

Contents

1 Graph and its representations	18
Source	27
2 Breadth First Traversal for a Graph	28
Source	37
3 Applications of Breadth First Traversal	38
Source	39
4 Depth First Traversal for a Graph	40
Source	53
5 Applications of Depth First Search	54
Source	55
6 Iterative Depth First Traversal of Graph	56
Source	65
7 Detect Cycle in a Directed Graph	66
Source	72
8 Union Find - (Detect Cycle in an Undirected Graph)	73
Source	80
9 Detect cycle in an Undirected graph	81
Source	87
10 Longest Path in a Directed Acyclic Graph	88
Source	93
11 Topological Sorting	94
Source	100
12 Check whether a given graph is Bipartite or not	101
Source	114
13 Snake and Ladder Problem	115
Source	123

14 Minimize Cash Flow among a given set of friends who have borrowed money from each other	124
Source	135
15 Boggle (Find all possible words in a board of characters)	136
Source	139
16 Assign directions to edges so that the directed graph remains acyclic	140
Source	141
17 Prim's Minimum Spanning Tree (MST))	142
Source	154
18 Applications of Minimum Spanning Tree Problem	155
Source	156
19 Prim's MST for Adjacency List Representation	157
Source	174
20 Kruskal's Minimum Spanning Tree Algorithm	175
Source	188
21 Dijkstra's shortest path algorithm	189
Source	198
22 Dijkstra's Algorithm for Adjacency List Representation	199
Source	213
23 Greedy Algorithms Set 9 (Boruvka's algorithm)	214
Source	224
24 Bellman-Ford Algorithm	225
Source	236
25 Floyd Warshall Algorithm	237
Source	247
26 Johnson's algorithm for All-pairs shortest paths	248
Source	250
27 Shortest Path in Directed Acyclic Graph	251
Source	261
28 Some interesting shortest path questions,	262
Source	263
29 Shortest path with exactly k edges in a directed and weighted graph	264
Source	270
30 Find if there is a path between two vertices in a directed graph	271
Source	278

31 Find if the strings can be chained to form a circle	279
Source	287
32 Find same contacts in a list of contacts	288
Source	291
33 Find the minimum cost to reach destination using a train	292
Source	302
34 Given a sorted dictionary of an alien language, find order of characters	303
Source	309
35 Length of shortest chain to reach a target word	310
Source	314
36 Minimum time required to rot all oranges	315
Source	323
37 Optimal read list for given number of days	324
Source	329
38 Print all paths from a given source to a destination	330
Source	337
39 Print all Jumping Numbers smaller than or equal to a given value	338
Source	342
40 Stable Marriage Problem	343
Source	347
41 Steiner Tree Problem	348
Source	350
42 Connectivity in a directed graph	351
Source	360
43 Articulation Points (or Cut Vertices) in a Graph	361
Source	372
44 Biconnected graph	373
Source	385
45 Bridges in a graph	386
Source	396
46 Eulerian path and circuit	397
Source	408
47 Fleury's Algorithm for printing Eulerian Path or Circuit	409
Source	421

48 Strongly Connected Components	422
Source	432
49 Tarjan's Algorithm to find Strongly Connected Components	433
Source	444
50 Transitive closure of a graph	445
Source	451
51 Find the number of islands	452
Source	463
52 Count all possible walks from a source to a destination with exactly k edges	464
Source	474
53 Euler Circuit in a Directed Graph	475
Source	484
54 Biconnected Components	485
Source	496
55 Graph Coloring (Introduction and Applications)	497
Source	498
56 Greedy Algorithm for Graph Coloring	499
Source	506
57 Travelling Salesman Problem (Naive and Dynamic Programming)	507
Source	508
58 Travelling Salesman Problem (Approximate using MST)	509
Source	511
59 Hamiltonian Cycle	512
Source	522
60 Vertex Cover Problem Set 1 (Introduction and Approximate Algorithm)	523
Source	530
61 K Centers Problem Set 1 (Greedy Approximate Algorithm)	531
Source	533
62 Ford-Fulkerson Algorithm for Maximum Flow Problem	534
Source	543
63 Find maximum number of edge disjoint paths between two vertices	544
Source	550
64 Find minimum s-t cut in a flow network	551
Source	560

65 Maximum Bipartite Matching	561
Source	571
66 Channel Assignment Problem	572
Source	575
67 Union Find Algorithm Set 2 (Union By Rank and Path Compression)	576
Source	584
68 Karger's algorithm for Minimum Cut	585
Source	592
69 Karger's algorithm for Minimum Cut Set 2 (Analysis and Applications)	593
Source	596
70 Hopcroft Karp Algorithm for Maximum Matching Set 1 (Introduction)	597
Source	599
71 Hopcroft-Karp Algorithm for Maximum Matching Set 2 (Implementation)	600
Source	605
72 0-1 BFS (Shortest Path in a Binary Weight Graph)	606
Source	609
73 2-Satisfiability (2-SAT) Problem	610
Source	617
74 A matrix probability question	618
Source	625
75 A Peterson Graph Problem	626
Source	628
76 All Topological Sorts of a Directed Acyclic Graph	629
Source	635
77 Applications of Graph Data Structure	636
Source	637
78 Barabasi Albert Graph (for Scale Free Models)	638
Source	642
79 Bellman-Ford Algorithm DP-23	643
Source	654
80 Best First Search (Informed Search)	655
Source	657
81 Betweenness Centrality (Centrality Measure)	658
Source	662

82 BFS for Disconnected Graph	663
Source	665
83 BFS using STL for competitive coding	666
Source	669
84 BFS using vectors & queue as per the algorithm of CLRS	670
Source	675
85 Bidirectional Search	676
Source	682
86 Boggle (Find all possible words in a board of characters) Set 1	683
Source	686
87 Boggle Set 2 (Using Trie)	687
Source	696
88 Boruvka's algorithm for Minimum Spanning Tree	697
Source	697
89 Boruvka's algorithm Greedy Algo-9	698
Source	708
90 Breadth First Search or BFS for a Graph	709
Source	718
91 Calculate number of nodes between two vertices in an acyclic Graph by Disjoint Union method	719
Source	725
92 Check for star graph	726
Source	734
93 Check if a given directed graph is strongly connected Set 2 (Kosaraju using BFS)	735
Source	741
94 Check if a given graph is Bipartite using DFS	742
Source	745
95 Check if a given graph is tree or not	746
Source	753
96 Check if a graph is strongly connected Set 1 (Kosaraju using DFS)	754
Source	763
97 Check if a graphs has a cycle of odd length	764
Source	770
98 Check if removing a given edge disconnects a graph	771

Source	774
99 Check if the given permutation is a valid DFS of graph	775
Source	779
100 Check if there is a cycle with odd weight sum in an undirected graph	780
Source	785
101 Check if two nodes are on same path in a tree	786
Source	790
102 Check loop in array according to given constraints	791
Source	793
103 Check whether given degrees of vertices represent a Graph or Tree	794
Source	796
104 Chinese Postman or Route Inspection Set 1 (introduction)	797
Source	800
105 Clone a Directed Acyclic Graph	801
Source	804
106 Clone an Undirected Graph	805
Source	813
107 Clustering Coefficient in Graph Theory	814
Source	817
108 Comparison of Dijkstra's and Floyd-Warshall algorithms	818
Source	819
109 Computer Networks Cuts and Network Flow	820
Source	824
110 Connected Components in an undirected graph	825
Source	828
111 Construct a graph from given degrees of all vertices	829
Source	831
112 Construct binary palindrome by repeated appending and trimming	832
Source	834
113 Count all possible paths between two vertices	835
Source	842
114 Count nodes within K-distance from all nodes in a set	843
Source	846
115 Count number of edges in an undirected graph	847

Source	849
116 Count number of trees in a forest	850
Source	852
117 Count single node isolated sub-graphs in a disconnected graph	853
Source	854
118 Count the number of nodes at given level in a tree using BFS.	855
Source	859
119 Count the number of non-reachable nodes	860
Source	863
120 Cycles of length n in an undirected and connected graph	864
Source	870
121 Degree Centrality (Centrality Measure)	871
Source	874
122 Delete Edge to minimize subtree sum difference	875
Source	877
123 Depth First Search or DFS for a Graph	878
Source	891
124 Detect a negative cycle in a Graph (Bellman Ford)	892
Source	901
125 Detect Cycle in a directed graph using colors	902
Source	907
126 Detect cycle in an undirected graph using BFS	908
Source	910
127 Detect cycle in an undirected graph	911
Source	917
128 Detecting negative cycle using Floyd Warshall	918
Source	928
129 Determine whether a universal sink exists in a directed graph	929
Source	934
130 DFS for a n-ary tree (acyclic graph) represented as adjacency list	935
Source	938
131 Dial's Algorithm (Optimized Dijkstra for small range weights)	939
Source	950
132 Dijkstra's Algorithm for Adjacency List Representation Greedy Algo-8	951

Source	965
133 Dijkstra's Shortest Path Algorithm using priority_queue of STL	966
Source	973
134 Dijkstra's shortest path algorithm using set in STL	974
Source	979
135 Dijkstra's shortest path algorithm Greedy Algo-7	980
Source	989
136 Dinic's algorithm for Maximum Flow	990
Source	999
137 Disjoint Set (Or Union-Find) Set 1 (Detect Cycle in an Undirected Graph)	1000
Source	1008
138 Distance of nearest cell having 1 in a binary matrix	1009
Source	1019
139 Dominant Set of a Graph	1020
Source	1023
140 Dynamic Connectivity Set 1 (Incremental)	1024
Source	1028
141 Erdos Renyl Model (for generating Random Graphs)	1029
Source	1033
142 Eulerian path and circuit for undirected graph	1034
Source	1045
143 Eulerian Path in undirected graph	1046
Source	1051
144 Fibonacci Cube Graph	1052
Source	1056
145 Find a Mother Vertex in a Graph	1057
Source	1062
146 Find all reachable nodes from every node present in a given set	1063
Source	1068
147 Find alphabetical order such that words can be considered sorted	1069
Source	1073
148 Find if an array of strings can be chained to form a circle Set 1	1074
Source	1082

149 Find if an array of strings can be chained to form a circle Set 2	1083
Source1087
150 Find if there is a path of more than k length from a source	1088
Source1092
151 Find k-cores of an undirected graph	1093
Source1100
152 Find length of the largest region in Boolean Matrix	1101
Source1104
153 Find minimum weight cycle in an undirected graph	1105
Source1110
154 Find Shortest distance from a guard in a Bank	1111
Source1114
155 Find the Degree of a Particular vertex in a Graph	1115
Source1119
156 Find the minimum number of moves needed to move from one cell of matrix to another	1120
Source1124
157 Find the number of islands Set 1 (Using DFS)	1125
Source1136
158 Find the number of Islands Set 2 (Using Disjoint Set)	1137
Source1141
159 Find the smallest binary digit multiple of given number	1142
Source1145
160 Find whether there is path between two cells in matrix	1146
Source1151
161 Finding minimum vertex cover size of a graph using binary search	1152
Source1157
162 Flood fill Algorithm - how to implement fill() in paint?	1158
Source1161
163 Floyd Warshall Algorithm DP-16	1162
Source1172
164 Generate a graph using Dictionary in Python	1173
Source1178
165 Given a matrix of 'O' and 'X', replace 'O' with 'X' if surrounded by 'X'	1179
Source1190

166 Graph Coloring Set 1 (Introduction and Applications)	1191
Source1192
167 Graph Coloring Set 2 (Greedy Algorithm)	1193
Source1200
168 Graph implementation using STL for competitive programming Set 1 (DFS of Unweighted and Undirected)	1201
Source1203
169 Graph implementation using STL for competitive programming Set 2 (Weighted graph)	1204
Source1206
170 Graph representations using set and hash	1207
Source1213
171 Hamiltonian Cycle Backtracking-6	1214
Source1224
172 Height of a generic tree from parent array	1225
Source1230
173 Hierholzer's Algorithm for directed graph	1231
Source1235
174 Hopcroft-Karp Algorithm for Maximum Matching Set 1 (Introduction)	1236
Source1238
175 Hungarian Algorithm for Assignment Problem Set 1 (Introduction)	1239
Source1243
176 Hypercube Graph	1244
Source1246
177 Implementation of Graph in JavaScript	1247
Source1254
178 Iterative Deepening Search(IDS) or Iterative Deepening Depth First Search(IDDFS)	1255
Source1261
179 Java Program for Dijkstra's Algorithm with Path Printing	1262
Source1264
180 k'th heaviest adjacent node in a graph where each vertex has weight	1265
Source1267
181 Kahn's algorithm for Topological Sorting	1268
Source1276

182 Karger's algorithm for Minimum Cut Set 1 (Introduction and Implementation)	1277
Source1284
183 Karp's minimum mean (or average) weight cycle algorithm	1285
Source1288
184 Katz Centrality (Centrality Measure)	1289
Source1295
185 Kruskal's Algorithm (Simple Implementation for Adjacency Matrix)	1296
Source1298
186 Kruskal's Minimum Spanning Tree Algorithm Greedy Algo-2	1299
Source1312
187 Kruskal's Minimum Spanning Tree using STL in C++	1313
Source1318
188 Largest connected component on a grid	1319
Source1329
189 Largest subset of Graph vertices with edges of 2 or more colors	1330
Source1335
190 Level Ancestor Problem	1336
Source1343
191 Level of Each node in a Tree from source node (using BFS)	1344
Source1347
192 Longest path between any pair of vertices	1348
Source1351
193 Longest Path in a Directed Acyclic Graph Set 2	1352
Source1356
194 m Coloring Problem Backtracking-5	1357
Source1364
195 Magical Indices in an array	1365
Source1374
196 Mathematics Graph theory practice questions	1375
Source1379
197 Max Flow Problem Introduction	1380
Source1383
198 Maximum edges that can be added to DAG so that it remains DAG	1384
Source1387

199 Maximum number of edges to be added to a tree so that it stays a Bipartite graph	1388
Source1390
200 Maximum product of two non-intersecting paths in a tree	1391
Source1394
201 Minimize the number of weakly connected nodes	1395
Source1399
202 Minimum cost path from source node to destination node via an intermediate node	1400
Source1404
203 Minimum Cost Path with Left, Right, Bottom and Up moves allowed	1405
Source1408
204 Minimum cost to connect all cities	1409
Source1412
205 Minimum cost to connect weighted nodes represented as array	1413
Source1416
206 Minimum edge reversals to make a root	1417
Source1421
207 Minimum edges required to add to make Euler Circuit	1422
Source1425
208 Minimum edges to reverse to make path from a source to a destination	1426
Source1431
209 Minimum initial vertices to traverse whole matrix with given conditions	1432
Source1435
210 Minimum number of edges between two vertices of a Graph	1436
Source1441
211 Minimum number of operation required to convert number x into y	1442
Source1446
212 Minimum number of swaps required to sort an array	1447
Source1453
213 Minimum Product Spanning Tree	1454
Source1458
214 Minimum steps to reach a destination	1459
Source1465
215 Minimum steps to reach end of array under constraints	1466

Source1469
216 Minimum steps to reach target by a Knight Set 1	1470
Source1472
217 Move weighting scale alternate under given constraints	1473
Source1475
218 Multi Source Shortest Path in Unweighted Graph	1476
Source1483
219 Multistage Graph (Shortest Path)	1484
Source1487
220 NetworkX : Python software package for study of complex networks	1488
Source1494
221 Number of cyclic elements in an array where we can jump according to value	1495
Source1499
222 Number of groups formed in a graph of friends	1500
Source1503
223 Number of loops of size k starting from a specific node	1504
Source1508
224 Number of pair of positions in matrix which are not accessible	1509
Source1511
225 Number of shortest paths in an unweighted and directed graph	1512
Source1516
226 Number of single cycle components in an undirected graph	1517
Source1520
227 Number of sink nodes in a graph	1521
Source1523
228 Number of Transpositions in a Permutation	1524
Source1530
229 Number of Triangles in an Undirected Graph	1531
Source1538
230 Number of Triangles in Directed and Undirected Graphs	1539
Source1548
231 Number of Unicolored Paths between two nodes	1549
Source1552

232 Path in a Rectangle with Circles	1553
Source1557
233 Paths to travel each nodes using each edge (Seven Bridges of Königsberg)	1558
Source1563
234 Permutation of numbers such that sum of two consecutive numbers is a perfect square	1564
Source1568
235 Prim's Algorithm (Simple Implementation for Adjacency Matrix Representation)	1569
Source1571
236 Prim's algorithm using priority_queue in STL	1572
Source1580
237 Prim's Minimum Spanning Tree (MST) Greedy Algo-5	1581
Source1593
238 Prim's MST for Adjacency List Representation Greedy Algo-6	1594
Source1611
239 Print all paths from a given source to a destination using BFS	1612
Source1615
240 Print all the cycles in an undirected graph	1616
Source1620
241 Print the DFS traversal step-wise (Backtracking also)	1621
Source1624
242 Printing Paths in Dijkstra's Shortest Path Algorithm	1625
Source1636
243 Product of lengths of all cycles in an undirected graph	1637
Source1642
244 Proof that Hamiltonian Path is NP-Complete	1643
Source1653
245 Push Relabel Algorithm Set 1 (Introduction and Illustration)	1654
Source1659
246 Push Relabel Algorithm Set 2 (Implementation)	1660
Source1667
247 Remove all outgoing edges except edge with minimum weight	1668
Source1673
248 Reverse Delete Algorithm for Minimum Spanning Tree	1674

Source1680
249 Roots of a tree which give minimum height	1681
Source1686
250 Shortest path in a Binary Maze	1687
Source1691
251 Shortest Path in a weighted Graph where weight of an edge is 1 or 2	1692
Source1698
252 Shortest path in an unweighted graph	1699
Source1703
253 Shortest path to reach one prime to other by changing single digit at a time	1704
Source1708
254 Some interesting shortest path questions Set 1	1709
Source1710
255 Stepping Numbers	1711
Source1723
256 Subtree of all nodes in a tree using DFS	1724
Source1728
257 Sum of dependencies in a graph	1729
Source1732
258 Sum of the minimum elements in all connected components of an undirected graph	1733
Source1736
259 Topological Sort of a graph using departure time of vertex	1737
Source1740
260 Total number of Spanning Trees in a Graph	1741
Source1742
261 Transitive Closure of a Graph using DFS	1743
Source1749
262 Transpose graph	1750
Source1752
263 Traveling Salesman Problem (TSP) Implementation	1753
Source1755
264 Travelling Salesman Problem Set 1 (Naive and Dynamic Programming)	1756
Source1757

265 Travelling Salesman Problem Set 2 (Approximate using MST)	1758
Source1760
266 Two Clique Problem (Check if Graph can be divided in two Cliques)	1761
Source1765
267 Undirected graph splitting and its application for number pairs	1766
Source1774
268 Union-Find Algorithm (Union By Rank and Find by Optimized Path Compression)	1775
Source1780
269 Union-Find Algorithm Set 2 (Union By Rank and Path Compression)	1781
Source1789
270 Water Connection Problem	1790
Source1796
271 Water Jug problem using BFS	1797
Source1800
272 Word Ladder (Length of shortest chain to reach a target word)	1801
Source1805

Contents

Chapter 1

Graph and its representations

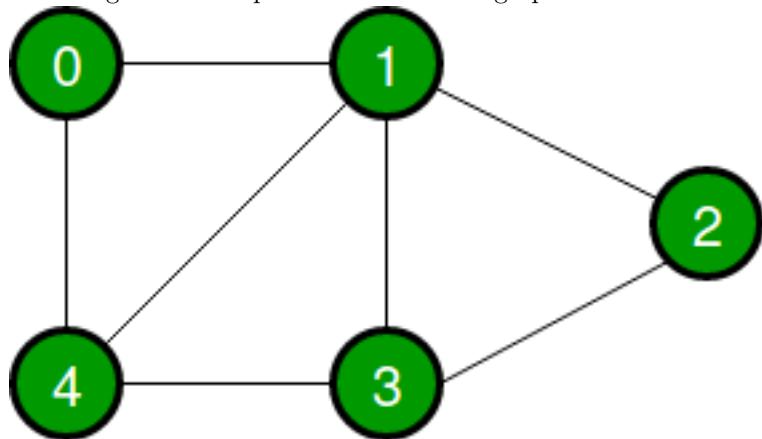
Graph and its representations - GeeksforGeeks

Graph is a data structure that consists of following two components:

1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not same as (v, u) in case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v . The edges may contain weight/value/cost.

Graphs are used to represent many real-life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender and locale. See [this](#)for more applications of graph.

Following is an example of an undirected graph with 5 vertices.



Following two are the most commonly used representations of a graph.

1. Adjacency Matrix
2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of the graph representation is situation specific. It totally depends on the type of operations to be performed and ease of use.

Adjacency Matrix:

Adjacency Matrix is a 2D array of size $V \times V$ where V is the number of vertices in a graph. Let the 2D array be $\text{adj}[][],$ a slot $\text{adj}[i][j] = 1$ indicates that there is an edge from vertex i to vertex $j.$ Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If $\text{adj}[i][j] = w,$ then there is an edge from vertex i to vertex j with weight $w.$

The adjacency matrix for the above example graph is:

0	1	2	3	4	
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

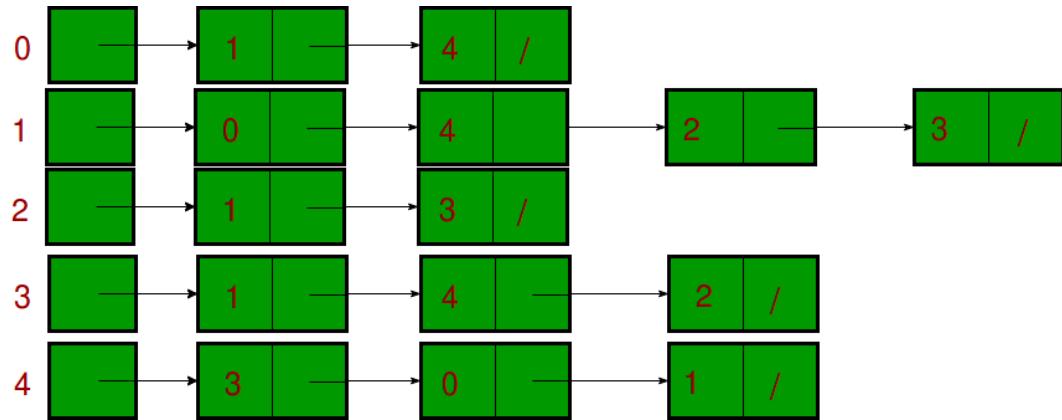
Pros: Representation is easier to implement and follow. Removing an edge takes $O(1)$ time. Queries like whether there is an edge from vertex ‘ u ’ to vertex ‘ v ’ are efficient and can be done $O(1).$

Cons: Consumes more space $O(V^2).$ Even if the graph is sparse(contains less number of

edges), it consumes the same space. Adding a vertex is $O(V^2)$ time.
 Please see [this](#) for a sample Python implementation of adjacency matrix.

Adjacency List:

An array of lists is used. Size of the array is equal to the number of vertices. Let the array be $\text{array}[]$. An entry $\text{array}[i]$ represents the list of vertices adjacent to the i th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is adjacency list representation of the above graph.



Note that in below implementation, we use dynamic arrays (vector in C++/ArrayList in Java) to represent adjacency lists instead of linked list. The vector implementation has advantages of cache friendliness.

C++

```
// A simple representation of graph using STL
#include<bits/stdc++.h>
using namespace std;

// A utility function to add an edge in an
// undirected graph.
void addEdge(vector<int> adj[], int u, int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}

// A utility function to print the adjacency list
// representation of graph
void printGraph(vector<int> adj[], int V)
{
    for (int v = 0; v < V; ++v)
    {
        cout << "\n Adjacency list of vertex "
            << v << "\n head ";
        for (auto x : adj[v])

```

```

        cout << "-> " << x;
        printf("\n");
    }
}

// Driver code
int main()
{
    int V = 5;
    vector<int> adj[V];
    addEdge(adj, 0, 1);
    addEdge(adj, 0, 4);
    addEdge(adj, 1, 2);
    addEdge(adj, 1, 3);
    addEdge(adj, 1, 4);
    addEdge(adj, 2, 3);
    addEdge(adj, 3, 4);
    printGraph(adj, V);
    return 0;
}

```

C

```

// A C Program to demonstrate adjacency list
// representation of graphs
#include <stdio.h>
#include <stdlib.h>

// A structure to represent an adjacency list node
struct AdjListNode
{
    int dest;
    struct AdjListNode* next;
};

// A structure to represent an adjacency list
struct AdjList
{
    struct AdjListNode *head;
};

// A structure to represent a graph. A graph
// is an array of adjacency lists.
// Size of array will be V (number of vertices
// in graph)
struct Graph
{
    int V;

```

```

        struct AdjList* array;
    };

// A utility function to create a new adjacency list node
struct AdjListNode* newAdjListNode(int dest)
{
    struct AdjListNode* newNode =
        (struct AdjListNode*) malloc(sizeof(struct AdjListNode));
    newNode->dest = dest;
    newNode->next = NULL;
    return newNode;
}

// A utility function that creates a graph of V vertices
struct Graph* createGraph(int V)
{
    struct Graph* graph =
        (struct Graph*) malloc(sizeof(struct Graph));
    graph->V = V;

    // Create an array of adjacency lists. Size of
    // array will be V
    graph->array =
        (struct AdjList*) malloc(V * sizeof(struct AdjList));

    // Initialize each adjacency list as empty by
    // making head as NULL
    int i;
    for (i = 0; i < V; ++i)
        graph->array[i].head = NULL;

    return graph;
}

// Adds an edge to an undirected graph
void addEdge(struct Graph* graph, int src, int dest)
{
    // Add an edge from src to dest. A new node is
    // added to the adjacency list of src. The node
    // is added at the begining
    struct AdjListNode* newNode = newAdjListNode(dest);
    newNode->next = graph->array[src].head;
    graph->array[src].head = newNode;

    // Since graph is undirected, add an edge from
    // dest to src also
    newNode = newAdjListNode(src);
    newNode->next = graph->array[dest].head;
}

```

```

graph->array[dest].head = newNode;
}

// A utility function to print the adjacency list
// representation of graph
void printGraph(struct Graph* graph)
{
    int v;
    for (v = 0; v < graph->V; ++v)
    {
        struct AdjListNode* pCrawl = graph->array[v].head;
        printf("\n Adjacency list of vertex %d\n head ", v);
        while (pCrawl)
        {
            printf("-> %d", pCrawl->dest);
            pCrawl = pCrawl->next;
        }
        printf("\n");
    }
}

// Driver program to test above functions
int main()
{
    // create the graph given in above figure
    int V = 5;
    struct Graph* graph = createGraph(V);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 4);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 4);

    // print the adjacency list representation of the above graph
    printGraph(graph);

    return 0;
}

```

Java

```

// Java Program to demonstrate adjacency list
// representation of graphs
import java.util.LinkedList;

public class GFG

```

```
{
    // A user define class to represent a graph.
    // A graph is an array of adjacency lists.
    // Size of array will be V (number of vertices
    // in graph)
    static class Graph
    {
        int V;
        LinkedList<Integer> adjListArray[];

        // constructor
        Graph(int V)
        {
            this.V = V;

            // define the size of array as
            // number of vertices
            adjListArray = new LinkedList[V];

            // Create a new list for each vertex
            // such that adjacent nodes can be stored
            for(int i = 0; i < V ; i++){
                adjListArray[i] = new LinkedList<>();
            }
        }

        // Adds an edge to an undirected graph
        static void addEdge(Graph graph, int src, int dest)
        {
            // Add an edge from src to dest.
            graph.adjListArray[src].addFirst(dest);

            // Since graph is undirected, add an edge from dest
            // to src also
            graph.adjListArray[dest].addFirst(src);
        }

        // A utility function to print the adjacency list
        // representation of graph
        static void printGraph(Graph graph)
        {
            for(int v = 0; v < graph.V; v++)
            {
                System.out.println("Adjacency list of vertex "+ v);
                System.out.print("head");
                for(Integer pCrawl: graph.adjListArray[v]){
                    System.out.print(" -> "+pCrawl);
                }
            }
        }
    }
}
```

```

        }
        System.out.println("\n");
    }
}

// Driver program to test above functions
public static void main(String args[])
{
    // create the graph given in above figure
    int V = 5;
    Graph graph = new Graph(V);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 4);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 4);

    // print the adjacency list representation of
    // the above graph
    printGraph(graph);
}
}

// This code is contributed by Sumit Ghosh

```

Python3

```

"""
A Python program to demonstrate the adjacency
list representation of the graph
"""

# A class to represent the adjacency list of the node
class AdjNode:
    def __init__(self, data):
        self.vertex = data
        self.next = None


# A class to represent a graph. A graph
# is the list of the adjacency lists.
# Size of the array will be the no. of the
# vertices "V"
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [None] * self.V

```

```

# Function to add an edge in an undirected graph
def add_edge(self, src, dest):
    # Adding the node to the source node
    node = AdjNode(dest)
    node.next = self.graph[src]
    self.graph[src] = node

    # Adding the source node to the destination as
    # it is the undirected graph
    node = AdjNode(src)
    node.next = self.graph[dest]
    self.graph[dest] = node

# Function to print the graph
def print_graph(self):
    for i in range(self.V):
        print("Adjacency list of vertex {}\\n head".format(i), end="")
        temp = self.graph[i]
        while temp:
            print(" -> {}".format(temp.vertex), end="")
            temp = temp.next
        print(" \\n")

# Driver program to the above graph class
if __name__ == "__main__":
    V = 5
    graph = Graph(V)
    graph.add_edge(0, 1)
    graph.add_edge(0, 4)
    graph.add_edge(1, 2)
    graph.add_edge(1, 3)
    graph.add_edge(1, 4)
    graph.add_edge(2, 3)
    graph.add_edge(3, 4)

    graph.print_graph()

# This code is contributed by Kanav Malhotra

```

Output:

```

Adjacency list of vertex 0
head -> 1-> 4

```

```
Adjacency list of vertex 1  
head -> 0-> 2-> 3-> 4
```

```
Adjacency list of vertex 2  
head -> 1-> 3
```

```
Adjacency list of vertex 3  
head -> 1-> 2-> 4
```

```
Adjacency list of vertex 4  
head -> 0-> 1-> 3
```

Pros: Saves space $O(V+E)$. In the worst case, there can be $C(V, 2)$ number of edges in a graph thus consuming $O(V^2)$ space. Adding a vertex is easier.

Cons: Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done $O(V)$.

Reference:

http://en.wikipedia.org/wiki/Graph_abstract_data_type

Related Post:

[Graph representation using STL for competitive programming Set 1 \(DFS of Unweighted and Undirected\)](#)

[Graph implementation using STL for competitive programming Set 2 \(Weighted graph\)](#)

This article is compiled by [Aashish Barnwal](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Improved By : [RajatSinghal](#), [kanavMalhotra](#)

Source

<https://www.geeksforgeeks.org/graph-and-its-representations/>

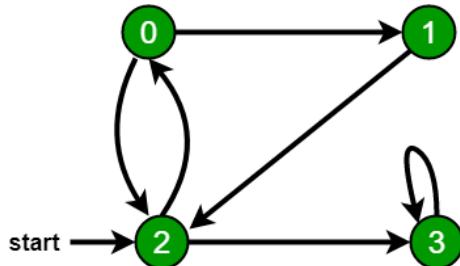
Chapter 2

Breadth First Traversal for a Graph

Breadth First Search or BFS for a Graph - GeeksforGeeks

[Breadth First Traversal \(or Search\)](#) for a graph is similar to Breadth First Traversal of a tree (See method 2 of [this post](#)). The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth First Traversal of the following graph is 2, 0, 3, 1.



Following are the implementations of simple Breadth First Traversal from a given source.

The implementation uses [adjacency list representation](#) of graphs. [STL's list container](#) is used to store lists of adjacent nodes and queue of nodes needed for BFS traversal.

C++

```
// Program to print BFS traversal from a given
// source vertex. BFS(int s) traverses vertices
// reachable from s.
```

```
#include<iostream>
#include <list>

using namespace std;

// This class represents a directed graph using
// adjacency list representation
class Graph
{
    int V;      // No. of vertices

    // Pointer to an array containing adjacency
    // lists
    list<int> *adj;
public:
    Graph(int V); // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // prints BFS traversal from a given source s
    void BFS(int s);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);
```

```
// 'i' will be used to get all adjacent
// vertices of a vertex
list<int>::iterator i;

while(!queue.empty())
{
    // Dequeue a vertex from queue and print it
    s = queue.front();
    cout << s << " ";
    queue.pop_front();

    // Get all adjacent vertices of the dequeued
    // vertex s. If a adjacent has not been visited,
    // then mark it visited and enqueue it
    for (i = adj[s].begin(); i != adj[s].end(); ++i)
    {
        if (!visited[*i])
        {
            visited[*i] = true;
            queue.push_back(*i);
        }
    }
}

// Driver program to test methods of graph class
int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Breadth First Traversal "
        << "(starting from vertex 2) \n";
    g.BFS(2);

    return 0;
}
```

Java

```
// Java program to print BFS traversal from a given source vertex.
```

```
// BFS(int s) traverses vertices reachable from s.
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    private int V;      // No. of vertices
    private LinkedList<Integer> adj[]; //Adjacency Lists

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    // Function to add an edge into the graph
    void addEdge(int v,int w)
    {
        adj[v].add(w);
    }

    // prints BFS traversal from a given source s
    void BFS(int s)
    {
        // Mark all the vertices as not visited(By default
        // set as false)
        boolean visited[] = new boolean[V];

        // Create a queue for BFS
        LinkedList<Integer> queue = new LinkedList<Integer>();

        // Mark the current node as visited and enqueue it
        visited[s]=true;
        queue.add(s);

        while (queue.size() != 0)
        {
            // Dequeue a vertex from queue and print it
            s = queue.poll();
            System.out.print(s+" ");

            // Get all adjacent vertices of the dequeued vertex s
            // If a adjacent has not been visited, then mark it
        }
    }
}
```

```
// visited and enqueue it
Iterator<Integer> i = adj[s].listIterator();
while (i.hasNext())
{
    int n = i.next();
    if (!visited[n])
    {
        visited[n] = true;
        queue.add(n);
    }
}
}

// Driver method to
public static void main(String args[])
{
    Graph g = new Graph(4);

    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    System.out.println("Following is Breadth First Traversal "+
                       "(starting from vertex 2)");

    g.BFS(2);
}
}

// This code is contributed by Aakash Hasija
```

Python3

```
# Python3 Program to print BFS traversal
# from a given source vertex. BFS(int s)
# traverses vertices reachable from s.
from collections import defaultdict

# This class represents a directed graph
# using adjacency list representation
class Graph:

    # Constructor
    def __init__(self):
```

```

# default dictionary to store graph
self.graph = defaultdict(list)

# function to add an edge to graph
def addEdge(self,u,v):
    self.graph[u].append(v)

# Function to print a BFS of graph
def BFS(self, s):

    # Mark all the vertices as not visited
    visited = [False] * (len(self.graph))

    # Create a queue for BFS
    queue = []

    # Mark the source node as
    # visited and enqueue it
    queue.append(s)
    visited[s] = True

    while queue:

        # Dequeue a vertex from
        # queue and print it
        s = queue.pop(0)
        print (s, end = " ")

        # Get all adjacent vertices of the
        # dequeued vertex s. If a adjacent
        # has not been visited, then mark it
        # visited and enqueue it
        for i in self.graph[s]:
            if visited[i] == False:
                queue.append(i)
                visited[i] = True

# Driver code

# Create a graph given in
# the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

```

```

print ("Following is Breadth First Traversal"
       " (starting from vertex 2)")
g.BFS(2)

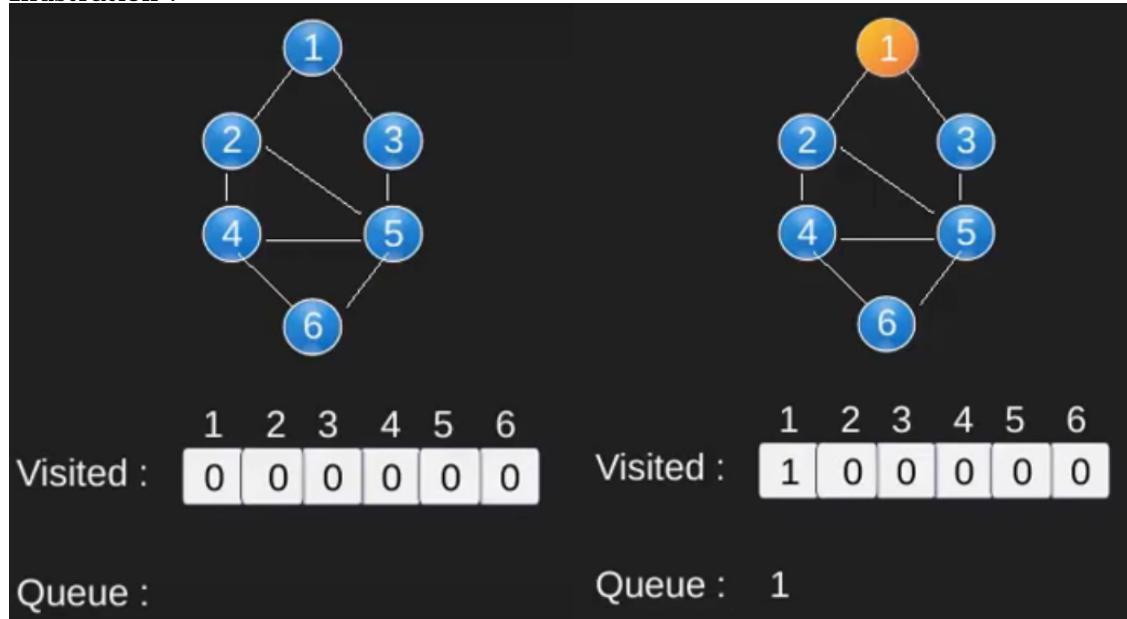
# This code is contributed by Neelam Yadav

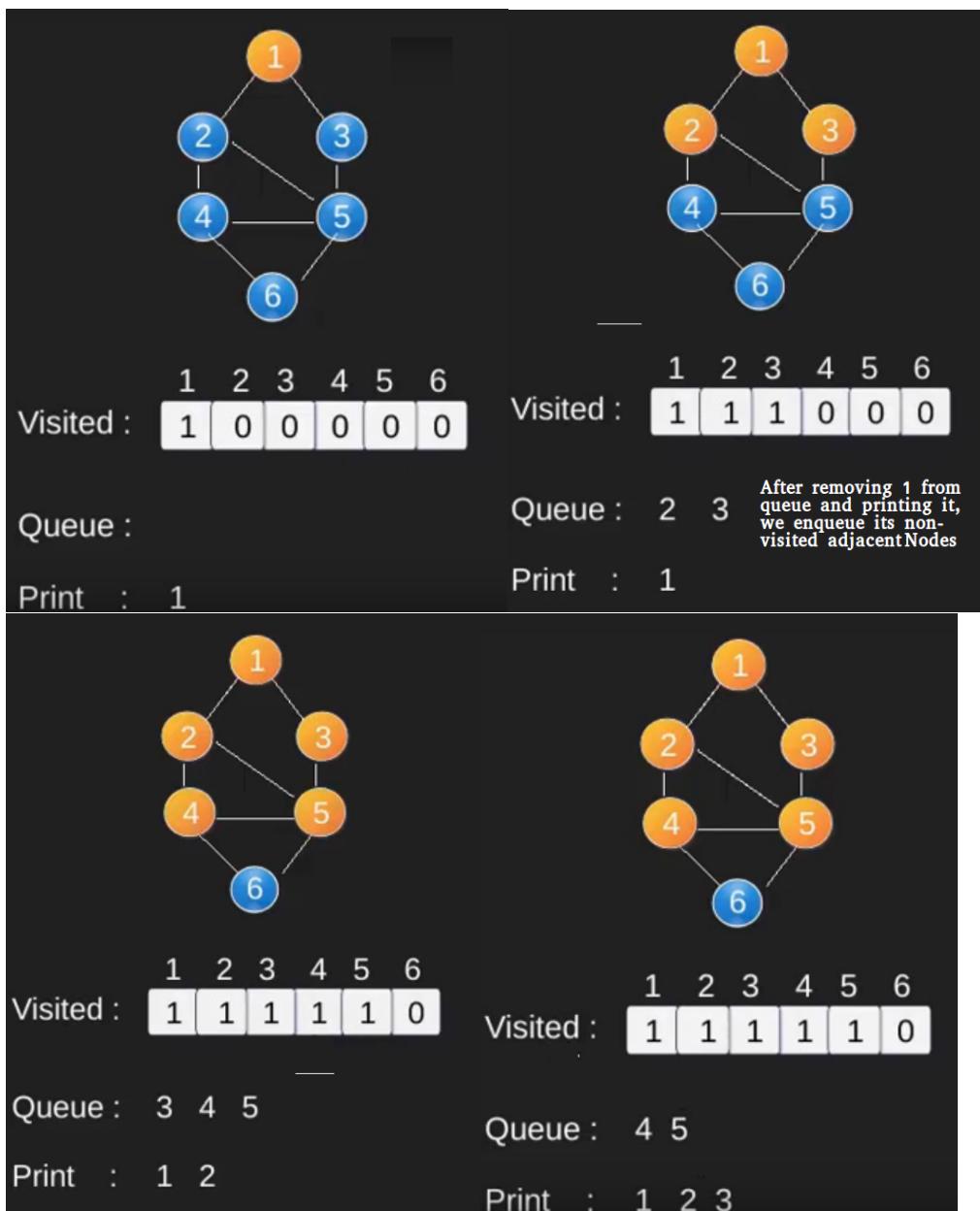
```

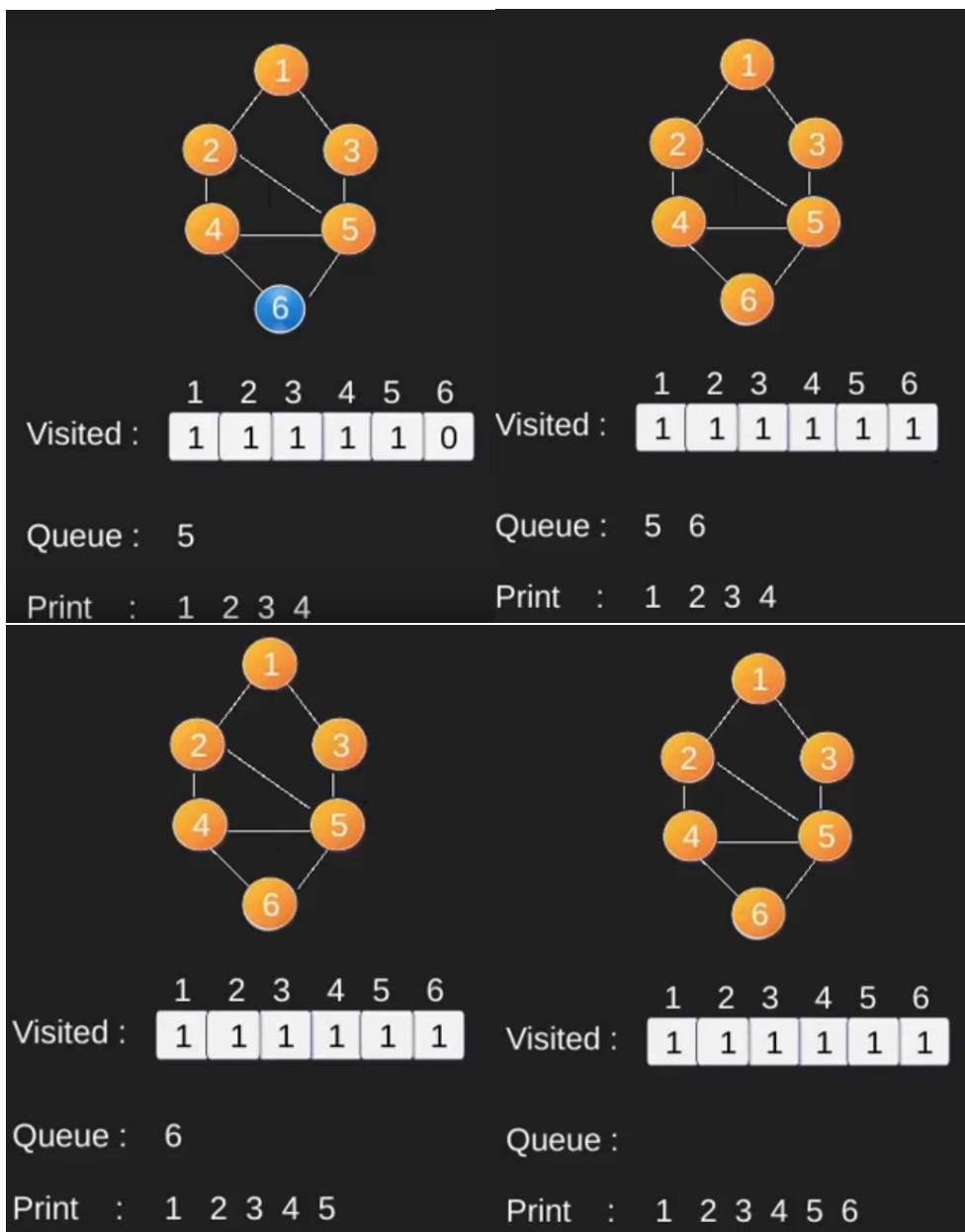
Output:

Following is Breadth First Traversal (starting from vertex 2)
 2 0 3 1

Illustration :







Note that the above code traverses only the vertices reachable from a given source vertex. All the vertices may not be reachable from a given vertex (example Disconnected graph). To print all the vertices, we can modify the BFS function to do traversal starting from all nodes one by one (Like the [DFS modified version](#)) .

Time Complexity: $O(V+E)$ where V is number of vertices in the graph and E is number of edges in the graph.

You may like to see below also :

- Recent Articles on BFS
- Depth First Traversal
- Applications of Breadth First Traversal
- Applications of Depth First Search

Source

<https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/>

Chapter 3

Applications of Breadth First Traversal

Applications of Breadth First Traversal - GeeksforGeeks

We have earlier discussed [Breadth First Traversal Algorithm](#) for Graphs. We have also discussed [Applications of Depth First Traversal](#). In this article, applications of Breadth First Search are discussed.

- 1) Shortest Path and Minimum Spanning Tree for unweighted graph** In an unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using the minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.
- 2) Peer to Peer Networks.** In Peer to Peer Networks like [BitTorrent](#), Breadth First Search is used to find all neighbor nodes.
- 3) Crawlers in Search Engines:** Crawlers build index using Breadth First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of the built tree can be limited.
- 4) Social Networking Websites:** In social networks, we can find people within a given distance ‘k’ from a person using Breadth First Search till ‘k’ levels.
- 5) GPS Navigation systems:** Breadth First Search is used to find all neighboring locations.
- 6) Broadcasting in Network:** In networks, a broadcasted packet follows Breadth First Search to reach all nodes.
- 7) In Garbage Collection:** Breadth First Search is used in copying garbage collection using [Cheney's algorithm](#). Refer[this](#) and for details. Breadth First Search is preferred over Depth First Search because of better locality of reference:

8) Cycle detection in undirected graph: In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. In directed graph, only depth first search can be used.

9) Ford–Fulkerson algorithm In Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worst case time complexity to $O(VE^2)$.

10) To test if a graph is Bipartite We can either use Breadth First or Depth First Traversal.

11) Path Finding We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.

12) Finding all nodes within one connected component: We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

Many algorithms like [Prim's Minimum Spanning Tree](#) and [Dijkstra's Single Source Shortest Path](#) use structure similar to Breadth First Search.

There can be many more applications as Breadth First Search is one of the core algorithms for Graphs.

This article is contributed by **Neeraj Jain**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

<https://www.geeksforgeeks.org/applications-of-breadth-first-traversal/>

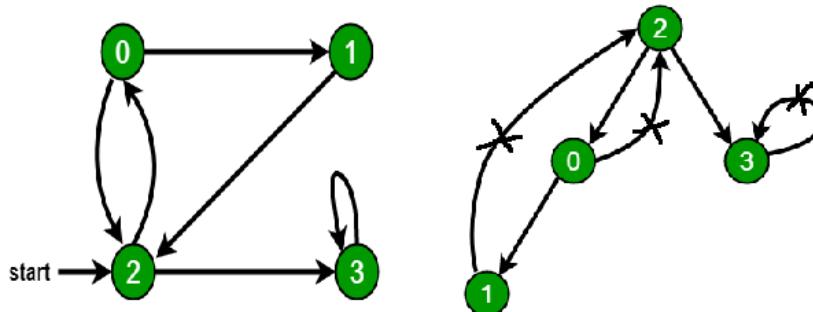
Chapter 4

Depth First Traversal for a Graph

Depth First Search or DFS for a Graph - GeeksforGeeks

Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Depth First Traversal of the following graph is 2, 0, 1, 3.



See [this post](#) for all applications of Depth First Traversal.

Following are implementations of simple Depth First Traversal. The C++ implementation uses adjacency list representation of graphs. STL's list container is used to store lists of adjacent nodes.

C++

```
// C++ program to print DFS traversal from
```

```
// a given vertex in a given graph
#include<iostream>
#include<list>
using namespace std;

// Graph class represents a directed graph
// using adjacency list representation
class Graph
{
    int V;      // No. of vertices

    // Pointer to an array containing
    // adjacency lists
    list<int> *adj;

    // A recursive function used by DFS
    void DFSUtil(int v, bool visited[]);

public:
    Graph(int V);    // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // DFS traversal of the vertices
    // reachable from v
    void DFS(int v);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and
    // print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent
    // to this vertex
```

```
list<int>::iterator i;
for (i = adj[v].begin(); i != adj[v].end(); ++i)
    if (!visited[*i])
        DFSUtil(*i, visited);
}

// DFS traversal of the vertices reachable from v.
// It uses recursive DFSUtil()
void Graph::DFS(int v)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function
    // to print DFS traversal
    DFSUtil(v, visited);
}

int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Depth First Traversal"
          " (starting from vertex 2) \n";
    g.DFS(2);

    return 0;
}
```

Java

```
// Java program to print DFS traversal from a given given graph
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
```

```
private int V;    // No. of vertices

// Array of lists for Adjacency List Representation
private LinkedList<Integer> adj[];

// Constructor
Graph(int v)
{
    V = v;
    adj = new LinkedList[v];
    for (int i=0; i<v; ++i)
        adj[i] = new LinkedList();
}

//Function to add an edge into the graph
void addEdge(int v, int w)
{
    adj[v].add(w);  // Add w to v's list.
}

// A function used by DFS
void DFSUtil(int v,boolean visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;
    System.out.print(v+" ");

    // Recur for all the vertices adjacent to this vertex
    Iterator<Integer> i = adj[v].listIterator();
    while (i.hasNext())
    {
        int n = i.next();
        if (!visited[n])
            DFSUtil(n, visited);
    }
}

// The function to do DFS traversal. It uses recursive DFSUtil()
void DFS(int v)
{
    // Mark all the vertices as not visited(set as
    // false by default in java)
    boolean visited[] = new boolean[V];

    // Call the recursive helper function to print DFS traversal
    DFSUtil(v, visited);
}
```

```
public static void main(String args[])
{
    Graph g = new Graph(4);

    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    System.out.println("Following is Depth First Traversal "+
        "(starting from vertex 2)");

    g.DFS(2);
}
}

// This code is contributed by Aakash Hasija
```

Python

```
# Python program to print DFS traversal from a
# given given graph
from collections import defaultdict

# This class represents a directed graph using
# adjacency list representation
class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # A function used by DFS
    def DFSUtil(self,v,visited):

        # Mark the current node as visited and print it
        visited[v]= True
        print v,

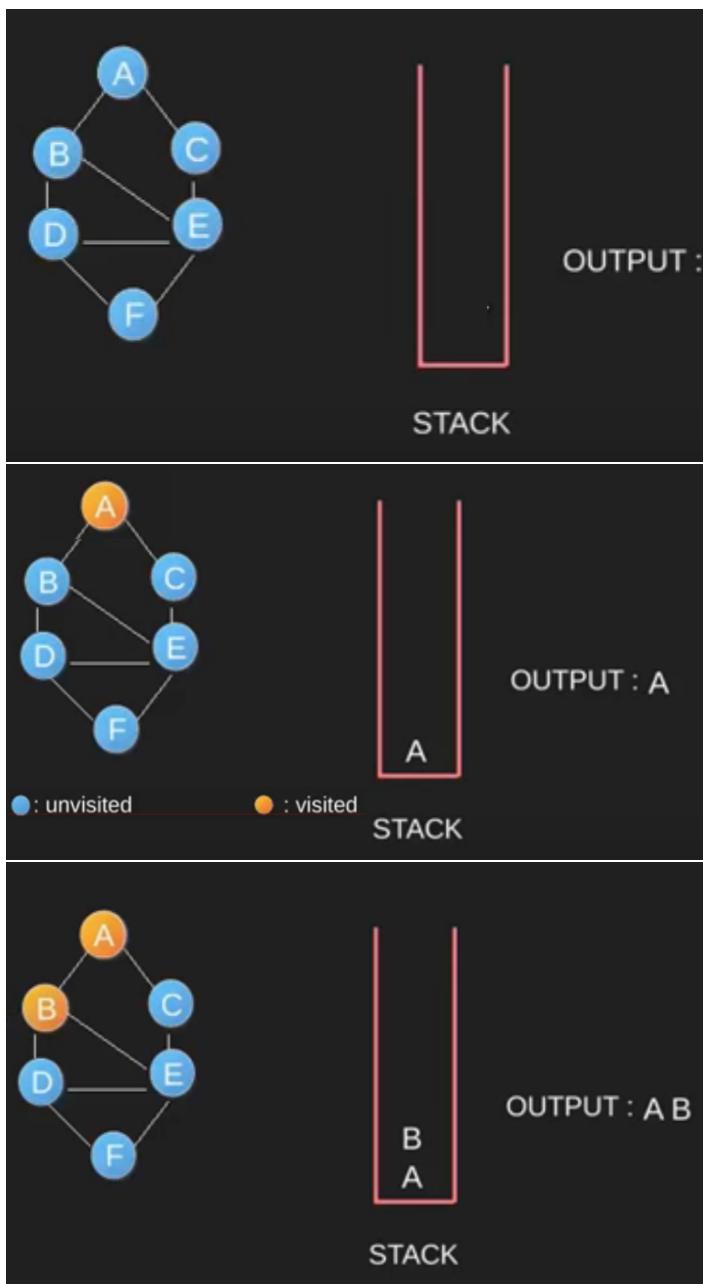
        # Recur for all the vertices adjacent to this vertex
        for i in self.graph[v]:
```

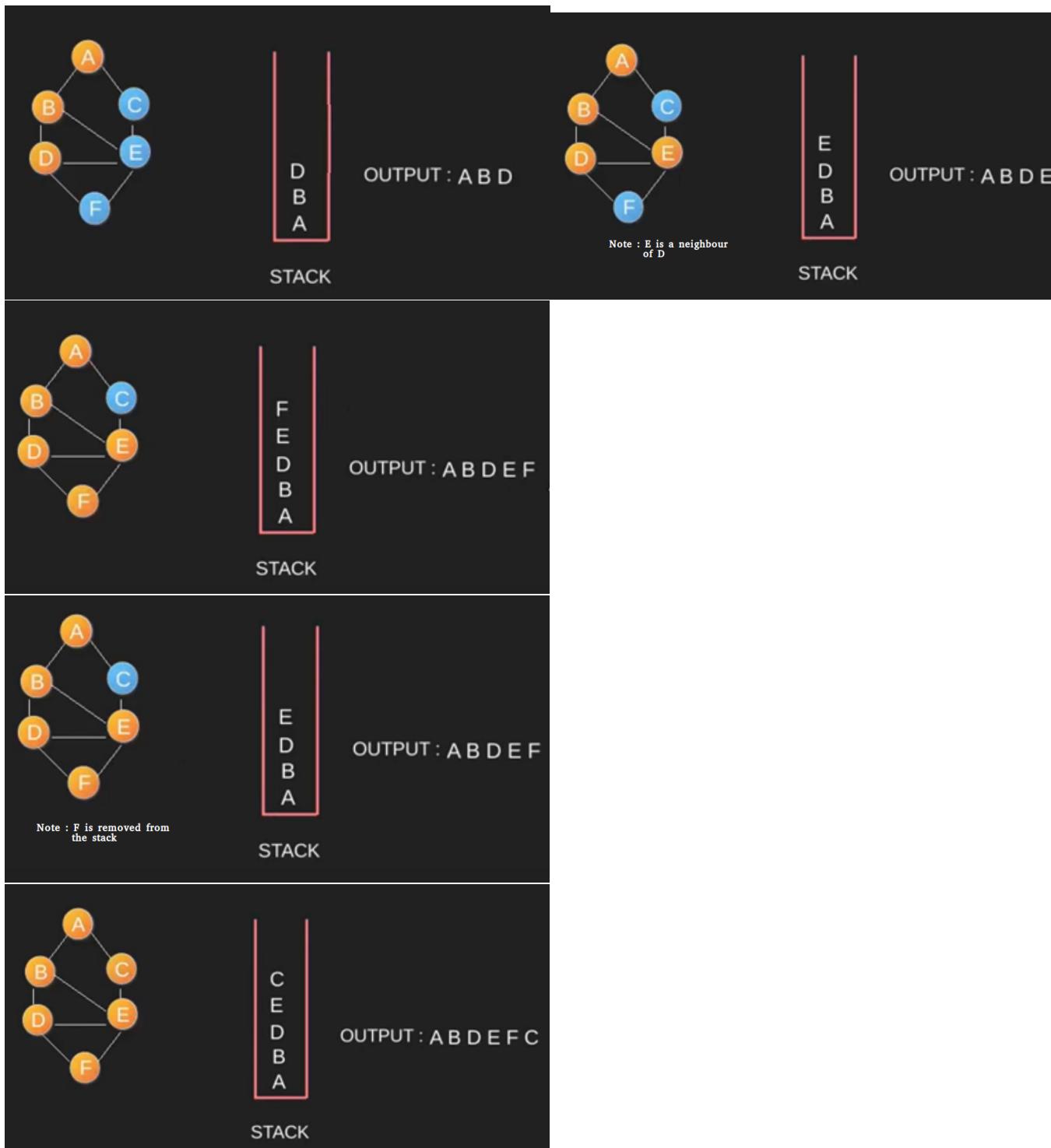
```
if visited[i] == False:  
    self.DFSUtil(i, visited)  
  
# The function to do DFS traversal. It uses  
# recursive DFSUtil()  
def DFS(self,v):  
  
    # Mark all the vertices as not visited  
    visited = [False]*(len(self.graph))  
  
    # Call the recursive helper function to print  
    # DFS traversal  
    self.DFSUtil(v,visited)  
  
# Driver code  
# Create a graph given in the above diagram  
g = Graph()  
g.addEdge(0, 1)  
g.addEdge(0, 2)  
g.addEdge(1, 2)  
g.addEdge(2, 0)  
g.addEdge(2, 3)  
g.addEdge(3, 3)  
  
print "Following is DFS from (starting from vertex 2)"  
g.DFS(2)  
  
# This code is contributed by Neelam Yadav
```

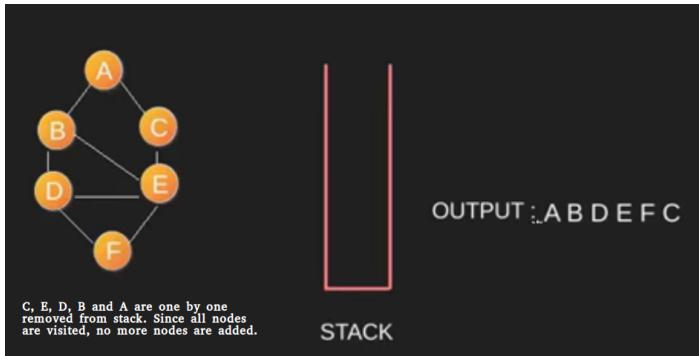
Output:

```
Following is Depth First Traversal (starting from vertex 2)  
2 0 1 3
```

Illustration for an Undirected Graph :







How to handle disconnected graph?

The above code traverses only the vertices reachable from a given source vertex. All the vertices may not be reachable from a given vertex (example Disconnected graph). To do complete DFS traversal of such graphs, we must call `DFSUtil()` for every vertex. Also, before calling `DFSUtil()`, we should check if it is already printed by some other call of `DFSUtil()`. Following implementation does the complete graph traversal even if the nodes are unreachable. The differences from the above code are highlighted in the below code.

C++

```
// C++ program to print DFS traversal for a given given graph
#include<iostream>
#include      <list>
using namespace std;

class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // Pointer to an array containing adjacency lists
    void DFSUtil(int v, bool visited[]); // A function used by DFS
public:
    Graph(int V);    // Constructor
    void addEdge(int v, int w);    // function to add an edge to graph
    void DFS();    // prints DFS traversal of the complete graph
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}
```

```
void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for(i = adj[v].begin(); i != adj[v].end(); ++i)
        if(!visited[*i])
            DFSUtil(*i, visited);
}

// The function to do DFS traversal. It uses recursive DFSUtil()
void Graph::DFS()
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to print DFS traversal
    // starting from all vertices one by one
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            DFSUtil(i, visited);
}

int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Depth First Traversal";
    g.DFS();
}

return 0;
}
```

Java

```
// Java program to print DFS traversal from a given graph
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    private int V;      // No. of vertices

    // Array of lists for Adjacency List Representation
    private LinkedList<Integer> adj[];

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    //Function to add an edge into the graph
    void addEdge(int v, int w)
    {
        adj[v].add(w);      // Add w to v's list.
    }

    // A function used by DFS
    void DFSUtil(int v,boolean visited[])
    {
        // Mark the current node as visited and print it
        visited[v] = true;
        System.out.print(v+" ");

        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> i = adj[v].listIterator();
        while (i.hasNext())
        {
            int n = i.next();
            if (!visited[n])
                DFSUtil(n,visited);
        }
    }

    // The function to do DFS traversal. It uses recursive DFSUtil()
    void DFS()
    {
```

```
// Mark all the vertices as not visited(set as
// false by default in java)
boolean visited[] = new boolean[V];

// Call the recursive helper function to print DFS traversal
// starting from all vertices one by one
for (int i=0; i<V; ++i)
    if (visited[i] == false)
        DFSUtil(i, visited);
}

public static void main(String args[])
{
    Graph g = new Graph(4);

    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    System.out.println("Following is Depth First Traversal");

    g.DFS();
}
}
// This code is contributed by Aakash Hasija
```

Python

```
# Python program to print DFS traversal for complete graph
from collections import defaultdict

# This class represents a directed graph using adjacency
# list representation
class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)
```

```
# A function used by DFS
def DFSUtil(self, v, visited):

    # Mark the current node as visited and print it
    visited[v] = True
    print v,

    # Recur for all the vertices adjacent to
    # this vertex
    for i in self.graph[v]:
        if visited[i] == False:
            self.DFSUtil(i, visited)

# The function to do DFS traversal. It uses
# recursive DFSUtil()
def DFS(self):
    V = len(self.graph) #total vertices

    # Mark all the vertices as not visited
    visited =[False]*(V)

    # Call the recursive helper function to print
    # DFS traversal starting from all vertices one
    # by one
    for i in range(V):
        if visited[i] == False:
            self.DFSUtil(i, visited)

# Driver code
# Create a graph given in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print "Following is Depth First Traversal"
g.DFS()

# This code is contributed by Neelam Yadav
```

Output:

Following is Depth First Traversal

0 1 2 3

Time Complexity: $O(V+E)$ where V is number of vertices in the graph and E is number of edges in the graph.

- [Applications of DFS.](#)
- [Breadth First Traversal for a Graph](#)
- [Recent Articles on DFS](#)

Source

<https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/>

Chapter 5

Applications of Depth First Search

Applications of Depth First Search - GeeksforGeeks

Depth-first search (DFS) is an algorithm (or technique) for traversing a graph.

Following are the problems that use DFS as a building block.

1) For an unweighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.

2) Detecting cycle in a graph

A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges. (See [this](#) for details)

3) Path Finding

We can specialize the DFS algorithm to find a path between two given vertices u and z.

- i) Call DFS(G, u) with u as the start vertex.
- ii) Use a stack S to keep track of the path between the start vertex and the current vertex.
- iii) As soon as destination vertex z is encountered, return the path as the contents of the stack

See [this](#) for details.

4) Topological Sorting

Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, data serialization, and resolving symbol dependencies in linkers [2].

5) To test if a graph is bipartite

We can augment either BFS or DFS when we first discover a new vertex, color it opposite its parents, and for each other edge, check it doesn't link two vertices of the same color. The first vertex in any connected component can be red or black! See [this](#) for details.

6) Finding Strongly Connected Components of a graph A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex. (See [this](#)for DFS based algo for finding Strongly Connected Components)

7) Solving puzzles with only one solution, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)

Sources:

<http://www8.cs.umu.se/kurser/TDBAfl/VT06/algorithms/LECTUR16/NODE16.HTM>

http://en.wikipedia.org/wiki/Depth-first_search

<http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/depthSearch.htm>

<http://ww3.algorithmdesign.net/handouts/DFS.pdf>

Improved By : JunyiYe

Source

<https://www.geeksforgeeks.org/applications-of-depth-first-search/>

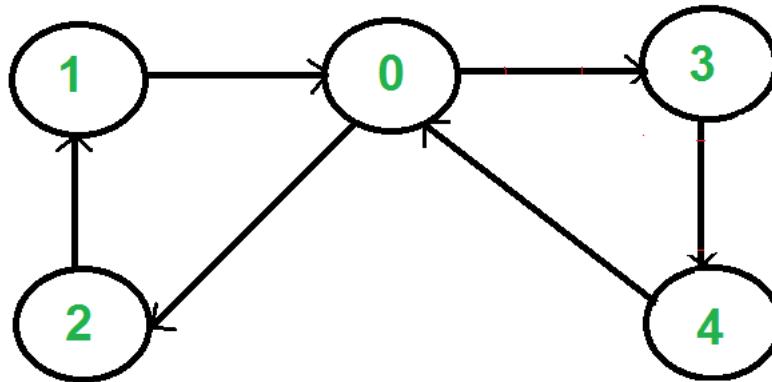
Chapter 6

Iterative Depth First Traversal of Graph

Iterative Depth First Traversal of Graph - GeeksforGeeks

Depth First Traversal (or Search) for a graph is similar to Depth First Traversal (DFS) of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.

For example, a DFS of below graph is “0 3 4 2 1”, other possible DFS is “0 2 1 3 4”.



We have discussed recursive implementation of DFS in previous in [previous post](#). In the post, iterative DFS is discussed. The recursive implementation uses function call stack. In iterative implementation, an explicit stack is used to hold visited vertices.

Below is implementation of Iterative DFS. *The implementation is similar to BFS, the only difference is queue is replaced by stack.*

C++

```
// An Iterative C++ program to do DFS traversal from
// a given source vertex. DFS(int s) traverses vertices
// reachable from s.
#include<bits/stdc++.h>
using namespace std;

// This class represents a directed graph using adjacency
// list representation
class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // adjacency lists
public:
    Graph(int V); // Constructor
    void addEdge(int v, int w); // to add an edge to graph
    void DFS(int s); // prints all vertices in DFS manner
    // from a given source.
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

// prints all not yet visited vertices reachable from s
void Graph::DFS(int s)
{
    // Initially mark all vertices as not visited
    vector<bool> visited(V, false);

    // Create a stack for DFS
    stack<int> stack;

    // Push the current source node.
    stack.push(s);

    while (!stack.empty())
    {
        // Pop a vertex from stack and print it
        s = stack.top();
        stack.pop();
    }
}
```

```
// Stack may contain same vertex twice. So
// we need to print the popped item only
// if it is not visited.
if (!visited[s])
{
    cout << s << " ";
    visited[s] = true;
}

// Get all adjacent vertices of the popped vertex s
// If a adjacent has not been visited, then push it
// to the stack.
for (auto i = adj[s].begin(); i != adj[s].end(); ++i)
    if (!visited[*i])
        stack.push(*i);
}

// Driver program to test methods of graph class
int main()
{
    Graph g(5); // Total 5 vertices in graph
    g.addEdge(1, 0);
    g.addEdge(0, 2);
    g.addEdge(2, 1);
    g.addEdge(0, 3);
    g.addEdge(1, 4);

    cout << "Following is Depth First Traversal\n";
    g.DFS(0);

    return 0;
}
```

Java

```
//An Iterative Java program to do DFS traversal from
//a given source vertex. DFS(int s) traverses vertices
//reachable from s.

import java.util.*;

public class GFG
{
    // This class represents a directed graph using adjacency
    // list representation
    static class Graph
    {
```

```

int V; //Number of Vertices

LinkedList<Integer>[] adj; // adjacency lists

//Constructor
Graph(int V)
{
    this.V = V;
    adj = new LinkedList[V];

    for (int i = 0; i < adj.length; i++)
        adj[i] = new LinkedList<Integer>();

}

//To add an edge to graph
void addEdge(int v, int w)
{
    adj[v].add(w); // Add w to v's list.
}

// prints all not yet visited vertices reachable from s
void DFS(int s)
{
    // Initially mark all vertices as not visited
    Vector<Boolean> visited = new Vector<Boolean>(V);
    for (int i = 0; i < V; i++)
        visited.add(false);

    // Create a stack for DFS
    Stack<Integer> stack = new Stack<>();

    // Push the current source node
    stack.push(s);

    while(stack.empty() == false)
    {
        // Pop a vertex from stack and print it
        s = stack.peek();
        stack.pop();

        // Stack may contain same vertex twice. So
        // we need to print the popped item only
        // if it is not visited.
        if(visited.get(s) == false)
        {
            System.out.print(s + " ");
            visited.set(s, true);
        }
    }
}

```

```

        }

        // Get all adjacent vertices of the popped vertex s
        // If a adjacent has not been visited, then push it
        // to the stack.
        Iterator<Integer> itr = adj[s].iterator();

        while (itr.hasNext())
        {
            int v = itr.next();
            if(!visited.get(v))
                stack.push(v);
        }

    }
}

// Driver program to test methods of graph class
public static void main(String[] args)
{
    // Total 5 vertices in graph
    Graph g = new Graph(5);

    g.addEdge(1, 0);
    g.addEdge(0, 2);
    g.addEdge(2, 1);
    g.addEdge(0, 3);
    g.addEdge(1, 4);

    System.out.println("Following is the Depth First Traversal");
    g.DFS(0);
}
}

```

Output:

```

Following is Depth First Traversal
0 3 2 1 4

```

Note that the above implementation prints only vertices that are reachable from a given vertex. For example, if we remove edges 0-3 and 0-2, the above program would only print 0. To print all vertices of a graph, we need to call DFS for every vertex. Below is implementation for the same.

C++

```

// An Iterative C++ program to do DFS traversal from
// a given source vertex. DFS(int s) traverses vertices
// reachable from s.
#include<bits/stdc++.h>
using namespace std;

// This class represents a directed graph using adjacency
// list representation
class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // adjacency lists
public:
    Graph(int V); // Constructor
    void addEdge(int v, int w); // to add an edge to graph
    void DFS(); // prints all vertices in DFS manner

    // prints all not yet visited vertices reachable from s
    void DFSUtil(int s, vector<bool> &visited);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

// prints all not yet visited vertices reachable from s
void Graph::DFSUtil(int s, vector<bool> &visited)
{
    // Create a stack for DFS
    stack<int> stack;

    // Push the current source node.
    stack.push(s);

    while (!stack.empty())
    {
        // Pop a vertex from stack and print it
        s = stack.top();
        stack.pop();

        // Stack may contain same vertex twice. So

```

```
// we need to print the popped item only
// if it is not visited.
if (!visited[s])
{
    cout << s << " ";
    visited[s] = true;
}

// Get all adjacent vertices of the popped vertex s
// If a adjacent has not been visited, then puah it
// to the stack.
for (auto i = adj[s].begin(); i != adj[s].end(); ++i)
    if (!visited[*i])
        stack.push(*i);
}

// prints all vertices in DFS manner
void Graph::DFS()
{
    // Mark all the vertices as not visited
    vector<bool> visited(V, false);

    for (int i = 0; i < V; i++)
        if (!visited[i])
            DFSUtil(i, visited);
}

// Driver program to test methods of graph class
int main()
{
    Graph g(5); // Total 5 vertices in graph
    g.addEdge(1, 0);
    g.addEdge(2, 1);
    g.addEdge(3, 4);
    g.addEdge(4, 0);

    cout << "Following is Depth First Traversal\n";
    g.DFS();

    return 0;
}
```

Java

```
//An Iterative Java program to do DFS traversal from
//a given source vertex. DFS() traverses vertices
//reachable from s.
```

```
import java.util.*;

public class GFG
{
    // This class represents a directed graph using adjacency
    // list representation
    static class Graph
    {
        int V; //Number of Vertices

        LinkedList<Integer>[] adj; // adjacency lists

        //Constructor
        Graph(int V)
        {
            this.V = V;
            adj = new LinkedList[V];

            for (int i = 0; i < adj.length; i++)
                adj[i] = new LinkedList<Integer>();

        }

        //To add an edge to graph
        void addEdge(int v, int w)
        {
            adj[v].add(w); // Add w to v's list.
        }

        // prints all not yet visited vertices reachable from s
        void DFSUtil(int s, Vector<Boolean> visited)
        {
            // Create a stack for DFS
            Stack<Integer> stack = new Stack<>();

            // Push the current source node
            stack.push(s);

            while(stack.empty() == false)
            {
                // Pop a vertex from stack and print it
                s = stack.peek();
                stack.pop();

                // Stack may contain same vertex twice. So
                // we need to print the popped item only
                // if it is not visited.
            }
        }
    }
}
```

```

        if(visited.get(s) == false)
        {
            System.out.print(s + " ");
            visited.set(s, true);
        }

        // Get all adjacent vertices of the popped vertex s
        // If a adjacent has not been visited, then puah it
        // to the stack.
        Iterator<Integer> itr = adj[s].iterator();

        while (itr.hasNext())
        {
            int v = itr.next();
            if(!visited.get(v))
                stack.push(v);
        }

    }

// prints all vertices in DFS manner
void DFS()
{
    Vector<Boolean> visited = new Vector<Boolean>(V);
    // Mark all the vertices as not visited
    for (int i = 0; i < V; i++)
        visited.add(false);

    for (int i = 0; i < V; i++)
        if (!visited.get(i))
            DFSUtil(i, visited);
}

// Driver program to test methods of graph class
public static void main(String[] args)
{
    Graph g = new Graph(5);
    g.addEdge(1, 0);
    g.addEdge(2, 1);
    g.addEdge(3, 4);
    g.addEdge(4, 0);

    System.out.println("Following is Depth First Traversal");
    g.DFS();
}
}
    
```

Output:

```
Following is Depth First Traversal
0 1 2 3 4
```

Like recursive traversal, time complexity of iterative implementation is $O(V + E)$.

This article is contributed by **Shivam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

<https://www.geeksforgeeks.org/iterative-depth-first-traversal/>

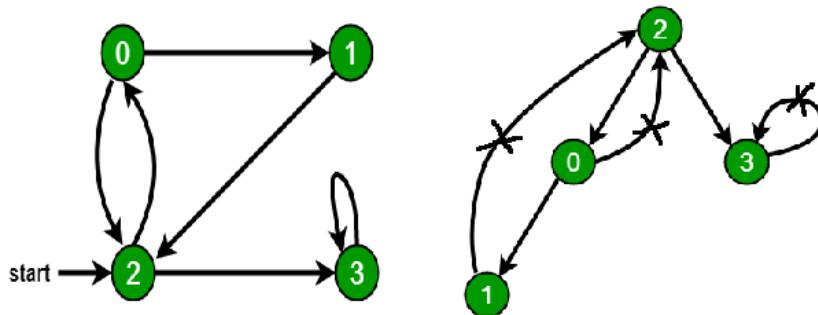
Chapter 7

Detect Cycle in a Directed Graph

Detect Cycle in a Directed Graph - GeeksforGeeks

Given a directed graph, check whether the graph contains a cycle or not. Your function should return true if the given graph contains at least one cycle, else return false. For example, the following graph contains three cycles $0 \rightarrow 2 \rightarrow 0$, $0 \rightarrow 1 \rightarrow 2 \rightarrow 0$ and $3 \rightarrow 3$, so your function must return true.

Depth First Traversal can be used to detect a cycle in a Graph. DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a [back edge](#) present in the graph. A back edge is an edge that is from a node to itself (self-loop) or one of its ancestor in the tree produced by DFS. In the following graph, there are 3 back edges, marked with a cross sign. We can observe that these 3 back edges indicate 3 cycles present in the graph.



For a disconnected graph, we get the DFS forest as output. To detect cycle, we can check for a cycle in individual trees by checking back edges.

To detect a back edge, we can keep track of vertices currently in recursion stack of function for DFS traversal. If we reach a vertex that is already in the recursion stack, then there is a cycle in the tree. The edge that connects current vertex to the vertex in the recursion

stack is a back edge. We have used recStack[] array to keep track of vertices in the recursion stack.

C++

```
// A C++ Program to detect cycle in a graph
#include<iostream>
#include <list>
#include <limits.h>

using namespace std;

class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // Pointer to an array containing adjacency lists
    bool isCyclicUtil(int v, bool visited[], bool *rs); // used by isCyclic()
public:
    Graph(int V);    // Constructor
    void addEdge(int v, int w);    // to add an edge to graph
    bool isCyclic();    // returns true if there is a cycle in this graph
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

// This function is a variation of DFSUtil() in https://www.geeksforgeeks.org/archives/18212
bool Graph::isCyclicUtil(int v, bool visited[], bool *recStack)
{
    if(visited[v] == false)
    {
        // Mark the current node as visited and part of recursion stack
        visited[v] = true;
        recStack[v] = true;

        // Recur for all the vertices adjacent to this vertex
        list<int>::iterator i;
        for(i = adj[v].begin(); i != adj[v].end(); ++i)
        {
            if ( !visited[*i] && isCyclicUtil(*i, visited, recStack) )
                return true;
        }
    }
    return false;
}
```

```

        else if (recStack[*i])
            return true;
    }

}

recStack[v] = false; // remove the vertex from recursion stack
return false;
}

// Returns true if the graph contains a cycle, else false.
// This function is a variation of DFS() in https://www.geeksforgeeks.org/archives/18212
bool Graph::isCyclic()
{
    // Mark all the vertices as not visited and not part of recursion
    // stack
    bool *visited = new bool[V];
    bool *recStack = new bool[V];
    for(int i = 0; i < V; i++)
    {
        visited[i] = false;
        recStack[i] = false;
    }

    // Call the recursive helper function to detect cycle in different
    // DFS trees
    for(int i = 0; i < V; i++)
        if (isCyclicUtil(i, visited, recStack))
            return true;

    return false;
}

int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    if(g.isCyclic())
        cout << "Graph contains cycle";
    else
        cout << "Graph doesn't contain cycle";
    return 0;
}

```

```
}
```

Java

```
// A Java Program to detect cycle in a graph
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

class Graph {

    private final int V;
    private final List<List<Integer>> adj;

    public Graph(int V)
    {
        this.V = V;
        adj = new ArrayList<>(V);

        for (int i = 0; i < V; i++)
            adj.add(new LinkedList<>());
    }

    // This function is a variation of DFSUtil() in
    // https://www.geeksforgeeks.org/archives/18212
    private boolean isCyclicUtil(int i, boolean[] visited,
                                 boolean[] recStack)
    {

        // Mark the current node as visited and
        // part of recursion stack
        if (recStack[i])
            return true;

        if (visited[i])
            return false;

        visited[i] = true;

        recStack[i] = true;
        List<Integer> children = adj.get(i);

        for (Integer c: children)
            if (isCyclicUtil(c, visited, recStack))
                return true;

        recStack[i] = false;
    }
}
```

```
        return false;
    }

private void addEdge(int source, int dest) {
    adj.get(source).add(dest);
}

// Returns true if the graph contains a
// cycle, else false.
// This function is a variation of DFS() in
// https://www.geeksforgeeks.org/archives/18212
private boolean isCyclic()
{
    // Mark all the vertices as not visited and
    // not part of recursion stack
    boolean[] visited = new boolean[V];
    boolean[] recStack = new boolean[V];

    // Call the recursive helper function to
    // detect cycle in different DFS trees
    for (int i = 0; i < V; i++)
        if (isCyclicUtil(i, visited, recStack))
            return true;

    return false;
}

// Driver code
public static void main(String[] args)
{
    Graph graph = new Graph(4);
    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 2);
    graph.addEdge(2, 0);
    graph.addEdge(2, 3);
    graph.addEdge(3, 3);

    if(graph.isCyclic())
        System.out.println("Graph contains cycle");
    else
        System.out.println("Graph doesn't "
                           + "contain cycle");
}
}
```

```
// This code is contributed by Sagar Shah.
```

Python

```
# Python program to detect cycle
# in a graph

from collections import defaultdict

class Graph():
    def __init__(self,vertices):
        self.graph = defaultdict(list)
        self.V = vertices

    def addEdge(self,u,v):
        self.graph[u].append(v)

    def isCyclicUtil(self, v, visited, recStack):

        # Mark current node as visited and
        # adds to recursion stack
        visited[v] = True
        recStack[v] = True

        # Recur for all neighbours
        # if any neighbour is visited and in
        # recStack then graph is cyclic
        for neighbour in self.graph[v]:
            if visited[neighbour] == False:
                if self.isCyclicUtil(neighbour, visited, recStack) == True:
                    return True
            elif recStack[neighbour] == True:
                return True

        # The node needs to be popped from
        # recursion stack before function ends
        recStack[v] = False
        return False

    # Returns true if graph is cyclic else false
    def isCyclic(self):
        visited = [False] * self.V
        recStack = [False] * self.V
        for node in range(self.V):
            if visited[node] == False:
                if self.isCyclicUtil(node,visited,recStack) == True:
                    return True
        return False
```

```
g = Graph(4)
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)
if g.isCyclic() == 1:
    print "Graph has a cycle"
else:
    print "Graph has no cycle"

# Thanks to Divyanshu Mehta for contributing this code
```

Output:

```
Graph contains cycle
```

Time Complexity of this method is same as time complexity of [DFS traversal](#) which is $O(V+E)$.

In the below article, another $O(V + E)$ method is discussed :
[Detect Cycle in a direct graph using colors](#)

Improved By : [SagarShah1](#)

Source

<https://www.geeksforgeeks.org/detect-cycle-in-a-graph/>

Chapter 8

Union Find - (Detect Cycle in an Undirected Graph)

Disjoint Set (Or Union-Find) Set 1 (Detect Cycle in an Undirected Graph) - GeeksforGeeks

A [*disjoint-set data structure*](#) is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. A [*union-find algorithm*](#) is an algorithm that performs two useful operations on such a data structure:

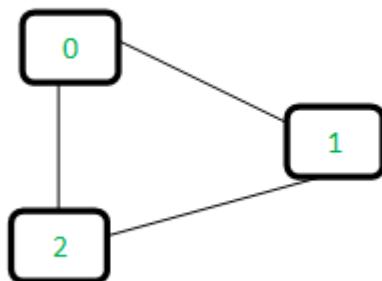
Find: Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.

Union: Join two subsets into a single subset.

In this post, we will discuss an application of Disjoint Set Data Structure. The application is to check whether a given graph contains a cycle or not.

Union-Find Algorithm can be used to check whether an undirected graph contains cycle or not. Note that we have discussed an [*algorithm to detect cycle*](#). This is another method based on *Union-Find*. This method assumes that graph doesn't contain any self-loops. We can keep track of the subsets in a 1D array, let's call it `parent[]`.

Let us consider the following graph:



For each edge, make subsets using both the vertices of the edge. If both the vertices are in the same subset, a cycle is found.

Initially, all slots of parent array are initialized to -1 (means there is only one item in every subset).

```
0   1   2
-1  -1  -1
```

Now process all edges one by one.

Edge 0-1: Find the subsets in which vertices 0 and 1 are. Since they are in different subsets, we take the union of them. For taking the union, either make node 0 as parent of node 1 or vice-versa.

```
0   1   2      <----- 1 is made parent of 0 (1 is now representative of subset {0, 1})
1   -1  -1
```

Edge 1-2: 1 is in subset 1 and 2 is in subset 2. So, take union.

```
0   1   2      <----- 2 is made parent of 1 (2 is now representative of subset {0, 1, 2})
1   2   -1
```

Edge 0-2: 0 is in subset 2 and 2 is also in subset 2. Hence, including this edge forms a cycle.

How subset of 0 is same as 2?

0->1->2 // 1 is parent of 0 and 2 is parent of 1

Based on the above explanation, below are implementations:

C/C++

```
// A union-find algorithm to detect cycle in a graph
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// a structure to represent an edge in graph
struct Edge
{
    int src, dest;
};

// a structure to represent a graph
struct Graph
{
```

```
// V-> Number of vertices, E-> Number of edges
int V, E;

// graph is represented as an array of edges
struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph =
        (struct Graph*) malloc( sizeof(struct Graph) );
    graph->V = V;
    graph->E = E;

    graph->edge =
        (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );

    return graph;
}

// A utility function to find the subset of an element i
int find(int parent[], int i)
{
    if (parent[i] == -1)
        return i;
    return find(parent, parent[i]);
}

// A utility function to do union of two subsets
void Union(int parent[], int x, int y)
{
    int xset = find(parent, x);
    int yset = find(parent, y);
    if(xset!=yset){
        parent[xset] = yset;
    }
}

// The main function to check whether a given graph contains
// cycle or not
int isCycle( struct Graph* graph )
{
    // Allocate memory for creating V subsets
    int *parent = (int*) malloc( graph->V * sizeof(int) );

    // Initialize all subsets as single element sets
    memset(parent, -1, sizeof(int) * graph->V);
```

```

// Iterate through all edges of graph, find subset of both
// vertices of every edge, if both subsets are same, then
// there is cycle in graph.
for(int i = 0; i < graph->E; ++i)
{
    int x = find(parent, graph->edge[i].src);
    int y = find(parent, graph->edge[i].dest);

    if (x == y)
        return 1;

    Union(parent, x, y);
}
return 0;
}

// Driver program to test above functions
int main()
{
    /* Let us create following graph
       0
       |
       |
       1----2 */
    int V = 3, E = 3;
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;

    // add edge 1-2
    graph->edge[1].src = 1;
    graph->edge[1].dest = 2;

    // add edge 0-2
    graph->edge[2].src = 0;
    graph->edge[2].dest = 2;

    if (isCycle(graph))
        printf( "graph contains cycle" );
    else
        printf( "graph doesn't contain cycle" );

    return 0;
}

```

Java

```
// Java Program for union-find algorithm to detect cycle in a graph
import java.util.*;
import java.lang.*;
import java.io.*;

class Graph
{
    int V, E;      // V-> no. of vertices & E->no.of edges
    Edge edge[]; // /collection of all edges

    class Edge
    {
        int src, dest;
    };

    // Creates a graph with V vertices and E edges
    Graph(int v,int e)
    {
        V = v;
        E = e;
        edge = new Edge[E];
        for (int i=0; i<e; ++i)
            edge[i] = new Edge();
    }

    // A utility function to find the subset of an element i
    int find(int parent[], int i)
    {
        if (parent[i] == -1)
            return i;
        return find(parent, parent[i]);
    }

    // A utility function to do union of two subsets
    void Union(int parent[], int x, int y)
    {
        int xset = find(parent, x);
        int yset = find(parent, y);
        parent[xset] = yset;
    }

    // The main function to check whether a given graph
    // contains cycle or not
    int isCycle( Graph graph)
    {
```

```
// Allocate memory for creating V subsets
int parent[] = new int[graph.V];

// Initialize all subsets as single element sets
for (int i=0; i<graph.V; ++i)
    parent[i]=-1;

// Iterate through all edges of graph, find subset of both
// vertices of every edge, if both subsets are same, then
// there is cycle in graph.
for (int i = 0; i < graph.E; ++i)
{
    int x = graph.find(parent, graph.edge[i].src);
    int y = graph.find(parent, graph.edge[i].dest);

    if (x == y)
        return 1;

    graph.Union(parent, x, y);
}
return 0;
}

// Driver Method
public static void main (String[] args)
{
    /* Let us create following graph
     0
     | \
     |   \
     1----2 */
    int V = 3, E = 3;
    Graph graph = new Graph(V, E);

    // add edge 0-1
    graph.edge[0].src = 0;
    graph.edge[0].dest = 1;

    // add edge 1-2
    graph.edge[1].src = 1;
    graph.edge[1].dest = 2;

    // add edge 0-2
    graph.edge[2].src = 0;
    graph.edge[2].dest = 2;

    if (graph.isCycle(graph)==1)
        System.out.println( "graph contains cycle" );
}
```

```

        else
            System.out.println( "graph doesn't contain cycle" );
    }
}

```

Python

```

# Python Program for union-find algorithm to detect cycle in a undirected graph
# we have one edge for any two vertex i.e 1-2 is either 1-2 or 2-1 but not both

from collections import defaultdict

#This class represents a undirected graph using adjacency list representation
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices
        self.graph = defaultdict(list) # default dictionary to store graph

        # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # A utility function to find the subset of an element i
    def find_parent(self, parent,i):
        if parent[i] == -1:
            return i
        if parent[i]!= -1:
            return self.find_parent(parent,parent[i])

    # A utility function to do union of two subsets
    def union(self,parent,x,y):
        x_set = self.find_parent(parent, x)
        y_set = self.find_parent(parent, y)
        parent[x_set] = y_set

    # The main function to check whether a given graph
    # contains cycle or not
    def isCyclic(self):

        # Allocate memory for creating V subsets and
        # Initialize all subsets as single element sets
        parent = [-1]*(self.V)

        # Iterate through all edges of graph, find subset of both

```

```
# vertices of every edge, if both subsets are same, then
# there is cycle in graph.
for i in self.graph:
    for j in self.graph[i]:
        x = self.find_parent(parent, i)
        y = self.find_parent(parent, j)
        if x == y:
            return True
        self.union(parent,x,y)

# Create a graph given in the above diagram
g = Graph(3)
g.addEdge(0, 1)
g.addEdge(1, 2)
g.addEdge(2, 0)

if g.isCyclic():
    print "Graph contains cycle"
else :
    print "Graph does not contain cycle "

#This code is contributed by Neelam Yadav
```

Output:

```
graph contains cycle
```

Note that the implementation of *union()* and *find()* is naive and takes $O(n)$ time in worst case. These methods can be improved to $O(\log n)$ using *Union by Rank or Height*. We will soon be discussing *Union by Rank* in a separate post.

Related Articles :

[Union-Find Algorithm Set 2 \(Union By Rank and Path Compression\)](#)

[Disjoint Set Data Structures \(Java Implementation\)](#)

[Greedy Algorithms Set 2 \(Kruskal's Minimum Spanning Tree Algorithm\)](#)

[Job Sequencing Problem Set 2 \(Using Disjoint Set\)](#)

This article is compiled by [Aashish Barnwal](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Improved By : [avinashr175](#)

Source

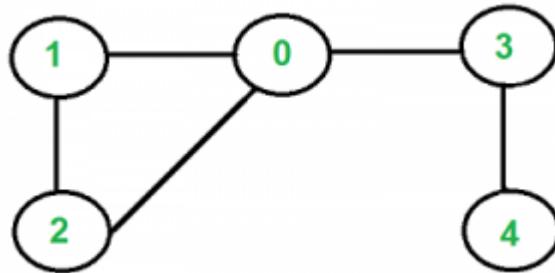
<https://www.geeksforgeeks.org/union-find/>

Chapter 9

Detect cycle in an Undirected graph

Detect cycle in an undirected graph - GeeksforGeeks

Given an undirected graph, how to check if there is a cycle in the graph? For example, the following graph has a cycle 1-0-2-1.



We have discussed [cycle detection for directed graph](#). We have also discussed a [union-find algorithm for cycle detection in undirected graphs](#). The time complexity of the union-find algorithm is $O(E\log V)$. Like directed graphs, we can use [DFS](#) to detect cycle in an undirected graph in $O(V+E)$ time. We do a DFS traversal of the given graph. For every visited vertex 'v', if there is an adjacent 'u' such that u is already visited and u is not parent of v, then there is a cycle in graph. If we don't find such an adjacent for any vertex, we say that there is no cycle. The assumption of this approach is that there are no parallel edges between any two vertices.

C++

```
// A C++ Program to detect cycle in an undirected graph
#include<iostream>
#include <list>
#include <limits.h>
```

```

using namespace std;

// Class for an undirected graph
class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // Pointer to an array containing adjacency lists
    bool isCyclicUtil(int v, bool visited[], int parent);

public:
    Graph(int V);    // Constructor
    void addEdge(int v, int w);    // to add an edge to graph
    bool isCyclic();    // returns true if there is a cycle
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
    adj[w].push_back(v); // Add v to w's list.
}

// A recursive function that uses visited[] and parent to detect
// cycle in subgraph reachable from vertex v.
bool Graph::isCyclicUtil(int v, bool visited[], int parent)
{
    // Mark the current node as visited
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
    {
        // If an adjacent is not visited, then recur for that adjacent
        if (!visited[*i])
        {
            if (isCyclicUtil(*i, visited, v))
                return true;
        }
    }

    // If an adjacent is visited and not parent of current vertex,
    // then there is a cycle.
    else if (*i != parent)
        return true;
}

```

```
        }
        return false;
    }

// Returns true if the graph contains a cycle, else false.
bool Graph::isCyclic()
{
    // Mark all the vertices as not visited and not part of recursion
    // stack
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to detect cycle in different
    // DFS trees
    for (int u = 0; u < V; u++)
        if (!visited[u]) // Don't recur for u if it is already visited
            if (isCyclicUtil(u, visited, -1))
                return true;

    return false;
}

// Driver program to test above functions
int main()
{
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 0);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.isCyclic()? cout << "Graph contains cycle\n":
                  cout << "Graph doesn't contain cycle\n";

    Graph g2(3);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.isCyclic()? cout << "Graph contains cycle\n":
                  cout << "Graph doesn't contain cycle\n";

    return 0;
}
```

Java

```
// A Java Program to detect cycle in an undirected graph
import java.io.*;
```

```
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    private int V;    // No. of vertices
    private LinkedList<Integer> adj[]; // Adjacency List Representation

    // Constructor
    Graph(int v) {
        V = v;
        adj = new LinkedList[v];
        for(int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    // Function to add an edge into the graph
    void addEdge(int v,int w) {
        adj[v].add(w);
        adj[w].add(v);
    }

    // A recursive function that uses visited[] and parent to detect
    // cycle in subgraph reachable from vertex v.
    Boolean isCyclicUtil(int v, Boolean visited[], int parent)
    {
        // Mark the current node as visited
        visited[v] = true;
        Integer i;

        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> it = adj[v].iterator();
        while (it.hasNext())
        {
            i = it.next();

            // If an adjacent is not visited, then recur for that
            // adjacent
            if (!visited[i])
            {
                if (isCyclicUtil(i, visited, v))
                    return true;
            }
        }

        // If an adjacent is visited and not parent of current
        // vertex, then there is a cycle.
        else if (i != parent)
```

```
        return true;
    }
    return false;
}

// Returns true if the graph contains a cycle, else false.
Boolean isCyclic()
{
    // Mark all the vertices as not visited and not part of
    // recursion stack
    Boolean visited[] = new Boolean[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to detect cycle in
    // different DFS trees
    for (int u = 0; u < V; u++)
        if (!visited[u]) // Don't recur for u if already visited
            if (isCyclicUtil(u, visited, -1))
                return true;

    return false;
}

// Driver method to test above methods
public static void main(String args[])
{
    // Create a graph given in the above diagram
    Graph g1 = new Graph(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 0);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    if (g1.isCyclic())
        System.out.println("Graph contains cycle");
    else
        System.out.println("Graph doesn't contain cycle");

    Graph g2 = new Graph(3);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    if (g2.isCyclic())
        System.out.println("Graph contains cycle");
    else
        System.out.println("Graph doesn't contain cycle");
}
```

```
}
```

// This code is contributed by Aakash Hasija

Python

```
# Python Program to detect cycle in an undirected graph

from collections import defaultdict

#This class represents a undirected graph using adjacency list representation
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices
        self.graph = defaultdict(list) # default dictionary to store graph

        # function to add an edge to graph
    def addEdge(self,v,w):
        self.graph[v].append(w) #Add w to v_s list
        self.graph[w].append(v) #Add v to w_s list

    # A recursive function that uses visited[] and parent to detect
    # cycle in subgraph reachable from vertex v.
    def isCyclicUtil(self,v,visited,parent):

        #Mark the current node as visited
        visited[v]= True

        #Recur for all the vertices adjacent to this vertex
        for i in self.graph[v]:
            # If the node is not visited then recurse on it
            if  visited[i]==False :
                if(self.isCyclicUtil(i,visited,v)):
                    return True
            # If an adjacent vertex is visited and not parent of current vertex,
            # then there is a cycle
            elif parent!=i:
                return True

        return False

    #Returns true if the graph contains a cycle, else false.
    def isCyclic(self):
        # Mark all the vertices as not visited
        visited =[False]*(self.V)
        # Call the recursive helper function to detect cycle in different
```

```
#DFS trees
for i in range(self.V):
    if visited[i] ==False: #Don't recur for u if it is already visited
        if(self.isCyclicUtil(i,visited,-1))== True:
            return True

    return False

# Create a graph given in the above diagram
g = Graph(5)
g.addEdge(1, 0)
g.addEdge(0, 2)
g.addEdge(2, 0)
g.addEdge(0, 3)
g.addEdge(3, 4)

if g.isCyclic():
    print "Graph contains cycle"
else :
    print "Graph does not contain cycle "
g1 = Graph(3)
g1.addEdge(0,1)
g1.addEdge(1,2)

if g1.isCyclic():
    print "Graph contains cycle"
else :
    print "Graph does not contain cycle "

#This code is contributed by Neelam Yadav
```

Output:

```
Graph contains cycle
Graph doesn't contain cycle
```

Time Complexity: The program does a simple DFS Traversal of graph and graph is represented using adjacency list. So the time complexity is $O(V+E)$

Exercise: Can we use BFS to detect cycle in an undirected graph in $O(V+E)$ time? What about directed graphs?

Source

<https://www.geeksforgeeks.org/detect-cycle-undirected-graph/>

Chapter 10

Longest Path in a Directed Acyclic Graph

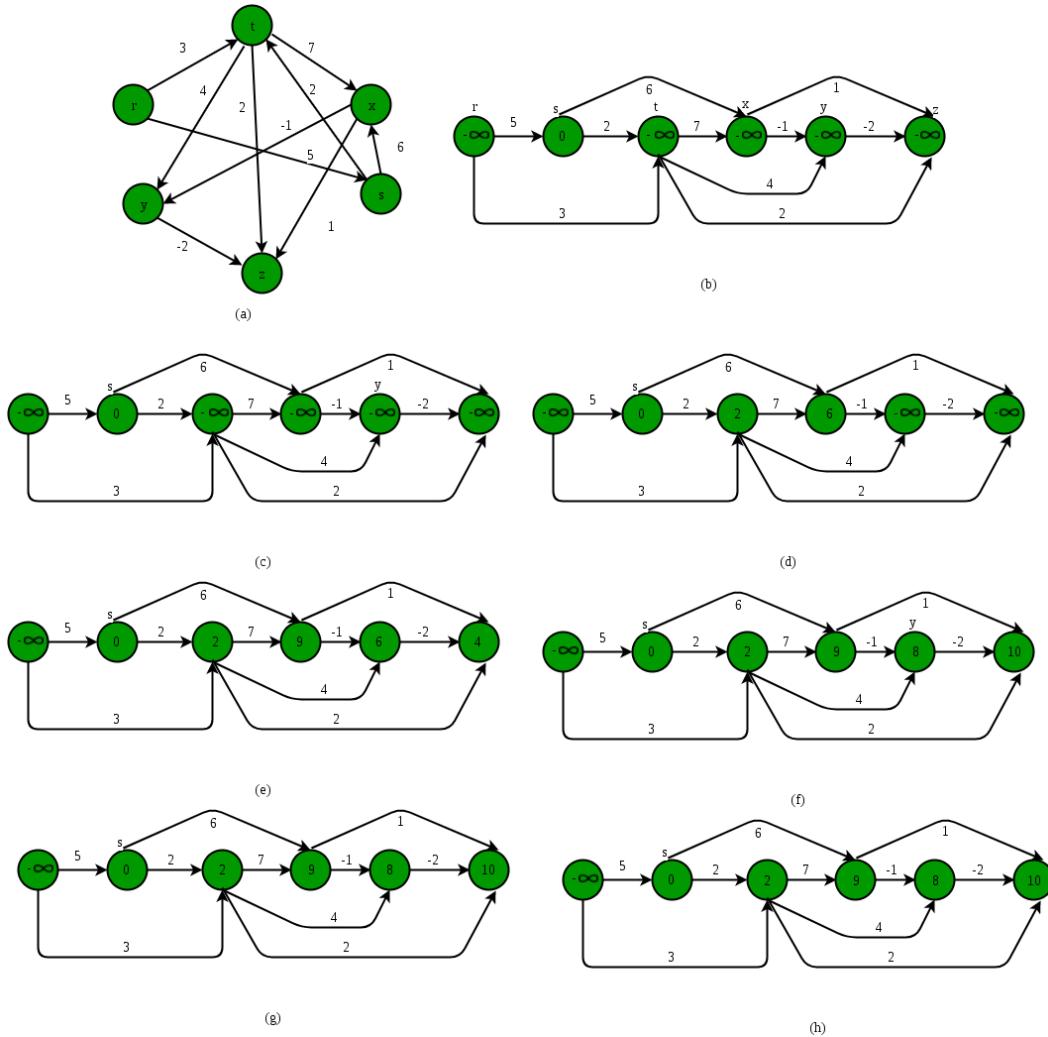
Longest Path in a Directed Acyclic Graph - GeeksforGeeks

Given a Weighted **Directed Acyclic Graph** (DAG) and a source vertex s in it, find the longest distances from s to all other vertices in the given graph.

The longest path problem for a general graph is not as easy as the shortest path problem because the longest path problem doesn't have [optimal substructure property](#). In fact, [the Longest Path problem is NP-Hard for a general graph](#). However, the longest path problem has a linear time solution for directed acyclic graphs. The idea is similar to [linear time solution for shortest path in a directed acyclic graph](#)., we use [Topological Sorting](#).

We initialize distances to all vertices as minus infinite and distance to source as 0, then we find a [topological sorting](#) of the graph. Topological Sorting of a graph represents a linear ordering of the graph (See below, figure (b) is a linear representation of figure (a)). Once we have topological order (or linear representation), we one by one process all vertices in topological order. For every vertex being processed, we update distances of its adjacent using distance of current vertex.

Following figure shows step by step process of finding longest paths.



Following is complete algorithm for finding longest distances.

1) Initialize $\text{dist}[] = \{\text{NINF}, \text{NINF}, \dots\}$ and $\text{dist}[s] = 0$ where s is the source vertex. Here NINF means negative infinite.

2) Create a topological order of all vertices.

3) Do following for every vertex u in topological order.

.....Do following for every adjacent vertex v of u

.....if ($\text{dist}[v] < \text{dist}[u] + \text{weight}(u, v)$)

..... $\text{dist}[v] = \text{dist}[u] + \text{weight}(u, v)$

Following is C++ implementation of the above algorithm.

```

// A C++ program to find single source longest distances
// in a DAG
#include <iostream>
#include <limits.h>

```

```

#include <list>
#include <stack>
#define NINF INT_MIN
using namespace std;

// Graph is represented using adjacency list. Every
// node of adjacency list contains vertex number of
// the vertex to which edge connects. It also
// contains weight of the edge
class AdjListNode {
    int v;
    int weight;

public:
    AdjListNode(int _v, int _w)
    {
        v = _v;
        weight = _w;
    }
    int getV() { return v; }
    int getWeight() { return weight; }
};

// Class to represent a graph using adjacency list
// representation
class Graph {
    int V; // No. of vertices'

    // Pointer to an array containing adjacency lists
    list<AdjListNode*>* adj;

    // A function used by longestPath
    void topologicalSortUtil(int v, bool visited[],
                           stack<int>& Stack);

public:
    Graph(int V); // Constructor

    // function to add an edge to graph
    void addEdge(int u, int v, int weight);

    // Finds longest distances from given source vertex
    void longestPath(int s);
};

Graph::Graph(int V) // Constructor
{
    this->V = V;
}

```

```

        adj = new list<AdjListNode>[V];
    }

void Graph::addEdge(int u, int v, int weight)
{
    AdjListNode node(v, weight);
    adj[u].push_back(node); // Add v to u's list
}

// A recursive function used by longestPath. See below
// link for details
// https://www.geeksforgeeks.org/topological-sorting/
void Graph::topologicalSortUtil(int v, bool visited[],
                                stack<int>& Stack)
{
    // Mark the current node as visited
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<AdjListNode>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i) {
        AdjListNode node = *i;
        if (!visited[node.getV()])
            topologicalSortUtil(node.getV(), visited, Stack);
    }

    // Push current vertex to stack which stores topological
    // sort
    Stack.push(v);
}

// The function to find longest distances from a given vertex.
// It uses recursive topologicalSortUtil() to get topological
// sorting.
void Graph::longestPath(int s)
{
    stack<int> Stack;
    int dist[V];

    // Mark all the vertices as not visited
    bool* visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to store Topological
    // Sort starting from all vertices one by one
    for (int i = 0; i < V; i++)
        if (visited[i] == false)

```

```

        topologicalSortUtil(i, visited, Stack);

        // Initialize distances to all vertices as infinite and
        // distance to source as 0
        for (int i = 0; i < V; i++)
            dist[i] = NINF;
        dist[s] = 0;

        // Process vertices in topological order
        while (Stack.empty() == false) {
            // Get the next vertex from topological order
            int u = Stack.top();
            Stack.pop();

            // Update distances of all adjacent vertices
            list<AdjListNode>::iterator i;
            if (dist[u] != NINF) {
                for (i = adj[u].begin(); i != adj[u].end(); ++i)
                    if (dist[i->getV()] < dist[u] + i->getWeight())
                        dist[i->getV()] = dist[u] + i->getWeight();
            }
        }

        // Print the calculated longest distances
        for (int i = 0; i < V; i++)
            (dist[i] == NINF) ? cout << "INF " : cout << dist[i] << " ";
    }

    // Driver program to test above functions
    int main()
    {
        // Create a graph given in the above diagram.
        // Here vertex numbers are 0, 1, 2, 3, 4, 5 with
        // following mappings:
        // 0=r, 1=s, 2=t, 3=x, 4=y, 5=z
        Graph g(6);
        g.addEdge(0, 1, 5);
        g.addEdge(0, 2, 3);
        g.addEdge(1, 3, 6);
        g.addEdge(1, 2, 2);
        g.addEdge(2, 4, 4);
        g.addEdge(2, 5, 2);
        g.addEdge(2, 3, 7);
        g.addEdge(3, 5, 1);
        g.addEdge(3, 4, -1);
        g.addEdge(4, 5, -2);

        int s = 1;
    }
}

```

```
cout << "Following are longest distances from "
      "source vertex "
      << s << "\n";
g.longestPath(s);

return 0;
}
```

Output:

```
Following are longest distances from source vertex 1
INF 0 2 9 8 10
```

Time Complexity: Time complexity of topological sorting is $O(V+E)$. After finding topological order, the algorithm process all vertices and for every vertex, it runs a loop for all adjacent vertices. Total adjacent vertices in a graph is $O(E)$. So the inner loop runs $O(V+E)$ times. Therefore, overall time complexity of this algorithm is $O(V+E)$.

Exercise: The above solution print longest distances, extend the code to print paths also.

Micro Focus

Improved By : [Rohit0803](#), [pranith maddineni](#)

Source

<https://www.geeksforgeeks.org/find-longest-path-directed-acyclic-graph/>

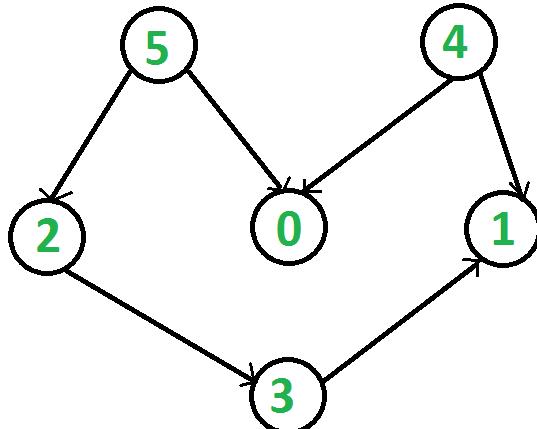
Chapter 11

Topological Sorting

Topological Sorting - GeeksforGeeks

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv , vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, a topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 3 1 0”. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges).



Topological Sorting vs Depth First Traversal (DFS):

In [DFS](#), we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex ‘5’ should be printed before vertex ‘0’, but unlike [DFS](#), the vertex ‘4’ should also be printed before vertex ‘0’. So Topological sorting is different from DFS. For example, a DFS of the shown graph is “5 2 3 1 0 4”, but it is not a topological sorting

Algorithm to find Topological Sorting:

We recommend to first see implementation of DFS [here](#). We can modify [DFS](#) to find Topological Sorting of a graph. In [DFS](#), we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don't print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in stack.

Following are the implementations of topological sorting. Please see the code for [Depth First Traversal for a disconnected Graph](#) and note the differences between the second code given there and the below code.

C++

```
// A C++ program to print topological sorting of a DAG
#include<iostream>
#include <list>
#include <stack>
using namespace std;

// Class to represent a graph
class Graph
{
    int V;      // No. of vertices'

    // Pointer to an array containing adjacency listsList
    list<int> *adj;

    // A function used by topologicalSort
    void topologicalSortUtil(int v, bool visited[], stack<int> &Stack);

public:
    Graph(int V);    // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // prints a Topological Sort of the complete graph
    void topologicalSort();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
```

```
}

// A recursive function used by topologicalSort
void Graph::topologicalSortUtil(int v, bool visited[],
                                stack<int> &Stack)
{
    // Mark the current node as visited.
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            topologicalSortUtil(*i, visited, Stack);

    // Push current vertex to stack which stores result
    Stack.push(v);
}

// The function to do Topological Sort. It uses recursive
// topologicalSortUtil()
void Graph::topologicalSort()
{
    stack<int> Stack;

    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to store Topological
    // Sort starting from all vertices one by one
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            topologicalSortUtil(i, visited, Stack);

    // Print contents of stack
    while (Stack.empty() == false)
    {
        cout << Stack.top() << " ";
        Stack.pop();
    }
}

// Driver program to test above functions
int main()
{
    // Create a graph given in the above diagram
```

```
Graph g(6);
g.addEdge(5, 2);
g.addEdge(5, 0);
g.addEdge(4, 0);
g.addEdge(4, 1);
g.addEdge(2, 3);
g.addEdge(3, 1);

cout << "Following is a Topological Sort of the given graph \n";
g.topologicalSort();

return 0;
}
```

Java

```
// A Java program to print topological sorting of a DAG
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency
// list representation
class Graph
{
    private int V;    // No. of vertices
    private LinkedList<Integer> adj[]; // Adjacency List

    //Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    // Function to add an edge into the graph
    void addEdge(int v,int w) { adj[v].add(w); }

    // A recursive function used by topologicalSort
    void topologicalSortUtil(int v, boolean visited[],
                           Stack stack)
    {
        // Mark the current node as visited.
        visited[v] = true;
        Integer i;

        // Recur for all the vertices adjacent to this
```

```
// vertex
Iterator<Integer> it = adj[v].iterator();
while (it.hasNext())
{
    i = it.next();
    if (!visited[i])
        topologicalSortUtil(i, visited, stack);
}

// Push current vertex to stack which stores result
stack.push(new Integer(v));
}

// The function to do Topological Sort. It uses
// recursive topologicalSortUtil()
void topologicalSort()
{
    Stack stack = new Stack();

    // Mark all the vertices as not visited
    boolean visited[] = new boolean[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to store
    // Topological Sort starting from all vertices
    // one by one
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            topologicalSortUtil(i, visited, stack);

    // Print contents of stack
    while (stack.empty() == false)
        System.out.print(stack.pop() + " ");
}

// Driver method
public static void main(String args[])
{
    // Create a graph given in the above diagram
    Graph g = new Graph(6);
    g.addEdge(5, 2);
    g.addEdge(5, 0);
    g.addEdge(4, 0);
    g.addEdge(4, 1);
    g.addEdge(2, 3);
    g.addEdge(3, 1);
```

```

        System.out.println("Following is a Topological " +
                           "sort of the given graph");
        g.topologicalSort();
    }
}

// This code is contributed by Aakash Hasija

```

Python

```

#Python program to print topological sorting of a DAG
from collections import defaultdict

#Class to represent a graph
class Graph:
    def __init__(self,vertices):
        self.graph = defaultdict(list) #dictionary containing adjacency List
        self.V = vertices #No. of vertices

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # A recursive function used by topologicalSort
    def topologicalSortUtil(self,v,visited,stack):

        # Mark the current node as visited.
        visited[v] = True

        # Recur for all the vertices adjacent to this vertex
        for i in self.graph[v]:
            if visited[i] == False:
                self.topologicalSortUtil(i,visited,stack)

        # Push current vertex to stack which stores result
        stack.insert(0,v)

    # The function to do Topological Sort. It uses recursive
    # topologicalSortUtil()
    def topologicalSort(self):
        # Mark all the vertices as not visited
        visited = [False]*self.V
        stack = []

        # Call the recursive helper function to store Topological
        # Sort starting from all vertices one by one
        for i in range(self.V):
            if visited[i] == False:
                self.topologicalSortUtil(i,visited,stack)

```

```
# Print contents of the stack
print stack

g= Graph(6)
g.addEdge(5, 2);
g.addEdge(5, 0);
g.addEdge(4, 0);
g.addEdge(4, 1);
g.addEdge(2, 3);
g.addEdge(3, 1);

print "Following is a Topological Sort of the given graph"
g.topologicalSort()
#This code is contributed by Neelam Yadav
```

Output:

```
Following is a Topological Sort of the given graph
5 4 2 3 1 0
```

Time Complexity: The above algorithm is simply DFS with an extra stack. So time complexity is same as DFS which is $O(V+E)$.

Applications:

Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, data serialization, and resolving symbol dependencies in linkers [2].

Related Articles:

[Kahn's algorithm for Topological Sorting](#) : Another $O(V + E)$ algorithm.
[All Topological Sorts of a Directed Acyclic Graph](#)

References:

[http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/
topoSort.htm](http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/topoSort.htm)
http://en.wikipedia.org/wiki/Topological_sorting

Improved By : [ConstantineShulyupin](#)

Source

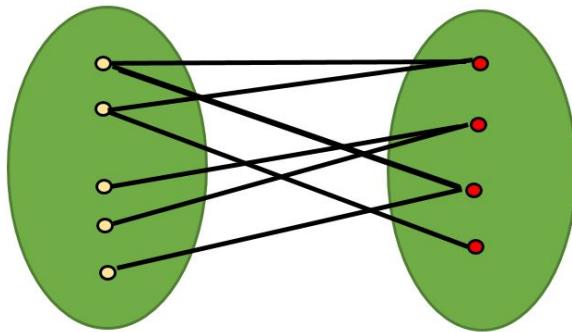
<https://www.geeksforgeeks.org/topological-sorting/>

Chapter 12

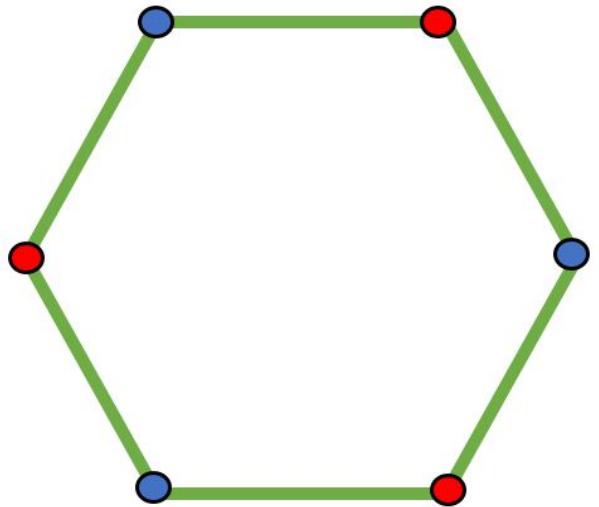
Check whether a given graph is Bipartite or not

Check whether a given graph is Bipartite or not - GeeksforGeeks

A [Bipartite Graph](#) is a graph whose vertices can be divided into two independent sets, U and V such that every edge (u, v) either connects a vertex from U to V or a vertex from V to U. In other words, for every edge (u, v) , either u belongs to U and v to V, or u belongs to V and v to U. We can also say that there is no edge that connects vertices of same set.

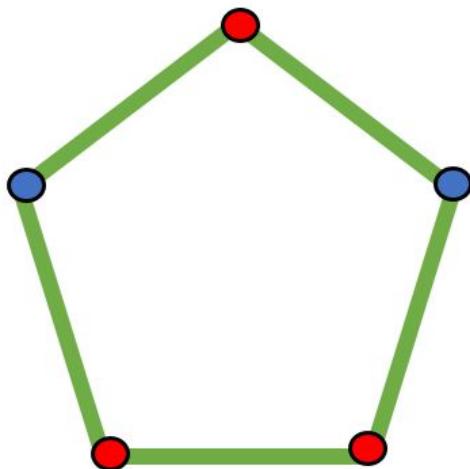


A bipartite graph is possible if the graph coloring is possible using two colors such that vertices in a set are colored with the same color. Note that it is possible to color a cycle graph with even cycle using two colors. For example, see the following graph.



Cycle graph of length 6

It is not possible to color a cycle graph with odd cycle using two colors.



Cycle graph of length 5

Algorithm to check if a graph is Bipartite:

One approach is to check whether the graph is 2-colorable or not using backtracking algo-

rithm m coloring problem.

Following is a simple algorithm to find out whether a given graph is Bipartite or not using Breadth First Search (BFS).

1. Assign RED color to the source vertex (putting into set U).
2. Color all the neighbors with BLUE color (putting into set V).
3. Color all neighbor's neighbor with RED color (putting into set U).
4. This way, assign color to all vertices such that it satisfies all the constraints of m way coloring problem where $m = 2$.
5. While assigning colors, if we find a neighbor which is colored with same color as current vertex, then the graph cannot be colored with 2 vertices (or graph is not Bipartite)

C++

```
// C++ program to find out whether a
// given graph is Bipartite or not
#include <iostream>
#include <queue>
#define V 4

using namespace std;

// This function returns true if graph
// G[V][V] is Bipartite, else false
bool isBipartite(int G[][V], int src)
{
    // Create a color array to store colors
    // assigned to all vertices. Vertex
    // number is used as index in this array.
    // The value '-1' of colorArr[i]
    // is used to indicate that no color
    // is assigned to vertex 'i'. The value 1
    // is used to indicate first color
    // is assigned and value 0 indicates
    // second color is assigned.
    int colorArr[V];
    for (int i = 0; i < V; ++i)
        colorArr[i] = -1;

    // Assign first color to source
    colorArr[src] = 1;

    // Create a queue (FIFO) of vertex
    // numbers and enqueue source vertex
    // for BFS traversal
    queue <int> q;
    q.push(src);

    // Run while there are vertices
    // in queue (Similar to BFS)
```

```
while (!q.empty())
{
    // Dequeue a vertex from queue ( Refer http://goo.gl/35oz8 )
    int u = q.front();
    q.pop();

    // Return false if there is a self-loop
    if (G[u][u] == 1)
        return false;

    // Find all non-colored adjacent vertices
    for (int v = 0; v < V; ++v)
    {
        // An edge from u to v exists and
        // destination v is not colored
        if (G[u][v] && colorArr[v] == -1)
        {
            // Assign alternate color to this adjacent v of u
            colorArr[v] = 1 - colorArr[u];
            q.push(v);
        }

        // An edge from u to v exists and destination
        // v is colored with same color as u
        else if (G[u][v] && colorArr[v] == colorArr[u])
            return false;
    }
}

// If we reach here, then all adjacent
// vertices can be colored with alternate color
return true;
}

// Driver program to test above function
int main()
{
    int G[][][V] = {{0, 1, 0, 1},
                    {1, 0, 1, 0},
                    {0, 1, 0, 1},
                    {1, 0, 1, 0}};
};

isBipartite(G, 0) ? cout << "Yes" : cout << "No";
return 0;
}
```

Java

```
// Java program to find out whether
// a given graph is Bipartite or not
import java.util.*;
import java.lang.*;
import java.io.*;

class Bipartite
{
    final static int V = 4; // No. of Vertices

    // This function returns true if
    // graph G[V][V] is Bipartite, else false
    boolean isBipartite(int G[][] ,int src)
    {
        // Create a color array to store
        // colors assigned to all vertices.
        // Vertex number is used as index
        // in this array. The value '-1'
        // of colorArr[i] is used to indicate
        // that no color is assigned
        // to vertex 'i'. The value 1 is
        // used to indicate first color
        // is assigned and value 0 indicates
        // second color is assigned.
        int colorArr[] = new int[V];
        for (int i=0; i<V; ++i)
            colorArr[i] = -1;

        // Assign first color to source
        colorArr[src] = 1;

        // Create a queue (FIFO) of vertex numbers
        // and enqueue source vertex for BFS traversal
        LinkedList<Integer>q = new LinkedList<Integer>();
        q.add(src);

        // Run while there are vertices in queue (Similar to BFS)
        while (q.size() != 0)
        {
            // Dequeue a vertex from queue
            int u = q.poll();

            // Return false if there is a self-loop
            if (G[u][u] == 1)
                return false;

            // Find all non-colored adjacent vertices
            for (int v=0; v<V; ++v)
```

```
{  
    // An edge from u to v exists  
    // and destination v is not colored  
    if (G[u][v]==1 && colorArr[v]==-1)  
    {  
        // Assign alternate color to this adjacent v of u  
        colorArr[v] = 1-colorArr[u];  
        q.add(v);  
    }  
  
    // An edge from u to v exists and destination  
    // v is colored with same color as u  
    else if (G[u][v]==1 && colorArr[v]==colorArr[u])  
        return false;  
    }  
}  
// If we reach here, then all adjacent vertices can  
// be colored with alternate color  
return true;  
}  
  
// Driver program to test above function  
public static void main (String[] args)  
{  
    int G[][] = {{0, 1, 0, 1},  
                {1, 0, 1, 0},  
                {0, 1, 0, 1},  
                {1, 0, 1, 0}  
    };  
    Bipartite b = new Bipartite();  
    if (b.isBipartite(G, 0))  
        System.out.println("Yes");  
    else  
        System.out.println("No");  
}  
}  
  
// Contributed by Aakash Hasija
```

Python

```
# Python program to find out whether a  
# given graph is Bipartite or not  
  
class Graph():  
  
    def __init__(self, V):  
        self.V = V
```

```
self.graph = [[0 for column in range(V)] \
              for row in range(V)]

# This function returns true if graph G[V][V]
# is Bipartite, else false
def isBipartite(self, src):

    # Create a color array to store colors
    # assigned to all vertices. Vertex
    # number is used as index in this array.
    # The value '-1' of colorArr[i] is used to
    # indicate that no color is assigned to
    # vertex 'i'. The value 1 is used to indicate
    # first color is assigned and value 0
    # indicates second color is assigned.
    colorArr = [-1] * self.V

    # Assign first color to source
    colorArr[src] = 1

    # Create a queue (FIFO) of vertex numbers and
    # enqueue source vertex for BFS traversal
    queue = []
    queue.append(src)

    # Run while there are vertices in queue
    # (Similar to BFS)
    while queue:

        u = queue.pop()

        # Return false if there is a self-loop
        if self.graph[u][u] == 1:
            return False;

        for v in range(self.V):

            # An edge from u to v exists and destination
            # v is not colored
            if self.graph[u][v] == 1 and colorArr[v] == -1:

                # Assign alternate color to this
                # adjacent v of u
                colorArr[v] = 1 - colorArr[u]
                queue.append(v)

            # An edge from u to v exists and destination
            # v is colored with same color as u
```

```
        elif self.graph[u][v] == 1 and colorArr[v] == colorArr[u]:
            return False

        # If we reach here, then all adjacent
        # vertices can be colored with alternate
        # color
        return True

# Driver program to test above function
g = Graph(4)
g.graph = [[0, 1, 0, 1],
            [1, 0, 1, 0],
            [0, 1, 0, 1],
            [1, 0, 1, 0]
            ]

print "Yes" if g.isBipartite() else "No"

# This code is contributed by Divyanshu Mehta
```

Output:

Yes

The above algorithm works only if the graph is strongly connected. In above code, we always start with source 0 and assume that vertices are visited from it. One important observation is a graph with no edges is also Bipartite. Note that the Bipartite condition says all edges should be from one set to another.

We can extend the above code to handle cases when a graph is not connected. The idea is repeatedly call above method for all not yet visited vertices.

C++

```
// C++ program to find out whether
// a given graph is Bipartite or not.
// It works for disconnected graph also.
#include <bits/stdc++.h>

using namespace std;

const int V = 4;

// This function returns true if
// graph G[V][V] is Bipartite, else false
bool isBipartiteUtil(int G[][V], int src, int colorArr[])
{
    colorArr[src] = 1;
```

```

// Create a queue (FIFO) of vertex numbers and enqueue source vertex for BFS traversal
queue <int> q;
q.push(src);

// Run while there are vertices in queue (Similar to BFS)
while (!q.empty())
{
    // Dequeue a vertex from queue ( Refer http://goo.gl/35oz8 )
    int u = q.front();
    q.pop();

    // Return false if there is a self-loop
    if (G[u][u] == 1)
        return false;

    // Find all non-colored adjacent vertices
    for (int v = 0; v < V; ++v)
    {
        // An edge from u to v exists and
        // destination v is not colored
        if (G[u][v] && colorArr[v] == -1)
        {
            // Assign alternate color to this
            // adjacent v of u
            colorArr[v] = 1 - colorArr[u];
            q.push(v);
        }

        // An edge from u to v exists and destination
        // v is colored with same color as u
        else if (G[u][v] && colorArr[v] == colorArr[u])
            return false;
    }
}

// If we reach here, then all adjacent vertices can
// be colored with alternate color
return true;
}

// Returns true if G[][] is Bipartite, else false
bool isBipartite(int G[][V])
{
    // Create a color array to store colors assigned to all
    // vertices. Vertex/ number is used as index in this
    // array. The value '-1' of colorArr[i] is used to
    // indicate that no color is assigned to vertex 'i'.
}

```

```
// The value 1 is used to indicate first color is
// assigned and value 0 indicates second color is
// assigned.
int colorArr[V];
for (int i = 0; i < V; ++i)
    colorArr[i] = -1;

// This code is to handle disconnected graph
for (int i = 0; i < V; i++)
    if (colorArr[i] == -1)
        if (isBipartiteUtil(G, i, colorArr) == false)
            return false;

    return true;
}

// Driver program to test above function
int main()
{
    int G[][][V] = {{0, 1, 0, 1},
                    {1, 0, 1, 0},
                    {0, 1, 0, 1},
                    {1, 0, 1, 0}
    };

    isBipartite(G) ? cout << "Yes" : cout << "No";
    return 0;
}
```

Java

```
// JAVA Code to check whether a given
// graph is Bipartite or not
import java.util.*;

class Bipartite {

    public static int V = 4;

    // This function returns true if graph
    // G[V][V] is Bipartite, else false
    public static boolean isBipartiteUtil(int G[][], int src,
                                         int colorArr[])
    {
        colorArr[src] = 1;

        // Create a queue (FIFO) of vertex numbers and
        // enqueue source vertex for BFS traversal
```

```
LinkedList<Integer> q = new LinkedList<Integer>();
q.add(src);

// Run while there are vertices in queue
// (Similar to BFS)
while (!q.isEmpty())
{
    // Dequeue a vertex from queue
    // ( Refer http://goo.gl/35oz8 )
    int u = q.getFirst();
    q.pop();

    // Return false if there is a self-loop
    if (G[u][u] == 1)
        return false;

    // Find all non-colored adjacent vertices
    for (int v = 0; v < V; ++v)
    {
        // An edge from u to v exists and
        // destination v is not colored
        if (G[u][v] == 1 && colorArr[v] == -1)
        {
            // Assign alternate color to this
            // adjacent v of u
            colorArr[v] = 1 - colorArr[u];
            q.push(v);
        }

        // An edge from u to v exists and
        // destination v is colored with same
        // color as u
        else if (G[u][v] == 1 && colorArr[v] ==
                  colorArr[u])
            return false;
    }
}

// If we reach here, then all adjacent vertices
// can be colored with alternate color
return true;
}

// Returns true if G[][] is Bipartite, else false
public static boolean isBipartite(int G[][])
{
    // Create a color array to store colors assigned
    // to all vertices. Vertex/ number is used as
```

```

// index in this array. The value '-1' of
// colorArr[i] is used to indicate that no color
// is assigned to vertex 'i'. The value 1 is used
// to indicate first color is assigned and value
// 0 indicates second color is assigned.
int colorArr[] = new int[V];
for (int i = 0; i < V; ++i)
    colorArr[i] = -1;

// This code is to handle disconnected graph
for (int i = 0; i < V; i++)
    if (colorArr[i] == -1)
        if (isBipartiteUtil(G, i, colorArr) == false)
            return false;

    return true;
}

/* Driver program to test above function */
public static void main(String[] args)
{
    int G[][] = {{0, 1, 0, 1},
                 {1, 0, 1, 0},
                 {0, 1, 0, 1},
                 {1, 0, 1, 0}};

    if (isBipartite(G))
        System.out.println("Yes");
    else
        System.out.println("No");
}
}

// This code is contributed by Arnav Kr. Mandal.

```

Output:

Yes

Time Complexity of the above approach is same as that Breadth First Search. In above implementation is $O(V^2)$ where V is number of vertices. If graph is represented using adjacency list, then the complexity becomes $O(V+E)$.

Exercise:

1. Can DFS algorithm be used to check the bipartite-ness of a graph? If yes, how?

Solution :

```
// C++ program to find out whether a given graph is Bipartite or not.
```

```
// Using recursion.
#include <iostream>

using namespace std;
#define V 4

bool colorGraph(int G[][][V],int color[],int pos, int c){

    if(color[pos] != -1 && color[pos] !=c)
        return false;

    // color this pos as c and all its neighbours and 1-c
    color[pos] = c;
    bool ans = true;
    for(int i=0;i<V;i++){
        if(G[pos][i]){
            if(color[i] == -1)
                ans &= colorGraph(G,color,i,1-c);

            if(color[i] !=-1 && color[i] != 1-c)
                return false;
        }
        if (!ans)
            return false;
    }

    return true;
}

bool isBipartite(int G[][][V]){
    int color[V];
    for(int i=0;i<V;i++)
        color[i] = -1;

    //start is vertex 0;
    int pos = 0;
    // two colors 1 and 0
    return colorGraph(G,color,pos,1);

}

int main()
{
    int G[][][V] = {{0, 1, 0, 1},
                    {1, 0, 1, 0},
                    {0, 1, 0, 1},
                    {1, 0, 1, 0}}
```

```
};

isBipartite(G) ? cout<< "Yes" : cout << "No";
return 0;
}
// This code is contributed By Mudit Verma
```

References:

http://en.wikipedia.org/wiki/Graph_coloring
http://en.wikipedia.org/wiki/Bipartite_graph

This article is compiled by [Aashish Barnwal](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Improved By : [Mudit Verma](#)

Source

<https://www.geeksforgeeks.org/bipartite-graph/>

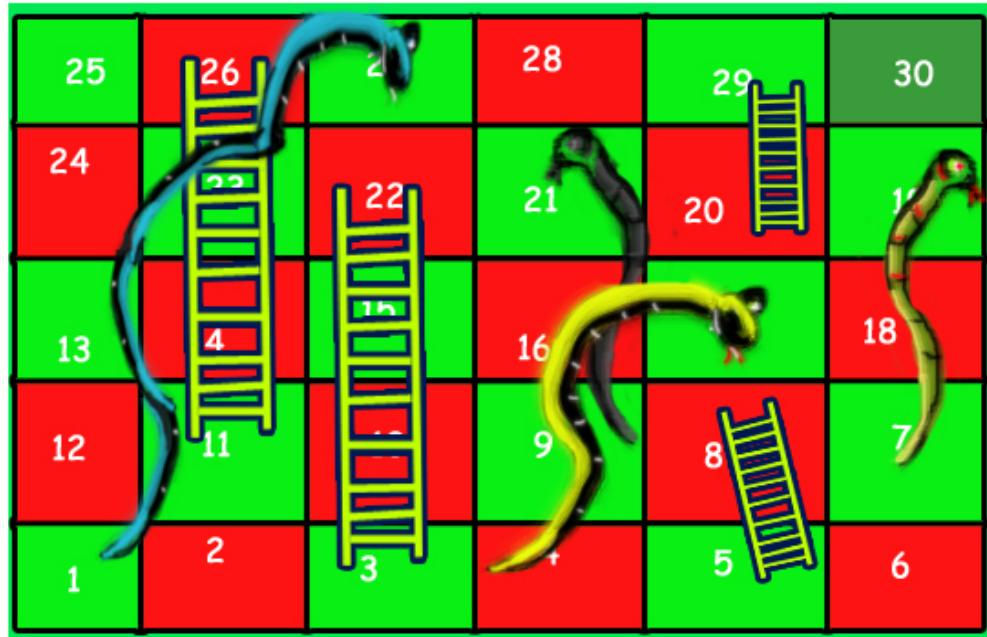
Chapter 13

Snake and Ladder Problem

[Snake and Ladder Problem - GeeksforGeeks](#)

Given a snake and ladder board, find the minimum number of dice throws required to reach the destination or last cell from source or 1st cell. Basically, the player has total control over outcome of dice throw and wants to find out minimum number of throws required to reach last cell.

If the player reaches a cell which is base of a ladder, the player has to climb up that ladder and if reaches a cell is mouth of the snake, has to go down to the tail of snake without a dice throw.



For example, consider the board shown, the minimum number of dice throws required to reach cell 30 from cell 1 is 3.

Following are the steps:

- a) First throw two on dice to reach cell number 3 and then ladder to reach 22
- b) Then throw 6 to reach 28.
- c) Finally through 2 to reach 30.

There can be other solutions as well like (2, 2, 6), (2, 4, 4), (2, 3, 5).. etc.

The idea is to consider the given snake and ladder board as a directed graph with number of vertices equal to the number of cells in the board. The problem reduces to finding the shortest path in a graph. Every vertex of the graph has an edge to next six vertices if next 6 vertices do not have a snake or ladder. If any of the next six vertices has a snake or ladder, then the edge from current vertex goes to the top of the ladder or tail of the snake. Since all edges are of equal weight, we can efficiently find shortest path using [Breadth First Search](#) of the graph.

Following is the implementation of the above idea. The input is represented by two things, first is 'N' which is number of cells in the given board, second is an array 'move[0...N-1]' of size N. An entry move[i] is -1 if there is no snake and no ladder from i, otherwise move[i] contains index of destination cell for the snake or the ladder at i.

C++

```
// C++ program to find minimum number of dice throws required to
// reach last cell from first cell of a given snake and ladder
// board
#include<iostream>
#include <queue>
using namespace std;

// An entry in queue used in BFS
struct queueEntry
{
    int v;      // Vertex number
    int dist;   // Distance of this vertex from source
};

// This function returns minimum number of dice throws required to
// Reach last cell from 0'th cell in a snake and ladder game.
// move[] is an array of size N where N is no. of cells on board
// If there is no snake or ladder from cell i, then move[i] is -1
// Otherwise move[i] contains cell to which snake or ladder at i
// takes to.
int getMinDiceThrows(int move[], int N)
{
    // The graph has N vertices. Mark all the vertices as
    // not visited
    bool *visited = new bool[N];
```

```

for (int i = 0; i < N; i++)
    visited[i] = false;

// Create a queue for BFS
queue<queueEntry> q;

// Mark the node 0 as visited and enqueue it.
visited[0] = true;
queueEntry s = {0, 0}; // distance of 0't vertex is also 0
q.push(s); // Enqueue 0'th vertex

// Do a BFS starting from vertex at index 0
queueEntry qe; // A queue entry (qe)
while (!q.empty())
{
    qe = q.front();
    int v = qe.v; // vertex no. of queue entry

    // If front vertex is the destination vertex,
    // we are done
    if (v == N-1)
        break;

    // Otherwise dequeue the front vertex and enqueue
    // its adjacent vertices (or cell numbers reachable
    // through a dice throw)
    q.pop();
    for (int j=v+1; j<=(v+6) && j<N; ++j)
    {
        // If this cell is already visited, then ignore
        if (!visited[j])
        {
            // Otherwise calculate its distance and mark it
            // as visited
            queueEntry a;
            a.dist = (qe.dist + 1);
            visited[j] = true;

            // Check if there a snake or ladder at 'j'
            // then tail of snake or top of ladder
            // become the adjacent of 'i'
            if (move[j] != -1)
                a.v = move[j];
            else
                a.v = j;
            q.push(a);
        }
    }
}

```

```
}

// We reach here when 'qe' has last vertex
// return the distance of vertex in 'qe'
return qe.dist;
}

// Driver program to test methods of graph class
int main()
{
    // Let us construct the board given in above diagram
    int N = 30;
    int moves[N];
    for (int i = 0; i < N; i++)
        moves[i] = -1;

    // Ladders
    moves[2] = 21;
    moves[4] = 7;
    moves[10] = 25;
    moves[19] = 28;

    // Snakes
    moves[26] = 0;
    moves[20] = 8;
    moves[16] = 3;
    moves[18] = 6;

    cout << "Min Dice throws required is " << getMinDiceThrows(moves, N);
    return 0;
}
```

Java

```
// Java program to find minimum number of dice
// throws required to reach last cell from first
// cell of a given snake and ladder board

import java.util.LinkedList;
import java.util.Queue;

public class SnakesLadder
{
    // An entry in queue used in BFS
    static class qentry
    {
        int v;// Vertex number
        int dist;// Distance of this vertex from source
```

```

}

// This function returns minimum number of dice
// throws required to Reach last cell from 0'th cell
// in a snake and ladder game. move[] is an array of
// size N where N is no. of cells on board If there
// is no snake or ladder from cell i, then move[i]
// is -1 Otherwise move[i] contains cell to which
// snake or ladder at i takes to.
static int getMinDiceThrows(int move[], int n)
{
    int visited[] = new int[n];
    Queue<qentry> q = new LinkedList<>();
    qentry qe = new qentry();
    qe.v = 0;
    qe.dist = 0;

    // Mark the node 0 as visited and enqueue it.
    visited[0] = 1;
    q.add(qe);

    // Do a BFS starting from vertex at index 0
    while (!q.isEmpty())
    {
        qe = q.remove();
        int v = qe.v;

        // If front vertex is the destination
        // vertex, we are done
        if (v == n - 1)
            break;

        // Otherwise dequeue the front vertex and
        // enqueue its adjacent vertices (or cell
        // numbers reachable through a dice throw)
        for (int j = v + 1; j <= (v + 6) && j < n; ++j)
        {
            // If this cell is already visited, then ignore
            if (visited[j] == 0)
            {
                // Otherwise calculate its distance and
                // mark it as visited
                qentry a = new qentry();
                a.dist = (qe.dist + 1);
                visited[j] = 1;

                // Check if there a snake or ladder at 'j'
                // then tail of snake or top of ladder
            }
        }
    }
}

```

```

        // become the adjacent of 'i'
        if (move[j] != -1)
            a.v = move[j];
        else
            a.v = j;
        q.add(a);
    }
}

// We reach here when 'qe' has last vertex
// return the distance of vertex in 'qe'
return qe.dist;
}

public static void main(String[] args)
{
    // Let us construct the board given in above diagram
    int N = 30;
    int moves[] = new int[N];
    for (int i = 0; i < N; i++)
        moves[i] = -1;

    // Ladders
    moves[2] = 21;
    moves[4] = 7;
    moves[10] = 25;
    moves[19] = 28;

    // Snakes
    moves[26] = 0;
    moves[20] = 8;
    moves[16] = 3;
    moves[18] = 6;

    System.out.println("Min Dice throws required is " +
                       getMinDiceThrows(moves, N));
}
}

```

Python3

```

# Python3 program to find minimum number
# of dice throws required to reach last
# cell from first cell of a given
# snake and ladder board

# An entry in queue used in BFS

```

```

class QueueEntry(object):
    def __init__(self, v = 0, dist = 0):
        self.v = v
        self.dist = dist

    '''This function returns minimum number of
    dice throws required to. Reach last cell
    from 0'th cell in a snake and ladder game.
    move[] is an array of size N where N is
    no. of cells on board. If there is no
    snake or ladder from cell i, then move[i]
    is -1. Otherwise move[i] contains cell to
    which snake or ladder at i takes to.'''
    def getMinDiceThrows(move, N):

        # The graph has N vertices. Mark all
        # the vertices as not visited
        visited = [False] * N

        # Create a queue for BFS
        queue = []

        # Mark the node 0 as visited and enqueue it
        visited[0] = True

        # Distance of 0't vertex is also 0
        # Enqueue 0'th vertex
        queue.append(QueueEntry(0, 0))

        # Do a BFS starting from vertex at index 0
        qe = QueueEntry() # A queue entry (qe)
        while queue:
            qe = queue.pop(0)
            v = qe.v # Vertex no. of queue entry

            # If front vertex is the destination
            # vertex, we are done
            if v == N - 1:
                break

            # Otherwise dequeue the front vertex
            # and enqueue its adjacent vertices
            # (or cell numbers reachable through
            # a dice throw)
            j = v + 1
            while j <= v + 6 and j < N:

                # If this cell is already visited,

```

```

# then ignore
if visited[j] is False:

    # Otherwise calculate its
    # distance and mark it
    # as visited
    a = QueueEntry()
    a.dist = qe.dist + 1
    visited[j] = True

    # Check if there a snake or ladder
    # at 'j' then tail of snake or top
    # of ladder become the adjacent of 'i'
    a.v = move[j] if move[j] != -1 else j

queue.append(a)

j += 1

# We reach here when 'qe' has last vertex
# return the distance of vertex in 'qe'
return qe.dist

# driver code
N = 30
moves = [-1] * N

# Ladders
moves[2] = 21
moves[4] = 7
moves[10] = 25
moves[19] = 28

# Snakes
moves[26] = 0
moves[20] = 8
moves[16] = 3
moves[18] = 6

print("Min Dice throws required is {0}".
      format(getMinDiceThrows(moves, N)))

# This code is contributed by Ajitesh Pathak

```

Output:

Min Dice throws required is 3

Time complexity of the above solution is $O(N)$ as every cell is added and removed only once from queue. And a typical enqueue or dequeue operation takes $O(1)$ time.

This article is contributed by **Siddharth**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

<https://www.geeksforgeeks.org/snake-ladder-problem-2/>

Chapter 14

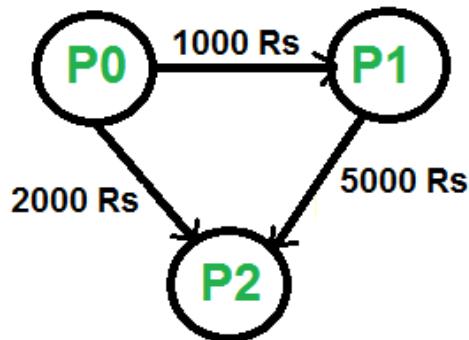
Minimize Cash Flow among a given set of friends who have borrowed money from each other

Minimize Cash Flow among a given set of friends who have borrowed money from each other
- GeeksforGeeks

Given a number of friends who have to give or take some amount of money from one another.
Design an algorithm by which the total cash flow among all the friends is minimized.

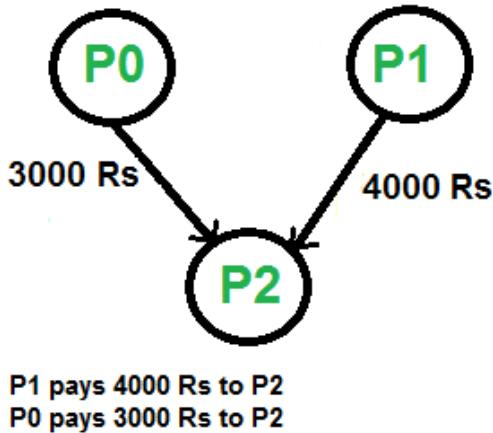
Example:

Following diagram shows input debts to be settled.



P0 has to pay 1000 Rs to P1
P0 also has to pay 2000 Rs to P2
P1 has to pay 5000 Rs to P2.

Above debts can be settled in following optimized way



The idea is to use [Greedy algorithm](#) where at every step, settle all amounts of one person and recur for remaining n-1 persons.

How to pick the first person? To pick the first person, calculate the net amount for every person where net amount is obtained by subtracting all debts (amounts to pay) from all credits (amounts to be paid). Once net amount for every person is evaluated, find two persons with maximum and minimum net amounts. These two persons are the most creditors and debtors. The person with minimum of two is our first person to be settled and removed from list. Let the minimum of two amounts be x. We pay ‘x’ amount from the maximum debtor to maximum creditor and settle one person. If x is equal to the maximum debit, then maximum debtor is settled, else maximum creditor is settled.

The following is detailed algorithm.

Do following for every person P_i where i is from 0 to $n-1$.

- 1) Compute the net amount for every person. The net amount for person ‘i’ can be computed by subtracting sum of all debts from sum of all credits.
- 2) Find the two persons that are maximum creditor and maximum debtor. Let the maximum amount to be credited maximum creditor be maxCredit and maximum amount to be debited from maximum debtor be maxDebit. Let the maximum debtor be P_d and maximum creditor be P_c .
- 3) Find the minimum of maxDebit and maxCredit. Let minimum of two be x. Debit ‘x’ from P_d and credit this amount to P_c
- 4) If x is equal to maxCredit, then remove P_c from set of persons and recur for remaining ($n-1$) persons.
- 5) If x is equal to maxDebit, then remove P_d from set of persons and recur for remaining ($n-1$) persons.

Thanks to Balaji S for suggesting this method in a comment [here](#).

The following is implementation of above algorithm.

C++

```
// C++ program to fin maximum cash flow among a set of persons
```

```
#include<iostream>
using namespace std;

// Number of persons (or vertices in the graph)
#define N 3

// A utility function that returns index of minimum value in arr[]
int getMin(int arr[])
{
    int minInd = 0;
    for (int i=1; i<N; i++)
        if (arr[i] < arr[minInd])
            minInd = i;
    return minInd;
}

// A utility function that returns index of maximum value in arr[]
int getMax(int arr[])
{
    int maxInd = 0;
    for (int i=1; i<N; i++)
        if (arr[i] > arr[maxInd])
            maxInd = i;
    return maxInd;
}

// A utility function to return minimum of 2 values
int minOf2(int x, int y)
{
    return (x<y)? x: y;
}

// amount[p] indicates the net amount to be credited/debited
// to/from person 'p'
// If amount[p] is positive, then i'th person will amount[i]
// If amount[p] is negative, then i'th person will give -amount[i]
void minCashFlowRec(int amount[])
{
    // Find the indexes of minimum and maximum values in amount[]
    // amount[mxCredit] indicates the maximum amount to be given
    //           (or credited) to any person .
    // And amount[mxD debit] indicates the maximum amount to be taken
    //           (or debited) from any person.
    // So if there is a positive value in amount[], then there must
    // be a negative value
    int mxCredit = getMax(amount), mxDebit = getMin(amount);

    // If both amounts are 0, then all amounts are settled
```

```
if (amount[mxCredit] == 0 && amount[mxDebit] == 0)
    return;

// Find the minimum of two amounts
int min = minOf2(-amount[mxDebit], amount[mxCredit]);
amount[mxCredit] -= min;
amount[mxDebit] += min;

// If minimum is the maximum amount to be
cout << "Person " << mxDebit << " pays " << min
    << " to " << "Person " << mxCredit << endl;

// Recur for the amount array. Note that it is guaranteed that
// the recursion would terminate as either amount[mxCredit]
// or amount[mxDebit] becomes 0
minCashFlowRec(amount);
}

// Given a set of persons as graph[] where graph[i][j] indicates
// the amount that person i needs to pay person j, this function
// finds and prints the minimum cash flow to settle all debts.
void minCashFlow(int graph[][] N)
{
    // Create an array amount[], initialize all value in it as 0.
    int amount[N] = {0};

    // Calculate the net amount to be paid to person 'p', and
    // stores it in amount[p]. The value of amount[p] can be
    // calculated by subtracting debts of 'p' from credits of 'p'
    for (int p=0; p<N; p++)
        for (int i=0; i<N; i++)
            amount[p] += (graph[i][p] - graph[p][i]);

    minCashFlowRec(amount);
}

// Driver program to test above function
int main()
{
    // graph[i][j] indicates the amount that person i needs to
    // pay person j
    int graph[N][N] = { {0, 1000, 2000},
                        {0, 0, 5000},
                        {0, 0, 0}, };

    // Print the solution
    minCashFlow(graph);
    return 0;
}
```

}

Java

```
// Java program to fin maximum cash
// flow among a set of persons

class GFG
{
    // Number of persons (or vertices in the graph)
    static final int N = 3;

    // A utility function that returns
    // index of minimum value in arr[]
    static int getMin(int arr[])
    {
        int minInd = 0;
        for (int i = 1; i < N; i++)
            if (arr[i] < arr[minInd])
                minInd = i;
        return minInd;
    }

    // A utility function that returns
    // index of maximum value in arr[]
    static int getMax(int arr[])
    {
        int maxInd = 0;
        for (int i = 1; i < N; i++)
            if (arr[i] > arr[maxInd])
                maxInd = i;
        return maxInd;
    }

    // A utility function to return minimum of 2 values
    static int minOf2(int x, int y)
    {
        return (x < y) ? x: y;
    }

    // amount[p] indicates the net amount
    // to be credited/debited to/from person 'p'
    // If amount[p] is positive, then
    // i'th person will amount[i]
    // If amount[p] is negative, then
    // i'th person will give -amount[i]
    static void minCashFlowRec(int amount[])
    {
```

```
// Find the indexes of minimum and
// maximum values in amount[]
// amount[mxCredit] indicates the maximum amount
// to be given (or credited) to any person .
// And amount[mxDabit] indicates the maximum amount
// to be taken(or debited) from any person.
// So if there is a positive value in amount[],
// then there must be a negative value
int mxCredit = getMax(amount), mxDebit = getMin(amount);

// If both amounts are 0, then
// all amounts are settled
if (amount[mxCredit] == 0 && amount[mxDabit] == 0)
    return;

// Find the minimum of two amounts
int min = minOf2(-amount[mxDabit], amount[mxCredit]);
amount[mxCredit] -= min;
amount[mxDabit] += min;

// If minimum is the maximum amount to be
System.out.println("Person " + mxDebit + " pays " + min
                    + " to " + "Person " + mxCredit);

// Recur for the amount array.
// Note that it is guaranteed that
// the recursion would terminate
// as either amount[mxCredit] or
// amount[mxDabit] becomes 0
minCashFlowRec(amount);
}

// Given a set of persons as graph[]
// where graph[i][j] indicates
// the amount that person i needs to
// pay person j, this function
// finds and prints the minimum
// cash flow to settle all debts.
static void minCashFlow(int graph[][])
{
    // Create an array amount[],
    // initialize all value in it as 0.
    int amount[] = new int[N];

    // Calculate the net amount to
    // be paid to person 'p', and
    // stores it in amount[p]. The
    // value of amount[p] can be
```

```
// calculated by subtracting
// debts of 'p' from credits of 'p'
for (int p = 0; p < N; p++)
    for (int i = 0; i < N; i++)
        amount[p] += (graph[i][p] - graph[p][i]);

    minCashFlowRec(amount);
}

// Driver code
public static void main (String[] args)
{
    // graph[i][j] indicates the amount
    // that person i needs to pay person j
    int graph[][] = { {0, 1000, 2000},
                      {0, 0, 5000},
                      {0, 0, 0},};

    // Print the solution
    minCashFlow(graph);
}
}

// This code is contributed by Anant Agarwal.
```

Python3

```
# Python3 program to fin maximum
# cash flow among a set of persons

# Number of persons(or vertices in graph)
N = 3

# A utility function that returns
# index of minimum value in arr[]
def getMin(arr):

    minInd = 0
    for i in range(1, N):
        if (arr[i] < arr[minInd]):
            minInd = i
    return minInd

# A utility function that returns
# index of maximum value in arr[]
def getMax(arr):

    maxInd = 0
```

```
for i in range(1, N):
    if (arr[i] > arr[maxInd]):
        maxInd = i
return maxInd

# A utility function to
# return minimum of 2 values
def minOf2(x, y):

    return x if x < y else y

# amount[p] indicates the net amount to
# be credited/debited to/from person 'p'
# If amount[p] is positive, then i'th
# person will amount[i]
# If amount[p] is negative, then i'th
# person will give -amount[i]
def minCashFlowRec(amount):

    # Find the indexes of minimum
    # and maximum values in amount[]
    # amount[mxCredit] indicates the maximum
    # amount to be given(or credited) to any person.
    # And amount[mxDabit] indicates the maximum amount
    # to be taken (or debited) from any person.
    # So if there is a positive value in amount[],
    # then there must be a negative value
    mxCredit = getMax(amount)
    mxDebit = getMin(amount)

    # If both amounts are 0,
    # then all amounts are settled
    if (amount[mxCredit] == 0 and amount[mxDabit] == 0):
        return 0

    # Find the minimum of two amounts
    min = minOf2(-amount[mxDabit], amount[mxCredit])
    amount[mxCredit] -= min
    amount[mxDabit] += min

    # If minimum is the maximum amount to be
    print("Person " , mxDebit , " pays " , min
          , " to " , "Person " , mxCredit)

    # Recur for the amount array. Note that
    # it is guaranteed that the recursion
    # would terminate as either amount[mxCredit]
    # or amount[mxDabit] becomes 0
```

```
minCashFlowRec(amount)

# Given a set of persons as graph[] where
# graph[i][j] indicates the amount that
# person i needs to pay person j, this
# function finds and prints the minimum
# cash flow to settle all debts.
def minCashFlow(graph):

    # Create an array amount[],
    # initialize all value in it as 0.
    amount = [0 for i in range(N)]

    # Calculate the net amount to be paid
    # to person 'p', and stores it in amount[p].
    # The value of amount[p] can be calculated by
    # subtracting debts of 'p' from credits of 'p'
    for p in range(N):
        for i in range(N):
            amount[p] += (graph[i][p] - graph[p][i])

    minCashFlowRec(amount)

# Driver code

# graph[i][j] indicates the amount
# that person i needs to pay person j
graph = [ [0, 1000, 2000],
          [0, 0, 5000],
          [0, 0, 0] ]

minCashFlow(graph)

# This code is contributed by Anant Agarwal.
```

C#

```
// C# program to fin maximum cash
// flow among a set of persons
using System;

class GFG
{
    // Number of persons (or
    // vertices in the graph)
    static int N = 3;

    // A utility function that returns
```

```
// index of minimum value in arr[]
static int getMin(int []arr)
{
    int minInd = 0;
    for (int i = 1; i < N; i++)
        if (arr[i] < arr[minInd])
            minInd = i;
    return minInd;
}

// A utility function that returns
// index of maximum value in arr[]
static int getMax(int []arr)
{
    int maxInd = 0;
    for (int i = 1; i < N; i++)
        if (arr[i] > arr[maxInd])
            maxInd = i;
    return maxInd;
}

// A utility function to return
// minimum of 2 values
static int minOf2(int x, int y)
{
    return (x < y) ? x: y;
}

// amount[p] indicates the net amount
// to be credited/debited to/from person 'p'
// If amount[p] is positive, then
// i'th person will amount[i]
// If amount[p] is negative, then
// i'th person will give -amount[i]
static void minCashFlowRec(int []amount)
{
    // Find the indexes of minimum and
    // maximum values in amount[]
    // amount[mxCredit] indicates the maximum amount
    // to be given (or credited) to any person .
    // And amount[mxD debit] indicates the maximum amount
    // to be taken(or debited) from any person.
    // So if there is a positive value in amount[],
    // then there must be a negative value
    int mxCredit = getMax(amount), mxDebit = getMin(amount);

    // If both amounts are 0, then
    // all amounts are settled
```

```
if (amount[mxCredit] == 0 &&
    amount[mxDebit] == 0)
    return;

// Find the minimum of two amounts
int min = minOf2(-amount[mxDebit], amount[mxCredit]);
amount[mxCredit] -= min;
amount[mxDebit] += min;

// If minimum is the maximum amount to be
Console.WriteLine("Person " + mxDebit +
                    " pays " + min + " to " +
                    "Person " + mxCredit);

// Recur for the amount array.
// Note that it is guaranteed that
// the recursion would terminate
// as either amount[mxCredit] or
// amount[mxDebit] becomes 0
minCashFlowRec(amount);
}

// Given a set of persons as graph[]
// where graph[i][j] indicates
// the amount that person i needs to
// pay person j, this function
// finds and prints the minimum
// cash flow to settle all debts.
static void minCashFlow(int [,]graph)
{
    // Create an array amount[],
    // initialize all value in it as 0.
    int []amount=new int[N];

    // Calculate the net amount to
    // be paid to person 'p', and
    // stores it in amount[p]. The
    // value of amount[p] can be
    // calculated by subtracting
    // debts of 'p' from credits of 'p'
    for (int p = 0; p < N; p++)
        for (int i = 0; i < N; i++)
            amount[p] += (graph[i,p] - graph[p,i]);

    minCashFlowRec(amount);
}

// Driver code
```

```
public static void Main ()  
{  
    // graph[i][j] indicates the amount  
    // that person i needs to pay person j  
    int [,]graph = { {0, 1000, 2000},  
                    {0, 0, 5000},  
                    {0, 0, 0},};  
  
    // Print the solution  
    minCashFlow(graph);  
}  
}  
  
// This code is contributed by nitin mittal.
```

Output:

```
Person 1 pays 4000 to Person 2  
Person 0 pays 3000 to Person 2
```

Algorithmic Paradigm: Greedy

Time Complexity: $O(N^2)$ where N is the number of persons.

This article is contributed by **Gaurav**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Improved By : [nitin mittal](#)

Source

<https://www.geeksforgeeks.org/minimize-cash-flow-among-given-set-friends-borrowed-money/>

Chapter 15

Boggle (Find all possible words in a board of characters)

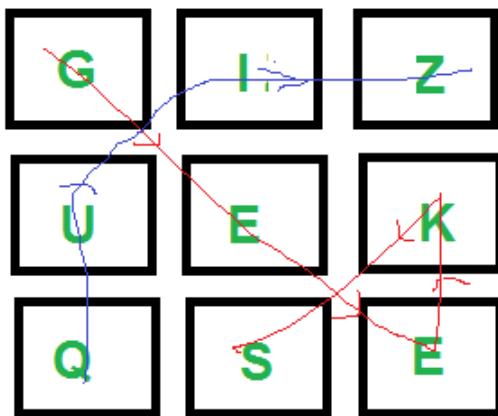
Boggle (Find all possible words in a board of characters) Set 1 - GeeksforGeeks

Given a dictionary, a method to do lookup in dictionary and a M x N board where every cell has one character. Find all possible words that can be formed by a sequence of adjacent characters. Note that we can move to any of 8 adjacent characters, but a word should not have multiple instances of same cell.

Example:

```
Input: dictionary[] = {"GEEKS", "FOR", "QUIZ", "GO"};
       boggle[][]   = {{'G','I','Z'},
                      {'U','E','K'},
                      {'Q','S','E'}};
       isWord(str): returns true if str is present in dictionary
                     else false.
```

```
Output: Following words of dictionary are present
        GEEKS
        QUIZ
```



The idea is to consider every character as a starting character and find all words starting with it. All words starting from a character can be found using [Depth First Traversal](#). We do depth first traversal starting from every cell. We keep track of visited cells to make sure that a cell is considered only once in a word.

```
// C++ program for Boggle game
#include<iostream>
#include<cstring>
using namespace std;

#define M 3
#define N 3

// Let the given dictionary be following
string dictionary[] = {"GEEKS", "FOR", "QUIZ", "GO"};
int n = sizeof(dictionary)/sizeof(dictionary[0]);

// A given function to check if a given string is present in
// dictionary. The implementation is naive for simplicity. As
// per the question dictionary is given to us.
bool isWord(string &str)
{
    // Linearly search all words
    for (int i=0; i<n; i++)
        if (str.compare(dictionary[i]) == 0)
            return true;
    return false;
}

// A recursive function to print all words present on boggle
void findWordsUtil(char boggle[M][N], bool visited[M][N], int i,
                    int j, string &str)
{
```

```

// Mark current cell as visited and append current character
// to str
visited[i][j] = true;
str = str + boggle[i][j];

// If str is present in dictionary, then print it
if (isWord(str))
    cout << str << endl;

// Traverse 8 adjacent cells of boggle[i][j]
for (int row=i-1; row<=i+1 && row<M; row++)
    for (int col=j-1; col<=j+1 && col<N; col++)
        if (row>=0 && col>=0 && !visited[row][col])
            findWordsUtil(boggle,visited, row, col, str);

// Erase current character from string and mark visited
// of current cell as false
str.erase(str.length()-1);
visited[i][j] = false;
}

// Prints all words present in dictionary.
void findWords(char boggle[M][N])
{
    // Mark all characters as not visited
    bool visited[M][N] = {{false}};

    // Initialize current string
    string str = "";

    // Consider every character and look for all words
    // starting with this character
    for (int i=0; i<M; i++)
        for (int j=0; j<N; j++)
            findWordsUtil(boggle, visited, i, j, str);
}

// Driver program to test above function
int main()
{
    char boggle[M][N] = {{'G','I','Z'},
                         {'U','E','K'},
                         {'Q','S','E'}};

    cout << "Following words of dictionary are present\n";
    findWords(boggle);
    return 0;
}

```

Output:

```
Following words of dictionary are present
GEEKS
QUIZ
```

Note that the above solution may print the same word multiple times. For example, if we add “SEEK” to the dictionary, it is printed multiple times. To avoid this, we can use hashing to keep track of all printed words.

In below set 2, we have discussed Trie based optimized solution:

Boggle Set 2 (Using Trie)

This article is contributed by **Rishabh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

<https://www.geeksforgeeks.org/boggle-find-possible-words-board-characters/>

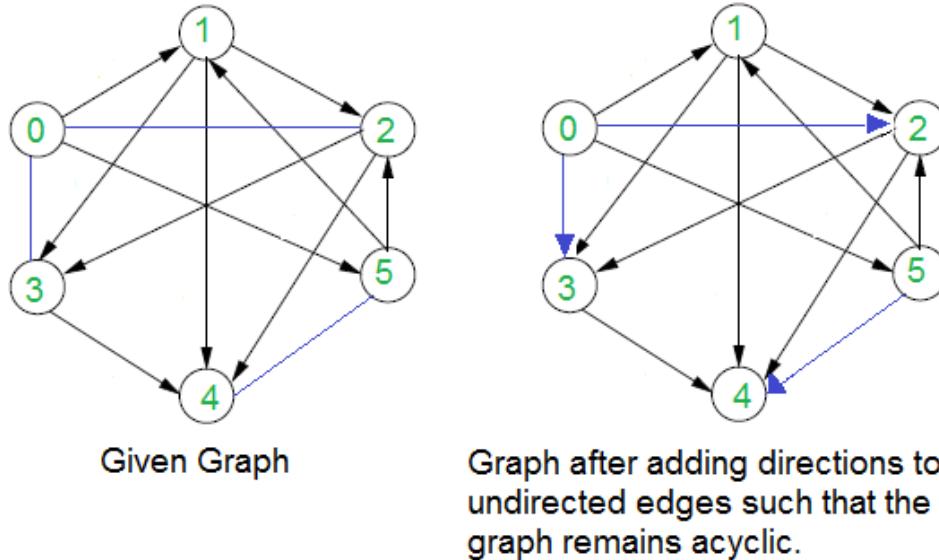
Chapter 16

Assign directions to edges so that the directed graph remains acyclic

Assign directions to edges so that the directed graph remains acyclic - GeeksforGeeks

Given a graph with both directed and undirected edges. It is given that the directed edges don't form cycle. How to assign directions to undirected edges so that the graph (with all directed edges) remains acyclic even after the assignment?

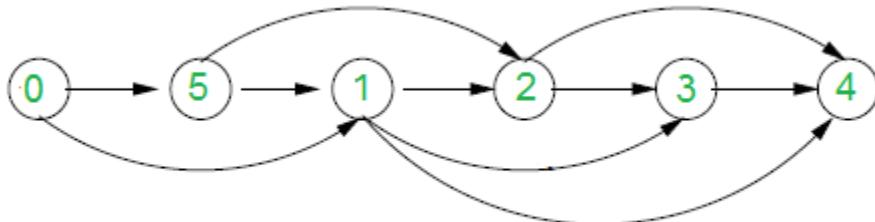
For example, in the below graph, blue edges don't have directions.



We strongly recommend to minimize your browser and try this yourself first.

The idea is to use [Topological Sorting](#). Following are two steps used in the algorithm.

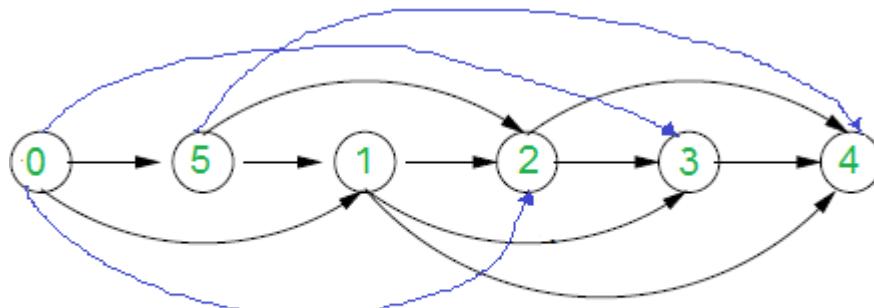
- 1) Consider the subgraph with directed edges only and find topological sorting of the subgraph. In the above example, topological sorting is $\{0, 5, 1, 2, 3, 4\}$. Below diagram shows topological sorting for the above example graph.



Step 1: Find Topological Sorting

- 2) Use above topological sorting to assign directions to undirected edges. For every undirected edge (u, v) , assign it direction from u to v if u comes before v in topological sorting, else assign it direction from v to u .

Below diagram shows assigned directions in the example graph.



Step 2: Add directions to undirected edges

Source: <http://courses.csail.mit.edu/6.006/oldquizzes/solutions/q2-f2009-sol.pdf>

This article is contributed by **Aditya Agrawal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

<https://www.geeksforgeeks.org/assign-directions-to-edges-so-that-the-directed-graph-remains-acyclic/>

Chapter 17

Prim's Minimum Spanning Tree (MST))

Prim's Minimum Spanning Tree (MST) Greedy Algo-5 - GeeksforGeeks

We have discussed [Kruskal's algorithm for Minimum Spanning Tree](#). Like Kruskal's algorithm, Prim's algorithm is also a [Greedy algorithm](#). It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

A group of edges that connects two set of vertices in a graph is called [cut in graph theory](#). So, at every step of Prim's algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the vertices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).

How does Prim's Algorithm Work? The idea behind Prim's algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a *Spanning Tree*. And they must be connected with the minimum weight edge to make it a *Minimum Spanning Tree*.

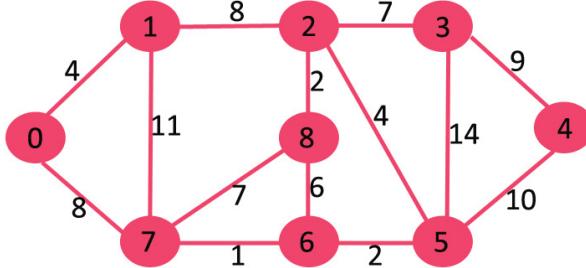
Algorithm

- 1) Create a set *mstSet* that keeps track of vertices already included in MST.
- 2) Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
- 3) While *mstSet* doesn't include all vertices
 -a) Pick a vertex *u* which is not there in *mstSet* and has minimum key value.
 -b) Include *u* to *mstSet*.
 -c) Update key value of all adjacent vertices of *u*. To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*, if weight of edge *u-v* is less than the previous key value of *v*, update the key value as weight of *u-v*

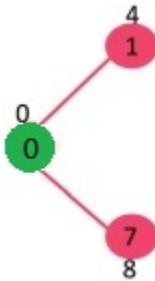
The idea of using key values is to pick the minimum weight edge from [cut](#). The key values

are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

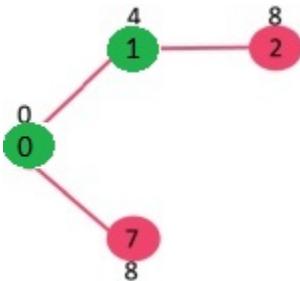
Let us understand with the following example:



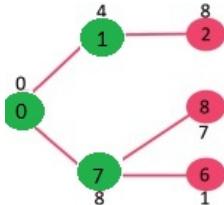
The set *mstSet* is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum key value. The vertex 0 is picked, include it in *mstSet*. So *mstSet* becomes {0}. After including to *mstSet*, update key values of adjacent vertices. Adjacent vertices of 0 are 1 and 7. The key values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their key values, only the vertices with finite key values are shown. The vertices included in MST are shown in green color.



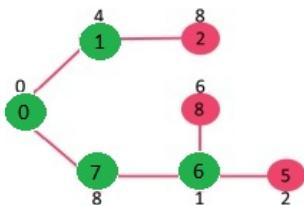
Pick the vertex with minimum key value and not already included in MST (not in *mstSET*). The vertex 1 is picked and added to *mstSet*. So *mstSet* now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.



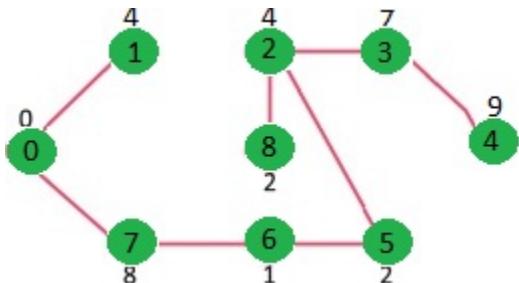
Pick the vertex with minimum key value and not already included in MST (not in *mstSET*). We can either pick vertex 7 or vertex 2, let vertex 7 is picked. So *mstSet* now becomes {0, 1, 7}. Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (7 and 1 respectively).



Pick the vertex with minimum key value and not already included in MST (not in *mstSet*). Vertex 6 is picked. So *mstSet* now becomes {0, 1, 7, 6}. Update the key values of adjacent vertices of 6. The key value of vertex 5 and 8 are updated.



We repeat the above steps until *mstSet* includes all vertices of given graph. Finally, we get the following graph.



How to implement the above algorithm?

We use a boolean array *mstSet[]* to represent the set of vertices included in MST. If a value *mstSet[v]* is true, then vertex v is included in MST, otherwise not. Array *key[]* is used to store key values of all vertices. Another array *parent[]* to store indexes of parent nodes in MST. The parent array is the output array which is used to show the constructed MST.

C/C++

```
// A C / C++ program for Prim's Minimum
// Spanning Tree (MST) algorithm. The program is
// for adjacency matrix representation of the graph
#include <stdio.h>
#include <limits.h>
#include<stdbool.h>
// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with
// minimum key value, from the set of vertices
```

```

// not yet included in MST
int minKey(int key[], bool mstSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

// A utility function to print the
// constructed MST stored in parent[]
int printMST(int parent[], int n, int graph[V][V])
{
    printf("Edge \tWeight\n");
    for (int i = 1; i < V; i++)
        printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
}

// Function to construct and print MST for
// a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
    // Array to store constructed MST
    int parent[V];
    // Key values used to pick minimum weight edge in cut
    int key[V];
    // To represent set of vertices not yet included in MST
    bool mstSet[V];

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    // Always include first 1st vertex in MST.
    // Make key 0 so that this vertex is picked as first vertex.
    key[0] = 0;
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (int count = 0; count < V-1; count++)
    {
        // Pick the minimum key vertex from the
        // set of vertices not yet included in MST

```

```
int u = minKey(key, mstSet);

// Add the picked vertex to the MST Set
mstSet[u] = true;

// Update key value and parent index of
// the adjacent vertices of the picked vertex.
// Consider only those vertices which are not
// yet included in MST
for (int v = 0; v < V; v++)

    // graph[u][v] is non zero only for adjacent vertices of m
    // mstSet[v] is false for vertices not yet included in MST
    // Update the key only if graph[u][v] is smaller than key[v]
    if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
        parent[v] = u, key[v] = graph[u][v];
}

// print the constructed MST
printMST(parent, V, graph);
}

// driver program to test above function
int main()
{
/* Let us create the following graph
   2 3
   (0)--(1)--(2)
   | / \ |
   6| 8/ \5 |7
   | /       \ |
   (3)-----(4)
   9           */
int graph[V][V] = {{0, 2, 0, 6, 0},
                    {2, 0, 3, 8, 5},
                    {0, 3, 0, 0, 7},
                    {6, 8, 0, 0, 9},
                    {0, 5, 7, 9, 0}};

// Print the solution
primMST(graph);

return 0;
}
```

Java

```
// A Java program for Prim's Minimum Spanning Tree (MST) algorithm.
// The program is for adjacency matrix representation of the graph

import java.util.*;
import java.lang.*;
import java.io.*;

class MST
{
    // Number of vertices in the graph
    private static final int V=5;

    // A utility function to find the vertex with minimum key
    // value, from the set of vertices not yet included in MST
    int minKey(int key[], Boolean mstSet[])
    {
        // Initialize min value
        int min = Integer.MAX_VALUE, min_index=-1;

        for (int v = 0; v < V; v++)
            if (mstSet[v] == false && key[v] < min)
            {
                min = key[v];
                min_index = v;
            }

        return min_index;
    }

    // A utility function to print the constructed MST stored in
    // parent[]
    void printMST(int parent[], int n, int graph[][])
    {
        System.out.println("Edge \tWeight");
        for (int i = 1; i < V; i++)
            System.out.println(parent[i]+ " - "+ i+"\t"+
                               graph[i][parent[i]]);
    }

    // Function to construct and print MST for a graph represented
    // using adjacency matrix representation
    void primMST(int graph[][])
    {
        // Array to store constructed MST
        int parent[] = new int[V];

        // Key values used to pick minimum weight edge in cut
        int key[] = new int [V];
```

```

// To represent set of vertices not yet included in MST
Boolean mstSet[] = new Boolean[V];

// Initialize all keys as INFINITE
for (int i = 0; i < V; i++)
{
    key[i] = Integer.MAX_VALUE;
    mstSet[i] = false;
}

// Always include first 1st vertex in MST.
key[0] = 0;      // Make key 0 so that this vertex is
                 // picked as first vertex
parent[0] = -1; // First node is always root of MST

// The MST will have V vertices
for (int count = 0; count < V-1; count++)
{
    // Pick thd minimum key vertex from the set of vertices
    // not yet included in MST
    int u = minKey(key, mstSet);

    // Add the picked vertex to the MST Set
    mstSet[u] = true;

    // Update key value and parent index of the adjacent
    // vertices of the picked vertex. Consider only those
    // vertices which are not yet included in MST
    for (int v = 0; v < V; v++)

        // graph[u][v] is non zero only for adjacent vertices of m
        // mstSet[v] is false for vertices not yet included in MST
        // Update the key only if graph[u][v] is smaller than key[v]
        if (graph[u][v] != 0 && mstSet[v] == false &&
            graph[u][v] < key[v])
        {
            parent[v] = u;
            key[v] = graph[u][v];
        }
    }

    // print the constructed MST
    printMST(parent, V, graph);
}

public static void main (String[] args)
{

```

```
/* Let us create the following graph
2 3
(0)--(1)--(2)
| / \ |
6| 8/ \5 |7
| /       \ |
(3)-----(4)
         9      */
MST t = new MST();
int graph[][] = new int[][] {{0, 2, 0, 6, 0},
                             {2, 0, 3, 8, 5},
                             {0, 3, 0, 0, 7},
                             {6, 8, 0, 0, 9},
                             {0, 5, 7, 9, 0}};

// Print the solution
t.primMST(graph);
}
}

// This code is contributed by Aakash Hasija
```

Python

```
# A Python program for Prim's Minimum Spanning Tree (MST) algorithm.
# The program is for adjacency matrix representation of the graph

import sys # Library for INT_MAX

class Graph():

    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                     for row in range(vertices)]

    # A utility function to print the constructed MST stored in parent[]
    def printMST(self, parent):
        print "Edge \tWeight"
        for i in range(1,self.V):
            print parent[i],"-",i," \t",self.graph[i][ parent[i] ]

    # A utility function to find the vertex with
    # minimum distance value, from the set of vertices
    # not yet included in shortest path tree
    def minKey(self, key, mstSet):

        # Initialize min value
        min = sys.maxint
```

```

for v in range(self.V):
    if key[v] < min and mstSet[v] == False:
        min = key[v]
        min_index = v

return min_index

# Function to construct and print MST for a graph
# represented using adjacency matrix representation
def primMST(self):

    #Key values used to pick minimum weight edge in cut
    key = [sys.maxint] * self.V
    parent = [None] * self.V # Array to store constructed MST
    # Make key 0 so that this vertex is picked as first vertex
    key[0] = 0
    mstSet = [False] * self.V

    parent[0] = -1 # First node is always the root of

    for cout in range(self.V):

        # Pick the minimum distance vertex from
        # the set of vertices not yet processed.
        # u is always equal to src in first iteration
        u = self.minKey(key, mstSet)

        # Put the minimum distance vertex in
        # the shortest path tree
        mstSet[u] = True

        # Update dist value of the adjacent vertices
        # of the picked vertex only if the current
        # distance is greater than new distance and
        # the vertex is not in the shortest path tree
        for v in range(self.V):
            # graph[u][v] is non zero only for adjacent vertices of m
            # mstSet[v] is false for vertices not yet included in MST
            # Update the key only if graph[u][v] is smaller than key[v]
            if self.graph[u][v] > 0 and mstSet[v] == False and key[v] > self.graph[u][v]:
                key[v] = self.graph[u][v]
                parent[v] = u

    self.printMST(parent)

g = Graph(5)
g.graph = [ [0, 2, 0, 6, 0],

```

```
[2, 0, 3, 8, 5],  
[0, 3, 0, 0, 7],  
[6, 8, 0, 0, 9],  
[0, 5, 7, 9, 0]]  
  
g.primMST();  
  
# Contributed by Divyanshu Mehta  
  
C#  
  
// A C# program for Prim's Minimum  
// Spanning Tree (MST) algorithm.  
// The program is for adjacency  
// matrix representation of the graph  
using System;  
class MST {  
  
    // Number of vertices in the graph  
    static int V = 5;  
  
    // A utility function to find  
    // the vertex with minimum key  
    // value, from the set of vertices  
    // not yet included in MST  
    static int minKey(int []key, bool []mstSet)  
{  
  
    // Initialize min value  
    int min = int.MaxValue, min_index = -1;  
  
    for (int v = 0; v < V; v++)  
        if (mstSet[v] == false && key[v] < min)  
        {  
            min = key[v];  
            min_index = v;  
        }  
  
    return min_index;  
}  
  
// A utility function to print  
// the constructed MST stored in  
// parent[]  
static void printMST(int []parent, int n, int [,]graph)  
{  
    Console.WriteLine("Edge \tWeight");  
    for (int i = 1; i < V; i++)
```

```
Console.WriteLine(parent[i]+ " - "+ i+"\t"+  
                  graph[i,parent[i]]);  
}  
  
// Function to construct and  
// print MST for a graph represented  
// using adjacency matrix representation  
static void primMST(int [,]graph)  
{  
  
    // Array to store constructed MST  
    int []parent = new int[V];  
  
    // Key values used to pick  
    // minimum weight edge in cut  
    int []key = new int [V];  
  
    // To represent set of vertices  
    // not yet included in MST  
    bool []mstSet = new bool[V];  
  
    // Initialize all keys  
    // as INFINITE  
    for (int i = 0; i < V; i++)  
    {  
        key[i] = int.MaxValue;  
        mstSet[i] = false;  
    }  
  
    // Always include first 1st vertex in MST.  
    // Make key 0 so that this vertex is  
    // picked as first vertex  
    // First node is always root of MST  
    key[0] = 0;  
    parent[0] = -1;  
  
    // The MST will have V vertices  
    for (int count = 0; count < V - 1; count++)  
    {  
  
        // Pick thd minimum key vertex  
        // from the set of vertices  
        // not yet included in MST  
        int u = minKey(key, mstSet);  
  
        // Add the picked vertex  
        // to the MST Set  
        mstSet[u] = true;
```

```
// Update key value and parent
// index of the adjacent vertices
// of the picked vertex. Consider
// only those vertices which are
// not yet included in MST
for (int v = 0; v < V; v++)

    // graph[u][v] is non zero only
    // for adjacent vertices of m
    // mstSet[v] is false for vertices
    // not yet included in MST Update
    // the key only if graph[u][v] is
    // smaller than key[v]
    if (graph[u,v] != 0 && mstSet[v] == false &&
        graph[u,v] < key[v])
    {
        parent[v] = u;
        key[v] = graph[u,v];
    }
}

// print the constructed MST
printMST(parent, V, graph);
}

// Driver Code
public static void Main ()
{

/* Let us create the following graph
2 3
(0)--(1)--(2)
| / \ |
6| 8/ \5 |7
| / \ |
(3)-----(4)
         9 */
}

int [,]graph = new int[,] {{0, 2, 0, 6, 0},
                           {2, 0, 3, 8, 5},
                           {0, 3, 0, 0, 7},
                           {6, 8, 0, 0, 9},
                           {0, 5, 7, 9, 0}};

// Print the solution
primMST(graph);
}
```

```
}
```

```
// This code is contributed by anuj_67.
```

Output:

Edge	Weight
0 - 1	2
1 - 2	3
0 - 3	6
1 - 4	5

Time Complexity of the above program is $O(V^2)$. If the input graph is represented using [adjacency list](#), then the time complexity of Prim's algorithm can be reduced to $O(E \log V)$ with the help of binary heap. Please see [Prim's MST for Adjacency List Representation](#) for more details.

Improved By : [vt_m](#), [AnkurKarmakar](#)

Source

<https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/>

Chapter 18

Applications of Minimum Spanning Tree Problem

Applications of Minimum Spanning Tree Problem - GeeksforGeeks

Minimum Spanning Tree (MST) problem: Given connected graph G with positive edge weights, find a min weight set of edges that connects all of the vertices.

MST is fundamental problem with diverse applications.

Network design.

- *telephone, electrical, hydraulic, TV cable, computer, road*

The standard application is to a problem like phone network design. You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. It should be a spanning tree, since if a network isn't a tree you can always remove some edges and save money.

Approximation algorithms for NP-hard problems.

- *traveling salesperson problem, Steiner tree*

A less obvious application is that the minimum spanning tree can be used to approximately solve the traveling salesman problem. A convenient formal way of defining this problem is to find the shortest path that visits each point at least once.

Note that if you have a path visiting all points exactly once, it's a special kind of tree. For instance in the example above, twelve of sixteen spanning trees are actually paths. If you have a path visiting some vertices more than once, you can always drop some edges to get a tree. So in general the MST weight is less than the TSP weight, because it's a minimization over a strictly larger set.

On the other hand, if you draw a path tracing around the minimum spanning tree, you trace each edge twice and visit all points, so the TSP weight is less than twice the MST weight. Therefore this tour is within a factor of two of optimal.

Indirect applications.

- max bottleneck paths
- LDPC codes for error correction
- image registration with Renyi entropy
- learning salient features for real-time face verification
- reducing data storage in sequencing amino acids in a protein
- model locality of particle interactions in turbulent fluid flows
- autoconfig protocol for Ethernet bridging to avoid cycles in a network

Cluster analysis

k clustering problem can be viewed as finding an MST and deleting the k-1 most expensive edges.

Sources:

<http://www.cs.princeton.edu/courses/archive/spr07/cos226/lectures/mst.pdf>
<http://www.ics.uci.edu/~eppstein/161/960206.html>

Source

<https://www.geeksforgeeks.org/applications-of-minimum-spanning-tree/>

Chapter 19

Prim's MST for Adjacency List Representation

Prim's MST for Adjacency List Representation Greedy Algo-6 - GeeksforGeeks

We recommend to read following two posts as a prerequisite of this post.

1. [Greedy Algorithms Set 5 \(Prim's Minimum Spanning Tree \(MST\)\)](#)
2. [Graph and its representations](#)

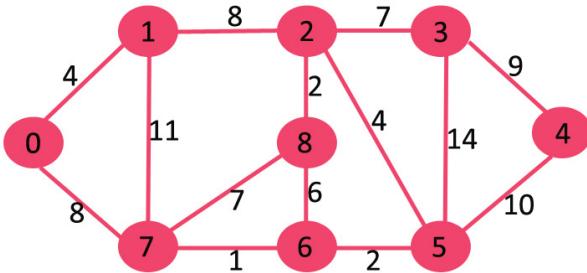
We have discussed [Prim's algorithm and its implementation for adjacency matrix representation of graphs](#). The time complexity for the matrix representation is $O(V^2)$. In this post, $O(E\log V)$ algorithm for adjacency list representation is discussed.

As discussed in the previous post, in Prim's algorithm, two sets are maintained, one set contains list of vertices already included in MST, other set contains vertices not yet included. With adjacency list representation, all vertices of a graph can be traversed in $O(V+E)$ time using [BFS](#). The idea is to traverse all vertices of graph using [BFS](#) and use a Min Heap to store the vertices not yet included in MST. Min Heap is used as a priority queue to get the minimum weight edge from the [cut](#). Min Heap is used as time complexity of operations like extracting minimum element and decreasing key value is $O(\log V)$ in Min Heap.

Following are the detailed steps.

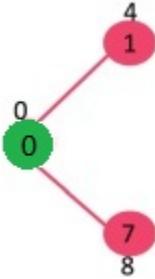
- 1) Create a Min Heap of size V where V is the number of vertices in the given graph. Every node of min heap contains vertex number and key value of the vertex.
- 2) Initialize Min Heap with first vertex as root (the key value assigned to first vertex is 0). The key value assigned to all other vertices is INF (infinite).
- 3) While Min Heap is not empty, do following
 -a) Extract the min value node from Min Heap. Let the extracted vertex be u .
 -b) For every adjacent vertex v of u , check if v is in Min Heap (not yet included in MST). If v is in Min Heap and its key value is more than weight of $u-v$, then update the key value of v as weight of $u-v$.

Let us understand the above algorithm with the following example:

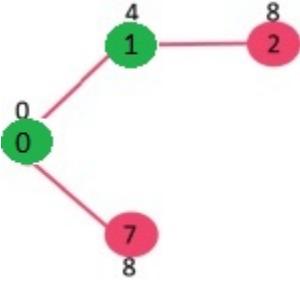


Initially, key value of first vertex is 0 and INF (infinite) for all other vertices. So vertex 0 is extracted from Min Heap and key values of vertices adjacent to 0 (1 and 7) are updated. Min Heap contains all vertices except vertex 0.

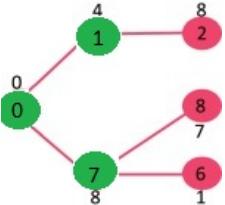
The vertices in green color are the vertices included in MST.



Since key value of vertex 1 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 1 are updated (Key is updated if the a vertex is not in Min Heap and previous key value is greater than the weight of edge from 1 to the adjacent). Min Heap contains all vertices except vertex 0 and 1.

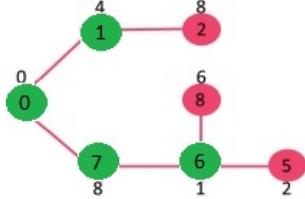


Since key value of vertex 7 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 7 are updated (Key is updated if the a vertex is not in Min Heap and previous key value is greater than the weight of edge from 7 to the adjacent). Min Heap contains all vertices except vertex 0, 1 and 7.

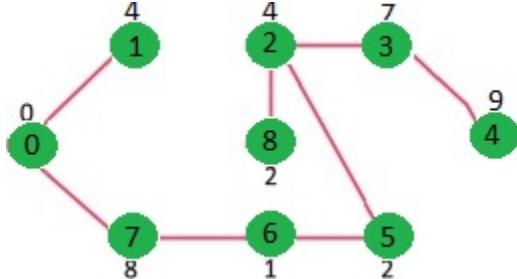


Since key value of vertex 6 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 6 are updated (Key is updated if the a

vertex is not in Min Heap and previous key value is greater than the weight of edge from 6 to the adjacent). Min Heap contains all vertices except vertex 0, 1, 7 and 6.



The above steps are repeated for rest of the nodes in Min Heap till Min Heap becomes empty



C++

```
// C / C++ program for Prim's MST for adjacency list representation of graph

#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

// A structure to represent a node in adjacency list
struct AdjListNode {
    int dest;
    int weight;
    struct AdjListNode* next;
};

// A structure to represent an adjacency list
struct AdjList {
    struct AdjListNode* head; // pointer to head node of list
};

// A structure to represent a graph. A graph is an array of adjacency lists.
// Size of array will be V (number of vertices in graph)
struct Graph {
    int V;
    struct AdjList* array;
};

// A utility function to create a new adjacency list node
struct AdjListNode* newAdjListNode(int dest, int weight)
```

```
{
    struct AdjListNode* newNode = (struct AdjListNode*)malloc(sizeof(struct AdjListNode));
    newNode->dest = dest;
    newNode->weight = weight;
    newNode->next = NULL;
    return newNode;
}

// A utility function that creates a graph of V vertices
struct Graph* createGraph(int V)
{
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->V = V;

    // Create an array of adjacency lists. Size of array will be V
    graph->array = (struct AdjList*)malloc(V * sizeof(struct AdjList));

    // Initialize each adjacency list as empty by making head as NULL
    for (int i = 0; i < V; ++i)
        graph->array[i].head = NULL;

    return graph;
}

// Adds an edge to an undirected graph
void addEdge(struct Graph* graph, int src, int dest, int weight)
{
    // Add an edge from src to dest. A new node is added to the adjacency
    // list of src. The node is added at the begining
    struct AdjListNode* newNode = newAdjListNode(dest, weight);
    newNode->next = graph->array[src].head;
    graph->array[src].head = newNode;

    // Since graph is undirected, add an edge from dest to src also
    newNode = newAdjListNode(src, weight);
    newNode->next = graph->array[dest].head;
    graph->array[dest].head = newNode;
}

// Structure to represent a min heap node
struct MinHeapNode {
    int v;
    int key;
};

// Structure to represent a min heap
struct MinHeap {
    int size; // Number of heap nodes present currently
```

```

        int capacity; // Capacity of min heap
        int* pos; // This is needed for decreaseKey()
        struct MinHeapNode** array;
    };

    // A utility function to create a new Min Heap Node
    struct MinHeapNode* newMinHeapNode(int v, int key)
    {
        struct MinHeapNode* minHeapNode = (struct MinHeapNode*)malloc(sizeof(struct MinHeapNode));
        minHeapNode->v = v;
        minHeapNode->key = key;
        return minHeapNode;
    }

    // A utilit function to create a Min Heap
    struct MinHeap* createMinHeap(int capacity)
    {
        struct MinHeap* minHeap = (struct MinHeap*)malloc(sizeof(struct MinHeap));
        minHeap->pos = (int*)malloc(capacity * sizeof(int));
        minHeap->size = 0;
        minHeap->capacity = capacity;
        minHeap->array = (struct MinHeapNode**)malloc(capacity * sizeof(struct MinHeapNode*));
        return minHeap;
    }

    // A utility function to swap two nodes of min heap. Needed for min heapify
    void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b)
    {
        struct MinHeapNode* t = *a;
        *a = *b;
        *b = t;
    }

    // A standard function to heapify at given idx
    // This function also updates position of nodes when they are swapped.
    // Position is needed for decreaseKey()
    void minHeapify(struct MinHeap* minHeap, int idx)
    {
        int smallest, left, right;
        smallest = idx;
        left = 2 * idx + 1;
        right = 2 * idx + 2;

        if (left < minHeap->size && minHeap->array[left]->key < minHeap->array[smallest]->key)
            smallest = left;

        if (right < minHeap->size && minHeap->array[right]->key < minHeap->array[smallest]->key)
            smallest = right;
    }
}

```

```

if (smallest != idx) {
    // The nodes to be swapped in min heap
    MinHeapNode* smallestNode = minHeap->array[smallest];
    MinHeapNode* idxNode = minHeap->array[idx];

    // Swap positions
    minHeap->pos[smallestNode->v] = idx;
    minHeap->pos[idxNode->v] = smallest;

    // Swap nodes
    swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

    minHeapify(minHeap, smallest);
}
}

// A utility function to check if the given minHeap is ampty or not
int isEmpty(struct MinHeap* minHeap)
{
    return minHeap->size == 0;
}

// Standard function to extract minimum node from heap
struct MinHeapNode* extractMin(struct MinHeap* minHeap)
{
    if (isEmpty(minHeap))
        return NULL;

    // Store the root node
    struct MinHeapNode* root = minHeap->array[0];

    // Replace root node with last node
    struct MinHeapNode* lastNode = minHeap->array[minHeap->size - 1];
    minHeap->array[0] = lastNode;

    // Update position of last node
    minHeap->pos[root->v] = minHeap->size - 1;
    minHeap->pos[lastNode->v] = 0;

    // Reduce heap size and heapify root
    --minHeap->size;
    minHeapify(minHeap, 0);

    return root;
}

// Function to decreasy key value of a given vertex v. This function

```

```

// uses pos[] of min heap to get the current index of node in min heap
void decreaseKey(struct MinHeap* minHeap, int v, int key)
{
    // Get the index of v in heap array
    int i = minHeap->pos[v];

    // Get the node and update its key value
    minHeap->array[i]->key = key;

    // Travel up while the complete tree is not hepified.
    // This is a O(Logn) loop
    while (i && minHeap->array[i]->key < minHeap->array[(i - 1) / 2]->key) {
        // Swap this node with its parent
        minHeap->pos[minHeap->array[i]->v] = (i - 1) / 2;
        minHeap->pos[minHeap->array[(i - 1) / 2]->v] = i;
        swapMinHeapNode(&minHeap->array[i], &minHeap->array[(i - 1) / 2]);

        // move to parent index
        i = (i - 1) / 2;
    }
}

// A utility function to check if a given vertex
// 'v' is in min heap or not
bool isInMinHeap(struct MinHeap* minHeap, int v)
{
    if (minHeap->pos[v] < minHeap->size)
        return true;
    return false;
}

// A utility function used to print the constructed MST
void printArr(int arr[], int n)
{
    for (int i = 1; i < n; ++i)
        printf("%d - %d\n", arr[i], i);
}

// The main function that constructs Minimum Spanning Tree (MST)
// using Prim's algorithm
void PrimMST(struct Graph* graph)
{
    int V = graph->V; // Get the number of vertices in graph
    int parent[V]; // Array to store constructed MST
    int key[V]; // Key values used to pick minimum weight edge in cut

    // minHeap represents set E
    struct MinHeap* minHeap = createMinHeap(V);

```

```

// Initialize min heap with all vertices. Key value of
// all vertices (except 0th vertex) is initially infinite
for (int v = 1; v < V; ++v) {
    parent[v] = -1;
    key[v] = INT_MAX;
    minHeap->array[v] = newMinHeapNode(v, key[v]);
    minHeap->pos[v] = v;
}

// Make key value of 0th vertex as 0 so that it
// is extracted first
key[0] = 0;
minHeap->array[0] = newMinHeapNode(0, key[0]);
minHeap->pos[0] = 0;

// Initially size of min heap is equal to V
minHeap->size = V;

// In the followin loop, min heap contains all nodes
// not yet added to MST.
while (!isEmpty(minHeap)) {
    // Extract the vertex with minimum key value
    struct MinHeapNode* minHeapNode = extractMin(minHeap);
    int u = minHeapNode->v; // Store the extracted vertex number

    // Traverse through all adjacent vertices of u (the extracted
    // vertex) and update their key values
    struct AdjListNode* pCrawl = graph->array[u].head;
    while (pCrawl != NULL) {
        int v = pCrawl->dest;

        // If v is not yet included in MST and weight of u-v is
        // less than key value of v, then update key value and
        // parent of v
        if (isInMinHeap(minHeap, v) && pCrawl->weight < key[v]) {
            key[v] = pCrawl->weight;
            parent[v] = u;
            decreaseKey(minHeap, v, key[v]);
        }
        pCrawl = pCrawl->next;
    }
}

// print edges of MST
printArr(parent, V);
}

```

```
// Driver program to test above functions
int main()
{
    // Let us create the graph given in above fugure
    int V = 9;
    struct Graph* graph = createGraph(V);
    addEdge(graph, 0, 1, 4);
    addEdge(graph, 0, 7, 8);
    addEdge(graph, 1, 2, 8);
    addEdge(graph, 1, 7, 11);
    addEdge(graph, 2, 3, 7);
    addEdge(graph, 2, 8, 2);
    addEdge(graph, 2, 5, 4);
    addEdge(graph, 3, 4, 9);
    addEdge(graph, 3, 5, 14);
    addEdge(graph, 4, 5, 10);
    addEdge(graph, 5, 6, 2);
    addEdge(graph, 6, 7, 1);
    addEdge(graph, 6, 8, 6);
    addEdge(graph, 7, 8, 7);

    PrimMST(graph);

    return 0;
}
```

Java

```
// Java program for Prim's MST for
// adjacency list representation of graph
import java.util.LinkedList;
import java.util.PriorityQueue;
import java.util.Comparator;

public class prims {
    class node1 {

        // Stores destination vertex in adjacency list
        int dest;

        // Stores weight of a vertex in adjacency list
        int weight;

        // Constructor
        node1(int a, int b)
        {
            dest = a;
            weight = b;
        }
    }
}
```

```

        }
    }
}

static class Graph {

    // Number of vertices in the graph
    int V;

    // List of adjacent nodes of a given vertex
    LinkedList<node1>[] adj;

    // Constructor
    Graph(int e)
    {
        V = e;
        adj = new LinkedList[V];
        for (int o = 0; o < V; o++)
            adj[o] = new LinkedList<>();
    }

    // class to represent a node in PriorityQueue
    // Stores a vertex and its corresponding
    // key value
    class node {
        int vertex;
        int key;
    }

    // Comparator class created for PriorityQueue
    // returns 1 if node0.key > node1.key
    // returns 0 if node0.key < node1.key and
    // returns -1 otherwise
    class comparator implements Comparator<node> {

        @Override
        public int compare(node node0, node node1)
        {
            return node0.key - node1.key;
        }
    }

    // method to add an edge
    // between two vertices
    void addEdge(Graph graph, int src, int dest, int weight)
    {

        node1 node0 = new node1(dest, weight);
        node1 node = new node1(src, weight);
    }
}

```

```

graph.adj[src].addLast(node0);
graph.adj[dest].addLast(node);
}

// method used to find the mst
void prims_mst(Graph graph)
{
    // Whether a vertex is in PriorityQueue or not
    Boolean[] mstset = new Boolean[graph.V];
    node[] e = new node[graph.V];

    // Stores the parents of a vertex
    int[] parent = new int[graph.V];

    for (int o = 0; o < graph.V; o++)
        e[o] = new node();

    for (int o = 0; o < graph.V; o++) {

        // Initialize to false
        mstset[o] = false;

        // Initialize key values to infinity
        e[o].key = Integer.MAX_VALUE;
        e[o].vertex = o;
        parent[o] = -1;
    }

    // Include the source vertex in mstset
    mstset[0] = true;

    // Set key value to 0
    // so that it is extracted first
    // out of PriorityQueue
    e[0].key = 0;

    // PriorityQueue
    PriorityQueue<node> queue = new PriorityQueue<>(graph.V, new comparator());

    for (int o = 0; o < graph.V; o++)
        queue.add(e[o]);

    // Loops until the PriorityQueue is not empty
    while (!queue.isEmpty()) {

        // Extracts a node with min key value
        node node0 = queue.poll();

```

```

// Include that node into mstset
mstset[node0.vertex] = true;

// For all adjacent vertex of the extracted vertex V
for (node1 iterator : graph.adj[node0.vertex]) {

    // If V is in PriorityQueue
    if (mstset[iterator.dest] == false) {
        // If the key value of the adjacent vertex is
        // more than the extracted key
        // update the key value of adjacent vertex
        // to update first remove and add the updated vertex
        if (e[iterator.dest].key > iterator.weight) {
            queue.remove(e[iterator.dest]);
            e[iterator.dest].key = iterator.weight;
            queue.add(e[iterator.dest]);
            parent[iterator.dest] = node0.vertex;
        }
    }
}

// Prints the vertex pair of mst
for (int o = 1; o < graph.V; o++)
    System.out.println(parent[o] + " "
                       + "-"
                       + " " + o);
}

public static void main(String[] args)
{
    int V = 9;

    Graph graph = new Graph(V);

    prims e = new prims();

    e.addEdge(graph, 0, 1, 4);
    e.addEdge(graph, 0, 7, 8);
    e.addEdge(graph, 1, 2, 8);
    e.addEdge(graph, 1, 7, 11);
    e.addEdge(graph, 2, 3, 7);
    e.addEdge(graph, 2, 8, 2);
    e.addEdge(graph, 2, 5, 4);
    e.addEdge(graph, 3, 4, 9);
    e.addEdge(graph, 3, 5, 14);
    e.addEdge(graph, 4, 5, 10);
}

```

```
    e.addEdge(graph, 5, 6, 2);
    e.addEdge(graph, 6, 7, 1);
    e.addEdge(graph, 6, 8, 6);
    e.addEdge(graph, 7, 8, 7);

    // Method invoked
    e.prims_mst(graph);
}
}

// This code is contributed by Vikash Kumar Dubey
```

Python

```
# A Python program for Prims's MST for
# adjacency list representation of graph

from collections import defaultdict
import sys

class Heap():

    def __init__(self):
        self.array = []
        self.size = 0
        self.pos = []

    def newMinHeapNode(self, v, dist):
        minHeapNode = [v, dist]
        return minHeapNode

    # A utility function to swap two nodes of
    # min heap. Needed for min heapify
    def swapMinHeapNode(self, a, b):
        t = self.array[a]
        self.array[a] = self.array[b]
        self.array[b] = t

    # A standard function to heapify at given idx
    # This function also updates position of nodes
    # when they are swapped. Position is needed
    # for decreaseKey()
    def minHeapify(self, idx):
        smallest = idx
        left = 2 * idx + 1
        right = 2 * idx + 2

        if left < self.size and self.array[left][1] < \
           self.array[smallest][1]:
```

```

smallest = left

if right < self.size and self.array[right][1] < \
    self.array[smallest][1]:
    smallest = right

# The nodes to be swapped in min heap
# if idx is not smallest
if smallest != idx:

    # Swap positions
    self.pos[ self.array[smallest][0] ] = idx
    self.pos[ self.array[idx][0] ] = smallest

    # Swap nodes
    self.swapMinHeapNode(smallest, idx)

    self.minHeapify(smallest)

# Standard function to extract minimum node from heap
def extractMin(self):

    # Return NULL wif heap is empty
    if self.isEmpty() == True:
        return

    # Store the root node
    root = self.array[0]

    # Replace root node with last node
    lastNode = self.array[self.size - 1]
    self.array[0] = lastNode

    # Update position of last node
    self.pos[lastNode[0]] = 0
    self.pos[root[0]] = self.size - 1

    # Reduce heap size and heapify root
    self.size -= 1
    self.minHeapify(0)

    return root

def isEmpty(self):
    return True if self.size == 0 else False

def decreaseKey(self, v, dist):

```

```

# Get the index of v in heap array
i = self.pos[v]

# Get the node and update its dist value
self.array[i][1] = dist

# Travel up while the complete tree is not
# heepified. This is a O(Logn) loop
while i > 0 and self.array[i][1] < \
    self.array[(i - 1) / 2][1]:

    # Swap this node with its parent
    self.pos[ self.array[i][0] ] = (i-1)/2
    self.pos[ self.array[(i-1)/2][0] ] = i
    self.swapMinHeapNode(i, (i - 1)/2)

    # move to parent index
    i = (i - 1) / 2;

# A utility function to check if a given vertex
# 'v' is in min heap or not
def isInMinHeap(self, v):

    if self.pos[v] < self.size:
        return True
    return False

def printArr(parent, n):
    for i in range(1, n):
        print "% d - % d" % (parent[i], i)

class Graph():

    def __init__(self, V):
        self.V = V
        self.graph = defaultdict(list)

    # Adds an edge to an undirected graph
    def addEdge(self, src, dest, weight):

        # Add an edge from src to dest. A new node is
        # added to the adjacency list of src. The node
        # is added at the begining. The first element of
        # the node has the destination and the second
        # elements has the weight

```

```

newNode = [dest, weight]
self.graph[src].insert(0, newNode)

# Since graph is undirected, add an edge from
# dest to src also
newNode = [src, weight]
self.graph[dest].insert(0, newNode)

# The main function that prints the Minimum
# Spanning Tree(MST) using the Prim's Algorithm.
# It is a O(ELogV) function
def PrimMST(self):
    # Get the number of vertices in graph
    V = self.V

    # key values used to pick minimum weight edge in cut
    key = []

    # List to store constructed MST
    parent = []

    # minHeap represents set E
    minHeap = Heap()

    # Initialize min heap with all vertices. Key values of all
    # vertices (except the 0th vertex) is initially infinite
    for v in range(V):
        parent.append(-1)
        key.append(sys.maxint)
        minHeap.array.append( minHeap.newMinHeapNode(v, key[v]) )
        minHeap.pos.append(v)

    # Make key value of 0th vertex as 0 so
    # that it is extracted first
    minHeap.pos[0] = 0
    key[0] = 0
    minHeap.decreaseKey(0, key[0])

    # Initially size of min heap is equal to V
    minHeap.size = V;

    # In the following loop, min heap contains all nodes
    # not yet added in the MST.
    while minHeap.isEmpty() == False:

        # Extract the vertex with minimum distance value
        newHeapNode = minHeap.extractMin()
        u = newHeapNode[0]

```

```
# Traverse through all adjacent vertices of u
# (the extracted vertex) and update their
# distance values
for pCrawl in self.graph[u]:
    v = pCrawl[0]

    # If shortest distance to v is not finalized
    # yet, and distance to v through u is less than
    # its previously calculated distance
    if minHeap.isInMinHeap(v) and pCrawl[1] < key[v]:
        key[v] = pCrawl[1]
        parent[v] = u

        # update distance value in min heap also
        minHeap.decreaseKey(v, key[v])

printArr(parent, V)

# Driver program to test the above functions
graph = Graph(9)
graph.addEdge(0, 1, 4)
graph.addEdge(0, 7, 8)
graph.addEdge(1, 2, 8)
graph.addEdge(1, 7, 11)
graph.addEdge(2, 3, 7)
graph.addEdge(2, 8, 2)
graph.addEdge(2, 5, 4)
graph.addEdge(3, 4, 9)
graph.addEdge(3, 5, 14)
graph.addEdge(4, 5, 10)
graph.addEdge(5, 6, 2)
graph.addEdge(6, 7, 1)
graph.addEdge(6, 8, 6)
graph.addEdge(7, 8, 7)
graph.PrimMST()

# This code is contributed by Divyanshu Mehta
```

Output:

```
0 - 1
5 - 2
2 - 3
3 - 4
6 - 5
```

7 - 6
0 - 7
2 - 8

Time Complexity: The time complexity of the above code/algorithm looks $O(V^2)$ as there are two nested while loops. If we take a closer look, we can observe that the statements in inner loop are executed $O(V+E)$ times (similar to BFS). The inner loop has decreaseKey() operation which takes $O(\log V)$ time. So overall time complexity is $O(E+V)*O(\log V)$ which is $O((E+V)*\log V) = O(E\log V)$ (For a connected graph, $V = O(E)$)

References:

[Introduction to Algorithms](#) by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.

http://en.wikipedia.org/wiki/Prim's_algorithm

This article is compiled by [Aashish Barnwal](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Improved By : [VikashDubey](#), [Sumon Nath](#)

Source

<https://www.geeksforgeeks.org/prims-mst-for-adjacency-list-representation-greedy-algo-6/>

Chapter 20

Kruskal's Minimum Spanning Tree Algorithm

Kruskal's Minimum Spanning Tree Algorithm Greedy Algo-2 - GeeksforGeeks

What is Minimum Spanning Tree?

Given a connected and undirected graph, a *spanning tree* of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A *minimum spanning tree (MST)* or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

How many edges does a minimum spanning tree has?

A minimum spanning tree has $(V - 1)$ edges where V is the number of vertices in the given graph.

What are the applications of Minimum Spanning Tree?

See [this](#)for applications of MST.

Below are the steps for finding MST using Kruskal's algorithm

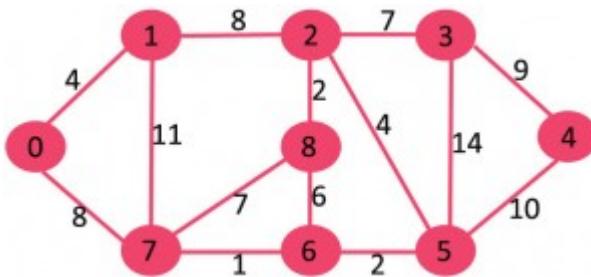
1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are $(V-1)$ edges in the spanning tree.

The step#2 uses [Union-Find algorithm](#) to detect cycle. So we recommend to read following post as a prerequisite.

[Union-Find Algorithm Set 1 \(Detect Cycle in a Graph\)](#)

[Union-Find Algorithm Set 2 \(Union By Rank and Path Compression\)](#)

The algorithm is a Greedy Algorithm. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far. Let us understand it with an example: Consider the below input graph.



The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having $(9 - 1) = 8$ edges.

After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

Now pick all edges one by one from sorted list of edges

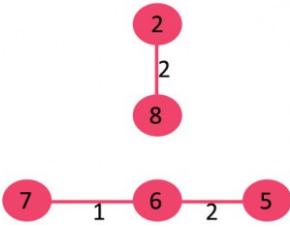
1. Pick edge 7-6: No cycle is formed, include it.



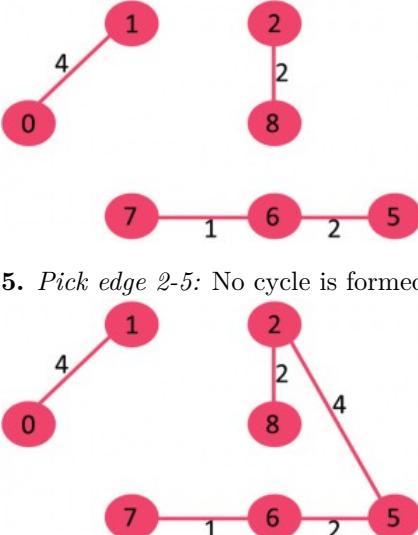
2. Pick edge 8-2: No cycle is formed, include it.



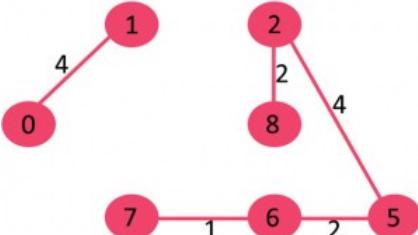
3. Pick edge 6-5: No cycle is formed, include it.



4. Pick edge 0-1: No cycle is formed, include it.

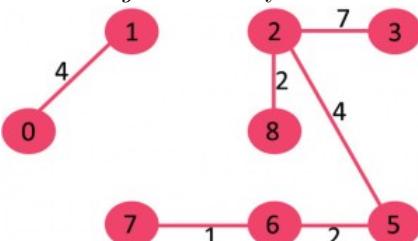


5. Pick edge 2-5: No cycle is formed, include it.



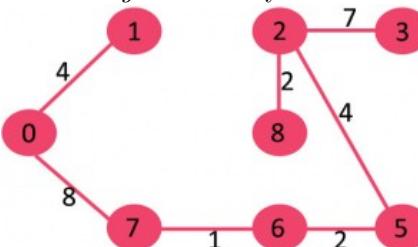
6. Pick edge 8-6: Since including this edge results in cycle, discard it.

7. Pick edge 2-3: No cycle is formed, include it.



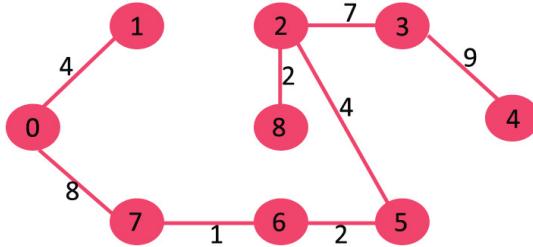
8. Pick edge 7-8: Since including this edge results in cycle, discard it.

9. Pick edge 0-7: No cycle is formed, include it.



10. Pick edge 1-2: Since including this edge results in cycle, discard it.

11. Pick edge 3-4: No cycle is formed, include it.



Since the number of edges included equals $(V - 1)$, the algorithm stops here.

C/C++

```

// C++ program for Kruskal's algorithm to find Minimum Spanning Tree
// of a given connected, undirected and weighted graph
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// a structure to represent a weighted edge in graph
struct Edge
{
    int src, dest, weight;
};

// a structure to represent a connected, undirected
// and weighted graph
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    // Since the graph is undirected, the edge
    // from src to dest is also edge from dest
    // to src. Both are counted as 1 edge here.
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
}

```

```

graph->edge = new Edge[E];

return graph;
}

// A structure to represent a subset for union-find
struct subset
{
    int parent;
    int rank;
};

// A utility function to find set of an element i
// (uses path compression technique)
int find(struct subset subsets[], int i)
{
    // find root and make root as parent of i
    // (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(struct subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high
    // rank tree (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and
    // increment its rank by one
    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Compare two edges according to their weights.

```

```

// Used in qsort() for sorting an array of edges
int myComp(const void* a, const void* b)
{
    struct Edge* a1 = (struct Edge*)a;
    struct Edge* b1 = (struct Edge*)b;
    return a1->weight > b1->weight;
}

// The main function to construct MST using Kruskal's algorithm
void KruskalMST(struct Graph* graph)
{
    int V = graph->V;
    struct Edge result[V]; // This will store the resultant MST
    int e = 0; // An index variable, used for result[]
    int i = 0; // An index variable, used for sorted edges

    // Step 1: Sort all the edges in non-decreasing
    // order of their weight. If we are not allowed to
    // change the given graph, we can create a copy of
    // array of edges
    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);

    // Allocate memory for creating V subsets
    struct subset *subsets =
        (struct subset*) malloc( V * sizeof(struct subset) );

    // Create V subsets with single elements
    for (int v = 0; v < V; ++v)
    {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Number of edges to be taken is equal to V-1
    while (e < V - 1)
    {
        // Step 2: Pick the smallest edge. And increment
        // the index for next iteration
        struct Edge next_edge = graph->edge[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        // If including this edge does't cause cycle,
        // include it in result and increment the index
        // of result for next edge
        if (x != y)
        {

```

```

        result[e++] = next_edge;
        Union(subsets, x, y);
    }
    // Else discard the next_edge
}

// print the contents of result[] to display the
// built MST
printf("Following are the edges in the constructed MST\n");
for (i = 0; i < e; ++i)
    printf("%d -- %d == %d\n", result[i].src, result[i].dest,
           result[i].weight);
return;
}

// Driver program to test above functions
int main()
{
/* Let us create following weighted graph
   10
   0-----1
   | \   |
   6|   5\  |15
   |     \ |
   2-----3
   4           */
    int V = 4; // Number of vertices in graph
    int E = 5; // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = 10;

    // add edge 0-2
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 6;

    // add edge 0-3
    graph->edge[2].src = 0;
    graph->edge[2].dest = 3;
    graph->edge[2].weight = 5;

    // add edge 1-3
    graph->edge[3].src = 1;
}

```

```
graph->edge[3].dest = 3;
graph->edge[3].weight = 15;

// add edge 2-3
graph->edge[4].src = 2;
graph->edge[4].dest = 3;
graph->edge[4].weight = 4;

KruskalMST(graph);

return 0;
}

Java

// Java program for Kruskal's algorithm to find Minimum
// Spanning Tree of a given connected, undirected and
// weighted graph
import java.util.*;
import java.lang.*;
import java.io.*;

class Graph
{
    // A class to represent a graph edge
    class Edge implements Comparable<Edge>
    {
        int src, dest, weight;

        // Comparator function used for sorting edges
        // based on their weight
        public int compareTo(Edge compareEdge)
        {
            return this.weight - compareEdge.weight;
        }
    };

    // A class to represent a subset for union-find
    class subset
    {
        int parent, rank;
    };

    int V, E;      // V-> no. of vertices & E->no.of edges
    Edge edge[]; // collection of all edges

    // Creates a graph with V vertices and E edges
    Graph(int v, int e)
```

```

{
    V = v;
    E = e;
    edge = new Edge[E];
    for (int i=0; i<e; ++i)
        edge[i] = new Edge();
}

// A utility function to find set of an element i
// (uses path compression technique)
int find(subset subsets[], int i)
{
    // find root and make root as parent of i (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high rank tree
    // (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and increment
    // its rank by one
    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// The main function to construct MST using Kruskal's algorithm
void KruskalMST()
{
    Edge result[] = new Edge[V]; // This will store the resultant MST
    int e = 0; // An index variable, used for result[]
    int i = 0; // An index variable, used for sorted edges
}

```

```

for (i=0; i<V; ++i)
    result[i] = new Edge();

// Step 1: Sort all the edges in non-decreasing order of their
// weight. If we are not allowed to change the given graph, we
// can create a copy of array of edges
Arrays.sort(edge);

// Allocate memory for creating V subsets
subset subsets[] = new subset[V];
for(i=0; i<V; ++i)
    subsets[i]=new subset();

// Create V subsets with single elements
for (int v = 0; v < V; ++v)
{
    subsets[v].parent = v;
    subsets[v].rank = 0;
}

i = 0; // Index used to pick next edge

// Number of edges to be taken is equal to V-1
while (e < V - 1)
{
    // Step 2: Pick the smallest edge. And increment
    // the index for next iteration
    Edge next_edge = new Edge();
    next_edge = edge[i++];

    int x = find(subsets, next_edge.src);
    int y = find(subsets, next_edge.dest);

    // If including this edge does't cause cycle,
    // include it in result and increment the index
    // of result for next edge
    if (x != y)
    {
        result[e++] = next_edge;
        Union(subsets, x, y);
    }
    // Else discard the next_edge
}

// print the contents of result[] to display
// the built MST
System.out.println("Following are the edges in " +
                    "the constructed MST");

```

```

        for (i = 0; i < e; ++i)
            System.out.println(result[i].src+" -- " +
                result[i].dest+" == " + result[i].weight);
    }

    // Driver Program
    public static void main (String[] args)
    {

        /* Let us create following weighted graph
           10
           0-----1
           |   \
           |   5\   |
           |       \ |
           2-----3
           4           */
        int V = 4; // Number of vertices in graph
        int E = 5; // Number of edges in graph
        Graph graph = new Graph(V, E);

        // add edge 0-1
        graph.edge[0].src = 0;
        graph.edge[0].dest = 1;
        graph.edge[0].weight = 10;

        // add edge 0-2
        graph.edge[1].src = 0;
        graph.edge[1].dest = 2;
        graph.edge[1].weight = 6;

        // add edge 0-3
        graph.edge[2].src = 0;
        graph.edge[2].dest = 3;
        graph.edge[2].weight = 5;

        // add edge 1-3
        graph.edge[3].src = 1;
        graph.edge[3].dest = 3;
        graph.edge[3].weight = 15;

        // add edge 2-3
        graph.edge[4].src = 2;
        graph.edge[4].dest = 3;
        graph.edge[4].weight = 4;

        graph.KruskalMST();
    }
}

```

```
}
```

//This code is contributed by Aakash Hasija

Python

```
# Python program for Kruskal's algorithm to find
# Minimum Spanning Tree of a given connected,
# undirected and weighted graph

from collections import defaultdict

#Class to represent a graph
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices
        self.graph = [] # default dictionary
                           # to store graph

        # function to add an edge to graph
    def addEdge(self,u,v,w):
        self.graph.append([u,v,w])

    # A utility function to find set of an element i
    # (uses path compression technique)
    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    # A function that does union of two sets of x and y
    # (uses union by rank)
    def union(self, parent, rank, x, y):
        xroot = self.find(parent, x)
        yroot = self.find(parent, y)

        # Attach smaller rank tree under root of
        # high rank tree (Union by Rank)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot

        # If ranks are same, then make one as root
        # and increment its rank by one
        else :
            parent[yroot] = xroot
```

```

rank[xroot] += 1

# The main function to construct MST using Kruskal's
# algorithm
def KruskalMST(self):

    result = [] #This will store the resultant MST

    i = 0 # An index variable, used for sorted edges
    e = 0 # An index variable, used for result[]

    # Step 1: Sort all the edges in non-decreasing
    # order of their
    # weight. If we are not allowed to change the
    # given graph, we can create a copy of graph
    self.graph = sorted(self.graph,key=lambda item: item[2])

    parent = [] ; rank = []

    # Create V subsets with single elements
    for node in range(self.V):
        parent.append(node)
        rank.append(0)

    # Number of edges to be taken is equal to V-1
    while e < self.V - 1 :

        # Step 2: Pick the smallest edge and increment
        # the index for next iteration
        u,v,w = self.graph[i]
        i = i + 1
        x = self.find(parent, u)
        y = self.find(parent ,v)

        # If including this edge does't cause cycle,
        # include it in result and increment the index
        # of result for next edge
        if x != y:
            e = e + 1
            result.append([u,v,w])
            self.union(parent, rank, x, y)
        # Else discard the edge

    # print the contents of result[] to display the built MST
    print "Following are the edges in the constructed MST"
    for u,v,weight in result:
        #print str(u) + " -- " + str(v) + " == " + str(weight)
        print ("%d -- %d == %d" % (u,v,weight))

```

```
# Driver code
g = Graph(4)
g.addEdge(0, 1, 10)
g.addEdge(0, 2, 6)
g.addEdge(0, 3, 5)
g.addEdge(1, 3, 15)
g.addEdge(2, 3, 4)

g.KruskalMST()

#This code is contributed by Neelam Yadav
```

Following are the edges in the constructed MST

```
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
```

Time Complexity: $O(E \log E)$ or $O(E \log V)$. Sorting of edges takes $O(E \log E)$ time. After sorting, we iterate through all edges and apply find-union algorithm. The find and union operations can take atmost $O(\log V)$ time. So overall complexity is $O(E \log E + E \log V)$ time. The value of E can be atmost $O(V^2)$, so $O(\log V)$ are $O(\log E)$ same. Therefore, overall time complexity is $O(E \log E)$ or $O(E \log V)$

References:

<http://www.ics.uci.edu/~eppstein/161/960206.html>
http://en.wikipedia.org/wiki/Minimum_spanning_tree

This article is compiled by [Aashish Barnwal](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

<https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/>

Chapter 21

Dijkstra's shortest path algorithm

Dijkstra's algorithm

Given a graph and a source vertex in the graph, find shortest paths from source to all vertices in the given graph.

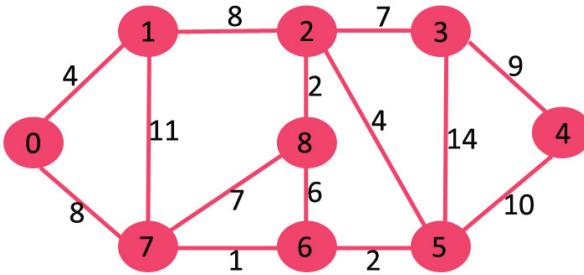
Dijkstra's algorithm is very similar to [Prim's algorithm for minimum spanning tree](#). Like Prim's MST, we generate a *SPT* (*shortest path tree*) with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

Below are the detailed steps used in Dijkstra's algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

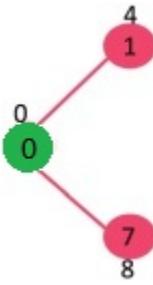
Algorithm

- 1) Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.
- 2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
- 3) While *sptSet* doesn't include all vertices
 -a) Pick a vertex *u* which is not there in *sptSet* and has minimum distance value.
 -b) Include *u* to *sptSet*.
 -c) Update distance value of all adjacent vertices of *u*. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex *v*, if sum of distance value of *u* (from source) and weight of edge *u-v*, is less than the distance value of *v*, then update the distance value of *v*.

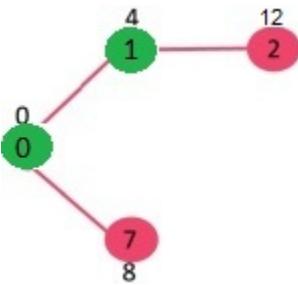
Let us understand with the following example:



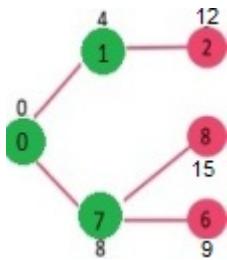
The set *sptSet* is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.



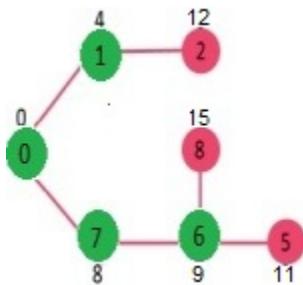
Pick the vertex with minimum distance value and not already included in SPT (not in *sptSET*). The vertex 1 is picked and added to *sptSet*. So *sptSet* now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



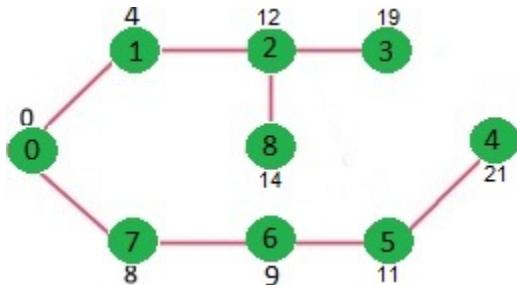
Pick the vertex with minimum distance value and not already included in SPT (not in *sptSET*). Vertex 7 is picked. So *sptSet* now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



Pick the vertex with minimum distance value and not already included in SPT (not in *sptSet*). Vertex 6 is picked. So *sptSet* now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



We repeat the above steps until *sptSet* doesn't include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).



How to implement the above algorithm?

We use a boolean array *sptSet[]* to represent the set of vertices included in SPT. If a value *sptSet[v]* is true, then vertex *v* is included in SPT, otherwise not. Array *dist[]* is used to store shortest distance values of all vertices.

C++

```
// A C++ program for Dijkstra's single source shortest path algorithm.
// The program is for adjacency matrix representation of the graph

#include <stdio.h>
#include <limits.h>

// Number of vertices in the graph
```

```

#define V 9

// A utility function to find the vertex with minimum distance value, from
// the set of vertices not yet included in shortest path tree
int minDistance(int dist[], bool sptSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min)
            min = dist[v], min_index = v;

    return min_index;
}

// A utility function to print the constructed distance array
int printSolution(int dist[], int n)
{
    printf("Vertex      Distance from Source\n");
    for (int i = 0; i < V; i++)
        printf("%d \t %d\n", i, dist[i]);
}

// Function that implements Dijkstra's single source shortest path algorithm
// for a graph represented using adjacency matrix representation
void dijkstra(int graph[V][V], int src)
{
    int dist[V];      // The output array. dist[i] will hold the shortest
                      // distance from src to i

    bool sptSet[V]; // sptSet[i] will true if vertex i is included in shortest
                    // path tree or shortest distance from src to i is finalized

    // Initialize all distances as INFINITE and stpSet[] as false
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;

    // Distance of source vertex from itself is always 0
    dist[src] = 0;

    // Find shortest path for all vertices
    for (int count = 0; count < V-1; count++)
    {
        // Pick the minimum distance vertex from the set of vertices not
        // yet processed. u is always equal to src in the first iteration.
        int u = minDistance(dist, sptSet);

```

```

// Mark the picked vertex as processed
sptSet[u] = true;

// Update dist value of the adjacent vertices of the picked vertex.
for (int v = 0; v < V; v++)

    // Update dist[v] only if is not in sptSet, there is an edge from
    // u to v, and total weight of path from src to v through u is
    // smaller than current value of dist[v]
    if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
        && dist[u]+graph[u][v] < dist[v])
        dist[v] = dist[u] + graph[u][v];
    }

    // print the constructed distance array
    printSolution(dist, V);
}

// driver program to test above function
int main()
{
    /* Let us create the example graph discussed above */
    int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},
                        {4, 0, 8, 0, 0, 0, 0, 11, 0},
                        {0, 8, 0, 7, 0, 4, 0, 0, 2},
                        {0, 0, 7, 0, 9, 14, 0, 0, 0},
                        {0, 0, 0, 9, 0, 10, 0, 0, 0},
                        {0, 0, 4, 14, 10, 0, 2, 0, 0},
                        {0, 0, 0, 0, 2, 0, 1, 6},
                        {8, 11, 0, 0, 0, 1, 0, 7},
                        {0, 0, 2, 0, 0, 0, 6, 7, 0}
                    };

    dijkstra(graph, 0);

    return 0;
}

```

Java

```

// A Java program for Dijkstra's single source shortest path algorithm.
// The program is for adjacency matrix representation of the graph
import java.util.*;
import java.lang.*;
import java.io.*;

class ShortestPath
{

```

```

// A utility function to find the vertex with minimum distance value,
// from the set of vertices not yet included in shortest path tree
static final int V=9;
int minDistance(int dist[], Boolean sptSet[])
{
    // Initialize min value
    int min = Integer.MAX_VALUE, min_index=-1;

    for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min)
        {
            min = dist[v];
            min_index = v;
        }

    return min_index;
}

// A utility function to print the constructed distance array
void printSolution(int dist[], int n)
{
    System.out.println("Vertex   Distance from Source");
    for (int i = 0; i < V; i++)
        System.out.println(i+" tt "+dist[i]);
}

// Function that implements Dijkstra's single source shortest path
// algorithm for a graph represented using adjacency matrix
// representation
void dijkstra(int graph[][] , int src)
{
    int dist[] = new int[V]; // The output array. dist[i] will hold
                           // the shortest distance from src to i

    // sptSet[i] will true if vertex i is included in shortest
    // path tree or shortest distance from src to i is finalized
    Boolean sptSet[] = new Boolean[V];

    // Initialize all distances as INFINITE and stpSet[] as false
    for (int i = 0; i < V; i++)
    {
        dist[i] = Integer.MAX_VALUE;
        sptSet[i] = false;
    }

    // Distance of source vertex from itself is always 0
    dist[src] = 0;
}

```

```

// Find shortest path for all vertices
for (int count = 0; count < V-1; count++)
{
    // Pick the minimum distance vertex from the set of vertices
    // not yet processed. u is always equal to src in first
    // iteration.
    int u = minDistance(dist, sptSet);

    // Mark the picked vertex as processed
    sptSet[u] = true;

    // Update dist value of the adjacent vertices of the
    // picked vertex.
    for (int v = 0; v < V; v++)

        // Update dist[v] only if is not in sptSet, there is an
        // edge from u to v, and total weight of path from src to
        // v through u is smaller than current value of dist[v]
        if (!sptSet[v] && graph[u][v]!=0 &&
            dist[u] != Integer.MAX_VALUE &&
            dist[u]+graph[u][v] < dist[v])
            dist[v] = dist[u] + graph[u][v];
    }

    // print the constructed distance array
    printSolution(dist, V);
}

// Driver method
public static void main (String[] args)
{
    /* Let us create the example graph discussed above */
    int graph[][] = new int[][]{{0, 4, 0, 0, 0, 0, 0, 8, 0},
                                {4, 0, 8, 0, 0, 0, 0, 11, 0},
                                {0, 8, 0, 7, 0, 4, 0, 0, 2},
                                {0, 0, 7, 0, 9, 14, 0, 0, 0},
                                {0, 0, 0, 9, 0, 10, 0, 0, 0},
                                {0, 0, 4, 14, 10, 0, 2, 0, 0},
                                {0, 0, 0, 0, 2, 0, 1, 6},
                                {8, 11, 0, 0, 0, 0, 1, 0, 7},
                                {0, 0, 2, 0, 0, 0, 6, 7, 0}
                            };
    ShortestPath t = new ShortestPath();
    t.dijkstra(graph, 0);
}
//This code is contributed by Aakash Hasija

```

Python

```
# Python program for Dijkstra's single
# source shortest path algorithm. The program is
# for adjacency matrix representation of the graph

# Library for INT_MAX
import sys

class Graph():

    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]

    def printSolution(self, dist):
        print "Vertex tDistance from Source"
        for node in range(self.V):
            print node,"t",dist[node]

    # A utility function to find the vertex with
    # minimum distance value, from the set of vertices
    # not yet included in shortest path tree
    def minDistance(self, dist, sptSet):

        # Initilaize minimum distance for next node
        min = sys.maxint

        # Search not nearest vertex not in the
        # shortest path tree
        for v in range(self.V):
            if dist[v] < min and sptSet[v] == False:
                min = dist[v]
                min_index = v

        return min_index

    # Function that implements Dijkstra's single source
    # shortest path algorithm for a graph represented
    # using adjacency matrix representation
    def dijkstra(self, src):

        dist = [sys.maxint] * self.V
        dist[src] = 0
        sptSet = [False] * self.V

        for cout in range(self.V):
```

```

# Pick the minimum distance vertex from
# the set of vertices not yet processed.
# u is always equal to src in first iteration
u = self.minDistance(dist, sptSet)

# Put the minimum distance vertex in the
# shortest path tree
sptSet[u] = True

# Update dist value of the adjacent vertices
# of the picked vertex only if the current
# distance is greater than new distance and
# the vertex is not in the shortest path tree
for v in range(self.V):
    if self.graph[u][v] > 0 and sptSet[v] == False and
       dist[v] > dist[u] + self.graph[u][v]:
        dist[v] = dist[u] + self.graph[u][v]

self.printSolution(dist)

# Driver program
g = Graph(9)
g.graph = [[0, 4, 0, 0, 0, 0, 0, 8, 0],
            [4, 0, 8, 0, 0, 0, 0, 11, 0],
            [0, 8, 0, 7, 0, 4, 0, 0, 2],
            [0, 0, 7, 0, 9, 14, 0, 0, 0],
            [0, 0, 0, 9, 0, 10, 0, 0, 0],
            [0, 0, 4, 14, 10, 0, 2, 0, 0],
            [0, 0, 0, 0, 2, 0, 1, 6],
            [8, 11, 0, 0, 0, 0, 1, 0, 7],
            [0, 0, 2, 0, 0, 0, 6, 7, 0]
];
g.dijkstra(0);

# This code is contributed by Divyanshu Mehta

```

Output:

Vertex	Distance from Source
0	0
1	4
2	12
3	19
4	21
5	11
6	9

7	8
8	14

Notes:

1) The code calculates shortest distance, but doesn't calculate the path information. We can create a parent array, update the parent array when distance is updated (like [prim's implementation](#)) and use it show the shortest path from source to different vertices.

2) The code is for undirected graph, same dijkstra function can be used for directed graphs also.

3) The code finds shortest distances from source to all vertices. If we are interested only in shortest distance from the source to a single target, we can break the for the loop when the picked minimum distance vertex is equal to target (Step 3.a of the algorithm).

4) Time Complexity of the implementation is $O(V^2)$. If the input [graph is represented using adjacency list](#), it can be reduced to $O(E \log V)$ with the help of binary heap. Please see

[Dijkstra's Algorithm for Adjacency List Representation](#) for more details.

5) Dijkstra's algorithm doesn't work for graphs with negative weight edges. For graphs with negative weight edges, [Bellman–Ford algorithm](#) can be used, we will soon be discussing it as a separate post.

[Dijkstra's Algorithm for Adjacency List Representation](#)

[Printing Paths in Dijkstra's Shortest Path Algorithm](#)

[Dijkstra's shortest path algorithm using set in STL](#)

Source

<https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/>

Chapter 22

Dijkstra's Algorithm for Adjacency List Representation

Dijkstra's Algorithm for Adjacency List Representation Greedy Algo-8 - GeeksforGeeks

We recommend to read following two posts as a prerequisite of this post.

1. [Greedy Algorithms Set 7 \(Dijkstra's shortest path algorithm\)](#)
2. [Graph and its representations](#)

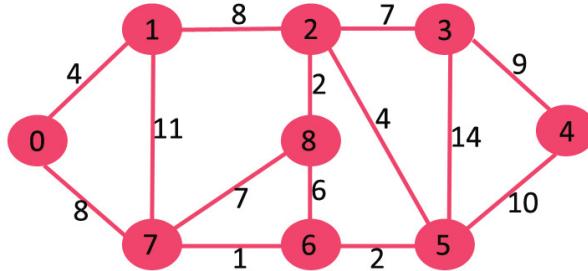
We have discussed [Dijkstra's algorithm and its implementation for adjacency matrix representation of graphs](#). The time complexity for the matrix representation is $O(V^2)$. In this post, $O(E\log V)$ algorithm for adjacency list representation is discussed.

As discussed in the previous post, in Dijkstra's algorithm, two sets are maintained, one set contains list of vertices already included in SPT (Shortest Path Tree), other set contains vertices not yet included. With adjacency list representation, all vertices of a graph can be traversed in $O(V+E)$ time using [BFS](#). The idea is to traverse all vertices of graph using [BFS](#) and use a Min Heap to store the vertices not yet included in SPT (or the vertices for which shortest distance is not finalized yet). Min Heap is used as a priority queue to get the minimum distance vertex from set of not yet included vertices. Time complexity of operations like extract-min and decrease-key value is $O(\log V)$ for Min Heap.

Following are the detailed steps.

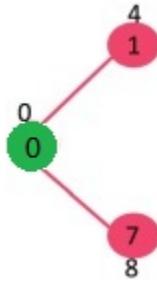
- 1) Create a Min Heap of size V where V is the number of vertices in the given graph. Every node of min heap contains vertex number and distance value of the vertex.
- 2) Initialize Min Heap with source vertex as root (the distance value assigned to source vertex is 0). The distance value assigned to all other vertices is INF (infinite).
- 3) While Min Heap is not empty, do following
 -a) Extract the vertex with minimum distance value node from Min Heap. Let the extracted vertex be u .
 -b) For every adjacent vertex v of u , check if v is in Min Heap. If v is in Min Heap and distance value is more than weight of $u-v$ plus distance value of u , then update the distance value of v .

Let us understand with the following example. Let the given source vertex be 0

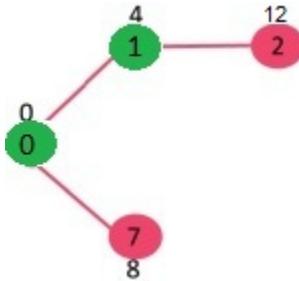


Initially, distance value of source vertex is 0 and INF (infinite) for all other vertices. So source vertex is extracted from Min Heap and distance values of vertices adjacent to 0 (1 and 7) are updated. Min Heap contains all vertices except vertex 0.

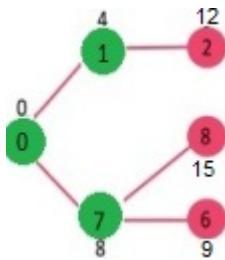
The vertices in green color are the vertices for which minimum distances are finalized and are not in Min Heap



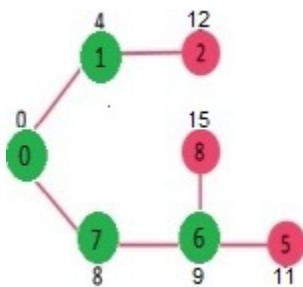
Since distance value of vertex 1 is minimum among all nodes in Min Heap, it is extracted from Min Heap and distance values of vertices adjacent to 1 are updated (distance is updated if the a vertex is not in Min Heap and distance through 1 is shorter than the previous distance). Min Heap contains all vertices except vertex 0 and 1.



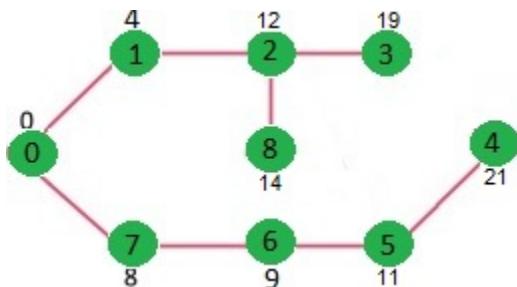
Pick the vertex with minimum distance value from min heap. Vertex 7 is picked. So min heap now contains all vertices except 0, 1 and 7. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



Pick the vertex with minimum distance from min heap. Vertex 6 is picked. So min heap now contains all vertices except 0, 1, 7 and 6. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



Above steps are repeated till min heap doesn't become empty. Finally, we get the following shortest path tree.



C++

```
// C / C++ program for Dijkstra's shortest path algorithm for adjacency
// list representation of graph

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a node in adjacency list
struct AdjListNode
{
    int dest;
    int weight;
    struct AdjListNode* next;
};

// A structure to represent a node in adjacency list
struct AdjList
{
    struct AdjListNode* head;
};
```

```
};

// A structure to represent an adjacency liat
struct AdjList
{
    struct AdjListNode *head; // pointer to head node of list
};

// A structure to represent a graph. A graph is an array of adjacency lists.
// Size of array will be V (number of vertices in graph)
struct Graph
{
    int V;
    struct AdjList* array;
};

// A utility function to create a new adjacency list node
struct AdjListNode* newAdjListNode(int dest, int weight)
{
    struct AdjListNode* newNode =
        (struct AdjListNode*) malloc(sizeof(struct AdjListNode));
    newNode->dest = dest;
    newNode->weight = weight;
    newNode->next = NULL;
    return newNode;
}

// A utility function that creates a graph of V vertices
struct Graph* createGraph(int V)
{
    struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
    graph->V = V;

    // Create an array of adjacency lists. Size of array will be V
    graph->array = (struct AdjList*) malloc(V * sizeof(struct AdjList));

    // Initialize each adjacency list as empty by making head as NULL
    for (int i = 0; i < V; ++i)
        graph->array[i].head = NULL;

    return graph;
}

// Adds an edge to an undirected graph
void addEdge(struct Graph* graph, int src, int dest, int weight)
{
    // Add an edge from src to dest. A new node is added to the adjacency
    // list of src. The node is added at the begining
```

```

struct AdjListNode* newNode = newAdjListNode(dest, weight);
newNode->next = graph->array[src].head;
graph->array[src].head = newNode;

// Since graph is undirected, add an edge from dest to src also
newNode = newAdjListNode(src, weight);
newNode->next = graph->array[dest].head;
graph->array[dest].head = newNode;
}

// Structure to represent a min heap node
struct MinHeapNode
{
    int v;
    int dist;
};

// Structure to represent a min heap
struct MinHeap
{
    int size;      // Number of heap nodes present currently
    int capacity; // Capacity of min heap
    int *pos;      // This is needed for decreaseKey()
    struct MinHeapNode **array;
};

// A utility function to create a new Min Heap Node
struct MinHeapNode* newMinHeapNode(int v, int dist)
{
    struct MinHeapNode* minHeapNode =
        (struct MinHeapNode*) malloc(sizeof(struct MinHeapNode));
    minHeapNode->v = v;
    minHeapNode->dist = dist;
    return minHeapNode;
}

// A utility function to create a Min Heap
struct MinHeap* createMinHeap(int capacity)
{
    struct MinHeap* minHeap =
        (struct MinHeap*) malloc(sizeof(struct MinHeap));
    minHeap->pos = (int *)malloc(capacity * sizeof(int));
    minHeap->size = 0;
    minHeap->capacity = capacity;
    minHeap->array =
        (struct MinHeapNode**) malloc(capacity * sizeof(struct MinHeapNode*));
    return minHeap;
}

```

```

// A utility function to swap two nodes of min heap. Needed for min heapify
void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b)
{
    struct MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}

// A standard function to heapify at given idx
// This function also updates position of nodes when they are swapped.
// Position is needed for decreaseKey()
void minHeapify(struct MinHeap* minHeap, int idx)
{
    int smallest, left, right;
    smallest = idx;
    left = 2 * idx + 1;
    right = 2 * idx + 2;

    if (left < minHeap->size &&
        minHeap->array[left]->dist < minHeap->array[smallest]->dist )
        smallest = left;

    if (right < minHeap->size &&
        minHeap->array[right]->dist < minHeap->array[smallest]->dist )
        smallest = right;

    if (smallest != idx)
    {
        // The nodes to be swapped in min heap
        MinHeapNode *smallestNode = minHeap->array[smallest];
        MinHeapNode *idxNode = minHeap->array[idx];

        // Swap positions
        minHeap->pos[smallestNode->v] = idx;
        minHeap->pos[idxNode->v] = smallest;

        // Swap nodes
        swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

        minHeapify(minHeap, smallest);
    }
}

// A utility function to check if the given minHeap is ampty or not
int isEmpty(struct MinHeap* minHeap)
{
    return minHeap->size == 0;
}

```

```

}

// Standard function to extract minimum node from heap
struct MinHeapNode* extractMin(struct MinHeap* minHeap)
{
    if (isEmpty(minHeap))
        return NULL;

    // Store the root node
    struct MinHeapNode* root = minHeap->array[0];

    // Replace root node with last node
    struct MinHeapNode* lastNode = minHeap->array[minHeap->size - 1];
    minHeap->array[0] = lastNode;

    // Update position of last node
    minHeap->pos[root->v] = minHeap->size-1;
    minHeap->pos[lastNode->v] = 0;

    // Reduce heap size and heapify root
    --minHeap->size;
    minHeapify(minHeap, 0);

    return root;
}

// Function to decrease dist value of a given vertex v. This function
// uses pos[] of min heap to get the current index of node in min heap
void decreaseKey(struct MinHeap* minHeap, int v, int dist)
{
    // Get the index of v in heap array
    int i = minHeap->pos[v];

    // Get the node and update its dist value
    minHeap->array[i]->dist = dist;

    // Travel up while the complete tree is not heapified.
    // This is a O(Logn) loop
    while (i && minHeap->array[i]->dist < minHeap->array[(i - 1) / 2]->dist)
    {
        // Swap this node with its parent
        minHeap->pos[minHeap->array[i]->v] = (i-1)/2;
        minHeap->pos[minHeap->array[(i-1)/2]->v] = i;
        swapMinHeapNode(&minHeap->array[i], &minHeap->array[(i - 1) / 2]);

        // move to parent index
        i = (i - 1) / 2;
    }
}

```

```

}

// A utility function to check if a given vertex
// 'v' is in min heap or not
bool isInMinHeap(struct MinHeap *minHeap, int v)
{
    if (minHeap->pos[v] < minHeap->size)
        return true;
    return false;
}

// A utility function used to print the solution
void printArr(int dist[], int n)
{
    printf("Vertex  Distance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

// The main function that calculates distances of shortest paths from src to all
// vertices. It is a O(ELogV) function
void dijkstra(struct Graph* graph, int src)
{
    int V = graph->V;// Get the number of vertices in graph
    int dist[V];      // dist values used to pick minimum weight edge in cut

    // minHeap represents set E
    struct MinHeap* minHeap = createMinHeap(V);

    // Initialize min heap with all vertices. dist value of all vertices
    for (int v = 0; v < V; ++v)
    {
        dist[v] = INT_MAX;
        minHeap->array[v] = newMinHeapNode(v, dist[v]);
        minHeap->pos[v] = v;
    }

    // Make dist value of src vertex as 0 so that it is extracted first
    minHeap->array[src] = newMinHeapNode(src, dist[src]);
    minHeap->pos[src] = src;
    dist[src] = 0;
    decreaseKey(minHeap, src, dist[src]);

    // Initially size of min heap is equal to V
    minHeap->size = V;

    // In the followin loop, min heap contains all nodes
    // whose shortest distance is not yet finalized.
}

```

```

while (!isEmpty(minHeap))
{
    // Extract the vertex with minimum distance value
    struct MinHeapNode* minHeapNode = extractMin(minHeap);
    int u = minHeapNode->v; // Store the extracted vertex number

    // Traverse through all adjacent vertices of u (the extracted
    // vertex) and update their distance values
    struct AdjListNode* pCrawl = graph->array[u].head;
    while (pCrawl != NULL)
    {
        int v = pCrawl->dest;

        // If shortest distance to v is not finalized yet, and distance to v
        // through u is less than its previously calculated distance
        if (isInMinHeap(minHeap, v) && dist[u] != INT_MAX &&
            pCrawl->weight + dist[u] < dist[v])
        {
            dist[v] = dist[u] + pCrawl->weight;

            // update distance value in min heap also
            decreaseKey(minHeap, v, dist[v]);
        }
        pCrawl = pCrawl->next;
    }
}

// print the calculated shortest distances
printArr(dist, V);
}

// Driver program to test above functions
int main()
{
    // create the graph given in above figure
    int V = 9;
    struct Graph* graph = createGraph(V);
    addEdge(graph, 0, 1, 4);
    addEdge(graph, 0, 7, 8);
    addEdge(graph, 1, 2, 8);
    addEdge(graph, 1, 7, 11);
    addEdge(graph, 2, 3, 7);
    addEdge(graph, 2, 8, 2);
    addEdge(graph, 2, 5, 4);
    addEdge(graph, 3, 4, 9);
    addEdge(graph, 3, 5, 14);
    addEdge(graph, 4, 5, 10);
}

```

```
    addEdge(graph, 5, 6, 2);
    addEdge(graph, 6, 7, 1);
    addEdge(graph, 6, 8, 6);
    addEdge(graph, 7, 8, 7);

    dijkstra(graph, 0);

    return 0;
}
```

Python

```
# A Python program for Dijkstra's shortest
# path algorithm for adjacency
# list representation of graph

from collections import defaultdict
import sys

class Heap():

    def __init__(self):
        self.array = []
        self.size = 0
        self.pos = []

    def newMinHeapNode(self, v, dist):
        minHeapNode = [v, dist]
        return minHeapNode

    # A utility function to swap two nodes
    # of min heap. Needed for min heapify
    def swapMinHeapNode(self, a, b):
        t = self.array[a]
        self.array[a] = self.array[b]
        self.array[b] = t

    # A standard function to heapify at given idx
    # This function also updates position of nodes
    # when they are swapped. Position is needed
    # for decreaseKey()
    def minHeapify(self, idx):
        smallest = idx
        left = 2*idx + 1
        right = 2*idx + 2

        if left < self.size and self.array[left][1] \
           < self.array[smallest][1]:
```

```

smallest = left

if right < self.size and self.array[right][1] \
    < self.array[smallest][1]:
    smallest = right

# The nodes to be swapped in min
# heap if idx is not smallest
if smallest != idx:

    # Swap positions
    self.pos[ self.array[smallest][0] ] = idx
    self.pos[ self.array[idx][0] ] = smallest

    # Swap nodes
    self.swapMinHeapNode(smallest, idx)

    self.minHeapify(smallest)

# Standard function to extract minimum
# node from heap
def extractMin(self):

    # Return NULL wif heap is empty
    if self.isEmpty() == True:
        return

    # Store the root node
    root = self.array[0]

    # Replace root node with last node
    lastNode = self.array[self.size - 1]
    self.array[0] = lastNode

    # Update position of last node
    self.pos[lastNode[0]] = 0
    self.pos[root[0]] = self.size - 1

    # Reduce heap size and heapify root
    self.size -= 1
    self.minHeapify(0)

    return root

def isEmpty(self):
    return True if self.size == 0 else False

def decreaseKey(self, v, dist):

```

```

# Get the index of v in heap array
i = self.pos[v]

# Get the node and update its dist value
self.array[i][1] = dist

# Travel up while the complete tree is
# not heapified. This is a O(Logn) loop
while i > 0 and self.array[i][1] < self.array[(i - 1) / 2][1]:

    # Swap this node with its parent
    self.pos[ self.array[i][0] ] = (i-1)/2
    self.pos[ self.array[(i-1)/2][0] ] = i
    self.swapMinHeapNode(i, (i - 1)/2 )

    # move to parent index
    i = (i - 1) / 2;

# A utility function to check if a given
# vertex 'v' is in min heap or not
def isInMinHeap(self, v):

    if self.pos[v] < self.size:
        return True
    return False

def printArr(dist, n):
    print "Vertex\tDistance from source"
    for i in range(n):
        print "%d\t\t%d" % (i,dist[i])

class Graph():

    def __init__(self, V):
        self.V = V
        self.graph = defaultdict(list)

    # Adds an edge to an undirected graph
    def addEdge(self, src, dest, weight):

        # Add an edge from src to dest. A new node
        # is added to the adjacency list of src. The
        # node is added at the begining. The first
        # element of the node has the destination

```

```

# and the second elements has the weight
newNode = [dest, weight]
self.graph[src].insert(0, newNode)

# Since graph is undirected, add an edge
# from dest to src also
newNode = [src, weight]
self.graph[dest].insert(0, newNode)

# The main function that calculates distances
# of shortest paths from src to all vertices.
# It is a O(ELogV) function
def dijkstra(self, src):

    V = self.V # Get the number of vertices in graph
    dist = [] # dist values used to pick minimum
              # weight edge in cut

    # minHeap represents set E
    minHeap = Heap()

    # Initialize min heap with all vertices.
    # dist value of all vertices
    for v in range(V):
        dist.append(sys.maxint)
        minHeap.array.append( minHeap.newMinHeapNode(v, dist[v]) )
        minHeap.pos.append(v)

    # Make dist value of src vertex as 0 so
    # that it is extracted first
    minHeap.pos[src] = src
    dist[src] = 0
    minHeap.decreaseKey(src, dist[src])

    # Initially size of min heap is equal to V
    minHeap.size = V;

    # In the following loop, min heap contains all nodes
    # whose shortest distance is not yet finalized.
    while minHeap.isEmpty() == False:

        # Extract the vertex with minimum distance value
        newHeapNode = minHeap.extractMin()
        u = newHeapNode[0]

        # Traverse through all adjacent vertices of
        # u (the extracted vertex) and update their
        # distance values

```

```

for pCrawl in self.graph[u]:
    v = pCrawl[0]

    # If shortest distance to v is not finalized
    # yet, and distance to v through u is less
    # than its previously calculated distance
    if minHeap.isInMinHeap(v) and dist[u] != sys.maxint and \
       pCrawl[1] + dist[u] < dist[v]:
        dist[v] = pCrawl[1] + dist[u]

        # update distance value
        # in min heap also
        minHeap.decreaseKey(v, dist[v])

printArr(dist,V)

# Driver program to test the above functions
graph = Graph(9)
graph.addEdge(0, 1, 4)
graph.addEdge(0, 7, 8)
graph.addEdge(1, 2, 8)
graph.addEdge(1, 7, 11)
graph.addEdge(2, 3, 7)
graph.addEdge(2, 8, 2)
graph.addEdge(2, 5, 4)
graph.addEdge(3, 4, 9)
graph.addEdge(3, 5, 14)
graph.addEdge(4, 5, 10)
graph.addEdge(5, 6, 2)
graph.addEdge(6, 7, 1)
graph.addEdge(6, 8, 6)
graph.addEdge(7, 8, 7)
graph.dijkstra(0)

# This code is contributed by Divyanshu Mehta

```

Output:

Vertex	Distance from Source
0	0
1	4
2	12
3	19
4	21
5	11
6	9

7	8
8	14

Time Complexity: The time complexity of the above code/algorithm looks $O(V^2)$ as there are two nested while loops. If we take a closer look, we can observe that the statements in inner loop are executed $O(V+E)$ times (similar to BFS). The inner loop has decreaseKey() operation which takes $O(\log V)$ time. So overall time complexity is $O(E+V)*O(\log V)$ which is $O((E+V)*\log V) = O(E\log V)$

Note that the above code uses Binary Heap for Priority Queue implementation. Time complexity can be reduced to $O(E + V\log V)$ using Fibonacci Heap. The reason is, Fibonacci Heap takes $O(1)$ time for decrease-key operation while Binary Heap takes $O(\log n)$ time.

Notes:

- 1) The code calculates shortest distance, but doesn't calculate the path information. We can create a parent array, update the parent array when distance is updated (like [prim's implementation](#)) and use it show the shortest path from source to different vertices.
- 2) The code is for undirected graph, same dijkstra function can be used for directed graphs also.
- 3) The code finds shortest distances from source to all vertices. If we are interested only in shortest distance from source to a single target, we can break the for loop when the picked minimum distance vertex is equal to target (Step 3.a of algorithm).
- 4) Dijkstra's algorithm doesn't work for graphs with negative weight edges. For graphs with negative weight edges, [Bellman–Ford algorithm](#) can be used, we will soon be discussing it as a separate post.

[Printing Paths in Dijkstra's Shortest Path Algorithm](#)

[Dijkstra's shortest path algorithm using set in STL](#)

References:

[Introduction to Algorithms](#) by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.

[Algorithms](#) by Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani

Source

<https://www.geeksforgeeks.org/dijkstras-algorithm-for-adjacency-list-representation-greedy-algo-8/>

Chapter 23

Greedy Algorithms Set 9 (Boruvka's algorithm)

Boruvka's algorithm Greedy Algo-9 - GeeksforGeeks

We have discussed following topics on Minimum Spanning Tree.

[Applications of Minimum Spanning Tree Problem](#)
[Kruskal's Minimum Spanning Tree Algorithm](#)
[Prim's Minimum Spanning Tree Algorithm](#)

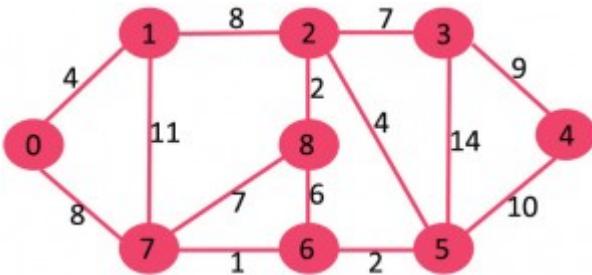
In this post, Boruvka's algorithm is discussed. Like Prim's and Kruskal's, Boruvka's algorithm is also a Greedy algorithm. Below is complete algorithm.

- 1) Input is a connected, weighted and directed graph.
- 2) Initialize all vertices as individual components (or sets).
- 3) Initialize MST as empty.
- 4) While there are more than one components, do following for each component.
 - a) Find the closest weight edge that connects this component to any other component.
 - b) Add this closest edge to MST if not already added.
- 5) Return MST.

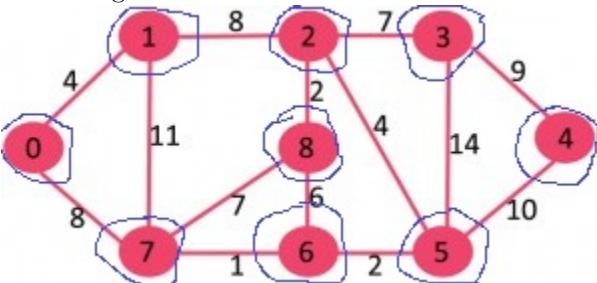
Below is the idea behind above algorithm (The idea is same as [Prim's MST algorithm](#)).

A spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a Spanning Tree. And they must be connected with the minimum weight edge to make it a Minimum Spanning Tree.

Let us understand the algorithm with below example.



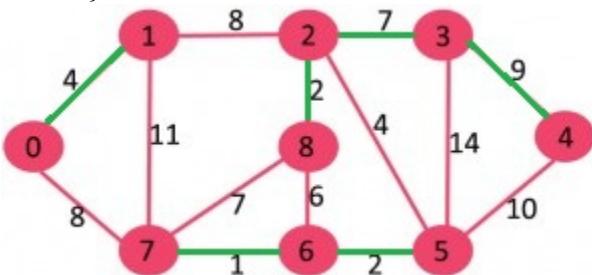
Initially MST is empty. Every vertex is single component as highlighted in blue color in below diagram.



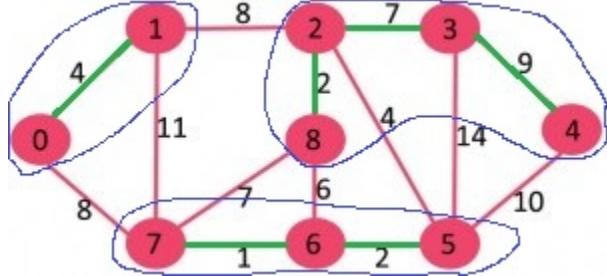
For every component, find the cheapest edge that connects it to some other component.

Component	Cheapest Edge that connects it to some other component
{0}	0-1
{1}	0-1
{2}	2-8
{3}	2-3
{4}	3-4
{5}	5-6
{6}	6-7
{7}	6-7
{8}	2-8

The cheapest edges are highlighted with green color. Now MST becomes {0-1, 2-8, 2-3, 3-4, 5-6, 6-7}.



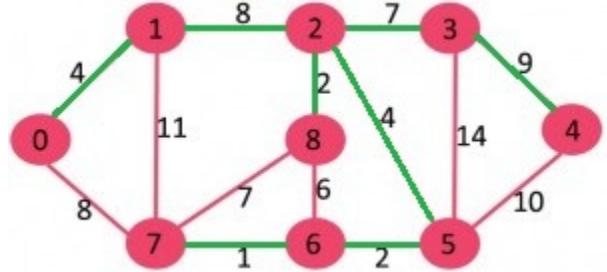
After above step, components are $\{\{0,1\}, \{2,3,4,8\}, \{5,6,7\}\}$. The components are encircled with blue color.



We again repeat the step, i.e., for every component, find the cheapest edge that connects it to some other component.

Component	Cheapest Edge that connects it to some other component
$\{0,1\}$	1-2 (or 0-7)
$\{2,3,4,8\}$	2-5
$\{5,6,7\}$	2-5

The cheapest edges are highlighted with green color. Now MST becomes $\{0-1, 2-8, 2-3, 3-4, 5-6, 6-7, 1-2, 2-5\}$



At this stage, there is only one component $\{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ which has all edges. Since there is only one component left, we stop and return MST.

Implementation:

Below is implementation of above algorithm. The input graph is represented as a collection of edges and [union-find data structure](#) is used to keep track of components.

C/C++

```
// Boruvka's algorithm to find Minimum Spanning
// Tree of a given connected, undirected and
// weighted graph
#include <stdio.h>

// a structure to represent a weighted edge in graph
```

```

struct Edge
{
    int src, dest, weight;
};

// a structure to represent a connected, undirected
// and weighted graph as a collection of edges.
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    // Since the graph is undirected, the edge
    // from src to dest is also edge from dest
    // to src. Both are counted as 1 edge here.
    Edge* edge;
};

// A structure to represent a subset for union-find
struct subset
{
    int parent;
    int rank;
};

// Function prototypes for union-find (These functions are defined
// after boruvkaMST() )
int find(struct subset subsets[], int i);
void Union(struct subset subsets[], int x, int y);

// The main function for MST using Boruvka's algorithm
void boruvkaMST(struct Graph* graph)
{
    // Get data of given graph
    int V = graph->V, E = graph->E;
    Edge *edge = graph->edge;

    // Allocate memory for creating V subsets.
    struct subset *subsets = new subset[V];

    // An array to store index of the cheapest edge of
    // subset. The stored index for indexing array 'edge[]'
    int *cheapest = new int[V];

    // Create V subsets with single elements
    for (int v = 0; v < V; ++v)
    {

```

```

        subsets[v].parent = v;
        subsets[v].rank = 0;
        cheapest[v] = -1;
    }

    // Initially there are V different trees.
    // Finally there will be one tree that will be MST
    int numTrees = V;
    int MSTweight = 0;

    // Keep combining components (or sets) until all
    // componentes are not combined into single MST.
    while (numTrees > 1)
    {
        // Everytime initialize cheapest array
        for (int v = 0; v < V; ++v)
        {
            cheapest[v] = -1;
        }

        // Traverse through all edges and update
        // cheapest of every component
        for (int i=0; i<E; i++)
        {
            // Find components (or sets) of two corners
            // of current edge
            int set1 = find(subsets, edge[i].src);
            int set2 = find(subsets, edge[i].dest);

            // If two corners of current edge belong to
            // same set, ignore current edge
            if (set1 == set2)
                continue;

            // Else check if current edge is closer to previous
            // cheapest edges of set1 and set2
            else
            {
                if (cheapest[set1] == -1 ||
                    edge[cheapest[set1]].weight > edge[i].weight)
                    cheapest[set1] = i;

                if (cheapest[set2] == -1 ||
                    edge[cheapest[set2]].weight > edge[i].weight)
                    cheapest[set2] = i;
            }
        }
    }
}

```

```

// Consider the above picked cheapest edges and add them
// to MST
for (int i=0; i<V; i++)
{
    // Check if cheapest for current set exists
    if (cheapest[i] != -1)
    {
        int set1 = find(subsets, edge[cheapest[i]].src);
        int set2 = find(subsets, edge[cheapest[i]].dest);

        if (set1 == set2)
            continue;
        MSTweight += edge[cheapest[i]].weight;
        printf("Edge %d-%d included in MST\n",
               edge[cheapest[i]].src, edge[cheapest[i]].dest);

        // Do a union of set1 and set2 and decrease number
        // of trees
        Union(subsets, set1, set2);
        numTrees--;
    }
}

printf("Weight of MST is %d\n", MSTweight);
return;
}

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}

// A utility function to find set of an element i
// (uses path compression technique)
int find(struct subset subsets[], int i)
{
    // find root and make root as parent of i
    // (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent =
            find(subsets, subsets[i].parent);
}

```

```

        return subsets[i].parent;
    }

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(struct subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high
    // rank tree (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and
    // increment its rank by one
    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Driver program to test above functions
int main()
{
    /* Let us create following weighted graph
       10
       0-----1
       | \     |
       6|   5\   |15
       |     \  |
       2-----3
       4      */
    int V = 4; // Number of vertices in graph
    int E = 5; // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = 10;

    // add edge 0-2
}

```

```
graph->edge[1].src = 0;
graph->edge[1].dest = 2;
graph->edge[1].weight = 6;

// add edge 0-3
graph->edge[2].src = 0;
graph->edge[2].dest = 3;
graph->edge[2].weight = 5;

// add edge 1-3
graph->edge[3].src = 1;
graph->edge[3].dest = 3;
graph->edge[3].weight = 15;

// add edge 2-3
graph->edge[4].src = 2;
graph->edge[4].dest = 3;
graph->edge[4].weight = 4;

boruvkaMST(graph);

return 0;
}

// Thanks to Anukul Chand for modifying above code.
```

Python

```
# Boruvka's algorithm to find Minimum Spanning
# Tree of a given connected, undirected and weighted graph

from collections import defaultdict

#Class to represent a graph
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices
        self.graph = [] # default dictionary to store graph

    # function to add an edge to graph
    def addEdge(self,u,v,w):
        self.graph.append([u,v,w])

    # A utility function to find set of an element i
    # (uses path compression technique)
    def find(self, parent, i):
```

```

if parent[i] == i:
    return i
return self.find(parent, parent[i])

# A function that does union of two sets of x and y
# (uses union by rank)
def union(self, parent, rank, x, y):
    xroot = self.find(parent, x)
    yroot = self.find(parent, y)

    # Attach smaller rank tree under root of high rank tree
    # (Union by Rank)
    if rank[xroot] < rank[yroot]:
        parent[xroot] = yroot
    elif rank[xroot] > rank[yroot]:
        parent[yroot] = xroot
    #If ranks are same, then make one as root and increment
    # its rank by one
    else :
        parent[yroot] = xroot
        rank[xroot] += 1

# The main function to construct MST using Kruskal's algorithm
def boruvkaMST(self):
    parent = [] ; rank = [];

    # An array to store index of the cheapest edge of
    # subset. It store [u,v,w] for each component
    cheapest =[]

    # Initially there are V different trees.
    # Finally there will be one tree that will be MST
    numTrees = self.V
    MSTweight = 0

    # Create V subsets with single elements
    for node in range(self.V):
        parent.append(node)
        rank.append(0)
        cheapest =[-1] * self.V

    # Keep combining components (or sets) until all
    # components are not combined into single MST

    while numTrees > 1:

        # Traverse through all edges and update
        # cheapest of every component

```

```

for i in range(len(self.graph)):

    # Find components (or sets) of two corners
    # of current edge
    u,v,w = self.graph[i]
    set1 = self.find(parent, u)
    set2 = self.find(parent ,v)

    # If two corners of current edge belong to
    # same set, ignore current edge. Else check if
    # current edge is closer to previous
    # cheapest edges of set1 and set2
    if set1 != set2:

        if cheapest[set1] == -1 or cheapest[set1][2] > w :
            cheapest[set1] = [u,v,w]

        if cheapest[set2] == -1 or cheapest[set2][2] > w :
            cheapest[set2] = [u,v,w]

    # Consider the above picked cheapest edges and add them
    # to MST
    for node in range(self.V):

        #Check if cheapest for current set exists
        if cheapest[node] != -1:
            u,v,w = cheapest[node]
            set1 = self.find(parent, u)
            set2 = self.find(parent ,v)

            if set1 != set2 :
                MSTweight += w
                self.union(parent, rank, set1, set2)
                print ("Edge %d-%d with weight %d included in MST" % (u,v,w))
                numTrees = numTrees - 1

        #reset cheapest array
        cheapest = [-1] * self.V

    print ("Weight of MST is %d" % MSTweight)

g = Graph(4)
g.addEdge(0, 1, 10)
g.addEdge(0, 2, 6)
g.addEdge(0, 3, 5)

```

```
g.addEdge(1, 3, 15)
g.addEdge(2, 3, 4)

g.boruvkaMST()

#This code is contributed by Neelam Yadav
```

Output:

```
Edge 0-3 included in MST
Edge 0-1 included in MST
Edge 2-3 included in MST
Weight of MST is 19
```

Interesting Facts about Boruvka's algorithm:

- 1) Time Complexity of Boruvka's algorithm is $O(E \log V)$ which is same as Kruskal's and Prim's algorithms.
- 2) Boruvka's algorithm is used as a step in a [faster randomized algorithm that works in linear time \$O\(E\)\$](#) .
- 3) Boruvka's algorithm is the oldest minimum spanning tree algorithm was discovered by Boruvka in 1926, long before computers even existed. The algorithm was published as a method of constructing an efficient electricity network.

Exercise:

The above code assumes that input graph is connected and it fails if a disconnected graph is given. Extend the above algorithm so that it works for a disconnected graph also and produces a forest.

References:

http://en.wikipedia.org/wiki/Bor%C5%AFvka%27s_algorithm

Improved By : [Cyberfreak](#)

Source

<https://www.geeksforgeeks.org/boruvkas-algorithm-greedy-algo-9/>

Chapter 24

Bellman-Ford Algorithm

Bellman-Ford Algorithm DP-23 - GeeksforGeeks

Given a graph and a source vertex src in graph, find shortest paths from src to all vertices in the given graph. The graph may contain negative weight edges.

We have discussed [Dijkstra's algorithm](#) for this problem. Dijkstra's algorithm is a Greedy algorithm and time complexity is $O(V \log V)$ (with the use of Fibonacci heap). *Dijkstra doesn't work for Graphs with negative weight edges, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is $O(VE)$, which is more than Dijkstra.*

Algorithm

Following are the detailed steps.

Input: Graph and a source vertex src

Output: Shortest distance to all vertices from src . If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

1) This step initializes distances from source to all vertices as infinite and distance to source itself as 0. Create an array $dist[]$ of size V with all values as infinite except $dist[src]$ where src is source vertex.

2) This step calculates shortest distances. Do following $V-1$ times where V is the number of vertices in given graph.

....**a)** Do following for each edge $u-v$

.....If $dist[v] > dist[u] + \text{weight of edge } uv$, then update $dist[v]$

..... $dist[v] = dist[u] + \text{weight of edge } uv$

3) This step reports if there is a negative weight cycle in graph. Do following for each edge $u-v$

.....If $dist[v] > dist[u] + \text{weight of edge } uv$, then “Graph contains negative weight cycle”

The idea of step 3 is, step 2 guarantees shortest distances if graph doesn't contain negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

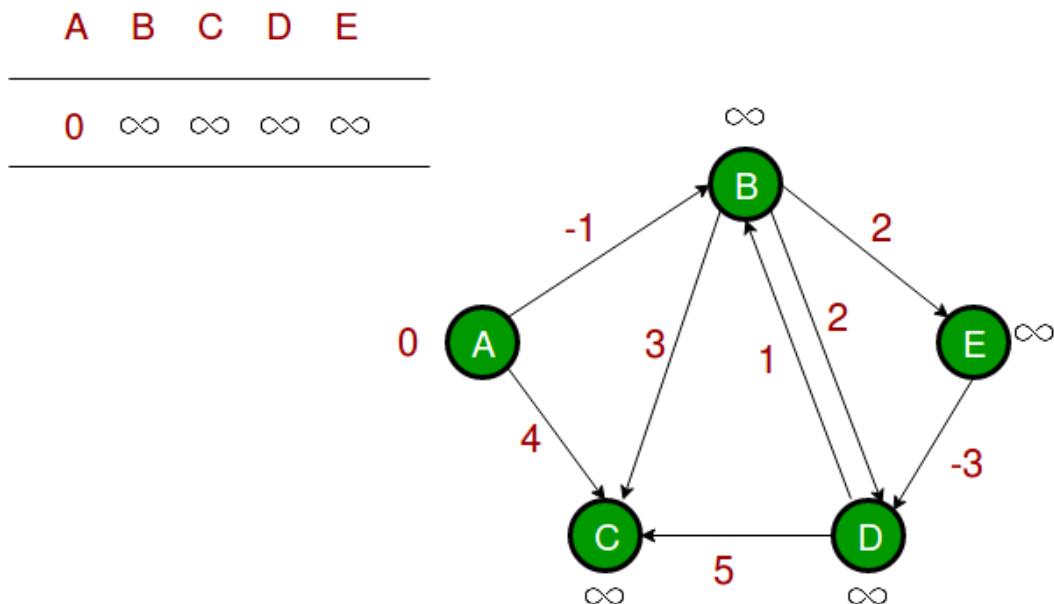
How does this work? Like other Dynamic Programming Problems, the algorithm calculate shortest paths in bottom-up manner. It first calculates the shortest distances which

have at-most one edge in the path. Then, it calculates shortest paths with at-most 2 edges, and so on. After the i -th iteration of outer loop, the shortest paths with at most i edges are calculated. There can be maximum $V - 1$ edges in any simple path, that is why the outer loop runs $v - 1$ times. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most $(i+1)$ edges (Proof is simple, you can refer [this](#) or [MIT Video Lecture](#))

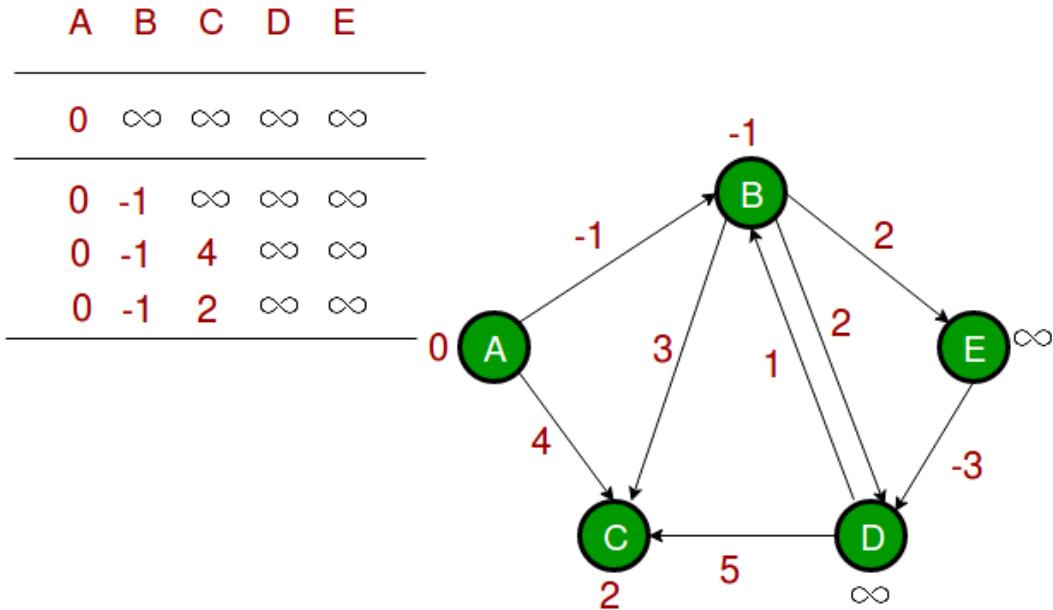
Example

Let us understand the algorithm with following example graph. The images are taken from [this source](#).

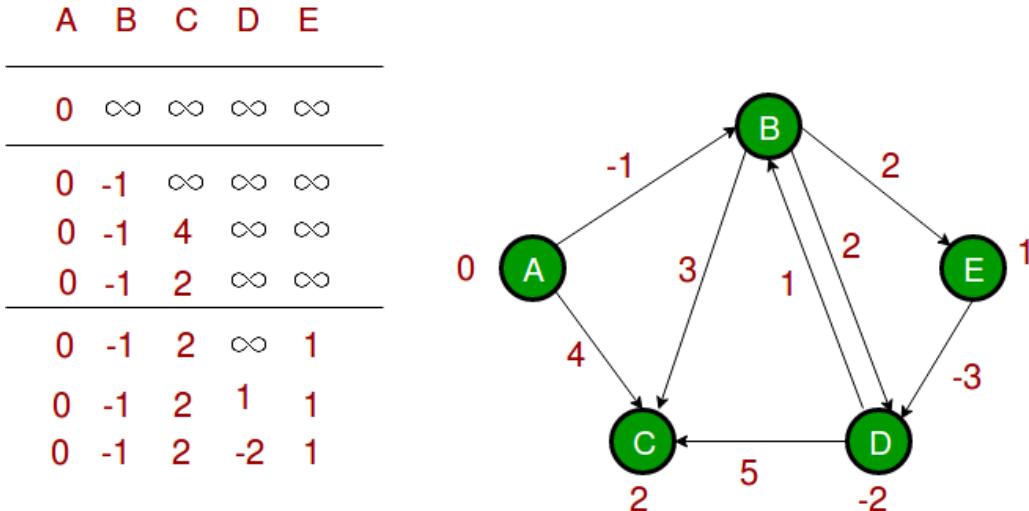
Let the given source vertex be 0. Initialize all distances as infinite, except the distance to source itself. Total number of vertices in the graph is 5, so *all edges must be processed 4 times*.



Let all edges are processed in following order: (B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D). We get following distances when all edges are processed first time. The first row in shows initial distances. The second row shows distances when edges (B,E), (D,B), (B,D) and (A,B) are processed. The third row shows distances when (A,C) is processed. The fourth row shows when (D,C), (B,C) and (E,D) are processed.



The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get following distances when all edges are processed second time (The last row shows final values).



The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don't update the distances.

Implementation:

C++

```
// A C++ program for Bellman-Ford's single source
```

```
// shortest path algorithm.
#include <bits/stdc++.h>

// a structure to represent a weighted edge in graph
struct Edge
{
    int src, dest, weight;
};

// a structure to represent a connected, directed and
// weighted graph
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}

// A utility function used to print the solution
void printArr(int dist[], int n)
{
    printf("Vertex      Distance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

// The main function that finds shortest distances from src to
// all other vertices using Bellman-Ford algorithm. The function
// also detects negative weight cycle
void BellmanFord(struct Graph* graph, int src)
{
    int V = graph->V;
    int E = graph->E;
    int dist[V];

    // Step 1: Initialize distances from src to all other vertices
```

```

// as INFINITE
for (int i = 0; i < V; i++)
    dist[i] = INT_MAX;
dist[src] = 0;

// Step 2: Relax all edges |V| - 1 times. A simple shortest
// path from src to any other vertex can have at-most |V| - 1
// edges
for (int i = 1; i <= V-1; i++)
{
    for (int j = 0; j < E; j++)
    {
        int u = graph->edge[j].src;
        int v = graph->edge[j].dest;
        int weight = graph->edge[j].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
            dist[v] = dist[u] + weight;
    }
}

// Step 3: check for negative-weight cycles. The above step
// guarantees shortest distances if graph doesn't contain
// negative weight cycle. If we get a shorter path, then there
// is a cycle.
for (int i = 0; i < E; i++)
{
    int u = graph->edge[i].src;
    int v = graph->edge[i].dest;
    int weight = graph->edge[i].weight;
    if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
        printf("Graph contains negative weight cycle");
}

printArr(dist, V);

return;
}

// Driver program to test above functions
int main()
{
    /* Let us create the graph given in above example */
    int V = 5; // Number of vertices in graph
    int E = 8; // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1 (or A-B in above figure)
    graph->edge[0].src = 0;
}

```

```
graph->edge[0].dest = 1;
graph->edge[0].weight = -1;

// add edge 0-2 (or A-C in above figure)
graph->edge[1].src = 0;
graph->edge[1].dest = 2;
graph->edge[1].weight = 4;

// add edge 1-2 (or B-C in above figure)
graph->edge[2].src = 1;
graph->edge[2].dest = 2;
graph->edge[2].weight = 3;

// add edge 1-3 (or B-D in above figure)
graph->edge[3].src = 1;
graph->edge[3].dest = 3;
graph->edge[3].weight = 2;

// add edge 1-4 (or A-E in above figure)
graph->edge[4].src = 1;
graph->edge[4].dest = 4;
graph->edge[4].weight = 2;

// add edge 3-2 (or D-C in above figure)
graph->edge[5].src = 3;
graph->edge[5].dest = 2;
graph->edge[5].weight = 5;

// add edge 3-1 (or D-B in above figure)
graph->edge[6].src = 3;
graph->edge[6].dest = 1;
graph->edge[6].weight = 1;

// add edge 4-3 (or E-D in above figure)
graph->edge[7].src = 4;
graph->edge[7].dest = 3;
graph->edge[7].weight = -3;

BellmanFord(graph, 0);

return 0;
}
```

Java

```
// A Java program for Bellman-Ford's single source shortest path
// algorithm.
import java.util.*;
```

```

import java.lang.*;
import java.io.*;

// A class to represent a connected, directed and weighted graph
class Graph
{
    // A class to represent a weighted edge in graph
    class Edge {
        int src, dest, weight;
        Edge() {
            src = dest = weight = 0;
        }
    };
    int V, E;
    Edge edge[];

    // Creates a graph with V vertices and E edges
    Graph(int v, int e)
    {
        V = v;
        E = e;
        edge = new Edge[e];
        for (int i=0; i<e; ++i)
            edge[i] = new Edge();
    }

    // The main function that finds shortest distances from src
    // to all other vertices using Bellman-Ford algorithm. The
    // function also detects negative weight cycle
    void BellmanFord(Graph graph,int src)
    {
        int V = graph.V, E = graph.E;
        int dist[] = new int[V];

        // Step 1: Initialize distances from src to all other
        // vertices as INFINITE
        for (int i=0; i<V; ++i)
            dist[i] = Integer.MAX_VALUE;
        dist[src] = 0;

        // Step 2: Relax all edges |V| - 1 times. A simple
        // shortest path from src to any other vertex can
        // have at-most |V| - 1 edges
        for (int i=1; i<V; ++i)
        {
            for (int j=0; j<E; ++j)
            {

```

```

        int u = graph.edge[j].src;
        int v = graph.edge[j].dest;
        int weight = graph.edge[j].weight;
        if (dist[u] != Integer.MAX_VALUE &&
            dist[u]+weight<dist[v])
            dist[v]=dist[u]+weight;
    }
}

// Step 3: check for negative-weight cycles. The above
// step guarantees shortest distances if graph doesn't
// contain negative weight cycle. If we get a shorter
// path, then there is a cycle.
for (int j=0; j<E; ++j)
{
    int u = graph.edge[j].src;
    int v = graph.edge[j].dest;
    int weight = graph.edge[j].weight;
    if (dist[u] != Integer.MAX_VALUE &&
        dist[u]+weight < dist[v])
        System.out.println("Graph contains negative weight cycle");
}
printArr(dist, V);
}

// A utility function used to print the solution
void printArr(int dist[], int V)
{
    System.out.println("Vertex      Distance from Source");
    for (int i=0; i<V; ++i)
        System.out.println(i+"\t\t"+dist[i]);
}

// Driver method to test above function
public static void main(String[] args)
{
    int V = 5; // Number of vertices in graph
    int E = 8; // Number of edges in graph

    Graph graph = new Graph(V, E);

    // add edge 0-1 (or A-B in above figure)
    graph.edge[0].src = 0;
    graph.edge[0].dest = 1;
    graph.edge[0].weight = -1;

    // add edge 0-2 (or A-C in above figure)
    graph.edge[1].src = 0;
}

```

```
graph.edge[1].dest = 2;
graph.edge[1].weight = 4;

// add edge 1-2 (or B-C in above figure)
graph.edge[2].src = 1;
graph.edge[2].dest = 2;
graph.edge[2].weight = 3;

// add edge 1-3 (or B-D in above figure)
graph.edge[3].src = 1;
graph.edge[3].dest = 3;
graph.edge[3].weight = 2;

// add edge 1-4 (or A-E in above figure)
graph.edge[4].src = 1;
graph.edge[4].dest = 4;
graph.edge[4].weight = 2;

// add edge 3-2 (or D-C in above figure)
graph.edge[5].src = 3;
graph.edge[5].dest = 2;
graph.edge[5].weight = 5;

// add edge 3-1 (or D-B in above figure)
graph.edge[6].src = 3;
graph.edge[6].dest = 1;
graph.edge[6].weight = 1;

// add edge 4-3 (or E-D in above figure)
graph.edge[7].src = 4;
graph.edge[7].dest = 3;
graph.edge[7].weight = -3;

graph.BellmanFord(graph, 0);
}

}

// Contributed by Aakash Hasija
```

Python

```
# Python program for Bellman-Ford's single source
# shortest path algorithm.

from collections import defaultdict

#Class to represent a graph
class Graph:
```

```

def __init__(self,vertices):
    self.V= vertices #No. of vertices
    self.graph = {} # default dictionary to store graph

# function to add an edge to graph
def addEdge(self,u,v,w):
    self.graph.append([u, v, w])

# utility function used to print the solution
def printArr(self, dist):
    print("Vertex   Distance from Source")
    for i in range(self.V):
        print("%d \t\t %d" % (i, dist[i]))

# The main function that finds shortest distances from src to
# all other vertices using Bellman-Ford algorithm. The function
# also detects negative weight cycle
def BellmanFord(self, src):

    # Step 1: Initialize distances from src to all other vertices
    # as INFINITE
    dist = [float("Inf")] * self.V
    dist[src] = 0

    # Step 2: Relax all edges |V| - 1 times. A simple shortest
    # path from src to any other vertex can have at-most |V| - 1
    # edges
    for i in range(self.V - 1):
        # Update dist value and parent index of the adjacent vertices of
        # the picked vertex. Consider only those vertices which are still in
        # queue
        for u, v, w in self.graph:
            if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w

    # Step 3: check for negative-weight cycles. The above step
    # guarantees shortest distances if graph doesn't contain
    # negative weight cycle. If we get a shorter path, then there
    # is a cycle.

    for u, v, w in self.graph:
        if dist[u] != float("Inf") and dist[u] + w < dist[v]:
            print "Graph contains negative weight cycle"
            return

    # print all distance
    self.printArr(dist)

```

```
g = Graph(5)
g.addEdge(0, 1, -1)
g.addEdge(0, 2, 4)
g.addEdge(1, 2, 3)
g.addEdge(1, 3, 2)
g.addEdge(1, 4, 2)
g.addEdge(3, 2, 5)
g.addEdge(3, 1, 1)
g.addEdge(4, 3, -3)

#print the solution
g.BellmanFord(0)

#This code is contributed by Neelam Yadav
```

Output:

Vertex	Distance from Source
0	0
1	-1
2	2
3	-2
4	1

Notes

- 1) Negative weights are found in various applications of graphs. For example, instead of paying cost for a path, we may get some advantage if we follow the path.
- 2) Bellman-Ford works better (better than Dijksra's) for distributed systems. Unlike Dijksra's where we need to find minimum value of all vertices, in Bellman-Ford, edges are considered one by one.

Exercise

- 1) The standard Bellman-Ford algorithm reports shortest path only if there is no negative weight cycles. Modify it so that it reports minimum distances even if there is a negative weight cycle.
- 2) Can we use Dijksra's algorithm for shortest paths for graphs with negative weights – one idea can be, calculate the minimum weight value, add a positive value (equal to absolute value of minimum weight value) to all weights and run the Dijksra's algorithm for the modified graph. Will this algorithm work?

References:

- <http://www.youtube.com/watch?v=Ttezuzs39nk>
- http://en.wikipedia.org/wiki/Bellman%20%93Ford_algorithm
- <http://www.cs.arizona.edu/classes/cs445/spring07/ShortestPath2.prn.pdf>

Source

<https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/>

Chapter 25

Floyd Warshall Algorithm

Floyd Warshall Algorithm DP-16 - GeeksforGeeks

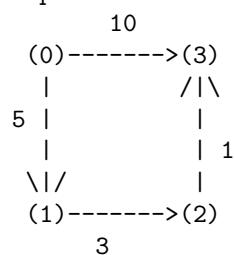
The [Floyd Warshall Algorithm](#) is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

Example:

Input:

```
graph[] [] = { {0, 5, INF, 10},
               {INF, 0, 3, INF},
               {INF, INF, 0, 1},
               {INF, INF, INF, 0} }
```

which represents the following graph



Note that the value of $\text{graph}[i][j]$ is 0 if i is equal to j .
And $\text{graph}[i][j]$ is INF (infinite) if there is no edge from vertex i to j .

Output:

Shortest distance matrix

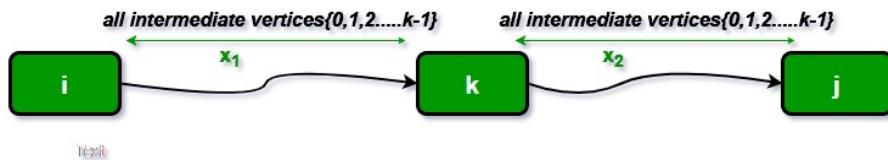
0	5	8	9
INF	0	3	4
INF	INF	0	1
INF	INF	INF	0

Floyd Warshall Algorithm

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices $\{0, 1, 2, \dots, k-1\}$ as intermediate vertices. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.

- 1) k is not an intermediate vertex in shortest path from i to j. We keep the value of $dist[i][j]$ as it is.
- 2) k is an intermediate vertex in shortest path from i to j. We update the value of $dist[i][j]$ as $dist[i][k] + dist[k][j]$.

The following figure shows the above optimal substructure property in the all-pairs shortest path problem.



Following is implementations of the Floyd Warshall algorithm.

C/C++

```
// C Program for Floyd Warshall Algorithm
#include<stdio.h>

// Number of vertices in the graph
#define V 4

/* Define Infinite as a large enough value. This value will be used
   for vertices not connected to each other */
#define INF 99999

// A function to print the solution matrix
void printSolution(int dist[][]);

// Solves the all-pairs shortest path problem using Floyd Warshall algorithm
void floydWarshall (int graph[][V])
{
    /* dist[][] will be the output matrix that will finally have the shortest
       distances between every pair of vertices */
    int dist[V][V], i, j, k;

    /* Initialize the solution matrix same as input graph matrix. Or
       we can say the initial values of shortest distances are based
       on shortest paths considering no intermediate vertex. */
    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];
}
```

```

        for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];

/* Add all vertices one by one to the set of intermediate vertices.
---> Before start of an iteration, we have shortest distances between all
pairs of vertices such that the shortest distances consider only the
vertices in set {0, 1, 2, .. k-1} as intermediate vertices.
----> After the end of an iteration, vertex no. k is added to the set of
intermediate vertices and the set becomes {0, 1, 2, .. k} */
for (k = 0; k < V; k++)
{
    // Pick all vertices as source one by one
    for (i = 0; i < V; i++)
    {
        // Pick all vertices as destination for the
        // above picked source
        for (j = 0; j < V; j++)
        {
            // If vertex k is on the shortest path from
            // i to j, then update the value of dist[i][j]
            if (dist[i][k] + dist[k][j] < dist[i][j])
                dist[i][j] = dist[i][k] + dist[k][j];
        }
    }
}

// Print the shortest distance matrix
printSolution(dist);
}

/* A utility function to print solution */
void printSolution(int dist[][])
{
    printf ("The following matrix shows the shortest distances"
           " between every pair of vertices \n");
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            if (dist[i][j] == INF)
                printf("%7s", "INF");
            else
                printf ("%7d", dist[i][j]);
        }
        printf("\n");
    }
}

```

```

// driver program to test above function
int main()
{
    /* Let us create the following weighted graph
       10
       (0)----->(3)
          |           /|\
          5 |           |
          |           | 1
          \|/           |
       (1)----->(2)
          3           */
    int graph[V][V] = { {0, 5, INF, 10},
                        {INF, 0, 3, INF},
                        {INF, INF, 0, 1},
                        {INF, INF, INF, 0}
                      };
}

// Print the solution
floydWarshall(graph);
return 0;
}

```

Java

```

// A Java program for Floyd Warshall All Pairs Shortest
// Path algorithm.
import java.util.*;
import java.lang.*;
import java.io.*;

class AllPairShortestPath
{
    final static int INF = 99999, V = 4;

    void floydWarshall(int graph[][])
    {
        int dist[][] = new int[V][V];
        int i, j, k;

        /* Initialize the solution matrix same as input graph matrix.
           Or we can say the initial values of shortest distances
           are based on shortest paths considering no intermediate
           vertex. */
        for (i = 0; i < V; i++)
            for (j = 0; j < V; j++)
                dist[i][j] = graph[i][j];
    }
}

```

```

/*
 * Add all vertices one by one to the set of intermediate
 * vertices.
 ----> Before start of an iteration, we have shortest
      distances between all pairs of vertices such that
      the shortest distances consider only the vertices in
      set {0, 1, 2, .. k-1} as intermediate vertices.
 -----> After the end of an iteration, vertex no. k is added
      to the set of intermediate vertices and the set
      becomes {0, 1, 2, .. k} */
for (k = 0; k < V; k++)
{
    // Pick all vertices as source one by one
    for (i = 0; i < V; i++)
    {
        // Pick all vertices as destination for the
        // above picked source
        for (j = 0; j < V; j++)
        {
            // If vertex k is on the shortest path from
            // i to j, then update the value of dist[i][j]
            if (dist[i][k] + dist[k][j] < dist[i][j])
                dist[i][j] = dist[i][k] + dist[k][j];
        }
    }
}

// Print the shortest distance matrix
printSolution(dist);
}

void printSolution(int dist[][])
{
    System.out.println("The following matrix shows the shortest "+
                       "distances between every pair of vertices");
    for (int i=0; i<V; ++i)
    {
        for (int j=0; j<V; ++j)
        {
            if (dist[i][j]==INF)
                System.out.print("INF ");
            else
                System.out.print(dist[i][j]+ " ");
        }
        System.out.println();
    }
}

```

```

// Driver program to test above function
public static void main (String[] args)
{
    /* Let us create the following weighted graph
       10
       (0)----->(3)
       |           /|\
      5 |           |
       |           | 1
      \|/           |
       (1)----->(2)
                   3
    */
    int graph[][] = { {0, 5, INF, 10},
                      {INF, 0, 3, INF},
                      {INF, INF, 0, 1},
                      {INF, INF, INF, 0}
                };
    AllPairShortestPath a = new AllPairShortestPath();

    // Print the solution
    a.floydWarshall(graph);
}
}

// Contributed by Aakash Hasija

```

Python

```

# Python Program for Floyd Warshall Algorithm

# Number of vertices in the graph
V = 4

# Define infinity as the large enough value. This value will be
# used for vertices not connected to each other
INF = 99999

# Solves all pair shortest path via Floyd Warshall Algorithm
def floydWarshall(graph):

    """ dist[][] will be the output matrix that will finally
        have the shortest distances between every pair of vertices """
    """ initializing the solution matrix same as input graph matrix
    OR we can say that the initial values of shortest distances
    are based on shortest paths considering no
    intermediate vertices """
    dist = map(lambda i : map(lambda j : j , i) , graph)

    # print the initial solution
    print ("The initial solution matrix is as follows")
    printMat(dist)
    print()

    # Compute shortest distance between all pairs of vertices
    for k in range(V):
        for i in range(V):
            for j in range(V):
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
    # print the final shortest distance matrix
    print ("The shortest distance matrix is as follows")
    printMat(dist)

```

```

""" Add all vertices one by one to the set of intermediate
vertices.
---> Before start of an iteration, we have shortest distances
between all pairs of vertices such that the shortest
distances consider only the vertices in the set
{0, 1, 2, .. k-1} as intermediate vertices.
----> After the end of a iteration, vertex no. k is
added to the set of intermediate vertices and the
set becomes {0, 1, 2, .. k}
"""
for k in range(V):

    # pick all vertices as source one by one
    for i in range(V):

        # Pick all vertices as destination for the
        # above picked source
        for j in range(V):

            # If vertex k is on the shortest path from
            # i to j, then update the value of dist[i][j]
            dist[i][j] = min(dist[i][j] ,
                               dist[i][k]+ dist[k][j]
                               )
printSolution(dist)

# A utility function to print the solution
def printSolution(dist):
    print "Following matrix shows the shortest distances\
between every pair of vertices"
    for i in range(V):
        for j in range(V):
            if(dist[i][j] == INF):
                print "%7s" %("INF"),
            else:
                print "%7d\t" %(dist[i][j]),
            if j == V-1:
                print ""

# Driver program to test the above program
# Let us create the following weighted graph
"""
          10
(0)----->(3)
           |
           / \ \

```

```

5 |           |
 |           | 1
 \|/           |
(1)----->(2)
      3           """
graph = [[0,5,INF,10],
         [INF,0,3,INF],
         [INF, INF, 0, 1],
         [INF, INF, INF, 0]
        ]
# Print the solution
floydWarshall(graph);
# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

C#

```

// A C# program for Floyd Warshall All
// Pairs Shortest Path algorithm.

using System;

public class AllPairShortestPath
{
    readonly static int INF = 99999, V = 4;

    void floydWarshall(int[,] graph)
    {
        int[,] dist = new int[V, V];
        int i, j, k;

        // Initialize the solution matrix
        // same as input graph matrix
        // Or we can say the initial
        // values of shortest distances
        // are based on shortest paths
        // considering no intermediate
        // vertex
        for (i = 0; i < V; i++) {
            for (j = 0; j < V; j++) {
                dist[i, j] = graph[i, j];
            }
        }

        /* Add all vertices one by one to
        the set of intermediate vertices.
        ---> Before start of a iteration,
        we have shortest distances
        between all pairs of vertices
```

```

such that the shortest distances
consider only the vertices in
set {0, 1, 2, .. k-1} as
intermediate vertices.
----> After the end of a iteration,
vertex no. k is added
to the set of intermediate
vertices and the set
becomes {0, 1, 2, .. k} */
for (k = 0; k < V; k++)
{
    // Pick all vertices as source
    // one by one
    for (i = 0; i < V; i++)
    {
        // Pick all vertices as destination
        // for the above picked source
        for (j = 0; j < V; j++)
        {
            // If vertex k is on the shortest
            // path from i to j, then update
            // the value of dist[i][j]
            if (dist[i, k] + dist[k, j] < dist[i, j])
            {
                dist[i, j] = dist[i, k] + dist[k, j];
            }
        }
    }
}

// Print the shortest distance matrix
printSolution(dist);
}

void printSolution(int[,] dist)
{
    Console.WriteLine("Following matrix shows the shortest "+
                      "distances between every pair of vertices");
    for (int i = 0; i < V; ++i)
    {
        for (int j = 0; j < V; ++j)
        {
            if (dist[i, j] == INF) {
                Console.Write("INF ");
            } else {
                Console.Write(dist[i, j] + " ");
            }
        }
    }
}

```

```

        Console.WriteLine();
    }

}

// Driver Code
public static void Main(string[] args)
{
    /* Let us create the following
       weighted graph
       10
       (0)----->(3)
       |           / \
      5 |           |
       |           | 1
      \|/           |
       (1)----->(2)
                   3          */
    int[,] graph = { {0, 5, INF, 10},
                    {INF, 0, 3, INF},
                    {INF, INF, 0, 1},
                    {INF, INF, INF, 0}
                };

    AllPairShortestPath a = new AllPairShortestPath();

    // Print the solution
    a.floydWarshall(graph);
}
}

// This article is contributed by
// Abdul Mateen Mohammed

```

Output:

```

Following matrix shows the shortest distances between every pair of vertices
      0      5      8      9
INF      0      3      4
INF      INF     0      1
INF      INF     INF     0

```

Time Complexity: $O(V^3)$

The above program only prints the shortest distances. We can modify the solution to print the shortest paths also by storing the predecessor information in a separate 2D matrix. Also, the value of INF can be taken as INT_MAX from limits.h to make sure that we

handle maximum possible value. When we take INF as INT_MAX, we need to change the if condition in the above program to avoid arithmetic overflow.

```
#include

#define INF INT_MAX
.....
if ( dist[i][k] != INF &&
    dist[k][j] != INF &&
    dist[i][k] + dist[k][j] < dist[i][j]
)
    dist[i][j] = dist[i][k] + dist[k][j];
....
```

Improved By : [Abdul Mateen Mohammed](#)

Source

<https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/>

Chapter 26

Johnson's algorithm for All-pairs shortest paths

Johnson's algorithm for All-pairs shortest paths - GeeksforGeeks

The problem is to find shortest paths between every pair of vertices in a given weighted directed Graph and weights may be negative. We have discussed [Floyd Warshall Algorithm](#) for this problem. Time complexity of Floyd Warshall Algorithm is $\Theta(V^3)$. *Using Johnson's algorithm, we can find all pair shortest paths in $O(V^2 \log V + VE)$ time.* Johnson's algorithm uses both [Dijkstra](#) and [Bellman-Ford](#) as subroutines.

If we apply [Dijkstra's Single Source shortest path algorithm](#) for every vertex, considering every vertex as source, we can find all pair shortest paths in $O(V^2V\log V)$ time. So using Dijkstra's single source shortest path seems to be a better option than [Floyd Warshell](#), but the problem with Dijkstra's algorithm is, it doesn't work for negative weight edge.
The idea of Johnson's algorithm is to re-weight all edges and make them all positive, then apply Dijkstra's algorithm for every vertex.

How to transform a given graph to a graph with all non-negative weight edges?
One may think of a simple approach of finding the minimum weight edge and adding this weight to all edges. Unfortunately, this doesn't work as there may be different number of edges in different paths (See [this](#) for an example). If there are multiple paths from a vertex u to v, then all paths must be increased by same amount, so that the shortest path remains the shortest in the transformed graph.

The idea of Johnson's algorithm is to assign a weight to every vertex. Let the weight assigned to vertex u be $h[u]$. We reweight edges using vertex weights. For example, for an edge (u, v) of weight $w(u, v)$, the new weight becomes $w(u, v) + h[u] - h[v]$. The great thing about this reweighting is, all set of paths between any two vertices are increased by same amount and all negative weights become non-negative. Consider any path between two vertices s and t, weight of every path is increased by $h[s] - h[t]$, all $h[]$ values of vertices on path from s to t cancel each other.

How do we calculate $h[]$ values? [Bellman-Ford algorithm](#) is used for this purpose. Following is the complete algorithm. A new vertex is added to the graph and connected to all existing vertices. The shortest distance values from new vertex to all existing vertices are $h[]$ values.

Algorithm:

- 1) Let the given graph be G. Add a new vertex s to the graph, add edges from new vertex to all vertices of G. Let the modified graph be G'.
- 2) Run [Bellman-Ford algorithm](#) on G' with s as source. Let the distances calculated by Bellman-Ford be $h[0], h[1], \dots, h[V-1]$. If we find a negative weight cycle, then return. Note that the negative weight cycle cannot be created by new vertex s as there is no edge to s. All edges are from s.
- 3) Reweight the edges of original graph. For each edge (u, v) , assign the new weight as "original weight + $h[u] - h[v]$ ".
- 4) Remove the added vertex s and run [Dijkstra's algorithm](#) for every vertex.

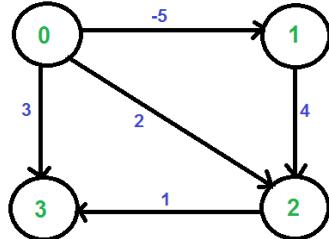
How does the transformation ensure nonnegative weight edges?

The following property is always true about $h[]$ values as they are shortest distances.

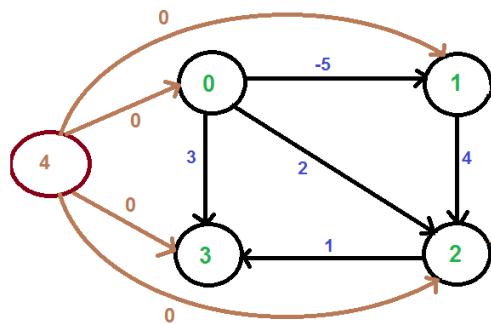
$$h[v] \leq h[u] + w(u, v)$$

The property simply means, shortest distance from s to v must be smaller than or equal to shortest distance from s to u plus weight of edge (u, v) . The new weights are $w(u, v) + h[u] - h[v]$. The value of the new weights must be greater than or equal to zero because of the inequality " $h[v] \leq h[u] + w(u, v)$ ". **Example:**

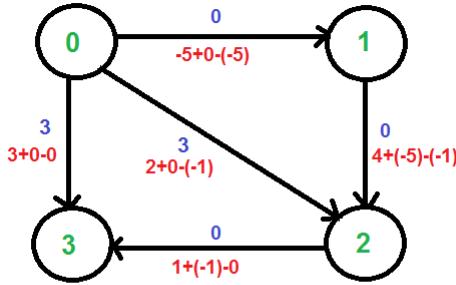
Let us consider the following graph.



We add a source s and add edges from s to all vertices of the original graph. In the following diagram s is 4.



We calculate the shortest distances from 4 to all other vertices using Bellman-Ford algorithm. The shortest distances from 4 to 0, 1, 2 and 3 are 0, -5, -1 and 0 respectively, i.e., $h[] = \{0, -5, -1, 0\}$. Once we get these distances, we remove the source vertex 4 and reweight the edges using following formula. $w(u, v) = w(u, v) + h[u] - h[v]$.



Distances from 4 to 0, 1, 2 and 3 are 0, -5, -1 and 0 respectively.

Since all weights are positive now, we can run Dijkstra's shortest path algorithm for every vertex as source.

Time Complexity: The main steps in algorithm are Bellman Ford Algorithm called once and Dijkstra called V times. Time complexity of Bellman Ford is $O(VE)$ and time complexity of Dijkstra is $O(V\log V)$. So overall time complexity is $O(V^2\log V + VE)$.
The time complexity of Johnson's algorithm becomes same as [Floyd Warshell](#) when the graphs is complete (For a complete graph $E = O(V^2)$). But for sparse graphs, the algorithm performs much better than [Floyd Warshell](#).

References:

- Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest
- <http://www.youtube.com/watch?v=b6LOHvCzmkI>
- <http://www.youtube.com/watch?v=TV2Z6nbolic>
- http://en.wikipedia.org/wiki/Johnson%27s_algorithm
- <http://www.youtube.com/watch?v=Sygq1e0xWnM>

Source

<https://www.geeksforgeeks.org/johnsons-algorithm/>

Chapter 27

Shortest Path in Directed Acyclic Graph

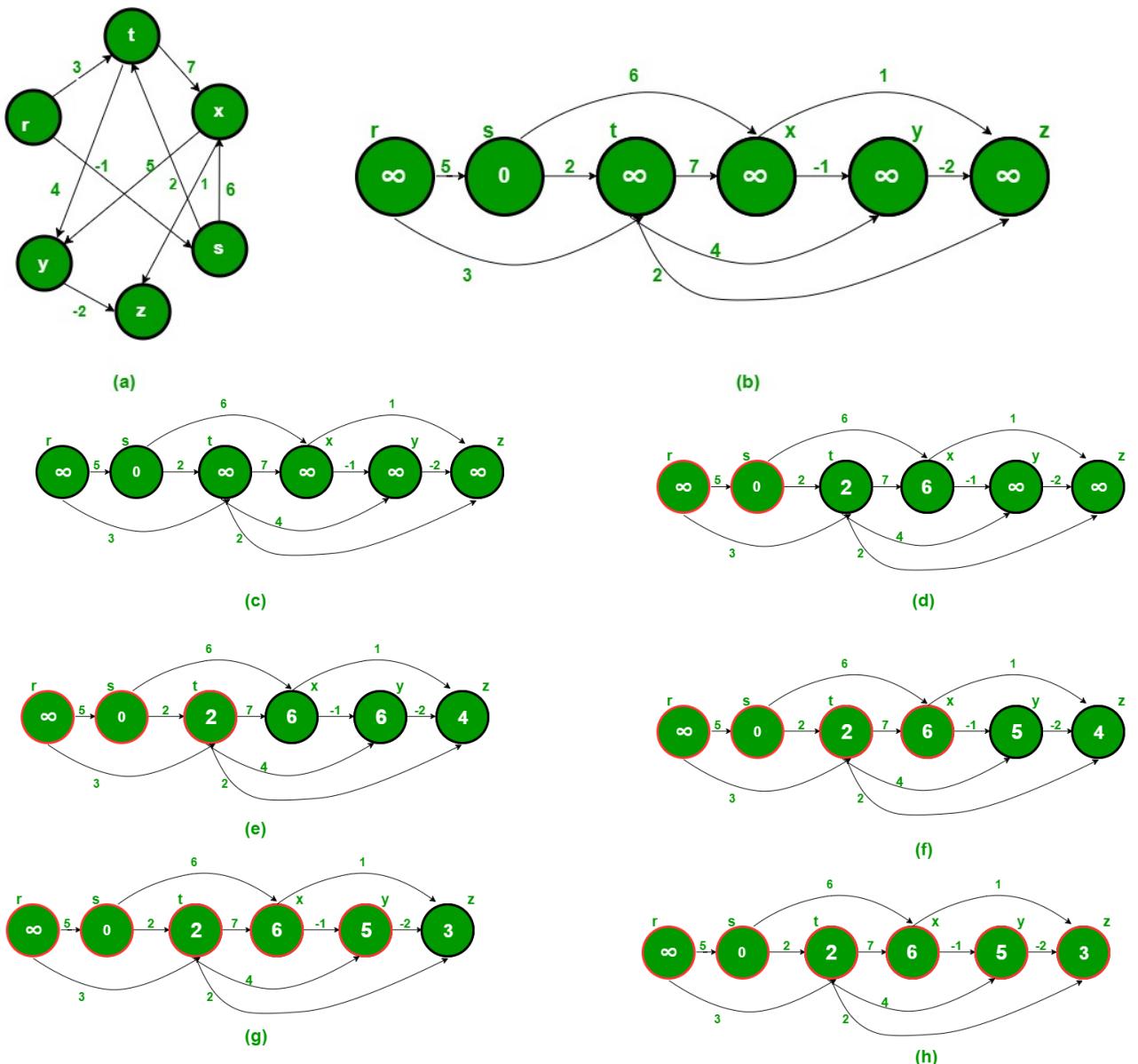
Shortest Path in Directed Acyclic Graph - GeeksforGeeks

Given a Weighted Directed Acyclic Graph and a source vertex in the graph, find the shortest paths from given source to all other vertices.

For a general weighted graph, we can calculate single source shortest distances in $O(VE)$ time using [Bellman–Ford Algorithm](#). For a graph with no negative weights, we can do better and calculate single source shortest distances in $O(E + V\log V)$ time using [Dijkstra's algorithm](#). Can we do even better for Directed Acyclic Graph (DAG)? We can calculate single source shortest distances in $O(V+E)$ time for DAGs. The idea is to use [Topological Sorting](#).

We initialize distances to all vertices as infinite and distance to source as 0, then we find a topological sorting of the graph. [Topological Sorting](#) of a graph represents a linear ordering of the graph (See below, figure (b) is a linear representation of figure (a)). Once we have topological order (or linear representation), we one by one process all vertices in topological order. For every vertex being processed, we update distances of its adjacent using distance of current vertex.

Following figure is taken from [this](#) source. It shows step by step process of finding shortest paths.



Following is complete algorithm for finding shortest distances.

- 1) Initialize $\text{dist}[] = \{\text{INF}, \text{INF}, \dots\}$ and $\text{dist}[s] = 0$ where s is the source vertex.
- 2) Create a topological order of all vertices.
- 3) Do following for every vertex u in topological order.
 -Do following for every adjacent vertex v of u
 -if ($\text{dist}[v] > \text{dist}[u] + \text{weight}(u, v)$)

```

.....dist[v] = dist[u] + weight(u, v)

C++

// C++ program to find single source shortest paths for Directed Acyclic Graphs
#include<iostream>
#include <list>
#include <stack>
#include <limits.h>
#define INF INT_MAX
using namespace std;

// Graph is represented using adjacency list. Every node of adjacency list
// contains vertex number of the vertex to which edge connects. It also
// contains weight of the edge
class AdjListNode
{
    int v;
    int weight;
public:
    AdjListNode(int _v, int _w) { v = _v; weight = _w; }
    int getV() { return v; }
    int getWeight() { return weight; }
};

// Class to represent a graph using adjacency list representation
class Graph
{
    int V;      // No. of vertices'

    // Pointer to an array containing adjacency lists
    list<AdjListNode> *adj;

    // A function used by shortestPath
    void topologicalSortUtil(int v, bool visited[], stack<int> &Stack);
public:
    Graph(int V);    // Constructor

    // function to add an edge to graph
    void addEdge(int u, int v, int weight);

    // Finds shortest paths from given source vertex
    void shortestPath(int s);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<AdjListNode>[V];

```

```

}

void Graph::addEdge(int u, int v, int weight)
{
    AdjListNode node(v, weight);
    adj[u].push_back(node); // Add v to u's list
}

// A recursive function used by shortestPath. See below link for details
// https://www.geeksforgeeks.org/topological-sorting/
void Graph::topologicalSortUtil(int v, bool visited[], stack<int> &Stack)
{
    // Mark the current node as visited
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<AdjListNode>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
    {
        AdjListNode node = *i;
        if (!visited[node.getV()])
            topologicalSortUtil(node.getV(), visited, Stack);
    }

    // Push current vertex to stack which stores topological sort
    Stack.push(v);
}

// The function to find shortest paths from given vertex. It uses recursive
// topologicalSortUtil() to get topological sorting of given graph.
void Graph::shortestPath(int s)
{
    stack<int> Stack;
    int dist[V];

    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to store Topological Sort
    // starting from all vertices one by one
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            topologicalSortUtil(i, visited, Stack);

    // Initialize distances to all vertices as infinite and distance
    // to source as 0
}

```

```

for (int i = 0; i < V; i++)
    dist[i] = INF;
dist[s] = 0;

// Process vertices in topological order
while (Stack.empty() == false)
{
    // Get the next vertex from topological order
    int u = Stack.top();
    Stack.pop();

    // Update distances of all adjacent vertices
    list<AdjListNode>::iterator i;
    if (dist[u] != INF)
    {
        for (i = adj[u].begin(); i != adj[u].end(); ++i)
            if (dist[i->getV()] > dist[u] + i->getWeight())
                dist[i->getV()] = dist[u] + i->getWeight();
    }
}

// Print the calculated shortest distances
for (int i = 0; i < V; i++)
    (dist[i] == INF)? cout << "INF ": cout << dist[i] << " ";
}

// Driver program to test above functions
int main()
{
    // Create a graph given in the above diagram. Here vertex numbers are
    // 0, 1, 2, 3, 4, 5 with following mappings:
    // 0=r, 1=s, 2=t, 3=x, 4=y, 5=z
    Graph g(6);
    g.addEdge(0, 1, 5);
    g.addEdge(0, 2, 3);
    g.addEdge(1, 3, 6);
    g.addEdge(1, 2, 2);
    g.addEdge(2, 4, 4);
    g.addEdge(2, 5, 2);
    g.addEdge(2, 3, 7);
    g.addEdge(3, 4, -1);
    g.addEdge(4, 5, -2);

    int s = 1;
    cout << "Following are shortest distances from source " << s << endl;
    g.shortestPath(s);

    return 0;
}

```

```
}
```

Java

```
// Java program to find single source shortest paths in Directed Acyclic Graphs
import java.io.*;
import java.util.*;

class ShortestPath
{
    static final int INF=Integer.MAX_VALUE;
    class AdjListNode
    {
        private int v;
        private int weight;
        AdjListNode(int _v, int _w) { v = _v;  weight = _w; }
        int getV() { return v; }
        int getWeight() { return weight; }
    }

    // Class to represent graph as an adjacency list of
    // nodes of type AdjListNode
    class Graph
    {
        private int V;
        private LinkedList<AdjListNode>adj[];
        Graph(int v)
        {
            V=v;
            adj = new LinkedList[V];
            for (int i=0; i<v; ++i)
                adj[i] = new LinkedList<AdjListNode>();
        }
        void addEdge(int u, int v, int weight)
        {
            AdjListNode node = new AdjListNode(v,weight);
            adj[u].add(node); // Add v to u's list
        }

        // A recursive function used by shortestPath.
        // See below link for details
        void topologicalSortUtil(int v, Boolean visited[], Stack stack)
        {
            // Mark the current node as visited.
            visited[v] = true;
            Integer i;

            // Recur for all the vertices adjacent to this vertex

```

```

Iterator<AdjListNode> it = adj[v].iterator();
while (it.hasNext())
{
    AdjListNode node = it.next();
    if (!visited[node.getV()])
        topologicalSortUtil(node.getV(), visited, stack);
}
// Push current vertex to stack which stores result
stack.push(new Integer(v));
}

// The function to find shortest paths from given vertex. It
// uses recursive topologicalSortUtil() to get topological
// sorting of given graph.
void shortestPath(int s)
{
    Stack stack = new Stack();
    int dist[] = new int[V];

    // Mark all the vertices as not visited
    Boolean visited[] = new Boolean[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to store Topological
    // Sort starting from all vertices one by one
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            topologicalSortUtil(i, visited, stack);

    // Initialize distances to all vertices as infinite and
    // distance to source as 0
    for (int i = 0; i < V; i++)
        dist[i] = INF;
    dist[s] = 0;

    // Process vertices in topological order
    while (stack.empty() == false)
    {
        // Get the next vertex from topological order
        int u = (int)stack.pop();

        // Update distances of all adjacent vertices
        Iterator<AdjListNode> it;
        if (dist[u] != INF)
        {
            it = adj[u].iterator();
            while (it.hasNext())

```

```

        {
            AdjListNode i= it.next();
            if (dist[i.getV()] > dist[u] + i.getWeight())
                dist[i.getV()] = dist[u] + i.getWeight();
        }
    }

    // Print the calculated shortest distances
    for (int i = 0; i < V; i++)
    {
        if (dist[i] == INF)
            System.out.print( "INF ");
        else
            System.out.print( dist[i] + " ");
    }
}

// Method to create a new graph instance through an object
// of ShortestPath class.
Graph newGraph(int number)
{
    return new Graph(number);
}

public static void main(String args[])
{
    // Create a graph given in the above diagram. Here vertex
    // numbers are 0, 1, 2, 3, 4, 5 with following mappings:
    // 0=r, 1=s, 2=t, 3=x, 4=y, 5=z
    ShortestPath t = new ShortestPath();
    Graph g = t.newGraph(6);
    g.addEdge(0, 1, 5);
    g.addEdge(0, 2, 3);
    g.addEdge(1, 3, 6);
    g.addEdge(1, 2, 2);
    g.addEdge(2, 4, 4);
    g.addEdge(2, 5, 2);
    g.addEdge(2, 3, 7);
    g.addEdge(3, 4, -1);
    g.addEdge(4, 5, -2);

    int s = 1;
    System.out.println("Following are shortest distances "+
                       "from source " + s );
    g.shortestPath(s);
}

```

```
}
```

//This code is contributed by Aakash Hasija

Python

```
# Python program to find single source shortest paths
# for Directed Acyclic Graphs Complexity :O(V(V+E))
from collections import defaultdict

# Graph is represented using adjacency list. Every
# node of adjacency list contains vertex number of
# the vertex to which edge connects. It also contains
# weight of the edge
class Graph:
    def __init__(self,vertices):

        self.V = vertices # No. of vertices

        # dictionary containing adjacency List
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v,w):
        self.graph[u].append((v,w))

    # A recursive function used by shortestPath
    def topologicalSortUtil(self,v,visited,stack):

        # Mark the current node as visited.
        visited[v] = True

        # Recur for all the vertices adjacent to this vertex
        if v in self.graph.keys():
            for node,weight in self.graph[v]:
                if visited[node] == False:
                    self.topologicalSortUtil(node,visited,stack)

        # Push current vertex to stack which stores topological sort
        stack.append(v)

    ''' The function to find shortest paths from given vertex.
    It uses recursive topologicalSortUtil() to get topological
    sorting of given graph.'''
    def shortestPath(self, s):

        # Mark all the vertices as not visited
```

```

visited = [False]*self.V
stack = []

# Call the recursive helper function to store Topological
# Sort starting from source vertice
for i in range(self.V):
    if visited[i] == False:
        self.topologicalSortUtil(s,visited,stack)

# Initialize distances to all vertices as infinite and
# distance to source as 0
dist = [float("Inf")] * (self.V)
dist[s] = 0

# Process vertices in topological order
while stack:

    # Get the next vertex from topological order
    i = stack.pop()

    # Update distances of all adjacent vertices
    for node,weight in self.graph[i]:
        if dist[node] > dist[i] + weight:
            dist[node] = dist[i] + weight

    # Print the calculated shortest distances
    for i in range(self.V):
        print ("%d" %dist[i]) if dist[i] != float("Inf") else "Inf" ,

g = Graph(6)
g.addEdge(0, 1, 5)
g.addEdge(0, 2, 3)
g.addEdge(1, 3, 6)
g.addEdge(1, 2, 2)
g.addEdge(2, 4, 4)
g.addEdge(2, 5, 2)
g.addEdge(2, 3, 7)
g.addEdge(3, 4, -1)
g.addEdge(4, 5, -2)

# source = 1
s = 1

print ("Following are shortest distances from source %d " % s)
g.shortestPath(s)

# This code is contributed by Neelam Yadav

```

Output:

```
Following are shortest distances from source 1
INF 0 2 6 5 3
```

Time Complexity: Time complexity of topological sorting is $O(V+E)$. After finding topological order, the algorithm process all vertices and for every vertex, it runs a loop for all adjacent vertices. Total adjacent vertices in a graph is $O(E)$. So the inner loop runs $O(V+E)$ times. Therefore, overall time complexity of this algorithm is $O(V+E)$.

References:

<http://www.utdallas.edu/~sizheng/CS4349.d/l-notes.d/L17.pdf>

Source

<https://www.geeksforgeeks.org/shortest-path-for-directed-acyclic-graphs/>

Chapter 28

Some interesting shortest path questions,

Some interesting shortest path questions Set 1 - GeeksforGeeks

Question 1: *Given a directed weighted graph. You are also given the shortest path from a source vertex ‘s’ to a destination vertex ‘t’. If weight of every edge is increased by 10 units, does the shortest path remain same in the modified graph?*

The shortest path may change. The reason is, there may be different number of edges in different paths from s to t. For example, let shortest path be of weight 15 and has 5 edges. Let there be another path with 2 edges and total weight 25. The weight of the shortest path is increased by 5×10 and becomes $15 + 50$. Weight of the other path is increased by 2×10 and becomes $25 + 20$. So the shortest path changes to the other path with weight as 45.

Question 2: *This is similar to above question. Does the shortest path change when weights of all edges are multiplied by 10?*

If we multiply all edge weights by 10, the shortest path doesn’t change. The reason is simple, weights of all paths from s to t get multiplied by same amount. The number of edges on a path doesn’t matter. It is like changing unit of weights.

Question 3: *Given a directed graph where every edge has weight as either 1 or 2, find the shortest path from a given source vertex ‘s’ to a given destination vertex ‘t’. Expected time complexity is $O(V+E)$.*

If we apply [Dijkstra’s shortest path algorithm](#), we can get a shortest path in $O(E + V \log V)$ time. How to do it in $O(V+E)$ time? The idea is to use [BFS](#). One important observation about [BFS](#)s, the path used in BFS always has least number of edges between any two vertices. So if all edges are of same weight, we can use BFS to find the shortest path. For this problem, we can modify the graph and split all edges of weight 2 into two edges of weight 1 each. In the modified graph, we can use BFS to find the shortest path. How is this approach $O(V+E)$? In worst case, all edges are of weight 2 and we need to do $O(E)$ operations to split all edges, so the time complexity becomes $O(E) + O(V+E)$ which is $O(V+E)$.

Question 4: *Given a directed acyclic weighted graph, how to find the shortest path from a source s to a destination t in $O(V+E)$ time?*

See: [Shortest Path in Directed Acyclic Graph](#)

More Questions See following links for more questions.

<http://algs4.cs.princeton.edu/44sp/>

<https://www.geeksforgeeks.org/algorithms-gq/graph-shortest-paths-gq/>

Source

<https://www.geeksforgeeks.org/interesting-shortest-path-questions-set-1/>

Chapter 29

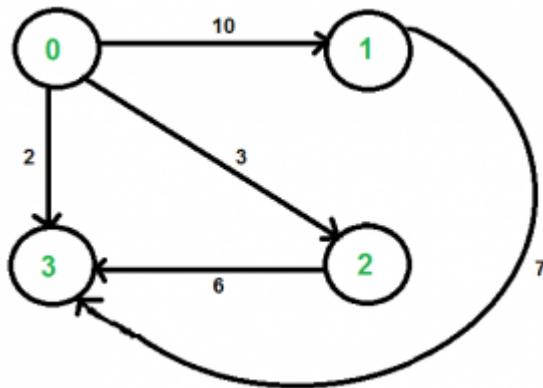
Shortest path with exactly k edges in a directed and weighted graph

Shortest path with exactly k edges in a directed and weighted graph - GeeksforGeeks

Given a directed and two vertices ‘u’ and ‘v’ in it, find shortest path from ‘u’ to ‘v’ with exactly k edges on the path.

The graph is given as [adjacency matrix representation](#) where value of $\text{graph}[i][j]$ indicates the weight of an edge from vertex i to vertex j and a value INF(infinite) indicates no edge from i to j.

For example consider the following graph. Let source ‘u’ be vertex 0, destination ‘v’ be 3 and k be 2. There are two walks of length 2, the walks are {0, 2, 3} and {0, 1, 3}. The shortest among the two is {0, 2, 3} and weight of path is $3+6 = 9$.



The idea is to browse through all paths of length k from u to v using the approach discussed in the [previous post](#) and return weight of the shortest path. A [simple solution](#) is to start

from u, go to all adjacent vertices and recur for adjacent vertices with k as k-1, source as adjacent vertex and destination as v. Following are C++ and Java implementations of this simple solution.

C++

```
// C++ program to find shortest path with exactly k edges
#include <iostream>
#include <climits>
using namespace std;

// Define number of vertices in the graph and infinite value
#define V 4
#define INF INT_MAX

// A naive recursive function to count walks from u to v with k edges
int shortestPath(int graph[][][V], int u, int v, int k)
{
    // Base cases
    if (k == 0 && u == v) return 0;
    if (k == 1 && graph[u][v] != INF) return graph[u][v];
    if (k <= 0) return INF;

    // Initialize result
    int res = INF;

    // Go to all adjacents of u and recur
    for (int i = 0; i < V; i++)
    {
        if (graph[u][i] != INF && u != i && v != i)
        {
            int rec_res = shortestPath(graph, i, v, k-1);
            if (rec_res != INF)
                res = min(res, graph[u][i] + rec_res);
        }
    }
    return res;
}

// driver program to test above function
int main()
{
    /* Let us create the graph shown in above diagram*/
    int graph[V][V] = { {0, 10, 3, 2},
                        {INF, 0, INF, 7},
                        {INF, INF, 0, 6},
                        {INF, INF, INF, 0}
                      };
    int u = 0, v = 3, k = 2;
```

```

cout << "Weight of the shortest path is " <<
      shortestPath(graph, u, v, k);
return 0;
}

```

Java

```

// Dynamic Programming based Java program to find shortest path
// with exactly k edges
import java.util.*;
import java.lang.*;
import java.io.*;

class ShortestPath
{
    // Define number of vertices in the graph and infinite value
    static final int V = 4;
    static final int INF = Integer.MAX_VALUE;

    // A naive recursive function to count walks from u to v
    // with k edges
    int shortestPath(int graph[][] , int u, int v, int k)
    {
        // Base cases
        if (k == 0 && u == v)           return 0;
        if (k == 1 && graph[u][v] != INF) return graph[u][v];
        if (k <= 0)                     return INF;

        // Initialize result
        int res = INF;

        // Go to all adjacents of u and recur
        for (int i = 0; i < V; i++)
        {
            if (graph[u][i] != INF && u != i && v != i)
            {
                int rec_res = shortestPath(graph, i, v, k-1);
                if (rec_res != INF)
                    res = Math.min(res, graph[u][i] + rec_res);
            }
        }
        return res;
    }

    public static void main (String[] args)
    {
        /* Let us create the graph shown in above diagram*/
        int graph[][] = new int[][]{ {0, 10, 3, 2},

```

```

        {INF, 0, INF, 7},
        {INF, INF, 0, 6},
        {INF, INF, INF, 0}
    };
ShortestPath t = new ShortestPath();
int u = 0, v = 3, k = 2;
System.out.println("Weight of the shortest path is "+
                    t.shortestPath(graph, u, v, k));
}
}

```

Output:

```
Weight of the shortest path is 9
```

The worst case time complexity of the above function is $O(V^k)$ where V is the number of vertices in the given graph. We can simply analyze the time complexity by drawing recursion tree. The worst occurs for a complete graph. In worst case, every internal node of recursion tree would have exactly V children.

We can optimize the above solution using **Dynamic Programming**. The idea is to build a 3D table where first dimension is source, second dimension is destination, third dimension is number of edges from source to destination, and the value is count of walks. Like other **Dynamic Programming problems**, we fill the 3D table in bottom up manner.

C++

```

// Dynamic Programming based C++ program to find shortest path with
// exactly k edges
#include <iostream>
#include <climits>
using namespace std;

// Define number of vertices in the graph and infinite value
#define V 4
#define INF INT_MAX

// A Dynamic programming based function to find the shortest path from
// u to v with exactly k edges.
int shortestPath(int graph[][V], int u, int v, int k)
{
    // Table to be filled up using DP. The value sp[i][j][e] will store
    // weight of the shortest path from i to j with exactly k edges
    int sp[V][V][k+1];

    // Loop for number of edges from 0 to k
    for (int e = 0; e <= k; e++)
    {
        for (int i = 0; i < V; i++) // for source

```

```

{
    for (int j = 0; j < V; j++) // for destination
    {
        // initialize value
        sp[i][j][e] = INF;

        // from base cases
        if (e == 0 && i == j)
            sp[i][j][e] = 0;
        if (e == 1 && graph[i][j] != INF)
            sp[i][j][e] = graph[i][j];

        //go to adjacent only when number of edges is more than 1
        if (e > 1)
        {
            for (int a = 0; a < V; a++)
            {
                // There should be an edge from i to a and a
                // should not be same as either i or j
                if (graph[i][a] != INF && i != a &&
                    j != a && sp[a][j][e-1] != INF)
                    sp[i][j][e] = min(sp[i][j][e], graph[i][a] +
                        sp[a][j][e-1]);
            }
        }
    }
    return sp[u][v][k];
}

// driver program to test above function
int main()
{
    /* Let us create the graph shown in above diagram*/
    int graph[V][V] = { {0, 10, 3, 2},
                        {INF, 0, INF, 7},
                        {INF, INF, 0, 6},
                        {INF, INF, INF, 0}
                      };
    int u = 0, v = 3, k = 2;
    cout << shortestPath(graph, u, v, k);
    return 0;
}

```

Java

```
// Dynamic Programming based Java program to find shortest path with
```

```

// exactly k edges
import java.util.*;
import java.lang.*;
import java.io.*;

class ShortestPath
{
    // Define number of vertices in the graph and infinite value
    static final int V = 4;
    static final int INF = Integer.MAX_VALUE;

    // A Dynamic programming based function to find the shortest path
    // from u to v with exactly k edges.
    int shortestPath(int graph[][][], int u, int v, int k)
    {
        // Table to be filled up using DP. The value sp[i][j][e] will
        // store weight of the shortest path from i to j with exactly
        // k edges
        int sp[][][] = new int[V][V][k+1];

        // Loop for number of edges from 0 to k
        for (int e = 0; e <= k; e++)
        {
            for (int i = 0; i < V; i++) // for source
            {
                for (int j = 0; j < V; j++) // for destination
                {
                    // initialize value
                    sp[i][j][e] = INF;

                    // from base cases
                    if (e == 0 && i == j)
                        sp[i][j][e] = 0;
                    if (e == 1 && graph[i][j] != INF)
                        sp[i][j][e] = graph[i][j];

                    // go to adjacent only when number of edges is
                    // more than 1
                    if (e > 1)
                    {
                        for (int a = 0; a < V; a++)
                        {
                            // There should be an edge from i to a and
                            // a should not be same as either i or j
                            if (graph[i][a] != INF && i != a &&
                                j != a && sp[a][j][e-1] != INF)
                                sp[i][j][e] = Math.min(sp[i][j][e],
                                                       graph[i][a] + sp[a][j][e-1]);
                        }
                    }
                }
            }
        }
    }
}

```

```
        }
    }
}
}
return sp[u][v][k];
}

public static void main (String[] args)
{
    /* Let us create the graph shown in above diagram*/
    int graph[][] = new int[][]{{0, 10, 3, 2},
                                {INF, 0, INF, 7},
                                {INF, INF, 0, 6},
                                {INF, INF, INF, 0}
                               };
    ShortestPath t = new ShortestPath();
    int u = 0, v = 3, k = 2;
    System.out.println("Weight of the shortest path is "+
                       t.shortestPath(graph, u, v, k));
}
//This code is contributed by Aakash Hasija
```

Output:

```
Weight of the shortest path is 9
```

Time complexity of the above DP based solution is $O(V^3K)$ which is much better than the naive solution.

This article is contributed by **Abhishek**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

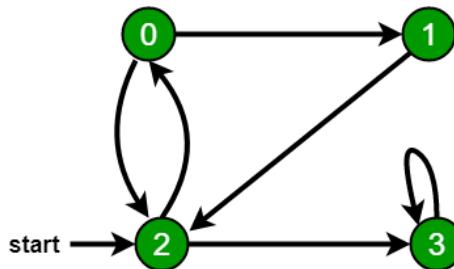
<https://www.geeksforgeeks.org/shortest-path-exactly-k-edges-directed-weighted-graph/>

Chapter 30

Find if there is a path between two vertices in a directed graph

Find if there is a path between two vertices in a directed graph - GeeksforGeeks

Given a Directed Graph and two vertices in it, check whether there is a path from the first given vertex to second. For example, in the following graph, there is a path from vertex 1 to 3. As another example, there is no path from 3 to 0.



We can either use [Breadth First Search \(BFS\)](#) or [Depth First Search \(DFS\)](#) to find path between two vertices. Take the first vertex as source in BFS (or DFS), follow the standard BFS (or DFS). If we see the second vertex in our traversal, then return true. Else return false.

Following are C++, Java and Python codes that use BFS for finding reachability of second vertex from first vertex.

C++

```
// C++ program to check if there is exist a path between two vertices
// of a graph.
#include<iostream>
#include <list>
using namespace std;
```

```
// This class represents a directed graph using adjacency list
// representation
class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // Pointer to an array containing adjacency lists
public:
    Graph(int V); // Constructor
    void addEdge(int v, int w); // function to add an edge to graph
    bool isReachable(int s, int d);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

// A BFS based function to check whether d is reachable from s.
bool Graph::isReachable(int s, int d)
{
    // Base case
    if (s == d)
        return true;

    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // it will be used to get all adjacent vertices of a vertex
    list<int>::iterator i;

    while (!queue.empty())
    {
        // Dequeue a vertex from queue and print it
```

```
s = queue.front();
queue.pop_front();

// Get all adjacent vertices of the dequeued vertex s
// If a adjacent has not been visited, then mark it visited
// and enqueue it
for (i = adj[s].begin(); i != adj[s].end(); ++i)
{
    // If this adjacent node is the destination node, then
    // return true
    if (*i == d)
        return true;

    // Else, continue to do BFS
    if (!visited[*i])
    {
        visited[*i] = true;
        queue.push_back(*i);
    }
}

// If BFS is complete without visiting d
return false;
}

// Driver program to test methods of graph class
int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    int u = 1, v = 3;
    if(g.isReachable(u, v))
        cout<< "\n There is a path from " << u << " to " << v;
    else
        cout<< "\n There is no path from " << u << " to " << v;

    u = 3, v = 1;
    if(g.isReachable(u, v))
        cout<< "\n There is a path from " << u << " to " << v;
    else
```

```
    cout<< "\n There is no path from " << u << " to " << v;  
  
    return 0;  
}
```

Java

```
// Java program to check if there is exist a path between two vertices  
// of a graph.  
import java.io.*;  
import java.util.*;  
import java.util.LinkedList;  
  
// This class represents a directed graph using adjacency list  
// representation  
class Graph  
{  
    private int V; // No. of vertices  
    private LinkedList<Integer> adj[]; //Adjacency List  
  
    //Constructor  
    Graph(int v)  
    {  
        V = v;  
        adj = new LinkedList[v];  
        for (int i=0; i<v; ++i)  
            adj[i] = new LinkedList();  
    }  
  
    //Function to add an edge into the graph  
    void addEdge(int v,int w) { adj[v].add(w); }  
  
    //prints BFS traversal from a given source s  
    Boolean isReachable(int s, int d)  
    {  
        LinkedList<Integer>temp;  
  
        // Mark all the vertices as not visited(By default set  
        // as false)  
        boolean visited[] = new boolean[V];  
  
        // Create a queue for BFS  
        LinkedList<Integer> queue = new LinkedList<Integer>();  
  
        // Mark the current node as visited and enqueue it  
        visited[s]=true;  
        queue.add(s);
```

```
// 'i' will be used to get all adjacent vertices of a vertex
Iterator<Integer> i;
while (queue.size()!=0)
{
    // Dequeue a vertex from queue and print it
    s = queue.poll();

    int n;
    i = adj[s].listIterator();

    // Get all adjacent vertices of the dequeued vertex s
    // If a adjacent has not been visited, then mark it
    // visited and enqueue it
    while (i.hasNext())
    {
        n = i.next();

        // If this adjacent node is the destination node,
        // then return true
        if (n==d)
            return true;

        // Else, continue to do BFS
        if (!visited[n])
        {
            visited[n] = true;
            queue.add(n);
        }
    }
}

// If BFS is complete without visited d
return false;
}

// Driver method
public static void main(String args[])
{
    // Create a graph given in the above diagram
    Graph g = new Graph(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    int u = 1;
```

```
int v = 3;
if (g.isReachable(u, v))
    System.out.println("There is a path from " + u +" to " + v);
else
    System.out.println("There is no path from " + u +" to " + v);;

u = 3;
v = 1;
if (g.isReachable(u, v))
    System.out.println("There is a path from " + u +" to " + v);
else
    System.out.println("There is no path from " + u +" to " + v);;
}

// This code is contributed by Aakash Hasija
```

Python

```
# program to check if there is exist a path between two vertices
# of a graph

from collections import defaultdict

#This class represents a directed graph using adjacency list representation
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices
        self.graph = defaultdict(list) # default dictionary to store graph

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # Use BFS to check path between s and d
    def isReachable(self, s, d):
        # Mark all the vertices as not visited
        visited =[False]*(self.V)

        # Create a queue for BFS
        queue=[]

        # Mark the source node as visited and enqueue it
        queue.append(s)
        visited[s] = True

        while queue:
```

```
#Dequeue a vertex from queue
n = queue.pop(0)

# If this adjacent node is the destination node,
# then return true
if n == d:
    return True

# Else, continue to do BFS
for i in self.graph[n]:
    if visited[i] == False:
        queue.append(i)
        visited[i] = True
# If BFS is complete without visited d
return False

# Create a graph given in the above diagram
g = Graph(4)
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

u = 1; v = 3

if g.isReachable(u, v):
    print("There is a path from %d to %d" % (u,v))
else :
    print("There is no path from %d to %d" % (u,v))

u = 3; v = 1
if g.isReachable(u, v) :
    print("There is a path from %d to %d" % (u,v))
else :
    print("There is no path from %d to %d" % (u,v))

#This code is contributed by Neelam Yadav
```

Output:

```
There is a path from 1 to 3
There is no path from 3 to 1
```

As an exercise, try an extended version of the problem where the complete path between two vertices is also needed.

Source

<https://www.geeksforgeeks.org/find-if-there-is-a-path-between-two-vertices-in-a-given-graph/>

Chapter 31

Find if the strings can be chained to form a circle

Find if an array of strings can be chained to form a circle Set 1 - GeeksforGeeks

Given an array of strings, find if the given strings can be chained to form a circle. A string X can be put before another string Y in circle if the last character of X is same as first character of Y.

Examples:

Input: arr[] = {"geek", "king"}
Output: Yes, the given strings can be chained.
Note that the last character of first string is same as first character of second string and vice versa is also true.

Input: arr[] = {"for", "geek", "rig", "kaf"}
Output: Yes, the given strings can be chained.
The strings can be chained as "for", "rig", "geek" and "kaf"

Input: arr[] = {"aab", "bac", "aaa", "cda"}
Output: Yes, the given strings can be chained.
The strings can be chained as "aaa", "aab", "bac" and "cda"

Input: arr[] = {"aaa", "bbb", "baa", "aab"};
Output: Yes, the given strings can be chained.
The strings can be chained as "aaa", "aab", "bbb" and "baa"

Input: arr[] = {"aaa"};
Output: Yes

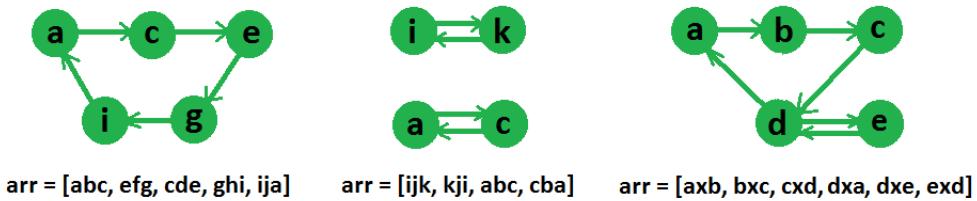
Input: arr[] = {"aaa", "bbb"};
Output: No

Input : arr[] = ["abc", "efg", "cde", "ghi", "ija"]
Output : Yes
These strings can be reordered as, "abc", "cde", "efg",
"ghi", "ija"

Input : arr[] = ["ijk", "kji", "abc", "cba"]
Output : No

The idea is to create a directed graph of all characters and then find if there is an [eulerian circuit](#) in the graph or not.

Graph representation of some string arrays are given in below diagram,



If there is an [eulerian circuit](#), then chain can be formed, otherwise not.

Note that a directed graph has [eulerian circuit](#) only if in degree and out degree of every vertex is same, and all non-zero degree vertices form a single strongly connected component.

Following are detailed steps of the algorithm.

- 1) Create a directed graph g with number of vertices equal to the size of alphabet. We have created a graph with 26 vertices in the below program.
- 2) Do following for every string in the given array of strings.
....a) Add an edge from first character to last character of the given graph.
- 3) If the created graph has [eulerian circuit](#), then return true, else return false.

Following are C++ and Python implementations of the above algorithm.

C/C++

```
// A C++ program to check if a given directed graph is Eulerian or not
#include<iostream>
#include <list>
#define CHARS 26
using namespace std;
```

```
// A class that represents an undirected graph
class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // A dynamic array of adjacency lists
    int *in;
public:
    // Constructor and destructor
    Graph(int V);
    ~Graph() { delete [] adj; delete [] in; }

    // function to add an edge to graph
    void addEdge(int v, int w) { adj[v].push_back(w); (in[w])++; }

    // Method to check if this graph is Eulerian or not
    bool isEulerianCycle();

    // Method to check if all non-zero degree vertices are connected
    bool isSC();

    // Function to do DFS starting from v. Used in isConnected();
    void DFSUtil(int v, bool visited[]);

    Graph getTranspose();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
    in = new int[V];
    for (int i = 0; i < V; i++)
        in[i] = 0;
}

/* This function returns true if the directed graph has an eulerian
cycle, otherwise returns false */
bool Graph::isEulerianCycle()
{
    // Check if all non-zero degree vertices are connected
    if (isSC() == false)
        return false;

    // Check if in degree and out degree of every vertex is same
    for (int i = 0; i < V; i++)
        if (adj[i].size() != in[i])
            return false;
```

```
        return true;
    }

// A recursive function to do DFS starting from v
void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

// Function that returns reverse (or transpose) of this graph
// This function is needed in isSC()
Graph Graph::getTranspose()
{
    Graph g(V);
    for (int v = 0; v < V; v++)
    {
        // Recur for all the vertices adjacent to this vertex
        list<int>::iterator i;
        for(i = adj[v].begin(); i != adj[v].end(); ++i)
        {
            g.adj[*i].push_back(v);
            (g.in[v])++;
        }
    }
    return g;
}

// This function returns true if all non-zero degree vertices of
// graph are strongly connected. Please refer
// https://www.geeksforgeeks.org/connectivity-in-a-directed-graph/
bool Graph::isSC()
{
    // Mark all the vertices as not visited (For first DFS)
    bool visited[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Find the first vertex with non-zero degree
    int n;
    for (n = 0; n < V; n++)
```

```
if (adj[n].size() > 0)
    break;

// Do DFS traversal starting from first non zero degree vertex.
DFSUtil(n, visited);

// If DFS traversal doesn't visit all vertices, then return false.
for (int i = 0; i < V; i++)
    if (adj[i].size() > 0 && visited[i] == false)
        return false;

// Create a reversed graph
Graph gr = getTranspose();

// Mark all the vertices as not visited (For second DFS)
for (int i = 0; i < V; i++)
    visited[i] = false;

// Do DFS for reversed graph starting from first vertex.
// Starting Vertex must be same starting point of first DFS
gr.DFSUtil(n, visited);

// If all vertices are not visited in second DFS, then
// return false
for (int i = 0; i < V; i++)
    if (adj[i].size() > 0 && visited[i] == false)
        return false;

return true;
}

// This function takes an of strings and returns true
// if the given array of strings can be chained to
// form cycle
bool canBeChained(string arr[], int n)
{
    // Create a graph with 'aplha' edges
    Graph g(CHARS);

    // Create an edge from first character to last character
    // of every string
    for (int i = 0; i < n; i++)
    {
        string s = arr[i];
        g.addEdge(s[0]-'a', s[s.length()-1]-'a');
    }

    // The given array of strings can be chained if there
```

```
// is an eulerian cycle in the created graph
return g.isEulerianCycle();
}

// Driver program to test above functions
int main()
{
    string arr1[] = {"for", "geek", "rig", "kaf"};
    int n1 = sizeof(arr1)/sizeof(arr1[0]);
    canBeChained(arr1, n1)? cout << "Can be chained n" :
                                cout << "Can't be chained n";

    string arr2[] = {"aab", "abb"};
    int n2 = sizeof(arr2)/sizeof(arr2[0]);
    canBeChained(arr2, n2)? cout << "Can be chained n" :
                                cout << "Can't be chained n";

    return 0;
}
```

Python

```
# Python program to check if a given directed graph is Eulerian or not
CHARS = 26

# A class that represents an undirected graph
class Graph(object):
    def __init__(self, V):
        self.V = V          # No. of vertices
        self.adj = [[] for x in xrange(V)]  # a dynamic array
        self.inp = [0] * V

    # function to add an edge to graph
    def addEdge(self, v, w):
        self.adj[v].append(w)
        self.inp[w]+=1

    # Method to check if this graph is Eulerian or not
    def isSC(self):
        # Mark all the vertices as not visited (For first DFS)
        visited = [False] * self.V

        # Find the first vertex with non-zero degree
        n = 0
        for n in xrange(self.V):
            if len(self.adj[n]) > 0:
                break
```

```
# Do DFS traversal starting from first non zero degree vertex.
self.DFSUtil(n, visited)

# If DFS traversal doesn't visit all vertices, then return false.
for i in xrange(self.V):
    if len(self.adj[i]) > 0 and visited[i] == False:
        return False

# Create a reversed graph
gr = self.getTranspose()

# Mark all the vertices as not visited (For second DFS)
for i in xrange(self.V):
    visited[i] = False

# Do DFS for reversed graph starting from first vertex.
# Starting Vertex must be same starting point of first DFS
gr.DFSUtil(n, visited)

# If all vertices are not visited in second DFS, then
# return false
for i in xrange(self.V):
    if len(self.adj[i]) > 0 and visited[i] == False:
        return False

return True

# This function returns true if the directed graph has an eulerian
# cycle, otherwise returns false
def isEulerianCycle(self):

    # Check if all non-zero degree vertices are connected
    if self.isSC() == False:
        return False

    # Check if in degree and out degree of every vertex is same
    for i in xrange(self.V):
        if len(self.adj[i]) != self.inp[i]:
            return False

    return True

# A recursive function to do DFS starting from v
def DFSUtil(self, v, visited):

    # Mark the current node as visited and print it
    visited[v] = True
```

```

# Recur for all the vertices adjacent to this vertex
for i in xrange(len(self.adj[v])):
    if not visited[self.adj[v][i]]:
        self.DFSUtil(self.adj[v][i], visited)

# Function that returns reverse (or transpose) of this graph
# This function is needed in isSC()
def getTranspose(self):
    g = Graph(self.V)
    for v in xrange(self.V):
        # Recur for all the vertices adjacent to this vertex
        for i in xrange(len(self.adj[v])):
            g.adj[self.adj[v][i]].append(v)
            g.inp[v]+=1
    return g

# This function takes an array of strings and returns true
# if the given array of strings can be chained to
# form cycle
def canBeChained(arr, n):

    # Create a graph with 'alpha' edges
    g = Graph(CHARS)

    # Create an edge from first character to last character
    # of every string
    for i in xrange(n):
        s = arr[i]
        g.addEdge(ord(s[0])-ord('a'), ord(s[len(s)-1])-ord('a'))

    # The given array of strings can be chained if there
    # is an eulerian cycle in the created graph
    return g.isEulerianCycle()

# Driver program
arr1 = ["for", "geek", "rig", "kaf"]
n1 = len(arr1)
if canBeChained(arr1, n1):
    print "Can be chained"
else:
    print "Cant be chained"

arr2 = ["aab", "abb"]
n2 = len(arr2)
if canBeChained(arr2, n2):
    print "Can be chained"
else:
    print "Can't be chained"

```

```
# This code is contributed by BHAVYA JAIN
```

Output:

```
Can be chained  
Can't be chained
```

[Find if an array of strings can be chained to form a circle Set 2](#)

This article is contributed by **Piyush Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

<https://www.geeksforgeeks.org/given-array-strings-find-strings-can-chained-form-circle/>

Chapter 32

Find same contacts in a list of contacts

Find same contacts in a list of contacts - GeeksforGeeks

Given a list of contacts containing username, email and phone number in any order. Identify the same contacts (i.e., same person having many different contacts) and output the same contacts together.

Notes:

- 1) A contact can store its three fields in any order, i.e., phone number can appear before username or username can appear before phone number.
- 2) Two contacts are same if they have either same username or email or phone number.

Example:

```
Input: contact[] =  
    { {"Gaurav", "gaurav@gmail.com", "gaurav@gfgQA.com"},  
     { "Lucky", "lucky@gmail.com", "+1234567"},  
     { "gaurav123", "+5412312", "gaurav123@skype.com"},  
     { "gaurav1993", "+5412312", "gaurav@gfgQA.com"}  
    }  
Output:  
    0 2 3  
    1  
contact[2] is same as contact[3] because they both have same  
contact number.  
contact[0] is same as contact[3] because they both have same  
e-mail address.  
Therefore, contact[0] and contact[2] are also same.
```

We strongly recommend you to minimize your browser and try this yourself first.

Input is basically an array of structures. A structure contains three fields such that any field can represent any detail about a contact.

The idea is to first create a graph of contacts using given array. In the graph, there is an edge between vertex i to vertex j if they both have either same username or same email or same phone number. Once the graph is constructed, the task reduces to finding [connected components in an undirected graph](#). We can find connected components either by doing DFS or BFS starting from every unvisited vertex. In below code, DFS is used.

Below is C++ implementation of this idea.

```
// A C++ program to find same contacts in a list of contacts
#include<bits/stdc++.h>
using namespace std;

// Structure for storing contact details.
struct contact
{
    string field1, field2, field3;
};

// A utility function to fill entries in adjacency matrix
// representation of graph
void buildGraph(contact arr[], int n, int *mat[])
{
    // Initialize the adjacency matrix
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            mat[i][j] = 0;

    // Traverse through all contacts
    for (int i = 0; i < n; i++) {

        // Add mat from i to j and vice versa, if possible.
        // Since length of each contact field is at max some
        // constant. (say 30) so body execution of this for
        // loop takes constant time.
        for (int j = i+1; j < n; j++)
            if (arr[i].field1 == arr[j].field1 ||
                arr[i].field1 == arr[j].field2 ||
                arr[i].field1 == arr[j].field3 ||
                arr[i].field2 == arr[j].field1 ||
                arr[i].field2 == arr[j].field2 ||
                arr[i].field2 == arr[j].field3 ||
                arr[i].field3 == arr[j].field1 ||
                arr[i].field3 == arr[j].field2 ||
                arr[i].field3 == arr[j].field3)
            {
                mat[i][j] = 1;
            }
    }
}
```

```
        mat[j][i] = 1;
        break;
    }
}

// A recursive function to perform DFS with vertex i as source
void DFSvisit(int i, int *mat[], bool visited[], vector<int>& sol, int n)
{
    visited[i] = true;
    sol.push_back(i);

    for (int j = 0; j < n; j++)
        if (mat[i][j] && !visited[j])
            DFSvisit(j, mat, visited, sol, n);
}

// Finds similar contacts in an array of contacts
void findSameContacts(contact arr[], int n)
{
    // vector for storing the solution
    vector<int> sol;

    // Declare 2D adjacency matrix for mats
    int **mat = new int*[n];

    for (int i = 0; i < n; i++)
        mat[i] = new int[n];

    // visited array to keep track of visited nodes
    bool visited[n];
    memset(visited, 0, sizeof(visited));

    // Fill adjacency matrix
    buildGraph(arr, n, mat);

    // Since, we made a graph with contacts as nodes with fields as links.
    // two nodes are linked if they represent the same person.
    // so, total number of connected components and nodes in each component
    // will be our answer.
    for (int i = 0; i < n; i++)
    {
        if (!visited[i])
        {
            DFSvisit(i, mat, visited, sol, n);

            // Add delimiter to separate nodes of one component from other.
            sol.push_back(-1);
        }
    }
}
```

```
        }
    }

    // Print the solution
    for (int i = 0; i < sol.size(); i++)
        if (sol[i] == -1) cout << endl;
        else cout << sol[i] << " ";
}

// Drive program
int main()
{
    contact arr[] = {{"Gaurav", "gaurav@gmail.com", "gaurav@gfgQA.com"},  

                      {"Lucky", "lucky@gmail.com", "+1234567"},  

                      {"gaurav123", "+5412312", "gaurav123@skype.com"},  

                      {"gaurav1993", "+5412312", "gaurav@gfgQA.com"},  

                      {"raja", "+2231210", "raja@gfg.com"},  

                      {"bahubali", "+878312", "raja"}  

    };

    int n = sizeof arr / sizeof arr[0];
    findSameContacts(arr, n);
    return 0;
}
```

Output:

```
0 3 2  
1  
4 5
```

Time complexity: $O(n^2)$ where n is number of contacts.

Thanks to [Gaurav Ahirwar](#) for above solution.

Source

<https://www.geeksforgeeks.org/find-same-contacts-in-a-list-of-contacts/>

Chapter 33

Find the minimum cost to reach destination using a train

Find the minimum cost to reach destination using a train - GeeksforGeeks

There are N stations on route of a train. The train goes from station 0 to N-1. The ticket cost for all pair of stations (i, j) is given where j is greater than i. Find the minimum cost to reach the destination.

Consider the following example:

Input:

```
cost[N][N] = { {0, 15, 80, 90},  
               {INF, 0, 40, 50},  
               {INF, INF, 0, 70},  
               {INF, INF, INF, 0}  
             };
```

There are 4 stations and `cost[i][j]` indicates cost to reach j from i. The entries where $j < i$ are meaningless.

Output:

The minimum cost is 65

The minimum cost can be obtained by first going to station 1 from 0. Then from station 1 to station 3.

We strongly recommend to minimize your browser and try this yourself first.

The minimum cost to reach N-1 from 0 can be recursively written as following:

```
minCost(0, N-1) = MIN { cost[0][n-1],  
                         cost[0][1] + minCost(1, N-1),
```

```
minCost(0, 2) + minCost(2, N-1),  
.....,  
minCost(0, N-2) + cost[N-2][n-1] }
```

The following is the implementation of above recursive formula.

C++

```
// A naive recursive solution to find min cost path from station 0  
// to station N-1  
#include<iostream>  
#include<climits>  
using namespace std;  
  
// infinite value  
#define INF INT_MAX  
  
// Number of stations  
#define N 4  
  
// A recursive function to find the shortest path from  
// source 's' to destination 'd'.  
int minCostRec(int cost[][][N], int s, int d)  
{  
    // If source is same as destination  
    // or destination is next to source  
    if (s == d || s+1 == d)  
        return cost[s][d];  
  
    // Initialize min cost as direct ticket from  
    // source 's' to destination 'd'.  
    int min = cost[s][d];  
  
    // Try every intermediate vertex to find minimum  
    for (int i = s+1; i < d; i++)  
    {  
        int c = minCostRec(cost, s, i) +  
                minCostRec(cost, i, d);  
        if (c < min)  
            min = c;  
    }  
    return min;  
}  
  
// This function returns the smallest possible cost to  
// reach station N-1 from station 0. This function mainly  
// uses minCostRec().  
int minCost(int cost[] [N])
```

```
{  
    return minCostRec(cost, 0, N-1);  
}  
  
// Driver program to test above function  
int main()  
{  
    int cost[N][N] = { {0, 15, 80, 90},  
                      {INF, 0, 40, 50},  
                      {INF, INF, 0, 70},  
                      {INF, INF, INF, 0}  
                    };  
    cout << "The Minimum cost to reach station "  
         << N << " is " << minCost(cost);  
    return 0;  
}
```

Java

```
// A naive recursive solution to find min cost path from station 0  
// to station N-1  
class shortest_path  
{  
  
    static int INF = Integer.MAX_VALUE, N = 4;  
    // A recursive function to find the shortest path from  
    // source 's' to destination 'd'.  
    static int minCostRec(int cost[][], int s, int d)  
    {  
        // If source is same as destination  
        // or destination is next to source  
        if (s == d || s+1 == d)  
            return cost[s][d];  
  
        // Initialize min cost as direct ticket from  
        // source 's' to destination 'd'.  
        int min = cost[s][d];  
  
        // Try every intermediate vertex to find minimum  
        for (int i = s+1; i < d; i++)  
        {  
            int c = minCostRec(cost, s, i) +  
                   minCostRec(cost, i, d);  
            if (c < min)  
                min = c;  
        }  
        return min;  
    }  
}
```

```
// This function returns the smallest possible cost to
// reach station N-1 from station 0. This function mainly
// uses minCostRec().
static int minCost(int cost[][])
{
    return minCostRec(cost, 0, N-1);
}

public static void main(String args[])
{
    int cost[][] = { {0, 15, 80, 90},
                    {INF, 0, 40, 50},
                    {INF, INF, 0, 70},
                    {INF, INF, INF, 0}
                };
    System.out.println("The Minimum cost to reach station "+ N+
                        " is "+minCost(cost));
}

/* This code is contributed by Rajat Mishra */
```

Python

```
# Python program to find min cost path
# from station 0 to station N-1

global N
N = 4
def minCostRec(cost, s, d):

    if s == d or s+1 == d:
        return cost[s][d]

    min = cost[s][d]

    for i in range(s+1, d):
        c = minCostRec(cost, s, i) + minCostRec(cost, i, d)
        if c < min:
            min = c
    return min

def minCost(cost):
    return minCostRec(cost, 0, N-1)
cost = [ [0, 15, 80, 90],
         [float("inf"), 0, 40, 50],
         [float("inf"), float("inf"), 0, 70],
         [float("inf"), float("inf"), float("inf"), 0]
```

```
        ]
print "The Minimum cost to reach station %d is %d" % \
(N, minCost(cost))
```

```
# This code is contributed by Divyanshu Mehta
```

C#

```
// A naive recursive solution to find min
// cost path from station 0 to station N-1
using System;

class GFG {

    static int INF = int.MaxValue, N = 4;

    // A recursive function to find the
    // shortest path from source 's' to
    // destination 'd'.
    static int minCostRec(int [,]cost, int s, int d)
    {

        // If source is same as destination
        // or destination is next to source
        if (s == d || s + 1 == d)
            return cost[s,d];

        // Initialize min cost as direct
        // ticket from source 's' to
        // destination 'd'.
        int min = cost[s,d];

        // Try every intermediate vertex to
        // find minimum
        for (int i = s + 1; i < d; i++)
        {
            int c = minCostRec(cost, s, i) +
                    minCostRec(cost, i, d);

            if (c < min)
                min = c;
        }

        return min;
    }

    // This function returns the smallest
    // possible cost to reach station N-1
```

```

// from station 0. This function mainly
// uses minCostRec().
static int minCost(int [,]cost)
{
    return minCostRec(cost, 0, N-1);
}

// Driver code
public static void Main()
{
    int [,]cost = { {0, 15, 80, 90},
                   {INF, 0, 40, 50},
                   {INF, INF, 0, 70},
                   {INF, INF, INF, 0} };
    Console.WriteLine("The Minimum cost to"
                      + " reach station " + N
                      + " is " + minCost(cost));
}
}

// This code is contributed by Sam007.

```

Output:

The Minimum cost to reach station 4 is 65

Time complexity of the above implementation is exponential as it tries every possible path from 0 to N-1. The above solution solves same subproblems multiple times (it can be seen by drawing recursion tree for minCostPathRec(0, 5)).

Since this problem has both properties of dynamic programming problems ((see [this](#) and [this](#)). Like other typical [Dynamic Programming\(DP\) problems](#), re-computations of same subproblems can be avoided by storing the solutions to subproblems and solving problems in bottom up manner.

One dynamic programming solution is to create a 2D table and fill the table using above given recursive formula. The extra space required in this solution would be $O(N^2)$ and time complexity would be $O(N^3)$

We can solve this problem using $O(N)$ extra space and $O(N^2)$ time. The idea is based on the fact that given input matrix is a Directed Acyclic Graph (DAG). The shortest path in DAG can be calculated using the approach discussed in below post.

[Shortest Path in Directed Acyclic Graph](#)

We need to do less work here compared to above mentioned post as we know [topological sorting](#) of the graph. The topological sorting of vertices here is 0, 1, ..., N-1. Following is the idea once topological sorting is known.

The idea in below code is to first calculate min cost for station 1, then for station 2, and so on. These costs are stored in an array dist[0...N-1].

- 1) The min cost for station 0 is 0, i.e., $\text{dist}[0] = 0$
- 2) The min cost for station 1 is $\text{cost}[0][1]$, i.e., $\text{dist}[1] = \text{cost}[0][1]$
- 3) The min cost for station 2 is minimum of following two.
 - a) $\text{dist}[0] + \text{cost}[0][2]$
 - b) $\text{dist}[1] + \text{cost}[1][2]$
- 3) The min cost for station 3 is minimum of following three.
 - a) $\text{dist}[0] + \text{cost}[0][3]$
 - b) $\text{dist}[1] + \text{cost}[1][3]$
 - c) $\text{dist}[2] + \text{cost}[2][3]$

Similarly, $\text{dist}[4]$, $\text{dist}[5]$, ... $\text{dist}[N-1]$ are calculated.

Below is the implementation of above idea.

C++

```
// A Dynamic Programming based solution to find min cost
// to reach station N-1 from station 0.
#include<iostream>
#include<climits>
using namespace std;

#define INF INT_MAX
#define N 4

// This function returns the smallest possible cost to
// reach station N-1 from station 0.
int minCost(int cost[][])
{
    // dist[i] stores minimum cost to reach station i
    // from station 0.
    int dist[N];
    for (int i=0; i<N; i++)
        dist[i] = INF;
    dist[0] = 0;

    // Go through every station and check if using it
    // as an intermediate station gives better path
    for (int i=0; i<N; i++)
        for (int j=i+1; j<N; j++)
            if (dist[j] > dist[i] + cost[i][j])
                dist[j] = dist[i] + cost[i][j];

    return dist[N-1];
}

// Driver program to test above function
int main()
```

```
{  
    int cost[N][N] = { {0, 15, 80, 90},  
                      {INF, 0, 40, 50},  
                      {INF, INF, 0, 70},  
                      {INF, INF, INF, 0}  
                };  
    cout << "The Minimum cost to reach station "  
         << N << " is " << minCost(cost);  
    return 0;  
}
```

Java

```
// A Dynamic Programming based solution to find min cost  
// to reach station N-1 from station 0.  
class shortest_path  
{  
  
    static int INF = Integer.MAX_VALUE, N = 4;  
    // A recursive function to find the shortest path from  
    // source 's' to destination 'd'.  
  
    // This function returns the smallest possible cost to  
    // reach station N-1 from station 0.  
    static int minCost(int cost[][])  
    {  
        // dist[i] stores minimum cost to reach station i  
        // from station 0.  
        int dist[] = new int[N];  
        for (int i=0; i<N; i++)  
            dist[i] = INF;  
        dist[0] = 0;  
  
        // Go through every station and check if using it  
        // as an intermediate station gives better path  
        for (int i=0; i<N; i++)  
            for (int j=i+1; j<N; j++)  
                if (dist[j] > dist[i] + cost[i][j])  
                    dist[j] = dist[i] + cost[i][j];  
  
        return dist[N-1];  
    }  
  
    public static void main(String args[])  
    {  
        int cost[][] = { {0, 15, 80, 90},  
                        {INF, 0, 40, 50},
```

```
        {INF, INF, 0, 70},
        {INF, INF, INF, 0}
    };
System.out.println("The Minimum cost to reach station "+ N+
                    " is "+minCost(cost));
}

/* This code is contributed by Rajat Mishra */
```

Python3

```
# A Dynamic Programming based
# solution to find min cost
# to reach station N-1
# from station 0.

INF = 2147483647
N = 4

# This function returns the
# smallest possible cost to
# reach station N-1 from station 0.
def minCost(cost):

    # dist[i] stores minimum
    # cost to reach station i
    # from station 0.
    dist=[0 for i in range(N)]
    for i in range(N):
        dist[i] = INF
    dist[0] = 0

    # Go through every station
    # and check if using it
    # as an intermediate station
    # gives better path
    for i in range(N):
        for j in range(i+1,N):
            if (dist[j] > dist[i] + cost[i][j]):
                dist[j] = dist[i] + cost[i][j]

    return dist[N-1]

# Driver program to
# test above function

cost= [ [0, 15, 80, 90],
```

```
[INF, 0, 40, 50],  
[INF, INF, 0, 70],  
[INF, INF, INF, 0]]  
  
print("The Minimum cost to reach station ",  
      N, " is ",minCost(cost))  
  
# This code is contributed  
# by Anant Agarwal.
```

C#

```
// A Dynamic Programming based solution  
// to find min cost to reach station N-1  
// from station 0.  
using System;  
  
class GFG {  
  
    static int INF = int.MaxValue, N = 4;  
    // A recursive function to find the  
    // shortest path from source 's' to  
    // destination 'd'.  
  
    // This function returns the smallest  
    // possible cost to reach station N-1  
    // from station 0.  
    static int minCost(int [,]cost)  
    {  
  
        // dist[i] stores minimum cost  
        // to reach station i from  
        // station 0.  
        int []dist = new int[N];  
  
        for (int i = 0; i < N; i++)  
            dist[i] = INF;  
  
        dist[0] = 0;  
  
        // Go through every station and check  
        // if using it as an intermediate  
        // station gives better path  
        for (int i = 0; i < N; i++)  
            for (int j = i + 1; j < N; j++)  
                if (dist[j] > dist[i] + cost[i,j])  
                    dist[j] = dist[i] + cost[i,j];  
    }  
}
```

```
        return dist[N-1];
    }

    public static void Main()
    {
        int [,]cost = { {0, 15, 80, 90},
                      {INF, 0, 40, 50},
                      {INF, INF, 0, 70},
                      {INF, INF, INF, 0} };
        Console.WriteLine("The Minimum cost to"
                          + " reach station "+ N
                          + " is "+minCost(cost));
    }
}

// This code is contributed by Sam007.
```

Output:

The Minimum cost to reach station 4 is 65

This article is contributed by **Udit Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Improved By : [Sam007](#)

Source

<https://www.geeksforgeeks.org/find-the-minimum-cost-to-reach-a-destination-where-every-station-is-connected-in->

Chapter 34

Given a sorted dictionary of an alien language, find order of characters

Given a sorted dictionary of an alien language, find order of characters - GeeksforGeeks

Given a sorted dictionary (array of words) of an alien language, find order of characters in the language.

Examples:

Input: words[] = {"baa", "abcd", "abca", "cab", "cad"}

Output: Order of characters is 'b', 'd', 'a', 'c'

Note that words are sorted and in the given language "baa" comes before "abcd", therefore 'b' is before 'a' in output.
Similarly we can find other orders.

Input: words[] = {"caa", "aaa", "aab"}

Output: Order of characters is 'c', 'a', 'b'

The idea is to create a graph of characters and then find [topological sorting](#) of the created graph. Following are the detailed steps.

- 1) Create a graph g with number of vertices equal to the size of alphabet in the given alien language. For example, if the alphabet size is 5, then there can be 5 characters in words. Initially there are no edges in graph.
- 2) Do following for every pair of adjacent words in given sorted array.
 -a) Let the current pair of words be $word1$ and $word2$. One by one compare characters of both words and find the first mismatching characters.
 -b) Create an edge in g from mismatching character of $word1$ to that of $word2$.
- 3) Print [topological sorting](#) of the above created graph.

Following is the implementation of the above algorithm.

C++

```
// A C++ program to order of characters in an alien language
#include<iostream>
#include <list>
#include <stack>
#include <cstring>
using namespace std;

// Class to represent a graph
class Graph
{
    int V;      // No. of vertices'

    // Pointer to an array containing adjacency listsList
    list<int> *adj;

    // A function used by topologicalSort
    void topologicalSortUtil(int v, bool visited[], stack<int> &Stack);
public:
    Graph(int V);    // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // prints a Topological Sort of the complete graph
    void topologicalSort();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

// A recursive function used by topologicalSort
void Graph::topologicalSortUtil(int v, bool visited[], stack<int> &Stack)
{
    // Mark the current node as visited.
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
```

```

list<int>::iterator i;
for (i = adj[v].begin(); i != adj[v].end(); ++i)
    if (!visited[*i])
        topologicalSortUtil(*i, visited, Stack);

// Push current vertex to stack which stores result
Stack.push(v);
}

// The function to do Topological Sort. It uses recursive topologicalSortUtil()
void Graph::topologicalSort()
{
    stack<int> Stack;

    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to store Topological Sort
    // starting from all vertices one by one
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            topologicalSortUtil(i, visited, Stack);

    // Print contents of stack
    while (Stack.empty() == false)
    {
        cout << (char) ('a' + Stack.top()) << " ";
        Stack.pop();
    }
}

int min(int x, int y)
{
    return (x < y)? x : y;
}

// This function finds and prints order of character from a sorted
// array of words. n is size of words[]. alpha is set of possible
// alphabets.
// For simplicity, this function is written in a way that only
// first 'alpha' characters can be there in words array. For
// example if alpha is 7, then words[] should have only 'a', 'b',
// 'c' 'd', 'e', 'f', 'g'
void printOrder(string words[], int n, int alpha)
{
    // Create a graph with 'alpha' edges

```

```
Graph g(alpha);

// Process all adjacent pairs of words and create a graph
for (int i = 0; i < n-1; i++)
{
    // Take the current two words and find the first mismatching
    // character
    string word1 = words[i], word2 = words[i+1];
    for (int j = 0; j < min(word1.length(), word2.length()); j++)
    {
        // If we find a mismatching character, then add an edge
        // from character of word1 to that of word2
        if (word1[j] != word2[j])
        {
            g.addEdge(word1[j]-'a', word2[j]-'a');
            break;
        }
    }
}

// Print topological sort of the above created graph
g.topologicalSort();
}

// Driver program to test above functions
int main()
{
    string words[] = {"caa", "aaa", "aab"};
    printOrder(words, 3, 3);
    return 0;
}
```

Java

```
// A Java program to order of
// characters in an alien language
import java.util.*;

// Class to represent a graph
class Graph
{

    // An array representing the graph as an adjacency list
    private final LinkedList<Integer>[] adjacencyList;

    Graph(int nVertices)
    {
        adjacencyList = new LinkedList[nVertices];
```

```
for (int vertexIndex = 0; vertexIndex < nVertices; vertexIndex++)
{
    adjacencyList[vertexIndex] = new LinkedList<>();
}
}

// function to add an edge to graph
void addEdge(int startVertex, int endVertex)
{
    adjacencyList[startVertex].add(endVertex);
}

private int getNoOfVertices()
{
    return adjacencyList.length;
}

// A recursive function used by topologicalSort
private void topologicalSortUtil(int currentVertex, boolean[] visited,
                                 Stack<Integer> stack)
{
    // Mark the current node as visited.
    visited[currentVertex] = true;

    // Recur for all the vertices adjacent to this vertex
    for (int adjacentVertex : adjacencyList[currentVertex])
    {
        if (!visited[adjacentVertex])
        {
            topologicalSortUtil(adjacentVertex, visited, stack);
        }
    }

    // Push current vertex to stack which stores result
    stack.push(currentVertex);
}

// prints a Topological Sort of the complete graph
void topologicalSort()
{
    Stack<Integer> stack = new Stack<>();

    // Mark all the vertices as not visited
    boolean[] visited = new boolean[getNoOfVertices()];
    for (int i = 0; i < getNoOfVertices(); i++)
    {
        visited[i] = false;
    }
}
```

```

// Call the recursive helper function to store Topological
// Sort starting from all vertices one by one
for (int i = 0; i < getNoOfVertices(); i++)
{
    if (!visited[i])
    {
        topologicalSortUtil(i, visited, stack);
    }
}

// Print contents of stack
while (!stack.isEmpty())
{
    System.out.print((char)('a' + stack.pop()) + " ");
}
}

public class OrderOfCharacters
{
    // This function finds and prints order
    // of character from a sorted array of words.
    // alpha is number of possible alphabets
    // starting from 'a'. For simplicity, this
    // function is written in a way that only
    // first 'alpha' characters can be there
    // in words array. For example if alpha
    // is 7, then words[] should contain words
    // having only 'a', 'b', 'c', 'd', 'e', 'f', 'g'
    private static void printOrder(String[] words, int alpha)
    {
        // Create a graph with 'alpha' edges
        Graph graph = new Graph(alpha);

        for (int i = 0; i < words.length - 1; i++)
        {
            // Take the current two words and find the first mismatching
            // character
            String word1 = words[i];
            String word2 = words[i+1];
            for (int j = 0; j < Math.min(word1.length(), word2.length()); j++)
            {
                // If we find a mismatching character, then add an edge
                // from character of word1 to that of word2
                if (word1.charAt(j) != word2.charAt(j))
                {
                    graph.addEdge(word1.charAt(j) - 'a', word2.charAt(j)- 'a');
                }
            }
        }
    }
}

```

```
        break;
    }
}
}

// Print topological sort of the above created graph
graph.topologicalSort();
}

// Driver program to test above functions
public static void main(String[] args)
{
    String[] words = {"caa", "aaa", "aab"};
    printOrder(words, 3);
}
}

//Contributed by Harikrishnan Rajan
```

Output:

c a b

Time Complexity: The first step to create a graph takes $O(n + \alpha)$ time where n is number of given words and α is number of characters in given alphabet. The second step is also topological sorting. Note that there would be α vertices and at-most $(n-1)$ edges in the graph. The time complexity of [topological sorting](#) is $O(V+E)$ which is $O(n + \alpha)$ here. So overall time complexity is $O(n + \alpha) + O(n + \alpha)$ which is $O(n + \alpha)$.

Exercise:

The above code doesn't work when the input is not valid. For example {"aba", "bba", "aaa"} is not valid, because from first two words, we can deduce 'a' should appear before 'b', but from last two words, we can deduce 'b' should appear before 'a' which is not possible. Extend the above program to handle invalid inputs and generate the output as "Not valid".

This article is contributed by **Piyush Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

<https://www.geeksforgeeks.org/given-sorted-dictionary-find-precedence-characters/>

Chapter 35

Length of shortest chain to reach a target word

Word Ladder (Length of shortest chain to reach a target word) - GeeksforGeeks

Given a dictionary, and two words ‘start’ and ‘target’ (both of same length). Find length of the smallest chain from ‘start’ to ‘target’ if it exists, such that adjacent words in the chain only differ by one character and each word in the chain is a valid word i.e., it exists in the dictionary. It may be assumed that the ‘target’ word exists in dictionary and length of all dictionary words is same.

Example:

```
Input: Dictionary = {POON, PLEE, SAME, POIE, PLEA, PLIE, POIN}
       start = TOON
       target = PLEA
Output: 7
Explanation: TOON - POON - POIN - POIE - PLIE - PLEE - PLEA
```

The idea is to use [BFS](#). We start from the given start word, traverse all words that adjacent (differ by one character) to it and keep doing so until we find the target word or we have traversed all words.

Below is C++ implementation of above idea.

C

```
// C++ program to find length of the shortest chain
// transformation from source to target
#include<bits/stdc++.h>
using namespace std;

// To check if strings differ by exactly one character
```

```

bool isadjacent(string& a, string& b)
{
    int count = 0; // to store count of differences
    int n = a.length();

    // Iterate through all characters and return false
    // if there are more than one mismatching characters
    for (int i = 0; i < n; i++)
    {
        if (a[i] != b[i]) count++;
        if (count > 1) return false;
    }
    return count == 1 ? true : false;
}

// A queue item to store word and minimum chain length
// to reach the word.
struct QItem
{
    string word;
    int len;
};

// Returns length of shortest chain to reach 'target' from 'start'
// using minimum number of adjacent moves. D is dictionary
int shortestChainLen(string& start, string& target, set<string> &D)
{
    // Create a queue for BFS and insert 'start' as source vertex
    queue<QItem> Q;
    QItem item = {start, 1}; // Chain length for start word is 1
    Q.push(item);

    // While queue is not empty
    while (!Q.empty())
    {
        // Take the front word
        QItem curr = Q.front();
        Q.pop();

        // Go through all words of dictionary
        for (set<string>::iterator it = D.begin(); it != D.end(); it++)
        {
            // Process a dictionary word if it is adjacent to current
            // word (or vertex) of BFS
            string temp = *it;
            if (isadjacent(curr.word, temp))
            {
                // Add the dictionary word to Q

```

```
        item.word = temp;
        item.len = curr.len + 1;
        Q.push(item);

        // Remove from dictionary so that this word is not
        // processed again. This is like marking visited
        D.erase(temp);

        // If we reached target
        if (temp == target)
            return item.len;
    }
}
return 0;
}

// Driver program
int main()
{
    // make dictionary
    set<string> D;
    D.insert("poon");
    D.insert("plee");
    D.insert("same");
    D.insert("poie");
    D.insert("plie");
    D.insert("poin");
    D.insert("plea");
    string start = "toon";
    string target = "plea";
    cout << "Length of shortest chain is: "
         << shortestChainLen(start, target, D);
    return 0;
}
```

Python

```
# To check if strings differ by
# exactly one character

def isadjacent(a, b):
    count = 0
    n = len(a)

    # Iterate through all characters and return false
    # if there are more than one mismatching characters
    for i in range(n):
```

```

if a[i] != b[i]:
    count += 1
if count > 1:
    break

return True if count == 1 else False

# A queue item to store word and minimum chain length
# to reach the word.
class QItem():

    def __init__(self, word, len):
        self.word = word
        self.len = len

    # Returns length of shortest chain to reach
    # 'target' from 'start' using minimum number
    # of adjacent moves. D is dictionary
    def shortestChainLen(start, target, D):

        # Create a queue for BFS and insert
        # 'start' as source vertex
        Q = []
        item = QItem(start, 1)
        Q.append(item)

        while( len(Q) > 0):

            curr = Q.pop()

            # Go through all words of dictionary
            for it in D:

                # Process a dictionary word if it is
                # adjacent to current word (or vertex) of BFS
                temp = it
                if isadjacent(curr.word, temp) == True:

                    # Add the dictionary word to Q
                    item.word = temp
                    item.len = curr.len + 1
                    Q.append(item)

                # Remove from dictionary so that this
                # word is not processed again. This is
                # like marking visited
                D.remove(temp)

```

```
# If we reached target
if temp == target:
    return item.len

D = []
D.append("poon")
D.append("plee")
D.append("same")
D.append("poie")
D.append("plie")
D.append("poin")
D.append("plea")
start = "toon"
target = "plea"
print "Length of shortest chain is: %d" \
      % shortestChainLen(start, target, D)

# This code is contributed by Divyanshu Mehta
```

Output:

```
Length of shortest chain is: 7
```

Time Complexity of the above code is $O(n^2m)$ where n is the number of entries originally in the dictionary and m is the size of the string

Thanks to Gaurav Ahirwar and Rajnish Kumar Jha for above solution.

Source

<https://www.geeksforgeeks.org/word-ladder-length-of-shortest-chain-to-reach-a-target-word/>

Chapter 36

Minimum time required to rot all oranges

Minimum time required to rot all oranges - GeeksforGeeks

Given a matrix of dimension $m \times n$ where each cell in the matrix can have values 0, 1 or 2 which has the following meaning:

0: Empty cell

1: Cells have fresh oranges

2: Cells have rotten oranges

So we have to determine what is the minimum time required so that all the oranges become rotten. A rotten orange at index [i,j] can rot other fresh orange at indexes [i-1,j], [i+1,j], [i,j-1], [i,j+1] (up, down, left and right). If it is impossible to rot every orange then simply return -1.

Examples:

Input: arr[0][C] = {{2, 1, 0, 2, 1},
 {1, 0, 1, 2, 1},
 {1, 0, 0, 2, 1}};

Output:

All oranges can become rotten in 2 time frames.

Input: arr[0][C] = {{2, 1, 0, 2, 1},
 {0, 0, 1, 2, 1},

```
{1, 0, 0, 2, 1};
```

Output:

All oranges cannot be rotten.

The idea is to user Breadth First Search. Below is algorithm.

- 1) Create an empty Q.
- 2) Find all rotten oranges and enqueue them to Q. Also enqueue a delimiter to indicate beginning of next time frame.
- 3) While Q is not empty do following
 - 3.a) While delimiter in Q is not reached
 - (i) Dequeue an orange from queue, rot all adjacent oranges. While rotting the adjacents, make sure that time frame is incremented only once. And time frame is not incremented if there are no adjacent oranges.
 - 3.b) Dequeue the old delimiter and enqueue a new delimiter. The oranges rotten in previous time frame lie between the two delimiters.

Below is implementation of the above idea.

C++

```
// C++ program to find minimum time required to make all
// oranges rotten
#include<bits/stdc++.h>
#define R 3
#define C 5
using namespace std;

// function to check whether a cell is valid / invalid
bool isValid(int i, int j)
{
    return (i >= 0 && j >= 0 && i < R && j < C);
}

// structure for storing coordinates of the cell
struct ele {
    int x, y;
};

// Function to check whether the cell is delimiter
// which is (-1, -1)
bool isdelim(ele temp)
{
    return (temp.x == -1 && temp.y == -1);
}
```

```
// Function to check whether there is still a fresh
// orange remaining
bool checkall(int arr[][])
{
    for (int i=0; i<R; i++)
        for (int j=0; j<C; j++)
            if (arr[i][j] == 1)
                return true;
    return false;
}

// This function finds if it is possible to rot all oranges or not.
// If possible, then it returns minimum time required to rot all,
// otherwise returns -1
int rotOranges(int arr[][])
{
    // Create a queue of cells
    queue<ele> Q;
    ele temp;
    int ans = 0;

    // Store all the cells having rotten orange in first time frame
    for (int i=0; i<R; i++)
    {
        for (int j=0; j<C; j++)
        {
            if (arr[i][j] == 2)
            {
                temp.x = i;
                temp.y = j;
                Q.push(temp);
            }
        }
    }

    // Separate these rotten oranges from the oranges which will rotten
    // due the oranges in first time frame using delimiter which is (-1, -1)
    temp.x = -1;
    temp.y = -1;
    Q.push(temp);

    // Process the grid while there are rotten oranges in the Queue
    while (!Q.empty())
    {
        // This flag is used to determine whether even a single fresh
        // orange gets rotten due to rotten oranges in current time
        // frame so we can increase the count of the required time.
```

```
bool flag = false;

// Process all the rotten oranges in current time frame.
while (!isdelim(Q.front()))
{
    temp = Q.front();

    // Check right adjacent cell that if it can be rotten
    if (isValid(temp.x+1, temp.y) && arr[temp.x+1][temp.y] == 1)
    {
        // if this is the first orange to get rotten, increase
        // count and set the flag.
        if (!flag) ans++, flag = true;

        // Make the orange rotten
        arr[temp.x+1][temp.y] = 2;

        // push the adjacent orange to Queue
        temp.x++;
        Q.push(temp);

        temp.x--; // Move back to current cell
    }

    // Check left adjacent cell that if it can be rotten
    if (isValid(temp.x-1, temp.y) && arr[temp.x-1][temp.y] == 1) {
        if (!flag) ans++, flag = true;
        arr[temp.x-1][temp.y] = 2;
        temp.x--;
        Q.push(temp); // push this cell to Queue
        temp.x++;
    }

    // Check top adjacent cell that if it can be rotten
    if (isValid(temp.x, temp.y+1) && arr[temp.x][temp.y+1] == 1) {
        if (!flag) ans++, flag = true;
        arr[temp.x][temp.y+1] = 2;
        temp.y++;
        Q.push(temp); // Push this cell to Queue
        temp.y--;
    }

    // Check bottom adjacent cell if it can be rotten
    if (isValid(temp.x, temp.y-1) && arr[temp.x][temp.y-1] == 1) {
        if (!flag) ans++, flag = true;
        arr[temp.x][temp.y-1] = 2;
        temp.y--;
        Q.push(temp); // push this cell to Queue
    }
}
```

```
    }

    Q.pop();
}

// Pop the delimiter
Q.pop();

// If oranges were rotten in current frame than separate the
// rotten oranges using delimiter for the next frame for processing.
if (!Q.empty()) {
    temp.x = -1;
    temp.y = -1;
    Q.push(temp);
}

// If Queue was empty than no rotten oranges left to process so exit
}

// Return -1 if all arranges could not rot, otherwise -1.
return (checkall(arr))? -1: ans;
}

// Drive program
int main()
{
    int arr[][][C] = { {2, 1, 0, 2, 1},
                      {1, 0, 1, 2, 1},
                      {1, 0, 0, 2, 1} };
    int ans = rotOranges(arr);
    if (ans == -1)
        cout << "All oranges cannot rot";
    else
        cout << "Time required for all oranges to rot => " << ans << endl;
    return 0;
}
```

Java

```
//Java program to find minimum time required to make all
//oranges rotten

import java.util.LinkedList;
import java.util.Queue;

public class RotOrange
{
    public final static int R = 3;
```

```
public final static int C = 5;

// structure for storing coordinates of the cell
static class Ele
{
    int x = 0;
    int y = 0;
    Ele(int x,int y)
    {
        this.x = x;
        this.y = y;
    }
}

// function to check whether a cell is valid / invalid
static boolean isValid(int i, int j)
{
    return (i >= 0 && j >= 0 && i < R && j < C);
}

// Function to check whether the cell is delimiter
// which is (-1, -1)
static boolean isDelim(Ele temp)
{
    return (temp.x == -1 && temp.y == -1);
}

// Function to check whether there is still a fresh
// orange remaining
static boolean checkAll(int arr[][])
{
    for (int i=0; i<R; i++)
        for (int j=0; j<C; j++)
            if (arr[i][j] == 1)
                return true;
    return false;
}

// This function finds if it is possible to rot all oranges or not.
// If possible, then it returns minimum time required to rot all,
// otherwise returns -1
static int rotOranges(int arr[][])
{
    // Create a queue of cells
    Queue<Ele> Q=new LinkedList<>();
    Ele temp;
    int ans = 0;
```

```

// Store all the cells having rotten orange in first time frame
for (int i=0; i < R; i++)
    for (int j=0; j < C; j++)
        if (arr[i][j] == 2)
            Q.add(new Ele(i,j));

// Separate these rotten oranges from the oranges which will rotten
// due the oranges in first time frame using delimiter which is (-1, -1)
Q.add(new Ele(-1,-1));

// Process the grid while there are rotten oranges in the Queue
while(!Q.isEmpty())
{
    // This flag is used to determine whether even a single fresh
    // orange gets rotten due to rotten oranges in current time
    // frame so we can increase the count of the required time.
    boolean flag = false;

    // Process all the rotten oranges in current time frame.
    while(!isDelim(Q.peek()))
    {
        temp = Q.peek();

        // Check right adjacent cell that if it can be rotten
        if(isValid(temp.x+1, temp.y+1) && arr[temp.x+1][temp.y] == 1)
        {
            if(!flag)
            {
                // if this is the first orange to get rotten, increase
                // count and set the flag.
                ans++;
                flag = true;
            }
            // Make the orange rotten
            arr[temp.x+1][temp.y] = 2;

            // push the adjacent orange to Queue
            temp.x++;
            Q.add(new Ele(temp.x,temp.y));

            // Move back to current cell
            temp.x--;
        }

        // Check left adjacent cell that if it can be rotten
        if (isValid(temp.x-1, temp.y) && arr[temp.x-1][temp.y] == 1)
        {
            if (!flag)

```

```

    {
        ans++;
        flag = true;
    }
    arr[temp.x-1][temp.y] = 2;
    temp.x--;
    Q.add(new Ele(temp.x,temp.y)); // push this cell to Queue
    temp.x++;
}

// Check top adjacent cell that if it can be rotten
if (isValid(temp.x, temp.y+1) && arr[temp.x][temp.y+1] == 1) {
    if(!flag)
    {
        ans++;
        flag = true;
    }
    arr[temp.x][temp.y+1] = 2;
    temp.y++;
    Q.add(new Ele(temp.x,temp.y)); // Push this cell to Queue
    temp.y--;
}

// Check bottom adjacent cell if it can be rotten
if (isValid(temp.x, temp.y-1) && arr[temp.x][temp.y-1] == 1)
{
    if (!flag)
    {
        ans++;
        flag = true;
    }
    arr[temp.x][temp.y-1] = 2;
    temp.y--;
    Q.add(new Ele(temp.x,temp.y)); // push this cell to Queue
}
Q.remove();

}

// Pop the delimiter
Q.remove();

// If oranges were rotten in current frame than separate the
// rotten oranges using delimiter for the next frame for processing.
if (!Q.isEmpty())
{
    Q.add(new Ele(-1,-1));
}

```

```
// If Queue was empty than no rotten oranges left to process so exit
}

// Return -1 if all arranges could not rot, otherwise -1.s
return (checkAll(arr))? -1: ans;

}

// Drive program
public static void main(String[] args)
{
    int arr[][] = { {2, 1, 0, 2, 1},
                    {1, 0, 1, 2, 1},
                    {1, 0, 0, 2, 1}};
    int ans = rotOranges(arr);
    if(ans == -1)
        System.out.println("All oranges cannot rot");
    else
        System.out.println("Time required for all oranges to rot = " + ans);
}

//This code is contributed by Sumit Ghosh
```

Output:

Time required for all oranges to rot => 2

Thanks to Gaurav Ahirwar for suggesting above solution.

Source

<https://www.geeksforgeeks.org/minimum-time-required-so-that-all-oranges-become-rotten/>

Chapter 37

Optimal read list for given number of days

Optimal read list for given number of days - GeeksforGeeks

A person is determined to finish the book in ‘k’ days but he never wants to stop a chapter in between. Find the optimal assignment of chapters, such that the person doesn’t read too many extra/less pages overall.

Example 1:

Input: Number of Days to Finish book = 2
Number of pages in chapters = {10, 5, 5}
Output: Day 1: Chapter 1
Day 2: Chapters 2 and 3

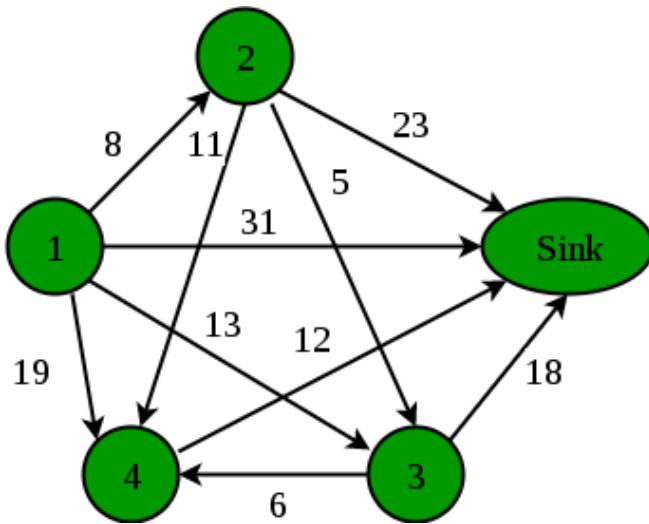
Example 2:

Input: Number of Days to Finish book = 3
Number of pages in chapters = {8, 5, 6, 12}
Output: Day 1: Chapter 1
Day 2: Chapters 2 and 3
Day 3: Chapter 4

The target is to minimize the sum of differences between the pages read on each day and average number of pages. If the average number of pages is a non-integer, then it should be rounded to closest integer.

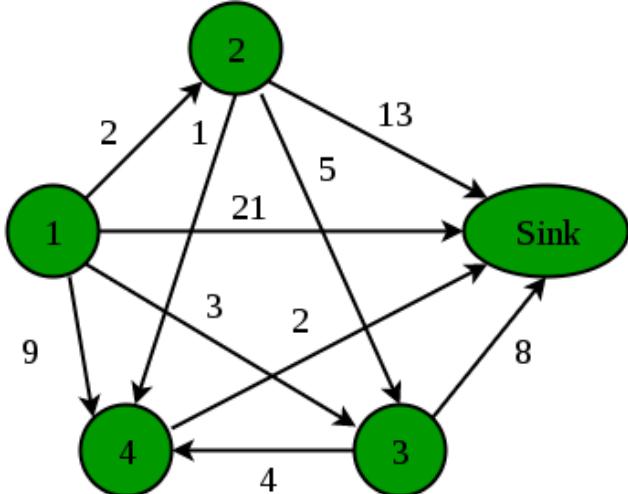
In above example 2, average number of pages is $(8 + 5 + 6 + 12)/3 = 31/3$ which is rounded to 10. So the difference between average and number of pages on each day for the output shown above is “ $\text{abs}(8-10) + \text{abs}(5+6-10) + \text{abs}(12-10)$ ” which is 5. The value 5 is the optimal value of sum of differences.

Consider the example 2 above where a book has 4 chapters with pages 8, 5, 6 and 12. User wishes to finish it in 3 days. The graphical representation of the above scenario is,



In the above graph vertex represents the chapter and an edge $e(u, v)$ represents number of pages to be read to reach ' v ' from ' u '. Sink node is added to symbolize the end of book.

First, calculate the average number of pages to read in a day (here $31/3$ roughly 10). New edge weight $e'(u, v)$ would be the mean difference $\text{avg} - e(u, v)$. Modified graph for the above problem would be,



Thanks to [Armadillo](#)for initiating this thought in a comment.

The idea is to start from chapter 1 and do a DFS to find sink with count of edges being ' k '. Keep storing the visited vertices in an array say 'path[]'. If we reach the destination vertex, and path sum is less than the optimal path update the optimal assignment `optimal_path[]`. Note, that the graph is DAG thus there is no need to take care of cycles during DFS. Following, is the C++ implementation of the same, adjacency matrix is used to represent the graph. The following program has mainly 4 phases.

- 1) Construct a directed acyclic graph.

- 2) Find the optimal path using DFS.
- 3) Print the found optimal path.

```
// C++ DFS solution to schedule chapters for reading in
// given days
# include <iostream>
# include <cstdlib>
# include <climits>
# include <cmath>
using namespace std;

// Define total chapters in the book
// Number of days user can spend on reading
# define CHAPTERS 4
# define DAYS 3
# define NOLINK -1

// Array to store the final balanced schedule
int optimal_path[DAYS+1];

// Graph - Node chapter+1 is the sink described in the
//         above graph
int DAG[CHAPTERS+1][CHAPTERS+1];

// Updates the optimal assignment with current assignment
void updateAssignment(int* path, int path_len);

// A DFS based recursive function to store the optimal path
// in path[] of size path_len. The variable sum stores sum of
// of all edges on current path. k is number of days spent so
// far.
void assignChapters(int u, int* path, int path_len, int sum, int k)
{
    static int min = INT_MAX;

    // Ignore the assignment which requires more than required days
    if (k < 0)
        return;

    // Current assignment of chapters to days
    path[path_len] = u;
    path_len++;

    // Update the optimal assignment if necessary
    if (k == 0 && u == CHAPTERS)
    {
        if (sum < min)
        {
```

```

        updateAssignment(path, path_len);
        min = sum;
    }
}

// DFS - Depth First Search for sink
for (int v = u+1; v <= CHAPTERS; v++)
{
    sum += DAG[u][v];
    assignChapters(v, path, path_len, sum, k-1);
    sum -= DAG[u][v];
}
}

// This function finds and prints optimal read list. It first creates a
// graph, then calls assignChapters().
void minAssignment(int pages[])
{
    // 1) .....CONSTRUCT GRAPH.....
    // Partial sum array construction S[i] = total pages
    // till ith chapter
    int avg_pages = 0, sum = 0, S[CHAPTERS+1], path[DAYST+1];
    S[0] = 0;

    for (int i = 0; i < CHAPTERS; i++)
    {
        sum += pages[i];
        S[i+1] = sum;
    }

    // Average pages to be read in a day
    avg_pages = round(sum/DAYS);

    /* DAG construction vertices being chapter name &
     * Edge weight being |avg_pages - pages in a chapter|
     * Adjacency matrix representation */
    for (int i = 0; i <= CHAPTERS; i++)
    {
        for (int j = 0; j <= CHAPTERS; j++)
        {
            if (j <= i)
                DAG[i][j] = NOLINK;
            else
            {
                sum = abs(avg_pages - (S[j] - S[i]));
                DAG[i][j] = sum;
            }
        }
    }
}

```

```
}

// 2) .....FIND OPTIMAL PATH.....
assignChapters(0, path, 0, 0, DAYS);

// 3) ..PRINT OPTIMAL READ LIST USING OPTIMAL PATH....
cout << "Optimal Chapter Assignment :" << endl;
int ch;
for (int i = 0; i < DAYS; i++)
{
    ch = optimal_path[i];
    cout << "Day" << i+1 << ":" << ch << " ";
    ch++;
    while ( (i < DAYS-1 && ch < optimal_path[i+1]) ||
            (i == DAYS-1 && ch <= CHAPTERS))
    {
        cout << ch << " ";
        ch++;
    }
    cout << endl;
}
}

// This function updates optimal_path[]
void updateAssignment(int* path, int path_len)
{
    for (int i = 0; i < path_len; i++)
        optimal_path[i] = path[i] + 1;
}

// Driver program to test the schedule
int main(void)
{
    int pages[CHAPTERS] = {7, 5, 6, 12};

    // Get read list for given days
    minAssignment(pages);

    return 0;
}
```

Output:

```
Optimal Chapter Assignment :
Day1: 1
Day2: 2 3
Day3: 4
```

This article is contributed by **Balaji S.** Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

<https://www.geeksforgeeks.org/optimal-read-list-given-number-days/>

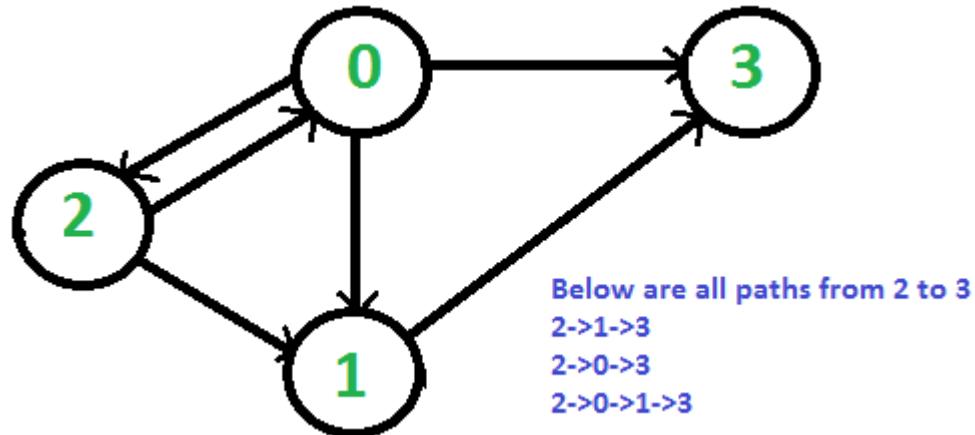
Chapter 38

Print all paths from a given source to a destination

Print all paths from a given source to a destination - GeeksforGeeks

Given a directed graph, a source vertex ‘s’ and a destination vertex ‘d’, print all paths from given ‘s’ to ‘d’.

Consider the following directed graph. Let the s be 2 and d be 3. There are 4 different paths from 2 to 3.



The idea is to do [Depth First Traversal](#) of given directed graph. Start the traversal from source. Keep storing the visited vertices in an array say ‘path[]’. If we reach the destination vertex, print contents of path[]. The important thing is to mark current vertices in path[] as visited also, so that the traversal doesn’t go in a cycle.

Following is implementation of above idea.

C/C++

```
// C++ program to print all paths from a source to destination.
```

```
#include<iostream>
#include <list>
using namespace std;

// A directed graph using adjacency list representation
class Graph
{
    int V;      // No. of vertices in graph
    list<int> *adj; // Pointer to an array containing adjacency lists

    // A recursive function used by printAllPaths()
    void printAllPathsUtil(int , int , bool [], int [], int &);

public:
    Graph(int V);    // Constructor
    void addEdge(int u, int v);
    void printAllPaths(int s, int d);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int u, int v)
{
    adj[u].push_back(v); // Add v to u's list.
}

// Prints all paths from 's' to 'd'
void Graph::printAllPaths(int s, int d)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];

    // Create an array to store paths
    int *path = new int[V];
    int path_index = 0; // Initialize path[] as empty

    // Initialize all vertices as not visited
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to print all paths
    printAllPathsUtil(s, d, visited, path, path_index);
}
```

```
// A recursive function to print all paths from 'u' to 'd'.
// visited[] keeps track of vertices in current path.
// path[] stores actual vertices and path_index is current
// index in path[]
void Graph::printAllPathsUtil(int u, int d, bool visited[],
                               int path[], int &path_index)
{
    // Mark the current node and store it in path[]
    visited[u] = true;
    path[path_index] = u;
    path_index++;

    // If current vertex is same as destination, then print
    // current path[]
    if (u == d)
    {
        for (int i = 0; i<path_index; i++)
            cout << path[i] << " ";
        cout << endl;
    }
    else // If current vertex is not destination
    {
        // Recur for all the vertices adjacent to current vertex
        list<int>::iterator i;
        for (i = adj[u].begin(); i != adj[u].end(); ++i)
            if (!visited[*i])
                printAllPathsUtil(*i, d, visited, path, path_index);
    }

    // Remove current vertex from path[] and mark it as unvisited
    path_index--;
    visited[u] = false;
}

// Driver program
int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(0, 3);
    g.addEdge(2, 0);
    g.addEdge(2, 1);
    g.addEdge(1, 3);

    int s = 2, d = 3;
    cout << "Following are all different paths from " << s
```

```
        << " to " << d << endl;
g.printAllPaths(s, d);

return 0;
}
```

Java

```
// JAVA program to print all
// paths from a source to
// destination.
import java.util.ArrayList;
import java.util.List;

// A directed graph using
// adjacency list representation
public class Graph {

    // No. of vertices in graph
    private int v;

    // adjacency list
    private ArrayList<Integer>[] adjList;

    //Constructor
    public Graph(int vertices){

        //initialise vertex count
        this.v = vertices;

        // initialise adjacency list
        initAdjList();
    }

    // utility method to initialise
    // adjacency list
    @SuppressWarnings("unchecked")
    private void initAdjList()
    {
        adjList = new ArrayList[v];

        for(int i = 0; i < v; i++)
        {
            adjList[i] = new ArrayList<>();
        }
    }

    // add edge from u to v
```

```
public void addEdge(int u, int v)
{
    // Add v to u's list.
    adjList[u].add(v);
}

// Prints all paths from
// 's' to 'd'
public void printAllPaths(int s, int d)
{
    boolean[] isVisited = new boolean[v];
    ArrayList<Integer> pathList = new ArrayList<>();

    //add source to path[]
    pathList.add(s);

    //Call recursive utility
    printAllPathsUtil(s, d, isVisited, pathList);
}

// A recursive function to print
// all paths from 'u' to 'd'.
// isVisited[] keeps track of
// vertices in current path.
// localPathList<> stores actual
// vertices in the current path
private void printAllPathsUtil(Integer u, Integer d,
                               boolean[] isVisited,
                               List<Integer> localPathList) {

    // Mark the current node
    isVisited[u] = true;

    if (u.equals(d))
    {
        System.out.println(localPathList);
    }

    // Recur for all the vertices
    // adjacent to current vertex
    for (Integer i : adjList[u])
    {
        if (!isVisited[i])
        {
            // store current node
            // in path[]
            localPathList.add(i);
            printAllPathsUtil(i, d, isVisited, localPathList);
        }
    }
}
```

```
// remove current node
// in path[]
localPathList.remove(i);
}
}

// Mark the current node
isVisited[u] = false;
}

// Driver program
public static void main(String[] args)
{
    // Create a sample graph
    Graph g = new Graph(4);
    g.addEdge(0,1);
    g.addEdge(0,2);
    g.addEdge(0,3);
    g.addEdge(2,0);
    g.addEdge(2,1);
    g.addEdge(1,3);

    // arbitrary source
    int s = 2;

    // arbitrary destination
    int d = 3;

    System.out.println("Following are all different paths from "+s+" to "+d);
    g.printAllPaths(s, d);
}

}

// This code is contributed by Himanshu Shekhar.
```

Python

```
# Python program to print all paths from a source to destination.

from collections import defaultdict

#This class represents a directed graph
# using adjacency list representation
class Graph:

    def __init__(self,vertices):
```

```
#No. of vertices
self.V= vertices

# default dictionary to store graph
self.graph = defaultdict(list)

# function to add an edge to graph
def addEdge(self,u,v):
    self.graph[u].append(v)

'''A recursive function to print all paths from 'u' to 'd'.
visited[] keeps track of vertices in current path.
path[] stores actual vertices and path_index is current
index in path[]
'''
def printAllPathsUtil(self, u, d, visited, path):

    # Mark the current node as visited and store in path
    visited[u]= True
    path.append(u)

    # If current vertex is same as destination, then print
    # current path[]
    if u ==d:
        print path
    else:
        # If current vertex is not destination
        #Recur for all the vertices adjacent to this vertex
        for i in self.graph[u]:
            if visited[i]==False:
                self.printAllPathsUtil(i, d, visited, path)

    # Remove current vertex from path[] and mark it as unvisited
    path.pop()
    visited[u]= False

# Prints all paths from 's' to 'd'
def printAllPaths(self,s, d):

    # Mark all the vertices as not visited
    visited =[False]*(self.V)

    # Create an array to store paths
    path = []

    # Call the recursive helper function to print all paths
    self.printAllPathsUtil(s, d,visited, path)
```

```
# Create a graph given in the above diagram
g = Graph(4)
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(0, 3)
g.addEdge(2, 0)
g.addEdge(2, 1)
g.addEdge(1, 3)

s = 2 ; d = 3
print ("Following are all different paths from %d to %d :" %(s, d))
g.printAllPaths(s, d)
#This code is contributed by Neelam Yadav
```

Output:

```
Following are all different paths from 2 to 3
2 0 1 3
2 0 3
2 1 3
```

This article is contributed by **Shivam Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

<https://www.geeksforgeeks.org/find-paths-given-source-destination/>

Chapter 39

Print all Jumping Numbers smaller than or equal to a given value

Print all Jumping Numbers smaller than or equal to a given value - GeeksforGeeks

A number is called as a Jumping Number if all adjacent digits in it differ by 1. The difference between ‘9’ and ‘0’ is not considered as 1.

All single digit numbers are considered as Jumping Numbers. For example 7, 8987 and 4343456 are Jumping numbers but 796 and 89098 are not.

Given a positive number x, print all Jumping Numbers smaller than or equal to x. The numbers can be printed in any order.

Example:

```
Input: x = 20
Output: 0 1 2 3 4 5 6 7 8 9 10 12
```

```
Input: x = 105
Output: 0 1 2 3 4 5 6 7 8 9 10 12
        21 23 32 34 43 45 54 56 65
        67 76 78 87 89 98 101
```

Note: Order of output doesn't matter,
i.e., numbers can be printed in any order

One **Simple Solution** is to traverse all numbers from 0 to x. For every traversed number, check if it is a Jumping number. If yes, then print it. Otherwise ignore it. Time Complexity of this solution is O(x).

An **Efficient Solution** can solve this problem in $O(k)$ time where k is number of Jumping Numbers smaller than or equal to x . The idea is use [BFS](#) or [DFS](#).

Assume that we have a graph where the starting node is 0 and we need to traverse it from the start node to all the reachable nodes.

With the restrictions given in the graph about the jumping numbers, what do you think should be the restrictions defining the next transitions in the graph.

Lets take a example for input $x = 90$

```
Start node = 0
From 0, we can move to 1 2 3 4 5 6 7 8 9
[these are not in our range so we don't add it]
```

Now from 1, we can move to 12 and 10

```
From 2, 23 and 21
From 3, 34 and 32
```

```
.
.
.
.
.
.
```

and so on.

Below is BFS based C++ implementation of above idea.

C++

```
// Finds and prints all jumping numbers smaller than or
// equal to x.
#include<bits/stdc++.h>
using namespace std;

// Prints all jumping numbers smaller than or equal to x starting
// with 'num'. It mainly does BFS starting from 'num'.
void bfs(int x, int num)
{
    // Create a queue and enqueue 'i' to it
    queue<int> q;
    q.push(num);

    // Do BFS starting from i
    while (!q.empty())
    {
        num = q.front();
        q.pop();
```

```
if (num <= x)
{
    cout << num << " ";
    int last_dig = num % 10;

    // If last digit is 0, append next digit only
    if (last_dig == 0)
        q.push((num*10) + (last_dig+1));

    // If last digit is 9, append previous digit only
    else if (last_dig == 9)
        q.push( (num*10) + (last_dig-1) );

    // If last digit is neither 0 nor 9, append both
    // previous and next digits
    else
    {
        q.push((num*10) + (last_dig-1));
        q.push((num*10) + (last_dig+1));
    }
}

}

// Prints all jumping numbers smaller than or equal to
// a positive number x
void printJumping(int x)
{
    cout << 0 << " ";
    for (int i=1; i<=9 && i<=x; i++)
        bfs(x, i);
}

// Driver program
int main()
{
    int x = 40;
    printJumping(x);
    return 0;
}
```

Python 3

```
#Class queue for use later
class Queue:
    def __init__(self):
```

```

self.lst = []

def is_empty(self):
    return self.lst == []

def enqueue(self, elem):
    self.lst.append(elem)

def dequeue(self):
    return self.lst.pop(0)

# Prints all jumping numbers smaller than or equal to
# x starting with 'num'. It mainly does BFS starting
# from 'num'.
def bfs(x,num):

    # Create a queue and enqueue i to it
    q = Queue()
    q.enqueue(num)

    # Do BFS starting from 1
    while (not q.is_empty()):
        num = q.dequeue()

        if num<=x:
            print(str(num),end=' ')
            last_dig = num % 10

            # If last digit is 0, append next digit only
            if last_dig == 0:
                q.enqueue((num * 10) + (last_dig + 1))

            # If last digit is 9, append previous digit
            # only
            elif last_dig == 9:
                q.enqueue((num * 10) + (last_dig - 1))

            # If last digit is neighter 0 nor 9, append
            # both previous digit and next digit
            else:
                q.enqueue((num * 10) + (last_dig - 1))
                q.enqueue((num * 10) + (last_dig + 1))

# Prints all jumping numbers smaller than or equal to
# a positive number x
def printJumping(x):
    print (str(0), end=' ')
    for i in range(1,10):

```

```
bfs(x, i)

# Driver Program ( Change value of x as desired )
x = 40
printJumping(x)

# This code is contributed by Saket Modi
```

Output:

```
0 1 10 12 2 21 23 3 32 34 4 5 6 7 8 9
```

Thanks to Gaurav Ahirwar for above solution.

Exercise:

1. Change the above solution to use DFS instead of BFS.
2. Extend your solution to print all numbers in sorted order instead of any order.
3. Further extend the solution to print all numbers in a given range.

Improved By : [fruit_jam](#)

Source

<https://www.geeksforgeeks.org/print-all-jumping-numbers-smaller-than-or-equal-to-a-given-value/>

Chapter 40

Stable Marriage Problem

Stable Marriage Problem - GeeksforGeeks

The Stable Marriage Problem states that given N men and N women, where each person has ranked all members of the opposite sex in order of preference, marry the men and women together such that there are no two people of opposite sex who would both rather have each other than their current partners. If there are no such people, all the marriages are “stable” (Source [Wiki](#)).

Consider the following example.

Let there be two men m1 and m2 and two women w1 and w2.

Let m1's list of preferences be {w1, w2}

Let m2's list of preferences be {w1, w2}

Let w1's list of preferences be {m1, m2}

Let w2's list of preferences be {m1, m2}

The matching { {m1, w2}, {w1, m2} } is not stable because m1 and w1 would prefer each other over their assigned partners. The matching {m1, w1} and {m2, w2} is stable because there are no two people of opposite sex that would prefer each other over their assigned partners.

It is always possible to form stable marriages from lists of preferences (See references for proof). Following is **Gale-Shapley algorithm** to find a stable matching:

The idea is to iterate through all free men while there is any free man available. Every free man goes to all women in his preference list according to the order. For every woman he goes to, he checks if the woman is free, if yes, they both become engaged. If the woman is not free, then the woman chooses either says no to him or dumps her current engagement according to her preference list. So an engagement done once can be broken if a woman gets better option. Time Complexity of Gale-Shapley Algorithm is $O(n^2)$.

Following is complete algorithm from [Wiki](#)

```
Initialize all men and women to free
while there exist a free man m who still has a woman w to propose to
{
```

```

w = m's highest ranked such woman to whom he has not yet proposed
if w is free
    (m, w) become engaged
else some pair (m', w) already exists
    if w prefers m to m'
        (m, w) become engaged
        m' becomes free
    else
        (m', w) remain engaged
}
    
```

Input & Output: Input is a 2D matrix of size $(2*N)*N$ where N is number of women or men. Rows from 0 to N-1 represent preference lists of men and rows from N to $2*N - 1$ represent preference lists of women. So men are numbered from 0 to N-1 and women are numbered from N to $2*N - 1$. The output is list of married pairs.

Following is C++ implementation of the above algorithm.

```

// C++ program for stable marriage problem
#include <iostream>
#include <string.h>
#include <stdio.h>
using namespace std;

// Number of Men or Women
#define N 4

// This function returns true if woman 'w' prefers man 'm1' over man 'm'
bool wPrefersM1OverM(int prefer[2*N][N], int w, int m, int m1)
{
    // Check if w prefers m over her current engagement m1
    for (int i = 0; i < N; i++)
    {
        // If m1 comes before m in list of w, then w prefers her
        // current engagement, don't do anything
        if (prefer[w][i] == m1)
            return true;

        // If m comes before m1 in w's list, then free her current
        // engagement and engage her with m
        if (prefer[w][i] == m)
            return false;
    }
}

// Prints stable matching for N boys and N girls. Boys are numbered as 0 to
// N-1. Girls are numbered as N to 2N-1.
void stableMarriage(int prefer[2*N][N])
    
```

```

{
    // Stores partner of women. This is our output array that
    // stores pairing information. The value of wPartner[i]
    // indicates the partner assigned to woman N+i. Note that
    // the woman numbers between N and 2*N-1. The value -1
    // indicates that (N+i)'th woman is free
    int wPartner[N];

    // An array to store availability of men. If mFree[i] is
    // false, then man 'i' is free, otherwise engaged.
    bool mFree[N];

    // Initialize all men and women as free
    memset(wPartner, -1, sizeof(wPartner));
    memset(mFree, false, sizeof(mFree));
    int freeCount = N;

    // While there are free men
    while (freeCount > 0)
    {
        // Pick the first free man (we could pick any)
        int m;
        for (m = 0; m < N; m++)
            if (mFree[m] == false)
                break;

        // One by one go to all women according to m's preferences.
        // Here m is the picked free man
        for (int i = 0; i < N && mFree[m] == false; i++)
        {
            int w = prefer[m][i];

            // The woman of preference is free, w and m become
            // partners (Note that the partnership maybe changed
            // later). So we can say they are engaged not married
            if (wPartner[w-N] == -1)
            {
                wPartner[w-N] = m;
                mFree[m] = true;
                freeCount--;
            }

            else // If w is not free
            {
                // Find current engagement of w
                int m1 = wPartner[w-N];

                // If w prefers m over her current engagement m1,

```

Output:

References:

<http://www.csee.wvu.edu/~ksmani/courses/fa01/random/lecnotes/lecture5.pdf>

<http://www.youtube.com/watch?v=5RSMLgy06Ew#t=11m4s>

Improved By : ParulShandilya

Source

<https://www.geeksforgeeks.org/stable-marriage-problem/>

Chapter 41

Steiner Tree Problem

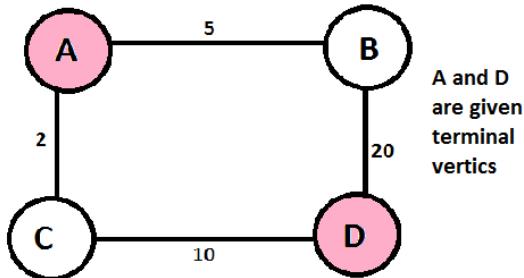
Steiner Tree Problem - GeeksforGeeks

What is Steiner Tree?

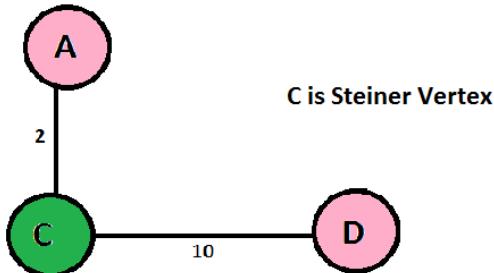
Given a graph and a **subset** of vertices in the graph, a steiner tree spans though the given subset. The Steiner Tree may contain some vertices which are not in given subset but are used to connect the vertices of subset.

The given set of vertices is called ***Terminal Vertices*** and other vertices that are used to construct Steiner tree are called ***Steiner vertices***.

The Steiner Tree Problem is to find the minimum cost Steiner Tree. See below for an example.



**Below is Minimum Steiner Tree for
above Graph**



Spanning Tree vs Steiner Tree

Minimum Spanning Tree is a minimum weight tree that spans through **all** vertices.

If given subset (or terminal) vertices is equal to set of all vertices in Steiner Tree problem, then the problem becomes Minimum Spanning Tree problem. And if the given subset contains only two vertices, then it shortest path problem between two vertices.

Finding out Minimum Spanning Tree is polynomial time solvable, but Minimum Steiner Tree problem is NP Hard and related decision problem is NP-Complete.

Applications of Steiner Tree

Any situation where the task is minimize cost of connection among some important locations like VLSI Design, Computer Networks, etc.

Shortest Path based Approximate Algorithm

Since Steiner Tree problem is NP-Hard, there are no polynomial time solutions that always give optimal cost. Therefore, there are approximate algorithms to solve the same. Below is one simple approximate algorithm.

- 1) Start with a subtree T consisting of one given terminal vertex
- 2) While T does not span all terminals
 - a) Select a terminal x not in T that is closest to a vertex in T.
 - b) Add to T the shortest path that connects x with T

The above algorithm is $(2 - 2/n)$ approximate, i.e., it guarantees that solution produced by this algorithm is not more than this ratio of optimized solution for a given graph with n vertices. There are better algorithms also that provide better ratio. Refer below reference for more details.

References:

www.cs.uu.nl/docs/vakken/an/teoud/an-steiner.ppt

This article is contributed by **Shivam Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

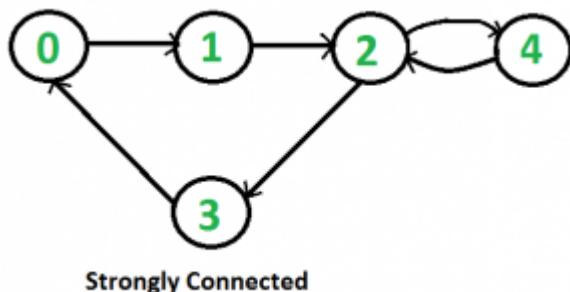
<https://www.geeksforgeeks.org/steiner-tree/>

Chapter 42

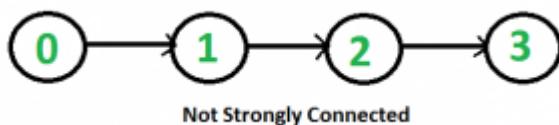
Connectivity in a directed graph

Check if a graph is strongly connected Set 1 (Kosaraju using DFS) - GeeksforGeeks

Given a directed graph, find out whether the graph is strongly connected or not. A directed graph is strongly connected if there is a path between any two pair of vertices. For example, following is a strongly connected graph.



It is easy for undirected graph, we can just do a BFS and DFS starting from any vertex. If BFS or DFS visits all vertices, then the given undirected graph is connected. This approach won't work for a directed graph. For example, consider the following graph which is not strongly connected. If we start DFS (or BFS) from vertex 0, we can reach all vertices, but if we start from any other vertex, we cannot reach all vertices.



How to do for directed graph?

A simple idea is to use a all pair shortest path algorithm like **Floyd Warshall** or find **Transitive Closure** of graph. Time complexity of this method would be $O(v^3)$.

We can also do **DFS V times** starting from every vertex. If any DFS, doesn't visit all vertices, then graph is not strongly connected. This algorithm takes $O(V^*(V+E))$ time which can be same as transitive closure for a dense graph.

A better idea can be **Strongly Connected Components (SCC) algorithm**. We can find all SCCs in $O(V+E)$ time. If number of SCCs is one, then graph is strongly connected. The algorithm for SCC does extra work as it finds all SCCs.

Following is **Kosaraju's DFS based simple algorithm that does two DFS traversals of graph:**

- 1) Initialize all vertices as not visited.
- 2) Do a DFS traversal of graph starting from any arbitrary vertex v. If DFS traversal doesn't visit all vertices, then return false.
- 3) Reverse all arcs (or find transpose or reverse of graph)
- 4) Mark all vertices as not-visited in reversed graph.
- 5) Do a DFS traversal of reversed graph starting from same vertex v (Same as step 2). If DFS traversal doesn't visit all vertices, then return false. Otherwise return true.

The idea is, if every node can be reached from a vertex v, and every node can reach v, then the graph is strongly connected. In step 2, we check if all vertices are reachable from v. In step 4, we check if all vertices can reach v (In reversed graph, if all vertices are reachable from v, then all vertices can reach v in original graph).

Following is the implementation of above algorithm.

C++

```
// C++ program to check if a given directed graph is strongly
// connected or not
#include <iostream>
#include <list>
#include <stack>
using namespace std;

class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // An array of adjacency lists

    // A recursive function to print DFS starting from v
    void DFSUtil(int v, bool visited[]);

public:
    // Constructor and Destructor
    Graph(int V) { this->V = V; adj = new list<int>[V]; }
    ~Graph() { delete [] adj; }

    // Method to add an edge
    void addEdge(int v, int w);
```

```

// The main function that returns true if the graph is strongly
// connected, otherwise false
bool isSC();

// Function that returns reverse (or transpose) of this graph
Graph getTranspose();
};

// A recursive function to print DFS starting from v
void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

// Function that returns reverse (or transpose) of this graph
Graph Graph::getTranspose()
{
    Graph g(V);
    for (int v = 0; v < V; v++)
    {
        // Recur for all the vertices adjacent to this vertex
        list<int>::iterator i;
        for(i = adj[v].begin(); i != adj[v].end(); ++i)
        {
            g.adj[*i].push_back(v);
        }
    }
    return g;
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

// The main function that returns true if graph is strongly connected
bool Graph::isSC()
{
    // Step 1: Mark all the vertices as not visited (For first DFS)
    bool visited[V];

```

```
for (int i = 0; i < V; i++)
    visited[i] = false;

// Step 2: Do DFS traversal starting from first vertex.
DFSUtil(0, visited);

// If DFS traversal doesn't visit all vertices, then return false.
for (int i = 0; i < V; i++)
    if (visited[i] == false)
        return false;

// Step 3: Create a reversed graph
Graph gr = getTranspose();

// Step 4: Mark all the vertices as not visited (For second DFS)
for(int i = 0; i < V; i++)
    visited[i] = false;

// Step 5: Do DFS for reversed graph starting from first vertex.
// Starting Vertex must be same starting point of first DFS
gr.DFSUtil(0, visited);

// If all vertices are not visited in second DFS, then
// return false
for (int i = 0; i < V; i++)
    if (visited[i] == false)
        return false;

return true;
}

// Driver program to test above functions
int main()
{
    // Create graphs given in the above diagrams
    Graph g1(5);
    g1.addEdge(0, 1);
    g1.addEdge(1, 2);
    g1.addEdge(2, 3);
    g1.addEdge(3, 0);
    g1.addEdge(2, 4);
    g1.addEdge(4, 2);
    g1.isSC()? cout << "Yes\n" : cout << "No\n";

    Graph g2(4);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.addEdge(2, 3);
```

```
g2.isSC()? cout << "Yes\n" : cout << "No\n";  
return 0;  
}
```

Java

```
// Java program to check if a given directed graph is strongly  
// connected or not  
import java.io.*;  
import java.util.*;  
import java.util.LinkedList;  
  
// This class represents a directed graph using adjacency  
// list representation  
class Graph  
{  
    private int V; // No. of vertices  
    private LinkedList<Integer> adj[]; //Adjacency List  
  
    //Constructor  
    Graph(int v)  
    {  
        V = v;  
        adj = new LinkedList[v];  
        for (int i=0; i<v; ++i)  
            adj[i] = new LinkedList();  
    }  
  
    //Function to add an edge into the graph  
    void addEdge(int v,int w) { adj[v].add(w); }  
  
    // A recursive function to print DFS starting from v  
    void DFSUtil(int v,Boolean visited[])  
    {  
        // Mark the current node as visited and print it  
        visited[v] = true;  
  
        int n;  
  
        // Recur for all the vertices adjacent to this vertex  
        Iterator<Integer> i = adj[v].iterator();  
        while (i.hasNext())  
        {  
            n = i.next();  
            if (!visited[n])  
                DFSUtil(n,visited);  
        }  
    }  
}
```

```
}

// Function that returns transpose of this graph
Graph getTranspose()
{
    Graph g = new Graph(V);
    for (int v = 0; v < V; v++)
    {
        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> i = adj[v].listIterator();
        while (i.hasNext())
            g.adj[i.next()].add(v);
    }
    return g;
}

// The main function that returns true if graph is strongly
// connected
Boolean isSC()
{
    // Step 1: Mark all the vertices as not visited
    // (For first DFS)
    Boolean visited[] = new Boolean[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Step 2: Do DFS traversal starting from first vertex.
    DFSUtil(0, visited);

    // If DFS traversal doesn't visit all vertices, then
    // return false.
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            return false;

    // Step 3: Create a reversed graph
    Graph gr = getTranspose();

    // Step 4: Mark all the vertices as not visited (For
    // second DFS)
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Step 5: Do DFS for reversed graph starting from
    // first vertex. Starting Vertex must be same starting
    // point of first DFS
    gr.DFSUtil(0, visited);
```

```
// If all vertices are not visited in second DFS, then
// return false
for (int i = 0; i < V; i++)
    if (visited[i] == false)
        return false;

    return true;
}

public static void main(String args[])
{
    // Create graphs given in the above diagrams
    Graph g1 = new Graph(5);
    g1.addEdge(0, 1);
    g1.addEdge(1, 2);
    g1.addEdge(2, 3);
    g1.addEdge(3, 0);
    g1.addEdge(2, 4);
    g1.addEdge(4, 2);
    if (g1.isSC())
        System.out.println("Yes");
    else
        System.out.println("No");

    Graph g2 = new Graph(4);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.addEdge(2, 3);
    if (g2.isSC())
        System.out.println("Yes");
    else
        System.out.println("No");
}
}

// This code is contributed by Aakash Hasija
```

Python

```
# Python program to check if a given directed graph is strongly
# connected or not

from collections import defaultdict

#This class represents a directed graph using adjacency list representation
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices
```

```
self.graph = defaultdict(list) # default dictionary to store graph

# function to add an edge to graph
def addEdge(self,u,v):
    self.graph[u].append(v)

#A function used by isSC() to perform DFS
def DFSUtil(self,v,visited):

    # Mark the current node as visited
    visited[v]= True

    #Recur for all the vertices adjacent to this vertex
    for i in self.graph[v]:
        if visited[i]==False:
            self.DFSUtil(i,visited)

# Function that returns reverse (or transpose) of this graph
def getTranspose(self):

    g = Graph(self.V)

    # Recur for all the vertices adjacent to this vertex
    for i in self.graph:
        for j in self.graph[i]:
            g.addEdge(j,i)

    return g

# The main function that returns true if graph is strongly connected
def isSC(self):

    # Step 1: Mark all the vertices as not visited (For first DFS)
    visited =[False]*(self.V)

    # Step 2: Do DFS traversal starting from first vertex.
    self.DFSUtil(0,visited)

    # If DFS traversal doesnt visit all vertices, then return false
    if any(i == False for i in visited):
        return False

    # Step 3: Create a reversed graph
    gr = self.getTranspose()
```

```

# Step 4: Mark all the vertices as not visited (For second DFS)
visited =[False]*(self.V)

# Step 5: Do DFS for reversed graph starting from first vertex.
# Starting Vertex must be same starting point of first DFS
gr.DFSUtil(0,visited)

# If all vertices are not visited in second DFS, then
# return false
if any(i == False for i in visited):
    return False

return True

# Create a graph given in the above diagram
g1 = Graph(5)
g1.addEdge(0, 1)
g1.addEdge(1, 2)
g1.addEdge(2, 3)
g1.addEdge(3, 0)
g1.addEdge(2, 4)
g1.addEdge(4, 2)
print "Yes" if g1.isSC() else "No"

g2 = Graph(4)
g2.addEdge(0, 1)
g2.addEdge(1, 2)
g2.addEdge(2, 3)
print "Yes" if g2.isSC() else "No"

#This code is contributed by Neelam Yadav

```

Output:

```

Yes
No

```

Time Complexity: Time complexity of above implementation is same as [Depth First Search](#) which is $O(V+E)$ if the graph is represented using adjacency list representation.

Can we improve further?

The above approach requires two traversals of graph. We can find whether a graph is strongly connected or not in one traversal using [Tarjan's Algorithm to find Strongly Connected Components](#).

Exercise:

Can we use BFS instead of DFS in above algorithm? See [this](#).

References:

<http://www.ieor.berkeley.edu/~hochbaum/files/ieor266-2012.pdf>

Source

<https://www.geeksforgeeks.org/connectivity-in-a-directed-graph/>

Chapter 43

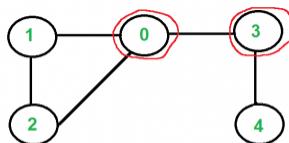
Articulation Points (or Cut Vertices) in a Graph

Articulation Points (or Cut Vertices) in a Graph - GeeksforGeeks

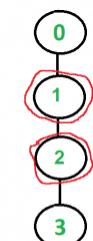
A vertex in an undirected connected graph is an articulation point (or cut vertex) iff removing it (and edges through it) disconnects the graph. Articulation points represent vulnerabilities in a connected network – single points whose failure would split the network into 2 or more disconnected components. They are useful for designing reliable networks.

For a disconnected undirected graph, an articulation point is a vertex removing which increases number of connected components.

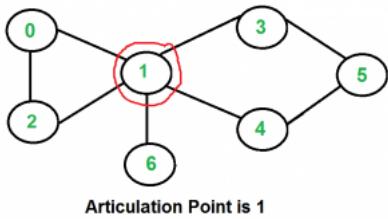
Following are some example graphs with articulation points encircled with red color.



Articulation points are 0 and 3



Artulation
Points are 1 & 2



How to find all articulation points in a given graph?

A simple approach is to one by one remove all vertices and see if removal of a vertex causes disconnected graph. Following are steps of simple approach for connected graph.

- 1) For every vertex v , do following
 -a) Remove v from graph
 -b) See if the graph remains connected (We can either use BFS or DFS)
 -c) Add v back to the graph

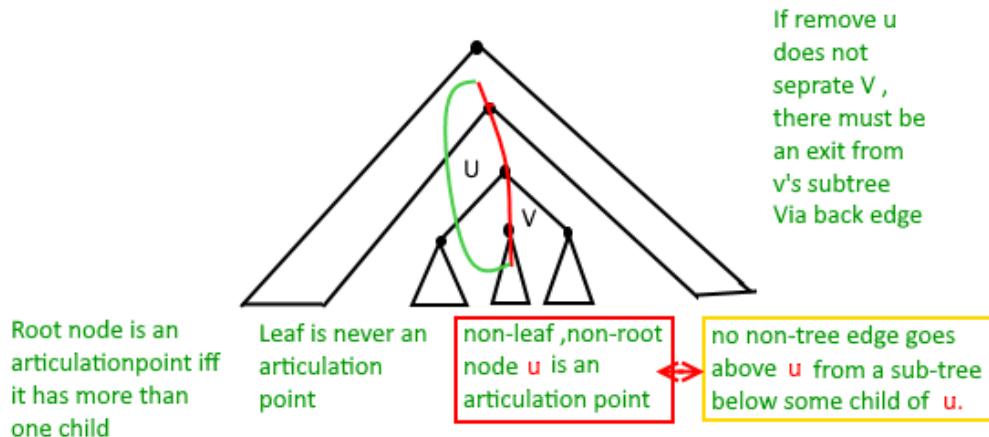
Time complexity of above method is $O(V*(V+E))$ for a graph represented using adjacency list. Can we do better?

A $O(V+E)$ algorithm to find all Articulation Points (APs)

The idea is to use DFS (Depth First Search). In DFS, we follow vertices in tree form called DFS tree. In DFS tree, a vertex u is parent of another vertex v , if v is discovered by u (obviously v is an adjacent of u in graph). In DFS tree, a vertex u is articulation point if one of the following two conditions is true.

- 1) u is root of DFS tree and it has at least two children.
- 2) u is not root of DFS tree and it has a child v such that no vertex in subtree rooted with v has a back edge to one of the ancestors (in DFS tree) of u .

Following figure shows same points as above with one additional point that a leaf in DFS Tree can never be an articulation point.



We do DFS traversal of given graph with additional code to find out Articulation Points

(APs). In DFS traversal, we maintain a parent[] array where parent[u] stores parent of vertex u. Among the above mentioned two cases, the first case is simple to detect. For every vertex, count children. If currently visited vertex u is root (parent[u] is NIL) and has more than two children, print it.

How to handle second case? The second case is trickier. We maintain an array disc[] to store discovery time of vertices. For every node u, we need to find out the earliest visited vertex (the vertex with minimum discovery time) that can be reached from subtree rooted with u. So we maintain an additional array low[] which is defined as follows.

```
low[u] = min(disc[u], disc[w])
where w is an ancestor of u and there is a back edge from
some descendant of u to w.
```

Following are C++, Java and Python implementation of Tarjan's algorithm for finding articulation points.

C++

```
// A C++ program to find articulation points in an undirected graph
#include<iostream>
#include <list>
#define NIL -1
using namespace std;

// A class that represents an undirected graph
class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // A dynamic array of adjacency lists
    void APUtil(int v, bool visited[], int disc[], int low[],
                int parent[], bool ap[]);
public:
    Graph(int V);    // Constructor
    void addEdge(int v, int w);    // function to add an edge to graph
    void AP();    // prints articulation points
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
    adj[w].push_back(v);  // Note: the graph is undirected
```

```

}

// A recursive function that find articulation points using DFS traversal
// u --> The vertex to be visited next
// visited[] --> keeps tract of visited vertices
// disc[] --> Stores discovery times of visited vertices
// parent[] --> Stores parent vertices in DFS tree
// ap[] --> Store articulation points
void Graph::APUtil(int u, bool visited[], int disc[],
                    int low[], int parent[], bool ap[])
{
    // A static variable is used for simplicity, we can avoid use of static
    // variable by passing a pointer.
    static int time = 0;

    // Count of children in DFS Tree
    int children = 0;

    // Mark the current node as visited
    visited[u] = true;

    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;

    // Go through all vertices adjacent to this
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        int v = *i; // v is current adjacent of u

        // If v is not visited yet, then make it a child of u
        // in DFS tree and recur for it
        if (!visited[v])
        {
            children++;
            parent[v] = u;
            APUtil(v, visited, disc, low, parent, ap);

            // Check if the subtree rooted with v has a connection to
            // one of the ancestors of u
            low[u] = min(low[u], low[v]);
        }
    }

    // u is an articulation point in following cases

    // (1) u is root of DFS tree and has two or more children.
    if (parent[u] == NIL && children > 1)
        ap[u] = true;
}

```

```

// (2) If u is not root and low value of one of its child is more
// than discovery value of u.
if (parent[u] != NIL && low[v] >= disc[u])
    ap[u] = true;
}

// Update low value of u for parent function calls.
else if (v != parent[u])
    low[u] = min(low[u], disc[v]);
}

// The function to do DFS traversal. It uses recursive function APUtil()
void Graph::AP()
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    int *disc = new int[V];
    int *low = new int[V];
    int *parent = new int[V];
    bool *ap = new bool[V]; // To store articulation points

    // Initialize parent and visited, and ap(articulation point) arrays
    for (int i = 0; i < V; i++)
    {
        parent[i] = NIL;
        visited[i] = false;
        ap[i] = false;
    }

    // Call the recursive helper function to find articulation points
    // in DFS tree rooted with vertex 'i'
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            APUtil(i, visited, disc, low, parent, ap);

    // Now ap[] contains articulation points, print them
    for (int i = 0; i < V; i++)
        if (ap[i] == true)
            cout << i << " ";
}

// Driver program to test above function
int main()
{
    // Create graphs given in above diagrams
    cout << "\nArticulation points in first graph \n";
    Graph g1(5);
}

```

```
g1.addEdge(1, 0);
g1.addEdge(0, 2);
g1.addEdge(2, 1);
g1.addEdge(0, 3);
g1.addEdge(3, 4);
g1.AP();

cout << "\nArticulation points in second graph \n";
Graph g2(4);
g2.addEdge(0, 1);
g2.addEdge(1, 2);
g2.addEdge(2, 3);
g2.AP();

cout << "\nArticulation points in third graph \n";
Graph g3(7);
g3.addEdge(0, 1);
g3.addEdge(1, 2);
g3.addEdge(2, 0);
g3.addEdge(1, 3);
g3.addEdge(1, 4);
g3.addEdge(1, 6);
g3.addEdge(3, 5);
g3.addEdge(4, 5);
g3.AP();

return 0;
}
```

Java

```
// A Java program to find articulation points in an undirected graph
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents an undirected graph using adjacency list
// representation
class Graph
{
    private int V;    // No. of vertices

    // Array of lists for Adjacency List Representation
    private LinkedList<Integer> adj[];
    int time = 0;
    static final int NIL = -1;

    // Constructor
```

```

Graph(int v)
{
    V = v;
    adj = new LinkedList[v];
    for (int i=0; i<v; ++i)
        adj[i] = new LinkedList();
}

//Function to add an edge into the graph
void addEdge(int v, int w)
{
    adj[v].add(w); // Add w to v's list.
    adj[w].add(v); //Add v to w's list
}

// A recursive function that find articulation points using DFS
// u --> The vertex to be visited next
// visited[] --> keeps tract of visited vertices
// disc[] --> Stores discovery times of visited vertices
// parent[] --> Stores parent vertices in DFS tree
// ap[] --> Store articulation points
void APUtil(int u, boolean visited[], int disc[],
            int low[], int parent[], boolean ap[])
{

    // Count of children in DFS Tree
    int children = 0;

    // Mark the current node as visited
    visited[u] = true;

    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;

    // Go through all vertices aadjacent to this
    Iterator<Integer> i = adj[u].iterator();
    while (i.hasNext())
    {
        int v = i.next(); // v is current adjacent of u

        // If v is not visited yet, then make it a child of u
        // in DFS tree and recur for it
        if (!visited[v])
        {
            children++;
            parent[v] = u;
            APUtil(v, visited, disc, low, parent, ap);
        }
    }
}

```

```

// Check if the subtree rooted with v has a connection to
// one of the ancestors of u
low[u] = Math.min(low[u], low[v]);

// u is an articulation point in following cases

// (1) u is root of DFS tree and has two or more children.
if (parent[u] == NIL && children > 1)
    ap[u] = true;

// (2) If u is not root and low value of one of its child
// is more than discovery value of u.
if (parent[u] != NIL && low[v] >= disc[u])
    ap[u] = true;
}

// Update low value of u for parent function calls.
else if (v != parent[u])
    low[u] = Math.min(low[u], disc[v]);
}

}

// The function to do DFS traversal. It uses recursive function APUtil()
void AP()
{
    // Mark all the vertices as not visited
    boolean visited[] = new boolean[V];
    int disc[] = new int[V];
    int low[] = new int[V];
    int parent[] = new int[V];
    boolean ap[] = new boolean[V]; // To store articulation points

    // Initialize parent and visited, and ap(articulation point)
    // arrays
    for (int i = 0; i < V; i++)
    {
        parent[i] = NIL;
        visited[i] = false;
        ap[i] = false;
    }

    // Call the recursive helper function to find articulation
    // points in DFS tree rooted with vertex 'i'
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            APUtil(i, visited, disc, low, parent, ap);

    // Now ap[] contains articulation points, print them
}

```

```
        for (int i = 0; i < V; i++)
            if (ap[i] == true)
                System.out.print(i+" ");
    }

    // Driver method
    public static void main(String args[])
    {
        // Create graphs given in above diagrams
        System.out.println("Articulation points in first graph ");
        Graph g1 = new Graph(5);
        g1.addEdge(1, 0);
        g1.addEdge(0, 2);
        g1.addEdge(2, 1);
        g1.addEdge(0, 3);
        g1.addEdge(3, 4);
        g1.AP();
        System.out.println();

        System.out.println("Articulation points in Second graph");
        Graph g2 = new Graph(4);
        g2.addEdge(0, 1);
        g2.addEdge(1, 2);
        g2.addEdge(2, 3);
        g2.AP();
        System.out.println();

        System.out.println("Articulation points in Third graph ");
        Graph g3 = new Graph(7);
        g3.addEdge(0, 1);
        g3.addEdge(1, 2);
        g3.addEdge(2, 0);
        g3.addEdge(1, 3);
        g3.addEdge(1, 4);
        g3.addEdge(1, 6);
        g3.addEdge(3, 5);
        g3.addEdge(4, 5);
        g3.AP();
    }
}

// This code is contributed by Aakash Hasija
```

Python

```
# Python program to find articulation points in an undirected graph

from collections import defaultdict
```

```

#This class represents an undirected graph
#using adjacency list representation
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices
        self.graph = defaultdict(list) # default dictionary to store graph
        self.Time = 0

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)
        self.graph[v].append(u)

    '''A recursive function that find articulation points
    using DFS traversal
    u --> The vertex to be visited next
    visited[] --> keeps tract of visited vertices
    disc[] --> Stores discovery times of visited vertices
    parent[] --> Stores parent vertices in DFS tree
    ap[] --> Store articulation points'''
    def APUtil(self,u, visited, ap, parent, low, disc):

        #Count of children in current node
        children =0

        # Mark the current node as visited and print it
        visited[u]= True

        # Initialize discovery time and low value
        disc[u] = self.Time
        low[u] = self.Time
        self.Time += 1

        #Recur for all the vertices adjacent to this vertex
        for v in self.graph[u]:
            # If v is not visited yet, then make it a child of u
            # in DFS tree and recur for it
            if visited[v] == False :
                parent[v] = u
                children += 1
                self.APUtil(v, visited, ap, parent, low, disc)

                # Check if the subtree rooted with v has a connection to
                # one of the ancestors of u
                low[u] = min(low[u], low[v])

        # u is an articulation point in following cases
        if children > 1:
            ap.append(u)

```

```

# (1) u is root of DFS tree and has two or more children.
if parent[u] == -1 and children > 1:
    ap[u] = True

#(2) If u is not root and low value of one of its child is more
# than discovery value of u.
if parent[u] != -1 and low[v] >= disc[u]:
    ap[u] = True

    # Update low value of u for parent function calls
elif v != parent[u]:
    low[u] = min(low[u], disc[v])

```

#The function to do DFS traversal. It uses recursive APUtil()

```

def AP(self):

    # Mark all the vertices as not visited
    # and Initialize parent and visited,
    # and ap(articulation point) arrays
    visited = [False] * (self.V)
    disc = [float("Inf")] * (self.V)
    low = [float("Inf")] * (self.V)
    parent = [-1] * (self.V)
    ap = [False] * (self.V) #To store articulation points

    # Call the recursive helper function
    # to find articulation points
    # in DFS tree rooted with vertex 'i'
    for i in range(self.V):
        if visited[i] == False:
            self.APUtil(i, visited, ap, parent, low, disc)

    for index, value in enumerate (ap):
        if value == True: print index,

    # Create a graph given in the above diagram
g1 = Graph(5)
g1.addEdge(1, 0)
g1.addEdge(0, 2)
g1.addEdge(2, 1)
g1.addEdge(0, 3)
g1.addEdge(3, 4)

print "\nArticulation points in first graph "
g1.AP()

g2 = Graph(4)

```

```
g2.addEdge(0, 1)
g2.addEdge(1, 2)
g2.addEdge(2, 3)
print "\nArticulation points in second graph "
g2.AP()

g3 = Graph (7)
g3.addEdge(0, 1)
g3.addEdge(1, 2)
g3.addEdge(2, 0)
g3.addEdge(1, 3)
g3.addEdge(1, 4)
g3.addEdge(1, 6)
g3.addEdge(3, 5)
g3.addEdge(4, 5)
print "\nArticulation points in third graph "
g3.AP()

#This code is contributed by Neelam Yadav
```

Output:

```
Articulation points in first graph
0 3
Articulation points in second graph
1 2
Articulation points in third graph
1
```

Time Complexity: The above function is simple DFS with additional arrays. So time complexity is same as DFS which is $O(V+E)$ for adjacency list representation of graph.

References:

<https://www.cs.washington.edu/education/courses/421/04su/slides/artic.pdf>
<http://www.slideshare.net/TraianRebedea/algorithm-design-and-complexity-course-8>
http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250_Weiss/L25-Connectivity.htm

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

<https://www.geeksforgeeks.org/articulation-points-or-cut-vertices-in-a-graph/>

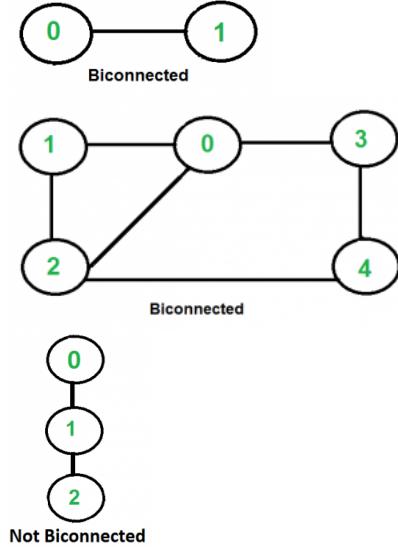
Chapter 44

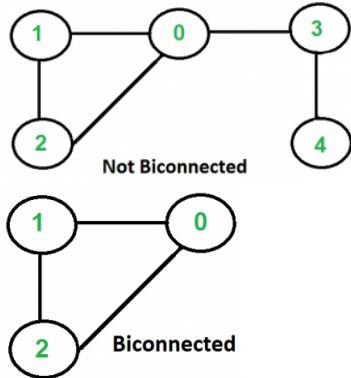
Biconnected graph

Biconnected graph - GeeksforGeeks

An undirected graph is called Biconnected if there are two vertex-disjoint paths between any two vertices. In a Biconnected Graph, there is a simple cycle through any two vertices. By convention, two nodes connected by an edge form a biconnected graph, but this does not verify the above properties. For a graph with more than two vertices, the above properties must be there for it to be Biconnected.

Following are some examples.





See [this](#)for more examples.

How to find if a given graph is Biconnected or not?

A connected graph is Biconnected if it is connected and doesn't have any Articulation Point.

We mainly need to check two things in a graph.

- 1) The graph is connected.
- 2) There is not articulation point in graph.

We start from any vertex and do DFS traversal. In DFS traversal, we check if there is any articulation point. If we don't find any articulation point, then the graph is Biconnected. Finally, we need to check whether all vertices were reachable in DFS or not. If all vertices were not reachable, then the graph is not even connected.

Following is C++ implementation of above approach.

C++

```
// A C++ program to find if a given undirected graph is
// biconnected
#include<iostream>
#include <list>
#define NIL -1
using namespace std;

// A class that represents an undirected graph
class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // A dynamic array of adjacency lists
    bool isBCUtil(int v, bool visited[], int disc[], int low[],
                  int parent[]);
public:
    Graph(int V);    // Constructor
    void addEdge(int v, int w); // to add an edge to graph
    bool isBC();     // returns true if graph is Biconnected
};
```

```

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
    adj[w].push_back(v); // Note: the graph is undirected
}

// A recursive function that returns true if there is an articulation
// point in given graph, otherwise returns false.
// This function is almost same as isAPUtil() here ( http://goo.gl/Me9Fw )
// u --> The vertex to be visited next
// visited[] --> keeps tract of visited vertices
// disc[] --> Stores discovery times of visited vertices
// parent[] --> Stores parent vertices in DFS tree
bool Graph::isBCUtil(int u, bool visited[], int disc[], int low[], int parent[])
{
    // A static variable is used for simplicity, we can avoid use of static
    // variable by passing a pointer.
    static int time = 0;

    // Count of children in DFS Tree
    int children = 0;

    // Mark the current node as visited
    visited[u] = true;

    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;

    // Go through all vertices adjacent to this
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        int v = *i; // v is current adjacent of u

        // If v is not visited yet, then make it a child of u
        // in DFS tree and recur for it
        if (!visited[v])
        {
            children++;
            parent[v] = u;

            // check if subgraph rooted with v has an articulation point
        }
    }
}

```

```

        if (isBCUtil(v, visited, disc, low, parent))
            return true;

        // Check if the subtree rooted with v has a connection to
        // one of the ancestors of u
        low[u] = min(low[u], low[v]);

        // u is an articulation point in following cases

        // (1) u is root of DFS tree and has two or more children.
        if (parent[u] == NIL && children > 1)
            return true;

        // (2) If u is not root and low value of one of its child is
        // more than discovery value of u.
        if (parent[u] != NIL && low[v] >= disc[u])
            return true;
    }

    // Update low value of u for parent function calls.
    else if (v != parent[u])
        low[u] = min(low[u], disc[v]);
}
return false;
}

// The main function that returns true if graph is Biconnected,
// otherwise false. It uses recursive function isBCUtil()
bool Graph::isBC()
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    int *disc = new int[V];
    int *low = new int[V];
    int *parent = new int[V];

    // Initialize parent and visited, and ap(articulation point)
    // arrays
    for (int i = 0; i < V; i++)
    {
        parent[i] = NIL;
        visited[i] = false;
    }

    // Call the recursive helper function to find if there is an articulation
    // point in given graph. We do DFS traversal starting from vertex 0
    if (isBCUtil(0, visited, disc, low, parent) == true)
        return false;
}

```

```
// Now check whether the given graph is connected or not. An undirected
// graph is connected if all vertices are reachable from any starting
// point (we have taken 0 as starting point)
for (int i = 0; i < V; i++)
    if (visited[i] == false)
        return false;

return true;
}

// Driver program to test above function
int main()
{
    // Create graphs given in above diagrams
    Graph g1(2);
    g1.addEdge(0, 1);
    g1.isBC()? cout << "Yes\n" : cout << "No\n";

    Graph g2(5);
    g2.addEdge(1, 0);
    g2.addEdge(0, 2);
    g2.addEdge(2, 1);
    g2.addEdge(0, 3);
    g2.addEdge(3, 4);
    g2.addEdge(2, 4);
    g2.isBC()? cout << "Yes\n" : cout << "No\n";

    Graph g3(3);
    g3.addEdge(0, 1);
    g3.addEdge(1, 2);
    g3.isBC()? cout << "Yes\n" : cout << "No\n";

    Graph g4(5);
    g4.addEdge(1, 0);
    g4.addEdge(0, 2);
    g4.addEdge(2, 1);
    g4.addEdge(0, 3);
    g4.addEdge(3, 4);
    g4.isBC()? cout << "Yes\n" : cout << "No\n";

    Graph g5(3);
    g5.addEdge(0, 1);
    g5.addEdge(1, 2);
    g5.addEdge(2, 0);
    g5.isBC()? cout << "Yes\n" : cout << "No\n";

    return 0;
}
```

}

Java

```
// A Java program to find if a given undirected graph is
// biconnected
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents a directed graph using adjacency
// list representation
class Graph
{
    private int V;    // No. of vertices

    // Array of lists for Adjacency List Representation
    private LinkedList<Integer> adj[];

    int time = 0;
    static final int NIL = -1;

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    //Function to add an edge into the graph
    void addEdge(int v, int w)
    {
        adj[v].add(w);  //Note that the graph is undirected.
        adj[w].add(v);
    }

    // A recursive function that returns true if there is an articulation
    // point in given graph, otherwise returns false.
    // This function is almost same as isAPUtil() @ http://goo.gl/Me9Fw
    // u --> The vertex to be visited next
    // visited[] --> keeps tract of visited vertices
    // disc[] --> Stores discovery times of visited vertices
    // parent[] --> Stores parent vertices in DFS tree
    boolean isBCUtil(int u, boolean visited[], int disc[],int low[],
                     int parent[])
    {
```

```

// Count of children in DFS Tree
int children = 0;

// Mark the current node as visited
visited[u] = true;

// Initialize discovery time and low value
disc[u] = low[u] = ++time;

// Go through all vertices adjacent to this
Iterator<Integer> i = adj[u].iterator();
while (i.hasNext())
{
    int v = i.next(); // v is current adjacent of u

    // If v is not visited yet, then make it a child of u
    // in DFS tree and recur for it
    if (!visited[v])
    {
        children++;
        parent[v] = u;

        // check if subgraph rooted with v has an articulation point
        if (isBCUtil(v, visited, disc, low, parent))
            return true;

        // Check if the subtree rooted with v has a connection to
        // one of the ancestors of u
        low[u] = Math.min(low[u], low[v]);
    }

    // u is an articulation point in following cases

    // (1) u is root of DFS tree and has two or more children.
    if (parent[u] == NIL && children > 1)
        return true;

    // (2) If u is not root and low value of one of its
    // child is more than discovery value of u.
    if (parent[u] != NIL && low[v] >= disc[u])
        return true;
}

// Update low value of u for parent function calls.
else if (v != parent[u])
    low[u] = Math.min(low[u], disc[v]);
}

return false;
}

```

```
}

// The main function that returns true if graph is Biconnected,
// otherwise false. It uses recursive function isBCUtil()
boolean isBC()
{
    // Mark all the vertices as not visited
    boolean visited[] = new boolean[V];
    int disc[] = new int[V];
    int low[] = new int[V];
    int parent[] = new int[V];

    // Initialize parent and visited, and ap(articulation point)
    // arrays
    for (int i = 0; i < V; i++)
    {
        parent[i] = NIL;
        visited[i] = false;
    }

    // Call the recursive helper function to find if there is an
    // articulation/ point in given graph. We do DFS traversal
    // starring from vertex 0
    if (isBCUtil(0, visited, disc, low, parent) == true)
        return false;

    // Now check whether the given graph is connected or not.
    // An undirected graph is connected if all vertices are
    // reachable from any starting point (we have taken 0 as
    // starting point)
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            return false;

    return true;
}

// Driver method
public static void main(String args[])
{
    // Create graphs given in above diagrams
    Graph g1 =new Graph(2);
    g1.addEdge(0, 1);
    if (g1.isBC())
        System.out.println("Yes");
    else
        System.out.println("No");
}
```

```
Graph g2 =new Graph(5);
g2.addEdge(1, 0);
g2.addEdge(0, 2);
g2.addEdge(2, 1);
g2.addEdge(0, 3);
g2.addEdge(3, 4);
g2.addEdge(2, 4);
if (g2.isBC())
    System.out.println("Yes");
else
    System.out.println("No");

Graph g3 = new Graph(3);
g3.addEdge(0, 1);
g3.addEdge(1, 2);
if (g3.isBC())
    System.out.println("Yes");
else
    System.out.println("No");

Graph g4 = new Graph(5);
g4.addEdge(1, 0);
g4.addEdge(0, 2);
g4.addEdge(2, 1);
g4.addEdge(0, 3);
g4.addEdge(3, 4);
if (g4.isBC())
    System.out.println("Yes");
else
    System.out.println("No");

Graph g5= new Graph(3);
g5.addEdge(0, 1);
g5.addEdge(1, 2);
g5.addEdge(2, 0);
if (g5.isBC())
    System.out.println("Yes");
else
    System.out.println("No");
}

}

// This code is contributed by Aakash Hasija
```

Python

```
# Python program to find if a given undirected graph is
# biconnected
```

```

from collections import defaultdict

#This class represents an undirected graph using adjacency list representation
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices
        self.graph = defaultdict(list) # default dictionary to store graph
        self.Time = 0

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)
        self.graph[v].append(u)

    '''A recursive function that returns true if there is an articulation
    point in given graph, otherwise returns false.
    This function is almost same as isAPUtil()
    u --> The vertex to be visited next
    visited[] --> keeps tract of visited vertices
    disc[] --> Stores discovery times of visited vertices
    parent[] --> Stores parent vertices in DFS tree'''
    def isBCUtil(self,u, visited, parent, low, disc):

        #Count of children in current node
        children =0

        # Mark the current node as visited and print it
        visited[u]= True

        # Initialize discovery time and low value
        disc[u] = self.Time
        low[u] = self.Time
        self.Time += 1

        #Recur for all the vertices adjacent to this vertex
        for v in self.graph[u]:
            # If v is not visited yet, then make it a child of u
            # in DFS tree and recur for it
            if visited[v] == False :
                parent[v] = u
                children += 1
                if self.isBCUtil(v, visited, parent, low, disc):
                    return True

            # Check if the subtree rooted with v has a connection to
            # one of the ancestors of u
            low[u] = min(low[u], low[v])


```

```

# u is an articulation point in following cases
# (1) u is root of DFS tree and has two or more children.
if parent[u] == -1 and children > 1:
    return True

#(2) If u is not root and low value of one of its child is more
# than discovery value of u.
if parent[u] != -1 and low[v] >= disc[u]:
    return True

elif v != parent[u]: # Update low value of u for parent function calls.
    low[u] = min(low[u], disc[v])

return False

# The main function that returns true if graph is Biconnected,
# otherwise false. It uses recursive function isBCUtil()
def isBC(self):

    # Mark all the vertices as not visited and Initialize parent and visited,
    # and ap(articulation point) arrays
    visited = [False] * (self.V)
    disc = [float("Inf")] * (self.V)
    low = [float("Inf")] * (self.V)
    parent = [-1] * (self.V)

    # Call the recursive helper function to find if there is an
    # articulation points in given graph. We do DFS traversal starting
    # from vertex 0
    if self.isBCUtil(0, visited, parent, low, disc):
        return False

    '''Now check whether the given graph is connected or not.
    An undirected graph is connected if all vertices are
    reachable from any starting point (we have taken 0 as
    starting point)'''
    if any(i == False for i in visited):
        return False

    return True

# Create a graph given in the above diagram
g1 = Graph(2)
g1.addEdge(0, 1)
print "Yes" if g1.isBC() else "No"

```

```
g2 = Graph(5)
g2.addEdge(1, 0)
g2.addEdge(0, 2)
g2.addEdge(2, 1)
g2.addEdge(0, 3)
g2.addEdge(3, 4)
g2.addEdge(2, 4)
print "Yes" if g2.isBC() else "No"

g3 = Graph(3)
g3.addEdge(0, 1)
g3.addEdge(1, 2)
print "Yes" if g3.isBC() else "No"

g4 = Graph (5)
g4.addEdge(1, 0)
g4.addEdge(0, 2)
g4.addEdge(2, 1)
g4.addEdge(0, 3)
g4.addEdge(3, 4)
print "Yes" if g4.isBC() else "No"

g5 = Graph(3)
g5.addEdge(0, 1)
g5.addEdge(1, 2)
g5.addEdge(2, 0)
print "Yes" if g5.isBC() else "No"

#This code is contributed by Neelam Yadav
```

Output:

```
Yes
Yes
No
No
Yes
```

Time Complexity: The above function is a simple DFS with additional arrays. So time complexity is same as DFS which is $O(V+E)$ for adjacency list representation of graph.

References:

<http://www.cs.purdue.edu/homes/ayg/CS251/slides/chap9d.pdf>

Source

<https://www.geeksforgeeks.org/biconnectivity-in-a-graph/>

Chapter 45

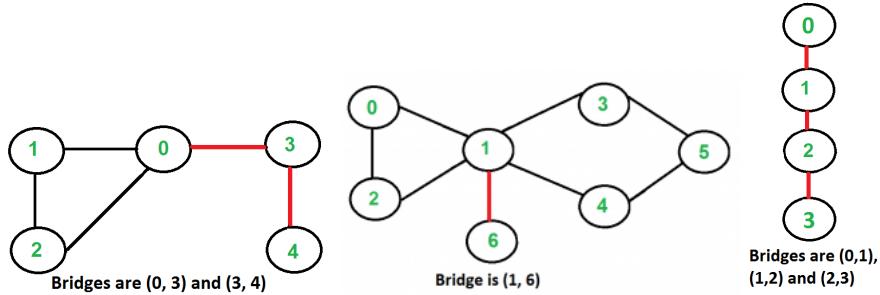
Bridges in a graph

Bridges in a graph - GeeksforGeeks

An edge in an undirected connected graph is a bridge iff removing it disconnects the graph. For a disconnected undirected graph, definition is similar, a bridge is an edge removing which increases number of disconnected components.

Like [Articulation Points](#), bridges represent vulnerabilities in a connected network and are useful for designing reliable networks. For example, in a wired computer network, an articulation point indicates the critical computers and a bridge indicates the critical wires or connections.

Following are some example graphs with bridges highlighted with red color.



How to find all bridges in a given graph?

A simple approach is to one by one remove all edges and see if removal of a edge causes disconnected graph. Following are steps of simple approach for connected graph.

- 1) For every edge (u, v) , do following
 -a) Remove (u, v) from graph
 -b) See if the graph remains connected (We can either use BFS or DFS)
 -c) Add (u, v) back to the graph.

Time complexity of above method is $O(E^*(V+E))$ for a graph represented using adjacency list. Can we do better?

A O(V+E) algorithm to find all Bridges

The idea is similar to [O\(V+E\) algorithm for Articulation Points](#). We do DFS traversal of the given graph. In DFS tree an edge (u, v) (u is parent of v in DFS tree) is bridge if there does not exist any other alternative to reach u or an ancestor of u from subtree rooted with v . As discussed in the [previous post](#), the value $\text{low}[v]$ indicates earliest visited vertex reachable from subtree rooted with v . *The condition for an edge (u, v) to be a bridge is, $\text{low}[v] > \text{disc}[u]$.*

Following are C++ and Java implementations of above approach.

C++

```
// A C++ program to find bridges in a given undirected graph
#include<iostream>
#include <list>
#define NIL -1
using namespace std;

// A class that represents an undirected graph
class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // A dynamic array of adjacency lists
    void bridgeUtil(int v, bool visited[], int disc[], int low[],
                    int parent[]);
public:
    Graph(int V);    // Constructor
    void addEdge(int v, int w);   // to add an edge to graph
    void bridge();    // prints all bridges
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
    adj[w].push_back(v);  // Note: the graph is undirected
}

// A recursive function that finds and prints bridges using
// DFS traversal
// u --> The vertex to be visited next
// visited[] --> keeps tract of visited vertices
// disc[] --> Stores discovery times of visited vertices
// parent[] --> Stores parent vertices in DFS tree
```

```

void Graph::bridgeUtil(int u, bool visited[], int disc[],
                      int low[], int parent[])
{
    // A static variable is used for simplicity, we can
    // avoid use of static variable by passing a pointer.
    static int time = 0;

    // Mark the current node as visited
    visited[u] = true;

    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;

    // Go through all vertices adjacent to this
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        int v = *i; // v is current adjacent of u

        // If v is not visited yet, then recur for it
        if (!visited[v])
        {
            parent[v] = u;
            bridgeUtil(v, visited, disc, low, parent);

            // Check if the subtree rooted with v has a
            // connection to one of the ancestors of u
            low[u] = min(low[u], low[v]);

            // If the lowest vertex reachable from subtree
            // under v is below u in DFS tree, then u-v
            // is a bridge
            if (low[v] > disc[u])
                cout << u << " " << v << endl;
        }

        // Update low value of u for parent function calls.
        else if (v != parent[u])
            low[u] = min(low[u], disc[v]);
    }
}

// DFS based function to find all bridges. It uses recursive
// function bridgeUtil()
void Graph::bridge()
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];

```

```
int *disc = new int[V];
int *low = new int[V];
int *parent = new int[V];

// Initialize parent and visited arrays
for (int i = 0; i < V; i++)
{
    parent[i] = NIL;
    visited[i] = false;
}

// Call the recursive helper function to find Bridges
// in DFS tree rooted with vertex 'i'
for (int i = 0; i < V; i++)
    if (visited[i] == false)
        bridgeUtil(i, visited, disc, low, parent);
}

// Driver program to test above function
int main()
{
    // Create graphs given in above diagrams
    cout << "\nBridges in first graph \n";
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 1);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.bridge();

    cout << "\nBridges in second graph \n";
    Graph g2(4);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.addEdge(2, 3);
    g2.bridge();

    cout << "\nBridges in third graph \n";
    Graph g3(7);
    g3.addEdge(0, 1);
    g3.addEdge(1, 2);
    g3.addEdge(2, 0);
    g3.addEdge(1, 3);
    g3.addEdge(1, 4);
    g3.addEdge(1, 6);
    g3.addEdge(3, 5);
    g3.addEdge(4, 5);
```

```
    g3.bridge();

    return 0;
}
```

Java

```
// A Java program to find bridges in a given undirected graph
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents a undirected graph using adjacency list
// representation
class Graph
{
    private int V;    // No. of vertices

    // Array of lists for Adjacency List Representation
    private LinkedList<Integer> adj[];
    int time = 0;
    static final int NIL = -1;

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    // Function to add an edge into the graph
    void addEdge(int v, int w)
    {
        adj[v].add(w);  // Add w to v's list.
        adj[w].add(v);  //Add v to w's list
    }

    // A recursive function that finds and prints bridges
    // using DFS traversal
    // u --> The vertex to be visited next
    // visited[] --> keeps tract of visited vertices
    // disc[] --> Stores discovery times of visited vertices
    // parent[] --> Stores parent vertices in DFS tree
    void bridgeUtil(int u, boolean visited[], int disc[],
                    int low[], int parent[])
    {
```

```

// Mark the current node as visited
visited[u] = true;

// Initialize discovery time and low value
disc[u] = low[u] = ++time;

// Go through all vertices adjacent to this
Iterator<Integer> i = adj[u].iterator();
while (i.hasNext())
{
    int v = i.next(); // v is current adjacent of u

    // If v is not visited yet, then make it a child
    // of u in DFS tree and recur for it.
    // If v is not visited yet, then recur for it
    if (!visited[v])
    {
        parent[v] = u;
        bridgeUtil(v, visited, disc, low, parent);

        // Check if the subtree rooted with v has a
        // connection to one of the ancestors of u
        low[u] = Math.min(low[u], low[v]);

        // If the lowest vertex reachable from subtree
        // under v is below u in DFS tree, then u-v is
        // a bridge
        if (low[v] > disc[u])
            System.out.println(u+" "+v);
    }

    // Update low value of u for parent function calls.
    else if (v != parent[u])
        low[u] = Math.min(low[u], disc[v]);
}
}

// DFS based function to find all bridges. It uses recursive
// function bridgeUtil()
void bridge()
{
    // Mark all the vertices as not visited
    boolean visited[] = new boolean[V];
    int disc[] = new int[V];
    int low[] = new int[V];
    int parent[] = new int[V];
}

```

```
// Initialize parent and visited, and ap(articulation point)
// arrays
for (int i = 0; i < V; i++)
{
    parent[i] = NIL;
    visited[i] = false;
}

// Call the recursive helper function to find Bridges
// in DFS tree rooted with vertex 'i'
for (int i = 0; i < V; i++)
    if (visited[i] == false)
        bridgeUtil(i, visited, disc, low, parent);
}

public static void main(String args[])
{
    // Create graphs given in above diagrams
    System.out.println("Bridges in first graph ");
    Graph g1 = new Graph(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 1);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.bridge();
    System.out.println();

    System.out.println("Bridges in Second graph");
    Graph g2 = new Graph(4);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.addEdge(2, 3);
    g2.bridge();
    System.out.println();

    System.out.println("Bridges in Third graph ");
    Graph g3 = new Graph(7);
    g3.addEdge(0, 1);
    g3.addEdge(1, 2);
    g3.addEdge(2, 0);
    g3.addEdge(1, 3);
    g3.addEdge(1, 4);
    g3.addEdge(1, 6);
    g3.addEdge(3, 5);
    g3.addEdge(4, 5);
```

```
        g3.bridge();
    }
}

// This code is contributed by Aakash Hasija
```

Python

```
# Python program to find bridges in a given undirected graph
#Complexity : O(V+E)

from collections import defaultdict

#This class represents an undirected graph using adjacency list representation
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices
        self.graph = defaultdict(list) # default dictionary to store graph
        self.Time = 0

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)
        self.graph[v].append(u)

    '''A recursive function that finds and prints bridges
    using DFS traversal
    u --> The vertex to be visited next
    visited[] --> keeps tract of visited vertices
    disc[] --> Stores discovery times of visited vertices
    parent[] --> Stores parent vertices in DFS tree'''
    def bridgeUtil(self,u, visited, parent, low, disc):

        # Mark the current node as visited and print it
        visited[u]= True

        # Initialize discovery time and low value
        disc[u] = self.Time
        low[u] = self.Time
        self.Time += 1

        #Recur for all the vertices adjacent to this vertex
        for v in self.graph[u]:
            # If v is not visited yet, then make it a child of u
            # in DFS tree and recur for it
            if visited[v] == False :
                parent[v] = u
                self.bridgeUtil(v, visited, parent, low, disc)
```

```

# Check if the subtree rooted with v has a connection to
# one of the ancestors of u
low[u] = min(low[u], low[v])

''' If the lowest vertex reachable from subtree
under v is below u in DFS tree, then u-v is
a bridge'''
if low[v] > disc[u]:
    print ("%d %d" %(u,v))

elif v != parent[u]: # Update low value of u for parent function calls.
    low[u] = min(low[u], disc[v])

# DFS based function to find all bridges. It uses recursive
# function bridgeUtil()
def bridge(self):

    # Mark all the vertices as not visited and Initialize parent and visited,
    # and ap(articulation point) arrays
    visited = [False] * (self.V)
    disc = [float("Inf")] * (self.V)
    low = [float("Inf")] * (self.V)
    parent = [-1] * (self.V)

    # Call the recursive helper function to find bridges
    # in DFS tree rooted with vertex 'i'
    for i in range(self.V):
        if visited[i] == False:
            self.bridgeUtil(i, visited, parent, low, disc)

# Create a graph given in the above diagram
g1 = Graph(5)
g1.addEdge(1, 0)
g1.addEdge(0, 2)
g1.addEdge(2, 1)
g1.addEdge(0, 3)
g1.addEdge(3, 4)

print "Bridges in first graph "
g1.bridge()

g2 = Graph(4)

```

```
g2.addEdge(0, 1)
g2.addEdge(1, 2)
g2.addEdge(2, 3)
print "\nBridges in second graph "
g2.bridge()

g3 = Graph (7)
g3.addEdge(0, 1)
g3.addEdge(1, 2)
g3.addEdge(2, 0)
g3.addEdge(1, 3)
g3.addEdge(1, 4)
g3.addEdge(1, 6)
g3.addEdge(3, 5)
g3.addEdge(4, 5)
print "\nBridges in third graph "
g3.bridge()

#This code is contributed by Neelam Yadav
```

Output:

```
Bridges in first graph
3 4
0 3
```

```
Bridges in second graph
2 3
1 2
0 1
```

```
Bridges in third graph
1 6
```

Time Complexity: The above function is simple DFS with additional arrays. So time complexity is same as DFS which is $O(V+E)$ for adjacency list representation of graph.

References:

<https://www.cs.washington.edu/education/courses/421/04su/slides/artic.pdf>
<http://www.slideshare.net/TraianRebedea/algorithm-design-and-complexity-course-8>
http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250_Weiss/L25-Connectivity.htm
<http://www.youtube.com/watch?v=bmyyxNyZKzI>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Improved By : [FurqanAziz](#), [MatthiasKoerber](#)

Source

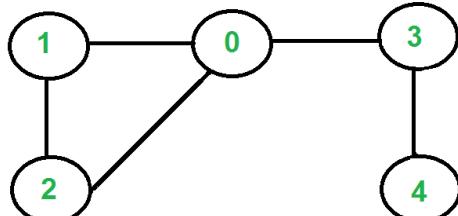
<https://www.geeksforgeeks.org/bridge-in-a-graph/>

Chapter 46

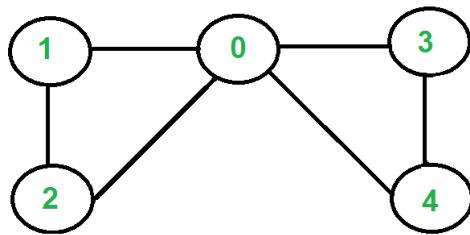
Eulerian path and circuit

Eulerian path and circuit for undirected graph - GeeksforGeeks

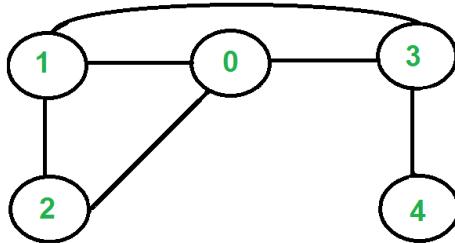
[Eulerian Path](#) is a path in graph that visits every edge exactly once. Eulerian Circuit is an Eulerian Path which starts and ends on the same vertex.



The graph has Eulerian Paths, for example "4 3 0 1 2 0", but no Eulerian Cycle. Note that there are two vertices with odd degree (4 and 0)



The graph has Eulerian Cycles, for example "2 1 0 3 4 0 2"
Note that all vertices have even degree



The graph is not Eulerian. Note that there are four vertices with odd degree (0, 1, 3 and 4)

How to find whether a given graph is Eulerian or not?

The problem is same as following question. “Is it possible to draw a given graph without lifting pencil from the paper and without tracing any of the edges more than once”.

A graph is called Eulerian if it has an Eulerian Cycle and called Semi-Eulerian if it has an Eulerian Path. The problem seems similar to [Hamiltonian Path](#) which is NP complete problem for a general graph. Fortunately, we can find whether a given graph has a Eulerian Path or not in polynomial time. In fact, we can find it in $O(V+E)$ time.

Following are some interesting properties of undirected graphs with an Eulerian path and cycle. We can use these properties to find whether a graph is Eulerian or not.

Eulerian Cycle

An undirected graph has Eulerian cycle if following two conditions are true.

-a) All vertices with non-zero degree are connected. We don't care about vertices with zero degree because they don't belong to Eulerian Cycle or Path (we only consider all edges).
-b) All vertices have even degree.

Eulerian Path

An undirected graph has Eulerian Path if following two conditions are true.

-a) Same as condition (a) for Eulerian Cycle
-b) If two vertices have odd degree and all other vertices have even degree. Note that only one vertex with odd degree is not possible in an undirected graph (sum of all degrees is always even in an undirected graph)

Note that a graph with no edges is considered Eulerian because there are no edges to traverse.

How does this work?

In Eulerian path, each time we visit a vertex v , we walk through two unvisited edges with one end point as v . Therefore, all middle vertices in Eulerian Path must have even degree. For Eulerian Cycle, any vertex can be middle vertex, therefore all vertices must have even degree.

C++

```
// A C++ program to check if a given graph is Eulerian or not
#include<iostream>
#include <list>
using namespace std;

// A class that represents an undirected graph
```

```

class Graph
{
    int V;      // No. of vertices
    list<int> *adj;     // A dynamic array of adjacency lists
public:
    // Constructor and destructor
    Graph(int V) {this->V = V; adj = new list<int>[V]; }
    ~Graph() { delete [] adj; } // To avoid memory leak

    // function to add an edge to graph
    void addEdge(int v, int w);

    // Method to check if this graph is Eulerian or not
    int isEulerian();

    // Method to check if all non-zero degree vertices are connected
    bool isConnected();

    // Function to do DFS starting from v. Used in isConnected();
    void DFSUtil(int v, bool visited[]);
};

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
    adj[w].push_back(v); // Note: the graph is undirected
}

void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

// Method to check if all non-zero degree vertices are connected.
// It mainly does DFS traversal starting from
bool Graph::isConnected()
{
    // Mark all the vertices as not visited
    bool visited[V];
    int i;
    for (i = 0; i < V; i++)

```

```

visited[i] = false;

// Find a vertex with non-zero degree
for (i = 0; i < V; i++)
    if (adj[i].size() != 0)
        break;

// If there are no edges in the graph, return true
if (i == V)
    return true;

// Start DFS traversal from a vertex with non-zero degree
DFSUtil(i, visited);

// Check if all non-zero degree vertices are visited
for (i = 0; i < V; i++)
    if (visited[i] == false && adj[i].size() > 0)
        return false;

return true;
}

/* The function returns one of the following values
0 --> If grpah is not Eulerian
1 --> If graph has an Euler path (Semi-Eulerian)
2 --> If graph has an Euler Circuit (Eulerian) */
int Graph::isEulerian()
{
    // Check if all non-zero degree vertices are connected
    if (isConnected() == false)
        return 0;

    // Count vertices with odd degree
    int odd = 0;
    for (int i = 0; i < V; i++)
        if (adj[i].size() & 1)
            odd++;

    // If count is more than 2, then graph is not Eulerian
    if (odd > 2)
        return 0;

    // If odd count is 2, then semi-eulerian.
    // If odd count is 0, then eulerian
    // Note that odd count can never be 1 for undirected graph
    return (odd)? 1 : 2;
}

```

```
// Function to run test cases
void test(Graph &g)
{
    int res = g.isEulerian();
    if (res == 0)
        cout << "graph is not Eulerian\n";
    else if (res == 1)
        cout << "graph has a Euler path\n";
    else
        cout << "graph has a Euler cycle\n";
}

// Driver program to test above function
int main()
{
    // Let us create and test graphs shown in above figures
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 1);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    test(g1);

    Graph g2(5);
    g2.addEdge(1, 0);
    g2.addEdge(0, 2);
    g2.addEdge(2, 1);
    g2.addEdge(0, 3);
    g2.addEdge(3, 4);
    g2.addEdge(4, 0);
    test(g2);

    Graph g3(5);
    g3.addEdge(1, 0);
    g3.addEdge(0, 2);
    g3.addEdge(2, 1);
    g3.addEdge(0, 3);
    g3.addEdge(3, 4);
    g3.addEdge(1, 3);
    test(g3);

    // Let us create a graph with 3 vertices
    // connected in the form of cycle
    Graph g4(3);
    g4.addEdge(0, 1);
    g4.addEdge(1, 2);
    g4.addEdge(2, 0);
```

```
    test(g4);

    // Let us create a graph with all vertices
    // with zero degree
    Graph g5(3);
    test(g5);

    return 0;
}
```

Java

```
// A Java program to check if a given graph is Eulerian or not
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents an undirected graph using adjacency list
// representation
class Graph
{
    private int V;      // No. of vertices

    // Array of lists for Adjacency List Representation
    private LinkedList<Integer> adj[];

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    //Function to add an edge into the graph
    void addEdge(int v, int w)
    {
        adj[v].add(w); // Add w to v's list.
        adj[w].add(v); //The graph is undirected
    }

    // A function used by DFS
    void DFSUtil(int v,boolean visited[])
    {
        // Mark the current node as visited
        visited[v] = true;
```

```

// Recur for all the vertices adjacent to this vertex
Iterator<Integer> i = adj[v].listIterator();
while (i.hasNext())
{
    int n = i.next();
    if (!visited[n])
        DFSUtil(n, visited);
}
}

// Method to check if all non-zero degree vertices are
// connected. It mainly does DFS traversal starting from
boolean isConnected()
{
    // Mark all the vertices as not visited
    boolean visited[] = new boolean[V];
    int i;
    for (i = 0; i < V; i++)
        visited[i] = false;

    // Find a vertex with non-zero degree
    for (i = 0; i < V; i++)
        if (adj[i].size() != 0)
            break;

    // If there are no edges in the graph, return true
    if (i == V)
        return true;

    // Start DFS traversal from a vertex with non-zero degree
    DFSUtil(i, visited);

    // Check if all non-zero degree vertices are visited
    for (i = 0; i < V; i++)
        if (visited[i] == false && adj[i].size() > 0)
            return false;

    return true;
}

/* The function returns one of the following values
0 --> If grpah is not Eulerian
1 --> If graph has an Euler path (Semi-Eulerian)
2 --> If graph has an Euler Circuit (Eulerian) */
int isEulerian()
{
    // Check if all non-zero degree vertices are connected
    if (isConnected() == false)

```

```

        return 0;

    // Count vertices with odd degree
    int odd = 0;
    for (int i = 0; i < V; i++)
        if (adj[i].size()%2!=0)
            odd++;

    // If count is more than 2, then graph is not Eulerian
    if (odd > 2)
        return 0;

    // If odd count is 2, then semi-eulerian.
    // If odd count is 0, then eulerian
    // Note that odd count can never be 1 for undirected graph
    return (odd==2)? 1 : 2;
}

// Function to run test cases
void test()
{
    int res = isEulerian();
    if (res == 0)
        System.out.println("graph is not Eulerian");
    else if (res == 1)
        System.out.println("graph has a Euler path");
    else
        System.out.println("graph has a Euler cycle");
}

// Driver method
public static void main(String args[])
{
    // Let us create and test graphs shown in above figures
    Graph g1 = new Graph(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 1);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.test();

    Graph g2 = new Graph(5);
    g2.addEdge(1, 0);
    g2.addEdge(0, 2);
    g2.addEdge(2, 1);
    g2.addEdge(0, 3);
    g2.addEdge(3, 4);
}

```

```
g2.addEdge(4, 0);
g2.test();

Graph g3 = new Graph(5);
g3.addEdge(1, 0);
g3.addEdge(0, 2);
g3.addEdge(2, 1);
g3.addEdge(0, 3);
g3.addEdge(3, 4);
g3.addEdge(1, 3);
g3.test();

// Let us create a graph with 3 vertices
// connected in the form of cycle
Graph g4 = new Graph(3);
g4.addEdge(0, 1);
g4.addEdge(1, 2);
g4.addEdge(2, 0);
g4.test();

// Let us create a graph with all vertices
// with zero degree
Graph g5 = new Graph(3);
g5.test();
}

}

// This code is contributed by Aakash Hasija
```

Python

```
# Python program to check if a given graph is Eulerian or not
#Complexity : O(V+E)

from collections import defaultdict

#This class represents a undirected graph using adjacency list representation
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices
        self.graph = defaultdict(list) # default dictionary to store graph

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)
        self.graph[v].append(u)

    #A function used by isConnected
```

```

def DFSUtil(self,v,visited):
    # Mark the current node as visited
    visited[v]= True

    #Recur for all the vertices adjacent to this vertex
    for i in self.graph[v]:
        if visited[i]==False:
            self.DFSUtil(i,visited)

'''Method to check if all non-zero degree vertices are
connected. It mainly does DFS traversal starting from
node with non-zero degree'''
def isConnected(self):

    # Mark all the vertices as not visited
    visited =[False]*(self.V)

    # Find a vertex with non-zero degree
    for i in range(self.V):
        if len(self.graph[i]) > 1:
            break

    # If there are no edges in the graph, return true
    if i == self.V-1:
        return True

    # Start DFS traversal from a vertex with non-zero degree
    self.DFSUtil(i,visited)

    # Check if all non-zero degree vertices are visited
    for i in range(self.V):
        if visited[i]==False and len(self.graph[i]) > 0:
            return False

    return True

'''The function returns one of the following values
0 --> If grpah is not Eulerian
1 --> If graph has an Euler path (Semi-Eulerian)
2 --> If graph has an Euler Circuit (Eulerian) '''
def isEulerian(self):
    # Check if all non-zero degree vertices are connected
    if self.isConnected() == False:
        return 0
    else:
        #Count vertices with odd degree

```

```

odd = 0
for i in range(self.V):
    if len(self.graph[i]) % 2 !=0:
        odd +=1

    '''If odd count is 2, then semi-eulerian.
    If odd count is 0, then eulerian
    If count is more than 2, then graph is not Eulerian
    Note that odd count can never be 1 for undirected graph'''
    if odd == 0:
        return 2
    elif odd == 2:
        return 1
    elif odd > 2:
        return 0

# Function to run test cases
def test(self):
    res = self.isEulerian()
    if res == 0:
        print "graph is not Eulerian"
    elif res ==1 :
        print "graph has a Euler path"
    else:
        print "graph has a Euler cycle"

#Let us create and test graphs shown in above figures
g1 = Graph(5);
g1.addEdge(1, 0)
g1.addEdge(0, 2)
g1.addEdge(2, 1)
g1.addEdge(0, 3)
g1.addEdge(3, 4)
g1.test()

g2 = Graph(5)
g2.addEdge(1, 0)
g2.addEdge(0, 2)
g2.addEdge(2, 1)
g2.addEdge(0, 3)
g2.addEdge(3, 4)
g2.addEdge(4, 0)
g2.test();

g3 = Graph(5)

```

```
g3.addEdge(1, 0)
g3.addEdge(0, 2)
g3.addEdge(2, 1)
g3.addEdge(0, 3)
g3.addEdge(3, 4)
g3.addEdge(1, 3)
g3.test()

#Let us create a graph with 3 vertices
# connected in the form of cycle
g4 = Graph(3)
g4.addEdge(0, 1)
g4.addEdge(1, 2)
g4.addEdge(2, 0)
g4.test()

# Let us create a graph with all veritces
# with zero degree
g5 = Graph(3)
g5.test()

#This code is contributed by Neelam Yadav
```

Output:

```
graph has a Euler path
graph has a Euler cycle
graph is not Eulerian
graph has a Euler cycle
graph has a Euler cycle
```

Time Complexity: O(V+E)

Next Articles:

[Eulerian Path and Circuit for a Directed Graphs.](#)
[Fleury's Algorithm to print a Eulerian Path or Circuit?](#)

References:

http://en.wikipedia.org/wiki/Eulerian_path

Improved By : StartingTheLife

Source

<https://www.geeksforgeeks.org/eulerian-path-and-circuit/>

Chapter 47

Fleury's Algorithm for printing Eulerian Path or Circuit

Fleury's Algorithm for printing Eulerian Path or Circuit - GeeksforGeeks

Eulerian Path is a path in graph that visits every edge exactly once. Eulerian Circuit is an Eulerian Path which starts and ends on the same vertex.

We strongly recommend to first read the following post on Euler Path and Circuit.

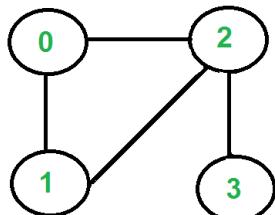
<https://www.geeksforgeeks.org/eulerian-path-and-circuit/>

In the above mentioned post, we discussed the problem of finding out whether a given graph is Eulerian or not. In this post, an algorithm to print Eulerian trail or circuit is discussed.

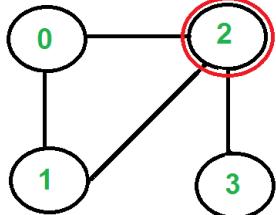
Following is Fleury's Algorithm for printing Eulerian trail or cycle (Source [Ref1](#)).

1. Make sure the graph has either 0 or 2 odd vertices.
2. If there are 0 odd vertices, start anywhere. If there are 2 odd vertices, start at one of them.
3. Follow edges one at a time. If you have a choice between a bridge and a non-bridge, *always choose the non-bridge*.
4. Stop when you run out of edges.

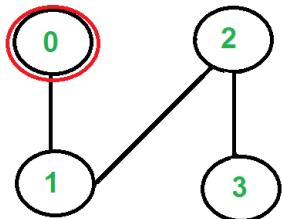
The idea is, “**don't burn bridges**“ so that we can come back to a vertex and traverse remaining edges. For example let us consider the following graph.



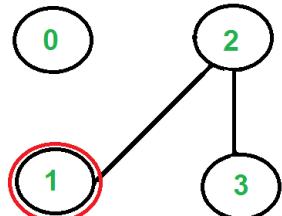
There are two vertices with odd degree, ‘2’ and ‘3’, we can start path from any of them. Let us start tour from vertex ‘2’.



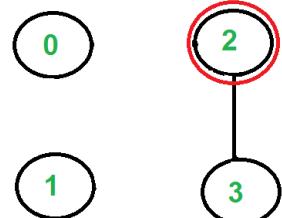
There are three edges going out from vertex ‘2’, which one to pick? We don’t pick the edge ‘2-3’ because that is a bridge (we won’t be able to come back to ‘3’). We can pick any of the remaining two edge. Let us say we pick ‘2-0’. We remove this edge and move to vertex ‘0’.



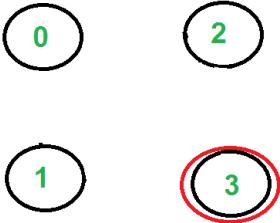
There is only one edge from vertex ‘0’, so we pick it, remove it and move to vertex ‘1’. Euler tour becomes ‘2-0 0-1’.



There is only one edge from vertex ‘1’, so we pick it, remove it and move to vertex ‘2’. Euler tour becomes ‘2-0 0-1 1-2’



Again there is only one edge from vertex 2, so we pick it, remove it and move to vertex 3. Euler tour becomes ‘2-0 0-1 1-2 2-3’



There are no more edges left, so we stop here. Final tour is '2-0 0-1 1-2 2-3'.

See [this](#)for and [this](#)fore more examples.

Following is C++ implementation of above algorithm. In the following code, it is assumed that the given graph has an Eulerian trail or Circuit. The main focus is to print an Eulerian trail or circuit. We can use [isEulerian\(\)](#) to first check whether there is an Eulerian Trail or Circuit in the given graph.

We first find the starting point which must be an odd vertex (if there are odd vertices) and store it in variable 'u'. If there are zero odd vertices, we start from vertex '0'. We call [printEulerUtil\(\)](#) to print Euler tour starting with u. We traverse all adjacent vertices of u, if there is only one adjacent vertex, we immediately consider it. If there are more than one adjacent vertices, we consider an adjacent v only if edge u-v is not a bridge. How to find if a given is edge is bridge? We count number of vertices reachable from u. We remove edge u-v and again count number of reachable vertices from u. If number of reachable vertices are reduced, then edge u-v is a bridge. To count reachable vertices, we can either use BFS or DFS, we have used DFS in the above code. The function [DFSCount\(u\)](#) returns number of vertices reachable from u.

Once an edge is processed (included in Euler tour), we remove it from the graph. To remove the edge, we replace the vertex entry with -1 in adjacency list. Note that simply deleting the node may not work as the code is recursive and a parent call may be in middle of adjacency list.

C/C++

```
// A C++ program print Eulerian Trail in a given Eulerian or Semi-Eulerian Graph
#include <iostream>
#include <string.h>
#include <algorithm>
#include <list>
using namespace std;

// A class that represents an undirected graph
class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // A dynamic array of adjacency lists
public:
    // Constructor and destructor
    Graph(int V) { this->V = V; adj = new list<int>[V]; }
```

```

~Graph()      { delete [] adj; }

// functions to add and remove edge
void addEdge(int u, int v) { adj[u].push_back(v); adj[v].push_back(u); }
void rmvEdge(int u, int v);

// Methods to print Eulerian tour
void printEulerTour();
void printEulerUtil(int s);

// This function returns count of vertices reachable from v. It does DFS
int DFSCount(int v, bool visited[]);

// Utility function to check if edge u-v is a valid next edge in
// Eulerian trail or circuit
bool isValidNextEdge(int u, int v);
};

/* The main function that print Eulerian Trail. It first finds an odd
degree vertex (if there is any) and then calls printEulerUtil()
to print the path */
void Graph::printEulerTour()
{
    // Find a vertex with odd degree
    int u = 0;
    for (int i = 0; i < V; i++)
        if (adj[i].size() & 1)
            { u = i; break; }

    // Print tour starting from oddv
    printEulerUtil(u);
    cout << endl;
}

// Print Euler tour starting from vertex u
void Graph::printEulerUtil(int u)
{
    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        int v = *i;

        // If edge u-v is not removed and it's a a valid next edge
        if (v != -1 && isValidNextEdge(u, v))
        {
            cout << u << "-" << v << " ";
            rmvEdge(u, v);
        }
    }
}

```

```

        printEulerUtil(v);
    }
}

// The function to check if edge u-v can be considered as next edge in
// Euler Tour
bool Graph::isValidNextEdge(int u, int v)
{
    // The edge u-v is valid in one of the following two cases:

    // 1) If v is the only adjacent vertex of u
    int count = 0; // To store count of adjacent vertices
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
        if (*i != -1)
            count++;
    if (count == 1)
        return true;

    // 2) If there are multiple adjacents, then u-v is not a bridge
    // Do following steps to check if u-v is a bridge

    // 2.a) count of vertices reachable from u
    bool visited[V];
    memset(visited, false, V);
    int count1 = DFSCount(u, visited);

    // 2.b) Remove edge (u, v) and after removing the edge, count
    // vertices reachable from u
    rmvEdge(u, v);
    memset(visited, false, V);
    int count2 = DFSCount(u, visited);

    // 2.c) Add the edge back to the graph
    addEdge(u, v);

    // 2.d) If count1 is greater, then edge (u, v) is a bridge
    return (count1 > count2)? false: true;
}

// This function removes edge u-v from graph. It removes the edge by
// replacing adjacent vertex value with -1.
void Graph::rmvEdge(int u, int v)
{
    // Find v in adjacency list of u and replace it with -1
    list<int>::iterator iv = find(adj[u].begin(), adj[u].end(), v);
}

```

```

*iv = -1;

// Find u in adjacency list of v and replace it with -1
list<int>::iterator iu = find(adj[v].begin(), adj[v].end(), u);
*iu = -1;
}

// A DFS based function to count reachable vertices from v
int Graph::DFSCount(int v, bool visited[])
{
    // Mark the current node as visited
    visited[v] = true;
    int count = 1;

    // Recur for all vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (*i != -1 && !visited[*i])
            count += DFSCount(*i, visited);

    return count;
}

// Driver program to test above function
int main()
{
    // Let us first create and test graphs shown in above figure
    Graph g1(4);
    g1.addEdge(0, 1);
    g1.addEdge(0, 2);
    g1.addEdge(1, 2);
    g1.addEdge(2, 3);
    g1.printEulerTour();

    Graph g2(3);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.addEdge(2, 0);
    g2.printEulerTour();

    Graph g3(5);
    g3.addEdge(1, 0);
    g3.addEdge(0, 2);
    g3.addEdge(2, 1);
    g3.addEdge(0, 3);
    g3.addEdge(3, 4);
    g3.addEdge(3, 2);
    g3.addEdge(3, 1);
}

```

```
g3.addEdge(2, 4);
g3.printEulerTour();

return 0;
}

Java

// A Java program print Eulerian Trail
// in a given Eulerian or Semi-Eulerian Graph
import java.util.ArrayList;

// An Undirected graph using
// adjacency list representation
public class Graph {

    private int vertices; // No. of vertices
    private ArrayList<Integer>[] adj; // adjacency list

    // Constructor
    Graph(int numVertices)
    {
        // initialise vertex count
        this.vertices = numVertices;

        // initialise adjacency list
        initGraph();
    }

    // utility method to initialise adjacency list
    @SuppressWarnings("unchecked")
    private void initGraph()
    {
        adj = new ArrayList[vertices];
        for (int i = 0; i < vertices; i++)
        {
            adj[i] = new ArrayList<>();
        }
    }

    // add edge u-v
    private void addEdge(Integer u, Integer v)
    {
        adj[u].add(v);
        adj[v].add(u);
    }

    // This function removes edge u-v from graph.
```

```

private void removeEdge(Integer u, Integer v)
{
    adj[u].remove(v);
    adj[v].remove(u);
}

/* The main function that print Eulerian Trail.
   It first finds an odd degree vertex (if there
   is any) and then calls printEulerUtil() to
   print the path */
private void printEulerTour()
{
    // Find a vertex with odd degree
    Integer u = 0;
    for (int i = 0; i < vertices; i++)
    {
        if (adj[i].size() % 2 == 1)
        {
            u = i;
            break;
        }
    }

    // Print tour starting from oddv
    printEulerUtil(u);
    System.out.println();
}

// Print Euler tour starting from vertex u
private void printEulerUtil(Integer u)
{
    // Recur for all the vertices adjacent to this vertex
    for (int i = 0; i < adj[u].size(); i++)
    {
        Integer v = adj[u].get(i);
        // If edge u-v is a valid next edge
        if (isValidNextEdge(u, v))
        {
            System.out.print(u + "-" + v + " ");

            // This edge is used so remove it now
            removeEdge(u, v);
            printEulerUtil(v);
        }
    }
}

// The function to check if edge u-v can be

```

```

// considered as next edge in Euler Tout
private boolean isValidNextEdge(Integer u, Integer v)
{
    // The edge u-v is valid in one of the
    // following two cases:

    // 1) If v is the only adjacent vertex of u
    // ie size of adjacent vertex list is 1
    if (adj[u].size() == 1) {
        return true;
    }

    // 2) If there are multiple adjacents, then
    // u-v is not a bridge Do following steps
    // to check if u-v is a bridge
    // 2.a) count of vertices reachable from u
    boolean[] isVisited = new boolean[this.vertices];
    int count1 = dfsCount(u, isVisited);

    // 2.b) Remove edge (u, v) and after removing
    // the edge, count vertices reachable from u
    removeEdge(u, v);
    isVisited = new boolean[this.vertices];
    int count2 = dfsCount(u, isVisited);

    // 2.c) Add the edge back to the graph
    addEdge(u, v);
    return (count1 > count2) ? false : true;
}

// A DFS based function to count reachable
// vertices from v
private int dfsCount(Integer v, boolean[] isVisited)
{
    // Mark the current node as visited
    isVisited[v] = true;
    int count = 1;
    // Recur for all vertices adjacent to this vertex
    for (int adj : adj[v])
    {
        if (!isVisited[adj])
        {
            count = count + dfsCount(adj, isVisited);
        }
    }
    return count;
}

```

```
// Driver program to test above function
public static void main(String a[])
{
    // Let us first create and test
    // graphs shown in above figure
    Graph g1 = new Graph(4);
    g1.addEdge(0, 1);
    g1.addEdge(0, 2);
    g1.addEdge(1, 2);
    g1.addEdge(2, 3);
    g1.printEulerTour();

    Graph g2 = new Graph(3);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.addEdge(2, 0);
    g2.printEulerTour();

    Graph g3 = new Graph(5);
    g3.addEdge(1, 0);
    g3.addEdge(0, 2);
    g3.addEdge(2, 1);
    g3.addEdge(0, 3);
    g3.addEdge(3, 4);
    g3.addEdge(3, 2);
    g3.addEdge(3, 1);
    g3.addEdge(2, 4);
    g3.printEulerTour();
}
}

// This code is contributed by Himanshu Shekhar
```

Python

```
# Python program print Eulerian Trail in a given Eulerian or Semi-Eulerian Graph

from collections import defaultdict

#This class represents an undirected graph using adjacency list representation
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices
        self.graph = defaultdict(list) # default dictionary to store graph
        self.Time = 0

    # function to add an edge to graph
```

```

def addEdge(self,u,v):
    self.graph[u].append(v)
    self.graph[v].append(u)

# This function removes edge u-v from graph
def rmvEdge(self, u, v):
    for index, key in enumerate(self.graph[u]):
        if key == v:
            self.graph[u].pop(index)
    for index, key in enumerate(self.graph[v]):
        if key == u:
            self.graph[v].pop(index)

# A DFS based function to count reachable vertices from v
def DFSCount(self, v, visited):
    count = 1
    visited[v] = True
    for i in self.graph[v]:
        if visited[i] == False:
            count = count + self.DFSCount(i, visited)
    return count

# The function to check if edge u-v can be considered as next edge in
# Euler Tour
def isValidNextEdge(self, u, v):
    # The edge u-v is valid in one of the following two cases:

    # 1) If v is the only adjacent vertex of u
    if len(self.graph[u]) == 1:
        return True
    else:
        '''
        2) If there are multiple adjacents, then u-v is not a bridge
           Do following steps to check if u-v is a bridge

        2.a) count of vertices reachable from u'''
    visited =[False]*(self.V)
    count1 = self.DFSCount(u, visited)

    '''2.b) Remove edge (u, v) and after removing the edge, count
           vertices reachable from u'''
    self.rmvEdge(u, v)
    visited =[False]*(self.V)
    count2 = self.DFSCount(u, visited)

    #2.c) Add the edge back to the graph
    self.addEdge(u,v)

```

```

# 2.d) If count1 is greater, then edge (u, v) is a bridge
return False if count1 > count2 else True

# Print Euler tour starting from vertex u
def printEulerUtil(self, u):
    #Recur for all the vertices adjacent to this vertex
    for v in self.graph[u]:
        #If edge u-v is not removed and it's a valid next edge
        if self.isValidNextEdge(u, v):
            print("%d-%d " %(u,v)),
            self.rmvEdge(u, v)
            self.printEulerUtil(v)

'''The main function that prints Eulerian Trail. It first finds an odd
degree vertex (if there is any) and then calls printEulerUtil()
to print the path '''
def printEulerTour(self):
    #Find a vertex with odd degree
    u = 0
    for i in range(self.V):
        if len(self.graph[i]) %2 != 0 :
            u = i
            break
    # Print tour starting from odd vertex
    print ("\n")
    self.printEulerUtil(u)

# Create a graph given in the above diagram

g1 = Graph(4)
g1.addEdge(0, 1)
g1.addEdge(0, 2)
g1.addEdge(1, 2)
g1.addEdge(2, 3)
g1.printEulerTour()

g2 = Graph(3)
g2.addEdge(0, 1)
g2.addEdge(1, 2)
g2.addEdge(2, 0)
g2.printEulerTour()

g3 = Graph (5)
g3.addEdge(1, 0)

```

```
g3.addEdge(0, 2)
g3.addEdge(2, 1)
g3.addEdge(0, 3)
g3.addEdge(3, 4)
g3.addEdge(3, 2)
g3.addEdge(3, 1)
g3.addEdge(2, 4)
g3.printEulerTour()
```

#This code is contributed by Neelam Yadav

Output:

```
2-0  0-1  1-2  2-3
0-1  1-2  2-0
0-1  1-2  2-0  0-3  3-4  4-2  2-3  3-1
```

Note that the above code modifies given graph, we can create a copy of graph if we don't want the given graph to be modified.

Time Complexity: Time complexity of the above implementation is $O((V+E)^2)$. The function printEulerUtil() is like DFS and it calls isValidNextEdge() which also does DFS two times. Time complexity of DFS for adjacency list representation is $O(V+E)$. Therefore overall time complexity is $O((V+E)*(V+E))$ which can be written as $O(E^2)$ for a connected graph.

There are better algorithms to print Euler tour, [Hierholzer's Algorithm](#) finds in $O(V+E)$ time.

References:

<http://www.math.ku.edu/~jmartin/courses/math105-F11/Lectures/chapter5-part2.pdf>
http://en.wikipedia.org/wiki/Eulerian_path#Fleury.27s_algorithm

Source

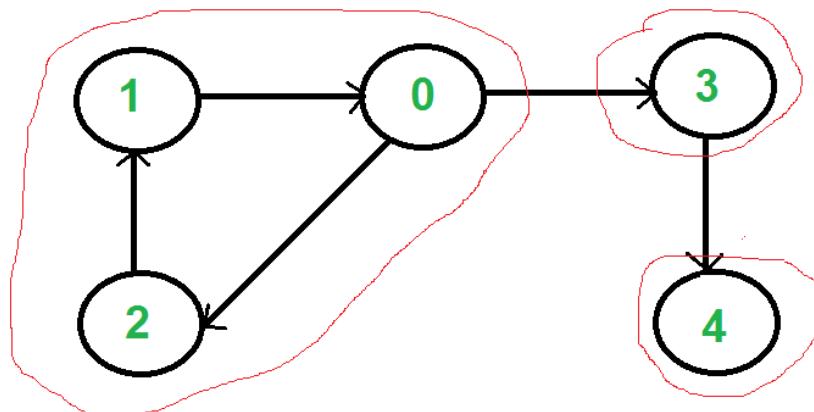
<https://www.geeksforgeeks.org/fleury-s-algorithm-for-printing-eulerian-path/>

Chapter 48

Strongly Connected Components

Strongly Connected Components - GeeksforGeeks

A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (**SCC**) of a directed graph is a maximal strongly connected subgraph. For example, there are 3 SCCs in the following graph.



We can find all strongly connected components in $O(V+E)$ time using [Kosaraju's algorithm](#). Following is detailed Kosaraju's algorithm.

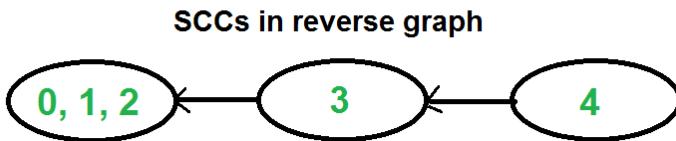
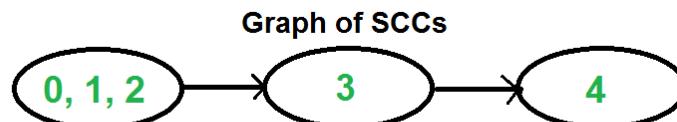
- 1) Create an empty stack 'S' and do DFS traversal of a graph. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack. In the above graph, if we start DFS from vertex 0, we get vertices in stack as 1, 2, 4, 3, 0.
- 2) Reverse directions of all arcs to obtain the transpose graph.
- 3) One by one pop a vertex from S while S is not empty. Let the popped vertex be 'v'. Perform a DFS traversal starting from v in the transpose graph. Mark all vertices reached during this traversal as part of the current SCC.

Take v as source and do DFS (call `DFSUtil(v)`). The DFS starting from v prints strongly connected component of v . In the above example, we process vertices in order 0, 3, 4, 2, 1 (One by one popped from stack).

How does this work?

The above algorithm is DFS based. It does DFS two times. DFS of a graph produces a single tree if all vertices are reachable from the DFS starting point. Otherwise DFS produces a forest. So DFS of a graph with only one SCC always produces a tree. The important point to note is DFS may produce a tree or a forest when there are more than one SCCs depending upon the chosen starting point. For example, in the above diagram, if we start DFS from vertices 0 or 1 or 2, we get a tree as output. And if we start from 3 or 4, we get a forest. To find and print all SCCs, we would want to start DFS from vertex 4 (which is a sink vertex), then move to 3 which is sink in the remaining set (set excluding 4) and finally any of the remaining vertices (0, 1, 2). So how do we find this sequence of picking vertices as starting points of DFS? Unfortunately, there is no direct way for getting this sequence. However, if we do a DFS of graph and store vertices according to their finish times, we make sure that the finish time of a vertex that connects to other SCCs (other than its own SCC), will always be greater than finish time of vertices in the other SCC (See [this](#)for proof). For example, in DFS of above example graph, finish time of 0 is always greater than 3 and 4 (irrespective of the sequence of vertices considered for DFS). And finish time of 3 is always greater than 4. DFS doesn't guarantee about other vertices, for example finish times of 1 and 2 may be smaller or greater than 3 and 4 depending upon the sequence of vertices considered for DFS. So to use this property, we do DFS traversal of complete graph and push every finished vertex to a stack. In stack, 3 always appears after 4, and 0 appear after both 3 and 4.

In the next step, we reverse the graph. Consider the graph of SCCs. In the reversed graph, the edges that connect two components are reversed. So the SCC $\{0, 1, 2\}$ becomes sink and the SCC $\{4\}$ becomes source. As discussed above, in stack, we always have 0 before 3 and 4. So if we do a DFS of the reversed graph using sequence of vertices in stack, we process vertices from sink to source (in reversed graph). That is what we wanted to achieve and that is all needed to print SCCs one by one.



Following is C++ implementation of Kosaraju's algorithm.
C++

```
// C++ Implementation of Kosaraju's algorithm to print all SCCs
```

```

#include <iostream>
#include <list>
#include <stack>
using namespace std;

class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // An array of adjacency lists

    // Fills Stack with vertices (in increasing order of finishing
    // times). The top element of stack has the maximum finishing
    // time
    void fillOrder(int v, bool visited[], stack<int> &Stack);

    // A recursive function to print DFS starting from v
    void DFSUtil(int v, bool visited[]);

public:
    Graph(int V);
    void addEdge(int v, int w);

    // The main function that finds and prints strongly connected
    // components
    void printSCCs();

    // Function that returns reverse (or transpose) of this graph
    Graph getTranspose();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

// A recursive function to print DFS starting from v
void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

```

```

Graph Graph::getTranspose()
{
    Graph g(V);
    for (int v = 0; v < V; v++)
    {
        // Recur for all the vertices adjacent to this vertex
        list<int>::iterator i;
        for(i = adj[v].begin(); i != adj[v].end(); ++i)
        {
            g.adj[*i].push_back(v);
        }
    }
    return g;
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

void Graph::fillOrder(int v, bool visited[], stack<int> &Stack)
{
    // Mark the current node as visited and print it
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for(i = adj[v].begin(); i != adj[v].end(); ++i)
        if(!visited[*i])
            fillOrder(*i, visited, Stack);

    // All vertices reachable from v are processed by now, push v
    Stack.push(v);
}

// The main function that finds and prints all strongly connected
// components
void Graph::printSCCs()
{
    stack<int> Stack;

    // Mark all the vertices as not visited (For first DFS)
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Fill vertices in stack according to their finishing times
}

```

```
for(int i = 0; i < V; i++)
    if(visited[i] == false)
        fillOrder(i, visited, Stack);

// Create a reversed graph
Graph gr = getTranspose();

// Mark all the vertices as not visited (For second DFS)
for(int i = 0; i < V; i++)
    visited[i] = false;

// Now process all vertices in order defined by Stack
while (Stack.empty() == false)
{
    // Pop a vertex from stack
    int v = Stack.top();
    Stack.pop();

    // Print Strongly connected component of the popped vertex
    if (visited[v] == false)
    {
        gr.DFSUtil(v, visited);
        cout << endl;
    }
}

// Driver program to test above functions
int main()
{
    // Create a graph given in the above diagram
    Graph g(5);
    g.addEdge(1, 0);
    g.addEdge(0, 2);
    g.addEdge(2, 1);
    g.addEdge(0, 3);
    g.addEdge(3, 4);

    cout << "Following are strongly connected components in "
          "given graph \n";
    g.printSCCs();

    return 0;
}
```

Java

```
// Java implementation of Kosaraju's algorithm to print all SCCs
```

```
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    private int V;    // No. of vertices
    private LinkedList<Integer> adj[]; //Adjacency List

    //Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    //Function to add an edge into the graph
    void addEdge(int v, int w)  { adj[v].add(w); }

    // A recursive function to print DFS starting from v
    void DFSUtil(int v,boolean visited[])
    {
        // Mark the current node as visited and print it
        visited[v] = true;
        System.out.print(v + " ");

        int n;

        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> i =adj[v].iterator();
        while (i.hasNext())
        {
            n = i.next();
            if (!visited[n])
                DFSUtil(n,visited);
        }
    }

    // Function that returns reverse (or transpose) of this graph
    Graph getTranspose()
    {
        Graph g = new Graph(V);
        for (int v = 0; v < V; v++)
        {
```

```

// Recur for all the vertices adjacent to this vertex
Iterator<Integer> i = adj[v].listIterator();
while(i.hasNext())
    g.adj[i.next()].add(v);
}
return g;
}

void fillOrder(int v, boolean visited[], Stack stack)
{
    // Mark the current node as visited and print it
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    Iterator<Integer> i = adj[v].iterator();
    while (i.hasNext())
    {
        int n = i.next();
        if(!visited[n])
            fillOrder(n, visited, stack);
    }

    // All vertices reachable from v are processed by now,
    // push v to Stack
    stack.push(new Integer(v));
}

// The main function that finds and prints all strongly
// connected components
void printSCCs()
{
    Stack stack = new Stack();

    // Mark all the vertices as not visited (For first DFS)
    boolean visited[] = new boolean[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Fill vertices in stack according to their finishing
    // times
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            fillOrder(i, visited, stack);

    // Create a reversed graph
    Graph gr = getTranspose();

    // Mark all the vertices as not visited (For second DFS)

```

```

        for (int i = 0; i < V; i++)
            visited[i] = false;

        // Now process all vertices in order defined by Stack
        while (stack.empty() == false)
        {
            // Pop a vertex from stack
            int v = (int)stack.pop();

            // Print Strongly connected component of the popped vertex
            if (visited[v] == false)
            {
                gr.DFSUtil(v, visited);
                System.out.println();
            }
        }

        // Driver method
        public static void main(String args[])
        {
            // Create a graph given in the above diagram
            Graph g = new Graph(5);
            g.addEdge(1, 0);
            g.addEdge(0, 2);
            g.addEdge(2, 1);
            g.addEdge(0, 3);
            g.addEdge(3, 4);

            System.out.println("Following are strongly connected components "+
                               "in given graph ");
            g.printSCCs();
        }
    }
    // This code is contributed by Aakash Hasija
}

```

Python

```

# Python implementation of Kosaraju's algorithm to print all SCCs

from collections import defaultdict

#This class represents a directed graph using adjacency list representation
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices
        self.graph = defaultdict(list) # default dictionary to store graph

```

```
# function to add an edge to graph
def addEdge(self,u,v):
    self.graph[u].append(v)

# A function used by DFS
def DFSUtil(self,v,visited):
    # Mark the current node as visited and print it
    visited[v]= True
    print v,
    #Recur for all the vertices adjacent to this vertex
    for i in self.graph[v]:
        if visited[i]==False:
            self.DFSUtil(i,visited)

def fillOrder(self,v,visited, stack):
    # Mark the current node as visited
    visited[v]= True
    #Recur for all the vertices adjacent to this vertex
    for i in self.graph[v]:
        if visited[i]==False:
            self.fillOrder(i, visited, stack)
    stack = stack.append(v)

# Function that returns reverse (or transpose) of this graph
def getTranspose(self):
    g = Graph(self.V)

    # Recur for all the vertices adjacent to this vertex
    for i in self.graph:
        for j in self.graph[i]:
            g.addEdge(j,i)
    return g

# The main function that finds and prints all strongly
# connected components
def printSCCs(self):

    stack = []
    # Mark all the vertices as not visited (For first DFS)
    visited =[False]*(self.V)
    # Fill vertices in stack according to their finishing
    # times
    for i in range(self.V):
```

```

        if visited[i]==False:
            self.fillOrder(i, visited, stack)

    # Create a reversed graph
    gr = self.getTranspose()

    # Mark all the vertices as not visited (For second DFS)
    visited =[False]*(self.V)

    # Now process all vertices in order defined by Stack
    while stack:
        i = stack.pop()
        if visited[i]==False:
            gr.DFSUtil(i, visited)
            print ""

    # Create a graph given in the above diagram
    g = Graph(5)
    g.addEdge(1, 0)
    g.addEdge(0, 2)
    g.addEdge(2, 1)
    g.addEdge(0, 3)
    g.addEdge(3, 4)

    print ("Following are strongly connected components " +
           "in given graph")
g.printSCCs()
#This code is contributed by Neelam Yadav

```

Output:

```

Following are strongly connected components in given graph
0 1 2
3
4

```

Time Complexity: The above algorithm calls DFS, finds reverse of the graph and again calls DFS. DFS takes $O(V+E)$ for a graph represented using adjacency list. Reversing a graph also takes $O(V+E)$ time. For reversing the graph, we simply traverse all adjacency lists.

The above algorithm is asymptotically best algorithm, but there are other algorithms like [Tarjan's algorithm](#) and [path-based](#) which have same time complexity but find SCCs using single DFS. The Tarjan's algorithm is discussed in the following post.

[Tarjan's Algorithm to find Strongly Connected Components](#)

Applications:

SCC algorithms can be used as a first step in many graph algorithms that work only on

strongly connected graph.

In social networks, a group of people are generally strongly connected (For example, students of a class or any other common place). Many people in these groups generally like some common pages or play common games. The SCC algorithms can be used to find such groups and suggest the commonly liked pages or games to the people in the group who have not yet liked commonly liked a page or played a game.

References:

http://en.wikipedia.org/wiki/Kosaraju%27s_algorithm
<https://www.youtube.com/watch?v=PZQ0Pdk15RA>

You may also like to see [Tarjan's Algorithm to find Strongly Connected Components](#).

Source

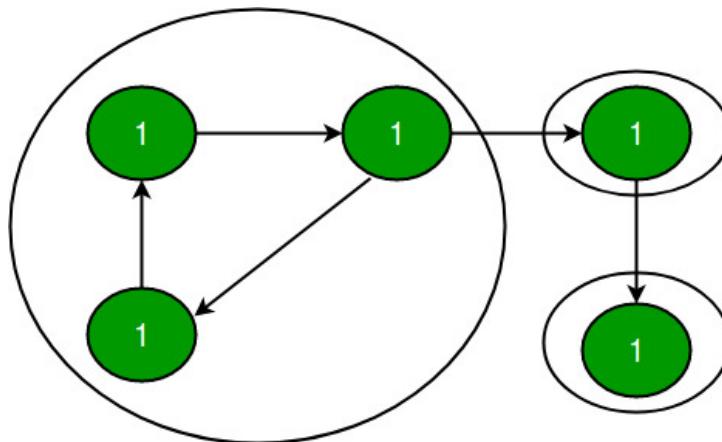
<https://www.geeksforgeeks.org/strongly-connected-components/>

Chapter 49

Tarjan's Algorithm to find Strongly Connected Components

Tarjan's Algorithm to find Strongly Connected Components - GeeksforGeeks

A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (**SCC**) of a directed graph is a maximal strongly connected subgraph. For example, there are 3 SCCs in the following graph.



We have discussed [Kosaraju's algorithm for strongly connected components](#). The previously discussed algorithm requires two DFS traversals of a Graph. In this post, [Tarjan's algorithm](#) is discussed that requires only one DFS traversal.

Tarjan Algorithm is based on following facts:

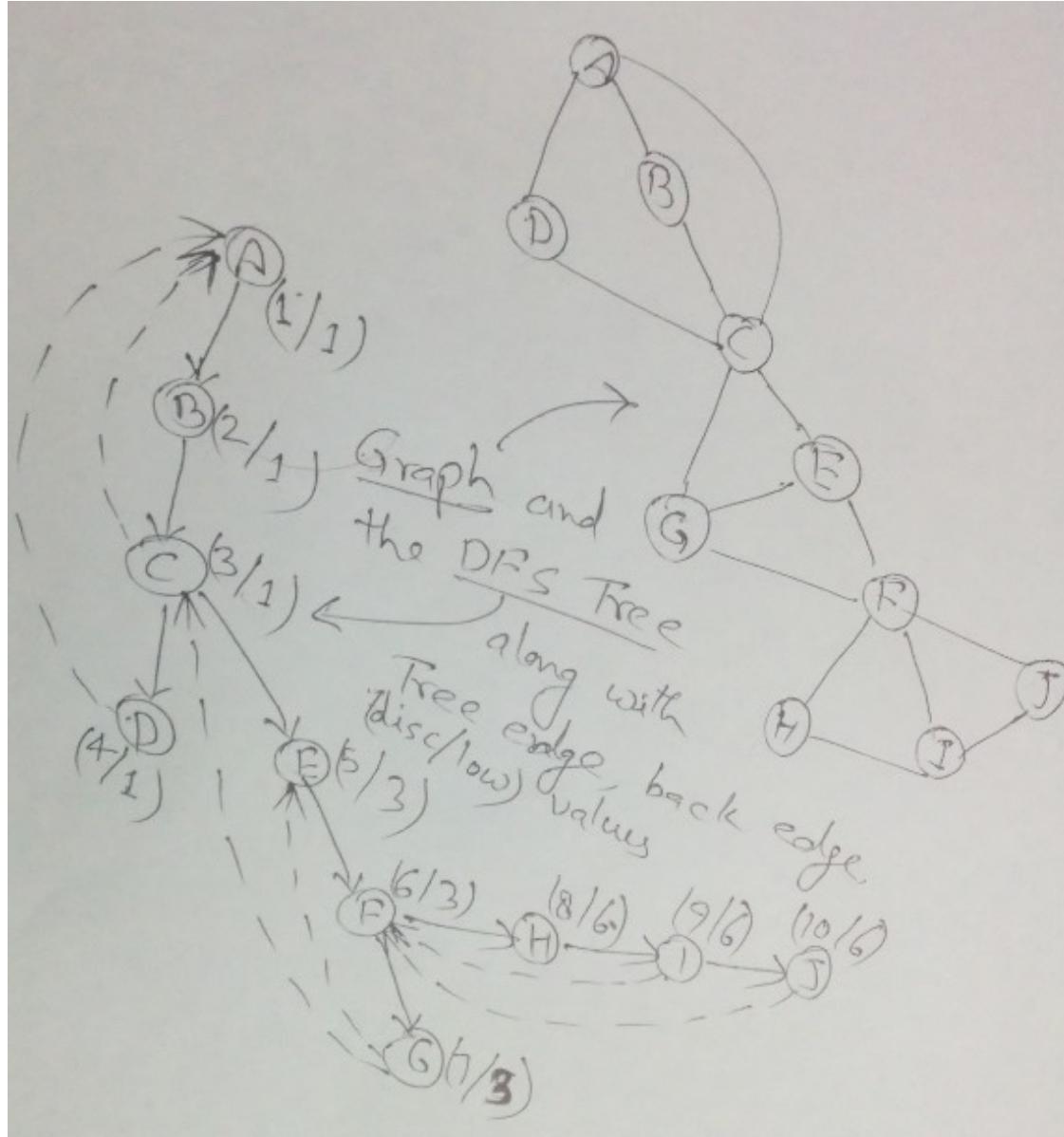
1. DFS search produces a DFS tree/forest
2. Strongly Connected Components form subtrees of the DFS tree.
3. If we can find head of such subtrees, we can print/store all the nodes in that subtree

(including head) and that will be one SCC.

4. There is no back edge from one SCC to another (There can be cross edges, but cross edges will not be used while processing the graph).

To find head of a SCC, we calculate desc and low array (as done for [articulation point](#), [bridge](#), [biconnected component](#)). As discussed in the previous posts, $\text{low}[u]$ indicates earliest visited vertex (the vertex with minimum discovery time) that can be reached from subtree rooted with u . A node u is head if $\text{disc}[u] = \text{low}[u]$.

Disc and Low Values



Strongly Connected Component relates to directed graph only, but Disc and Low values

relate to both directed and undirected graph, so in above pic we have taken an undirected graph.

In above Figure, we have shown a graph and it's one of DFS tree (There could be different DFS trees on same graph depending on order in which edges are traversed).

In DFS tree, continuous arrows are tree edges and dashed arrows are back edges ([DFS Tree Edges](#))

Disc and Low values are showin in Figure for every node as (Disc/Low).

Disc: This is the time when a node is visited 1st time while DFS traversal. For nodes A, B, C, .., J in DFS tree, Disc values are 1, 2, 3, .., 10.

Low: In DFS tree, Tree edges take us forward, from ancestor node to one of it's descendants. For example, from node C, tree edges can take us to node node G, node I etc. Back edges take us backward, from a descendant node to one of its ancestors. For example, from node G, Back edges take us to E or C. If we look at both Tree and Back edge together, then we can see that if we start traversal from one node, we may go down the tree via Tree edges and then go up via back edges. For example, from node E, we can go down to G and then go up to C. Similarly from E, we can go down to I or J and then go up to F. "Low" value of a node tells the topmost reachable ancestor (with minimum possible Disc value) via the subtree of that node. So for any node, Low value equal to it's Disc value anyway (A node is ancestor of itself). Then we look into it's subtree and see if there is any node which can take us to any of its ancestor. If there are multiple back edges in subtree which take us to different ancestors, then we take the one with minimum Disc value (i.e. the topmost one). If we look at node F, it has two subtrees. Subtree with node G, takes us to E and C. The other subtree takes us back to F only. Here topmost ancestor is C where F can reach and so Low value of F is 3 (The Disc value of C).

Based on above discussion, it should be clear that Low values of B, C, and D are 1 (As A is the topmost node where B, C and D can reach). In same way, Low values of E, F, G are 3 and Low values of H, I, J are 6.

For any node u, when DFS starts, Low will be set to it's Disc 1st.

Then later on DFS will be performed on each of it's children v one by one, Low value of u can change it two case:

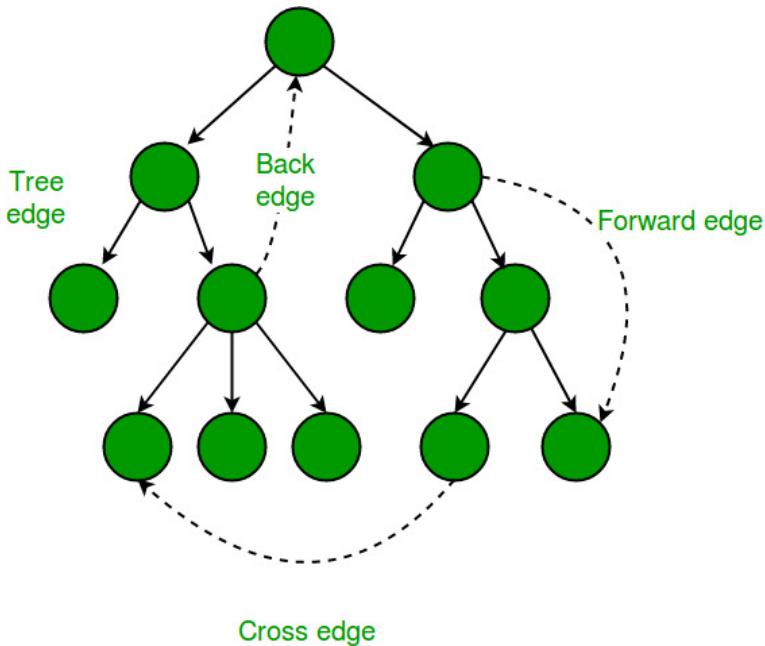
Case1 (Tree Edge): If node v is not visited already, then after DFS of v is complete, then minimum of low[u] and low[v] will be updated to low[u].

$\text{low}[u] = \min(\text{low}[u], \text{low}[v]);$

Case 2 (Back Edge): When child v is already visited, then minimum of low[u] and Disc[v] will be updated to low[u].

$\text{low}[u] = \min(\text{low}[u], \text{disc}[v]);$

In case two, can we take low[v] instead of disc[v] ?? . Answer is **NO**. If you can think why answer is **NO**, you probably understood the Low and Disc concept.



Same Low and Disc values help to solve other graph problems like [articulation point](#), [bridge](#) and [biconnected component](#).

To track the subtree rooted at head, we can use a stack (keep pushing node while visiting). When a head node found, pop all nodes from stack till you get head out of stack.

To make sure, we don't consider cross edges, when we reach a node which is already visited, we should process the visited node only if it is present in stack, else ignore the node.

Following is implementation of Tarjan's algorithm to print all SCCs.

C/C++

```
// A C++ program to find strongly connected components in a given
// directed graph using Tarjan's algorithm (single DFS)
#include<iostream>
#include <list>
#include <stack>
#define NIL -1
using namespace std;

// A class that represents an directed graph
class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // A dynamic array of adjacency lists

    // A Recursive DFS based function used by SCC()
    void SCCUtil(int u, int disc[], int low[],
                 stack<int> *st, bool stackMember[]);
}
```

```

public:
    Graph(int V); // Constructor
    void addEdge(int v, int w); // function to add an edge to graph
    void SCC(); // prints strongly connected components
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
}

// A recursive function that finds and prints strongly connected
// components using DFS traversal
// u --> The vertex to be visited next
// disc[] --> Stores discovery times of visited vertices
// low[] --> earliest visited vertex (the vertex with minimum
//           discovery time) that can be reached from subtree
//           rooted with current vertex
// *st --> To store all the connected ancestors (could be part
//           of SCC)
// stackMember[] --> bit/index array for faster check whether
//           a node is in stack
void Graph::SCCUtil(int u, int disc[], int low[], stack<int> *st,
                    bool stackMember[])
{
    // A static variable is used for simplicity, we can avoid use
    // of static variable by passing a pointer.
    static int time = 0;

    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;
    st->push(u);
    stackMember[u] = true;

    // Go through all vertices adjacent to this
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        int v = *i; // v is current adjacent of 'u'

        // If v is not visited yet, then recur for it
        if (disc[v] == -1)

```

```

{
    SCCUtil(v, disc, low, st, stackMember);

    // Check if the subtree rooted with 'v' has a
    // connection to one of the ancestors of 'u'
    // Case 1 (per above discussion on Disc and Low value)
    low[u] = min(low[u], low[v]);
}

// Update low value of 'u' only if 'v' is still in stack
// (i.e. it's a back edge, not cross edge).
// Case 2 (per above discussion on Disc and Low value)
else if (stackMember[v] == true)
    low[u] = min(low[u], disc[v]);
}

// head node found, pop the stack and print an SCC
int w = 0; // To store stack extracted vertices
if (low[u] == disc[u])
{
    while (st->top() != u)
    {
        w = (int) st->top();
        cout << w << " ";
        stackMember[w] = false;
        st->pop();
    }
    w = (int) st->top();
    cout << w << "n";
    stackMember[w] = false;
    st->pop();
}
}

// The function to do DFS traversal. It uses SCCUtil()
void Graph::SCC()
{
    int *disc = new int[V];
    int *low = new int[V];
    bool *stackMember = new bool[V];
    stack<int> *st = new stack<int>();

    // Initialize disc and low, and stackMember arrays
    for (int i = 0; i < V; i++)
    {
        disc[i] = NIL;
        low[i] = NIL;
        stackMember[i] = false;
    }
}

```

```
}

// Call the recursive helper function to find strongly
// connected components in DFS tree with vertex 'i'
for (int i = 0; i < V; i++)
    if (disc[i] == NIL)
        SCCUtil(i, disc, low, st, stackMember);
}

// Driver program to test above function
int main()
{
    cout << "nSCCs in first graph n";
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 1);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.SCC();

    cout << "nSCCs in second graph n";
    Graph g2(4);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.addEdge(2, 3);
    g2.SCC();

    cout << "nSCCs in third graph n";
    Graph g3(7);
    g3.addEdge(0, 1);
    g3.addEdge(1, 2);
    g3.addEdge(2, 0);
    g3.addEdge(1, 3);
    g3.addEdge(1, 4);
    g3.addEdge(1, 6);
    g3.addEdge(3, 5);
    g3.addEdge(4, 5);
    g3.SCC();

    cout << "nSCCs in fourth graph n";
    Graph g4(11);
    g4.addEdge(0,1);g4.addEdge(0,3);
    g4.addEdge(1,2);g4.addEdge(1,4);
    g4.addEdge(2,0);g4.addEdge(2,6);
    g4.addEdge(3,2);
    g4.addEdge(4,5);g4.addEdge(4,6);
    g4.addEdge(5,6);g4.addEdge(5,7);g4.addEdge(5,8);g4.addEdge(5,9);
```

```
g4.addEdge(6,4);
g4.addEdge(7,9);
g4.addEdge(8,9);
g4.addEdge(9,8);
g4.SCC();

cout << "nSCCs in fifth graph n";
Graph g5(5);
g5.addEdge(0,1);
g5.addEdge(1,2);
g5.addEdge(2,3);
g5.addEdge(2,4);
g5.addEdge(3,0);
g5.addEdge(4,2);
g5.SCC();

return 0;
}
```

Python

```
# Python program to find strongly connected components in a given
# directed graph using Tarjan's algorithm (single DFS)
#Complexity : O(V+E)

from collections import defaultdict

#This class represents an directed graph
# using adjacency list representation
class Graph:

    def __init__(self,vertices):
        #No. of vertices
        self.V= vertices

        # default dictionary to store graph
        self.graph = defaultdict(list)

        self.Time = 0

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    '''A recursive function that finds and prints strongly connected
    components using DFS traversal
    u --> The vertex to be visited next
```

```

disc[] --> Stores discovery times of visited vertices
low[] --> earliest visited vertex (the vertex with minimum
           discovery time) that can be reached from subtree
           rooted with current vertex
st --> To store all the connected ancestors (could be part
       of SCC)
stackMember[] --> bit/index array for faster check whether
                   a node is in stack
...
def SCCUtil(self,u, low, disc, stackMember, st):

    # Initialize discovery time and low value
    disc[u] = self.Time
    low[u] = self.Time
    self.Time += 1
    stackMember[u] = True
    st.append(u)

    # Go through all vertices adjacent to this
    for v in self.graph[u]:

        # If v is not visited yet, then recur for it
        if disc[v] == -1 :

            self.SCCUtil(v, low, disc, stackMember, st)

            # Check if the subtree rooted with v has a connection to
            # one of the ancestors of u
            # Case 1 (per above discussion on Disc and Low value)
            low[u] = min(low[u], low[v])

        elif stackMember[v] == True:

            '''Update low value of 'u' only if 'v' is still in stack
            (i.e. it's a back edge, not cross edge).
            Case 2 (per above discussion on Disc and Low value)'''
            low[u] = min(low[u], disc[v])

    # head node found, pop the stack and print an SCC
    w = -1 #To store stack extracted vertices
    if low[u] == disc[u]:
        while w != u:
            w = st.pop()
            print w,
            stackMember[w] = False

    print ""

```

```

#The function to do DFS traversal.
# It uses recursive SCCUtil()
def SCC(self):

    # Mark all the vertices as not visited
    # and Initialize parent and visited,
    # and ap(articulation point) arrays
    disc = [-1] * (self.V)
    low = [-1] * (self.V)
    stackMember = [False] * (self.V)
    st = []

    # Call the recursive helper function
    # to find articulation points
    # in DFS tree rooted with vertex 'i'
    for i in range(self.V):
        if disc[i] == -1:
            self.SCCUtil(i, low, disc, stackMember, st)

# Create a graph given in the above diagram
g1 = Graph(5)
g1.addEdge(1, 0)
g1.addEdge(0, 2)
g1.addEdge(2, 1)
g1.addEdge(0, 3)
g1.addEdge(3, 4)
print "SSC in first graph "
g1.SCC()

g2 = Graph(4)
g2.addEdge(0, 1)
g2.addEdge(1, 2)
g2.addEdge(2, 3)
print "nSSC in second graph "
g2.SCC()

g3 = Graph(7)
g3.addEdge(0, 1)
g3.addEdge(1, 2)
g3.addEdge(2, 0)

```

```
g3.addEdge(1, 3)
g3.addEdge(1, 4)
g3.addEdge(1, 6)
g3.addEdge(3, 5)
g3.addEdge(4, 5)
print "nSSC in third graph "
g3.SCC()

g4 = Graph(11)
g4.addEdge(0, 1)
g4.addEdge(0, 3)
g4.addEdge(1, 2)
g4.addEdge(1, 4)
g4.addEdge(2, 0)
g4.addEdge(2, 6)
g4.addEdge(3, 2)
g4.addEdge(4, 5)
g4.addEdge(4, 6)
g4.addEdge(5, 6)
g4.addEdge(5, 7)
g4.addEdge(5, 8)
g4.addEdge(5, 9)
g4.addEdge(6, 4)
g4.addEdge(7, 9)
g4.addEdge(8, 9)
g4.addEdge(9, 8)
print "nSSC in fourth graph "
g4.SCC();

g5 = Graph (5)
g5.addEdge(0, 1)
g5.addEdge(1, 2)
g5.addEdge(2, 3)
g5.addEdge(2, 4)
g5.addEdge(3, 0)
g5.addEdge(4, 2)
print "nSSC in fifth graph "
g5.SCC();
```

#This code is contributed by Neelam Yadav

Output:

```
SCCs in first graph
4
3
```

1 2 0

SCCs in second graph
3
2
1
0

SCCs in third graph
5
3
4
6
2 1 0

SCCs in fourth graph
8 9
7
5 4 6
3 2 1 0
10

SCCs in fifth graph
4 3 2 1 0

Time Complexity: The above algorithm mainly calls DFS, DFS takes $O(V+E)$ for a graph represented using adjacency list.

References:

http://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm
<http://www.ics.uci.edu/~eppstein/161/960220.html#sca>

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

<https://www.geeksforgeeks.org/tarjan-algorithm-find-strongly-connected-components/>

Chapter 50

Transitive closure of a graph

Transitive closure of a graph - GeeksforGeeks

Given a directed graph, find out if a vertex j is reachable from another vertex i for all vertex pairs (i, j) in the given graph. Here reachable mean that there is a path from vertex i to j. The reach-ability matrix is called transitive closure of a graph.

For example, consider below graph

Transitive closure of above graphs is

```
1 1 1 1  
1 1 1 1  
1 1 1 1  
0 0 0 1
```

The graph is given in the form of adjacency matrix say ‘graph[V][V]’ where $\text{graph}[i][j]$ is 1 if there is an edge from vertex i to vertex j or i is equal to j, otherwise $\text{graph}[i][j]$ is 0.

[Floyd Warshall Algorithm](#) can be used, we can calculate the distance matrix $\text{dist}[V][V]$ using [Floyd Warshall](#), if $\text{dist}[i][j]$ is infinite, then j is not reachable from i, otherwise j is reachable and value of $\text{dist}[i][j]$ will be less than V.

Instead of directly using Floyd Warshall, we can optimize it in terms of space and time, for this particular problem. Following are the optimizations:

- 1) Instead of integer resultant matrix ([dist\[V\]\[V\] in floyd warshall](#)), we can create a boolean reach-ability matrix $\text{reach}[V][V]$ (we save space). The value $\text{reach}[i][j]$ will be 1 if j is reachable from i, otherwise 0.
- 2) Instead of using arithmetic operations, we can use logical operations. For arithmetic operation ‘+’, logical and ‘&&’ is used, and for min, logical or ‘’ is used. (We save time by a constant factor. Time complexity is same though)

C++

```

// Program for transitive closure using Floyd Warshall Algorithm
#include<stdio.h>

// Number of vertices in the graph
#define V 4

// A function to print the solution matrix
void printSolution(int reach[][]);

// Prints transitive closure of graph[][] using Floyd Warshall algorithm
void transitiveClosure(int graph[][])
{
    /* reach[][] will be the output matrix that will finally have the
       shortest distances between every pair of vertices */
    int reach[V][V], i, j, k;

    /* Initialize the solution matrix same as input graph matrix. Or
       we can say the initial values of shortest distances are based
       on shortest paths considering no intermediate vertex. */
    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            reach[i][j] = graph[i][j];

    /* Add all vertices one by one to the set of intermediate vertices.
       ---> Before start of a iteration, we have reachability values for
           all pairs of vertices such that the reachability values
           consider only the vertices in set {0, 1, 2, .. k-1} as
           intermediate vertices.
       ----> After the end of a iteration, vertex no. k is added to the
           set of intermediate vertices and the set becomes {0, 1, .. k} */
    for (k = 0; k < V; k++)
    {
        // Pick all vertices as source one by one
        for (i = 0; i < V; i++)
        {
            // Pick all vertices as destination for the
            // above picked source
            for (j = 0; j < V; j++)
            {
                // If vertex k is on a path from i to j,
                // then make sure that the value of reach[i][j] is 1
                // reach[i][j] = reach[i][j] || (reach[i][k] && reach[k][j]);
                reach[i][j] = reach[i][j] || (reach[i][k] && reach[k][j]);
            }
        }
    }

    // Print the shortest distance matrix
    printSolution(reach);
}

```

```

}

/* A utility function to print solution */
void printSolution(int reach[][] [V])
{
    printf ("Following matrix is transitive closure of the given graph\n");
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
            printf ("%d ", reach[i][j]);
        printf("\n");
    }
}

// driver program to test above function
int main()
{
    /* Let us create the following weighted graph
       10
       (0)----->(3)
           |           /|\
           5 |           |
           |           | 1
           \|/           |
       (1)----->(2)
           3           */
    int graph[V][V] = { {1, 1, 0, 1},
                        {0, 1, 1, 0},
                        {0, 0, 1, 1},
                        {0, 0, 0, 1}
    };

    // Print the solution
    transitiveClosure(graph);
    return 0;
}

```

Java

```

// Program for transitive closure using Floyd Warshall Algorithm
import java.util.*;
import java.lang.*;
import java.io.*;

class GraphClosure
{
    final static int V = 4; //Number of vertices in a graph

```

```

// Prints transitive closure of graph[][] using Floyd
// Warshall algorithm
void transitiveClosure(int graph[][])
{
    /* reach[][] will be the output matrix that will finally
       have the shortest distances between every pair of
       vertices */
    int reach[][] = new int[V][V];
    int i, j, k;

    /* Initialize the solution matrix same as input graph
       matrix. Or we can say the initial values of shortest
       distances are based on shortest paths considering
       no intermediate vertex. */
    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            reach[i][j] = graph[i][j];

    /* Add all vertices one by one to the set of intermediate
       vertices.
       ---> Before start of a iteration, we have reachability
           values for all pairs of vertices such that the
           reachability values consider only the vertices in
           set {0, 1, 2, .. k-1} as intermediate vertices.
       ----> After the end of a iteration, vertex no. k is
           added to the set of intermediate vertices and the
           set becomes {0, 1, 2, .. k} */
    for (k = 0; k < V; k++)
    {
        // Pick all vertices as source one by one
        for (i = 0; i < V; i++)
        {
            // Pick all vertices as destination for the
            // above picked source
            for (j = 0; j < V; j++)
            {
                // If vertex k is on a path from i to j,
                // then make sure that the value of reach[i][j] is 1
                reach[i][j] = (reach[i][j]!=0) ||
                               ((reach[i][k]!=0) && (reach[k][j]!=0))?1:0;
            }
        }
    }

    // Print the shortest distance matrix
    printSolution(reach);
}

```

```

/* A utility function to print solution */
void printSolution(int reach[][])
{
    System.out.println("Following matrix is transitive closure"+
                       " of the given graph");
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
            System.out.print(reach[i][j]+" ");
        System.out.println();
    }
}

// Driver program to test above function
public static void main (String[] args)
{
    /* Let us create the following weighted graph
       10
       (0)----->(3)
       |           /|\
      5 |           |
       |           | 1
      \|/           |
       (1)----->(2)
       3           */
}

/* Let us create the following weighted graph

       10
       (0)----->(3)
       |           /|\
      5 |           |
       |           | 1
      \|/           |
       (1)----->(2)
       3           */
    int graph[][] = new int[][]{{1, 1, 0, 1},
                               {0, 1, 1, 0},
                               {0, 0, 1, 1},
                               {0, 0, 0, 1}};
}

// Print the solution
GraphClosure g = new GraphClosure();
g.transitiveClosure(graph);
}
}

// This code is contributed by Aakash Hasija

```

Python

```
# Python program for transitive closure using Floyd Warshall Algorithm
#Complexity : O(V^3)

from collections import defaultdict

#Class to represent a graph
class Graph:

    def __init__(self, vertices):
        self.V = vertices

        # A utility function to print the solution
    def printSolution(self, reach):
        print ("Following matrix transitive closure of the given graph ")
        for i in range(self.V):
            for j in range(self.V):
                print "%7d\t" %(reach[i][j]),
            print ""

    # Prints transitive closure of graph[][] using Floyd Warshall algorithm
    def transitiveClosure(self,graph):
        '''reach[][] will be the output matrix that will finally
        have reachability values.
        Initialize the solution matrix same as input graph matrix'''
        reach =[i[:] for i in graph]
        '''Add all vertices one by one to the set of intermediate
        vertices.
        ---> Before start of a iteration, we have reachability value
        for all pairs of vertices such that the reachability values
        consider only the vertices in set
        {0, 1, 2, .. k-1} as intermediate vertices.
        ----> After the end of an iteration, vertex no. k is
        added to the set of intermediate vertices and the
        set becomes {0, 1, 2, .. k}'''
        for k in range(self.V):

            # Pick all vertices as source one by one
            for i in range(self.V):

                # Pick all vertices as destination for the
                # above picked source
                for j in range(self.V):

                    # If vertex k is on a path from i to j,
                    # then make sure that the value of reach[i][j] is 1
```

```
reach[i][j] = reach[i][j] or (reach[i][k] and reach[k][j])  
  
self.printSolution(reach)  
  
g= Graph(4)  
  
graph = [[1, 1, 0, 1],  
         [0, 1, 1, 0],  
         [0, 0, 1, 1],  
         [0, 0, 0, 1]]  
  
#Print the solution  
g.transitiveClosure(graph)  
  
#This code is contributed by Neelam Yadav
```

Output:

```
Following matrix is transitive closure of the given graph  
1 1 1 1  
0 1 1 1  
0 0 1 1  
0 0 0 1
```

Time Complexity: $O(V^3)$ where V is number of vertices in the given graph.

See below post for a $O(V^2)$ solution.

[Transitive Closure of a Graph using DFS](#)

References:

[Introduction to Algorithms](#) by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.

Source

<https://www.geeksforgeeks.org/transitive-closure-of-a-graph/>

Chapter 51

Find the number of islands

Find the number of islands Set 1 (Using DFS) - GeeksforGeeks

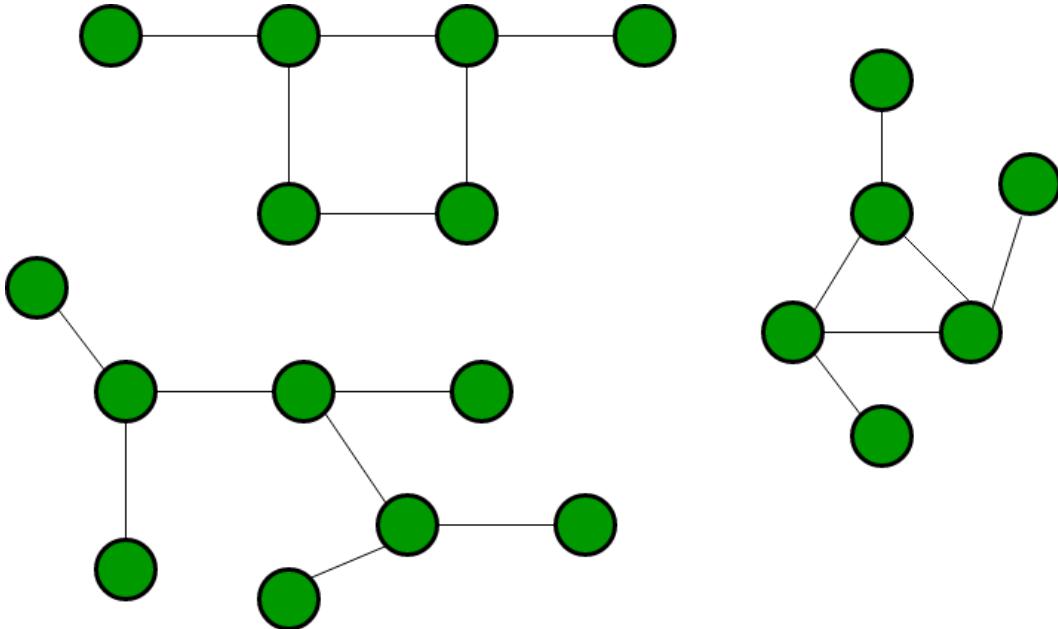
Given a boolean 2D matrix, find the number of islands. A group of connected 1s forms an island. For example, the below matrix contains 5 islands

Example:

```
Input : mat[][] = {{1, 1, 0, 0, 0},  
                  {0, 1, 0, 0, 1},  
                  {1, 0, 0, 1, 1},  
                  {0, 0, 0, 0, 0},  
                  {1, 0, 1, 0, 1}}  
Output : 5
```

This is a variation of the standard problem: “Counting the number of connected components in an undirected graph”.

Before we go to the problem, let us understand what is a connected component. A [connected component](#) of an undirected graph is a subgraph in which every two vertices are connected to each other by a path(s), and which is connected to no other vertices outside the subgraph. For example, the graph shown below has three connected components.



A graph where all vertices are connected with each other has exactly one connected component, consisting of the whole graph. Such graph with only one connected component is called as Strongly Connected Graph.

The problem can be easily solved by applying DFS() on each component. In each DFS() call, a component or a sub-graph is visited. We will call DFS on the next un-visited component. The number of calls to DFS() gives the number of connected components. BFS can also be used.

What is an island?

A group of connected 1s forms an island. For example, the below matrix contains 5 islands

```
{1, 1, 0, 0, 0},
{0, 1, 0, 0, 1},
{1, 0, 0, 1, 1},
{0, 0, 0, 0, 0},
{1, 0, 1, 0, 1}
```

A cell in 2D matrix can be connected to 8 neighbours. So, unlike standard DFS(), where we recursively call for all adjacent vertices, here we can recursively call for 8 neighbours only. We keep track of the visited 1s so that they are not visited again.

C/C++

```
// Program to count islands in boolean 2D matrix
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
```

```

#define ROW 5
#define COL 5

// A function to check if a given cell (row, col) can be included in DFS
int isSafe(int M[] [COL], int row, int col, bool visited[] [COL])
{
    // row number is in range, column number is in range and value is 1
    // and not yet visited
    return (row >= 0) && (row < ROW) &&
           (col >= 0) && (col < COL) &&
           (M[row] [col] && !visited[row] [col]);
}

// A utility function to do DFS for a 2D boolean matrix. It only considers
// the 8 neighbours as adjacent vertices
void DFS(int M[] [COL], int row, int col, bool visited[] [COL])
{
    // These arrays are used to get row and column numbers of 8 neighbours
    // of a given cell
    static int rowNbr[] = {-1, -1, -1, 0, 0, 1, 1, 1};
    static int colNbr[] = {-1, 0, 1, -1, 1, -1, 0, 1};

    // Mark this cell as visited
    visited[row] [col] = true;

    // Recur for all connected neighbours
    for (int k = 0; k < 8; ++k)
        if (isSafe(M, row + rowNbr[k], col + colNbr[k], visited) )
            DFS(M, row + rowNbr[k], col + colNbr[k], visited);
}

// The main function that returns count of islands in a given boolean
// 2D matrix
int countIslands(int M[] [COL])
{
    // Make a bool array to mark visited cells.
    // Initially all cells are unvisited
    bool visited[ROW] [COL];
    memset(visited, 0, sizeof(visited));

    // Initialize count as 0 and traverse through the all cells of
    // given matrix
    int count = 0;
    for (int i = 0; i < ROW; ++i)
        for (int j = 0; j < COL; ++j)
            if (M[i] [j] && !visited[i] [j]) // If a cell with value 1 is not
            {                                // visited yet, then new island found

```

```
        DFS(M, i, j, visited);      // Visit all cells in this island.
        ++count;                  // and increment island count
    }

    return count;
}

// Driver program to test above function
int main()
{
    int M[][] [COL]={ {1, 1, 0, 0, 0},
                      {0, 1, 0, 0, 1},
                      {1, 0, 0, 1, 1},
                      {0, 0, 0, 0, 0},
                      {1, 0, 1, 0, 1}
    };

    printf("Number of islands is: %d\n", countIslands(M));

    return 0;
}
```

Java

```
// Java program to count islands in boolean 2D matrix
import java.util.*;
import java.lang.*;
import java.io.*;

class Islands
{
    //No of rows and columns
    static final int ROW = 5, COL = 5;

    // A function to check if a given cell (row, col) can
    // be included in DFS
    boolean isSafe(int M[][], int row, int col,
                  boolean visited[][])
    {
        // row number is in range, column number is in range
        // and value is 1 and not yet visited
        return (row >= 0) && (row < ROW) &&
               (col >= 0) && (col < COL) &&
               (M[row][col]==1 && !visited[row][col]);
    }

    // A utility function to do DFS for a 2D boolean matrix.
    // It only considers the 8 neighbors as adjacent vertices
```

```

void DFS(int M[][] , int row, int col, boolean visited[][])
{
    // These arrays are used to get row and column numbers
    // of 8 neighbors of a given cell
    int rowNbr[] = new int[] {-1, -1, -1, 0, 0, 1, 1, 1};
    int colNbr[] = new int[] {-1, 0, 1, -1, 1, -1, 0, 1};

    // Mark this cell as visited
    visited[row][col] = true;

    // Recur for all connected neighbours
    for (int k = 0; k < 8; ++k)
        if (isSafe(M, row + rowNbr[k], col + colNbr[k], visited) )
            DFS(M, row + rowNbr[k], col + colNbr[k], visited);
}

// The main function that returns count of islands in a given
// boolean 2D matrix
int countIslands(int M[][])
{
    // Make a bool array to mark visited cells.
    // Initially all cells are unvisited
    boolean visited[][] = new boolean[ROW][COL];

    // Initialize count as 0 and traverse through the all cells
    // of given matrix
    int count = 0;
    for (int i = 0; i < ROW; ++i)
        for (int j = 0; j < COL; ++j)
            if (M[i][j]==1 && !visited[i][j]) // If a cell with
            {                                     // value 1 is not
                // visited yet, then new island found, Visit all
                // cells in this island and increment island count
                DFS(M, i, j, visited);
                ++count;
            }
    return count;
}

// Driver method
public static void main (String[] args) throws java.lang.Exception
{
    int M[][]= new int[][] {{1, 1, 0, 0, 0},
                           {0, 1, 0, 0, 1},
                           {1, 0, 0, 1, 1},
                           {0, 0, 0, 0, 0},
                           {0, 0, 1, 0, 1}};
}

```

```

        {1, 0, 1, 0, 1}
    };
Islands I = new Islands();
System.out.println("Number of islands is: "+ I.countIslands(M));
}
} //Contributed by Aakash Hasija

```

Python

```

# Program to count islands in boolean 2D matrix
class Graph:

    def __init__(self, row, col, g):
        self.ROW = row
        self.COL = col
        self.graph = g

    # A function to check if a given cell
    # (row, col) can be included in DFS
    def isSafe(self, i, j, visited):
        # row number is in range, column number
        # is in range and value is 1
        # and not yet visited
        return (i >= 0 and i < self.ROW and
                j >= 0 and j < self.COL and
                not visited[i][j] and self.graph[i][j])

    # A utility function to do DFS for a 2D
    # boolean matrix. It only considers
    # the 8 neighbours as adjacent vertices
    def DFS(self, i, j, visited):

        # These arrays are used to get row and
        # column numbers of 8 neighbours
        # of a given cell
        rowNbr = [-1, -1, -1, 0, 0, 1, 1, 1];
        colNbr = [-1, 0, 1, -1, 1, -1, 0, 1];

        # Mark this cell as visited
        visited[i][j] = True

        # Recur for all connected neighbours
        for k in range(8):
            if self.isSafe(i + rowNbr[k], j + colNbr[k], visited):
                self.DFS(i + rowNbr[k], j + colNbr[k], visited)

```

```
# The main function that returns
# count of islands in a given boolean
# 2D matrix
def countIslands(self):
    # Make a bool array to mark visited cells.
    # Initially all cells are unvisited
    visited = [[False for j in range(self.COL)]for i in range(self.ROW)]

    # Initialize count as 0 and traverse
    # through the all cells of
    # given matrix
    count = 0
    for i in range(self.ROW):
        for j in range(self.COL):
            # If a cell with value 1 is not visited yet,
            # then new island found
            if visited[i][j] == False and self.graph[i][j] == 1:
                # Visit all cells in this island
                # and increment island count
                self.DFS(i, j, visited)
                count += 1

    return count

graph = [[1, 1, 0, 0, 0],
         [0, 1, 0, 0, 1],
         [1, 0, 0, 1, 1],
         [0, 0, 0, 0, 0],
         [1, 0, 1, 0, 1]]

row = len(graph)
col = len(graph[0])

g= Graph(row, col, graph)

print "Number of islands is:"
print g.countIslands()

#This code is contributed by Neelam Yadav
```

C#

```
// C# program to count
// islands in boolean
// 2D matrix
using System;
```

```

class GFG
{
    // No of rows
    // and columns
    static int ROW = 5, COL = 5;

    // A function to check if
    // a given cell (row, col)
    // can be included in DFS
    static bool isSafe(int [,]M, int row,
                       int col, bool [,]visited)
    {
        // row number is in range,
        // column number is in range
        // and value is 1 and not
        // yet visited
        return (row >= 0) && (row < ROW) &&
               (col >= 0) && (col < COL) &&
               (M[row, col] == 1 &&
                !visited[row, col]);
    }

    // A utility function to do
    // DFS for a 2D boolean matrix.
    // It only considers the 8
    // neighbors as adjacent vertices
    static void DFS(int [,]M, int row,
                    int col, bool [,]visited)
    {
        // These arrays are used to
        // get row and column numbers
        // of 8 neighbors of a given cell
        int []rowNbr = new int[] {-1, -1, -1, 0,
                                0, 1, 1, 1};
        int []colNbr = new int[] {-1, 0, 1, -1,
                                1, -1, 0, 1};

        // Mark this cell
        // as visited
        visited[row, col] = true;

        // Recur for all
        // connected neighbours
        for (int k = 0; k < 8; ++k)
            if (isSafe(M, row + rowNbr[k], col +
                       colNbr[k], visited))
                DFS(M, row + rowNbr[k],

```

```
        col + colNbr[k], visited);
    }

    // The main function that
    // returns count of islands
    // in a given boolean 2D matrix
    static int countIslands(int [,]M)
    {
        // Make a bool array to
        // mark visited cells.
        // Initially all cells
        // are unvisited
        bool [,]visited = new bool[ROW, COL];

        // Initialize count as 0 and
        // traverse through the all
        // cells of given matrix
        int count = 0;
        for(int i = 0; i < ROW; ++i)
            for (int j = 0; j < COL; ++j)
                if (M[i, j] == 1 &&
                    !visited[i, j])
                {
                    // If a cell with value 1 is not
                    // visited yet, then new island
                    // found, Visit all cells in this
                    // island and increment island count
                    DFS(M, i, j, visited);
                    ++count;
                }
        return count;
    }

    // Driver Code
    public static void Main ()
    {
        int [,]M = new int[,] {{1, 1, 0, 0, 0},
                               {0, 1, 0, 0, 1},
                               {1, 0, 0, 1, 1},
                               {0, 0, 0, 0, 0},
                               {1, 0, 1, 0, 1}};
        Console.WriteLine("Number of islands is: " +
                          countIslands(M));
    }
}
```

```
// This code is contributed
// by shiv_bhakt.
```

PHP

```
<?php
// Program to count islands
// in boolean 2D matrix

$ROW = 5;
$COL = 5;

// A function to check if a
// given cell (row, col) can
// be included in DFS
function isSafe(&$M, $row, $col,
    &$visited)
{
    global $ROW, $COL;

    // row number is in range,
    // column number is in
    // range and value is 1
    // and not yet visited
    return ($row >= 0) && ($row < $ROW) &&
        ($col >= 0) && ($col < $COL) &&
        ($M[$row][$col] &&
        !isset($visited[$row][$col]));
}

// A utility function to do DFS
// for a 2D boolean matrix. It
// only considers the 8 neighbours
// as adjacent vertices
function DFS(&$M, $row, $col,
    &$visited)
{
    // These arrays are used to
    // get row and column numbers
    // of 8 neighbours of a given cell
    $rowNbr = array(-1, -1, -1, 0,
        0, 1, 1, 1);
    $colNbr = array(-1, 0, 1, -1,
        1, -1, 0, 1);

    // Mark this cell as visited
    $visited[$row][$col] = true;
```

```

// Recur for all
// connected neighbours
for ($k = 0; $k < 8; ++$k)
    if (isSafe($M, $row + $rowNbr[$k],
               $col + $colNbr[$k], $visited))
        DFS($M, $row + $rowNbr[$k],
             $col + $colNbr[$k], $visited);
}

// The main function that returns
// count of islands in a given
// boolean 2D matrix
function countIslands(&$M)
{
    global $ROW, $COL;

    // Make a bool array to
    // mark visited cells.
    // Initially all cells
    // are unvisited
    $visited = array(array());

    // Initialize count as 0 and
    // traverse through the all
    // cells of given matrix
    $count = 0;
    for ($i = 0; $i < $ROW; ++$i)
        for ($j = 0; $j < $COL; ++$j)
            if ($M[$i][$j] &&
                !isset($visited[$i][$j])) // If a cell with value 1
            {
                // is not visited yet,
                DFS($M, $i, $j, $visited); // then new island found
                ++$count; // Visit all cells in this
            } // island and increment
            // island count.

    return $count;
}

// Driver Code
$M = array(array(1, 1, 0, 0, 0),
           array(0, 1, 0, 0, 1),
           array(1, 0, 0, 1, 1),
           array(0, 0, 0, 0, 0),
           array(1, 0, 1, 0, 1));

echo "Number of islands is: ",
     countIslands($M);

```

```
// This code is contributed  
// by ChitraNayal  
?>
```

Output:

Number of islands is: 5

Time complexity: O(ROW x COL)

[Find the number of Islands Set 2 \(Using Disjoint Set\)](#)

Reference:

http://en.wikipedia.org/wiki/Connected_component_%28graph_theory%29

[Improved By : shiv_bhakt, ChitraNayal](#)

Source

<https://www.geeksforgeeks.org/find-number-of-islands/>

Chapter 52

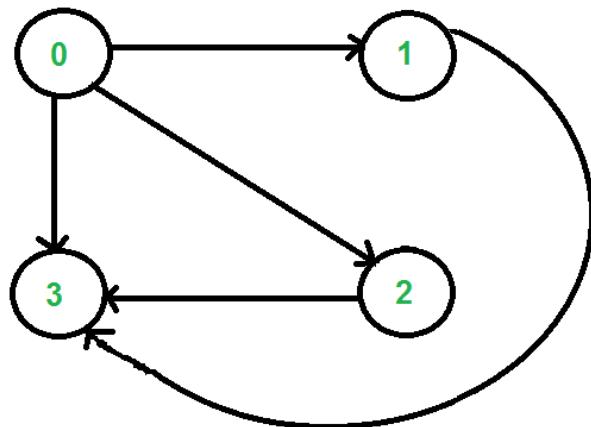
Count all possible walks from a source to a destination with exactly k edges

Count all possible walks from a source to a destination with exactly k edges - GeeksforGeeks

Given a directed graph and two vertices 'u' and 'v' in it, count all possible walks from 'u' to 'v' with exactly k edges on the walk.

The graph is given as [adjacency matrix representation](#) where value of $\text{graph}[i][j]$ as 1 indicates that there is an edge from vertex i to vertex j and a value 0 indicates no edge from i to j.

For example consider the following graph. Let source 'u' be vertex 0, destination 'v' be 3 and k be 2. The output should be 2 as there are two walk from 0 to 3 with exactly 2 edges. The walks are {0, 2, 3} and {0, 1, 3}



A **simple solution** is to start from u, go to all adjacent vertices and recur for adjacent

vertices with k as $k-1$, source as adjacent vertex and destination as v . Following is the implementation of this simple solution.

C++

```
// C++ program to count walks from u to v with exactly k edges
#include <iostream>
using namespace std;

// Number of vertices in the graph
#define V 4

// A naive recursive function to count walks from u to v with k edges
int countwalks(int graph[][][V], int u, int v, int k)
{
    // Base cases
    if (k == 0 && u == v)      return 1;
    if (k == 1 && graph[u][v]) return 1;
    if (k <= 0)                return 0;

    // Initialize result
    int count = 0;

    // Go to all adjacents of u and recur
    for (int i = 0; i < V; i++)
        if (graph[u][i] == 1) // Check if is adjacent of u
            count += countwalks(graph, i, v, k-1);

    return count;
}

// driver program to test above function
int main()
{
    /* Let us create the graph shown in above diagram*/
    int graph[V][V] = { {0, 1, 1, 1},
                        {0, 0, 0, 1},
                        {0, 0, 0, 1},
                        {0, 0, 0, 0}
                      };
    int u = 0, v = 3, k = 2;
    cout << countwalks(graph, u, v, k);
    return 0;
}
```

Java

```
// Java program to count walks from u to v with exactly k edges
```

```
import java.util.*;
import java.lang.*;
import java.io.*;

class KPaths
{
    static final int V = 4; //Number of vertices

    // A naive recursive function to count walks from u
    // to v with k edges
    int countwalks(int graph[][] , int u, int v, int k)
    {
        // Base cases
        if (k == 0 && u == v)           return 1;
        if (k == 1 && graph[u][v] == 1)  return 1;
        if (k <= 0)                     return 0;

        // Initialize result
        int count = 0;

        // Go to all adjacents of u and recur
        for (int i = 0; i < V; i++)
            if (graph[u][i] == 1) // Check if is adjacent of u
                count += countwalks(graph, i, v, k-1);

        return count;
    }

    // Driver method
    public static void main (String[] args) throws java.lang.Exception
    {
        /* Let us create the graph shown in above diagram*/
        int graph[][] =new int[][] { {0, 1, 1, 1},
                                    {0, 0, 0, 1},
                                    {0, 0, 0, 1},
                                    {0, 0, 0, 0}
                                };
        int u = 0, v = 3, k = 2;
        KPaths p = new KPaths();
        System.out.println(p.countwalks(graph, u, v, k));
    }
}//Contributed by Aakash Hasija
```

Python3

```
# Python3 program to count walks from
# u to v with exactly k edges
```

```
# Number of vertices in the graph
V = 4

# A naive recursive function to count
# walks from u to v with k edges
def countwalks(graph, u, v, k):

    # Base cases
    if (k == 0 and u == v):
        return 1
    if (k == 1 and graph[u][v]):
        return 1
    if (k <= 0):
        return 0

    # Initialize result
    count = 0

    # Go to all adjacents of u and recur
    for i in range(0, V):

        # Check if is adjacent of u
        if (graph[u][i] == 1):
            count += countwalks(graph, i, v, k-1)

    return count

# Driver Code

# Let us create the graph shown in above diagram
graph = [[0, 1, 1, 1], 
          [0, 0, 0, 1], 
          [0, 0, 0, 1], 
          [0, 0, 0, 0] ]

u = 0; v = 3; k = 2
print(countwalks(graph, u, v, k))

# This code is contributed by Smitha Dinesh Semwal.

C#
// C# program to count walks from u to
// v with exactly k edges
using System;

class GFG {
```

```
// Number of vertices
static int V = 4;

// A naive recursive function to
// count walks from u to v with
// k edges
static int countwalks(int [,]graph, int u,
                      int v, int k)
{

    // Base cases
    if (k == 0 && u == v)
        return 1;
    if (k == 1 && graph[u,v] == 1)
        return 1;
    if (k <= 0)
        return 0;

    // Initialize result
    int count = 0;

    // Go to all adjacents of u and recur
    for (int i = 0; i < V; i++)

        // Check if is adjacent of u
        if (graph[u,i] == 1)
            count +=
                countwalks(graph, i, v, k-1);

    return count;
}

// Driver method
public static void Main ()
{

    /* Let us create the graph shown
    in above diagram*/
    int [,]graph =
        new int[,] { {0, 1, 1, 1},
                    {0, 0, 0, 1},
                    {0, 0, 0, 1},
                    {0, 0, 0, 0} };

    int u = 0, v = 3, k = 2;

    Console.WriteLine(
        countwalks(graph, u, v, k));
}
```

```
    }
}

// This code is contributed by nitin mittal.
```

PHP

```
<?php
// PHP program to count walks from u
// to v with exactly k edges

// Number of vertices in the graph
$V = 4;

// A naive recursive function to count
// walks from u to v with k edges
function countwalks( $graph, $u, $v, $k)
{
    global $V;

    // Base cases
    if ($k == 0 and $u == $v)
        return 1;
    if ($k == 1 and $graph[$u] [$v])
        return 1;
    if ($k <= 0)
        return 0;

    // Initialize result
    $count = 0;

    // Go to all adjacents of u and recur
    for ( $i = 0; $i < $V; $i++)

        // Check if is adjacent of u
        if ($graph[$u] [$i] == 1)
            $count += countwalks($graph, $i,
                $v, $k - 1);

    return $count;
}

// Driver Code
/* Let us create the graph
   shown in above diagram*/
$graph = array(array(0, 1, 1, 1),
              array(0, 0, 0, 1),
              array(0, 0, 0, 1),
```

```
        array(0, 0, 0, 0));
$u = 0; $v = 3; $k = 2;
echo countwalks($graph, $u, $v, $k);

// This code is contributed by anuj_67.
?>
```

Output:

2

The worst case time complexity of the above function is $O(V^k)$ where V is the number of vertices in the given graph. We can simply analyze the time complexity by drawing recursion tree. The worst occurs for a complete graph. In worst case, every internal node of recursion tree would have exactly n children.

We can optimize the above solution using **Dynamic Programming**. The idea is to build a 3D table where first dimension is source, second dimension is destination, third dimension is number of edges from source to destination, and the value is count of walks. Like other [Dynamic Programming problems](#), we fill the 3D table in bottom up manner.

C++

```
// C++ program to count walks from u to v with exactly k edges
#include <iostream>
using namespace std;

// Number of vertices in the graph
#define V 4

// A Dynamic programming based function to count walks from u
// to v with k edges
int countwalks(int graph[][V], int u, int v, int k)
{
    // Table to be filled up using DP. The value count[i][j][e] will
    // store count of possible walks from i to j with exactly k edges
    int count[V][V][k+1];

    // Loop for number of edges from 0 to k
    for (int e = 0; e <= k; e++)
    {
        for (int i = 0; i < V; i++) // for source
        {
            for (int j = 0; j < V; j++) // for destination
            {
                // initialize value
                count[i][j][e] = 0;

                // from base cases
                if (e == 0)
                    count[i][j][e] = 1;
                else if (i == j)
                    count[i][j][e] = e + 1;
                else
                    for (int l = 0; l < V; l++)
                        count[i][j][e] += count[i][l][e-1];
            }
        }
    }
}
```

```

        if (e == 0 && i == j)
            count[i][j][e] = 1;
        if (e == 1 && graph[i][j])
            count[i][j][e] = 1;

        // go to adjacent only when number of edges is more than 1
        if (e > 1)
        {
            for (int a = 0; a < V; a++) // adjacent of source i
                if (graph[i][a])
                    count[i][j][e] += count[a][j][e-1];
        }
    }
}
return count[u][v][k];
}

// driver program to test above function
int main()
{
    /* Let us create the graph shown in above diagram*/
    int graph[V][V] = { {0, 1, 1, 1},
                        {0, 0, 0, 1},
                        {0, 0, 0, 1},
                        {0, 0, 0, 0}
                      };
    int u = 0, v = 3, k = 2;
    cout << countwalks(graph, u, v, k);
    return 0;
}

```

Java

```

// Java program to count walks from u to v with exactly k edges
import java.util.*;
import java.lang.*;
import java.io.*;

class KPaths
{
    static final int V = 4; //Number of vertices

    // A Dynamic programming based function to count walks from u
    // to v with k edges
    int countwalks(int graph[][], int u, int v, int k)
    {
        // Table to be filled up using DP. The value count[i][j][e]

```

```
// will/ store count of possible walks from i to j with
// exactly k edges
int count[][][] = new int[V][V][k+1];

// Loop for number of edges from 0 to k
for (int e = 0; e <= k; e++)
{
    for (int i = 0; i < V; i++) // for source
    {
        for (int j = 0; j < V; j++) // for destination
        {
            // initialize value
            count[i][j][e] = 0;

            // from base cases
            if (e == 0 && i == j)
                count[i][j][e] = 1;
            if (e == 1 && graph[i][j] != 0)
                count[i][j][e] = 1;

            // go to adjacent only when number of edges
            // is more than 1
            if (e > 1)
            {
                for (int a = 0; a < V; a++) // adjacent of i
                    if (graph[i][a] != 0)
                        count[i][j][e] += count[a][j][e-1];
            }
        }
    }
}
return count[u][v][k];
}

// Driver method
public static void main (String[] args) throws java.lang.Exception
{
    /* Let us create the graph shown in above diagram*/
    int graph[][] =new int[][] {{0, 1, 1, 1},
                                {0, 0, 0, 1},
                                {0, 0, 0, 1},
                                {0, 0, 0, 0}};
    int u = 0, v = 3, k = 2;
    KPaths p = new KPaths();
    System.out.println(p.countwalks(graph, u, v, k));
}
}//Contributed by Aakash Hasija
```

C#

```
// C# program to count walks from u to v
// with exactly k edges
using System;

class GFG {
    static int V = 4; //Number of vertices

    // A Dynamic programming based function
    // to count walks from u to v with k edges
    static int countwalks(int [,]graph, int u,
                          int v, int k)
    {
        // Table to be filled up using DP. The
        // value count[i][j][e] will/ store
        // count of possible walks from i to
        // j with exactly k edges
        int [, ,]count = new int[V,V,k+1];

        // Loop for number of edges from 0 to k
        for (int e = 0; e <= k; e++)
        {

            // for source
            for (int i = 0; i < V; i++)
            {

                // for destination
                for (int j = 0; j < V; j++)
                {
                    // initialize value
                    count[i,j,e] = 0;

                    // from base cases
                    if (e == 0 && i == j)
                        count[i,j,e] = 1;
                    if (e == 1 && graph[i,j] != 0)
                        count[i,j,e] = 1;

                    // go to adjacent only when
                    // number of edges
                    // is more than 1
                    if (e > 1)
                    {
                        // adjacent of i
                        for (int a = 0; a < V; a++)
                            if (graph[i,a]!=0)
```

```
        count[i,j,e] +=  
        count[a,j,e-1];  
    }  
}  
}  
  
return count[u,v,k];  
}  
  
// Driver method  
public static void Main ()  
{  
    /* Let us create the graph shown in  
    above diagram*/  
    int [,]graph = { {0, 1, 1, 1},  
                    {0, 0, 0, 1},  
                    {0, 0, 0, 1},  
                    {0, 0, 0, 0} };  
    int u = 0, v = 3, k = 2;  
  
    Console.WriteLine(  
        countwalks(graph, u, v, k));  
}  
}  
  
// This is Code Contributed by anuj_67.
```

Output:

2

Time complexity of the above DP based solution is $O(V^3K)$ which is much better than the naive solution.

We can also use **Divide and Conquer** to solve the above problem in $O(V^3 \log k)$ time. The count of walks of length k from u to v is the $[u][v]$ 'th entry in $(\text{graph}[V][V])^k$. We can calculate power of by doing $O(\log k)$ multiplication by using the [divide and conquer technique to calculate power](#). A multiplication between two matrices of size $V \times V$ takes $O(V^3)$ time. Therefore overall time complexity of this method is $O(V^3 \log k)$.

Improved By : [nitin mittal, vt_m](#)

Source

<https://www.geeksforgeeks.org/count-possible-paths-source-destination-exactly-k-edges/>

Chapter 53

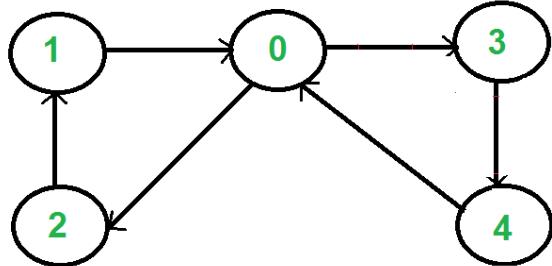
Euler Circuit in a Directed Graph

Euler Circuit in a Directed Graph - GeeksforGeeks

Eulerian Path is a path in graph that visits every edge exactly once. Eulerian Circuit is an Eulerian Path which starts and ends on the same vertex.

A graph is said to be eulerian if it has eulerian cycle. We have discussed [eulerian circuit for an undirected graph](#). In this post, same is discussed for a directed graph.

For example, the following graph has eulerian cycle as {1, 0, 3, 4, 0, 2, 1}



How to check if a directed graph is eulerian?

A directed graph has an eulerian cycle if following conditions are true (Source: [Wiki](#))

- 1) All vertices with nonzero degree belong to a single **strongly connected component**.
- 2) In degree and out degree of every vertex is same.

We can detect singly connected component using [Kosaraju's DFS based simple algorithm](#). To compare in degree and out degree, we need to store in degree and out degree of every vertex. Out degree can be obtained by size of adjacency list. In degree can be stored by creating an array of size equal to number of vertices.

Following are C++ and Java implementations of above approach.

C++

```

// A C++ program to check if a given directed graph is Eulerian or not
#include<iostream>
#include <list>
#define CHAR 26
using namespace std;

// A class that represents an undirected graph
class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // A dynamic array of adjacency lists
    int *in;
public:
    // Constructor and destructor
    Graph(int V);
    ~Graph() { delete [] adj; delete [] in; }

    // function to add an edge to graph
    void addEdge(int v, int w) { adj[v].push_back(w); (in[w])++; }

    // Method to check if this graph is Eulerian or not
    bool isEulerianCycle();

    // Method to check if all non-zero degree vertices are connected
    bool isSC();

    // Function to do DFS starting from v. Used in isConnected();
    void DFSUtil(int v, bool visited[]);

    Graph getTranspose();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
    in = new int[V];
    for (int i = 0; i < V; i++)
        in[i] = 0;
}

/* This function returns true if the directed graph has an eulerian
cycle, otherwise returns false */
bool Graph::isEulerianCycle()
{
    // Check if all non-zero degree vertices are connected
    if (isSC() == false)
        return false;
}

```

```

// Check if in degree and out degree of every vertex is same
for (int i = 0; i < V; i++)
    if (adj[i].size() != in[i])
        return false;

return true;
}

// A recursive function to do DFS starting from v
void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

// Function that returns reverse (or transpose) of this graph
// This function is needed in isSC()
Graph Graph::getTranspose()
{
    Graph g(V);
    for (int v = 0; v < V; v++)
    {
        // Recur for all the vertices adjacent to this vertex
        list<int>::iterator i;
        for(i = adj[v].begin(); i != adj[v].end(); ++i)
        {
            g.adj[*i].push_back(v);
            (g.in[v])++;
        }
    }
    return g;
}

// This function returns true if all non-zero degree vertices of
// graph are strongly connected (Please refer
// https://www.geeksforgeeks.org/connectivity-in-a-directed-graph/ )
bool Graph::isSC()
{
    // Mark all the vertices as not visited (For first DFS)
    bool visited[V];
    for (int i = 0; i < V; i++)

```

```

visited[i] = false;

// Find the first vertex with non-zero degree
int n;
for (n = 0; n < V; n++)
    if (adj[n].size() > 0)
        break;

// Do DFS traversal starting from first non zero degree vertex.
DFSUtil(n, visited);

// If DFS traversal doesn't visit all vertices, then return false.
for (int i = 0; i < V; i++)
    if (adj[i].size() > 0 && visited[i] == false)
        return false;

// Create a reversed graph
Graph gr = getTranspose();

// Mark all the vertices as not visited (For second DFS)
for (int i = 0; i < V; i++)
    visited[i] = false;

// Do DFS for reversed graph starting from first vertex.
// Starting Vertex must be same starting point of first DFS
gr.DFSUtil(n, visited);

// If all vertices are not visited in second DFS, then
// return false
for (int i = 0; i < V; i++)
    if (adj[i].size() > 0 && visited[i] == false)
        return false;

return true;
}

// Driver program to test above functions
int main()
{
    // Create a graph given in the above diagram
    Graph g(5);
    g.addEdge(1, 0);
    g.addEdge(0, 2);
    g.addEdge(2, 1);
    g.addEdge(0, 3);
    g.addEdge(3, 4);
    g.addEdge(4, 0);
}

```

```
if (g.isEulerianCycle())
    cout << "Given directed graph is eulerian n";
else
    cout << "Given directed graph is NOT eulerian n";
return 0;
}
```

Java

```
// A Java program to check if a given directed graph is Eulerian or not

// A class that represents an undirected graph
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents a directed graph using adjacency list
class Graph
{
    private int V;    // No. of vertices
    private LinkedList<Integer> adj[];//Adjacency List
    private int in[];           //maintaining in degree

    //Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        in = new int[V];
        for (int i=0; i<v; ++i)
        {
            adj[i] = new LinkedList();
            in[i] = 0;
        }
    }

    //Function to add an edge into the graph
    void addEdge(int v,int w)
    {
        adj[v].add(w);
        in[w]++;
    }

    // A recursive function to print DFS starting from v
    void DFSUtil(int v,Boolean visited[])
    {
        // Mark the current node as visited
        visited[v] = true;
```

```

int n;

// Recur for all the vertices adjacent to this vertex
Iterator<Integer> i = adj[v].iterator();
while (i.hasNext())
{
    n = i.next();
    if (!visited[n])
        DFSUtil(n, visited);
}
}

// Function that returns reverse (or transpose) of this graph
Graph getTranspose()
{
    Graph g = new Graph(V);
    for (int v = 0; v < V; v++)
    {
        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> i = adj[v].listIterator();
        while (i.hasNext())
        {
            g.adj[i.next()].add(v);
            (g.in[v])++;
        }
    }
    return g;
}

// The main function that returns true if graph is strongly
// connected
Boolean isSC()
{
    // Step 1: Mark all the vertices as not visited (For
    // first DFS)
    Boolean visited[] = new Boolean[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Step 2: Do DFS traversal starting from first vertex.
    DFSUtil(0, visited);

    // If DFS traversal doesn't visit all vertices, then return false.
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            return false;
}

```

```

// Step 3: Create a reversed graph
Graph gr = getTranspose();

// Step 4: Mark all the vertices as not visited (For second DFS)
for (int i = 0; i < V; i++)
    visited[i] = false;

// Step 5: Do DFS for reversed graph starting from first vertex.
// Starting Vertex must be same starting point of first DFS
gr.DFSUtil(0, visited);

// If all vertices are not visited in second DFS, then
// return false
for (int i = 0; i < V; i++)
    if (visited[i] == false)
        return false;

return true;
}

/* This function returns true if the directed graph has an eulerian
cycle, otherwise returns false */
Boolean isEulerianCycle()
{
    // Check if all non-zero degree vertices are connected
    if (isSC() == false)
        return false;

    // Check if in degree and out degree of every vertex is same
    for (int i = 0; i < V; i++)
        if (adj[i].size() != in[i])
            return false;

    return true;
}

public static void main (String[] args) throws java.lang.Exception
{
    Graph g = new Graph(5);
    g.addEdge(1, 0);
    g.addEdge(0, 2);
    g.addEdge(2, 1);
    g.addEdge(0, 3);
    g.addEdge(3, 4);
    g.addEdge(4, 0);

    if (g.isEulerianCycle())
        System.out.println("Given directed graph is eulerian ");
}

```

```
        else
            System.out.println("Given directed graph is NOT eulerian ");
    }
}
//This code is contributed by Aakash Hasija
```

Python

```
# A Python program to check if a given
# directed graph is Eulerian or not

from collections import defaultdict

class Graph():

    def __init__(self, vertices):
        self.V = vertices
        self.graph = defaultdict(list)
        self.IN = [0] * vertices

    def addEdge(self, v, u):
        self.graph[v].append(u)
        self.IN[u] += 1

    def DFSUtil(self, v, visited):
        visited[v] = True
        for node in self.graph[v]:
            if visited[node] == False:
                self.DFSUtil(node, visited)

    def getTranspose(self):
        gr = Graph(self.V)

        for node in range(self.V):
            for child in self.graph[node]:
                gr.addEdge(child, node)

        return gr

    def isSC(self):
        visited = [False] * self.V

        v = 0
        for v in range(self.V):
            if len(self.graph[v]) > 0:
                break
```

```

        self.DFSUtil(v, visited)

        # If DFS traversal doesn't visit all
        # vertices, then return false.
        for i in range(self.V):
            if visited[i] == False:
                return False

        gr = self.getTranspose()

        visited = [False] * self.V
        gr.DFSUtil(v, visited)

        for i in range(self.V):
            if visited[i] == False:
                return False

        return True

    def isEulerianCycle(self):

        # Check if all non-zero degree vertices
        # are connected
        if self.isSC() == False:
            return False

        # Check if in degree and out degree of
        # every vertex is same
        for v in range(self.V):
            if len(self.graph[v]) != self.IN[v]:
                return False

        return True

g = Graph(5);
g.addEdge(1, 0);
g.addEdge(0, 2);
g.addEdge(2, 1);
g.addEdge(0, 3);
g.addEdge(3, 4);
g.addEdge(4, 0);
if g.isEulerianCycle():
    print "Given directed graph is eulerian";
else:
    print "Given directed graph is NOT eulerian";

# This code is contributed by Divyanshu Mehta

```

Output:

```
Given directed graph is eulerian
```

Time complexity of the above implementation is $O(V + E)$ as [Kosaraju's algorithm](#) takes $O(V + E)$ time. After running [Kosaraju's algorithm](#) we traverse all vertices and compare in degree with out degree which takes $O(V)$ time.

See following as an application of this.

[Find if the given array of strings can be chained to form a circle.](#)

Source

<https://www.geeksforgeeks.org/euler-circuit-directed-graph/>

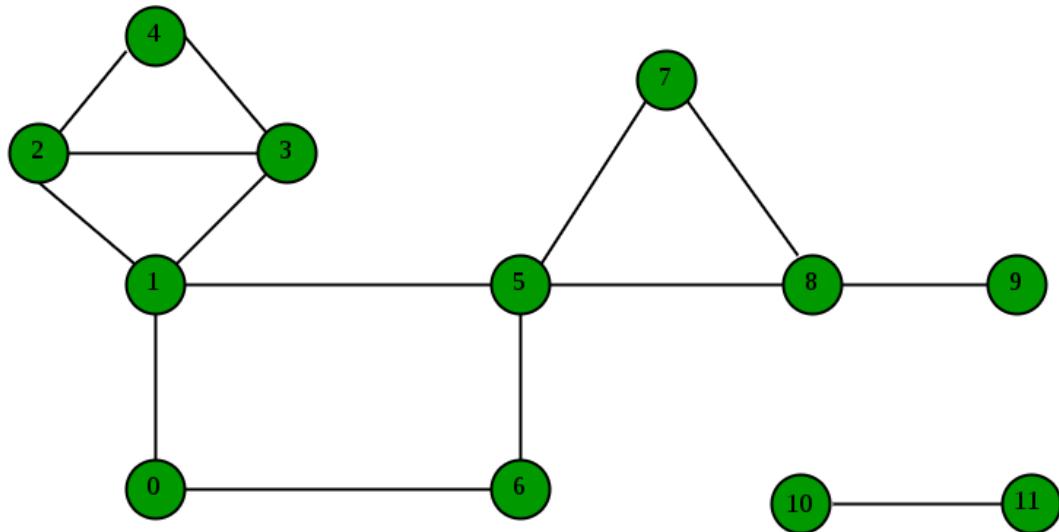
Chapter 54

Biconnected Components

Biconnected Components - GeeksforGeeks

A [biconnected component](#) is a maximal [biconnected subgraph](#).

[Biconnected Graph](#) is already discussed [here](#). In this article, we will see how to find [biconnected component](#) in a graph using algorithm by John Hopcroft and Robert Tarjan.



In above graph, following are the biconnected components:

- 4–2 3–4 3–1 2–3 1–2
- 8–9
- 8–5 7–8 5–7
- 6–0 5–6 1–5 0–1

- 10–11

Algorithm is based on Disc and Low Values discussed in [Strongly Connected Components Article](#).

Idea is to store visited edges in a stack while DFS on a graph and keep looking for [Articulation Points](#) (highlighted in above figure). As soon as an [Articulation Point](#) u is found, all edges visited while DFS from node u onwards will form one [biconnected component](#). When DFS completes for one [connected component](#), all edges present in stack will form a biconnected component.

If there is no [Articulation Point](#) in graph, then graph is biconnected and so there will be one biconnected component which is the graph itself.

C++

```
// A C++ program to find biconnected components in a given undirected graph
#include <iostream>
#include <list>
#include <stack>
#define NIL -1
using namespace std;
int count = 0;
class Edge {
public:
    int u;
    int v;
    Edge(int u, int v);
};
Edge::Edge(int u, int v)
{
    this->u = u;
    this->v = v;
}

// A class that represents an directed graph
class Graph {
    int V; // No. of vertices
    int E; // No. of edges
    list<int*>* adj; // A dynamic array of adjacency lists

    // A Recursive DFS based function used by BCC()
    void BCCUtil(int u, int disc[], int low[],
                 list<Edge*>* st, int parent[]);

public:
    Graph(int V); // Constructor
    void addEdge(int v, int w); // function to add an edge to graph
    void BCC(); // prints strongly connected components
};
```

```

Graph::Graph(int V)
{
    this->V = V;
    this->E = 0;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
    E++;
}

// A recursive function that finds and prints strongly connected
// components using DFS traversal
// u --> The vertex to be visited next
// disc[] --> Stores discovery times of visited vertices
// low[] --> earliest visited vertex (the vertex with minimum
// discovery time) that can be reached from subtree
// rooted with current vertex
// *st --> To store visited edges
void Graph::BCCUtil(int u, int disc[], int low[], list<Edge*>* st,
                     int parent[])
{
    // A static variable is used for simplicity, we can avoid use
    // of static variable by passing a pointer.
    static int time = 0;

    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;
    int children = 0;

    // Go through all vertices adjacent to this
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i) {
        int v = *i; // v is current adjacent of 'u'

        // If v is not visited yet, then recur for it
        if (disc[v] == -1) {
            children++;
            parent[v] = u;
            // store the edge in stack
            st->push_back(Edge(u, v));
            BCCUtil(v, disc, low, st, parent);

            // Check if the subtree rooted with 'v' has a
            // connection to one of the ancestors of 'u'
        }
    }
}

```

```

// Case 1 -- per Strongly Connected Components Article
low[u] = min(low[u], low[v]);

// If u is an articulation point,
// pop all edges from stack till u == v
if ((disc[u] == 1 && children > 1) || (disc[u] > 1 && low[v] >= disc[u])) {
    while (st->back().u != u || st->back().v != v) {
        cout << st->back().u << "--" << st->back().v << " ";
        st->pop_back();
    }
    cout << st->back().u << "--" << st->back().v;
    st->pop_back();
    cout << endl;
    count++;
}
}

// Update low value of 'u' only if 'v' is still in stack
// (i.e. it's a back edge, not cross edge).
// Case 2 -- per Strongly Connected Components Article
else if (v != parent[u]) {
    low[u] = min(low[u], disc[v]);
    if (disc[v] < disc[u]) {
        st->push_back(Edge(u, v));
    }
}
}

// The function to do DFS traversal. It uses BCCUtil()
void Graph::BCC()
{
    int* disc = new int[V];
    int* low = new int[V];
    int* parent = new int[V];
    list<Edge>* st = new list<Edge>[E];

    // Initialize disc and low, and parent arrays
    for (int i = 0; i < V; i++) {
        disc[i] = NIL;
        low[i] = NIL;
        parent[i] = NIL;
    }

    for (int i = 0; i < V; i++) {
        if (disc[i] == NIL)
            BCCUtil(i, disc, low, st, parent);
    }
}

```

```
int j = 0;
// If stack is not empty, pop all edges from stack
while (st->size() > 0) {
    j = 1;
    cout << st->back().u << "--" << st->back().v << " ";
    st->pop_back();
}
if (j == 1) {
    cout << endl;
    count++;
}
}

// Driver program to test above function
int main()
{
    Graph g(12);
    g.addEdge(0, 1);
    g.addEdge(1, 0);
    g.addEdge(1, 2);
    g.addEdge(2, 1);
    g.addEdge(1, 3);
    g.addEdge(3, 1);
    g.addEdge(2, 3);
    g.addEdge(3, 2);
    g.addEdge(2, 4);
    g.addEdge(4, 2);
    g.addEdge(3, 4);
    g.addEdge(4, 3);
    g.addEdge(1, 5);
    g.addEdge(5, 1);
    g.addEdge(0, 6);
    g.addEdge(6, 0);
    g.addEdge(5, 6);
    g.addEdge(6, 5);
    g.addEdge(5, 7);
    g.addEdge(7, 5);
    g.addEdge(5, 8);
    g.addEdge(8, 5);
    g.addEdge(7, 8);
    g.addEdge(8, 7);
    g.addEdge(8, 9);
    g.addEdge(9, 8);
    g.addEdge(10, 11);
    g.addEdge(11, 10);
    g.BCC();
    cout << "Above are " << count << " biconnected components in graph";
}
```

```
    return 0;
}
```

Java

```
// A Java program to find biconnected components in a given
// undirected graph
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency
// list representation
class Graph {
    private int V, E; // No. of vertices & Edges respectively
    private LinkedList<Integer> adj[]; // Adjacency List

    // Count is number of biconnected components. time is
    // used to find discovery times
    static int count = 0, time = 0;

    class Edge {
        int u;
        int v;
        Edge(int u, int v)
        {
            this.u = u;
            this.v = v;
        }
    };

    // Constructor
    Graph(int v)
    {
        V = v;
        E = 0;
        adj = new LinkedList[v];
        for (int i = 0; i < v; ++i)
            adj[i] = new LinkedList();
    }

    // Function to add an edge into the graph
    void addEdge(int v, int w)
    {
        adj[v].add(w);
        E++;
    }

    // A recursive function that finds and prints strongly connected
```

```

// components using DFS traversal
// u --> The vertex to be visited next
// disc[] --> Stores discovery times of visited vertices
// low[] --> earliest visited vertex (the vertex with minimum
// discovery time) that can be reached from subtree
// rooted with current vertex
// *st --> To store visited edges
void BCCUtil(int u, int disc[], int low[], LinkedList<Edge> st,
             int parent[])
{
    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;
    int children = 0;

    // Go through all vertices adjacent to this
    Iterator<Integer> it = adj[u].iterator();
    while (it.hasNext()) {
        int v = it.next(); // v is current adjacent of 'u'

        // If v is not visited yet, then recur for it
        if (disc[v] == -1) {
            children++;
            parent[v] = u;

            // store the edge in stack
            st.add(new Edge(u, v));
            BCCUtil(v, disc, low, st, parent);

            // Check if the subtree rooted with 'v' has a
            // connection to one of the ancestors of 'u'
            // Case 1 -- per Strongly Connected Components Article
            if (low[u] > low[v])
                low[u] = low[v];
        }

        // If u is an articulation point,
        // pop all edges from stack till u == v
        if ((disc[u] == 1 && children > 1) || (disc[u] > 1 && low[v] >= disc[u])) {
            while (st.getLast().u != u || st.getLast().v != v) {
                System.out.print(st.getLast().u + "--" + st.getLast().v + " ");
                st.removeLast();
            }
            System.out.println(st.getLast().u + "--" + st.getLast().v + " ");
            st.removeLast();

            count++;
        }
    }
}

```

```

// Update low value of 'u' only if 'v' is still in stack
// (i.e. it's a back edge, not cross edge).
// Case 2 -- per Strongly Connected Components Article
else if (v != parent[u] && disc[v] < low[u]) {
    if (low[u] > disc[v])
        low[u] = disc[v];
    st.add(new Edge(u, v));
}
}

// The function to do DFS traversal. It uses BCCUtil()
void BCC()
{
    int disc[] = new int[V];
    int low[] = new int[V];
    int parent[] = new int[V];
    LinkedList<Edge> st = new LinkedList<Edge>();

    // Initialize disc and low, and parent arrays
    for (int i = 0; i < V; i++) {
        disc[i] = -1;
        low[i] = -1;
        parent[i] = -1;
    }

    for (int i = 0; i < V; i++) {
        if (disc[i] == -1)
            BCCUtil(i, disc, low, st, parent);

        int j = 0;

        // If stack is not empty, pop all edges from stack
        while (st.size() > 0) {
            j = 1;
            System.out.print(st.getLast().u + "--" + st.getLast().v + " ");
            st.removeLast();
        }
        if (j == 1) {
            System.out.println();
            count++;
        }
    }
}

public static void main(String args[])
{

```

```
Graph g = new Graph(12);
g.addEdge(0, 1);
g.addEdge(1, 0);
g.addEdge(1, 2);
g.addEdge(2, 1);
g.addEdge(1, 3);
g.addEdge(3, 1);
g.addEdge(2, 3);
g.addEdge(3, 2);
g.addEdge(2, 4);
g.addEdge(4, 2);
g.addEdge(3, 4);
g.addEdge(4, 3);
g.addEdge(1, 5);
g.addEdge(5, 1);
g.addEdge(0, 6);
g.addEdge(6, 0);
g.addEdge(5, 6);
g.addEdge(6, 5);
g.addEdge(5, 7);
g.addEdge(7, 5);
g.addEdge(5, 8);
g.addEdge(8, 5);
g.addEdge(7, 8);
g.addEdge(8, 7);
g.addEdge(8, 9);
g.addEdge(9, 8);
g.addEdge(10, 11);
g.addEdge(11, 10);

g.BCC();

System.out.println("Above are " + g.count + " biconnected components in graph");
}

}

// This code is contributed by Aakash Hasija
```

Python

```
# Python program to find biconnected components in a given
# undirected graph
# Complexity : O(V + E)

from collections import defaultdict

# This class represents an directed graph
# using adjacency list representation
```

```

class Graph:

    def __init__(self, vertices):
        # No. of vertices
        self.V = vertices

        # default dictionary to store graph
        self.graph = defaultdict(list)

        # time is used to find discovery times
        self.Time = 0

        # Count is number of biconnected components
        self.count = 0

    # function to add an edge to graph
    def addEdge(self, u, v):
        self.graph[u].append(v)
        self.graph[v].append(u)

    '''A recursive function that finds and prints strongly connected
    components using DFS traversal
    u --> The vertex to be visited next
    disc[] --> Stores discovery times of visited vertices
    low[] --> earliest visited vertex (the vertex with minimum
               discovery time) that can be reached from subtree
               rooted with current vertex
    st --> To store visited edges'''
    def BCCUtil(self, u, parent, low, disc, st):

        # Count of children in current node
        children = 0

        # Initialize discovery time and low value
        disc[u] = self.Time
        low[u] = self.Time
        self.Time += 1

        # Recur for all the vertices adjacent to this vertex
        for v in self.graph[u]:
            # If v is not visited yet, then make it a child of u
            # in DFS tree and recur for it
            if disc[v] == -1 :
                parent[v] = u
                children += 1
                st.append((u, v)) # store the edge in stack
                self.BCCUtil(v, parent, low, disc, st)

```

```

# Check if the subtree rooted with v has a connection to
# one of the ancestors of u
# Case 1 -- per Strongly Connected Components Article
low[u] = min(low[u], low[v])

# If u is an articulation point, pop
# all edges from stack till (u, v)
if parent[u] == -1 and children > 1 or parent[u] != -1 and low[v] >= disc[u]:
    self.count += 1 # increment count
    w = -1
    while w != (u, v):
        w = st.pop()
        print w,
    print ""

elif v != parent[u] and low[u] > disc[v]:
    '''Update low value of 'u' only if 'v' is still in stack
    (i.e. it's a back edge, not cross edge).
    Case 2
    -- per Strongly Connected Components Article'''

    low[u] = min(low [u], disc[v])

    st.append((u, v))

# The function to do DFS traversal.
# It uses recursive BCCUtil()
def BCC(self):

    # Initialize disc and low, and parent arrays
    disc = [-1] * (self.V)
    low = [-1] * (self.V)
    parent = [-1] * (self.V)
    st = []

    # Call the recursive helper function to
    # find articulation points
    # in DFS tree rooted with vertex 'i'
    for i in range(self.V):
        if disc[i] == -1:
            self.BCCUtil(i, parent, low, disc, st)

    # If stack is not empty, pop all edges from stack
    if st:
        self.count = self.count + 1

```

```
while st:  
    w = st.pop()  
    print w,  
    print ""  
  
# Create a graph given in the above diagram  
  
g = Graph(12)  
g.addEdge(0, 1)  
g.addEdge(1, 2)  
g.addEdge(1, 3)  
g.addEdge(2, 3)  
g.addEdge(2, 4)  
g.addEdge(3, 4)  
g.addEdge(1, 5)  
g.addEdge(0, 6)  
g.addEdge(5, 6)  
g.addEdge(5, 7)  
g.addEdge(5, 8)  
g.addEdge(7, 8)  
g.addEdge(8, 9)  
g.addEdge(10, 11)  
  
g.BCC();  
print ("Above are % d biconnected components in graph" %(g.count));  
  
# This code is contributed by Neelam Yadav
```

Output:

```
4--2 3--4 3--1 2--3 1--2  
8--9  
8--5 7--8 5--7  
6--0 5--6 1--5 0--1  
10--11  
Above are 5 biconnected components in graph
```

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Improved By : [VismayMalondkar](#)

Source

<https://www.geeksforgeeks.org/biconnected-components/>

Chapter 55

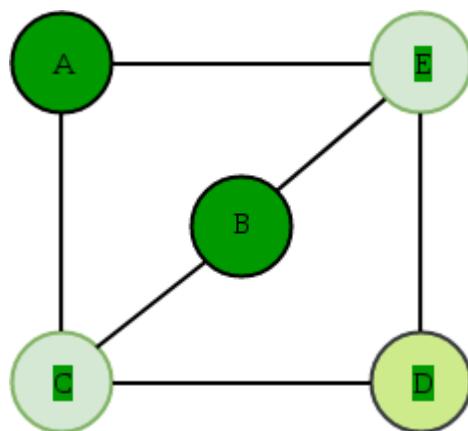
Graph Coloring (Introduction and Applications)

Graph Coloring Set 1 (Introduction and Applications) - GeeksforGeeks

[Graph coloring](#) problem is to assign colors to certain elements of a graph subject to certain constraints.

Vertex coloring is the most common graph coloring problem. The problem is, given m colors, find a way of coloring the vertices of a graph such that no two adjacent vertices are colored using same color. The other graph coloring problems like **Edge Coloring** (No vertex is incident to two edges of same color) and **Face Coloring** (Geographical Map Coloring) can be transformed into vertex coloring.

Chromatic Number: The smallest number of colors needed to color a graph G is called its chromatic number. For example, the following can be colored minimum 3 colors.



The problem to find chromatic number of a given graph is [NP Complete](#).

Applications of Graph Coloring:

The graph coloring problem has huge number of applications.

- 1) **Making Schedule or Time Table:** Suppose we want to make an exam schedule for a university. We have a list of different subjects and students enrolled in every subject. Many subjects would have common students (of same batch, some backlog students, etc). *How do we schedule the exam so that no two exams with a common student are scheduled at same time? How many minimum time slots are needed to schedule all exams?* This problem can be represented as a graph where every vertex is a subject and an edge between two vertices means there is a common student. So this is a graph coloring problem where minimum number of time slots is equal to the chromatic number of the graph.
- 2) **Mobile Radio Frequency Assignment:** When frequencies are assigned to towers, frequencies assigned to all towers at the same location must be different. How to assign frequencies with this constraint? What is the minimum number of frequencies needed? This problem is also an instance of graph coloring problem where every tower represents a vertex and an edge between two towers represents that they are in range of each other.
- 3) **Sudoku:** Sudoku is also a variation of Graph coloring problem where every cell represents a vertex. There is an edge between two vertices if they are in same row or same column or same block.
- 4) **Register Allocation:** In compiler optimization, register allocation is the process of assigning a large number of target program variables onto a small number of CPU registers. This problem is also a graph coloring problem.
- 5) **Bipartite Graphs:** We can check if a graph is Bipartite or not by coloring the graph using two colors. If a given graph is 2-colorable, then it is Bipartite, otherwise not. See [this](#) for more details.
- 6) **Map Coloring:** Geographical maps of countries or states where no two adjacent cities cannot be assigned same color. Four colors are sufficient to color any map (See [Four Color Theorem](#))

There can be many more applications: For example the below reference video lecture has a case study at 1:18.

Akamai runs a network of thousands of servers and the servers are used to distribute content on Internet. They install a new software or update existing softwares pretty much every week. The update cannot be deployed on every server at the same time, because the server may have to be taken down for the install. Also, the update should not be done one at a time, because it will take a lot of time. There are sets of servers that cannot be taken down together, because they have certain critical functions. This is a typical scheduling application of graph coloring problem. It turned out that 8 colors were good enough to color the graph of 75000 nodes. So they could install updates in 8 passes.

We will soon be discussing different ways to solve the graph coloring problem.

References:

[Lec 6 MIT 6.042J Mathematics for Computer Science, Fall 2010 Video Lecture](#)

Source

<https://www.geeksforgeeks.org/graph-coloring-applications/>

Chapter 56

Greedy Algorithm for Graph Coloring

Graph Coloring Set 2 (Greedy Algorithm) - GeeksforGeeks

We introduced [graph coloring and applications](#) in previous post. As discussed in the previous post, graph coloring is widely used. Unfortunately, there is no efficient algorithm available for coloring a graph with minimum number of colors as the problem is a known [NP Complete problem](#). There are approximate algorithms to solve the problem though. Following is the basic Greedy Algorithm to assign colors. It doesn't guarantee to use minimum colors, but it guarantees an upper bound on the number of colors. The basic algorithm never uses more than $d+1$ colors where d is the maximum degree of a vertex in the given graph.

Basic Greedy Coloring Algorithm:

1. Color first vertex with first color.
2. Do following for remaining $V-1$ vertices.
 - a) Consider the currently picked vertex and color it with the lowest numbered color that has not been used on any previously colored vertices adjacent to it. If all previously used colors appear on vertices adjacent to v , assign a new color to it.

Following are C++ and Java implementations of the above Greedy Algorithm.

C++

```
// A C++ program to implement greedy algorithm for graph coloring
#include <iostream>
#include <list>
using namespace std;

// A class that represents an undirected graph
class Graph
```

```

{
    int V;      // No. of vertices
    list<int> *adj;     // A dynamic array of adjacency lists
public:
    // Constructor and destructor
    Graph(int V)   { this->V = V; adj = new list<int>[V]; }
    ~Graph()        { delete [] adj; }

    // function to add an edge to graph
    void addEdge(int v, int w);

    // Prints greedy coloring of the vertices
    void greedyColoring();
};

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
    adj[w].push_back(v); // Note: the graph is undirected
}

// Assigns colors (starting from 0) to all vertices and prints
// the assignment of colors
void Graph::greedyColoring()
{
    int result[V];

    // Assign the first color to first vertex
    result[0] = 0;

    // Initialize remaining V-1 vertices as unassigned
    for (int u = 1; u < V; u++)
        result[u] = -1; // no color is assigned to u

    // A temporary array to store the available colors. True
    // value of available[cr] would mean that the color cr is
    // assigned to one of its adjacent vertices
    bool available[V];
    for (int cr = 0; cr < V; cr++)
        available[cr] = false;

    // Assign colors to remaining V-1 vertices
    for (int u = 1; u < V; u++)
    {
        // Process all adjacent vertices and flag their colors
        // as unavailable
        list<int>::iterator i;
        for (i = adj[u].begin(); i != adj[u].end(); ++i)

```

```

        if (result[*i] != -1)
            available[result[*i]] = true;

        // Find the first available color
        int cr;
        for (cr = 0; cr < V; cr++)
            if (available[cr] == false)
                break;

        result[u] = cr; // Assign the found color

        // Reset the values back to false for the next iteration
        for (i = adj[u].begin(); i != adj[u].end(); ++i)
            if (result[*i] != -1)
                available[result[*i]] = false;
    }

    // print the result
    for (int u = 0; u < V; u++)
        cout << "Vertex " << u << " ---> Color "
            << result[u] << endl;
}

// Driver program to test above function
int main()
{
    Graph g1(5);
    g1.addEdge(0, 1);
    g1.addEdge(0, 2);
    g1.addEdge(1, 2);
    g1.addEdge(1, 3);
    g1.addEdge(2, 3);
    g1.addEdge(3, 4);
    cout << "Coloring of graph 1 \n";
    g1.greedyColoring();

    Graph g2(5);
    g2.addEdge(0, 1);
    g2.addEdge(0, 2);
    g2.addEdge(1, 2);
    g2.addEdge(1, 4);
    g2.addEdge(2, 4);
    g2.addEdge(4, 3);
    cout << "\nColoring of graph 2 \n";
    g2.greedyColoring();

    return 0;
}

```

Java

```
// A Java program to implement greedy algorithm for graph coloring
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents an undirected graph using adjacency list
class Graph
{
    private int V;    // No. of vertices
    private LinkedList<Integer> adj[]; //Adjacency List

    //Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    //Function to add an edge into the graph
    void addEdge(int v,int w)
    {
        adj[v].add(w);
        adj[w].add(v); //Graph is undirected
    }

    // Assigns colors (starting from 0) to all vertices and
    // prints the assignment of colors
    void greedyColoring()
    {
        int result[] = new int[V];

        // Initialize all vertices as unassigned
        Arrays.fill(result, -1);

        // Assign the first color to first vertex
        result[0] = 0;

        // A temporary array to store the available colors. False
        // value of available[cr] would mean that the color cr is
        // assigned to one of its adjacent vertices
        boolean available[] = new boolean[V];

        // Initially, all colors are available
        Arrays.fill(available, true);
```

```

// Assign colors to remaining V-1 vertices
for (int u = 1; u < V; u++)
{
    // Process all adjacent vertices and flag their colors
    // as unavailable
    Iterator<Integer> it = adj[u].iterator() ;
    while (it.hasNext())
    {
        int i = it.next();
        if (result[i] != -1)
            available[result[i]] = false;
    }

    // Find the first available color
    int cr;
    for (cr = 0; cr < V; cr++){
        if (available[cr])
            break;
    }

    result[u] = cr; // Assign the found color

    // Reset the values back to true for the next iteration
    Arrays.fill(available, true);
}

// print the result
for (int u = 0; u < V; u++)
    System.out.println("Vertex " + u + " ---> Color "
                      + result[u]);
}

// Driver method
public static void main(String args[])
{
    Graph g1 = new Graph(5);
    g1.addEdge(0, 1);
    g1.addEdge(0, 2);
    g1.addEdge(1, 2);
    g1.addEdge(1, 3);
    g1.addEdge(2, 3);
    g1.addEdge(3, 4);
    System.out.println("Coloring of graph 1");
    g1.greedyColoring();

    System.out.println();
    Graph g2 = new Graph(5);
}

```

```
g2.addEdge(0, 1);
g2.addEdge(0, 2);
g2.addEdge(1, 2);
g2.addEdge(1, 4);
g2.addEdge(2, 4);
g2.addEdge(4, 3);
System.out.println("Coloring of graph 2 ");
g2.greedyColoring();
}
}

// This code is contributed by Aakash Hasija
```

Output:

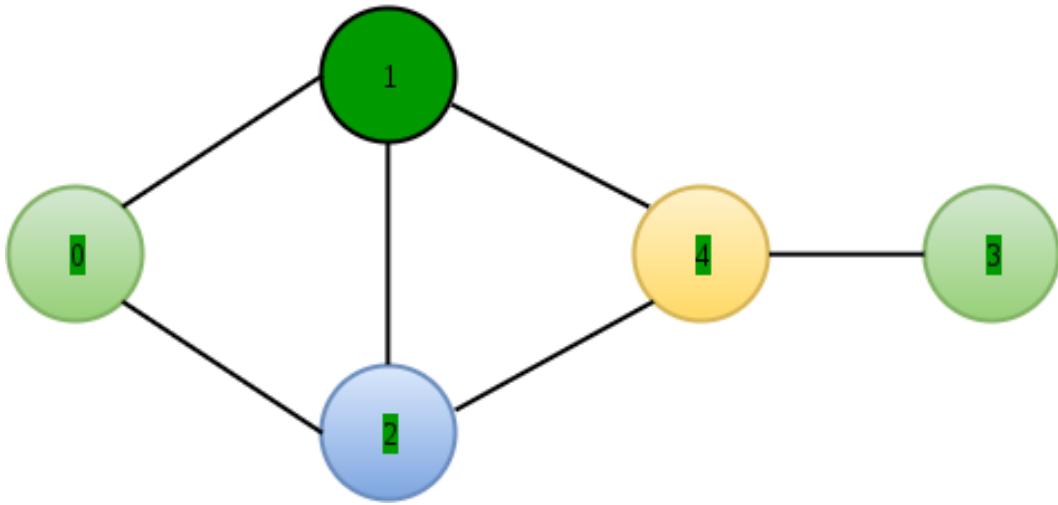
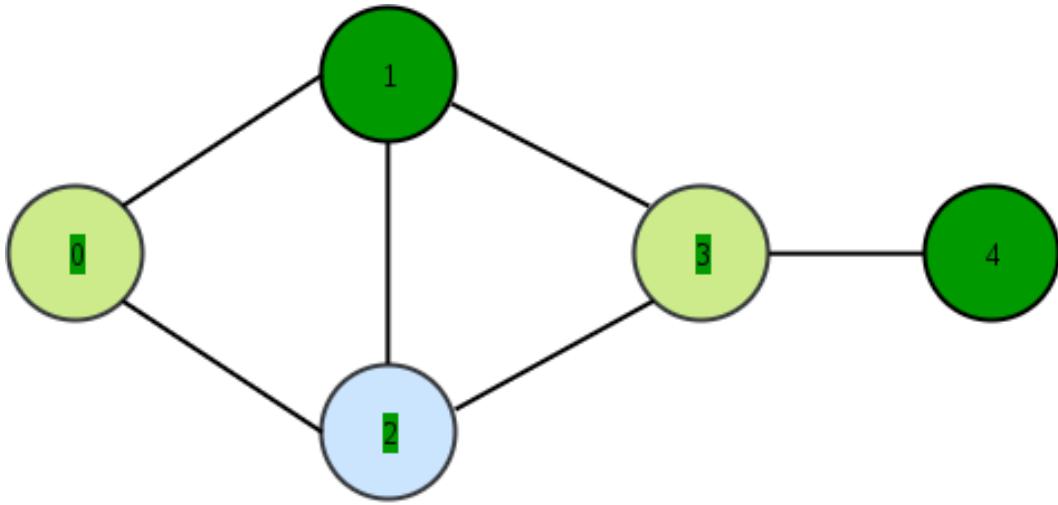
```
Coloring of graph 1
Vertex 0 ----> Color 0
Vertex 1 ----> Color 1
Vertex 2 ----> Color 2
Vertex 3 ----> Color 0
Vertex 4 ----> Color 1
```

```
Coloring of graph 2
Vertex 0 ----> Color 0
Vertex 1 ----> Color 1
Vertex 2 ----> Color 2
Vertex 3 ----> Color 0
Vertex 4 ----> Color 3
```

Time Complexity: $O(V^2 + E)$ in worst case.

Analysis of Basic Algorithm

The above algorithm doesn't always use minimum number of colors. Also, the number of colors used sometime depend on the order in which vertices are processed. For example, consider the following two graphs. Note that in graph on right side, vertices 3 and 4 are swapped. If we consider the vertices 0, 1, 2, 3, 4 in left graph, we can color the graph using 3 colors. But if we consider the vertices 0, 1, 2, 3, 4 in right graph, we need 4 colors.



So the order in which the vertices are picked is important. Many people have suggested different ways to find an ordering that work better than the basic algorithm on average. The most common is [Welsh–Powell Algorithm](#) which considers vertices in descending order of degrees.

How does the basic algorithm guarantee an upper bound of $d+1$?

Here d is the maximum degree in the given graph. Since d is maximum degree, a vertex cannot be attached to more than d vertices. When we color a vertex, at most d colors could have already been used by its adjacent. To color this vertex, we need to pick the smallest numbered color that is not used by the adjacent vertices. If colors are numbered like 1, 2, ..., then the value of such smallest number must be between 1 to $d+1$ (Note that d numbers are already picked by adjacent vertices).

This can also be proved using induction. See [this](#)video lecture for proof.

We will soon be discussing some interesting facts about chromatic number and graph coloring.

Improved By : [ChamanJhinga](#)

Source

<https://www.geeksforgeeks.org/graph-coloring-set-2-greedy-algorithm/>

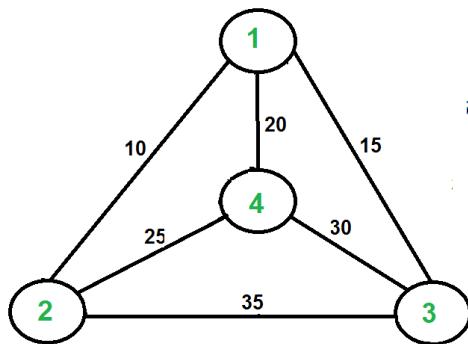
Chapter 57

Travelling Salesman Problem (Naive and Dynamic Programming)

Travelling Salesman Problem Set 1 (Naive and Dynamic Programming) - GeeksforGeeks

Travelling Salesman Problem (TSP): Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

Note the difference between [Hamiltonian Cycle](#) and TSP. The Hamiltonian cycle problem is to find if there exist a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.



For example, consider the graph shown in figure on right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is $10+25+30+15$ which is 80.

The problem is a famous [NP hard](#) problem. There is no polynomial time known solution for this problem.

Following are different solutions for the traveling salesman problem.

Naive Solution:

- 1) Consider city 1 as the starting and ending point.
- 2) Generate all $(n-1)!$ Permutations of cities.
- 3) Calculate cost of every permutation and keep track of minimum cost permutation.
- 4) Return the permutation with minimum cost.

Time Complexity: $\Theta(n!)$

Dynamic Programming:

Let the given set of vertices be $\{1, 2, 3, 4, \dots, n\}$. Let us consider 1 as starting and ending point of output. For every other vertex i (other than 1), we find the minimum cost path with 1 as the starting point, i as the ending point and all vertices appearing exactly once. Let the cost of this path be $\text{cost}(i)$, the cost of corresponding Cycle would be $\text{cost}(i) + \text{dist}(i, 1)$ where $\text{dist}(i, 1)$ is the distance from i to 1. Finally, we return the minimum of all $[\text{cost}(i) + \text{dist}(i, 1)]$ values. This looks simple so far. Now the question is how to get $\text{cost}(i)$?

To calculate $\text{cost}(i)$ using Dynamic Programming, we need to have some recursive relation in terms of sub-problems. Let us define a term $C(S, i)$ be the cost of the minimum cost path visiting each vertex in set S exactly once, starting at 1 and ending at i .

We start with all subsets of size 2 and calculate $C(S, i)$ for all subsets where S is the subset, then we calculate $C(S, i)$ for all subsets S of size 3 and so on. Note that 1 must be present in every subset.

```
If size of S is 2, then S must be {1, i},  
C(S, i) = dist(1, i)  
Else if size of S is greater than 2.  
C(S, i) = min { C(S-{i}, j) + dis(j, i)} where j belongs to S, j != i and j != 1.
```

For a set of size n , we consider $n-2$ subsets each of size $n-1$ such that all subsets don't have nth in them.

Using the above recurrence relation, we can write dynamic programming based solution. There are at most $O(n^*2^n)$ subproblems, and each one takes linear time to solve. The total running time is therefore $O(n^2*2^n)$. The time complexity is much less than $O(n!)$, but still exponential. Space required is also exponential. So this approach is also infeasible even for slightly higher number of vertices.

We will soon be discussing approximate algorithms for travelling salesman problem.

Next Article: [Traveling Salesman Problem Set 2](#)

References:

<http://www.lsi.upc.edu/~mjserna/docencia/algofib/P07/dynprog.pdf>
<http://www.cs.berkeley.edu/~vazirani/algorithms/chap6.pdf>

Source

<https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/>

Chapter 58

Travelling Salesman Problem (Approximate using MST)

Travelling Salesman Problem Set 2 (Approximate using MST) - GeeksforGeeks

We introduced [Travelling Salesman Problem](#) and discussed Naive and Dynamic Programming Solutions for the problem in the [previous post](#). Both of the solutions are infeasible. In fact, there is no polynomial time solution available for this problem as the problem is a known NP-Hard problem. There are approximate algorithms to solve the problem though. The approximate algorithms work only if the problem instance satisfies Triangle-Inequality.

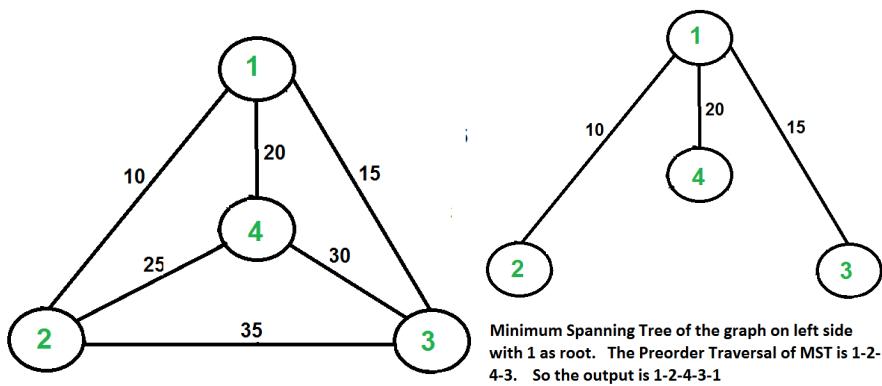
Triangle-Inequality: The least distant path to reach a vertex j from i is always to reach j directly from i , rather than through some other vertex k (or vertices), i.e., $\text{dis}(i, j)$ is always less than or equal to $\text{dis}(i, k) + \text{dist}(k, j)$. The Triangle-Inequality holds in many practical situations.

When the cost function satisfies the triangle inequality, we can design an approximate algorithm for TSP that returns a tour whose cost is never more than twice the cost of an optimal tour. The idea is to use [Minimum Spanning Tree \(MST\)](#). Following is the MST based algorithm.

Algorithm:

- 1) Let 1 be the starting and ending point for salesman.
- 2) Construct MST from with 1 as root using [Prim's Algorithm](#).
- 3) List vertices visited in preorder walk of the constructed MST and add 1 at the end.

Let us consider the following example. The first diagram is the given graph. The second diagram shows MST constructed with 1 as root. The preorder traversal of MST is 1-2-4-3. Adding 1 at the end gives 1-2-4-3-1 which is the output of this algorithm.



In this case, the approximate algorithm produces the optimal tour, but it may not produce optimal tour in all cases.

How is this algorithm 2-approximate? The cost of the output produced by the above algorithm is never more than twice the cost of best possible output. Let us see how is this guaranteed by the above algorithm.

Let us define a term **full walk** to understand this. A full walk is lists all vertices when they are first visited in preorder, it also list vertices when they are returned after a subtree is visited in preorder. The full walk of above tree would be 1-2-1-4-1-3-1.

Following are some important facts that prove the 2-approximateneess.

- 1) The cost of best possible Travelling Salesman tour is never less than the cost of MST. (The definition of [MST](#)says, it is a minimum cost tree that connects all vertices).
- 2) The total cost of full walk is at most twice the cost of MST (Every edge of MST is visited at-most twice)
- 3) The output of the above algorithm is less than the cost of full walk. In above algorithm, we print preorder walk as output. In preorder walk, two or more edges of full walk are replaced with a single edge. For example, 2-1 and 1-4 are replaced by 1 edge 2-4. So if the graph follows triangle inequality, then this is always true.

From the above three statements, we can conclude that the cost of output produced by the approximate algorithm is never more than twice the cost of best possible solution.

We have discussed a very simple 2-approximate algorithm for the travelling salesman problem. There are other better approximate algorithms for the problem. For example [Christofides algorithm](#) is 1.5 approximate algorithm. We will soon be discussing these algorithms as separate posts.

References:

- [Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest](#)
<http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/AproxAlgor/TSP/tsp.htm>

Improved By : [PranamLashkari](#)

Source

<https://www.geeksforgeeks.org/travelling-salesman-problem-set-2-approximate-using-mst/>

Chapter 59

Hamiltonian Cycle

Hamiltonian Cycle Backtracking-6 - GeeksforGeeks

[Hamiltonian Path](#) in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in graph) from the last vertex to the first vertex of the Hamiltonian Path. Determine whether a given graph contains Hamiltonian Cycle or not. If it contains, then print the path. Following are the input and output of the required function.

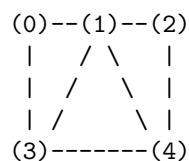
Input:

A 2D array $\text{graph}[V][V]$ where V is the number of vertices in graph and $\text{graph}[V][V]$ is adjacency matrix representation of the graph. A value $\text{graph}[i][j]$ is 1 if there is a direct edge from i to j , otherwise $\text{graph}[i][j]$ is 0.

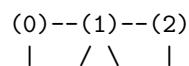
Output:

An array $\text{path}[V]$ that should contain the Hamiltonian Path. $\text{path}[i]$ should represent the i th vertex in the Hamiltonian Path. The code should also return false if there is no Hamiltonian Cycle in the graph.

For example, a Hamiltonian Cycle in the following graph is $\{0, 1, 2, 4, 3, 0\}$. There are more Hamiltonian Cycles in the graph like $\{0, 3, 4, 2, 1, 0\}$



And the following graph doesn't contain any Hamiltonian Cycle.



```

| / \ |
| / \ |
(3)   (4)

```

Naive Algorithm

Generate all possible configurations of vertices and print a configuration that satisfies the given constraints. There will be $n!$ (n factorial) configurations.

```

while there are untried conflagrations
{
    generate the next configuration
    if ( there are edges between two consecutive vertices of this
        configuration and there is an edge from the last vertex to
        the first ).

    {
        print this configuration;
        break;
    }
}

```

Backtracking Algorithm

Create an empty path array and add vertex 0 to it. Add other vertices, starting from the vertex 1. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added. If we find such a vertex, we add the vertex as part of the solution. If we do not find a vertex then we return false.

Implementation of Backtracking solution

Following are implementations of the Backtracking solution.

C/C++

```

/* C/C++ program for solution of Hamiltonian Cycle problem
   using backtracking */
#include<stdio.h>

// Number of vertices in the graph
#define V 5

void printSolution(int path[]);

/* A utility function to check if the vertex v can be added at
   index 'pos' in the Hamiltonian Cycle constructed so far (stored
   in 'path[]') */
bool isSafe(int v, bool graph[V][V], int path[], int pos)
{
    /* Check if this vertex is an adjacent vertex of the previously
       added vertex. */

```

```

if (graph [ path[pos-1] ][ v ] == 0)
    return false;

/* Check if the vertex has already been included.
   This step can be optimized by creating an array of size V */
for (int i = 0; i < pos; i++)
    if (path[i] == v)
        return false;

return true;
}

/* A recursive utility function to solve hamiltonian cycle problem */
bool hamCycleUtil(bool graph[V][V], int path[], int pos)
{
    /* base case: If all vertices are included in Hamiltonian Cycle */
    if (pos == V)
    {
        // And if there is an edge from the last included vertex to the
        // first vertex
        if (graph[ path[pos-1] ][ path[0] ] == 1 )
            return true;
        else
            return false;
    }

    // Try different vertices as a next candidate in Hamiltonian Cycle.
    // We don't try for 0 as we included 0 as starting point in in hamCycle()
    for (int v = 1; v < V; v++)
    {
        /* Check if this vertex can be added to Hamiltonian Cycle */
        if (isSafe(v, graph, path, pos))
        {
            path[pos] = v;

            /* recur to construct rest of the path */
            if (hamCycleUtil (graph, path, pos+1) == true)
                return true;

            /* If adding vertex v doesn't lead to a solution,
               then remove it */
            path[pos] = -1;
        }
    }

    /* If no vertex can be added to Hamiltonian Cycle constructed so far,
       then return false */
    return false;
}

```

```

}

/* This function solves the Hamiltonian Cycle problem using Backtracking.
   It mainly uses hamCycleUtil() to solve the problem. It returns false
   if there is no Hamiltonian Cycle possible, otherwise return true and
   prints the path. Please note that there may be more than one solutions,
   this function prints one of the feasible solutions. */
bool hamCycle(bool graph[V][V])
{
    int *path = new int[V];
    for (int i = 0; i < V; i++)
        path[i] = -1;

    /* Let us put vertex 0 as the first vertex in the path. If there is
       a Hamiltonian Cycle, then the path can be started from any point
       of the cycle as the graph is undirected */
    path[0] = 0;
    if (hamCycleUtil(graph, path, 1) == false)
    {
        printf("\nSolution does not exist");
        return false;
    }

    printSolution(path);
    return true;
}

/* A utility function to print solution */
void printSolution(int path[])
{
    printf ("Solution Exists:\n"
           " Following is one Hamiltonian Cycle \n");
    for (int i = 0; i < V; i++)
        printf(" %d ", path[i]);

    // Let us print the first vertex again to show the complete cycle
    printf(" %d ", path[0]);
    printf("\n");
}

// driver program to test above function
int main()
{
    /* Let us create the following graph
       (0)--(1)--(2)
       |    / \   |
       |   /     \  |
       |  /       \ |
    */
}

```

```

(3)----- (4)      */
bool graph1[V][V] = {{0, 1, 0, 1, 0},
                      {1, 0, 1, 1, 1},
                      {0, 1, 0, 0, 1},
                      {1, 1, 0, 0, 1},
                      {0, 1, 1, 1, 0},
                    };

// Print the solution
hamCycle(graph1);

/* Let us create the following graph
(0)--(1)--(2)
    |   / \   |
    |   /     \  |
    | /         \ |
(3)       (4)      */
bool graph2[V][V] = {{0, 1, 0, 1, 0},
                      {1, 0, 1, 1, 1},
                      {0, 1, 0, 0, 1},
                      {1, 1, 0, 0, 0},
                      {0, 1, 1, 0, 0},
                    };

// Print the solution
hamCycle(graph2);

return 0;
}

```

Java

```

/* Java program for solution of Hamiltonian Cycle problem
   using backtracking */
class HamiltonianCycle
{
    final int V = 5;
    int path[];

    /* A utility function to check if the vertex v can be
       added at index 'pos' in the Hamiltonian Cycle
       constructed so far (stored in 'path[]') */
    boolean isSafe(int v, int graph[][][], int path[], int pos)
    {
        /* Check if this vertex is an adjacent vertex of
           the previously added vertex. */
        if (graph[path[pos - 1]][v] == 0)
            return false;
    }
}

```

```

/* Check if the vertex has already been included.
   This step can be optimized by creating an array
   of size V */
for (int i = 0; i < pos; i++)
    if (path[i] == v)
        return false;

    return true;
}

/* A recursive utility function to solve hamiltonian
   cycle problem */
boolean hamCycleUtil(int graph[][] , int path[], int pos)
{
    /* base case: If all vertices are included in
       Hamiltonian Cycle */
    if (pos == V)
    {
        // And if there is an edge from the last included
        // vertex to the first vertex
        if (graph[path[pos - 1]][path[0]] == 1)
            return true;
        else
            return false;
    }

    // Try different vertices as a next candidate in
    // Hamiltonian Cycle. We don't try for 0 as we
    // included 0 as starting point in in hamCycle()
    for (int v = 1; v < V; v++)
    {
        /* Check if this vertex can be added to Hamiltonian
           Cycle */
        if (isSafe(v, graph, path, pos))
        {
            path[pos] = v;

            /* recur to construct rest of the path */
            if (hamCycleUtil(graph, path, pos + 1) == true)
                return true;

            /* If adding vertex v doesn't lead to a solution,
               then remove it */
            path[pos] = -1;
        }
    }
}

```

```

    /* If no vertex can be added to Hamiltonian Cycle
       constructed so far, then return false */
    return false;
}

/* This function solves the Hamiltonian Cycle problem using
   Backtracking. It mainly uses hamCycleUtil() to solve the
   problem. It returns false if there is no Hamiltonian Cycle
   possible, otherwise return true and prints the path.
   Please note that there may be more than one solutions,
   this function prints one of the feasible solutions. */
int hamCycle(int graph[][])
{
    path = new int[V];
    for (int i = 0; i < V; i++)
        path[i] = -1;

    /* Let us put vertex 0 as the first vertex in the path.
       If there is a Hamiltonian Cycle, then the path can be
       started from any point of the cycle as the graph is
       undirected */
    path[0] = 0;
    if (hamCycleUtil(graph, path, 1) == false)
    {
        System.out.println("\nSolution does not exist");
        return 0;
    }

    printSolution(path);
    return 1;
}

/* A utility function to print solution */
void printSolution(int path[])
{
    System.out.println("Solution Exists: Following" +
                       " is one Hamiltonian Cycle");
    for (int i = 0; i < V; i++)
        System.out.print(" " + path[i] + " ");

    // Let us print the first vertex again to show the
    // complete cycle
    System.out.println(" " + path[0] + " ");
}

// driver program to test above function
public static void main(String args[])
{

```

```

HamiltonianCycle hamiltonian =
    new HamiltonianCycle();
/* Let us create the following graph
(0)--(1)--(2)
|   / \   |
|   /     \ |
| /         \|
(3)-----(4) */
int graph1[][] = {{0, 1, 0, 1, 0},
                  {1, 0, 1, 1, 1},
                  {0, 1, 0, 0, 1},
                  {1, 1, 0, 0, 1},
                  {0, 1, 1, 1, 0},
                };
// Print the solution
hamiltonian.hamCycle(graph1);

/* Let us create the following graph
(0)--(1)--(2)
|   / \   |
|   /     \ |
| /         \|
(3)-----(4) */
int graph2[][] = {{0, 1, 0, 1, 0},
                  {1, 0, 1, 1, 1},
                  {0, 1, 0, 0, 1},
                  {1, 1, 0, 0, 0},
                  {0, 1, 1, 0, 0},
                };
// Print the solution
hamiltonian.hamCycle(graph2);
}
}

// This code is contributed by Abhishek Shankhadhar

```

Python

```

# Python program for solution of
# hamiltonian cycle problem

class Graph():
    def __init__(self, vertices):
        self.graph = [[0 for column in range(vertices)]\n                     for row in range(vertices)]
        self.V = vertices

```

```

''' Check if this vertex is an adjacent vertex
   of the previously added vertex and is not
   included in the path earlier '''
def isSafe(self, v, pos, path):
    # Check if current vertex and last vertex
    # in path are adjacent
    if self.graph[ path[pos-1] ][v] == 0:
        return False

    # Check if current vertex not already in path
    for vertex in path:
        if vertex == v:
            return False

    return True

# A recursive utility function to solve
# hamiltonian cycle problem
def hamCycleUtil(self, path, pos):

    # base case: if all vertices are
    # included in the path
    if pos == self.V:
        # Last vertex must be adjacent to the
        # first vertex in path to make a cycle
        if self.graph[ path[pos-1] ][ path[0] ] == 1:
            return True
        else:
            return False

    # Try different vertices as a next candidate
    # in Hamiltonian Cycle. We don't try for 0 as
    # we included 0 as starting point in in hamCycle()
    for v in range(1,self.V):

        if self.isSafe(v, pos, path) == True:

            path[pos] = v

            if self.hamCycleUtil(path, pos+1) == True:
                return True

            # Remove current vertex if it doesn't
            # lead to a solution
            path[pos] = -1

    return False

```

```

def hamCycle(self):
    path = [-1] * self.V

    ''' Let us put vertex 0 as the first vertex
       in the path. If there is a Hamiltonian Cycle,
       then the path can be started from any point
       of the cycle as the graph is undirected '''
    path[0] = 0

    if self.hamCycleUtil(path,1) == False:
        print "Solution does not exist\n"
        return False

    self.printSolution(path)
    return True

def printSolution(self, path):
    print "Solution Exists: Following is one Hamiltonian Cycle"
    for vertex in path:
        print vertex,
    print path[0], "\n"

# Driver Code

''' Let us create the following graph
    (0)--(1)--(2)
    |   / \   |
    |   /     \   |
    | /         \   |
    (3)-----(4)      ...
g1 = Graph(5)
g1.graph = [ [0, 1, 0, 1, 0], [1, 0, 1, 1, 1],
            [0, 1, 0, 0, 1],[1, 1, 0, 0, 1],
            [0, 1, 1, 1, 0], ]

# Print the solution
g1.hamCycle();

''' Let us create the following graph
    (0)--(1)--(2)
    |   / \   |
    |   /     \   |
    | /         \   |
    (3)          (4)      ...
g2 = Graph(5)
g2.graph = [ [0, 1, 0, 1, 0], [1, 0, 1, 1, 1],
            [0, 1, 0, 0, 1],[1, 1, 0, 0, 0],
            [0, 1, 1, 0, 0], ]

```

```
# Print the solution
g2.hamCycle();

# This code is contributed by Divyanshu Mehta
```

Output:

```
Solution Exists: Following is one Hamiltonian Cycle
0 1 2 4 3 0
```

```
Solution does not exist
```

Note that the above code always prints cycle starting from 0. Starting point should not matter as cycle can be started from any point. If you want to change the starting point, you should make two changes to above code.

Change “path[0] = 0;” to “path[0] = s;” where s is your new starting point. Also change loop “for (int v = 1; v < V; v++)” in hamCycleUtil() to ”for (int v = 0; v < V; v++)”. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

<https://www.geeksforgeeks.org/hamiltonian-cycle-backtracking-6/>

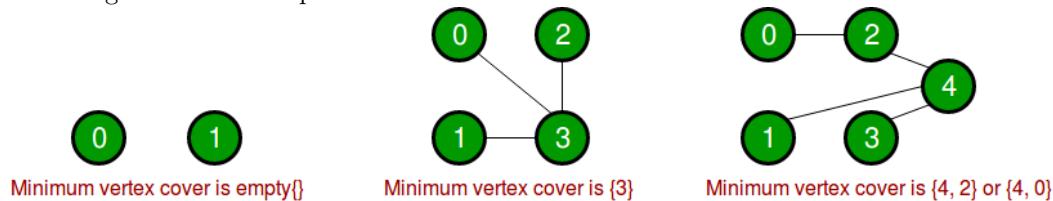
Chapter 60

Vertex Cover Problem Set 1 (Introduction and Approximate Algorithm)

Vertex Cover Problem Set 1 (Introduction and Approximate Algorithm) - GeeksforGeeks

A vertex cover of an undirected graph is a subset of its vertices such that for every edge (u, v) of the graph, either ' u ' or ' v ' is in vertex cover. Although the name is Vertex Cover, the set covers all edges of the given graph. *Given an undirected graph, the vertex cover problem is to find minimum size vertex cover.*

Following are some examples.



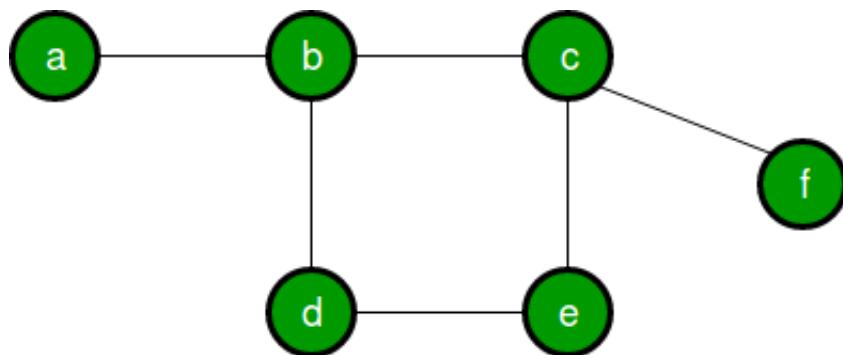
Vertex Cover Problem is a known [NP Complete problem](#), i.e., there is no polynomial time solution for this unless $P = NP$. There are approximate polynomial time algorithms to solve the problem though. Following is a simple approximate algorithm adapted from [CLRS book](#).

Approximate Algorithm for Vertex Cover:

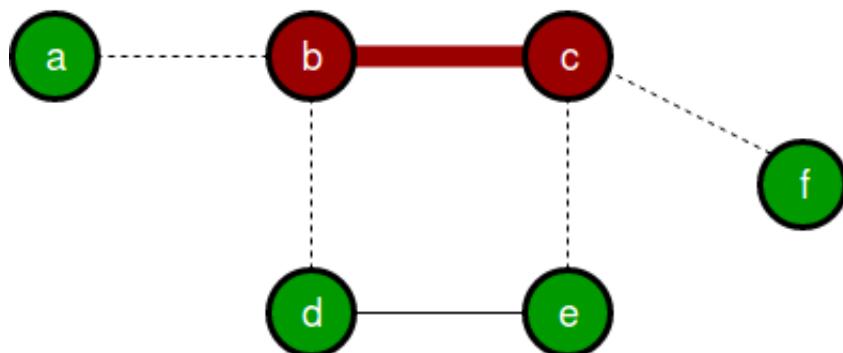
- 1) Initialize the result as {}
- 2) Consider a set of all edges in given graph. Let the set be E.
- 3) Do following while E is not empty
 - ...a) Pick an arbitrary edge (u, v) from set E and add ' u ' and ' v ' to result
 - ...b) Remove all edges from E which are either incident on u or v .

4) **Return result**

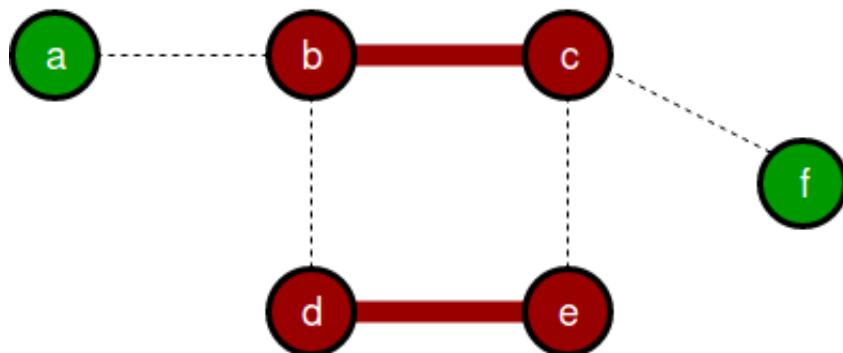
Below diagram to show execution of above approximate algorithm:



(a)



(b)



(c)

Minimum Vertex Cover is {b, c, d} or {b, c, e}

How well the above algorithm perform?

It can be proved that the above approximate algorithm never finds a vertex cover whose size is more than twice the size of minimum possible vertex cover (Refer [this](#)for proof)

Implementation:

Following are C++ and Java implementations of above approximate algorithm.

C++

```
// Program to print Vertex Cover of a given undirected graph
#include<iostream>
#include <list>
using namespace std;

// This class represents a undirected graph using adjacency list
class Graph
{
    int V;      // No. of vertices
    list<int> *adj; // Pointer to an array containing adjacency lists
public:
    Graph(int V); // Constructor
    void addEdge(int v, int w); // function to add an edge to graph
    void printVertexCover(); // prints vertex cover
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
    adj[w].push_back(v); // Since the graph is undirected
}

// The function to print vertex cover
void Graph::printVertexCover()
{
    // Initialize all vertices as not visited.
    bool visited[V];
    for (int i=0; i<V; i++)
        visited[i] = false;

    list<int>::iterator i;

    // Consider all edges one by one
    for (int u=0; u<V; u++)
```

```
{  
    // An edge is only picked when both visited[u] and visited[v]  
    // are false  
    if (visited[u] == false)  
    {  
        // Go through all adjacents of u and pick the first not  
        // yet visited vertex (We are basically picking an edge  
        // (u, v) from remaining edges.  
        for (i= adj[u].begin(); i != adj[u].end(); ++i)  
        {  
            int v = *i;  
            if (visited[v] == false)  
            {  
                // Add the vertices (u, v) to the result set.  
                // We make the vertex u and v visited so that  
                // all edges from/to them would be ignored  
                visited[v] = true;  
                visited[u] = true;  
                break;  
            }  
        }  
    }  
  
    // Print the vertex cover  
    for (int i=0; i<V; i++)  
        if (visited[i])  
            cout << i << " ";  
}  
  
// Driver program to test methods of graph class  
int main()  
{  
    // Create a graph given in the above diagram  
    Graph g(7);  
    g.addEdge(0, 1);  
    g.addEdge(0, 2);  
    g.addEdge(1, 3);  
    g.addEdge(3, 4);  
    g.addEdge(4, 5);  
    g.addEdge(5, 6);  
  
    g.printVertexCover();  
  
    return 0;  
}
```

Java

```
// Java Program to print Vertex Cover of a given undirected graph
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents an undirected graph using adjacency list
class Graph
{
    private int V;    // No. of vertices

    // Array of lists for Adjacency List Representation
    private LinkedList<Integer> adj[];

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    //Function to add an edge into the graph
    void addEdge(int v, int w)
    {
        adj[v].add(w);  // Add w to v's list.
        adj[w].add(v); //Graph is undirected
    }

    // The function to print vertex cover
    void printVertexCover()
    {
        // Initialize all vertices as not visited.
        boolean visited[] = new boolean[V];
        for (int i=0; i<V; i++)
            visited[i] = false;

        Iterator<Integer> i;

        // Consider all edges one by one
        for (int u=0; u<V; u++)
        {
            // An edge is only picked when both visited[u]
            // and visited[v] are false
            if (visited[u] == false)
            {
                // Go through all adjacents of u and pick the
                // first not yet visited vertex (We are basically
```

```

        // picking an edge (u, v) from remaining edges.
        i = adj[u].iterator();
        while (i.hasNext())
        {
            int v = i.next();
            if (visited[v] == false)
            {
                // Add the vertices (u, v) to the result
                // set. We make the vertex u and v visited
                // so that all edges from/to them would
                // be ignored
                visited[v] = true;
                visited[u] = true;
                break;
            }
        }
    }

    // Print the vertex cover
    for (int j=0; j<V; j++)
        if (visited[j])
            System.out.print(j+" ");
    }

    // Driver method
    public static void main(String args[])
    {
        // Create a graph given in the above diagram
        Graph g = new Graph(7);
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 3);
        g.addEdge(3, 4);
        g.addEdge(4, 5);
        g.addEdge(5, 6);

        g.printVertexCover();
    }
}
// This code is contributed by Aakash Hasija

```

Output:

0 1 3 4 5 6

Time Complexity of above algorithm is $O(V + E)$.

Exact Algorithms:

Although the problem is NP complete, it can be solved in polynomial time for following types of graphs.

- 1) Bipartite Graph
- 2) Tree Graph

The problem to check whether there is a vertex cover of size smaller than or equal to a given number k can also be solved in polynomial time if k is bounded by $O(\log V)$ (Refer [this](#))

We will soon be discussing exact algorithms for vertex cover.

This article is contributed by **Shubham**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

<https://www.geeksforgeeks.org/vertex-cover-problem-set-1-introduction-approximate-algorithm-2/>

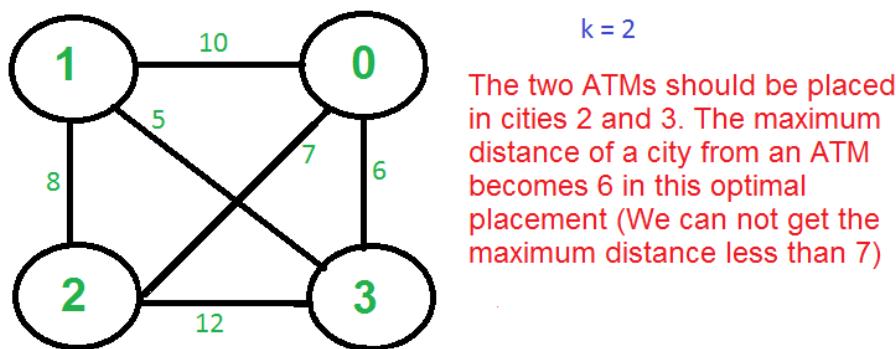
Chapter 61

K Centers Problem Set 1 (Greedy Approximate Algorithm)

K Centers Problem Set 1 (Greedy Approximate Algorithm) - GeeksforGeeks

Given n cities and distances between every pair of cities, select k cities to place warehouses (or ATMs or Cloud Server) such that the maximum distance of a city to a warehouse (or ATM or Cloud Server) is minimized.

For example consider the following four cities, 0, 1, 2 and 3 and distances between them, how do place 2 ATMs among these 4 cities so that the maximum distance of a city to an ATM is minimized.

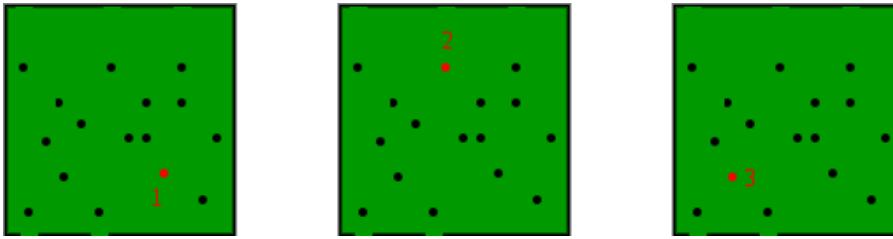


There is no polynomial time solution available for this problem as the problem is a known NP-Hard problem. There is a polynomial time Greedy approximate algorithm, the greedy algorithm provides a solution which is never worse than twice the optimal solution. The greedy solution works only if the distances between cities follow [Triangular Inequality](#) (Distance between two points is always smaller than sum of distances through a third point).

The 2-Approximate Greedy Algorithm:

- 1) Choose the first center arbitrarily.
- 2) Choose remaining $k-1$ centers using the following criteria.
Let $c_1, c_2, c_3, \dots, c_i$ be the already chosen centers. Choose $(i+1)$ 'th center by picking the city which is farthest from already selected centers, i.e., the point p which has following value as maximum

$$\text{Min}[\text{dist}(p, c_1), \text{dist}(p, c_2), \text{dist}(p, c_3), \dots, \text{dist}(p, c_i)]$$



Example ($k = 3$ in the above shown Graph)

- a) Let the first arbitrarily picked vertex be 0.
- b) The next vertex is 1 because 1 is the farthest vertex from 0.
- c) Remaining cities are 2 and 3. Calculate their distances from already selected centers (0 and 1). The greedy algorithm basically calculates following values.

Minimum of all distances from 2 to already considered centers
 $\text{Min}[\text{dist}(2, 0), \text{dist}(2, 1)] = \text{Min}[7, 8] = 7$

Minimum of all distances from 3 to already considered centers
 $\text{Min}[\text{dist}(3, 0), \text{dist}(3, 1)] = \text{Min}[6, 5] = 5$

After computing the above values, the city 2 is picked as the value corresponding to 2 is maximum.

Note that the greedy algorithm doesn't give best solution for $k = 2$ as this is just an approximate algorithm with bound as twice of optimal.

Proof that the above greedy algorithm is 2 approximate.

Let OPT be the maximum distance of a city from a center in the Optimal solution. We need to show that the maximum distance obtained from Greedy algorithm is $2 \cdot \text{OPT}$.

The proof can be done using contradiction.

- a) Assume that the distance from the furthest point to all centers is $> 2 \cdot \text{OPT}$.
- b) This means that distances between all centers are also $> 2 \cdot \text{OPT}$.
- c) We have $k + 1$ points with distances $> 2 \cdot \text{OPT}$ between every pair.
- d) Each point has a center of the optimal solution with distance $\leq \text{OPT}$ to it.
- e) There exists a pair of points with the same center X in the optimal solution (pigeonhole principle: k optimal centers, $k+1$ points)
- f) The distance between them is at most $2 \cdot \text{OPT}$ (triangle inequality) which is a contradiction.

Source:

<http://algo2.iti.kit.edu/vanstee/courses/kcenter.pdf>

This article is contributed by **Harshit**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

<https://www.geeksforgeeks.org/k-centers-problem-set-1-greedy-approximate-algorithm/>

Chapter 62

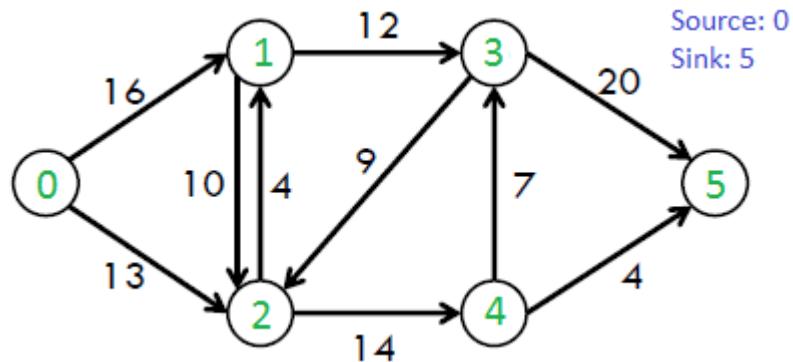
Ford-Fulkerson Algorithm for Maximum Flow Problem

Ford-Fulkerson Algorithm for Maximum Flow Problem - GeeksforGeeks

Given a graph which represents a flow network where every edge has a capacity. Also given two vertices *source* 's' and *sink* 't' in the graph, find the maximum possible flow from s to t with following constraints:

- a) Flow on an edge doesn't exceed the given capacity of the edge.
- b) Incoming flow is equal to outgoing flow for every vertex except s and t.

For example, consider the following graph from CLRS book.



The maximum possible flow in the above graph is 23.

Prerequisite : [Max Flow Problem Introduction](#)

Ford-Fulkerson Algorithm

The following is simple idea of Ford-Fulkerson algorithm:

- 1) Start with initial flow as 0.
- 2) While there is a augmenting path from source to sink.
Add this path-flow to flow.

3) Return flow.

Time Complexity: Time complexity of the above algorithm is $O(\text{max_flow} * E)$. We run a loop while there is an augmenting path. In worst case, we may add 1 unit flow in every iteration. Therefore the time complexity becomes $O(\text{max_flow} * E)$.

How to implement the above simple algorithm?

Let us first define the concept of Residual Graph which is needed for understanding the implementation.

Residual Graph of a flow network is a graph which indicates additional possible flow. If there is a path from source to sink in residual graph, then it is possible to add flow. Every edge of a residual graph has a value called **residual capacity** which is equal to original capacity of the edge minus current flow. Residual capacity is basically the current capacity of the edge.

Let us now talk about implementation details. Residual capacity is 0 if there is no edge between two vertices of residual graph. We can initialize the residual graph as original graph as there is no initial flow and initially residual capacity is equal to original capacity. To find an augmenting path, we can either do a BFS or DFS of the residual graph. We have used BFS in below implementation. Using BFS, we can find out if there is a path from source to sink. BFS also builds parent[] array. Using the parent[] array, we traverse through the found path and find possible flow through this path by finding minimum residual capacity along the path. We later add the found path flow to overall flow.

The important thing is, we need to update residual capacities in the residual graph. We subtract path flow from all edges along the path and we add path flow along the reverse edges. We need to add path flow along reverse edges because may later need to send flow in reverse direction (See following link for example).

<https://www.geeksforgeeks.org/max-flow-problem-introduction/>

Below is the implementation of Ford-Fulkerson algorithm. To keep things simple, graph is represented as a 2D matrix.

C++

```
// C++ program for implementation of Ford Fulkerson algorithm
#include <iostream>
#include <limits.h>
#include <string.h>
#include <queue>
using namespace std;

// Number of vertices in given graph
#define V 6

/* Returns true if there is a path from source 's' to sink 't' in
   residual graph. Also fills parent[] to store the path */
bool bfs(int rGraph[V][V], int s, int t, int parent[])
{
    // Create a visited array and mark all vertices as not visited
    bool visited[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;
    queue<int> q;
    q.push(s);
    visited[s] = true;
    while (!q.empty())
    {
        int u = q.front();
        q.pop();
        for (int v = 0; v < V; v++)
        {
            if (rGraph[u][v] > 0 && !visited[v])
            {
                parent[v] = u;
                if (v == t)
                    return true;
                q.push(v);
                visited[v] = true;
            }
        }
    }
    return false;
}
```

```

        memset(visited, 0, sizeof(visited));

        // Create a queue, enqueue source vertex and mark source vertex
        // as visited
        queue <int> q;
        q.push(s);
        visited[s] = true;
        parent[s] = -1;

        // Standard BFS Loop
        while (!q.empty())
        {
            int u = q.front();
            q.pop();

            for (int v=0; v<V; v++)
            {
                if (visited[v]==false && rGraph[u][v] > 0)
                {
                    q.push(v);
                    parent[v] = u;
                    visited[v] = true;
                }
            }
        }

        // If we reached sink in BFS starting from source, then return
        // true, else false
        return (visited[t] == true);
    }

    // Returns the maximum flow from s to t in the given graph
    int fordFulkerson(int graph[V][V], int s, int t)
    {
        int u, v;

        // Create a residual graph and fill the residual graph with
        // given capacities in the original graph as residual capacities
        // in residual graph
        int rGraph[V][V]; // Residual graph where rGraph[i][j] indicates
                           // residual capacity of edge from i to j (if there
                           // is an edge. If rGraph[i][j] is 0, then there is not)
        for (u = 0; u < V; u++)
            for (v = 0; v < V; v++)
                rGraph[u][v] = graph[u][v];

        int parent[V]; // This array is filled by BFS and to store path

```

```

int max_flow = 0; // There is no flow initially

// Augment the flow while there is path from source to sink
while (bfs(rGraph, s, t, parent))
{
    // Find minimum residual capacity of the edges along the
    // path filled by BFS. Or we can say find the maximum flow
    // through the path found.
    int path_flow = INT_MAX;
    for (v=t; v!=s; v=parent[v])
    {
        u = parent[v];
        path_flow = min(path_flow, rGraph[u][v]);
    }

    // update residual capacities of the edges and reverse edges
    // along the path
    for (v=t; v != s; v=parent[v])
    {
        u = parent[v];
        rGraph[u][v] -= path_flow;
        rGraph[v][u] += path_flow;
    }

    // Add path flow to overall flow
    max_flow += path_flow;
}

// Return the overall flow
return max_flow;
}

// Driver program to test above functions
int main()
{
    // Let us create a graph shown in the above example
    int graph[V][V] = { {0, 16, 13, 0, 0, 0},
                        {0, 0, 10, 12, 0, 0},
                        {0, 4, 0, 0, 14, 0},
                        {0, 0, 9, 0, 0, 20},
                        {0, 0, 0, 7, 0, 4},
                        {0, 0, 0, 0, 0, 0}
                    };

    cout << "The maximum possible flow is " << fordFulkerson(graph, 0, 5);

    return 0;
}

```

Java

```
// Java program for implementation of Ford Fulkerson algorithm
import java.util.*;
import java.lang.*;
import java.io.*;
import java.util.LinkedList;

class MaxFlow
{
    static final int V = 6;      //Number of vertices in graph

    /* Returns true if there is a path from source 's' to sink
       't' in residual graph. Also fills parent[] to store the
       path */
    boolean bfs(int rGraph[][] , int s, int t, int parent[])
    {
        // Create a visited array and mark all vertices as not
        // visited
        boolean visited[] = new boolean[V];
        for(int i=0; i<V; ++i)
            visited[i]=false;

        // Create a queue, enqueue source vertex and mark
        // source vertex as visited
        LinkedList<Integer> queue = new LinkedList<Integer>();
        queue.add(s);
        visited[s] = true;
        parent[s]=-1;

        // Standard BFS Loop
        while (queue.size()!=0)
        {
            int u = queue.poll();

            for (int v=0; v<V; v++)
            {
                if (visited[v]==false && rGraph[u][v] > 0)
                {
                    queue.add(v);
                    parent[v] = u;
                    visited[v] = true;
                }
            }
        }

        // If we reached sink in BFS starting from source, then
        // return true, else false
    }
}
```

```

        return (visited[t] == true);
    }

// Returns the maximum flow from s to t in the given graph
int fordFulkerson(int graph[][] , int s, int t)
{
    int u, v;

    // Create a residual graph and fill the residual graph
    // with given capacities in the original graph as
    // residual capacities in residual graph

    // Residual graph where rGraph[i][j] indicates
    // residual capacity of edge from i to j (if there
    // is an edge. If rGraph[i][j] is 0, then there is
    // not)
    int rGraph[][] = new int[V][V];

    for (u = 0; u < V; u++)
        for (v = 0; v < V; v++)
            rGraph[u][v] = graph[u][v];

    // This array is filled by BFS and to store path
    int parent[] = new int[V];

    int max_flow = 0; // There is no flow initially

    // Augment the flow while there is path from source
    // to sink
    while (bfs(rGraph, s, t, parent))
    {
        // Find minimum residual capacity of the edges
        // along the path filled by BFS. Or we can say
        // find the maximum flow through the path found.
        int path_flow = Integer.MAX_VALUE;
        for (v=t; v!=s; v=parent[v])
        {
            u = parent[v];
            path_flow = Math.min(path_flow, rGraph[u][v]);
        }

        // update residual capacities of the edges and
        // reverse edges along the path
        for (v=t; v != s; v=parent[v])
        {
            u = parent[v];
            rGraph[u][v] -= path_flow;
            rGraph[v][u] += path_flow;
        }
    }
}

```

```

        }

        // Add path flow to overall flow
        max_flow += path_flow;
    }

    // Return the overall flow
    return max_flow;
}

// Driver program to test above functions
public static void main (String[] args) throws java.lang.Exception
{
    // Let us create a graph shown in the above example
    int graph[][] =new int[][] { {0, 16, 13, 0, 0, 0},
                                {0, 0, 10, 12, 0, 0},
                                {0, 4, 0, 0, 14, 0},
                                {0, 0, 9, 0, 0, 20},
                                {0, 0, 0, 7, 0, 4},
                                {0, 0, 0, 0, 0, 0}
                            };
    MaxFlow m = new MaxFlow();

    System.out.println("The maximum possible flow is " +
                        m.fordFulkerson(graph, 0, 5));

}
}

```

Python

```

# Python program for implementation of Ford Fulkerson algorithm

from collections import defaultdict

#This class represents a directed graph using adjacency matrix representation
class Graph:

    def __init__(self,graph):
        self.graph = graph # residual graph
        self. ROW = len(graph)
        #self.COL = len(gr[0])

        '''Returns true if there is a path from source 's' to sink 't' in
        residual graph. Also fills parent[] to store the path '''
        def BFS(self,s, t, parent):


```

```

# Mark all the vertices as not visited
visited =[False]*(self.ROW)

# Create a queue for BFS
queue=[]

# Mark the source node as visited and enqueue it
queue.append(s)
visited[s] = True

# Standard BFS Loop
while queue:

    #Dequeue a vertex from queue and print it
    u = queue.pop(0)

    # Get all adjacent vertices of the dequeued vertex u
    # If a adjacent has not been visited, then mark it
    # visited and enqueue it
    for ind, val in enumerate(self.graph[u]):
        if visited[ind] == False and val > 0 :
            queue.append(ind)
            visited[ind] = True
            parent[ind] = u

    # If we reached sink in BFS starting from source, then return
    # true, else false
    return True if visited[t] else False

# Returns the maximum flow from s to t in the given graph
def FordFulkerson(self, source, sink):

    # This array is filled by BFS and to store path
    parent = [-1]*(self.ROW)

    max_flow = 0 # There is no flow initially

    # Augment the flow while there is path from source to sink
    while self.BFS(source, sink, parent) :

        # Find minimum residual capacity of the edges along the
        # path filled by BFS. Or we can say find the maximum flow
        # through the path found.
        path_flow = float("Inf")
        s = sink
        while(s != source):
            path_flow = min (path_flow, self.graph[parent[s]][s])
            s = parent[s]

    return max_flow

```

```

s = parent[s]

# Add path flow to overall flow
max_flow += path_flow

# update residual capacities of the edges and reverse edges
# along the path
v = sink
while(v != source):
    u = parent[v]
    self.graph[u][v] -= path_flow
    self.graph[v][u] += path_flow
    v = parent[v]

return max_flow

# Create a graph given in the above diagram

graph = [[0, 16, 13, 0, 0, 0],
          [0, 0, 10, 12, 0, 0],
          [0, 4, 0, 0, 14, 0],
          [0, 0, 9, 0, 0, 20],
          [0, 0, 0, 7, 0, 4],
          [0, 0, 0, 0, 0, 0]]

g = Graph(graph)

source = 0; sink = 5

print ("The maximum possible flow is %d " % g.FordFulkerson(source, sink))

#This code is contributed by Neelam Yadav

```

Output:

The maximum possible flow is 23

The above implementation of Ford Fulkerson Algorithm is called **Edmonds-Karp Algorithm**. The idea of Edmonds-Karp is to use BFS in Ford Fulkerson implementation as BFS always picks a path with minimum number of edges. When BFS is used, the worst case time complexity can be reduced to $O(VE^2)$. The above implementation uses adjacency matrix representation though where BFS takes $O(V^2)$ time, the time complexity of the above implementation is $O(EV^3)$ (Refer [CLRS book](#) for proof of time complexity)

This is an important problem as it arises in many practical situations. Examples include, maximizing the transportation with given traffic limits, maximizing packet flow in computer networks.

Dinic's Algorithm for Max-Flow.

Exercise:

Modify the above implementation so that it runs in $O(VE^2)$ time.

References:

<http://www.stanford.edu/class/cs97si/08-network-flow-problems.pdf>

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

Improved By : S3Reuis

Source

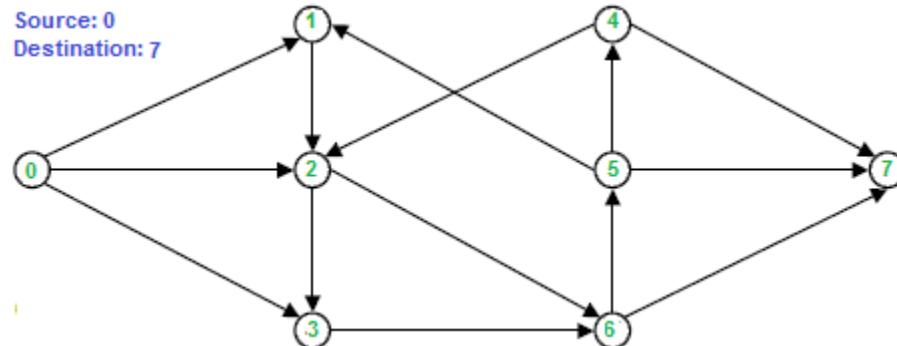
<https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/>

Chapter 63

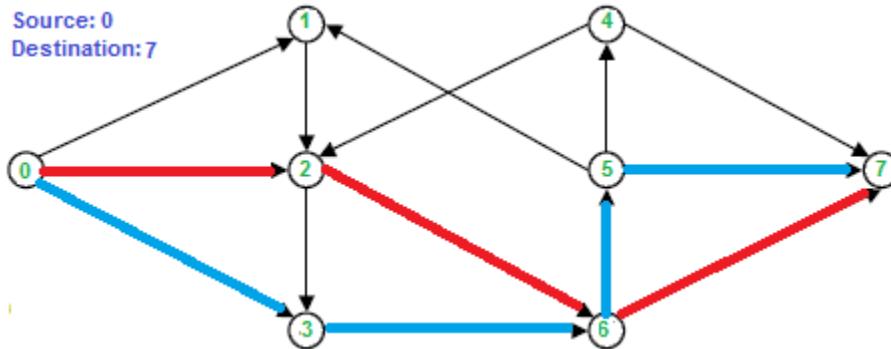
Find maximum number of edge disjoint paths between two vertices

Find maximum number of edge disjoint paths between two vertices - GeeksforGeeks

Given a directed graph and two vertices in it, source ‘s’ and destination ‘t’, find out the maximum number of edge disjoint paths from s to t. Two paths are said edge disjoint if they don’t share any edge.



There can be maximum two edge disjoint paths from source 0 to destination 7 in the above graph. Two edge disjoint paths are highlighted below in red and blue colors are 0-2-6-7 and 0-3-5-7.



Note that the paths may be different, but the maximum number is same. For example, in the above diagram, another possible set of paths is 0-1-2-6-7 and 0-3-6-5-7 respectively.

This problem can be solved by reducing it to [maximum flow problem](#). Following are steps.

- 1) Consider the given source and destination as source and sink in flow network. Assign unit capacity to each edge.
- 2) Run Ford-Fulkerson algorithm to find the maximum flow from source to sink.
- 3) The maximum flow is equal to the maximum number of edge-disjoint paths.

When we run Ford-Fulkerson, we reduce the capacity by a unit. Therefore, the edge can not be used again. So the maximum flow is equal to the maximum number of edge-disjoint paths.

Following is C++ implementation of the above algorithm. Most of the code is taken from [here](#).

C/C++

```
// C++ program to find maximum number of edge disjoint paths
#include <iostream>
#include <limits.h>
#include <string.h>
#include <queue>
using namespace std;

// Number of vertices in given graph
#define V 8

/* Returns true if there is a path from source 's' to sink 't' in
   residual graph. Also fills parent[] to store the path */
bool bfs(int rGraph[V][V], int s, int t, int parent[])
{
    // Create a visited array and mark all vertices as not visited
    bool visited[V];
    memset(visited, 0, sizeof(visited));

    // Create a queue, enqueue source vertex and mark source vertex
    // as visited
    queue <int> q;
    q.push(s);
    visited[s] = true;
    while (!q.empty())
    {
        int u = q.front();
        q.pop();
        for (int v = 0; v < V; v++)
        {
            if (rGraph[u][v] > 0 && !visited[v])
            {
                parent[v] = u;
                if (v == t)
                    return true;
                q.push(v);
                visited[v] = true;
            }
        }
    }
    return false;
}

// A utility function to print shortest distance from s to t
void printPath(int parent[], int t)
{
    if (parent[t] == -1)
        cout << t;
    else
        printPath(parent, parent[t]);
    cout << " " << t;
}
```

```

q.push(s);
visited[s] = true;
parent[s] = -1;

// Standard BFS Loop
while (!q.empty())
{
    int u = q.front();
    q.pop();

    for (int v=0; v<V; v++)
    {
        if (visited[v]==false && rGraph[u][v] > 0)
        {
            q.push(v);
            parent[v] = u;
            visited[v] = true;
        }
    }
}

// If we reached sink in BFS starting from source, then return
// true, else false
return (visited[t] == true);
}

// Returns the maximum number of edge-disjoint paths from s to t.
// This function is copy of forFulkerson() discussed at http://goo.gl/wtQ4Ks
int findDisjointPaths(int graph[V][V], int s, int t)
{
    int u, v;

    // Create a residual graph and fill the residual graph with
    // given capacities in the original graph as residual capacities
    // in residual graph
    int rGraph[V][V]; // Residual graph where rGraph[i][j] indicates
                      // residual capacity of edge from i to j (if there
                      // is an edge. If rGraph[i][j] is 0, then there is not)
    for (u = 0; u < V; u++)
        for (v = 0; v < V; v++)
            rGraph[u][v] = graph[u][v];

    int parent[V]; // This array is filled by BFS and to store path

    int max_flow = 0; // There is no flow initially

    // Augment the flow while there is path from source to sink
    while (bfs(rGraph, s, t, parent))

```

```

{
    // Find minimum residual capacity of the edges along the
    // path filled by BFS. Or we can say find the maximum flow
    // through the path found.
    int path_flow = INT_MAX;

    for (v=t; v!=s; v=parent[v])
    {
        u = parent[v];
        path_flow = min(path_flow, rGraph[u][v]);
    }

    // update residual capacities of the edges and reverse edges
    // along the path
    for (v=t; v != s; v=parent[v])
    {
        u = parent[v];
        rGraph[u][v] -= path_flow;
        rGraph[v][u] += path_flow;
    }

    // Add path flow to overall flow
    max_flow += path_flow;
}

// Return the overall flow (max_flow is equal to maximum
// number of edge-disjoint paths)
return max_flow;
}

// Driver program to test above functions
int main()
{
    // Let us create a graph shown in the above example
    int graph[V][V] = { {0, 1, 1, 1, 0, 0, 0, 0}, 
                        {0, 0, 1, 0, 0, 0, 0, 0}, 
                        {0, 0, 0, 1, 0, 0, 1, 0}, 
                        {0, 0, 0, 0, 0, 0, 1, 0}, 
                        {0, 0, 1, 0, 0, 0, 0, 1}, 
                        {0, 1, 0, 0, 0, 0, 0, 1}, 
                        {0, 0, 0, 0, 0, 1, 0, 1}, 
                        {0, 0, 0, 0, 0, 0, 0, 0} 
                    };
    
    int s = 0;
    int t = 7;
    cout << "There can be maximum " << findDisjointPaths(graph, s, t)
        << " edge-disjoint paths from " << s << " to " << t ;
}

```

```
    return 0;
}
```

Python

```
# Python program to find maximum number of edge disjoint paths
# Complexity : (E*(V^3))
# Total augmenting path = VE
# and BFS with adj matrix takes :V^2 times

from collections import defaultdict

#This class represents a directed graph using
# adjacency matrix representation
class Graph:

    def __init__(self,graph):
        self.graph = graph # residual graph
        self. ROW = len(graph)

    '''Returns true if there is a path from source 's' to sink 't' in
    residual graph. Also fills parent[] to store the path '''
    def BFS(self,s, t, parent):

        # Mark all the vertices as not visited
        visited =[False]*(self.ROW)

        # Create a queue for BFS
        queue=[]

        # Mark the source node as visited and enqueue it
        queue.append(s)
        visited[s] = True

        # Standard BFS Loop
        while queue:

            #Dequeue a vertex from queue and print it
            u = queue.pop(0)

            # Get all adjacent vertices of the dequeued vertex u
            # If a adjacent has not been visited, then mark it
            # visited and enqueue it
            for ind, val in enumerate(self.graph[u]):
                if visited[ind] == False and val > 0 :
                    queue.append(ind)
                    parent[ind] = u
```

```

        visited[ind] = True
        parent[ind] = u

    # If we reached sink in BFS starting from source, then return
    # true, else false
    return True if visited[t] else False

# Returns the maximum number of edge-disjoint paths from
#s to t in the given graph
def findDisjointPaths(self, source, sink):

    # This array is filled by BFS and to store path
    parent = [-1]*(self.ROW)

    max_flow = 0 # There is no flow initially

    # Augment the flow while there is path from source to sink
    while self.BFS(source, sink, parent) :

        # Find minimum residual capacity of the edges along the
        # path filled by BFS. Or we can say find the maximum flow
        # through the path found.
        path_flow = float("Inf")
        s = sink
        while(s != source):
            path_flow = min (path_flow, self.graph[parent[s]][s])
            s = parent[s]

        # Add path flow to overall flow
        max_flow += path_flow

        # update residual capacities of the edges and reverse edges
        # along the path
        v = sink
        while(v != source):
            u = parent[v]
            self.graph[u][v] -= path_flow
            self.graph[v][u] += path_flow
            v = parent[v]

    return max_flow

# Create a graph given in the above diagram

graph = [[0, 1, 1, 1, 0, 0, 0, 0], ,
          [0, 0, 1, 0, 0, 0, 0, 0], ,

```

```
[0, 0, 0, 1, 0, 0, 1, 0],  
[0, 0, 0, 0, 0, 1, 0],  
[0, 0, 1, 0, 0, 0, 0, 1],  
[0, 1, 0, 0, 0, 0, 0, 1],  
[0, 0, 0, 0, 1, 0, 1],  
[0, 0, 0, 0, 0, 0, 0]]
```

```
g = Graph(graph)  
  
source = 0; sink = 7  
  
print ("There can be maximum %d edge-disjoint paths from %d to %d" %  
       (g.findDisjointPaths(source, sink), source, sink))
```

#This code is contributed by Neelam Yadav

Output:

```
There can be maximum 2 edge-disjoint paths from 0 to 7
```

Time Complexity: Same as time complexity of Edmonds-Karp implementation of Ford-Fulkerson (See time complexity discussed [here](#))

References:

<http://www.win.tue.nl/~nikhil/courses/2012/2WO08/max-flow-applications-4up.pdf>

Source

<https://www.geeksforgeeks.org/find-edge-disjoint-paths-two-vertices/>

Chapter 64

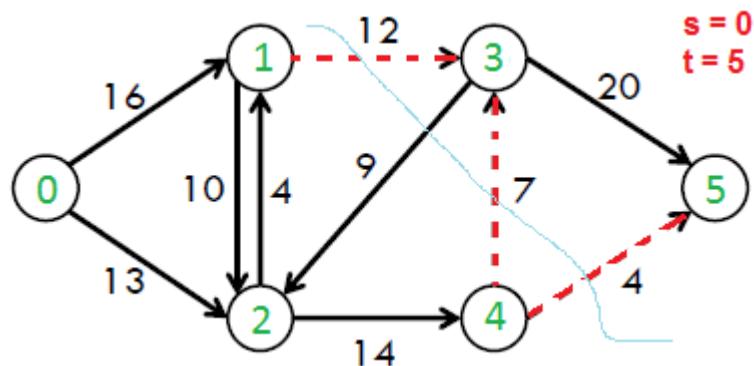
Find minimum s-t cut in a flow network

Find minimum s-t cut in a flow network - GeeksforGeeks

In a flow network, an s-t cut is a cut that requires the source 's' and the sink 't' to be in different subsets, and it consists of edges going from the source's side to the sink's side. The capacity of an s-t cut is defined by the sum of the capacity of each edge in the cut-set. (Source: [Wiki](#))

The problem discussed here is to find minimum capacity s-t cut of the given network. Expected output is all edges of the minimum cut.

For example, in the following flow network, example s-t cuts are $\{\{0, 1\}, \{0, 2\}\}$, $\{\{0, 2\}, \{1, 2\}, \{1, 3\}\}$, etc. The minimum s-t cut is $\{\{1, 3\}, \{4, 3\}, \{4, 5\}\}$ which has capacity as $12+7+4 = 23$.



We strongly recommend to read the below post first.

[Ford-Fulkerson Algorithm for Maximum Flow Problem](#)

Minimum Cut and Maximum Flow

Like [Maximum Bipartite Matching](#), this is another problem which can be solved using [Ford-Fulkerson Algorithm](#). This is based on max-flow min-cut theorem.

The [max-flow min-cut theorem](#) states that in a flow network, the amount of maximum flow is equal to capacity of the minimum cut. See [CLRS book](#) for proof of this theorem.

From Ford-Fulkerson, we get capacity of minimum cut. How to print all edges that form the minimum cut? The idea is to use [residual graph](#).

Following are steps to print all edges of the minimum cut.

- 1) Run Ford-Fulkerson algorithm and consider the final [residual graph](#).
- 2) Find the set of vertices that are reachable from the source in the residual graph.
- 3) All edges which are from a reachable vertex to non-reachable vertex are minimum cut edges. Print all such edges.

Following is C++ implementation of the above approach.

C++

```
// C++ program for finding minimum cut using Ford-Fulkerson
#include <iostream>
#include <limits.h>
#include <string.h>
#include <queue>
using namespace std;

// Number of vertices in given graph
#define V 6

/* Returns true if there is a path from source 's' to sink 't' in
   residual graph. Also fills parent[] to store the path */
int bfs(int rGraph[V][V], int s, int t, int parent[])
{
    // Create a visited array and mark all vertices as not visited
    bool visited[V];
    memset(visited, 0, sizeof(visited));

    // Create a queue, enqueue source vertex and mark source vertex
    // as visited
    queue <int> q;
    q.push(s);
    visited[s] = true;
    parent[s] = -1;

    // Standard BFS Loop
    while (!q.empty())
    {
        int u = q.front();
        q.pop();

        for (int v=0; v<V; v++)
        {
            if (rGraph[u][v] > 0 && !visited[v])
            {
                q.push(v);
                visited[v] = true;
                parent[v] = u;
            }
        }
    }
}

// A utility function to print the constructed s-t cut
void printCut(int parent[], int s, int t)
{
    cout << "The Minimum Cut is : ";
    for (int i=s; i!=t; i=parent[i])
        cout << i << " -> ";
    cout << t;
}
```

```

        if (visited[v]==false && rGraph[u] [v] > 0)
        {
            q.push(v);
            parent[v] = u;
            visited[v] = true;
        }
    }

// If we reached sink in BFS starting from source, then return
// true, else false
return (visited[t] == true);
}

// A DFS based function to find all reachable vertices from s. The function
// marks visited[i] as true if i is reachable from s. The initial values in
// visited[] must be false. We can also use BFS to find reachable vertices
void dfs(int rGraph[V][V], int s, bool visited[])
{
    visited[s] = true;
    for (int i = 0; i < V; i++)
        if (rGraph[s][i] && !visited[i])
            dfs(rGraph, i, visited);
}

// Prints the minimum s-t cut
void minCut(int graph[V][V], int s, int t)
{
    int u, v;

    // Create a residual graph and fill the residual graph with
    // given capacities in the original graph as residual capacities
    // in residual graph
    int rGraph[V][V]; // rGraph[i][j] indicates residual capacity of edge i-j
    for (u = 0; u < V; u++)
        for (v = 0; v < V; v++)
            rGraph[u][v] = graph[u][v];

    int parent[V]; // This array is filled by BFS and to store path

    // Augment the flow while there is a path from source to sink
    while (bfs(rGraph, s, t, parent))
    {
        // Find minimum residual capacity of the edges along the
        // path filled by BFS. Or we can say find the maximum flow
        // through the path found.
        int path_flow = INT_MAX;
        for (v=t; v!=s; v=parent[v])

```

```

{
    u = parent[v];
    path_flow = min(path_flow, rGraph[u][v]);
}

// update residual capacities of the edges and reverse edges
// along the path
for (v=t; v != s; v=parent[v])
{
    u = parent[v];
    rGraph[u][v] -= path_flow;
    rGraph[v][u] += path_flow;
}
}

// Flow is maximum now, find vertices reachable from s
bool visited[V];
memset(visited, false, sizeof(visited));
dfs(rGraph, s, visited);

// Print all edges that are from a reachable vertex to
// non-reachable vertex in the original graph
for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
        if (visited[i] && !visited[j] && graph[i][j])
            cout << i << " - " << j << endl;

return;
}

// Driver program to test above functions
int main()
{
    // Let us create a graph shown in the above example
    int graph[V][V] = { {0, 16, 13, 0, 0, 0},
                        {0, 0, 10, 12, 0, 0},
                        {0, 4, 0, 0, 14, 0},
                        {0, 0, 9, 0, 0, 20},
                        {0, 0, 0, 7, 0, 4},
                        {0, 0, 0, 0, 0, 0} };
}

minCut(graph, 0, 5);

return 0;
}

```

Java

```

// Java program for finding min-cut in the given graph
import java.util.LinkedList;
import java.util.Queue;

public class Graph {

    // Returns true if there is a path
    // from source 's' to sink 't' in residual
    // graph. Also fills parent[] to store the path
    private static boolean bfs(int[][] rGraph, int s,
                               int t, int[] parent) {

        // Create a visited array and mark
        // all vertices as not visited
        boolean[] visited = new boolean[rGraph.length];

        // Create a queue, enqueue source vertex
        // and mark source vertex as visited
        Queue<Integer> q = new LinkedList<Integer>();
        q.add(s);
        visited[s] = true;
        parent[s] = -1;

        // Standard BFS Loop
        while (!q.isEmpty()) {
            int v = q.poll();
            for (int i = 0; i < rGraph.length; i++) {
                if (rGraph[v][i] > 0 && !visited[i]) {
                    q.offer(i);
                    visited[i] = true;
                    parent[i] = v;
                }
            }
        }

        // If we reached sink in BFS starting
        // from source, then return true, else false
        return (visited[t] == true);
    }

    // A DFS based function to find all reachable
    // vertices from s. The function marks visited[i]
    // as true if i is reachable from s. The initial
    // values in visited[] must be false. We can also
    // use BFS to find reachable vertices
    private static void dfs(int[][] rGraph, int s,
                           boolean[] visited) {
        visited[s] = true;

```

```

        for (int i = 0; i < rGraph.length; i++) {
            if (rGraph[s][i] > 0 && !visited[i]) {
                dfs(rGraph, i, visited);
            }
        }

        // Prints the minimum s-t cut
        private static void minCut(int[][] graph, int s, int t) {
            int u,v;

            // Create a residual graph and fill the residual
            // graph with given capacities in the original
            // graph as residual capacities in residual graph
            // rGraph[i][j] indicates residual capacity of edge i-j
            int[][] rGraph = new int[graph.length][graph.length];
            for (int i = 0; i < graph.length; i++) {
                for (int j = 0; j < graph.length; j++) {
                    rGraph[i][j] = graph[i][j];
                }
            }

            // This array is filled by BFS and to store path
            int[] parent = new int[graph.length];

            // Augment the flow while there is path from source to sink
            while (bfs(rGraph, s, t, parent)) {

                // Find minimum residual capacity of the edges
                // along the path filled by BFS. Or we can say
                // find the maximum flow through the path found.
                int pathFlow = Integer.MAX_VALUE;
                for (v = t; v != s; v = parent[v]) {
                    u = parent[v];
                    pathFlow = Math.min(pathFlow, rGraph[u][v]);
                }

                // update residual capacities of the edges and
                // reverse edges along the path
                for (v = t; v != s; v = parent[v]) {
                    u = parent[v];
                    rGraph[u][v] = rGraph[u][v] - pathFlow;
                    rGraph[v][u] = rGraph[v][u] + pathFlow;
                }
            }

            // Flow is maximum now, find vertices reachable from s
            boolean[] isVisited = new boolean[graph.length];
        }
    }
}

```

```

dfs(rGraph, s, isVisited);

// Print all edges that are from a reachable vertex to
// non-reachable vertex in the original graph
for (int i = 0; i < graph.length; i++) {
    for (int j = 0; j < graph.length; j++) {
        if (graph[i][j] > 0 && isVisited[i] && !isVisited[j]) {
            System.out.println(i + " - " + j);
        }
    }
}

//Driver Program
public static void main(String args[]) {

    // Let us create a graph shown in the above example
    int graph[][] = { {0, 16, 13, 0, 0, 0},
                      {0, 0, 10, 12, 0, 0},
                      {0, 4, 0, 0, 14, 0},
                      {0, 0, 9, 0, 0, 20},
                      {0, 0, 0, 7, 0, 4},
                      {0, 0, 0, 0, 0, 0}
                  };
    minCut(graph, 0, 5);
}
// This code is contributed by Himanshu Shekhar

```

Python

```

# Python program for finding min-cut in the given graph
# Complexity : (E*(V^3))
# Total augmenting path = VE and BFS with adj matrix takes :V^2 times

from collections import defaultdict

# This class represents a directed graph using adjacency matrix representation
class Graph:

    def __init__(self,graph):
        self.graph = graph # residual graph
        self.org_graph = [i[:] for i in graph]
        self.ROW = len(graph)
        self.COL = len(graph[0])

        '''Returns true if there is a path from source 's' to sink 't' in

```

```

residual graph. Also fills parent[] to store the path ''
def BFS(self,s, t, parent):

    # Mark all the vertices as not visited
    visited =[False]*(self.ROW)

    # Create a queue for BFS
    queue=[]

    # Mark the source node as visited and enqueue it
    queue.append(s)
    visited[s] = True

    # Standard BFS Loop
    while queue:

        #Dequeue a vertex from queue and print it
        u = queue.pop(0)

        # Get all adjacent vertices of the dequeued vertex u
        # If a adjacent has not been visited, then mark it
        # visited and enqueue it
        for ind, val in enumerate(self.graph[u]):
            if visited[ind] == False and val > 0 :
                queue.append(ind)
                visited[ind] = True
                parent[ind] = u

    # If we reached sink in BFS starting from source, then return
    # true, else false
    return True if visited[t] else False

# Returns the min-cut of the given graph
def minCut(self, source, sink):

    # This array is filled by BFS and to store path
    parent = [-1]*(self.ROW)

    max_flow = 0 # There is no flow initially

    # Augment the flow while there is path from source to sink
    while self.BFS(source, sink, parent) :

        # Find minimum residual capacity of the edges along the
        # path filled by BFS. Or we can say find the maximum flow
        # through the path found.
        path_flow = float("Inf")

```

```

s = sink
while(s != source):
    path_flow = min (path_flow, self.graph[parent[s]][s])
    s = parent[s]

    # Add path flow to overall flow
    max_flow += path_flow

    # update residual capacities of the edges and reverse edges
    # along the path
    v = sink
    while(v != source):
        u = parent[v]
        self.graph[u][v] -= path_flow
        self.graph[v][u] += path_flow
        v = parent[v]

    # print the edges which initially had weights
    # but now have 0 weight
    for i in range(self.ROW):
        for j in range(self.COL):
            if self.graph[i][j] == 0 and self.org_graph[i][j] > 0:
                print str(i) + " - " + str(j)

# Create a graph given in the above diagram
graph = [[0, 16, 13, 0, 0, 0],
          [0, 0, 10, 12, 0, 0],
          [0, 4, 0, 0, 14, 0],
          [0, 0, 9, 0, 0, 20],
          [0, 0, 0, 7, 0, 4],
          [0, 0, 0, 0, 0, 0]]

g = Graph(graph)

source = 0; sink = 5

g.minCut(source, sink)

# This code is contributed by Neelam Yadav

```

Output:

```

1 - 3
4 - 3
4 - 5

```

References:

<http://www.stanford.edu/class/cs97si/08-network-flow-problems.pdf>

<http://www.cs.princeton.edu/courses/archive/spring06/cos226/lectures/maxflow.pdf>

Source

<https://www.geeksforgeeks.org/minimum-cut-in-a-directed-graph/>

Chapter 65

Maximum Bipartite Matching

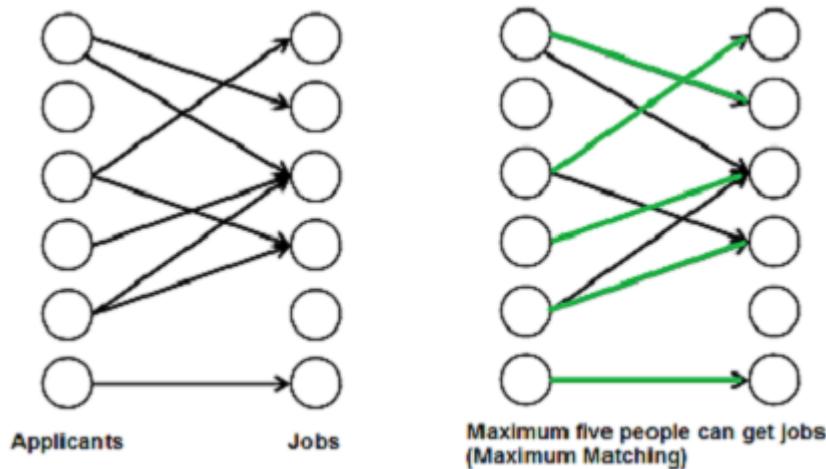
Maximum Bipartite Matching - GeeksforGeeks

A matching in a [Bipartite Graph](#) is a set of the edges chosen in such a way that no two edges share an endpoint. A maximum matching is a matching of maximum size (maximum number of edges). In a maximum matching, if any edge is added to it, it is no longer a matching. There can be more than one maximum matchings for a given Bipartite Graph.

Why do we care?

There are many real world problems that can be formed as Bipartite Matching. For example, consider the following problem:

There are M job applicants and N jobs. Each applicant has a subset of jobs that he/she is interested in. Each job opening can only accept one applicant and a job applicant can be appointed for only one job. Find an assignment of jobs to applicants in such that as many applicants as possible get jobs.

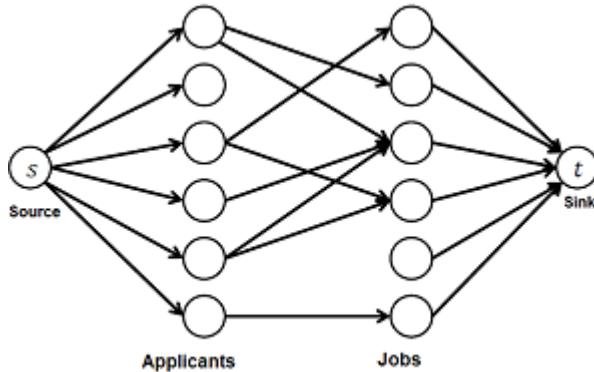


We strongly recommend to read the following post first.

[Ford-Fulkerson Algorithm for Maximum Flow Problem](#)

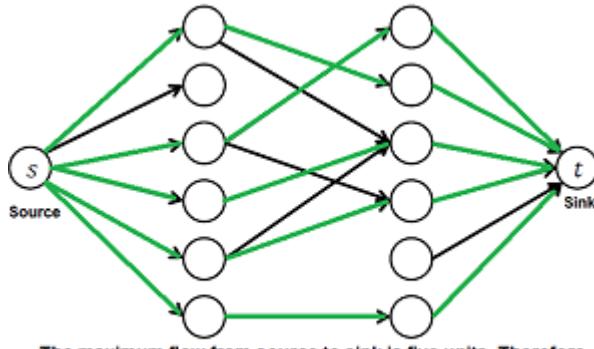
Maximum Bipartite Matching and Max Flow Problem

Maximum Bipartite Matching (MBP) problem can be solved by converting it into a flow network (See [this video](#) to know how did we arrive at this conclusion). Following are the steps.



1) Build a Flow Network

There must be a source and sink in a flow network. So we add a source and add edges from source to all applicants. Similarly, add edges from all jobs to sink. The capacity of every edge is marked as 1 unit.



The maximum flow from source to sink is five units. Therefore, maximum five people can get jobs.

2) Find the maximum flow.

We use [Ford-Fulkerson algorithm](#) to find the maximum flow in the flow network built in step 1. The maximum flow is actually the MBP we are looking for.

How to implement the above approach?

Let us first define input and output forms. Input is in the form of [Edmonds matrix](#) which is a 2D array ‘bpGraph[M][N]’ with M rows (for M job applicants) and N columns (for N jobs). The value $bpGraph[i][j]$ is 1 if i'th applicant is interested in j'th job, otherwise 0. Output is number maximum number of people that can get jobs.

A simple way to implement this is to create a matrix that represents [adjacency matrix representation](#) of a directed graph with $M+N+2$ vertices. Call the [fordFulkerson\(\)](#) for the matrix. This implementation requires $O((M+N)*(M+N))$ extra space.

Extra space can be reduced and code can be simplified using the fact that the graph is bipartite and capacity of every edge is either 0 or 1. The idea is to use DFS traversal to find a job for an applicant (similar to augmenting path in Ford-Fulkerson). We call bpm() for every applicant, bpm() is the DFS based function that tries all possibilities to assign a job to the applicant.

In bpm(), we one by one try all jobs that an applicant ‘u’ is interested in until we find a job, or all jobs are tried without luck. For every job we try, we do following.

If a job is not assigned to anybody, we simply assign it to the applicant and return true. If a job is assigned to somebody else say x, then we recursively check whether x can be assigned some other job. To make sure that x doesn’t get the same job again, we mark the job ‘v’ as seen before we make recursive call for x. If x can get other job, we change the applicant for job ‘v’ and return true. We use an array maxR[0..N-1] that stores the applicants assigned to different jobs.

If bmp() returns true, then it means that there is an augmenting path in flow network and 1 unit of flow is added to the result in maxBPM().

C++

```
// A C++ program to find maximal
// Bipartite matching.
#include <iostream>
#include <string.h>
using namespace std;

// M is number of applicants
// and N is number of jobs
#define M 6
#define N 6

// A DFS based recursive function
// that returns true if a matching
// for vertex u is possible
bool bpm(bool bpGraph[M][N], int u,
         bool seen[], int matchR[])
{
    // Try every job one by one
    for (int v = 0; v < N; v++)
    {
        // If applicant u is interested in
        // job v and v is not visited
        if (bpGraph[u][v] && !seen[v])
        {
            // Mark v as visited
            seen[v] = true;

            // If job 'v' is not assigned to an
            // applicant OR previously assigned
```

```

        // applicant for job v (which is matchR[v])
        // has an alternate job available.
        // Since v is marked as visited in
        // the above line, matchR[v] in the following
        // recursive call will not get job 'v' again
        if (matchR[v] < 0 || bpm(bpGraph, matchR[v],
                                seen, matchR))
        {
            matchR[v] = u;
            return true;
        }
    }
    return false;
}

// Returns maximum number
// of matching from M to N
int maxBPM(bool bpGraph[M][N])
{
    // An array to keep track of the
    // applicants assigned to jobs.
    // The value of matchR[i] is the
    // applicant number assigned to job i,
    // the value -1 indicates nobody is
    // assigned.
    int matchR[N];

    // Initially all jobs are available
    memset(matchR, -1, sizeof(matchR));

    // Count of jobs assigned to applicants
    int result = 0;
    for (int u = 0; u < M; u++)
    {
        // Mark all jobs as not seen
        // for next applicant.
        bool seen[N];
        memset(seen, 0, sizeof(seen));

        // Find if the applicant 'u' can get a job
        if (bpm(bpGraph, u, seen, matchR))
            result++;
    }
    return result;
}

// Driver Code

```

```

int main()
{
    // Let us create a bpGraph
    // shown in the above example
    bool bpGraph[M][N] = {{0, 1, 1, 0, 0, 0},
                          {1, 0, 0, 1, 0, 0},
                          {0, 0, 1, 0, 0, 0},
                          {0, 0, 1, 1, 0, 0},
                          {0, 0, 0, 0, 0, 0},
                          {0, 0, 0, 0, 0, 1}};

    cout << "Maximum number of applicants that can get job is "
         << maxBPM(bpGraph);

    return 0;
}

```

Java

```

// A Java program to find maximal
// Bipartite matching.
import java.util.*;
import java.lang.*;
import java.io.*;

class GFG
{
    // M is number of applicants
    // and N is number of jobs
    static final int M = 6;
    static final int N = 6;

    // A DFS based recursive function that
    // returns true if a matching for
    // vertex u is possible
    boolean bpm(boolean bpGraph[][], int u,
                boolean seen[], int matchR[])
    {
        // Try every job one by one
        for (int v = 0; v < N; v++)
        {
            // If applicant u is interested
            // in job v and v is not visited
            if (bpGraph[u][v] && !seen[v])
            {

                // Mark v as visited
                seen[v] = true;

```

```

        // If job 'v' is not assigned to
        // an applicant OR previously
        // assigned applicant for job v (which
        // is matchR[v]) has an alternate job available.
        // Since v is marked as visited in the
        // above line, matchR[v] in the following
        // recursive call will not get job 'v' again
        if (matchR[v] < 0 || bpm(bpGraph, matchR[v],
                                seen, matchR))
        {
            matchR[v] = u;
            return true;
        }
    }
    return false;
}

// Returns maximum number
// of matching from M to N
int maxBPM(boolean bpGraph[][])
{
    // An array to keep track of the
    // applicants assigned to jobs.
    // The value of matchR[i] is the
    // applicant number assigned to job i,
    // the value -1 indicates nobody is assigned.
    int matchR[] = new int[N];

    // Initially all jobs are available
    for(int i = 0; i < N; ++i)
        matchR[i] = -1;

    // Count of jobs assigned to applicants
    int result = 0;
    for (int u = 0; u < M; u++)
    {
        // Mark all jobs as not seen
        // for next applicant.
        boolean seen[] =new boolean[N] ;
        for(int i = 0; i < N; ++i)
            seen[i] = false;

        // Find if the applicant 'u' can get a job
        if (bpm(bpGraph, u, seen, matchR))
            result++;
    }
}

```

```

        return result;
    }

    // Driver Code
    public static void main (String[] args)
        throws java.lang.Exception
    {
        // Let us create a bpGraph shown
        // in the above example
        boolean bpGraph[][] = new boolean[][]{
            {false, true, true,
             false, false, false},
            {true, false, false,
             true, false, false},
            {false, false, true,
             false, false, false},
            {false, false, true,
             true, false, false},
            {false, false, false,
             false, false, false},
            {false, false, false,
             false, false, true}};
        GFG m = new GFG();
        System.out.println( "Maximum number of applicants that can"+
                           " get job is "+m.maxBPM(bpGraph));
    }
}

```

Python

```

# Python program to find
# maximal Bipartite matching.

class GFG:
    def __init__(self,graph):

        # residual graph
        self.graph = graph
        self.ppl = len(graph)
        self.jobs = len(graph[0])

    # A DFS based recursive function
    # that returns true if a matching
    # for vertex u is possible
    def bpm(self, u, matchR, seen):

        # Try every job one by one
        for v in range(self.jobs):

```

```

# If applicant u is interested
# in job v and v is not seen
if self.graph[u][v] and seen[v] == False:

    # Mark v as visited
    seen[v] = True

    '''If job 'v' is not assigned to
    an applicant OR previously assigned
    applicant for job v (which is matchR[v])
    has an alternate job available.
    Since v is marked as visited in the
    above line, matchR[v] in the following
    recursive call will not get job 'v' again'''
    if matchR[v] == -1 or self.bpm(matchR[v],
                                    matchR, seen):
        matchR[v] = u
        return True
    return False

# Returns maximum number of matching
def maxBPM(self):
    '''An array to keep track of the
    applicants assigned to jobs.
    The value of matchR[i] is the
    applicant number assigned to job i,
    the value -1 indicates nobody is assigned.'''
    matchR = [-1] * self.jobs

    # Count of jobs assigned to applicants
    result = 0
    for i in range(self.ppl):

        # Mark all jobs as not seen for next applicant.
        seen = [False] * self.jobs

        # Find if the applicant 'u' can get a job
        if self.bpm(i, matchR, seen):
            result += 1
    return result

bpGraph = [[0, 1, 1, 0, 0, 0],
           [1, 0, 0, 1, 0, 0],
           [0, 0, 1, 0, 0, 0],
           [0, 0, 1, 1, 0, 0],
           [0, 0, 0, 0, 0, 0],
           [0, 0, 0, 0, 0, 0],
           [0, 0, 0, 0, 0, 0],
           [0, 0, 0, 0, 0, 0],
           [0, 0, 0, 0, 0, 0],
           [0, 0, 0, 0, 0, 0]]

```

```
[0, 0, 0, 0, 0, 1]

g = GFG(bpGraph)

print ("Maximum number of applicants that can get job is %d " % g.maxBPM())

# This code is contributed by Neelam Yadav

C#

// A C# program to find maximal
// Bipartite matching.
using System;

class GFG
{
    // M is number of applicants
    // and N is number of jobs
    static int M = 6;
    static int N = 6;

    // A DFS based recursive function
    // that returns true if a matching
    // for vertex u is possible
    bool bpm(bool[,] bpGraph, int u,
             bool[] seen, int[] matchR)
    {
        // Try every job one by one
        for (int v = 0; v < N; v++)
        {
            // If applicant u is interested
            // in job v and v is not visited
            if (bpGraph[u, v] && !seen[v])
            {
                // Mark v as visited
                seen[v] = true;

                // If job 'v' is not assigned to
                // an applicant OR previously assigned
                // applicant for job v (which is matchR[v])
                // has an alternate job available.
                // Since v is marked as visited in the above
                // line, matchR[v] in the following recursive
                // call will not get job 'v' again
                if (matchR[v] < 0 || bpm(bpGraph, matchR[v],
                                           seen, matchR))
                {
                    matchR[v] = u;
                    return true;
                }
            }
        }
        return false;
    }

    // This function is used by dfs()
    // It returns true if a matching
    // for vertex u is possible
    int maxBPM()
    {
        int[] matchR = new int[N];
        for (int i = 0; i < N; i++)
            matchR[i] = -1;
        int count = 0;
        for (int u = 0; u < M; u++)
            if (bpm(bpGraph, u, seen, matchR))
                count++;
        return count;
    }
}
```

```

        return true;
    }
}
}
return false;
}

// Returns maximum number of
// matching from M to N
int maxBPM(bool[,] bpGraph)
{
    // An array to keep track of the
    // applicants assigned to jobs.
    // The value of matchR[i] is the
    // applicant number assigned to job i,
    // the value -1 indicates nobody is assigned.
    int []matchR = new int[N];

    // Initially all jobs are available
    for(int i = 0; i < N; ++i)
        matchR[i] = -1;

    // Count of jobs assigned to applicants
    int result = 0;
    for (int u = 0; u < M; u++)
    {
        // Mark all jobs as not
        // seen for next applicant.
        bool []seen = new bool[N];
        for(int i = 0; i < N; ++i)
            seen[i] = false;

        // Find if the applicant
        // 'u' can get a job
        if (bpm(bpGraph, u, seen, matchR))
            result++;
    }
    return result;
}

// Driver Code
public static void Main ()
{
    // Let us create a bpGraph shown
    // in the above example
    bool [,]bpGraph = new bool[,]
        {{false, true, true,
          false, false, false},

```

```
        {true, false, false,
         true, false, false},
        {false, false, true,
         false, false, false},
        {false, false, true,
         true, false, false},
        {false, false, false,
         false, false, false},
        {false, false, false,
         false, false, true}};

GFG m = new GFG();
Console.WriteLine("Maximum number of applicants that can"+
                  " get job is "+m.maxBPM(bpGraph));
}
}

//This code is contributed by nitin mittal.
```

Output :

Maximum number of applicants that can get job is 5

You may like to see below also:

[Hopcroft–Karp Algorithm for Maximum Matching Set 1 \(Introduction\)](#)

[Hopcroft–Karp Algorithm for Maximum Matching Set 2 \(Implementation\)](#)

References:

http://www.cs.cornell.edu/~wdtseng/icpc/notes/graph_part5.pdf

<http://www.youtube.com/watch?v=NlQqmEXuiC8>

http://en.wikipedia.org/wiki/Maximum_matching

<http://www.stanford.edu/class/cs97si/08-network-flow-problems.pdf>

<http://www.cs.princeton.edu/courses/archive/spring13/cos423/lectures/07NetworkFlowII-2x2.pdf>

http://www.ise.ncsu.edu/fangroup/or766.dir/or766_ch7.pdf

Improved By : nitin mittal

Source

<https://www.geeksforgeeks.org/maximum-bipartite-matching/>

Chapter 66

Channel Assignment Problem

Channel Assignment Problem - GeeksforGeeks

There are M transmitter and N receiver stations. Given a matrix that keeps track of the number of packets to be transmitted from a given transmitter to a receiver. If the (i; j)-th entry of the matrix is k, it means at that time the station i has k packets for transmission to station j.

During a time slot, a transmitter can send only one packet and a receiver can receive only one packet. Find the channel assignments so that maximum number of packets are transferred from transmitters to receivers during the next time slot.

Example:

```
0 2 0  
3 0 1  
2 4 0
```

The above is the input format. We call the above matrix M. Each value $M[i; j]$ represents the number of packets Transmitter ‘i’ has to send to Receiver ‘j’. The output should be:

```
The number of maximum packets sent in the time slot is 3  
T1 -> R2  
T2 -> R3  
T3 -> R1
```

Note that the maximum number of packets that can be transferred in any slot is $\min(M, N)$.

Algorithm:

The channel assignment problem between sender and receiver can be easily transformed into Maximum Bipartite Matching(MBP) problem that can be solved by converting it into a flow network.

Step 1: Build a Flow Network

There must be a source and sink in a flow network. So we add a dummy source and add edges from source to all senders. Similarly, add edges from all receivers to dummy sink. The capacity of all added edges is marked as 1 unit.

Step 2: Find the maximum flow.

We use [Ford-Fulkerson algorithm](#) to find the maximum flow in the flow network built in step 1. The maximum flow is actually the maximum number of packets that can be transmitted without bandwidth interference in a time slot.

Implementation:

Let us first define input and output forms. Input is in the form of Edmonds matrix which is a 2D array ‘table[M][N]’ with M rows (for M senders) and N columns (for N receivers). The value $\text{table}[i][j]$ is the number of packets that has to be sent from transmitter ‘i’ to receiver ‘j’. Output is the maximum number of packets that can be transmitted without bandwidth interference in a time slot.

A simple way to implement this is to create a matrix that represents adjacency matrix representation of a directed graph with $M+N+2$ vertices. Call the `fordFulkerson()` for the matrix. This implementation requires $O((M+N)*(M+N))$ extra space.

Extra space can be reduced and code can be simplified using the fact that the graph is bipartite. The idea is to use DFS traversal to find a receiver for a transmitter (similar to augmenting path in Ford-Fulkerson). We call `bpm()` for every applicant, `bpm()` is the DFS based function that tries all possibilities to assign a receiver to the sender. In `bpm()`, we one by one try all receivers that a sender ‘u’ is interested in until we find a receiver, or all receivers are tried without luck.

For every receiver we try, we do following:

If a receiver is not assigned to anybody, we simply assign it to the sender and return true. If a receiver is assigned to somebody else say x, then we recursively check whether x can be assigned some other receiver. To make sure that x doesn’t get the same receiver again, we mark the receiver ‘v’ as seen before we make recursive call for x. If x can get other receiver, we change the sender for receiver ‘v’ and return true. We use an array `maxR[0..N-1]` that stores the senders assigned to different receivers.

If `bmp()` returns true, then it means that there is an augmenting path in flow network and 1 unit of flow is added to the result in `maxBPM()`.

Time and space complexity analysis:

In case of bipartite matching problem, $F \leq V$ since there can be only V possible edges coming out from source node. So the total running time is $O(m'n) = O((m + n)n)$. The space complexity is also substantially reduces from $O((M+N)*(M+N))$ to just a single dimensional array of size M thus storing the mapping between M and N .

```
#include <iostream>
#include <string.h>
#include <vector>
#define M 3
#define N 4
using namespace std;

// A Depth First Search based recursive function that returns true
```

```

// if a matching for vertex u is possible
bool bpm(int table[M][N], int u, bool seen[], int matchR[])
{
    // Try every receiver one by one
    for (int v = 0; v < N; v++)
    {
        // If sender u has packets to send to receiver v and
        // receiver v is not already mapped to any other sender
        // just check if the number of packets is greater than '0'
        // because only one packet can be sent in a time frame anyways
        if (table[u][v]>0 && !seen[v])
        {
            seen[v] = true; // Mark v as visited

            // If receiver 'v' is not assigned to any sender OR
            // previously assigned sender for receiver v (which is
            // matchR[v]) has an alternate receiver available. Since
            // v is marked as visited in the above line, matchR[v] in
            // the following recursive call will not get receiver 'v' again
            if (matchR[v] < 0 || bpm(table, matchR[v], seen, matchR))
            {
                matchR[v] = u;
                return true;
            }
        }
    }
    return false;
}

// Returns maximum number of packets that can be sent parallelly in 1
// time slot from sender to receiver
int maxBPM(int table[M][N])
{
    // An array to keep track of the receivers assigned to the senders.
    // The value of matchR[i] is the sender ID assigned to receiver i.
    // the value -1 indicates nobody is assigned.
    int matchR[N];

    // Initially all receivers are not mapped to any senders
    memset(matchR, -1, sizeof(matchR));

    int result = 0; // Count of receivers assigned to senders
    for (int u = 0; u < M; u++)
    {
        // Mark all receivers as not seen for next sender
        bool seen[N];
        memset(seen, 0, sizeof(seen));

```

```
// Find if the sender 'u' can be assigned to the receiver
if (bpm(table, u, seen, matchR))
    result++;
}

cout << "The number of maximum packets sent in the time slot is "
<< result << "\n";

for (int x=0; x<N; x++)
    if (matchR[x]+1!=0)
        cout << "T" << matchR[x]+1 << "-> R" << x+1 << "\n";
return result;
}

// Driver program to test above function
int main()
{
    int table[M][N] = {{0, 2, 0}, {3, 0, 1}, {2, 4, 0}};
    int max_flow = maxBPM(table);
    return 0;
}
```

Output:

```
The number of maximum packets sent in the time slot is 3
T3-> R1
T1-> R2
T2-> R3
```

This article is contributed by [Vignesh Narayanan](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

<https://www.geeksforgeeks.org/channel-assignment-problem/>

Chapter 67

Union Find Algorithm Set 2 (Union By Rank and Path Compression)

Union-Find Algorithm Set 2 (Union By Rank and Path Compression) - GeeksforGeeks

In the [previous post](#), we introduced *union find algorithm* and used it to detect cycle in a graph. We used following *union()* and *find()* operations for subsets.

```
// Naive implementation of find
int find(int parent[], int i)
{
    if (parent[i] == -1)
        return i;
    return find(parent, parent[i]);
}

// Naive implementation of union()
void Union(int parent[], int x, int y)
{
    int xset = find(parent, x);
    int yset = find(parent, y);
    parent[xset] = yset;
}
```

The above *union()* and *find()* are naive and the worst case time complexity is linear. The trees created to represent subsets can be skewed and can become like a linked list. Following is an example worst case scenario.

Let there be 4 elements 0, 1, 2, 3

Initially, all elements are single element subsets.
0 1 2 3

```
Do Union(0, 1)
    1   2   3
    /
0
```

```
Do Union(1, 2)
    2   3
    /
1
/
0
```

```
Do Union(2, 3)
    3
    /
2
/
1
/
0
```

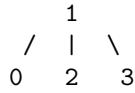
The above operations can be optimized to $O(\log n)$ in worst case. The idea is to always attach smaller depth tree under the root of the deeper tree. This technique is called **union by rank**. The term *rank* is preferred instead of height because if path compression technique (we have discussed it below) is used, then *rank* is not always equal to height. Also, size (in place of height) of trees can also be used as *rank*. Using size as *rank* also yields worst case time complexity as $O(\log n)$ (See [this](#) for proof)

Let us see the above example with union by rank
Initially, all elements are single element subsets.
0 1 2 3

```
Do Union(0, 1)
    1   2   3
    /
0
```

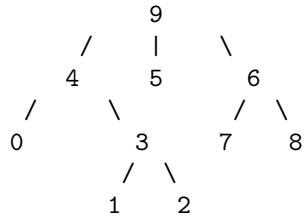
```
Do Union(1, 2)
    1   3
    /   \
0     2
```

```
Do Union(2, 3)
```

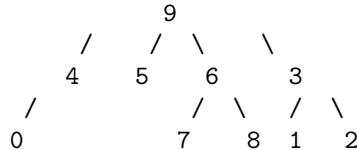


The second optimization to naive method is ***Path Compression***. The idea is to flatten the tree when *find()* is called. When *find()* is called for an element x, root of the tree is returned. The *find()* operation traverses up from x to find root. The idea of path compression is to make the found root as parent of x so that we don't have to traverse all intermediate nodes again. If x is root of a subtree, then path (to root) from all nodes under x also compresses.

Let the subset {0, 1, .. 9} be represented as below and *find()* is called for element 3.



When *find()* is called for 3, we traverse up and find 9 as representative of this subset. With path compression, we also make 3 as the child of 9 so that when *find()* is called next time for 1, 2 or 3, the path to root is reduced.



The two techniques complement each other. The time complexity of each operation becomes even smaller than O(Logn). In fact, amortized time complexity effectively becomes small constant.

Following is union by rank and path compression based implementation to find a cycle in a graph.

C++

```
// A union by rank and path compression based program to detect cycle in a graph
#include <stdio.h>
#include <stdlib.h>

// a structure to represent an edge in the graph
struct Edge
{
    int src, dest;
}
```

```

};

// a structure to represent a graph
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges
    struct Edge* edge;
};

struct subset
{
    int parent;
    int rank;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = (struct Graph*) malloc( sizeof(struct Graph) );
    graph->V = V;
    graph->E = E;

    graph->edge = (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );

    return graph;
}

// A utility function to find set of an element i
// (uses path compression technique)
int find(struct subset subsets[], int i)
{
    // find root and make root as parent of i (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(struct subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);
}

```

```

// Attach smaller rank tree under root of high rank tree
// (Union by Rank)
if (subsets[xroot].rank < subsets[yroot].rank)
    subsets[xroot].parent = yroot;
else if (subsets[xroot].rank > subsets[yroot].rank)
    subsets[yroot].parent = xroot;

// If ranks are same, then make one as root and increment
// its rank by one
else
{
    subsets[yroot].parent = xroot;
    subsets[xroot].rank++;
}
}

// The main function to check whether a given graph contains cycle or not
int isCycle( struct Graph* graph )
{
    int V = graph->V;
    int E = graph->E;

    // Allocate memory for creating V sets
    struct subset *subsets =
        (struct subset*) malloc( V * sizeof(struct subset) );

    for (int v = 0; v < V; ++v)
    {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Iterate through all edges of graph, find sets of both
    // vertices of every edge, if sets are same, then there is
    // cycle in graph.
    for(int e = 0; e < E; ++e)
    {
        int x = find(subsets, graph->edge[e].src);
        int y = find(subsets, graph->edge[e].dest);

        if (x == y)
            return 1;

        Union(subsets, x, y);
    }
    return 0;
}

```

```
// Driver program to test above functions
int main()
{
    /* Let us create the following graph
       0
       |
       |
       |   \
       1-----2 */

    int V = 3, E = 3;
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;

    // add edge 1-2
    graph->edge[1].src = 1;
    graph->edge[1].dest = 2;

    // add edge 0-2
    graph->edge[2].src = 0;
    graph->edge[2].dest = 2;

    if (isCycle(graph))
        printf( "Graph contains cycle" );
    else
        printf( "Graph doesn't contain cycle" );

    return 0;
}
```

Java

```
// A union by rank and path compression
// based program to detect cycle in a graph
class Graph
{
    int V, E;
    Edge[] edge;

    Graph(int nV, int nE)
    {
        V = nV;
        E = nE;
        edge = new Edge[E];
        for (int i = 0; i < E; i++)
        {
```

```
        edge[i] = new Edge();
    }
}

// class to represent edge
class Edge
{
    int src, dest;
}

// class to represent Subset
class subset
{
    int parent;
    int rank;
}

// A utility function to find
// set of an element i (uses
// path compression technique)
int find(subset [] subsets , int i)
{
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets,
                                subsets[i].parent);
    return subsets[i].parent;
}

// A function that does union
// of two sets of x and y
// (uses union by rank)
void Union(subset [] subsets,
           int x , int y )
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[yroot].rank < subsets[xroot].rank)
        subsets[yroot].parent = xroot;
    else
    {
        subsets[xroot].parent = yroot;
        subsets[yroot].rank++;
    }
}
```

```
// The main function to check whether
// a given graph contains cycle or not
int isCycle(Graph graph)
{
    int V = graph.V;
    int E = graph.E;

    subset [] subsets = new subset[V];
    for (int v = 0; v < V; v++)
    {

        subsets[v] = new subset();
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    for (int e = 0; e < E; e++)
    {
        int x = find(subsets, graph.edge[e].src);
        int y = find(subsets, graph.edge[e].dest);
        if(x == y)
            return 1;
        Union(subsets, x, y);
    }
    return 0;
}

// Driver Code
public static void main(String [] args)
{
/* Let us create the following graph
   0
   | \
   | \
   1----2 */
}

int V = 3, E = 3;
Graph graph = new Graph(V, E);

// add edge 0-1
graph.edge[0].src = 0;
graph.edge[0].dest = 1;

// add edge 1-2
graph.edge[1].src = 1;
graph.edge[1].dest = 2;

// add edge 0-2
```

```
graph.edge[2].src = 0;
graph.edge[2].dest = 2;

if (graph.isCycle(graph) == 1)
    System.out.println("Graph contains cycle");
else
    System.out.println("Graph doesn't contain cycle");
}
}

// This code is contributed
// by ashwani khemani
```

Output:

Graph contains cycle

Related Articles :

[Union-Find Algorithm Set 1 \(Detect Cycle in a an Undirected Graph\)](#)
[Disjoint Set Data Structures \(Java Implementation\)](#)
[Greedy Algorithms Set 2 \(Kruskal's Minimum Spanning Tree Algorithm\)](#)
[Job Sequencing Problem Set 2 \(Using Disjoint Set\)](#)

References:

http://en.wikipedia.org/wiki/Disjoint-set_data_structure
[IITD Video Lecture](#)

Improved By : [DeepeshThakur](#), [ashwani khemani](#)

Source

<https://www.geeksforgeeks.org/union-find-algorithm-set-2-union-by-rank/>

Chapter 68

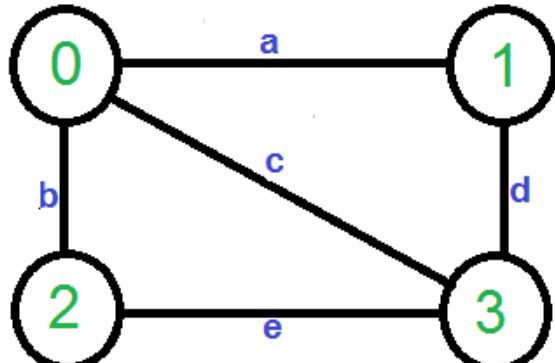
Karger's algorithm for Minimum Cut

Karger's algorithm for Minimum Cut Set 1 (Introduction and Implementation) - GeeksforGeeks

Given an undirected and unweighted graph, find the smallest cut (smallest number of edges that disconnects the graph into two components).

The input graph may have parallel edges.

For example consider the following example, the smallest cut has 2 edges.



Min-Cut for above graph is either {**a, d**} OR {**b, e**}

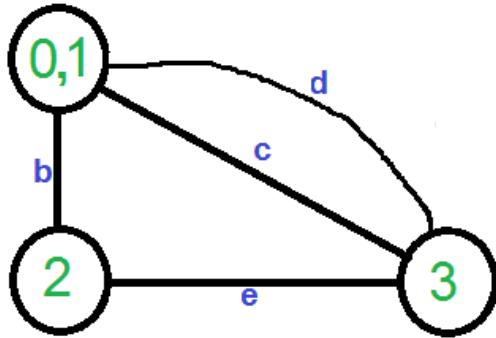
A Simple Solution use [Max-Flow based s-t cut algorithm](#) to find minimum cut. Consider every pair of vertices as source 's' and sink 't', and call minimum s-t cut algorithm to find the s-t cut. Return minimum of all s-t cuts. Best possible time complexity of this algorithm is $O(V^5)$ for a graph. [How? there are total possible V^2 pairs and s-t cut algorithm for one pair takes $O(V^*E)$ time and $E = O(V^2)$].

Below is simple Karger's Algorithm for this purpose. Below Karger's algorithm can be implemented in $O(E) = O(V^2)$ time.

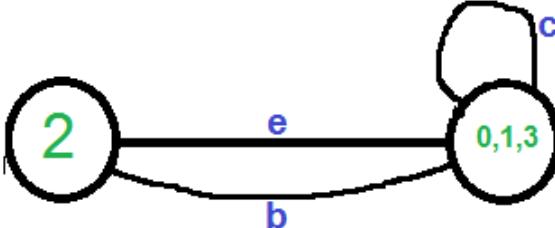
- 1) Initialize contracted graph CG as copy of original graph
- 2) While there are more than 2 vertices.
 - a) Pick a random edge (u, v) in the contracted graph.
 - b) Merge (or contract) u and v into a single vertex (update the contracted graph).
 - c) Remove self-loops
- 3) Return cut represented by two vertices.

Let us understand above algorithm through the example given.

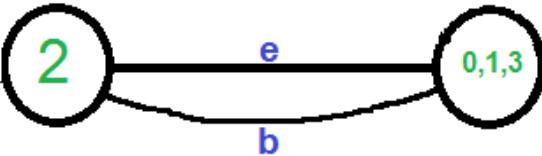
Let the first randomly picked vertex be ‘a’ which connects vertices 0 and 1. We remove this edge and contract the graph (combine vertices 0 and 1). We get the following graph.



Let the next randomly picked edge be ‘d’. We remove this edge and combine vertices (0,1) and 3.

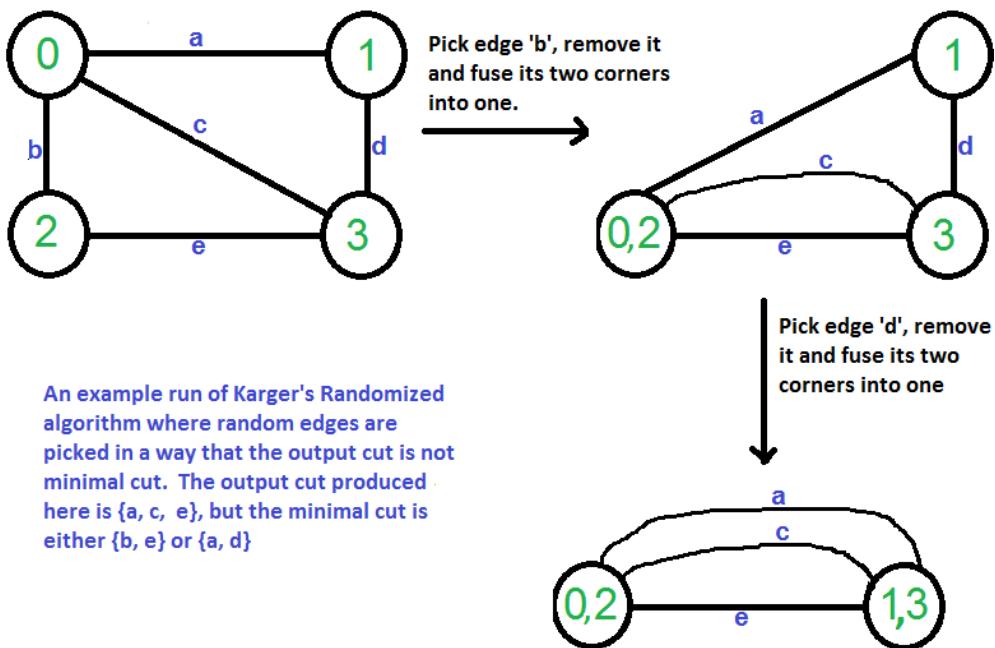


We need to remove self-loops in the graph. So we remove edge ‘c’



Now graph has two vertices, so we stop. The number of edges in the resultant graph is the cut produced by Karger’s algorithm.

Karger’s algorithm is a Monte Carlo algorithm and cut produced by it may not be minimum. For example, the following diagram shows that a different order of picking random edges produces a min-cut of size 3.



Below is C++ implementation of above algorithm. The input graph is represented as a collection of edges and [union-find data structure](#) is used to keep track of components.

```

// Karger's algorithm to find Minimum Cut in an
// undirected, unweighted and connected graph.
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// a structure to represent a unweighted edge in graph
struct Edge
{
    int src, dest;
};

// a structure to represent a connected, undirected
// and unweighted graph as a collection of edges.
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    // Since the graph is undirected, the edge
    // from src to dest is also edge from dest
    // to src. Both are counted as 1 edge here.
    Edge* edge;
};

```

```

};

// A structure to represent a subset for union-find
struct subset
{
    int parent;
    int rank;
};

// Function prototypes for union-find (These functions are defined
// after kargerMinCut() )
int find(struct subset subsets[], int i);
void Union(struct subset subsets[], int x, int y);

// A very basic implementation of Karger's randomized
// algorithm for finding the minimum cut. Please note
// that Karger's algorithm is a Monte Carlo Randomized algo
// and the cut returned by the algorithm may not be
// minimum always
int kargerMinCut(struct Graph* graph)
{
    // Get data of given graph
    int V = graph->V, E = graph->E;
    Edge *edge = graph->edge;

    // Allocate memory for creating V subsets.
    struct subset *subsets = new subset[V];

    // Create V subsets with single elements
    for (int v = 0; v < V; ++v)
    {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Initially there are V vertices in
    // contracted graph
    int vertices = V;

    // Keep contracting vertices until there are
    // 2 vertices.
    while (vertices > 2)
    {
        // Pick a random edge
        int i = rand() % E;

        // Find vertices (or sets) of two corners
        // of current edge

```

```

int subset1 = find(subsets, edge[i].src);
int subset2 = find(subsets, edge[i].dest);

// If two corners belong to same subset,
// then no point considering this edge
if (subset1 == subset2)
    continue;

// Else contract the edge (or combine the
// corners of edge into one vertex)
else
{
    printf("Contracting edge %d-%d\n",
           edge[i].src, edge[i].dest);
    vertices--;
    Union(subsets, subset1, subset2);
}
}

// Now we have two vertices (or subsets) left in
// the contracted graph, so count the edges between
// two components and return the count.
int cutedges = 0;
for (int i=0; i<E; i++)
{
    int subset1 = find(subsets, edge[i].src);
    int subset2 = find(subsets, edge[i].dest);
    if (subset1 != subset2)
        cutedges++;
}

return cutedges;
}

// A utility function to find set of an element i
// (uses path compression technique)
int find(struct subset subsets[], int i)
{
    // find root and make root as parent of i
    // (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent =
            find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// A function that does union of two sets of x and y

```

```

// (uses union by rank)
void Union(struct subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high
    // rank tree (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and
    // increment its rank by one
    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}

// Driver program to test above functions
int main()
{
    /* Let us create following unweighted graph
       0-----1
       | \     |
       |   \   |
       |     \| |
       2-----3 */
    int V = 4; // Number of vertices in graph
    int E = 5; // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
}

```

```
// add edge 0-2
graph->edge[1].src = 0;
graph->edge[1].dest = 2;

// add edge 0-3
graph->edge[2].src = 0;
graph->edge[2].dest = 3;

// add edge 1-3
graph->edge[3].src = 1;
graph->edge[3].dest = 3;

// add edge 2-3
graph->edge[4].src = 2;
graph->edge[4].dest = 3;

// Use a different seed value for every run.
srand(time(NULL));

printf("\nCut found by Karger's randomized algo is %d\n",
      kargerMinCut(graph));

return 0;
}
```

Output:

```
Contracting edge 0-2
Contracting edge 0-3

Cut found by Karger's randomized algo is 2
```

Note that the above program is based on outcome of a random function and may produce different output.

In this post, we have discussed simple Karger's algorithm and have seen that the algorithm doesn't always produce min-cut. The above algorithm produces min-cut with probability greater or equal to that $1/(n^2)$. See next post on [Analysis and Applications of Karger's Algorithm](#), applications, proof of this probability and improvements are discussed.

References:

- http://en.wikipedia.org/wiki/Karger%27s_algorithm
- <https://www.youtube.com/watch?v=P0l8jMDQTEQ>
- <https://www.cs.princeton.edu/courses/archive/fall13/cos521/lecnotes/lec2final.pdf>
- <http://web.stanford.edu/class/archive/cs/cs161/cs161.1138/lectures/11/Small11.pdf>

Source

<https://www.geeksforgeeks.org/kargers-algorithm-for-minimum-cut-set-1-introduction-and-implementation/>

Chapter 69

Karger's algorithm for Minimum Cut Set 2 (Analysis and Applications)

Karger's algorithm for Minimum Cut Set 2 (Analysis and Applications) - GeeksforGeeks

We have introduced and discussed below [Karger's algorithm](#) in set 1.

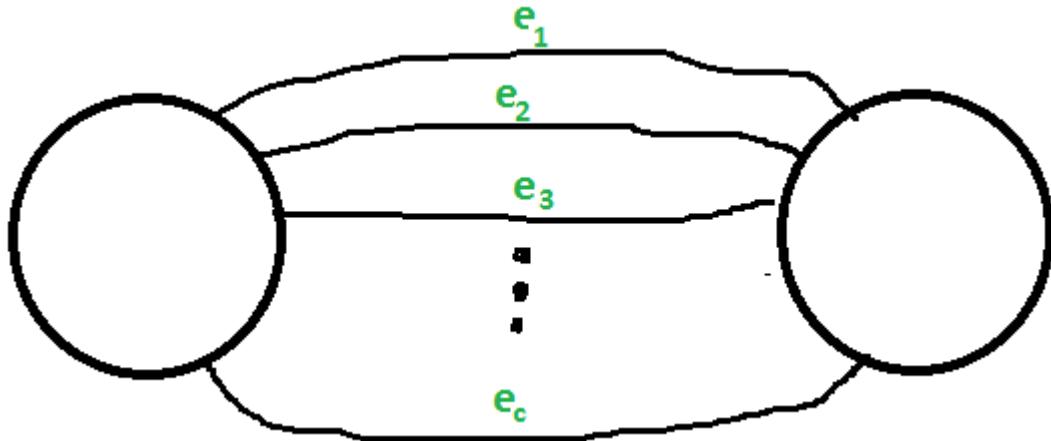
- 1) Initialize contracted graph CG as copy of original graph
- 2) While there are more than 2 vertices.
 - a) Pick a random edge (u, v) in the contracted graph.
 - b) Merge (or contract) u and v into a single vertex (update the contracted graph).
 - c) Remove self-loops
- 3) Return cut represented by two vertices.

As discussed in the previous post, [Karger's algorithm](#) doesn't always find min cut. In this post, the probability of finding min-cut is discussed.

Probability that the cut produced by Karger's Algorithm is Min-Cut is greater than or equal to $1/(n^2)$

Proof:

Let there be a unique Min-Cut of given graph and let there be C edges in the Min-Cut and the edges be $\{e_1, e_2, e_3, \dots, e_c\}$. The Karger's algorithm would produce this Min-Cut if and only if none of the edges in set $\{e_1, e_2, e_3, \dots, e_c\}$ is removed in iterations in the main while loop of above algorithm.



c is number of edges in min-cut

m is total number of edges

n is total number of vertices

S1 = Event that one of the edges in {e1, e2, e3, .. ec} is chosen in 1st iteration.

S2 = Event that one of the edges in {e1, e2, e3, .. ec} is chosen in 2nd iteration.

S3 = Event that one of the edges in {e1, e2, e3, .. ec} is chosen in 3rd iteration.

.....

The cut produced by Karger's algorithm would be a min-cut if none of the above events happen.

So the required probability is $P[S1' \cap S2' \cap S3' \cap \dots]$

Probability that a min-cut edge is chosen in first iteration:

Let us calculate $P[S1']$

$$P[S1] = c/m$$

$$P[S1'] = (1 - c/m)$$

Above value is in terms of m (or edges), let us convert it in terms of n (or vertices) using below 2 facts..

- 1) Since size of min-cut is c, degree of all vertices must be greater than or equal to c.

2) As per Handshaking Lemma, sum of degrees of all vertices = $2m$

From above two facts, we can conclude below.

$$\begin{aligned} n*c &\leq 2m \\ m &\geq nc/2 \end{aligned}$$

$$\begin{aligned} P[S_1] &\leq c / (cn/2) \\ &\leq 2/n \end{aligned}$$

$$\begin{aligned} P[S_1'] &\leq c / (cn/2) \\ &\leq 2/n \end{aligned}$$

$$P[S_1'] \geq (1 - 2/n) \quad \text{-----(1)}$$

Probability that a min-cut edge is chosen in second iteration:

$$P[S_1' \cap S_2'] = P[S_2' | S_1'] * P[S_1']$$

In the above expression, we know value of $P[S_1'] \geq (1 - 2/n)$

$P[S_2' | S_1']$ is conditional probability that is, a min cut is not chosen in second iteration given that it is not chosen in first iteration

Since there are total $(n-1)$ edges left now and number of cut edges is still c , we can replace n by $n-1$ in inequality (1). So we get.

$$P[S_2' | S_1'] \geq (1 - 2/(n-1))$$

$$P[S_1' \cap S_2'] \geq (1 - 2/n) \times (1 - 2/(n-1))$$

Probability that a min-cut edge is chosen in all iterations:

$$P[S_1' \cap S_2' \cap S_3' \cap \dots \cap S_{n-2'}]$$

$$\begin{aligned} &\geq [1 - 2/n] * [1 - 2/(n-1)] * [1 - 2/(n-2)] * [1 - 2/(n-3)] * \dots \\ &\quad \dots * [1 - 2/(n - (n-4))] * [1 - 2/(n - (n-3))] \end{aligned}$$

$$\geq [(n-2)/n] * [(n-3)/(n-1)] * [(n-4)/(n-2)] * \dots * 2/4 * 2/3$$

$$\begin{aligned} &\geq 2/(n * (n-1)) \\ &\geq 1/n^2 \end{aligned}$$

How to increase probability of success?

The above probability of success of basic algorithm is very less. For example, for a graph

with 10 nodes, the probability of finding the min-cut is greater than or equal to $1/100$. The probability can be increased by repeated runs of basic algorithm and return minimum of all cuts found.

Applications:

- 1) In war situation, a party would be interested in finding minimum number of links that break communication network of enemy.
- 2) The min-cut problem can be used to study reliability of a network (smallest number of edges that can fail).
- 3) Study of network optimization (find a maximum flow).
- 4) Clustering problems (edges like associations rules) Matching problems (an NC algorithm for min-cut in directed graphs would result in an NC algorithm for maximum matching in bipartite graphs)
- 5) Matching problems (an NC algorithm for min-cut in directed graphs would result in an NC algorithm for maximum matching in bipartite graphs)

Sources:

<https://www.youtube.com/watch?v=-UuivvyHPas>
<http://disi.unal.edu.co/~gj hernandezp/psc/lectures/02/MinCut.pdf>

Source

<https://www.geeksforgeeks.org/kargers-algorithm-for-minimum-cut-set-2-analysis-and-applications/>

Chapter 70

Hopcroft Karp Algorithm for Maximum Matching Set 1 (Introduction)

Hopcroft-Karp Algorithm for Maximum Matching Set 1 (Introduction) - GeeksforGeeks

A matching in a [Bipartite Graph](#) is a set of the edges chosen in such a way that no two edges share an endpoint. A maximum matching is a matching of maximum size (maximum number of edges). In a maximum matching, if any edge is added to it, it is no longer a matching. There can be more than one maximum matching for a given Bipartite Graph.

We have discussed importance of maximum matching and [Ford Fulkerson Based approach for maximal Bipartite Matching](#) in [previous post](#). Time complexity of the Ford Fulkerson based algorithm is $O(V \times E)$.

Hopcroft Karp algorithm is an improvement that runs in $O(\sqrt{V} \times E)$ time. Let us define few terms before we discuss the algorithm

Free Node or Vertex: Given a matching M, a node that is not part of matching is called free node. Initially all vertices are free (See first graph of below diagram). In second graph, u2 and v2 are free. In third graph, no vertex is free.

Matching and Not-Matching edges: Given a matching M, edges that are part of matching are called Matching edges and edges that are not part of M (or connect free nodes) are called Not-Matching edges. In first graph, all edges are non-matching. In second graph, (u0, v1), (u1, v0) and (u3, v3) are matching and others not-matching.

Alternating Paths: Given a matching M, an alternating path is a path in which the edges belong alternatively to the matching and not matching. All single edges paths are alternating paths. Examples of alternating paths in middle graph are u0-v1-u2 and u2-v1-u0-v2.

Augmenting path: Given a matching M, an augmenting path is an alternating path that starts from and ends on free vertices. All single edge paths that start and end with free vertices are augmenting paths. In below diagram, augmenting paths are highlighted with blue color. Note that the augmenting path always has one extra matching edge.

The Hopcroft Karp algorithm is based on below concept.

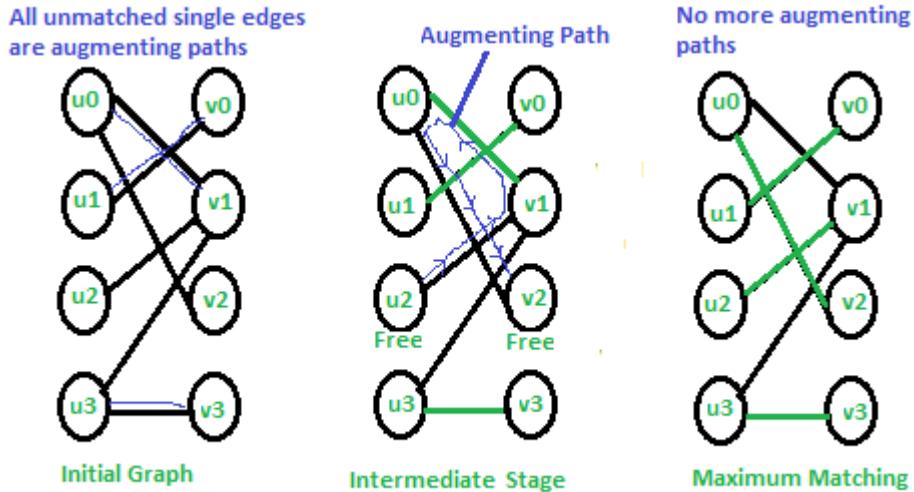
A matching M is not maximum if there exists an augmenting path. It is also true other way, i.e, a matching is maximum if no augmenting path exists

So the idea is to one by one look for augmenting paths. And add the found paths to current matching.

Hopcroft Karp Algorithm

- 1) Initialize Maximal Matching M as empty.
- 2) While there exists an Augmenting Path p
 - Remove matching edges of p from M and add not-matching edges of p to M
(This increases size of M by 1 as p starts and ends with a free vertex)
- 3) Return M .

Below diagram shows working of the algorithm.



In the initial graph all single edges are augmenting paths and we can pick in any order. In the middle stage, there is only one augmenting path. We remove matching edges of this path and add not-matching edges. In final matching, there are no augmenting paths so the matching is maximum.

Implementation of Hopcroft Karp algorithm is discussed in set 2.

Hopcroft–Karp Algorithm for Maximum Matching Set 2 (Implementation)

References:

- https://en.wikipedia.org/wiki/Hopcroft%E2%80%93Karp_algorithm
- <http://www.dis.uniroma1.it/~leon/tcs/lecture2.pdf>

Source

<https://www.geeksforgeeks.org/hopcroft-karp-algorithm-for-maximum-matching-set-1-introduction/>

Chapter 71

Hopcroft-Karp Algorithm for Maximum Matching Set 2 (Implementation)

Hopcroft-Karp Algorithm for Maximum Matching Set 2 (Implementation) - GeeksforGeeks

We strongly recommend to refer below post as a prerequisite.

[Hopcroft-Karp Algorithm for Maximum Matching Set 1 \(Introduction\)](#)

There are few important things to note before we start implementation.

1. We need to **find an augmenting path** (A path that alternates between matching and not matching edges, and has free vertices as starting and ending points).
2. Once we find alternating path, we need to **add the found path to existing Matching**. Here adding path means, making previous matching edges on this path as not-matching and previous not-matching edges as matching.

The idea is to use BFS (Breadth First Search) to find augmenting paths. Since BFS traverses level by level, it is used to divide the graph in layers of matching and not matching edges. A dummy vertex NIL is added that is connected to all vertices on left side and all vertices on right side. Following arrays are used to find augmenting path. Distance to NIL is initialized as INF (infinite). If we start from dummy vertex and come back to it using alternating path of distinct vertices, then there is an augmenting path.

1. pairU[]: An array of size m+1 where m is number of vertices on left side of Bipartite Graph. pairU[u] stores pair of u on right side if u is matched and NIL otherwise.
2. pairV[]: An array of size n+1 where n is number of vertices on right side of Bipartite Graph. pairV[v] stores pair of v on left side if v is matched and NIL otherwise.

3. dist[]: An array of size m+1 where m is number of vertices on left side of Bipartite Graph. dist[u] is initialized as 0 if u is not matching and INF (infinite) otherwise. dist[] of NIL is also initialized as INF

Once an augmenting path is found, DFS (Depth First Search) is used to add augmenting paths to current matching. DFS simply follows the distance array setup by BFS. It fills values in pairU[u] and pairV[v] if v is next to u in BFS.

Below is C++ implementation of above Hopcroft Karp algorithm.

```
// C++ implementation of Hopcroft Karp algorithm for
// maximum matching
#include<bits/stdc++.h>
using namespace std;
#define NIL 0
#define INF INT_MAX

// A class to represent Bipartite graph for Hopcroft
// Karp implementation
class BipGraph
{
    // m and n are number of vertices on left
    // and right sides of Bipartite Graph
    int m, n;

    // adj[u] stores adjacents of left side
    // vertex 'u'. The value of u ranges from 1 to m.
    // 0 is used for dummy vertex
    list<int> *adj;

    // These are basically pointers to arrays needed
    // for hopcroftKarp()
    int *pairU, *pairV, *dist;

public:
    BipGraph(int m, int n); // Constructor
    void addEdge(int u, int v); // To add edge

    // Returns true if there is an augmenting path
    bool bfs();

    // Adds augmenting path if there is one beginning
    // with u
    bool dfs(int u);

    // Returns size of maximum matcing
    int hopcroftKarp();
};

};
```

```
// Returns size of maximum matching
int BipGraph::hopcroftKarp()
{
    // pairU[u] stores pair of u in matching where u
    // is a vertex on left side of Bipartite Graph.
    // If u doesn't have any pair, then pairU[u] is NIL
    pairU = new int[m+1];

    // pairV[v] stores pair of v in matching. If v
    // doesn't have any pair, then pairU[v] is NIL
    pairV = new int[n+1];

    // dist[u] stores distance of left side vertices
    // dist[u] is one more than dist[u'] if u is next
    // to u' in augmenting path
    dist = new int[m+1];

    // Initialize NIL as pair of all vertices
    for (int u=0; u<m; u++)
        pairU[u] = NIL;
    for (int v=0; v<n; v++)
        pairV[v] = NIL;

    // Initialize result
    int result = 0;

    // Keep updating the result while there is an
    // augmenting path.
    while (bfs())
    {
        // Find a free vertex
        for (int u=1; u<=m; u++)

            // If current vertex is free and there is
            // an augmenting path from current vertex
            if (pairU[u]==NIL && dfs(u))
                result++;

    }
    return result;
}

// Returns true if there is an augmenting path, else returns
// false
bool BipGraph::bfs()
{
    queue<int> Q; //an integer queue
```

```
// First layer of vertices (set distance as 0)
for (int u=1; u<=m; u++)
{
    // If this is a free vertex, add it to queue
    if (pairU[u]==NIL)
    {
        // u is not matched
        dist[u] = 0;
        Q.push(u);
    }

    // Else set distance as infinite so that this vertex
    // is considered next time
    else dist[u] = INF;
}

// Initialize distance to NIL as infinite
dist[NIL] = INF;

// Q is going to contain vertices of left side only.
while (!Q.empty())
{
    // Dequeue a vertex
    int u = Q.front();
    Q.pop();

    // If this node is not NIL and can provide a shorter path to NIL
    if (dist[u] < dist[NIL])
    {
        // Get all adjacent vertices of the dequeued vertex u
        list<int>::iterator i;
        for (i=adj[u].begin(); i!=adj[u].end(); ++i)
        {
            int v = *i;

            // If pair of v is not considered so far
            // (v, pairV[v]) is not yet explored edge.
            if (dist[pairV[v]] == INF)
            {
                // Consider the pair and add it to queue
                dist[pairV[v]] = dist[u] + 1;
                Q.push(pairV[v]);
            }
        }
    }
}

// If we could come back to NIL using alternating path of distinct
```

```
// vertices then there is an augmenting path
return (dist[NIL] != INF);
}

// Returns true if there is an augmenting path beginning with free vertex u
bool BipGraph::dfs(int u)
{
    if (u != NIL)
    {
        list<int>::iterator i;
        for (i=adj[u].begin(); i!=adj[u].end(); ++i)
        {
            // Adjacent to u
            int v = *i;

            // Follow the distances set by BFS
            if (dist[pairV[v]] == dist[u]+1)
            {
                // If dfs for pair of v also returns
                // true
                if (dfs(pairV[v]) == true)
                {
                    pairV[v] = u;
                    pairU[u] = v;
                    return true;
                }
            }
        }

        // If there is no augmenting path beginning with u.
        dist[u] = INF;
        return false;
    }
    return true;
}

// Constructor
BipGraph::BipGraph(int m, int n)
{
    this->m = m;
    this->n = n;
    adj = new list<int>[m+1];
}

// To add edge from u to v and v to u
void BipGraph::addEdge(int u, int v)
{
    adj[u].push_back(v); // Add u to v's list.
```

```
adj[v].push_back(u); // Add u to v's list.  
}  
  
// Driver Program  
int main()  
{  
    BipGraph g(4, 4);  
    g.addEdge(1, 2);  
    g.addEdge(1, 3);  
    g.addEdge(2, 1);  
    g.addEdge(3, 2);  
    g.addEdge(4, 2);  
    g.addEdge(4, 4);  
  
    cout << "Size of maximum matching is " << g.hopcroftKarp();  
  
    return 0;  
}
```

Output:

```
Size of maximum matching is 4
```

The above implementation is mainly adopted from the algorithm provided on Wiki page of [Hopcroft Karp algorithm](#).

This article is contributed by **Rahul Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Improved By : [am10](#)

Source

<https://www.geeksforgeeks.org/hopcroft-karp-algorithm-for-maximum-matching-set-2-implementation/>

Chapter 72

0-1 BFS (Shortest Path in a Binary Weight Graph)

0-1 BFS (Shortest Path in a Binary Weight Graph) - GeeksforGeeks

Given a graph where every edge has weight as either 0 or 1. A source vertex is also given in the graph. Find the shortest path from source vertex to every other vertex.

Input : Source Vertex = 0 and below graph

Output : Shortest distances from given source
0 0 1 1 2 1 2 1 2

Explanation :

Shortest distance from 0 to 0 is 0
Shortest distance from 0 to 1 is 0
Shortest distance from 0 to 2 is 1
.....

In normal [BFS](#) of a graph all edges have equal weight but in 0-1 BFS some edges may have 0 weight and some may have 1 weight. In this we will not use bool array to mark visited nodes but at each step we will check for the optimal distance condition. We use [double ended queue](#) to store the node. While performing BFS if a edge having weight = 0 is found node is pushed at front of double ended queue and if a edge having weight = 1 is found, it is pushed at back of double ended queue.

The approach is similar to [Dijkstra](#) that the if the shortest distance to node is relaxed by the previous node then only it will be pushed in the queue.

The above idea works in all cases, when pop a vertex (like Dijkstra), it is the minimum weight vertex among remaining vertices. If there is a 0 weight vertex adjacent to it, then this

adjacent has same distance. If there is a 1 weight adjacent, then this adjacent has maximum distance among all vertices in dequeue (because all other vertices are either adjacent of currently popped vertex or adjacent of previously popped vertices).

Below is C++ implementation of the above idea.

```

// C++ program to implement single source
// shortest path for a Binary Graph
#include<bits/stdc++.h>
using namespace std;

/* no.of vertices */
#define V 9

// a structure to represent edges
struct node
{
    // two variable one denote the node
    // and other the weight
    int to, weight;
};

// vector to store edges
vector <node> edges[V];

// Prints shortest distance from given source to
// every other vertex
void zeroOneBFS(int src)
{
    // Initialize distances from given source
    int dist[V];
    for (int i=0; i<V; i++)
        dist[i] = INT_MAX;

    // double ende queue to do BFS.
    deque <int> Q;
    dist[src] = 0;
    Q.push_back(src);

    while (!Q.empty())
    {
        int v = Q.front();
        Q.pop_front();

        for (int i=0; i<edges[v].size(); i++)
        {
            // checking for the optimal distance
            if (dist[edges[v][i].to] > dist[v] + edges[v][i].weight)
            {

```

```

        dist[edges[v][i].to] = dist[v] + edges[v][i].weight;

        // Put 0 weight edges to front and 1 weight
        // edges to back so that vertices are processed
        // in increasing order of weights.
        if (edges[v][i].weight == 0)
            Q.push_front(edges[v][i].to);
        else
            Q.push_back(edges[v][i].to);
    }
}

// printing the shortest distances
for (int i=0; i<V; i++)
    cout << dist[i] << " ";
}

void addEdge(int u, int v, int wt)
{
    edges[u].push_back({v, wt});
    edges[v].push_back({u, wt});
}

// Driver function
int main()
{
    addEdge(0, 1, 0);
    addEdge(0, 7, 1);
    addEdge(1, 7, 1);
    addEdge(1, 2, 1);
    addEdge(2, 3, 0);
    addEdge(2, 5, 0);
    addEdge(2, 8, 1);
    addEdge(3, 4, 1);
    addEdge(3, 5, 1);
    addEdge(4, 5, 1);
    addEdge(5, 6, 1);
    addEdge(6, 7, 1);
    addEdge(7, 8, 1);
    int src = 0;//source node
    zeroOneBFS(src);
    return 0;
}

```

Output:

0 0 1 1 2 1 2 1 2

This problem can also be solved by Dijkstra but the time complexity will be $O(E + V \log V)$ whereas by BFS it will be $O(V+E)$.

Reference :

<http://codeforces.com/blog/entry/22276>

Source

<https://www.geeksforgeeks.org/0-1-bfs-shortest-path-binary-graph/>

Chapter 73

2-Satisfiability (2-SAT) Problem

2-Satisfiability (2-SAT) Problem - GeeksforGeeks

Boolean Satisfiability or simply **SAT** is the problem of determining if a Boolean formula is satisfiable or unsatisfiable.

- **Satisfiable** : If the Boolean variables can be assigned values such that the formula turns out to be TRUE, then we say that the formula is satisfiable.
- **Unsatisfiable** : If it is not possible to assign such values, then we say that the formula is unsatisfiable.

Examples:

- $\frac{A \wedge \neg A}{F} \vee B$, is satisfiable, because $A = \text{TRUE}$ and $B = \text{FALSE}$ makes $F = \text{TRUE}$.
- $\frac{G \equiv A \wedge \neg A}{F}$, is unsatisfiable, because:

A	$\neg A$	G
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE

Note : Boolean satisfiability problem is **NP-complete** (For proof, refer [Cook's Theorem](#)).

2-SAT is a special case of Boolean Satisfiability Problem and can be solved in polynomial time.

To understand this better, first let us see what is Conjunctive Normal Form (CNF) or also known as Product of Sums (POS).

CNF : CNF is a conjunction (AND) of clauses, where every clause is a disjunction (OR).

Now, 2-SAT limits the problem of SAT to only those Boolean formula which are expressed as a CNF with every clause having only **2 terms**(also called **2-CNF**).

Example: $\Phi = (A_1 \vee B_1) \wedge (A_2 \vee B_2) \wedge (A_3 \vee B_3) \wedge \dots \wedge (A_m \vee B_m)$

Thus, Problem of 2-Satisfiability can be stated as:

Given CNF with each clause having only 2 terms, is it possible to assign such values to the variables so that the CNF is TRUE?

Examples:

Input :

Output : The given expression is satisfiable.
(for $x_1 = \text{FALSE}$, $x_2 = \text{TRUE}$)

Input :

Output : The given expression is unsatisfiable.
(for all possible combinations of x_1 and x_2)

For the CNF value to come TRUE, value of every clause should be TRUE. Let one of the

clause be $(A \vee B)$.

$$(A \vee B) = \text{TRUE}$$

- If $A = 0$, B must be 1 i.e. $(0 \vee 1) = \text{TRUE}$

- If $B = 0$, A must be 1 i.e. $(1 \vee 0) = \text{TRUE}$

Thus,

= TRUE is equivalent to

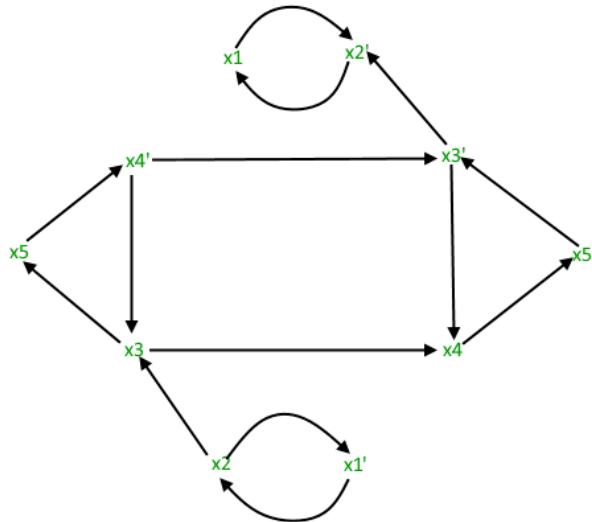
Now, we can express the CNF as an Implication. So, we create an Implication Graph which has 2 edges for every clause of the CNF.

$(A \vee B)$ is expressed in Implication Graph as $\text{edge}(A \rightarrow B) \text{ and } (B \rightarrow A)$
Thus, for a Boolean formula with 'm' clauses, we make an Implication Graph with:

- 2 edges for every clause i.e. ‘2m’ edges.
- 1 node for every Boolean variable involved in the Boolean formula.

Let's see one example of Implication Graph.

$$F = (x_1 \vee x_2) \wedge (\bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2) \wedge (x_3 \vee x_4) \wedge (\bar{x}_3 \vee x_5)$$



Note: The implication (if A then B) is equivalent to its contrapositive (if \bar{B} then \bar{A}).

Now, consider the following cases:

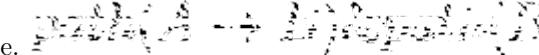
CASE 1: If x_1 exists in the graph
 This means
 If $x_1 = \text{TRUE}$, $x_2 = \text{TRUE}$, which is a contradiction.
 But if $x_1 = \text{FALSE}$, there are no implication constraints.
 Thus, $x_1 = \text{FALSE}$

CASE 2: If \bar{x}_1 exists in the graph
 This means
 If $\bar{x}_1 = \text{TRUE}$, $x_2 = \text{TRUE}$, which is a contradiction.
 But if $\bar{x}_1 = \text{FALSE}$, there are no implication constraints.
 Thus, $\bar{x}_1 = \text{FALSE}$ i.e. $x_1 = \text{TRUE}$

CASE 3: If both exist in the graph

One edge requires X to be TRUE and the other one requires X to be FALSE.

Thus, there is no possible assignment in such a case.

CONCLUSION: If any two variables X and \bar{X} are on a cycle i.e.  both exists, then the CNF is unsatisfiable. Otherwise, there is a possible assignment and the CNF is satisfiable.

Note here that, we use path due to the following property of implication:

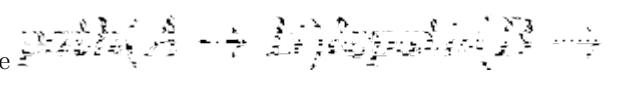
If we have 

Thus, if we have a path in the Implication Graph, that is pretty much same as having a direct edge.

CONCLUSION FROM IMPLEMENTATION POINT OF VIEW:

If both X and \bar{X} lie in the same SCC (Strongly Connected Component), the CNF is unsatisfiable.

A Strongly Connected Component of a directed graph has nodes such that every node can be reached from every another node in that SCC.

Now, if X and \bar{X} lie on the same SCC, we will definitely have  present and hence the conclusion.

Checking of the SCC can be done in $O(E+V)$ using the [Kosaraju's Algorithm](#)

```
// C++ implementation to find if the given
// expression is satisfiable using the
// Kosaraju's Algorithm
#include <bits/stdc++.h>
using namespace std;

const int MAX = 100000;

// data structures used to implement Kosaraju's
// Algorithm. Please refer
// http://www.geeksforgeeks.org/strongly-connected-components/
vector<int> adj[MAX];
vector<int> adjInv[MAX];
bool visited[MAX];
bool visitedInv[MAX];
stack<int> s;

// this array will store the SCC that the
// particular node belongs to
int scc[MAX];

// counter maintains the number of the SCC
int counter = 1;
```

```

// adds edges to form the original graph
void addEdges(int a, int b)
{
    adj[a].push_back(b);
}

// add edges to form the inverse graph
void addEdgesInverse(int a, int b)
{
    adjInv[b].push_back(a);
}

// for STEP 1 of Kosaraju's Algorithm
void dfsFirst(int u)
{
    if(visited[u])
        return;

    visited[u] = 1;

    for (int i=0;i<adj[u].size();i++)
        dfsFirst(adj[u][i]);

    s.push(u);
}

// for STEP 2 of Kosaraju's Algorithm
void dfsSecond(int u)
{
    if(visitedInv[u])
        return;

    visitedInv[u] = 1;

    for (int i=0;i<adjInv[u].size();i++)
        dfsSecond(adjInv[u][i]);

    scc[u] = counter;
}

// function to check 2-Satisfiability
void is2Satisfiable(int n, int m, int a[], int b[])
{
    // adding edges to the graph
    for(int i=0;i<m;i++)
    {
        // variable x is mapped to x

```

```

// variable -x is mapped to n+x = n-(-x)

// for a[i] or b[i], addEdges -a[i] -> b[i]
// AND -b[i] -> a[i]
if (a[i]>0 && b[i]>0)
{
    addEdges(a[i]+n, b[i]);
    addEdgesInverse(a[i]+n, b[i]);
    addEdges(b[i]+n, a[i]);
    addEdgesInverse(b[i]+n, a[i]);
}

else if (a[i]>0 && b[i]<0)
{
    addEdges(a[i]+n, n-b[i]);
    addEdgesInverse(a[i]+n, n-b[i]);
    addEdges(-b[i], a[i]);
    addEdgesInverse(-b[i], a[i]);
}

else if (a[i]<0 && b[i]>0)
{
    addEdges(-a[i], b[i]);
    addEdgesInverse(-a[i], b[i]);
    addEdges(b[i]+n, n-a[i]);
    addEdgesInverse(b[i]+n, n-a[i]);
}

else
{
    addEdges(-a[i], n-b[i]);
    addEdgesInverse(-a[i], n-b[i]);
    addEdges(-b[i], n-a[i]);
    addEdgesInverse(-b[i], n-a[i]);
}
}

// STEP 1 of Kosaraju's Algorithm which
// traverses the original graph
for (int i=1;i<=2*n;i++)
    if (!visited[i])
        dfsFirst(i);

// STEP 2 pf Kosaraju's Algorithm which
// traverses the inverse graph. After this,
// array scc[] stores the corresponding value
while (!s.empty())
{

```

```

int n = s.top();
s.pop();

if (!visitedInv[n])
{
    dfsSecond(n);
    counter++;
}
}

for (int i=1;i<=n;i++)
{
    // for any 2 variable x and -x lie in
    // same SCC
    if(scc[i]==scc[i+n])
    {
        cout << "The given expression "
            "is unsatisfiable." << endl;
        return;
    }
}

// no such variables x and -x exist which lie
// in same SCC
cout << "The given expression is satisfiable."
    << endl;
return;
}

// Driver function to test above functions
int main()
{
    // n is the number of variables
    // 2n is the total number of nodes
    // m is the number of clauses
    int n = 5, m = 7;

    // each clause is of the form a or b
    // for m clauses, we have a[m], b[m]
    // representing a[i] or b[i]

    // Note:
    // 1 <= x <= N for an uncomplemented variable x
    // -N <= x <= -1 for a complemented variable x
    // -x is the complement of a variable x

    // The CNF being handled is:
    // '+' implies 'OR' and '*' implies 'AND'
}

```

```
// (x1+x2)*(x2'+x3)*(x1'+x2')*(x3+x4)*(x3'+x5)*  
// (x4'+x5')*(x3'+x4)  
int a[] = {1, -2, -1, 3, -3, -4, -3};  
int b[] = {2, 3, -2, 4, 5, -5, 4};  
  
// We have considered the same example for which  
// Implication Graph was made  
is2Satisfiable(n, m, a, b);  
  
return 0;  
}
```

Output:

The given expression is satisfiable.

More Test Cases:

```
Input : n = 2, m = 3  
        a[] = {1, 2, -1}  
        b[] = {2, -1, -2}  
Output : The given expression is satisfiable.
```

```
Input : n = 2, m = 4  
        a[] = {1, -1, 1, -1}  
        b[] = {2, 2, -2, -2}  
Output : The given expression is unsatisfiable.
```

Source

<https://www.geeksforgeeks.org/2-satisfiability-2-sat-problem/>

Chapter 74

A matrix probability question

A matrix probability question - GeeksforGeeks

Given a rectangular matrix, we can move from current cell in 4 directions with equal probability. The 4 directions are right, left, top or bottom. Calculate the Probability that after N moves from a given position (i, j) in the matrix, we will not cross boundaries of the matrix at any point.

We strongly recommend you to minimize your browser and try this yourself first.

The idea is to perform something similar to DFS. We recursively traverse in each of the 4 allowed direction and for each cell encountered, we calculate the required probability with one less move. As each direction has equal probability, each direction will contribute to 1/4 of total probability of that cell i.e. 0.25. We return 0 if we step outside the matrix and return 1 if N steps are completed without crossing matrix boundaries.

Below is the implementation of above idea :

C++

```
/// C++ program to find the probability
// that we do not cross boundary of a
// matrix after N moves.
#include <iostream>
using namespace std;

// check if (x, y) is valid matrix coordinate
bool isSafe(int x, int y, int m, int n)
{
    return (x >= 0 && x < m &&
            y >= 0 && y < n);
}

// Function to calculate probability
// that after N moves from a given
```

```
// position (x, y) in m x n matrix,
// boundaries of the matrix will not be crossed.
double findProbability(int m, int n, int x,
                      int y, int N)
{
    // boundary crossed
    if (!isSafe(x, y, m, n))
        return 0.0;

    // N steps taken
    if (N == 0)
        return 1.0;

    // Initialize result
    double prob = 0.0;

    // move up
    prob += findProbability(m, n, x - 1,
                           y, N - 1) * 0.25;

    // move right
    prob += findProbability(m, n, x,
                           y + 1, N - 1) * 0.25;

    // move down
    prob += findProbability(m, n, x + 1,
                           y, N - 1) * 0.25;

    // move left
    prob += findProbability(m, n, x,
                           y - 1, N - 1) * 0.25;

    return prob;
}

// Driver code
int main()
{
    // matrix size
    int m = 5, n = 5;

    // coordinates of starting point
    int i = 1, j = 1;

    // Number of steps
    int N = 2;

    cout << "Probability is "
```

```
<< findProbability(m, n, i, j, N);

return 0;
}
```

Java

```
// Java program to find the probability
// that we do not cross boundary
// of a matrix after N moves.
import java.io.*;

class GFG {

    // check if (x, y) is valid
    // matrix coordinate
    static boolean isSafe(int x, int y,
                          int m, int n)
    {
        return (x >= 0 && x < m &&
                y >= 0 && y < n);
    }

    // Function to calculate probability
    // that after N moves from a given
    // position (x, y) in m x n matrix,
    // boundaries of the matrix will
    // not be crossed.
    static double findProbability(int m, int n,
                                 int x, int y,
                                 int N)
    {

        // boundary crossed
        if (! isSafe(x, y, m, n))
            return 0.0;

        // N steps taken
        if (N == 0)
            return 1.0;

        // Initialize result
        double prob = 0.0;

        // move up
        prob += findProbability(m, n, x - 1,
                               y, N - 1) * 0.25;
```

```
// move right
prob += findProbability(m, n, x, y + 1,
                        N - 1) * 0.25;

// move down
prob += findProbability(m, n, x + 1,
                        y, N - 1) * 0.25;

// move left
prob += findProbability(m, n, x, y - 1,
                        N - 1) * 0.25;

return prob;
}

// Driver code
public static void main (String[] args)
{
    // matrix size
    int m = 5, n = 5;

    // coordinates of starting point
    int i = 1, j = 1;

    // Number of steps
    int N = 2;

    System.out.println("Probability is " +
                       findProbability(m, n, i,
                                         j, N));

}
}

// This code is contributed by KRV.

C#

// C# program to find the probability
// that we do not cross boundary
// of a matrix after N moves.
using System;

class GFG
{

    // check if (x, y) is valid
```

```
// matrix coordinate
static bool isSafe(int x, int y,
                    int m, int n)
{
    return (x >= 0 && x < m &&
            y >= 0 && y < n);
}

// Function to calculate probability
// that after N moves from a given
// position (x, y) in m x n matrix,
// boundaries of the matrix will
// not be crossed.
static double findProbability(int m, int n,
                               int x, int y,
                               int N)
{
    // boundary crossed
    if (! isSafe(x, y, m, n))
        return 0.0;

    // N steps taken
    if (N == 0)
        return 1.0;

    // Initialize result
    double prob = 0.0;

    // move up
    prob += findProbability(m, n, x - 1,
                            y, N - 1) * 0.25;

    // move right
    prob += findProbability(m, n, x, y + 1,
                            N - 1) * 0.25;

    // move down
    prob += findProbability(m, n, x + 1,
                            y, N - 1) * 0.25;

    // move left
    prob += findProbability(m, n, x, y - 1,
                            N - 1) * 0.25;

    return prob;
}
```

```
// Driver code
public static void Main ()
{
    // matrix size
    int m = 5, n = 5;

    // coordinates of starting point
    int i = 1, j = 1;

    // Number of steps
    int N = 2;

    Console.WriteLine("Probability is " +
                      findProbability(m, n, i,
                                      j, N));

}

}

// This code is contributed by nitin mittal.
```

PHP

```
<?php
// PHP program to find the probability
// that we do not cross boundary of a
// matrix after N moves.

// check if (x, y) is valid
// matrix coordinate
function isSafe($x, $y, $m, $n)
{
    return ($x >= 0 && $x < $m &&
            $y >= 0 && $y < $n);
}

// Function to calculate probability
// that after N moves from a given
// position (x, y) in m x n matrix,
// boundaries of the matrix will
// not be crossed.
function findProbability($m, $n, $x,
                           $y, $N)
{
    // boundary crossed
    if (!isSafe($x, $y, $m, $n))
        return 0.0;
```

```
// N steps taken
if ($N == 0)
    return 1.0;

// Initialize result
$prob = 0.0;

// move up
$prob += findProbability($m, $n, $x - 1,
                         $y, $N - 1) * 0.25;

// move right
$prob += findProbability($m, $n, $x,
                         $y + 1, $N - 1) * 0.25;

// move down
$prob += findProbability($m, $n, $x + 1,
                         $y, $N - 1) * 0.25;

// move left
$prob += findProbability($m, $n, $x,
                         $y - 1, $N - 1) * 0.25;

return $prob;
}

// Driver code

// matrix size
$m = 5; $n = 5;

// coordinates of starting point
$i = 1; $j = 1;

// Number of steps
$N = 2;

echo "Probability is ",
     findProbability($m, $n, $i, $j, $N);

// This code is contributed by nitin mittal.
?>
```

Output :

```
Probability is 0.875
```

This article is contributed by **Aditya Goel**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Improved By : [KRV](#), [nitin mittal](#)

Source

<https://www.geeksforgeeks.org/a-matrix-probability-question/>

Chapter 75

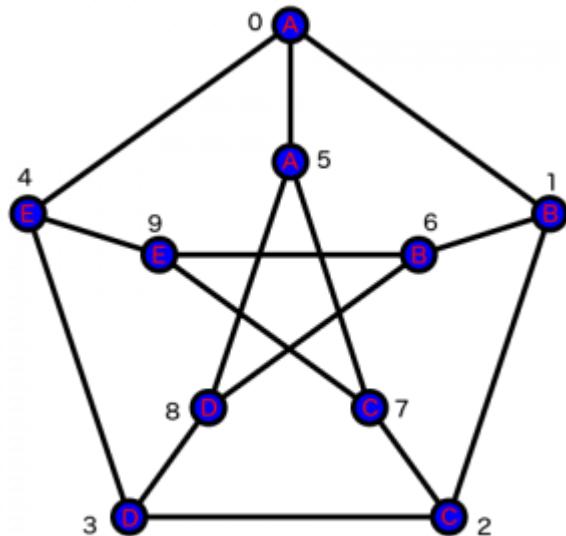
A Peterson Graph Problem

A Peterson Graph Problem - GeeksforGeeks

The following graph G is called a Petersen graph and its vertices have been numbered from 0 to 9. Some letters have also been assigned to vertices of G, as can be seen from the following picture:

Let's consider a walk W in graph G, which consists of L vertices W₁, W₂, ..., W_L. A string S of L letters 'A' – 'E' is realized by walk W if the sequence of letters written along W is equal to S. Vertices can be visited multiple times while walking along W.

For example, S = 'ABBECCD' is realized by W = (0, 1, 6, 9, 7, 2, 3). Determine whether there is a walk W which realizes a given string S in graph G, and if so then find the lexicographically least such walk. The only line of input contains one string S. If there is no walk W which realizes S, then output -1 otherwise, you should output the least lexicographical walk W which realizes S.



Examples:

```
Input : s = 'ABB'
Output: 016
Explanation: As we can see in the graph
             the path from ABB is 016.

Input : s = 'AABE'
Output :-1
Explanation: As there is no path that
             exists, hence output is -1.
```

We apply breadth first search to visit each vertex of the graph.

```
// C++ program to find the
// path in Peterson graph
#include <bits/stdc++.h>
using namespace std;

// path to be checked
char S[100005];

// adjacency matrix.
bool adj[10][10];

// resulted path - way
char result[100005];

// we are applying breadth first search
// here
bool findthepath(char* S, int v)
{
    result[0] = v + '0';
    for (int i = 1; S[i]; i++) {

        // first traverse the outer graph
        if (adj[v][S[i] - 'A'] || adj[S[i] -
                                         'A'][v]) {
            v = S[i] - 'A';
        }

        // then traverse the inner graph
        else if (adj[v][S[i] - 'A' + 5] ||
                 adj[S[i] - 'A' + 5][v]) {
            v = S[i] - 'A' + 5;
        }
    }
}
```

```
// if the condition failed to satisfy
// return false
else
    return false;

result[i] = v + '0';
}

return true;
}

// driver code
int main()
{
    // here we have used adjacency matrix to make
    // connections between the connected nodes
    adj[0][1] = adj[1][2] = adj[2][3] = adj[3][4] =
    adj[4][0] = adj[0][5] = adj[1][6] = adj[2][7] =
    adj[3][8] = adj[4][9] = adj[5][7] = adj[7][9] =
    adj[9][6] = adj[6][8] = adj[8][5] = true;

    // path to be checked
    char S[] = "ABB";

    if (findthepath(S, S[0] - 'A') ||
        findthepath(S, S[0] - 'A' + 5)) {
        cout << result;
    } else {
        cout << "-1";
    }
    return 0;
}
```

Output:

016

Source

<https://www.geeksforgeeks.org/peterson-graph/>

Chapter 76

All Topological Sorts of a Directed Acyclic Graph

All Topological Sorts of a Directed Acyclic Graph - GeeksforGeeks

Topological sorting for **Directed Acyclic Graph** (DAG) is a linear ordering of vertices such that for every directed edge uv , vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

Given a DAG, print all topological sorts of the graph.

For example, consider the below graph.

All topological sorts of the given graph are:

```
4 5 0 2 3 1  
4 5 2 0 3 1  
4 5 2 3 0 1  
4 5 2 3 1 0  
5 2 3 4 0 1  
5 2 3 4 1 0  
5 2 4 0 3 1  
5 2 4 3 0 1  
5 2 4 3 1 0  
5 4 0 2 3 1  
5 4 2 0 3 1  
5 4 2 3 0 1  
5 4 2 3 1 0
```

In a Directed acyclic graph many a times we can have vertices which are unrelated to each other because of which we can order them in many ways. These various topological sorting

is important in many cases, for example if some relative weight is also available between the vertices, which is to minimize then we need to take care of relative ordering as well as their relative weight, which creates the need of checking through all possible topological ordering. We can go through all possible ordering via backtracking , the algorithm step are as follows :

1. Initialize all vertices as unvisited.
2. Now choose vertex which is unvisited and has zero indegree and decrease indegree of all those vertices by 1 (corresponding to removing edges) now add this vertex to result and call the recursive function again and backtrack.
3. After returning from function reset values of visited, result and indegree for enumeration of other possibilities.

Below is implementation of above steps.

C++

```
// C++ program to print all topological sorts of a graph
#include <bits/stdc++.h>
using namespace std;

class Graph
{
    int V;      // No. of vertices

    // Pointer to an array containing adjacency list
    list<int> *adj;

    // Vector to store indegree of vertices
    vector<int> indegree;

    // A function used by alltopologicalSort
    void alltopologicalSortUtil(vector<int>& res,
                                bool visited[]);

public:
    Graph(int V);    // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // Prints all Topological Sorts
    void alltopologicalSort();
};

// Constructor of graph
```

```

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];

    // Initialising all indegree with 0
    for (int i = 0; i < V; i++)
        indegree.push_back(0);
}

// Utility function to add edge
void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.

    // increasing inner degree of w by 1
    indegree[w]++;
}

// Main recursive function to print all possible
// topological sorts
void Graph::alltopologicalSortUtil(vector<int>& res,
                                    bool visited[])
{
    // To indicate whether all topological are found
    // or not
    bool flag = false;

    for (int i = 0; i < V; i++)
    {
        // If indegree is 0 and not yet visited then
        // only choose that vertex
        if (indegree[i] == 0 && !visited[i])
        {
            // reducing indegree of adjacent vertices
            list<int>:: iterator j;
            for (j = adj[i].begin(); j != adj[i].end(); j++)
                indegree[*j]--;
        }

        // including in result
        res.push_back(i);
        visited[i] = true;
        alltopologicalSortUtil(res, visited);

        // resetting visited, res and indegree for
        // backtracking
        visited[i] = false;
        res.erase(res.end() - 1);
    }
}

```

```
        for (j = adj[i].begin(); j != adj[i].end(); j++)
            indegree[*j]++;

        flag = true;
    }
}

// We reach here if all vertices are visited.
// So we print the solution here
if (!flag)
{
    for (int i = 0; i < res.size(); i++)
        cout << res[i] << " ";
    cout << endl;
}
}

// The function does all Topological Sort.
// It uses recursive alltopologicalSortUtil()
void Graph::alltopologicalSort()
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    vector<int> res;
    alltopologicalSortUtil(res, visited);
}

// Driver program to test above functions
int main()
{
    // Create a graph given in the above diagram
    Graph g(6);
    g.addEdge(5, 2);
    g.addEdge(5, 0);
    g.addEdge(4, 0);
    g.addEdge(4, 1);
    g.addEdge(2, 3);
    g.addEdge(3, 1);

    cout << "All Topological sorts\n";

    g.alltopologicalSort();

    return 0;
}
```

Java

```
//Java program to print all topolgical sorts of a graph
import java.util.*;

class Graph {
    int V; // No. of vertices

    List<Integer> adjListArray[];

    public Graph(int V) {

        this.V = V;

        @SuppressWarnings("unchecked")
        List<Integer> adjListArray[] = new LinkedList[V];

        this.adjListArray = adjListArray;

        for (int i = 0; i < V; i++) {
            adjListArray[i] = new LinkedList<>();
        }
    }

    // Utility function to add edge
    public void addEdge(int src, int dest) {

        this.adjListArray[src].add(dest);
    }

    // Main recursive function to print all possible
    // topological sorts
    private void allTopologicalSortsUtil(boolean[] visited,
                                          int[] indegree, ArrayList<Integer> stack) {
        // To indicate whether all topological are found
        // or not
        boolean flag = false;

        for (int i = 0; i < this.V; i++) {
            // If indegree is 0 and not yet visited then
            // only choose that vertex
            if (!visited[i] && indegree[i] == 0) {

                // including in result
                visited[i] = true;
                stack.add(i);
                for (int adjacent : this.adjListArray[i]) {
                    indegree[adjacent]--;
                }
            }
        }
    }
}
```

```

        }
        allTopologicalSortsUtil(visited, indegree, stack);

        // resetting visited, res and indegree for
        // backtracking
        visited[i] = false;
        stack.remove(stack.size() - 1);
        for (int adjacent : this.adjListArray[i]) {
            indegree[adjacent]++;
        }

        flag = true;
    }
}

// We reach here if all vertices are visited.
// So we print the solution here
if (!flag) {
    stack.forEach(i -> System.out.print(i + " "));
    System.out.println();
}
}

// The function does all Topological Sort.
// It uses recursive alltopologicalSortUtil()
public void allTopologicalSorts() {
    // Mark all the vertices as not visited
    boolean[] visited = new boolean[this.V];

    int[] indegree = new int[this.V];

    for (int i = 0; i < this.V; i++) {

        for (int var : this.adjListArray[i]) {
            indegree[var]++;
        }
    }

    ArrayList<Integer> stack = new ArrayList<>();

    allTopologicalSortsUtil(visited, indegree, stack);
}

// Driver code
public static void main(String[] args) {

    // Create a graph given in the above diagram
    Graph graph = new Graph(6);
}

```

```
graph.addEdge(5, 2);
graph.addEdge(5, 0);
graph.addEdge(4, 0);
graph.addEdge(4, 1);
graph.addEdge(2, 3);
graph.addEdge(3, 1);

System.out.println("All Topological sorts");
graph.allTopologicalSorts();
}

}
```

Output :

```
All Topological sorts
4 5 0 2 3 1
4 5 2 0 3 1
4 5 2 3 0 1
4 5 2 3 1 0
5 2 3 4 0 1
5 2 3 4 1 0
5 2 4 0 3 1
5 2 4 3 0 1
5 2 4 3 1 0
5 4 0 2 3 1
5 4 2 0 3 1
5 4 2 3 0 1
5 4 2 3 1 0
```

This article is contributed by Utkarsh Trivedi. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Improved By : [sakshamcse](#)

Source

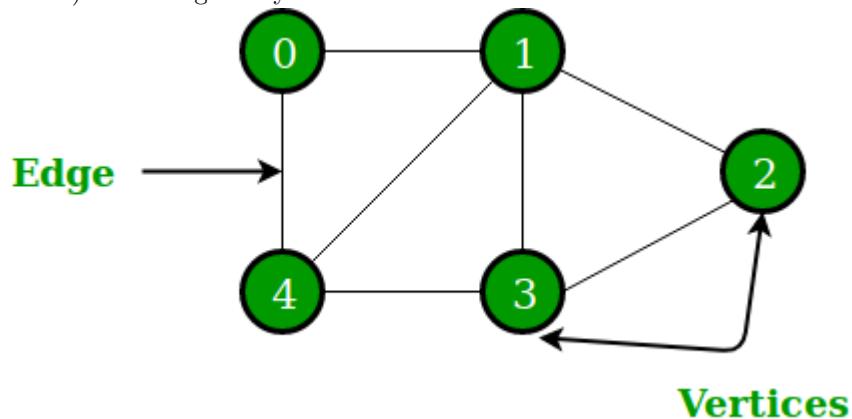
<https://www.geeksforgeeks.org/all-topological-sorts-of-a-directed-acyclic-graph/>

Chapter 77

Applications of Graph Data Structure

Applications of Graph Data Structure - GeeksforGeeks

A graph is a non-linear data structure, which consists of vertices(or nodes) connected by edges(or arcs) where edges may be directed or undirected.



- In **Computer science** graphs are used to represent the flow of computation.
- **Google maps** uses graphs for building transportation systems, where intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices.
- In **Facebook**, users are considered to be the vertices and if they are friends then there is an edge running between them. Facebook's Friend suggestion algorithm uses graph theory.
- In **Operating System**, we come across the Resource Allocation Graph where each process and resources are considered to be vertices. Edges are drawn from resources

to the allocated process, or from requesting process to the requested resource. If this leads to any formation of a cycle then a deadlock will occur.

Thus the development of algorithms to handle graphs is of major interest in the field of computer science.

Source

<https://www.geeksforgeeks.org/applications-of-graph-data-structure/>

Chapter 78

Barabasi Albert Graph (for Scale Free Models)

Barabasi Albert Graph (for Scale Free Models) - GeeksforGeeks

The current article would deal with the concepts surrounding the complex networks using the python library Networkx. It is a Python language software package for the creation, manipulation, and study of the structure, dynamics, and function of complex networks. With NetworkX you can load and store networks in standard and nonstandard data formats, generate many types of random and classic networks, analyze network structure, build network models, design new network algorithms, draw networks, and much more.

The current article would deal with the algorithm for generating random scale free networks for using preferential attachment model. The reason of interest behind this model dates back to the 1990s when Albert Lazlo Barabasi and Reka Albert came out with the path breaking research describing the model followed by the scale free networks around the world. They suggested that several natural and human-made systems, including the Internet, the World Wide Web, citation networks, and some social networks are thought to be approximately scale-free networks.

A scale-free network is a network whose degree distribution follows a power law, at least asymptotically. That is, the fraction $P(k)$ of nodes in the network having k connections to other nodes goes for large values of k as

$$P(k) \propto k^{-\gamma}$$

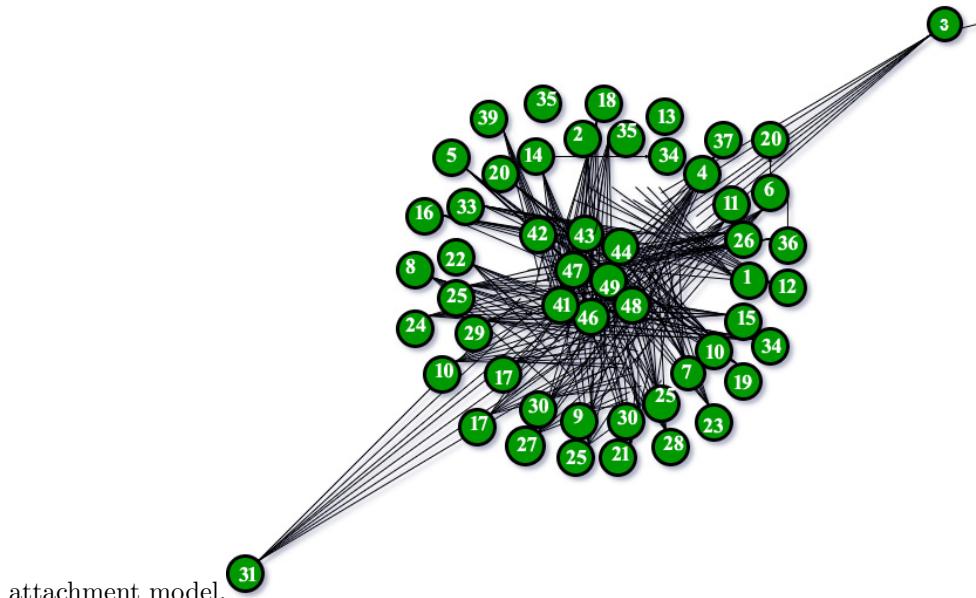
Where γ is a parameter whose value is typically in the range $2 < \gamma < 3$, although occasionally it may lie outside these bounds and c is a proportionality constant.

The Barabási-Albert model is one of several proposed models that generate scale-free networks. It incorporates two important general concepts: growth and preferential attachment. Both growth and preferential attachment exist widely in real networks. Growth means that the number of nodes in the network increases over time.

Preferential attachment means that the more connected a node is, the more likely it is to receive new links. Nodes with higher degree have stronger ability to grab links added to the

network. Intuitively, the preferential attachment can be understood if we think in terms of social networks connecting people. Here a link from A to B means that person A "knows" or "is acquainted with" person B. Heavily linked nodes represent well-known people with lots of relations. When a newcomer enters the community, s/he is more likely to become acquainted with one of those more visible people rather than with a relative unknown. The BA model was proposed by assuming that in the World Wide Web, new pages link preferentially to hubs, i.e. very well-known sites such as Google, rather than to pages that hardly anyone knows. If someone selects a new page to link to by randomly choosing an existing link, the probability of selecting a particular page would be proportional to its degree.

Following image will describe the BA Model graph with 50 nodes following the preferential



attachment model.

The above graph completely satisfies the logic of the rich getting richer and the poor getting poorer.

Code:

The following code is a part of the function which we will eventually implement using the networkx library.

```
def barabasi_albert_graph(n, m, seed=None):
    """Returns a random graph according to the Barabási-Albert preferential
    Attachment model.

    A graph of ``n`` nodes is grown by attaching new nodes each with ``m``.
    Edges that are preferentially attached to existing nodes with high degree.

    Parameters
    -----
    n : int
        Number of nodes
```

```

m : int
    Number of edges to attach from a new node to existing nodes
seed : int, optional
    Seed for random number generator (default=None).

Returns
-----
G : Graph

Raises
-----
NetworkXError
    If ``m`` does not satisfy ``1 <= m < n``.

if m < 1 or m >=n:
    raise nx.NetworkXError("Barabási-Albert network must have m >= 1"
                           " and m < n, m = %d, n = %d" % (m, n))
if seed is not None:
    random.seed(seed)

# Add m initial nodes (m0 in barabasi-speak)
G=empty_graph(m)
G.name="barabasi_albert_graph(%s,%s)"%(n,m)
# Target nodes for new edges
targets=list(range(m))
# List of existing nodes, with nodes repeated once for each adjacent edge
repeated_nodes=[]
# Start adding the other n-m nodes. The first node is m.
source=m
while source<n:
    # Add edges to m nodes from the source.
    G.add_edges_from(zip(*m,targets))
    # Add one node to the list for each new edge just created.
    repeated_nodes.extend(targets)
    # And the new node "source" has m edges to add to the list.
    repeated_nodes.extend(*m)
    # Now choose m unique nodes from the existing nodes
    # Pick uniformly from repeated_nodes (preferential attachment)
    targets = _random_subset(repeated_nodes,m)
    source += 1
return G

```

The above code is a part of the networkx library which is used to handle the random graphs efficiently in python. One will have to install it before running the following code.

```

>>> import networkx as nx
>>> G= nx.barabasi_albert_graph(50,40)

```

```
>>> nx.draw(G, with_labels=True)
```

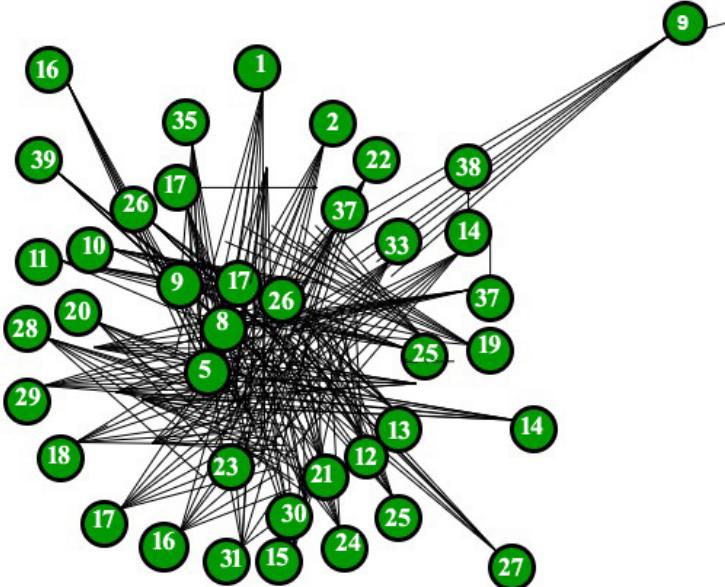
To display the above graph, I used the matplotlib library. We need to install it before the execution of the below codes.

```
>>> import matplotlib.pyplot as plt  
>>> plt.show()
```

So the final code seemed like:

```
>>> import networkx as nx  
>>> import matplotlib.pyplot as plt  
  
>>> G= nx.barabasi_albert_graph(40,15)  
>>> nx.draw(G, with_labels=True)  
>>> plt.show()
```

Output:



BA model for 40 nodes

Thus I would further like to describe more about the networkx library and its modules basically focusing on the centrality measure of a network (especially the scale free models).

References

You can read more about the same at

https://en.wikipedia.org/wiki/Barab%C3%A1si%E2%80%93Albert_model

<http://networkx.readthedocs.io/en/networkx-1.10/index.html>

Source

<https://www.geeksforgeeks.org/barabasi-albert-graph-scale-free-models/>

Chapter 79

Bellman-Ford Algorithm DP-23

Bellman-Ford Algorithm DP-23 - GeeksforGeeks

Given a graph and a source vertex src in graph, find shortest paths from src to all vertices in the given graph. The graph may contain negative weight edges.

We have discussed [Dijkstra's algorithm](#) for this problem. Dijkstra's algorithm is a Greedy algorithm and time complexity is $O(V \log V)$ (with the use of Fibonacci heap). *Dijkstra doesn't work for Graphs with negative weight edges, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is $O(VE)$, which is more than Dijkstra.*

Algorithm

Following are the detailed steps.

Input: Graph and a source vertex src

Output: Shortest distance to all vertices from src . If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

1) This step initializes distances from source to all vertices as infinite and distance to source itself as 0. Create an array $dist[]$ of size V with all values as infinite except $dist[src]$ where src is source vertex.

2) This step calculates shortest distances. Do following $V-1$ times where V is the number of vertices in given graph.

....**a)** Do following for each edge $u-v$

.....If $dist[v] > dist[u] + \text{weight of edge } uv$, then update $dist[v]$

..... $dist[v] = dist[u] + \text{weight of edge } uv$

3) This step reports if there is a negative weight cycle in graph. Do following for each edge $u-v$

.....If $dist[v] > dist[u] + \text{weight of edge } uv$, then “Graph contains negative weight cycle”

The idea of step 3 is, step 2 guarantees shortest distances if graph doesn't contain negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

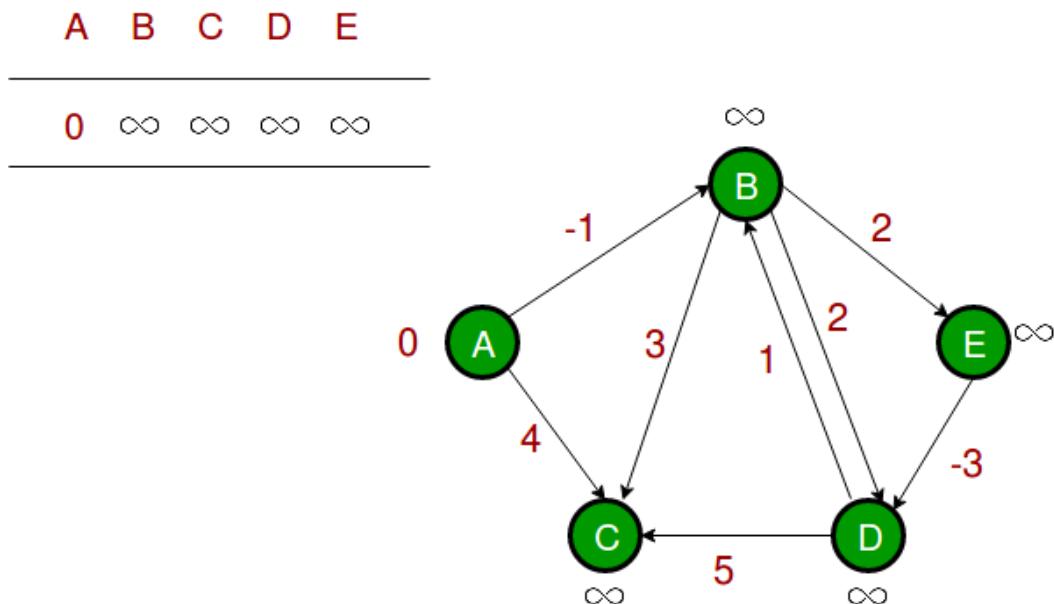
How does this work? Like other Dynamic Programming Problems, the algorithm calculate shortest paths in bottom-up manner. It first calculates the shortest distances which

have at-most one edge in the path. Then, it calculates shortest paths with at-most 2 edges, and so on. After the i -th iteration of outer loop, the shortest paths with at most i edges are calculated. There can be maximum $V - 1$ edges in any simple path, that is why the outer loop runs $v - 1$ times. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most $(i+1)$ edges (Proof is simple, you can refer [this](#) or [MIT Video Lecture](#))

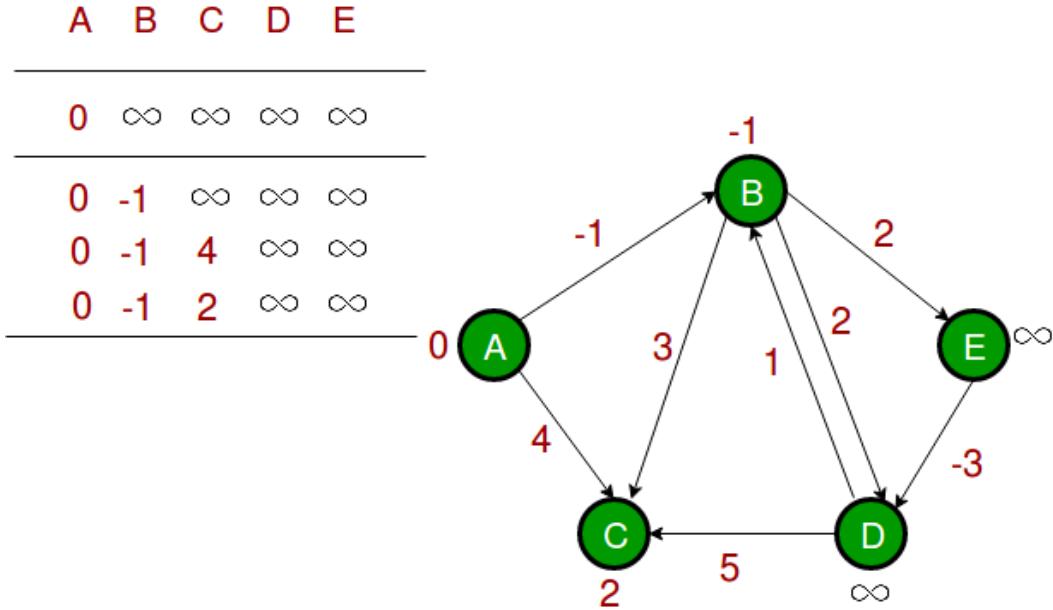
Example

Let us understand the algorithm with following example graph. The images are taken from [this source](#).

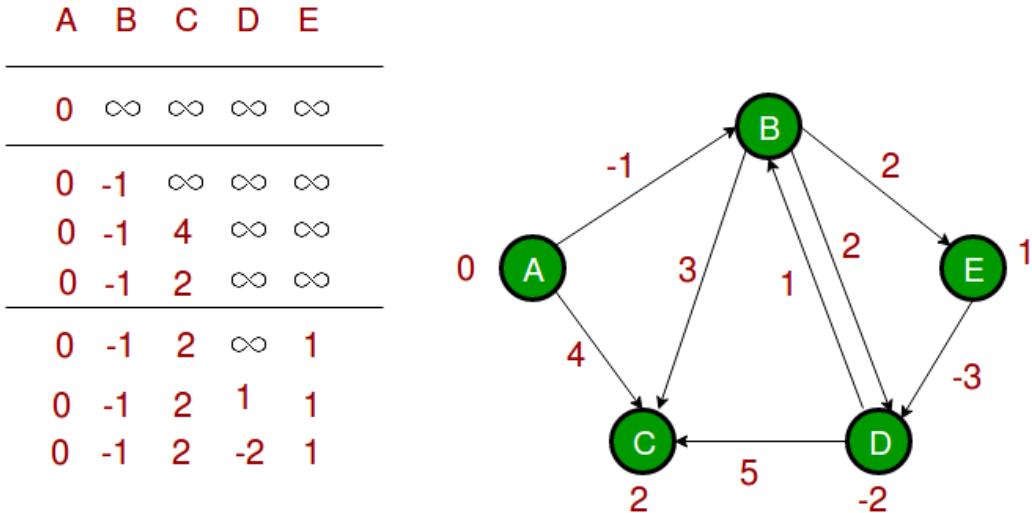
Let the given source vertex be 0. Initialize all distances as infinite, except the distance to source itself. Total number of vertices in the graph is 5, so *all edges must be processed 4 times*.



Let all edges are processed in following order: (B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D). We get following distances when all edges are processed first time. The first row in shows initial distances. The second row shows distances when edges (B,E), (D,B), (B,D) and (A,B) are processed. The third row shows distances when (A,C) is processed. The fourth row shows when (D,C), (B,C) and (E,D) are processed.



The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get following distances when all edges are processed second time (The last row shows final values).



The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don't update the distances.

Implementation:

C++

```
// A C++ program for Bellman-Ford's single source
```

```

// shortest path algorithm.
#include <bits/stdc++.h>

// a structure to represent a weighted edge in graph
struct Edge
{
    int src, dest, weight;
};

// a structure to represent a connected, directed and
// weighted graph
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}

// A utility function used to print the solution
void printArr(int dist[], int n)
{
    printf("Vertex      Distance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

// The main function that finds shortest distances from src to
// all other vertices using Bellman-Ford algorithm. The function
// also detects negative weight cycle
void BellmanFord(struct Graph* graph, int src)
{
    int V = graph->V;
    int E = graph->E;
    int dist[V];

    // Step 1: Initialize distances from src to all other vertices

```

```

// as INFINITE
for (int i = 0; i < V; i++)
    dist[i] = INT_MAX;
dist[src] = 0;

// Step 2: Relax all edges |V| - 1 times. A simple shortest
// path from src to any other vertex can have at-most |V| - 1
// edges
for (int i = 1; i <= V-1; i++)
{
    for (int j = 0; j < E; j++)
    {
        int u = graph->edge[j].src;
        int v = graph->edge[j].dest;
        int weight = graph->edge[j].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
            dist[v] = dist[u] + weight;
    }
}

// Step 3: check for negative-weight cycles. The above step
// guarantees shortest distances if graph doesn't contain
// negative weight cycle. If we get a shorter path, then there
// is a cycle.
for (int i = 0; i < E; i++)
{
    int u = graph->edge[i].src;
    int v = graph->edge[i].dest;
    int weight = graph->edge[i].weight;
    if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
        printf("Graph contains negative weight cycle");
}

printArr(dist, V);

return;
}

// Driver program to test above functions
int main()
{
    /* Let us create the graph given in above example */
    int V = 5; // Number of vertices in graph
    int E = 8; // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1 (or A-B in above figure)
    graph->edge[0].src = 0;
}

```

```
graph->edge[0].dest = 1;
graph->edge[0].weight = -1;

// add edge 0-2 (or A-C in above figure)
graph->edge[1].src = 0;
graph->edge[1].dest = 2;
graph->edge[1].weight = 4;

// add edge 1-2 (or B-C in above figure)
graph->edge[2].src = 1;
graph->edge[2].dest = 2;
graph->edge[2].weight = 3;

// add edge 1-3 (or B-D in above figure)
graph->edge[3].src = 1;
graph->edge[3].dest = 3;
graph->edge[3].weight = 2;

// add edge 1-4 (or A-E in above figure)
graph->edge[4].src = 1;
graph->edge[4].dest = 4;
graph->edge[4].weight = 2;

// add edge 3-2 (or D-C in above figure)
graph->edge[5].src = 3;
graph->edge[5].dest = 2;
graph->edge[5].weight = 5;

// add edge 3-1 (or D-B in above figure)
graph->edge[6].src = 3;
graph->edge[6].dest = 1;
graph->edge[6].weight = 1;

// add edge 4-3 (or E-D in above figure)
graph->edge[7].src = 4;
graph->edge[7].dest = 3;
graph->edge[7].weight = -3;

BellmanFord(graph, 0);

return 0;
}
```

Java

```
// A Java program for Bellman-Ford's single source shortest path
// algorithm.
import java.util.*;
```

```

import java.lang.*;
import java.io.*;

// A class to represent a connected, directed and weighted graph
class Graph
{
    // A class to represent a weighted edge in graph
    class Edge {
        int src, dest, weight;
        Edge() {
            src = dest = weight = 0;
        }
    };
    int V, E;
    Edge edge[];

    // Creates a graph with V vertices and E edges
    Graph(int v, int e)
    {
        V = v;
        E = e;
        edge = new Edge[e];
        for (int i=0; i<e; ++i)
            edge[i] = new Edge();
    }

    // The main function that finds shortest distances from src
    // to all other vertices using Bellman-Ford algorithm. The
    // function also detects negative weight cycle
    void BellmanFord(Graph graph,int src)
    {
        int V = graph.V, E = graph.E;
        int dist[] = new int[V];

        // Step 1: Initialize distances from src to all other
        // vertices as INFINITE
        for (int i=0; i<V; ++i)
            dist[i] = Integer.MAX_VALUE;
        dist[src] = 0;

        // Step 2: Relax all edges |V| - 1 times. A simple
        // shortest path from src to any other vertex can
        // have at-most |V| - 1 edges
        for (int i=1; i<V; ++i)
        {
            for (int j=0; j<E; ++j)
            {

```

```

        int u = graph.edge[j].src;
        int v = graph.edge[j].dest;
        int weight = graph.edge[j].weight;
        if (dist[u] != Integer.MAX_VALUE &&
            dist[u]+weight<dist[v])
            dist[v]=dist[u]+weight;
    }
}

// Step 3: check for negative-weight cycles. The above
// step guarantees shortest distances if graph doesn't
// contain negative weight cycle. If we get a shorter
// path, then there is a cycle.
for (int j=0; j<E; ++j)
{
    int u = graph.edge[j].src;
    int v = graph.edge[j].dest;
    int weight = graph.edge[j].weight;
    if (dist[u] != Integer.MAX_VALUE &&
        dist[u]+weight < dist[v])
        System.out.println("Graph contains negative weight cycle");
}
printArr(dist, V);
}

// A utility function used to print the solution
void printArr(int dist[], int V)
{
    System.out.println("Vertex      Distance from Source");
    for (int i=0; i<V; ++i)
        System.out.println(i+"\t\t"+dist[i]);
}

// Driver method to test above function
public static void main(String[] args)
{
    int V = 5; // Number of vertices in graph
    int E = 8; // Number of edges in graph

    Graph graph = new Graph(V, E);

    // add edge 0-1 (or A-B in above figure)
    graph.edge[0].src = 0;
    graph.edge[0].dest = 1;
    graph.edge[0].weight = -1;

    // add edge 0-2 (or A-C in above figure)
    graph.edge[1].src = 0;
}

```

```
graph.edge[1].dest = 2;
graph.edge[1].weight = 4;

// add edge 1-2 (or B-C in above figure)
graph.edge[2].src = 1;
graph.edge[2].dest = 2;
graph.edge[2].weight = 3;

// add edge 1-3 (or B-D in above figure)
graph.edge[3].src = 1;
graph.edge[3].dest = 3;
graph.edge[3].weight = 2;

// add edge 1-4 (or A-E in above figure)
graph.edge[4].src = 1;
graph.edge[4].dest = 4;
graph.edge[4].weight = 2;

// add edge 3-2 (or D-C in above figure)
graph.edge[5].src = 3;
graph.edge[5].dest = 2;
graph.edge[5].weight = 5;

// add edge 3-1 (or D-B in above figure)
graph.edge[6].src = 3;
graph.edge[6].dest = 1;
graph.edge[6].weight = 1;

// add edge 4-3 (or E-D in above figure)
graph.edge[7].src = 4;
graph.edge[7].dest = 3;
graph.edge[7].weight = -3;

graph.BellmanFord(graph, 0);
}

}

// Contributed by Aakash Hasija
```

Python

```
# Python program for Bellman-Ford's single source
# shortest path algorithm.

from collections import defaultdict

#Class to represent a graph
class Graph:
```

```

def __init__(self,vertices):
    self.V= vertices #No. of vertices
    self.graph = {} # default dictionary to store graph

# function to add an edge to graph
def addEdge(self,u,v,w):
    self.graph.append([u, v, w])

# utility function used to print the solution
def printArr(self, dist):
    print("Vertex   Distance from Source")
    for i in range(self.V):
        print("%d \t\t %d" % (i, dist[i]))

# The main function that finds shortest distances from src to
# all other vertices using Bellman-Ford algorithm. The function
# also detects negative weight cycle
def BellmanFord(self, src):

    # Step 1: Initialize distances from src to all other vertices
    # as INFINITE
    dist = [float("Inf")] * self.V
    dist[src] = 0

    # Step 2: Relax all edges |V| - 1 times. A simple shortest
    # path from src to any other vertex can have at-most |V| - 1
    # edges
    for i in range(self.V - 1):
        # Update dist value and parent index of the adjacent vertices of
        # the picked vertex. Consider only those vertices which are still in
        # queue
        for u, v, w in self.graph:
            if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w

    # Step 3: check for negative-weight cycles. The above step
    # guarantees shortest distances if graph doesn't contain
    # negative weight cycle. If we get a shorter path, then there
    # is a cycle.

    for u, v, w in self.graph:
        if dist[u] != float("Inf") and dist[u] + w < dist[v]:
            print "Graph contains negative weight cycle"
            return

    # print all distance
    self.printArr(dist)

```

```
g = Graph(5)
g.addEdge(0, 1, -1)
g.addEdge(0, 2, 4)
g.addEdge(1, 2, 3)
g.addEdge(1, 3, 2)
g.addEdge(1, 4, 2)
g.addEdge(3, 2, 5)
g.addEdge(3, 1, 1)
g.addEdge(4, 3, -3)

#print the solution
g.BellmanFord(0)

#This code is contributed by Neelam Yadav
```

Output:

Vertex	Distance from Source
0	0
1	-1
2	2
3	-2
4	1

Notes

- 1) Negative weights are found in various applications of graphs. For example, instead of paying cost for a path, we may get some advantage if we follow the path.
- 2) Bellman-Ford works better (better than Dijksra's) for distributed systems. Unlike Dijksra's where we need to find minimum value of all vertices, in Bellman-Ford, edges are considered one by one.

Exercise

- 1) The standard Bellman-Ford algorithm reports shortest path only if there is no negative weight cycles. Modify it so that it reports minimum distances even if there is a negative weight cycle.
- 2) Can we use Dijksra's algorithm for shortest paths for graphs with negative weights – one idea can be, calculate the minimum weight value, add a positive value (equal to absolute value of minimum weight value) to all weights and run the Dijksra's algorithm for the modified graph. Will this algorithm work?

References:

- <http://www.youtube.com/watch?v=Ttezuzs39nk>
- http://en.wikipedia.org/wiki/Bellman%20%93Ford_algorithm
- <http://www.cs.arizona.edu/classes/cs445/spring07/ShortestPath2.prn.pdf>

Source

<https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/>

Chapter 80

Best First Search (Informed Search)

Best First Search (Informed Search) - GeeksforGeeks

Prerequisites : [BFS](#), [DFS](#)

In BFS and DFS, when we are at a node, we can consider any of the adjacent as next node. So both BFS and DFS blindly explore paths without considering any cost function. The idea of Best First Search is to use an evaluation function to decide which adjacent is most promising and then explore. Best First Search falls under the category of Heuristic Search or Informed Search.

We use a priority queue to store costs of nodes. So the implementation is a variation of BFS, we just need to change Queue to PriorityQueue.

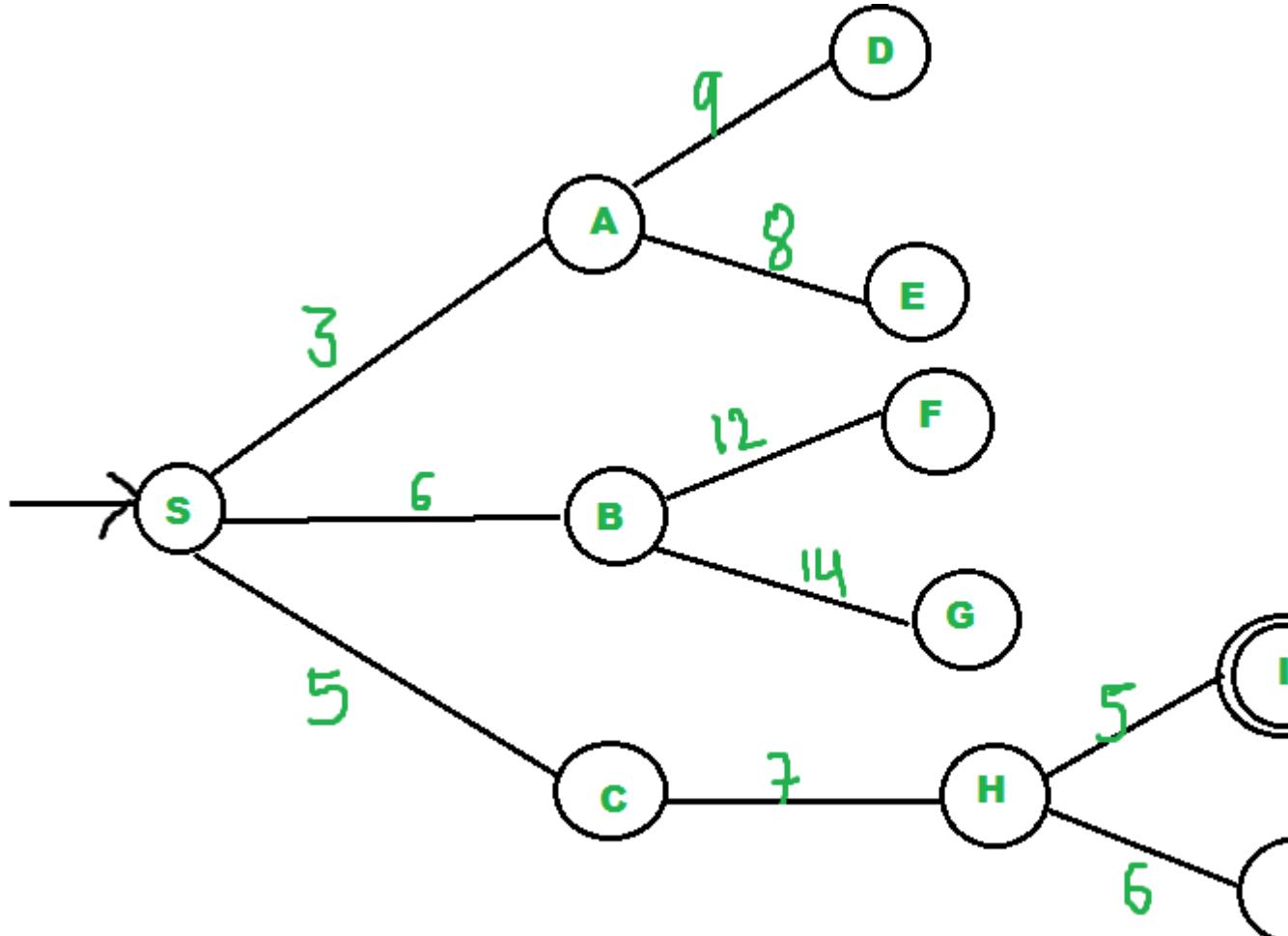
```
// This pseudocode is adapted from below
// source:
// https://courses.cs.washington.edu/
Best-First-Search(Grah g, Node start)
    1) Create an empty PriorityQueue
        PriorityQueue pq;
    2) Insert "start" in pq.
        pq.insert(start)
    3) Until PriorityQueue is empty
        u = PriorityQueue.DeleteMin
        If u is the goal
            Exit
        Else
            Foreach neighbor v of u
                If v "Unvisited"
                    Mark v "Visited"
```

```

    pq.insert(v)
    Mark v "Examined"
End procedure

```

Let us consider below example.



We start from source "S" and search for goal "I" using given costs and Best First search.

pq initially contains S
 We remove S from and process unvisited neighbors of S to pq.
 pq now contains {A, C, B} (C is put

before B because C has lesser cost)

We remove A from pq and process unvisited
neighbors of A to pq.
pq now contains {C, B, E, D}

We remove C from pq and process unvisited
neighbors of C to pq.
pq now contains {B, H, E, D}

We remove B from pq and process unvisited
neighbors of B to pq.
pq now contains {H, E, D, F, G}

We remove H from pq. Since our goal
"I" is a neighbor of H, we return.

Analysis :

- The worst case time complexity for Best First Search is $O(n * \log n)$ where n is number of nodes. In worst case, we may have to visit all nodes before we reach goal. Note that priority queue is implemented using Min(or Max) Heap, and insert and remove operations take $O(\log n)$ time.
- Performance of the algorithm depends on how well the cost or evaluation function is designed.

Related Article:

[A* Search Algorithm](#)

Source

<https://www.geeksforgeeks.org/best-first-search-informed-search/>

Chapter 81

Betweenness Centrality (Centrality Measure)

Betweenness Centrality (Centrality Measure) - GeeksforGeeks

In graph theory, betweenness centrality is a measure of centrality in a graph based on shortest paths. For every pair of vertices in a connected graph, there exists at least one shortest path between the vertices such that either the number of edges that the path passes through (for unweighted graphs) or the sum of the weights of the edges (for weighted graphs) is minimized. The betweenness centrality for each vertex is the number of these shortest paths that pass through the vertex.

Betweenness centrality finds wide application in network theory: it represents the degree of which nodes stand between each other. For example, in a telecommunications network, a node with higher betweenness centrality would have more control over the network, because more information will pass through that node. Betweenness centrality was devised as a general measure of centrality: it applies to a wide range of problems in network theory, including problems related to social networks, biology, transport and scientific cooperation.

Definition

The betweenness centrality of a node v is given by the expression:

$$g(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

where σ_{st} is the total number of shortest paths from node s to node t and $\sigma_{st}(v)$ is the number of those paths that pass through v .

Note that the betweenness centrality of a node scales with the number of pairs of nodes as implied by the summation indices. Therefore, the calculation may be rescaled by dividing through by the number of pairs of nodes not including v , so that $\sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} = \frac{N(N-1)(N-2)}{3}$. The division is done by $\frac{N(N-1)(N-2)}{3}$ for directed

graphs and $\frac{(N-1)(N-2)/2}{N}$ for undirected graphs, where N is the number of nodes in the giant component. Note that this scales for the highest possible value, where one node is crossed by every single shortest path. This is often not the case, and a normalization can be performed without a loss of precision

$$\text{normalized } \sigma(v) = \frac{\sigma(v)}{\frac{(N-1)(N-2)/2}{N}}$$

which results in:

$$\begin{aligned} \text{normalized } \sigma(v) &= \frac{1}{\frac{(N-1)(N-2)/2}{N}} \\ \text{normalized } \sigma(v) &= \frac{N}{(N-1)(N-2)/2} \end{aligned}$$

Note that this will always be a scaling from a smaller range into a larger range, so no precision is lost.

Weighted Networks

In a weighted network the links connecting the nodes are no longer treated as binary interactions, but are weighted in proportion to their capacity, influence, frequency, etc., which adds another dimension of heterogeneity within the network beyond the topological effects. A node's strength in a weighted network is given by the sum of the weights of its adjacent edges.

$$S_i = \sum_{j \neq i} w_{ij}$$

With A_{ij} and w_{ij} being adjacency and weight matrices between nodes i and j , respectively. Analogous to the power law distribution of degree found in scale free networks, the strength of a given node follows a power law distribution as well.

$$S_i \propto t^b$$

A study of the average value $\bar{S}(b)$ of the strength for vertices with betweenness b shows that the functional behavior can be approximated by a scaling form

$$\bar{S}(b) \propto b^{-\alpha}$$

Following is the code for the calculation of the betweenness centrality of the graph and its various nodes.

```
def betweenness_centrality(G, k=None, normalized=True, weight=None,
                           endpoints=False, seed=None):
    """Compute the shortest-path betweenness centrality for nodes.

    Betweenness centrality of a node $v$ is the sum of the
    fraction of all-pairs shortest paths that pass through $v$.

    ... math::
```

$$c_B(v) = \sum_{s,t \in V} \frac{\sigma(s, t|v)}{\sigma(s, t)}$$

where V is the set of nodes, $\sigma(s, t)$ is the number of shortest (s, t) -paths, and $\sigma(s, t|v)$ is the number of those paths passing through some node v other than s, t . If $s = t$, $\sigma(s, t) = 1$, and if $v \in \{s, t\}$, $\sigma(s, t|v) = 0$ [2].

Parameters

`G : graph`
A NetworkX graph.

`k : int, optional (default=None)`

If `k` is not `None` use `k` node samples to estimate betweenness.

The value of `k <= n` where `n` is the number of nodes in the graph.

Higher values give better approximation.

`normalized : bool, optional`

If `True` the betweenness values are normalized by $2/((n-1)(n-2))$ for graphs, and $1/((n-1)(n-2))$ for directed graphs where `n` is the number of nodes in `G`.

`weight : None or string, optional (default=None)`

If `None`, all edge weights are considered equal.

Otherwise holds the name of the edge attribute used as weight.

`endpoints : bool, optional`

If `True` include the endpoints in the shortest path counts.

Returns

`nodes : dictionary`

Dictionary of nodes with betweenness centrality as the value.

Notes

The algorithm is from Ulrik Brandes [1].

See [4] for the original first published version and [2] for details on algorithms for variations and related metrics.

For approximate betweenness calculations set `k=#samples` to use `k` nodes ("pivots") to estimate the betweenness values. For an estimate of the number of pivots needed see [3].

For weighted graphs the edge weights must be greater than zero.

Zero edge weights can produce an infinite number of equal length paths between pairs of nodes.

```
"""
betweenness = dict.fromkeys(G, 0.0) # b[v]=0 for v in G
if k is None:
    nodes = G
else:
    random.seed(seed)
    nodes = random.sample(G.nodes(), k)
for s in nodes:

    # single source shortest paths
    if weight is None: # use BFS
        S, P, sigma = _single_source_shortest_path_basic(G, s)
    else: # use Dijkstra's algorithm
        S, P, sigma = _single_source_dijkstra_path_basic(G, s, weight)

    # accumulation
    if endpoints:
        betweenness = _accumulate_endpoints(betweenness, S, P, sigma, s)
    else:
        betweenness = _accumulate_basic(betweenness, S, P, sigma, s)

# rescaling
betweenness = _rescale(betweenness, len(G), normalized=normalized,
                      directed=G.is_directed(), k=k)
return betweenness
```

The above function is invoked using the networkx library and once the library is installed, you can eventually use it and the following code has to be written in python for the implementation of the betweenness centrality of a node.

```
>>> import networkx as nx
>>> G=nx.erdos_renyi_graph(50,0.5)
>>> b=nx.betweenness_centrality(G)
>>> print(b)
```

The result of it is:

```
{0: 0.01220586070437195, 1: 0.009125402885768874, 2: 0.010481510111098788, 3: 0.0146456909071822
4: 0.013407129955492722, 5: 0.008165902336070403, 6: 0.008515486873573529, 7: 0.00673628833379575
8: 0.009167651113672941, 9: 0.012386122359980324, 10: 0.00711685931010503, 11: 0.0114635883585897
12: 0.010392276809830674, 13: 0.0071149912635190965, 14: 0.011112503660641336, 15: 0.008013362669
16: 0.01332441710128969, 17: 0.009307485134691016, 18: 0.006974541084171777, 19: 0.0065346360683
20: 0.007794762718607258, 21: 0.012297442232146375, 22: 0.011081427155225095, 23: 0.018715475770}
```

```
24: 0.011527827410298818, 25: 0.012294312339823964, 26: 0.008103941622217354, 27: 0.011063824793  
28: 0.00876321613116331, 29: 0.01539738650994337, 30: 0.014968892689224241, 31: 0.00694256978632  
32: 0.01389881951343378, 33: 0.005315473883526104, 34: 0.012485048548223817, 35: 0.0091478490104  
36: 0.00755662592209711, 37: 0.007387027127423285, 38: 0.015993065123210606, 39: 0.0111516804297  
40: 0.010720274864419366, 41: 0.007769933231367805, 42: 0.009986222659285306, 43: 0.005102869708  
44: 0.007652686310399397, 45: 0.017408689421606432, 46: 0.008512679806690831, 47: 0.010277611517  
48: 0.008908600658162324, 49: 0.013439198921385216}
```

The above result is a dictionary depicting the value of betweenness centrality of each node.
The above is an extension of my article series on the centrality measures. Keep networking!!!

References

You can read more about the same at

https://en.wikipedia.org/wiki/Betweenness_centrality

<http://networkx.readthedocs.io/en/networkx-1.10/index.html>

Source

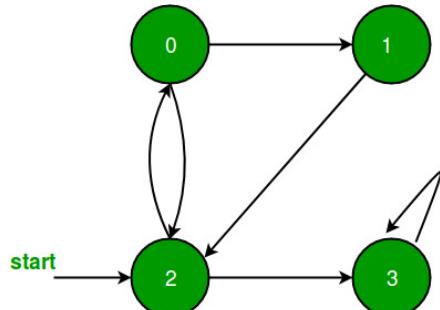
<https://www.geeksforgeeks.org/betweenness-centrality-centrality-measure/>

Chapter 82

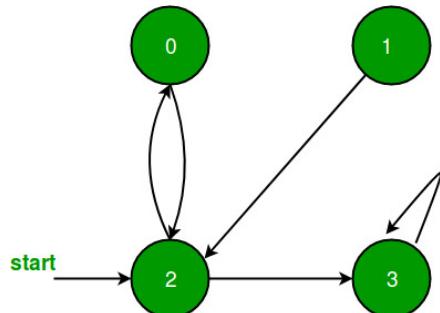
BFS for Disconnected Graph

BFS for Disconnected Graph - GeeksforGeeks

In previous [post](#), BFS only with a particular vertex is performed i.e. it is assumed that all vertices are reachable from the starting vertex. But in the case of disconnected graph or any vertex that is unreachable from all vertex, the previous implementation will not give the desired output, so in this post, a modification is done in BFS.



All vertices are reachable. So, for above graph simple [BFS](#) will work.



As in above graph a vertex 1 is unreachable from all vertex, so simple BFS wouldn't work for it.

Just to modify BFS, perform simple BFS from each unvisited vertex of given graph.

```
// C++ implementation of modified BFS
#include<bits/stdc++.h>
using namespace std;

// A utility function to add an edge in an
// undirected graph.
void addEdge(vector<int> adj[], int u, int v)
{
    adj[u].push_back(v);
}

// A utility function to do BFS of graph
// from a given vertex u.
void BFSUtil(int u, vector<int> adj[],
             vector<bool> &visited)
{

    // Create a queue for BFS
    list<int> q;

    // Mark the current node as visited and enqueue it
    visited[u] = true;
    q.push_back(u);

    // 'i' will be used to get all adjacent vertices 4
    // of a vertex list<int>::iterator i;

    while(!q.empty())
    {
        // Dequeue a vertex from queue and print it
        u = q.front();
        cout << u << " ";
        q.pop_front();

        // Get all adjacent vertices of the dequeued
        // vertex s. If an adjacent has not been visited,
        // then mark it visited and enqueue it
        for (int i = 0; i != adj[u].size(); ++i)
        {
            if (!visited[adj[u][i]])
            {
                visited[adj[u][i]] = true;
                q.push_back(adj[u][i]);
            }
        }
    }
}
```

```
        }
    }

// This function does BFSUtil() for all
// unvisited vertices.
void BFS(vector<int> adj[], int V)
{
    vector<bool> visited(V, false);
    for (int u=0; u<V; u++)
        if (visited[u] == false)
            BFSUtil(u, adj, visited);
}

// Driver code
int main()
{
    int V = 5;
    vector<int> adj[V];

    addEdge(adj, 0, 4);
    addEdge(adj, 1, 2);
    addEdge(adj, 1, 3);
    addEdge(adj, 1, 4);
    addEdge(adj, 2, 3);
    addEdge(adj, 3, 4);
    BFS(adj, V);
    return 0;
}
```

Output:

0 4 1 2 3

Source

<https://www.geeksforgeeks.org/bfs-disconnected-graph/>

Chapter 83

BFS using STL for competitive coding

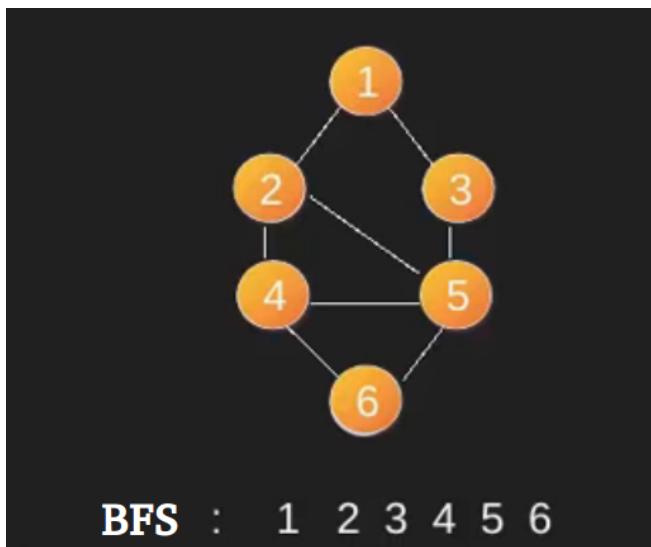
BFS using STL for competitive coding - GeeksforGeeks

A STL based simple implementation of BFS using [queue](#) and [vector](#) in STL. The adjacency list is represented using vectors of vector.

In BFS, we start with a node.

- 1) Create a queue and enqueue source into it.
Mark source as visited.
- 2) While queue is not empty, do following
 - a) Dequeue a vertex from queue. Let this be f.
 - b) Print f
 - c) Enqueue all not yet visited adjacent of f and mark them visited.

Below is an example BFS starting from source vertex 1. Note that there can be multiple BFSs possible for a graph (even from a particular vertex).



For more details of BFS, refer this post .

The code here is simplified such that it could be used in competitive coding.

```
// A Quick implementation of BFS using
// vectors and queue
#include <bits/stdc++.h>
#define pb push_back

using namespace std;

vector<bool> v;
vector<vector<int> > g;

void edge(int a, int b)
{
    g[a].pb(b);

    // for undirected graph add this line
    // g[b].pb(a);
}

void bfs(int u)
{
    queue<int> q;

    q.push(u);
    v[u] = true;

    while (!q.empty()) {
```

```
int f = q.front();
q.pop();

cout << f << " ";

// Enqueue all adjacent of f and mark them visited
for (auto i = g[f].begin(); i != g[f].end(); i++) {
    if (!v[*i]) {
        q.push(*i);
        v[*i] = true;
    }
}
}

// Driver code
int main()
{
    int n, e;
    cin >> n >> e;

    v.assign(n, false);
    g.assign(n, vector<int>());

    int a, b;
    for (int i = 0; i < e; i++) {
        cin >> a >> b;
        edge(a, b);
    }

    for (int i = 0; i < n; i++) {
        if (!v[i])
            bfs(i);
    }

    return 0;
}
```

Input:

```
8 10
0 1
0 2
0 3
0 4
1 5
2 5
3 6
```

```
4 6  
5 7  
6 7
```

Output:

```
0 1 2 3 4 5 6 7
```

Source

<https://www.geeksforgeeks.org/bfs-using-stl-competitive-coding/>

Chapter 84

BFS using vectors & queue as per the algorithm of CLRS

BFS using vectors & queue as per the algorithm of CLRS - GeeksforGeeks

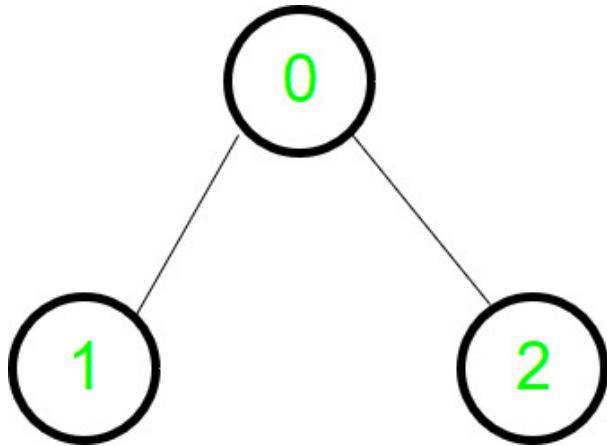
[Breadth-first search](#) traversal of a graph using the algorithm given in [CLRS book](#).

BFS is one of the ways to traverse a graph. It is named so because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. What it means is that the algorithm first discovers all the vertices connected to “u” at a distance of k before discovering the vertices at a distance of k+1 from u. The algorithm given in CLRS uses the concept of “colour” to check if a vertex is discovered fully or partially or undiscovered. It also keeps a track of the distance a vertex u is from the source s.

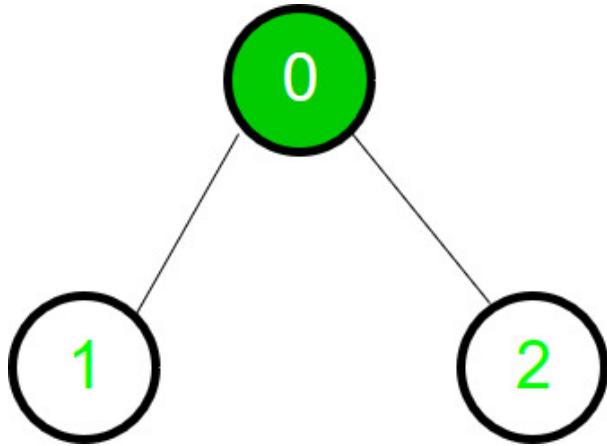
```
BFS(G,s)
1  for each vertex u in G.V - {s}
2      u.color = white
3      u.d = INF
4      u.p = NIL
5  s.color = green
6  s.d = 0
7  s.p = NIL
8  Q = NULL
9  ENQUEUE(Q,s)
10 while Q != NULL
11     u = DEQUEUE(Q)
12     for each v in G.Adj[u]
13         if v.color == white
14             v.color = green
15             v.d = u.d + 1
16             v.p = u
17             ENQUEUE(Q,v)
```

```
18     u.color = dark_green
```

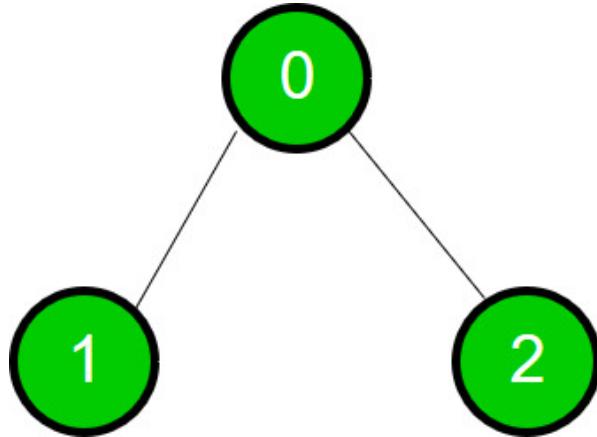
It produces a “breadth-first tree” with root s that contains all reachable vertices. Let’s take a simple directed graph and see how BFS traverses it.



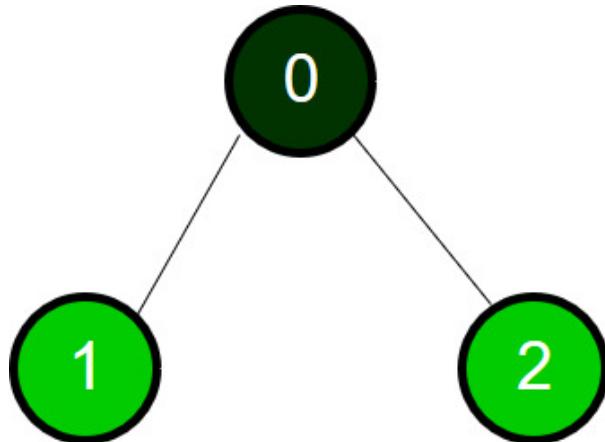
The graph



Starting of traversal



1st traversal



1st traversal completes

```
// CPP program to implement BFS as per CLRS
// algorithm.
#include <bits/stdc++.h>
using namespace std;

// Declaring the vectors to store color, distance
// and parent
vector<string> colour;
vector<int> d;
vector<int> p;

/* This function adds an edge to the graph.
It is an undirected graph. So edges are
added for both the nodes. */
```

```

void addEdge(vector <int> g[], int u, int v)
{
    g[u].push_back(v);
    g[v].push_back(u);
}

/* This function does the Breadth First Search*/
void BFSSingleSource(vector <int> g[], int s)
{
    // The Queue used for the BFS operation
    queue<int> q;

    // Pushing the root node inside the queue
    q.push(s);

    /* Distance of root node is 0 & colour
    is gray as it is visited partially now */
    d[s] = 0;
    colour[s] = "green";

    /* Loop to traverse the graph. Traversal
    will happen traverse until the queue is
    not empty.*/
    while (!q.empty())
    {
        /* Extracting the front element(node)
        and poping it out of queue. */
        int u = q.front();
        q.pop();

        cout << u << " ";

        /* This loop traverses all the child nodes of u */
        for (auto i = g[u].begin(); i != g[u].end(); i++)
        {
            /* If the colour is white then the said node
            is not traversed. */
            if (colour[*i] == "white")
            {
                colour[*i] = "green";
                d[*i] = d[u] + 1;
                p[*i] = u;

                /* Pushing the node inside queue
                to traverse its children. */
                q.push(*i);
            }
        }
    }
}

```

```
/* Now the node u is completely traversed
and colour is changed to black. */
colour[u] = "dark_green";
}
}

void BFSSingleSource(vector <int> g[], int n)
{
    /* Initially all nodes are not traversed.
    Therefore, the colour is white. */
    colour.assign(n, "white");
    d.assign(n, 0);
    p.assign(n, -1);

    // Calling BFSSingleSource() for all white
    // vertices.
    for (int i = 0; i < n; i++)
        if (colour[i] == "white")
            BFSSingleSource(g, i);
}

// Driver Function
int main()
{
    // Graph with 7 nodes and 6 edges.
    int n = 7;

    // The Graph vector
    vector <int> g[n];

    addEdge(g, 0, 1);
    addEdge(g, 0, 2);
    addEdge(g, 1, 3);
    addEdge(g, 1, 4);
    addEdge(g, 2, 5);
    addEdge(g, 2, 6);

    BFSSingleSource(g, n);

    return 0;
}
```

Output:

0 1 2 3 4 5 6

Improved By : [Kunjal Rupala](#)

Source

<https://www.geeksforgeeks.org/bfs-using-vectors-queue-per-algorithm-clrs/>

Chapter 85

Bidirectional Search

Bidirectional Search - GeeksforGeeks

Searching a graph is quite famous problem and have a lot of practical use. We have already discussed [here](#) how to search for a goal vertex starting from a source vertex using [BFS](#). In normal graph search using BFS/DFS we begin our search in one direction usually from source vertex toward the goal vertex, **but what if we start search form both direction simultaneously.**

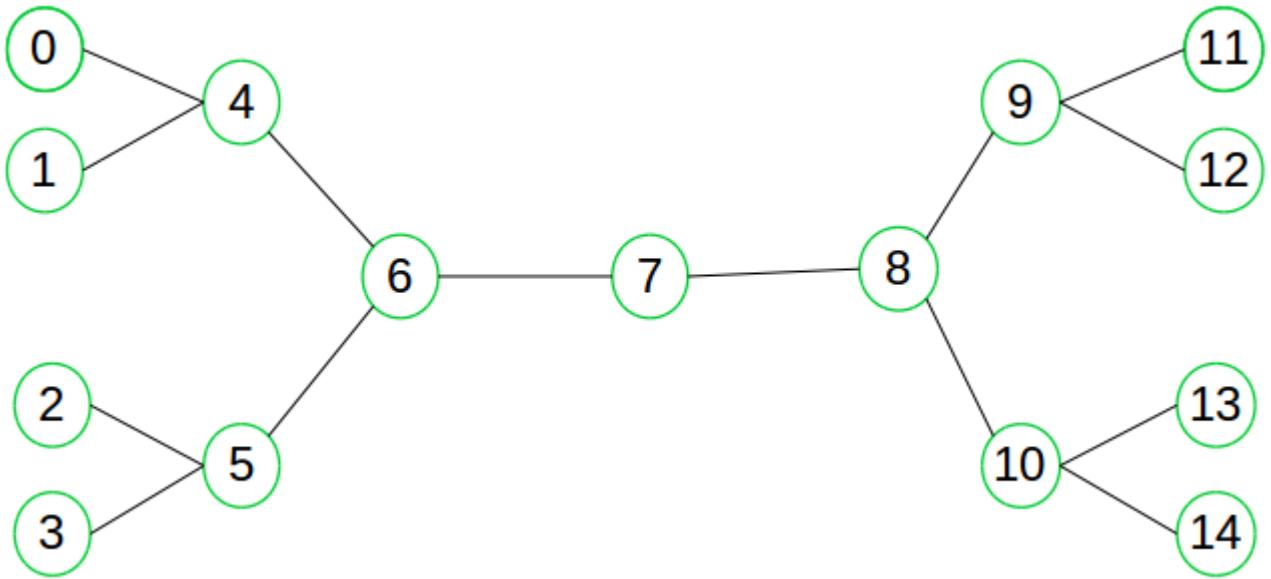
Bidirectional search is a graph search algorithm which find smallest path form source to goal vertex. It runs two simultaneous search –

1. Forward search form source/initial vertex toward goal vertex
2. Backward search form goal/target vertex toward source vertex

Bidirectional search replaces single search graph(which is likely to grow exponentially) with two smaller sub graphs – one starting from initial vertex and other starting from goal vertex. **The search terminates when two graphs intersect.**

Just like [A* algorithm](#), bidirectional search can be guided by a [heuristic](#) estimate of remaining distance from source to goal and vice versa for finding shortest path possible.

Consider following simple example-



Suppose we want to find if there exists a path from vertex 0 to vertex 14. Here we can execute two searches, one from vertex 0 and other from vertex 14. When both forward and backward search meet at vertex 7, we know that we have found a path from node 0 to 14 and search can be terminated now. We can clearly see that we have successfully avoided unnecessary exploration.

Why bidirectional approach?

Because in many cases it is faster, it dramatically reduce the amount of required exploration. Suppose if branching factor of tree is b and distance of goal vertex from source is d , then

the normal BFS/DFS searching complexity would be $\mathcal{O}(b^d)$. On the other hand, if we

execute two search operation then the complexity would be $\mathcal{O}(b^{d/2})$ for each search

and total complexity would be $\mathcal{O}(b^{d/2} + b^{d/2})$ which is far less than $\mathcal{O}(b^d)$.

When to use bidirectional approach?

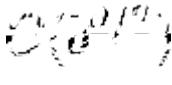
We can consider bidirectional approach when-

1. Both initial and goal states are unique and completely defined.
2. The branching factor is exactly the same in both directions.

Performance measures

- Completeness : Bidirectional search is complete if BFS is used in both searches.

- Optimality : It is optimal if BFS is used for search and paths have uniform cost.

- Time and Space Complexity : Time and space complexity is 

Below is very simple implementation representing the concept of bidirectional search using BFS. This implementation considers undirected paths without any weight.

```

// C++ program for Bidirectional BFS search
// to check path between two vertices
#include <bits/stdc++.h>
using namespace std;

// class representing undirected graph
// using adjacency list
class Graph
{
    //number of nodes in graph
    int V;

    // Adjacency list
    list<int> *adj;
public:
    Graph(int V);
    int isIntersecting(bool *s_visited, bool *t_visited);
    void addEdge(int u, int v);
    void printPath(int *s_parent, int *t_parent, int s,
                   int t, int intersectNode);
    void BFS(list<int> *queue, bool *visited, int *parent);
    int biDirSearch(int s, int t);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

// Method for adding undirected edge
void Graph::addEdge(int u, int v)
{
    this->adj[u].push_back(v);
    this->adj[v].push_back(u);
}

// Method for Breadth First Search
void Graph::BFS(list<int> *queue, bool *visited,
                int *parent)

```

```

{
    int current = queue->front();
    queue->pop_front();
    list<int>::iterator i;
    for (i=adj[current].begin();i != adj[current].end();i++)
    {
        // If adjacent vertex is not visited earlier
        // mark it visited by assigning true value
        if (!visited[*i])
        {
            // set current as parent of this vertex
            parent[*i] = current;

            // Mark this vertex visited
            visited[*i] = true;

            // Push to the end of queue
            queue->push_back(*i);
        }
    }
};

// check for intersecting vertex
int Graph::isIntersecting(bool *s_visited, bool *t_visited)
{
    int intersectNode = -1;
    for(int i=0;i<V;i++)
    {
        // if a vertex is visited by both front
        // and back BFS search return that node
        // else return -1
        if(s_visited[i] && t_visited[i])
            return i;
    }
    return -1;
};

// Print the path from source to target
void Graph::printPath(int *s_parent, int *t_parent,
                      int s, int t, int intersectNode)
{
    vector<int> path;
    path.push_back(intersectNode);
    int i = intersectNode;
    while (i != s)
    {
        path.push_back(s_parent[i]);
        i = s_parent[i];
    }
}

```

```
}

reverse(path.begin(), path.end());
i = intersectNode;
while(i != t)
{
    path.push_back(t_parent[i]);
    i = t_parent[i];
}

vector<int>::iterator it;
cout<<"*****Path*****\n";
for(it = path.begin();it != path.end();it++)
    cout<<*it<<" ";
cout<<"\n";
};

// Method for bidirectional searching
int Graph::biDirSearch(int s, int t)
{
    // boolean array for BFS started from
    // source and target(front and backward BFS)
    // for keeping track on visited nodes
    bool s_visited[V], t_visited[V];

    // Keep track on parents of nodes
    // for front and backward search
    int s_parent[V], t_parent[V];

    // queue for front and backward search
    list<int> s_queue, t_queue;

    int intersectNode = -1;

    // necessary initialization
    for(int i=0; i<V; i++)
    {
        s_visited[i] = false;
        t_visited[i] = false;
    }

    s_queue.push_back(s);
    s_visited[s] = true;

    // parent of source is set to -1
    s_parent[s]=-1;

    t_queue.push_back(t);
    t_visited[t] = true;
```

```

// parent of target is set to -1
t_parent[t] = -1;

while (!s_queue.empty() && !t_queue.empty())
{
    // Do BFS from source and target vertices
    BFS(&s_queue, s_visited, s_parent);
    BFS(&t_queue, t_visited, t_parent);

    // check for intersecting vertex
    intersectNode = isIntersecting(s_visited, t_visited);

    // If intersecting vertex is found
    // that means there exist a path
    if(intersectNode != -1)
    {
        cout << "Path exist between " << s << " and "
            << t << "\n";
        cout << "Intersection at: " << intersectNode << "\n";

        // print the path and exit the program
        printPath(s_parent, t_parent, s, t, intersectNode);
        exit(0);
    }
}
return -1;
}

// Driver code
int main()
{
    // no of vertices in graph
    int n=15;

    // source vertex
    int s=0;

    // target vertex
    int t=14;

    // create a graph given in above diagram
    Graph g(n);
    g.addEdge(0, 4);
    g.addEdge(1, 4);
    g.addEdge(2, 5);
    g.addEdge(3, 5);
    g.addEdge(4, 6);
}

```

```
g.addEdge(5, 6);
g.addEdge(6, 7);
g.addEdge(7, 8);
g.addEdge(8, 9);
g.addEdge(8, 10);
g.addEdge(9, 11);
g.addEdge(9, 12);
g.addEdge(10, 13);
g.addEdge(10, 14);
if (g.biDirSearch(s, t) == -1)
    cout << "Path don't exist between "
    << s << " and " << t << "\n";

return 0;
}
```

Output:

```
Path exist between 0 and 14
Intersection at: 7
*****Path*****
0 4 6 7 8 10 14
```

References

- https://en.wikipedia.org/wiki/Bidirectional_search

Source

<https://www.geeksforgeeks.org/bidirectional-search/>

Chapter 86

Boggle (Find all possible words in a board of characters) Set 1

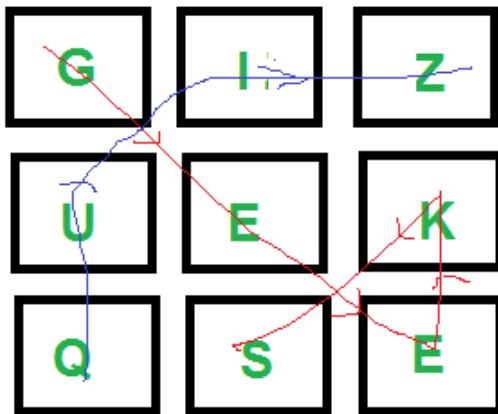
Boggle (Find all possible words in a board of characters) Set 1 - GeeksforGeeks

Given a dictionary, a method to do lookup in dictionary and a M x N board where every cell has one character. Find all possible words that can be formed by a sequence of adjacent characters. Note that we can move to any of 8 adjacent characters, but a word should not have multiple instances of same cell.

Example:

```
Input: dictionary[] = {"GEEKS", "FOR", "QUIZ", "GO"};
       boggle[][]   = {{'G','I','Z'},
                      {'U','E','K'},
                      {'Q','S','E'}};
       isWord(str): returns true if str is present in dictionary
                     else false.
```

```
Output: Following words of dictionary are present
        GEEKS
        QUIZ
```



The idea is to consider every character as a starting character and find all words starting with it. All words starting from a character can be found using [Depth First Traversal](#). We do depth first traversal starting from every cell. We keep track of visited cells to make sure that a cell is considered only once in a word.

```
// C++ program for Boggle game
#include<iostream>
#include<cstring>
using namespace std;

#define M 3
#define N 3

// Let the given dictionary be following
string dictionary[] = {"GEEKS", "FOR", "QUIZ", "GO"};
int n = sizeof(dictionary)/sizeof(dictionary[0]);

// A given function to check if a given string is present in
// dictionary. The implementation is naive for simplicity. As
// per the question dictionary is given to us.
bool isWord(string &str)
{
    // Linearly search all words
    for (int i=0; i<n; i++)
        if (str.compare(dictionary[i]) == 0)
            return true;
    return false;
}

// A recursive function to print all words present on boggle
void findWordsUtil(char boggle[M][N], bool visited[M][N], int i,
                    int j, string &str)
{
```

```

// Mark current cell as visited and append current character
// to str
visited[i][j] = true;
str = str + boggle[i][j];

// If str is present in dictionary, then print it
if (isWord(str))
    cout << str << endl;

// Traverse 8 adjacent cells of boggle[i][j]
for (int row=i-1; row<=i+1 && row<M; row++)
    for (int col=j-1; col<=j+1 && col<N; col++)
        if (row>=0 && col>=0 && !visited[row][col])
            findWordsUtil(boggle,visited, row, col, str);

// Erase current character from string and mark visited
// of current cell as false
str.erase(str.length()-1);
visited[i][j] = false;
}

// Prints all words present in dictionary.
void findWords(char boggle[M][N])
{
    // Mark all characters as not visited
    bool visited[M][N] = {{false}};

    // Initialize current string
    string str = "";

    // Consider every character and look for all words
    // starting with this character
    for (int i=0; i<M; i++)
        for (int j=0; j<N; j++)
            findWordsUtil(boggle, visited, i, j, str);
}

// Driver program to test above function
int main()
{
    char boggle[M][N] = {{'G','I','Z'},
                         {'U','E','K'},
                         {'Q','S','E'}};

    cout << "Following words of dictionary are present\n";
    findWords(boggle);
    return 0;
}

```

Output:

```
Following words of dictionary are present
GEEKS
QUIZ
```

Note that the above solution may print the same word multiple times. For example, if we add “SEEK” to the dictionary, it is printed multiple times. To avoid this, we can use hashing to keep track of all printed words.

In below set 2, we have discussed Trie based optimized solution:

Boggle Set 2 (Using Trie)

This article is contributed by **Rishabh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

<https://www.geeksforgeeks.org/boggle-find-possible-words-board-characters/>

Chapter 87

Boggle Set 2 (Using Trie)

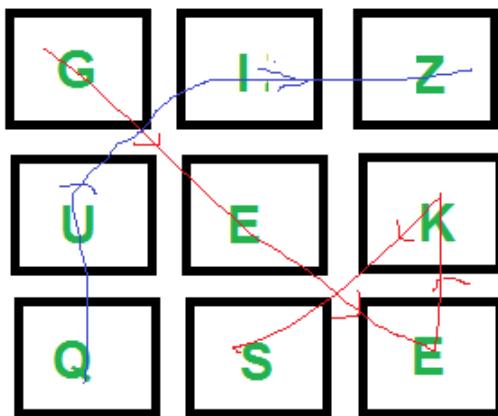
Boggle Set 2 (Using Trie) - GeeksforGeeks

Given a dictionary, a method to do lookup in dictionary and a M x N board where every cell has one character. Find all possible words that can be formed by a sequence of adjacent characters. Note that we can move to any of 8 adjacent characters, but a word should not have multiple instances of same cell.

Example:

```
Input: dictionary[] = {"GEEKS", "FOR", "QUIZ", "GO"};
       boggle[][]   = {{'G','I','Z'},
                      {'U','E','K'},
                      {'Q','S','E'}};
       isWord(str): returns true if str is present in dictionary
                     else false.

Output: Following words of the dictionary are present
        GEEKS
        QUIZ
```



We have discussed a Graph DFS based solution in below post.

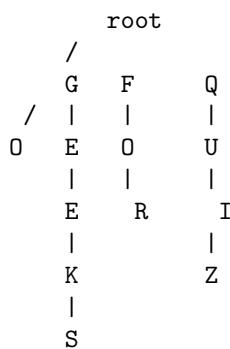
[Boggle \(Find all possible words in a board of characters\) Set 1](#)

Here we discuss a [Trie](#) based solution which is better than DFS based solution.

Given Dictionary `dictionary[] = {"GEEKS", "FOR", "QUIZ", "GO"}`

1. Create an Empty trie and insert all words of given dictionary into trie

After insertion, Trie looks like (leaf nodes are in RED)



2. After that we have pick only those character in `boggle[][]` which are child of root of Trie
Let for above we pick 'G' `boggle[0][0]` , 'Q' `boggle[2][0]` (they both are present in boggle matrix)

3. search a word in a trie which start with character that we pick in step 2

- 1) Create bool visited boolean matrix (`Visited[M][N] = false`)
- 2) Call `SearchWord()` for every cell (i, j) which has one of the first characters of dictionary words. In above example, we have 'G' and 'Q' as first characters.

`SearchWord(Trie *root, i, j, visited[] [N])`

```
if root->leaf == true
    print word

if we have seen this element first time then make it visited.
visited[i][j] = true
do
    traverse all child of current root
    k goes (0 to 26 ) [there are only 26 Alphabet]
    add current char and search for next character

    find next character which is adjacent to boggle[i][j]
    they are 8 adjacent cells of boggle[i][j] (i+1, j+1),
    (i+1, j) (i-1, j) and so on.

make it unvisited visited[i][j] = false
```

Below is the implementation of above idea
C++

```
// C++ program for Boggle game
#include<bits/stdc++.h>
using namespace std;

// Converts key current character into index
// use only 'A' through 'Z'
#define char_int(c) ((int)c - (int)'A')

// Alphabet size
#define SIZE (26)

#define M 3
#define N 3

// trie Node
struct TrieNode
{
    TrieNode *Child[SIZE];

    // isLeaf is true if the node represents
    // end of a word
    bool leaf;
};

// Returns new trie node (initialized to NULLs)
TrieNode *getNode()
{
    TrieNode *newNode = new TrieNode;
    newNode->leaf = false;
```

```

        for (int i =0 ; i< SIZE ; i++)
            newNode->Child[i] = NULL;
        return newNode;
    }

// If not present, inserts a key into the trie
// If the key is a prefix of trie node, just
// marks leaf node
void insert(TrieNode *root, char *Key)
{
    int n = strlen(Key);
    TrieNode * pChild = root;

    for (int i=0; i<n; i++)
    {
        int index = char_int(Key[i]);

        if (pChild->Child[index] == NULL)
            pChild->Child[index] = getNode();

        pChild = pChild->Child[index];
    }

    // make last node as leaf node
    pChild->leaf = true;
}

// function to check that current location
// (i and j) is in matrix range
bool isSafe(int i, int j, bool visited[M][N])
{
    return (i >=0 && i < M && j >=0 &&
            j < N && !visited[i][j]);
}

// A recursive function to print all words present on boggle
void searchWord(TrieNode *root, char boggle[M][N], int i,
                int j, bool visited[][][N], string str)
{
    // if we found word in trie / dictionary
    if (root->leaf == true)
        cout << str << endl ;

    // If both I and j in range and we visited
    // that element of matrix first time
    if (isSafe(i, j, visited))
    {
        // make it visited

```

```

visited[i][j] = true;

// traverse all childs of current root
for (int K =0; K < SIZE; K++)
{
    if (root->Child[K] != NULL)
    {
        // current character
        char ch = (char)K + (char)'A' ;

        // Recursively search reaming character of word
        // in trie for all 8 adjacent cells of boggle[i][j]
        if (isSafe(i+1,j+1,visited) && boggle[i+1][j+1] == ch)
            searchWord(root->Child[K],boggle,i+1,j+1,visited,str+ch);
        if (isSafe(i, j+1,visited) && boggle[i][j+1] == ch)
            searchWord(root->Child[K],boggle,i, j+1,visited,str+ch);
        if (isSafe(i-1,j+1,visited) && boggle[i-1][j+1] == ch)
            searchWord(root->Child[K],boggle,i-1, j+1,visited,str+ch);
        if (isSafe(i+1,j, visited) && boggle[i+1][j] == ch)
            searchWord(root->Child[K],boggle,i+1, j,visited,str+ch);
        if (isSafe(i+1,j-1,visited) && boggle[i+1][j-1] == ch)
            searchWord(root->Child[K],boggle,i+1, j-1,visited,str+ch);
        if (isSafe(i, j-1,visited)&& boggle[i][j-1] == ch)
            searchWord(root->Child[K],boggle,i, j-1,visited,str+ch);
        if (isSafe(i-1,j-1,visited) && boggle[i-1][j-1] == ch)
            searchWord(root->Child[K],boggle,i-1, j-1,visited,str+ch);
        if (isSafe(i-1, j,visited) && boggle[i-1][j] == ch)
            searchWord(root->Child[K],boggle,i-1, j, visited,str+ch);
    }
}

// make current element unvisited
visited[i][j] = false;
}

// Prints all words present in dictionary.
void findWords(char boggle[M][N], TrieNode *root)
{
    // Mark all characters as not visited
    bool visited[M][N];
    memset(visited,false,sizeof(visited));

    TrieNode *pChild = root ;

    string str = "";

    // traverse all matrix elements
}

```

```

for (int i = 0 ; i < M; i++)
{
    for (int j = 0 ; j < N ; j++)
    {
        // we start searching for word in dictionary
        // if we found a character which is child
        // of Trie root
        if (pChild->Child[char_int(boggle[i][j])] )
        {
            str = str+boggle[i][j];
            searchWord(pChild->Child[char_int(boggle[i][j])],
                       boggle, i, j, visited, str);
            str = "";
        }
    }
}

//Driver program to test above function
int main()
{
    // Let the given dictionary be following
    char *dictionary[] = {"GEEKS", "FOR", "QUIZ", "GEE"};

    // root Node of trie
    TrieNode *root = getNode();

    // insert all words of dictionary into trie
    int n = sizeof(dictionary)/sizeof(dictionary[0]);
    for (int i=0; i<n; i++)
        insert(root, dictionary[i]);

    char boggle[M][N] = {{'G','I','Z'},
                        {'U','E','K'},
                        {'Q','S','E'}};
}

findWords(boggle, root);

return 0;
}

```

Java

```

// Java program for Boggle game
public class Boggle {

    // Alphabet size

```

```

static final int SIZE = 26;

static final int M = 3;
static final int N = 3;

// trie Node
static class TrieNode
{
    TrieNode[] Child = new TrieNode[SIZE];

    // isLeaf is true if the node represents
    // end of a word
    boolean leaf;

    //constructor
    public TrieNode() {
        leaf = false;
        for (int i = 0 ; i < SIZE ; i++)
            Child[i] = null;
    }
}

// If not present, inserts a key into the trie
// If the key is a prefix of trie node, just
// marks leaf node
static void insert(TrieNode root, String Key)
{
    int n = Key.length();
    TrieNode pChild = root;

    for (int i=0; i<n; i++)
    {
        int index = Key.charAt(i) - 'A';

        if (pChild.Child[index] == null)
            pChild.Child[index] = new TrieNode();

        pChild = pChild.Child[index];
    }

    // make last node as leaf node
    pChild.leaf = true;
}

// function to check that current location
// (i and j) is in matrix range
static boolean isSafe(int i, int j, boolean visited[][])
{

```

```

        return (i >=0 && i < M && j >=0 &&
               j < N && !visited[i][j]);
    }

    // A recursive function to print all words present on boggle
    static void searchWord(TrieNode root, char boggle[][][], int i,
                           int j, boolean visited[][][], String str)
    {
        // if we found word in trie / dictionary
        if (root.leaf == true)
            System.out.println(str);

        // If both I and j in range and we visited
        // that element of matrix first time
        if (isSafe(i, j, visited))
        {
            // make it visited
            visited[i][j] = true;

            // traverse all child of current root
            for (int K =0; K < SIZE; K++)
            {
                if (root.Child[K] != null)
                {
                    // current character
                    char ch = (char) (K + 'A') ;

                    // Recursively search reaming character of word
                    // in trie for all 8 adjacent cells of
                    // boggle[i][j]
                    if (isSafe(i+1,j+1,visited) && boggle[i+1][j+1]
                        == ch)
                        searchWord(root.Child[K],boggle,i+1,j+1,
                                  visited,str+ch);
                    if (isSafe(i, j+1,visited) && boggle[i][j+1]
                        == ch)
                        searchWord(root.Child[K],boggle,i, j+1,
                                  visited,str+ch);
                    if (isSafe(i-1,j+1,visited) && boggle[i-1][j+1]
                        == ch)
                        searchWord(root.Child[K],boggle,i-1, j+1,
                                  visited,str+ch);
                    if (isSafe(i+1,j, visited) && boggle[i+1][j]
                        == ch)
                        searchWord(root.Child[K],boggle,i+1, j,
                                  visited,str+ch);
                    if (isSafe(i+1,j-1,visited) && boggle[i+1][j-1]
                        == ch)

```

```

        searchWord(root.Child[K], boggle, i+1, j-1,
                   visited,str+ch);
        if (isSafe(i, j-1,visited)&& boggle[i][j-1]
                        == ch)
            searchWord(root.Child[K], boggle, i, j-1,
                       visited,str+ch);
        if (isSafe(i-1,j-1,visited) && boggle[i-1][j-1]
                        == ch)
            searchWord(root.Child[K], boggle, i-1, j-1,
                       visited,str+ch);
        if (isSafe(i-1, j,visited) && boggle[i-1][j]
                        == ch)
            searchWord(root.Child[K], boggle, i-1, j,
                       visited,str+ch);
    }
}

// make current element unvisited
visited[i][j] = false;
}
}

// Prints all words present in dictionary.
static void findWords(char boggle[][] , TrieNode root)
{
    // Mark all characters as not visited
    boolean[][] visited = new boolean[M][N];
    TrieNode pChild = root ;

    String str = "";

    // traverse all matrix elements
    for (int i = 0 ; i < M; i++)
    {
        for (int j = 0 ; j < N ; j++)
        {
            // we start searching for word in dictionary
            // if we found a character which is child
            // of Trie root
            if (pChild.Child[(boggle[i][j]) - 'A'] != null)
            {
                str = str+boggle[i][j];
                searchWord(pChild.Child[(boggle[i][j]) - 'A'],
                           boggle, i, j, visited, str);
                str = "";
            }
        }
    }
}

```

```
}

// Driver program to test above function
public static void main(String args[])
{
    // Let the given dictionary be following
    String dictionary[] = {"GEEKS", "FOR", "QUIZ", "GEE"};

    // root Node of trie
    TrieNode root = new TrieNode();

    // insert all words of dictionary into trie
    int n = dictionary.length;
    for (int i=0; i<n; i++)
        insert(root, dictionary[i]);

    char boggle[][] = {{'G','I','Z'},
                      {'U','E','K'},
                      {'Q','S','E'}
                     };

    findWords(boggle, root);
}

// This code is contributed by Sumit Ghosh
```

Output:

GEE, GEEKS, QUIZ

Source

<https://www.geeksforgeeks.org/boggle-set-2-using-trie/>

Chapter 88

Boruvka's algorithm for Minimum Spanning Tree

Boruvka's algorithm for Minimum Spanning Tree - GeeksforGeeks

Following two algorithms are generally taught for Minimum Spanning Tree (MST) problem.

[Prim's algorithm](#)

[Kruskal's algorithm](#)

There is a third algorithm called [Boruvka's algorithm](#) for MST which (like the above two) is also Greedy algorithm. The [Boruvka's algorithm](#) is the oldest minimum spanning tree algorithm was discovered by Boruvka in 1926, long before computers even existed. The algorithm was published as a method of constructing an efficient electricity network. See following links for the working and applications of the algorithm.

Sources:

http://en.wikipedia.org/wiki/Bor%C5%AFvka%27s_algorithm

Source

<https://www.geeksforgeeks.org/g-fact-83/>

Chapter 89

Boruvka's algorithm Greedy Algo-9

Boruvka's algorithm Greedy Algo-9 - GeeksforGeeks

We have discussed following topics on Minimum Spanning Tree.

[Applications of Minimum Spanning Tree Problem](#)

[Kruskal's Minimum Spanning Tree Algorithm](#)

[Prim's Minimum Spanning Tree Algorithm](#)

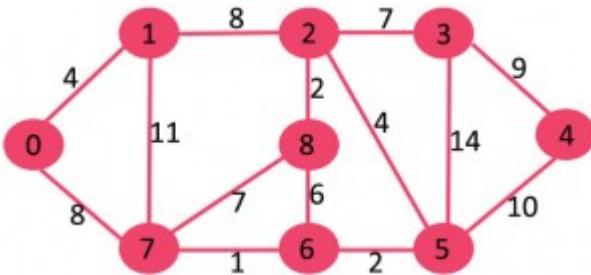
In this post, Boruvka's algorithm is discussed. Like Prim's and Kruskal's, Boruvka's algorithm is also a Greedy algorithm. Below is complete algorithm.

- 1) Input is a connected, weighted and directed graph.
- 2) Initialize all vertices as individual components (or sets).
- 3) Initialize MST as empty.
- 4) While there are more than one components, do following for each component.
 - a) Find the closest weight edge that connects this component to any other component.
 - b) Add this closest edge to MST if not already added.
- 5) Return MST.

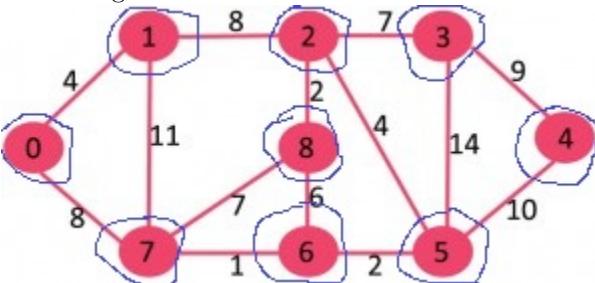
Below is the idea behind above algorithm (The idea is same as [Prim's MST algorithm](#)).

A spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a Spanning Tree. And they must be connected with the minimum weight edge to make it a Minimum Spanning Tree.

Let us understand the algorithm with below example.



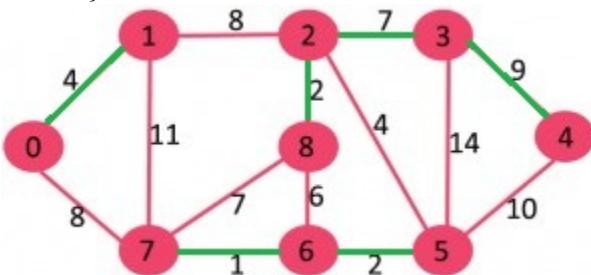
Initially MST is empty. Every vertex is single component as highlighted in blue color in below diagram.



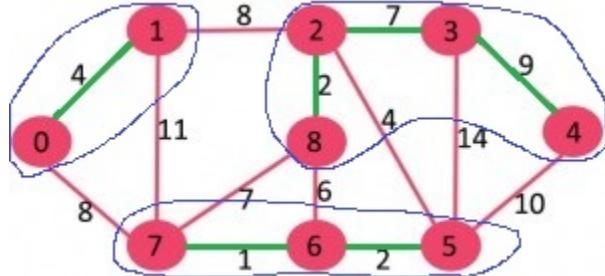
For every component, find the cheapest edge that connects it to some other component.

Component	Cheapest Edge that connects it to some other component
{0}	0-1
{1}	0-1
{2}	2-8
{3}	2-3
{4}	3-4
{5}	5-6
{6}	6-7
{7}	6-7
{8}	2-8

The cheapest edges are highlighted with green color. Now MST becomes {0-1, 2-8, 2-3, 3-4, 5-6, 6-7}.



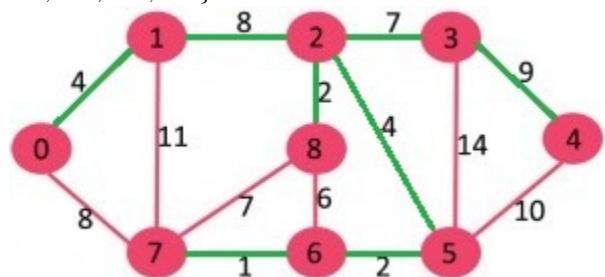
After above step, components are $\{\{0,1\}, \{2,3,4,8\}, \{5,6,7\}\}$. The components are encircled with blue color.



We again repeat the step, i.e., for every component, find the cheapest edge that connects it to some other component.

Component	Cheapest Edge that connects it to some other component
$\{0,1\}$	1-2 (or 0-7)
$\{2,3,4,8\}$	2-5
$\{5,6,7\}$	2-5

The cheapest edges are highlighted with green color. Now MST becomes $\{0-1, 2-8, 2-3, 3-4, 5-6, 6-7, 1-2, 2-5\}$



At this stage, there is only one component $\{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ which has all edges. Since there is only one component left, we stop and return MST.

Implementation:

Below is implementation of above algorithm. The input graph is represented as a collection of edges and [union-find data structure](#) is used to keep track of components.

C/C++

```
// Boruvka's algorithm to find Minimum Spanning
// Tree of a given connected, undirected and
// weighted graph
#include <stdio.h>

// a structure to represent a weighted edge in graph
```

```

struct Edge
{
    int src, dest, weight;
};

// a structure to represent a connected, undirected
// and weighted graph as a collection of edges.
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    // Since the graph is undirected, the edge
    // from src to dest is also edge from dest
    // to src. Both are counted as 1 edge here.
    Edge* edge;
};

// A structure to represent a subset for union-find
struct subset
{
    int parent;
    int rank;
};

// Function prototypes for union-find (These functions are defined
// after boruvkaMST() )
int find(struct subset subsets[], int i);
void Union(struct subset subsets[], int x, int y);

// The main function for MST using Boruvka's algorithm
void boruvkaMST(struct Graph* graph)
{
    // Get data of given graph
    int V = graph->V, E = graph->E;
    Edge *edge = graph->edge;

    // Allocate memory for creating V subsets.
    struct subset *subsets = new subset[V];

    // An array to store index of the cheapest edge of
    // subset. The stored index for indexing array 'edge[]'
    int *cheapest = new int[V];

    // Create V subsets with single elements
    for (int v = 0; v < V; ++v)
    {

```

```

        subsets[v].parent = v;
        subsets[v].rank = 0;
        cheapest[v] = -1;
    }

    // Initially there are V different trees.
    // Finally there will be one tree that will be MST
    int numTrees = V;
    int MSTweight = 0;

    // Keep combining components (or sets) until all
    // componentes are not combined into single MST.
    while (numTrees > 1)
    {
        // Everytime initialize cheapest array
        for (int v = 0; v < V; ++v)
        {
            cheapest[v] = -1;
        }

        // Traverse through all edges and update
        // cheapest of every component
        for (int i=0; i<E; i++)
        {
            // Find components (or sets) of two corners
            // of current edge
            int set1 = find(subsets, edge[i].src);
            int set2 = find(subsets, edge[i].dest);

            // If two corners of current edge belong to
            // same set, ignore current edge
            if (set1 == set2)
                continue;

            // Else check if current edge is closer to previous
            // cheapest edges of set1 and set2
            else
            {
                if (cheapest[set1] == -1 ||
                    edge[cheapest[set1]].weight > edge[i].weight)
                    cheapest[set1] = i;

                if (cheapest[set2] == -1 ||
                    edge[cheapest[set2]].weight > edge[i].weight)
                    cheapest[set2] = i;
            }
        }
    }
}

```

```

// Consider the above picked cheapest edges and add them
// to MST
for (int i=0; i<V; i++)
{
    // Check if cheapest for current set exists
    if (cheapest[i] != -1)
    {
        int set1 = find(subsets, edge[cheapest[i]].src);
        int set2 = find(subsets, edge[cheapest[i]].dest);

        if (set1 == set2)
            continue;
        MSTweight += edge[cheapest[i]].weight;
        printf("Edge %d-%d included in MST\n",
               edge[cheapest[i]].src, edge[cheapest[i]].dest);

        // Do a union of set1 and set2 and decrease number
        // of trees
        Union(subsets, set1, set2);
        numTrees--;
    }
}
printf("Weight of MST is %d\n", MSTweight);
return;
}

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}

// A utility function to find set of an element i
// (uses path compression technique)
int find(struct subset subsets[], int i)
{
    // find root and make root as parent of i
    // (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent =
            find(subsets, subsets[i].parent);
}

```

```

        return subsets[i].parent;
    }

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(struct subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high
    // rank tree (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and
    // increment its rank by one
    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Driver program to test above functions
int main()
{
    /* Let us create following weighted graph
       10
       0-----1
       | \     |
       6|   5\   |15
       |     \  |
       2-----3
       4      */
    int V = 4; // Number of vertices in graph
    int E = 5; // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = 10;

    // add edge 0-2
}

```

```

graph->edge[1].src = 0;
graph->edge[1].dest = 2;
graph->edge[1].weight = 6;

// add edge 0-3
graph->edge[2].src = 0;
graph->edge[2].dest = 3;
graph->edge[2].weight = 5;

// add edge 1-3
graph->edge[3].src = 1;
graph->edge[3].dest = 3;
graph->edge[3].weight = 15;

// add edge 2-3
graph->edge[4].src = 2;
graph->edge[4].dest = 3;
graph->edge[4].weight = 4;

boruvkaMST(graph);

return 0;
}

// Thanks to Anukul Chand for modifying above code.

```

Python

```

# Boruvka's algorithm to find Minimum Spanning
# Tree of a given connected, undirected and weighted graph

from collections import defaultdict

#Class to represent a graph
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices
        self.graph = [] # default dictionary to store graph

    # function to add an edge to graph
    def addEdge(self,u,v,w):
        self.graph.append([u,v,w])

    # A utility function to find set of an element i
    # (uses path compression technique)
    def find(self, parent, i):

```

```

if parent[i] == i:
    return i
return self.find(parent, parent[i])

# A function that does union of two sets of x and y
# (uses union by rank)
def union(self, parent, rank, x, y):
    xroot = self.find(parent, x)
    yroot = self.find(parent, y)

    # Attach smaller rank tree under root of high rank tree
    # (Union by Rank)
    if rank[xroot] < rank[yroot]:
        parent[xroot] = yroot
    elif rank[xroot] > rank[yroot]:
        parent[yroot] = xroot
    #If ranks are same, then make one as root and increment
    # its rank by one
    else :
        parent[yroot] = xroot
        rank[xroot] += 1

# The main function to construct MST using Kruskal's algorithm
def boruvkaMST(self):
    parent = [] ; rank = [];

    # An array to store index of the cheapest edge of
    # subset. It store [u,v,w] for each component
    cheapest =[]

    # Initially there are V different trees.
    # Finally there will be one tree that will be MST
    numTrees = self.V
    MSTweight = 0

    # Create V subsets with single elements
    for node in range(self.V):
        parent.append(node)
        rank.append(0)
        cheapest =[-1] * self.V

    # Keep combining components (or sets) until all
    # components are not combined into single MST

    while numTrees > 1:

        # Traverse through all edges and update
        # cheapest of every component

```

```

for i in range(len(self.graph)):

    # Find components (or sets) of two corners
    # of current edge
    u,v,w = self.graph[i]
    set1 = self.find(parent, u)
    set2 = self.find(parent ,v)

    # If two corners of current edge belong to
    # same set, ignore current edge. Else check if
    # current edge is closer to previous
    # cheapest edges of set1 and set2
    if set1 != set2:

        if cheapest[set1] == -1 or cheapest[set1][2] > w :
            cheapest[set1] = [u,v,w]

        if cheapest[set2] == -1 or cheapest[set2][2] > w :
            cheapest[set2] = [u,v,w]

    # Consider the above picked cheapest edges and add them
    # to MST
    for node in range(self.V):

        #Check if cheapest for current set exists
        if cheapest[node] != -1:
            u,v,w = cheapest[node]
            set1 = self.find(parent, u)
            set2 = self.find(parent ,v)

            if set1 != set2 :
                MSTweight += w
                self.union(parent, rank, set1, set2)
                print ("Edge %d-%d with weight %d included in MST" % (u,v,w))
                numTrees = numTrees - 1

        #reset cheapest array
        cheapest = [-1] * self.V

    print ("Weight of MST is %d" % MSTweight)

g = Graph(4)
g.addEdge(0, 1, 10)
g.addEdge(0, 2, 6)
g.addEdge(0, 3, 5)

```

```
g.addEdge(1, 3, 15)
g.addEdge(2, 3, 4)

g.boruvkaMST()

#This code is contributed by Neelam Yadav
```

Output:

```
Edge 0-3 included in MST
Edge 0-1 included in MST
Edge 2-3 included in MST
Weight of MST is 19
```

Interesting Facts about Boruvka's algorithm:

- 1) Time Complexity of Boruvka's algorithm is $O(E \log V)$ which is same as Kruskal's and Prim's algorithms.
- 2) Boruvka's algorithm is used as a step in a [faster randomized algorithm that works in linear time \$O\(E\)\$](#) .
- 3) Boruvka's algorithm is the oldest minimum spanning tree algorithm was discovered by Boruvka in 1926, long before computers even existed. The algorithm was published as a method of constructing an efficient electricity network.

Exercise:

The above code assumes that input graph is connected and it fails if a disconnected graph is given. Extend the above algorithm so that it works for a disconnected graph also and produces a forest.

References:

http://en.wikipedia.org/wiki/Bor%C5%AFvka%27s_algorithm

Improved By : [Cyberfreak](#)

Source

<https://www.geeksforgeeks.org/boruvkas-algorithm-greedy-algo-9/>

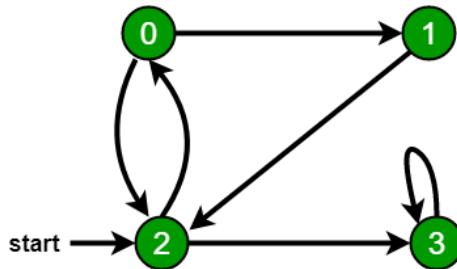
Chapter 90

Breadth First Search or BFS for a Graph

Breadth First Search or BFS for a Graph - GeeksforGeeks

[Breadth First Traversal \(or Search\)](#) for a graph is similar to Breadth First Traversal of a tree (See method 2 of [this post](#)). The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth First Traversal of the following graph is 2, 0, 3, 1.



Following are the implementations of simple Breadth First Traversal from a given source.

The implementation uses [adjacency list representation](#) of graphs. [STL's list container](#) is used to store lists of adjacent nodes and queue of nodes needed for BFS traversal.

C++

```
// Program to print BFS traversal from a given
// source vertex. BFS(int s) traverses vertices
// reachable from s.
```

```
#include<iostream>
#include <list>

using namespace std;

// This class represents a directed graph using
// adjacency list representation
class Graph
{
    int V;      // No. of vertices

    // Pointer to an array containing adjacency
    // lists
    list<int> *adj;
public:
    Graph(int V); // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // prints BFS traversal from a given source s
    void BFS(int s);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);
```

```
// 'i' will be used to get all adjacent
// vertices of a vertex
list<int>::iterator i;

while(!queue.empty())
{
    // Dequeue a vertex from queue and print it
    s = queue.front();
    cout << s << " ";
    queue.pop_front();

    // Get all adjacent vertices of the dequeued
    // vertex s. If a adjacent has not been visited,
    // then mark it visited and enqueue it
    for (i = adj[s].begin(); i != adj[s].end(); ++i)
    {
        if (!visited[*i])
        {
            visited[*i] = true;
            queue.push_back(*i);
        }
    }
}

// Driver program to test methods of graph class
int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Breadth First Traversal "
        << "(starting from vertex 2) \n";
    g.BFS(2);

    return 0;
}
```

Java

```
// Java program to print BFS traversal from a given source vertex.
```

```
// BFS(int s) traverses vertices reachable from s.
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    private int V;    // No. of vertices
    private LinkedList<Integer> adj[]; //Adjacency Lists

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    // Function to add an edge into the graph
    void addEdge(int v,int w)
    {
        adj[v].add(w);
    }

    // prints BFS traversal from a given source s
    void BFS(int s)
    {
        // Mark all the vertices as not visited(By default
        // set as false)
        boolean visited[] = new boolean[V];

        // Create a queue for BFS
        LinkedList<Integer> queue = new LinkedList<Integer>();

        // Mark the current node as visited and enqueue it
        visited[s]=true;
        queue.add(s);

        while (queue.size() != 0)
        {
            // Dequeue a vertex from queue and print it
            s = queue.poll();
            System.out.print(s+" ");

            // Get all adjacent vertices of the dequeued vertex s
            // If a adjacent has not been visited, then mark it
        }
    }
}
```

```
// visited and enqueue it
Iterator<Integer> i = adj[s].listIterator();
while (i.hasNext())
{
    int n = i.next();
    if (!visited[n])
    {
        visited[n] = true;
        queue.add(n);
    }
}
}

// Driver method to
public static void main(String args[])
{
    Graph g = new Graph(4);

    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    System.out.println("Following is Breadth First Traversal "+
                       "(starting from vertex 2)");

    g.BFS(2);
}
}

// This code is contributed by Aakash Hasija
```

Python3

```
# Python3 Program to print BFS traversal
# from a given source vertex. BFS(int s)
# traverses vertices reachable from s.
from collections import defaultdict

# This class represents a directed graph
# using adjacency list representation
class Graph:

    # Constructor
    def __init__(self):
```

```
# default dictionary to store graph
self.graph = defaultdict(list)

# function to add an edge to graph
def addEdge(self,u,v):
    self.graph[u].append(v)

# Function to print a BFS of graph
def BFS(self, s):

    # Mark all the vertices as not visited
    visited = [False] * (len(self.graph))

    # Create a queue for BFS
    queue = []

    # Mark the source node as
    # visited and enqueue it
    queue.append(s)
    visited[s] = True

    while queue:

        # Dequeue a vertex from
        # queue and print it
        s = queue.pop(0)
        print (s, end = " ")

        # Get all adjacent vertices of the
        # dequeued vertex s. If a adjacent
        # has not been visited, then mark it
        # visited and enqueue it
        for i in self.graph[s]:
            if visited[i] == False:
                queue.append(i)
                visited[i] = True

# Driver code

# Create a graph given in
# the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)
```

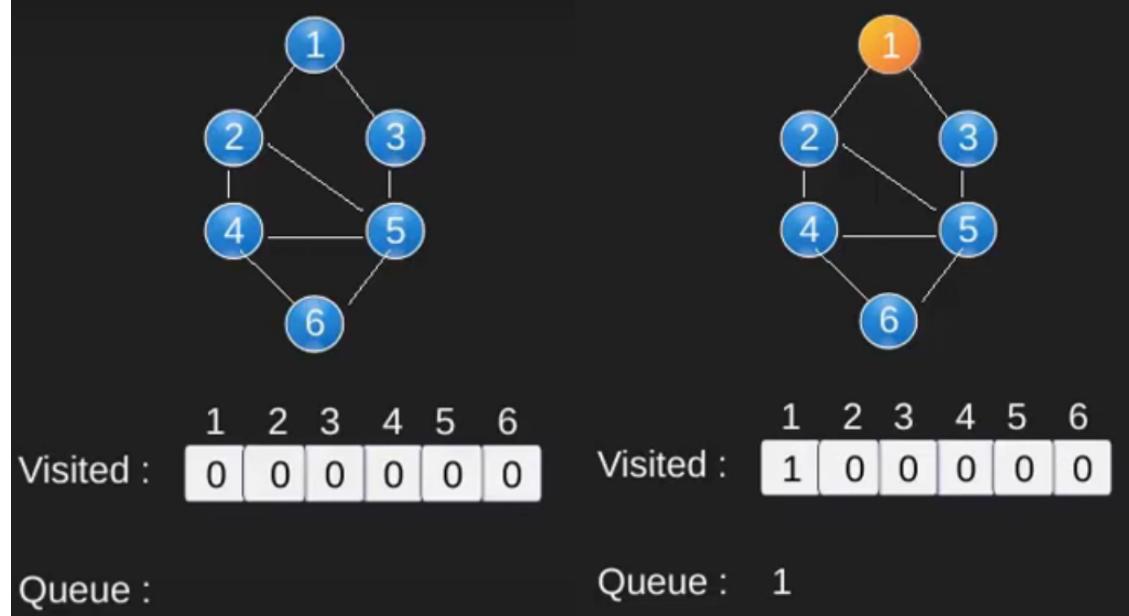
```
print ("Following is Breadth First Traversal"
      " (starting from vertex 2)")
g.BFS(2)

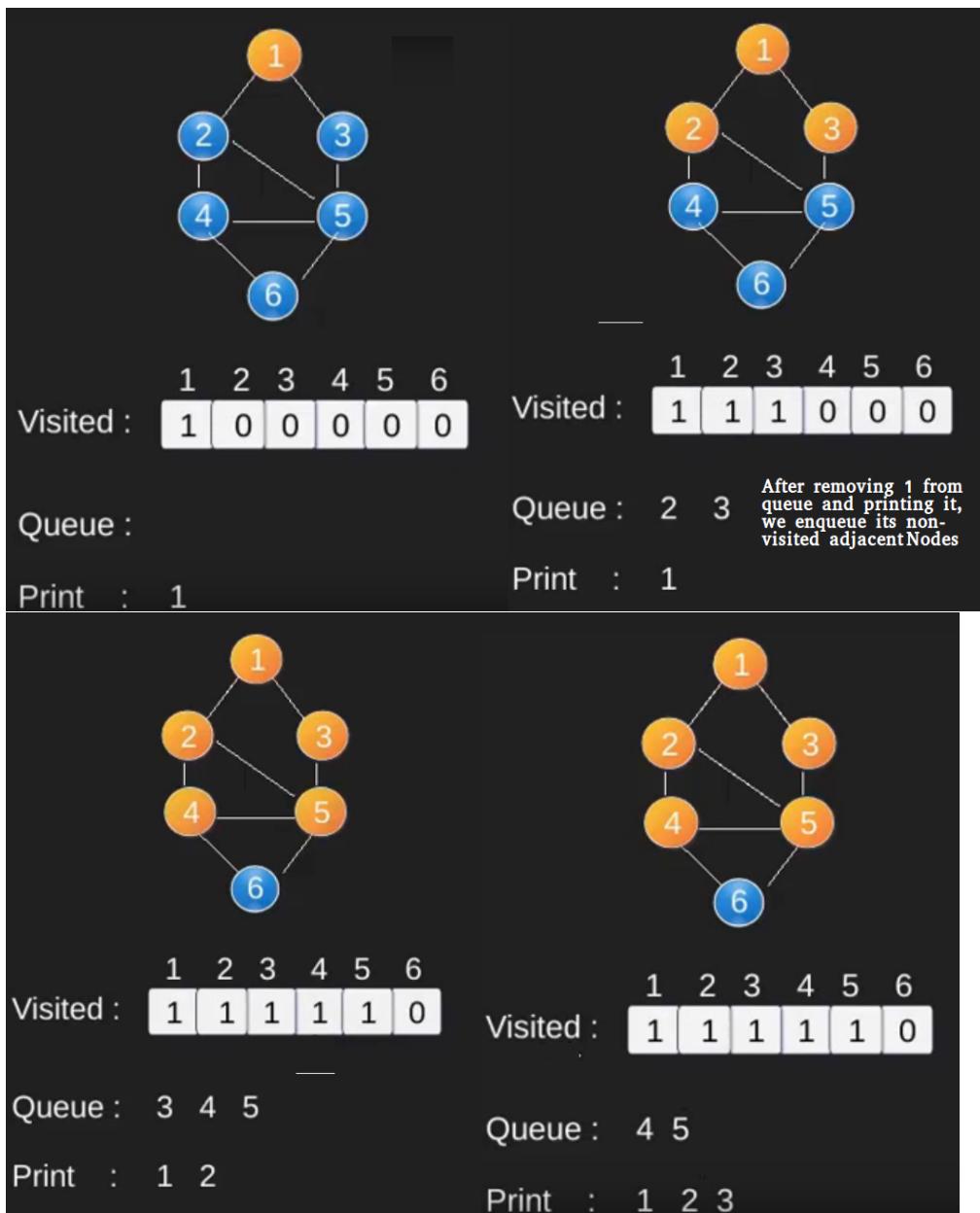
# This code is contributed by Neelam Yadav
```

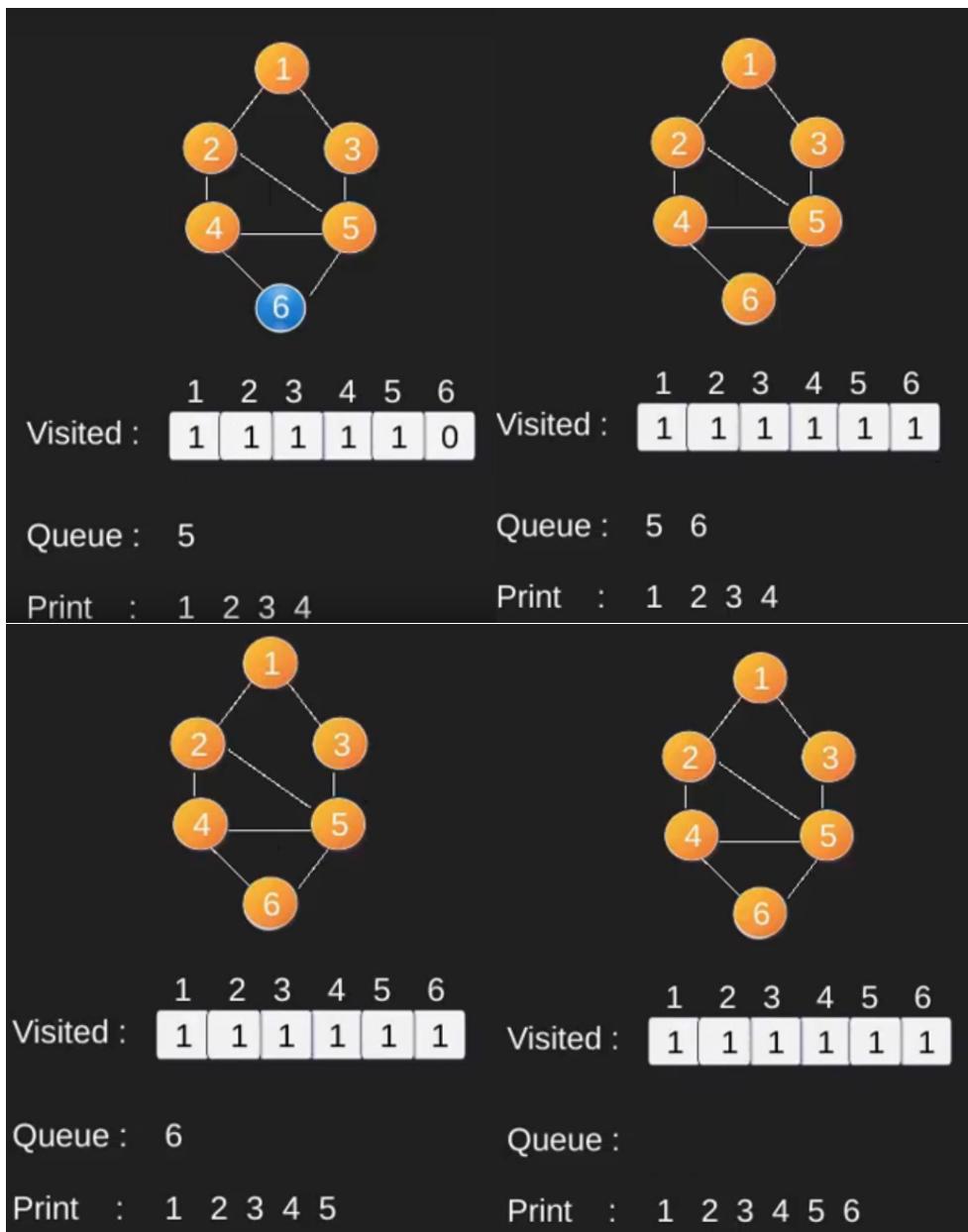
Output:

```
Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1
```

Illustration :







Note that the above code traverses only the vertices reachable from a given source vertex. All the vertices may not be reachable from a given vertex (example Disconnected graph). To print all the vertices, we can modify the BFS function to do traversal starting from all nodes one by one (Like the [DFS modified version](#)) .

Time Complexity: $O(V+E)$ where V is number of vertices in the graph and E is number of edges in the graph.

You may like to see below also :

- Recent Articles on BFS
- Depth First Traversal
- Applications of Breadth First Traversal
- Applications of Depth First Search

Source

<https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/>

Chapter 91

Calculate number of nodes between two vertices in an acyclic Graph by Disjoint Union method

Calculate number of nodes between two vertices in an acyclic Graph by Disjoint Union method - GeeksforGeeks

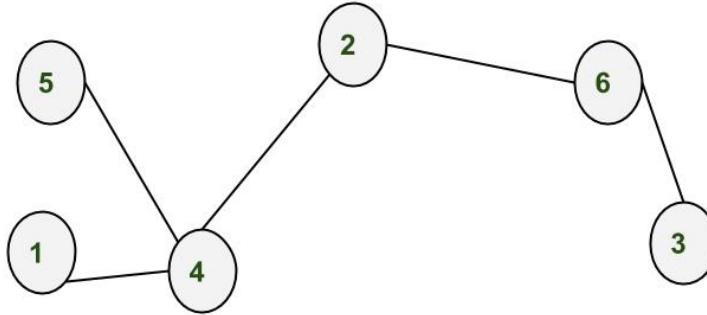
Given a connected acyclic graph and a source vertex and a destination vertex, your task is to count the number of vertices between the given source and destination vertex by **Disjoint Union Method**.

Examples:

```
Input : 1 4  
        4 5  
        4 2  
        2 6  
        6 3  
        1 3
```

```
Output : 3
```

In the input 6 is the total number of vertices labeled from 1 to 6 and next 5 lines are connection between verticies . Please see the figure for more explanation. And in last line 1 is the source vertex and 3 is the destination vertex. From the figure it is clear that there are 3 nodes(4, 2, 6) present between 1 and 3 .



To use disjoint union method we have to first check the parent of each of the node of the given graph. We can use [BFS](#) to traverse through the graph and calculate the parent vertex of each vertices of graph. For example, if we traverse the graph (i.e starts our bfs) from vertex 1 then 1 is the parent of 4 , then 4 is the parent of 5 and 2 , again 2 is the parent of 6 and 6 is the parent of 3 .

Now to calculate number of nodes between the source node and destination node, we have to make a loop that starts with parent of destination node and after every iteration we will update this node with parent of current node , keeping the count of number of iterations. The execution of loop will terminate when we reach the source vertex and the count variable gives the number of nodes in the connected path of source node and destination node.

In the above method the disjoint sets are all the sets with single vertex, and we have used union operation to merge two sets where one contains the parent node and other contains the child node.

Below are implementations of above approach .

C++

```
// C++ program to calculate number
// of nodes between two nodes
#include <bits/stdc++.h>
using namespace std;

// function to calculate no of nodes
// between two nodes
int totalNodes(vector<int> adjac[], int n,
               int x, int y)
{
    // x is the source node and
    // y is destination node

    // visited array take account of
    // the nodes visited through the bfs
```

```
bool visited[n+1] = {0};

// parent array to store each nodes
// parent value
int p[n] ;

queue<int> q;
q.push(x);

// take our first node(x) as first element
// of queue and marked it as
// visited through visited[] array
visited[x] = true;

int m;

// normal bfs method starts
while (!q.empty())
{
    m = q.front() ;
    q.pop();
    for (int i=0; i<adjac[m].size(); ++i)
    {
        int h = adjac[m][i];
        if (!visited[h])
        {
            visited[h] = true;

            // when new node is encountered
            // we assign it's parent value
            // in parent array p
            p[h] = m ;
            q.push(h);
        }
    }
}

// count variable stores the result
int count = 0;

// loop start with parent of y
// till we encountered x
int i = p[y];
while (i != x)
{
    // count increases for counting
    // the nodes
    count++;
}
```

```
i = p[i];
}

return count ;
}

// Driver program to test above function
int main()
{
    // adjacency list for graph
    vector < int > adjac[7];

    // creating graph, keeping length of
    // adjacency list as (1 + no of nodes)
    // as index ranges from (0 to n-1)
    adjac[1].push_back(4);
    adjac[4].push_back(1);
    adjac[5].push_back(4);
    adjac[4].push_back(5);
    adjac[4].push_back(2);
    adjac[2].push_back(4);
    adjac[2].push_back(6);
    adjac[6].push_back(2);
    adjac[6].push_back(3);
    adjac[3].push_back(6);

    cout << totalNodes(adjac, 7, 1, 3);

    return 0;
}
```

Java

```
// Java program to calculate number
// of nodes between two nodes
import java.util.Arrays;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Vector;

public class GFG
{
    // function to calculate no of nodes
    // between two nodes
    static int totalNodes(Vector<Integer> adjac[],
                          int n, int x, int y)
    {
```

```
// x is the source node and
// y is destination node

// visited array take account of
// the nodes visited through the bfs
Boolean visited[] = new Boolean[n + 1];

//filling boolean value with false
Arrays.fill(visited, false);

// parent array to store each nodes
// parent value
int p[] = new int[n];

Queue<Integer> q = new LinkedList<>();
q.add(x);

// take our first node(x) as first element
// of queue and marked it as
// visited through visited[] array
visited[x] = true;

int m;

// normal bfs method starts
while(!q.isEmpty())
{
    m = q.peek();
    q.poll();
    for(int i=0; i < adjac[m].size() ; ++i)
    {

        int h = adjac[m].get(i);

        if(visited[h] != true )
        {
            visited[h] = true;

            // when new node is encountered
            // we assign it's parent value
            // in parent array p
            p[h] = m;
            q.add(h);
        }
    }
}
```

```
// count variable stores the result
int count = 0;

// loop start with parent of y
// till we encountered x
int i = p[y];
while(i != x)
{
    // count increases for counting
    // the nodes
    count++;
    i = p[i];
}
return count;
}

// Driver program to test above function
public static void main(String[] args)
{
    // adjacency list for graph
    Vector<Integer> adjac[] = new Vector[7];

    //Initialzing Vector for each nodes
    for (int i = 0; i < 7; i++)
        adjac[i] = new Vector<>();

    // creating graph, keeping length of
    // adjacency list as (1 + no of nodes)
    // as index ranges from (0 to n-1)
    adjac[1].add(4);
    adjac[4].add(1);
    adjac[5].add(4);
    adjac[4].add(5);
    adjac[4].add(2);
    adjac[2].add(4);
    adjac[2].add(6);
    adjac[6].add(2);
    adjac[6].add(3);
    adjac[3].add(6);

    System.out.println(totalNodes(adjac, 7, 1, 3));
}

// This code is contributed by Sumit Ghosh
```

Output:

3

Time Complexity: $O(n)$, where n is total number of nodes in the graph.

Source

<https://www.geeksforgeeks.org/number-nodes-two-vertices-acyclic-graph-disjoint-union-method/>

Chapter 92

Check for star graph

Check for star graph - GeeksforGeeks

You are given an $n * n$ matrix which represent a graph with n -vertices, check whether the input matrix represent a star graph or not.

Example:

```
Input : Mat[] [] = {{0, 1, 0},  
                    {1, 0, 1},  
                    {0, 1, 0}}
```

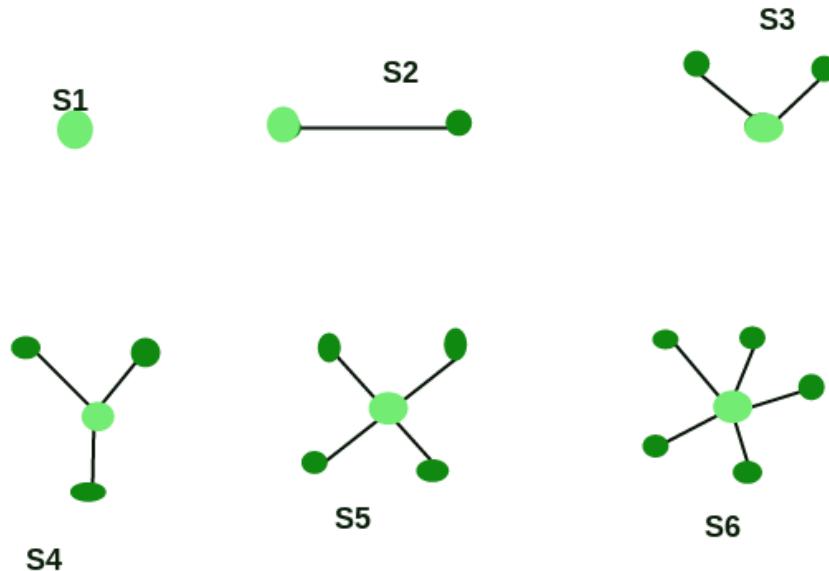
Output : Star graph

```
Input : Mat[] [] = {{0, 1, 0},  
                    {1, 1, 1},  
                    {0, 1, 0}}
```

Output : Not a Star graph

Star graph : Star graph is a special type of graph in which $n-1$ vertices have degree 1 and a single vertex have degree $n - 1$. This looks like that $n - 1$ vertices are connected to a single central vertex. A star graph with total $n -$ vertex is termed as S_n .

Here is an illustration for the star graph :



Star Graph of order-n (Sn)

Approach : Just traverse whole matrix and record the number of vertices having degree 1 and degree n-1. If number of vertices having degree 1 is n-1 and number of vertex having degree n-1 is 1 then our graph should be a star graph other-wise it should be not.

Note:

- For S1, there must be only one vertex with no edges.
- For S2, there must be two vertices each with degree one or can say, both are connected by a single edge.
- For Sn ($n > 2$) simply check the above explained criteria.

C++

```
// CPP to find whether given graph is star or not
#include<bits/stdc++.h>
using namespace std;

// define the size of incidence matrix
#define size 4

// function to find star graph
```

```
bool checkStar(int mat[][] [size])
{
    // initialize number of vertex
    // with deg 1 and n-1
    int vertexD1 = 0, vertexDn_1 = 0;

    // check for S1
    if (size == 1)
        return (mat[0][0] == 0);

    // check for S2
    if (size == 2)
        return (mat[0][0] == 0 && mat[0][1] == 1 &&
                mat[1][0] == 1 && mat[1][1] == 0);

    // check for Sn (n>2)
    for (int i = 0; i < size; i++)
    {
        int degreeI = 0;
        for (int j = 0; j < size; j++)
            if (mat[i][j])
                degreeI++;

        if (degreeI == 1)
            vertexD1++;
        else if (degreeI == size-1)
            vertexDn_1++;
    }

    return (vertexD1 == (size-1) &&
            vertexDn_1 == 1);
}

// driver code
int main()
{
    int mat[size][size] = { {0, 1, 1, 1},
                           {1, 0, 0, 0},
                           {1, 0, 0, 0},
                           {1, 0, 0, 0}};

    checkStar(mat) ? cout << "Star Graph" :
                    cout << "Not a Star Graph";
    return 0;
}
```

Java

```
// Java program to find whether
// given graph is star or not
import java.io.*;

class GFG
{
    // define the size of
    // incidence matrix
    static int size = 4;

    // function to find
    // star graph
    static boolean checkStar(int mat[] [])
    {
        // initialize number of
        // vertex with deg 1 and n-1
        int vertexD1 = 0,
            vertexDn_1 = 0;

        // check for S1
        if (size == 1)
            return (mat[0][0] == 0);

        // check for S2
        if (size == 2)
            return (mat[0][0] == 0 &&
                    mat[0][1] == 1 &&
                    mat[1][0] == 1 &&
                    mat[1][1] == 0);

        // check for Sn (n>2)
        for (int i = 0; i < size; i++)
        {
            int degreeI = 0;
            for (int j = 0; j < size; j++)
                if (mat[i][j] == 1)
                    degreeI++;

            if (degreeI == 1)
                vertexD1++;
            else if (degreeI == size - 1)
                vertexDn_1++;
        }

        return (vertexD1 == (size - 1) &&
                vertexDn_1 == 1);
    }
}
```

```
// Driver code
public static void main(String args[])
{
    int mat[][] = {{0, 1, 1, 1},
                   {1, 0, 0, 0},
                   {1, 0, 0, 0},
                   {1, 0, 0, 0}};

    if (checkStar(mat))
        System.out.print("Star Graph");
    else
        System.out.print("Not a Star Graph");
}
}

// This code is contributed by
// Manish Shaw(manishshaw1)
```

Python3

```
# Python to find whether
# given graph is star
# or not

# define the size
# of incidence matrix
size = 4

# def to
# find star graph
def checkStar(mat) :

    global size

    # initialize number of
    # vertex with deg 1 and n-1
    vertexD1 = 0
    vertexDn_1 = 0

    # check for S1
    if (size == 1) :
        return (mat[0][0] == 0)

    # check for S2
    if (size == 2) :
        return (mat[0][0] == 0 and
                mat[0][1] == 1 and
                mat[1][0] == 1 and
```

```
mat[1][1] == 0)

# check for Sn (n>2)
for i in range(0, size) :

    degreeI = 0
    for j in range(0, size) :
        if (mat[i][j]) :
            degreeI = degreeI + 1

    if (degreeI == 1) :
        vertexD1 = vertexD1 + 1

    elif (degreeI == size - 1):
        vertexDn_1 = vertexDn_1 + 1

return (vertexD1 == (size - 1) and
        vertexDn_1 == 1)

# Driver code
mat = [[0, 1, 1, 1],
       [1, 0, 0, 0],
       [1, 0, 0, 0],
       [1, 0, 0, 0]]

if(checkStar(mat)) :
    print ("Star Graph")
else :
    print ("Not a Star Graph")

# This code is contributed by
# Manish Shaw(manishshaw1)

C#
// C# to find whether given
// graph is star or not
using System;

class GFG
{
    // define the size of
    // incidence matrix
    static int size = 4;

    // function to find
    // star graph
    static bool checkStar(int [,]mat)
```

```
{  
    // initialize number of  
    // vertex with deg 1 and n-1  
    int vertexD1 = 0, vertexDn_1 = 0;  
  
    // check for S1  
    if (size == 1)  
        return (mat[0, 0] == 0);  
  
    // check for S2  
    if (size == 2)  
        return (mat[0, 0] == 0 &&  
                mat[0, 1] == 1 &&  
                mat[1, 0] == 1 &&  
                mat[1, 1] == 0);  
  
    // check for Sn (n>2)  
    for (int i = 0; i < size; i++)  
    {  
        int degreeI = 0;  
        for (int j = 0; j < size; j++)  
            if (mat[i, j] == 1)  
                degreeI++;  
  
        if (degreeI == 1)  
            vertexD1++;  
        else if (degreeI == size - 1)  
            vertexDn_1++;  
    }  
  
    return (vertexD1 == (size - 1) &&  
            vertexDn_1 == 1);  
}  
  
// Driver code  
static void Main()  
{  
    int [,]mat = new int[4, 4]{ {0, 1, 1, 1},  
                               {1, 0, 0, 0},  
                               {1, 0, 0, 0},  
                               {1, 0, 0, 0} };  
  
    if (checkStar(mat))  
        Console.WriteLine("Star Graph");  
    else  
        Console.WriteLine("Not a Star Graph");  
}  
}
```

```
// This code is contributed by  
// Manish Shaw(manishshaw1)
```

PHP

```
<?php  
// PHP to find whether  
// given graph is star  
// or not  
  
// define the size  
// of incidence matrix  
$size = 4;  
  
// function to  
// find star graph  
function checkStar($mat)  
{  
    global $size;  
  
    // initialize number of  
    // vertex with deg 1 and n-1  
    $vertexD1 = 0;  
    $vertexDn_1 = 0;  
  
    // check for S1  
    if ($size == 1)  
        return ($mat[0][0] == 0);  
  
    // check for S2  
    if ($size == 2)  
        return ($mat[0][0] == 0 &&  
                $mat[0][1] == 1 &&  
                $mat[1][0] == 1 &&  
                $mat[1][1] == 0 );  
  
    // check for Sn (n>2)  
    for ($i = 0; $i < $size; $i++)  
    {  
        $degreeI = 0;  
        for ($j = 0; $j < $size; $j++)  
            if ($mat[$i][$j])  
                $degreeI++;  
  
        if ($degreeI == 1)  
            $vertexD1++;  
        else if ($degreeI == $size - 1)  
            $vertexDn_1++;
```

```
}

return ($vertexD1 == ($size - 1) &&
       $vertexDn_1 == 1);
}

// Driver code
$mat = array(array(0, 1, 1, 1),
             array(1, 0, 0, 0),
             array(1, 0, 0, 0),
             array(1, 0, 0, 0));

if(checkStar($mat))
    echo ("Star Graph");
else
    echo ("Not a Star Graph");

// This code is contributed by
// Manish Shaw(manishshaw1)
?>
```

Output:

Star Graph

Improved By : [manishshaw1](#)

Source

<https://www.geeksforgeeks.org/check-star-graph/>

Chapter 93

Check if a given directed graph is strongly connected Set 2 (Kosaraju using BFS)

Check if a given directed graph is strongly connected Set 2 (Kosaraju using BFS) - Geeks-forGeeks

Given a directed graph, find out whether the graph is strongly connected or not. A directed graph is strongly connected if there is a path between any two pairs of vertices. There are different methods to check the connectivity of directed graph but one of the optimized method is [Kosaraju's DFS based simple algorithm](#).

Kosaraju's BFS based simple algorithm also work on the same principle as DFS based algorithm does.

Following is Kosaraju's BFS based simple algorithm
that does two BFS traversals of graph:

- 1) Initialize all vertices as not visited.
- 2) Do a BFS traversal of graph starting from
any arbitrary vertex v. If BFS traversal
doesn't visit all vertices, then return false.
- 3) Reverse all edges (or find transpose or reverse
of graph)
- 4) Mark all vertices as not visited in reversed graph.
- 5) Again do a BFS traversal of reversed graph starting
from same vertex v (Same as step 2). If BFS traversal
doesn't visit all vertices, then return false.

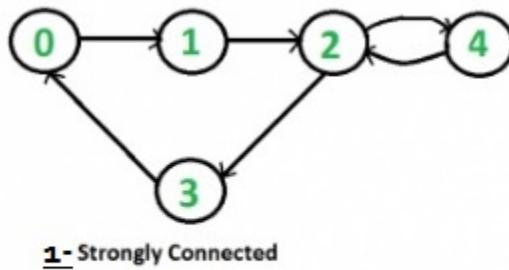
Otherwise, return true.

The idea is again simple if every node can be reached from a vertex v, and every node can reach same vertex v, then the graph is strongly connected. In step 2, we check if all vertices are reachable from v. In step 5, we check if all vertices can reach v (In reversed graph, if all vertices are reachable from v, then all vertices can reach v in original graph).

Explanation with some examples:

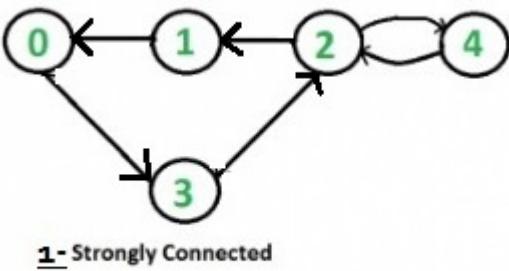
Example 1 :

Given a directed to check if it is strongly connected or not.



step 1: Starting with vertex 2 BFS obtained is 2 3 4 0 1

step 2: After reversing the given graph we got listed graph.



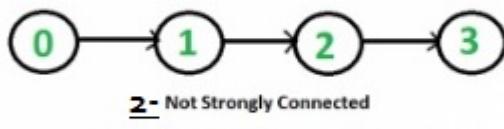
step 3: Again after starting with vertex 2 the BFS is 2 1 4 0 3

step 4: No vertex in both case (step 1 & step 3) remains unvisited.

step 5: So, given graph is strongly connected.

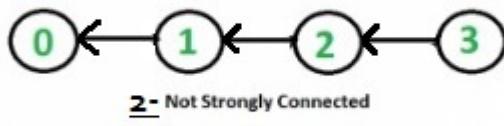
Example 2 :

Given a directed to check if it is strongly connected or not.



step 1: Starting with vertex 2 BFS obtained is 2 3 4

step 2: After reversing the given graph we got listed graph.



step 3: Again after starting with vertex 2 the BFS is 2 1 0

step 4: vertex 0, 1 in original graph and 3, 4 in reverse graph remains unvisited.

step 5: So, given graph is not strongly connected.

Following is the implementation of above algorithm.

```
// C++ program to check if a given directed graph
// is strongly connected or not with BFS use
#include <bits/stdc++.h>
using namespace std;

class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // An array of adjacency lists
```

```
// A recursive function to print DFS starting from v
void BFSUtil(int v, bool visited[]);
public:

// Constructor and Destructor
Graph(int V) { this->V = V; adj = new list<int>[V];}
~Graph() { delete [] adj; }

// Method to add an edge
void addEdge(int v, int w);

// The main function that returns true if the
// graph is strongly connected, otherwise false
bool isSC();

// Function that returns reverse (or transpose)
// of this graph
Graph getTranspose();
};

// A recursive function to print DFS starting from v
void Graph::BFSUtil(int v, bool visited[])
{
    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[v] = true;
    queue.push_back(v);

    // 'i' will be used to get all adjacent vertices
    // of a vertex
    list<int>::iterator i;

    while (!queue.empty())
    {
        // Dequeue a vertex from queue
        v = queue.front();
        queue.pop_front();

        // Get all adjacent vertices of the dequeued vertex s
        // If a adjacent has not been visited, then mark it
        // visited and enqueue it
        for (i = adj[v].begin(); i != adj[v].end(); ++i)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
```

```
        queue.push_back(*i);
    }
}
}

// Function that returns reverse (or transpose) of this graph
Graph Graph::getTranspose()
{
    Graph g(V);
    for (int v = 0; v < V; v++)
    {
        // Recur for all the vertices adjacent to this vertex
        list<int>::iterator i;
        for (i = adj[v].begin(); i != adj[v].end(); ++i)
            g.adj[*i].push_back(v);
    }
    return g;
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

// The main function that returns true if graph
// is strongly connected
bool Graph::isSC()
{
    // Step 1: Mark all the vertices as not
    // visited (For first BFS)
    bool visited[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Step 2: Do BFS traversal starting
    // from first vertex.
    BFSUtil(0, visited);

    // If BFS traversal doesn't visit all
    // vertices, then return false.
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            return false;

    // Step 3: Create a reversed graph
    Graph gr = getTranspose();
```

```
// Step 4: Mark all the vertices as not
// visited (For second BFS)
for(int i = 0; i < V; i++)
    visited[i] = false;

// Step 5: Do BFS for reversed graph
// starting from first vertex.
// Starting Vertex must be same starting
// point of first DFS
gr.BFSUtil(0, visited);

// If all vertices are not visited in
// second DFS, then return false
for (int i = 0; i < V; i++)
    if (visited[i] == false)
        return false;

return true;
}

// Driver program to test above functions
int main()
{
    // Create graphs given in the above diagrams
    Graph g1(5);
    g1.addEdge(0, 1);
    g1.addEdge(1, 2);
    g1.addEdge(2, 3);
    g1.addEdge(3, 0);
    g1.addEdge(2, 4);
    g1.addEdge(4, 2);
    g1.isSC()? cout << "Yes\n" : cout << "No\n";

    Graph g2(4);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.addEdge(2, 3);
    g2.isSC()? cout << "Yes\n" : cout << "No\n";

    return 0;
}
```

Output:

Yes
No

Time Complexity: Time complexity of above implementation is same as Breadth First Search which is $O(V+E)$ if the graph is represented using adjacency matrix representation.

Can we improve further?

The above approach requires two traversals of graph. We can find whether a graph is strongly connected or not in one traversal using [Tarjan's Algorithm to find Strongly Connected Components](#).

Source

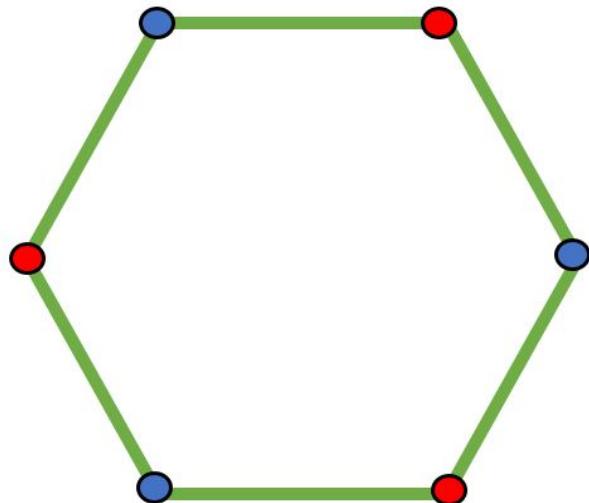
<https://www.geeksforgeeks.org/check-given-directed-graph-strongly-connected-set-2-kosaraju-using-bfs/>

Chapter 94

Check if a given graph is Bipartite using DFS

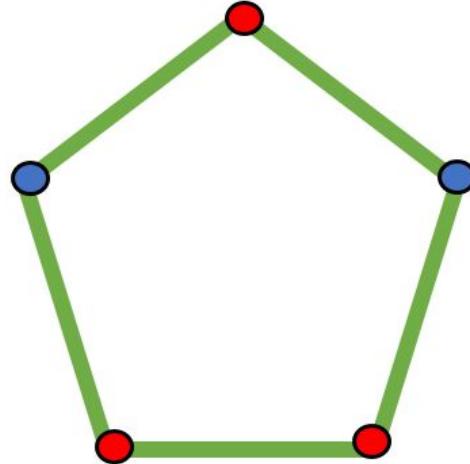
Check if a given graph is Bipartite using DFS - GeeksforGeeks

Given a connected graph, check if the graph is bipartite or not. A bipartite graph is possible if the graph coloring is possible using two colors such that vertices in a set are colored with the same color. Note that it is possible to color a cycle graph with even cycle using two colors. For example, see the following graph.



Cycle graph of length 6

It is not possible to color a cycle graph with an odd cycle using two colors.



Cycle graph of length 5

In the previous [post](#), an approach using [BFS](#) has been discussed. In this post, an approach using [DFS](#) has been implemented. Given below is the algorithm to check for bipartiteness of a graph.

- Use a `color[]` array which stores 0 or 1 for every node which denotes opposite colors.
- Call the function [DFS](#) from any node.
- If the node `u` has not been visited previously, then assign `!color[v]` to `color[u]` and call [DFS](#) again to visit nodes connected to `u`.
- If at any point, `color[u]` is equal to `color[v]`, then the node is bipartite.
- Modify the [DFS](#) function such that it returns a boolean value at the end.

Below is the implementation of the above approach:

```
// C++ program to check if a connected
// graph is bipartite or not suing DFS
#include <bits/stdc++.h>
using namespace std;

// function to store the connected nodes
void addEdge(vector<int> adj[], int u, int v)
{
    adj[u].push_back(v);
}
```

```
adj[v].push_back(u);
}

// function to check whether a graph is bipartite or not
bool isBipartite(vector<int> adj[], int v,
                  vector<bool>& visited, vector<int>& color)
{
    for (int u : adj[v]) {

        // if vertex u is not explored before
        if (visited[u] == false) {

            // mark present vertic as visited
            visited[u] = true;

            // mark its color opposite to its parent
            color[u] = !color[v];

            // if the subtree rooted at vertex v is not bipartite
            if (!isBipartite(adj, u, visited, color))
                return false;
        }
    }

    // if two adjacent are colored with same color then
    // the graph is not bipartite
    else if (color[u] == color[v])
        return false;
}
return true;
}

// Driver Code
int main()
{
    // no of nodes
    int N = 6;

    // to maintain the adjacency list of graph
    vector<int> adj[N + 1];

    // to keep a check on whether
    // a node is discovered or not
    vector<bool> visited(N + 1);

    // to color the vertices
    // of graph with 2 color
    vector<int> color(N + 1);
```

```
// adding edges to the graph
addEdge(adj, 1, 2);
addEdge(adj, 2, 3);
addEdge(adj, 3, 4);
addEdge(adj, 4, 5);
addEdge(adj, 5, 6);
addEdge(adj, 6, 1);

// marking the source node as visited
visited[1] = true;

// marking the source node with a color
color[1] = 0;

// Function to check if the graph
// is Bipartite or not
if (isBipartite(adj, 1, visited, color)) {
    cout << "Graph is Bipartite";
}
else {
    cout << "Graph is not Bipartite";
}

return 0;
}
```

Output:

Graph is Bipartite

Time Complexity: $O(N)$

Auxiliary Space: $O(N)$

Source

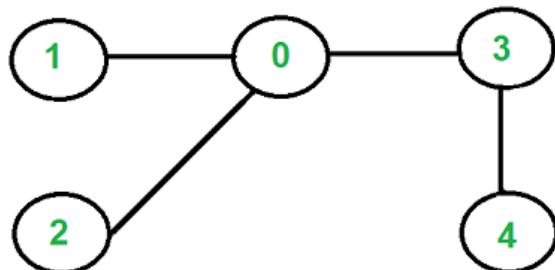
<https://www.geeksforgeeks.org/check-if-a-given-graph-is-bipartite-using-dfs/>

Chapter 95

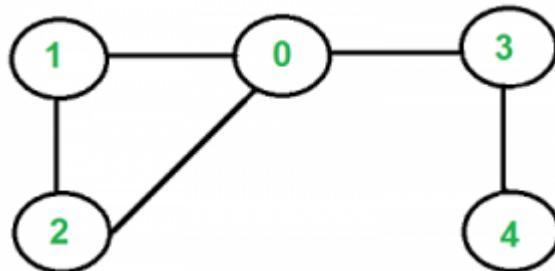
Check if a given graph is tree or not

Check if a given graph is tree or not - GeeksforGeeks

Write a function that returns true if a given undirected graph is tree and false otherwise.
For example, the following graph is a tree.



But the following graph is not a tree.



An undirected graph is tree if it has following properties.

- 1) There is no cycle.
- 2) The graph is connected.

For an undirected graph we can either use [BFS](#) or [DFS](#) to detect above two properties.

How to detect cycle in an undirected graph?

We can either use BFS or DFS. For every visited vertex ‘v’, if there is an adjacent ‘u’ such that u is already visited and u is not parent of v, then there is a cycle in graph. If we don’t find such an adjacent for any vertex, we say that there is no cycle ([See Detect cycle in an undirected graph](#) for more details).

How to check for connectivity?

Since the graph is undirected, we can start BFS or DFS from any vertex and check if all vertices are reachable or not. If all vertices are reachable, then graph is connected, otherwise not.

C++

```
// A C++ Program to check whether a graph is tree or not
#include<iostream>
#include <list>
#include <limits.h>
using namespace std;

// Class for an undirected graph
class Graph
{
    int V;      // No. of vertices
    list<int> *adj; // Pointer to an array for adjacency lists
    bool isCyclicUtil(int v, bool visited[], int parent);
public:
    Graph(int V);    // Constructor
    void addEdge(int v, int w);    // to add an edge to graph
    bool isTree();    // returns true if graph is tree
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
    adj[w].push_back(v); // Add v to w's list.
}

// A recursive function that uses visited[] and parent to
// detect cycle in subgraph reachable from vertex v.
bool Graph::isCyclicUtil(int v, bool visited[], int parent)
{
```

```
// Mark the current node as visited
visited[v] = true;

// Recur for all the vertices adjacent to this vertex
list<int>::iterator i;
for (i = adj[v].begin(); i != adj[v].end(); ++i)
{
    // If an adjacent is not visited, then recur for
    // that adjacent
    if (!visited[*i])
    {
        if (isCyclicUtil(*i, visited, v))
            return true;
    }

    // If an adjacent is visited and not parent of current
    // vertex, then there is a cycle.
    else if (*i != parent)
        return true;
}
return false;
}

// Returns true if the graph is a tree, else false.
bool Graph::isTree()
{
    // Mark all the vertices as not visited and not part of
    // recursion stack
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // The call to isCyclicUtil serves multiple purposes.
    // It returns true if graph reachable from vertex 0
    // is cyclic. It also marks all vertices reachable
    // from 0.
    if (isCyclicUtil(0, visited, -1))
        return false;

    // If we find a vertex which is not reachable from 0
    // (not marked by isCyclicUtil()), then we return false
    for (int u = 0; u < V; u++)
        if (!visited[u])
            return false;

    return true;
}
```

```
// Driver program to test above functions
int main()
{
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.isTree()? cout << "Graph is Tree\n":
                  cout << "Graph is not Tree\n";

    Graph g2(5);
    g2.addEdge(1, 0);
    g2.addEdge(0, 2);
    g2.addEdge(2, 1);
    g2.addEdge(0, 3);
    g2.addEdge(3, 4);
    g2.isTree()? cout << "Graph is Tree\n":
                  cout << "Graph is not Tree\n";

    return 0;
}
```

Java

```
// A Java Program to check whether a graph is tree or not
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency
// list representation
class Graph
{
    private int V;    // No. of vertices
    private LinkedList<Integer> adj[]; //Adjacency List

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    // Function to add an edge into the graph
    void addEdge(int v,int w)
    {
```

```
adj[v].add(w);
adj[w].add(v);
}

// A recursive function that uses visited[] and parent
// to detect cycle in subgraph reachable from vertex v.
Boolean isCyclicUtil(int v, Boolean visited[], int parent)
{
    // Mark the current node as visited
    visited[v] = true;
    Integer i;

    // Recur for all the vertices adjacent to this vertex
    Iterator<Integer> it = adj[v].iterator();
    while (it.hasNext())
    {
        i = it.next();

        // If an adjacent is not visited, then recur for
        // that adjacent
        if (!visited[i])
        {
            if (isCyclicUtil(i, visited, v))
                return true;
        }

        // If an adjacent is visited and not parent of
        // current vertex, then there is a cycle.
        else if (i != parent)
            return true;
    }
    return false;
}

// Returns true if the graph is a tree, else false.
Boolean isTree()
{
    // Mark all the vertices as not visited and not part
    // of recursion stack
    Boolean visited[] = new Boolean[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // The call to isCyclicUtil serves multiple purposes
    // It returns true if graph reachable from vertex 0
    // is cyclcic. It also marks all vertices reachable
    // from 0.
    if (isCyclicUtil(0, visited, -1))
```

```
        return false;

    // If we find a vertex which is not reachable from 0
    // (not marked by isCyclicUtil(), then we return false
    for (int u = 0; u < V; u++)
        if (!visited[u])
            return false;

    return true;
}

// Driver method
public static void main(String args[])
{
    // Create a graph given in the above diagram
    Graph g1 = new Graph(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    if (g1.isTree())
        System.out.println("Graph is Tree");
    else
        System.out.println("Graph is not Tree");

    Graph g2 = new Graph(5);
    g2.addEdge(1, 0);
    g2.addEdge(0, 2);
    g2.addEdge(2, 1);
    g2.addEdge(0, 3);
    g2.addEdge(3, 4);

    if (g2.isTree())
        System.out.println("Graph is Tree");
    else
        System.out.println("Graph is not Tree");

}
}

// This code is contributed by Aakash Hasija
```

Python

```
# Python Program to check whether
# a graph is tree or not

from collections import defaultdict
```

```
class Graph():

    def __init__(self, V):
        self.V = V
        self.graph = defaultdict(list)

    def addEdge(self, v, w):
        # Add w to v list.
        self.graph[v].append(w)
        # Add v to w list.
        self.graph[w].append(v)

    # A recursive function that uses visited[]
    # and parent to detect cycle in subgraph
    # reachable from vertex v.
    def isCyclicUtil(self, v, visited, parent):

        # Mark current node as visited
        visited[v] = True

        # Recur for all the vertices adjacent
        # for this vertex
        for i in self.graph[v]:
            # If an adjacent is not visited,
            # then recur for that adjacent
            if visited[i] == False:
                if self.isCyclicUtil(i, visited, v) == True:
                    return True

            # If an adjacent is visited and not
            # parent of current vertex, then there
            # is a cycle.
            elif i != parent:
                return True

        return False

    # Returns true if the graph is a tree,
    # else false.
    def isTree(self):
        # Mark all the vertices as not visited
        # and not part of recursion stack
        visited = [False] * self.V

        # The call to isCyclicUtil serves multiple
        # purposes. It returns true if graph reachable
        # from vertex 0 is cyclic. It also marks
        # all vertices reachable from 0.
```

```
if self.isCyclicUtil(0, visited, -1) == True:
    return False

# If we find a vertex which is not reachable
# from 0 (not marked by isCyclicUtil()),
# then we return false
for i in range(self.V):
    if visited[i] == False:
        return False

return True

# Driver program to test above functions
g1 = Graph(5)
g1.addEdge(1, 0)
g1.addEdge(0, 2)
g1.addEdge(0, 3)
g1.addEdge(3, 4)
print "Graph is a Tree" if g1.isTree() == True \
else "Graph is a not a Tree"

g2 = Graph(5)
g2.addEdge(1, 0)
g2.addEdge(0, 2)
g2.addEdge(2, 1)
g2.addEdge(0, 3)
g2.addEdge(3, 4)
print "Graph is a Tree" if g2.isTree() == True \
else "Graph is a not a Tree"

# This code is contributed by Divyanshu Mehta
```

Output:

```
Graph is Tree
Graph is not Tree
```

Thanks to **Vinit Verma** for suggesting this problem and initial solution. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

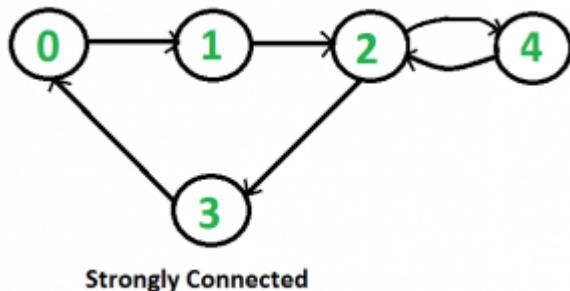
<https://www.geeksforgeeks.org/check-given-graph-tree/>

Chapter 96

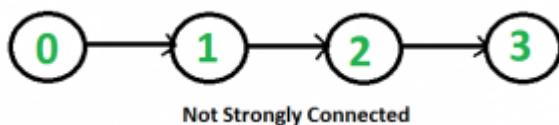
Check if a graph is strongly connected Set 1 (Kosaraju using DFS)

Check if a graph is strongly connected Set 1 (Kosaraju using DFS) - GeeksforGeeks

Given a directed graph, find out whether the graph is strongly connected or not. A directed graph is strongly connected if there is a path between any two pair of vertices. For example, following is a strongly connected graph.



It is easy for undirected graph, we can just do a BFS and DFS starting from any vertex. If BFS or DFS visits all vertices, then the given undirected graph is connected. This approach won't work for a directed graph. For example, consider the following graph which is not strongly connected. If we start DFS (or BFS) from vertex 0, we can reach all vertices, but if we start from any other vertex, we cannot reach all vertices.



How to do for directed graph?

A simple idea is to use a all pair shortest path algorithm like **Floyd Warshall** or **find Transitive Closure** of graph. Time complexity of this method would be $O(v^3)$.

We can also **do DFSV times** starting from every vertex. If any DFS, doesn't visit all vertices, then graph is not strongly connected. This algorithm takes $O(V^*(V+E))$ time which can be same as transitive closure for a dense graph.

A better idea can be **Strongly Connected Components (SCC) algorithm**. We can find all SCCs in $O(V+E)$ time. If number of SCCs is one, then graph is strongly connected. The algorithm for SCC does extra work as it finds all SCCs.

Following is **Kosaraju's DFS based simple algorithm that does two DFS traversals of graph:**

- 1) Initialize all vertices as not visited.
- 2) Do a DFS traversal of graph starting from any arbitrary vertex v. If DFS traversal doesn't visit all vertices, then return false.
- 3) Reverse all arcs (or find transpose or reverse of graph)
- 4) Mark all vertices as not-visited in reversed graph.
- 5) Do a DFS traversal of reversed graph starting from same vertex v (Same as step 2). If DFS traversal doesn't visit all vertices, then return false. Otherwise return true.

The idea is, if every node can be reached from a vertex v, and every node can reach v, then the graph is strongly connected. In step 2, we check if all vertices are reachable from v. In step 4, we check if all vertices can reach v (In reversed graph, if all vertices are reachable from v, then all vertices can reach v in original graph).

Following is the implementation of above algorithm.

C++

```
// C++ program to check if a given directed graph is strongly
// connected or not
#include <iostream>
#include <list>
#include <stack>
using namespace std;

class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // An array of adjacency lists

    // A recursive function to print DFS starting from v
    void DFSUtil(int v, bool visited[]);

public:
    // Constructor and Destructor
    Graph(int V) { this->V = V; adj = new list<int>[V]; }
```

```
~Graph() { delete [] adj; }

// Method to add an edge
void addEdge(int v, int w);

// The main function that returns true if the graph is strongly
// connected, otherwise false
bool isSC();

// Function that returns reverse (or transpose) of this graph
Graph getTranspose();
};

// A recursive function to print DFS starting from v
void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

// Function that returns reverse (or transpose) of this graph
Graph Graph::getTranspose()
{
    Graph g(V);
    for (int v = 0; v < V; v++)
    {
        // Recur for all the vertices adjacent to this vertex
        list<int>::iterator i;
        for(i = adj[v].begin(); i != adj[v].end(); ++i)
        {
            g.adj[*i].push_back(v);
        }
    }
    return g;
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

// The main function that returns true if graph is strongly connected
```

```
bool Graph::isSC()
{
    // Step 1: Mark all the vertices as not visited (For first DFS)
    bool visited[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Step 2: Do DFS traversal starting from first vertex.
    DFSUtil(0, visited);

    // If DFS traversal doesn't visit all vertices, then return false.
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            return false;

    // Step 3: Create a reversed graph
    Graph gr = getTranspose();

    // Step 4: Mark all the vertices as not visited (For second DFS)
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Step 5: Do DFS for reversed graph starting from first vertex.
    // Starting Vertex must be same starting point of first DFS
    gr.DFSUtil(0, visited);

    // If all vertices are not visited in second DFS, then
    // return false
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            return false;

    return true;
}

// Driver program to test above functions
int main()
{
    // Create graphs given in the above diagrams
    Graph g1(5);
    g1.addEdge(0, 1);
    g1.addEdge(1, 2);
    g1.addEdge(2, 3);
    g1.addEdge(3, 0);
    g1.addEdge(2, 4);
    g1.addEdge(4, 2);
    g1.isSC()? cout << "Yes\n" : cout << "No\n";
}
```

```
Graph g2(4);
g2.addEdge(0, 1);
g2.addEdge(1, 2);
g2.addEdge(2, 3);
g2.isSC()? cout << "Yes\n" : cout << "No\n";

return 0;
}
```

Java

```
// Java program to check if a given directed graph is strongly
// connected or not
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents a directed graph using adjacency
// list representation
class Graph
{
    private int V;    // No. of vertices
    private LinkedList<Integer> adj[]; //Adjacency List

    //Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    //Function to add an edge into the graph
    void addEdge(int v,int w) { adj[v].add(w); }

    // A recursive function to print DFS starting from v
    void DFSUtil(int v,Boolean visited[])
    {
        // Mark the current node as visited and print it
        visited[v] = true;

        int n;

        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> i = adj[v].iterator();
        while (i.hasNext())
        {
```

```
n = i.next();
if (!visited[n])
    DFSUtil(n,visited);
}

// Function that returns transpose of this graph
Graph getTranspose()
{
    Graph g = new Graph(V);
    for (int v = 0; v < V; v++)
    {
        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> i = adj[v].listIterator();
        while (i.hasNext())
            g.adj[i.next()].add(v);
    }
    return g;
}

// The main function that returns true if graph is strongly
// connected
Boolean isSC()
{
    // Step 1: Mark all the vertices as not visited
    // (For first DFS)
    Boolean visited[] = new Boolean[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Step 2: Do DFS traversal starting from first vertex.
    DFSUtil(0, visited);

    // If DFS traversal doesn't visit all vertices, then
    // return false.
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            return false;

    // Step 3: Create a reversed graph
    Graph gr = getTranspose();

    // Step 4: Mark all the vertices as not visited (For
    // second DFS)
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Step 5: Do DFS for reversed graph starting from
```

```
// first vertex. Starting Vertex must be same starting
// point of first DFS
gr.DFSUtil(0, visited);

// If all vertices are not visited in second DFS, then
// return false
for (int i = 0; i < V; i++)
    if (visited[i] == false)
        return false;

return true;
}

public static void main(String args[])
{
    // Create graphs given in the above diagrams
    Graph g1 = new Graph(5);
    g1.addEdge(0, 1);
    g1.addEdge(1, 2);
    g1.addEdge(2, 3);
    g1.addEdge(3, 0);
    g1.addEdge(2, 4);
    g1.addEdge(4, 2);
    if (g1.isSC())
        System.out.println("Yes");
    else
        System.out.println("No");

    Graph g2 = new Graph(4);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.addEdge(2, 3);
    if (g2.isSC())
        System.out.println("Yes");
    else
        System.out.println("No");
}
}

// This code is contributed by Aakash Hasija
```

Python

```
# Python program to check if a given directed graph is strongly
# connected or not

from collections import defaultdict

#This class represents a directed graph using adjacency list representation
```

```
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices
        self.graph = defaultdict(list) # default dictionary to store graph

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    #A function used by isSC() to perform DFS
    def DFSUtil(self,v,visited):

        # Mark the current node as visited
        visited[v]= True

        #Recur for all the vertices adjacent to this vertex
        for i in self.graph[v]:
            if visited[i]==False:
                self.DFSUtil(i,visited)

    # Function that returns reverse (or transpose) of this graph
    def getTranspose(self):

        g = Graph(self.V)

        # Recur for all the vertices adjacent to this vertex
        for i in self.graph:
            for j in self.graph[i]:
                g.addEdge(j,i)

        return g

    # The main function that returns true if graph is strongly connected
    def isSC(self):

        # Step 1: Mark all the vertices as not visited (For first DFS)
        visited =[False]*(self.V)

        # Step 2: Do DFS traversal starting from first vertex.
        self.DFSUtil(0,visited)

        # If DFS traversal doesnt visit all vertices, then return false
        if any(i == False for i in visited):
            return False
```

```
# Step 3: Create a reversed graph
gr = self.getTranspose()

# Step 4: Mark all the vertices as not visited (For second DFS)
visited =[False]*(self.V)

# Step 5: Do DFS for reversed graph starting from first vertex.
# Starting Vertex must be same starting point of first DFS
gr.DFSUtil(0,visited)

# If all vertices are not visited in second DFS, then
# return false
if any(i == False for i in visited):
    return False

return True

# Create a graph given in the above diagram
g1 = Graph(5)
g1.addEdge(0, 1)
g1.addEdge(1, 2)
g1.addEdge(2, 3)
g1.addEdge(3, 0)
g1.addEdge(2, 4)
g1.addEdge(4, 2)
print "Yes" if g1.isSC() else "No"

g2 = Graph(4)
g2.addEdge(0, 1)
g2.addEdge(1, 2)
g2.addEdge(2, 3)
print "Yes" if g2.isSC() else "No"

#This code is contributed by Neelam Yadav
```

Output:

```
Yes
No
```

Time Complexity: Time complexity of above implementation is same as [Depth First Search](#) which is $O(V+E)$ if the graph is represented using adjacency list representation.

Can we improve further?

The above approach requires two traversals of graph. We can find whether a graph is strongly

connected or not in one traversal using [Tarjan's Algorithm to find Strongly Connected Components](#).

Exercise:

Can we use BFS instead of DFS in above algorithm? See [this](#).

References:

<http://www.ieor.berkeley.edu/~hochbaum/files/ieor266-2012.pdf>

Source

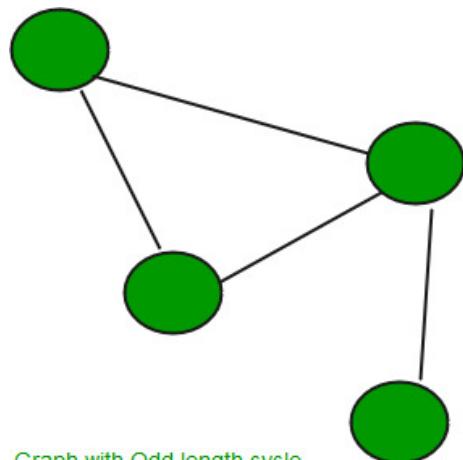
<https://www.geeksforgeeks.org/connectivity-in-a-directed-graph/>

Chapter 97

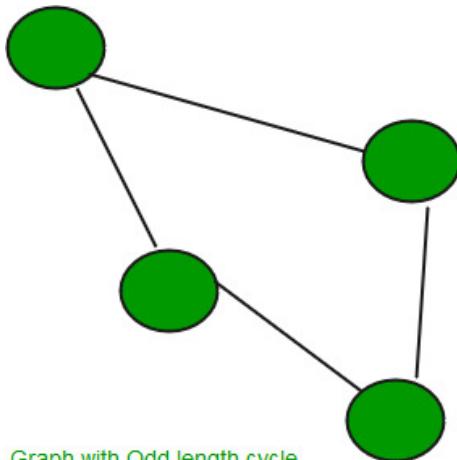
Check if a graphs has a cycle of odd length

Check if a graphs has a cycle of odd length - GeeksforGeeks

Given a graph, the task is to find if it has a cycle of odd length or not.



Graph with Odd length cycle



Graph with Odd length cycle

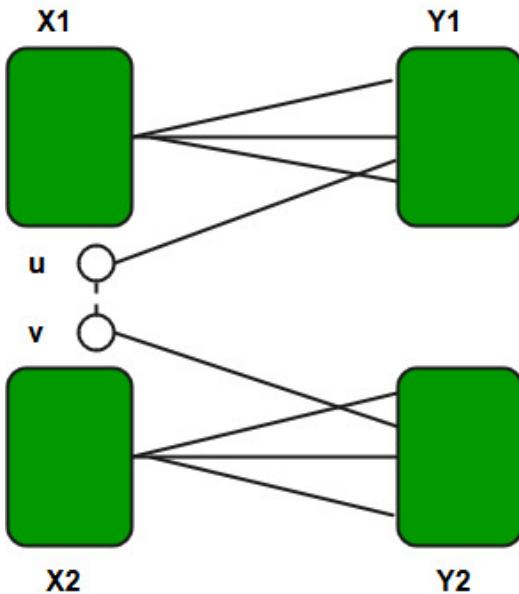
The idea is based on an important fact that a graph does **not** contain a cycle of odd length if and only if it is **Bipartite**, i.e., it can be colored with two colors.

It is obvious that if a graph has odd length cycle then it cannot be Bipartite. In Bipartite graph there are two sets of vertices such that no vertex in a set is connected with any other vertex of same set). For a cycle of odd length, two vertices must of same set must be connected which contradicts Bipartite definition.

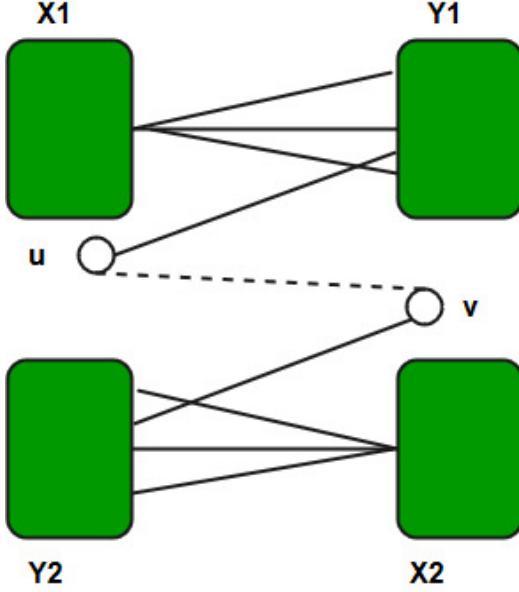
Let us understand converse, if a graph has no odd cycle then it must be Bipartite. Below is a induction based proof of this taken from <http://infohost.nmt.edu/~math/faculty/barefoot/Math321Spring98/BipartiteGraphsAndEvenCycles.html>

Assume that (X, Y) is a bipartition of G and let $C = u_1, u_2, \dots, u_k$ be a cycle of G , where u_1 is in the vertex set X (abbreviated $u_1 \in X$). If $u_1 \in X$ then $u_2 \in Y, \dots$ and, in general, $u_{2j+1} \in X$ and $u_{2i} \in Y$. Since C is a cycle, $u_k \in Y$, so that $k = 2s$ for some positive integer s . Therefore cycle C is even.

Assume that graph G has no odd cycles. It will be shown that such a graph is bipartite. The proof is induction on the number of edges. The assertion is clearly true for a graph with at most one edge. Assume that every graph with no odd cycles and at most q edges is bipartite and let G be a graph with $q + 1$ edges and with no odd cycles. Let $e = uv$ be an edge of G and consider the graph $H = G - uv$. By induction, H has a bipartition (X, Y) . If e has one end in X and the other end in Y then (X, Y) is a bipartition of G . Hence, assume that u and v are in X . If there were a path, P , between u and v in H then the length of P would be even. Thus, $P + uv$ would be an odd cycle of G . Therefore, u and v must be in lie in differenet “pieces” or components of H . Thus, we have:



where $X = X_1 \cup X_2$ and $Y = Y_1 \cup Y_2$. In this case it is clear that $(X_1 \cup Y_2, X_2 \cup Y_1)$ is a bipartition of G .



Therefore we conclude that every graph with no odd cycles is bipartite. One can construct a bipartition as follows:

- (1) Choose an arbitrary vertex x_0 and set $X_0 = \{x_0\}$.
- (2) Let Y_0 be the set of all vertices adjacent to x_0 and iterate steps 3-4.
- (3) Let X_k be the set of vertices not chosen that are adjacent to a vertex of Y_{k-1} .
- (4) Let Y_k be the set of vertices not chosen that are adjacent to a vertex of X_{k-1} .
- (5) If all vertices of G have been chosen then

$X = X_0 \cup X_1 \cup X_2 \cup \dots$ and $Y = Y_0 \cup Y_1 \cup Y_2 \cup \dots$

Below is code to check if a graph has odd cycle or not. The code basically checks whether graph is Bipartite.

C++

```
// C++ program to find out whether a given graph is
// Bipartite or not
#include <iostream>
#include <queue>
#define V 4
using namespace std;

// This function returns true if graph G[V][V] contains
// odd cycle, else false
bool containsOdd(int G[][] , int src)
{
    // Create a color array to store colors assigned
    // to all vertices. Vertex number is used as index
    // in this array. The value '-1' of colorArr[i]
    // is used to indicate that no color is assigned to
    // vertex 'i'. The value 1 is used to indicate first
    // color is assigned and value 0 indicates second
    // color is assigned.
    int colorArr[V];
    for (int i = 0; i < V; ++i)
        colorArr[i] = -1;

    // Assign first color to source
    colorArr[src] = 1;

    // Create a queue (FIFO) of vertex numbers and
    // enqueue source vertex for BFS traversal
    queue <int> q;
    q.push(src);

    // Run while there are vertices in queue (Similar to BFS)
    while (!q.empty())
    {
        // Dequeue a vertex from queue
        int u = q.front();
        q.pop();

        // Return true if there is a self-loop
        if (G[u][u] == 1)
            return true;

        // Find all non-colored adjacent vertices
        for (int v = 0; v < V; ++v)
        {
```

```
// An edge from u to v exists and destination
// v is not colored
if (G[u][v] && colorArr[v] == -1)
{
    // Assign alternate color to this adjacent
    // v of u
    colorArr[v] = 1 - colorArr[u];
    q.push(v);
}

// An edge from u to v exists and destination
// v is colored with same color as u
else if (G[u][v] && colorArr[v] == colorArr[u])
    return true;
}
}

// If we reach here, then all adjacent
// vertices can be colored with alternate
// color
return false;
}

// Driver program to test above function
int main()
{
    int G[][][V] = {{0, 1, 0, 1},
                    {1, 0, 1, 0},
                    {0, 1, 0, 1},
                    {1, 0, 1, 0}};
    containsOdd(G, 0) ? cout << "Yes" : cout << "No";
    return 0;
}
```

Java

```
// JAVA Code For Check if a graphs has a cycle
// of odd length
import java.util.*;

class GFG {

    public static int V = 4;

    // This function returns true if graph G[V][V]
    // contains odd cycle, else false
```

```

public static boolean containsOdd(int G[][] , int src)
{
    // Create a color array to store colors assigned
    // to all vertices. Vertex number is used as
    // index in this array. The value '-1' of
    // colorArr[i] is used to indicate that no color
    // is assigned to vertex 'i'. The value 1 is
    // used to indicate first color is assigned and
    // value 0 indicates second color is assigned.
    int colorArr[] = new int[V];
    for (int i = 0; i < V; ++i)
        colorArr[i] = -1;

    // Assign first color to source
    colorArr[src] = 1;

    // Create a queue (FIFO) of vertex numbers and
    // enqueue source vertex for BFS traversal
    LinkedList<Integer> q = new LinkedList<Integer>();
    q.add(src);

    // Run while there are vertices in queue
    // (Similar to BFS)
    while (!q.isEmpty())
    {
        // Dequeue a vertex from queue
        int u = q.peek();
        q.pop();

        // Return true if there is a self-loop
        if (G[u][u] == 1)
            return true;

        // Find all non-colored adjacent vertices
        for (int v = 0; v < V; ++v)
        {
            // An edge from u to v exists and
            // destination v is not colored
            if (G[u][v] == 1 && colorArr[v] == -1)
            {
                // Assign alternate color to this
                // adjacent v of u
                colorArr[v] = 1 - colorArr[u];
                q.push(v);
            }
        }

        // An edge from u to v exists and
        // destination v is colored with same
    }
}

```

```
// color as u
else if (G[u][v] == 1 && colorArr[v] ==
          colorArr[u])
    return true;
}
}

// If we reach here, then all adjacent
// vertices can be colored with alternate
// color
return false;
}

/* Driver program to test above function */
public static void main(String[] args)
{
    int G[][] = {{0, 1, 0, 1},
                 {1, 0, 1, 0},
                 {0, 1, 0, 1},
                 {1, 0, 1, 0}};

    if (containsOdd(G, 0))
        System.out.println("Yes") ;
    else
        System.out.println("No");
}
}

// This code is contributed by Arnav Kr. Mandal.
```

Output :

No

The above algorithm works only if the graph is strongly connected. We can extend it for the cases when graph is not strongly connected (Please refer [this](#) for details). In above code, we always start with source 0 and assume that vertices are visited from it. One important observation is a graph with no edges is also Bipartite. Note that the Bipartite condition says all edges should be from one set to another.

Source

<https://www.geeksforgeeks.org/check-graphs-cycle-odd-length/>

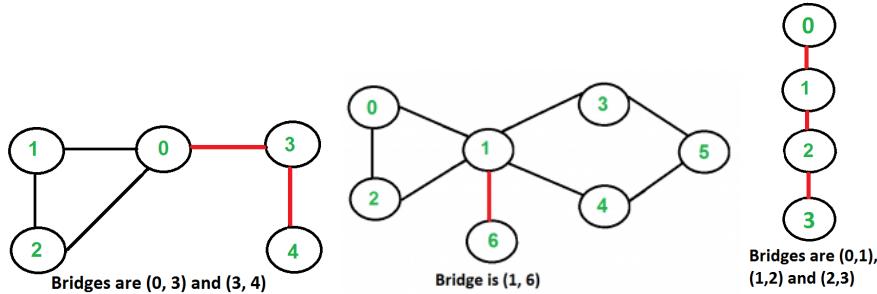
Chapter 98

Check if removing a given edge disconnects a graph

Check if removing a given edge disconnects a graph - GeeksforGeeks

Given an undirected graph and an edge, the task is to find if the given edge is a [bridge](#) in graph, i.e., removing the edge disconnects the graph.

Following are some example graphs with bridges highlighted with red color.



One solution is to [find all bridges in given graph](#) and then check if given edge is a bridge or not.

A simpler solution is to remove the edge, check if graph remains connected after removal or not, finally add the edge back. We can always find if an undirected graph is connected or not by finding all reachable vertices from any vertex. If count of reachable vertices is equal to number of vertices in graph, then the graph is connected else not. We can find all reachable vertices either using BFS or DFS. Below are complete steps.

- 1) Remove the given edge
- 2) Find all reachable vertices from any vertex. We have chosen first vertex in below implementation.
- 3) If count of reachable nodes is V, then return false [given is not Bridge]. Else return yes.

```
// C++ program to check if removing an
```

```
// edge disconnects a graph or not.
#include<bits/stdc++.h>
using namespace std;

// Graph class represents a directed graph
// using adjacency list representation
class Graph
{
    int V;      // No. of vertices
    list<int> *adj;
    void DFS(int v, bool visited[]);
public:
    Graph(int V);    // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // Returns true if graph is connected
    bool isConnected();

    bool isBridge(int u, int v);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int u, int v)
{
    adj[u].push_back(v); // Add w to v's list.
    adj[v].push_back(u); // Add w to v's list.
}

void Graph::DFS(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;

    // Recur for all the vertices adjacent to
    // this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFS(*i, visited);
}
```

```
// Returns true if given graph is connected, else false
bool Graph::isConnected()
{
    bool visited[V];
    memset(visited, false, sizeof(visited));

    // Find all reachable vertices from first vertex
    DFS(0, visited);

    // If set of reachable vertices includes all,
    // return true.
    for (int i=1; i<V; i++)
        if (visited[i] == false)
            return false;

    return true;
}

// This function assumes that edge (u, v)
// exists in graph or not,
bool Graph::isBridge(int u, int v)
{
    // Remove edge from undirected graph
    adj[u].remove(v);
    adj[v].remove(u);

    bool res = isConnected();

    // Adding the edge back
    addEdge(u, v);

    // Return true if graph becomes disconnected
    // after removing the edge.
    return (res == false);
}

// Driver code
int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(1, 2);
    g.addEdge(2, 3);

    g.isBridge(1, 2)? cout << "Yes" : cout << "No";

    return 0;
}
```

}

Output :

Yes

Time Complexity : $O(V + E)$

Source

<https://www.geeksforgeeks.org/check-removing-given-edge-disconnects-given-graph/>

Chapter 99

Check if the given permutation is a valid DFS of graph

Check if the given permutation is a valid DFS of graph - GeeksforGeeks

Given a graph with N nodes numbered from 1 to N and M edges and an array of numbers from 1 to N. Check if it is possible to obtain any permutation of array by applying DFS (Depth First Traversal) on given graph.

Prerequisites : [DFS Map in CPP](#)

Examples :

Input : N = 3, M = 2

Edges are:

- 1) 1-2
- 2) 2-3

P = {1, 2, 3}

Output : YES

Explanation :

Since there are edges between 1-2 and 2-3, therefore we can have DFS in the order 1-2-3

Input : N = 3, M = 2

Edges are:

- 1) 1-2
- 2) 2-3

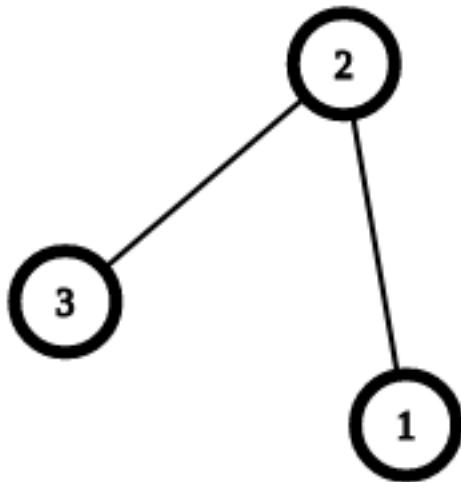
P = {1, 3, 2}

Output : NO

Explanation :

Since there is no edge between 1 and 3, the DFS traversal is not possible

in the order of given permutation.



Possible Permutations in which whole graph can be traversed are:

- 1) 1 2 3
- 2) 3 2 1

Approach : We assume that the input graph is represented as adjacency list. The idea is to first sort all adjacency lists according to input order, then traverse the given graph starting from first node in given given permutation. If we visit all vertices in same order, then given permutation is a valid DFS.

1. Store the indexes of each number in the given permutation in a Hash map.
2. Sort every adjacency list according to the indexes of permutation since there is need to maintain the order.
3. Perform the Depth First Traversal Search with source node as 1st number of given permutation.
4. Keep a counter variable and at every recursive call, check if the counter has reached the number of nodes, i.e. N and set the flag as 1. If the flag is 0 after complete DFS, answer is ‘NO’ otherwise ‘YES’

Below is the implementation of above approach :

```
// CPP program to check if given  
// permutation can be obtained  
// upon DFS traversal on given graph  
#include <bits/stdc++.h>  
using namespace std;  
  
// To track of DFS is valid or not.
```

```
bool flag = false;

// HashMap to store the indexes
// of given permutation
map<int, int> mp;

// Comparator function for sort
bool cmp(int a, int b)
{
    // Sort according ascending
    // order of indexes
    return mp[a] < mp[b];
}

// Graph class represents a undirected
// using adjacency list representation
class Graph
{
    int V; // No. of vertices
    int counter; // Counter variable

public:
    // Pointer to an array containing
    // adjacency lists
    list<int>*> adj;

    Graph(int V); // Constructor

    // function to add an edge to graph
    void addEdge(int u, int v);

    // DFS traversal of the vertices
    // reachable from v
    void DFS(int v, int Perm[]);
};

Graph::Graph(int V)
{
    this->V = V;
    this->counter = 0;
    adj = new list<int>[V + 1];
}

void Graph::addEdge(int u, int v)
{
    adj[u].push_back(v); // Add v to u's list.
    adj[v].push_back(u); // Add u to v's list
}
```

```
// DFS traversal of the
// vertices reachable from v.
void Graph::DFS(int v, int Perm[])
{
    // Increment counter for
    // every node being traversed
    counter++;

    // Check if counter has
    // reached number of vertices
    if (counter == V) {

        // Set flag to 1
        flag = 1;
        return;
    }

    // Recur for all vertices adjacent
    // to this vertex only if it
    // lies in the given permutation
    list<int>::iterator i;
    for (i = adj[v].begin();
         i != adj[v].end(); i++)
    {

        // if the current node equals to
        // current element of permutation
        if (*i == Perm[counter])
            DFS(*i, Perm);
    }
}

// Returns true if P[] is a valid DFS of given
// graph. In other words P[] can be obtained by
// doing a DFS of the graph.
bool checkPermutation(int N, int M,
                      vector<pair<int, int> > V, int P[])
{
    // Create the required graph with
    // N vertices and M edges
    Graph G(N);

    // Add Edges to Graph G
    for (int i = 0; i < M; i++)
        G.addEdge(V[i].first, V[i].second);

    for (int i = 0; i < N; i++)
```

```
mp[P[i]] = i;

// Sort every adjacency
// list according to HashMap
for (int i = 1; i <= N; i++)
    G.adj[i].sort(cmp);

// Call DFS with source node as P[0]
G.DFS(P[0], P);

// If Flag has been set to 1, means
// given permutation is obtained
// by DFS on given graph
return flag;
}

// Driver code
int main()
{
    // Number of vertices and number of edges
    int N = 3, M = 2;

    // Vector of pair to store edges
    vector<pair<int, int> > V;

    V.push_back(make_pair(1, 2));
    V.push_back(make_pair(2, 3));

    int P[] = { 1, 2, 3 };

    // Return the answer
    if (checkPermutation(N, M, V, P))
        cout << "YES" << endl;
    else
        cout << "NO" << endl;

    return 0;
}
```

Output:

YES

Source

<https://www.geeksforgeeks.org/check-given-permutation-valid-dfs-graph/>

Chapter 100

Check if there is a cycle with odd weight sum in an undirected graph

Check if there is a cycle with odd weight sum in an undirected graph - GeeksforGeeks

Given a weighted and undirected graph, we need to find if a cycle exist in this graph such that the sum of weights of all the edges in that cycle comes out to be odd.

Examples:

```
Input : Number of vertices, n = 4,  
        Number of edges, m = 4  
        Weighted Edges =  
        1 2 12  
        2 3 1  
        4 3 1  
        4 1 20  
Output : No! There is no odd weight  
          cycle in the given graph
```

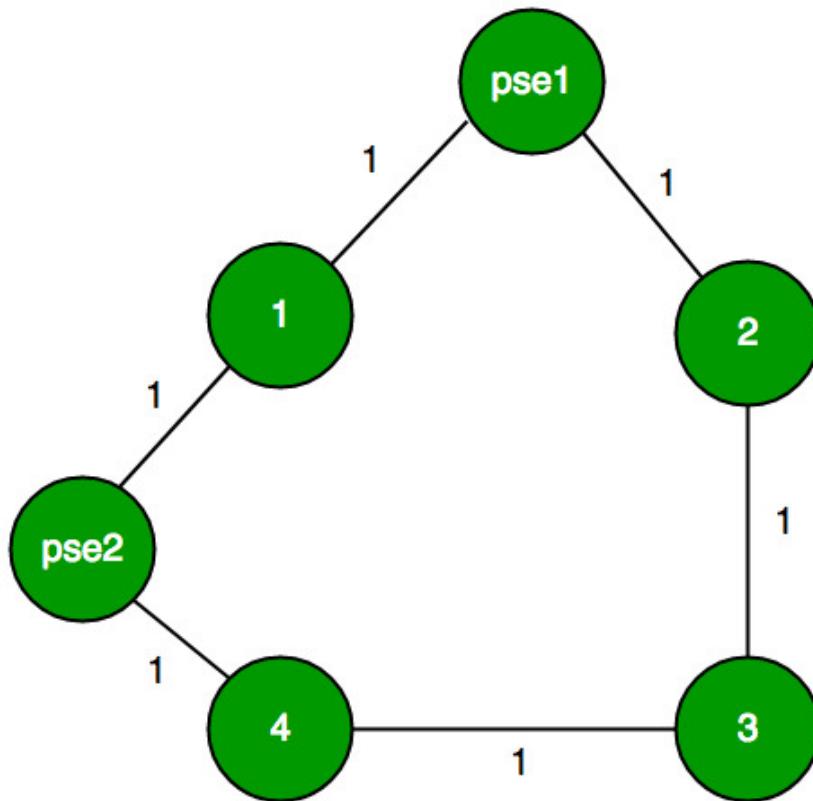
```
Input : Number of vertices, n = 5,  
        Number of edges, m = 3  
        Weighted Edges =  
        1 2 1  
        3 2 1  
        3 1 1  
Output : Yes! There is an odd weight  
          cycle in the given graph
```

The solution is based on the fact that “[If a graph has no odd length cycle then it must be Bipartite, i.e., it can be colored with two colors](#)”

The idea is to convert given problem to a simpler problem where we have to just check if there is cycle of odd length or not. To convert, we do following

1. Convert all even weight edges into two edges of unit weight.
2. Convert all odd weight edges to a single edge of unit weight.

Let's make an another graph for graph shown above (in example 1)



Here, edges [1 — 2] have been broken in two parts such that [1-pseudo1-2] a pseudo node has been introduced. We are doing this so that each of our even weighted edge is taken into consideration twice while the edge with odd weight is counted only once. Doing this would help us further when we color our cycle. We assign all the edges with weight 1 and then by using 2 color method traverse the whole graph. Now we start coloring our modified graph using two colors only. In a cycle with even number of nodes, when we color it using two colors only, none of the two adjacent edges have the same color. While if we try coloring a cycle having odd number of edges, surely a situation arises where two adjacent edges have the same color. This is our pick! Thus, if we are able to color the modified graph completely

using 2 colors only in a way no two adjacent edges get the same color assigned to them then there must be either no cycle in the graph or a cycle with even number of nodes. If any conflict arises while coloring a cycle with 2 colors only, then we have an odd cycle in our graph.

```
// C++ program to check if there is a cycle of
// total odd weight
#include <bits/stdc++.h>
using namespace std;

// This function returns true if the current subpart
// of the forest is two colorable, else false.
bool twoColorUtil(vector<int>G[], int src, int N,
                  int colorArr[]) {

    // Assign first color to source
    colorArr[src] = 1;

    // Create a queue (FIFO) of vertex numbers and
    // enqueue source vertex for BFS traversal
    queue <int> q;
    q.push(src);

    // Run while there are vertices in queue
    // (Similar to BFS)
    while (!q.empty()){

        int u = q.front();
        q.pop();

        // Find all non-colored adjacent vertices
        for (int v = 0; v < G[u].size(); ++v){

            // An edge from u to v exists and
            // destination v is not colored
            if (colorArr[G[u][v]] == -1){

                // Assign alternate color to this
                // adjacent v of u
                colorArr[G[u][v]] = 1 - colorArr[u];
                q.push(G[u][v]);
            }

            // An edge from u to v exists and destination
            // v is colored with same color as u
            else if (colorArr[G[u][v]] == colorArr[u])
                return false;
        }
    }
}
```

```
        }
        return true;
    }

// This function returns true if graph G[V] [V] is two
// colorable, else false
bool twoColor(vector<int>G[], int N){

    // Create a color array to store colors assigned
    // to all veritces. Vertex number is used as index
    // in this array. The value '-1' of colorArr[i]
    // is used to indicate that no color is assigned
    // to vertex 'i'. The value 1 is used to indicate
    // first color is assigned and value 0 indicates
    // second color is assigned.
    int colorArr[N];
    for (int i = 1; i <= N; ++i)
        colorArr[i] = -1;

    // As we are dealing with graph, the input might
    // come as a forest, thus start coloring from a
    // node and if true is returned we'll know that
    // we successfully colored the subpart of our
    // forest and we start coloring again from a new
    // uncolored node. This way we cover the entire forest.
    for (int i = 1; i <= N; i++)
        if (colorArr[i] == -1)
            if (twoColorUtil(G, i, N, colorArr) == false)
                return false;

    return true;
}

// Returns false if an odd cycle is present else true
// int info[][] is the information about our graph
// int n is the number of nodes
// int m is the number of informations given to us
bool isOddSum(int info[][3], int n, int m){

    // Declaring adjacency list of a graph
    // Here at max, we can encounter all the edges with
    // even weight thus there will be 1 pseudo node
    // for each edge
    vector<int> G[2*n];

    int pseudo = n+1;
    int pseudo_count = 0;
```

```
for (int i=0; i<m; i++){

    // For odd weight edges, we directly add it
    // in our graph
    if (info[i][2]%2 == 1){

        int u = info[i][0];
        int v = info[i][1];
        G[u].push_back(v);
        G[v].push_back(u);
    }

    // For even weight edges, we break it
    else{

        int u = info[i][0];
        int v = info[i][1];

        // Entering a pseudo node between u---v
        G[u].push_back(pseudo);
        G[pseudo].push_back(u);
        G[v].push_back(pseudo);
        G[pseudo].push_back(v);

        // Keeping a record of number of pseudo nodes
        // inserted
        pseudo_count++;

        // Making a new pseudo node for next time
        pseudo++;
    }
}

// We pass number graph G[][] and total number
// of node = actual number of nodes + number of
// pseudo nodes added.
return twoColor(G,n+pseudo_count);
}

// Driver function
int main() {

    // 'n' correspond to number of nodes in our
    // graph while 'm' correspond to the number
    // of information about this graph.
    int n = 4, m = 3;
    int info[4][3] = {{1, 2, 12},
                      {2, 3, 1},
```

```
{4, 3, 1},  
{4, 1, 20}};  
  
// This function break the even weighted edges in  
// two parts. Makes the adjacency representation  
// of the graph and sends it for two coloring.  
if (isOddSum(info, n, m) == true)  
    cout << "No\n";  
else  
    cout << "Yes\n";  
  
return 0;  
}
```

Output:

No

Source

<https://www.geeksforgeeks.org/check-if-there-is-a-cycle-with-odd-weight-sum-in-an-undirected-graph/>

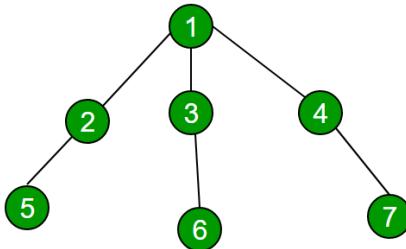
Chapter 101

Check if two nodes are on same path in a tree

Check if two nodes are on same path in a tree - GeeksforGeeks

Given a tree (not necessarily a binary tree) and a number of queries such that every query takes two nodes of tree as parameters. For every query pair, find if two nodes are on the same path from root to the bottom.

For example, consider the below tree, if given queries are (1, 5), (1, 6) and (2, 6), then answers should be true, true and false respectively.



Note that 1 and 5 lie on same root to leaf path, so do 1 and 6, but 2 and 6 are not on same root to leaf path.

It is obvious that Depth First Search technique is to be used to solve above problem, the main problem is how to respond to multiple queries fast. Here our graph is a tree which may have any number of children. Now DFS in a tree if started from root node proceeds in a depth search manner i.e. Suppose root has three children and those children have only one child with them so if DFS is started then it first visits the first child of root node then will go deep to the child of that node. The situation with a small tree can be shown as follows:

The order of visiting the nodes will be – 1 2 5 3 6 4 7 .

Thus other children nodes are visited later until completely one child is successfully visited till depth. To simplify this if we assume that we have a watch in our hand and we start

walking from root in DFS manner.

Intime – When we visit the node for the first time

Outtime- If we again visit the node later but there is no children unvisited we call it outtime,

Note: Any node in its sub-tree will always have intime < its children (or children of children) because it is always visited first before children (due to DFS) and will have outtime > all nodes in its sub-tree because before noting the outtime it waits for all of its children to be marked visited.

For any two nodes u, v if they are in same path then,

```
Intime[v] < Intime[u] and Outtime[v] > Outtime[u]
OR
Intime[u] < Intime[v] and Outtime[u] > Outtime[v]
```

- If given pair of nodes follows any of the two conditions, then they are on the same root to leaf path.
- Else not on same path (If two nodes are on different paths it means that no one is in subtree of each other).

Pseudo Code

We use a global variable time which will be incremented as dfs for a node begins and will also be incremented after

```
DFS(v)
    increment timer
    Intime[v] = timer
    mark v as visited
    for all u that are children of v
        DFS(u)
    increment timer
    Outtime[v] = timer
end
```

Time Complexity – O(n) for preprocessing and O(1) per query.

Implementation:

Below is C++ implementation of above pseudo code.

```
// C++ program to check if given pairs lie on same
// path or not.
#include<bits/stdc++.h>
using namespace std;
const int MAX = 100001;
```

```
// To keep track of visited vertices in DFS
bool visit[MAX] = {0};

// To store start and end time of all vertices
// during DFS.
int intime[MAX];
int outtime[MAX];

// initially timer is zero
int timer = 0;

// Does DFS of given graph and fills arrays
// intime[] and outtime[]. These arrays are used
// to answer given queries.
void dfs(vector<int> graph[], int v)
{
    visit[v] = true;

    // Increment the timer as you enter
    // the recursion for v
    ++timer;

    // Upgrade the in time for the vertex
    intime[v] = timer;

    vector<int>::iterator it = graph[v].begin();
    while (it != graph[v].end())
    {
        if (visit[*it]==false)
            dfs(graph, *it);
        it++;
    }

    // increment the timer as you exit the
    // recursion for v
    ++timer;

    // upgrade the outtime for that node
    outtime[v] = timer;
}

// Returns true if 'u' and 'v' lie on same root to leaf path
// else false.
bool query(int u, int v)
{
    return ( (intime[u]<intime[v] && outtime[u]>outtime[v]) ||
             (intime[v]<intime[u] && outtime[v]>outtime[u]) );
}
```

```
// Driver code
int main()
{
    // Let us create above shown tree
    int n = 9; // total number of nodes
    vector<int> graph[n+1];
    graph[1].push_back(2);
    graph[1].push_back(3);
    graph[3].push_back(6);
    graph[2].push_back(4);
    graph[2].push_back(5);
    graph[5].push_back(7);
    graph[5].push_back(8);
    graph[5].push_back(9);

    // Start dfs (here root node is 1)
    dfs(graph, 1);

    // below are calls for few pairs of nodes
    query(1, 5)? cout << "Yes\n" : cout << "No\n";
    query(2, 9)? cout << "Yes\n" : cout << "No\n";
    query(2, 6)? cout << "Yes\n" : cout << "No\n";

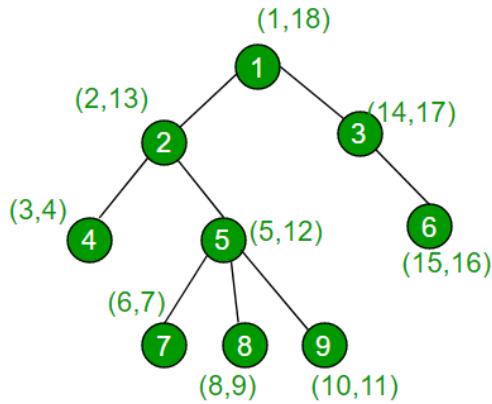
    return 0;
}
```

Output:

```
Yes
Yes
No
```

Illustration:

From the diagram below to understand more we can have some examples. DFS algorithm as modified above will result in the following intime and outtime for the vertex of the tree as labelled there. Now we will consider all the cases.



Case 1 : Nodes 2 and 4 : Node 2 has intime less than node 4 but since 4 is in its sub tree so it will have a greater exit time than 4 . Thus condition is valid and both are on same path.

Case 2 : Nodes 7 and 6 : Node 7 has intime less than node 6 but since both nodes are not in each other's sub tree so their exit time does not follow the required condition.

Source

<https://www.geeksforgeeks.org/check-if-two-nodes-are-on-same-path-in-a-tree/>

Chapter 102

Check loop in array according to given constraints

Check loop in array according to given constraints - GeeksforGeeks

Given an array $\text{arr}[0..n-1]$ of positive and negative numbers we need to find if there is a cycle in array with given rules of movements. If a number at an i index is positive, then move $\text{arr}[i]\%n$ forward steps, i.e., next index to visit is $(i + \text{arr}[i])\%n$. Conversely, if it's negative, move backward $\text{arr}[i]\%n$ steps i.e., next index to visit is $(i - \text{arr}[i])\%n$. Here n is size of array. If value of $\text{arr}[i]\%n$ is zero, then it means no move from index i .

Examples:

Input: $\text{arr}[] = \{2, -1, 1, 2, 2\}$
Output: Yes

Explanation: There is a loop in this array because 0 moves to 2, 2 moves to 3, and 3 moves to 0.

Input : $\text{arr}[] = \{1, 1, 1, 1, 1, 1\}$
Output : Yes
Whole array forms a loop.

Input : $\text{arr}[] = \{1, 2\}$
Output : No
We move from 0 to index 1. From index 1, there is no move as $2\%n$ is 0. Note that n is 2.

Note that self loops are not considered a cycle. For example $\{0\}$ is not cyclic.

The idea is to form a directed graph of array elements using given set of rules. While forming the graph we don't make self loops as value $\text{arr}[i]\%n$ equals to 0 means no moves. Finally

our task reduces to [detecting cycle in a directed graph](#). For detecting cycle, we use DFS and in DFS if reach a node which is visited and recursion call stack, we say there is a cycle.

```
// C++ program to check if a given array is cyclic or not
#include<bits/stdc++.h>
using namespace std;

// A simple Graph DFS based recursive function to check if
// there is cycle in graph with vertex v as root of DFS.
// Refer below article for details.
// https://www.geeksforgeeks.org/detect-cycle-in-a-graph/
bool isCycleRec(int v, vector<int>adj[],
                 vector<bool> &visited, vector<bool> &recur)
{
    visited[v] = true;
    recur[v] = true;
    for (int i=0; i<adj[v].size(); i++)
    {
        if (visited[adj[v][i]] == false)
        {
            if (isCycleRec(adj[v][i], adj, visited, recur))
                return true;
        }
        // There is a cycle if an adjacent is visited
        // and present in recursion call stack recur[]
        else if (visited[adj[v][i]] == true &&
                  recur[adj[v][i]] == true)
            return true;
    }

    recur[v] = false;
    return false;
}

// Returns true if arr[] has cycle
bool isCycle(int arr[], int n)
{
    // Create a graph using given moves in arr[]
    vector<int>adj[n];
    for (int i=0; i<n; i++)
        if (i != (i+arr[i]+n)%n)
            adj[i].push_back((i+arr[i]+n)%n);

    // Do DFS traversal of graph to detect cycle
    vector<bool> visited(n, false);
    vector<bool> recur(n, false);
    for (int i=0; i<n; i++)
```

```
        if (visited[i]==false)
            if (isCycleRec(i, adj, visited, recur))
                return true;
        return true;
    }

// Driver code
int main(void)
{
    int arr[] = {2, -1, 1, 2, 2};
    int n = sizeof(arr)/sizeof(arr[0]);
    if (isCycle(arr, n))
        cout << "Yes" << endl;
    else
        cout << "No" << endl;
    return 0;
}
```

Output :

Yes

Source

<https://www.geeksforgeeks.org/check-loop-array-according-given-constraints/>

Chapter 103

Check whether given degrees of vertices represent a Graph or Tree

Check whether given degrees of vertices represent a Graph or Tree - GeeksforGeeks

Given the number of vertices and the degree of each vertex where vertex numbers are 1, 2, 3,...n. The task is to identify whether it is a graph or a tree. It may be assumed that the graph is connected.

```
Input : 5
        2 3 1 1 1
Output : Tree
Explanation : The input array indicates that
              vertex one has degree 2, vertex
              two has degree 3, vertices 3, 4
              and 5 have degree 1.
              1
              / \
              2   3
              / \
              4   5
```

```
Input : 3
        2 2 2
Output : Graph
          1
          / \
          2 - 3
```

The **degree of a vertex** is given by the number of edges incident or leaving from it. This can simply be done using the properties of trees like –

1. Tree is **connected** and has **no cycles** while graphs can have cycles.
2. Tree has exactly **n-1** edges while there is no such constraint for graph.
3. It is given that the input graph is connected. We need at least n-1 edges to connect n nodes.

If we take the sum of all the degrees, each edge will be counted twice. Hence, for a tree with **n** vertices and **n - 1** edges, sum of all degrees should be **2 * (n - 1)**. Please refer [Handshaking Lemma](#) for details.

So basically we need to check if sum of all degrees is $2*(n-1)$ or not.

C++

```
// C++ program to check whether input degree
// sequence is graph or tree
#include<bits/stdc++.h>
using namespace std;

// Function returns true for tree
// false for graph
bool check(int degree[], int n)
{
    // Find sum of all degrees
    int deg_sum = 0;
    for (int i = 0; i < n; i++)
        deg_sum += degree[i];

    // Graph is tree if sum is equal to 2(n-1)
    return (2*(n-1) == deg_sum);
}

// Driver program to test above function
int main()
{
    int n = 5;
    int degree[] = {2, 3, 1, 1, 1};

    if (check(degree, n))
        cout << "Tree";
    else
        cout << "Graph";

    return 0;
}
```

Python

```
# Python program to check whether input degree
# sequence is graph or tree
def check(degree, n):

    # Find sum of all degrees
    deg_sum = sum(degree)

    # It is tree if sum is equal to 2(n-1)
    if (2*(n-1) == deg_sum):
        return True
    else:
        return False

#main
n = 5
degree = [2, 3, 1, 1, 1];
if (check(degree, n)):
    print "Tree"
else:
    print "Graph"
```

Output:

Tree

Source

<https://www.geeksforgeeks.org/check-whether-given-degrees-vertices-represent-graph-tree/>

Chapter 104

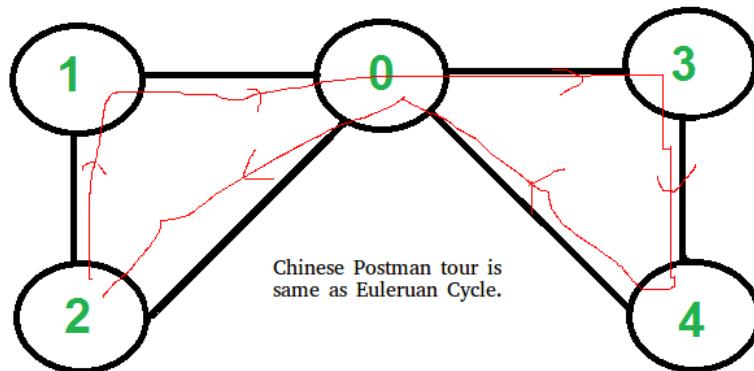
Chinese Postman or Route Inspection Set 1 (introduction)

Chinese Postman or Route Inspection Set 1 (introduction) - GeeksforGeeks

Chinese Postman Problem is a variation of **Eulerian circuit** problem for undirected graphs. An Euler Circuit is a closed walk that covers every edge once starting and ending position is same. Chinese Postman problem is defined for connected and undirected graph. The problem is to find shortest path or circuitry that visits every edge of the graph at least once.

If input graph contains Euler Circuit, then a solution of the problem is Euler Circuit

An undirected and connected graph has Eulerian cycle if “all vertices have even degree”.



The graph has Eulerian Cycles, for example "2 1 0 3 4 0 2"
Note that all vertices have even degree

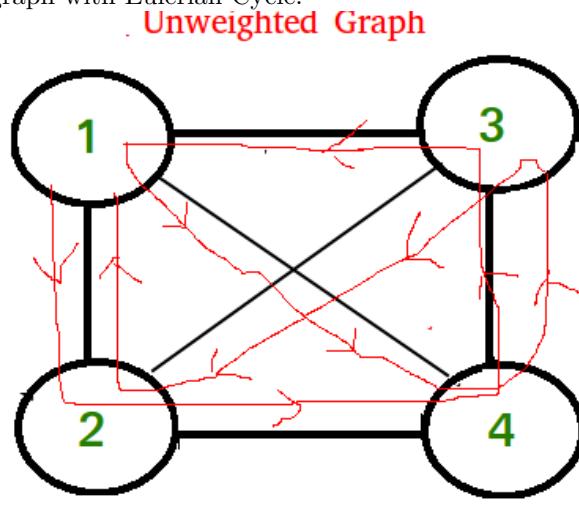
It doesn't matter whether graph is weighted or unweighted, the Chinese Postman Route is always same as Eulerian Circuit if it exists. In weighted graph the minimum possible weight

of Postman tour is sum of all edge weights which we get through Eulerian Circuit. We can't get a shorter route as we must visit all edges at-least once.

If input graph does NOT contain Euler Circuit

In this case, the task reduces to following.

- 1) In unweighted graph, minimum number of edges to duplicate so that the given graph converts to a graph with Eulerian Cycle.

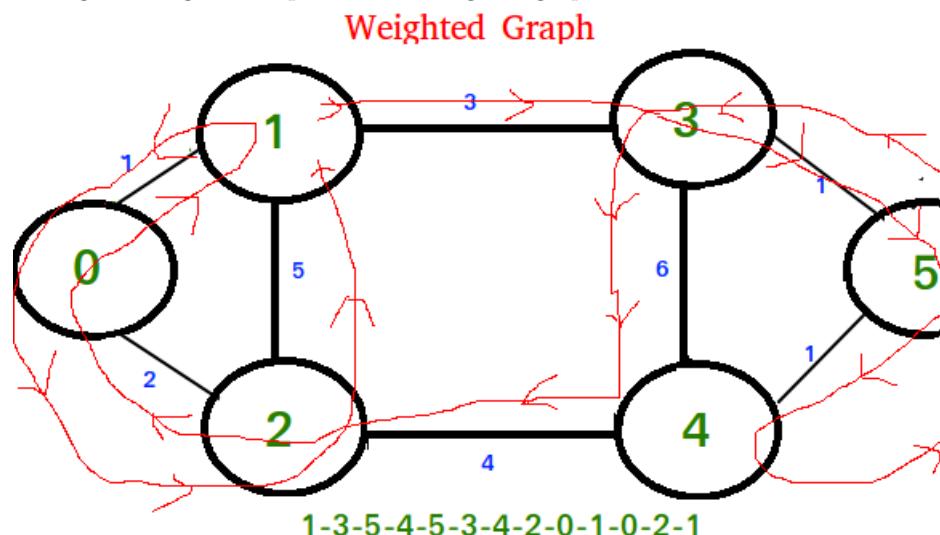


1-2-4-3-1-4-3-2-1

Chinese Postman Tour for Non-Eulerian Graph

The tour has two edges traversed twice, 1-2 and 3-4

- 2) In weighted graph, minimum total weight of edges to duplicate so that given graph converts to a graph with Eulerian Cycle.



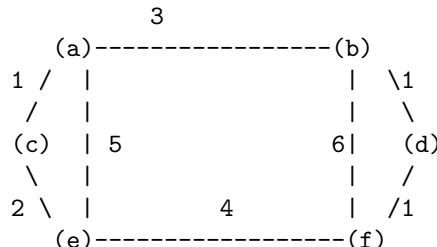
Chinese Postman Tour : Total weight = $3+1+1+1+1+6+4+2+1+1$

verts to a graph with Eulerian Cycle.

Algorithm to find shortest closed path or optimal Chinese postman route in a weighted graph that may not be Eulerian.

- step 1 : If graph is Eulerian, return sum of all edge weights. Else do following steps.
- step 2 : We find all the vertices with odd degree
- step 3 : List all possible pairings of odd vertices
For n odd vertices total number of pairings possible are, $(n-1) * (n-3) * (n-5) \dots * 1$
- step 4 : For each set of pairings, find the shortest path connecting them.
- step 5 : Find the pairing with minimum shortest path connecting pairs.
- step 6 : Modify the graph by adding all the edges that have been found in step 5.
- step 7 : Weight of Chinese Postman Tour is sum of all edges in the modified graph.
- step 8 : Print Euler Circuit of the modified graph.
This Euler Circuit is Chinese Postman Tour.

Illustration :



As we see above graph does not contain Eulerian circuit because it has odd degree vertices [a, b, e, f] they all are odd degree vertices .

First we make all possible pairs of odd degree vertices [ae, bf], [ab, ef], [af, eb]
so pairs with min sum of weight are [ae, bc] :
 $ae = (ac + ce = 3)$, $bf = (ad + df = 2)$
Total : 5

We add edges ac, ce, ad and df to the original graph and create a modified graph.

Optimal chinese postman route is of length : $5 + 23 = 28$ [23 = sum of all edges of modified graph]

Chinese Postman Route :

a - b - d - f - d - b - f - e - c - a - c - e - a

This route is Euler Circuit of the modified graph.

References:

https://en.wikipedia.org/wiki/Route_inspection_problem

<http://www.suffolkmaths.co.uk/pages/Maths%20Projects/Projects/Topology%20and%20Graph%20Theory/Chinese%20Postman%20Problem.pdf>

Source

<https://www.geeksforgeeks.org/chinese-postman-route-inspection-set-1-introduction/>

Chapter 105

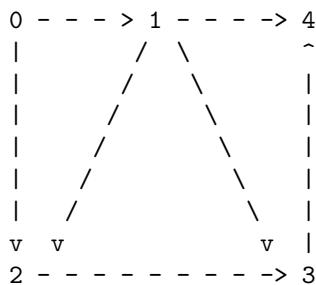
Clone a Directed Acyclic Graph

Clone a Directed Acyclic Graph - GeeksforGeeks

A directed acyclic graph (DAG) is a graph which doesn't contain a cycle and has directed edges. We are given a DAG, we need to clone it, i.e., create another graph that has copy of its vertices and edges connecting them.

Examples:

Input :



Output : Printing the output of the cloned graph gives:

```
0-1
1-2
2-3
3-4
1-3
1-4
0-2
```

To clone a DAG without storing the graph itself within a hash (or dictionary in Python). To clone, it we basically do a depth-first traversal of the nodes, taking original node's value and

initializing new neighboring nodes with the same value, recursively doing, till the original graph is fully traversed. Below is the recursive approach to cloning a DAG (in Python). We make use of dynamic lists in Python, append operation to this list happens in constant time, hence, fast and efficient initialization of the graph.

```

# Python program to clon a directed acyclic graph.

# Class to create a new graph node
class Node():

    # key is the value of the node
    # adj will be holding a dynamic
    # list of all Node type neighboring
    # nodes
    def __init__(self, key = None, adj = None):
        self.key = key
        self.adj = adj

# Function to print a graph, depth-wise, recursively
def printGraph(startNode, visited):

    # Visit only those nodes who have any
    # neighboring nodes to be traversed
    if startNode.adj is not None:

        # Loop through the neighboring nodes
        # of this node. If source node not already
        # visited, print edge from source to
        # neighboring nodes. After visiting all
        # neighbors of source node, mark its visited
        # flag to true
        for i in startNode.adj:
            if visited[startNode.key] == False :
                print("edge %s-%s:%s-%s"%(hex(id(startNode)), hex(id(i)), startNode.key, i.key))
                if visited[i.key] == False:
                    printGraph(i, visited)
                    visited[i.key] = True

# Function to clone a graph. To do this, we start
# reading the original graph depth-wise, recursively
# If we encounter an unvisited node in original graph,
# we initialize a new instance of Node for
# cloned graph with key of original node
def cloneGraph(oldSource, newSource, visited):
    clone = None
    if visited[oldSource.key] is False and oldSource.adj is not None:
        for old in oldSource.adj:

```

```

# Below check is for backtracking, so new
# nodes don't get initialized everytime
if clone is None or(clone is not None and clone.key != old.key):
    clone = Node(old.key, [])
newSource.adj.append(clone)
cloneGraph(old, clone, visited)

# Once, all neighbors for that particular node
# are created in cloned graph, code backtracks
# and exits from that node, mark the node as
# visited in original graph, and traverse the
# next unvisited
visited[old.key] = True
return newSource

# Creating DAG to be cloned
# In Python, we can do as many assignments of
# variables in one single line by using commas
n0, n1, n2 = Node(0, []), Node(1, []), Node(2, [])
n3, n4 = Node(3, []), Node(4)
n0.adj.append(n1)
n0.adj.append(n2)
n1.adj.append(n2)
n1.adj.append(n3)
n1.adj.append(n4)
n2.adj.append(n3)
n3.adj.append(n4)

# flag to check if a node is already visited.
# Stops indefinite looping during recursion
visited = [False]* (5)
print("Graph Before Cloning:-")
printGraph(n0, visited)

visited = [False]* (5)
print("\nCloning Process Starts")
clonedGraphHead = cloneGraph(n0, Node(n0.key, []), visited)
print("Cloning Process Completes.")

visited = [False]*(5)
print("\nGraph After Cloning:-")
printGraph(clonedGraphHead, visited)

```

Output:

```

Graph Before Cloning:-
edge 0x7fa03dd43878-0x7fa03dd43908:0-1

```

```
edge 0x7fa03dd43908-0x7fa03dd43950:1-2
edge 0x7fa03dd43950-0x7fa03dd43998:2-3
edge 0x7fa03dd43998-0x7fa03dd439e0:3-4
edge 0x7fa03dd43908-0x7fa03dd43998:1-3
edge 0x7fa03dd43908-0x7fa03dd439e0:1-4
edge 0x7fa03dd43878-0x7fa03dd43950:0-2
```

Cloning Process Starts
Cloning Process Completes.

Graph After Cloning:-

```
edge 0x7fa03dd43a28-0x7fa03dd43a70:0-1
edge 0x7fa03dd43a70-0x7fa03dd43ab8:1-2
edge 0x7fa03dd43ab8-0x7fa03dd43b00:2-3
edge 0x7fa03dd43b00-0x7fa03dd43b48:3-4
edge 0x7fa03dd43a70-0x7fa03dd43b90:1-3
edge 0x7fa03dd43a70-0x7fa03dd43bd8:1-4
edge 0x7fa03dd43a28-0x7fa03dd43c20:0-2
```

Creating the DAG by appending adjacent edges to the vertex happens in $O(1)$ time. Cloning of the graph takes $O(E+V)$ time.

Source

<https://www.geeksforgeeks.org/clone-directed-acyclic-graph/>

Chapter 106

Clone an Undirected Graph

Clone an Undirected Graph - GeeksforGeeks

Cloning of a [LinkedList](#) and a [Binary Tree](#) with random pointers has already been discussed. The idea behind cloning a graph is pretty much similar.

The idea is to do a [BFS traversal](#) of the graph and while visiting a node make a clone node of it (a copy of original node). If a node is encountered which is already visited then it already has a clone node.

How to keep track of the visited/cloned nodes?

A HashMap/Map is required in order to maintain all the nodes which have already been created.

Key stores: Reference/Address of original Node

Value stores: Reference/Address of cloned Node

A copy of all the graph nodes has been made, **how to connect clone nodes?**

While visiting the neighboring vertices of a node u get the corresponding cloned node for u , let's call that $cloneNodeU$, now visit all the neighboring nodes for u and for each neighbor find the corresponding clone node(if not found create one) and then push into the neighboring vector of $cloneNodeU$ node.

How to verify if the cloned graph is a correct?

Do a BFS traversal before and after the cloning of graph. In BFS traversal display the value of a node along with its address/reference.

Compare the order in which nodes are displayed, if the values are same but the address/reference is different for both the traversals than the cloned graph is correct.

C++

```
// A C++ program to Clone an Undirected Graph
#include<bits/stdc++.h>
using namespace std;

struct GraphNode
{
```

```

int val;

//A neighbour vector which contains addresses to
//all the neighbours of a GraphNode
vector<GraphNode*> neighbours;
};

// A function which clones a Graph and
// returns the address to the cloned
// src node
GraphNode *cloneGraph(GraphNode *src)
{
    //A Map to keep track of all the
    //nodes which have already been created
    map<GraphNode*, GraphNode*> m;
    queue<GraphNode*> q;

    // Enqueue src node
    q.push(src);
    GraphNode *node;

    // Make a clone Node
    node = new GraphNode();
    node->val = src->val;

    // Put the clone node into the Map
    m[src] = node;
    while (!q.empty())
    {
        //Get the front node from the queue
        //and then visit all its neighbours
        GraphNode *u = q.front();
        q.pop();
        vector<GraphNode *> v = u->neighbours;
        int n = v.size();
        for (int i = 0; i < n; i++)
        {
            // Check if this node has already been created
            if (m[v[i]] == NULL)
            {
                // If not then create a new Node and
                // put into the HashMap
                node = new GraphNode();
                node->val = v[i]->val;
                m[v[i]] = node;
                q.push(v[i]);
            }
        }
    }
}

```

```

        // add these neighbours to the cloned graph node
        m[u]->neighbours.push_back(m[v[i]]);
    }
}

// Return the address of cloned src Node
return m[src];
}

// Build the desired graph
GraphNode *buildGraph()
{
    /*
        Note : All the edges are Undirected
        Given Graph:
        1--2
        | |
        4--3
    */
    GraphNode *node1 = new GraphNode();
    node1->val = 1;
    GraphNode *node2 = new GraphNode();
    node2->val = 2;
    GraphNode *node3 = new GraphNode();
    node3->val = 3;
    GraphNode *node4 = new GraphNode();
    node4->val = 4;
    vector<GraphNode *> v;
    v.push_back(node2);
    v.push_back(node4);
    node1->neighbours = v;
    v.clear();
    v.push_back(node1);
    v.push_back(node3);
    node2->neighbours = v;
    v.clear();
    v.push_back(node2);
    v.push_back(node4);
    node3->neighbours = v;
    v.clear();
    v.push_back(node3);
    v.push_back(node1);
    node4->neighbours = v;
    return node1;
}

// A simple bfs traversal of a graph to
// check for proper cloning of the graph

```

```

void bfs(GraphNode *src)
{
    map<GraphNode*, bool> visit;
    queue<GraphNode*> q;
    q.push(src);
    visit[src] = true;
    while (!q.empty())
    {
        GraphNode *u = q.front();
        cout << "Value of Node " << u->val << "\n";
        cout << "Address of Node " << u << "\n";
        q.pop();
        vector<GraphNode *> v = u->neighbours;
        int n = v.size();
        for (int i = 0; i < n; i++)
        {
            if (!visit[v[i]])
            {
                visit[v[i]] = true;
                q.push(v[i]);
            }
        }
    }
    cout << endl;
}

// Driver program to test above function
int main()
{
    GraphNode *src = buildGraph();
    cout << "BFS Traversal before cloning\n";
    bfs(src);
    GraphNode *newsrcl = cloneGraph(src);
    cout << "BFS Traversal after cloning\n";
    bfs(newsrc);
    return 0;
}

```

Java

```

// Java program to Clone an Undirected Graph
import java.util.*;

// GraphNode class represents each
// Node of the Graph
class GraphNode
{
    int val;

```

```

// A neighbour Vector which contains references to
// all the neighbours of a GraphNode
Vector<GraphNode> neighbours;
public GraphNode(int val)
{
    this.val = val;
    neighbours = new Vector<GraphNode>();
}
}

class Graph
{
    // A method which clones the graph and
    // returns the reference of new cloned source node
    public GraphNode cloneGraph(GraphNode source)
    {
        Queue<GraphNode> q = new LinkedList<GraphNode>();
        q.add(source);

        // An HashMap to keep track of all the
        // nodes which have already been created
        HashMap<GraphNode,GraphNode> hm =
            new HashMap<GraphNode,GraphNode>();

        //Put the node into the HashMap
        hm.put(source,new GraphNode(source.val));

        while (!q.isEmpty())
        {
            // Get the front node from the queue
            // and then visit all its neighbours
            GraphNode u = q.poll();

            // Get corresponding Cloned Graph Node
            GraphNode cloneNodeU = hm.get(u);
            if (u.neighbours != null)
            {
                Vector<GraphNode> v = u.neighbours;
                for (GraphNode graphNode : v)
                {
                    // Get the corresponding cloned node
                    // If the node is not cloned then we will
                    // simply get a null
                    GraphNode cloneNodeG = hm.get(graphNode);

                    // Check if this node has already been created
                    if (cloneNodeG == null)

```

```

    {
        q.add(graphNode);

        // If not then create a new Node and
        // put into the HashMap
        cloneNodeG = new GraphNode(graphNode.val);
        hm.put(graphNode,cloneNodeG);
    }

    // add the 'cloneNodeG' to neighbour
    // vector of the cloneNodeG
    cloneNodeU.neighbours.add(cloneNodeG);
}
}

// Return the reference of cloned source Node
return hm.get(source);
}

// Build the desired graph
public GraphNode buildGraph()
{
    /*
        Note : All the edges are Undirected
        Given Graph:
        1--2
        |   |
        4--3
    */
    GraphNode node1 = new GraphNode(1);
    GraphNode node2 = new GraphNode(2);
    GraphNode node3 = new GraphNode(3);
    GraphNode node4 = new GraphNode(4);
    Vector<GraphNode> v = new Vector<GraphNode>();
    v.add(node2);
    v.add(node4);
    node1.neighbours = v;
    v = new Vector<GraphNode>();
    v.add(node1);
    v.add(node3);
    node2.neighbours = v;
    v = new Vector<GraphNode>();
    v.add(node2);
    v.add(node4);
    node3.neighbours = v;
    v = new Vector<GraphNode>();
    v.add(node3);
}

```

```

        v.add(node1);
        node4.neighbours = v;
        return node1;
    }

    // BFS traversal of a graph to
    // check if the cloned graph is correct
    public void bfs(GraphNode source)
    {
        Queue<GraphNode> q = new LinkedList<GraphNode>();
        q.add(source);
        HashMap<GraphNode,Boolean> visit =
            new HashMap<GraphNode,Boolean>();
        visit.put(source,true);
        while (!q.isEmpty())
        {
            GraphNode u = q.poll();
            System.out.println("Value of Node " + u.val);
            System.out.println("Address of Node " + u);
            if (u.neighbours != null)
            {
                Vector<GraphNode> v = u.neighbours;
                for (GraphNode g : v)
                {
                    if (visit.get(g) == null)
                    {
                        q.add(g);
                        visit.put(g,true);
                    }
                }
            }
        }
        System.out.println();
    }

    // Driver code
    class Main
    {
        public static void main(String args[])
        {
            Graph graph = new Graph();
            GraphNode source = graph.buildGraph();
            System.out.println("BFS traversal of a graph before cloning");
            graph.bfs(source);
            GraphNode newSource = graph.cloneGraph(source);
            System.out.println("BFS traversal of a graph after cloning");
            graph.bfs(newSource);
        }
    }
}

```

```
    }  
}
```

Output in Java:

```
BFS traversal of a graph before cloning  
Value of Node 1  
Address of Node GraphNode@15db9742  
Value of Node 2  
Address of Node GraphNode@6d06d69c  
Value of Node 4  
Address of Node GraphNode@7852e922  
Value of Node 3  
Address of Node GraphNode@4e25154f  
  
BFS traversal of a graph after cloning  
Value of Node 1  
Address of Node GraphNode@70dea4e  
Value of Node 2  
Address of Node GraphNode@5c647e05  
Value of Node 4  
Address of Node GraphNode@33909752  
Value of Node 3  
Address of Node GraphNode@55f96302
```

Output in C++:

```
BFS Traversal before cloning  
Value of Node 1  
Address of Node 0x24ccc20  
Value of Node 2  
Address of Node 0x24ccc50  
Value of Node 4  
Address of Node 0x24cccb0  
Value of Node 3  
Address of Node 0x24ccc80  
  
BFS Traversal after cloning  
Value of Node 1  
Address of Node 0x24cd030  
Value of Node 2  
Address of Node 0x24cd0e0  
Value of Node 4  
Address of Node 0x24cd170  
Value of Node 3  
Address of Node 0x24cd200
```

Source

<https://www.geeksforgeeks.org/clone-an-undirected-graph/>

Chapter 107

Clustering Coefficient in Graph Theory

Clustering Coefficient in Graph Theory - GeeksforGeeks

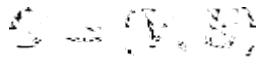
In graph theory, a clustering coefficient is a measure of the degree to which nodes in a graph tend to cluster together. Evidence suggests that in most real-world networks, and in particular social networks, nodes tend to create tightly knit groups characterized by a relatively high density of ties; this likelihood tends to be greater than the average probability of a tie randomly established between two nodes (Holland and Leinhardt, 1971; Watts and Strogatz, 1998).

Two versions of this measure exist: the global and the local. The global version was designed to give an overall indication of the clustering in the network, whereas the local gives an indication of the embeddedness of single nodes.

Global clustering coefficient

The global clustering coefficient is based on triplets of nodes. A triplet consists of three connected nodes. A triangle therefore includes three closed triplets, one centered on each of the nodes (n.b. this means the three triplets in a triangle come from overlapping selections of nodes). The global clustering coefficient is the number of closed triplets (or $3 \times$ triangles) over the total number of triplets (both open and closed). The first attempt to measure it was made by Luce and Perry (1949). This measure gives an indication of the clustering in the whole network (global), and can be applied to both undirected and directed networks.

Local clustering coefficient

A graph  formally consists of a set of vertices V and a set of edges E between them. An edge  connects vertex  with vertex .

The neighborhood  for a vertex  is defined as its immediately connected neighbors as follows:

$$N_i = \{v_j : e_{ij} \in E, e_{ji} \in E\}$$

We define N_i as the number of vertices, $|N_i|$, in the neighbourhood, N_i , of a vertex.

The local clustering coefficient C_i for a vertex v_i is then given by the proportion of links between the vertices within its neighborhood divided by the number of links that could possibly exist between them. For a directed graph, e_{ij} is distinct from e_{ji} , and therefore for each neighborhood N_i there are $\binom{|N_i|}{2} - 1$ links that could exist among the vertices within the neighborhood ($|N_i|$ is the number of neighbors of a vertex). Thus, the local clustering coefficient for directed graphs is given as [2]

$$C_i = \frac{\sum_{j \in N_i} \sum_{k \in N_i} e_{jk}}{\binom{|N_i|}{2} - 1}$$

An undirected graph has the property that e_{ij} and e_{ji} are considered identical. There-

fore, if a vertex v_i has n_i neighbors, $\frac{n_i(n_i-1)}{2}$ edges could exist among the vertices within the neighborhood. Thus, the local clustering coefficient for undirected graphs can be defined as

$$C_i = \frac{\sum_{j \in N_i} \sum_{k \in N_i} e_{jk}}{\frac{n_i(n_i-1)}{2}}$$

Let $\lambda_G(v)$ be the number of triangles on $v \in V(G)$ for undirected graph G.

That is, $\lambda_G(v)$ is the number of sub-graphs of G with 3 edges and 3 vertices, one of which is v. Let $\tau_G(v)$ be the number of triples on $v \in G$. That is, $\tau_G(v)$ is the number of sub-graphs (not necessarily induced) with 2 edges and 3 vertices, one of which is v and such that v is incident to both edges. Then we can also define the clustering coefficient as

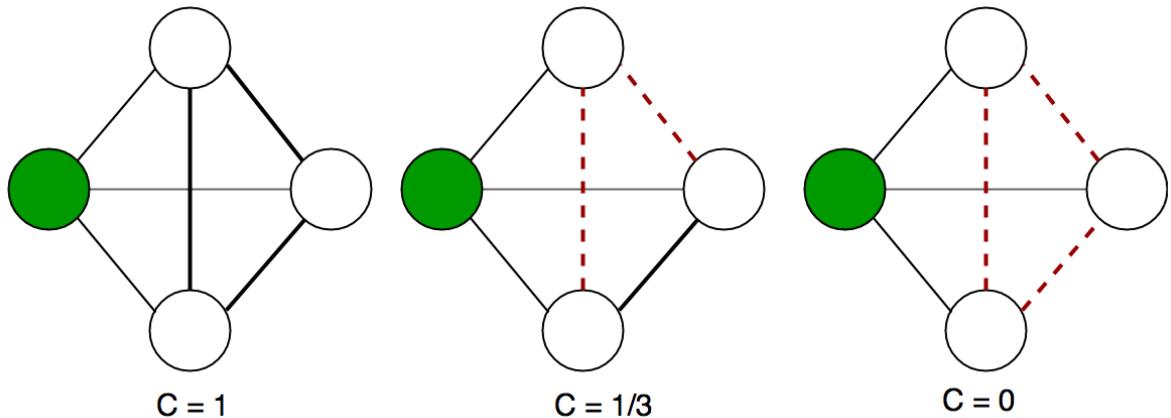
lue

$$\lambda_G(v) = \frac{\tau_G(v)}{\binom{n_i}{2}}$$

It is simple to show that the two preceding definitions are the same, since

$$\lambda_G(v) = C_i(v, 2) = \frac{1}{2} \lambda_G(v) - 1$$

These measures are 1 if every neighbor connected to v_i is also connected to every other vertex within the neighborhood, and 0 if no vertex that is connected to v_i connects to any other vertex that is connected to v_i .



Example local clustering coefficient on an undirected graph. The local clustering coefficient of the green node is computed as the proportion of connections among its neighbours.

Here is the code to implement the above clustering coefficient in a graph. It is a part of the networkx library and can be directly accessed using it.

```
def average_clustering(G, trials=1000):
    """Estimates the average clustering coefficient of G.

    The local clustering of each node in `G` is the
    fraction of triangles that actually exist over
    all possible triangles in its neighborhood.
    The average clustering coefficient of a graph
    `G` is the mean of local clusterings.

    This function finds an approximate average
    clustering coefficient for G by repeating `n`
    times (defined in `trials`) the following
    experiment: choose a node at random, choose
    two of its neighbors at random, and check if
    they are connected. The approximate coefficient
    is the fraction of triangles found over the
    number of trials [1]_.

    Parameters
    -----
    G : NetworkX graph

    trials : integer
        Number of trials to perform (default 1000).

    Returns
    -----
    c : float
```

Approximated average clustering coefficient.

```
"""
n = len(G)
triangles = 0
nodes = G.nodes()
for i in [int(random.random() * n) for i in range(trials)]:
    nbrs = list(G[nodes[i]])
    if len(nbrs) < 2:
        continue
    u, v = random.sample(nbrs, 2)
    if u in G[v]:
        triangles += 1
return triangles / float(trials)
```

Note: The above code is valid for undirected networks and not for the directed networks. The code below has been run on IDLE(Python IDE of windows). You would need to download the networkx library before you run this code. The part inside the curly braces represent the output. It is almost similar as Ipython(for Ububtu users).

```
>>> import networkx as nx
>>> G=nx.erdos_renyi_graph(10,0.4)
>>> cc=nx.average_clustering(G)
>>> cc
#Output of Global CC
0.0833333333333333
>>> c=nx.clustering(G)
>>> c
# Output of local CC
{0: 0.0, 1: 0.3333333333333333, 2: 0.0, 3: 0.0, 4: 0.0, 5: 0.0, 6: 0.0,
 7: 0.3333333333333333, 8: 0.0, 9: 0.1666666666666666}
```

The above two values give us the global clustering coefficient of a network as well as local clustering coefficient of a network.

Next into this series, we will talk about another centrality measure for any given network.

References

You can read more about the same at

https://en.wikipedia.org/wiki/Clustering_coefficient

<http://networkx.readthedocs.io/en/networkx-1.10/index.html>

Source

<https://www.geeksforgeeks.org/clustering-coefficient-graph-theory/>

Chapter 108

Comparison of Dijkstra's and Floyd-Warshall algorithms

Comparison of Dijkstra's and Floyd–Warshall algorithms - GeeksforGeeks

Main Purposes:

- [Dijkstra's Algorithm](#) is one example of a single-source shortest or SSSP algorithm, i.e., given a source vertex it finds shortest path from source to all other vertices.
- [Floyd Warshall Algorithm](#) is an example of all-pairs shortest path algorithm, meaning it computes the shortest path between all pair of nodes.

Time Complexities :

- Time Complexity of Dijkstra's Algorithm: $O(E \log V)$
- Time Complexity of Floyd Warshall: $O(V^3)$

Other Points:

- We can use Dijkstra's shortest path algorithm for finding all pair shortest paths by running it for every vertex. But time complexity of this would be $O(VE \log V)$ which can go $(V^3 \log V)$ in worst case.
- Another important differentiating factor between the algorithms is their working towards distributed systems. Unlike Dijkstra's algorithm, Floyd Warshall can be implemented in a distributed system, making it suitable for data structures such as Graph of Graphs (Used in Maps).
- Lastly Floyd Warshall works for negative edge but no [negative cycle](#), whereas Dijkstra's algorithm don't work for negative edges.

Source

<https://www.geeksforgeeks.org/comparison-dijkstras-floyd-warshall-algorithms/>

Chapter 109

Computer Networks Cuts and Network Flow

Computer Networks Cuts and Network Flow - GeeksforGeeks

The backbone analysis of any network is broadly accomplished by using Graph Theory and its Algorithms. The performance constraints are **Reliability**, **Delay/Throughput** and the goal is to **minimize cost**.

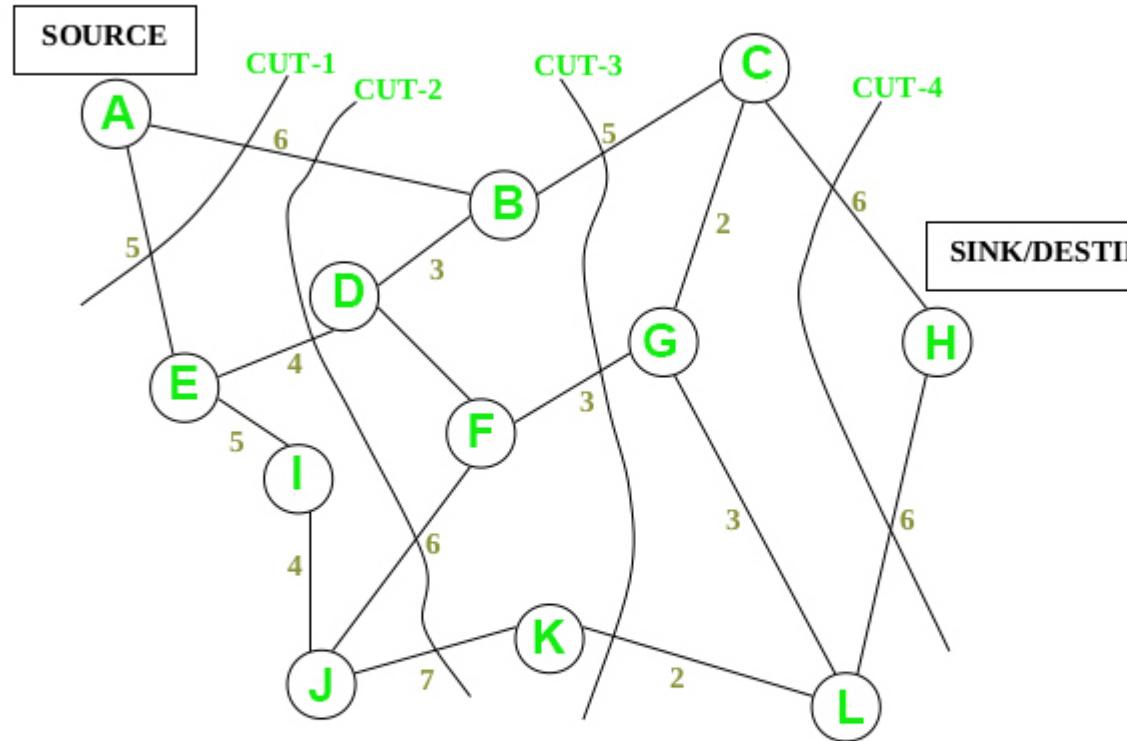
In the backbone designing of a network the concerned points and considerations are :

1. What should be the backbone topology ?
2. Assignment of Line Capacities.
3. Flow Assignment of the lines and hence the whole network.

Some Common Definitions :

- **Network** : A network is a circuit which is a sequence of adjacent nodes that comes back to the starting node. A circuit containing all the nodes of a graph is known as **Hamiltonian Circuit**.
- **Spanning Tree** : A spanning tree of a graph is a sub graph containing all the nodes of the graph and some collection of arcs chosen so that there is exactly one path between each pair of nodes.
- **Cut** : A concept from graph theory that is useful for modelling the carrying capacity of a network is the cut. An X-Y cut is a set of arcs whose removal disconnects node

X from node Y.



CUT-1 : AB, AE

CUT-2 : AB, ED, JF, JK

CUT-3 : BC, FG, KL

CUT-4 : CH, HL

Cuts in weighted Graph

Four A-H cuts are shown in the figure above.

Cut 1 : AB, AE (capacity = 11)

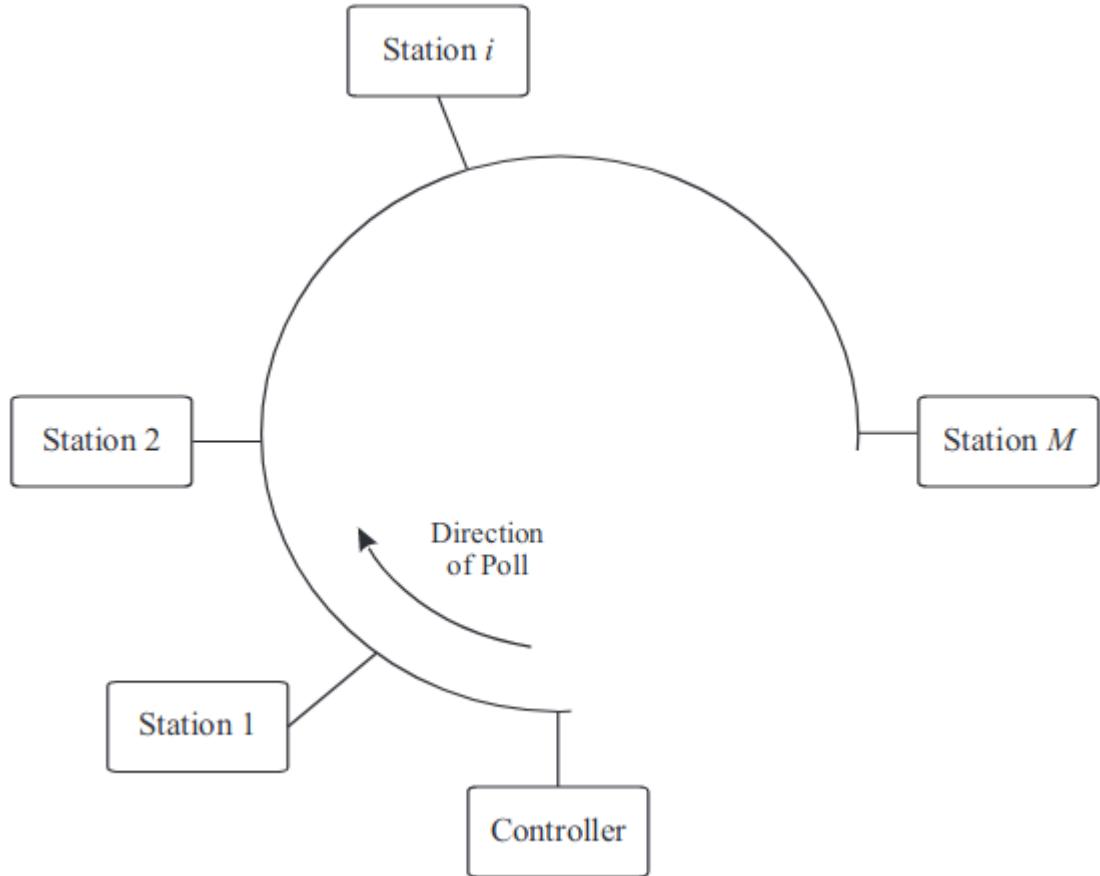
Cut 2 : AB, ED, JF, JK (capacity = 23)

Cut 3 : BC, FG, KL (capacity = 10)

Cut 4 : CH, CL (capacity = 12)

- **Minimal Cut :** It is one which replacement of any of its member reconnects the Graph. In other words, in a minimal cut, all the arcs are essential. The set of arcs AB, AE and FG form A-H cut, but the cut is not minimal, because restoring arc FG does not reconnect node A to node H.
- **Minimum Cut :** In a weighted graph each cut has capacity. A cut with minimum capacity is minimum cut. In the diagram shown above the Cut-3 with capacity=10, is the minimum cut.
- **Max Flow Min Cut Theorem :** The maximum flow between any two arbitrary nodes in any graph cannot exceed the capacity of the minimum cut separating those two nodes.
- **Polling :** Each station on the network is polled in some predetermined order. Between polls, stations accumulate messages in their queues but do not transmit until they are polled. The process by which terminals on a line are successively invited to send data. This may occur by having a master station use a polling list to invite terminals to send data, or by each terminal sending a poll message to the next terminal in sequence so that it might send data, or via the use of a token as in a token ring to control the sending of data.
- **Polling Sequences :** The sequence in which terminals are invited to transmit data. This may be based on a polling list in which terminal ids are stored in the sequence that they are to be polled.

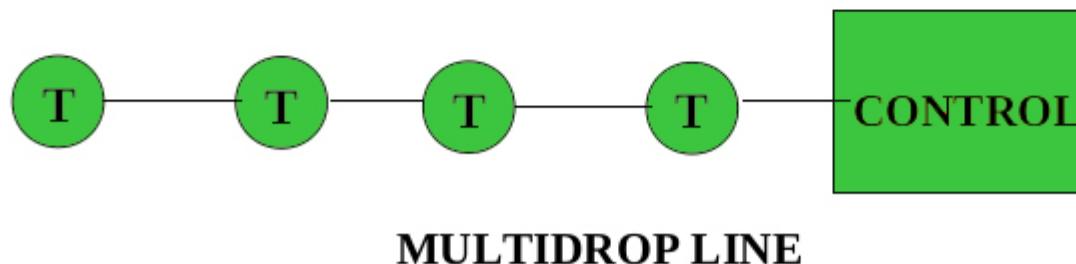
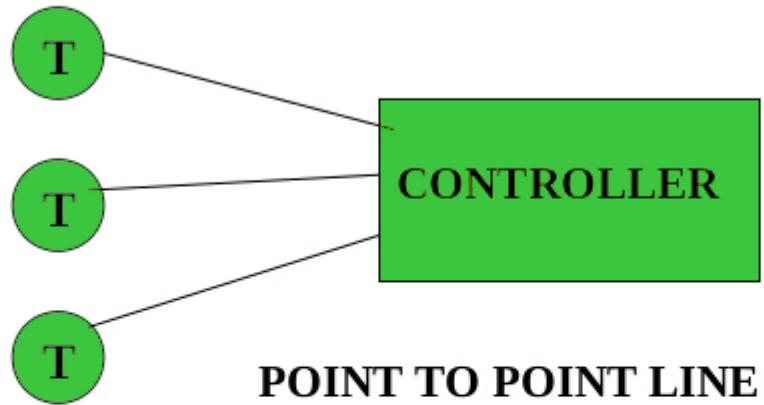
- **Polling Techniques :**



A typical Polling Network

1. **Roll call Polling :** A master station uses one or more polling lists to determine the next terminal in sequence to be polled. Each station has to be polled in turn by the central computer (controller). After the station has transmitted its backlog of messages, it notifies the central controller with a suffix to its last packet. After receiving this suffix packet, the controller sends a poll to the next station in the polling sequence.
2. **Hub Polling :** The terminal currently in polled mode polls the next terminal in sequence. In this case, the go-ahead (suffix) packet contains the next station address.
A monitoring channel must be provided to indicate to the appropriate station that it should start transmitting. Essentially the go-ahead is transmitted directly from one station to another.
3. **Token Passing :** A token is passed to the next device on the network (ring or bus) which may use it to transmit data or may let it pass to the next device.

ROLL CALL POLLING AND HUB POLLING



Roll Call Polling and Hub Polling

Source

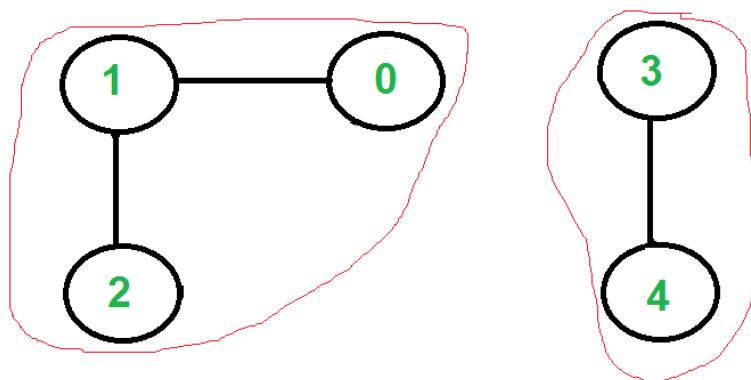
<https://www.geeksforgeeks.org/cuts-network-flow/>

Chapter 110

Connected Components in an undirected graph

Connected Components in an undirected graph - GeeksforGeeks

Given an undirected graph, print all connected components line by line. For example consider the following graph.



There are two connected components in above undirected graph

0 1 2

3 4

We strongly recommend to minimize your browser and try this yourself first.

We have discussed algorithms for finding strongly connected components in directed graphs in following posts.

[Kosaraju's algorithm for strongly connected components.](#)

[Tarjan's Algorithm to find Strongly Connected Components](#)

Finding connected components for an undirected graph is an easier task. We simple need to do either BFS or DFS starting from every unvisited vertex, and we get all strongly connected components. Below are steps based on DFS.

```
1) Initialize all vertices as not visited.  
2) Do following for every vertex 'v'.  
   (a) If 'v' is not visited before, call DFSUtil(v)  
   (b) Print new line character  
  
DFSUtil(v)  
1) Mark 'v' as visited.  
2) Print 'v'  
3) Do following for every adjacent 'u' of 'v'.  
   If 'u' is not visited, then recursively call DFSUtil(u)
```

Below is C++ implementation of above algorithm.

```
// C++ program to print connected components in  
// an undirected graph  
#include<iostream>  
#include <list>  
using namespace std;  
  
// Graph class represents a undirected graph  
// using adjacency list representation  
class Graph  
{  
    int V;      // No. of vertices  
  
    // Pointer to an array containing adjacency lists  
    list<int> *adj;  
  
    // A function used by DFS  
    void DFSUtil(int v, bool visited[]);  
public:  
    Graph(int V);    // Constructor  
    void addEdge(int v, int w);  
    void connectedComponents();  
};  
  
// Method to print connected components in an  
// undirected graph  
void Graph::connectedComponents()  
{  
    // Mark all the vertices as not visited  
    bool *visited = new bool[V];  
    for(int v = 0; v < V; v++)  
        visited[v] = false;  
  
    for (int v=0; v<V; v++)  
    {
```

```
        if (visited[v] == false)
        {
            // print all reachable vertices
            // from v
            DFSUtil(v, visited);

            cout << "\n";
        }
    }
}

void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices
    // adjacent to this vertex
    list<int>::iterator i;
    for(i = adj[v].begin(); i != adj[v].end(); ++i)
        if(!visited[*i])
            DFSUtil(*i, visited);
}

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

// method to add an undirected edge
void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
    adj[w].push_back(v);
}

// Drive program to test above
int main()
{
    // Create a graph given in the above diagram
    Graph g(5); // 5 vertices numbered from 0 to 4
    g.addEdge(1, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 4);

    cout << "Following are connected components \n";
```

```
g.connectedComponents();  
return 0;  
}
```

Output

```
0 1  
2 3 4
```

Time complexity of above solution is $O(V + E)$ as it does simple DFS for given graph.

Source

<https://www.geeksforgeeks.org/connected-components-in-an-undirected-graph/>

Chapter 111

Construct a graph from given degrees of all vertices

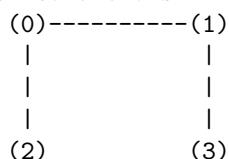
Construct a graph from given degrees of all vertices - GeeksforGeeks

This is a C++ program to generate a graph for a given fixed degree sequence. This algorithm generates a undirected graph for the given degree sequence. It does not include self-edge and multiple edges.

Examples:

```
Input : degrees[] = {2, 2, 1, 1}
Output : (0)  (1)  (2)  (3)
         (0)    0    1    1    0
         (1)    1    0    0    1
         (2)    1    0    0    0
         (3)    0    1    0    0
```

Explanation : We are given that there are four vertices with degree of vertex 0 as 2, degree of vertex 1 as 2, degree of vertex 2 as 1 and degree of vertex 3 as 1. Following is graph that follows given conditions.



Approach :

- 1- Take the input of the number of vertexes and their corresponding degree.
- 2- Declare adjacency matrix, mat[][] to store the graph.

- 3- To create the graph, create the first loop to connect each vertex ‘i’.
- 4- Second nested loop to connect the vertex ‘i’ to the every valid vertex ‘j’, next to it.
- 5- If the degree of vertex ‘i’ and ‘j’ are more than zero then connect them.
- 6- Print the adjacency matrix.

Based on the above explanation, below are implementations:

```

// C++ program to generate a graph for a
// given fixed degrees
#include <bits/stdc++.h>
using namespace std;

// A function to print the adjacency matrix.
void printMat(int degseq[], int n)
{
    // n is number of vertices
    int mat[n][n];
    memset(mat, 0, sizeof(mat));

    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {

            // For each pair of vertex decrement
            // the degree of both vertex.
            if (degseq[i] > 0 && degseq[j] > 0) {
                degseq[i]--;
                degseq[j]--;
                mat[i][j] = 1;
                mat[j][i] = 1;
            }
        }
    }

    // Print the result in specified format
    cout << "\n"
        << setw(3) << "    ";
    for (int i = 0; i < n; i++)
        cout << setw(3) << "(" << i << ")";
    cout << "\n\n";
    for (int i = 0; i < n; i++) {
        cout << setw(4) << "(" << i << ")";
        for (int j = 0; j < n; j++)
            cout << setw(5) << mat[i][j];
        cout << "\n";
    }
}

// driver program to test above function
int main()

```

```
{  
    int degseq[] = { 2, 2, 1, 1, 1 };  
    int n = sizeof(degseq) / sizeof(degseq[0]);  
    printMat(degseq, n);  
    return 0;  
}
```

Output:

	(0)	(1)	(2)	(3)	(4)
(0)	0	1	1	0	0
(1)	1	0	0	1	0
(2)	1	0	0	0	0
(3)	0	1	0	0	0
(4)	0	0	0	0	0

Time Complexity: $O(v^*v)$.

Source

<https://www.geeksforgeeks.org/construct-graph-given-degrees-vertices/>

Chapter 112

Construct binary palindrome by repeated appending and trimming

Construct binary palindrome by repeated appending and trimming - GeeksforGeeks

Given n and k, Construct a palindrome of size n using a binary number of size k repeating itself to wrap into the palindrome. **The palindrome must always begin with 1 and contains maximum number of zeros.**

Examples :

Input : n = 5, k = 3
Output : 11011
Explanation : the 3 sized substring is 110 combined twice and trimming the extra 0 in the end to give 11011.

Input : n = 2, k = 8
Output : 11
Explanation : the 8 sized substring is 11..... wrapped to two places to give 11.

The **naive approach** would be to try every palindrome of size k starting with 1 such that a palindrome of size n is formed. This approach has an exponential complexity.

A **better way** to do this is to initialize the k sized binary number with the index and connect the palindrome in the way it should be. Like last character of palindrome should match to first, find which indexes will be present at those locations and link them. Set every character linked with 0th index to 1 and the string is ready. This approach will have a linear complexity.

In this approach, first lay the index of the k sized binary to hold into an array, for example if $n = 7$, $k = 3$ arr becomes $[0, 1, 2, 0, 1, 2, 0]$. Following that in the connectchars graph, connect the indices of the k sized binary which should be same by going through the property of palindrome which is k th and $(n - k - 1)$ th variable should be same, such that 0 is linked to 1 (and vice versa), 1 is linked to 2 (and vice versa) and so on. After that, check what is linked with 0 in connectchars array and make all of the associated indices one (because the first number should be non-zero) by using dfs approach. In the dfs, pass 0, the final answer string and the graph. Begin by making the parent 1 and checking if its children are zero, if they are make them and their children 1. This makes only the required indices of the k sized string one, others are left zero. Finally, the answer contains the 0 to $k - 1$ indexes and corresponding to arr the digits are printed.

```
// CPP code to form binary palindrome
#include <iostream>
#include <vector>
using namespace std;

// function to apply DFS
void dfs(int parent, int ans[], vector<int> connectchars[])
{
    // set the parent marked
    ans[parent] = 1;

    // if the node has not been visited set it and
    // its children marked
    for (int i = 0; i < connectchars[parent].size(); i++) {
        if (!ans[connectchars[parent][i]])
            dfs(connectchars[parent][i], ans, connectchars);
    }
}

void printBinaryPalindrome(int n, int k)
{
    int arr[n], ans[n] = { 0 };

    // link which digits must be equal
    vector<int> connectchars[k];

    for (int i = 0; i < n; i++)
        arr[i] = i % k;

    // connect the two indices
    for (int i = 0; i < n / 2; i++) {
        connectchars[arr[i]].push_back(arr[n - i - 1]);
        connectchars[arr[n - i - 1]].push_back(arr[i]);
    }

    // set everything connected to
}
```

```
// first character as 1
dfs(0, ans, connectchars);

for (int i = 0; i < n; i++)
    cout << ans[arr[i]];

// driver code
int main()
{
    int n = 10, k = 4;
    printBinaryPalindrome(n, k);
    return 0;
}
```

Output:

1100110011

Time Complexity : $O(n)$

Source

<https://www.geeksforgeeks.org/construct-binary-palindrome-by-repeated-appending-and-trimming/>

Chapter 113

Count all possible paths between two vertices

Count all possible paths between two vertices - GeeksforGeeks

Count the total number of ways or paths that exist between two vertices in a directed graph. These paths doesn't contain a cycle, the simple enough reason is that a cycle contain infinite number of paths and hence they create problem.

Examples:

Input : Count paths between A and E

Output : Total paths between A and E are 4

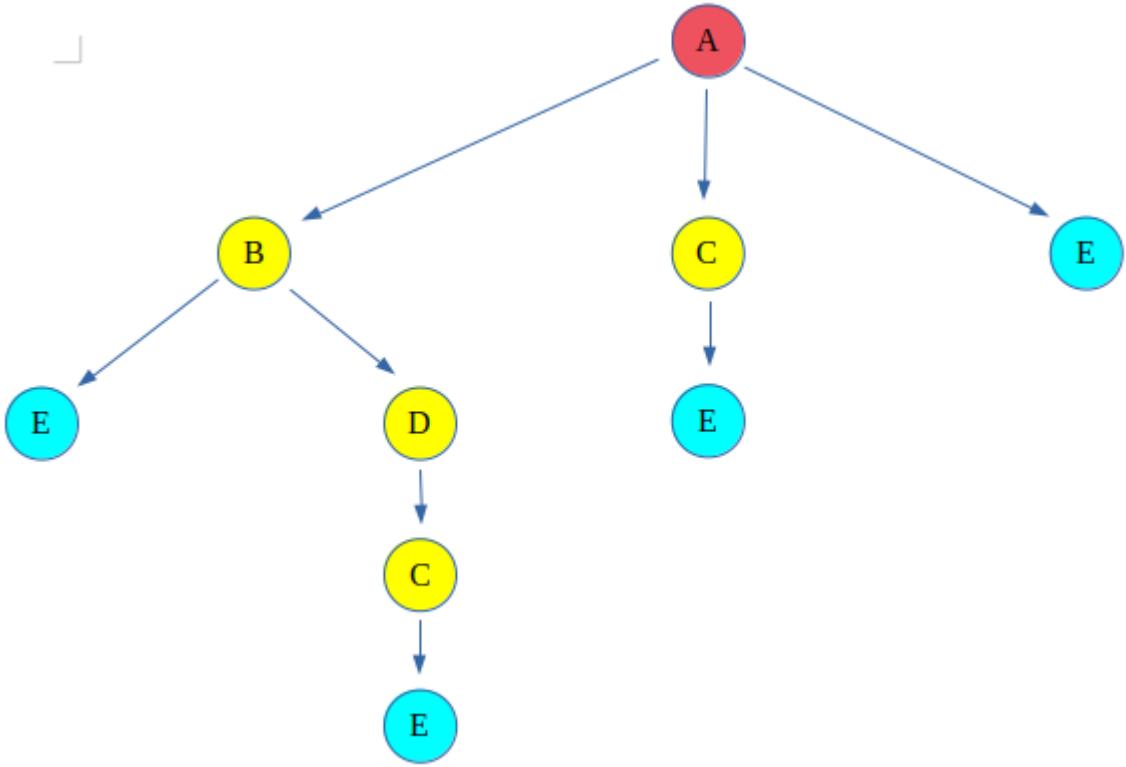
Explanation: The 4 paths between A and E are:

A → E
A → B → E
A → C → E
A → B → D → C → E

The problem can be solved using [backtracking](#), that is we take a path and start walking it, if it leads us to the destination vertex then we count the path and backtrack to take another path. If the path doesn't lead us to the destination vertex, we discard the path.

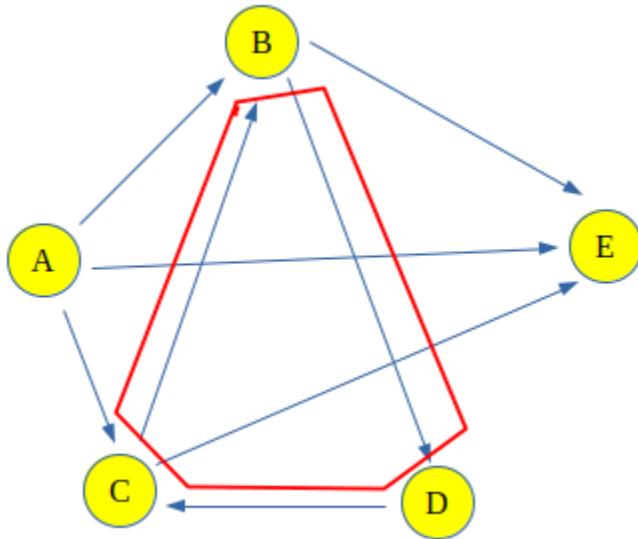
Backtracking for above graph can be shown like this:

The red color vertex is the source vertex and the light-blue color vertex is destination, rest are either intermediate or discarded paths.



This gives us four paths between **source(A)** and **destination(E)** vertex.

Problem Associated with this: Now if we add just one more edge between C and B, it would make a cycle (**B -> D -> C -> B**). And hence we could loop the cycles any number of times to get a new path, and there would be infinitely many paths because of the cycle.



C++

```

// C++ program to count all paths from a
// source to a destination.
#include<bits/stdc++.h>

using namespace std;

// A directed graph using adjacency list
// representation
class Graph
{

    // No. of vertices in graph
    int V;
    list<int> *adj;

    // A recursive function
    // used by countPaths()
    void countPathsUtil(int, int, bool [], int &);

public:

    // Constructor
    Graph(int V);
    void addEdge(int u, int v);
    int countPaths(int s, int d);
}

```

```
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int u, int v)
{

    // Add v to u's list.
    adj[u].push_back(v);
}

// Returns count of paths from 's' to 'd'
int Graph::countPaths(int s, int d)
{

    // Mark all the vertices
    // as not visited
    bool *visited = new bool[V];
    memset(visited, false, sizeof(visited));

    // Call the recursive helper
    // function to print all paths
    int pathCount = 0;
    countPathsUtil(s, d, visited, pathCount);
    return pathCount;
}

// A recursive function to print all paths
// from 'u' to 'd'. visited[] keeps track of
// vertices in current path. path[] stores
// actual vertices and path_index is
// current index in path[]
void Graph::countPathsUtil(int u, int d, bool visited[],
                           int &pathCount)
{
    visited[u] = true;

    // If current vertex is same as destination,
    // then increment count
    if (u == d)
        pathCount++;

    // If current vertex is not destination
    else
```

```
{  
    // Recur for all the vertices adjacent to  
    // current vertex  
    list<int>::iterator i;  
    for (i = adj[u].begin(); i != adj[u].end(); ++i)  
        if (!visited[*i])  
            countPathsUtil(*i, d, visited,  
                           pathCount);  
}  
  
visited[u] = false;  
}  
  
// Driver Code  
int main()  
{  
  
    // Create a graph given in the above diagram  
    Graph g(4);  
    g.addEdge(0, 1);  
    g.addEdge(0, 2);  
    g.addEdge(0, 3);  
    g.addEdge(2, 0);  
    g.addEdge(2, 1);  
    g.addEdge(1, 3);  
  
    int s = 2, d = 3;  
    cout << g.countPaths(s, d);  
  
    return 0;  
}
```

Java

```
// Java program to count all paths from a source  
// to a destination.  
import java.util.Arrays;  
import java.util.Iterator;  
import java.util.LinkedList;  
  
// This class represents a directed graph using  
// adjacency list representation  
  
class Graph {  
  
    // No. of vertices  
    private int V;
```

```
// Array of lists for
// Adjacency List
// Representation
private LinkedList<Integer> adj[];

@SuppressWarnings("unchecked")
Graph(int v)
{
    V = v;
    adj = new LinkedList[v];
    for (int i = 0; i < v; ++i)
        adj[i] = new LinkedList<>();
}

// Method to add an edge into the graph
void addEdge(int v, int w)
{
    // Add w to v's list.
    adj[v].add(w);
}

// A recursive method to count
// all paths from 'u' to 'd'.
int countPathsUtil(int u, int d,
                   boolean visited[], int pathCount)
{
    // Mark the current node as
    // visited and print it
    visited[u] = true;

    // If current vertex is same as
    // destination, then increment count
    if (u == d)
    {
        pathCount++;
    }

    // Recur for all the vertices
    // adjacent to this vertex
    else
    {
        Iterator<Integer> i = adj[u].listIterator();
        while (i.hasNext())
        {
```

```
        int n = i.next();
        if (!visited[n])
        {
            pathCount = countPathsUtil(n, d,
                                         visited,
                                         pathCount);
        }
    }

    visited[u] = false;
    return pathCount;
}

// Returns count of
// paths from 's' to 'd'
int countPaths(int s, int d)
{
    // Mark all the vertices
    // as not visited
    boolean visited[] = new boolean[V];
    Arrays.fill(visited, false);

    // Call the recursive method
    // to count all paths
    int pathCount = 0;
    pathCount = countPathsUtil(s, d,
                               visited,
                               pathCount);
    return pathCount;
}

// Driver Code
public static void main(String args[])
{
    Graph g = new Graph(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(0, 3);
    g.addEdge(2, 0);
    g.addEdge(2, 1);
    g.addEdge(1, 3);

    int s = 2, d = 3;
    System.out.println(g.countPaths(s, d));
}
```

```
// This code is contributed by shubhamjd.
```

Output:

3

Improved By : [shubhamjd](#), [IshaanKanwar](#)

Source

<https://www.geeksforgeeks.org/count-possible-paths-two-vertices/>

Chapter 114

Count nodes within K-distance from all nodes in a set

Count nodes within K-distance from all nodes in a set - GeeksforGeeks

Given an undirected tree with some marked nodes and a positive number K. We need to print the count of all such nodes which have distance from all marked nodes less than K that means every node whose distance from all marked nodes is less than K, should be counted in the result.

Examples:

```
In above tree we can see that node with index
0, 2, 3, 5, 6, 7 have distances less than 3
from all the marked nodes.
so answer will be 6
```

We can solve this problem using breadth first search. Main thing to observe in this problem is that if we find two marked nodes which are at largest distance from each other considering all pairs of marked nodes then if a node is at a distance less than K from both of these two nodes then it will be at a distance less than K from all the marked nodes because these two nodes represents the extreme limit of all marked nodes, if a node lies in this limit then it will be at a distance less than K from all marked nodes otherwise not.

As in above example, node-1 and node-4 are most distant marked node so nodes which are at distance less than 3 from these two nodes will also be at distance less than 3 from node 2 also. Now first distant marked node we can get by doing a bfs from any random node, second distant marked node we can get by doing another bfs from marked node we just found from the first bfs and in this bfs we can also found distance of all nodes from first distant marked node and to find distance of all nodes from second distant marked node we will do one more bfs, so after doing these three bfs we can get distance of all nodes from

two extreme marked nodes which can be compared with K to know which nodes fall in K-distance range from all marked nodes.

```
// C++ program to count nodes inside K distance
// range from marked nodes
#include <bits/stdc++.h>
using namespace std;

// Utility bfs method to fill distance vector and returns
// most distant marked node from node u
int bfsWithDistance(vector<int> g[], bool mark[], int u,
                     vector<int>& dis)
{
    int lastMarked;
    queue<int> q;

    // push node u in queue and initialize its distance as 0
    q.push(u);
    dis[u] = 0;

    // loop until all nodes are processed
    while (!q.empty())
    {
        u = q.front();      q.pop();
        // if node is marked, update lastMarked variable
        if (mark[u])
            lastMarked = u;

        // loop over all neighbors of u and update their
        // distance before pushing in queue
        for (int i = 0; i < g[u].size(); i++)
        {
            int v = g[u][i];

            // if not given value already
            if (dis[v] == -1)
            {
                dis[v] = dis[u] + 1;
                q.push(v);
            }
        }
    }
    // return last updated marked value
    return lastMarked;
}

// method returns count of nodes which are in K-distance
// range from marked nodes
```

```
int nodesKDistanceFromMarked(int edges[][] , int V,
                             int marked[], int N, int K)
{
    //  vertices in a tree are one more than number of edges
    V = V + 1;
    vector<int> g[V];

    //  fill vector for graph
    int u, v;
    for (int i = 0; i < (V - 1); i++)
    {
        u = edges[i][0];
        v = edges[i][1];

        g[u].push_back(v);
        g[v].push_back(u);
    }

    //  fill boolean array mark from marked array
    bool mark[V] = {false};
    for (int i = 0; i < N; i++)
        mark[marked[i]] = true;

    //  vectors to store distances
    vector<int> tmp(V, -1), dl(V, -1), dr(V, -1);

    //  first bfs(from any random node) to get one
    //  distant marked node
    u = bfsWithDistance(g, mark, 0, tmp);

    /* second bfs to get other distant marked node
       and also dl is filled with distances from first
       chosen marked node */
    u = bfsWithDistance(g, mark, u, dl);

    //  third bfs to fill dr by distances from second
    //  chosen marked node
    bfsWithDistance(g, mark, u, dr);

    int res = 0;
    //  loop over all nodes
    for (int i = 0; i < V; i++)
    {
        // increase res by 1, if current node has distance
        // less than K from both extreme nodes
        if (dl[i] <= K && dr[i] <= K)
            res++;
    }
}
```

```
    return res;
}

// Driver code to test above methods
int main()
{
    int edges[] [2] =
    {
        {1, 0}, {0, 3}, {0, 8}, {2, 3},
        {3, 5}, {3, 6}, {3, 7}, {4, 5},
        {5, 9}
    };
    int V = sizeof(edges) / sizeof(edges[0]);

    int marked[] = {1, 2, 4};
    int N = sizeof(marked) / sizeof(marked[0]);

    int K = 3;
    cout << nodesKDistanceFromMarked(edges, V, marked, N, K);
    return 0;
}
```

Output:

6

Source

<https://www.geeksforgeeks.org/count-nodes-within-k-distance-from-all-nodes-in-a-set/>

Chapter 115

Count number of edges in an undirected graph

Count number of edges in an undirected graph - GeeksforGeeks

Given an adjacency list representation undirected graph. Write a function to count the number of edges in the undirected graph.

Expected time complexity : $O(V)$

Examples:

```
Input : Adjacency list representation of
        below graph.
Output : 9
```

Idea is based on Handshaking Lemma. [Handshaking lemma](#) is about undirected graph. In every finite undirected graph number of vertices with odd degree is always even. The handshaking lemma is a consequence of the degree sum formula (also sometimes called the handshaking lemma)

So we traverse all vertices, compute sum of sizes of their adjacency lists, and finally returns sum/2. Below c++ implementation of above idea

```
// C++ program to count number of edge in
// undirected graph
#include<bits/stdc++.h>
using namespace std;
```

```
// Adjacency list representation of graph
class Graph
{
    int V ;
    list < int > *adj;
public :
    Graph( int V )
    {
        this->V = V ;
        adj = new list<int>[V];
    }
    void addEdge ( int u, int v ) ;
    int countEdges () ;
};

// add edge to graph
void Graph :: addEdge ( int u, int v )
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}

// Returns count of edge in undirected graph
int Graph :: countEdges()
{
    int sum = 0;

    //traverse all vertex
    for (int i = 0 ; i < V ; i++)

        // add all edge that are linked to the
        // current vertex
        sum += adj[i].size();

    // The count of edge is always even because in
    // undirected graph every edge is connected
    // twice between two vertices
    return sum/2;
}

// driver program to check above function
int main()
{
    int V = 9 ;
    Graph g(V);

    // making above uhown graph
```

```
g.addEdge(0, 1 );
g.addEdge(0, 7 );
g.addEdge(1, 2 );
g.addEdge(1, 7 );
g.addEdge(2, 3 );
g.addEdge(2, 8 );
g.addEdge(2, 5 );
g.addEdge(3, 4 );
g.addEdge(3, 5 );
g.addEdge(4, 5 );
g.addEdge(5, 6 );
g.addEdge(6, 7 );
g.addEdge(6, 8 );
g.addEdge(7, 8 );

cout << g.countEdges() << endl;

return 0;
}
```

Output:

14

Time Complexity : $O(V)$

Source

<https://www.geeksforgeeks.org/count-number-edges-undirected-graph/>

Chapter 116

Count number of trees in a forest

Count number of trees in a forest - GeeksforGeeks

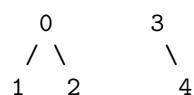
Given n nodes of a forest (collection of trees), find the number of trees in the forest.

Examples :

Input : edges[] = {0, 1}, {0, 2}, {3, 4}

Output : 2

Explanation : There are 2 trees



Approach :

1. Apply DFS on every node.
2. Increment count by one if every connected node is visited from one source.
3. Again perform DFS traversal if some nodes yet not visited.
4. Count will give the number of trees in forest.

C++

```
// CPP program to count number of trees in
// a forest.
#include<bits/stdc++.h>
using namespace std;

// A utility function to add an edge in an
// undirected graph.
void addEdge(vector<int> adj[], int u, int v)
```

```
{  
    adj[u].push_back(v);  
    adj[v].push_back(u);  
}  
  
// A utility function to do DFS of graph  
// recursively from a given vertex u.  
void DFSUtil(int u, vector<int> adj[],  
             vector<bool> &visited)  
{  
    visited[u] = true;  
    for (int i=0; i<adj[u].size(); i++)  
        if (visited[adj[u][i]] == false)  
            DFSUtil(adj[u][i], adj, visited);  
}  
  
// Returns count of tree in the forest  
// given as adjacency list.  
int countTrees(vector<int> adj[], int V)  
{  
    vector<bool> visited(V, false);  
    int res = 0;  
    for (int u=0; u<V; u++)  
    {  
        if (visited[u] == false)  
        {  
            DFSUtil(u, adj, visited);  
            res++;  
        }  
    }  
    return res;  
}  
  
// Driver code  
int main()  
{  
    int V = 5;  
    vector<int> adj[V];  
    addEdge(adj, 0, 1);  
    addEdge(adj, 0, 2);  
    addEdge(adj, 3, 4);  
    cout << countTrees(adj, V);  
    return 0;  
}
```

Output:

2

Time Complexity : $O(V + E)$

Improved By : [Rajesh Raj 2](#)

Source

<https://www.geeksforgeeks.org/count-number-trees-forest/>

Chapter 117

Count single node isolated sub-graphs in a disconnected graph

Count single node isolated sub-graphs in a disconnected graph - GeeksforGeeks

A disconnected Graph with N vertices and K edges is given. The task is to find the count of singleton sub-graphs. A singleton graph is one with only single vertex.

Examples:

Input :

Vertices : 6

Edges : 1 2
 1 3
 5 6

Output : 1

Explanation : The Graph has 3 components : {1-2-3}, {5-6}, {4}

Out of these, the only component forming singleton graph is {4}.

The idea is simple for graph given as adjacency list representation. We traverse the list and find the indices(representing a node) with no elements in list, i.e. no connected components.

Below is the C++ representation :

```
// CPP code to count the singleton sub-graphs
// in a disconnected graph
#include <bits/stdc++.h>
using namespace std;

// Function to compute the count
```

```
int compute(vector<int> graph[], int N)
{
    // Storing intermediate result
    int count = 0;

    // Traversing the Nodes
    for (int i = 1; i <= N; i++)

        // Singleton component
        if (graph[i].size() == 0)
            count++;

    // Returning the result
    return count;
}

// Driver
int main()
{
    // Number of nodes
    int N = 6;

    // Adjacency list for edges 1..6
    vector<int> graph[7];

    // Representing edges
    graph[1].push_back(2);
    graph[2].push_back(1);

    graph[2].push_back(3);
    graph[3].push_back(2);

    graph[5].push_back(6);
    graph[6].push_back(5);

    cout << compute(graph, N);
}
```

Output:

1

Source

<https://www.geeksforgeeks.org/count-single-node-isolated-sub-graphs-disconnected-graph/>

Chapter 118

Count the number of nodes at given level in a tree using BFS.

Count the number of nodes at given level in a tree using BFS. - GeeksforGeeks

Given a tree represented as undirected graph. Count the number of nodes at given level l. It may be assumed that vertex 0 is root of the tree.

Examples:

```
Input : 7
        0 1
        0 2
        1 3
        1 4
        1 5
        2 6
        2
Output : 4
```

```
Input : 6
        0 1
        0 2
        1 3
        2 4
        2 5
        2
Output : 3
```

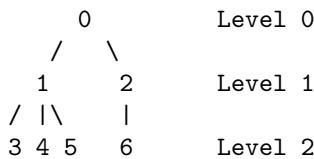
BFS is a traversing algorithm which start traversing from a selected node (source or starting node) and traverse the graph layer wise thus exploring the neighbour nodes (nodes which

are directly connected to source node). Then, move towards the next-level neighbour nodes. As the name BFS suggests, traverse the graph breadth wise as follows:

1. First move horizontally and visit all the nodes of the current layer.
2. Move to the next layer.

In this code, while visiting each node, the level of that node is set with an increment in the level of its parent node i.e., $\text{level}[\text{child}] = \text{level}[\text{parent}] + 1$. This is how the level of each node is determined. The root node lies at level zero in the tree.

Explanation :



Given a tree with 7 nodes and 6 edges in which node 0 lies at 0 level. Level of 1 can be updated as : $\text{level}[1] = \text{level}[0] + 1$ as 0 is the parent node of 1. Similarly, the level of other nodes can be updated by adding 1 to the level of their parent.

$\text{level}[2] = \text{level}[0] + 1$, i.e $\text{level}[2] = 0 + 1 = 1$.

$\text{level}[3] = \text{level}[1] + 1$, i.e $\text{level}[3] = 1 + 1 = 2$.

$\text{level}[4] = \text{level}[1] + 1$, i.e $\text{level}[4] = 1 + 1 = 2$.

$\text{level}[5] = \text{level}[1] + 1$, i.e $\text{level}[5] = 1 + 1 = 2$.

$\text{level}[6] = \text{level}[2] + 1$, i.e $\text{level}[6] = 1 + 1 = 2$.

Then, count of number of nodes which are at level l(i.e, l=2) is 4 (node:- 3, 4, 5, 6)

C++

```
// C++ Program to print
// count of nodes
// at given level.
#include <iostream>
#include <list>

using namespace std;

// This class represents
// a directed graph
// using adjacency
// list representation
class Graph {
    // No. of vertices
    int V;

    // Pointer to an
    // array containing
```

```
// adjacency lists
list<int>* adj;

public:
    // Constructor
    Graph(int V);

    // function to add
    // an edge to graph
    void addEdge(int v, int w);

    // Returns count of nodes at
    // level l from given source.
    int BFS(int s, int l);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    // Add w to v's list.
    adj[v].push_back(w);

    // Add v to w's list.
    adj[w].push_back(v);
}

int Graph::BFS(int s, int l)
{
    // Mark all the vertices
    // as not visited
    bool* visited = new bool[V];
    int level[V];

    for (int i = 0; i < V; i++) {
        visited[i] = false;
        level[i] = 0;
    }

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as
    // visited and enqueue it
```

```
visited[s] = true;
queue.push_back(s);
level[s] = 0;

while (!queue.empty()) {

    // Dequeue a vertex from
    // queue and print it
    s = queue.front();
    queue.pop_front();

    // Get all adjacent vertices
    // of the dequeued vertex s.
    // If a adjacent has not been
    // visited, then mark it
    // visited and enqueue it
    for (auto i = adj[s].begin();
         i != adj[s].end(); ++i) {
        if (!visited[*i]) {

            // Setting the level
            // of each node with
            // an increment in the
            // level of parent node
            level[*i] = level[s] + 1;
            visited[*i] = true;
            queue.push_back(*i);
        }
    }
}

int count = 0;
for (int i = 0; i < V; i++)
    if (level[i] == 1)
        count++;
return count;
}

// Driver program to test
// methods of graph class
int main()
{
    // Create a graph given
    // in the above diagram
    Graph g(6);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
```

```
g.addEdge(2, 4);
g.addEdge(2, 5);

int level = 2;

cout << g.BFS(0, level);

return 0;
}
```

Output:

3

Source

<https://www.geeksforgeeks.org/count-number-nodes-given-level-using-bfs/>

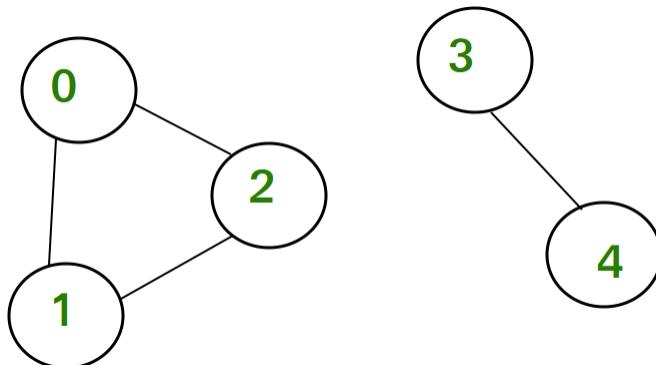
Chapter 119

Count the number of non-reachable nodes

Count the number of non-reachable nodes - GeeksforGeeks

Given an undirected graph and a set of vertices, we have to count the number of non reachable nodes from the given head node using depth first search.

Consider below undirected graph with two disconnected components:



In this graph if we consider 0 as head node, then the node 0, 1 and 2 are reachable. We mark all the reachable nodes as visited. All those nodes which are not mark as visited i.e, node 3 and 4 are non reachable nodes. Hence their count is 2.

Examples:

```
Input : 5
        0 1
        0 2
        1 2
```

```
3 4  
Output : 2
```

We can either use BFS or DFS for this purpose. In below implementation DFS is used. We do DFS from given source. Since the given graph is undirected, all the vertices that belong to the disconnected component are non-reachable nodes. We use the visit array for this purpose, the array which is used to keep track of non-visited vertices in DFS. In DFS, if we start from head node it will mark all the nodes connected to the head node as visited. Then after traversing the graph we count the number of nodes that are not mark as visited from the head node.

```
// C++ program to count non-reachable nodes  
// from a given source using DFS.  
#include <iostream>  
#include <list>  
using namespace std;  
  
// Graph class represents a directed graph  
// using adjacency list representation  
class Graph {  
    int V; // No. of vertices  
  
    // Pointer to an array containing  
    // adjacency lists  
    list<int>*> adj;  
  
    // A recursive function used by DFS  
    void DFSUtil(int v, bool visited[]);  
  
public:  
    Graph(int V); // Constructor  
  
    // function to add an edge to graph  
    void addEdge(int v, int w);  
  
    // DFS traversal of the vertices  
    // reachable from v  
    int countNotReach(int v);  
};  
  
Graph::Graph(int V)  
{  
    this->V = V;  
    adj = new list<int>[V];  
}  
  
void Graph::addEdge(int v, int w)  
{
```

```
adj[v].push_back(w); // Add w to v's list.
adj[w].push_back(v); // Add v to w's list.
}

void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and
    // print it
    visited[v] = true;

    // Recur for all the vertices adjacent
    // to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

// Returns count of not reachable nodes from
// vertex v.
// It uses recursive DFSUtil()
int Graph::countNotReach(int v)
{
    // Mark all the vertices as not visited
    bool* visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function
    // to print DFS traversal
    DFSUtil(v, visited);

    // Return count of not visited nodes
    int count = 0;
    for (int i = 0; i < V; i++) {
        if (visited[i] == false)
            count++;
    }
    return count;
}

int main()
{
    // Create a graph given in the above diagram
    Graph g(8);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
```

```
g.addEdge(3, 4);
g.addEdge(4, 5);
g.addEdge(6, 7);

cout << g.countNotReach(2);

return 0;
}
```

Output:

5

Source

<https://www.geeksforgeeks.org/count-number-non-reachable-nodes/>

Chapter 120

Cycles of length n in an undirected and connected graph

Cycles of length n in an undirected and connected graph - GeeksforGeeks

Given an undirected and connected graph and a number n, count total number of cycles of length n in the graph. A cycle of length n simply means that the cycle contains n vertices and n edges. And we have to count all such cycles that exist.

Example :

Input : n = 4

Output : Total cycles = 3

Explanation : Following 3 unique cycles

0 → 1 → 2 → 3 → 0

0 → 1 → 4 → 3 → 0

1 → 2 → 3 → 4 → 1

Note* : There are more cycles but
these 3 are unique as 0 → 3 → 2 → 1
→ 0 and 0 → 1 → 2 → 3 → 0 are
same cycles and hence will be counted as 1.

To solve this Problem, [DFS\(Depth First Search\)](#) can be effectively used. Using DFS we find every possible path of length (n-1) for a particular source (or starting point). Then we check if this path ends with the vertex it started with, if yes then we count this as the cycle of length n. Notice that we looked for path of length (n-1) because the nth edge will be the closing edge of cycle.

Every possible path of length (n-1) can be searched using only $V - (n - 1)$ vertices (where V is the total number of vertices).

For above example, all the cycles of length 4 can be searched using only $5 - (4 - 1) = 2$ vertices. The reason behind this is quite simple, because we search for all possible path of length

$(n-1) = 3$ using these 2 vertices which include the remaining 3 vertices. So, these 2 vertices cover the cycles of remaining 3 vertices as well, and using only 3 vertices we can't form a cycle of length 4 anyways.

One more thing to notice is that, every vertex finds 2 duplicate cycles for every cycle that it forms. For above example 0th vertex finds two duplicate cycle namely **0 -> 3 -> 2 -> 1 -> 0** and **0 -> 1 -> 2 -> 3 -> 0**. Hence the total count must be divided by 2 because every cycle is counted twice.

C++

```
// CPP Program to count cycles of length n
// in a given graph.
#include <bits/stdc++.h>
using namespace std;

// Number of vertices
const int V = 5;

void DFS(bool graph[][][V], bool marked[], int n,
         int vert, int start, int &count)
{
    // mark the vertex vert as visited
    marked[vert] = true;

    // if the path of length (n-1) is found
    if (n == 0) {

        // mark vert as un-visited to make
        // it usable again.
        marked[vert] = false;

        // Check if vertex vert can end with
        // vertex start
        if (graph[vert][start])
        {
            count++;
            return;
        } else
            return;
    }

    // For searching every possible path of
    // length (n-1)
    for (int i = 0; i < V; i++)
        if (!marked[i] && graph[vert][i])

            // DFS for searching path by decreasing
            // length by 1
```

```
DFS(graph, marked, n-1, i, start, count);

// marking vert as unvisited to make it
// usable again.
marked[vert] = false;
}

// Counts cycles of length N in an undirected
// and connected graph.
int countCycles(bool graph[][][V], int n)
{
    // all vertex are marked un-visited intially.
    bool marked[V];
    memset(marked, 0, sizeof(marked));

    // Searching for cycle by using v-n+1 vertices
    int count = 0;
    for (int i = 0; i < V - (n - 1); i++) {
        DFS(graph, marked, n-1, i, i, count);

        // ith vertex is marked as visited and
        // will not be visited again.
        marked[i] = true;
    }

    return count/2;
}

int main()
{
    bool graph[][][V] = {{0, 1, 0, 1, 0},
                        {1, 0, 1, 0, 1},
                        {0, 1, 0, 1, 0},
                        {1, 0, 1, 0, 1},
                        {0, 1, 0, 1, 0}};
    int n = 4;
    cout << "Total cycles of length " << n << " are "
         << countCycles(graph, n);
    return 0;
}
```

Java

```
// Java program to calculate cycles of
// length n in a given graph
public class Main {

    // Number of vertices
```

```
public static final int V = 5;
static int count = 0;

static void DFS(int graph[][] , boolean marked[],
                int n, int vert, int start) {

    // mark the vertex vert as visited
    marked[vert] = true;

    // if the path of length (n-1) is found
    if (n == 0) {

        // mark vert as un-visited to
        // make it usable again
        marked[vert] = false;

        // Check if vertex vert end
        // with vertex start
        if (graph[vert][start] == 1) {
            count++;
            return;
        } else
            return;
    }

    // For searching every possible
    // path of length (n-1)
    for (int i = 0; i < V; i++)
        if (!marked[i] && graph[vert][i] == 1)

            // DFS for searching path by
            // decreasing length by 1
            DFS(graph, marked, n-1, i, start);

    // marking vert as unvisited to make it
    // usable again
    marked[vert] = false;
}

// Count cycles of length N in an
// undirected and connected graph.
static int countCycles(int graph[][], int n) {

    // all vertex are marked un-visited
    // initially.
    boolean marked[] = new boolean[V];

    // Searching for cycle by using
```

```
// v-n+1 vertices
for (int i = 0; i < V - (n - 1); i++) {
    DFS(graph, marked, n-1, i, i);

    // ith vertex is marked as visited
    // and will not be visited again
    marked[i] = true;
}

return count / 2;
}

// driver code
public static void main(String[] args) {
    int graph[][] = {{0, 1, 0, 1, 0},
                     {1, 0, 1, 0, 1},
                     {0, 1, 0, 1, 0},
                     {1, 0, 1, 0, 1},
                     {0, 1, 0, 1, 0}};

    int n = 4;

    System.out.println("Total cycles of length "+
                       n + " are "+
                       countCycles(graph, n));
}
}

// This code is contributed by nuclode
```

Python3

```
# Python Program to count
# cycles of length n
# in a given graph.

# Number of vertices
V = 5

def DFS(graph, marked, n, vert, start, count):

    # mark the vertex vert as visited
    marked[vert] = True

    # if the path of length (n-1) is found
    if n == 0:

        # mark vert as un-visited to make
```

```

# it usable again.
marked[vert] = False

# Check if vertex vert can end with
# vertex start
if graph[vert][start] == 1:
    count = count + 1
    return count
else:
    return count

# For searching every possible path of
# length (n-1)
for i in range(V):
    if marked[i] == False and graph[vert][i] == 1:

        # DFS for searching path by decreasing
        # length by 1
        count = DFS(graph, marked, n-1, i, start, count)

    # marking vert as unvisited to make it
    # usable again.
    marked[vert] = False
    return count

# Counts cycles of length
# N in an undirected
# and connected graph.
def countCycles( graph, n):

    # all vertex are marked un-visited intially.
    marked = [False] * V

    # Searching for cycle by using v-n+1 vertices
    count = 0
    for i in range(V-(n-1)):
        count = DFS(graph, marked, n-1, i, i, count)

    # ith vertex is marked as visited and
    # will not be visited again.
    marked[i] = True

    return int(count/2)

# main :
graph = [[0, 1, 0, 1, 0],
          [1, 0, 1, 0, 1],
          [0, 1, 0, 1, 0],

```

```
[1, 0, 1, 0, 1],  
[0, 1, 0, 1, 0]]  
  
n = 4  
print("Total cycles of length ",n," are ",countCycles(graph, n))  
  
# this code is contributed by Shivani Ghugtyal
```

Output:

```
Total cycles of length 4 are 3
```

Source

<https://www.geeksforgeeks.org/cycles-of-length-n-in-an-undirected-and-connected-graph/>

Chapter 121

Degree Centrality (Centrality Measure)

Degree Centrality (Centrality Measure) - GeeksforGeeks

Degree

In graph theory, the degree (or valency) of a vertex of a graph is the number of edges incident to the vertex, with loops counted twice.[1] The degree of a vertex v is denoted $\deg(v)$ or $\text{度数}(v)$. The maximum degree of a graph G , denoted by $\Delta(G)$, and the minimum degree of a graph, denoted by $\delta(G)$, are the maximum and minimum degree of its vertices. In the graph on the right, the maximum degree is 5 and the minimum degree is 0. In a regular graph, all degrees are the same, and so we can speak of the degree of the graph.

Degree Centrality

Historically first and conceptually simplest is degree centrality, which is defined as the number of links incident upon a node (i.e., the number of ties that a node has). The degree can be interpreted in terms of the immediate risk of a node for catching whatever is flowing through the network (such as a virus, or some information). In the case of a directed network (where ties have direction), we usually define two separate measures of degree centrality, namely indegree and outdegree. Accordingly, indegree is a count of the number of ties directed to the node and outdegree is the number of ties that the node directs to others. When ties are associated to some positive aspects such as friendship or collaboration, indegree is often interpreted as a form of popularity, and outdegree as gregariousness.

The degree centrality of a vertex v , for a given graph $G = (V, E)$ with $|V|$ vertices and $|E|$ edges, is defined as

$$C_D(v) = \deg(v)$$

Calculating degree centrality for all the nodes in a graph takes $\mathcal{O}(V^2)$ in a dense adjacency matrix representation of the graph, and for edges takes $\mathcal{O}(E)$ in a sparse matrix representation.

The definition of centrality on the node level can be extended to the whole graph, in which case we are speaking of graph centralization. Let v^* be the node with highest degree centrality in G . Let $X := \{v^*, v_1, v_2, \dots, v_n\}$ be the $|X|$ node connected graph that maximizes the following quantity (with y^* being the node with highest degree centrality in X):

$$\Sigma = \sum_{i=1}^n \frac{1}{d(v_i)} (y^* - d_G(v_i))$$

Correspondingly, the degree centralization of the graph G is as follows:

$$C_D(G) = \frac{\sum_{i=1}^n \frac{1}{d(v_i)} (y^* - d_G(v_i))}{n}$$

The value of Σ is maximized when the graph X contains one central node to which all other nodes are connected (a star graph), and in this case

$$\Sigma = \sum_{i=1}^n \frac{1}{d(v_i)} (y^* - d_G(v_i)) = \sum_{i=1}^n \frac{1}{d(v_i)} (y^* - 1) = y^* - \frac{1}{n} \sum_{i=1}^n \frac{1}{d(v_i)}$$

Following is the code for the calculation of the degree centrality of the graph and its various nodes.

```
import networkx as nx

def degree_centrality(G, nodes):
    """Compute the degree centrality for nodes in a bipartite network.

    The degree centrality for a node `v` is the fraction of nodes
    connected to it.

    Parameters
    -----
    G : graph
        A bipartite network

    nodes : list or container
        Container with all nodes in one bipartite node set.
```

Returns

centrality : dictionary

Dictionary keyed by node with bipartite degree centrality as the value.

Notes

The nodes input parameter must contain all nodes in one bipartite node set, but the dictionary returned contains all nodes from both bipartite node sets.

For unipartite networks, the degree centrality values are normalized by dividing by the maximum possible degree (which is $n-1$ where n is the number of nodes in G).

In the bipartite case, the maximum possible degree of a node in a bipartite node set is the number of nodes in the opposite node set [1]. The degree centrality for a node v in the bipartite sets U with n nodes and V with m nodes is

.. math::

$$d_v = \frac{\deg(v)}{m}, \quad v \in U,$$

$$d_v = \frac{\deg(v)}{n}, \quad v \in V,$$

where $\deg(v)$ is the degree of node v .

....

```
top = set(nodes)
bottom = set(G) - top
s = 1.0/len(bottom)
centrality = dict((n,d*s) for n,d in G.degree_iter(top))
s = 1.0/len(top)
centrality.update(dict((n,d*s) for n,d in G.degree_iter(bottom)))
return centrality
```

The above function is invoked using the networkx library and once the library is installed, you can eventually use it and the following code has to be written in python for the implementation of the Degree centrality of a node.

```
import networkx as nx
G=nx.erdos_renyi_graph(100,0.5)
d=nx.degree_centrality(G)
print(d)
```

The result is as follows:

```
{0: 0.5252525252525253, 1: 0.4444444444444445, 2: 0.5454545454545455, 3: 0.36363636363636365, 4: 0.42424242424242425, 5: 0.494949494949495, 6: 0.5454545454545455, 7: 0.494949494949495, 8: 0.5555555555555556, 9: 0.51515151515152, 10: 0.5454545454545455, 11: 0.5151515151515152, 12: 0.494949494949495, 13: 0.4444444444444445, 14: 0.494949494949495, 15: 0.4141414141414142, 16: 0.43434343434343436, 17: 0.5555555555555556, 18: 0.494949494949495, 19: 0.5151515151515152, 20: 0.42424242424242425, 21: 0.494949494949495, 22: 0.5555555555555556, 23: 0.5151515151515152, 24: 0.4646464646464647, 25: 0.4747474747474748, 26: 0.4747474747474748, 27: 0.494949494949495, 28: 0.5656565656565657, 29: 0.5353535353535354, 30: 0.4747474747474748, 31: 0.494949494949495, 32: 0.43434343434343436, 33: 0.4444444444444445, 34: 0.51515151515152, 35: 0.48484848484848486, 36: 0.43434343434343436, 37: 0.4040404040404041, 38: 0.5656565656565657, 39: 0.5656565656565657, 40: 0.494949494949495, 41: 0.52525252525253, 42: 0.4545454545454546, 43: 0.42424242424242425, 44: 0.494949494949495, 45: 0.595959595959596, 46: 0.5454545454545455, 47: 0.5050505050505051, 48: 0.4646464646464647, 49: 0.48484848484848486, 50: 0.5353535353535354, 51: 0.5454545454545455, 52: 0.5252525252525253, 53: 0.5252525252525253, 54: 0.5353535353535354, 55: 0.6464646464646465, 56: 0.4444444444444445, 57: 0.48484848484848486, 58: 0.5353535353535354, 59: 0.494949494949495, 60: 0.4646464646464647, 61: 0.5858585858585859, 62: 0.494949494949495, 63: 0.48484848484848486, 64: 0.4444444444444445, 65: 0.6262626262626263, 66: 0.51515151515152, 67: 0.4444444444444445, 68: 0.4747474747474748, 69: 0.5454545454545455, 70: 0.48484848484848486, 71: 0.5050505050505051, 72: 0.4646464646464647, 73: 0.4646464646464647, 74: 0.5454545454545455, 75: 0.4444444444444445, 76: 0.42424242424242425, 77: 0.4545454545454546, 78: 0.494949494949495, 79: 0.494949494949495, 80: 0.4444444444444445, 81: 0.48484848484848486, 82: 0.48484848484848486, 83: 0.51515151515152, 84: 0.494949494949495, 85: 0.51515151515152, 86: 0.5252525252525253, 87: 0.4545454545454546, 88: 0.5252525252525253, 89: 0.5353535353535354, 90: 0.5252525252525253, 91: 0.4646464646464647, 92: 0.4646464646464647, 93: 0.5555555555555556, 94: 0.5656565656565657, 95: 0.4646464646464647, 96: 0.494949494949495, 97: 0.494949494949495, 98: 0.5050505050505051, 99: 0.5050505050505051}
```

The above result is a dictionary depicting the value of degree centrality of each node. The above is an extension of my article series on the centrality measures. Keep networking!!!

References

You can read more about the same at

https://en.wikipedia.org/wiki/Centrality#Degree_centrality
<http://networkx.readthedocs.io/en/networkx-1.10/index.html>

Source

<https://www.geeksforgeeks.org/degree-centrality-centrality-measure/>

Chapter 122

Delete Edge to minimize subtree sum difference

Delete Edge to minimize subtree sum difference - GeeksforGeeks

Given an undirected tree whose each **node is associated with a weight**. We need to delete an edge in such a way that difference between sum of weight in one subtree to sum of weight in other subtree is minimized.

Example:

```
In above tree,  
We have 6 choices for edge deletion,  
edge 0-1,    subtree sum difference = 21 - 2 = 19  
edge 0-2,    subtree sum difference = 14 - 9 = 5  
edge 0-3,    subtree sum difference = 15 - 8 = 7  
edge 2-4,    subtree sum difference = 20 - 3 = 17  
edge 2-5,    subtree sum difference = 18 - 5 = 13  
edge 3-6,    subtree sum difference = 21 - 2 = 19
```

We can solve this problem using DFS. One **simple solution** is to delete each edge one by one and check subtree sum difference. Finally choose the minimum of them. This approach takes quadratic amount of time. An **efficient method** can solve this problem in linear time by calculating the sum of both subtrees using total sum of the tree. We can get the sum of other tree by subtracting sum of one subtree from the total sum of tree, in this way subtree sum difference can be calculated at each node in $O(1)$ time. First we calculate the weight of complete tree and then while doing the DFS at each node, we calculate its subtree sum, by using these two values we can calculate subtree sum difference.

In below code, another array `subtree` is used to store sum of subtree rooted at node i in `subtree[i]`. DFS is called with current node index and parent index each time to loop over

children only at each node.

Please see below code for better understanding.

```
// C++ program to minimize subtree sum
// difference by one edge deletion
#include <bits/stdc++.h>
using namespace std;

/* DFS method to traverse through edges,
   calculating subtree sum at each node and
   updating the difference between subtrees */
void dfs(int u, int parent, int totalSum,
         vector<int> edge[], int subtree[], int& res)
{
    int sum = subtree[u];

    /* loop for all neighbors except parent and
       aggregate sum over all subtrees */
    for (int i = 0; i < edge[u].size(); i++)
    {
        int v = edge[u][i];
        if (v != parent)
        {
            dfs(v, u, totalSum, edge, subtree, res);
            sum += subtree[v];
        }
    }

    // store sum in current node's subtree index
    subtree[u] = sum;

    /* at one side subtree sum is 'sum' and other side
       subtree sum is 'totalSum - sum' so their difference
       will be totalSum - 2*sum, by which we'll update
       res */
    if (u != 0 && abs(totalSum - 2*sum) < res)
        res = abs(totalSum - 2*sum);
}

// Method returns minimum subtree sum difference
int getMinSubtreeSumDifference(int vertex[],
                               int edges[][][2], int N)
{
    int totalSum = 0;
    int subtree[N];

    // Calculating total sum of tree and initializing
    // subtree sum's by vertex values
```

```
for (int i = 0; i < N; i++)
{
    subtree[i] = vertex[i];
    totalSum += vertex[i];
}

// filling edge data structure
vector<int> edge[N];
for (int i = 0; i < N - 1; i++)
{
    edge[edges[i][0]].push_back(edges[i][1]);
    edge[edges[i][1]].push_back(edges[i][0]);
}

int res = INT_MAX;

// calling DFS method at node 0, with parent as -1
dfs(0, -1, totalSum, edge, subtree, res);
return res;
}

// Driver code to test above methods
int main()
{
    int vertex[] = {4, 2, 1, 6, 3, 5, 2};
    int edges[][2] = {{0, 1}, {0, 2}, {0, 3},
                      {2, 4}, {2, 5}, {3, 6}};
    int N = sizeof(vertex) / sizeof(vertex[0]);

    cout << getMinSubtreeSumDifference(vertex, edges, N);
    return 0;
}
```

Output:

5

Source

<https://www.geeksforgeeks.org/delete-edge-minimize-subtree-sum-difference/>

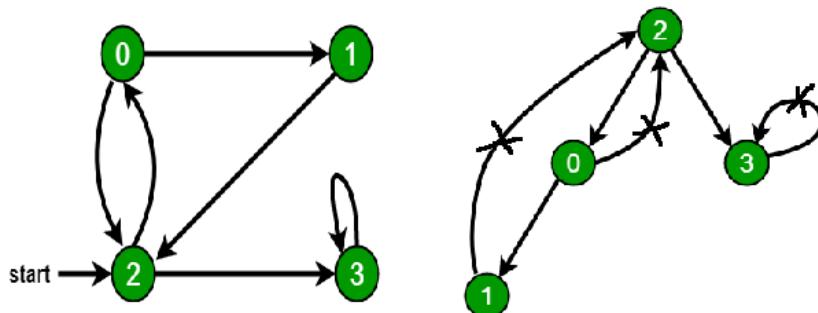
Chapter 123

Depth First Search or DFS for a Graph

Depth First Search or DFS for a Graph - GeeksforGeeks

Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Depth First Traversal of the following graph is 2, 0, 1, 3.



See [this post](#) for all applications of Depth First Traversal.

Following are implementations of simple Depth First Traversal. The C++ implementation uses [adjacency list representation](#) of graphs. [STL's list container](#) is used to store lists of adjacent nodes.

C++

```
// C++ program to print DFS traversal from
```

```
// a given vertex in a given graph
#include<iostream>
#include<list>
using namespace std;

// Graph class represents a directed graph
// using adjacency list representation
class Graph
{
    int V;      // No. of vertices

    // Pointer to an array containing
    // adjacency lists
    list<int> *adj;

    // A recursive function used by DFS
    void DFSUtil(int v, bool visited[]);

public:
    Graph(int V);    // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // DFS traversal of the vertices
    // reachable from v
    void DFS(int v);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and
    // print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent
    // to this vertex
```

```
list<int>::iterator i;
for (i = adj[v].begin(); i != adj[v].end(); ++i)
    if (!visited[*i])
        DFSUtil(*i, visited);
}

// DFS traversal of the vertices reachable from v.
// It uses recursive DFSUtil()
void Graph::DFS(int v)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function
    // to print DFS traversal
    DFSUtil(v, visited);
}

int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Depth First Traversal"
          " (starting from vertex 2) \n";
    g.DFS(2);

    return 0;
}
```

Java

```
// Java program to print DFS traversal from a given given graph
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
```

```
private int V;    // No. of vertices

// Array of lists for Adjacency List Representation
private LinkedList<Integer> adj[];

// Constructor
Graph(int v)
{
    V = v;
    adj = new LinkedList[v];
    for (int i=0; i<v; ++i)
        adj[i] = new LinkedList();
}

//Function to add an edge into the graph
void addEdge(int v, int w)
{
    adj[v].add(w);  // Add w to v's list.
}

// A function used by DFS
void DFSUtil(int v,boolean visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;
    System.out.print(v+" ");

    // Recur for all the vertices adjacent to this vertex
    Iterator<Integer> i = adj[v].listIterator();
    while (i.hasNext())
    {
        int n = i.next();
        if (!visited[n])
            DFSUtil(n, visited);
    }
}

// The function to do DFS traversal. It uses recursive DFSUtil()
void DFS(int v)
{
    // Mark all the vertices as not visited(set as
    // false by default in java)
    boolean visited[] = new boolean[V];

    // Call the recursive helper function to print DFS traversal
    DFSUtil(v, visited);
}
```

```
public static void main(String args[])
{
    Graph g = new Graph(4);

    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    System.out.println("Following is Depth First Traversal "+
                       "(starting from vertex 2)");

    g.DFS(2);
}
// This code is contributed by Aakash Hasija
```

Python

```
# Python program to print DFS traversal from a
# given given graph
from collections import defaultdict

# This class represents a directed graph using
# adjacency list representation
class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # A function used by DFS
    def DFSUtil(self,v,visited):

        # Mark the current node as visited and print it
        visited[v]= True
        print v,

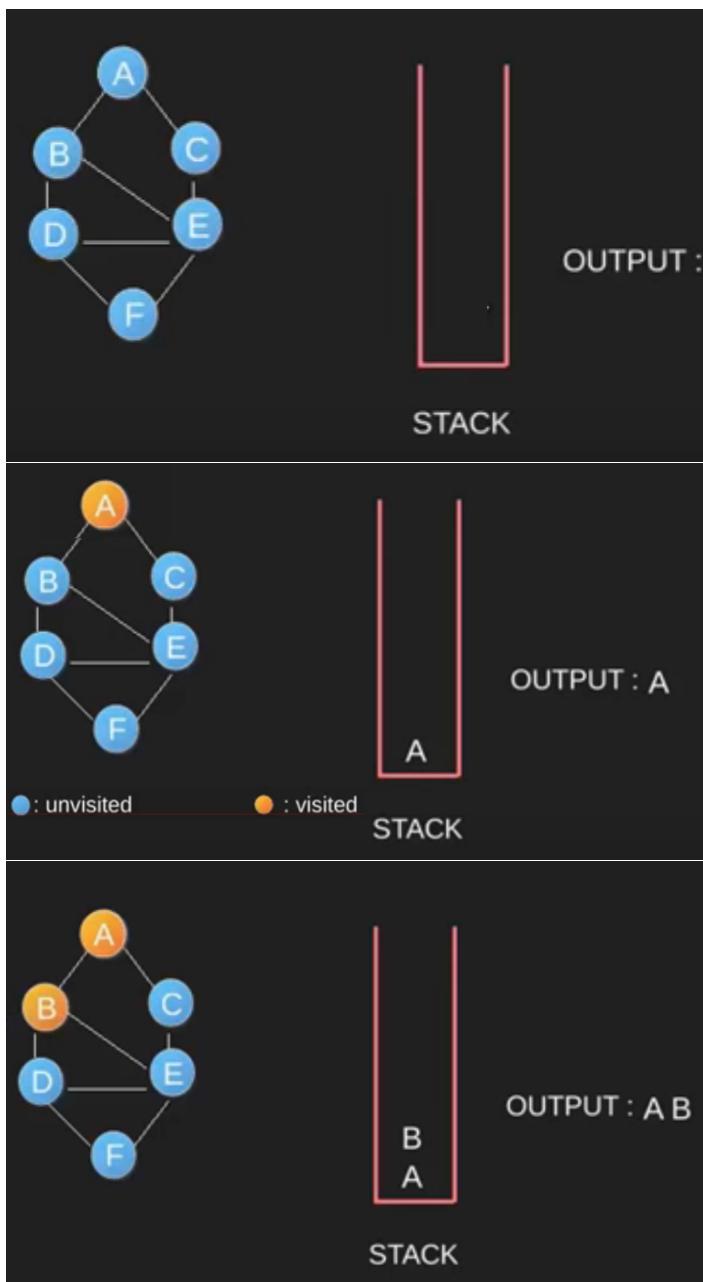
        # Recur for all the vertices adjacent to this vertex
        for i in self.graph[v]:
```

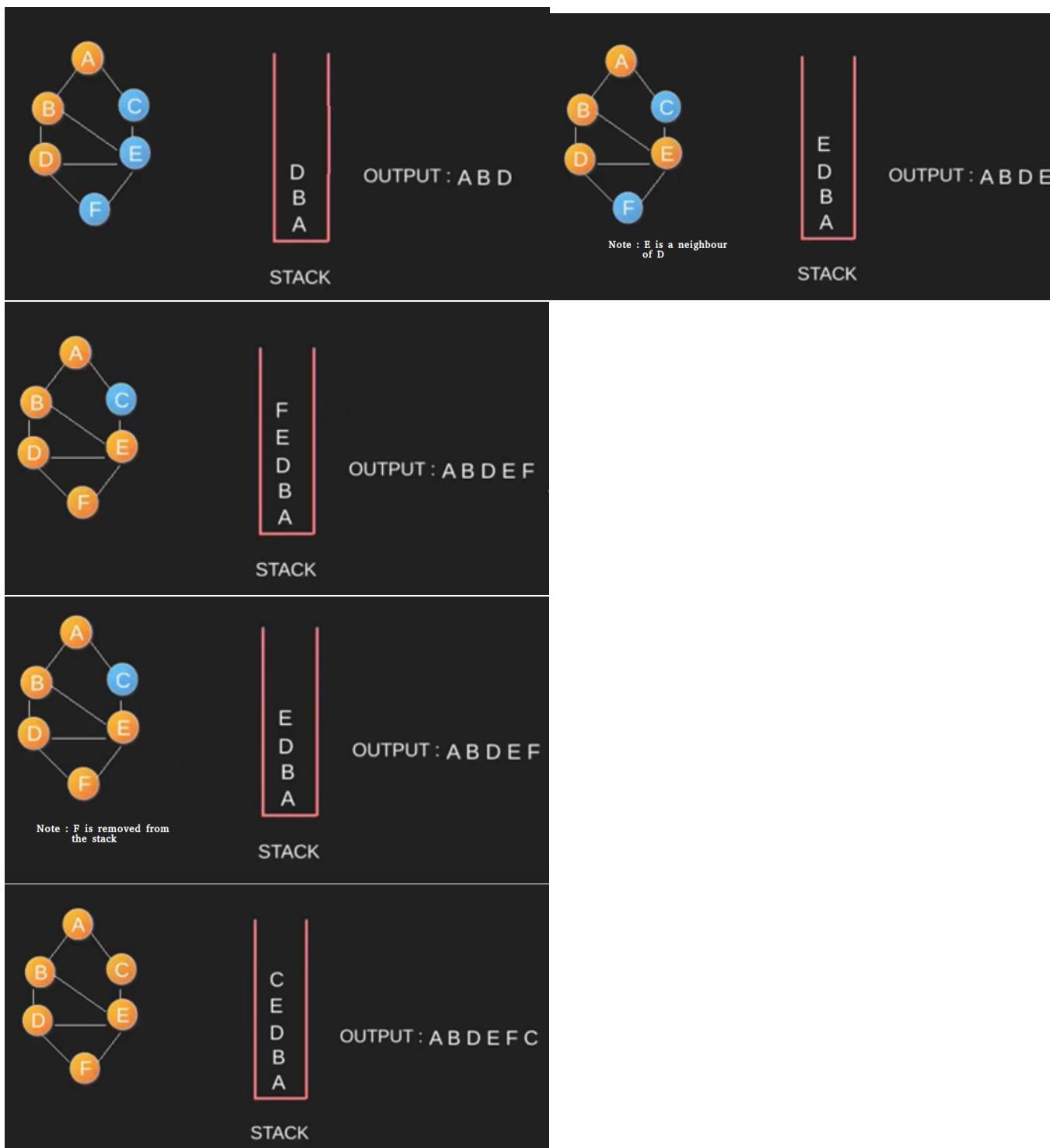
```
if visited[i] == False:  
    self.DFSUtil(i, visited)  
  
# The function to do DFS traversal. It uses  
# recursive DFSUtil()  
def DFS(self,v):  
  
    # Mark all the vertices as not visited  
    visited = [False]*(len(self.graph))  
  
    # Call the recursive helper function to print  
    # DFS traversal  
    self.DFSUtil(v,visited)  
  
# Driver code  
# Create a graph given in the above diagram  
g = Graph()  
g.addEdge(0, 1)  
g.addEdge(0, 2)  
g.addEdge(1, 2)  
g.addEdge(2, 0)  
g.addEdge(2, 3)  
g.addEdge(3, 3)  
  
print "Following is DFS from (starting from vertex 2)"  
g.DFS(2)  
  
# This code is contributed by Neelam Yadav
```

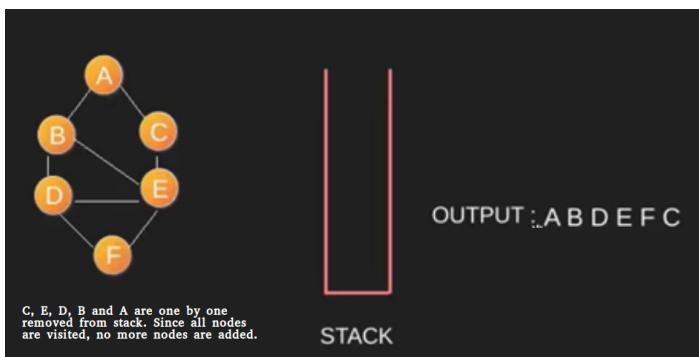
Output:

```
Following is Depth First Traversal (starting from vertex 2)  
2 0 1 3
```

Illustration for an Undirected Graph :







How to handle disconnected graph?

The above code traverses only the vertices reachable from a given source vertex. All the vertices may not be reachable from a given vertex (example Disconnected graph). To do complete DFS traversal of such graphs, we must call `DFSUtil()` for every vertex. Also, before calling `DFSUtil()`, we should check if it is already printed by some other call of `DFSUtil()`. Following implementation does the complete graph traversal even if the nodes are unreachable. The differences from the above code are highlighted in the below code.

C++

```
// C++ program to print DFS traversal for a given given graph
#include<iostream>
#include      <list>
using namespace std;

class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // Pointer to an array containing adjacency lists
    void DFSUtil(int v, bool visited[]); // A function used by DFS
public:
    Graph(int V);    // Constructor
    void addEdge(int v, int w);    // function to add an edge to graph
    void DFS();    // prints DFS traversal of the complete graph
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}
```

```
void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for(i = adj[v].begin(); i != adj[v].end(); ++i)
        if(!visited[*i])
            DFSUtil(*i, visited);
}

// The function to do DFS traversal. It uses recursive DFSUtil()
void Graph::DFS()
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to print DFS traversal
    // starting from all vertices one by one
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            DFSUtil(i, visited);
}

int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Depth First Traversaln";
    g.DFS();

    return 0;
}
```

Java

```
// Java program to print DFS traversal from a given graph
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    private int V; // No. of vertices

    // Array of lists for Adjacency List Representation
    private LinkedList<Integer> adj[];

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    //Function to add an edge into the graph
    void addEdge(int v, int w)
    {
        adj[v].add(w); // Add w to v's list.
    }

    // A function used by DFS
    void DFSUtil(int v,boolean visited[])
    {
        // Mark the current node as visited and print it
        visited[v] = true;
        System.out.print(v+" ");

        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> i = adj[v].listIterator();
        while (i.hasNext())
        {
            int n = i.next();
            if (!visited[n])
                DFSUtil(n,visited);
        }
    }

    // The function to do DFS traversal. It uses recursive DFSUtil()
    void DFS()
    {
```

```
// Mark all the vertices as not visited(set as
// false by default in java)
boolean visited[] = new boolean[V];

// Call the recursive helper function to print DFS traversal
// starting from all vertices one by one
for (int i=0; i<V; ++i)
    if (visited[i] == false)
        DFSUtil(i, visited);
}

public static void main(String args[])
{
    Graph g = new Graph(4);

    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    System.out.println("Following is Depth First Traversal");

    g.DFS();
}
}
// This code is contributed by Aakash Hasija
```

Python

```
# Python program to print DFS traversal for complete graph
from collections import defaultdict

# This class represents a directed graph using adjacency
# list representation
class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)
```

```
# A function used by DFS
def DFSUtil(self, v, visited):

    # Mark the current node as visited and print it
    visited[v] = True
    print v,

    # Recur for all the vertices adjacent to
    # this vertex
    for i in self.graph[v]:
        if visited[i] == False:
            self.DFSUtil(i, visited)

# The function to do DFS traversal. It uses
# recursive DFSUtil()
def DFS(self):
    V = len(self.graph) #total vertices

    # Mark all the vertices as not visited
    visited = [False]*(V)

    # Call the recursive helper function to print
    # DFS traversal starting from all vertices one
    # by one
    for i in range(V):
        if visited[i] == False:
            self.DFSUtil(i, visited)

# Driver code
# Create a graph given in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print "Following is Depth First Traversal"
g.DFS()

# This code is contributed by Neelam Yadav
```

Output:

Following is Depth First Traversal

0 1 2 3

Time Complexity: $O(V+E)$ where V is number of vertices in the graph and E is number of edges in the graph.

- [Applications of DFS.](#)
- [Breadth First Traversal for a Graph](#)
- [Recent Articles on DFS](#)

Source

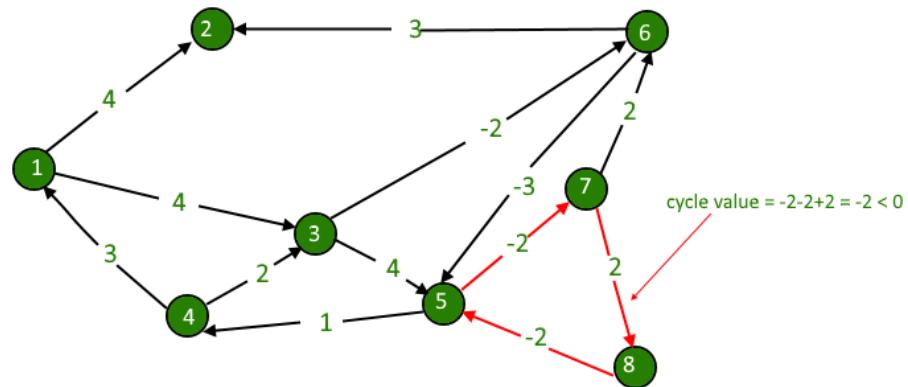
<https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/>

Chapter 124

Detect a negative cycle in a Graph (Bellman Ford)

Detect a negative cycle in a Graph (Bellman Ford) - GeeksforGeeks

We are given a directed graph. We need compute whether the graph has negative cycle or not. A negative cycle is one in which the overall sum of the cycle comes negative.



Negative weights are found in various applications of graphs. For example, instead of paying cost for a path, we may get some advantage if we follow the path.

Examples:

```
Input : 4 4
        0 1 1
        1 2 -1
        2 3 -1
        3 0 -1
```

```
Output : Yes
The graph contains a negative cycle.
```

The idea is to use [Bellman Ford Algorithm](#).

Below is algorithm find if there is a negative weight cycle reachable from given source.

- 1) Initialize distances from source to all vertices as infinite and distance to source itself as 0. Create an array dist[] of size V with all values as infinite except dist[src] where src is source vertex.
- 2) This step calculates shortest distances. Do following V-1 times where V is the number of vertices in given graph.
 - a) Do following for each edge u-v
 - If dist[v] > dist[u] + weight of edge uv, then update dist[v]
 - dist[v] = dist[u] + weight of edge uv
- 3) This step reports if there is a negative weight cycle in graph. Do following for each edge u-v
 - If dist[v] > dist[u] + weight of edge uv, then “Graph contains negative weight cycle”The idea of step 3 is, step 2 guarantees shortest distances if graph doesn’t contain negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle.

```
// A C++ program to check if a graph contains negative
// weight cycle using Bellman-Ford algorithm. This program
// works only if all vertices are reachable from a source
// vertex 0.
#include <bits/stdc++.h>
using namespace std;

// a structure to represent a weighted edge in graph
struct Edge {
    int src, dest, weight;
};

// a structure to represent a connected, directed and
// weighted graph
```

```

struct Graph {
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[graph->E];
    return graph;
}

// The main function that finds shortest distances
// from src to all other vertices using Bellman-
// Ford algorithm. The function also detects
// negative weight cycle
bool isNegCycleBellmanFord(struct Graph* graph,
                           int src)
{
    int V = graph->V;
    int E = graph->E;
    int dist[V];

    // Step 1: Initialize distances from src
    // to all other vertices as INFINITE
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX;
    dist[src] = 0;

    // Step 2: Relax all edges |V| - 1 times.
    // A simple shortest path from src to any
    // other vertex can have at-most |V| - 1
    // edges
    for (int i = 1; i <= V - 1; i++) {
        for (int j = 0; j < E; j++) {
            int u = graph->edge[j].src;
            int v = graph->edge[j].dest;
            int weight = graph->edge[j].weight;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }
}

```

```
// Step 3: check for negative-weight cycles.  
// The above step guarantees shortest distances  
// if graph doesn't contain negative weight cycle.  
// If we get a shorter path, then there  
// is a cycle.  
for (int i = 0; i < E; i++) {  
    int u = graph->edge[i].src;  
    int v = graph->edge[i].dest;  
    int weight = graph->edge[i].weight;  
    if (dist[u] != INT_MAX && dist[u] + weight < dist[v])  
        return true;  
}  
  
return false;  
}  
  
// Driver program to test above functions  
int main()  
{  
    /* Let us create the graph given in above example */  
    int V = 5; // Number of vertices in graph  
    int E = 8; // Number of edges in graph  
    struct Graph* graph = createGraph(V, E);  
  
    // add edge 0-1 (or A-B in above figure)  
    graph->edge[0].src = 0;  
    graph->edge[0].dest = 1;  
    graph->edge[0].weight = -1;  
  
    // add edge 0-2 (or A-C in above figure)  
    graph->edge[1].src = 0;  
    graph->edge[1].dest = 2;  
    graph->edge[1].weight = 4;  
  
    // add edge 1-2 (or B-C in above figure)  
    graph->edge[2].src = 1;  
    graph->edge[2].dest = 2;  
    graph->edge[2].weight = 3;  
  
    // add edge 1-3 (or B-D in above figure)  
    graph->edge[3].src = 1;  
    graph->edge[3].dest = 3;  
    graph->edge[3].weight = 2;  
  
    // add edge 1-4 (or A-E in above figure)  
    graph->edge[4].src = 1;  
    graph->edge[4].dest = 4;
```

```
graph->edge[4].weight = 2;

// add edge 3-2 (or D-C in above figure)
graph->edge[5].src = 3;
graph->edge[5].dest = 2;
graph->edge[5].weight = 5;

// add edge 3-1 (or D-B in above figure)
graph->edge[6].src = 3;
graph->edge[6].dest = 1;
graph->edge[6].weight = 1;

// add edge 4-3 (or E-D in above figure)
graph->edge[7].src = 4;
graph->edge[7].dest = 3;
graph->edge[7].weight = -3;

if (isNegCycleBellmanFord(graph, 0))
    cout << "Yes";
else
    cout << "No";

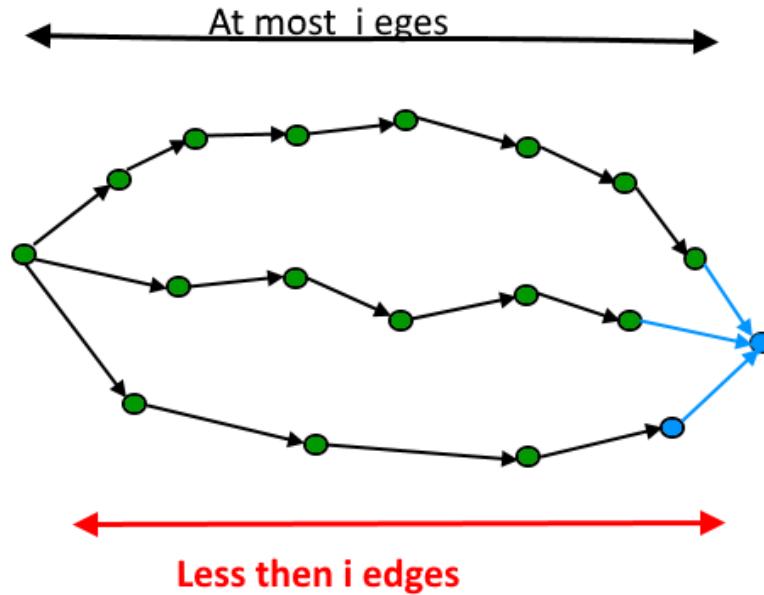
return 0;
}
```

Output :

No

How does it work?

As discussed in [Bellman Ford algorithm](#), for a given source, it first calculates the shortest distances which have at-most one edge in the path. Then, it calculates shortest paths with at-most 2 edges, and so on. After the i -th iteration of outer loop, the shortest paths with at most i edges are calculated. There can be maximum $V - 1$ edges in any simple path, that is why the outer loop runs $v - 1$ times. If there is a negative weight cycle, then one more iteration would give a shorter path.



How to handle disconnected graph (If cycle is not reachable from source)?

The above algorithm and program might not work if the given graph is disconnected. It works when all vertices are reachable from source vertex 0.

To handle disconnected graph, we can repeat the process for vertices for which distance is infinite.

```

// A C++ program for Bellman-Ford's single source
// shortest path algorithm.
#include <bits/stdc++.h>
using namespace std;

// a structure to represent a weighted edge in graph
struct Edge {
    int src, dest, weight;
};

// a structure to represent a connected, directed and
// weighted graph
struct Graph {
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges

```

```

struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[graph->E];
    return graph;
}

// The main function that finds shortest distances
// from src to all other vertices using Bellman-
// Ford algorithm. The function also detects
// negative weight cycle
bool isNegCycleBellmanFord(struct Graph* graph,
                           int src, int dist[])
{
    int V = graph->V;
    int E = graph->E;

    // Step 1: Initialize distances from src
    // to all other vertices as INFINITE
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX;
    dist[src] = 0;

    // Step 2: Relax all edges |V| - 1 times.
    // A simple shortest path from src to any
    // other vertex can have at-most |V| - 1
    // edges
    for (int i = 1; i <= V - 1; i++) {
        for (int j = 0; j < E; j++) {
            int u = graph->edge[j].src;
            int v = graph->edge[j].dest;
            int weight = graph->edge[j].weight;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }

    // Step 3: check for negative-weight cycles.
    // The above step guarantees shortest distances
    // if graph doesn't contain negative weight cycle.
    // If we get a shorter path, then there
    // is a cycle.
    for (int i = 0; i < E; i++) {
        int u = graph->edge[i].src;
        int v = graph->edge[i].dest;
        int weight = graph->edge[i].weight;

```

```
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
            return true;
    }

    return false;
}

// Returns true if given graph has negative weight
// cycle.
bool isNegCycleDisconnected(struct Graph* graph)
{

    int V = graph->V;
    // To keep track of visited vertices to avoid
    // recomputations.
    bool visited[V];
    memset(visited, 0, sizeof(visited));

    // This array is filled by Bellman-Ford
    int dist[V];

    // Call Bellman-Ford for all those vertices
    // that are not visited
    for (int i = 0; i < V; i++) {
        if (visited[i] == false) {
            // If cycle found
            if (isNegCycleBellmanFord(graph, i, dist))
                return true;

            // Mark all vertices that are visited
            // in above call.
            for (int i = 0; i < V; i++)
                if (dist[i] != INT_MAX)
                    visited[i] = true;
        }
    }

    return false;
}

// Driver program to test above functions
int main()
{
    /* Let us create the graph given in above example */
    int V = 5; // Number of vertices in graph
    int E = 8; // Number of edges in graph
    struct Graph* graph = createGraph(V, E);
```

```
// add edge 0-1 (or A-B in above figure)
graph->edge[0].src = 0;
graph->edge[0].dest = 1;
graph->edge[0].weight = -1;

// add edge 0-2 (or A-C in above figure)
graph->edge[1].src = 0;
graph->edge[1].dest = 2;
graph->edge[1].weight = 4;

// add edge 1-2 (or B-C in above figure)
graph->edge[2].src = 1;
graph->edge[2].dest = 2;
graph->edge[2].weight = 3;

// add edge 1-3 (or B-D in above figure)
graph->edge[3].src = 1;
graph->edge[3].dest = 3;
graph->edge[3].weight = 2;

// add edge 1-4 (or A-E in above figure)
graph->edge[4].src = 1;
graph->edge[4].dest = 4;
graph->edge[4].weight = 2;

// add edge 3-2 (or D-C in above figure)
graph->edge[5].src = 3;
graph->edge[5].dest = 2;
graph->edge[5].weight = 5;

// add edge 3-1 (or D-B in above figure)
graph->edge[6].src = 3;
graph->edge[6].dest = 1;
graph->edge[6].weight = 1;

// add edge 4-3 (or E-D in above figure)
graph->edge[7].src = 4;
graph->edge[7].dest = 3;
graph->edge[7].weight = -3;

if (isNegCycleDisconnected(graph))
    cout << "Yes";
else
    cout << "No";

return 0;
}
```

Output :

No

Detecting negative cycle using Floyd Warshall

Source

<https://www.geeksforgeeks.org/detect-negative-cycle-graph-bellman-ford/>

Chapter 125

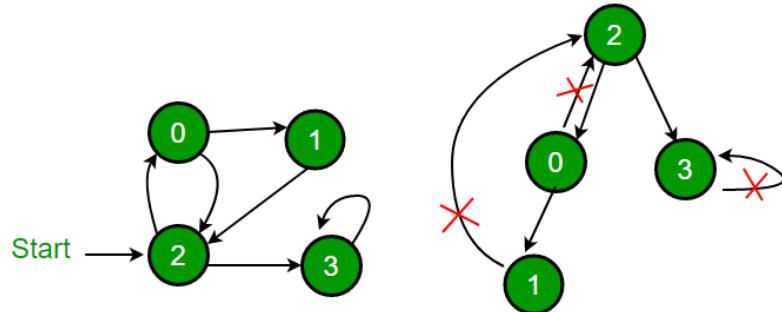
Detect Cycle in a directed graph using colors

Detect Cycle in a directed graph using colors - GeeksforGeeks

Given a directed graph, check whether the graph contains a cycle or not. Your function should return true if the given graph contains at least one cycle, else return false. For example, the following graph contains three cycles $0 \rightarrow 2 \rightarrow 0$, $0 \rightarrow 1 \rightarrow 2 \rightarrow 0$ and $3 \rightarrow 3$, so your function must return true.

Solution

Depth First Traversal can be used to detect cycle in a Graph. DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a [back edge](#) present in the graph. A back edge is an edge that is from a node to itself (selfloop) or one of its ancestor in the tree produced by DFS. In the following graph, there are 3 back edges, marked with cross sign. We can observe that these 3 back edges indicate 3 cycles present in the graph.



For a disconnected graph, we get the DFS forest as output. To detect cycle, we can check for cycle in individual trees by checking back edges.

[image](#)

Image Source: <http://www.cs.yale.edu/homes/aspnes/pinewiki/DepthFirstSearch.html>

In the [previous post](#), we have discussed a solution that stores visited vertices in a separate array which stores vertices of current recursion call stack.

In this post a different solution is discussed. The solution is from [CLRS book](#). The idea is to do DFS of given graph and while doing traversal, assign one of the below three colors to every vertex.

WHITE : Vertex is not processed yet. Initially all vertices are WHITE.

GRAY : Vertex is being processed (DFS for this vertex has started, but not finished which means that all descendants (in DFS tree) of this vertex are not processed yet (or this vertex is in function call stack)

BLACK : Vertex and all its descendants are processed.

While doing DFS, if we encounter an edge from current vertex to a GRAY vertex, then this edge is back edge and hence there is a cycle.

Below is C++ implementation based on above idea.

C++

```
// A DFS based approach to find if there is a cycle
// in a directed graph. This approach strictly follows
// the algorithm given in CLRS book.
#include <bits/stdc++.h>
using namespace std;

enum Color {WHITE, GRAY, BLACK};

// Graph class represents a directed graph using
// adjacency list representation
class Graph
{
    int V; // No. of vertices
    list<int>* adj; // adjacency lists

    // DFS traversal of the vertices reachable from v
    bool DFSUtil(int v, int color[]);

public:
    Graph(int V); // Constructor
```

```
// function to add an edge to graph
void addEdge(int v, int w);

    bool isCyclic();
};

// Constructor
Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

// Utility function to add an edge
void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

// Recursive function to find if there is back edge
// in DFS subtree tree rooted with 'u'
bool Graph::DFSUtil(int u, int color[])
{
    // GRAY : This vertex is being processed (DFS
    //         for this vertex has started, but not
    //         ended (or this vertex is in function
    //         call stack)
    color[u] = GRAY;

    // Iterate through all adjacent vertices
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        int v = *i; // An adjacent of u

        // If there is
        if (color[v] == GRAY)
            return true;

        // If v is not processed and there is a back
        // edge in subtree rooted with v
        if (color[v] == WHITE && DFSUtil(v, color))
            return true;
    }

    // Mark this vertex as processed
    color[u] = BLACK;
```

```
        return false;
    }

// Returns true if there is a cycle in graph
bool Graph::isCyclic()
{
    // Initialize color of all vertices as WHITE
    int *color = new int[V];
    for (int i = 0; i < V; i++)
        color[i] = WHITE;

    // Do a DFS traversal beginning with all
    // vertices
    for (int i = 0; i < V; i++)
        if (color[i] == WHITE)
            if (DFSUtil(i, color) == true)
                return true;

    return false;
}

// Driver code to test above
int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    if (g.isCyclic())
        cout << "Graph contains cycle";
    else
        cout << "Graph doesn't contain cycle";

    return 0;
}
```

Python

```
# Python program to detect cycle in
# a directed graph

from collections import defaultdict
```

```

class Graph():
    def __init__(self, V):
        self.V = V
        self.graph = defaultdict(list)

    def addEdge(self, u, v):
        self.graph[u].append(v)

    def DFSUtil(self, u, color):
        # GRAY : This vertex is being processed (DFS
        #         for this vertex has started, but not
        #         ended (or this vertex is in function
        #         call stack)
        color[u] = "GRAY"

        for v in self.graph[u]:
            if color[v] == "GRAY":
                return True

            if color[v] == "WHITE" and self.DFSUtil(v, color) == True:
                return True

        color[u] = "BLACK"
        return False

    def isCyclic(self):
        color = ["WHITE"] * self.V

        for i in range(self.V):
            if color[i] == "WHITE":
                if self.DFSUtil(i, color) == True:
                    return True
        return False

# Driver program to test above functions

g = Graph(4)
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)
print "Graph contains cycle" if g.isCyclic() == True\
else "Graph doesn't contain cycle"

# This program is contributed by Divyanshu Mehta

```

Output :

```
Graph contains cycle
```

Time complexity of above solution is $O(V + E)$ where V is number of vertices and E is number of edges in the graph.

This article is contributed by **Aditya Goel**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

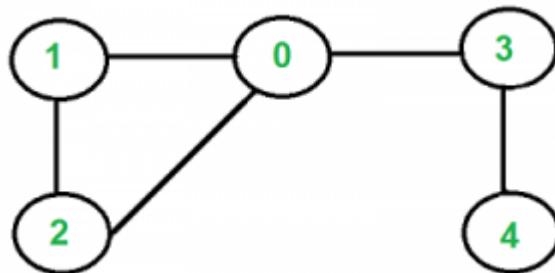
<https://www.geeksforgeeks.org/detect-cycle-direct-graph-using-colors/>

Chapter 126

Detect cycle in an undirected graph using BFS

Detect cycle in an undirected graph using BFS - GeeksforGeeks

Given an undirected graph, how to check if there is a cycle in the graph? For example, the following graph has a cycle 1-0-2-1.



We have discussed [cycle detection for directed graph](#). We have also discussed a [union-find algorithm for cycle detection in undirected graphs](#). The time complexity of the union-find algorithm is $O(E\log V)$. Like directed graphs, we can use [DFS](#) to detect cycle in an undirected graph in $O(V+E)$ time. We have discussed [DFS based solution for cycle detection in undirected graph](#).

In this article, [BFS](#) based solution is discussed. We do a BFS traversal of the given graph. For every visited vertex 'v', if there is an adjacent 'u' such that u is already visited and u is not parent of v, then there is a cycle in graph. If we don't find such an adjacent for any vertex, we say that there is no cycle. The assumption of this approach is that there are no parallel edges between any two vertices.

We use a parent array to keep track of parent vertex for a vertex so that we do not consider visited parent as cycle.

```
// C++ program to detect cycle in an undirected graph
```

```
// using BFS.
#include <bits/stdc++.h>
using namespace std;

void addEdge(vector<int> adj[], int u, int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}

bool isCyclicConnected(vector<int> adj[], int s,
                      int V, vector<bool>& visited)
{
    // Set parent vertex for every vertex as -1.
    vector<int> parent(V, -1);

    // Create a queue for BFS
    queue<int> q;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    q.push(s);

    while (!q.empty()) {

        // Dequeue a vertex from queue and print it
        int u = q.front();
        q.pop();

        // Get all adjacent vertices of the dequeued
        // vertex s. If a adjacent has not been visited,
        // then mark it visited and enqueue it. We also
        // mark parent so that parent is not considered
        // for cycle.
        for (auto v : adj[u]) {
            if (!visited[v]) {
                visited[v] = true;
                q.push(v);
                parent[v] = u;
            }
            else if (parent[u] != v)
                return true;
        }
    }
    return false;
}

bool isCyclicDisconnected(vector<int> adj[], int V)
```

```
{  
    // Mark all the vertices as not visited  
    vector<bool> visited(V, false);  
  
    for (int i = 0; i < V; i++)  
        if (!visited[i] && isCyclicConnected(adj, i,  
                                              V, visited))  
            return true;  
    return false;  
}  
  
// Driver program to test methods of graph class  
int main()  
{  
    int V = 4;  
    vector<int> adj[V];  
    addEdge(adj, 0, 1);  
    addEdge(adj, 1, 2);  
    addEdge(adj, 2, 0);  
    addEdge(adj, 2, 3);  
  
    if (isCyclicDisconnected(adj, V))  
        cout << "Yes";  
    else  
        cout << "No";  
  
    return 0;  
}
```

Output :

Yes

Time Complexity: The program does a simple BFS Traversal of graph and graph is represented using adjacency list. So the time complexity is $O(V+E)$

Source

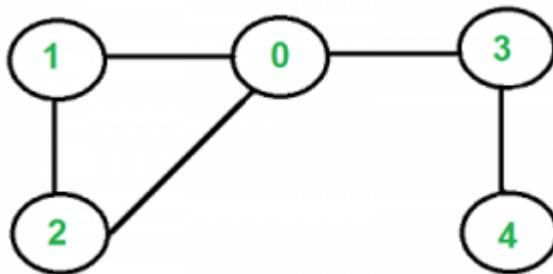
<https://www.geeksforgeeks.org/detect-cycle-in-an-undirected-graph-using-bfs/>

Chapter 127

Detect cycle in an undirected graph

Detect cycle in an undirected graph - GeeksforGeeks

Given an undirected graph, how to check if there is a cycle in the graph? For example, the following graph has a cycle 1-0-2-1.



We have discussed [cycle detection for directed graph](#). We have also discussed a [union-find algorithm for cycle detection in undirected graphs](#). The time complexity of the union-find algorithm is $O(E\log V)$. Like directed graphs, we can use [DFS](#) to detect cycle in an undirected graph in $O(V+E)$ time. We do a DFS traversal of the given graph. For every visited vertex 'v', if there is an adjacent 'u' such that u is already visited and u is not parent of v, then there is a cycle in graph. If we don't find such an adjacent for any vertex, we say that there is no cycle. The assumption of this approach is that there are no parallel edges between any two vertices.

C++

```
// A C++ Program to detect cycle in an undirected graph
#include<iostream>
#include <list>
#include <limits.h>
```

```
using namespace std;

// Class for an undirected graph
class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // Pointer to an array containing adjacency lists
    bool isCyclicUtil(int v, bool visited[], int parent);
public:
    Graph(int V);    // Constructor
    void addEdge(int v, int w);    // to add an edge to graph
    bool isCyclic();    // returns true if there is a cycle
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
    adj[w].push_back(v); // Add v to w's list.
}

// A recursive function that uses visited[] and parent to detect
// cycle in subgraph reachable from vertex v.
bool Graph::isCyclicUtil(int v, bool visited[], int parent)
{
    // Mark the current node as visited
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
    {
        // If an adjacent is not visited, then recur for that adjacent
        if (!visited[*i])
        {
            if (isCyclicUtil(*i, visited, v))
                return true;
        }
    }

    // If an adjacent is visited and not parent of current vertex,
    // then there is a cycle.
    else if (*i != parent)
        return true;
}
```

```
        }
        return false;
    }

// Returns true if the graph contains a cycle, else false.
bool Graph::isCyclic()
{
    // Mark all the vertices as not visited and not part of recursion
    // stack
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to detect cycle in different
    // DFS trees
    for (int u = 0; u < V; u++)
        if (!visited[u]) // Don't recur for u if it is already visited
            if (isCyclicUtil(u, visited, -1))
                return true;

    return false;
}

// Driver program to test above functions
int main()
{
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 0);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.isCyclic()? cout << "Graph contains cycle\n":
                  cout << "Graph doesn't contain cycle\n";

    Graph g2(3);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.isCyclic()? cout << "Graph contains cycle\n":
                  cout << "Graph doesn't contain cycle\n";

    return 0;
}
```

Java

```
// A Java Program to detect cycle in an undirected graph
import java.io.*;
```

```
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    private int V;    // No. of vertices
    private LinkedList<Integer> adj[]; // Adjacency List Representation

    // Constructor
    Graph(int v) {
        V = v;
        adj = new LinkedList[v];
        for(int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    // Function to add an edge into the graph
    void addEdge(int v,int w) {
        adj[v].add(w);
        adj[w].add(v);
    }

    // A recursive function that uses visited[] and parent to detect
    // cycle in subgraph reachable from vertex v.
    Boolean isCyclicUtil(int v, Boolean visited[], int parent)
    {
        // Mark the current node as visited
        visited[v] = true;
        Integer i;

        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> it = adj[v].iterator();
        while (it.hasNext())
        {
            i = it.next();

            // If an adjacent is not visited, then recur for that
            // adjacent
            if (!visited[i])
            {
                if (isCyclicUtil(i, visited, v))
                    return true;
            }
        }

        // If an adjacent is visited and not parent of current
        // vertex, then there is a cycle.
        else if (i != parent)
    }
}
```

```
        return true;
    }
    return false;
}

// Returns true if the graph contains a cycle, else false.
Boolean isCyclic()
{
    // Mark all the vertices as not visited and not part of
    // recursion stack
    Boolean visited[] = new Boolean[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to detect cycle in
    // different DFS trees
    for (int u = 0; u < V; u++)
        if (!visited[u]) // Don't recur for u if already visited
            if (isCyclicUtil(u, visited, -1))
                return true;

    return false;
}

// Driver method to test above methods
public static void main(String args[])
{
    // Create a graph given in the above diagram
    Graph g1 = new Graph(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 0);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    if (g1.isCyclic())
        System.out.println("Graph contains cycle");
    else
        System.out.println("Graph doesn't contain cycle");

    Graph g2 = new Graph(3);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    if (g2.isCyclic())
        System.out.println("Graph contains cycle");
    else
        System.out.println("Graph doesn't contain cycle");
}
```

```
}
```

// This code is contributed by Aakash Hasija

Python

```
# Python Program to detect cycle in an undirected graph

from collections import defaultdict

#This class represents a undirected graph using adjacency list representation
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices
        self.graph = defaultdict(list) # default dictionary to store graph

    # function to add an edge to graph
    def addEdge(self,v,w):
        self.graph[v].append(w) #Add w to v_s list
        self.graph[w].append(v) #Add v to w_s list

    # A recursive function that uses visited[] and parent to detect
    # cycle in subgraph reachable from vertex v.
    def isCyclicUtil(self,v,visited,parent):

        #Mark the current node as visited
        visited[v]= True

        #Recur for all the vertices adjacent to this vertex
        for i in self.graph[v]:
            # If the node is not visited then recurse on it
            if  visited[i]==False :
                if(self.isCyclicUtil(i,visited,v)):
                    return True
            # If an adjacent vertex is visited and not parent of current vertex,
            # then there is a cycle
            elif parent!=i:
                return True

        return False

    #Returns true if the graph contains a cycle, else false.
    def isCyclic(self):
        # Mark all the vertices as not visited
        visited =[False]*(self.V)
        # Call the recursive helper function to detect cycle in different
```

```
#DFS trees
for i in range(self.V):
    if visited[i] ==False: #Don't recur for u if it is already visited
        if(self.isCyclicUtil(i,visited,-1))== True:
            return True

    return False

# Create a graph given in the above diagram
g = Graph(5)
g.addEdge(1, 0)
g.addEdge(0, 2)
g.addEdge(2, 0)
g.addEdge(0, 3)
g.addEdge(3, 4)

if g.isCyclic():
    print "Graph contains cycle"
else :
    print "Graph does not contain cycle "
g1 = Graph(3)
g1.addEdge(0,1)
g1.addEdge(1,2)

if g1.isCyclic():
    print "Graph contains cycle"
else :
    print "Graph does not contain cycle "

#This code is contributed by Neelam Yadav
```

Output:

```
Graph contains cycle
Graph doesn't contain cycle
```

Time Complexity: The program does a simple DFS Traversal of graph and graph is represented using adjacency list. So the time complexity is $O(V+E)$

Exercise: Can we use BFS to detect cycle in an undirected graph in $O(V+E)$ time? What about directed graphs?

Source

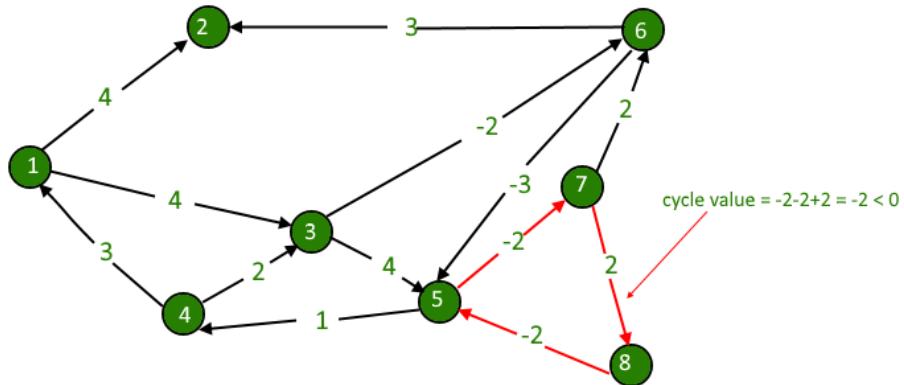
<https://www.geeksforgeeks.org/detect-cycle-undirected-graph/>

Chapter 128

Detecting negative cycle using Floyd Warshall

Detecting negative cycle using Floyd Warshall - GeeksforGeeks

We are given a directed graph. We need compute whether the graph has negative cycle or not. A negative cycle is one in which the overall sum of the cycle comes negative.



Negative weights are found in various applications of graphs. For example, instead of paying cost for a path, we may get some advantage if we follow the path.

Examples:

```
Input : 4 4
        0 1 1
        1 2 -1
        2 3 -1
        3 0 -1

Output : Yes
The graph contains a negative cycle.
```

We have discussed [Bellman Ford Algorithm](#) based solution for this problem.

In this post, [Floyd Warshall Algorithm](#) based solution is discussed that works for both connected and disconnected graphs.

Distance of any node from itself is always zero. But in some cases, as in this example, when we traverse further from 4 to 1, the distance comes out to be -2, i.e. distance of 1 from 1 will become -2. This is our catch, we just have to check the nodes distance from itself and if it comes out to be negative, we will detect the required negative cycle.

C++

```
// C++ Program to check if there is a negative weight
// cycle using Floyd Warshall Algorithm
#include<iostream>
using namespace std;

// Number of vertices in the graph
#define V 4

/* Define Infinite as a large enough value. This
   value will be used for vertices not connected
   to each other */
#define INF 99999

// A function to print the solution matrix
void printSolution(int dist[][]);

// Returns true if graph has negative weight cycle
// else false.
bool negCyclefloydWarshall(int graph[][V])
{
    /* dist[][] will be the output matrix that will
       finally have the shortest
```

```

distances between every pair of vertices */
int dist[V][V], i, j, k;

/* Initialize the solution matrix same as input
graph matrix. Or we can say the initial values
of shortest distances are based on shortest
paths considering no intermediate vertex. */
for (i = 0; i < V; i++)
    for (j = 0; j < V; j++)
        dist[i][j] = graph[i][j];

/* Add all vertices one by one to the set of
intermediate vertices.
---> Before start of a iteration, we have shortest
distances between all pairs of vertices such
that the shortest distances consider only the
vertices in set {0, 1, 2, .. k-1} as intermediate
vertices.
----> After the end of a iteration, vertex no. k is
added to the set of intermediate vertices and
the set becomes {0, 1, 2, .. k} */
for (k = 0; k < V; k++)
{
    // Pick all vertices as source one by one
    for (i = 0; i < V; i++)
    {
        // Pick all vertices as destination for the
        // above picked source
        for (j = 0; j < V; j++)
        {
            // If vertex k is on the shortest path from
            // i to j, then update the value of dist[i][j]
            if (dist[i][k] + dist[k][j] < dist[i][j])
                dist[i][j] = dist[i][k] + dist[k][j];
        }
    }
}

// If distance of any vertex from itself
// becomes negative, then there is a negative
// weight cycle.
for (int i = 0; i < V; i++)
    if (dist[i][i] < 0)
        return true;
return false;
}

// driver program

```

```
int main()
{
    /* Let us create the following weighted graph
       1
      (0)----->(1)
      /|\           |
      |           |
     -1 |           | -1
      |           \|/
      (3)<----- (2)
      -1           */
}

int graph[V][V] = { {0, 1, INF, INF},
                    {INF, 0, -1, INF},
                    {INF, INF, 0, -1},
                    {-1, INF, INF, 0}};

if (negCyclefloydWarshall(graph))
    cout << "Yes";
else
    cout << "No";
return 0;
}
```

Java

```
// Java Program to check if there is a negative weight
// cycle using Floyd Warshall Algorithm
class GFG
{

    // Number of vertices in the graph
    static final int V = 4;

    /* Define Infinite as a large enough value. This
    value will be used for vertices not connected
    to each other */
    static final int INF = 99999;

    // Returns true if graph has negative weight cycle
    // else false.
    static boolean negCyclefloydWarshall(int graph[][])
    {

        /* dist[][] will be the output matrix that will
        finally have the shortest
        distances between every pair of vertices */
        int dist[][] = new int[V][V], i, j, k;
```

```

/* Initialize the solution matrix same as input
graph matrix. Or we can say the initial values
of shortest distances are based on shortest
paths considering no intermediate vertex. */
for (i = 0; i < V; i++)
    for (j = 0; j < V; j++)
        dist[i][j] = graph[i][j];

/* Add all vertices one by one to the set of
intermediate vertices.
----> Before start of a iteration, we have shortest
distances between all pairs of vertices such
that the shortest distances consider only the
vertices in set {0, 1, 2, .. k-1} as intermediate
vertices.
----> After the end of a iteration, vertex no. k is
added to the set of intermediate vertices and
the set becomes {0, 1, 2, .. k} */
for (k = 0; k < V; k++)
{
    // Pick all vertices as source one by one
    for (i = 0; i < V; i++)
    {
        // Pick all vertices as destination for the
        // above picked source
        for (j = 0; j < V; j++)
        {
            // If vertex k is on the shortest path from
            // i to j, then update the value of dist[i][j]
            if (dist[i][k] + dist[k][j] < dist[i][j])
                dist[i][j] = dist[i][k] + dist[k][j];
        }
    }

    // If distance of any vertex from itself
    // becomes negative, then there is a negative
    // weight cycle.
    for (i = 0; i < V; i++)
        if (dist[i][i] < 0)
            return true;

    return false;
}

```

```
// Driver code
public static void main (String[] args)
{
    /* Let us create the following weighted graph
       1
      (0)----->(1)
      /|\           |
      |           |
     -1 |           | -1
      |           \|/
     (3)<----- (2)
      -1      */
    int graph[][] = { {0, 1, INF, INF},
                      {INF, 0, -1, INF},
                      {INF, INF, 0, -1},
                      {-1, INF, INF, 0}};

    if (negCyclefloydWarshall(graph))
        System.out.print("Yes");
    else
        System.out.print("No");
}
}

// This code is contributed by Anant Agarwal.
```

Python3

```
# Python Program to check
# if there is a
# negative weight
# cycle using Floyd
# Warshall Algorithm

# Number of vertices
# in the graph
V = 4

# Define Infinite as a
# large enough value. This
# value will be used
#for vertices not connected
# to each other
INF = 99999
```

```

# Returns true if graph has
# negative weight cycle
# else false.
def negCyclefloydWarshall(graph):

    # dist[][] will be the
    # output matrix that will
    # finally have the shortest
    # distances between every
    # pair of vertices
    dist=[[0 for i in range(V+1)]for j in range(V+1)]

    # Initialize the solution
    # matrix same as input
    # graph matrix. Or we can
    # say the initial values
    # of shortest distances
    # are based on shortest
    # paths considering no
    # intermediate vertex.
    for i in range(V):
        for j in range(V):
            dist[i][j] = graph[i][j]

    ''' Add all vertices one
        by one to the set of
        intermediate vertices.
    ---> Before start of a iteration,
        we have shortest
        distances between all pairs
        of vertices such
        that the shortest distances
        consider only the
        vertices in set {0, 1, 2, .. k-1}
        as intermediate vertices.
    ----> After the end of a iteration,
        vertex no. k is
        added to the set of
        intermediate vertices and
        the set becomes {0, 1, 2, .. k} '''
    for k in range(V):

        # Pick all vertices
        # as source one by one
        for i in range(V):

            # Pick all vertices as

```

```

# destination for the
# above picked source
for j in range(V):

    # If vertex k is on
    # the shortest path from
    # i to j, then update
    # the value of dist[i][j]
    if (dist[i][k] + dist[k][j] < dist[i][j]):
        dist[i][j] = dist[i][k] + dist[k][j]

# If distance of any
# vertex from itself
# becomes negative, then
# there is a negative
# weight cycle.
for i in range(V):
    if (dist[i][i] < 0):
        return True

return False

# Driver code

''' Let us create the
following weighted graph
      1
      (0)----->(1)
      /|\           |
      |           |
      -1|           | -1
      |           \|/
      (3)<----->(2)
      -1           '''

graph = [ [0, 1, INF, INF],
          [INF, 0, -1, INF],
          [INF, INF, 0, -1],
          [-1, INF, INF, 0]]

if (negCyclefloydWarshall(graph)):
    print("Yes")
else:
    print("No")

# This code is contributed

```

```
# by Anant Agarwal.

C#

// C# Program to check if there
// is a negative weight cycle
// using Floyd Warshall Algorithm

using System;

namespace Cycle
{
public class GFG
{

    // Number of vertices in the graph
    static int V = 4;

    /* Define Infinite as a large enough value. This
    value will be used for vertices not connected
    to each other */
    static int INF = 99999;

    // Returns true if graph has negative weight cycle
    // else false.
    static bool negCyclefloydWarshall(int [,]graph)
    {

        /* dist[][] will be the output matrix that will
        finally have the shortest
        distances between every pair of vertices */
        int [,]dist = new int[V,V];
        int i, j, k;

        /* Initialize the solution matrix same as input
        graph matrix. Or we can say the initial values
        of shortest distances are based on shortest
        paths considering no intermediate vertex. */
        for (i = 0; i < V; i++)
            for (j = 0; j < V; j++)
                dist[i,j] = graph[i,j];

        /* Add all vertices one by one to the set of
        intermediate vertices.
        ---> Before start of a iteration, we have shortest
        distances between all pairs of vertices such
        that the shortest distances consider only the
```

```

vertices in set {0, 1, 2, .. k-1} as intermediate
vertices.
----> After the end of a iteration, vertex no. k is
      added to the set of intermediate vertices and
      the set becomes {0, 1, 2, .. k} */
for (k = 0; k < V; k++)
{
    // Pick all vertices as source one by one
    for (i = 0; i < V; i++)
    {

        // Pick all vertices as destination for the
        // above picked source
        for (j = 0; j < V; j++)
        {

            // If vertex k is on the shortest path from
            // i to j, then update the value of dist[i][j]
            if (dist[i,k] + dist[k,j] < dist[i,j])
                dist[i,j] = dist[i,k] + dist[k,j];
        }
    }
}

// If distance of any vertex from itself
// becomes negative, then there is a negative
// weight cycle.
for (i = 0; i < V; i++)
    if (dist[i,i] < 0)
        return true;

return false;
}

// Driver code
public static void Main()
{
    /* Let us create the following weighted graph
       1
       (0)----->(1)
       /|\           |
       |             |
      -1 |           | -1
       |             \|/
       (3)<----->(2)
       -1          */
}

```

```
int[,]graph = { {0, 1, INF, INF},
                {INF, 0, -1, INF},
                {INF, INF, 0, -1},
                {-1, INF, INF, 0}};

if (negCyclefloydWarshall(graph))
    Console.WriteLine("Yes");
else
    Console.WriteLine("No");
}
}
}

// This code is contributed by Sam007.
```

Output:

Yes

Source

<https://www.geeksforgeeks.org/detecting-negative-cycle-using-floyd-warshall/>

Chapter 129

Determine whether a universal sink exists in a directed graph

Determine whether a universal sink exists in a directed graph - GeeksforGeeks

Determine whether a universal sink exists in a directed graph. A universal sink is a vertex which has no edge emanating from it, and all other vertices have an edge towards the sink.

```
Input :  
v1 -> v2 (implies vertex 1 is connected to vertex 2)  
v3 -> v2  
v4 -> v2  
v5 -> v2  
v6 -> v2  
Output :  
Sink found at vertex 2
```

```
Input :  
v1 -> v6  
v2 -> v3  
v2 -> v4  
v4 -> v3  
v5 -> v3  
Output :  
No Sink
```

We try to eliminate $n - 1$ non-sink vertices in $O(n)$ time and check the remaining vertex for the sink property.

To eliminate vertices, we check whether a particular index ($A[i][j]$) in the adjacency matrix is a 1 or a 0. If it is a 0, it means that the vertex corresponding to index j cannot be a sink. If the index is a 1, it means the vertex corresponding to i cannot be a sink. We keep

increasing i and j in this fashion until either i or j exceeds the number of vertices. Using this method allows us to carry out the universal sink test for only one vertex instead of all n vertices. Suppose we are left with only vertex i. We now check for whether row i has only 0s and whether row j as only 1s except for $A[i][i]$, which will be 0.

Illustration :

```
v1 -> v2
v3 -> v2
v4 -> v2
v5 -> v2
v6 -> v2

We can visualize the adjacency matrix for
the above as follows:
0 1 0 0 0 0
0 0 0 0 0 0
0 1 0 0 0 0
0 1 0 0 0 0
0 1 0 0 0 0
```

We observe that vertex 2 does not have any emanating edge, and that every other vertex has an edge in vertex 2. At $A[0][0]$ ($A[i][j]$), we encounter a 0, so we increment j and next look at $A[0][1]$. Here we encounter a 1. So we have to increment i by 1. $A[1][1]$ is 0, so we keep increasing j. We notice that $A[1][2]$, $A[1][3]$.. etc are all 0, so j will exceed the number of vertices (6 in this example). We now check row i and column i for the sink property. Row i must be completely 0, and column i must be completely 1 except for the index $A[i][i]$.

	V1	V2	V3	V4	V5	V6
V1	0 →	1	0	0	0	0
V2	0	0 →	0 →	0 →	0 →	0 →
V3	0	1	0	0	0	0
V4	0	1	0	0	0	0
V5	0	1	0	0	0	0
V6	0	1	0	0	0	0

Adjacency Matrix

Second Example:

v1 -> v6

v2 -> v3

v2 -> v4

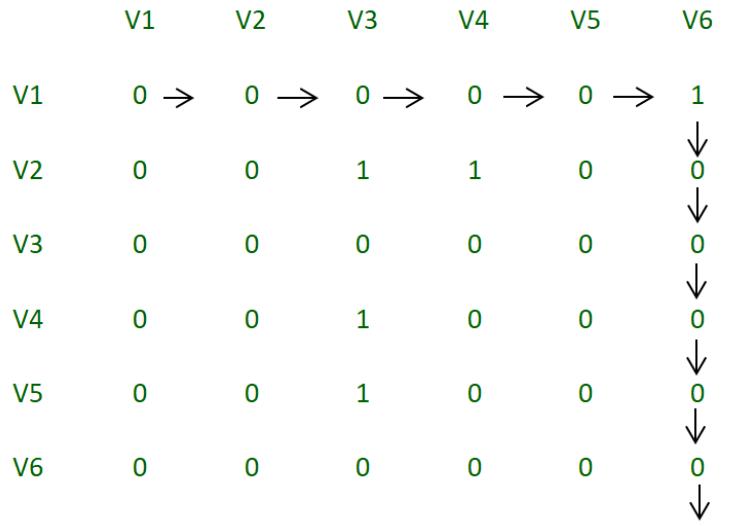
v4 -> v3

v5 -> v3

We can visualize the adjacency matrix
for the above as follows:

```
0 0 0 0 0 1
0 0 1 1 0 0
0 0 0 0 0 0
0 0 1 0 0 0
0 0 1 0 0 0
0 0 0 0 0 0
```

In this example, we observe that in row 1, every element is 0 except for the last column. So we will increment j until we reach the 1. When we reach 1, we increment i as long as the value of A[i][j] is 0. If i exceeds the number of vertices, it is not possible to have a sink, and in this case, i will exceed the number of vertices.



Adjacency Matrix

```
// Java program to find whether a universal sink
// exists in a directed graph
import java.io.*;
import java.util.*;
```

```
class Graph
{
    int vertices;
    int[][] adjacency_matrix;

    // constructor to initialize number of vertices and
    // size of adjacency matrix
    public Graph(int vertices)
    {
        this.vertices = vertices;
        adjacency_matrix = new int[vertices][vertices];
    }

    public void insert(int source, int destination)
    {
        // make adjacency_matrix[i][j] = 1 if there is
        // an edge from i to j
        adjacency_matrix[destination-1] = 1;
    }

    public boolean issink(int i)
    {
        for (int j = 0 ; j < vertices ; j++)
        {
            // if any element in the row i is 1, it means
            // that there is an edge emanating from the
            // vertex, which means it cannot be a sink
            if (adjacency_matrix[i][j] == 1)
                return false;

            // if any element other than i in the column
            // i is 0, it means that there is no edge from
            // that vertex to the vertex we are testing
            // and hence it cannot be a sink
            if (adjacency_matrix[j][i] == 0 && j != i)
                return false;
        }
        //if none of the checks fails, return true
        return true;
    }

    // we will eliminate n-1 non sink vertices so that
    // we have to check for only one vertex instead of
    // all n vertices
    public int eliminate()
    {
        int i = 0, j = 0;
        while (i < vertices && j < vertices)
```

```
{  
    // If the index is 1, increment the row we are  
    // checking by 1  
    // else increment the column  
    if (adjacency_matrix[i][j] == 1)  
        i = i + 1;  
    else  
        j = j + 1;  
  
}  
  
// If i exceeds the number of vertices, it  
// means that there is no valid vertex in  
// the given vertices that can be a sink  
if (i > vertices)  
    return -1;  
else if (!issink(i))  
    return -1;  
else return i;  
}  
}  
  
public class Sink  
{  
    public static void main(String[] args) throws IOException  
{  
        int number_of_vertices = 6;  
        int number_of_edges = 5;  
        graph g = new graph(number_of_vertices);  
        /*  
        //input set 1  
        g.insert(1, 6);  
        g.insert(2, 6);  
        g.insert(3, 6);  
        g.insert(4, 6);  
        g.insert(5, 6);  
        */  
        //input set 2  
        g.insert(1, 6);  
        g.insert(2, 3);  
        g.insert(2, 4);  
        g.insert(4, 3);  
        g.insert(5, 3);  
  
        int vertex = g.eliminate();  
  
        // returns 0 based indexing of vertex. returns  
        // -1 if no sink exists.  
    }  
}
```

```
// returns the vertex number-1 if sink is found
if (vertex >= 0)
    System.out.println("Sink found at vertex "
        + (vertex + 1));
else
    System.out.println("No Sink");
}
}
```

Output:

```
input set 1:
Sink found at vertex 6
input set 2:
No Sink
```

This program eliminates non-sink vertices in $O(n)$ complexity and checks for the sink property in $O(n)$ complexity.

You may also try [The Celebrity Problem](#), which is an application of this concept

Source

<https://www.geeksforgeeks.org/determine-whether-universal-sink-exists-directed-graph/>

Chapter 130

DFS for a n-ary tree (acyclic graph) represented as adjacency list

DFS for a n-ary tree (acyclic graph) represented as adjacency list - GeeksforGeeks

A tree consisting of n nodes is given, we need to print its [DFS](#).

Examples :

Input : Edges of graph

```
1 2
1 3
2 4
3 5
```

Output : 1 2 4 3 5

A simple solution is to do implement standard [DFS](#).

We can modify our approach to avoid extra space for visited nodes. Instead of using the visited array, we can keep track of parent. We traverse all adjacent nodes but the parent.

Below is the implementation :

C++

```
/* CPP code to perform DFS of given tree : */
#include <bits/stdc++.h>
using namespace std;

// DFS on tree
void dfs(vector<int> list[], int node, int arrival)
```

```
{  
    // Printing traversed node  
    cout << node << '\n';  
  
    // Traversing adjacent edges  
    for (int i = 0; i < list[node].size(); i++) {  
  
        // Not traversing the parent node  
        if (list[node][i] != arrival)  
            dfs(list, list[node][i], node);  
    }  
}  
  
int main()  
{  
    // Number of nodes  
    int nodes = 5;  
  
    // Adjacency list  
    vector<int> list[10000];  
  
    // Designing the tree  
    list[1].push_back(2);  
    list[2].push_back(1);  
  
    list[1].push_back(3);  
    list[3].push_back(1);  
  
    list[2].push_back(4);  
    list[4].push_back(2);  
  
    list[3].push_back(5);  
    list[5].push_back(3);  
  
    // Function call  
    dfs(list, 1, 0);  
  
    return 0;  
}
```

Java

```
//JAVA Code For DFS for a n-ary tree (acyclic graph)  
// represented as adjacency list  
import java.util.*;  
  
class GFG {
```

```
// DFS on tree
public static void dfs(LinkedList<Integer> list[],
                      int node, int arrival)
{
    // Printing traversed node
    System.out.println(node);

    // Traversing adjacent edges
    for (int i = 0; i < list[node].size(); i++) {

        // Not traversing the parent node
        if (list[node].get(i) != arrival)
            dfs(list, list[node].get(i), node);
    }
}

/* Driver program to test above function */
public static void main(String[] args)
{

    // Number of nodes
    int nodes = 5;

    // Adjacency list
    LinkedList<Integer> list[] = new LinkedList[nodes+1];

    for (int i = 0; i < list.length; i++){
        list[i] = new LinkedList<Integer>();
    }

    // Designing the tree
    list[1].add(2);
    list[2].add(1);

    list[1].add(3);
    list[3].add(1);

    list[2].add(4);
    list[4].add(2);

    list[3].add(5);
    list[5].add(3);

    // Function call
    dfs(list, 1, 0);
}
```

```
}
```

```
// This code is contributed by Arnav Kr. Mandal.
```

Output:

```
1
2
4
3
5
```

Source

<https://www.geeksforgeeks.org/dfs-n-ary-tree-acyclic-graph-represented-adjacency-list/>

Chapter 131

Dial's Algorithm (Optimized Dijkstra for small range weights)

Dial's Algorithm (Optimized Dijkstra for small range weights) - GeeksforGeeks

Dijkstra's shortest path algorithm runs in $O(E \log V)$ time when implemented with adjacency list representation (See [C implementation](#) and [STL based C++ implementations](#) for details).

```
Input : Source = 0, Maximum Weight W = 14
Output :
    Vertex      Distance from Source
        0                  0
        1                  4
        2                 12
        3                 19
        4                 21
        5                 11
        6                  9
        7                  8
        8                 14
```

Can we optimize Dijkstra's shortest path algorithm to work better than $O(E \log V)$ if maximum weight is small (or range of edge weights is small)?

For example, in the above diagram, maximum weight is 14. Many a times the range of weights on edges in is in small range (i.e. all edge weight can be mapped to 0, 1, 2.. w where w is a small number). In that case, Dijkstra's algorithm can be modified by using different data structure, buckets, which is called dial implementation of dijkstra's algorithm.

time complexity is $\mathbf{O}(E + WV)$ where W is maximum weight on any edge of graph, so we can see that, if W is small then this implementation runs much faster than traditional algorithm. Following are important observations.

- Maximum distance between any two node can be at max $w(V - 1)$ (w is maximum edge weight and we can have at max $V-1$ edges between two vertices).
- In Dijkstra algorithm, distances are finalized in non-decreasing, i.e., distance of the closer (to given source) vertices is finalized before the distant vertices.

Algorithm

Below is complete algorithm:

1. Maintains some buckets, numbered $0, 1, 2, \dots, wV$.
2. Bucket k contains all temporarily labeled nodes with distance equal to k .
3. Nodes in each bucket are represented by list of vertices.
4. Buckets $0, 1, 2, \dots, wV$ are checked sequentially until the first non-empty bucket is found. Each node contained in the first non-empty bucket has the minimum distance label by definition.
5. One by one, these nodes with minimum distance label are permanently labeled and deleted from the bucket during the scanning process.
6. Thus operations involving vertex include:
 - Checking if a bucket is empty
 - Adding a vertex to a bucket
 - Deleting a vertex from a bucket.
7. The position of a temporarily labeled vertex in the buckets is updated accordingly when the distance label of a vertex changes.
8. Process repeated until all vertices are permanently labeled (or distances of all vertices are finalized).

Implementation

Since the maximum distance can be $w(V - 1)$, we create wV buckets (more for simplicity of code) for implementation of algorithm which can be large if w is big.

```
// C++ Program for Dijkstra's dial implementation
#include<bits/stdc++.h>
using namespace std;
# define INF 0x3f3f3f3f

// This class represents a directed graph using
// adjacency list representation
class Graph
{
    int V; // No. of vertices

    // In a weighted graph, we need to store vertex
    // and weight pair for every edge
    list< pair<int, int> > *adj;

public:
    Graph(int V); // Constructor

    // function to add an edge to graph
    void addEdge(int u, int v, int w);

    // prints shortest path from s
    void shortestPath(int s, int W);
};

// Allocates memory for adjacency list
Graph::Graph(int V)
{
    this->V = V;
    adj = new list< pair<int, int> >[V];
}

// adds edge between u and v of weight w
void Graph::addEdge(int u, int v, int w)
{
    adj[u].push_back(make_pair(v, w));
    adj[v].push_back(make_pair(u, w));
}

// Prints shortest paths from src to all other vertices.
// W is the maximum weight of an edge
void Graph::shortestPath(int src, int W)
{
    /* With each distance, iterator to that vertex in
       its bucket is stored so that vertex can be deleted
       in O(1) at time of updation. So
       dist[i].first = distance of ith vertex from src vertex
       dist[i].second = iterator to vertex i in bucket number */
```

```

vector<pair<int, list<int>::iterator> > dist(V);

// Initialize all distances as infinite (INF)
for (int i = 0; i < V; i++)
    dist[i].first = INF;

// Create buckets B[] .
// B[i] keep vertex of distance label i
list<int> B[W * V + 1];

B[0].push_back(src);
dist[src].first = 0;

//
int idx = 0;
while (1)
{
    // Go sequentially through buckets till one non-empty
    // bucket is found
    while (B[idx].size() == 0 && idx < W*V)
        idx++;

    // If all buckets are empty, we are done.
    if (idx == W * V)
        break;

    // Take top vertex from bucket and pop it
    int u = B[idx].front();
    B[idx].pop_front();

    // Process all adjacents of extracted vertex 'u' and
    // update their distanced if required.
    for (auto i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        int v = (*i).first;
        int weight = (*i).second;

        int du = dist[u].first;
        int dv = dist[v].first;

        // If there is shorted path to v through u.
        if (dv > du + weight)
        {
            // If dv is not INF then it must be in B[dv]
            // bucket, so erase its entry using iterator
            // in O(1)
            if (dv != INF)
                B[dv].erase(dist[v].second);
        }
    }
}

```

```
// updating the distance
dist[v].first = du + weight;
dv = dist[v].first;

// pushing vertex v into updated distance's bucket
B[dv].push_front(v);

// storing updated iterator in dist[v].second
dist[v].second = B[dv].begin();
}

}

// Print shortest distances stored in dist[]
printf("Vertex    Distance from Source\n");
for (int i = 0; i < V; ++i)
    printf("%d      %d\n", i, dist[i].first);
}

// Driver program to test methods of graph class
int main()
{
    // create the graph given in above figure
    int V = 9;
    Graph g(V);

    // making above shown graph
    g.addEdge(0, 1, 4);
    g.addEdge(0, 7, 8);
    g.addEdge(1, 2, 8);
    g.addEdge(1, 7, 11);
    g.addEdge(2, 3, 7);
    g.addEdge(2, 8, 2);
    g.addEdge(2, 5, 4);
    g.addEdge(3, 4, 9);
    g.addEdge(3, 5, 14);
    g.addEdge(4, 5, 10);
    g.addEdge(5, 6, 2);
    g.addEdge(6, 7, 1);
    g.addEdge(6, 8, 6);
    g.addEdge(7, 8, 7);

    // maximum weighted edge - 14
    g.shortestPath(0, 14);

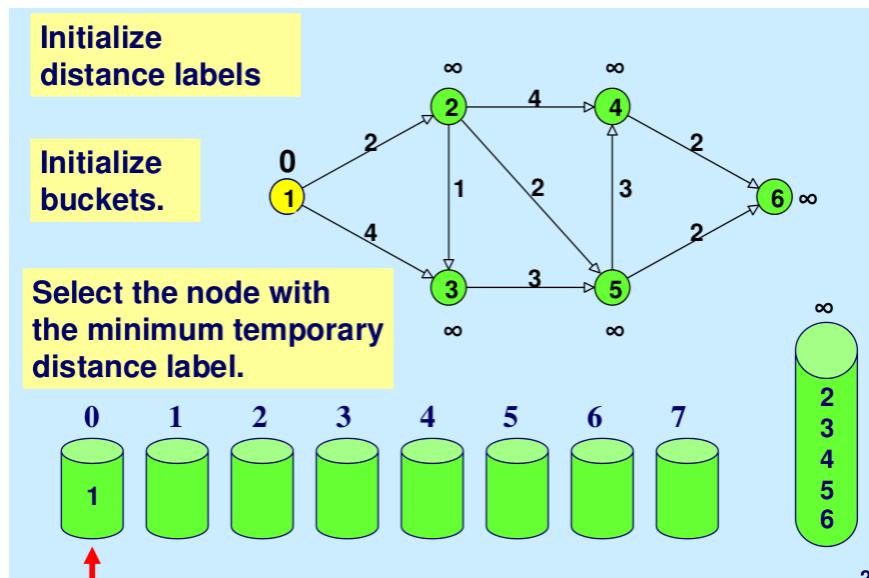
    return 0;
}
```

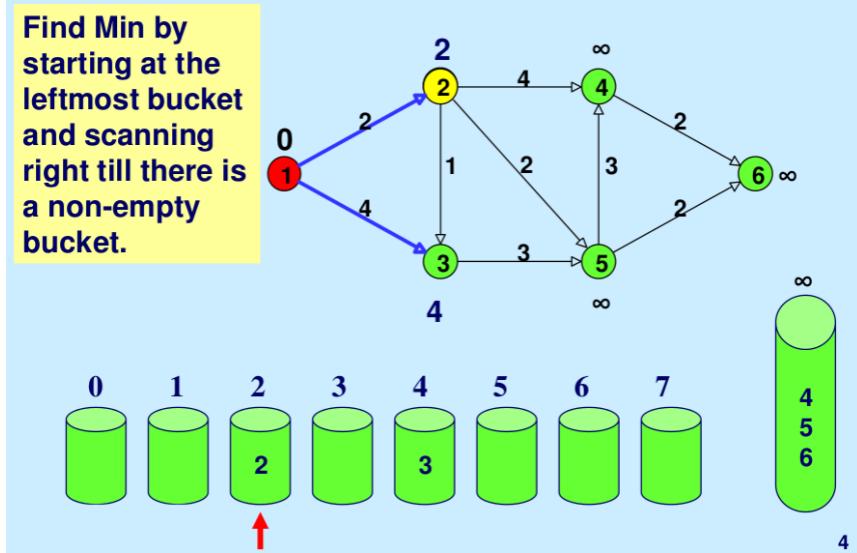
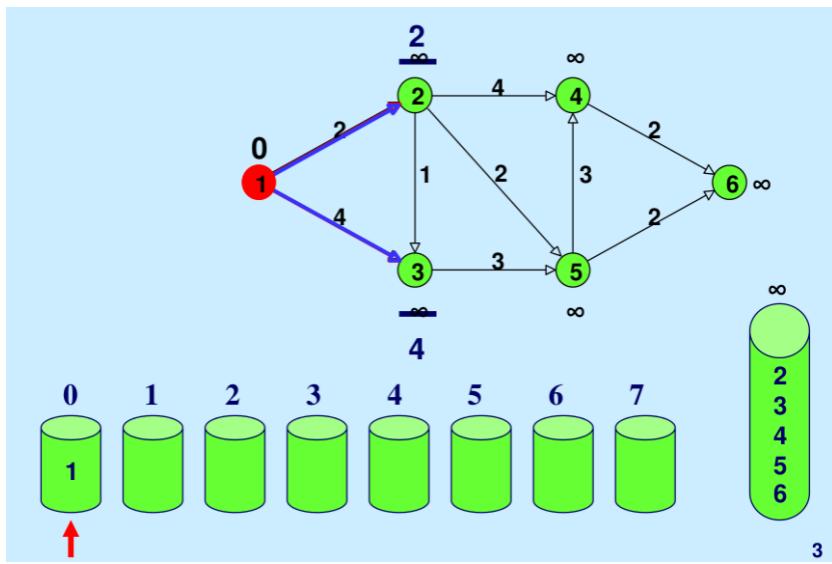
Output:

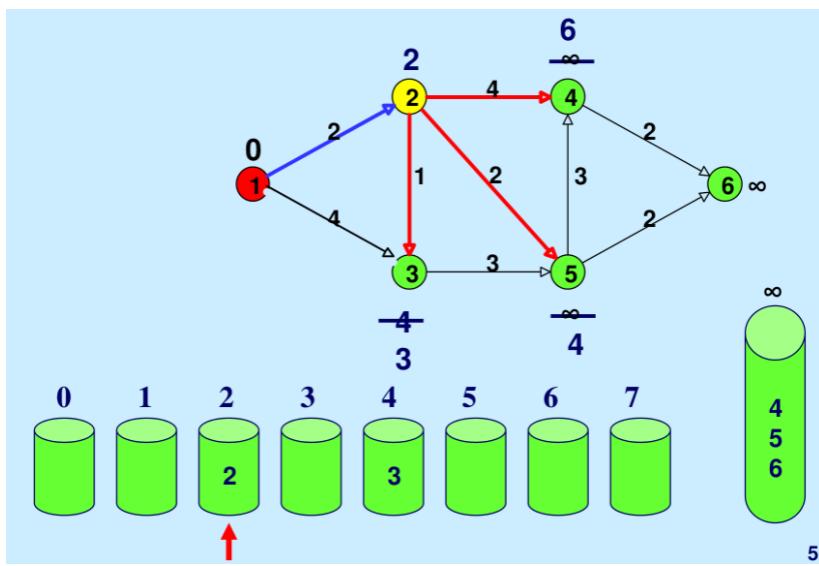
Vertex Distance from Source	
0	0
1	4
2	12
3	19
4	21
5	11
6	9
7	8
8	14

Illustration

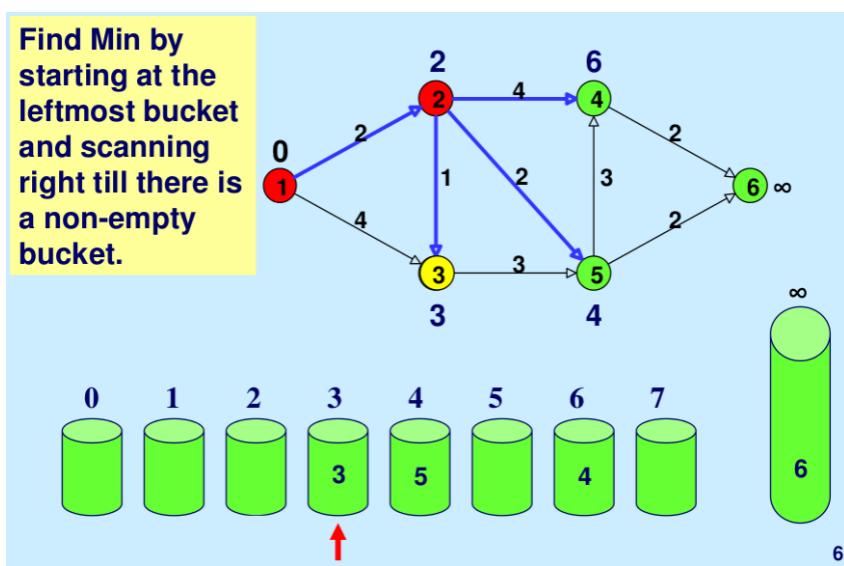
Below is step by step illustration taken from [here](#).

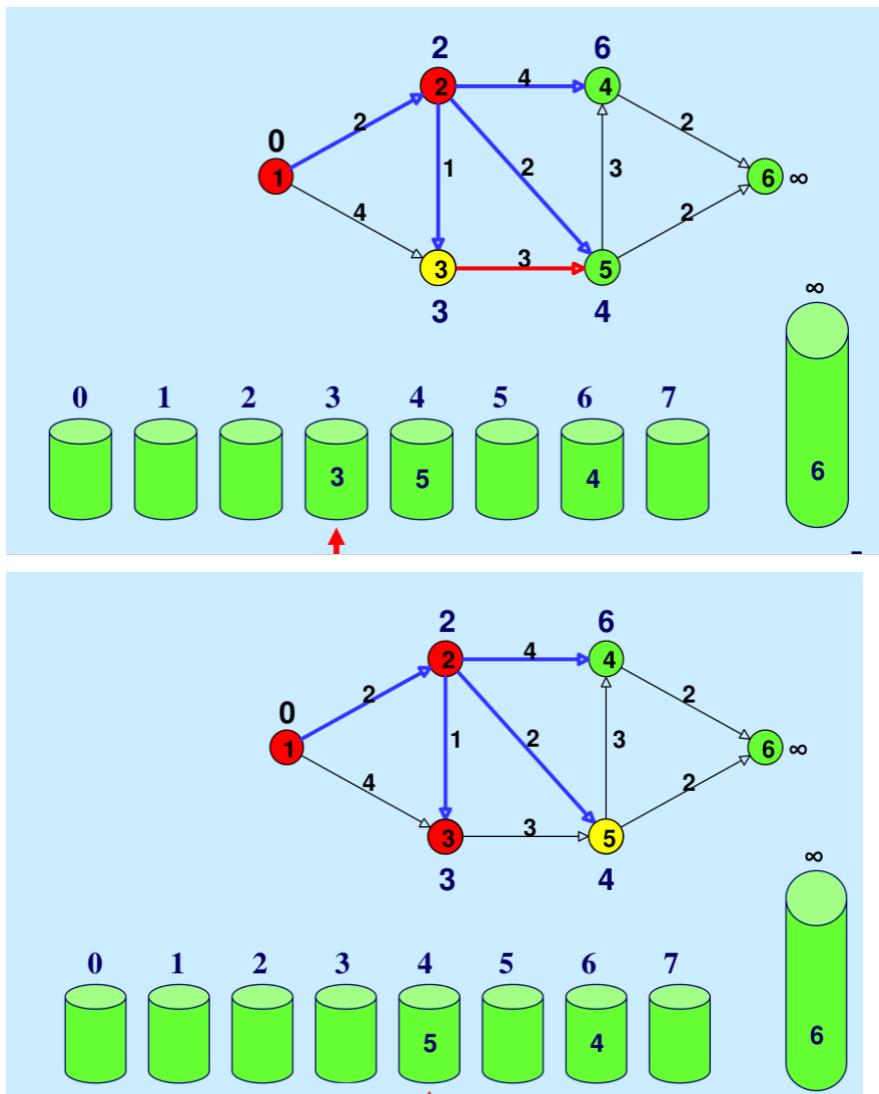


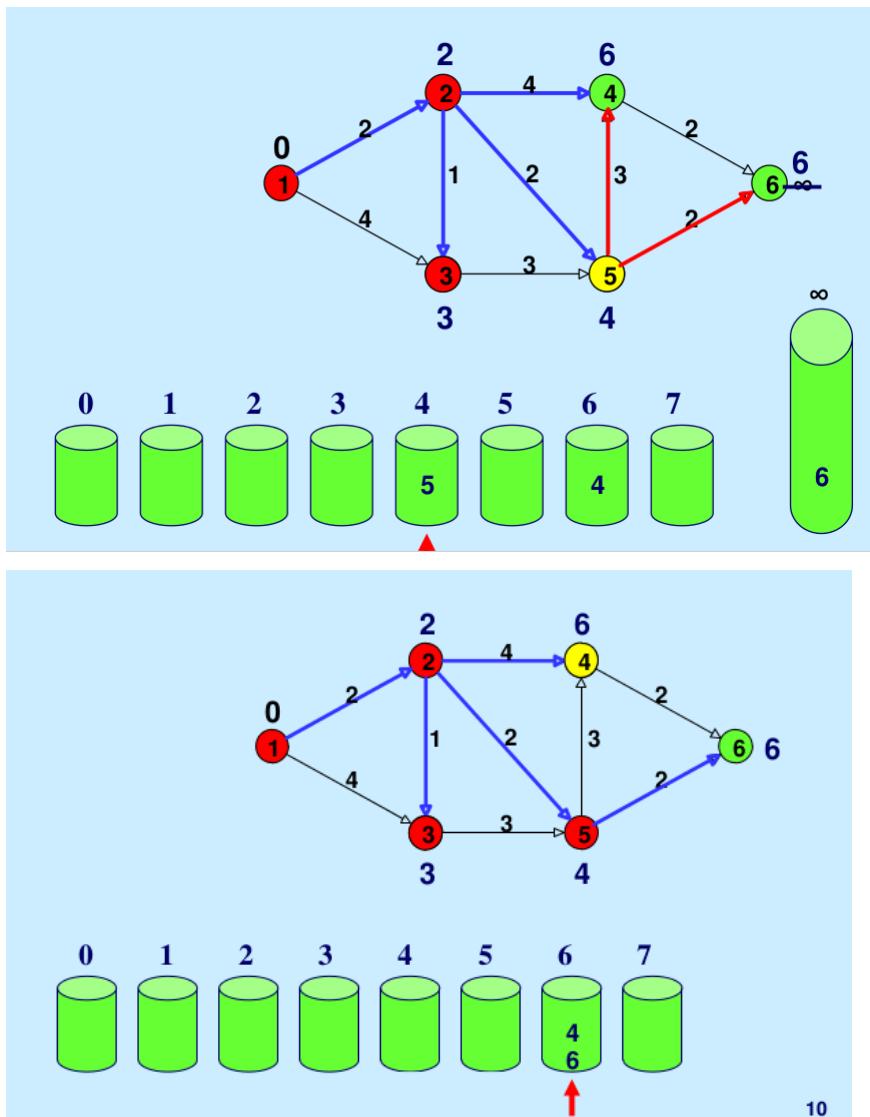


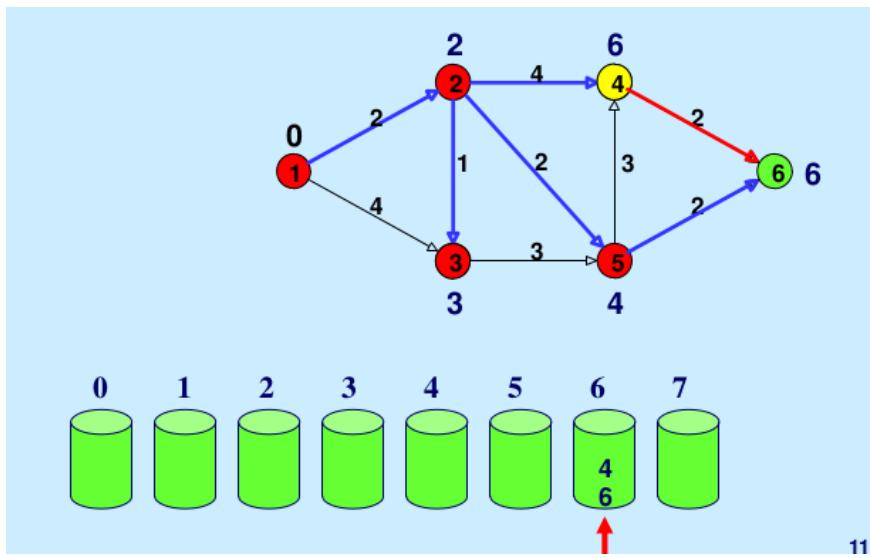


**Find Min by
starting at the
leftmost bucket
and scanning
right till there is
a non-empty
bucket.**

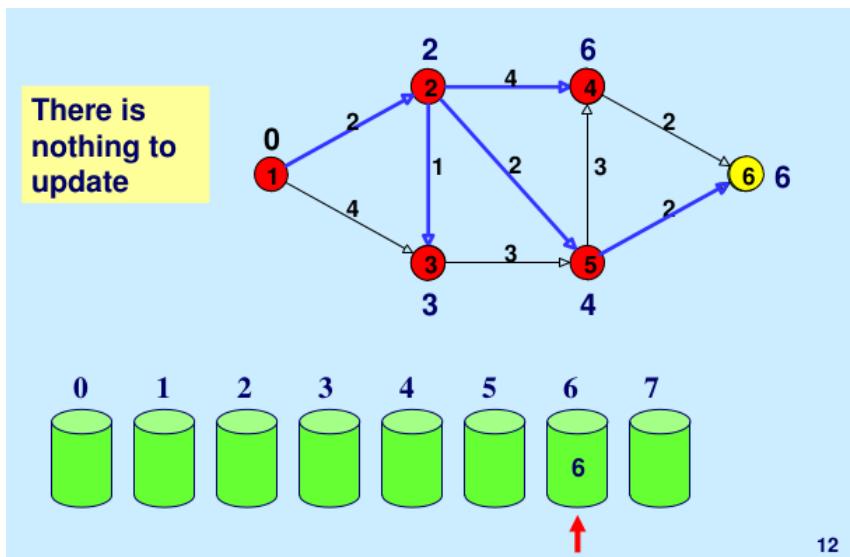




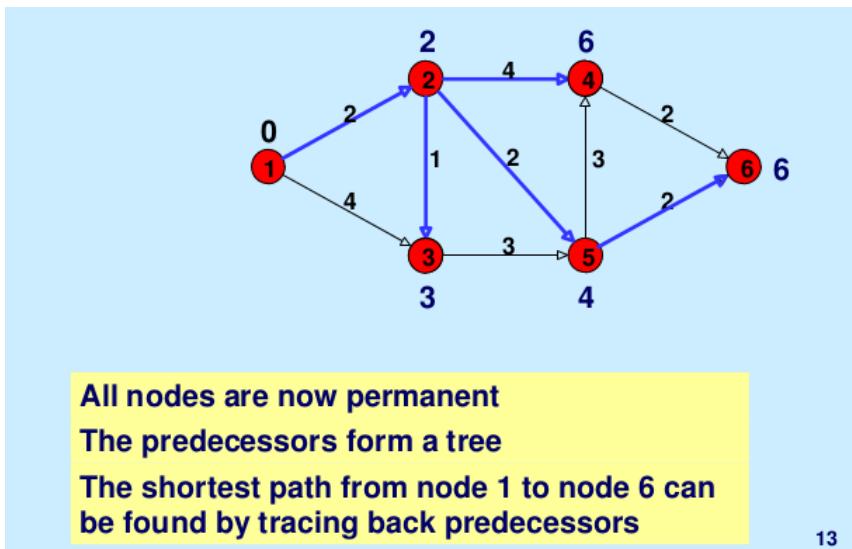




11



12



This article is contributed by Utkarsh Trivedi. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

<https://www.geeksforgeeks.org/dials-algorithm-optimized-dijkstra-for-small-range-weights/>

Chapter 132

Dijkstra's Algorithm for Adjacency List Representation Greedy Algo-8

Dijkstra's Algorithm for Adjacency List Representation Greedy Algo-8 - GeeksforGeeks

We recommend to read following two posts as a prerequisite of this post.

1. [Greedy Algorithms Set 7 \(Dijkstra's shortest path algorithm\)](#)
2. [Graph and its representations](#)

We have discussed [Dijkstra's algorithm and its implementation for adjacency matrix representation of graphs](#). The time complexity for the matrix representation is $O(V^2)$. In this post, $O(E\log V)$ algorithm for adjacency list representation is discussed.

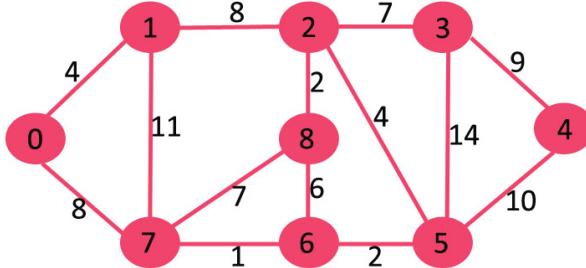
As discussed in the previous post, in Dijkstra's algorithm, two sets are maintained, one set contains list of vertices already included in SPT (Shortest Path Tree), other set contains vertices not yet included. With adjacency list representation, all vertices of a graph can be traversed in $O(V+E)$ time using [BFS](#). The idea is to traverse all vertices of graph using [BFS](#) and use a Min Heap to store the vertices not yet included in SPT (or the vertices for which shortest distance is not finalized yet). Min Heap is used as a priority queue to get the minimum distance vertex from set of not yet included vertices. Time complexity of operations like extract-min and decrease-key value is $O(\log V)$ for Min Heap.

Following are the detailed steps.

- 1) Create a Min Heap of size V where V is the number of vertices in the given graph. Every node of min heap contains vertex number and distance value of the vertex.
- 2) Initialize Min Heap with source vertex as root (the distance value assigned to source vertex is 0). The distance value assigned to all other vertices is INF (infinite).
- 3) While Min Heap is not empty, do following
 -a) Extract the vertex with minimum distance value node from Min Heap. Let the extracted vertex be u .
 -b) For every adjacent vertex v of u , check if v is in Min Heap. If v is in Min Heap and

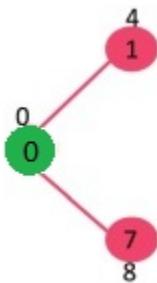
distance value is more than weight of $u-v$ plus distance value of u , then update the distance value of v .

Let us understand with the following example. Let the given source vertex be 0

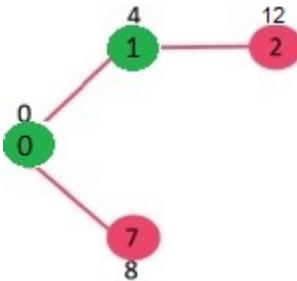


Initially, distance value of source vertex is 0 and INF (infinite) for all other vertices. So source vertex is extracted from Min Heap and distance values of vertices adjacent to 0 (1 and 7) are updated. Min Heap contains all vertices except vertex 0.

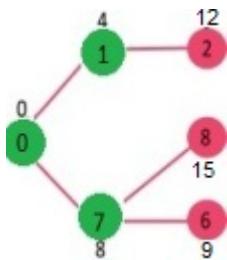
The vertices in green color are the vertices for which minimum distances are finalized and are not in Min Heap



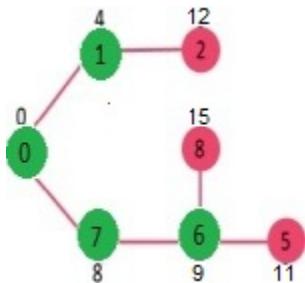
Since distance value of vertex 1 is minimum among all nodes in Min Heap, it is extracted from Min Heap and distance values of vertices adjacent to 1 are updated (distance is updated if the a vertex is not in Min Heap and distance through 1 is shorter than the previous distance). Min Heap contains all vertices except vertex 0 and 1.



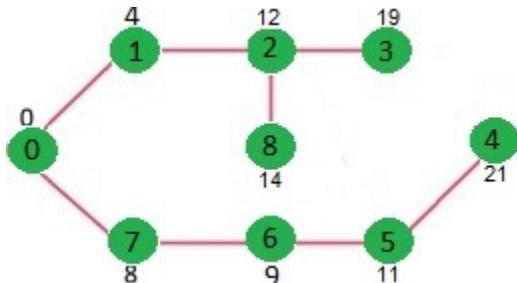
Pick the vertex with minimum distance value from min heap. Vertex 7 is picked. So min heap now contains all vertices except 0, 1 and 7. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



Pick the vertex with minimum distance from min heap. Vertex 6 is picked. So min heap now contains all vertices except 0, 1, 7 and 6. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



Above steps are repeated till min heap doesn't become empty. Finally, we get the following shortest path tree.



C++

```
// C / C++ program for Dijkstra's shortest path algorithm for adjacency
// list representation of graph

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a node in adjacency list
struct AdjListNode
{
    int dest;
    int weight;
    struct AdjListNode* next;
};

// A structure to represent a node in adjacency list
struct AdjList
{
    struct AdjListNode* head;
};
```

```
};

// A structure to represent an adjacency list
struct AdjList
{
    struct AdjListNode *head; // pointer to head node of list
};

// A structure to represent a graph. A graph is an array of adjacency lists.
// Size of array will be V (number of vertices in graph)
struct Graph
{
    int V;
    struct AdjList* array;
};

// A utility function to create a new adjacency list node
struct AdjListNode* newAdjListNode(int dest, int weight)
{
    struct AdjListNode* newNode =
        (struct AdjListNode*) malloc(sizeof(struct AdjListNode));
    newNode->dest = dest;
    newNode->weight = weight;
    newNode->next = NULL;
    return newNode;
}

// A utility function that creates a graph of V vertices
struct Graph* createGraph(int V)
{
    struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
    graph->V = V;

    // Create an array of adjacency lists. Size of array will be V
    graph->array = (struct AdjList*) malloc(V * sizeof(struct AdjList));

    // Initialize each adjacency list as empty by making head as NULL
    for (int i = 0; i < V; ++i)
        graph->array[i].head = NULL;

    return graph;
}

// Adds an edge to an undirected graph
void addEdge(struct Graph* graph, int src, int dest, int weight)
{
    // Add an edge from src to dest. A new node is added to the adjacency
    // list of src. The node is added at the begining
```

```
struct AdjListNode* newNode = newAdjListNode(dest, weight);
newNode->next = graph->array[src].head;
graph->array[src].head = newNode;

// Since graph is undirected, add an edge from dest to src also
newNode = newAdjListNode(src, weight);
newNode->next = graph->array[dest].head;
graph->array[dest].head = newNode;
}

// Structure to represent a min heap node
struct MinHeapNode
{
    int v;
    int dist;
};

// Structure to represent a min heap
struct MinHeap
{
    int size;      // Number of heap nodes present currently
    int capacity; // Capacity of min heap
    int *pos;      // This is needed for decreaseKey()
    struct MinHeapNode **array;
};

// A utility function to create a new Min Heap Node
struct MinHeapNode* newMinHeapNode(int v, int dist)
{
    struct MinHeapNode* minHeapNode =
        (struct MinHeapNode*) malloc(sizeof(struct MinHeapNode));
    minHeapNode->v = v;
    minHeapNode->dist = dist;
    return minHeapNode;
}

// A utility function to create a Min Heap
struct MinHeap* createMinHeap(int capacity)
{
    struct MinHeap* minHeap =
        (struct MinHeap*) malloc(sizeof(struct MinHeap));
    minHeap->pos = (int *)malloc(capacity * sizeof(int));
    minHeap->size = 0;
    minHeap->capacity = capacity;
    minHeap->array =
        (struct MinHeapNode**) malloc(capacity * sizeof(struct MinHeapNode*));
    return minHeap;
}
```

```

// A utility function to swap two nodes of min heap. Needed for min heapify
void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b)
{
    struct MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}

// A standard function to heapify at given idx
// This function also updates position of nodes when they are swapped.
// Position is needed for decreaseKey()
void minHeapify(struct MinHeap* minHeap, int idx)
{
    int smallest, left, right;
    smallest = idx;
    left = 2 * idx + 1;
    right = 2 * idx + 2;

    if (left < minHeap->size &&
        minHeap->array[left]->dist < minHeap->array[smallest]->dist )
        smallest = left;

    if (right < minHeap->size &&
        minHeap->array[right]->dist < minHeap->array[smallest]->dist )
        smallest = right;

    if (smallest != idx)
    {
        // The nodes to be swapped in min heap
        MinHeapNode *smallestNode = minHeap->array[smallest];
        MinHeapNode *idxNode = minHeap->array[idx];

        // Swap positions
        minHeap->pos[smallestNode->v] = idx;
        minHeap->pos[idxNode->v] = smallest;

        // Swap nodes
        swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

        minHeapify(minHeap, smallest);
    }
}

// A utility function to check if the given minHeap is ampty or not
int isEmpty(struct MinHeap* minHeap)
{
    return minHeap->size == 0;
}

```

```

}

// Standard function to extract minimum node from heap
struct MinHeapNode* extractMin(struct MinHeap* minHeap)
{
    if (isEmpty(minHeap))
        return NULL;

    // Store the root node
    struct MinHeapNode* root = minHeap->array[0];

    // Replace root node with last node
    struct MinHeapNode* lastNode = minHeap->array[minHeap->size - 1];
    minHeap->array[0] = lastNode;

    // Update position of last node
    minHeap->pos[root->v] = minHeap->size-1;
    minHeap->pos[lastNode->v] = 0;

    // Reduce heap size and heapify root
    --minHeap->size;
    minHeapify(minHeap, 0);

    return root;
}

// Function to decrease dist value of a given vertex v. This function
// uses pos[] of min heap to get the current index of node in min heap
void decreaseKey(struct MinHeap* minHeap, int v, int dist)
{
    // Get the index of v in heap array
    int i = minHeap->pos[v];

    // Get the node and update its dist value
    minHeap->array[i]->dist = dist;

    // Travel up while the complete tree is not heapified.
    // This is a O(Logn) loop
    while (i && minHeap->array[i]->dist < minHeap->array[(i - 1) / 2]->dist)
    {
        // Swap this node with its parent
        minHeap->pos[minHeap->array[i]->v] = (i-1)/2;
        minHeap->pos[minHeap->array[(i-1)/2]->v] = i;
        swapMinHeapNode(&minHeap->array[i], &minHeap->array[(i - 1) / 2]);

        // move to parent index
        i = (i - 1) / 2;
    }
}

```

```
}

// A utility function to check if a given vertex
// 'v' is in min heap or not
bool isInMinHeap(struct MinHeap *minHeap, int v)
{
    if (minHeap->pos[v] < minHeap->size)
        return true;
    return false;
}

// A utility function used to print the solution
void printArr(int dist[], int n)
{
    printf("Vertex      Distance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

// The main function that calculates distances of shortest paths from src to all
// vertices. It is a O(ELogV) function
void dijkstra(struct Graph* graph, int src)
{
    int V = graph->V;// Get the number of vertices in graph
    int dist[V];      // dist values used to pick minimum weight edge in cut

    // minHeap represents set E
    struct MinHeap* minHeap = createMinHeap(V);

    // Initialize min heap with all vertices. dist value of all vertices
    for (int v = 0; v < V; ++v)
    {
        dist[v] = INT_MAX;
        minHeap->array[v] = newMinHeapNode(v, dist[v]);
        minHeap->pos[v] = v;
    }

    // Make dist value of src vertex as 0 so that it is extracted first
    minHeap->array[src] = newMinHeapNode(src, dist[src]);
    minHeap->pos[src] = src;
    dist[src] = 0;
    decreaseKey(minHeap, src, dist[src]);

    // Initially size of min heap is equal to V
    minHeap->size = V;

    // In the followin loop, min heap contains all nodes
    // whose shortest distance is not yet finalized.
```

```

while (!isEmpty(minHeap))
{
    // Extract the vertex with minimum distance value
    struct MinHeapNode* minHeapNode = extractMin(minHeap);
    int u = minHeapNode->v; // Store the extracted vertex number

    // Traverse through all adjacent vertices of u (the extracted
    // vertex) and update their distance values
    struct AdjListNode* pCrawl = graph->array[u].head;
    while (pCrawl != NULL)
    {
        int v = pCrawl->dest;

        // If shortest distance to v is not finalized yet, and distance to v
        // through u is less than its previously calculated distance
        if (isInMinHeap(minHeap, v) && dist[u] != INT_MAX &&
            pCrawl->weight + dist[u] < dist[v])
        {
            dist[v] = dist[u] + pCrawl->weight;

            // update distance value in min heap also
            decreaseKey(minHeap, v, dist[v]);
        }
        pCrawl = pCrawl->next;
    }
}

// print the calculated shortest distances
printArr(dist, V);
}

// Driver program to test above functions
int main()
{
    // create the graph given in above figure
    int V = 9;
    struct Graph* graph = createGraph(V);
    addEdge(graph, 0, 1, 4);
    addEdge(graph, 0, 7, 8);
    addEdge(graph, 1, 2, 8);
    addEdge(graph, 1, 7, 11);
    addEdge(graph, 2, 3, 7);
    addEdge(graph, 2, 8, 2);
    addEdge(graph, 2, 5, 4);
    addEdge(graph, 3, 4, 9);
    addEdge(graph, 3, 5, 14);
    addEdge(graph, 4, 5, 10);
}

```

```
    addEdge(graph, 5, 6, 2);
    addEdge(graph, 6, 7, 1);
    addEdge(graph, 6, 8, 6);
    addEdge(graph, 7, 8, 7);

    dijkstra(graph, 0);

    return 0;
}
```

Python

```
# A Python program for Dijkstra's shortest
# path algorithm for adjacency
# list representation of graph

from collections import defaultdict
import sys

class Heap():

    def __init__(self):
        self.array = []
        self.size = 0
        self.pos = []

    def newMinHeapNode(self, v, dist):
        minHeapNode = [v, dist]
        return minHeapNode

    # A utility function to swap two nodes
    # of min heap. Needed for min heapify
    def swapMinHeapNode(self, a, b):
        t = self.array[a]
        self.array[a] = self.array[b]
        self.array[b] = t

    # A standard function to heapify at given idx
    # This function also updates position of nodes
    # when they are swapped. Position is needed
    # for decreaseKey()
    def minHeapify(self, idx):
        smallest = idx
        left = 2*idx + 1
        right = 2*idx + 2

        if left < self.size and self.array[left][1] \
           < self.array[smallest][1]:
```

```
smallest = left

if right < self.size and self.array[right][1] \
    < self.array[smallest][1]:
    smallest = right

# The nodes to be swapped in min
# heap if idx is not smallest
if smallest != idx:

    # Swap positions
    self.pos[ self.array[smallest][0] ] = idx
    self.pos[ self.array[idx][0] ] = smallest

    # Swap nodes
    self.swapMinHeapNode(smallest, idx)

    self.minHeapify(smallest)

# Standard function to extract minimum
# node from heap
def extractMin(self):

    # Return NULL wif heap is empty
    if self.isEmpty() == True:
        return

    # Store the root node
    root = self.array[0]

    # Replace root node with last node
    lastNode = self.array[self.size - 1]
    self.array[0] = lastNode

    # Update position of last node
    self.pos[lastNode[0]] = 0
    self.pos[root[0]] = self.size - 1

    # Reduce heap size and heapify root
    self.size -= 1
    self.minHeapify(0)

    return root

def isEmpty(self):
    return True if self.size == 0 else False

def decreaseKey(self, v, dist):
```

```
# Get the index of v in heap array
i = self.pos[v]

# Get the node and update its dist value
self.array[i][1] = dist

# Travel up while the complete tree is
# not hepified. This is a O(Logn) loop
while i > 0 and self.array[i][1] < self.array[(i - 1) / 2][1]:

    # Swap this node with its parent
    self.pos[ self.array[i][0] ] = (i-1)/2
    self.pos[ self.array[(i-1)/2][0] ] = i
    self.swapMinHeapNode(i, (i - 1)/2 )

    # move to parent index
    i = (i - 1) / 2;

# A utility function to check if a given
# vertex 'v' is in min heap or not
def isInMinHeap(self, v):

    if self.pos[v] < self.size:
        return True
    return False

def printArr(dist, n):
    print "Vertex\tDistance from source"
    for i in range(n):
        print "%d\t\t%d" % (i,dist[i])

class Graph():

    def __init__(self, V):
        self.V = V
        self.graph = defaultdict(list)

    # Adds an edge to an undirected graph
    def addEdge(self, src, dest, weight):

        # Add an edge from src to dest. A new node
        # is added to the adjacency list of src. The
        # node is added at the begining. The first
        # element of the node has the destination
```

```

# and the second elements has the weight
newNode = [dest, weight]
self.graph[src].insert(0, newNode)

# Since graph is undirected, add an edge
# from dest to src also
newNode = [src, weight]
self.graph[dest].insert(0, newNode)

# The main function that calculates distances
# of shortest paths from src to all vertices.
# It is a O(ELogV) function
def dijkstra(self, src):

    V = self.V # Get the number of vertices in graph
    dist = [] # dist values used to pick minimum
              # weight edge in cut

    # minHeap represents set E
    minHeap = Heap()

    # Initialize min heap with all vertices.
    # dist value of all vertices
    for v in range(V):
        dist.append(sys.maxint)
        minHeap.array.append( minHeap.newMinHeapNode(v, dist[v]) )
        minHeap.pos.append(v)

    # Make dist value of src vertex as 0 so
    # that it is extracted first
    minHeap.pos[src] = src
    dist[src] = 0
    minHeap.decreaseKey(src, dist[src])

    # Initially size of min heap is equal to V
    minHeap.size = V;

    # In the following loop, min heap contains all nodes
    # whose shortest distance is not yet finalized.
    while minHeap.isEmpty() == False:

        # Extract the vertex with minimum distance value
        newHeapNode = minHeap.extractMin()
        u = newHeapNode[0]

        # Traverse through all adjacent vertices of
        # u (the extracted vertex) and update their
        # distance values

```

```
for pCrawl in self.graph[u]:  
  
    v = pCrawl[0]  
  
    # If shortest distance to v is not finalized  
    # yet, and distance to v through u is less  
    # than its previously calculated distance  
    if minHeap isInMinHeap(v) and dist[u] != sys.maxint and \  
        pCrawl[1] + dist[u] < dist[v]:  
        dist[v] = pCrawl[1] + dist[u]  
  
        # update distance value  
        # in min heap also  
        minHeap.decreaseKey(v, dist[v])  
  
    printArr(dist, V)  
  
# Driver program to test the above functions  
graph = Graph(9)  
graph.addEdge(0, 1, 4)  
graph.addEdge(0, 7, 8)  
graph.addEdge(1, 2, 8)  
graph.addEdge(1, 7, 11)  
graph.addEdge(2, 3, 7)  
graph.addEdge(2, 8, 2)  
graph.addEdge(2, 5, 4)  
graph.addEdge(3, 4, 9)  
graph.addEdge(3, 5, 14)  
graph.addEdge(4, 5, 10)  
graph.addEdge(5, 6, 2)  
graph.addEdge(6, 7, 1)  
graph.addEdge(6, 8, 6)  
graph.addEdge(7, 8, 7)  
graph.dijkstra(0)  
  
# This code is contributed by Divyanshu Mehta
```

Output:

Vertex	Distance from Source
0	0
1	4
2	12
3	19
4	21
5	11
6	9

7	8
8	14

Time Complexity: The time complexity of the above code/algorithm looks $O(V^2)$ as there are two nested while loops. If we take a closer look, we can observe that the statements in inner loop are executed $O(V+E)$ times (similar to BFS). The inner loop has decreaseKey() operation which takes $O(\log V)$ time. So overall time complexity is $O(E+V)*O(\log V)$ which is $O((E+V)*\log V) = O(E\log V)$

Note that the above code uses Binary Heap for Priority Queue implementation. Time complexity can be reduced to $O(E + V\log V)$ using Fibonacci Heap. The reason is, Fibonacci Heap takes $O(1)$ time for decrease-key operation while Binary Heap takes $O(\log n)$ time.

Notes:

- 1) The code calculates shortest distance, but doesn't calculate the path information. We can create a parent array, update the parent array when distance is updated (like [prim's implementation](#)) and use it show the shortest path from source to different vertices.
- 2) The code is for undirected graph, same dijkstra function can be used for directed graphs also.
- 3) The code finds shortest distances from source to all vertices. If we are interested only in shortest distance from source to a single target, we can break the for loop when the picked minimum distance vertex is equal to target (Step 3.a of algorithm).
- 4) Dijkstra's algorithm doesn't work for graphs with negative weight edges. For graphs with negative weight edges, [Bellman–Ford algorithm](#) can be used, we will soon be discussing it as a separate post.

[Printing Paths in Dijkstra's Shortest Path Algorithm](#)

[Dijkstra's shortest path algorithm using set in STL](#)

References:

[Introduction to Algorithms](#) by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.

[Algorithms](#) by Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani

Source

<https://www.geeksforgeeks.org/dijkstras-algorithm-for-adjacency-list-representation-greedy-algo-8/>

Chapter 133

Dijkstra's Shortest Path Algorithm using priority_queue of STL

Dijkstra's Shortest Path Algorithm using priority_queue of STL - GeeksforGeeks

Given a graph and a source vertex in graph, find shortest paths from source to all vertices in the given graph.

```
Input : Source = 0
Output :
    Vertex      Distance from Source
        0                  0
        1                  4
        2                 12
        3                 19
        4                 21
        5                 11
        6                  9
        7                  8
        8                14
```

We have discussed Dijkstra's shortest Path implementations.

- Dijkstra's Algorithm for Adjacency Matrix Representation (In C/C++ with time complexity $O(v^2)$)
- Dijkstra's Algorithm for Adjacency List Representation (In C with Time Complexity $O(E\log V)$)

- Dijkstra's shortest path algorithm using set in STL (In C++ with Time Complexity O(ELogV))

The second implementation is time complexity wise better, but is really complex as we have implemented our own priority queue.

The Third implementation is simpler as it uses STL. The issue with third implementation is, it uses set which in turn uses Self-Balancing Binary Search Trees. For Dijkstra's algorithm, it is always recommended to use heap (or priority queue) as the required operations (extract minimum and decrease key) match with speciality of heap (or priority queue). However, the problem is, priority_queue doesn't support decrease key. To resolve this problem, do not update a key, but insert one more copy of it. So we allow multiple instances of same vertex in priority queue. This approach doesn't require decrease key operation and has below important properties.

- Whenever distance of a vertex is reduced, we add one more instance of vertex in priority_queue. Even if there are multiple instances, we only consider the instance with minimum distance and ignore other instances.
- The time complexity remains O(ELogV)) as there will be at most O(E) vertices in priority queue and O(Log E) is same as O(Log V)

Below is algorithm based on above idea.

- 1) Initialize distances of all vertices as infinite.
- 2) Create an empty priority_queue pq. Every item of pq is a pair (weight, vertex). Weight (or distance) is used as first item of pair as first item is by default used to compare two pairs
- 3) Insert source vertex into pq and make its distance as 0.
- 4) While either pq doesn't become empty
 - a) Extract minimum distance vertex from pq.
Let the extracted vertex be u.
 - b) Loop through all adjacent of u and do following for every vertex v.

```
// If there is a shorter path to v  
// through u.  
If dist[v] > dist[u] + weight(u, v)  
  
(i) Update distance of v, i.e., do  
dist[v] = dist[u] + weight(u, v)
```

- (ii) Insert v into the pq (Even if v is already there)
- 5) Print distance array dist[] to print all shortest paths.

Below is C++ implementation of above idea.

```
// Program to find Dijkstra's shortest path using
// priority_queue in STL
#include<bits/stdc++.h>
using namespace std;
#define INF 0x3f3f3f3f

// iPair ==> Integer Pair
typedef pair<int, int> iPair;

// This class represents a directed graph using
// adjacency list representation
class Graph
{
    int V;      // No. of vertices

    // In a weighted graph, we need to store vertex
    // and weight pair for every edge
    list< pair<int, int> > *adj;

public:
    Graph(int V); // Constructor

    // function to add an edge to graph
    void addEdge(int u, int v, int w);

    // prints shortest path from s
    void shortestPath(int s);
};

// Allocates memory for adjacency list
Graph::Graph(int V)
{
    this->V = V;
    adj = new list<iPair> [V];
}

void Graph::addEdge(int u, int v, int w)
{
    adj[u].push_back(make_pair(v, w));
    adj[v].push_back(make_pair(u, w));
}
```

```

}

// Prints shortest paths from src to all other vertices
void Graph::shortestPath(int src)
{
    // Create a priority queue to store vertices that
    // are being preprocessed. This is weird syntax in C++.
    // Refer below link for details of this syntax
    // https://www.geeksforgeeks.org/implement-min-heap-using-stl/
    priority_queue< iPair, vector <iPair> , greater<iPair> > pq;

    // Create a vector for distances and initialize all
    // distances as infinite (INF)
    vector<int> dist(V, INF);

    // Insert source itself in priority queue and initialize
    // its distance as 0.
    pq.push(make_pair(0, src));
    dist[src] = 0;

    /* Looping till priority queue becomes empty (or all
       distances are not finalized) */
    while (!pq.empty())
    {
        // The first vertex in pair is the minimum distance
        // vertex, extract it from priority queue.
        // vertex label is stored in second of pair (it
        // has to be done this way to keep the vertices
        // sorted distance (distance must be first item
        // in pair)
        int u = pq.top().second;
        pq.pop();

        // 'i' is used to get all adjacent vertices of a vertex
        list< pair<int, int> ::iterator i;
        for (i = adj[u].begin(); i != adj[u].end(); ++i)
        {
            // Get vertex label and weight of current adjacent
            // of u.
            int v = (*i).first;
            int weight = (*i).second;

            // If there is shorted path to v through u.
            if (dist[v] > dist[u] + weight)
            {
                // Updating distance of v
                dist[v] = dist[u] + weight;
                pq.push(make_pair(dist[v], v));
            }
        }
    }
}

```

```
        }
    }
}

// Print shortest distances stored in dist[]
printf("Vertex  Distance from Source\n");
for (int i = 0; i < V; ++i)
    printf("%d \t\t %d\n", i, dist[i]);
}

// Driver program to test methods of graph class
int main()
{
    // create the graph given in above figure
    int V = 9;
    Graph g(V);

    // making above shown graph
    g.addEdge(0, 1, 4);
    g.addEdge(0, 7, 8);
    g.addEdge(1, 2, 8);
    g.addEdge(1, 7, 11);
    g.addEdge(2, 3, 7);
    g.addEdge(2, 8, 2);
    g.addEdge(2, 5, 4);
    g.addEdge(3, 4, 9);
    g.addEdge(3, 5, 14);
    g.addEdge(4, 5, 10);
    g.addEdge(5, 6, 2);
    g.addEdge(6, 7, 1);
    g.addEdge(6, 8, 6);
    g.addEdge(7, 8, 7);

    g.shortestPath(0);

    return 0;
}
```

Output:

Vertex	Distance from Source
0	0
1	4
2	12
3	19

```

4      21
5      11
6      9
7      8
8      14

```

A Quicker Implementation using [vector of pairs representation of weighted graph](#):

```

// Program to find Dijkstra's shortest path using
// priority_queue in STL
#include<bits/stdc++.h>
using namespace std;
#define INF 0x3f3f3f3f

// iPair ==> Integer Pair
typedef pair<int, int> iPair;

// To add an edge
void addEdge(vector <pair<int, int> > adj[], int u,
             int v, int wt)
{
    adj[u].push_back(make_pair(v, wt));
    adj[v].push_back(make_pair(u, wt));
}

// Prints shortest paths from src to all other vertices
void shortestPath(vector<pair<int,int> > adj[], int V, int src)
{
    // Create a priority queue to store vertices that
    // are being preprocessed. This is weird syntax in C++.
    // Refer below link for details of this syntax
    // http://geeksquiz.com/implement-min-heap-using-stl/
    priority_queue< iPair, vector <iPair> , greater<iPair> > pq;

    // Create a vector for distances and initialize all
    // distances as infinite (INF)
    vector<int> dist(V, INF);

    // Insert source itself in priority queue and initialize
    // its distance as 0.
    pq.push(make_pair(0, src));
    dist[src] = 0;

    /* Looping till priority queue becomes empty (or all
    distances are not finalized) */
    while (!pq.empty())

```

```

{
    // The first vertex in pair is the minimum distance
    // vertex, extract it from priority queue.
    // vertex label is stored in second of pair (it
    // has to be done this way to keep the vertices
    // sorted distance (distance must be first item
    // in pair)
    int u = pq.top().second;
    pq.pop();

    // Get all adjacent of u.
    for (auto x : adj[u])
    {
        // Get vertex label and weight of current adjacent
        // of u.
        int v = x.first;
        int weight = x.second;

        // If there is shorted path to v through u.
        if (dist[v] > dist[u] + weight)
        {
            // Updating distance of v
            dist[v] = dist[u] + weight;
            pq.push(make_pair(dist[v], v));
        }
    }
}

// Print shortest distances stored in dist[]
printf("Vertex Distance from Source\n");
for (int i = 0; i < V; ++i)
    printf("%d \t\t %d\n", i, dist[i]);
}

// Driver program to test methods of graph class
int main()
{
    int V = 9;
    vector<iPair> adj[V];

    // making above shown graph
    addEdge(adj, 0, 1, 4);
    addEdge(adj, 0, 7, 8);
    addEdge(adj, 1, 2, 8);
    addEdge(adj, 1, 7, 11);
    addEdge(adj, 2, 3, 7);
    addEdge(adj, 2, 8, 2);
    addEdge(adj, 2, 5, 4);
}

```

```
addEdge(adj, 3, 4, 9);
addEdge(adj, 3, 5, 14);
addEdge(adj, 4, 5, 10);
addEdge(adj, 5, 6, 2);
addEdge(adj, 6, 7, 1);
addEdge(adj, 6, 8, 6);
addEdge(adj, 7, 8, 7);

shortestPath(adj, V, 0);

return 0;
}
```

Output:

Vertex	Distance from Source
0	0
1	4
2	12
3	19
4	21
5	11
6	9
7	8
8	14

Further Optimization

We can use a flag array to store what all vertices have been extracted from priority queue. This way we can avoid updating weights of items that have already been extracted. Please see [this](#) for optimized implementation.

This article is contributed by **Shubham Agrawal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-using-priority_queue-stl/

Chapter 134

Dijkstra's shortest path algorithm using set in STL

Dijkstra's shortest path algorithm using set in STL - GeeksforGeeks

Given a graph and a source vertex in graph, find shortest paths from source to all vertices in the given graph.

```
Input : Source = 0
Output :
    Vertex      Distance from Source
        0                  0
        1                  4
        2                 12
        3                 19
        4                 21
        5                 11
        6                  9
        7                  8
        8                14
```

We have discussed Dijkstra's shortest Path implementations.

- Dijkstra's Algorithm for Adjacency Matrix Representation (In C/C++ with time complexity $O(v^2)$)
- Dijkstra's Algorithm for Adjacency List Representation (In C with Time Complexity $O(E\log V)$)

The second implementation is time complexity wise better, but is really complex as we have implemented our own priority queue. STL provides [priority_queue](#), but the provided priority queue doesn't support decrease key and delete operations. And in Dijkstra's algorithm, we need a priority queue and below operations on priority queue :

- ExtractMin : from all those vertices whose shortest distance is not yet found, we need to get vertex with minimum distance.
- DecreaseKey : After extracting vertex we need to update distance of its adjacent vertices, and if new distance is smaller, then update that in data structure.

Above operations can be easily implemented by [set data structure of c++ STL](#), set keeps all its keys in sorted order so minimum distant vertex will always be at beginning, we can extract it from there, which is the ExtractMin operation and update other adjacent vertex accordingly if any vertex's distance become smaller then delete its previous entry and insert new updated entry which is DecreaseKey operation.

Below is algorithm based on set data structure.

- 1) Initialize distances of all vertices as infinite.
- 2) Create an empty set. Every item of set is a pair (weight, vertex). Weight (or distance) is used as first item of pair as first item is by default used to compare two pairs.
- 3) Insert source vertex into the set and make its distance as 0.
- 4) While Set doesn't become empty, do following
 - a) Extract minimum distance vertex from Set.
Let the extracted vertex be u.
 - b) Loop through all adjacent of u and do following for every vertex v.


```
// If there is a shorter path to v
// through u.
If dist[v] > dist[u] + weight(u, v)

(i) Update distance of v, i.e., do
    dist[v] = dist[u] + weight(u, v)
(ii) If v is in set, update its distance
    in set by removing it first, then
    inserting with new distance
(iii) If v is not in set, then insert
    it in set with new distance
```

- 5) Print distance array dist[] to print all shortest paths.

Below is C++ implementation of above idea.

```
// Program to find Dijkstra's shortest path using STL set
#include<bits/stdc++.h>
using namespace std;
#define INF 0x3f3f3f3f

// This class represents a directed graph using
// adjacency list representation
class Graph
{
    int V;      // No. of vertices

    // In a weighted graph, we need to store vertex
    // and weight pair for every edge
    list< pair<int, int> > *adj;

public:
    Graph(int V); // Constructor

    // function to add an edge to graph
    void addEdge(int u, int v, int w);

    // prints shortest path from s
    void shortestPath(int s);
};

// Allocates memory for adjacency list
Graph::Graph(int V)
{
    this->V = V;
    adj = new list< pair<int, int> >[V];
}

void Graph::addEdge(int u, int v, int w)
{
    adj[u].push_back(make_pair(v, w));
    adj[v].push_back(make_pair(u, w));
}

// Prints shortest paths from src to all other vertices
void Graph::shortestPath(int src)
{
    // Create a set to store vertices that are being
    // processed
```

```

set< pair<int, int> > setds;

// Create a vector for distances and initialize all
// distances as infinite (INF)
vector<int> dist(V, INF);

// Insert source itself in Set and initialize its
// distance as 0.
setds.insert(make_pair(0, src));
dist[src] = 0;

/* Looping till all shortest distance are finalized
   then setds will become empty */
while (!setds.empty())
{
    // The first vertex in Set is the minimum distance
    // vertex, extract it from set.
    pair<int, int> tmp = *(setds.begin());
    setds.erase(setds.begin());

    // vertex label is stored in second of pair (it
    // has to be done this way to keep the vertices
    // sorted distance (distance must be first item
    // in pair)
    int u = tmp.second;

    // 'i' is used to get all adjacent vertices of a vertex
    list< pair<int, int> >::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        // Get vertex label and weight of current adjacent
        // of u.
        int v = (*i).first;
        int weight = (*i).second;

        // If there is shorter path to v through u.
        if (dist[v] > dist[u] + weight)
        {
            /* If distance of v is not INF then it must be in
               our set, so removing it and inserting again
               with updated less distance.
               Note : We extract only those vertices from Set
               for which distance is finalized. So for them,
               we would never reach here. */
            if (dist[v] != INF)
                setds.erase(setds.find(make_pair(dist[v], v)));
        }
        // Updating distance of v
    }
}

```

```
        dist[v] = dist[u] + weight;
        setds.insert(make_pair(dist[v], v));
    }
}

// Print shortest distances stored in dist[]
printf("Vertex  Distance from Source\n");
for (int i = 0; i < V; ++i)
    printf("%d \t\t %d\n", i, dist[i]);
}

// Driver program to test methods of graph class
int main()
{
    // create the graph given in above figure
    int V = 9;
    Graph g(V);

    // making above shown graph
    g.addEdge(0, 1, 4);
    g.addEdge(0, 7, 8);
    g.addEdge(1, 2, 8);
    g.addEdge(1, 7, 11);
    g.addEdge(2, 3, 7);
    g.addEdge(2, 8, 2);
    g.addEdge(2, 5, 4);
    g.addEdge(3, 4, 9);
    g.addEdge(3, 5, 14);
    g.addEdge(4, 5, 10);
    g.addEdge(5, 6, 2);
    g.addEdge(6, 7, 1);
    g.addEdge(6, 8, 6);
    g.addEdge(7, 8, 7);

    g.shortestPath(0);

    return 0;
}
```

Output :

Vertex	Distance from Source
0	0
1	4
2	12
3	19

4	21
5	11
6	9
7	8
8	14

Time Complexity : Set in C++ are typically implemented using Self-balancing binary search trees. Therefore, time complexity of set operations like insert, delete is logarithmic and time complexity of above solution is $O(E\log V)$.

Dijkstra's Shortest Path Algorithm using priority_queue of STL

This article is contributed by [Utkarsh Trivedi](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

<https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-using-set-in-stl/>

Chapter 135

Dijkstra's shortest path algorithm Greedy Algo-7

Dijkstra's algorithm

Given a graph and a source vertex in the graph, find shortest paths from source to all vertices in the given graph.

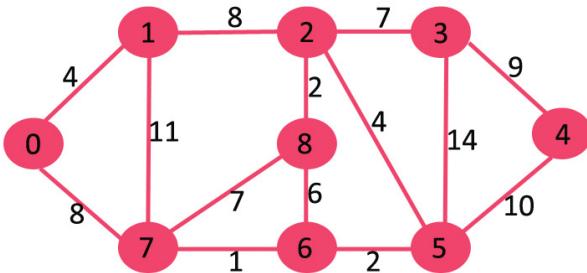
Dijkstra's algorithm is very similar to [Prim's algorithm for minimum spanning tree](#). Like Prim's MST, we generate a *SPT (shortest path tree)* with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

Below are the detailed steps used in Dijkstra's algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

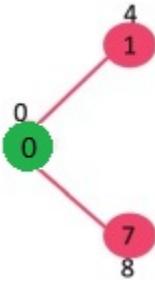
Algorithm

- 1) Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.
- 2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
- 3) While *sptSet* doesn't include all vertices
 -a) Pick a vertex *u* which is not there in *sptSet* and has minimum distance value.
 -b) Include *u* to *sptSet*.
 -c) Update distance value of all adjacent vertices of *u*. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex *v*, if sum of distance value of *u* (from source) and weight of edge *u-v*, is less than the distance value of *v*, then update the distance value of *v*.

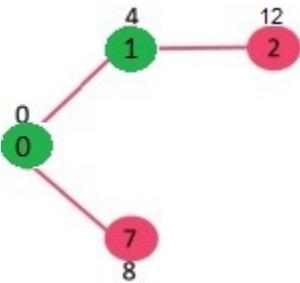
Let us understand with the following example:



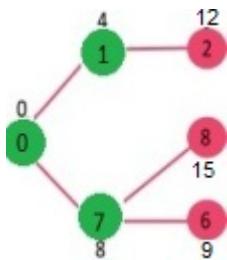
The set *sptSet* is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.



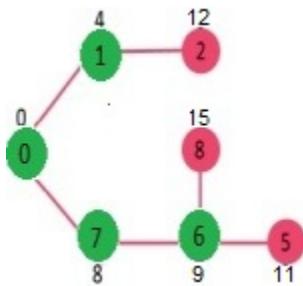
Pick the vertex with minimum distance value and not already included in SPT (not in *sptSET*). The vertex 1 is picked and added to *sptSet*. So *sptSet* now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



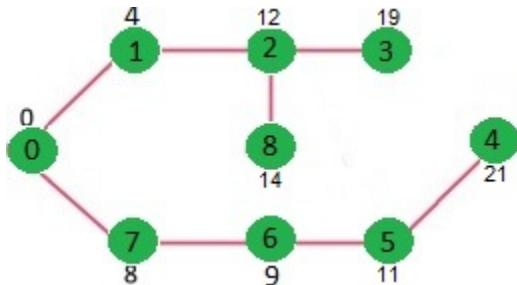
Pick the vertex with minimum distance value and not already included in SPT (not in *sptSET*). Vertex 7 is picked. So *sptSet* now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



Pick the vertex with minimum distance value and not already included in SPT (not in sptSet). Vertex 6 is picked. So sptSet now becomes $\{0, 1, 7, 6\}$. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



We repeat the above steps until *sptSet* doesn't include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).



How to implement the above algorithm?

We use a boolean array *sptSet[]* to represent the set of vertices included in SPT. If a value *sptSet[v]* is true, then vertex *v* is included in SPT, otherwise not. Array *dist[]* is used to store shortest distance values of all vertices.

C++

```
// A C++ program for Dijkstra's single source shortest path algorithm.
// The program is for adjacency matrix representation of the graph

#include <stdio.h>
#include <limits.h>

// Number of vertices in the graph
```

```
#define V 9

// A utility function to find the vertex with minimum distance value, from
// the set of vertices not yet included in shortest path tree
int minDistance(int dist[], bool sptSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min)
            min = dist[v], min_index = v;

    return min_index;
}

// A utility function to print the constructed distance array
int printSolution(int dist[], int n)
{
    printf("Vertex      Distance from Source\n");
    for (int i = 0; i < V; i++)
        printf("%d \t %d\n", i, dist[i]);
}

// Function that implements Dijkstra's single source shortest path algorithm
// for a graph represented using adjacency matrix representation
void dijkstra(int graph[V][V], int src)
{
    int dist[V];      // The output array. dist[i] will hold the shortest
                      // distance from src to i

    bool sptSet[V]; // sptSet[i] will true if vertex i is included in shortest
                    // path tree or shortest distance from src to i is finalized

    // Initialize all distances as INFINITE and stpSet[] as false
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;

    // Distance of source vertex from itself is always 0
    dist[src] = 0;

    // Find shortest path for all vertices
    for (int count = 0; count < V-1; count++)
    {
        // Pick the minimum distance vertex from the set of vertices not
        // yet processed. u is always equal to src in the first iteration.
        int u = minDistance(dist, sptSet);
```

```
// Mark the picked vertex as processed
sptSet[u] = true;

// Update dist value of the adjacent vertices of the picked vertex.
for (int v = 0; v < V; v++)

    // Update dist[v] only if is not in sptSet, there is an edge from
    // u to v, and total weight of path from src to v through u is
    // smaller than current value of dist[v]
    if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
        && dist[u]+graph[u][v] < dist[v])
        dist[v] = dist[u] + graph[u][v];
    }

    // print the constructed distance array
    printSolution(dist, V);
}

// driver program to test above function
int main()
{
    /* Let us create the example graph discussed above */
    int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},
                        {4, 0, 8, 0, 0, 0, 0, 11, 0},
                        {0, 8, 0, 7, 0, 4, 0, 0, 2},
                        {0, 0, 7, 0, 9, 14, 0, 0, 0},
                        {0, 0, 0, 9, 0, 10, 0, 0, 0},
                        {0, 0, 4, 14, 10, 0, 2, 0, 0},
                        {0, 0, 0, 0, 2, 0, 1, 6},
                        {8, 11, 0, 0, 0, 0, 1, 0, 7},
                        {0, 0, 2, 0, 0, 0, 6, 7, 0}
                    };

    dijkstra(graph, 0);

    return 0;
}
```

Java

```
// A Java program for Dijkstra's single source shortest path algorithm.
// The program is for adjacency matrix representation of the graph
import java.util.*;
import java.lang.*;
import java.io.*;

class ShortestPath
{
```

```
// A utility function to find the vertex with minimum distance value,
// from the set of vertices not yet included in shortest path tree
static final int V=9;
int minDistance(int dist[], Boolean sptSet[])
{
    // Initialize min value
    int min = Integer.MAX_VALUE, min_index=-1;

    for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min)
        {
            min = dist[v];
            min_index = v;
        }

    return min_index;
}

// A utility function to print the constructed distance array
void printSolution(int dist[], int n)
{
    System.out.println("Vertex   Distance from Source");
    for (int i = 0; i < V; i++)
        System.out.println(i+" tt "+dist[i]);
}

// Function that implements Dijkstra's single source shortest path
// algorithm for a graph represented using adjacency matrix
// representation
void dijkstra(int graph[][] , int src)
{
    int dist[] = new int[V]; // The output array. dist[i] will hold
                           // the shortest distance from src to i

    // sptSet[i] will true if vertex i is included in shortest
    // path tree or shortest distance from src to i is finalized
    Boolean sptSet[] = new Boolean[V];

    // Initialize all distances as INFINITE and stpSet[] as false
    for (int i = 0; i < V; i++)
    {
        dist[i] = Integer.MAX_VALUE;
        sptSet[i] = false;
    }

    // Distance of source vertex from itself is always 0
    dist[src] = 0;
```

```

// Find shortest path for all vertices
for (int count = 0; count < V-1; count++)
{
    // Pick the minimum distance vertex from the set of vertices
    // not yet processed. u is always equal to src in first
    // iteration.
    int u = minDistance(dist, sptSet);

    // Mark the picked vertex as processed
    sptSet[u] = true;

    // Update dist value of the adjacent vertices of the
    // picked vertex.
    for (int v = 0; v < V; v++)

        // Update dist[v] only if is not in sptSet, there is an
        // edge from u to v, and total weight of path from src to
        // v through u is smaller than current value of dist[v]
        if (!sptSet[v] && graph[u][v]!=0 &&
            dist[u] != Integer.MAX_VALUE &&
            dist[u]+graph[u][v] < dist[v])
            dist[v] = dist[u] + graph[u][v];
}

// print the constructed distance array
printSolution(dist, V);
}

// Driver method
public static void main (String[] args)
{
    /* Let us create the example graph discussed above */
    int graph[][] = new int[][]{{0, 4, 0, 0, 0, 0, 0, 8, 0},
                                {4, 0, 8, 0, 0, 0, 0, 11, 0},
                                {0, 8, 0, 7, 0, 4, 0, 0, 2},
                                {0, 0, 7, 0, 9, 14, 0, 0, 0},
                                {0, 0, 0, 9, 0, 10, 0, 0, 0},
                                {0, 0, 4, 14, 10, 0, 2, 0, 0},
                                {0, 0, 0, 0, 2, 0, 1, 6},
                                {8, 11, 0, 0, 0, 0, 1, 0, 7},
                                {0, 0, 2, 0, 0, 0, 6, 7, 0}
                            };
    ShortestPath t = new ShortestPath();
    t.dijkstra(graph, 0);
}
}

//This code is contributed by Aakash Hasija

```

Python

```
# Python program for Dijkstra's single
# source shortest path algorithm. The program is
# for adjacency matrix representation of the graph

# Library for INT_MAX
import sys

class Graph():

    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]

    def printSolution(self, dist):
        print "Vertex tDistance from Source"
        for node in range(self.V):
            print node,"t",dist[node]

    # A utility function to find the vertex with
    # minimum distance value, from the set of vertices
    # not yet included in shortest path tree
    def minDistance(self, dist, sptSet):

        # Initilaize minimum distance for next node
        min = sys.maxint

        # Search not nearest vertex not in the
        # shortest path tree
        for v in range(self.V):
            if dist[v] < min and sptSet[v] == False:
                min = dist[v]
                min_index = v

        return min_index

    # Function that implements Dijkstra's single source
    # shortest path algorithm for a graph represented
    # using adjacency matrix representation
    def dijkstra(self, src):

        dist = [sys.maxint] * self.V
        dist[src] = 0
        sptSet = [False] * self.V

        for cout in range(self.V):
```

```

# Pick the minimum distance vertex from
# the set of vertices not yet processed.
# u is always equal to src in first iteration
u = self.minDistance(dist, sptSet)

# Put the minimum distance vertex in the
# shortest path tree
sptSet[u] = True

# Update dist value of the adjacent vertices
# of the picked vertex only if the current
# distance is greater than new distance and
# the vertex is not in the shortest path tree
for v in range(self.V):
    if self.graph[u][v] > 0 and sptSet[v] == False and
       dist[v] > dist[u] + self.graph[u][v]:
        dist[v] = dist[u] + self.graph[u][v]

self.printSolution(dist)

# Driver program
g = Graph(9)
g.graph = [[0, 4, 0, 0, 0, 0, 0, 8, 0],
            [4, 0, 8, 0, 0, 0, 0, 11, 0],
            [0, 8, 0, 7, 0, 4, 0, 0, 2],
            [0, 0, 7, 0, 9, 14, 0, 0, 0],
            [0, 0, 0, 9, 0, 10, 0, 0, 0],
            [0, 0, 4, 14, 10, 0, 2, 0, 0],
            [0, 0, 0, 0, 2, 0, 1, 6],
            [8, 11, 0, 0, 0, 0, 1, 0, 7],
            [0, 0, 2, 0, 0, 0, 6, 7, 0]
        ];
g.dijkstra(0);

# This code is contributed by Divyanshu Mehta

```

Output:

Vertex	Distance from Source
0	0
1	4
2	12
3	19
4	21
5	11
6	9

7	8
8	14

Notes:

- 1) The code calculates shortest distance, but doesn't calculate the path information. We can create a parent array, update the parent array when distance is updated (like [prim's implementation](#)) and use it show the shortest path from source to different vertices.
- 2) The code is for undirected graph, same dijkstra function can be used for directed graphs also.
- 3) The code finds shortest distances from source to all vertices. If we are interested only in shortest distance from the source to a single target, we can break the for the loop when the picked minimum distance vertex is equal to target (Step 3.a of the algorithm).
- 4) Time Complexity of the implementation is $O(V^2)$. If the input [graph is represented using adjacency list](#), it can be reduced to $O(E \log V)$ with the help of binary heap. Please see [Dijkstra's Algorithm for Adjacency List Representation](#) for more details.
- 5) Dijkstra's algorithm doesn't work for graphs with negative weight edges. For graphs with negative weight edges, [Bellman–Ford algorithm](#) can be used, we will soon be discussing it as a separate post.

[Dijkstra's Algorithm for Adjacency List Representation](#)

[Printing Paths in Dijkstra's Shortest Path Algorithm](#)

[Dijkstra's shortest path algorithm using set in STL](#)

Source

<https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/>

Chapter 136

Dinic's algorithm for Maximum Flow

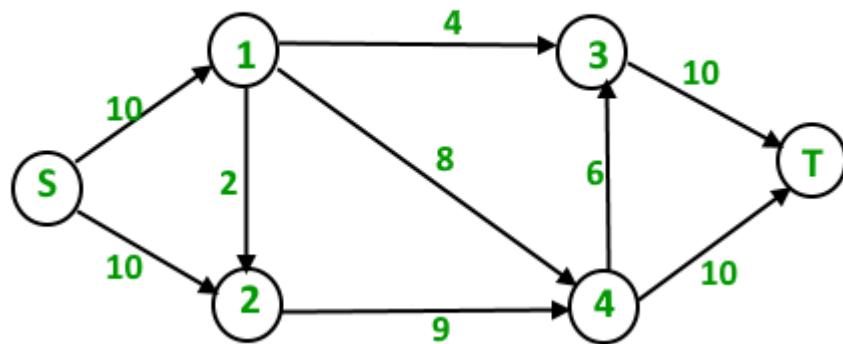
Dinic's algorithm for Maximum Flow - GeeksforGeeks

Problem Statement :

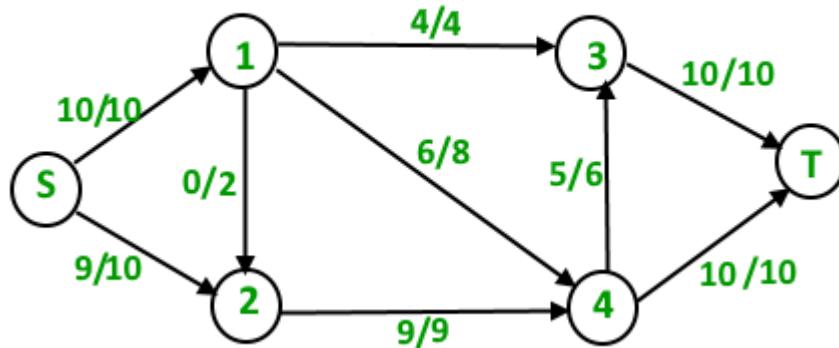
Given a graph which represents a flow network where every edge has a capacity. Also given two vertices source 's' and sink 't' in the graph, find the maximum possible flow from s to t with following constraints :

1. Flow on an edge doesn't exceed the given capacity of the edge.
2. Incoming flow is equal to outgoing flow for every vertex except s and t.

For example, in following input graph,



the maximum s-t flow is 19 which is shown below.



Background :

1. [Max Flow Problem Introduction](#) : We introduced Maximum Flow problem, discussed Greedy Algorithm and introduced residual graph.
2. [Ford-Fulkerson Algorithm and Edmond Karp Implementation](#) : We discussed Ford-Fulkerson algorithm and its implementation. We also discussed residual graph in detail.

Time complexity of [Edmond Karp Implementation](#) is $O(VE^2)$. In this post, a new Dinic's algorithm is discussed which is a faster algorithm and takes $O(EV^2)$.

Like Edmond Karp's algorithm, Dinic's algorithm uses following concepts :

1. A flow is maximum if there is no s to t path in residual graph.
2. BFS is used in a loop. There is a difference though in the way we use BFS in both algorithms.

In Edmond's Karp algorithm, we use BFS to find an augmenting path and send flow across this path. In Dinic's algorithm, we use BFS to check if more flow is possible and to construct level graph. In **level graph**, we assign levels to all nodes, level of a node is shortest distance (in terms of number of edges) of the node from source. Once level graph is constructed, we send multiple flows using this level graph. This is the reason it works better than Edmond Karp. In Edmond Karp, we send only flow that is send across the path found by BFS.

Outline of Dinic's algorithm :

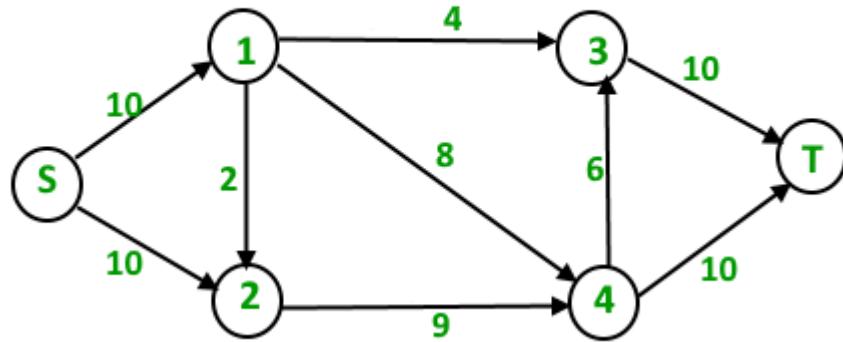
- 1) Initialize residual graph G as given graph.
- 1) Do BFS of G to construct a level graph (or assign levels to vertices) and also check if more flow is possible.

- a) If more flow is not possible, then return.
- b) Send multiple flows in G using level graph until blocking flow is reached. Here using level graph means, in every flow, levels of path nodes should be 0, 1, 2... (in order) from s to t.

A flow is **Blocking Flow** if no more flow can be sent using level graph, i.e., no more s-t path exists such that path vertices have current levels 0, 1, 2... in order. Blocking Flow can be seen same as maximum flow path in Greedy algorithm discussed [here](#).

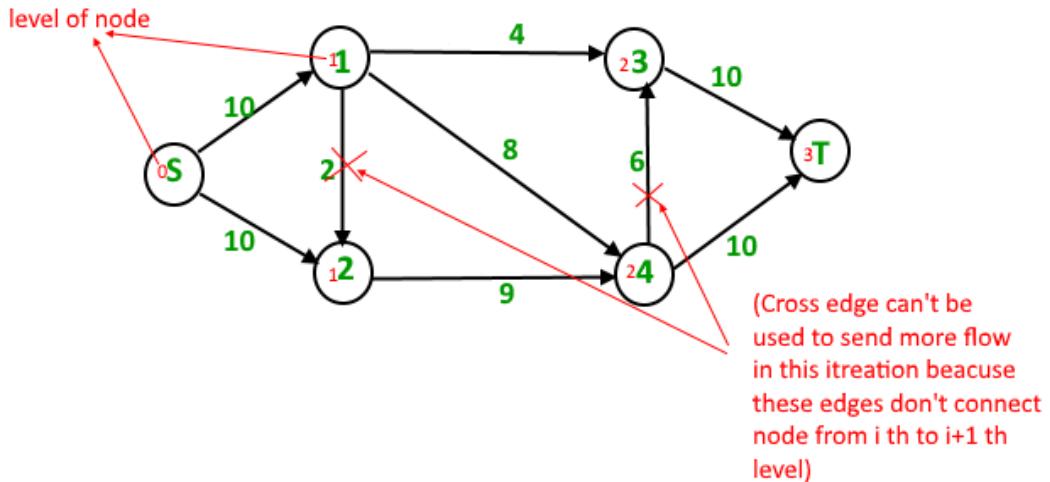
Illustration :

Initial Residual Graph (Same as given Graph)



Total Flow = 0

First Iteration : We assign levels to all nodes using BFS. We also check if more flow is possible (or there is a s-t path in residual graph).



Now we find blocking flow using levels (means every flow path should have levels as 0, 1, 2, 3). We send three flows together. This is where it is optimized compared to Edmond Karp where we send one flow at a time.

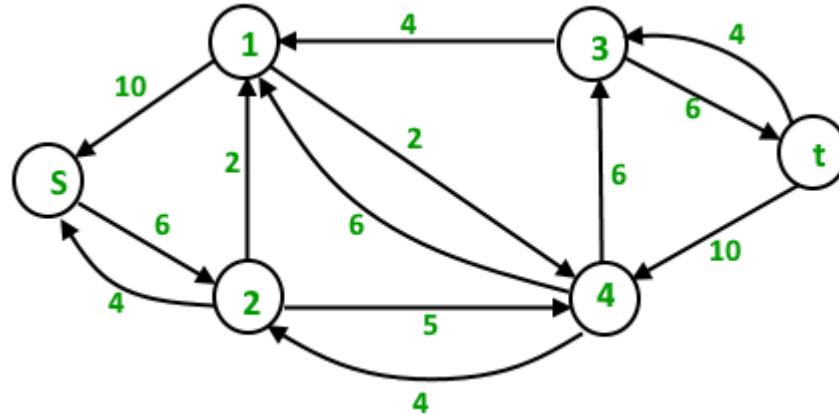
4 units of flow on path $s - 1 - 3 - t$.

6 units of flow on path $s - 1 - 4 - t$.

4 units of flow on path $s - 2 - 4 - t$.

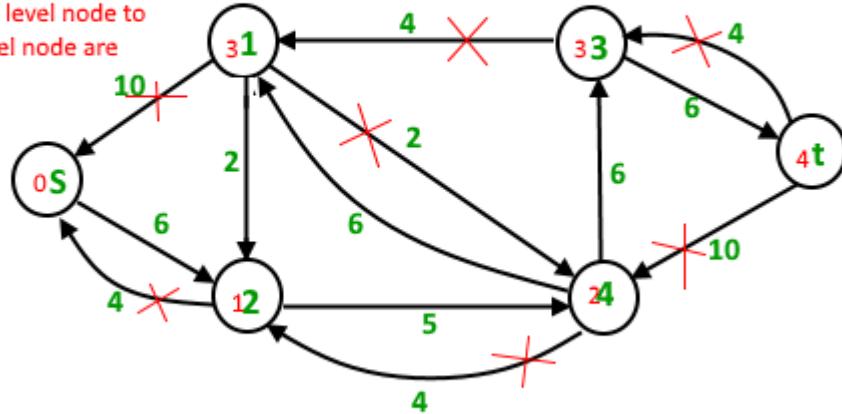
Total flow = Total flow + 4 + 6 + 4 = 14

After one iteration, residual graph changes to following.



Second Iteration : We assign new levels to all nodes using BFS of above modified residual graph. We also check if more flow is possible (or there is a $s-t$ path in residual graph).

Those edges that are not go from i th level node to $(i+1)$ th level node are crossed

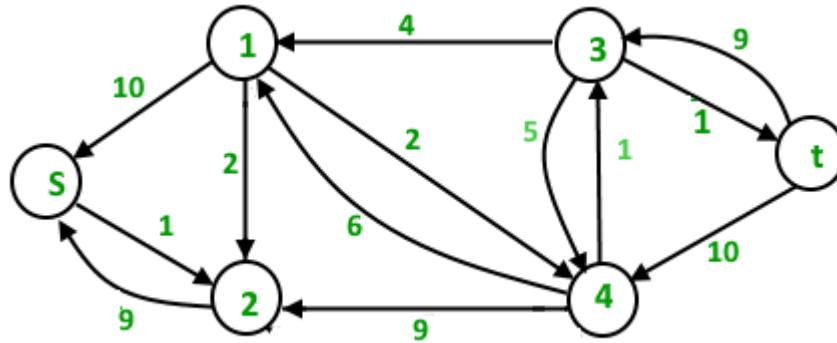


Now we find blocking flow using levels (means every flow path should have levels as 0, 1, 2, 3, 4). We can send only one flow this time.

5 units of flow on path $s - 2 - 4 - 3 - t$

Total flow = Total flow + 5 = 19

The new residual graph is



Third Iteration : We run BFS and create a level graph. We also check if more flow is possible and proceed only if possible. This time there is no $s-t$ path in residual graph, so we terminate the algorithm.

Implementation :

Below is c++ implementation of Dinic's algorithm:

```
// C++ implementation of Dinic's Algorithm
#include<bits/stdc++.h>
using namespace std;

// A structure to represent a edge between
// two vertex
struct Edge
{
    int v ; // Vertex v (or "to" vertex)
    // of a directed edge u-v. "From"
    // vertex u can be obtained using
    // index in adjacent array.

    int flow ; // flow of data in edge

    int C; // capacity

    int rev ; // To store index of reverse
    // edge in adjacency list so that
    // we can quickly find it.
};


```

```

// Residual Graph
class Graph
{
    int V; // number of vertex
    int *level ; // stores level of a node
    vector< Edge > *adj;
public :
    Graph(int V)
    {
        adj = new vector<Edge>[V];
        this->V = V;
        level = new int[V];
    }

    // add edge to the graph
    void addEdge(int u, int v, int C)
    {
        // Forward edge : 0 flow and C capacity
        Edge a{v, 0, C, adj[v].size()};
        adj[u].push_back(a);

        // Back edge : 0 flow and 0 capacity
        Edge b{u, 0, 0, adj[u].size()};

        adj[u].push_back(b); // reverse edge
    }

    bool BFS(int s, int t);
    int sendFlow(int s, int flow, int t, int ptr[]);
    int DinicMaxflow(int s, int t);
};

// Finds if more flow can be sent from s to t.
// Also assigns levels to nodes.
bool Graph::BFS(int s, int t)
{
    for (int i = 0 ; i < V ; i++)
        level[i] = -1;

    level[s] = 0; // Level of source vertex

    // Create a queue, enqueue source vertex
    // and mark source vertex as visited here
    // level[] array works as visited array also.
    list< int > q;
    q.push_back(s);
}

```

```

vector<Edge>::iterator i ;
while (!q.empty())
{
    int u = q.front();
    q.pop_front();
    for (i = adj[u].begin(); i != adj[u].end(); i++)
    {
        Edge &e = *i;
        if (level[e.v] < 0 && e.flow < e.C)
        {
            // Level of current vertex is,
            // level of parent + 1
            level[e.v] = level[u] + 1;

            q.push_back(e.v);
        }
    }
}

// IF we can not reach to the sink we
// return false else true
return level[t] < 0 ? false : true ;
}

// A DFS based function to send flow after BFS has
// figured out that there is a possible flow and
// constructed levels. This function called multiple
// times for a single call of BFS.
// flow : Current flow send by parent function call
// start[] : To keep track of next edge to be explored.
//             start[i] stores count of edges explored
//             from i.
// u : Current vertex
// t : Sink
int Graph::sendFlow(int u, int flow, int t, int start[])
{
    // Sink reached
    if (u == t)
        return flow;

    // Traverse all adjacent edges one -by - one.
    for ( ; start[u] < adj[u].size(); start[u]++)
    {
        // Pick next edge from adjacency list of u
        Edge &e = adj[u][start[u]];

        if (level[e.v] == level[u]+1 && e.flow < e.C)
        {

```

```

// find minimum flow from u to t
int curr_flow = min(flow, e.C - e.flow);

int temp_flow = sendFlow(e.v, curr_flow, t, start);

// flow is greater than zero
if (temp_flow > 0)
{
    // add flow to current edge
    e.flow += temp_flow;

    // subtract flow from reverse edge
    // of current edge
    adj[e.v][e.rev].flow -= temp_flow;
    return temp_flow;
}
}

return 0;
}

// Returns maximum flow in graph
int Graph::DinicMaxflow(int s, int t)
{
    // Corner case
    if (s == t)
        return -1;

    int total = 0; // Initialize result

    // Augment the flow while there is path
    // from source to sink
    while (BFS(s, t) == true)
    {
        // store how many edges are visited
        // from V { 0 to V }
        int *start = new int[V+1];

        // while flow is not zero in graph from S to D
        while (int flow = sendFlow(s, INT_MAX, t, start))

            // Add path flow to overall flow
            total += flow;
    }

    // return maximum flow
    return total;
}

```

```
}  
  
// Driver program to test above functions  
int main()  
{  
    Graph g(6);  
    g.addEdge(0, 1, 16 );  
    g.addEdge(0, 2, 13 );  
    g.addEdge(1, 2, 10 );  
    g.addEdge(1, 3, 12 );  
    g.addEdge(2, 1, 4 );  
    g.addEdge(2, 4, 14);  
    g.addEdge(3, 2, 9 );  
    g.addEdge(3, 5, 20 );  
    g.addEdge(4, 3, 7 );  
    g.addEdge(4, 5, 4);  
  
    // next exmp  
/*g.addEdge(0, 1, 3 );  
    g.addEdge(0, 2, 7 ) ;  
    g.addEdge(1, 3, 9);  
    g.addEdge(1, 4, 9 );  
    g.addEdge(2, 1, 9 );  
    g.addEdge(2, 4, 9);  
    g.addEdge(2, 5, 4);  
    g.addEdge(3, 5, 3);  
    g.addEdge(4, 5, 7 );  
    g.addEdge(0, 4, 10);  
  
    // next exp  
    g.addEdge(0, 1, 10);  
    g.addEdge(0, 2, 10);  
    g.addEdge(1, 3, 4 );  
    g.addEdge(1, 4, 8 );  
    g.addEdge(1, 2, 2 );  
    g.addEdge(2, 4, 9 );  
    g.addEdge(3, 5, 10 );  
    g.addEdge(4, 3, 6 );  
    g.addEdge(4, 5, 10 ); */  
  
    cout << "Maximum flow " << g.DinicMaxflow(0, 5);  
    return 0;  
}
```

Output:

```
Maximum flow 23
```

Time Complexity : $O(EV^2)$. Doing a BFS to construct level graph takes $O(E)$ time. Sending multiple more flows until a blocking flow is reached takes $O(VE)$ time. The outer loop runs at-most $O(V)$ time. In each iteration, we construct new level graph and find blocking flow. It can be proved that the number of levels increase at least by one in every iteration (Refer the below reference video for the proof). So the outer loop runs at most $O(V)$ times. Therefore overall time complexity is $O(EV^2)$.

References :

[https://en.wikipedia.org/wiki/Dinic's_algorithm](https://en.wikipedia.org/wiki/Dinic%27s_algorithm)

<https://www.youtube.com/watch?v=uM06jHdIC70>

Source

<https://www.geeksforgeeks.org/dinics-algorithm-maximum-flow/>

Chapter 137

Disjoint Set (Or Union-Find) Set 1 (Detect Cycle in an Undirected Graph)

Disjoint Set (Or Union-Find) Set 1 (Detect Cycle in an Undirected Graph) - GeeksforGeeks

A [*disjoint-set data structure*](#) is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. A [*union-find algorithm*](#) is an algorithm that performs two useful operations on such a data structure:

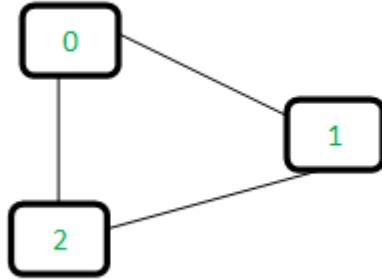
Find: Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.

Union: Join two subsets into a single subset.

In this post, we will discuss an application of Disjoint Set Data Structure. The application is to check whether a given graph contains a cycle or not.

Union-Find Algorithm can be used to check whether an undirected graph contains cycle or not. Note that we have discussed an [*algorithm to detect cycle*](#). This is another method based on *Union-Find*. This method assumes that graph doesn't contain any self-loops. We can keep track of the subsets in a 1D array, let's call it `parent[]`.

Let us consider the following graph:



For each edge, make subsets using both the vertices of the edge. If both the vertices are in the same subset, a cycle is found.

Initially, all slots of parent array are initialized to -1 (means there is only one item in every subset).

0	1	2
-1	-1	-1

Now process all edges one by one.

Edge 0-1: Find the subsets in which vertices 0 and 1 are. Since they are in different subsets, we take the union of them. For taking the union, either make node 0 as parent of node 1 or vice-versa.

0	1	2	----- 1 is made parent of 0 (1 is now representative of subset {0, 1})
1	-1	-1	

Edge 1-2: 1 is in subset 1 and 2 is in subset 2. So, take union.

0	1	2	----- 2 is made parent of 1 (2 is now representative of subset {0, 1, 2})
1	2	-1	

Edge 0-2: 0 is in subset 2 and 2 is also in subset 2. Hence, including this edge forms a cycle.

How subset of 0 is same as 2?

0->1->2 // 1 is parent of 0 and 2 is parent of 1

Based on the above explanation, below are implementations:

C/C++

```
// A union-find algorithm to detect cycle in a graph
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// a structure to represent an edge in graph
struct Edge
{
    int src, dest;
};

// a structure to represent a graph
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph =
        (struct Graph*) malloc( sizeof(struct Graph) );
    graph->V = V;
    graph->E = E;

    graph->edge =
        (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );

    return graph;
}

// A utility function to find the subset of an element i
int find(int parent[], int i)
{
    if (parent[i] == -1)
        return i;
    return find(parent, parent[i]);
}

// A utility function to do union of two subsets
void Union(int parent[], int x, int y)
{
    int xset = find(parent, x);
    int yset = find(parent, y);
```

```
if(xset!=yset){
    parent[xset] = yset;
}
}

// The main function to check whether a given graph contains
// cycle or not
int isCycle( struct Graph* graph )
{
    // Allocate memory for creating V subsets
    int *parent = (int*) malloc( graph->V * sizeof(int) );

    // Initialize all subsets as single element sets
    memset(parent, -1, sizeof(int) * graph->V);

    // Iterate through all edges of graph, find subset of both
    // vertices of every edge, if both subsets are same, then
    // there is cycle in graph.
    for(int i = 0; i < graph->E; ++i)
    {
        int x = find(parent, graph->edge[i].src);
        int y = find(parent, graph->edge[i].dest);

        if (x == y)
            return 1;

        Union(parent, x, y);
    }
    return 0;
}

// Driver program to test above functions
int main()
{
    /* Let us create following graph
       0
       |
       |
       1-----2 */
    int V = 3, E = 3;
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;

    // add edge 1-2
    graph->edge[1].src = 1;
```

```
graph->edge[1].dest = 2;

// add edge 0-2
graph->edge[2].src = 0;
graph->edge[2].dest = 2;

if (isCycle(graph))
    printf( "graph contains cycle" );
else
    printf( "graph doesn't contain cycle" );

return 0;
}
```

Java

```
// Java Program for union-find algorithm to detect cycle in a graph
import java.util.*;
import java.lang.*;
import java.io.*;

class Graph
{
    int V, E;      // V-> no. of vertices & E->no.of edges
    Edge edge[]; // /collection of all edges

    class Edge
    {
        int src, dest;
    };

    // Creates a graph with V vertices and E edges
    Graph(int v,int e)
    {
        V = v;
        E = e;
        edge = new Edge[E];
        for (int i=0; i<e; ++i)
            edge[i] = new Edge();
    }

    // A utility function to find the subset of an element i
    int find(int parent[], int i)
    {
        if (parent[i] == -1)
            return i;
        return find(parent, parent[i]);
    }
}
```

```
// A utility function to do union of two subsets
void Union(int parent[], int x, int y)
{
    int xset = find(parent, x);
    int yset = find(parent, y);
    parent[xset] = yset;
}

// The main function to check whether a given graph
// contains cycle or not
int isCycle( Graph graph)
{
    // Allocate memory for creating V subsets
    int parent[] = new int[graph.V];

    // Initialize all subsets as single element sets
    for (int i=0; i<graph.V; ++i)
        parent[i]=-1;

    // Iterate through all edges of graph, find subset of both
    // vertices of every edge, if both subsets are same, then
    // there is cycle in graph.
    for (int i = 0; i < graph.E; ++i)
    {
        int x = graph.find(parent, graph.edge[i].src);
        int y = graph.find(parent, graph.edge[i].dest);

        if (x == y)
            return 1;

        graph.Union(parent, x, y);
    }
    return 0;
}

// Driver Method
public static void main (String[] args)
{
    /* Let us create following graph
     0
     | \
     |   \
     1---2 */
    int V = 3, E = 3;
    Graph graph = new Graph(V, E);
```

```
// add edge 0-1
graph.edge[0].src = 0;
graph.edge[0].dest = 1;

// add edge 1-2
graph.edge[1].src = 1;
graph.edge[1].dest = 2;

// add edge 0-2
graph.edge[2].src = 0;
graph.edge[2].dest = 2;

if (graph.isCycle(graph)==1)
    System.out.println( "graph contains cycle" );
else
    System.out.println( "graph doesn't contain cycle" );
}
}
```

Python

```
# Python Program for union-find algorithm to detect cycle in a undirected graph
# we have one egde for any two vertex i.e 1-2 is either 1-2 or 2-1 but not both

from collections import defaultdict

#This class represents a undirected graph using adjacency list representation
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices
        self.graph = defaultdict(list) # default dictionary to store graph

        # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # A utility function to find the subset of an element i
    def find_parent(self, parent,i):
        if parent[i] == -1:
            return i
        if parent[i]!= -1:
            return self.find_parent(parent,parent[i])

    # A utility function to do union of two subsets
    def union(self,parent,x,y):
        x_set = self.find_parent(parent, x)
```

```
y_set = self.find_parent(parent, y)
parent[x_set] = y_set

# The main function to check whether a given graph
# contains cycle or not
def isCyclic(self):

    # Allocate memory for creating V subsets and
    # Initialize all subsets as single element sets
    parent = [-1]*(self.V)

    # Iterate through all edges of graph, find subset of both
    # vertices of every edge, if both subsets are same, then
    # there is cycle in graph.
    for i in self.graph:
        for j in self.graph[i]:
            x = self.find_parent(parent, i)
            y = self.find_parent(parent, j)
            if x == y:
                return True
            self.union(parent,x,y)

# Create a graph given in the above diagram
g = Graph(3)
g.addEdge(0, 1)
g.addEdge(1, 2)
g.addEdge(2, 0)

if g.isCyclic():
    print "Graph contains cycle"
else :
    print "Graph does not contain cycle "

#This code is contributed by Neelam Yadav
```

Output:

```
graph contains cycle
```

Note that the implementation of *union()* and *find()* is naive and takes O(n) time in worst case. These methods can be improved to O(Logn) using *Union by Rank or Height*. We will soon be discussing *Union by Rank* in a separate post.

Related Articles :

[Union-Find Algorithm Set 2 \(Union By Rank and Path Compression\)](#)

Chapter 137. Disjoint Set (Or Union-Find) Set 1 (Detect Cycle in an Undirected Graph)

[Disjoint Set Data Structures \(Java Implementation\)](#)

[Greedy Algorithms Set 2 \(Kruskal's Minimum Spanning Tree Algorithm\)](#)

[Job Sequencing Problem Set 2 \(Using Disjoint Set\)](#)

This article is compiled by [Aashish Barnwal](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Improved By : [avinashr175](#)

Source

<https://www.geeksforgeeks.org/union-find/>

Chapter 138

Distance of nearest cell having 1 in a binary matrix

Distance of nearest cell having 1 in a binary matrix - GeeksforGeeks

Given a binary matrix of $N \times M$, containing at least a value 1. The task is to find the distance of nearest 1 in the matrix for each cell. The distance is calculated as $i_1 - i_2 + j_1 - j_2$, where i_1, j_1 are the row number and column number of the current cell and i_2, j_2 are the row number and column number of the nearest cell having value 1.

Examples:

```
Input : N = 3, M = 4
        mat[][] = {
            0, 0, 0, 1,
            0, 0, 1, 1,
            0, 1, 1, 0
        }
Output : 3 2 1 0
         2 1 0 0
         1 0 0 1
```

For cell at (0, 0), nearest 1 is at (0, 3),
so distance = (0 - 0) + (3 - 0) = 3.
Similarly all the distance can be calculated.

Method 1 (Brute Force):

The idea is to traverse the matrix for each cell and find the minimum distance.

Below is the implementation of this approach:

C++

```
// C++ program to find distance of nearest
// cell having 1 in a binary matrix.
#include<bits/stdc++.h>
#define N 3
#define M 4
using namespace std;

// Print the distance of nearest cell
// having 1 for each cell.
void printDistance(int mat[N][M])
{
    int ans[N][M];

    // Initialize the answer matrix with INT_MAX.
    for (int i = 0; i < N; i++)
        for (int j = 0; j < M; j++)
            ans[i][j] = INT_MAX;

    // For each cell
    for (int i = 0; i < N; i++)
        for (int j = 0; j < M; j++)
    {
        // Traversing the whole matrix
        // to find the minimum distance.
        for (int k = 0; k < N; k++)
            for (int l = 0; l < M; l++)
        {
            // If cell contain 1, check
            // for minimum distance.
            if (mat[k][l] == 1)
                ans[i][j] = min(ans[i][j],
                                 abs(i-k) + abs(j-l));
        }
    }

    // Printing the answer.
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < M; j++)
            cout << ans[i][j] << " ";
        cout << endl;
    }
}

// Driven Program
int main()
{
```

```
int mat[N][M] =
{
    0, 0, 0, 1,
    0, 0, 1, 1,
    0, 1, 1, 0
};

printDistance(mat);

return 0;
}
```

Java

```
// Java program to find distance of nearest
// cell having 1 in a binary matrix.

import java.io.*;

class GFG {

    static int N = 3;
    static int M = 4;

    // Print the distance of nearest cell
    // having 1 for each cell.
    static void printDistance(int mat[][])
    {
        int ans[][] = new int[N][M];

        // Initialize the answer matrix with INT_MAX.
        for (int i = 0; i < N; i++)
            for (int j = 0; j < M; j++)
                ans[i][j] = Integer.MAX_VALUE;

        // For each cell
        for (int i = 0; i < N; i++)
            for (int j = 0; j < M; j++)
            {
                // Traversing the whole matrix
                // to find the minimum distance.
                for (int k = 0; k < N; k++)
                    for (int l = 0; l < M; l++)
                    {
                        // If cell contain 1, check
                        // for minimum distance.
                        if (mat[k][l] == 1)
                            ans[i][j] =
```

```
        Math.min(ans[i][j],
                  Math.abs(i-k)
                  + Math.abs(j-l));
    }
}

// Printing the answer.
for (int i = 0; i < N; i++)
{
    for (int j = 0; j < M; j++)
        System.out.print( ans[i][j] + " ");

    System.out.println();
}
}

// Driven Program
public static void main (String[] args)
{
    int mat[][] = { {0, 0, 0, 1},
                   {0, 0, 1, 1},
                   {0, 1, 1, 0}};

    printDistance(mat);
}
}

// This code is contributed by anuj_67.
```

C#

```
// C# program to find distance of nearest
// cell having 1 in a binary matrix.

using System;

class GFG {

    static int N = 3;
    static int M = 4;

    // Print the distance of nearest cell
    // having 1 for each cell.
    static void printDistance(int [,]mat)
    {
        int [,]ans = new int[N,M];

        // Initialise the answer matrix with int.MaxValue.
```

```
for (int i = 0; i < N; i++)
    for (int j = 0; j < M; j++)
        ans[i,j] = int.MaxValue;

// For each cell
for (int i = 0; i < N; i++)
    for (int j = 0; j < M; j++)
    {
        // Traversing the whole matrix
        // to find the minimum distance.
        for (int k = 0; k < N; k++)
            for (int l = 0; l < M; l++)
            {
                // If cell contain 1, check
                // for minimum distance.
                if (mat[k,l] == 1)
                    ans[i,j] =
                        Math.Min(ans[i,j],
                                Math.Abs(i-k)
                                + Math.Abs(j-l));
            }
    }

// Printing the answer.
for (int i = 0; i < N; i++)
{
    for (int j = 0; j < M; j++)
        Console.Write( ans[i,j] + " ");

    Console.WriteLine();
}
}

// Driven Program
public static void Main ()
{
    int [,]mat = { {0, 0, 0, 1},
                  {0, 0, 1, 1},
                  {0, 1, 1, 0} };

    printDistance(mat);
}
}

// This code is contributed by anuj_67.
```

PHP

```
<?php
// PHP program to find distance of nearest
// cell having 1 in a binary matrix.
$N = 3;
$M = 4;

// Print the distance of nearest cell
// having 1 for each cell.
function printDistance( $mat)
{
    global $N,$M;
    $ans = array(array());

    // Initialize the answer
    // matrix with INT_MAX.
    for($i = 0; $i < $N; $i++)
        for ( $j = 0; $j < $M; $j++)
            $ans[$i][$j] = PHP_INT_MAX;

    // For each cell
    for ( $i = 0; $i < $N; $i++)
        for ( $j = 0; $j < $M; $j++)
    {

        // Traversing the whole matrix
        // to find the minimum distance.
        for ($k = 0; $k < $N; $k++)
            for ( $l = 0; $l < $M; $l++)
            {

                // If cell contain 1, check
                // for minimum distance.
                if ($mat[$k][$l] == 1)
                    $ans[$i][$j] = min($ans[$i][$j],
                        abs($i-$k) + abs($j - $l));
            }
    }

    // Printing the answer.
    for ( $i = 0; $i < $N; $i++)
    {
        for ( $j = 0; $j < $M; $j++)
            echo $ans[$i][$j] , " ";

        echo "\n";
    }
}
```

```
// Driver Code
$mat = array(array(0, 0, 0, 1),
             array(0, 0, 1, 1),
             array(0, 1, 1, 0));

printDistance($mat);

// This code is contributed by anuj_67.
?>
```

Output:

```
3 2 1 0
2 1 0 0
1 0 0 1
```

Time Complexity: $O(N^2 \cdot M^2)$.

Method 2 (using BFS):

The idea is to use multisource Breadth First Search. Consider each cell as a node and each boundary between any two adjacent cells be an edge. Number each cell from 1 to $N \cdot M$. Now, push all the node whose corresponding cell value is 1 in the matrix in the queue. Apply BFS using this queue to find the minimum distance of the adjacent node.

1. Create a graph with values assigned from 1 to $M \cdot N$ to all vertices. The purpose is to store position and adjacent information.
2. Create an empty queue.
3. Traverse all matrix elements and insert positions of all 1s in queue.
4. Now do a BFS traversal of graph using above created queue. In BFS, we first explore immediate adjacent of all 1's, then adjacent of adjacent, and so on. Therefore we find minimum distance.

Below is C++ implementation of this approach:

```
// C++ program to find distance of nearest
// cell having 1 in a binary matrix.
#include<bits/stdc++.h>
#define MAX 500
#define N 3
#define M 4
using namespace std;
```

```
// Making a class of graph with bfs function.
class graph
{
private:
    vector<int> g[MAX];
    int n,m;

public:
    graph(int a, int b)
    {
        n = a;
        m = b;
    }

    // Function to create graph with N*M nodes
    // considering each cell as a node and each
    // boundary as an edge.
    void createGraph()
    {
        int k = 1; // A number to be assigned to a cell

        for (int i = 1; i <= n; i++)
        {
            for (int j = 1; j <= m; j++)
            {
                // If last row, then add edge on right side.
                if (i == n)
                {
                    // If not bottom right cell.
                    if (j != m)
                    {
                        g[k].push_back(k+1);
                        g[k+1].push_back(k);
                    }
                }

                // If last column, then add edge toward down.
                else if (j == m)
                {
                    g[k].push_back(k+m);
                    g[k+m].push_back(k);
                }

                // Else make edge in all four direction.
                else
                {
                    g[k].push_back(k+1);
                    g[k+1].push_back(k);
                }
            }
        }
    }
}
```

```
        g[k].push_back(k+m);
        g[k+m].push_back(k);
    }

    k++;
}
}

// BFS function to find minimum distance
void bfs(bool visit[], int dist[], queue<int> q)
{
    while (!q.empty())
    {
        int temp = q.front();
        q.pop();

        for (int i = 0; i < g[temp].size(); i++)
        {
            if (visit[g[temp][i]] != 1)
            {
                dist[g[temp][i]] =
                    min(dist[g[temp][i]], dist[temp]+1);

                q.push(g[temp][i]);
                visit[g[temp][i]] = 1;
            }
        }
    }
}

// Printing the solution.
void print(int dist[])
{
    for (int i = 1, c = 1; i <= n*m; i++, c++)
    {
        cout << dist[i] << " ";

        if (c%m == 0)
            cout << endl;
    }
};

// Find minimum distance
void findMinDistance(bool mat[N][M])
{
    // Creating a graph with nodes values assigned
```

```
// from 1 to N x M and matrix adjacent.
graph g1(N, M);
g1.createGraph();

// To store minimum distance
int dist[MAX];

// To mark each node as visited or not in BFS
bool visit[MAX] = { 0 };

// Initialising the value of distance and visit.
for (int i = 1; i <= M*N; i++)
{
    dist[i] = INT_MAX;
    visit[i] = 0;
}

// Inserting nodes whose value in matrix
// is 1 in the queue.
int k = 1;
queue<int> q;
for (int i = 0; i < N; i++)
{
    for (int j = 0; j < M; j++)
    {
        if (mat[i][j] == 1)
        {
            dist[k] = 0;
            visit[k] = 1;
            q.push(k);
        }
        k++;
    }
}

// Calling for Bfs with given Queue.
g1.bfs(visit, dist, q);

// Printing the solution.
g1.print(dist);
}

// Driven Program
int main()
{
    bool mat[N][M] =
    {
        0, 0, 0, 1,
```

```
    0, 0, 1, 1,  
    0, 1, 1, 0  
};  
  
findMinDistance(mat);  
  
return 0;  
}
```

Output :

```
3 2 1 0  
2 1 0 0  
1 0 0 1
```

Time Complexity: $O(N * M)$.

Improved By : [vt_m](#)

Source

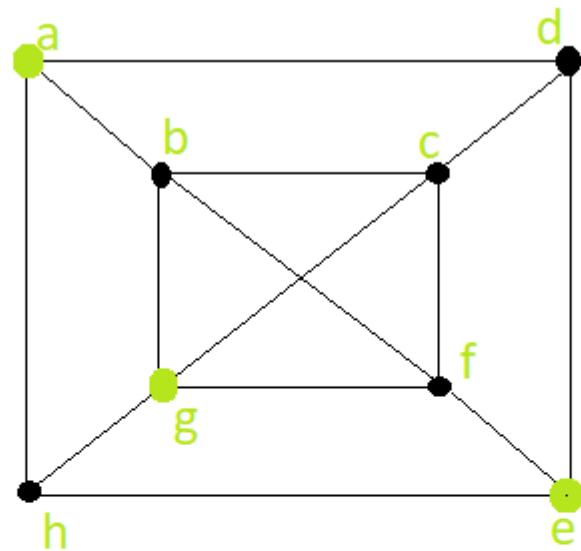
<https://www.geeksforgeeks.org/distance-nearest-cell-1-binary-matrix/>

Chapter 139

Dominant Set of a Graph

Dominant Set of a Graph - GeeksforGeeks

In graph theory, a dominating set for a graph $G = (V, E)$ is a subset D of V such that every vertex not in D is adjacent to at least one member of D . The domination number is the number of vertices in a smallest dominating set for G .



In this graph of 8 vertexs and 14 edges. Every black vertex is adjacent to at least one green vertex , which is then said that the black vertex is dominated by the green vertex. So, the Dominant Set of this graph is $S = \{ a, g, e \}$

Examples:

Input : A graph with 4 vertex and 4 edges

Output : The Dominant Set $S= \{ a, b \}$ or $\{ a, d \}$ or $\{ a, c \}$ and more.

Input : A graph with 6 vertex and 7 edges

Output : The Dominant Set $S= \{ a, d, f \}$ or $\{ e, c \}$ and more.

It is believed that there may be no efficient algorithm that finds a smallest dominating set for all graphs, but there are efficient approximation algorithms.

Algorithm :

- First we have to initialize a set ‘S’ as empty
- Take any edge ‘e’ of the graph connecting the vertices (say A and B)
- Add one vertex between A and B (let say A) to our set S

- Delete all the edges in the graph connected to A
- Go back to step 2 and repeat, if some edge is still left in the graph
- The final set S is a Dominant Set of the graph

```

// C++ program to find the Dominant Set of a graph
#include <bits/stdc++.h>
using namespace std;

vector<vector<int> > g;
bool box[100000];

vector<int> Dominant(int ver, int edge)
{
    vector<int> S; // set S
    for (int i = 0; i < ver; i++) {
        if (!box[i]) {
            S.push_back(i);
            box[i] = true;
            for (int j = 0; j < (int)g[i].size(); j++) {
                if (!box[g[i][j]]) {
                    box[g[i][j]] = true;
                    break;
                }
            }
        }
    }
    return S;
}

// Driver function
int main()
{
    int ver, edge, x, y;

    ver = 5; // Enter number of vertices
    edge = 6; // Enter number of Edges
    g.resize(ver);

    // Setting all index value of an array as 0
    memset(box, 0, sizeof(box));

    // Enter all the end-points of all the Edges
    // g[x--].push_back[y--]      g[y--].push_back[x--]
    g[0].push_back(1);
    g[1].push_back(0); // x = 1, y = 2 ;
    g[1].push_back(2);
    g[2].push_back(1); // x = 2, y = 3 ;

```

```
g[2].push_back(3);
g[3].push_back(2); // x = 3, y = 4 ;
g[0].push_back(3);
g[3].push_back(0); // x = 1, y = 4 ;
g[3].push_back(4);
g[4].push_back(3); // x = 4, y = 5 ;
g[2].push_back(4);
g[4].push_back(2); // x = 3, y = 5 ;

vector<int> S = Dominant(ver, edge);
cout << "The Dominant Set is : { ";
for (int i = 0; i < (int)S.size(); i++)
    cout << S[i] + 1 << " ";
cout << "}";
return 0;
}
```

Output:

The Dominant Set is : { 1 3 5 }

Reference : [wiki](#)

Source

<https://www.geeksforgeeks.org/dominant-set-of-a-graph/>

Chapter 140

Dynamic Connectivity Set 1 (Incremental)

Dynamic Connectivity Set 1 (Incremental) - GeeksforGeeks

Dynamic connectivity is a data structure that dynamically maintains the information about three connected components of graph. In simple words suppose there is a graph $G(V, E)$ in which no. of vertices V is constant but no. of edges E is variable. There are three ways in which we can change the number of edges

1. Incremental Connectivity : Edges are only added to the graph.
2. Decremental Connectivity : Edges are only deleted from the graph.
3. Fully Dynamic Connectivity : Edges can both be deleted and added to the graph.

In this article only **Incremental connectivity** is discussed. There are mainly two operations that need to be handled.

1. An edge is added to the graph.
2. Information about two nodes x and y whether they are in the same connected components or not.

Example:

```
Input : V = 7
        Number of operations = 11
        1 0 1
        2 0 1
        2 1 2
```

```
1 0 2
2 0 2
2 2 3
2 3 4
1 0 5
2 4 5
2 5 6
1 2 6

Note: 7 represents number of nodes,
      11 represents number of queries.
      There are two types of queries
      Type 1 : 1 x y in this if the node
                 x and y are connected print
                 Yes else No
      Type 2 : 2 x y in this add an edge
                 between node x and y

Output : No
         Yes
         No
         Yes

Explanation :
Initially there are no edges so node 0 and 1
will be disconnected so answer will be No
Node 0 and 2 will be connected through node
1 so answer will be Yes similarly for other
queries we can find whether two nodes are
connected or not
```

To solve the problems of incremental connectivity [disjoint data structure](#) is used. Here each connected component represents a set and if the two nodes belong to the same set it means that they are connected.

C++ implementation is given below here we are using [union by rank and path compression](#)

```
// C++ implementation of incremental connectivity
#include<bits/stdc++.h>
using namespace std;

// Finding the root of node i
int root(int arr[], int i)
{
    while (arr[i] != i)
    {
        arr[i] = arr[arr[i]];
        i = arr[i];
    }
    return i;
}
```

```
// union of two nodes a and b
void weighted_union(int arr[], int rank[],
                     int a, int b)
{
    int root_a = root(arr, a);
    int root_b = root(arr, b);

    // union based on rank
    if (rank[root_a] < rank[root_b])
    {
        arr[root_a] = arr[root_b];
        rank[root_b] += rank[root_a];
    }
    else
    {
        arr[root_b] = arr[root_a];
        rank[root_a] += rank[root_b];
    }
}

// Returns true if two nodes have same root
bool areSame(int arr[], int a, int b)
{
    return (root(arr, a) == root(arr, b));
}

// Performing an operation according to query type
void query(int type, int x, int y, int arr[], int rank[])
{
    // type 1 query means checking if node x and y
    // are connected or not
    if (type == 1)
    {
        // If roots of x and y is same then yes
        // is the answer
        if (areSame(arr, x, y) == true)
            cout << "Yes" << endl;
        else
            cout << "No" << endl;
    }

    // type 2 query refers union of x and y
    else if (type == 2)
    {
        // If x and y have different roots then
        // union them
        if (areSame(arr, x, y) == false)
```

```
        weighted_union(arr, rank, x, y);
    }
}

// Driver function
int main()
{
    // No.of nodes
    int n = 7;

    // The following two arrays are used to
    // implement disjoint set data structure.
    // arr[] holds the parent nodes while rank
    // array holds the rank of subset
    int arr[n], rank[n];

    // initializing both array and rank
    for (int i=0; i<n; i++)
    {
        arr[i] = i;
        rank[i] = 1;
    }

    // number of queries
    int q = 11;
    query(1, 0, 1, arr, rank);
    query(2, 0, 1, arr, rank);
    query(2, 1, 2, arr, rank);
    query(1, 0, 2, arr, rank);
    query(2, 0, 2, arr, rank);
    query(2, 2, 3, arr, rank);
    query(2, 3, 4, arr, rank);
    query(1, 0, 5, arr, rank);
    query(2, 4, 5, arr, rank);
    query(2, 5, 6, arr, rank);
    query(1, 2, 6, arr, rank);
    return 0;
}
```

Output:

No
Yes
No
Yes

Time Complexity:

The amortized time complexity is $O(\alpha(n))$ per operation where α is [inverse acker-mann function](#) which is nearly constant.

Reference:

https://en.wikipedia.org/wiki/Dynamic_connectivity

Source

<https://www.geeksforgeeks.org/dynamic-connectivity-set-1-incremental/>

Chapter 141

Erdos Renyl Model (for generating Random Graphs)

Erdos Renyl Model (for generating Random Graphs) - GeeksforGeeks

In graph theory, the Erdos–Rényi model is either of two closely related models for generating random graphs.

There are two closely related variants of the Erdos–Rényi (ER) random graph model.

In the $G(n, M)$ model, a graph is chosen uniformly at random from the collection of all graphs which have n nodes and M edges. For example, in the $G(3, 2)$ model, each of the three possible graphs on three vertices and two edges are included with probability $1/3$.

In the $G(n, p)$ model, a graph is constructed by connecting nodes randomly. Each edge is included in the graph with probability p independent from every other edge. Equivalently, all graphs with n nodes and M edges have equal probability of



A graph generated by the binomial model of Erdos and Rényi ($p = 0.01$)

The parameter p in this model can be thought of as a weighting function; as p increases from 0 to 1, the model becomes more and more likely to include graphs with more edges and less and less likely to include graphs with fewer edges. In particular, the case $p =$

0.5 corresponds to the case where all graphs on n vertices are chosen with equal probability.

The article will basically deal with the $G(n, p)$ model where n is the no of nodes to be created and p defines the probability of joining of each node to the other.

Properties of $G(n, p)$

With the notation above, a graph in $G(n, p)$ has on average $\frac{np}{2}$ edges. The distribution of the degree of any particular vertex is binomial:

$$\sum_{k=0}^n \binom{n}{k} p^k (1-p)^{n-k} = \binom{n}{k} p^k (1-p)^{n-k}$$

Where n is the total number of vertices in the graph. Since

$$\sum_{k=0}^n \binom{n}{k} p^k (1-p)^{n-k} \approx \frac{np}{2} e^{-np} n! \quad \text{as } np \rightarrow \text{constant}$$

This distribution is Poisson for large n and $np = \text{const}$.

In a 1960 paper, Erdos and Rényi described the behaviour of $G(n, p)$ very precisely for various values of p . Their results included that:

If $np < 1$, then a graph in $G(n, p)$ will almost surely have no connected components of size larger than $O(\log(n))$.

If $np = 1$, then a graph in $G(n, p)$ will almost surely have a largest component whose size is of order \sqrt{n} .

If $np \rightarrow c > 1$, where c is a constant, then a graph in $G(n, p)$ will almost surely have a unique giant component containing a positive fraction of the vertices. No other component will contain more than $O(\log(n))$ vertices.

If $p < \frac{\ln n - \gamma + \epsilon}{n}$, then a graph in $G(n, p)$ will almost surely contain isolated vertices, and thus be disconnected.

If $p > \frac{\ln n - \gamma + \epsilon}{n}$, then a graph in $G(n, p)$ will almost surely be connected.

Thus $\frac{\ln n - \gamma + \epsilon}{n}$ is a sharp threshold for the connectedness of $G(n, p)$.

Further properties of the graph can be described almost precisely as n tends to infinity. For example, there is a $k(n)$ (approximately equal to $2\log_2(n)$) such that the largest clique in $G(n, 0.5)$ has almost surely either size $k(n)$ or $k(n) + 1$.

Thus, even though finding the size of the largest clique in a graph is NP-complete, the size of the largest clique in a "typical" graph (according to this model) is very well understood. Interestingly, edge-dual graphs of Erdos-Renyi graphs are graphs with nearly the same degree distribution, but with degree correlations and a significantly higher clustering coefficient.

Next I'll describe the code to be used for making the ER graph. For implementation of the code below, you'll need to install the networkx library as well you'll need to install the matplotlib library. Following you'll see the exact code of the graph which has been used as a function of the networkx library lately in this article.

Erdos_renyi_graph(n, p, seed=None, directed=False)

Returns a $G(n,p)$ random graph, also known as an Erdős-Rényi graph or a binomial graph. The $G(n,p)$ model chooses each of the possible edges with probability p .

The functions `binomial_graph()` and `erdos_renyi_graph()` are aliases of this function.

Parameters: n (int) – The number of nodes.

p (float) – Probability for edge creation.

`seed` (int, optional) – Seed for random number generator (default=`None`).

`directed` (bool, optional (default=`False`)) – If True, this function returns a directed graph.

```
#importing the networkx library
>>> import networkx as nx

#imorting the matplotlib library for plotting the graph
>>> import matplotlib.pyplot as plt

>>> G= nx.erdos_renyi_graph(50,0.5)
>>> nx.draw(G, with_labels=True)
>>> plt.show()
```

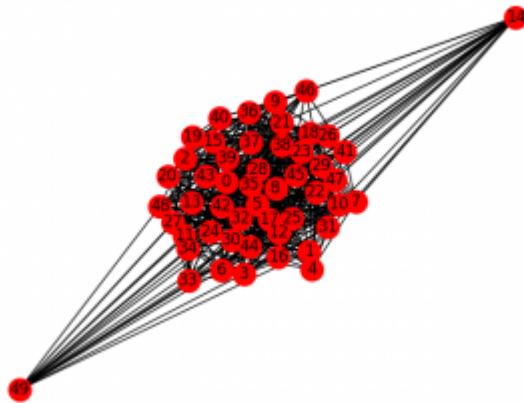


Figure 1: For $n=50$, $p=0.5$

The above example is for 50 nodes and is thus a bit unclear.

When considering the case for lesser no of nodes (for example 10), you can clearly see the difference.

Using the codes for various probabilities, we can see the difference easily:

```
>>> I= nx.erdos_renyi_graph(10,0)
>>> nx.draw(I, with_labels=True)
>>> plt.show()
```

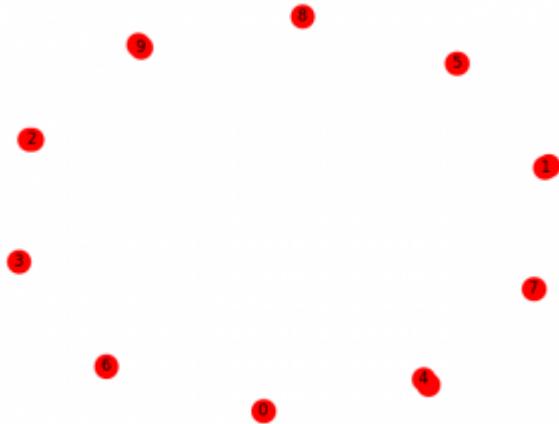


Figure 2: For $n=10$, $p=0$

```
>>> K=nx.erdos_renyi_graph(10,0.25)
>>> nx.draw(K, with_labels=True)
>>> plt.show()
```

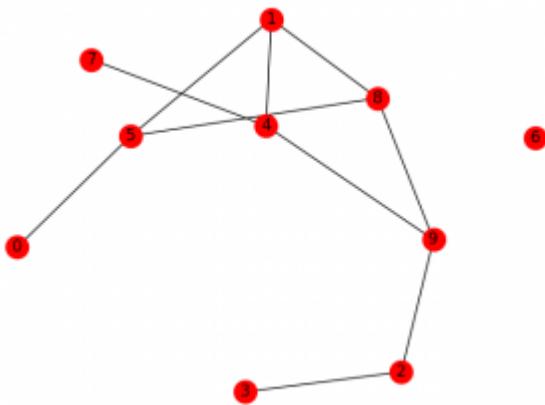


Figure 3: For $n=10$, $p=0.25$

```
>>>H= nx.erdos_renyi_graph(10,0.5)
>>> nx.draw(H, with_labels=True)
>>> plt.show()
```

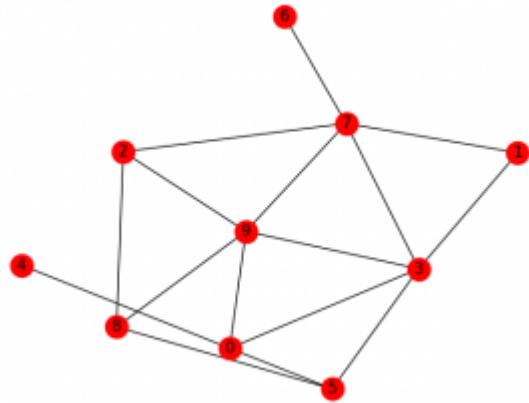


Figure 4: For n=10, p=0.5

This algorithm runs in $O(n^2)$ time. For sparse graphs (that is, for small values of p), `fast_gnp_random_graph()` is a faster algorithm.

Thus the above examples clearly define the use of erdos renyi model to make random graphs and how to use the foresaid using the networkx library of python.

Next we will discuss the ego graph and various other types of graphs in python using the library networkx.

References

You can read more about the same at

https://en.wikipedia.org/wiki/Erd%91s%E2%80%93R%C3%A9nyi_model

<http://networkx.readthedocs.io/en/networkx-1.10/index.html>

Source

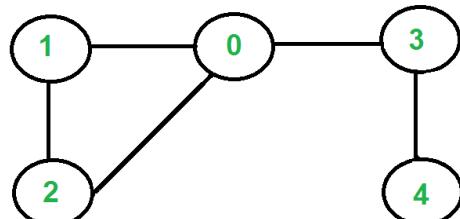
<https://www.geeksforgeeks.org/erdos-renyl-model-generating-random-graphs/>

Chapter 142

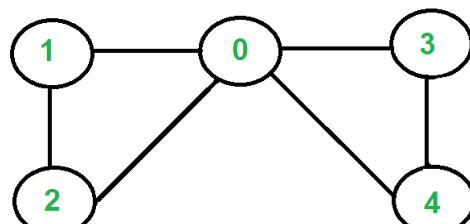
Eulerian path and circuit for undirected graph

Eulerian path and circuit for undirected graph - GeeksforGeeks

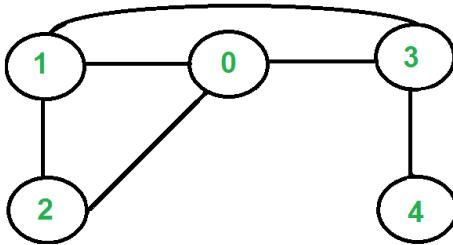
Eulerian Path is a path in graph that visits every edge exactly once. Eulerian Circuit is an Eulerian Path which starts and ends on the same vertex.



The graph has Eulerian Paths, for example "4 3 0 1 2 0", but no Eulerian Cycle. Note that there are two vertices with odd degree (4 and 0)



The graph has Eulerian Cycles, for example "2 1 0 3 4 0 2"
Note that all vertices have even degree



The graph is not Eulerian. Note that there are four vertices with odd degree (0, 1, 3 and 4)

How to find whether a given graph is Eulerian or not?

The problem is same as following question. “Is it possible to draw a given graph without lifting pencil from the paper and without tracing any of the edges more than once”.

A graph is called Eulerian if it has an Eulerian Cycle and called Semi-Eulerian if it has an Eulerian Path. The problem seems similar to [Hamiltonian Path](#) which is NP complete problem for a general graph. Fortunately, we can find whether a given graph has a Eulerian Path or not in polynomial time. In fact, we can find it in $O(V+E)$ time.

Following are some interesting properties of undirected graphs with an Eulerian path and cycle. We can use these properties to find whether a graph is Eulerian or not.

Eulerian Cycle

An undirected graph has Eulerian cycle if following two conditions are true.

-a) All vertices with non-zero degree are connected. We don't care about vertices with zero degree because they don't belong to Eulerian Cycle or Path (we only consider all edges).
-b) All vertices have even degree.

Eulerian Path

An undirected graph has Eulerian Path if following two conditions are true.

-a) Same as condition (a) for Eulerian Cycle
-b) If two vertices have odd degree and all other vertices have even degree. Note that only one vertex with odd degree is not possible in an undirected graph (sum of all degrees is always even in an undirected graph)

Note that a graph with no edges is considered Eulerian because there are no edges to traverse.

How does this work?

In Eulerian path, each time we visit a vertex v , we walk through two unvisited edges with one end point as v . Therefore, all middle vertices in Eulerian Path must have even degree. For Eulerian Cycle, any vertex can be middle vertex, therefore all vertices must have even degree.

C++

```

// A C++ program to check if a given graph is Eulerian or not
#include<iostream>
#include <list>
using namespace std;

// A class that represents an undirected graph
  
```

```

class Graph
{
    int V;      // No. of vertices
    list<int> *adj;     // A dynamic array of adjacency lists
public:
    // Constructor and destructor
    Graph(int V) {this->V = V; adj = new list<int>[V]; }
    ~Graph() { delete [] adj; } // To avoid memory leak

    // function to add an edge to graph
    void addEdge(int v, int w);

    // Method to check if this graph is Eulerian or not
    int isEulerian();

    // Method to check if all non-zero degree vertices are connected
    bool isConnected();

    // Function to do DFS starting from v. Used in isConnected();
    void DFSUtil(int v, bool visited[]);
};

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
    adj[w].push_back(v); // Note: the graph is undirected
}

void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

// Method to check if all non-zero degree vertices are connected.
// It mainly does DFS traversal starting from
bool Graph::isConnected()
{
    // Mark all the vertices as not visited
    bool visited[V];
    int i;
    for (i = 0; i < V; i++)

```

```

visited[i] = false;

// Find a vertex with non-zero degree
for (i = 0; i < V; i++)
    if (adj[i].size() != 0)
        break;

// If there are no edges in the graph, return true
if (i == V)
    return true;

// Start DFS traversal from a vertex with non-zero degree
DFSUtil(i, visited);

// Check if all non-zero degree vertices are visited
for (i = 0; i < V; i++)
    if (visited[i] == false && adj[i].size() > 0)
        return false;

return true;
}

/* The function returns one of the following values
0 --> If grpah is not Eulerian
1 --> If graph has an Euler path (Semi-Eulerian)
2 --> If graph has an Euler Circuit (Eulerian) */
int Graph::isEulerian()
{
    // Check if all non-zero degree vertices are connected
    if (isConnected() == false)
        return 0;

    // Count vertices with odd degree
    int odd = 0;
    for (int i = 0; i < V; i++)
        if (adj[i].size() & 1)
            odd++;

    // If count is more than 2, then graph is not Eulerian
    if (odd > 2)
        return 0;

    // If odd count is 2, then semi-eulerian.
    // If odd count is 0, then eulerian
    // Note that odd count can never be 1 for undirected graph
    return (odd)? 1 : 2;
}

```

```
// Function to run test cases
void test(Graph &g)
{
    int res = g.isEulerian();
    if (res == 0)
        cout << "graph is not Eulerian\n";
    else if (res == 1)
        cout << "graph has a Euler path\n";
    else
        cout << "graph has a Euler cycle\n";
}

// Driver program to test above function
int main()
{
    // Let us create and test graphs shown in above figures
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 1);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    test(g1);

    Graph g2(5);
    g2.addEdge(1, 0);
    g2.addEdge(0, 2);
    g2.addEdge(2, 1);
    g2.addEdge(0, 3);
    g2.addEdge(3, 4);
    g2.addEdge(4, 0);
    test(g2);

    Graph g3(5);
    g3.addEdge(1, 0);
    g3.addEdge(0, 2);
    g3.addEdge(2, 1);
    g3.addEdge(0, 3);
    g3.addEdge(3, 4);
    g3.addEdge(1, 3);
    test(g3);

    // Let us create a graph with 3 vertices
    // connected in the form of cycle
    Graph g4(3);
    g4.addEdge(0, 1);
    g4.addEdge(1, 2);
    g4.addEdge(2, 0);
```

```
    test(g4);

    // Let us create a graph with all vertices
    // with zero degree
    Graph g5(3);
    test(g5);

    return 0;
}
```

Java

```
// A Java program to check if a given graph is Eulerian or not
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents an undirected graph using adjacency list
// representation
class Graph
{
    private int V;    // No. of vertices

    // Array of lists for Adjacency List Representation
    private LinkedList<Integer> adj[];

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    //Function to add an edge into the graph
    void addEdge(int v, int w)
    {
        adj[v].add(w); // Add w to v's list.
        adj[w].add(v); //The graph is undirected
    }

    // A function used by DFS
    void DFSUtil(int v,boolean visited[])
    {
        // Mark the current node as visited
        visited[v] = true;
```

```

// Recur for all the vertices adjacent to this vertex
Iterator<Integer> i = adj[v].listIterator();
while (i.hasNext())
{
    int n = i.next();
    if (!visited[n])
        DFSUtil(n, visited);
}
}

// Method to check if all non-zero degree vertices are
// connected. It mainly does DFS traversal starting from
boolean isConnected()
{
    // Mark all the vertices as not visited
    boolean visited[] = new boolean[V];
    int i;
    for (i = 0; i < V; i++)
        visited[i] = false;

    // Find a vertex with non-zero degree
    for (i = 0; i < V; i++)
        if (adj[i].size() != 0)
            break;

    // If there are no edges in the graph, return true
    if (i == V)
        return true;

    // Start DFS traversal from a vertex with non-zero degree
    DFSUtil(i, visited);

    // Check if all non-zero degree vertices are visited
    for (i = 0; i < V; i++)
        if (visited[i] == false && adj[i].size() > 0)
            return false;

    return true;
}

/* The function returns one of the following values
0 --> If grpah is not Eulerian
1 --> If graph has an Euler path (Semi-Eulerian)
2 --> If graph has an Euler Circuit (Eulerian) */
int isEulerian()
{
    // Check if all non-zero degree vertices are connected
    if (isConnected() == false)

```

```
        return 0;

    // Count vertices with odd degree
    int odd = 0;
    for (int i = 0; i < V; i++)
        if (adj[i].size()%2!=0)
            odd++;

    // If count is more than 2, then graph is not Eulerian
    if (odd > 2)
        return 0;

    // If odd count is 2, then semi-eulerian.
    // If odd count is 0, then eulerian
    // Note that odd count can never be 1 for undirected graph
    return (odd==2)? 1 : 2;
}

// Function to run test cases
void test()
{
    int res = isEulerian();
    if (res == 0)
        System.out.println("graph is not Eulerian");
    else if (res == 1)
        System.out.println("graph has a Euler path");
    else
        System.out.println("graph has a Euler cycle");
}

// Driver method
public static void main(String args[])
{
    // Let us create and test graphs shown in above figures
    Graph g1 = new Graph(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 1);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.test();

    Graph g2 = new Graph(5);
    g2.addEdge(1, 0);
    g2.addEdge(0, 2);
    g2.addEdge(2, 1);
    g2.addEdge(0, 3);
    g2.addEdge(3, 4);
```

```
g2.addEdge(4, 0);
g2.test();

Graph g3 = new Graph(5);
g3.addEdge(1, 0);
g3.addEdge(0, 2);
g3.addEdge(2, 1);
g3.addEdge(0, 3);
g3.addEdge(3, 4);
g3.addEdge(1, 3);
g3.test();

// Let us create a graph with 3 vertices
// connected in the form of cycle
Graph g4 = new Graph(3);
g4.addEdge(0, 1);
g4.addEdge(1, 2);
g4.addEdge(2, 0);
g4.test();

// Let us create a graph with all vertices
// with zero degree
Graph g5 = new Graph(3);
g5.test();
}

}

// This code is contributed by Aakash Hasija
```

Python

```
# Python program to check if a given graph is Eulerian or not
#Complexity : O(V+E)

from collections import defaultdict

#This class represents a undirected graph using adjacency list representation
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices
        self.graph = defaultdict(list) # default dictionary to store graph

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)
        self.graph[v].append(u)

    #A function used by isConnected
```

```
def DFSUtil(self,v,visited):
    # Mark the current node as visited
    visited[v]= True

    #Recur for all the vertices adjacent to this vertex
    for i in self.graph[v]:
        if visited[i]==False:
            self.DFSUtil(i,visited)

'''Method to check if all non-zero degree vertices are
connected. It mainly does DFS traversal starting from
node with non-zero degree'''
def isConnected(self):

    # Mark all the vertices as not visited
    visited =[False]*(self.V)

    # Find a vertex with non-zero degree
    for i in range(self.V):
        if len(self.graph[i]) > 1:
            break

    # If there are no edges in the graph, return true
    if i == self.V-1:
        return True

    # Start DFS traversal from a vertex with non-zero degree
    self.DFSUtil(i,visited)

    # Check if all non-zero degree vertices are visited
    for i in range(self.V):
        if visited[i]==False and len(self.graph[i]) > 0:
            return False

    return True

'''The function returns one of the following values
0 --> If grpah is not Eulerian
1 --> If graph has an Euler path (Semi-Eulerian)
2 --> If graph has an Euler Circuit (Eulerian)  '''
def isEulerian(self):
    # Check if all non-zero degree vertices are connected
    if self.isConnected() == False:
        return 0
    else:
        #Count vertices with odd degree
```

```
odd = 0
for i in range(self.V):
    if len(self.graph[i]) % 2 !=0:
        odd +=1

    '''If odd count is 2, then semi-eulerian.
    If odd count is 0, then eulerian
    If count is more than 2, then graph is not Eulerian
    Note that odd count can never be 1 for undirected graph'''
    if odd == 0:
        return 2
    elif odd == 2:
        return 1
    elif odd > 2:
        return 0

# Function to run test cases
def test(self):
    res = self.isEulerian()
    if res == 0:
        print "graph is not Eulerian"
    elif res ==1 :
        print "graph has a Euler path"
    else:
        print "graph has a Euler cycle"

#Let us create and test graphs shown in above figures
g1 = Graph(5);
g1.addEdge(1, 0)
g1.addEdge(0, 2)
g1.addEdge(2, 1)
g1.addEdge(0, 3)
g1.addEdge(3, 4)
g1.test()

g2 = Graph(5)
g2.addEdge(1, 0)
g2.addEdge(0, 2)
g2.addEdge(2, 1)
g2.addEdge(0, 3)
g2.addEdge(3, 4)
g2.addEdge(4, 0)
g2.test();

g3 = Graph(5)
```

```
g3.addEdge(1, 0)
g3.addEdge(0, 2)
g3.addEdge(2, 1)
g3.addEdge(0, 3)
g3.addEdge(3, 4)
g3.addEdge(1, 3)
g3.test()

#Let us create a graph with 3 vertices
# connected in the form of cycle
g4 = Graph(3)
g4.addEdge(0, 1)
g4.addEdge(1, 2)
g4.addEdge(2, 0)
g4.test()

# Let us create a graph with all veritces
# with zero degree
g5 = Graph(3)
g5.test()

#This code is contributed by Neelam Yadav
```

Output:

```
graph has a Euler path
graph has a Euler cycle
graph is not Eulerian
graph has a Euler cycle
graph has a Euler cycle
```

Time Complexity: $O(V+E)$

Next Articles:

[Eulerian Path and Circuit for a Directed Graphs.](#)
[Fleury's Algorithm to print a Eulerian Path or Circuit?](#)

References:

http://en.wikipedia.org/wiki/Eulerian_path

Improved By : StartingTheLife

Source

<https://www.geeksforgeeks.org/eulerian-path-and-circuit/>

Chapter 143

Eulerian Path in undirected graph

Eulerian Path in undirected graph - GeeksforGeeks

Given an adjacency matrix representation of an undirected graph. Find if there is any [Eulerian Path](#) in the graph. If there is no path print “No Solution”. If there is any path print the path.

Examples:

```
Input : [[0, 1, 0, 0, 1],  
         [1, 0, 1, 1, 0],  
         [0, 1, 0, 1, 0],  
         [0, 1, 1, 0, 0],  
         [1, 0, 0, 0, 0]]  
  
Output : 5 -> 1 -> 2 -> 4 -> 3 -> 2
```

```
Input : [[0, 1, 0, 1, 1],  
         [1, 0, 1, 0, 1],  
         [0, 1, 0, 1, 1],  
         [1, 1, 1, 0, 0],  
         [1, 0, 1, 0, 0]]  
Output : "No Solution"
```

The base case of this problem is if number of vertices with odd number of edges(i.e. odd degree) is greater than 2 then there is no Eulerian path.

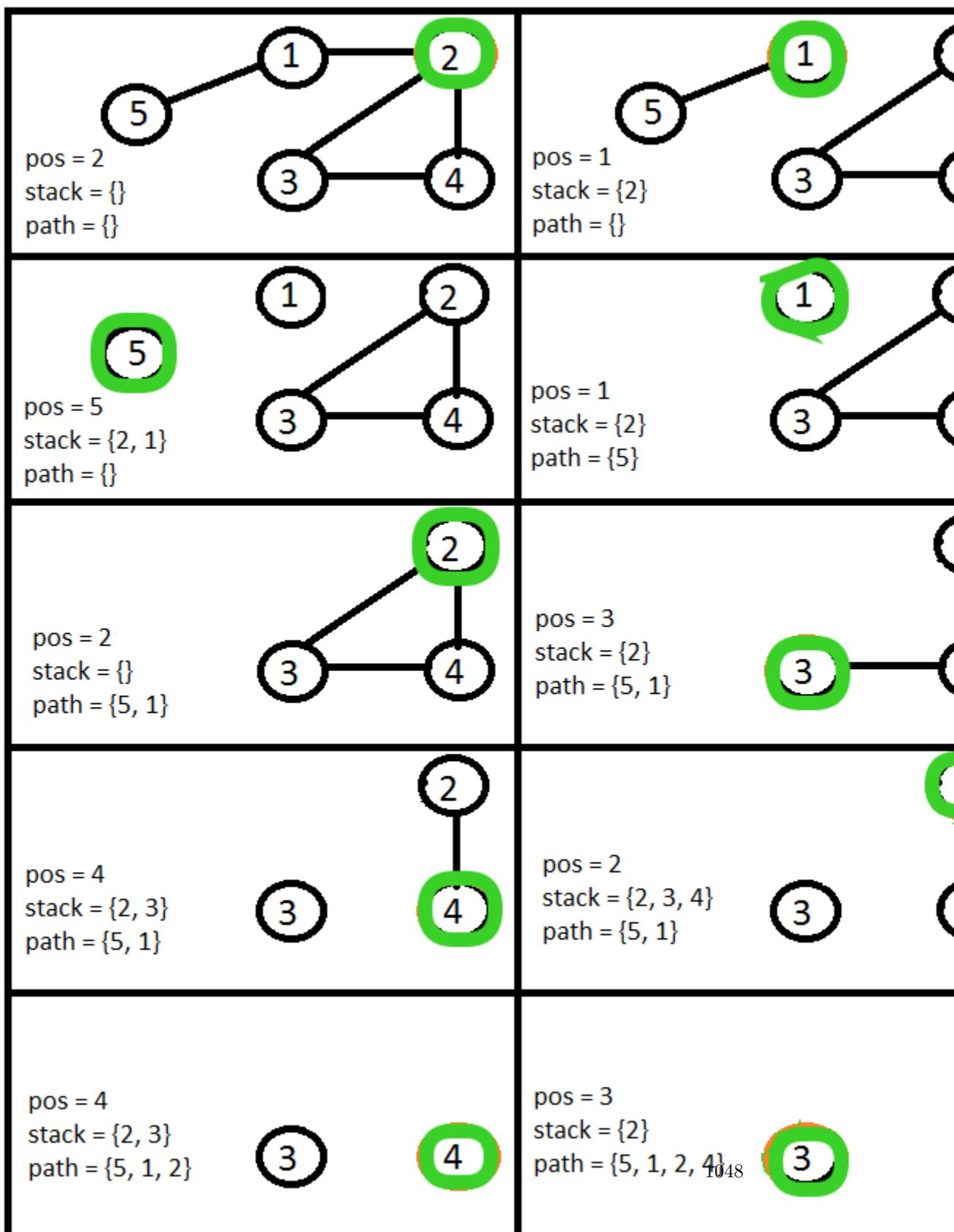
If it has solution and all the nodes has even number of edges then we can start our path from any of the nodes.

If it has solution and exactly two vertices has odd number of edges then we has to start our path from one of these two vertices.

There will not be the case where exactly one vertex has odd number of edges, as there is even number of edges in total.

Process to Find the Path:

1. First take an empty stack and an empty path.
2. If all the vertices has even number of edges then start from any of them. If two of the vertices has odd number of edges then start from one of them. Set variable current to this starting vertex.
3. If the current vertex has at least one adjacent node then first discover that node and then discover the current node by backtracking. To do so add the current node to stack, remove the edge between current node and neighbour node, set current to neighbour node.
4. If current node has not any neighbour then add it to path and pop stack set current to popped vertex.
5. Repeat process 3 and 4 until the stack is empty and current node has not any neighbour.



In next iteration: pos = 2, stack = {}, path = {5, 1, 2, 4, 3}

And in last iteration: stack = {}, path = {5, 1, 2, 4, 3, 2}

After the process path variable holds the Eulerian path.

Python3

```
# Efficient python program to find out Eulerian path

# Function to find out the path
# It takes the adjacency matrix representation of the
# graph as input
def findpath(graph):
    n = len(graph)
    numofadj = list()

    # Find out number of edges each vertex has
    for i in range(n):
        numofadj.append(sum(graph[i]))

    # Find out how many vertex has odd number edges
    startpoint = 0
    numofodd = 0
    for i in range(n-1, -1, -1):
        if (numofadj[i] % 2 == 1):
            numofodd += 1
            startpoint = i

    # If number of vertex with odd number of edges
    # is greater than two return "No Solution".
    if (numofodd > 2):
        print("No Solution")
        return

    # If there is a path find the path
    # Initialize empty stack and path
    # take the starting current as discussed
    stack = list()
    path = list()
    cur = startpoint

    # Loop will run until there is element in the stack
    # or current edge has some neighbour.
    while(stack != [] or sum(graph[cur]) != 0):

        # If current node has not any neighbour
        # add it to path and pop stack
        # set new current to the popped element
        if (sum(graph[cur]) == 0):
            path.append(cur + 1)
            cur = stack.pop(-1)
```

```

# If the current vertex has at least one
# neighbour add the current vertex to stack,
# remove the edge between them and set the
# current to its neighbour.
else:
    for i in range(n):
        if graph[cur][i] == 1:
            stack.append(cur)
            graph[cur][i] = 0
            graph[i][cur] = 0
            cur = i
            break
# print the path
for ele in path:
    print(ele, "-> ", end = '')
print(cur + 1)

# Driver Program
# Test case 1
graph1 = [[0, 1, 0, 0, 1],
           [1, 0, 1, 1, 0],
           [0, 1, 0, 1, 0],
           [0, 1, 1, 0, 0],
           [1, 0, 0, 0, 0]]
findpath(graph1)

# Test case 2
graph2 = [[0, 1, 0, 1, 1],
           [1, 0, 1, 0, 1],
           [0, 1, 0, 1, 1],
           [1, 1, 1, 0, 0],
           [1, 0, 1, 0, 0]]
findpath(graph2)

# Test case 3
graph3 = [[0, 1, 0, 0, 1],
           [1, 0, 1, 1, 1],
           [0, 1, 0, 1, 0],
           [0, 1, 1, 0, 1],
           [1, 1, 0, 1, 0]]
findpath(graph3)

```

Output:

```

4 -> 0 -> 1 -> 3 -> 2 -> 1
No Solution
4 -> 3 -> 2 -> 1 -> 4 -> 0 -> 1 -> 3

```

Time Complexity:

The runtime complexity of this algorithm is $O(E)$. This algorithm can also be used to find the Eulerian circuit. If the first and last vertex of the path is same then it will be an Eulerian circuit.

Source

<https://www.geeksforgeeks.org/eulerian-path-undirected-graph/>

Chapter 144

Fibonacci Cube Graph

Fibonacci Cube Graph - GeeksforGeeks

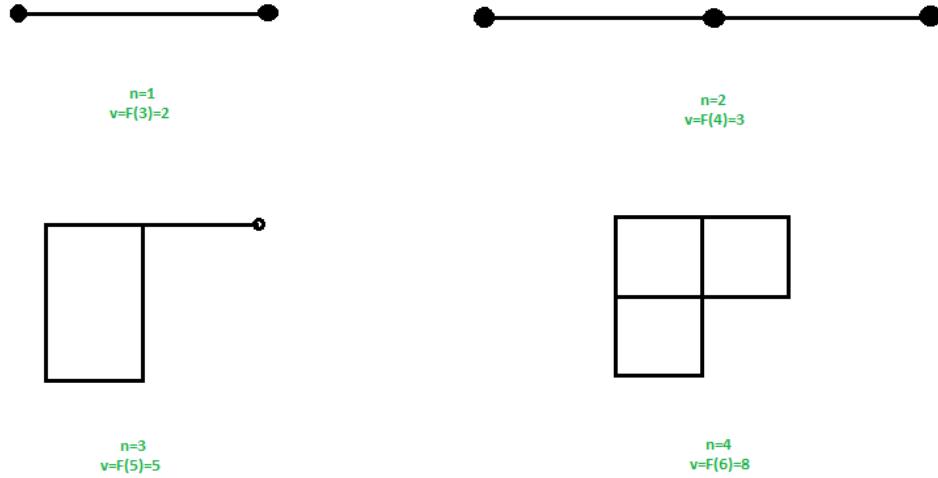
You are given input as order of graph n (highest number of edges connected to a node), you have to find number of vertices in a fibonacci cube graph of order n.

Examples :

```
Input : n = 3
Output : 5
Explanation :
Fib(n + 2) = Fib(5) = 5
```

```
Input : n = 2
Output : 3
```

A Fibonacci Cube Graph is similar to [hypercube graph](#), but with a [fibonacci number](#) of vertices. In fibonacci cube graph only 1 vertex has degree n rest all has degree less than n. Fibonacci cube graph of order n has $F(n + 2)$ vertices, where $F(n)$ is a n-th [fibonacci number](#), Fibonacci series : 1, 1, 2, 3, 5, 8, 13, 21, 34.....



For input n as order of graph, find the corresponding fibonacci number at the position $n + 2$.

$$\text{where } F(n) = F(n - 1) + F(n - 2)$$

Approach : Find the $(n + 2)$ -th fibonacci number.

Below is the implementation of above approach :

C++

```
// CPP code to find vertices in a fibonacci
// cube graph of order n
#include<iostream>
using namespace std;

// function to find fibonacci number
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n - 1) + fib(n - 2);
}

// function for finding number of vertices
// in fibonacci cube graph
int findVertices (int n)
{
    // return fibonacci number for f(n + 2)
    return fib(n + 2);
```

```
}
```

```
// driver program
int main()
{
    // n is the order of the graph
    int n = 3;
    cout << findVertices(n);
    return 0;
}
```

Java

```
// java code to find vertices in a fibonacci
// cube graph of order n
public class GFG {

    // function to find fibonacci number
    static int fib(int n)
    {
        if (n <= 1)
            return n;
        return fib(n - 1) + fib(n - 2);
    }

    // function for finding number of vertices
    // in fibonacci cube graph
    static int findVertices (int n)
    {
        // return fibonacci number for f(n + 2)
        return fib(n + 2);
    }

    public static void main(String args[])
    {
        // n is the order of the graph
        int n = 3;
        System.out.println(findVertices(n));
    }
}

// This code is contributed by Sam007
```

C#

```
// C# code to find vertices in a fibonacci
// cube graph of order n
```

```
using System;

class GFG {

    // function to find fibonacci number
    static int fib(int n)
    {
        if (n <= 1)
            return n;
        return fib(n - 1) + fib(n - 2);
    }

    // function for finding number of
    // vertices in fibonacci cube graph
    static int findVertices (int n)
    {

        // return fibonacci number for
        // f(n + 2)
        return fib(n + 2);
    }

    // Driver code
    static void Main()
    {

        // n is the order of the graph
        int n = 3;

        Console.WriteLine(findVertices(n));
    }
}

// This code is contributed by Sam007
```

PHP

```
<?php
// PHP code to find vertices in a
// fibonacci cube graph of order n

// function to find fibonacci number
function fib($n)
{
    if ($n <= 1)
        return $n;
    return fib($n - 1) + fib($n - 2);
}
```

```
// function for finding number of
// vertices in fibonacci cube graph
function findVertices ($n)
{
    // return fibonacci number
    // for f(n + 2)
    return fib($n + 2);
}

// Driver Code

// n is the order of the graph
$n = 3;
echo findVertices($n);

// This code is contributed by Sam007
?>
```

Output :

5

Note that the above code can be optimized to work in $O(\log n)$ using efficient implementations discussed in [Program for Fibonacci numbers](#)

Improved By : [Sam007](#)

Source

<https://www.geeksforgeeks.org/fibonacci-cube-graph/>

Chapter 145

Find a Mother Vertex in a Graph

Find a Mother Vertex in a Graph - GeeksforGeeks

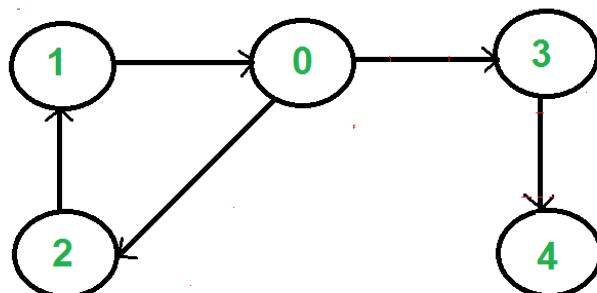
What is a Mother Vertex?

A mother vertex in a graph $G = (V,E)$ is a vertex v such that all other vertices in G can be reached by a path from v .

Example :

Input : Below Graph
Output : 5

There can be more than one mother vertices in a graph. We need to output anyone of them.
For example, in the below graph, vertices 0, 1 and 2 are mother vertices.



Mother Vertices : 0, 1 and 2

We strongly recommend you to minimize your browser and try this yourself first.

How to find mother vertex?

- **Case 1:- Undirected Connected Graph :** In this case, all the vertices are mother vertices as we can reach to all the other nodes in the graph.
- **Case 2:- Undirected/Directed Disconnected Graph :** In this case, there is no mother vertices as we cannot reach to all the other nodes in the graph.
- **Case 3:- Directed Connected Graph :** In this case, we have to find a vertex $-v$ in the graph such that we can reach to all the other nodes in the graph through a directed path.

A Naive approach :

A trivial approach will be to perform a DFS/BFS on all the vertices and find whether we can reach all the vertices from that vertex. This approach takes $O(V(E+V))$ time, which is very inefficient for large graphs.

Can we do better?

We can find a mother vertex in $O(V+E)$ time. The idea is based on [Kosaraju's Strongly Connected Component Algorithm](#). In a graph of strongly connected components, mother vertices are always vertices of source component in component graph. The idea is based on below fact.

If there exist mother vertex (or vertices), then one of the mother vertices is the last finished vertex in DFS. (Or a mother vertex has the maximum finish time in DFS traversal).

A vertex is said to be finished in DFS if a recursive call for its DFS is over, i.e., all descendants of the vertex have been visited.

How does the above idea work?

Let the last finished vertex be v . Basically, we need to prove that there cannot be an edge from another vertex u to v if u is not another mother vertex (Or there cannot exist a non-mother vertex u such that $u \rightarrow v$ is an edge). There can be two possibilities.

1. Recursive DFS call is made for u before v . If an edge $u \rightarrow v$ exists, then v must have finished before u because v is reachable through u and a vertex finishes after all its descendants.
2. Recursive DFS call is made for v before u . In this case also, if an edge $u \rightarrow v$ exists, then either v must finish before u (which contradicts our assumption that v is finished at the end) OR u should be reachable from v (which means u is another mother vertex).

Algorithm :

1. Do DFS traversal of the given graph. While doing traversal keep track of last finished vertex ' v '. This step takes $O(V+E)$ time.
2. If there exist mother vertex (or vertices), then v must be one (or one of them). Check if v is a mother vertex by doing DFS/BFS from v . This step also takes $O(V+E)$ time.

Below is implementation of above algorithm.

C/C++

```
// C++ program to find a mother vertex in O(V+E) time
#include <bits/stdc++.h>
using namespace std;

class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // adjacency lists

    // A recursive function to print DFS starting from v
    void DFSUtil(int v, vector<bool> &visited);

public:
    Graph(int V);
    void addEdge(int v, int w);
    int findMother();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

// A recursive function to print DFS starting from v
void Graph::DFSUtil(int v, vector<bool> &visited)
{
    // Mark the current node as visited and print it
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

// Returns a mother vertex if exists. Otherwise returns -1
int Graph::findMother()
{
    // visited[] is used for DFS. Initially all are
    // initialized as not visited
    vector <bool> visited(V, false);
```

```
// To store last finished vertex (or mother vertex)
int v = 0;

// Do a DFS traversal and find the last finished
// vertex
for (int i = 0; i < V; i++)
{
    if (visited[i] == false)
    {
        DFSUtil(i, visited);
        v = i;
    }
}

// If there exist mother vertex (or vertices) in given
// graph, then v must be one (or one of them)

// Now check if v is actually a mother vertex (or graph
// has a mother vertex). We basically check if every vertex
// is reachable from v or not.

// Reset all values in visited[] as false and do
// DFS beginning from v to check if all vertices are
// reachable from it or not.
fill(visited.begin(), visited.end(), false);
DFSUtil(v, visited);
for (int i=0; i<V; i++)
    if (visited[i] == false)
        return -1;

return v;
}

// Driver program to test above functions
int main()
{
    // Create a graph given in the above diagram
    Graph g(7);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(4, 1);
    g.addEdge(6, 4);
    g.addEdge(5, 6);
    g.addEdge(5, 2);
    g.addEdge(6, 0);

    cout << "A mother vertex is " << g.findMother();
```

```
    return 0;
}
```

Python

```
# program to find a mother vertex in O(V+E) time
from collections import defaultdict

# This class represents a directed graph using adjacency list
# representation
class Graph:

    def __init__(self,vertices):
        self.V = vertices #No. of vertices
        self.graph = defaultdict(list) # default dictionary

    # A recursive function to print DFS starting from v
    def DFSUtil(self, v, visited):

        # Mark the current node as visited and print it
        visited[v] = True

        # Recur for all the vertices adjacent to this vertex
        for i in self.graph[v]:
            if visited[i] == False:
                self.DFSUtil(i, visited)

    # Add w to the list of v
    def addEdge(self, v, w):
        self.graph[v].append(w)

    # Returns a mother vertex if exists. Otherwise returns -1
    def findMother(self):

        # visited[] is used for DFS. Initially all are
        # initialized as not visited
        visited =[False]*(self.V)

        # To store last finished vertex (or mother vertex)
        v=0

        # Do a DFS traversal and find the last finished
        # vertex
        for i in range(self.V):
            if visited[i]==False:
                self.DFSUtil(i,visited)
                v = i
```

```
# If there exist mother vertex (or vertices) in given
# graph, then v must be one (or one of them)

# Now check if v is actually a mother vertex (or graph
# has a mother vertex). We basically check if every vertex
# is reachable from v or not.

# Reset all values in visited[] as false and do
# DFS beginning from v to check if all vertices are
# reachable from it or not.
visited = [False]*(self.V)
self.DFSUtil(v, visited)
if any(i == False for i in visited):
    return -1
else:
    return v

# Create a graph given in the above diagram
g = Graph(7)
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 3)
g.addEdge(4, 1)
g.addEdge(6, 4)
g.addEdge(5, 6)
g.addEdge(5, 2)
g.addEdge(6, 0)
print "A mother vertex is " + str(g.findMother())

# This code is contributed by Neelam Yadav
```

Output :

A mother vertex is 5

Time Complexity : $O(V + E)$

This article is contributed by **Rachit Belwariar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Improved By : [ShubhamDixit](#)

Source

<https://www.geeksforgeeks.org/find-a-mother-vertex-in-a-graph/>

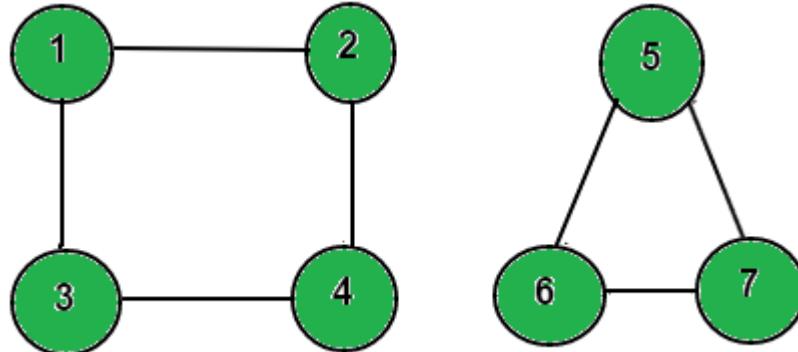
Chapter 146

Find all reachable nodes from every node present in a given set

Find all reachable nodes from every node present in a given set - GeeksforGeeks

Given an undirected graph and a set of vertices, find all reachable nodes from every vertex present in the given set.

Consider below undirected graph with 2 disconnected components.



```
arr[] = {1 , 2 , 5}
Reachable nodes from 1 are 1, 2, 3, 4
Reachable nodes from 2 are 1, 2, 3, 4
Reachable nodes from 5 are 5, 6, 7
```

Method 1 (Simple)

One straight forward solution is to do a [BFS traversal](#) for every node present in the set and then find all the reachable nodes.

Assume that we need to find reachable nodes for n nodes, the time complexity for this solution would be $O(n*(V+E))$ where V is number of nodes in the graph and E is number of edges in the graph. Please note that we need to call BFS as a separate call for every node without using the visited array of previous traversals because a same vertex may need to be printed multiple times. This seems to be an effective solution but consider the case when $E = \Theta(V^2)$ and $n = V$, time complexity becomes $O(V^3)$.

Method 2 (Efficient)

Since the given graph is undirected, all vertices that belong to same component have same set of reachable nodes. So we keep track of vertex and component mappings. Every component in the graph is assigned a number and every vertex in this component is assigned this number. We use the visit array for this purpose, the array which is used to keep track of visited vertices in BFS.

```
For a node u,  
if visit[u] is 0 then  
    u has not been visited before  
else // if not zero then  
    visit[u] represents the component number.
```

```
For any two nodes u and v belonging to same  
component, visit[u] is equal to visit[v]
```

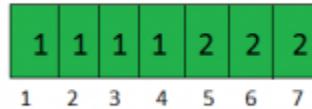
To store the reachable nodes, use a map m with key as component number and value as a vector which stores all the reachable nodes.

To find reachable nodes for a node u return $m[visit[u]]$

Look at the pseudo code below in order to understand how to assign component numbers.

```
componentNum = 0  
for i=1 to n  
    If visit[i] is NOT 0 then  
        componentNum++  
  
        // bfs() returns a list (or vector)  
        // for given vertex 'i'  
        list = bfs(i, componentNum)  
        m[visit[i]] = list
```

For the graph shown in the example the visit array would be.



For the nodes 1, 2, 3 and 4 the component number is 1. For nodes 5, 6 and 7 the component number is 2.

C++ Implementation of above idea

```
// C++ program to find all the reachable nodes
// for every node present in arr[0..n-1].
#include <bits/stdc++.h>
using namespace std;

// This class represents a directed graph using
// adjacency list representation
class Graph
{
public:
    int V;      // No. of vertices

    // Pointer to an array containing adjacency lists
    list<int> *adj;

    Graph(int); // Constructor

    void addEdge(int, int);

    vector<int> BFS(int, int []);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V+1];
}

void Graph::addEdge(int u, int v)
{
    adj[u].push_back(v); // Add w to v's list.
    adj[v].push_back(u); // Add v to w's list.
}
```

```
vector<int> Graph::BFS(int componentNum, int src,
                      int visited[])
{
    // Mark all the vertices as not visited
    // Create a queue for BFS
    queue<int> queue;

    queue.push(src);

    // Assign Component Number
    visited[src] = componentNum;

    // Vector to store all the reachable nodes from 'src'
    vector<int> reachableNodes;

    while(!queue.empty())
    {
        // Dequeue a vertex from queue
        int u = queue.front();
        queue.pop();

        reachableNodes.push_back(u);

        // Get all adjacent vertices of the dequeued
        // vertex u. If a adjacent has not been visited,
        // then mark it visited nd enqueue it
        for (auto itr = adj[u].begin();
             itr != adj[u].end(); itr++)
        {
            if (!visited[*itr])
            {
                // Assign Component Number to all the
                // reachable nodes
                visited[*itr] = componentNum;
                queue.push(*itr);
            }
        }
    }
    return reachableNodes;
}

// Display all the Reachable Nodes from a node 'n'
void displayReachableNodes(int n,
                           unordered_map <int, vector<int> > m)
{
    vector<int> temp = m[n];
    for (int i=0; i<temp.size(); i++)
        cout << temp[i] << " ";
```

```
        cout << endl;
    }

// Find all the reachable nodes for every element
// in the arr
void findReachableNodes(Graph g, int arr[], int n)
{
    // Get the number of nodes in the graph
    int V = g.V;

    // Take a integer visited array and initialize
    // all the elements with 0
    int visited[V+1];
    memset(visited, 0, sizeof(visited));

    // Map to store list of reachable Nodes for a
    // given node.
    unordered_map <int, vector<int> > m;

    // Initialize component Number with 0
    int componentNum = 0;

    // For each node in arr[] find reachable
    // Nodes
    for (int i = 0 ; i < n ; i++)
    {
        int u = arr[i];

        // Visit all the nodes of the component
        if (!visited[u])
        {
            componentNum++;

            // Store the reachable Nodes corresponding to
            // the node 'i'
            m[visited[u]] = g.BFS(componentNum, u, visited);
        }

        // At this point, we have all reachable nodes
        // from u, print them by doing a look up in map m.
        cout << "Reachable Nodes from " << u << " are\n";
        displayReachableNodes(visited[u], m);
    }
}

// Driver program to test above functions
int main()
```

```
{  
    // Create a graph given in the above diagram  
    int V = 7;  
    Graph g(V);  
    g.addEdge(1, 2);  
    g.addEdge(2, 3);  
    g.addEdge(3, 4);  
    g.addEdge(3, 1);  
    g.addEdge(5, 6);  
    g.addEdge(5, 7);  
  
    // For every ith element in the arr  
    // find all reachable nodes from query[i]  
    int arr[] = {2, 4, 5};  
  
    // Find number of elements in Set  
    int n = sizeof(arr)/sizeof(int);  
  
    findReachableNodes(g, arr, n);  
  
    return 0;  
}
```

Output:

```
Reachable Nodes from 2 are  
2 1 3 4  
Reachable Nodes from 4 are  
2 1 3 4  
Reachable Nodes from 5 are  
5 6 7
```

Time Complexity Analysis:

n = Size of the given set

E = Number of Edges

V = Number of Nodes

O(V+E) for BFS

In worst case all the V nodes are displayed for each node present in the given, i.e only one component in the graph so it takes O(n*V) time.

Worst Case Time Complexity : O(V+E) + O(n*V)

Source

<https://www.geeksforgeeks.org/find-all-reachable-nodes-from-every-node-present-in-a-given-set/>

Chapter 147

Find alphabetical order such that words can be considered sorted

Find alphabetical order such that words can be considered sorted - GeeksforGeeks

Given an array of words, find any alphabetical order in the English alphabet such that the given words can be considered sorted (increasing), if there exists such an order, otherwise output impossible.

Examples:

```
Input : words[] = {"zy", "ab"}  
Output : zabcdefghijklmnopqrstuvwxyz  
Basically we need to make sure that 'z' comes  
before 'a'.
```

```
Input : words[] = {"geeks", "gamers", "coders",  
                  "everyoneelse"}  
Output : zyxwvutsrqponmlkjihgfedb
```

```
Input : words[] = {"marvel", "superman", "spiderman",  
                  "batman"}  
Output : zyxwvuptrqonmsbdlkjihgfeca
```

Naive approach: The brute-force approach would be to check all the possible orders, and check if any of them satisfy the given order of words. Considering there are **26** alphabets in the English language, there are **26!** number of permutations that can be valid orders. Considering we check every pair for verifying an order, the complexity of this approach goes to **O(26!*N^2)**, which is well beyond practically preferred time complexity.

Using topological sort: This solution requires knowledge of [Graphs and its representation as adjacency lists](#), [DFS](#) and [Topological sorting](#).

In our required order, it is required to print letters such that each letter must be followed by the letters that are placed in lower priority than them. It seems somewhat similar to what [topological sort](#) is defined as – In topological sorting, we need to print a vertex before its adjacent vertices. Let's define each letter in the alphabet as nodes in a standard directed graph. **A** is said to be connected to **B** ($A \rightarrow B$) if **A** precedes **B** in the order. The algorithm can be formulated as follows:

1. If **n** is **1**, then any order is valid.
2. Take the first two words. Identify the first different letter (at the same index of the words) in the words. The letter in the first word will precede the letter in the second word.
3. If there exists no such letter, then the first string must be smaller in length than the second string.
4. Assign the second word to the first word and input the third word into the second word. Repeat **2**, **3** and **4 (n-1)** times.
5. Run a [DFS](#) traversal in topological order.
6. Check if all the nodes are visited. In topological order, if there are cycles in the graph, the nodes in the cycles remain not visited, since it is not possible to visit these nodes after visiting every node adjacent to it. In such a case, order does not exist. In this case, it means that the order in our list contradicts itself.

```
/* CPP program to find an order of alphabets
so that given set of words are considered
sorted */
#include <bits/stdc++.h>
using namespace std;
#define MAX_CHAR 26

void findOrder(vector<string> v)
{
    int n = v.size();

    /* If n is 1, then any order works */
    if (n == 1) {
        cout << "abcdefghijklmnopqrstuvwxyz";
        return;
    }

    /* Adjacency list of 26 characters*/
    vector<int> adj[MAX_CHAR];

    /* Array tracking the number of edges that are
```

```
inward to each node*/
vector<int> in(MAX_CHAR, 0);

// Traverse through all words in given array
string prev = v[0];

/* (n-1) loops because we already acquired the
first word in the list*/
for (int i = 1; i < n; ++i) {
    string s = v[i];

    /* Find first such letter in the present string that is different
from the letter in the previous string at the same index*/
    int j;
    for (j = 0; j < min(prev.length(), s.length()); ++j)
        if (s[j] != prev[j])
            break;

    if (j < min(prev.length(), s.length())) {

        /* The letter in the previous string precedes the the one
in the present string, hence add the letter in the present
string as the child of the letter in the previous string*/
        adj[prev[j] - 'a'].push_back(s[j] - 'a');

        /* The number of inward pointing edges to the node representing
the letter in the present string increases by one*/
        in[s[j] - 'a']++;

        /* Assign present string to previous string for the next
iteration. */
        prev = s;
        continue;
    }

    /* If there exists no such letter then the string length of
the previous string must be less than or equal to the
present string, otherwise no such order exists*/
    if (prev.length() > s.length()) {
        cout << "Impossible";
        return;
    }

    /* Assign present string to previous string for the next
iteration */
    prev = s;
}
```

```
/* Topological ordering requires the source nodes
that have no parent nodes*/
stack<int> stk;
for (int i = 0; i < MAX_CHAR; ++i)
    if (in[i] == 0)
        stk.push(i);

/* Vector storing required order (anyone that satisfies) */
vector<char> out;

/* Array to keep track of visited nodes */
bool vis[26];
memset(vis, false, sizeof(vis));

/* Standard DFS */
while (!stk.empty()) {

    /* Acquire present character */
    char x = stk.top();
    stk.pop();

    /* Mark as visited */
    vis[x] = true;

    /* Insert character to output vector */
    out.push_back(x + 'a');

    for (int i = 0; i < adj[x].size(); ++i) {
        if (vis[adj[x][i]])
            continue;

        /* Since we have already included the the present
        character in the order, the number edges inward
        to this child node can be reduced*/
        in[adj[x][i]]--;

        /* If the number of inward edges have been removed,
        we can include this node as a source node*/
        if (in[adj[x][i]] == 0)
            stk.push(adj[x][i]);
    }
}

/* Check if all nodes(alphabets) have been visited.
Order impossible if any one is unvisited*/
for (int i = 0; i < MAX_CHAR; ++i)
    if (!vis[i]) {
        cout << "Impossible";
```

```
        return;
    }

    for (int i = 0; i < out.size(); ++i)
        cout << out[i];
}

// Driver code
int main()
{
    vector<string> v{ "efgh", "abcd" };
    findOrder(v);
    return 0;
}
```

Output :

```
zyxwvutsrqponmlkjihgfdeadcb
```

The complexity of this approach is $\mathbf{O(N*S) + O(V+E)}$, where $V=26$ (number of nodes is the same as number of alphabets) and $E < N$ (since at most 1 edge is created for each word as input). Hence overall complexity is $\mathbf{O(N*S+N)}$. S represents the length of each word.

Source

<https://www.geeksforgeeks.org/find-alphabetical-order-such-that-words-can-be-considered-sorted/>

Chapter 148

Find if an array of strings can be chained to form a circle Set 1

Find if an array of strings can be chained to form a circle Set 1 - GeeksforGeeks

Given an array of strings, find if the given strings can be chained to form a circle. A string X can be put before another string Y in circle if the last character of X is same as first character of Y.

Examples:

Input: arr[] = {"geek", "king"}
Output: Yes, the given strings can be chained.
Note that the last character of first string is same as first character of second string and vice versa is also true.

Input: arr[] = {"for", "geek", "rig", "kaf"}
Output: Yes, the given strings can be chained.
The strings can be chained as "for", "rig", "geek" and "kaf"

Input: arr[] = {"aab", "bac", "aaa", "cda"}
Output: Yes, the given strings can be chained.
The strings can be chained as "aaa", "aab", "bac" and "cda"

Input: arr[] = {"aaa", "bbb", "baa", "aab"};
Output: Yes, the given strings can be chained.
The strings can be chained as "aaa", "aab", "bbb" and "baa"

Input: arr[] = {"aaa"};
Output: Yes

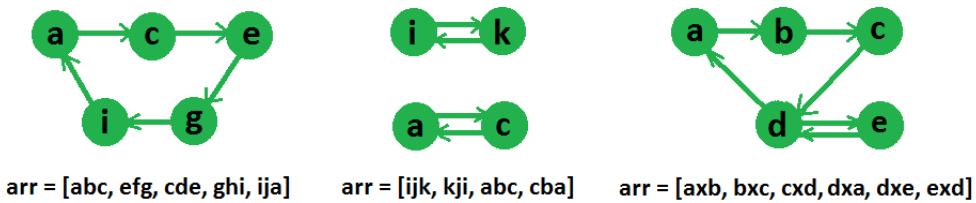
Input: arr[] = {"aaa", "bbb"};
Output: No

Input : arr[] = ["abc", "efg", "cde", "ghi", "ija"]
Output : Yes
These strings can be reordered as, "abc", "cde", "efg",
"ghi", "ija"

Input : arr[] = ["ijk", "kji", "abc", "cba"]
Output : No

The idea is to create a directed graph of all characters and then find if there is an [eulerian circuit](#) in the graph or not.

Graph representation of some string arrays are given in below diagram,



If there is an [eulerian circuit](#), then chain can be formed, otherwise not.

Note that a directed graph has [eulerian circuit](#) only if in degree and out degree of every vertex is same, and all non-zero degree vertices form a single strongly connected component.

Following are detailed steps of the algorithm.

- 1) Create a directed graph g with number of vertices equal to the size of alphabet. We have created a graph with 26 vertices in the below program.
- 2) Do following for every string in the given array of strings.
....a) Add an edge from first character to last character of the given graph.
- 3) If the created graph has [eulerian circuit](#), then return true, else return false.

Following are C++ and Python implementations of the above algorithm.

C/C++

```
// A C++ program to check if a given directed graph is Eulerian or not
#include<iostream>
#include <list>
#define CHARS 26
using namespace std;
```

```
// A class that represents an undirected graph
class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // A dynamic array of adjacency lists
    int *in;
public:
    // Constructor and destructor
    Graph(int V);
    ~Graph() { delete [] adj; delete [] in; }

    // function to add an edge to graph
    void addEdge(int v, int w) { adj[v].push_back(w); (in[w])++; }

    // Method to check if this graph is Eulerian or not
    bool isEulerianCycle();

    // Method to check if all non-zero degree vertices are connected
    bool isSC();

    // Function to do DFS starting from v. Used in isConnected();
    void DFSUtil(int v, bool visited[]);

    Graph getTranspose();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
    in = new int[V];
    for (int i = 0; i < V; i++)
        in[i] = 0;
}

/* This function returns true if the directed graph has an eulerian
cycle, otherwise returns false */
bool Graph::isEulerianCycle()
{
    // Check if all non-zero degree vertices are connected
    if (isSC() == false)
        return false;

    // Check if in degree and out degree of every vertex is same
    for (int i = 0; i < V; i++)
        if (adj[i].size() != in[i])
            return false;
```

```
        return true;
    }

// A recursive function to do DFS starting from v
void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

// Function that returns reverse (or transpose) of this graph
// This function is needed in isSC()
Graph Graph::getTranspose()
{
    Graph g(V);
    for (int v = 0; v < V; v++)
    {
        // Recur for all the vertices adjacent to this vertex
        list<int>::iterator i;
        for(i = adj[v].begin(); i != adj[v].end(); ++i)
        {
            g.adj[*i].push_back(v);
            (g.in[v])++;
        }
    }
    return g;
}

// This function returns true if all non-zero degree vertices of
// graph are strongly connected. Please refer
// https://www.geeksforgeeks.org/connectivity-in-a-directed-graph/
bool Graph::isSC()
{
    // Mark all the vertices as not visited (For first DFS)
    bool visited[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Find the first vertex with non-zero degree
    int n;
    for (n = 0; n < V; n++)
```

```
if (adj[n].size() > 0)
    break;

// Do DFS traversal starting from first non zero degree vertex.
DFSUtil(n, visited);

// If DFS traversal doesn't visit all vertices, then return false.
for (int i = 0; i < V; i++)
    if (adj[i].size() > 0 && visited[i] == false)
        return false;

// Create a reversed graph
Graph gr = getTranspose();

// Mark all the vertices as not visited (For second DFS)
for (int i = 0; i < V; i++)
    visited[i] = false;

// Do DFS for reversed graph starting from first vertex.
// Starting Vertex must be same starting point of first DFS
gr.DFSUtil(n, visited);

// If all vertices are not visited in second DFS, then
// return false
for (int i = 0; i < V; i++)
    if (adj[i].size() > 0 && visited[i] == false)
        return false;

return true;
}

// This function takes an of strings and returns true
// if the given array of strings can be chained to
// form cycle
bool canBeChained(string arr[], int n)
{
    // Create a graph with 'aplha' edges
    Graph g(CHARS);

    // Create an edge from first character to last character
    // of every string
    for (int i = 0; i < n; i++)
    {
        string s = arr[i];
        g.addEdge(s[0]-'a', s[s.length()-1]-'a');
    }

    // The given array of strings can be chained if there
```

```
// is an eulerian cycle in the created graph
return g.isEulerianCycle();
}

// Driver program to test above functions
int main()
{
    string arr1[] = {"for", "geek", "rig", "kaf"};
    int n1 = sizeof(arr1)/sizeof(arr1[0]);
    canBeChained(arr1, n1)? cout << "Can be chained n" :
                                cout << "Can't be chained n";

    string arr2[] = {"aab", "abb"};
    int n2 = sizeof(arr2)/sizeof(arr2[0]);
    canBeChained(arr2, n2)? cout << "Can be chained n" :
                                cout << "Can't be chained n";

    return 0;
}
```

Python

```
# Python program to check if a given directed graph is Eulerian or not
CHARS = 26

# A class that represents an undirected graph
class Graph(object):
    def __init__(self, V):
        self.V = V          # No. of vertices
        self.adj = [[] for x in xrange(V)]  # a dynamic array
        self.inp = [0] * V

    # function to add an edge to graph
    def addEdge(self, v, w):
        self.adj[v].append(w)
        self.inp[w]+=1

    # Method to check if this graph is Eulerian or not
    def isSC(self):
        # Mark all the vertices as not visited (For first DFS)
        visited = [False] * self.V

        # Find the first vertex with non-zero degree
        n = 0
        for n in xrange(self.V):
            if len(self.adj[n]) > 0:
                break
```

```
# Do DFS traversal starting from first non zero degree vertex.
self.DFSUtil(n, visited)

# If DFS traversal doesn't visit all vertices, then return false.
for i in xrange(self.V):
    if len(self.adj[i]) > 0 and visited[i] == False:
        return False

# Create a reversed graph
gr = self.getTranspose()

# Mark all the vertices as not visited (For second DFS)
for i in xrange(self.V):
    visited[i] = False

# Do DFS for reversed graph starting from first vertex.
# Starting Vertex must be same starting point of first DFS
gr.DFSUtil(n, visited)

# If all vertices are not visited in second DFS, then
# return false
for i in xrange(self.V):
    if len(self.adj[i]) > 0 and visited[i] == False:
        return False

return True

# This function returns true if the directed graph has an eulerian
# cycle, otherwise returns false
def isEulerianCycle(self):

    # Check if all non-zero degree vertices are connected
    if self.isSC() == False:
        return False

    # Check if in degree and out degree of every vertex is same
    for i in xrange(self.V):
        if len(self.adj[i]) != self.inp[i]:
            return False

    return True

# A recursive function to do DFS starting from v
def DFSUtil(self, v, visited):

    # Mark the current node as visited and print it
    visited[v] = True
```

```
# Recur for all the vertices adjacent to this vertex
for i in xrange(len(self.adj[v])):
    if not visited[self.adj[v][i]]:
        self.DFSUtil(self.adj[v][i], visited)

# Function that returns reverse (or transpose) of this graph
# This function is needed in isSC()
def getTranspose(self):
    g = Graph(self.V)
    for v in xrange(self.V):
        # Recur for all the vertices adjacent to this vertex
        for i in xrange(len(self.adj[v])):
            g.adj[self.adj[v][i]].append(v)
            g.inp[v]+=1
    return g

# This function takes an of strings and returns true
# if the given array of strings can be chained to
# form cycle
def canBeChained(arr, n):

    # Create a graph with 'aplha' edges
    g = Graph(CHARS)

    # Create an edge from first character to last character
    # of every string
    for i in xrange(n):
        s = arr[i]
        g.addEdge(ord(s[0])-ord('a'), ord(s[len(s)-1])-ord('a'))

    # The given array of strings can be chained if there
    # is an eulerian cycle in the created graph
    return g.isEulerianCycle()

# Driver program
arr1 = ["for", "geek", "rig", "kaf"]
n1 = len(arr1)
if canBeChained(arr1, n1):
    print "Can be chained"
else:
    print "Cant be chained"

arr2 = ["aab", "abb"]
n2 = len(arr2)
if canBeChained(arr2, n2):
    print "Can be chained"
else:
    print "Can't be chained"
```

```
# This code is contributed by BHAVYA JAIN
```

Output:

```
Can be chained  
Can't be chained
```

[Find if an array of strings can be chained to form a circle Set 2](#)

This article is contributed by **Piyush Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

<https://www.geeksforgeeks.org/given-array-strings-find-strings-can-chained-form-circle/>

Chapter 149

Find if an array of strings can be chained to form a circle Set 2

Find if an array of strings can be chained to form a circle Set 2 - GeeksforGeeks

Given an array of strings, find if the given strings can be chained to form a circle. A string X can be put before another string Y in circle if the last character of X is same as first character of Y.

Examples:

Input: arr[] = {"geek", "king"}
Output: Yes, the given strings can be chained.
Note that the last character of first string is same as first character of second string and vice versa is also true.

Input: arr[] = {"for", "geek", "rig", "kaf"}
Output: Yes, the given strings can be chained.
The strings can be chained as "for", "rig", "geek" and "kaf"

Input: arr[] = {"aab", "bac", "aaa", "cda"}
Output: Yes, the given strings can be chained.
The strings can be chained as "aaa", "aab", "bac" and "cda"

Input: arr[] = {"aaa", "bbb", "baa", "aab"};
Output: Yes, the given strings can be chained.
The strings can be chained as "aaa", "aab", "bbb" and "baa"

Input: arr[] = {"aaa"};
Output: Yes

Input: arr[] = {"aaa", "bbb"};
Output: No

Input : arr[] = ["abc", "efg", "cde", "ghi", "ija"]
Output : Yes
These strings can be reordered as, "abc", "cde", "efg",
"ghi", "ija"

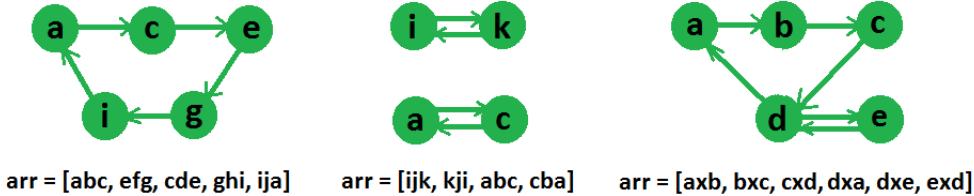
Input : arr[] = ["ijk", "kji", "abc", "cba"]
Output : No

We have discussed one approach to this problem in below post.

[Find if an array of strings can be chained to form a circle Set 1](#)

In this post another approach is discussed. We solve this problem by treating this as a graph problem, where vertices will be first and last character of strings and we will draw an edge between two vertices if they are first and last character of same string, so number of edges in graph will be same as number of strings in the array.

Graph representation of some string arrays are given in below diagram,



Now it can be clearly seen after graph representation that if a loop among graph vertices is possible then we can reorder the strings otherwise not. As in above diagram's example a loop can be found in first and third array of string but not in second array of string. Now to check whether **this graph can have a loop which goes through all the vertices**, we'll check two conditions,

- 1) Indegree and Outdegree of each vertex should be same.
- 2) Graph should be strongly connected.

First condition can be checked easily by keeping two arrays, in and out for each character. For checking whether graph is having a loop which goes through all vertices is same as checking complete directed graph is strongly connected or not because if it has a loop which goes through all vertices then we can reach to any vertex from any other vertex that is, graph will be strongly connected and same argument can be given for reverse statement also.

Now for checking second condition we will just run a DFS from any character and visit all reachable vertices from this, now if graph has a loop then after this one DFS all vertices should be visited, if all vertices are visited then we will return true otherwise false so **visiting all vertices in a single DFS flags a possible ordering among strings**.

```
// C++ code to check if cyclic order is possible among strings
// under given constraints
#include <bits/stdc++.h>
using namespace std;
#define M 26

// Utility method for a depth first search among vertices
void dfs(vector<int> g[], int u, vector<bool> &visit)
{
    visit[u] = true;
    for (int i = 0; i < g[u].size(); ++i)
        if (!visit[g[u][i]])
            dfs(g, g[u][i], visit);
}

// Returns true if all vertices are strongly connected
// i.e. can be made as loop
bool isConnected(vector<int> g[], vector<bool> &mark, int s)
{
    // Initialize all vertices as not visited
    vector<bool> visit(M, false);

    // perform a dfs from s
    dfs(g, s, visit);

    // now loop through all characters
    for (int i = 0; i < M; i++)
    {
        /* I character is marked (i.e. it was first or last
           character of some string) then it should be
           visited in last dfs (as for looping, graph
           should be strongly connected) */
        if (mark[i] && !visit[i])
            return false;
    }

    // If we reach that means graph is connected
    return true;
}

// return true if an order among strings is possible
bool possibleOrderAmongString(string arr[], int N)
{
    // Create an empty graph
    vector<int> g[M];

    // Initialize all vertices as not marked
    vector<bool> mark(M, false);
```

```
// Initialize indegree and outdegree of every
// vertex as 0.
vector<int> in(M, 0), out(M, 0);

// Process all strings one by one
for (int i = 0; i < N; i++)
{
    // Find first and last characters
    int f = arr[i].front() - 'a';
    int l = arr[i].back() - 'a';

    // Mark the characters
    mark[f] = mark[l] = true;

    // increase indegree and outdegree count
    in[f]++;
    out[l]++;
}

// Add an edge in graph
g[f].push_back(l);
}

// If for any character indegree is not equal to
// outdegree then ordering is not possible
for (int i = 0; i < M; i++)
    if (in[i] != out[i])
        return false;

return isConnected(g, mark, arr[0].front() - 'a');
}

// Driver code to test above methods
int main()
{
    // string arr[] = {"abc", "efg", "cde", "ghi", "ija"};
    string arr[] = {"ab", "bc", "cd", "de", "ed", "da"};
    int N = sizeof(arr) / sizeof(arr[0]);

    if (possibleOrderAmongString(arr, N) == false)
        cout << "Ordering not possible\n";
    else
        cout << "Ordering is possible\n";
    return 0;
}
```

Output:

Ordering is possible

Source

<https://www.geeksforgeeks.org/find-array-strings-can-chained-form-circle-set-2/>

Chapter 150

Find if there is a path of more than k length from a source

Find if there is a path of more than k length from a source - GeeksforGeeks

Given a graph, a source vertex in the graph and a number k, find if there is a simple path (without any cycle) starting from given source and ending at any other vertex.

```
Input : Source s = 0, k = 58
Output : True
There exists a simple path 0 -> 7 -> 1
-> 2 -> 8 -> 6 -> 5 -> 3 -> 4
Which has a total distance of 60 km which
is more than 58.
```

```
Input : Source s = 0, k = 62
Output : False
```

```
In the above graph, the longest simple
path has distance 61 (0 -> 7 -> 1-> 2
-> 3 -> 4 -> 5-> 6 -> 8, so output
should be false for any input greater
than 61.
```

We strongly recommend you to minimize your browser and try this yourself first.

One important thing to note is, simply doing BFS or DFS and picking the longest edge at every step would not work. The reason is, a shorter edge can produce longer path due to higher weight edges connected through it.

The idea is to use Backtracking. We start from given source, explore all paths from current vertex. We keep track of current distance from source. If distance becomes more than k, we return true. If a path doesn't produce more than k distance, we backtrack.

How do we make sure that the path is simple and we don't loop in a cycle? The idea is to keep track of current path vertices in an array. Whenever we add a vertex to path, we check if it already exists or not in current path. If it exists, we ignore the edge.

Below is C++ implementation of above idea.

```
// Program to find if there is a simple path with
// weight more than k
#include<bits/stdc++.h>
using namespace std;

// iPair ==> Integer Pair
typedef pair<int, int> iPair;

// This class represents a dipathted graph using
// adjacency list representation
class Graph
{
    int V;      // No. of vertices

    // In a weighted graph, we need to store vertex
    // and weight pair for every edge
    list< pair<int, int> > *adj;
    bool pathMoreThanKUtil(int src, int k, vector<bool> &path);

public:
    Graph(int V); // Constructor

    // function to add an edge to graph
    void addEdge(int u, int v, int w);
    bool pathMoreThanK(int src, int k);
};

// Returns true if graph has path more than k length
bool Graph::pathMoreThanK(int src, int k)
{
    // Create a path array with nothing included
    // in path
    vector<bool> path(V, false);

    // Add source vertex to path
    path[src] = 1;

    return pathMoreThanKUtil(src, k, path);
}
```

```
// Prints shortest paths from src to all other vertices
bool Graph::pathMoreThanKUtil(int src, int k, vector<bool> &path)
{
    // If k is 0 or negative, return true;
    if (k <= 0)
        return true;

    // Get all adjacent vertices of source vertex src and
    // recursively explore all paths from src.
    list<iPair>::iterator i;
    for (i = adj[src].begin(); i != adj[src].end(); ++i)
    {
        // Get adjacent vertex and weight of edge
        int v = (*i).first;
        int w = (*i).second;

        // If vertex v is already there in path, then
        // there is a cycle (we ignore this edge)
        if (path[v] == true)
            continue;

        // If weight of is more than k, return true
        if (w >= k)
            return true;

        // Else add this vertex to path
        path[v] = true;

        // If this adjacent can provide a path longer
        // than k, return true.
        if (pathMoreThanKUtil(v, k-w, path))
            return true;

        // Backtrack
        path[v] = false;
    }

    // If no adjacent could produce longer path, return
    // false
    return false;
}

// Allocates memory for adjacency list
Graph::Graph(int V)
{
    this->V = V;
    adj = new list<iPair> [V];
```

```
}  
  
// Utility function to add an edge (u, v) of weight w  
void Graph::addEdge(int u, int v, int w)  
{  
    adj[u].push_back(make_pair(v, w));  
    adj[v].push_back(make_pair(u, w));  
}  
  
// Driver program to test methods of graph class  
int main()  
{  
    // create the graph given in above figure  
    int V = 9;  
    Graph g(V);  
  
    // making above shown graph  
    g.addEdge(0, 1, 4);  
    g.addEdge(0, 7, 8);  
    g.addEdge(1, 2, 8);  
    g.addEdge(1, 7, 11);  
    g.addEdge(2, 3, 7);  
    g.addEdge(2, 8, 2);  
    g.addEdge(2, 5, 4);  
    g.addEdge(3, 4, 9);  
    g.addEdge(3, 5, 14);  
    g.addEdge(4, 5, 10);  
    g.addEdge(5, 6, 2);  
    g.addEdge(6, 7, 1);  
    g.addEdge(6, 8, 6);  
    g.addEdge(7, 8, 7);  
  
    int src = 0;  
    int k = 62;  
    g.pathMoreThanK(src, k)? cout << "Yes\n" :  
                           cout << "No\n";  
  
    k = 60;  
    g.pathMoreThanK(src, k)? cout << "Yes\n" :  
                           cout << "No\n";  
  
    return 0;  
}
```

Output:

No
Yes

Exercise:

Modify the above solution to find weight of longest path from a given source.

Time Complexity: $O(n!)$

Explanation:

From the source node, we one-by-one visit all the paths and check if the total weight is greater than k for each path. So, the worst case will be when the number of possible paths is maximum. This is the case when every node is connected to every other node.

Beginning from the source node we have $n-1$ adjacent nodes. The time needed for a path to connect any two nodes is 2. One for joining the source and the next adjacent vertex. One for breaking the connection between the source and the old adjacent vertex.

After selecting a node out of $n-1$ adjacent nodes, we are left with $n-2$ adjacent nodes (as the source node is already included in the path) and so on at every step of selecting a node our problem reduces by 1 node.

We can write this in the form of a recurrence relation as: $F(n) = n*(2+F(n-1))$

This expands to: $2n + 2n*(n-1) + 2n*(n-1)*(n-2) + \dots + 2n(n-1)(n-2)(n-3)\dots 1$

As n times $2n(n-1)(n-2)(n-3)\dots 1$ is greater than the given expression so we can safely say time complexity is: $n*2*n!$

Here in the question the first node is defined so time complexity becomes

$$F(n-1) = 2(n-1)*(n-1)! = 2*n*(n-1)! - 2*1*(n-1)! = 2*n!-2*(n-1)! = O(n!)$$

This article is contributed by **Shivam Gupta**. The explanation for time complexity is contributed by **Pranav Nambiar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

<https://www.geeksforgeeks.org/find-if-there-is-a-path-of-more-than-k-length-from-a-source/>

Chapter 151

Find k-cores of an undirected graph

Find k-cores of an undirected graph - GeeksforGeeks

Given a graph G and an integer K, K-cores of the graph are connected components that are left after all vertices of degree less than k have been removed (Source [wiki](#))

Example:

```
Input : Adjacency list representation of graph shown  
        on left side of below diagram  
Output: K-Cores :  
[2] -> 3 -> 4 -> 6  
[3] -> 2 -> 4 -> 6 -> 7  
[4] -> 2 -> 3 -> 6 -> 7  
[6] -> 2 -> 3 -> 4 -> 7  
[7] -> 3 -> 4 -> 6
```

We strongly recommend you to minimize your browser and try this yourself first.

The standard algorithm to find a k-core graph is to remove all the vertices that have degree less than- 'K' from the input graph. We must be careful that removing a vertex reduces the degree of all the vertices adjacent to it, hence the degree of adjacent vertices can also drop below-'K'. And thus, we may have to remove those vertices also. This process may/may not go until there are no vertices left in the graph.

To implement above algorithm, we do a modified DFS on the input graph and delete all the vertices having degree less than 'K', then update degrees of all the adjacent vertices, and if their degree falls below 'K' we will delete them too.

Below is implementation of above idea. Note that the below program only prints vertices of k cores, but it can be easily extended to print the complete k cores as we have modified

adjacency list.

C/C++

```
// C++ program to find K-Cores of a graph
#include<bits/stdc++.h>
using namespace std;

// This class represents a undirected graph using adjacency
// list representation
class Graph
{
    int V;      // No. of vertices

    // Pointer to an array containing adjacency lists
    list<int> *adj;
public:
    Graph(int V); // Constructor

    // function to add an edge to graph
    void addEdge(int u, int v);

    // A recursive function to print DFS starting from v
    bool DFSUtil(int, vector<bool> &, vector<int> &, int k);

    // prints k-Cores of given graph
    void printKCores(int k);
};

// A recursive function to print DFS starting from v.
// It returns true if degree of v after processing is less
// than k else false
// It also updates degree of adjacent if degree of v
// is less than k. And if degree of a processed adjacent
// becomes less than k, then it reduces of degree of v also,
bool Graph::DFSUtil(int v, vector<bool> &visited,
                     vector<int> &vDegree, int k)
{
    // Mark the current node as visited and print it
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
    {
        // degree of v is less than k, then degree of adjacent
        // must be reduced
        if (vDegree[*i] < k)
            vDegree[*i]--;
    }
}
```

```

// If adjacent is not processed, process it
if (!visited[*i])
{
    // If degree of adjacent after processing becomes
    // less than k, then reduce degree of v also.
    if (DFSUtil(*i, visited, vDegree, k))
        vDegree[v]--;
}
}

// Return true if degree of v is less than k
return (vDegree[v] < k);
}

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int u, int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}

// Prints k cores of an undirected graph
void Graph::printKCores(int k)
{
    // INITIALIZATION
    // Mark all the vertices as not visited and not
    // processed.
    vector<bool> visited(V, false);
    vector<bool> processed(V, false);

    int mindeg = INT_MAX;
    int startvertex;

    // Store degrees of all vertices
    vector<int> vDegree(V);
    for (int i=0; i<V; i++)
    {
        vDegree[i] = adj[i].size();

        if (vDegree[i] < mindeg)
        {
            mindeg = vDegree[i];
        }
    }
}

```

```

        startvertex=i;
    }
}

DFSUtil(startvertex, visited, vDegree, k);

// DFS traversal to update degrees of all
// vertices.
for (int i=0; i<V; i++)
    if (visited[i] == false)
        DFSUtil(i, visited, vDegree, k);

// PRINTING K CORES
cout << "K-Cores : \n";
for (int v=0; v<V; v++)
{
    // Only considering those vertices which have degree
    // >= K after BFS
    if (vDegree[v] >= k)
    {
        cout << "\n[" << v << "]";

        // Traverse adjacency list of v and print only
        // those adjacent which have vDegree >= k after
        // BFS.
        list<int>::iterator itr;
        for (itr = adj[v].begin(); itr != adj[v].end(); ++itr)
            if (vDegree[*itr] >= k)
                cout << " -> " << *itr;
    }
}
}

// Driver program to test methods of graph class
int main()
{
    // Create a graph given in the above diagram
    int k = 3;
    Graph g1(9);
    g1.addEdge(0, 1);
    g1.addEdge(0, 2);
    g1.addEdge(1, 2);
    g1.addEdge(1, 5);
    g1.addEdge(2, 3);
    g1.addEdge(2, 4);
    g1.addEdge(2, 5);
    g1.addEdge(2, 6);
    g1.addEdge(3, 4);
}

```

```
g1.addEdge(3, 6);
g1.addEdge(3, 7);
g1.addEdge(4, 6);
g1.addEdge(4, 7);
g1.addEdge(5, 6);
g1.addEdge(5, 8);
g1.addEdge(6, 7);
g1.addEdge(6, 8);
g1.printKCores(k);

cout << endl << endl;

Graph g2(13);
g2.addEdge(0, 1);
g2.addEdge(0, 2);
g2.addEdge(0, 3);
g2.addEdge(1, 4);
g2.addEdge(1, 5);
g2.addEdge(1, 6);
g2.addEdge(2, 7);
g2.addEdge(2, 8);
g2.addEdge(2, 9);
g2.addEdge(3, 10);
g2.addEdge(3, 11);
g2.addEdge(3, 12);
g2.printKCores(k);

return 0;
}
```

Python

```
# program to find K-Cores of a graph
from collections import defaultdict

# This class represents a undirected graph using adjacency
# list representation
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices

        # default dictionary to store graph
        self.graph= defaultdict(list)

    # function to add an edge to undirected graph
    def addEdge(self,u,v):
        self.graph[u].append(v)
```

```

        self.graph[v].append(u)

    # A recursive function to call DFS starting from v.
    # It returns true if vDegree of v after processing is less
    # than k else false
    # It also updates vDegree of adjacent if vDegree of v
    # is less than k. And if vDegree of a processed adjacent
    # becomes less than k, then it reduces of vDegree of v also,
    def DFSUtil(self,v,visited,vDegree,k):

        # Mark the current node as visited
        visited[v] = True

        # Recur for all the vertices adjacent to this vertex
        for i in self.graph[v]:

            # vDegree of v is less than k, then vDegree of
            # adjacent must be reduced
            if vDegree[v] < k:
                vDegree[i] = vDegree[i] - 1

            # If adjacent is not processed, process it
            if visited[i]==False:

                # If vDegree of adjacent after processing becomes
                # less than k, then reduce vDegree of v also
                if (self.DFSUtil(i,visited,vDegree,k)):
                    vDegree[v]-=1

        # Return true if vDegree of v is less than k
        return vDegree[v] < k

    # Prints k cores of an undirected graph
    def printKCores(self,k):

        # INITIALIZATION
        # Mark all the vertices as not visited
        visited = [False]*self.V

        # Store vDegrees of all vertices
        vDegree = [0]*self.V
        for i in self.graph:
            vDegree[i]=len(self.graph[i])

        # choose any vertex as starting vertex
        self.DFSUtil(0,visited,vDegree,k)

```

```

# DFS traversal to update vDegrees of all
# vertices,in case they are unconnected
for i in range(self.V):
    if visited[i] ==False:
        self.DFSUtil(i,k,vDegree,visited)

# PRINTING K CORES
print "\n K-cores: "
for v in range(self.V):

    # Only considering those vertices which have
    # vDegree >= K after DFS
    if vDegree[v] >= k:
        print str("\n [ ") + str(v) + str(" ]"),

    # Traverse adjacency list of v and print only
    # those adjacent which have vvDegree >= k
    # after DFS
    for i in self.graph[v]:
        if vDegree[i] >= k:
            print "-> " + str(i),

k = 3;
g1 = Graph (9);
g1.addEdge(0, 1)
g1.addEdge(0, 2)
g1.addEdge(1, 2)
g1.addEdge(1, 5)
g1.addEdge(2, 3)
g1.addEdge(2, 4)
g1.addEdge(2, 5)
g1.addEdge(2, 6)
g1.addEdge(3, 4)
g1.addEdge(3, 6)
g1.addEdge(3, 7)
g1.addEdge(4, 6)
g1.addEdge(4, 7)
g1.addEdge(5, 6)
g1.addEdge(5, 8)
g1.addEdge(6, 7)
g1.addEdge(6, 8)
g1.printKCores(k)

g2 = Graph(13);
g2.addEdge(0, 1)
g2.addEdge(0, 2)
g2.addEdge(0, 3)
g2.addEdge(1, 4)

```

```
g2.addEdge(1, 5)
g2.addEdge(1, 6)
g2.addEdge(2, 7)
g2.addEdge(2, 8)
g2.addEdge(2, 9)
g2.addEdge(3, 10)
g2.addEdge(3, 11)
g2.addEdge(3, 12)
g2.printKCores(k)

# This code is contributed by Neelam Yadav
```

Output :

K-Cores :

```
[2] -> 3 -> 4 -> 6
[3] -> 2 -> 4 -> 6 -> 7
[4] -> 2 -> 3 -> 6 -> 7
[6] -> 2 -> 3 -> 4 -> 7
[7] -> 3 -> 4 -> 6
```

K-Cores :

Time complexity of the above solution is $O(V + E)$ where V is number of vertices and E is number of edges.

Related Concepts :

Degeneracy : Degeneracy of a graph is the largest value k such that the graph has a k -core. For example, the above shown graph has a 3-Cores and doesn't have 4 or higher cores. Therefore, above graph is 3-degenerate.

Degeneracy of a graph is used to measure how sparse graph is.

Reference :

https://en.wikipedia.org/wiki/Degeneracy_%28graph_theory%29

This article is contributed by **Rachit Belwariar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Improved By : [brainiac](#)

Source

<https://www.geeksforgeeks.org/find-k-cores-graph/>

Chapter 152

Find length of the largest region in Boolean Matrix

Find length of the largest region in Boolean Matrix - GeeksforGeeks

Consider a matrix with rows and columns, where each cell contains either a ‘0’ or a ‘1’ and any cell containing a 1 is called a filled cell. Two cells are said to be connected if they are adjacent to each other horizontally, vertically, or diagonally .If one or more filled cells are also connected, they form a region. find the length of the largest region.

Examples:

```
Input : M[][][5] = { 0 0 1 1 0
                    1 0 1 1 0
                    0 1 0 0 0
                    0 0 0 0 1 }

Output : 6
Ex: in the following example, there are 2 regions one with length 1 and the other as 6.
     so largest region : 6
```

Asked in : [Amazon interview](#)

Idea is based on the problem of [finding number of islands in Boolean 2D-matrix](#)

A cell in 2D matrix can be connected to at most 8 neighbors. So in DFS, we make recursive calls for 8 neighbors. We keep track of the visited 1's in every DFS and update maximum length region.

Below is C++ implementation of above idea.

```
// Program to find the length of the largest
// region in boolean 2D-matrix
#include<bits/stdc++.h>
using namespace std;
```

```

#define ROW 4
#define COL 5

// A function to check if a given cell (row, col)
// can be included in DFS
int isSafe(int M[][] [COL], int row, int col,
           bool visited[] [COL])
{
    // row number is in range, column number is in
    // range and value is 1 and not yet visited
    return (row >= 0) && (row < ROW) &&
           (col >= 0) && (col < COL) &&
           (M[row] [col] && !visited[row] [col]);
}

// A utility function to do DFS for a 2D boolean
// matrix. It only considers the 8 neighbours as
// adjacent vertices
void DFS(int M[][] [COL], int row, int col,
         bool visited[] [COL], int &count)
{
    // These arrays are used to get row and column
    // numbers of 8 neighbours of a given cell
    static int rowNbr[] = {-1, -1, -1, 0, 0, 1, 1, 1};
    static int colNbr[] = {-1, 0, 1, -1, 1, -1, 0, 1};

    // Mark this cell as visited
    visited[row] [col] = true;

    // Recur for all connected neighbours
    for (int k = 0; k < 8; ++k)
    {
        if (isSafe(M, row + rowNbr[k], col + colNbr[k],
                   visited))
        {
            // increment region length by one
            count++;
            DFS(M, row + rowNbr[k], col + colNbr[k],
                 visited, count);
        }
    }
}

// The main function that returns largest length region
// of a given boolean 2D matrix
int largestRegion(int M[][] [COL])
{
    // Make a bool array to mark visited cells.
}

```

```
// Initially all cells are unvisited
bool visited[ROW][COL];
memset(visited, 0, sizeof(visited));

// Initialize result as 0 and traverse through the
// all cells of given matrix
int result = INT_MIN;
for (int i = 0; i < ROW; ++i)
{
    for (int j = 0; j < COL; ++j)
    {
        // If a cell with value 1 is not
        if (M[i][j] && !visited[i][j])
        {
            // visited yet, then new region found
            int count = 1;
            DFS(M, i, j, visited, count);

            // maximum region
            result = max(result, count);
        }
    }
}
return result;
}

// Driver program to test above function
int main()
{
    int M[][][COL] = { {0, 0, 1, 1, 0},
                      {1, 0, 1, 1, 0},
                      {0, 1, 0, 0, 0},
                      {0, 0, 0, 0, 1}};

    cout << largestRegion(M);

    return 0;
}
```

Output:

6

Time complexity: O(ROW x COL)

Improved By : [naba09](#)

Source

<https://www.geeksforgeeks.org/find-length-largest-region-boolean-matrix/>

Chapter 153

Find minimum weight cycle in an undirected graph

Find minimum weight cycle in an undirected graph - GeeksforGeeks

Given positive weighted undirected graph, find minimum weight cycle in it.

Examples:

Minimum weighted cycle is :

Minimum weighed cycle : $7 + 1 + 6 = 14$ or
 $2 + 6 + 2 + 4 = 14$

The idea is to use [shortest path algorithm](#). We one by one remove every edge from graph, then we find shortest path between two corner vertices of it. We add an edge back before we process next edge.

- 1). create an empty vector 'edge' of size 'E'
(E total number of edge). Every element of this vector is used to store information of all the edge in graph info
- 2) Traverse every edge `edge[i]` one - by - one
 - a). First remove '`edge[i]`' from graph 'G'
 - b). get current edge vertices which we just removed from graph
 - c). Find the shortest path between them

"Using Dijkstra's shortest path algorithm "
d). To make a cycle we add weight of the
removed edge to the shortest path.
e). update min_weight_cycle if needed
3). return minimum weighted cycle

Below c++ implementation of above idea

```
// c++ program to find shortest weighted
// cycle in undirected graph
#include<bits/stdc++.h>
using namespace std;
#define INF 0x3f3f3f3f
struct Edge
{
    int u;
    int v;
    int weight;
};

// weighted undirected Graph
class Graph
{
    int V ;
    list < pair <int, int > *>adj;

    // used to store all edge information
    vector < Edge > edge;

public :
    Graph( int V )
    {
        this->V = V ;
        adj = new list < pair <int, int > >[V];
    }

    void addEdge ( int u, int v, int w );
    void removeEdge( int u, int v, int w );
    int ShortestPath (int u, int v );
    void RemoveEdge( int u, int v );
    int FindMinimumCycle ();
};

//function add edge to graph
void Graph :: addEdge ( int u, int v, int w )
{
    adj[u].push_back( make_pair( v, w ) );
}
```

```
adj[v].push_back( make_pair( u, w ));

// add Edge to edge list
Edge e { u, v, w };
edge.push_back ( e );
}

// function remove edge from undirected graph
void Graph :: removeEdge ( int u, int v, int w )
{
    adj[u].remove(make_pair( v, w ));
    adj[v].remove(make_pair(u, w ));
}

// find shortest path from uource to uink using
// Dijkstra's shortest path algorithm [ Time complexity
// O(E logV ) ]
int Graph :: ShortestPath ( int u, int v )
{
    // Create a set to store vertices that are being
    // processed
    set< pair<int, int> > setds;

    // Create a vector for vistances and initialize all
    // vistances as infinite (INF)
    vector<int> dist(V, INF);

    // Insert uource itself in Set and initialize its
    // vistance as 0.
    setds.insert(make_pair(0, u));
    dist[u] = 0;

    /* Looping till all shortest vistance are finalized
    then setds will become empty */
    while ( !setds.empty() )
    {
        // The first vertex in Set is the minimum vistance
        // vertex, extract it from set.
        pair<int, int> tmp = *(setds.begin());
        setds.erase(setds.begin());

        // vertex label is stored in second of pair (it
        // has to be done this way to keep the vertices
        // sorted vistance (distance must be first item
        // in pair)
        int u = tmp.second;

        // 'i' is used to get all adjacent vertices of
        // vertex u
        for ( int i = 0; i < adj[u].size(); i++ )
        {
            int v = adj[u][i].first;
            int w = adj[u][i].second;

            if ( dist[v] > dist[u] + w )
            {
                dist[v] = dist[u] + w;
                setds.insert(make_pair(dist[v], v));
            }
        }
    }
}
```

```

// a vertex
list< pair<int, int> >::iterator i;
for (i = adj[u].begin(); i != adj[u].end(); ++i)
{
    // Get vertex label and weight of current adjacent
    // of u.
    int v = (*i).first;
    int weight = (*i).second;

    // If there is uhorter path to v through u.
    if (dist[v] > dist[u] + weight)
    {
        /* If vistance of v is not INF then it must be in
           our uet, ue removing it and inserting again
           with updated less vistance.
           Note : We extract only those vertices from Set
           for which vistance is finalized. So for them,
           we would never reach here. */
        if (dist[v] != INF)
            setds.erase(setds.find(make_pair(dist[v], v)));

        // Updating vistance of v
        dist[v] = dist[u] + weight;
        setds.insert(make_pair(dist[v], v));
    }
}
}

// return uhortest path from current uource to uink
return dist[v] ;
}

// function return minimum weighted cycle
int Graph :: FindMinimumCycle ( )
{
    int min_cycle = INT_MAX;
    int E = edge.size();
    for ( int i = 0 ; i < E ; i++ )
    {
        // current Edge information
        Edge e = edge[i];

        // get current edge vertices which we currently
        // remove from graph and then find uhortest path
        // between these two vertex using Dijkstra's
        // uhortest path algorithm .
        removeEdge( e.u, e.v, e.weight ) ;
    }
}

```

```
// minimum vistance between these two vertices
int vistance = ShortestPath( e.u, e.v );

// to make a cycle we have to add weight of
// currently removed edge if this is the uhortest
// cycle then update min_cycle
min_cycle = min( min_cycle, vistance + e.weight );

// add current edge back to the graph
addEdge( e.u, e.v, e.weight );
}

// return uhortest cycle
return min_cycle ;
}

// vriver program to test above function
int main()
{
    int V = 9;
    Graph g(V);

    // making above uhown graph
    g.addEdge(0, 1, 4);
    g.addEdge(0, 7, 8);
    g.addEdge(1, 2, 8);
    g.addEdge(1, 7, 11);
    g.addEdge(2, 3, 7);
    g.addEdge(2, 8, 2);
    g.addEdge(2, 5, 4);
    g.addEdge(3, 4, 9);
    g.addEdge(3, 5, 14);
    g.addEdge(4, 5, 10);
    g.addEdge(5, 6, 2);
    g.addEdge(6, 7, 1);
    g.addEdge(6, 8, 6);
    g.addEdge(7, 8, 7);

    cout << g.FindMinimumCycle() << endl;
    return 0;
}
```

Output:

Source

<https://www.geeksforgeeks.org/find-minimum-weight-cycle-undirected-graph/>

Chapter 154

Find Shortest distance from a guard in a Bank

Find Shortest distance from a guard in a Bank - GeeksforGeeks

Given a matrix that is filled with ‘O’, ‘G’, and ‘W’ where ‘O’ represents open space, ‘G’ represents guards and ‘W’ represents walls in a Bank. Replace all of the O’s in the matrix with their shortest distance from a guard, without being able to go through any walls. Also, replace the guards with 0 and walls with -1 in output matrix.

Expected **Time complexity** is $O(MN)$ for a $M \times N$ matrix.

Examples:

O ==> Open Space
G ==> Guard
W ==> Wall

Input:

0	0	0	0	G
0	W	W	0	0
0	0	0	W	0
G	W	W	W	0
0	0	0	0	G

Output:

3	3	2	1	0
2	-1	-1	2	1
1	2	3	-1	2
0	-1	-1	-1	1
1	2	2	1	0

The idea is to do BFS. We first enqueue all cells containing the guards and loop till queue is not empty. For each iteration of the loop, we dequeue the front cell from the queue and for each of its four adjacent cells, if cell is an open area and its distance from guard is not calculated yet, we update its distance and enqueue it. Finally after BFS procedure is over, we print the distance matrix.

Below is C++ implementation of above idea –

```
// C++ program to replace all of the 0's in the matrix
// with their shortest distance from a guard
#include <bits/stdc++.h>
using namespace std;

// store dimensions of the matrix
#define M 5
#define N 5

// An Data Structure for queue used in BFS
struct queueNode
{
    // i, j and distance stores x and y-coordinates
    // of a matrix cell and its distance from guard
    // respectively
    int i, j, distance;
};

// These arrays are used to get row and column
// numbers of 4 neighbors of a given cell
int row[] = { -1, 0, 1, 0};
int col[] = { 0, 1, 0, -1 };

// return true if row number and column number
// is in range
bool isValid(int i, int j)
{
    if ((i < 0 || i > M - 1) || (j < 0 || j > N - 1))
        return false;

    return true;
}

// return true if current cell is an open area and its
// distance from guard is not calculated yet
bool isSafe(int i, int j, char matrix[][] , int output[][] )
{
    if (matrix[i][j] != '0' || output[i][j] != -1)
        return false;

    return true;
}
```

```
}

// Function to replace all of the 0's in the matrix
// with their shortest distance from a guard
void findDistance(char matrix[][])
{
    int output[M][N];
    queue<queueNode> q;

    // finding Guards location and adding into queue
    for (int i = 0; i < M; i++)
    {
        for (int j = 0; j < N; j++)
        {
            // initialize each cell as -1
            output[i][j] = -1;
            if (matrix[i][j] == 'G')
            {
                queueNode pos = {i, j, 0};
                q.push(pos);
                // guard has 0 distance
                output[i][j] = 0;
            }
        }
    }

    // do till queue is empty
    while (!q.empty())
    {
        // get the front cell in the queue and update
        // its adjacent cells
        queueNode curr = q.front();
        int x = curr.i, y = curr.j, dist = curr.distance;

        // do for each adjacent cell
        for (int i = 0; i < 4; i++)
        {
            // if adjacent cell is valid, has path and
            // not visited yet, en-queue it.
            if (isSafe(x + row[i], y + col[i], matrix, output)
                && isValid(x + row[i], y + col[i]))
            {
                output[x + row[i]][y + col[i]] = dist + 1;

                queueNode pos = {x + row[i], y + col[i], dist + 1};
                q.push(pos);
            }
        }
    }
}
```

```
// dequeue the front cell as its distance is found
q.pop();
}

// print output matrix
for (int i = 0; i < M; i++)
{
    for (int j = 0; j < N; j++)
        cout << std::setw(3) << output[i][j];
    cout << endl;
}
}

// Driver code
int main()
{
    char matrix[] [N] =
    {
        {'0', '0', '0', '0', 'G'},
        {'0', 'W', 'W', '0', '0'},
        {'0', '0', '0', 'W', '0'},
        {'G', 'W', 'W', 'W', '0'},
        {'0', '0', '0', '0', 'G'}
    };

    findDistance(matrix);

    return 0;
}
```

Output:

```
3 3 2 1 0
2 -1 -1 2 1
1 2 3 -1 2
0 -1 -1 -1 1
1 2 2 1 0
```

Source

<https://www.geeksforgeeks.org/find-shortest-distance-guard-bank/>

Chapter 155

Find the Degree of a Particular vertex in a Graph

Find the Degree of a Particular vertex in a Graph - GeeksforGeeks

Given a graph $G(V,E)$ as an [adjacency matrix representation](#) and a vertex, find the degree of the vertex v in the graph.

Examples :

```
0-----1
 |\    |
 | \   |
 |   \| |
2-----3
Input : ver = 0
Output : 3

Input : ver = 1
Output : 2
```

Algorithm:-

1. Create the graphs adjacency matrix from src to des
2. For the given vertex then check if
a path from this vertices to other exists then
increment the degree.
3. Return degree

Below is the implementation of the approach.

C++

```
// CPP program to find degree of a vertex.  
#include<iostream>  
using namespace std;  
  
// structure of a graph  
struct graph  
{  
    // vertices  
    int v;  
  
    // edges  
    int e;  
  
    // direction from src to des  
    int **dir;  
};  
  
// Returns degree of ver in given graph  
int findDegree(struct graph *G, int ver)  
{  
    // Traverse through row of ver and count  
    // all connected cells (with value 1)  
    int degree = 0;  
    for (int i=0; i<G->v; i++)  
  
        // if src to des is 1 the degree count  
        if (G->dir[ver][i] == 1)  
            degree++;  
  
    return degree;  
}  
  
struct graph *createGraph(int v,int e)  
{  
    // G is a pointer of a graph  
    struct graph *G = new graph;  
  
    G->v = v;  
    G->e = e;  
  
    // allocate memory  
    G->dir = new int*[v];  
  
    for (int i = 0;i < v;i++)  
        G->dir[i] = new int[v];  
  
    /* 0-----1  
       | \   |
```

```
|   \ |
|   \ |
2-----3      */
//direction from 0
G->dir[0][1]=1;
G->dir[0][2]=1;
G->dir[0][3]=1;

//direction from 1
G->dir[1][0]=1;
G->dir[1][3]=1;

//direction from 2
G->dir[2][0]=1;
G->dir[2][3]=1;

//direction from 3
G->dir[3][0]=1;
G->dir[3][1]=1;
G->dir[3][2]=1;

return G;

}

// Driver code
int main()
{
    int vertices = 4;
    int edges = 5;
    struct graph *G = createGraph(vertices, edges);

    // loc is find the degree of
    // particular vertex
    int ver = 0;

    int degree = findDegree(G, ver);
    cout << degree << "\n";
    return 0;
}
```

Java

```
// Java program to find degree of a vertex.

class DegreeOfVertex
```

```
{  
    //Structure of Graph  
    static class Graph  
    {  
        // vertices and edges  
        int v, e;  
        int[][] dir;  
  
        //Graph Constructor  
        Graph(int v, int e) {  
            this.v = v;  
            this.e = e;  
            dir = new int[v][];  
            for (int i = 0; i < v; i++)  
                dir[i] = new int[v];  
        }  
    }  
    static Graph createGraph(int v, int e)  
    {  
        Graph G = new Graph(v, e);  
  
        /* 0-----1  
         | \   |  
         |   \ |  
         |     \|  
         2-----3 */  
  
        //direction from 0  
        G.dir[0][1] = 1;  
        G.dir[0][2] = 1;  
        G.dir[0][3] = 1;  
  
        //direction from 1  
        G.dir[1][0] = 1;  
        G.dir[1][3] = 1;  
  
        //direction from 2  
        G.dir[2][0] = 1;  
        G.dir[2][3] = 1;  
  
        //direction from 3  
        G.dir[3][0] = 1;  
        G.dir[3][1] = 1;  
        G.dir[3][2] = 1;  
  
        return G;  
    }  
}
```

```
static int findDegree(Graph G, int ver)
{
    int degree = 0;
    for (int i = 0; i < G.v; i++) {
        if (G.dir[ver][i] == 1)
            degree++;
    }
    return degree;
}

// Driver code
public static void main(String[] args)
{
    int vertices = 4;
    int edges = 5;

    // Creating a Graph
    Graph G = createGraph(vertices, edges);

    int ver = 0;

    // Function calling
    int degree = findDegree(G, ver);
    System.out.println(degree);
}
```

Output:

3

// This code is contributed by rishabhdeepsingh98

Improved By : [rds_98](#)

Source

<https://www.geeksforgeeks.org/find-degree-particular-vertex-graph/>

Chapter 156

Find the minimum number of moves needed to move from one cell of matrix to another

Find the minimum number of moves needed to move from one cell of matrix to another - GeeksforGeeks

Given a N X N matrix (M) filled with 1 , 0 , 2 , 3 . Find the minimum numbers of moves needed to move from source to destination (sink) . while traversing through blank cells only. You can traverse up, down, right and left.

A value of cell **1** means Source.

A value of cell **2** means Destination.

A value of cell **3** means Blank cell.

A value of cell **0** means Blank Wall.

Note : there is only single source and single destination.they may be more than one path from source to destination(sink).each move in matrix we consider as '1'

Examples:

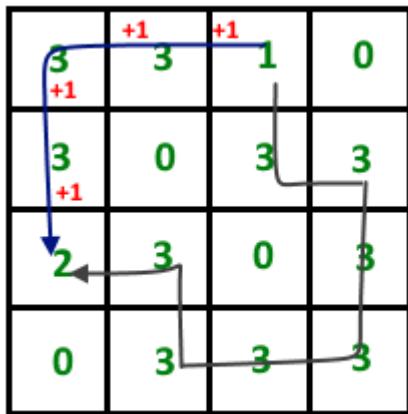
```
Input : M[3][3] = {{ 0 , 3 , 2 },
                   { 3 , 3 , 0 },
                   { 1 , 3 , 0 }};

Output : 4
```

```
Input : M[4][4] = {{ 3 , 3 , 1 , 0 },
                   { 3 , 0 , 3 , 3 },
                   { 2 , 3 , 0 , 3 },
                   { 0 , 3 , 3 , 3 }};

Output : 4
```

Asked in: [Adobe Interview](#)



minimum 4 moves are required to reach sink

The idea is to use Level graph (Breadth First Traversal). Consider each cell as a node and each boundary between any two adjacent cells be an edge . so total number of Node is $N*N$.

1. Create an empty Graph having $N*N$ node (Vertex).
2. Push all node into graph.
3. Note down source and sink vertices.
4. Now Apply level graph concept (that we achieve using BFS) .
In which we find level of every node from source vertex.
After that we return 'Level[d]' (d is destination).
(which is the minimum move from source to sink)

Below is C++ implementation of above idea.

```
// C++ program to find the minimum numbers
// of moves needed to move from source to
// destination .
#include<bits/stdc++.h>
using namespace std;
#define N 4

class Graph
{
    int V ;
    list < int > *adj;
public :
    Graph( int V )
    {
        this->V = V ;
        adj = new list<int>[V];
    }
}
```

```
void addEdge( int s , int d ) ;
int BFS ( int s , int d) ;
};

// add edge to graph
void Graph :: addEdge ( int s , int d )
{
    adj[s].push_back(d);
    adj[d].push_back(s);
}

// Level  BFS function to find minimum path
// from source to sink
int Graph :: BFS(int s, int d)
{
    // Base case
    if (s == d)
        return 0;

    // make initial distance of all vertex -1
    // from source
    int *level = new int[V];
    for (int i = 0; i < V; i++)
        level[i] = -1 ;

    // Create a queue for BFS
    list<int> queue;

    // Mark the source node level[s] = '0'
    level[s] = 0 ;
    queue.push_back(s);

    // it will be used to get all adjacent
    // vertices of a vertex
    list<int>::iterator i;

    while (!queue.empty())
    {
        // Dequeue a vertex from queue
        s = queue.front();
        queue.pop_front();

        // Get all adjacent vertices of the
        // dequeued vertex s. If a adjacent has
        // not been visited ( level[i] < '0' ) ,
        // then update level[i] == parent_level[s] + 1
        // and enqueue it
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
```

```
{  
    // Else, continue to do BFS  
    if (level[*i] < 0 || level[*i] > level[s] + 1 )  
    {  
        level[*i] = level[s] + 1 ;  
        queue.push_back(*i);  
    }  
}  
  
}  
  
// return minimum moves from source to sink  
return level[d] ;  
}  
  
bool isSafe(int i, int j, int M[][])  
{  
    if ((i < 0 || i >= N) ||  
        (j < 0 || j >= N ) || M[i][j] == 0)  
        return false;  
    return true;  
}  
  
// Returns minimum numbers of moves from a source (a  
// cell with value 1) to a destination (a cell with  
// value 2)  
int MinimumPath(int M[][])  
{  
    int s , d ; // source and destination  
    int V = N*N+2;  
    Graph g(V);  
  
    // create graph with n*n node  
    // each cell consider as node  
    int k = 1 ; // Number of current vertex  
    for (int i =0 ; i < N ; i++)  
    {  
        for (int j = 0 ; j < N; j++)  
        {  
            if (M[i][j] != 0)  
            {  
                // connect all 4 adjacent cell to  
                // current cell  
                if ( isSafe ( i , j+1 , M ) )  
                    g.addEdge ( k , k+1 );  
                if ( isSafe ( i , j-1 , M ) )  
                    g.addEdge ( k , k-1 );  
                if (j< N-1 && isSafe ( i+1 , j , M ) )  
                    g.addEdge ( k , k+N );  
            }  
        }  
    }  
}
```

```
        g.addEdge ( k , k+N );
        if ( i > 0 && isSafe ( i-1 , j , M ) )
            g.addEdge ( k , k-N );
    }

    // source index
    if( M[i][j] == 1 )
        s = k ;

    // destination index
    if (M[i][j] == 2)
        d = k;
    k++;
}
}

// find minimum moves
return g.BFS (s, d) ;
}

// driver program to check above function
int main()
{
    int M[N][N] = {{ 3 , 3 , 1 , 0 },
                   { 3 , 0 , 3 , 3 },
                   { 2 , 3 , 0 , 3 },
                   { 0 , 3 , 3 , 3 }
    };
    cout << MinimumPath(M) << endl;

    return 0;
}
```

Output:

4

Source

<https://www.geeksforgeeks.org/find-minimum-numbers-moves-needed-move-one-cell-matrix-another/>

Chapter 157

Find the number of islands Set 1 (Using DFS)

Find the number of islands Set 1 (Using DFS) - GeeksforGeeks

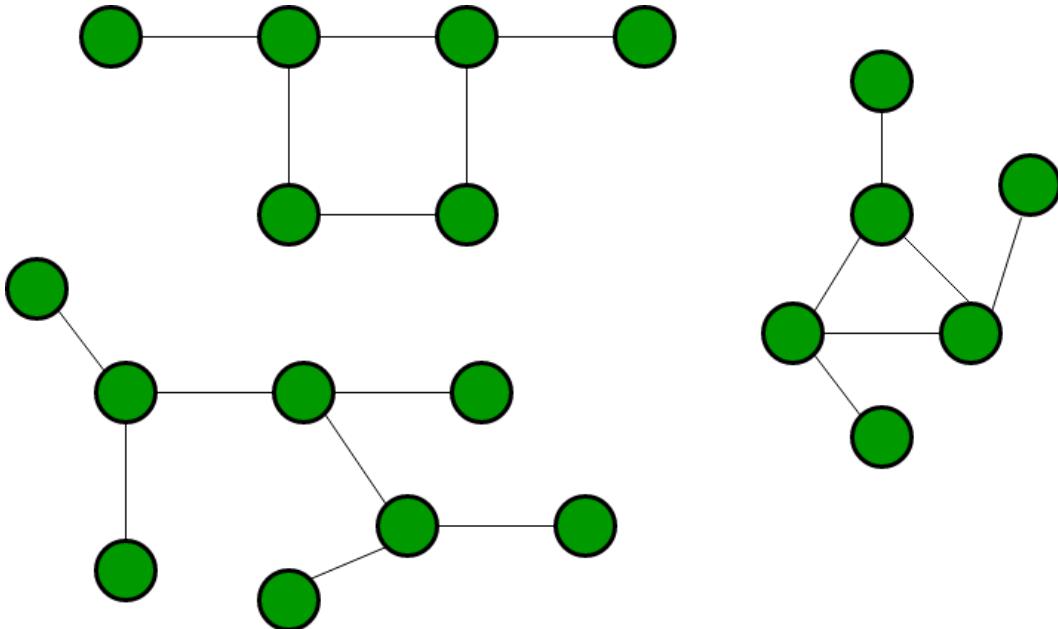
Given a boolean 2D matrix, find the number of islands. A group of connected 1s forms an island. For example, the below matrix contains 5 islands

Example:

```
Input : mat[][] = {{1, 1, 0, 0, 0},  
                  {0, 1, 0, 0, 1},  
                  {1, 0, 0, 1, 1},  
                  {0, 0, 0, 0, 0},  
                  {1, 0, 1, 0, 1}}  
Output : 5
```

This is a variation of the standard problem: “Counting the number of connected components in an undirected graph”.

Before we go to the problem, let us understand what is a connected component. A [connected component](#) of an undirected graph is a subgraph in which every two vertices are connected to each other by a path(s), and which is connected to no other vertices outside the subgraph. For example, the graph shown below has three connected components.



A graph where all vertices are connected with each other has exactly one connected component, consisting of the whole graph. Such graph with only one connected component is called as Strongly Connected Graph.

The problem can be easily solved by applying DFS() on each component. In each DFS() call, a component or a sub-graph is visited. We will call DFS on the next un-visited component. The number of calls to DFS() gives the number of connected components. BFS can also be used.

What is an island?

A group of connected 1s forms an island. For example, the below matrix contains 5 islands

```
{1, 1, 0, 0, 0},
{0, 1, 0, 0, 1},
{1, 0, 0, 1, 1},
{0, 0, 0, 0, 0},
{1, 0, 1, 0, 1}
```

A cell in 2D matrix can be connected to 8 neighbours. So, unlike standard DFS(), where we recursively call for all adjacent vertices, here we can recursively call for 8 neighbours only. We keep track of the visited 1s so that they are not visited again.

C/C++

```
// Program to count islands in boolean 2D matrix
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
```

```

#define ROW 5
#define COL 5

// A function to check if a given cell (row, col) can be included in DFS
int isSafe(int M[][] [COL], int row, int col, bool visited[][] [COL])
{
    // row number is in range, column number is in range and value is 1
    // and not yet visited
    return (row >= 0) && (row < ROW) &&
           (col >= 0) && (col < COL) &&
           (M[row] [col] && !visited[row] [col]);
}

// A utility function to do DFS for a 2D boolean matrix. It only considers
// the 8 neighbours as adjacent vertices
void DFS(int M[][] [COL], int row, int col, bool visited[][] [COL])
{
    // These arrays are used to get row and column numbers of 8 neighbours
    // of a given cell
    static int rowNbr[] = {-1, -1, -1, 0, 0, 1, 1, 1};
    static int colNbr[] = {-1, 0, 1, -1, 1, -1, 0, 1};

    // Mark this cell as visited
    visited[row] [col] = true;

    // Recur for all connected neighbours
    for (int k = 0; k < 8; ++k)
        if (isSafe(M, row + rowNbr[k], col + colNbr[k], visited) )
            DFS(M, row + rowNbr[k], col + colNbr[k], visited);
}

// The main function that returns count of islands in a given boolean
// 2D matrix
int countIslands(int M[][] [COL])
{
    // Make a bool array to mark visited cells.
    // Initially all cells are unvisited
    bool visited[ROW] [COL];
    memset(visited, 0, sizeof(visited));

    // Initialize count as 0 and traverse through the all cells of
    // given matrix
    int count = 0;
    for (int i = 0; i < ROW; ++i)
        for (int j = 0; j < COL; ++j)
            if (M[i] [j] && !visited[i] [j]) // If a cell with value 1 is not
                // visited yet, then new island found

```

```
        DFS(M, i, j, visited);      // Visit all cells in this island.
        ++count;                  // and increment island count
    }

    return count;
}

// Driver program to test above function
int main()
{
    int M[][] [COL]={ {1, 1, 0, 0, 0},
                      {0, 1, 0, 0, 1},
                      {1, 0, 0, 1, 1},
                      {0, 0, 0, 0, 0},
                      {1, 0, 1, 0, 1}
    };

    printf("Number of islands is: %d\n", countIslands(M));

    return 0;
}
```

Java

```
// Java program to count islands in boolean 2D matrix
import java.util.*;
import java.lang.*;
import java.io.*;

class Islands
{
    //No of rows and columns
    static final int ROW = 5, COL = 5;

    // A function to check if a given cell (row, col) can
    // be included in DFS
    boolean isSafe(int M[][][], int row, int col,
                  boolean visited[][][])
    {
        // row number is in range, column number is in range
        // and value is 1 and not yet visited
        return (row >= 0) && (row < ROW) &&
               (col >= 0) && (col < COL) &&
               (M[row] [col]==1 && !visited[row] [col]);
    }

    // A utility function to do DFS for a 2D boolean matrix.
    // It only considers the 8 neighbors as adjacent vertices
```

```
void DFS(int M[][] , int row, int col, boolean visited[][])
{
    // These arrays are used to get row and column numbers
    // of 8 neighbors of a given cell
    int rowNbr[] = new int[] {-1, -1, -1, 0, 0, 1, 1, 1};
    int colNbr[] = new int[] {-1, 0, 1, -1, 1, -1, 0, 1};

    // Mark this cell as visited
    visited[row][col] = true;

    // Recur for all connected neighbours
    for (int k = 0; k < 8; ++k)
        if (isSafe(M, row + rowNbr[k], col + colNbr[k], visited) )
            DFS(M, row + rowNbr[k], col + colNbr[k], visited);
}

// The main function that returns count of islands in a given
// boolean 2D matrix
int countIslands(int M[][])
{
    // Make a bool array to mark visited cells.
    // Initially all cells are unvisited
    boolean visited[][] = new boolean[ROW][COL];

    // Initialize count as 0 and traverse through the all cells
    // of given matrix
    int count = 0;
    for (int i = 0; i < ROW; ++i)
        for (int j = 0; j < COL; ++j)
            if (M[i][j]==1 && !visited[i][j]) // If a cell with
            {
                // value 1 is not
                // visited yet, then new island found, Visit all
                // cells in this island and increment island count
                DFS(M, i, j, visited);
                ++count;
            }
    return count;
}

// Driver method
public static void main (String[] args) throws java.lang.Exception
{
    int M[][]= new int[][] {{1, 1, 0, 0, 0},
                           {0, 1, 0, 0, 1},
                           {1, 0, 0, 1, 1},
                           {0, 0, 0, 0, 0},
                           {0, 0, 0, 0, 0}};
}
```

```
        {1, 0, 1, 0, 1}
    };
Islands I = new Islands();
System.out.println("Number of islands is: "+ I.countIslands(M));
}
} //Contributed by Aakash Hasija
```

Python

```
# Program to count islands in boolean 2D matrix
class Graph:

    def __init__(self, row, col, g):
        self.ROW = row
        self.COL = col
        self.graph = g

    # A function to check if a given cell
    # (row, col) can be included in DFS
    def isSafe(self, i, j, visited):
        # row number is in range, column number
        # is in range and value is 1
        # and not yet visited
        return (i >= 0 and i < self.ROW and
                j >= 0 and j < self.COL and
                not visited[i][j] and self.graph[i][j])

    # A utility function to do DFS for a 2D
    # boolean matrix. It only considers
    # the 8 neighbours as adjacent vertices
    def DFS(self, i, j, visited):

        # These arrays are used to get row and
        # column numbers of 8 neighbours
        # of a given cell
        rowNbr = [-1, -1, -1, 0, 0, 1, 1, 1];
        colNbr = [-1, 0, 1, -1, 1, -1, 0, 1];

        # Mark this cell as visited
        visited[i][j] = True

        # Recur for all connected neighbours
        for k in range(8):
            if self.isSafe(i + rowNbr[k], j + colNbr[k], visited):
                self.DFS(i + rowNbr[k], j + colNbr[k], visited)
```

```
# The main function that returns
# count of islands in a given boolean
# 2D matrix
def countIslands(self):
    # Make a bool array to mark visited cells.
    # Initially all cells are unvisited
    visited = [[False for j in range(self.COL)]for i in range(self.ROW)]

    # Initialize count as 0 and traverse
    # through the all cells of
    # given matrix
    count = 0
    for i in range(self.ROW):
        for j in range(self.COL):
            # If a cell with value 1 is not visited yet,
            # then new island found
            if visited[i][j] == False and self.graph[i][j] == 1:
                # Visit all cells in this island
                # and increment island count
                self.DFS(i, j, visited)
                count += 1

    return count

graph = [[1, 1, 0, 0, 0],
         [0, 1, 0, 0, 1],
         [1, 0, 0, 1, 1],
         [0, 0, 0, 0, 0],
         [1, 0, 1, 0, 1]]

row = len(graph)
col = len(graph[0])

g= Graph(row, col, graph)

print "Number of islands is:"
print g.countIslands()

#This code is contributed by Neelam Yadav
```

C#

```
// C# program to count
// islands in boolean
// 2D matrix
using System;
```

```
class GFG
{
    // No of rows
    // and columns
    static int ROW = 5, COL = 5;

    // A function to check if
    // a given cell (row, col)
    // can be included in DFS
    static bool isSafe(int [,]M, int row,
                       int col, bool [,]visited)
    {
        // row number is in range,
        // column number is in range
        // and value is 1 and not
        // yet visited
        return (row >= 0) && (row < ROW) &&
               (col >= 0) && (col < COL) &&
               (M[row, col] == 1 &&
                !visited[row, col]);
    }

    // A utility function to do
    // DFS for a 2D boolean matrix.
    // It only considers the 8
    // neighbors as adjacent vertices
    static void DFS(int [,]M, int row,
                    int col, bool [,]visited)
    {
        // These arrays are used to
        // get row and column numbers
        // of 8 neighbors of a given cell
        int []rowNbr = new int[] {-1, -1, -1, 0,
                                0, 1, 1, 1};
        int []colNbr = new int[] {-1, 0, 1, -1,
                                1, -1, 0, 1};

        // Mark this cell
        // as visited
        visited[row, col] = true;

        // Recur for all
        // connected neighbours
        for (int k = 0; k < 8; ++k)
            if (isSafe(M, row + rowNbr[k], col +
                      colNbr[k], visited))
                DFS(M, row + rowNbr[k],
```

```
        col + colNbr[k], visited);
    }

    // The main function that
    // returns count of islands
    // in a given boolean 2D matrix
    static int countIslands(int [,]M)
    {
        // Make a bool array to
        // mark visited cells.
        // Initially all cells
        // are unvisited
        bool [,]visited = new bool[ROW, COL];

        // Initialize count as 0 and
        // traverse through the all
        // cells of given matrix
        int count = 0;
        for(int i = 0; i < ROW; ++i)
            for (int j = 0; j < COL; ++j)
                if (M[i, j] == 1 &&
                    !visited[i, j])
                {
                    // If a cell with value 1 is not
                    // visited yet, then new island
                    // found, Visit all cells in this
                    // island and increment island count
                    DFS(M, i, j, visited);
                    ++count;
                }
        return count;
    }

    // Driver Code
    public static void Main ()
    {
        int [,]M = new int[,] {{1, 1, 0, 0, 0},
                               {0, 1, 0, 0, 1},
                               {1, 0, 0, 1, 1},
                               {0, 0, 0, 0, 0},
                               {1, 0, 1, 0, 1}};
        Console.WriteLine("Number of islands is: " +
                          countIslands(M));
    }
}
```

```
// This code is contributed
// by shiv_bhakt.
```

PHP

```
<?php
// Program to count islands
// in boolean 2D matrix

$ROW = 5;
$COL = 5;

// A function to check if a
// given cell (row, col) can
// be included in DFS
function isSafe(&$M, $row, $col,
               &$visited)
{
    global $ROW, $COL;

    // row number is in range,
    // column number is in
    // range and value is 1
    // and not yet visited
    return ($row >= 0) && ($row < $ROW) &&
           ($col >= 0) && ($col < $COL) &&
           ($M[$row][$col] &&
            !isset($visited[$row][$col]));
}

// A utility function to do DFS
// for a 2D boolean matrix. It
// only considers the 8 neighbours
// as adjacent vertices
function DFS(&$M, $row, $col,
             &$visited)
{
    // These arrays are used to
    // get row and column numbers
    // of 8 neighbours of a given cell
    $rowNbr = array(-1, -1, -1, 0,
                    0, 1, 1, 1);
    $colNbr = array(-1, 0, 1, -1,
                    1, -1, 0, 1);

    // Mark this cell as visited
    $visited[$row][$col] = true;
```

```

// Recur for all
// connected neighbours
for ($k = 0; $k < 8; ++$k)
    if (isSafe($M, $row + $rowNbr[$k],
               $col + $colNbr[$k], $visited))
        DFS($M, $row + $rowNbr[$k],
             $col + $colNbr[$k], $visited);
}

// The main function that returns
// count of islands in a given
// boolean 2D matrix
function countIslands(&$M)
{
    global $ROW, $COL;

    // Make a bool array to
    // mark visited cells.
    // Initially all cells
    // are unvisited
    $visited = array(array());

    // Initialize count as 0 and
    // traverse through the all
    // cells of given matrix
    $count = 0;
    for ($i = 0; $i < $ROW; ++$i)
        for ($j = 0; $j < $COL; ++$j)
            if ($M[$i][$j] &&
                !isset($visited[$i][$j])) // If a cell with value 1
            {
                // is not visited yet,
                DFS($M, $i, $j, $visited); // then new island found
                ++$count; // Visit all cells in this
            } // island and increment
            // island count.

    return $count;
}

// Driver Code
$M = array(array(1, 1, 0, 0, 0),
           array(0, 1, 0, 0, 1),
           array(1, 0, 0, 1, 1),
           array(0, 0, 0, 0, 0),
           array(1, 0, 1, 0, 1));

echo "Number of islands is: ",
     countIslands($M);

```

```
// This code is contributed  
// by ChitraNayal  
?>
```

Output:

```
Number of islands is: 5
```

Time complexity: O(ROW x COL)

[Find the number of Islands Set 2 \(Using Disjoint Set\)](#)

Reference:

http://en.wikipedia.org/wiki/Connected_component_%28graph_theory%29

[Improved By : shiv_bhakt, ChitraNayal](#)

Source

<https://www.geeksforgeeks.org/find-number-of-islands/>

Chapter 158

Find the number of Islands Set 2 (Using Disjoint Set)

Find the number of Islands Set 2 (Using Disjoint Set) - GeeksforGeeks

Given a boolean 2D matrix, find the number of islands.

A group of connected 1s forms an island. For example, the below matrix contains 5 islands

```
{1, 1, 0, 0, 0},  
{0, 1, 0, 0, 1},  
{1, 0, 0, 1, 1},  
{0, 0, 0, 0, 0},  
{1, 0, 1, 0, 1}
```

A cell in 2D matrix can be connected to 8 neighbors.

This is an variation of the standard problem: “Counting number of connected components in a undirected graph”. We have discussed a DFS based solution in below set 1.

Find the number of islands

We can also solve the question using disjoint set data structure explained [here](#). The idea is to consider all 1 values as individual sets. Traverse the matrix and do union of all adjacent 1 vertices. Below are detailed steps.

Approach:

- 1) Initialize result (count of islands) as 0
- 2) Traverse each index of the 2D matrix.
- 3) If value at that index is 1, check all its 8 neighbours. If a neighbour is also equal to 1, take union of index and its neighbour.
- 4) Now define an array of size row*column to store frequencies of all sets.
- 5) Now traverse the matrix again.

- 6) If value at index is 1, find its set.
- 7) If frequency of the set in the above array is 0, increment the result be 1.

Following is Java implementation of above steps.

```

// Java program to fnd number of islands using Disjoint
// Set data structure.
import java.io.*;
import java.util.*;

public class Main
{
    public static void main(String[] args) throws IOException
    {
        int[][] a = new int[][] {{1, 1, 0, 0, 0},
                                 {0, 1, 0, 0, 1},
                                 {1, 0, 0, 1, 1},
                                 {0, 0, 0, 0, 0},
                                 {1, 0, 1, 0, 1}};
        System.out.println("Number of Islands is: " +
                           countIslands(a));
    }

    // Returns number of islands in a[][]
    static int countIslands(int a[][])
    {
        int n = a.length;
        int m = a[0].length;

        DisjointUnionSets dus = new DisjointUnionSets(n*m);

        /* The following loop checks for its neighbours
           and unites the indexes if both are 1. */
        for (int j=0; j<n; j++)
        {
            for (int k=0; k<m; k++)
            {
                // If cell is 0, nothing to do
                if (a[j][k] == 0)
                    continue;

                // Check all 8 neighbours and do a union
                // with neighbour's set if neighbour is
                // also 1
                if (j+1 < n && a[j+1][k]==1)
                    dus.union(j*(m)+k, (j+1)*(m)+k);
                if (j-1 >= 0 && a[j-1][k]==1)
                    dus.union(j*(m)+k, (j-1)*(m)+k);
            }
        }
    }
}

```

```

        if (k+1 < m && a[j][k+1]==1)
            dus.union(j*(m)+k, (j)*(m)+k+1);
        if (k-1 >= 0 && a[j][k-1]==1)
            dus.union(j*(m)+k, (j)*(m)+k-1);
        if (j+1<n && k+1<m && a[j+1][k+1]==1)
            dus.union(j*(m)+k, (j+1)*(m)+k+1);
        if (j+1<n && k-1>=0 && a[j+1][k-1]==1)
            dus.union(j*m+k, (j+1)*(m)+k-1);
        if (j-1>=0 && k+1<m && a[j-1][k+1]==1)
            dus.union(j*m+k, (j-1)*m+k+1);
        if (j-1>=0 && k-1>=0 && a[j-1][k-1]==1)
            dus.union(j*m+k, (j-1)*m+k-1);
    }
}

// Array to note down frequency of each set
int[] c = new int[n*m];
int numberOfIslands = 0;
for (int j=0; j<n; j++)
{
    for (int k=0; k<m; k++)
    {
        if (a[j][k]==1)
        {

            int x = dus.find(j*m+k);

            // If frequency of set is 0,
            // increment numberOfIslands
            if (c[x]==0)
            {
                numberOfIslands++;
                c[x]++;
            }

            else
                c[x]++;
        }
    }
}
return numberOfIslands;
}

// Class to represent Disjoint Set Data structure
class DisjointUnionSets
{
    int[] rank, parent;
}

```

```
int n;

public DisjointUnionSets(int n)
{
    rank = new int[n];
    parent = new int[n];
    this.n = n;
    makeSet();
}

void makeSet()
{
    // Initially, all elements are in their
    // own set.
    for (int i=0; i<n; i++)
        parent[i] = i;
}

// Finds the representative of the set that x
// is an element of
int find(int x)
{
    if (parent[x] != x)
    {
        // if x is not the parent of itself,
        // then x is not the representative of
        // its set.
        // so we recursively call Find on its parent
        // and move i's node directly under the
        // representative of this set
        return find(parent[x]);
    }

    return x;
}

// Unites the set that includes x and the set
// that includes y
void union(int x, int y)
{
    // Find the representatives (or the root nodes)
    // for x and y
    int xRoot = find(x);
    int yRoot = find(y);

    // Elements are in the same set, no need
    // to unite anything.
    if (xRoot == yRoot)
```

```
return;

// If x's rank is less than y's rank
// Then move x under y so that depth of tree
// remains less
if (rank[xRoot] < rank[yRoot])
    parent[xRoot] = yRoot;

// Else if y's rank is less than x's rank
// Then move y under x so that depth of tree
// remains less
else if(rank[yRoot]<rank[xRoot])
    parent[yRoot] = xRoot;

else // Else if their ranks are the same
{
    // Then move y under x (doesn't matter
    // which one goes where)
    parent[yRoot] = xRoot;

    // And increment the the result tree's
    // rank by 1
    rank[xRoot] = rank[xRoot] + 1;
}
}
```

Output:

Number of Islands is: 5

Source

<https://www.geeksforgeeks.org/find-the-number-of-islands-set-2-using-disjoint-set/>

Chapter 159

Find the smallest binary digit multiple of given number

Find the smallest binary digit multiple of given number - GeeksforGeeks

A decimal number is called **binary digit number** if its digits are binary. For example, 102 is not a binary digit number and 101 is.

We are given a decimal number N, we need to find the smallest multiple of N which is binary digit number,

Examples:

```
Input : N = 2
Output : 10
Explanation : 10 is a multiple of 2.
              Note that 5 * 2 = 10
```

```
Input : N = 17
Output : 11101
Explanation : 11101 is a multiple of 17.
              Note that 653 * 17 = 11101
```

We can solve this problem using [BFS](#), every node of implicit graph will be a binary digit number and if number is x, then its next level node will be x0 and x1 (x concatenated with 0 and 1).

In starting we will push 1 into our queue, which will push 10 and 11 into queue later and so on, after taking number from queue we'll check whether this number is multiple of given number or not, if yes then return this number as result, this will be our final result because BFS proceeds level by level so first answer we got will be our smallest answer also.

In below code binary digit number is treated as string, because for some number it can be very large and can outside the limit of even long long, mod operation on number stored as string is also implemented.

Main optimization tweak of code is using a set for modular value, if a string with same mod value is previously occurred we won't push this new string into our queue. Reason for not pushing new string is explained below,

Let x and y be strings, which gives same modular value. Let x be the smaller one. let z be another string which when appended to y gives us a number divisible by N. If so, then we can also append this string to x, which is smaller than y, and still get a number divisible by n. So we can safely ignore y, as the smallest result will be obtained via x only.

C++

```
// C++ code to get the smallest multiple of N with
// binary digits only.
#include <bits/stdc++.h>
using namespace std;

// Method return true if N has only 1s and 0s in its
// decimal representation
bool isBinaryNum(int N)
{
    while (N > 0)
    {
        int digit = N % 10;
        if (digit != 0 && digit != 1)
            return false;
        N /= 10;
    }
    return true;
}

// Method return t % N, where t is stored as
// a string
int mod(string t, int N)
{
    int r = 0;
    for (int i = 0; i < t.length(); i++)
    {
        r = r * 10 + (t[i] - '0');
        r %= N;
    }
    return r;
}

// method returns smallest multiple which has
// binary digits
string getMinimumMultipleOfBinaryDigit(int N)
{
    queue<string> q;
    set<int> visit;
```

```
string t = "1";

// In starting push 1 into our queue
q.push(t);

// loop untill queue is not empty
while (!q.empty())
{
    // Take the front number from queue.
    t = q.front();      q.pop();

    // Find remainder of t with respect to N.
    int rem = mod(t, N);

    // if remainder is 0 then we have
    // found our solution
    if (rem == 0)
        return t;

    // If this remainder is not previously seen,
    // then push t0 and t1 in our queue
    else if(visit.find(rem) == visit.end())
    {
        visit.insert(rem);
        q.push(t + "0");
        q.push(t + "1");
    }
}

// Driver code to test above methods
int main()
{
    int N = 12;
    cout << getMinimumMultipleOfBinaryDigit(N);
    return 0;
}
```

Python3

```
# python code to get the
# smallest multiple of N
# with binary digits only
def multiple(A):
    flag = False
    i = 1
    while(flag != True):
```

```
# s = str(A)
# print(s)
# print(A)
result = A * i

# method returns smallest
# multiple which has
# binary digits
allowed_chars = set('01')
validationString = str(result)
if set(validationString).issubset(allowed_chars):
    flag = True;
    break;
else:
    i = i + 1

return validationString

# Driver code
n = 12
print(multiple(n))

# This code is contributed
# by HARDY_123
```

Output:

11100

Improved By : [HARDY_123](#)

Source

<https://www.geeksforgeeks.org/find-the-smallest-binary-digit-multiple-of-given-number/>

Chapter 160

Find whether there is path between two cells in matrix

Find whether there is path between two cells in matrix - GeeksforGeeks

Given N X N matrix filled with 1 , 0 , 2 , 3 . Find whether there is a path possible from source to destination, traversing through blank cells only. You can traverse up, down, right and left.

- A value of cell **1** means Source.
- A value of cell **2** means Destination.
- A value of cell **3** means Blank cell.
- A value of cell **0** means Blank Wall.

Note : there is only single source and single destination(sink).

Examples:

```
Input : M[3] [3] = {{ 0 , 3 , 2 },
                    { 3 , 3 , 0 },
                    { 1 , 3 , 0 }};
```

```
Output : Yes
```

```
Input : M[4] [4] = {{ 0 , 3 , 1 , 0 },
                    { 3 , 0 , 3 , 3 },
                    { 2 , 3 , 0 , 3 },
                    { 0 , 3 , 3 , 3 }};
```

```
Output : Yes
```

Asked in: [Adobe Interview](#)

Simple solution is that find the source index of cell in matrix and then recursively find a path from source index to destination in matrix .
algorithm :

```
Find source index in matrix , let we consider ( i , j )
After that Find path from source(1) to sink(2)
FindPathUtil ( M[] [N] , i , j )

    IF we seen M[i] [j] == 0 (wall) ||
        ((i, j) is out of matrix index)
        return false ;
    IF we seen destination M[i] [j] == 2
        return true ;

    Next move in path by traverse all 4 adjacent
    cell of current cell
    IF (FindPathUtil(M[] [N], i+1, j) ||
        FindPathUtil(M[] [N], i-1, j) ||
        FindPathUtil(M[] [N], i, j+1) ||
        FindPathUtil(M[] [N], i, j-1));
        return true ;

    return false ;
```

Efficient solution (Graph)

The idea is to use Breadth First Search. Consider each cell as a node and each boundary between any two adjacent cells be an edge. so total number of Node is N^2 .

1. Create an empty Graph having N^2 node (Vertex).
2. push all node into graph.
3. notedown source and sink vertex
4. Now Applying BFS to find is there is path between to vertex or not in graph
 - IF Path found return true
 - Else return False

below c++ implementation of above idea

```
// c++ program to find path between two
// cell in matrix
#include<bits/stdc++.h>
using namespace std;
#define N 4
```

```
class Graph
{
    int V ;
    list < int > *adj;
public :
    Graph( int V )
    {
        this->V = V ;
        adj = new list<int>[V];
    }
    void addEdge( int s , int d ) ;
    bool BFS ( int s , int d) ;
};

// add edge to graph
void Graph :: addEdge ( int s , int d )
{
    adj[s].push_back(d);
    adj[d].push_back(s);
}

// BFS function to find path from source to sink
bool Graph :: BFS(int s, int d)
{
    // Base case
    if (s == d)
        return true;

    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and
    // enqueue it
    visited[s] = true;
    queue.push_back(s);

    // it will be used to get all adjacent
    // vertices of a vertex
    list<int>::iterator i;

    while (!queue.empty())
    {
        // Dequeue a vertex from queue
```

```

s = queue.front();
queue.pop_front();

// Get all adjacent vertices of the
// dequeued vertex s. If a adjacent has
// not been visited, then mark it visited
// and enqueue it
for (i = adj[s].begin(); i != adj[s].end(); ++i)
{
    // If this adjacent node is the destination
    // node, then return true
    if (*i == d)
        return true;

    // Else, continue to do BFS
    if (!visited[*i])
    {
        visited[*i] = true;
        queue.push_back(*i);
    }
}

// If BFS is complete without visiting d
return false;
}

bool isSafe(int i, int j, int M[][])
{
    if ((i < 0 || i >= N) ||
        (j < 0 || j >= N ) || M[i][j] == 0)
        return false;
    return true;
}

// Returns true if there is a path from a source (a
// cell with value 1) to a destination (a cell with
// value 2)
bool findPath(int M[][])
{
    int s , d ; // source and destination
    int V = N*N+2;
    Graph g(V);

    // create graph with n*n node
    // each cell consider as node
    int k = 1 ; // Number of current vertex
    for (int i =0 ; i < N ; i++)

```

```
{  
    for (int j = 0 ; j < N; j++)  
    {  
        if (M[i][j] != 0)  
        {  
            // connect all 4 adjacent cell to  
            // current cell  
            if (isSafe ( i , j+1 , M ) )  
                g.addEdge ( k , k+1 );  
            if (isSafe ( i , j-1 , M ) )  
                g.addEdge ( k , k-1 );  
            if (j< N-1 && isSafe ( i+1 , j , M ) )  
                g.addEdge ( k , k+N );  
            if ( i > 0 && isSafe ( i-1 , j , M ) )  
                g.addEdge ( k , k-N );  
        }  
  
        // source index  
        if( M[i][j] == 1 )  
            s = k ;  
  
        // destination index  
        if (M[i][j] == 2)  
            d = k;  
        k++;  
    }  
}  
  
// find path Using BFS  
return g.BFS (s, d) ;  
}  
  
// driver program to check above function  
int main()  
{  
    int M[N][N] = {{ 0 , 3 , 0 , 1 },  
    { 3 , 0 , 3 , 3 },  
    { 2 , 3 , 3 , 3 },  
    { 0 , 3 , 3 , 3 }  
};  
  
(findPath(M) == true) ?  
    cout << "Yes" : cout << "No" << endl ;  
  
return 0;  
}
```

Output:

Yes

Source

<https://www.geeksforgeeks.org/find-whether-path-two-cells-matrix/>

Chapter 161

Finding minimum vertex cover size of a graph using binary search

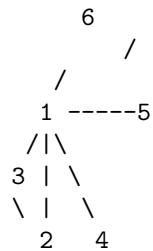
Finding minimum vertex cover size of a graph using binary search - GeeksforGeeks

A [vertex cover](#) of an undirected graph is a subset of its vertices such that for every edge (u, v) of the graph, either ‘ u ’ or ‘ v ’ is in vertex cover. There may be a lot of vertex covers possible for a graph.

Problem Find the size of the minimum size vertex cover, that is, cardinality of a vertex cover with minimum cardinality, for an undirected connected graph with V vertices and m edges.

Examples:

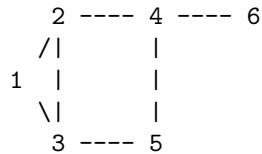
Input: $V = 6, E = 6$



Output: Minimum vertex cover size = 2

Consider subset of vertices {1, 2}, every edge in above graph is either incident on vertex 1 or 2. Hence the minimum vertex cover = {1, 2}, the size of which is 2.

Input: V = 6, E = 7



Output: Minimum vertex cover size = 3

Consider subset of vertices {2, 3, 4}, every edge in above graph is either incident on vertex 2, 3 or 4. Hence the minimum size of a vertex cover can be 3.

Method 1 (Naive)

We can check in $O(E + V)$ time if a given subset of vertices is a vertex cover or not, using the following algorithm.

Generate all $2V$ subsets of vertices in graph and do following for every subset.

1. edges_covered = 0
2. for each vertex in current subset
3. for all edges emerging out of current vertex
4. if the edge is not already marked visited
5. mark the edge visited
6. edges_covered++
7. if edges_covered is equal to total number edges
8. return size of current subset

An upper bound on time complexity of this solution is $O((E + V) * 2^V)$

Method 2 (Binary Search)

If we generate 2^V subsets first by generating $V C_V$ subsets, then $V C_{(V-1)}$ subsets, and so on upto $V C_0$ subsets ($2^V = V C_V + V C_{(V-1)} + \dots + V C_1 + V C_0$). Our objective is now to find the minimum k such that at least one subset of size 'k' amongst $V C_k$ subsets is a vertex cover [We know that if minimum size vertex cover is of size k, then there will exist a vertex cover of all sizes more than k. That is, there will be a vertex cover of size k + 1, k + 2, k + 3, ..., n.]

Now let's imagine a boolean array of size n and call it isCover[]. So if the answer of the question; "Does a vertex cover of size x exist?" is yes, we put a '1' at xth position, otherwise '0'.

The array isCover[] will look like:

1	2	3	.	.	.	k	.	.	.	n
0	0	0	.	.	.	1	.	.	.	1

The array is sorted and hence binary searchable, as no index before **k** will have a ‘1’, and every index after **k**(inclusive) will have a ‘1’, so **k** is the answer.

So we can apply Binary Search to find the minimum size vertex set that covers all edges (this problem is equivalent to finding last 1 in `isCover[]`). Now the problem is how to generate all subsets of a given size. The idea is to use Gosper’s hack.

What is Gosper’s Hack?

Gosper’s hack is a technique to get the next number with same number of bits set. So we set the first x bits from right and generate next number with x bits set until the number is less than 2^V . In this way, we can generate all $V \times C_x$ numbers with x bits set.

```
int set = (1 << k) - 1;
int limit = (1 << V);
while (set < limit)
{
    // Do your stuff with current set
    doStuff(set);

    // Gosper's hack:
    int c = set & -set;
    int r = set + c;
    set = (((r^set) >>> 2) / c) | r;
}
```

Source : [StackExchange](#)

We use gosper’s hack to generate all subsets of size $x(0 < x \leq V)$, that is, to check whether we have a ‘1’ or ‘0’ at any index x in `isCover[]` array.

```
// A C++ program to find size of minimum vertex
// cover using Binary Search
#include<bits/stdc++.h>
#define maxn 25

using namespace std;

// Global array to store the graph
// Note: since the array is global, all the
// elements are 0 initially
bool gr[maxn][maxn];

// Returns true if there is a possible subset
// of size 'k' that can be a vertex cover
bool isCover(int V, int k, int E)
{
    // Set has first 'k' bits high initially
    int set = (1 << k) - 1;
```

```
int limit = (1 << V);

// to mark the edges covered in each subset
// of size 'k'
bool vis[maxn][maxn];

while (set < limit)
{
    // Reset visited array for every subset
    // of vertices
    memset(vis, 0, sizeof vis);

    // set counter for number of edges covered
    // from this subset of vertices to zero
    int cnt = 0;

    // selected vertex cover is the the indices
    // where 'set' has its bit high
    for (int j = 1, v = 1 ; j < limit ; j = j << 1, v++)
    {
        if (set & j)
        {
            // Mark all edges emerging out of this
            // vertex visited
            for (int k = 1 ; k <= V ; k++)
            {
                if (gr[v][k] && !vis[v][k])
                {
                    vis[v][k] = 1;
                    vis[k][v] = 1;
                    cnt++;
                }
            }
        }
    }

    // If the current subset covers all the edges
    if (cnt == E)
        return true;

    // Generate previous combination with k bits high
    // set & -set = (1 << last bit high in set)
    int c = set & -set;
    int r = set + c;
    set = (((r^set) >> 2) / c) | r;
}

return false;
}
```

```
// Returns answer to graph stored in gr[][]  
int findMinCover(int n, int m)  
{  
    // Binary search the answer  
    int left = 1, right = n;  
    while (right > left)  
    {  
        int mid = (left + right) >> 1;  
        if (isCover(n, mid, m) == false)  
            left = mid + 1;  
        else  
            right = mid;  
    }  
  
    // at the end of while loop both left and  
    // right will be equal,/ as when they are  
    // not, the while loop won't exit the minimum  
    // size vertex cover = left = right  
    return left;  
}  
  
// Inserts an edge in the graph  
void insertEdge(int u, int v)  
{  
    gr[u][v] = 1;  
    gr[v][u] = 1; // Undirected graph  
}  
  
// Driver code  
int main()  
{  
    /*  
       6  
      /  
     1 ----- 5   vertex cover = {1, 2}  
    /|\\  
   3 | \\\br/>  \ |  \  
   2   4   */  
    int V = 6, E = 6;  
    insertEdge(1, 2);  
    insertEdge(2, 3);  
    insertEdge(1, 3);  
    insertEdge(1, 4);  
    insertEdge(1, 5);  
    insertEdge(1, 6);  
    cout << "Minimum size of a vertex cover = "
```

```
<< findMinCover(V, E) << endl;

// Let us create another graph
memset(gr, 0, sizeof gr);
/*
    2 ---- 4 ---- 6
    /|        |
1  |        |    vertex cover = {2, 3, 4}
 \ |        |
  3 ---- 5    */

V = 6, E = 7;
insertEdge(1, 2);
insertEdge(1, 3);
insertEdge(2, 3);
insertEdge(2, 4);
insertEdge(3, 5);
insertEdge(4, 5);
insertEdge(4, 6);
cout << "Minimum size of a vertex cover = "
    << findMinCover(V, E) << endl;

return 0;
}
```

Output:

```
Minimum size of a vertex cover = 2
Minimum size of a vertex cover = 3
```

Conclusion:

Time Complexity : $O(E * ({}^V C_{V/2} + {}^V C_{V/4} + {}^V C_{V/8} + \dots \text{upto } {}^V C_k))$

These terms are not more than $\log(V)$ in worst case.

Note: Gosper's hack works for upto $V = 31$ only, if we take 'long long int' instead of 'int' it can work upto $V = 63$.

Source

<https://www.geeksforgeeks.org/finding-minimum-vertex-cover-graph-using-binary-search/>

Chapter 162

Flood fill Algorithm - how to implement fill() in paint?

Flood fill Algorithm - how to implement fill() in paint? - GeeksforGeeks

In MS-Paint, when we take the brush to a pixel and click, the color of the region of that pixel is replaced with a new selected color. Following is the problem statement to do this task.

Given a 2D screen, location of a pixel in the screen and a color, replace color of the given pixel and all adjacent same colored pixels with the given color.

Example:

Input:

```
screen[M][N] = {{1, 1, 1, 1, 1, 1, 1, 1},  
                 {1, 1, 1, 1, 1, 1, 0, 0},  
                 {1, 0, 0, 1, 1, 0, 1, 1},  
                 {1, 2, 2, 2, 2, 0, 1, 0},  
                 {1, 1, 1, 2, 2, 0, 1, 0},  
                 {1, 1, 1, 2, 2, 2, 2, 0},  
                 {1, 1, 1, 1, 1, 2, 1, 1},  
                 {1, 1, 1, 1, 1, 2, 2, 1},  
                 };  
x = 4, y = 4, newColor = 3
```

The values in the given 2D screen indicate colors of the pixels.

x and y are coordinates of the brush, newColor is the color that should replace the previous color on screen[x][y] and all surrounding pixels with same color.

Output:

Screen should be changed to following.
screen[M][N] = {{1, 1, 1, 1, 1, 1, 1, 1},

```
{1, 1, 1, 1, 1, 1, 0, 0},  
{1, 0, 0, 1, 1, 0, 1, 1},  
{1, 3, 3, 3, 3, 0, 1, 0},  
{1, 1, 1, 3, 3, 0, 1, 0},  
{1, 1, 1, 3, 3, 3, 0},  
{1, 1, 1, 1, 1, 3, 1, 1},  
{1, 1, 1, 1, 1, 3, 3, 1},  
};
```

Flood Fill Algorithm:

The idea is simple, we first replace the color of current pixel, then recur for 4 surrounding points. The following is detailed algorithm.

```
// A recursive function to replace previous color 'prevC' at '(x, y)'  
// and all surrounding pixels of (x, y) with new color 'newC' and  
floodFill(screen[M][N], x, y, prevC, newC)  
1) If x or y is outside the screen, then return.  
2) If color of screen[x][y] is not same as prevC, then return  
3) Recur for north, south, east and west.  
    floodFillUtil(screen, x+1, y, prevC, newC);  
    floodFillUtil(screen, x-1, y, prevC, newC);  
    floodFillUtil(screen, x, y+1, prevC, newC);  
    floodFillUtil(screen, x, y-1, prevC, newC);
```

The following is C++ implementation of above algorithm.

```
// A C++ program to implement flood fill algorithm  
#include<iostream>  
using namespace std;  
  
// Dimentions of paint screen  
#define M 8  
#define N 8  
  
// A recursive function to replace previous color 'prevC' at '(x, y)'  
// and all surrounding pixels of (x, y) with new color 'newC' and  
void floodFillUtil(int screen[][][N], int x, int y, int prevC, int newC)  
{  
    // Base cases  
    if (x < 0 || x >= M || y < 0 || y >= N)  
        return;  
    if (screen[x][y] != prevC)  
        return;  
  
    // Replace the color at (x, y)  
    screen[x][y] = newC;
```

```
// Recur for north, east, south and west
floodFillUtil(screen, x+1, y, prevC, newC);
floodFillUtil(screen, x-1, y, prevC, newC);
floodFillUtil(screen, x, y+1, prevC, newC);
floodFillUtil(screen, x, y-1, prevC, newC);
}

// It mainly finds the previous color on (x, y) and
// calls floodFillUtil()
void floodFill(int screen[][] [N], int x, int y, int newC)
{
    int prevC = screen[x][y];
    floodFillUtil(screen, x, y, prevC, newC);
}

// Driver program to test above function
int main()
{
    int screen[M][N] = {{1, 1, 1, 1, 1, 1, 1, 1, 1},
                        {1, 1, 1, 1, 1, 1, 0, 0},
                        {1, 0, 0, 1, 1, 0, 1, 1},
                        {1, 2, 2, 2, 2, 0, 1, 0},
                        {1, 1, 1, 2, 2, 0, 1, 0},
                        {1, 1, 1, 2, 2, 2, 2, 0},
                        {1, 1, 1, 1, 1, 2, 1, 1},
                        {1, 1, 1, 1, 1, 2, 2, 1},
                        };
    int x = 4, y = 4, newC = 3;
    floodFill(screen, x, y, newC);

    cout << "Updated screen after call to floodFill: n";
    for (int i=0; i<M; i++)
    {
        for (int j=0; j<N; j++)
            cout << screen[i][j] << " ";
        cout << endl;
    }
}
```

Output:

```
Updated screen after call to floodFill:
1 1 1 1 1 1 1 1
1 1 1 1 1 1 0 0
1 0 0 1 1 0 1 1
1 3 3 3 3 0 1 0
1 1 1 3 3 0 1 0
```

```
1 1 1 3 3 3 3 0  
1 1 1 1 1 3 1 1  
1 1 1 1 1 3 3 1
```

References:

http://en.wikipedia.org/wiki/Flood_fill

This article is contributed by **Anmol**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

<https://www.geeksforgeeks.org/flood-fill-algorithm-implement-fill-paint/>

Chapter 163

Floyd Warshall Algorithm DP-16

Floyd Warshall Algorithm DP-16 - GeeksforGeeks

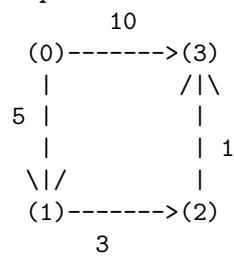
The [Floyd Warshall Algorithm](#) is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

Example:

Input:

```
graph[] [] = { {0, 5, INF, 10},
               {INF, 0, 3, INF},
               {INF, INF, 0, 1},
               {INF, INF, INF, 0} }
```

which represents the following graph



Note that the value of `graph[i][j]` is 0 if i is equal to j
And `graph[i][j]` is INF (infinite) if there is no edge from vertex i to j.

Output:

Shortest distance matrix

0	5	8	9
INF	0	3	4

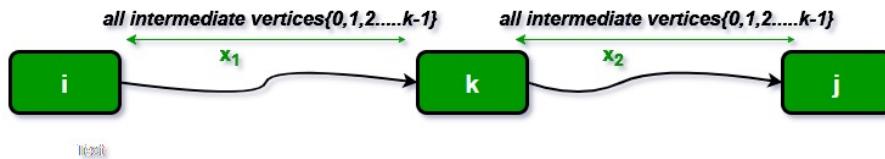
INF	INF	0	1
INF	INF	INF	0

Floyd Warshall Algorithm

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices $\{0, 1, 2, \dots, k-1\}$ as intermediate vertices. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.

- 1) k is not an intermediate vertex in shortest path from i to j. We keep the value of $dist[i][j]$ as it is.
- 2) k is an intermediate vertex in shortest path from i to j. We update the value of $dist[i][j]$ as $dist[i][k] + dist[k][j]$.

The following figure shows the above optimal substructure property in the all-pairs shortest path problem.



Following is implementations of the Floyd Warshall algorithm.
C/C++

```

// C Program for Floyd Warshall Algorithm
#include<stdio.h>

// Number of vertices in the graph
#define V 4

/* Define Infinite as a large enough value. This value will be used
   for vertices not connected to each other */
#define INF 99999

// A function to print the solution matrix
void printSolution(int dist[][]);

// Solves the all-pairs shortest path problem using Floyd Warshall algorithm
void floydWarshall (int graph[][V])
{
    /* dist[][] will be the output matrix that will finally have the shortest
       distances between every pair of vertices */
    int dist[V][V], i, j, k;

    /* Initialize the solution matrix same as input graph matrix. Or
       we can say that initialize all distances as infinite */
    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];
    ...
}

```

```

we can say the initial values of shortest distances are based
on shortest paths considering no intermediate vertex. */
for (i = 0; i < V; i++)
    for (j = 0; j < V; j++)
        dist[i][j] = graph[i][j];

/* Add all vertices one by one to the set of intermediate vertices.
---> Before start of an iteration, we have shortest distances between all
pairs of vertices such that the shortest distances consider only the
vertices in set {0, 1, 2, .. k-1} as intermediate vertices.
----> After the end of an iteration, vertex no. k is added to the set of
intermediate vertices and the set becomes {0, 1, 2, .. k} */
for (k = 0; k < V; k++)
{
    // Pick all vertices as source one by one
    for (i = 0; i < V; i++)
    {
        // Pick all vertices as destination for the
        // above picked source
        for (j = 0; j < V; j++)
        {
            // If vertex k is on the shortest path from
            // i to j, then update the value of dist[i][j]
            if (dist[i][k] + dist[k][j] < dist[i][j])
                dist[i][j] = dist[i][k] + dist[k][j];
        }
    }
}

// Print the shortest distance matrix
printSolution(dist);
}

/* A utility function to print solution */
void printSolution(int dist[][])
{
    printf ("The following matrix shows the shortest distances"
           " between every pair of vertices \n");
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            if (dist[i][j] == INF)
                printf("%7s", "INF");
            else
                printf ("%7d", dist[i][j]);
        }
        printf("\n");
    }
}

```

```
        }
    }

// driver program to test above function
int main()
{
    /* Let us create the following weighted graph
       10
       (0)----->(3)
           |           /|\ \
           5 |           |
           |           | 1
           \|/           |
       (1)----->(2)
           3           */
    int graph[V][V] = { {0, 5, INF, 10},
                        {INF, 0, 3, INF},
                        {INF, INF, 0, 1},
                        {INF, INF, INF, 0}
                    };
}

// Print the solution
floydWarshall(graph);
return 0;
}
```

Java

```
// A Java program for Floyd Warshall All Pairs Shortest
// Path algorithm.
import java.util.*;
import java.lang.*;
import java.io.*;

class AllPairShortestPath
{
    final static int INF = 99999, V = 4;

    void floydWarshall(int graph[][])
    {
        int dist[][] = new int[V][V];
        int i, j, k;

        /* Initialize the solution matrix same as input graph matrix.
           Or we can say the initial values of shortest distances
           are based on shortest paths considering no intermediate
           vertex. */
    }
}
```

```

for (i = 0; i < V; i++)
    for (j = 0; j < V; j++)
        dist[i][j] = graph[i][j];

/* Add all vertices one by one to the set of intermediate
   vertices.
---> Before start of an iteration, we have shortest
       distances between all pairs of vertices such that
       the shortest distances consider only the vertices in
       set {0, 1, 2, .. k-1} as intermediate vertices.
----> After the end of an iteration, vertex no. k is added
       to the set of intermediate vertices and the set
       becomes {0, 1, 2, .. k} */
for (k = 0; k < V; k++)
{
    // Pick all vertices as source one by one
    for (i = 0; i < V; i++)
    {
        // Pick all vertices as destination for the
        // above picked source
        for (j = 0; j < V; j++)
        {
            // If vertex k is on the shortest path from
            // i to j, then update the value of dist[i][j]
            if (dist[i][k] + dist[k][j] < dist[i][j])
                dist[i][j] = dist[i][k] + dist[k][j];
        }
    }
}

// Print the shortest distance matrix
printSolution(dist);
}

void printSolution(int dist[][])
{
    System.out.println("The following matrix shows the shortest "+
                       "distances between every pair of vertices");
    for (int i=0; i<V; ++i)
    {
        for (int j=0; j<V; ++j)
        {
            if (dist[i][j]==INF)
                System.out.print("INF ");
            else
                System.out.print(dist[i][j]+    " ");
        }
        System.out.println();
    }
}

```

```

        }
    }

// Driver program to test above function
public static void main (String[] args)
{
    /* Let us create the following weighted graph
       10
       (0)----->(3)
          |           /|\
          |           |
      5 |           |
          |           | 1
          \|/           |
       (1)----->(2)
          3           */
    int graph[][] = { {0, 5, INF, 10},
                      {INF, 0, 3, INF},
                      {INF, INF, 0, 1},
                      {INF, INF, INF, 0}
                  };
    AllPairShortestPath a = new AllPairShortestPath();

    // Print the solution
    a.floydWarshall(graph);
}
}

// Contributed by Aakash Hasija

```

Python

```

# Python Program for Floyd Warshall Algorithm

# Number of vertices in the graph
V = 4

# Define infinity as the large enough value. This value will be
# used for vertices not connected to each other
INF = 99999

# Solves all pair shortest path via Floyd Warshall Algorithm
def floydWarshall(graph):

    """ dist[][] will be the output matrix that will finally
        have the shortest distances between every pair of vertices """
    """ initializing the solution matrix same as input graph matrix
    OR we can say that the initial values of shortest distances
    are based on shortest paths considering no

```

```

intermediate vertices """
dist = map(lambda i : map(lambda j : j , i) , graph)

""" Add all vertices one by one to the set of intermediate
vertices.
---> Before start of an iteration, we have shortest distances
between all pairs of vertices such that the shortest
distances consider only the vertices in the set
{0, 1, 2, .. k-1} as intermediate vertices.
-----> After the end of a iteration, vertex no. k is
added to the set of intermediate vertices and the
set becomes {0, 1, 2, .. k}
"""
for k in range(V):

    # pick all vertices as source one by one
    for i in range(V):

        # Pick all vertices as destination for the
        # above picked source
        for j in range(V):

            # If vertex k is on the shortest path from
            # i to j, then update the value of dist[i][j]
            dist[i][j] = min(dist[i][j] ,
                               dist[i][k]+ dist[k][j]
                               )
printSolution(dist)

# A utility function to print the solution
def printSolution(dist):
    print "Following matrix shows the shortest distances\
between every pair of vertices"
    for i in range(V):
        for j in range(V):
            if(dist[i][j] == INF):
                print "%7s" %(("INF"),
            else:
                print "%7d\t" %(dist[i][j]),
            if j == V-1:
                print ""
"""

# Driver program to test the above program
# Let us create the following weighted graph
"""

```

```

          10
      (0)----->(3)
      |           /|\
      |           |
      |           | 1
      \|/           |
      (1)----->(2)
      3           """
graph = [[0,5,INF,10],
         [INF,0,3,INF],
         [INF, INF, 0, 1],
         [INF, INF, INF, 0]
        ]
# Print the solution
floydWarshall(graph);
# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

C#

```

// A C# program for Floyd Warshall All
// Pairs Shortest Path algorithm.

using System;

public class AllPairShortestPath
{
    readonly static int INF = 99999, V = 4;

    void floydWarshall(int[,] graph)
    {
        int[,] dist = new int[V, V];
        int i, j, k;

        // Initialize the solution matrix
        // same as input graph matrix
        // Or we can say the initial
        // values of shortest distances
        // are based on shortest paths
        // considering no intermediate
        // vertex
        for (i = 0; i < V; i++) {
            for (j = 0; j < V; j++) {
                dist[i, j] = graph[i, j];
            }
        }

        /* Add all vertices one by one to
        the set of intermediate vertices.
```

```

----> Before start of a iteration,
      we have shortest distances
      between all pairs of vertices
      such that the shortest distances
      consider only the vertices in
      set {0, 1, 2, .. k-1} as
      intermediate vertices.
----> After the end of a iteration,
      vertex no. k is added
      to the set of intermediate
      vertices and the set
      becomes {0, 1, 2, .. k} */
for (k = 0; k < V; k++)
{
    // Pick all vertices as source
    // one by one
    for (i = 0; i < V; i++)
    {
        // Pick all vertices as destination
        // for the above picked source
        for (j = 0; j < V; j++)
        {
            // If vertex k is on the shortest
            // path from i to j, then update
            // the value of dist[i][j]
            if (dist[i, k] + dist[k, j] < dist[i, j])
            {
                dist[i, j] = dist[i, k] + dist[k, j];
            }
        }
    }
}

// Print the shortest distance matrix
printSolution(dist);
}

void printSolution(int[,] dist)
{
    Console.WriteLine("Following matrix shows the shortest "+
                      "distances between every pair of vertices");
    for (int i = 0; i < V; ++i)
    {
        for (int j = 0; j < V; ++j)
        {
            if (dist[i, j] == INF) {
                Console.Write("INF ");
            } else {

```

```

        Console.WriteLine();
    }
}

Console.WriteLine();
}

// Driver Code
public static void Main(string[] args)
{
    /* Let us create the following
       weighted graph
          10
    (0)----->(3)
    |           /|\ \
    5 |           |
    |           | 1
    \|/           |
    (1)----->(2)
            3           */
    int[,] graph = { {0, 5, INF, 10},
                    {INF, 0, 3, INF},
                    {INF, INF, 0, 1},
                    {INF, INF, INF, 0}
                };
}

AllPairShortestPath a = new AllPairShortestPath();

// Print the solution
a.floydWarshall(graph);
}

// This article is contributed by
// Abdul Mateen Mohammed

```

Output:

```

Following matrix shows the shortest distances between every pair of vertices
      0      5      8      9
INF      0      3      4
INF      INF     0      1
INF      INF     INF     0

```

Time Complexity: $O(V^3)$

The above program only prints the shortest distances. We can modify the solution to print the shortest paths also by storing the predecessor information in a separate 2D matrix. Also, the value of INF can be taken as INT_MAX from limits.h to make sure that we handle maximum possible value. When we take INF as INT_MAX, we need to change the if condition in the above program to avoid arithmetic overflow.

```
#include  
  
#define INF INT_MAX  
.....  
if ( dist[i][k] != INF &&  
    dist[k][j] != INF &&  
    dist[i][k] + dist[k][j] < dist[i][j]  
)  
    dist[i][j] = dist[i][k] + dist[k][j];  
.....
```

Improved By : [Abdul Mateen Mohammed](#)

Source

<https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/>

Chapter 164

Generate a graph using Dictionary in Python

Generate a graph using Dictionary in Python - GeeksforGeeks

Prerequisite – [Graphs](#)

To draw graph using in built libraries – [Graph plotting in Python](#)

In this article, we will see how to implement graph in python using [dictionary](#) data structure in python.

The keys of the dictionary used are the nodes of our graph and the corresponding values are lists with each nodes, which are connecting by an edge.

This simple graph has six nodes (a-f) and five arcs:

```
a -> c  
b -> c  
b -> e  
c -> a  
c -> b  
c -> d  
c -> e  
d -> c  
e -> c  
e -> b
```

It can be represented by the following Python data structure. This is a dictionary whose keys are the nodes of the graph. For each key, the corresponding value is a list containing the nodes that are connected by a direct arc from this node.

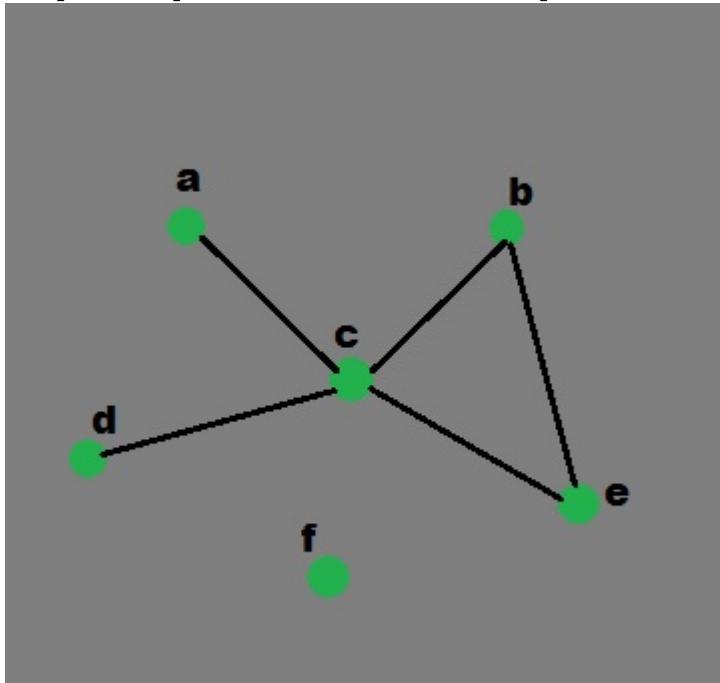
```
graph = { "a" : ["c"],
```

```

    "b" : ["c", "e"],
    "c" : ["a", "b", "d", "e"],
    "d" : ["c"],
    "e" : ["c", "b"],
    "f" : []
}

```

Graphical representation of above example:



defaultdict: Usually, a Python dictionary throws a KeyError if you try to get an item with a key that is not currently in the dictionary. defaultdict allows that if a key is not found in the dictionary, then instead of a KeyError being thrown, a new entry is created. The type of this new entry is given by the argument of defaultdict.

Python Function to generate graph:

```

# definition of function
def generate_edges(graph):
    edges = []

    # for each node in graph
    for node in graph:

        # for each neighbour node of a single node
        for neighbour in graph[node]:
            # if edge exists then append
            edges.append((node, neighbour))

```

```
return edges

# Python program for
# validation of a graph

# import dictionary for graph
from collections import defaultdict

# function for adding edge to graph
graph = defaultdict(list)
def addEdge(graph,u,v):
    graph[u].append(v)

# definition of function
def generate_edges(graph):
    edges = []

    # for each node in graph
    for node in graph:

        # for each neighbour node of a single node
        for neighbour in graph[node]:

            # if edge exists then append
            edges.append((node, neighbour))
    return edges

# declaration of graph as dictionary
addEdge(graph,'a','c')
addEdge(graph,'b','c')
addEdge(graph,'b','e')
addEdge(graph,'c','d')
addEdge(graph,'c','e')
addEdge(graph,'c','a')
addEdge(graph,'c','b')
addEdge(graph,'e','b')
addEdge(graph,'d','c')
addEdge(graph,'e','c')

# Driver Function call
# to print generated graph
print(generate_edges(graph))
```

Output:

```
[('a', 'c'), ('c', 'd'), ('c', 'e'), ('c', 'a'), ('c', 'b'),
 ('b', 'c'), ('b', 'e'), ('e', 'b'), ('e', 'c'), ('d', 'c')]
```

As we have taken example of undirected graph, so we have print same edge twice say as ('a','c') and ('c','a'). We can overcome this with use of directed graph.

Below are some more programs on graphs in python:

1. To generate the path from one node to the other node:

Using Python dictionary, we can find the path from one node to the other in a Graph. The idea is similar to [DFS](#) in graphs.

In the function, initially, the path is an empty list. In the starting, if the start node matches with the end node, the function will return the path. Otherwise the code goes forward and hits all the values of the starting node and searches for the path using recursion. If there is no such path, it returns None.

```
# Python program to generate the first
# path of the graph from the nodes provided

graph ={
    'a':['c'],
    'b':['d'],
    'c':['e'],
    'd':['a', 'd'],
    'e':['b', 'c']
}

# function to find path
def find_path(graph, start, end, path =[]):
    path = path + [start]
    if start == end:
        return path
    for node in graph[start]:
        if node not in path:
            newpath = find_path(graph, node, end, path)
            if newpath:
                return newpath
    return None

# Driver function call to print the path
print(find_path(graph, 'd', 'c'))
```

Output:

```
['d', 'a', 'c']
```

2. Program to generate all the possible paths from one node to the other.:

In the above discussed program, we generated the first possible path. Now, let us generate all the possible paths from the start node to the end node. The basic functioning works same as the functioning of the above code. The place where the difference comes

is instead of instantly returning the first path, it saves that path in a list named as ‘paths’ in the example given below. Finally, after iterating over all the possible ways, it returns the list of paths. If there is no path from the starting node to the ending node, it returns None.

```
# Python program to generate the all possible
# path of the graph from the nodes provided
graph ={
    'a':['c'],
    'b':['d'],
    'c':['e'],
    'd':['a', 'd'],
    'e':['b', 'c']
}

# function to generate all possible paths
def find_all_paths(graph, start, end, path =[]):
    path = path + [start]
    if start == end:
        return [path]
    paths = []
    for node in graph[start]:
        if node not in path:
            newpaths = find_all_paths(graph, node, end, path)
            for newpath in newpaths:
                paths.append(newpath)
    return paths

# Driver function call to print all
# generated paths
print(find_all_paths(graph, 'd', 'c'))
```

Output:

```
[[['d', 'a', 'c'], ['d', 'a', 'c']]]
```

3. Program to generate the shortest path.:

To get to the shortest from all the paths, we use a little different approach as shown below. In this, as we get the path from the start node to the end node, we compare the length of the path with a variable named as shortest which is initialized with the None value. If the length of generated path is less than the length of shortest, if shortest is not None, the newly generated path is set as the value of shortest. Again, if there is no path, it returns None

```
# Python program to generate shortest path

graph ={
```

```
'a':['c'],
'b':['d'],
'c':['e'],
'd':['a', 'd'],
'e':['b', 'c']
}

# function to find the shortest path
def find_shortest_path(graph, start, end, path =[]):
    path = path + [start]
    if start == end:
        return path
    shortest = None
    for node in graph[start]:
        if node not in path:
            newpath = find_shortest_path(graph, node, end, path)
            if newpath:
                if not shortest or len(newpath) < len(shortest):
                    shortest = newpath
    return shortest

# Driver function call to print
# the shortest path
print(find_shortest_path(graph, 'd', 'c'))
```

Output:

```
['d', 'a', 'c']
```

Source

<https://www.geeksforgeeks.org/generate-graph-using-dictionary-python/>

Chapter 165

**Given a matrix of 'O' and 'X',
replace 'O' with 'X' if
surrounded by 'X'**

Given a matrix of 'O' and 'X', replace 'O' with 'X' if surrounded by 'X' - GeeksforGeeks

Given a matrix where every element is either 'O' or 'X', replace 'O' with 'X' if surrounded by 'X'. A 'O' (or a set of 'O') is considered to be surrounded by 'X' if there are 'X' at locations just below, just above, just left and just right of it.

Examples:

```
Input: mat[M][N] = {{'X', 'O', 'X', 'X', 'X', 'X'},  
                     {'X', 'O', 'X', 'X', 'O', 'X'},  
                     {'X', 'X', 'X', 'O', 'O', 'X'},  
                     {'O', 'X', 'X', 'X', 'X', 'X'},  
                     {'X', 'X', 'X', 'O', 'X', 'O'},  
                     {'O', 'O', 'X', 'O', 'O', 'O'},  
                     };
```

```
Output: mat[M][N] = {{'X', 'O', 'X', 'X', 'X', 'X'},  
                      {'X', 'O', 'X', 'X', 'X', 'X'},  
                      {'X', 'X', 'X', 'X', 'X', 'X'},  
                      {'O', 'X', 'X', 'X', 'X', 'X'},  
                      {'X', 'X', 'X', 'O', 'X', 'O'},  
                      {'O', 'O', 'X', 'O', 'O', 'O'},  
                      };
```

```
Input: mat[M][N] = {{'X', 'X', 'X', 'X'}  
                     {'X', 'O', 'X', 'X'}  
                     {'X', 'O', 'O', 'X'}  
                     {'X', 'O', 'X', 'X'}}
```

```
{'X', 'X', 'O', 'O'}
};
```

```
Input: mat[M][N] = {{'X', 'X', 'X', 'X'},
{'X', 'X', 'X', 'X'},
{'X', 'X', 'X', 'X'},
{'X', 'X', 'X', 'X'},
{'X', 'X', 'O', 'O'}
};
```

This is mainly an application of [Flood-Fill algorithm](#). The main difference here is that a 'O' is not replaced by 'X' if it lies in region that ends on a boundary. Following are simple steps to do this special flood fill.

- 1) Traverse the given matrix and replace all 'O' with a special character '-'.
- 2) Traverse four edges of given matrix and call [floodFill\('-', 'O'\)](#) for every '-' on edges. The remaining '-' are the characters that indicate 'O's (in the original matrix) to be replaced by 'X'.
- 3) Traverse the matrix and replace all '-'s with 'X's.

Let us see steps of above algorithm with an example. Let following be the input matrix.

```
mat[M][N] = {{'X', 'O', 'X', 'X', 'X', 'X'},
{'X', 'O', 'X', 'X', 'O', 'X'},
{'X', 'X', 'X', 'O', 'O', 'X'},
{'O', 'X', 'X', 'X', 'X', 'X'},
{'X', 'X', 'X', 'O', 'X', 'O'},
{'O', 'O', 'X', 'O', 'O', 'O'},
};
```

Step 1: Replace all 'O' with '-'.

```
mat[M][N] = {{'X', 'O', 'X', 'X', 'X', 'X'},
{'X', 'O', 'X', 'X', 'O', 'X'},
{'X', 'X', 'X', 'O', 'O', 'X'},
{'O', 'X', 'X', 'X', 'X', 'X'},
{'X', 'X', 'X', 'O', 'X', 'O'},
{'O', 'O', 'X', 'O', 'O', 'O'},
};
```

Step 2: Call [floodFill\(' ', 'O'\)](#) for all edge elements with value equals to '-'

```
mat[M][N] = {{'X', 'O', 'X', 'X', 'X', 'X'},  
              {'X', 'O', 'X', 'X', '—', 'X'},  
              {'X', 'X', 'X', '—', '—', 'X'},  
              {'O', 'X', 'X', 'X', 'X', 'X'},  
              {'X', 'X', 'X', 'O', 'X', 'O'},  
              {'O', 'O', 'X', 'O', 'O', 'O'},  
              };
```

Step 3: Replace all ‘—’ with ‘X’.

```
mat[M][N] = {{'X', 'O', 'X', 'X', 'X', 'X'},  
              {'X', 'O', 'X', 'X', 'X', 'X'},  
              {'X', 'X', 'X', 'X', 'X', 'X'},  
              {'O', 'X', 'X', 'X', 'X', 'X'},  
              {'X', 'X', 'X', 'O', 'X', 'O'},  
              {'O', 'O', 'X', 'O', 'O', 'O'},  
              };
```

The following is implementation of above algorithm.

C++

```
// A C++ program to replace all 'O's with 'X''s if surrounded by 'X'  
#include<iostream>  
using namespace std;  
  
// Size of given matrix is M X N  
#define M 6  
#define N 6  
  
// A recursive function to replace previous value 'prevV' at '(x, y)'  
// and all surrounding values of (x, y) with new value 'newV'.  
void floodFillUtil(char mat[][][N], int x, int y, char prevV, char newV)  
{  
    // Base cases  
    if (x < 0 || x >= M || y < 0 || y >= N)  
        return;  
    if (mat[x][y] != prevV)  
        return;  
  
    // Replace the color at (x, y)  
    mat[x][y] = newV;  
  
    // Recur for north, east, south and west  
    floodFillUtil(mat, x+1, y, prevV, newV);
```

```
floodFillUtil(mat, x-1, y, prevV, newV);
floodFillUtil(mat, x, y+1, prevV, newV);
floodFillUtil(mat, x, y-1, prevV, newV);
}

// Returns size of maximum size subsquare matrix
// surrounded by 'X'
int replaceSurrounded(char mat[][] [N])
{
    // Step 1: Replace all 'O' with '-'
    for (int i=0; i<M; i++)
        for (int j=0; j<N; j++)
            if (mat[i][j] == 'O')
                mat[i][j] = '-';

    // Call floodFill for all '-' lying on edges
    for (int i=0; i<M; i++)    // Left side
        if (mat[i][0] == '-')
            floodFillUtil(mat, i, 0, '-', 'O');
    for (int i=0; i<M; i++)    // Right side
        if (mat[i][N-1] == '-')
            floodFillUtil(mat, i, N-1, '-', 'O');
    for (int i=0; i<N; i++)    // Top side
        if (mat[0][i] == '-')
            floodFillUtil(mat, 0, i, '-', 'O');
    for (int i=0; i<N; i++)    // Bottom side
        if (mat[M-1][i] == '-')
            floodFillUtil(mat, M-1, i, '-', 'O');

    // Step 3: Replace all '-' with 'X'
    for (int i=0; i<M; i++)
        for (int j=0; j<N; j++)
            if (mat[i][j] == '-')
                mat[i][j] = 'X';
}

// Driver program to test above function
int main()
{
    char mat[][] [N] = {{'X', 'O', 'X', 'O', 'X', 'X'}, 
                        {'X', 'O', 'X', 'X', 'O', 'X'}, 
                        {'X', 'X', 'X', 'O', 'X', 'X'}, 
                        {'O', 'X', 'X', 'X', 'X', 'X'}, 
                        {'X', 'X', 'X', 'O', 'X', 'O'}, 
                        {'O', 'O', 'X', 'O', 'O', 'O'}, 
                        {''};
    replaceSurrounded(mat);
```

```
for (int i=0; i<M; i++)
{
    for (int j=0; j<N; j++)
        cout << mat[i][j] << " ";
    cout << endl;
}
return 0;
}
```

Java

```
// A Java program to replace
// all 'O's with 'X''s if
// surrounded by 'X'
import java.io.*;

class GFG
{
    static int M = 6;
    static int N = 6;
    static void floodFillUtil(char mat[][] , int x,
                                int y, char prevV,
                                char newV)
    {
        // Base cases
        if (x < 0 || x >= M ||
            y < 0 || y >= N)
            return;

        if (mat[x][y] != prevV)
            return;

        // Replace the color at (x, y)
        mat[x][y] = newV;

        // Recur for north,
        // east, south and west
        floodFillUtil(mat, x + 1, y,
                      prevV, newV);
        floodFillUtil(mat, x - 1, y,
                      prevV, newV);
        floodFillUtil(mat, x, y + 1,
                      prevV, newV);
        floodFillUtil(mat, x, y - 1,
                      prevV, newV);
    }
}
```

```
// Returns size of maximum
// size subsquare matrix
// surrounded by 'X'
static void replaceSurrounded(char mat[][])
{
    // Step 1: Replace
    // all 'O' with '-'
    for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++)
            if (mat[i][j] == 'O')
                mat[i][j] = '-';

    // Call floodFill for
    // all '-' lying on edges
    for (int i = 0; i < M; i++) // Left side
        if (mat[i][0] == '-')
            floodFillUtil(mat, i, 0,
                           '-', 'O');

    for (int i = 0; i < M; i++) // Right side
        if (mat[i][N - 1] == '-')
            floodFillUtil(mat, i, N - 1,
                           '-', 'O');

    for (int i = 0; i < N; i++) // Top side
        if (mat[0][i] == '-')
            floodFillUtil(mat, 0, i,
                           '-', 'O');

    for (int i = 0; i < N; i++) // Bottom side
        if (mat[M - 1][i] == '-')
            floodFillUtil(mat, M - 1,
                           i, '-', 'O');

    // Step 3: Replace
    // all '-' with 'X'
    for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++)
            if (mat[i][j] == '-')
                mat[i][j] = 'X';
}

// Driver Code
public static void main (String[] args)
{
    char[][] mat = {{'X', 'O', 'X',
                     'O', 'X', 'X'},
                    {'X', 'O', 'X',
                     'X', 'O', 'X'},
                    {'X', 'O', 'X',
                     'X', 'O', 'X'}},

```

Chapter 165. Given a matrix of 'O' and 'X', replace 'O' with 'X' if surrounded by 'X'

```
{'X', 'X', 'X',
 'O', 'X', 'X'},
 {'O', 'X', 'X',
  'X', 'X', 'X'},
 {'X', 'X', 'X',
  'O', 'X', 'O'},
 {'O', 'O', 'X',
  'O', 'O'}};

replaceSurrounded(mat);

for (int i = 0; i < M; i++)
{
    for (int j = 0; j < N; j++)
        System.out.print(mat[i][j] + " ");

    System.out.println("");
}
}

// This code is contributed
// by shiv_bhakt
```

C#

```
// A C# program to replace
// all 'O's with 'X's if
// surrounded by 'X'
using System;

class GFG
{
    static int M = 6;
    static int N = 6;
    static void floodFillUtil(char [,]mat, int x,
                                int y, char prevV,
                                char newV)
    {
        // Base cases
        if (x < 0 || x >= M ||
            y < 0 || y >= N)
            return;

        if (mat[x, y] != prevV)
            return;

        // Replace the color at (x, y)
```

```
mat[x, y] = newV;

// Recur for north,
// east, south and west
floodFillUtil(mat, x + 1, y,
              prevV, newV);
floodFillUtil(mat, x - 1, y,
              prevV, newV);
floodFillUtil(mat, x, y + 1,
              prevV, newV);
floodFillUtil(mat, x, y - 1,
              prevV, newV);
}

// Returns size of maximum
// size subsquare matrix
// surrounded by 'X'
static void replaceSurrounded(char [,]mat)
{

    // Step 1: Replace
    // all 'O' with '-'
    for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++)
            if (mat[i, j] == 'O')
                mat[i, j] = '-';

    // Call floodFill for
    // all '-' lying on edges
    for (int i = 0; i < M; i++) // Left side
        if (mat[i, 0] == '-')
            floodFillUtil(mat, i, 0,
                           '-', 'O');
    for (int i = 0; i < M; i++) // Right side
        if (mat[i, N - 1] == '-')
            floodFillUtil(mat, i, N - 1,
                           '-', 'O');
    for (int i = 0; i < N; i++) // Top side
        if (mat[0, i] == '-')
            floodFillUtil(mat, 0, i,
                           '-', 'O');
    for (int i = 0; i < N; i++) // Bottom side
        if (mat[M - 1, i] == '-')
            floodFillUtil(mat, M - 1,
                           i, '-', 'O');

    // Step 3: Replace
    // all '-' with 'X'
```

Chapter 165. Given a matrix of 'O' and 'X', replace 'O' with 'X' if surrounded by 'X'

```
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
        if (mat[i, j] == '-')
            mat[i, j] = 'X';
}

// Driver Code
public static void Main ()
{
    char [,]mat = new char[,]
        {{'X', 'O', 'X',
          'O', 'X', 'X'},
         {'X', 'O', 'X',
          'X', 'O', 'X'},
         {'X', 'X', 'X',
          'O', 'X', 'X'},
         {'O', 'X', 'X',
          'X', 'X', 'O'},
         {'O', 'O', 'X',
          'O', 'X', 'O'},
         {'O', 'O', 'O'}};

    replaceSurrounded(mat);

    for (int i = 0; i < M; i++)
    {
        for (int j = 0; j < N; j++)
            Console.Write(mat[i, j] + " ");

        Console.WriteLine("");
    }
}

// This code is contributed
// by shiv_bhakt
```

PHP

```
<?php
// A PHP program to replace all
// 'O's with 'X''s if surrounded by 'X'

// Size of given
// matrix is M X N
$M = 6;
$N = 6;
```

```
// A recursive function to replace
// previous value 'prevV' at '(x, y)'
// and all surrounding values of
// (x, y) with new value 'newV'.
function floodFillUtil(&$mat, $x, $y,
                      $prevV, $newV)
{
    // Base cases
    if ($x < 0 || $x >= $GLOBALS['M'] ||
        $y < 0 || $y >= $GLOBALS['N'])
        return;
    if ($mat[$x][$y] != $prevV)
        return;

    // Replace the color at (x, y)
    $mat[$x][$y] = $newV;

    // Recur for north,
    // east, south and west
    floodFillUtil($mat, $x + 1, $y, $prevV, $newV);
    floodFillUtil($mat, $x - 1, $y, $prevV, $newV);
    floodFillUtil($mat, $x, $y + 1, $prevV, $newV);
    floodFillUtil($mat, $x, $y - 1, $prevV, $newV);
}

// Returns size of maximum
// size subsquare matrix
// surrounded by 'X'
function replaceSurrounded(&$mat)
{

    // Step 1: Replace all 'O' with '-'
    for ($i = 0; $i < $GLOBALS['M']; $i++)
        for ($j = 0; $j < $GLOBALS['N']; $j++)
            if ($mat[$i][$j] == 'O')
                $mat[$i][$j] = '-';

    // Call floodFill for all
    // '-' lying on edges
    for ($i = 0;
         $i < $GLOBALS['M']; $i++) // Left side
        if ($mat[$i][0] == '-')
            floodFillUtil($mat, $i, 0, '-', 'O');

    for ($i = 0; $i < $GLOBALS['M']; $i++) // Right side
        if ($mat[$i][$GLOBALS['N'] - 1] == '-')
            floodFillUtil($mat, $i, $GLOBALS['N'] - 1, '-', 'O');
}
```

```
floodFillUtil($mat, $i,
              $GLOBALS['N'] - 1, '-', '0');

for ($i = 0; $i < $GLOBALS['N']; $i++) // Top side
    if ($mat[0][$i] == '-')
        floodFillUtil($mat, 0, $i, '-', '0');

for ($i = 0; $i < $GLOBALS['N']; $i++) // Bottom side
    if ($mat[$GLOBALS['M'] - 1][$i] == '-')
        floodFillUtil($mat, $GLOBALS['M'] - 1,
                      $i, '-', '0');

// Step 3: Replace all '-' with 'X'
for ($i = 0; $i < $GLOBALS['M']; $i++)
    for ($j = 0; $j < $GLOBALS['N']; $j++)
        if ($mat[$i][$j] == '-')
            $mat[$i][$j] = 'X';

}

// Driver Code
$mat = array(array('X', 'O', 'X', 'O', 'X', 'X'),
             array('X', 'O', 'X', 'X', 'O', 'X'),
             array('X', 'X', 'X', 'O', 'X', 'X'),
             array('O', 'X', 'X', 'X', 'X', 'X'),
             array('X', 'X', 'X', 'O', 'X', 'O'),
             array('O', 'O', 'X', 'O', 'O', 'O'));
replaceSurrounded($mat);

for ($i = 0; $i < $GLOBALS['M']; $i++)
{
    for ($j = 0; $j < $GLOBALS['N']; $j++)
        echo $mat[$i][$j]." ";
    echo "\n";
}

// This code is contributed by ChitraNayal
?>
```

Output:

```
X O X O X X
X O X X X X
X X X X X X
O X X X X X
X X X O X O
O O X O O O
```

Chapter 165. Given a matrix of 'O' and 'X', replace 'O' with 'X' if surrounded by 'X'

Time Complexity of the above solution is $O(MN)$. Note that every element of matrix is processed at most three times.

This article is contributed by **Anmol**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Improved By : [shiv_bhakt](#), [ChitraNayal](#)

Source

<https://www.geeksforgeeks.org/given-matrix-o-x-replace-o-x-surrounded-x/>

Chapter 166

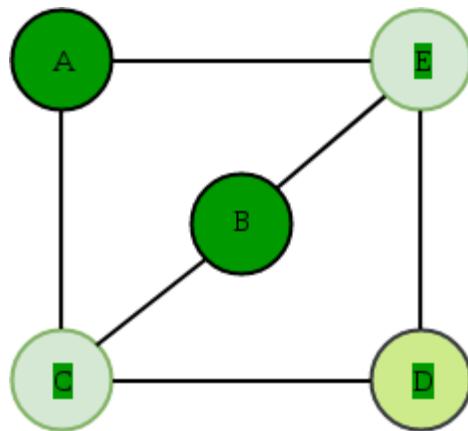
Graph Coloring Set 1 (Introduction and Applications)

Graph Coloring Set 1 (Introduction and Applications) - GeeksforGeeks

[Graph coloring](#) problem is to assign colors to certain elements of a graph subject to certain constraints.

Vertex coloring is the most common graph coloring problem. The problem is, given m colors, find a way of coloring the vertices of a graph such that no two adjacent vertices are colored using same color. The other graph coloring problems like **Edge Coloring** (No vertex is incident to two edges of same color) and **Face Coloring** (Geographical Map Coloring) can be transformed into vertex coloring.

Chromatic Number: The smallest number of colors needed to color a graph G is called its chromatic number. For example, the following can be colored minimum 3 colors.



The problem to find chromatic number of a given graph is [NP Complete](#).

Applications of Graph Coloring:

The graph coloring problem has huge number of applications.

1) **Making Schedule or Time Table:** Suppose we want to make an exam schedule for a university. We have listed different subjects and students enrolled in every subject. Many subjects would have common students (of same batch, some backlog students, etc). *How do we schedule the exam so that no two exams with a common student are scheduled at same time? How many minimum time slots are needed to schedule all exams?* This problem can be represented as a graph where every vertex is a subject and an edge between two vertices means there is a common student. So this is a graph coloring problem where minimum number of time slots is equal to the chromatic number of the graph.

2) **Mobile Radio Frequency Assignment:** When frequencies are assigned to towers, frequencies assigned to all towers at the same location must be different. How to assign frequencies with this constraint? What is the minimum number of frequencies needed? This problem is also an instance of graph coloring problem where every tower represents a vertex and an edge between two towers represents that they are in range of each other.

3) **Sudoku:** Sudoku is also a variation of Graph coloring problem where every cell represents a vertex. There is an edge between two vertices if they are in same row or same column or same block.

4) **Register Allocation:** In compiler optimization, register allocation is the process of assigning a large number of target program variables onto a small number of CPU registers. This problem is also a graph coloring problem.

5) **Bipartite Graphs:** We can check if a graph is Bipartite or not by coloring the graph using two colors. If a given graph is 2-colorable, then it is Bipartite, otherwise not. See [this](#) for more details.

6) **Map Coloring:** Geographical maps of countries or states where no two adjacent cities cannot be assigned same color. Four colors are sufficient to color any map (See [Four Color Theorem](#))

There can be many more applications: For example the below reference video lecture has a case study at 1:18.

[Akamai](#) runs a network of thousands of servers and the servers are used to distribute content on Internet. They install a new software or update existing softwares pretty much every week. The update cannot be deployed on every server at the same time, because the server may have to be taken down for the install. Also, the update should not be done one at a time, because it will take a lot of time. There are sets of servers that cannot be taken down together, because they have certain critical functions. This is a typical scheduling application of graph coloring problem. It turned out that 8 colors were good enough to color the graph of 75000 nodes. So they could install updates in 8 passes.

We will soon be discussing different ways to solve the graph coloring problem.

References:

[Lec 6 MIT 6.042J Mathematics for Computer Science, Fall 2010 Video Lecture](#)

Source

<https://www.geeksforgeeks.org/graph-coloring-applications/>

Chapter 167

Graph Coloring Set 2 (Greedy Algorithm)

Graph Coloring Set 2 (Greedy Algorithm) - GeeksforGeeks

We introduced [graph coloring and applications](#) in previous post. As discussed in the previous post, graph coloring is widely used. Unfortunately, there is no efficient algorithm available for coloring a graph with minimum number of colors as the problem is a known [NP Complete problem](#). There are approximate algorithms to solve the problem though. Following is the basic Greedy Algorithm to assign colors. It doesn't guarantee to use minimum colors, but it guarantees an upper bound on the number of colors. The basic algorithm never uses more than $d+1$ colors where d is the maximum degree of a vertex in the given graph.

Basic Greedy Coloring Algorithm:

1. Color first vertex with first color.
2. Do following for remaining $V-1$ vertices.
..... a) Consider the currently picked vertex and color it with the lowest numbered color that has not been used on any previously colored vertices adjacent to it. If all previously used colors appear on vertices adjacent to v , assign a new color to it.

Following are C++ and Java implementations of the above Greedy Algorithm.

C++

```
// A C++ program to implement greedy algorithm for graph coloring
#include <iostream>
#include <list>
using namespace std;

// A class that represents an undirected graph
class Graph
```

```
{
    int V;      // No. of vertices
    list<int> *adj;     // A dynamic array of adjacency lists
public:
    // Constructor and destructor
    Graph(int V) { this->V = V; adj = new list<int>[V]; }
    ~Graph() { delete [] adj; }

    // function to add an edge to graph
    void addEdge(int v, int w);

    // Prints greedy coloring of the vertices
    void greedyColoring();
};

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
    adj[w].push_back(v); // Note: the graph is undirected
}

// Assigns colors (starting from 0) to all vertices and prints
// the assignment of colors
void Graph::greedyColoring()
{
    int result[V];

    // Assign the first color to first vertex
    result[0] = 0;

    // Initialize remaining V-1 vertices as unassigned
    for (int u = 1; u < V; u++)
        result[u] = -1; // no color is assigned to u

    // A temporary array to store the available colors. True
    // value of available[cr] would mean that the color cr is
    // assigned to one of its adjacent vertices
    bool available[V];
    for (int cr = 0; cr < V; cr++)
        available[cr] = false;

    // Assign colors to remaining V-1 vertices
    for (int u = 1; u < V; u++)
    {
        // Process all adjacent vertices and flag their colors
        // as unavailable
        list<int>::iterator i;
        for (i = adj[u].begin(); i != adj[u].end(); ++i)
            available[*i] = true;
    }

    // Print the result
    cout << "Greedy Coloring Result: ";
    for (int u = 0; u < V; u++)
        cout << result[u] << " ";
}
```

```

        if (result[*i] != -1)
            available[result[*i]] = true;

        // Find the first available color
        int cr;
        for (cr = 0; cr < V; cr++)
            if (available[cr] == false)
                break;

        result[u] = cr; // Assign the found color

        // Reset the values back to false for the next iteration
        for (i = adj[u].begin(); i != adj[u].end(); ++i)
            if (result[*i] != -1)
                available[result[*i]] = false;
    }

    // print the result
    for (int u = 0; u < V; u++)
        cout << "Vertex " << u << " ---> Color "
            << result[u] << endl;
}

// Driver program to test above function
int main()
{
    Graph g1(5);
    g1.addEdge(0, 1);
    g1.addEdge(0, 2);
    g1.addEdge(1, 2);
    g1.addEdge(1, 3);
    g1.addEdge(2, 3);
    g1.addEdge(3, 4);
    cout << "Coloring of graph 1 \n";
    g1.greedyColoring();

    Graph g2(5);
    g2.addEdge(0, 1);
    g2.addEdge(0, 2);
    g2.addEdge(1, 2);
    g2.addEdge(1, 4);
    g2.addEdge(2, 4);
    g2.addEdge(4, 3);
    cout << "\nColoring of graph 2 \n";
    g2.greedyColoring();

    return 0;
}

```

Java

```
// A Java program to implement greedy algorithm for graph coloring
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents an undirected graph using adjacency list
class Graph
{
    private int V; // No. of vertices
    private LinkedList<Integer> adj[]; //Adjacency List

    //Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    //Function to add an edge into the graph
    void addEdge(int v,int w)
    {
        adj[v].add(w);
        adj[w].add(v); //Graph is undirected
    }

    // Assigns colors (starting from 0) to all vertices and
    // prints the assignment of colors
    void greedyColoring()
    {
        int result[] = new int[V];

        // Initialize all vertices as unassigned
        Arrays.fill(result, -1);

        // Assign the first color to first vertex
        result[0] = 0;

        // A temporary array to store the available colors. False
        // value of available[cr] would mean that the color cr is
        // assigned to one of its adjacent vertices
        boolean available[] = new boolean[V];

        // Initially, all colors are available
        Arrays.fill(available, true);
```

```
// Assign colors to remaining V-1 vertices
for (int u = 1; u < V; u++)
{
    // Process all adjacent vertices and flag their colors
    // as unavailable
    Iterator<Integer> it = adj[u].iterator() ;
    while (it.hasNext())
    {
        int i = it.next();
        if (result[i] != -1)
            available[result[i]] = false;
    }

    // Find the first available color
    int cr;
    for (cr = 0; cr < V; cr++){
        if (available[cr])
            break;
    }

    result[u] = cr; // Assign the found color

    // Reset the values back to true for the next iteration
    Arrays.fill(available, true);
}

// print the result
for (int u = 0; u < V; u++)
    System.out.println("Vertex " + u + " ---> Color "
                      + result[u]);
}

// Driver method
public static void main(String args[])
{
    Graph g1 = new Graph(5);
    g1.addEdge(0, 1);
    g1.addEdge(0, 2);
    g1.addEdge(1, 2);
    g1.addEdge(1, 3);
    g1.addEdge(2, 3);
    g1.addEdge(3, 4);
    System.out.println("Coloring of graph 1");
    g1.greedyColoring();

    System.out.println();
    Graph g2 = new Graph(5);
```

```
g2.addEdge(0, 1);
g2.addEdge(0, 2);
g2.addEdge(1, 2);
g2.addEdge(1, 4);
g2.addEdge(2, 4);
g2.addEdge(4, 3);
System.out.println("Coloring of graph 2 ");
g2.greedyColoring();
}
}

// This code is contributed by Aakash Hasija
```

Output:

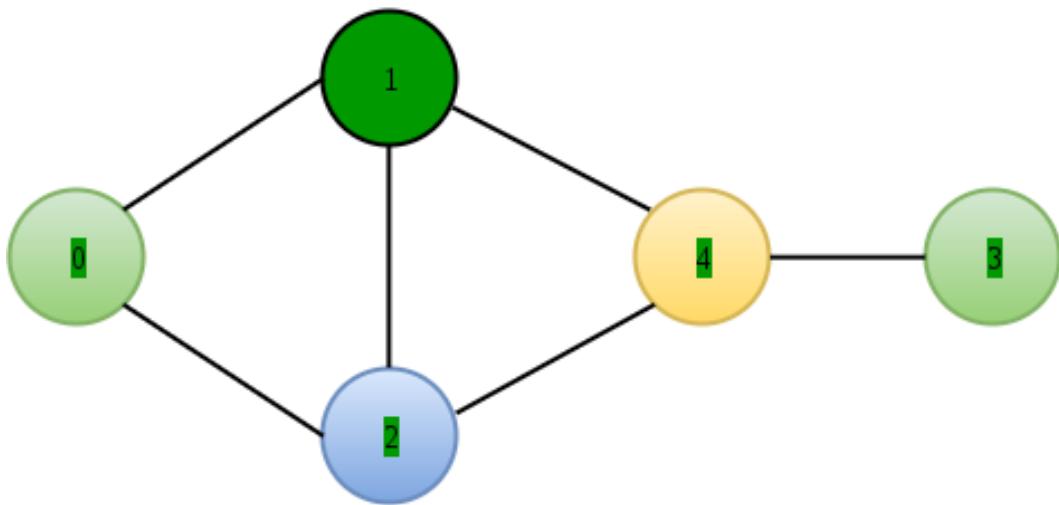
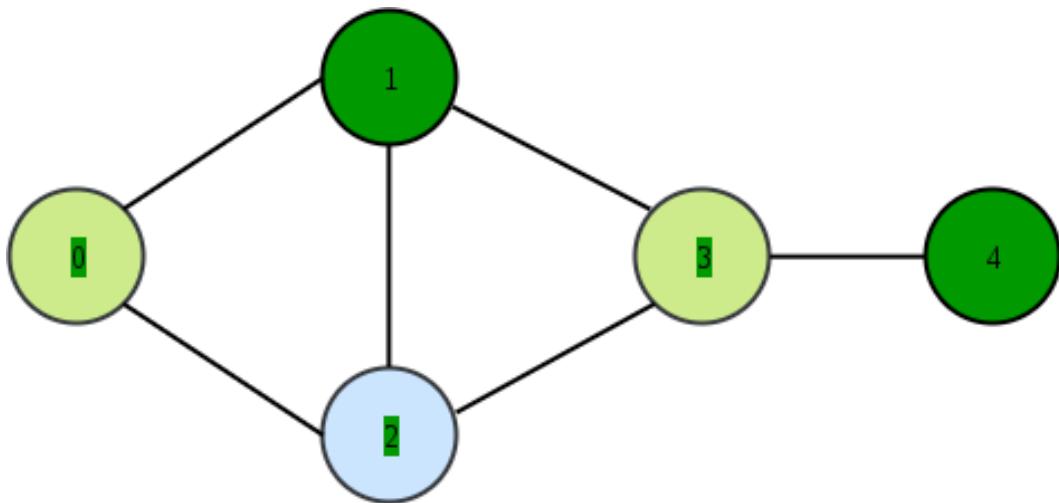
```
Coloring of graph 1
Vertex 0 ----> Color 0
Vertex 1 ----> Color 1
Vertex 2 ----> Color 2
Vertex 3 ----> Color 0
Vertex 4 ----> Color 1
```

```
Coloring of graph 2
Vertex 0 ----> Color 0
Vertex 1 ----> Color 1
Vertex 2 ----> Color 2
Vertex 3 ----> Color 0
Vertex 4 ----> Color 3
```

Time Complexity: $O(V^2 + E)$ in worst case.

Analysis of Basic Algorithm

The above algorithm doesn't always use minimum number of colors. Also, the number of colors used sometime depend on the order in which vertices are processed. For example, consider the following two graphs. Note that in graph on right side, vertices 3 and 4 are swapped. If we consider the vertices 0, 1, 2, 3, 4 in left graph, we can color the graph using 3 colors. But if we consider the vertices 0, 1, 2, 3, 4 in right graph, we need 4 colors.



So the order in which the vertices are picked is important. Many people have suggested different ways to find an ordering that work better than the basic algorithm on average. The most common is [Welsh–Powell Algorithm](#) which considers vertices in descending order of degrees.

How does the basic algorithm guarantee an upper bound of $d+1$?

Here d is the maximum degree in the given graph. Since d is maximum degree, a vertex cannot be attached to more than d vertices. When we color a vertex, at most d colors could have already been used by its adjacent. To color this vertex, we need to pick the smallest numbered color that is not used by the adjacent vertices. If colors are numbered like 1, 2, ..., then the value of such smallest number must be between 1 to $d+1$ (Note that d numbers are already picked by adjacent vertices).

This can also be proved using induction. See [this](#)video lecture for proof.

We will soon be discussing some interesting facts about chromatic number and graph coloring.

Improved By : [ChamanJhinga](#)

Source

<https://www.geeksforgeeks.org/graph-coloring-set-2-greedy-algorithm/>

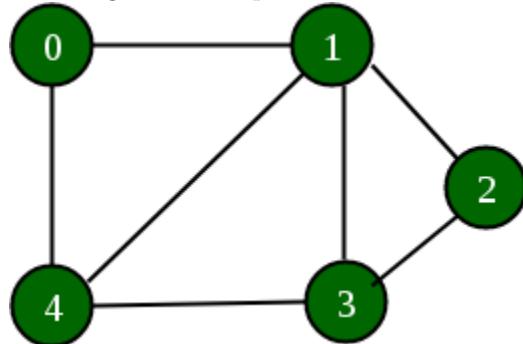
Chapter 168

Graph implementation using STL for competitive programming Set 1 (DFS of Unweighted and Undirected)

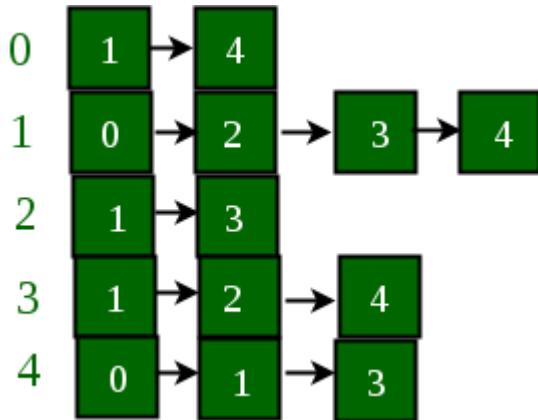
Graph implementation using STL for competitive programming Set 1 (DFS of Unweighted and Undirected) - GeeksforGeeks

We have introduced Graph basics in [Graph and its representations](#). In this post, a different STL based representation is used that can be helpful to quickly implement graph using [vectors](#). The implementation is for adjacency list representation of graph.

Following is an example undirected and unweighted graph with 5 vertices.



Below is adjacency list representation of the graph.



We use vector in STL to implement graph using adjacency list representation.

- **vector** : A sequence container. Here we use it to store adjacency lists of all vertices.
We use vertex number as index in this vector.

The idea is to represent graph as an array of vectors such that every vector represents adjacency list of a vertex. Below is complete STL based C++ program for [DFS Traversal](#).

```

// A simple representation of graph using STL,
// for the purpose of competitive programming
#include<bits/stdc++.h>
using namespace std;

// A utility function to add an edge in an
// undirected graph.
void addEdge(vector<int> adj[], int u, int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}

// A utility function to do DFS of graph
// recursively from a given vertex u.
void DFSUtil(int u, vector<int> adj[],
             vector<bool> &visited)
{
    visited[u] = true;
    cout << u << " ";
    for (int i=0; i<adj[u].size(); i++)
        if (visited[adj[u][i]] == false)
            DFSUtil(adj[u][i], adj, visited);
}

// This function does DFSUtil() for all
  
```

```
// unvisited vertices.  
void DFS(vector<int> adj[], int V)  
{  
    vector<bool> visited(V, false);  
    for (int u=0; u<V; u++)  
        if (visited[u] == false)  
            DFSUtil(u, adj, visited);  
}  
  
// Driver code  
int main()  
{  
    int V = 5;  
  
    // The below line may not work on all  
    // compilers. If it does not work on  
    // your compiler, please replace it with  
    // following  
    // vector<int> *adj = new vector<int>[V];  
    vector<int> adj[V];  
  
    // Vertex numbers should be from 0 to 4.  
    addEdge(adj, 0, 1);  
    addEdge(adj, 0, 4);  
    addEdge(adj, 1, 2);  
    addEdge(adj, 1, 3);  
    addEdge(adj, 1, 4);  
    addEdge(adj, 2, 3);  
    addEdge(adj, 3, 4);  
    DFS(adj, V);  
    return 0;  
}
```

Output :

0 1 2 3 4

Below are related articles:

[Graph implementation using STL for competitive programming Set 2 \(Weighted graph\)](#)
[Dijkstra's Shortest Path Algorithm using priority_queue of STL](#)
[Dijkstra's shortest path algorithm using set in STL](#)
[Kruskal's Minimum Spanning Tree using STL in C++](#)
[Prim's algorithm using priority_queue in STL](#)

Source

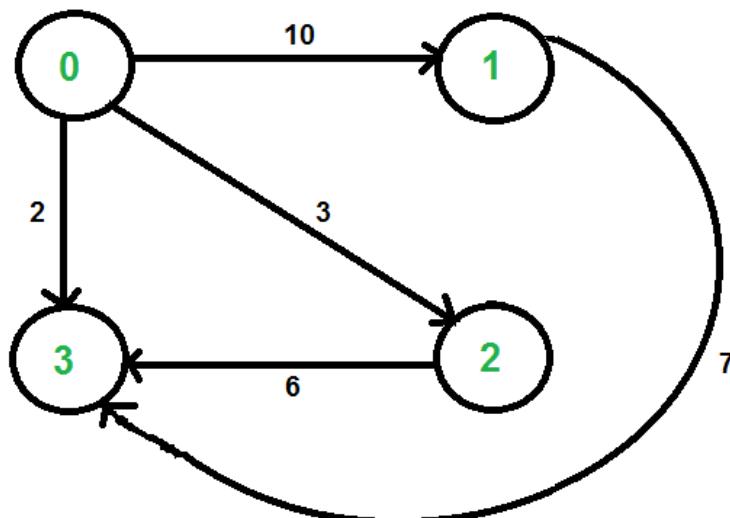
<https://www.geeksforgeeks.org/graph-implementation-using-stl-for-competitive-programming-set-1-dfs-of-unweight>

Chapter 169

Graph implementation using STL for competitive programming Set 2 (Weighted graph)

Graph implementation using STL for competitive programming Set 2 (Weighted graph) - GeeksforGeeks

In [Set 1](#), unweighted graph is discussed. In this post, weighted graph representation using STL is discussed. The implementation is for adjacency list representation of weighted graph.



We use two STL containers to represent graph:

- **vector** : A sequence container. Here we use it to store adjacency lists of all vertices. We use vertex number as index in this vector.
- **pair** : A simple container to store pair of elements. Here we use it to store adjacent vertex number and weight of edge connecting to the adjacent.

The idea is to use a vector of pair vectors. Below code implements the same.

```
// C++ program to represent undirected and weighted graph
// using STL. The program basically prints adjacency list
// representation of graph
#include <bits/stdc++.h>
using namespace std;

// To add an edge
void addEdge(vector<pair<int, int>> adj[], int u,
             int v, int wt)
{
    adj[u].push_back(make_pair(v, wt));
    adj[v].push_back(make_pair(u, wt));
}

// Print adjacency list representation of graph
void printGraph(vector<pair<int,int>> adj[], int V)
{
    int v, w;
    for (int u = 0; u < V; u++)
    {
        cout << "Node " << u << " makes an edge with \n";
        for (auto it = adj[u].begin(); it!=adj[u].end(); it++)
        {
            v = it->first;
            w = it->second;
            cout << "\tNode " << v << " with edge weight ="
                << w << "\n";
        }
        cout << "\n";
    }
}

// Driver code
int main()
{
    int V = 5;
    vector<pair<int, int>> adj[V];
    addEdge(adj, 0, 1, 10);
    addEdge(adj, 0, 4, 20);
    addEdge(adj, 1, 2, 30);
```

```
    addEdge(adj, 1, 3, 40);
    addEdge(adj, 1, 4, 50);
    addEdge(adj, 2, 3, 60);
    addEdge(adj, 3, 4, 70);
    printGraph(adj, V);
    return 0;
}
```

Output:

```
Node 0 makes an edge with
    Node 1 with edge weight =10
    Node 4 with edge weight =20

Node 1 makes an edge with
    Node 0 with edge weight =10
    Node 2 with edge weight =30
    Node 3 with edge weight =40
    Node 4 with edge weight =50

Node 2 makes an edge with
    Node 1 with edge weight =30
    Node 3 with edge weight =60

Node 3 makes an edge with
    Node 1 with edge weight =40
    Node 2 with edge weight =60
    Node 4 with edge weight =70

Node 4 makes an edge with
    Node 0 with edge weight =20
    Node 1 with edge weight =50
    Node 3 with edge weight =70
```

Improved By : [Ritesh Ghorse](#)

Source

<https://www.geeksforgeeks.org/graph-implementation-using-stl-for-competitive-programming-set-2-weighted-graph/>

Chapter 170

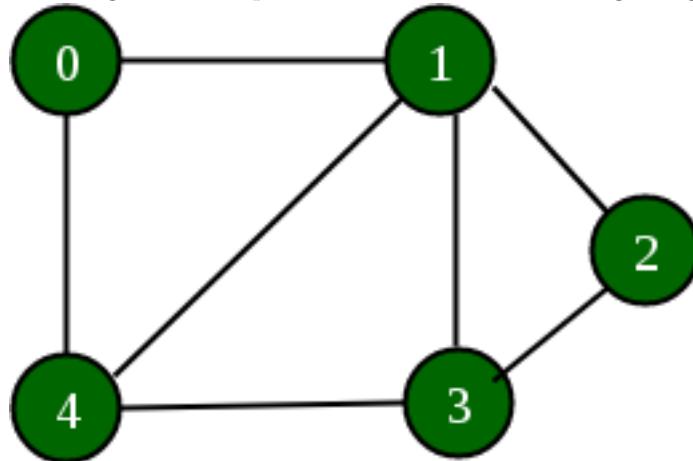
Graph representations using set and hash

Graph representations using set and hash - GeeksforGeeks

We have introduced Graph implementation using array of vectors in [Graph implementation using STL for competitive programming Set 1](#). In this post, a different implementation is used which can be used to implement graphs using [sets](#). The implementation is for [adjacency list representation of graph](#).

A set is different from a vector in two ways: it stores elements in a sorted way, and duplicate elements are not allowed. Therefore, this approach cannot be used for graphs containing parallel edges. Since sets are internally implemented as binary search trees, **an edge between two vertices can be searched in $O(\log V)$ time**, where V is the number of vertices in the graph.

Following is an example of an undirected and unweighted graph with 5 vertices.



Below is adjacency list representation of this graph using array of [sets](#).

0	1	4		
1	0	2	3	4
2	1	3		
3	1	2	4	
4	0	1	3	

Below is the code for adjacency list representation of an undirected graph using sets:

```

// A C++ program to demonstrate adjacency list
// representation of graphs using sets
#include <bits/stdc++.h>
using namespace std;

struct Graph {
    int V;
    set<int, greater<int> *> adjList;
};

// A utility function that creates a graph of V vertices
Graph* createGraph(int V)
{
    Graph* graph = new Graph;
    graph->V = V;

    // Create an array of sets representing
    // adjacency lists. Size of the array will be V
    graph->adjList = new set<int, greater<int> >[V];

    return graph;
}

// Adds an edge to an undirected graph
void addEdge(Graph* graph, int src, int dest)
{
    // Add an edge from src to dest. A new
    // element is inserted to the adjacent
    // list of src.
    graph->adjList[src].insert(dest);

    // Since graph is undirected, add an edge

```

```

        // from dest to src also
        graph->adjList[dest].insert(src);
    }

    // A utility function to print the adjacency
    // list representation of graph
    void printGraph(Graph* graph)
    {
        for (int i = 0; i < graph->V; ++i) {
            set<int, greater<int> > lst = graph->adjList[i];
            cout << endl << "Adjacency list of vertex "
            << i << endl;

            for (auto itr = lst.begin(); itr != lst.end(); ++itr)
                cout << *itr << " ";
            cout << endl;
        }
    }

    // Searches for a given edge in the graph
    void searchEdge(Graph* graph, int src, int dest)
    {
        auto itr = graph->adjList[src].find(dest);
        if (itr == graph->adjList[src].end())
            cout << endl << "Edge from " << src
            << " to " << dest << " not found."
            << endl;
        else
            cout << endl << "Edge from " << src
            << " to " << dest << " found."
            << endl;
    }

    // Driver code
    int main()
    {
        // Create the graph given in the above figure
        int V = 5;
        struct Graph* graph = createGraph(V);
        addEdge(graph, 0, 1);
        addEdge(graph, 0, 4);
        addEdge(graph, 1, 2);
        addEdge(graph, 1, 3);
        addEdge(graph, 1, 4);
        addEdge(graph, 2, 3);
        addEdge(graph, 3, 4);

        // Print the adjacency list representation of

```

```
// the above graph  
printGraph(graph);  
  
// Search the given edge in the graph  
searchEdge(graph, 2, 1);  
searchEdge(graph, 0, 3);  
  
return 0;  
}
```

Output:

```
Adjacency list of vertex 0  
4 1
```

```
Adjacency list of vertex 1  
4 3 2 0
```

```
Adjacency list of vertex 2  
3 1
```

```
Adjacency list of vertex 3  
4 2 1
```

```
Adjacency list of vertex 4  
3 1 0
```

```
Edge from 2 to 1 found.
```

```
Edge from 0 to 3 not found.
```

Pros: Queries like whether there is an edge from vertex u to vertex v can be done in $O(\log V)$.

Cons:

- Adding an edge takes $O(\log V)$, as opposed to $O(1)$ in vector implementation.
- Graphs containing parallel edge(s) cannot be implemented through this method.

Further Optimization of Edge Search Operation using `unordered_set` (or hashing):

The edge search operation can be further optimized to $O(1)$ using `unordered_set` which uses hashing internally.

```
// A C++ program to demonstrate adjacency list
```

```

// representation of graphs using sets
#include <bits/stdc++.h>
using namespace std;

struct Graph {
    int V;
    unordered_set<int>* adjList;
};

// A utility function that creates a graph of
// V vertices
Graph* createGraph(int V)
{
    Graph* graph = new Graph;
    graph->V = V;

    // Create an array of sets representing
    // adjacency lists. Size of the array will be V
    graph->adjList = new unordered_set<int>[V];

    return graph;
}

// Adds an edge to an undirected graph
void addEdge(Graph* graph, int src, int dest)
{
    // Add an edge from src to dest. A new
    // element is inserted to the adjacent
    // list of src.
    graph->adjList[src].insert(dest);

    // Since graph is undirected, add an edge
    // from dest to src also
    graph->adjList[dest].insert(src);
}

// A utility function to print the adjacency
// list representation of graph
void printGraph(Graph* graph)
{
    for (int i = 0; i < graph->V; ++i) {
        unordered_set<int> lst = graph->adjList[i];
        cout << endl << "Adjacency list of vertex "
        << i << endl;

        for (auto itr = lst.begin(); itr != lst.end(); ++itr)
            cout << *itr << " ";
        cout << endl;
}

```

```
        }
    }

// Searches for a given edge in the graph
void searchEdge(Graph* graph, int src, int dest)
{
    auto itr = graph->adjList[src].find(dest);
    if (itr == graph->adjList[src].end())
        cout << endl << "Edge from " << src
            << " to " << dest << " not found."
            << endl;
    else
        cout << endl << "Edge from " << src
            << " to " << dest << " found."
            << endl;
}

// Driver code
int main()
{
    // Create the graph given in the above figure
    int V = 5;
    struct Graph* graph = createGraph(V);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 4);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 4);

    // Print the adjacency list representation of
    // the above graph
    printGraph(graph);

    // Search the given edge in the graph
    searchEdge(graph, 2, 1);
    searchEdge(graph, 0, 3);

    return 0;
}
```

Output :

```
Adjacency list of vertex 0
4 1
```

```
Adjacency list of vertex 1  
4 3 2 0
```

```
Adjacency list of vertex 2  
3 1
```

```
Adjacency list of vertex 3  
4 2 1
```

```
Adjacency list of vertex 4  
3 1 0
```

Edge from 2 to 1 found.

Edge from 0 to 3 not found.

Pros:

- Queries like whether there is an edge from vertex u to vertex v can be done in O(1).
- Adding an edge takes O(1).

Cons:

- Graphs containing parallel edge(s) cannot be implemented through this method.
- Edges are not stored in any order.

Note : **adjacency matrix representation** is the most optimized for edge search, but space requirements of adjacency matrix are comparatively high for big sparse graphs. Moreover adjacency matrix has other disadvantages as well like **BFS** and **DFS** become costly as we can't quickly get all adjacent of a node.

Source

<https://www.geeksforgeeks.org/graph-representations-using-set-hash/>

Chapter 171

Hamiltonian Cycle Backtracking-6

Hamiltonian Cycle Backtracking-6 - GeeksforGeeks

Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in graph) from the last vertex to the first vertex of the Hamiltonian Path. Determine whether a given graph contains Hamiltonian Cycle or not. If it contains, then print the path. Following are the input and output of the required function.

Input:

A 2D array $\text{graph}[V][V]$ where V is the number of vertices in graph and $\text{graph}[V][V]$ is adjacency matrix representation of the graph. A value $\text{graph}[i][j]$ is 1 if there is a direct edge from i to j , otherwise $\text{graph}[i][j]$ is 0.

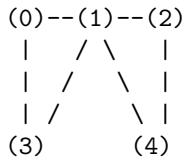
Output:

An array $\text{path}[V]$ that should contain the Hamiltonian Path. $\text{path}[i]$ should represent the i th vertex in the Hamiltonian Path. The code should also return false if there is no Hamiltonian Cycle in the graph.

For example, a Hamiltonian Cycle in the following graph is $\{0, 1, 2, 4, 3, 0\}$. There are more Hamiltonian Cycles in the graph like $\{0, 3, 4, 2, 1, 0\}$

```
(0)--(1)--(2)
 |   / \   |
 |   /   \   |
 | /       \ |
(3)-----(4)
```

And the following graph doesn't contain any Hamiltonian Cycle.



Naive Algorithm

Generate all possible configurations of vertices and print a configuration that satisfies the given constraints. There will be $n!$ (n factorial) configurations.

```
while there are untried conflagrations
{
    generate the next configuration
    if ( there are edges between two consecutive vertices of this
        configuration and there is an edge from the last vertex to
        the first ).
    {
        print this configuration;
        break;
    }
}
```

Backtracking Algorithm

Create an empty path array and add vertex 0 to it. Add other vertices, starting from the vertex 1. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added. If we find such a vertex, we add the vertex as part of the solution. If we do not find a vertex then we return false.

Implementation of Backtracking solution

Following are implementations of the Backtracking solution.

C/C++

```
/* C/C++ program for solution of Hamiltonian Cycle problem
   using backtracking */
#include<stdio.h>

// Number of vertices in the graph
#define V 5

void printSolution(int path[]);

/* A utility function to check if the vertex v can be added at
   index 'pos' in the Hamiltonian Cycle constructed so far (stored
   in 'path[]') */
bool isSafe(int v, bool graph[V][V], int path[], int pos)
{
```

```

/* Check if this vertex is an adjacent vertex of the previously
   added vertex. */
if (graph [ path[pos-1] ][ v ] == 0)
   return false;

/* Check if the vertex has already been included.
   This step can be optimized by creating an array of size V */
for (int i = 0; i < pos; i++)
   if (path[i] == v)
      return false;

return true;
}

/* A recursive utility function to solve hamiltonian cycle problem */
bool hamCycleUtil(bool graph[V][V], int path[], int pos)
{
    /* base case: If all vertices are included in Hamiltonian Cycle */
    if (pos == V)
    {
        // And if there is an edge from the last included vertex to the
        // first vertex
        if ( graph[ path[pos-1] ][ path[0] ] == 1 )
            return true;
        else
            return false;
    }

    // Try different vertices as a next candidate in Hamiltonian Cycle.
    // We don't try for 0 as we included 0 as starting point in in hamCycle()
    for (int v = 1; v < V; v++)
    {
        /* Check if this vertex can be added to Hamiltonian Cycle */
        if (isSafe(v, graph, path, pos))
        {
            path[pos] = v;

            /* recur to construct rest of the path */
            if (hamCycleUtil (graph, path, pos+1) == true)
                return true;

            /* If adding vertex v doesn't lead to a solution,
               then remove it */
            path[pos] = -1;
        }
    }

    /* If no vertex can be added to Hamiltonian Cycle constructed so far,

```

```

        then return false */
    return false;
}

/* This function solves the Hamiltonian Cycle problem using Backtracking.
It mainly uses hamCycleUtil() to solve the problem. It returns false
if there is no Hamiltonian Cycle possible, otherwise return true and
prints the path. Please note that there may be more than one solutions,
this function prints one of the feasible solutions. */
bool hamCycle(bool graph[V][V])
{
    int *path = new int[V];
    for (int i = 0; i < V; i++)
        path[i] = -1;

    /* Let us put vertex 0 as the first vertex in the path. If there is
       a Hamiltonian Cycle, then the path can be started from any point
       of the cycle as the graph is undirected */
    path[0] = 0;
    if ( hamCycleUtil(graph, path, 1) == false )
    {
        printf("\nSolution does not exist");
        return false;
    }

    printSolution(path);
    return true;
}

/* A utility function to print solution */
void printSolution(int path[])
{
    printf ("Solution Exists:\n"
           " Following is one Hamiltonian Cycle \n");
    for (int i = 0; i < V; i++)
        printf(" %d ", path[i]);

    // Let us print the first vertex again to show the complete cycle
    printf(" %d ", path[0]);
    printf("\n");
}

// driver program to test above function
int main()
{
    /* Let us create the following graph
       (0)--(1)--(2)
          |     |
          / \   /

```

```

    | / \ |
    | / \ |
(3)-----(4) */
bool graph1[V][V] = {{0, 1, 0, 1, 0},
                      {1, 0, 1, 1, 1},
                      {0, 1, 0, 0, 1},
                      {1, 1, 0, 0, 1},
                      {0, 1, 1, 1, 0},
};

// Print the solution
hamCycle(graph1);

/* Let us create the following graph
(0)--(1)--(2)
    | / \ |
    | / \ |
    | / \ |
(3)      (4) */
bool graph2[V][V] = {{0, 1, 0, 1, 0},
                      {1, 0, 1, 1, 1},
                      {0, 1, 0, 0, 1},
                      {1, 1, 0, 0, 0},
                      {0, 1, 1, 0, 0},
};

// Print the solution
hamCycle(graph2);

return 0;
}

```

Java

```

/* Java program for solution of Hamiltonian Cycle problem
   using backtracking */
class HamiltonianCycle
{
    final int V = 5;
    int path[];

    /* A utility function to check if the vertex v can be
       added at index 'pos' in the Hamiltonian Cycle
       constructed so far (stored in 'path[]') */
    boolean isSafe(int v, int graph[][] , int path[], int pos)
    {
        /* Check if this vertex is an adjacent vertex of
           the previously added vertex. */

```

```

if (graph[path[pos - 1]][v] == 0)
    return false;

/* Check if the vertex has already been included.
   This step can be optimized by creating an array
   of size V */
for (int i = 0; i < pos; i++)
    if (path[i] == v)
        return false;

return true;
}

/* A recursive utility function to solve hamiltonian
   cycle problem */
boolean hamCycleUtil(int graph[][], int path[], int pos)
{
    /* base case: If all vertices are included in
       Hamiltonian Cycle */
    if (pos == V)
    {
        // And if there is an edge from the last included
        // vertex to the first vertex
        if (graph[path[pos - 1]][path[0]] == 1)
            return true;
        else
            return false;
    }

    // Try different vertices as a next candidate in
    // Hamiltonian Cycle. We don't try for 0 as we
    // included 0 as starting point in in hamCycle()
    for (int v = 1; v < V; v++)
    {
        /* Check if this vertex can be added to Hamiltonian
           Cycle */
        if (isSafe(v, graph, path, pos))
        {
            path[pos] = v;

            /* recur to construct rest of the path */
            if (hamCycleUtil(graph, path, pos + 1) == true)
                return true;

            /* If adding vertex v doesn't lead to a solution,
               then remove it */
            path[pos] = -1;
        }
    }
}

```

```
}

/* If no vertex can be added to Hamiltonian Cycle
   constructed so far, then return false */
return false;
}

/* This function solves the Hamiltonian Cycle problem using
   Backtracking. It mainly uses hamCycleUtil() to solve the
   problem. It returns false if there is no Hamiltonian Cycle
   possible, otherwise return true and prints the path.
   Please note that there may be more than one solutions,
   this function prints one of the feasible solutions. */
int hamCycle(int graph[][])
{
    path = new int[V];
    for (int i = 0; i < V; i++)
        path[i] = -1;

    /* Let us put vertex 0 as the first vertex in the path.
       If there is a Hamiltonian Cycle, then the path can be
       started from any point of the cycle as the graph is
       undirected */
    path[0] = 0;
    if (hamCycleUtil(graph, path, 1) == false)
    {
        System.out.println("\nSolution does not exist");
        return 0;
    }

    printSolution(path);
    return 1;
}

/* A utility function to print solution */
void printSolution(int path[])
{
    System.out.println("Solution Exists: Following" +
                       " is one Hamiltonian Cycle");
    for (int i = 0; i < V; i++)
        System.out.print(" " + path[i] + " ");

    // Let us print the first vertex again to show the
    // complete cycle
    System.out.println(" " + path[0] + " ");
}

// driver program to test above function
```

```
public static void main(String args[])
{
    HamiltonianCycle hamiltonian =
        new HamiltonianCycle();
    /* Let us create the following graph
    (0)--(1)--(2)
    |   / \   |
    |   /     \ |
    | /         \ |
    (3)-----(4) */
    int graph1[][] = {{0, 1, 0, 1, 0},
                      {1, 0, 1, 1, 1},
                      {0, 1, 0, 0, 1},
                      {1, 1, 0, 0, 1},
                      {0, 1, 1, 1, 0},
    };
    // Print the solution
    hamiltonian.hamCycle(graph1);

    /* Let us create the following graph
    (0)--(1)--(2)
    |   / \   |
    |   /     \ |
    | /         \ |
    (3)----- (4) */
    int graph2[][] = {{0, 1, 0, 1, 0},
                      {1, 0, 1, 1, 1},
                      {0, 1, 0, 0, 1},
                      {1, 1, 0, 0, 0},
                      {0, 1, 1, 0, 0},
    };
    // Print the solution
    hamiltonian.hamCycle(graph2);
}
// This code is contributed by Abhishek Shankhadhar
```

Python

```
# Python program for solution of
# hamiltonian cycle problem

class Graph():
    def __init__(self, vertices):
        self.graph = [[0 for column in range(vertices)]\
                     for row in range(vertices)]
```

```

self.V = vertices

''' Check if this vertex is an adjacent vertex
   of the previously added vertex and is not
   included in the path earlier '''
def isSafe(self, v, pos, path):
    # Check if current vertex and last vertex
    # in path are adjacent
    if self.graph[ path[pos-1] ][v] == 0:
        return False

    # Check if current vertex not already in path
    for vertex in path:
        if vertex == v:
            return False

    return True

# A recursive utility function to solve
# hamiltonian cycle problem
def hamCycleUtil(self, path, pos):

    # base case: if all vertices are
    # included in the path
    if pos == self.V:
        # Last vertex must be adjacent to the
        # first vertex in path to make a cycle
        if self.graph[ path[pos-1] ][ path[0] ] == 1:
            return True
        else:
            return False

    # Try different vertices as a next candidate
    # in Hamiltonian Cycle. We don't try for 0 as
    # we included 0 as starting point in in hamCycle()
    for v in range(1,self.V):

        if self.isSafe(v, pos, path) == True:

            path[pos] = v

            if self.hamCycleUtil(path, pos+1) == True:
                return True

            # Remove current vertex if it doesn't
            # lead to a solution
            path[pos] = -1

```

```

        return False

def hamCycle(self):
    path = [-1] * self.V

    ''' Let us put vertex 0 as the first vertex
        in the path. If there is a Hamiltonian Cycle,
        then the path can be started from any point
        of the cycle as the graph is undirected '''
    path[0] = 0

    if self.hamCycleUtil(path,1) == False:
        print "Solution does not exist\n"
        return False

    self.printSolution(path)
    return True

def printSolution(self, path):
    print "Solution Exists: Following is one Hamiltonian Cycle"
    for vertex in path:
        print vertex,
    print path[0], "\n"

# Driver Code

''' Let us create the following graph
    (0)--(1)--(2)
    |   / \   |
    |   /     \   |
    | /         \   |
    (3)-----(4)      ...
g1 = Graph(5)
g1.graph = [ [0, 1, 0, 1, 0], [1, 0, 1, 1, 1],
            [0, 1, 0, 0, 1],[1, 1, 0, 0, 1],
            [0, 1, 1, 1, 0], ]

# Print the solution
g1.hamCycle();

''' Let us create the following graph
    (0)--(1)--(2)
    |   / \   |
    |   /     \   |
    | /         \   |
    (3)      (4)      ...
g2 = Graph(5)
g2.graph = [ [0, 1, 0, 1, 0], [1, 0, 1, 1, 1],
            [0, 1, 0, 0, 1],[1, 1, 0, 0, 1],
            [0, 1, 1, 1, 0], ]

```

```
[0, 1, 0, 0, 1], [1, 1, 0, 0, 0],  
[0, 1, 1, 0, 0], ]  
  
# Print the solution  
g2.hamCycle();  
  
# This code is contributed by Divyanshu Mehta
```

Output:

```
Solution Exists: Following is one Hamiltonian Cycle  
0 1 2 4 3 0
```

```
Solution does not exist
```

Note that the above code always prints cycle starting from 0. Starting point should not matter as cycle can be started from any point. If you want to change the starting point, you should make two changes to above code.

Change “path[0] = 0;” to “path[0] = s;” where s is your new starting point. Also change loop “for (int v = 1; v < V; v++)” in hamCycleUtil() to ”for (int v = 0; v < V; v++)”. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

<https://www.geeksforgeeks.org/hamiltonian-cycle-backtracking-6/>

Chapter 172

Height of a generic tree from parent array

Height of a generic tree from parent array - GeeksforGeeks

We are given a tree of size n as array parent[0..n-1] where every index i in parent[] represents a node and the value at i represents the immediate parent of that node. For root node value will be -1. Find the height of the generic tree given the parent links.

Examples:

```
Input : parent[] = {-1, 0, 0, 0, 3, 1, 1, 2}
Output : 2
```

```
Input : parent[] = {-1, 0, 1, 2, 3}
Output : 4
```

Here, generic tree is sometimes also called as N-ary tree or N-way tree where N denotes the maximum number of child a node can have. In this problem array represents n number of nodes in the tree.

Approach 1:

One solution is to traverse up the tree from node till root node is reached with node value -1. While Traversing for each node store maximum path length.

Time Complexity of this solution is $O(n^2)$.

Approach 2:

Build graph for N-ary Tree in $O(n)$ time and apply BFS on the stored graph in $O(n)$ time and while doing BFS store maximum reached level. This solution does two iterations to find the height of N-ary tree.

```
// C++ code to find height of N-ary
// tree in O(n)
```

```
#include <bits/stdc++.h>
#define MAX 1001
using namespace std;

// Adjacency list to
// store N-ary tree
vector<int> adj[MAX];

// Build tree in tree in O(n)
int build_tree(int arr[], int n)
{
    int root_index = 0;

    // Iterate for all nodes
    for (int i = 0; i < n; i++) {

        // if root node, store index
        if (arr[i] == -1)
            root_index = i;

        else {
            adj[i].push_back(arr[i]);
            adj[arr[i]].push_back(i);
        }
    }
    return root_index;
}

// Applying BFS
int BFS(int start)
{
    // map is used as visited array
    map<int, int> vis;

    queue<pair<int, int> > q;
    int max_level_reached = 0;

    // height of root node is zero
    q.push({ start, 0 });

    // p.first denotes node in adjacency list
    // p.second denotes level of p.first
    pair<int, int> p;

    while (!q.empty()) {

        p = q.front();
        vis[p.first] = 1;
```

```

// store the maximum level reached
max_level_reached = max(max_level_reached,
                        p.second);

q.pop();

for (int i = 0; i < adj[p.first].size(); i++)

    // adding 1 to previous level
    // stored on node p.first
    // which is parent of node adj[p.first][i]
    // if adj[p.first][i] is not visited
    if (!vis[adj[p.first][i]])
        q.push({adj[p.first][i], p.second + 1});
}

return max_level_reached;
}

// Driver Function
int main()
{
    // node 0 to node n-1
    int parent[] = { -1, 0, 1, 2, 3 };

    // Number of nodes in tree
    int n = sizeof(parent) / sizeof(parent[0]);

    int root_index = build_tree(parent, n);

    int ma = BFS(root_index);
    cout << "Height of N-ary Tree=" << ma;
    return 0;
}

```

Output:

Height of N-ary Tree=4

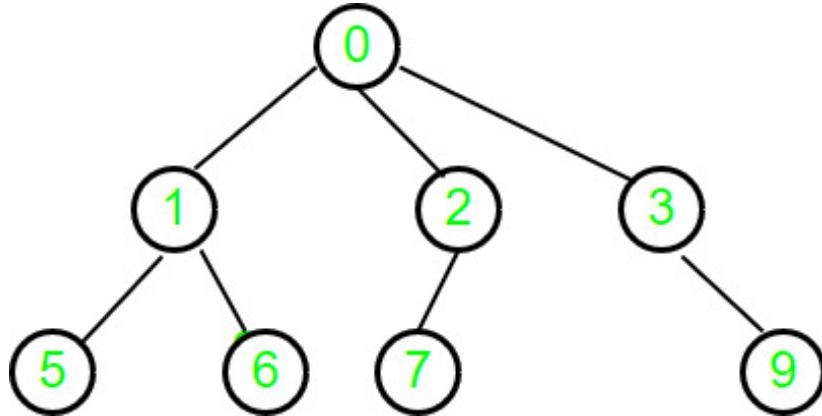
Time Complexity of this solution is **O(2n)** which converges to O(n) for very large n.

Approach 3:

We can find the height of N-ary Tree in only one iteration. We visit nodes from 0 to n-1 iteratively and mark the unvisited ancestors recursively if they are not visited before till we reach a node which is visited or we reach root node. If we reach visited node while

traversing up the tree using parent links, then we use its height and will not go further in recursion.

Explanation For Example 1::



For **node 0** : Check for Root node is true,
Return 0 as height, Mark node 0 as visited

For **node 1** : Recur for immediate ancestor, i.e 0, which is already visited
So, Use it's height and return height(node 0) +1
Mark node 1 as visited

For **node 2** : Recur for immediate ancestor, i.e 0, which is already visited
So, Use it's height and return height(node 0) +1
Mark node 2 as visited

For **node 3** : Recur for immediate ancestor, i.e 0, which is already visited
So, Use it's height and return height(node 0) +1
Mark node 3 as visited

For **node 4** : Recur for immediate ancestor, i.e 3, which is already visited
So, Use it's height and return height(node 3) +1
Mark node 3 as visited

For **node 5** : Recur for immediate ancestor, i.e 1, which is already visited
So, Use it's height and return height(node 1) +1
Mark node 5 as visited

For **node 6** : Recur for immediate ancestor, i.e 1, which is already visited
So, Use it's height and return height(node 1) +1
Mark node 6 as visited

For **node 7** : Recur for immediate ancestor, i.e 2, which is already visited
So, Use it's height and return height(node 2) +1
Mark node 7 as visited

Hence, we processed each node in N-ary tree only once.

```

// C++ code to find height of N-ary
// tree in O(n) (Efficient Approach)
#include <bits/stdc++.h>
using namespace std;
  
```

```
// Recur For Ancestors of node and
// store height of node at last
int fillHeight(int p[], int node, int visited[],
               int height[])
{
    // If root node
    if (p[node] == -1) {

        // mark root node as visited
        visited[node] = 1;
        return 0;
    }

    // If node is already visited
    if (visited[node])
        return height[node];

    // Visit node and calculate its height
    visited[node] = 1;

    // recur for the parent node
    height[node] = 1 + fillHeight(p, p[node],
                                  visited, height);

    // return calculated height for node
    return height[node];
}

int findHeight(int parent[], int n)
{
    // To store max height
    int ma = 0;

    // To check whether or not node is visited before
    int visited[n];

    // For Storing Height of node
    int height[n];

    memset(visited, 0, sizeof(visited));
    memset(height, 0, sizeof(height));

    for (int i = 0; i < n; i++) {

        // If not visited before
        if (!visited[i])
            height[i] = fillHeight(parent, i,
                                   visited, height);
    }
}
```

```
// store maximum height so far
ma = max(ma, height[i]);
}

return ma;
}

// Driver Function
int main()
{
    int parent[] = { -1, 0, 0, 0, 3, 1, 1, 2 };
    int n = sizeof(parent) / sizeof(parent[0]);

    cout << "Height of N-ary Tree = "
        << findHeight(parent, n);
    return 0;
}
```

Output:

Height of N-ary Tree = 2

Time Complexity: O(n)

Source

<https://www.geeksforgeeks.org/height-generic-tree-parent-array/>

Chapter 173

Hierholzer's Algorithm for directed graph

Hierholzer's Algorithm for directed graph - GeeksforGeeks

Given a directed Eulerian graph, print an [Euler circuit](#). Euler circuit is a path that traverses every edge of a graph, and the path ends on the starting vertex.

Examples:

Input : Adjacency list for the below graph

Output : 0 → 1 → 2 → 0

Input : Adjacency list for the below graph

Output : 0 → 6 → 4 → 5 → 0 → 1
→ 2 → 3 → 4 → 2 → 0

Explanation:

In both the cases, we can trace the Euler circuit by following the edges as indicated in the output.

We have discussed the [problem of finding out whether a given graph is Eulerian or not](#). In this post, an algorithm to print Eulerian trail or circuit is discussed. The same problem can be solved using [Fleury's Algorithm](#), however its complexity is $O(E^*E)$. Using Hierholzer's Algorithm, we can find the circuit/path in $O(E)$, i.e., linear time.

Below is the Algorithm: ref ([wiki](#)). Remember that a directed graph has an Eulerian cycle if following conditions are true (1) All vertices with nonzero degree belong to a single strongly connected component. (2) In degree and out degree of every vertex is same. The algorithm assumes that the given graph has Eulerian Circuit.

- Choose any starting vertex v , and follow a trail of edges from that vertex until returning to v . It is not possible to get stuck at any vertex other than v , because indegree and outdegree of every vertex must be same, when the trail enters another vertex w there must be an unused edge leaving w .
The tour formed in this way is a closed tour, but may not cover all the vertices and edges of the initial graph.
- As long as there exists a vertex u that belongs to the current tour but that has adjacent edges not part of the tour, start another trail from u , following unused edges until returning to u , and join the tour formed in this way to the previous tour.

Thus the idea is to keep following unused edges and removing them until we get stuck. Once we get stuck, we back-track to the nearest vertex in our current path that has unused edges, and we repeat the process until all the edges have been used. We can use another container to maintain the final path.

Let's take an example:

Let the initial directed graph be as below

Let's start our path from 0.

Thus, curr_path = {0} and circuit = {}

Now let's use the edge 0->1

Now, curr_path = {0,1} and circuit = {}

similarly we reach up to 2 and then to 0 again as

Now, curr_path = {0,1,2} and circuit = {}

Then we go to 0, now since 0 haven't got any unused edge we put 0 in circuit and back track till we find an edge

We then have curr_path = {0,1,2} and circuit = {0}

Similarly when we backtrack to 2, we don't find any unused edge. Hence put 2 in circuit and backtrack again.

curr_path = {0,1} and circuit = {0,2}

After reaching 1 we go to through unused edge 1->3 and then 3->4, 4->1 until all edges have been traversed.

The contents of the two containers look as:

curr_path = {0,1,3,4,1} and circuit = {0,2}

now as all edges have been used, the curr_path is popped one by one into circuit.

Finally we've circuit = {0,2,1,4,3,1,0}

We print the circuit in reverse to obtain the path followed.
i.e., 0->1->3->4->1->1->2->0

Below is the C++ program for the same.

```
// A C++ program to print Eulerian circuit in given
// directed graph using Hierholzer algorithm
#include <bits/stdc++.h>
using namespace std;

void printCircuit(vector< vector<int> > adj)
{
    // adj represents the adjacency list of
    // the directed graph
    // edge_count represents the number of edges
    // emerging from a vertex
    unordered_map<int,int> edge_count;

    for (int i=0; i<adj.size(); i++)
    {
        //find the count of edges to keep track
        //of unused edges
        edge_count[i] = adj[i].size();
    }

    if (!adj.size())
        return; //empty graph

    // Maintain a stack to keep vertices
    stack<int> curr_path;

    // vector to store final circuit
    vector<int> circuit;

    // start from any vertex
    curr_path.push(0);
    int curr_v = 0; // Current vertex

    while (!curr_path.empty())
    {
        // If there's remaining edge
        if (edge_count[curr_v])
        {
            // Push the vertex
            curr_path.push(curr_v);
```

```
// Find the next vertex using an edge
int next_v = adj[curr_v].back();

// and remove that edge
edge_count[curr_v]--;
adj[curr_v].pop_back();

// Move to next vertex
curr_v = next_v;
}

// back-track to find remaining circuit
else
{
    circuit.push_back(curr_v);

    // Back-tracking
    curr_v = curr_path.top();
    curr_path.pop();
}
}

// we've got the circuit, now print it in reverse
for (int i=circuit.size()-1; i>=0; i--)
{
    cout << circuit[i];
    if (i)
        cout<<" -> ";
}
}

// Driver program to check the above function
int main()
{
    vector< vector<int> > adj1, adj2;

    // Input Graph 1
    adj1.resize(3);

    // Build the edges
    adj1[0].push_back(1);
    adj1[1].push_back(2);
    adj1[2].push_back(0);
    printCircuit(adj1);
    cout << endl;

    // Input Graph 2
    adj2.resize(7);
```

```
adj2[0].push_back(1);
adj2[0].push_back(6);
adj2[1].push_back(2);
adj2[2].push_back(0);
adj2[2].push_back(3);
adj2[3].push_back(4);
adj2[4].push_back(2);
adj2[4].push_back(5);
adj2[5].push_back(0);
adj2[6].push_back(4);
printCircuit(adj2);

return 0;
}
```

Output:

```
0 -> 1 -> 2 -> 0
0 -> 6 -> 4 -> 5 -> 0 -> 1 -> 2 -> 3 -> 4 -> 2 -> 0
```

Time Complexity : $O(V+E)$.

Source

<https://www.geeksforgeeks.org/hierholzers-algorithm-directed-graph/>

Chapter 174

Hopcroft-Karp Algorithm for Maximum Matching Set 1 (Introduction)

Hopcroft-Karp Algorithm for Maximum Matching Set 1 (Introduction) - GeeksforGeeks

A matching in a [Bipartite Graph](#) is a set of the edges chosen in such a way that no two edges share an endpoint. A maximum matching is a matching of maximum size (maximum number of edges). In a maximum matching, if any edge is added to it, it is no longer a matching. There can be more than one maximum matching for a given Bipartite Graph.

We have discussed importance of maximum matching and [Ford Fulkerson Based approach for maximal Bipartite Matching](#) in [previous post](#). Time complexity of the Ford Fulkerson based algorithm is $O(V \times E)$.

Hopcroft Karp algorithm is an improvement that runs in $O(\sqrt{V} \times E)$ time. Let us define few terms before we discuss the algorithm

Free Node or Vertex: Given a matching M, a node that is not part of matching is called free node. Initially all vertices are free (See first graph of below diagram). In second graph, u2 and v2 are free. In third graph, no vertex is free.

Matching and Not-Matching edges: Given a matching M, edges that are part of matching are called Matching edges and edges that are not part of M (or connect free nodes) are called Not-Matching edges. In first graph, all edges are non-matching. In second graph, (u0, v1), (u1, v0) and (u3, v3) are matching and others not-matching.

Alternating Paths: Given a matching M, an alternating path is a path in which the edges belong alternatively to the matching and not matching. All single edges paths are alternating paths. Examples of alternating paths in middle graph are u0-v1-u2 and u2-v1-u0-v2.

Augmenting path: Given a matching M, an augmenting path is an alternating path that starts from and ends on free vertices. All single edge paths that start and end with free vertices are augmenting paths. In below diagram, augmenting paths are highlighted with blue color. Note that the augmenting path always has one extra matching edge.

The Hopcroft Karp algorithm is based on below concept.

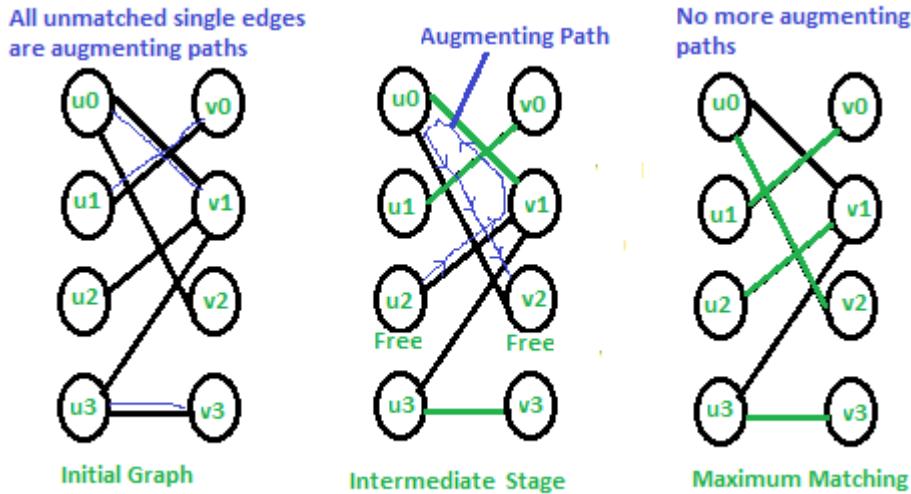
A matching M is not maximum if there exists an augmenting path. It is also true other way, i.e, a matching is maximum if no augmenting path exists

So the idea is to one by one look for augmenting paths. And add the found paths to current matching.

Hopcroft Karp Algorithm

- 1) Initialize Maximal Matching M as empty.
- 2) While there exists an Augmenting Path p
 - Remove matching edges of p from M and add not-matching edges of p to M
(This increases size of M by 1 as p starts and ends with a free vertex)
- 3) Return M .

Below diagram shows working of the algorithm.



In the initial graph all single edges are augmenting paths and we can pick in any order. In the middle stage, there is only one augmenting path. We remove matching edges of this path from M and add not-matching edges. In final matching, there are no augmenting paths so the matching is maximum.

Implementation of Hopcroft Karp algorithm is discussed in set 2.

Hopcroft-Karp Algorithm for Maximum Matching Set 2 (Implementation)

References:

- https://en.wikipedia.org/wiki/Hopcroft%E2%80%93Karp_algorithm
- <http://www.dis.uniroma1.it/~leon/tcs/lecture2.pdf>

Source

<https://www.geeksforgeeks.org/hopcroft-karp-algorithm-for-maximum-matching-set-1-introduction/>

Chapter 175

Hungarian Algorithm for Assignment Problem Set 1 (Introduction)

Hungarian Algorithm for Assignment Problem Set 1 (Introduction) - GeeksforGeeks

Let there be n agents and n tasks. Any agent can be assigned to perform any task, incurring some cost that may vary depending on the agent-task assignment. It is required to perform all tasks by assigning exactly one agent to each task and exactly one task to each agent in such a way that the total cost of the assignment is minimized.

Example: You work as a manager for a chip manufacturer, and you currently have 3 people on the road meeting clients. Your salespeople are in Jaipur, Pune and Bangalore, and you want them to fly to three other cities: Delhi, Mumbai and Kerala. The table below shows the cost of airline tickets in INR between the cities:

	Delhi	Kerala	Mumbai
Jaipur	2500	4000	3500
Pune	4000	6000	3500
Bangalore	2000	4000	2500

The question: where would you send each of your salespeople in order to minimize fair?

Possible assignment: Cost = 11000 INR

	Delhi	Kerala	Mumbai
Jaipur	2500	4000	3500
Pune	4000	6000	3500
Bangalore	2000	4000	2500

Other Possible assignment: Cost = **9500** INR and this is the best of the **3!** possible assignments.

	Delhi	Kerala	Mumbai
Jaipur	2500	4000	3500
Pune	4000	6000	3500
Bangalore	2000	4000	2500

Brute force solution is to consider every possible assignment implies a complexity of $\Omega(n!)$.

The **Hungarian algorithm**, aka **Munkres assignment algorithm**, utilizes the following theorem for polynomial runtime complexity (**worst case $O(n^3)$**) and guaranteed optimality: *If a number is added to or subtracted from all of the entries of any one row or column of a cost matrix, then an optimal assignment for the resulting cost matrix is also an optimal assignment for the original cost matrix.*

We reduce our original weight matrix to contain zeros, by using the above theorem. We try to assign tasks to agents such that each agent is doing only one task and the penalty incurred in each case is **zero**.

Core of the algorithm (assuming square matrix):

1. For each row of the matrix, find the smallest element and subtract it from every element in its row.
2. Do the same (as step 1) for all columns.
3. Cover all zeros in the matrix using minimum number of horizontal and vertical lines.
4. *Test for Optimality:* If the minimum number of covering lines is n, an optimal assignment is possible and we are finished. Else if lines are lesser than n, we haven't found the optimal assignment, and must proceed to step 5.
5. Determine the smallest entry not covered by any line. Subtract this entry from each uncovered row, and then add it to each covered column. Return to step 3.

Explanation for above simple example:

Below is the cost matrix of example given in above diagrams.

2500	4000	3500
4000	6000	3500
2000	4000	2500

Step 1: Subtract minimum of every row.

2500, 4000 and 2000 are subtracted from rows 1, 2 and 3 respectively.

0	1500	1000
500	2500	0
0	2000	500

Step 2: Subtract minimum of every column.

0, 1500 and 0 are subtracted from columns 1, 2 and 3 respectively.

0	0	1000
500	1000	0
0	500	500

Step 3: Cover all zeroes with minimum number of horizontal and vertical lines.

Step 4: Since we need 3 lines to cover all zeroes, we have found the optimal assignment.

2500	4000	3500
4000	6000	3500
2000	4000	2500

So the optimal cost is $4000 + 3500 + 2000 = 9500$

An example that doesn't lead to optimal value in first attempt:

In the above example, the first check for optimality did give us solution. What if we the number covering lines is less than n.

cost matrix:

1500	4000	4500
2000	6000	3500
2000	4000	2500

Step 1: Subtract minimum of every row.

1500, 2000 and 2000 are subtracted from rows 1, 2 and 3 respectively.

0	2500	3000
0	4000	1500
0	2000	500

Step 2: Subtract minimum of every column.

0, 2000 and 500 are subtracted from columns 1, 2 and 3 respectively.

0	500	2500
0	2000	1000
0	0	0

Step 3: Cover all zeroes with minimum number of horizontal and vertical lines.

Step 4: Since we only need 2 lines to cover all zeroes, we have NOT found the optimal assignment.

Step 5: We subtract the smallest uncovered entry from all uncovered rows. Smallest entry is 500.

-500	0	2000
-500	1500	500
0	0	0

Then we add the smallest entry to all covered columns, we get

0	0	2000
0	1500	500
500	0	0

Now we return to Step 3:. Here we cover again using lines. and go to Step 4:. Since we need 3 lines to cover, we found the optimal solution.

1500	4000	4500
2000	6000	3500
2000	4000	2500

So the optimal cost is $4000 + 2000 + 2500 = 8500$

In the next post, we will be discussing implementation of the above algorithm. The implementation requires more steps as we need to find minimum number of lines to cover all 0's using a program.

References:

http://www.math.harvard.edu/archive/20_spring_05/handouts/assignment_overheads.pdf

<https://www.youtube.com/watch?v=dQDZNHwuuOY>

This article is contributed by **Yash Varyani**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

<https://www.geeksforgeeks.org/hungarian-algorithm-assignment-problem-set-1-introduction/>

Chapter 176

Hypercube Graph

Hypercube Graph - GeeksforGeeks

You are given input as order of graph n (highest number of edges connected to a node), you have to find number of vertices in a Hypercube graph of order n.

Examples:

Input : n = 3
Output : 8

Input : n = 2
Output : 4

In hypercube graph $Q(n)$, n represents the degree of the graph. Hypercube graph represents the maximum number of edges that can be connected to a graph to make it a n degree graph, every vertex has same degree n and in that representation only a fixed number of edges and vertices are added as shown in the figure below:



Q(0) with vertex 1

(Single vertex)



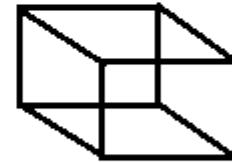
Q(1) with vertex 2

(Complete graph)



Q(2) with vertex 4

(Cycle of length 4)



Q(3) with vertex 8

(Cubical graph)

All hypercube graphs are Hamiltonian, **hypercube graph of order n has (2^n) vertices**, for input n as order of graph we have to find the corresponding power of 2.

```
// program to find vertices in a hypercube
// graph of order n
#include <iostream>
using namespace std;

// function to find power of 2
int power(int n)
{
    if (n == 1)
        return 2;
    return 2 * power(n - 1);
}
```

```
// driver program
int main()
{
    // n is the order of the graph
    int n = 4;
    cout << power(n);
    return 0;
}
```

Output:

16

Source

<https://www.geeksforgeeks.org/hypercube-graph/>

Chapter 177

Implementation of Graph in JavaScript

Implementation of Graph in JavaScript - GeeksforGeeks

In this article we would be implementing the [Graph data structure](#) in JavaScript. Graph is a non-linear data structure. A graph **G** contains a set of vertices **V** and set of Edges **E**. Graph has lots of application in computer science.

Graph is basically divided into two broad categories :

- **Directed Graph (Di-graph)** – Where edges have direction.
- **Undirected Graph** – Where edges do not represent any directed

There are various ways to represent a Graph :-

- **Adjacency Matrix**
- **Adjacency List**

There are several other ways like incidence matrix, etc. but these two are most commonly used. Refer to [Graph and its representations](#) for the explanation of Adjacency matrix and list.

In this article, we would be using **Adjacency List** to represent a graph because in most cases it has certain advantage over the other representation.

Now Lets see an example of Graph class-

```
// create a graph class
class Graph {
    // defining vertex array and
    // adjacent list
    constructor(noOfVertices)
```

```

{
    this.noOfVertices = noOfVertices;
    this.AdjList = new Map();
}

// functions to be implemented

// addVertex(v)
// addEdge(v, w)
// printGraph()

// bfs(v)
// dfs(v)
}

```

The above example shows a framework of *Graph* class. We define two private variable i.e *noOfVertices* to store the number of vertices in the graph and *AdjList*, which stores a adjacency list of a particular vertex. We used a **Map** Object provided by ES6 in order to implement Adjacency list. Where key of a map holds a vertex and values holds an array of adjacent node.

Now lets implement functions to perform basic operations on graph:

1. **addVertex(v)** – It adds the vertex *v* as key to *adjList* and initialize its values with an array.

```

// add vertex to the graph
addVertex(v)
{
    // initialize the adjacent list with a
    // null array
    this.AdjList.set(v, []);
}

```

2. **addEdge(src, dest)** – It adds an edge between the *src* and *dest*.

```

// add edge to the graph
addEdge(v, w)
{
    // get the list for vertex v and put the
    // vertex w denoting edge betweeen v and w
    this.AdjList.get(v).push(w);

    // Since graph is undirected,
    // add an edge from w to v also
    this.AdjList.get(w).push(v);
}

```

In order to add edge we get the adjacency list of the corresponding *src* vertex and add the *dest* to the adjacency list.

3. **printGraph()** – It prints vertices and its adjacency list.

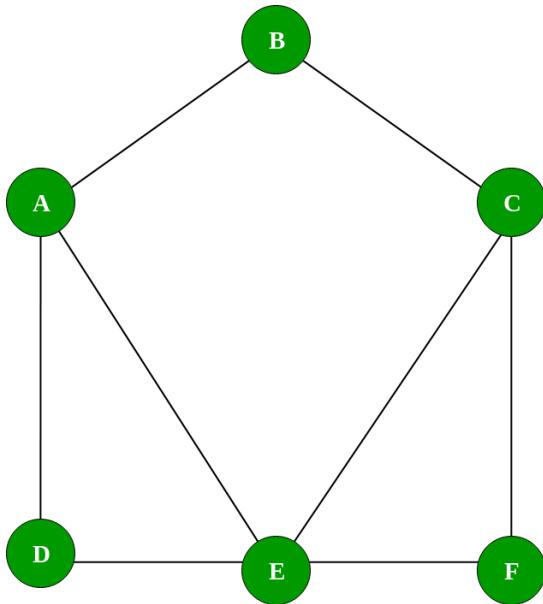
```
// Prints the vertex and adjacency list
printGraph()
{
    // get all the vertices
    var get_keys = this.AdjList.keys();

    // iterate over the vertices
    for (var i of get_keys)
    {
        // great the corresponding adjacency list
        // for the vertex
        var get_values = this.AdjList.get(i);
        var conc = "";

        // iterate over the adjacency list
        // concatenate the values into a string
        for (var j of get_values)
            conc += j + " ";

        // print the vertex and its adjacency list
        console.log(i + " -> " + conc);
    }
}
```

Lets see an example of a graph



Now we will use the graph class to implement the graph shown above:

```

// Using the above implemented graph class
var g = new Graph(6);
var vertices = [ 'A', 'B', 'C', 'D', 'E', 'F' ];

// adding vertices
for (var i = 0; i < vertices.length; i++) {
    g.addVertex(vertices[i]);
}

// adding edges
g.addEdge('A', 'B');
g.addEdge('A', 'D');
g.addEdge('A', 'E');
g.addEdge('B', 'C');
g.addEdge('D', 'E');
g.addEdge('E', 'F');
g.addEdge('E', 'C');
g.addEdge('C', 'F');

// prints all vertex and
// its adjacency list
// A -> B D E
// B -> A C
// C -> B E F
// D -> A E
// E -> A D F C
// F -> E C
  
```

```
g.printGraph();
```

Graph Traversal

We will implement the most common graph traversal algorithm:

- Breadth First Traversal for a Graph
- Depth First Traversal for a Graph

Implementation of BFS and DFS:

1. **bfs(startingNode)** – It performs Breadth First Search from the given *startingNode*

```
// function to performs BFS
bfs(startingNode)
{
    // create a visited array
    var visited = [];
    for (var i = 0; i < this.noOfVertices; i++)
        visited[i] = false;

    // Create an object for queue
    var q = new Queue();

    // add the starting node to the queue
    visited[startingNode] = true;
    q.enqueue(startingNode);

    // loop until queue is empty
    while (!q.isEmpty()) {
        // get the element from the queue
        var getQueueElement = q.dequeue();

        // passing the current vertex to callback function
        console.log(getQueueElement);

        // get the adjacent list for current vertex
        var get_List = this.AdjList.get(getQueueElement);

        // loop through the list and add the element to the
        // queue if it is not processed yet
        for (var i in get_List) {
            var neigh = get_List[i];

            if (!visited[neigh]) {
                visited[neigh] = true;
                q.enqueue(neigh);
            }
        }
    }
}
```

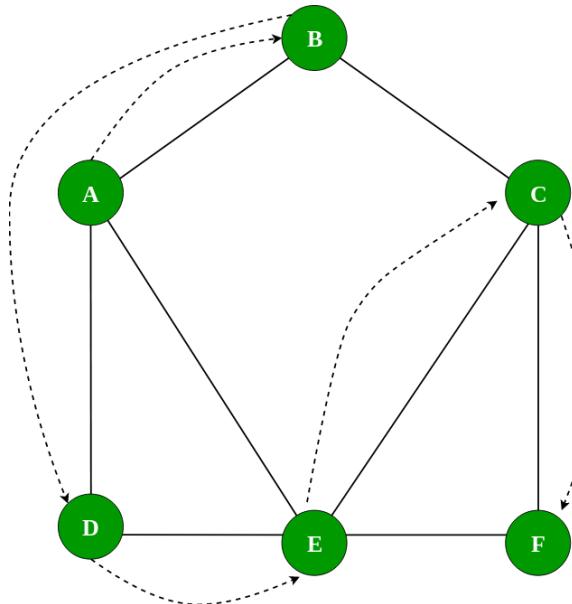
```
        q.enqueue(neigh);  
    }  
}  
}
```

In the above method we have implemented the BFS algorithm. A [Queue](#) is used to keep the unvisited nodes

Lets use the above method and traverse along the graph

```
// prints  
// BFS  
// A B D E C F  
console.log("BFS");  
g.bfs('A');
```

The Diagram below shows the BFS on the example graph:



2. `dfs(startingNode)` – It performs the Depth first traversal on a graph.

```
// Main DFS method
dfs(startingNode)
{
    var visited = [];
    for (var i = 0; i < this.noOfVertices; i++)
        visited[i] = false;

    this.DFSUtil(startingNode, visited);
```

```

}

// Recursive function which process and explore
// all the adjacent vertex of the vertex with which it is called
DFSUtil(vert, visited)
{
    visited[vert] = true;
    console.log(vert);

    var get_neighbours = this.AdjList.get(vert);

    for (var i in get_neighbours) {
        var get_elem = get_neighbours[i];
        if (!visited[get_elem])
            this.DFSUtil(get_elem, visited);
    }
}

```

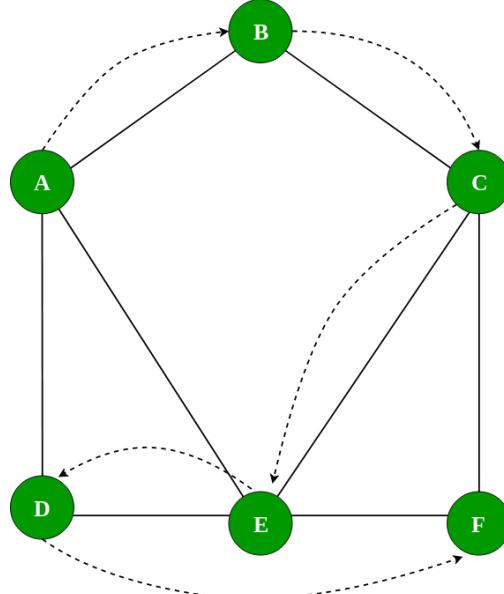
In the above example `dfs(startingNode)` is used to initialize a visited array and `DFSUtil(vert, visited)` contains the implementation of DFS algorithm
Lets use the above method to traverse along the graph

```

// prints
// DFS
// A B C E D F
console.log("DFS");
g.dfs('A');

```

The Diagram below shows the DFS on the example graph



Source

<https://www.geeksforgeeks.org/implementation-graph-javascript/>

Chapter 178

Iterative Deepening Search(IDS) or Iterative Deepening Depth First Search(IDDFS)

Iterative Deepening Search(IDS) or Iterative Deepening Depth First Search(IDDFS) - Geeks-forGeeks

There are two common ways to traverse a graph, [BFS](#) and [DFS](#). Considering a Tree (or Graph) of huge height and width, both BFS and DFS are not very efficient due to following reasons.

1. **DFS** first traverses nodes going through one adjacent of root, then next adjacent. The problem with this approach is, if there is a node close to root, but not in first few subtrees explored by DFS, then DFS reaches that node very late. Also, DFS may not find shortest path to a node (in terms of number of edges).
2. **BFS** goes level by level, but requires more space. The space required by DFS is $O(d)$ where d is depth of tree, but space required by BFS is $O(n)$ where n is number of nodes in tree (Why? Note that the last level of tree can have around $n/2$ nodes and second last level $n/4$ nodes and in BFS we need to have every level one by one in queue).

IDDFS combines depth-first search's space-efficiency and breadth-first search's fast search (for nodes closer to root).

How does IDDFS work?

IDDFS calls DFS for different depths starting from an initial value. In every call, DFS is restricted from going beyond given depth. So basically we do DFS in a BFS fashion.

Algorithm:

```
// Returns true if target is reachable from
// src within max_depth
bool IDDFS(src, target, max_depth)
    for limit from 0 to max_depth
        if DLS(src, target, limit) == true
            return true
    return false

bool DLS(src, target, limit)
    if (src == target)
        return true;

    // If reached the maximum depth,
    // stop recursing.
    if (limit <= 0)
        return false;

    foreach adjacent i of src
        if DLS(i, target, limit?1)
            return true

    return false
```

An important thing to note is, we visit top level nodes multiple times. The last (or max depth) level is visited once, second last level is visited twice, and so on. It may seem expensive, but it turns out to be not so costly, since in a tree most of the nodes are in the bottom level. So it does not matter much if the upper levels are visited multiple times.

Below is implementation of above algorithm

C/C++

```
// C++ program to search if a target node is reachable from
// a source with given max depth.
#include<bits/stdc++.h>
using namespace std;

// Graph class represents a directed graph using adjacency
// list representation.
class Graph
{
    int V;      // No. of vertices

    // Pointer to an array containing
    // adjacency lists
    list<int> *adj;

    // A function used by IDDFS
```

```
bool DLS(int v, int target, int limit);

public:
    Graph(int V);    // Constructor
    void addEdge(int v, int w);

    // IDDFS traversal of the vertices reachable from v
    bool IDDFS(int v, int target, int max_depth);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

// A function to perform a Depth-Limited search
// from given source 'src'
bool Graph::DLS(int src, int target, int limit)
{
    if (src == target)
        return true;

    // If reached the maximum depth, stop recursing.
    if (limit <= 0)
        return false;

    // Recur for all the vertices adjacent to source vertex
    for (auto i = adj[src].begin(); i != adj[src].end(); ++i)
        if (DLS(*i, target, limit-1) == true)
            return true;

    return false;
}

// IDDFS to search if target is reachable from v.
// It uses recursive DFSUtil().
bool Graph::IDDFS(int src, int target, int max_depth)
{
    // Repeatedly depth-limit search till the
    // maximum depth.
    for (int i = 0; i <= max_depth; i++)
        if (DLS(src, target, i) == true)
```

```
        return true;

    return false;
}

// Driver code
int main()
{
    // Let us create a Directed graph with 7 nodes
    Graph g(7);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(1, 4);
    g.addEdge(2, 5);
    g.addEdge(2, 6);

    int target = 6, maxDepth = 3, src = 0;
    if (g.IDDFS(src, target, maxDepth) == true)
        cout << "Target is reachable from source "
              "within max depth";
    else
        cout << "Target is NOT reachable from source "
              "within max depth";
    return 0;
}
```

Python

```
# Python program to print DFS traversal from a given
# given graph
from collections import defaultdict

# This class represents a directed graph using adjacency
# list representation
class Graph:

    def __init__(self,vertices):

        # No. of vertices
        self.V = vertices

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)
```

```
# A function to perform a Depth-Limited search
# from given source 'src'
def DLS(self,src,target,maxDepth):

    if src == target : return True

    # If reached the maximum depth, stop recursing.
    if maxDepth <= 0 : return False

    # Recur for all the vertices adjacent to this vertex
    for i in self.graph[src]:
        if(self.DLS(i,target,maxDepth-1)):
            return True
    return False

# IDDFS to search if target is reachable from v.
# It uses recursive DLS()
def IDDFS(self,src, target, maxDepth):

    # Repeatedly depth-limit search till the
    # maximum depth
    for i in range(maxDepth):
        if (self.DLS(src, target, i)):
            return True
    return False

# Create a graph given in the above diagram
g = Graph (7);
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 3)
g.addEdge(1, 4)
g.addEdge(2, 5)
g.addEdge(2, 6)

target = 6; maxDepth = 3; src = 0

if g.IDDFS(src, target, maxDepth) == True:
    print ("Target is reachable from source " +
          "within max depth")
else :
    print ("Target is NOT reachable from source " +
          "within max depth")

# This code is contributed by Neelam Pandey
```

Output :

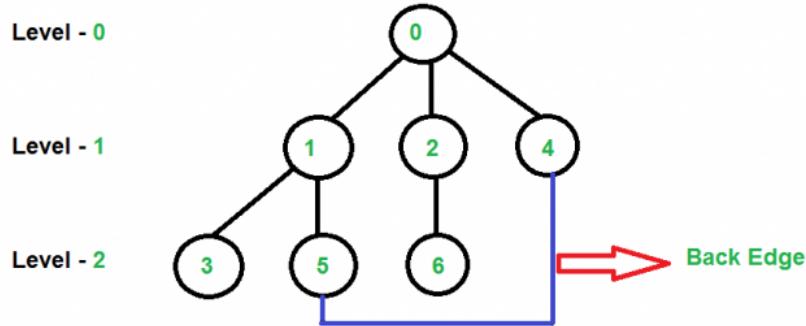
Target is reachable from source within max depth

Illustration:

There can be two cases-

a) *When the graph has no cycle:* This case is simple. We can DFS multiple times with different height limits.

b) *When the graph has cycles.* This is interesting as there is no visited flag in IDDFS.



Although, at first sight, it may seem that since there are only 3 levels, so we might think that Iterative Deepening Depth First Search of level 3, 4, 5,...and so on will remain same. But, this is not the case. You can see that there is a cycle in the above graph, hence IDDFS will change for level-3,4,5..and so on.

Depth	Iterative Deepening Depth First Search
0	0
1	0 1 2 4
2	0 1 3 5 2 6 4 5
3	0 1 3 5 4 2 6 4 5 1

The diagram shows the same search tree as the previous figure, but with nodes numbered 0 through 6. The nodes are grouped by level: Level 0 (root 0), Level 1 (children of 0: 1, 2, 4), and Level 2 (children of 1: 3, 5; children of 2: 6).

The explanation of the above pattern is left to the readers.

Time Complexity: Suppose we have a tree having branching factor 'b' (number of children of each node), and its depth 'd', i.e., there are b^d nodes.

In an iterative deepening search, the nodes on the bottom level are expanded once, those on the next to bottom level are expanded twice, and so on, up to the root of the search tree,

which is expanded $d+1$ times. So the total number of expansions in an iterative deepening search is-

$$(d)b + (d-1)b^2 + \dots + 3b^{d-2} + 2b^{d-1} + b^d$$

That is,

$\text{Summation}[(d + 1 - i) b^i]$, from $i = 0$ to $i = d$
Which is same as $O(b^d)$

After evaluating the above expression, we find that asymptotically IDDFS takes the same time as that of DFS and BFS, but it is indeed slower than both of them as it has a higher constant factor in its time complexity expression.

IDDFS is best suited for a complete infinite tree

A comparison table between DFS, BFS and IDDFS

	Time Complexity	Space Complexity	When to Use ?
DFS	$O(b^d)$	$O(d)$	=> Don't care if the answer is closest to the starting vertex/root. => When graph/tree is not very big/infinite.
BFS	$O(b^d)$	$O(b^d)$	=> When space is not an issue => When we do care/want the closest answer to the root.
IDDFS	$O(b^d)$	$O(bd)$	=> You want a BFS, you don't have enough memory, and somewhat slower performance is accepted. In short, you want a BFS + DFS.

References:

https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search

Source

<https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/>

Chapter 179

Java Program for Dijkstra's Algorithm with Path Printing

Java Program for Dijkstra's Algorithm with Path Printing - GeeksforGeeks

```
import java.util.Scanner; //Scanner Function to take in the Input Values

public class Dijkstra
{
    static Scanner scan; // scan is a Scanner Object

    public static void main(String[] args)
    {
        int[] preD = new int[5];
        int min = 999, nextNode = 0; // min holds the minimum value, nextNode holds the value for
        scan = new Scanner(System.in);
        int[] distance = new int[5]; // the distance matrix
        int[][] matrix = new int[5][5]; // the actual matrix
        int[] visited = new int[5]; // the visited array

        System.out.println("Enter the cost matrix");

        for (int i = 0; i < distance.length; i++)
        {
            visited[i] = 0; //initialize visited array to zeros
            preD[i] = 0;

            for (int j = 0; j < distance.length; j++)
            {
                matrix[i][j] = scan.nextInt(); //fill the matrix
                if (matrix[i][j]==0)
                    matrix[i][j] = 999; // make the zeros as 999
```

```
        }
    }

    distance = matrix[0]; //initialize the distance array
    visited[0] = 1; //set the source node as visited
    distance[0] = 0; //set the distance from source to source to zero which is the starting point

    for (int counter = 0; counter < 5; counter++)
    {
        min = 999;
        for (int i = 0; i < 5; i++)
        {
            if (min > distance[i] && visited[i]!=1)
            {
                min = distance[i];
                nextNode = i;
            }
        }

        visited[nextNode] = 1;
        for (int i = 0; i < 5; i++)
        {
            if (visited[i]!=1)
            {
                if (min+matrix[nextNode][i] < distance[i])
                {
                    distance[i] = min+matrix[nextNode][i];
                    preD[i] = nextNode;
                }
            }
        }
    }

    for(int i = 0; i < 5; i++)
        System.out.print("|" + distance[i]);

    System.out.println("|");

    int j;
    for (int i = 0; i < 5; i++)
    {
        if (i!=0)
        {

            System.out.print("Path = " + i);
        }
    }
}
```

```
j = i;
do
{
    j = preD[j];
    System.out.print(" <- " + j);
}
while(j != 0);
System.out.println();
}
}
```

This program is contributed by by Raj Miglani. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

<https://www.geeksforgeeks.org/java-program-for-dijkstras-algorithm-with-path-printing/>

Chapter 180

k'th heaviest adjacent node in a graph where each vertex has weight

k'th heaviest adjacent node in a graph where each vertex has weight - GeeksforGeeks

Given a positive number **k** and an undirected graph of **N** nodes, numbered from 0 to N-1, each having a weight associated with it. Note that this is different from a normal weighted graph where every edge has a weight.

For each node, if we sort the nodes (according to their weights), which are directly connected to it, in decreasing order, then what will be the number of the node at the **kth** position. Print kth node number(not weight) for each node and if it does not exist, print -1.

Examples:

```
Input : N = 3, k = 2, wt[] = { 2, 4, 3 }.
edge 1: 0 2
edge 2: 0 1
edge 3: 1 2
```

```
Output : 2 0 0
Graph:
      0 (weight 2)
      / \
      /   \
     1     2
(weight 4) (weight 3)
For node 0, sorted (decreasing order) nodes
according to their weights are node 1(weight 4),
node 2(weight 3). The node at 2nd position for
```

node 0 is node 2.
For node 1, sorted (decreasing order) nodes according to their weight are node 2(weight 3), node 0(weight 2). The node at 2nd position for node 1 is node 0.
For node 2, sorted (decreasing order) nodes according to their weight are node 1(weight 4), node 0(weight 2). The node at 2nd position for node 2 is node 0.

The idea is to sort Adjacency List of each node on the basis of adjacent node weights. First, create Adjacency List for all the nodes. Now for each node, all the nodes which are directly connected to it stored in a list. In adjacency list, store the nodes along with their weights.

Now, for each node sort the weights of all nodes which are directly connected to it in reverse order, and then print the node number which is at kth position in the list of each node.

Below is C++ implementation of this approach:

```
// C++ program to find Kth node weight after sorting of nodes directly connected to a node.
#include<bits/stdc++.h>
using namespace std;

// Print Kth node number for each node after sorting
// connected node according to their weight.
void printkthnode(vector< pair<int, int> > adj[],
                  int wt[], int n, int k)
{
    // Sort Adjacency List of all node on the basis
    // of its weight.
    for (int i = 0; i < n; i++)
        sort(adj[i].begin(), adj[i].end());

    // Printing Kth node for each node.
    for (int i = 0; i < n; i++)
    {
        if (adj[i].size() >= k)
            cout << adj[i][adj[i].size() - k].second;
        else
            cout << "-1";
    }
}

// Driven Program
int main()
{
    int n = 3, k = 2;
```

```
int wt[] = { 2, 4, 3 };

// Making adjacency list, storing the nodes
// along with their weight.
vector< pair<int, int> > adj[n+1];

adj[0].push_back(make_pair(wt[2], 2));
adj[2].push_back(make_pair(wt[0], 0));

adj[0].push_back(make_pair(wt[1], 1));
adj[1].push_back(make_pair(wt[0], 0));

adj[1].push_back(make_pair(wt[2], 2));
adj[2].push_back(make_pair(wt[1], 1));

printkthnode(adj, wt, n, k);
return 0;
}
```

Output:

2 0 0

Source

<https://www.geeksforgeeks.org/kth-adjacent-node-graph-vertex-weight/>

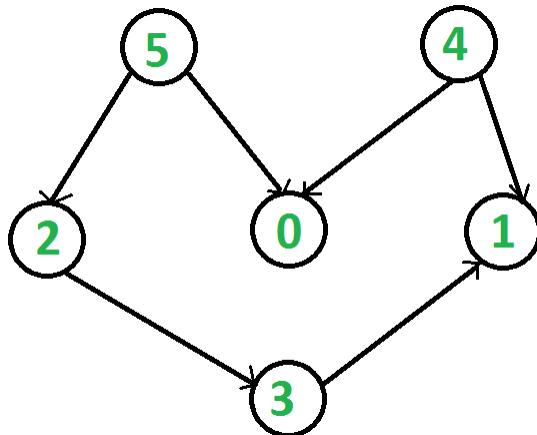
Chapter 181

Kahn's algorithm for Topological Sorting

Kahn's algorithm for Topological Sorting - GeeksforGeeks

Topological sorting for **Directed Acyclic Graph** (DAG) is a linear ordering of vertices such that for every directed edge uv , vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, a topological sorting of the following graph is “5 4 2 3 1 0?”. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 0 3 1”. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no in-coming edges).



A [DFS based solution to find a topological sort](#) has already been discussed.

In this article we will see another way to find the linear ordering of vertices in a directed acyclic graph (DAG). The approach is based on the below fact :

A DAG G has at least one vertex with in-degree 0 and one vertex with out-degree 0.

Proof: There's a simple proof to the above fact is that a DAG does not contain a cycle which means that all paths will be of finite length. Now let S be the longest path from u(source) to v(destination). Since S is the longest path there can be no incoming edge to u and no outgoing edge from v, if this situation had occurred then S would not have been the longest path

$$\Rightarrow \text{indegree}(u) = 0 \text{ and } \text{outdegree}(v) = 0$$

Algorithm:

Steps involved in finding the topological ordering of a DAG:

Step-1: Compute in-degree (number of incoming edges) for each of the vertex present in the DAG and initialize the count of visited nodes as 0.

Step-2: Pick all the vertices with in-degree as 0 and add them into a queue (Enqueue operation)

Step-3: Remove a vertex from the queue (Dequeue operation) and then.

1. Increment count of visited nodes by 1.
2. Decrease in-degree by 1 for all its neighboring nodes.
3. If in-degree of a neighboring nodes is reduced to zero, then add it to the queue.

Step 5: Repeat Step 3 until the queue is empty.

Step 5: If count of visited nodes is **not** equal to the number of nodes in the graph then the topological sort is not possible for the given graph.

How to find in-degree of each node?

There are 2 ways to calculate in-degree of every vertex:

Take an in-degree array which will keep track of

1) Traverse the array of edges and simply increase the counter of the destination node by 1.

```
for each node in Nodes
    indegree[node] = 0;
for each edge(src,dest) in Edges
    indegree[dest]++
```

Time Complexity: O(V+E)

2) Traverse the list for every node and then increment the in-degree of all the nodes connected to it by 1.

```
for each node in Nodes
    If (list[node].size()!=0) then
        for each dest in list
            indegree[dest]++;
```

Time Complexity: The outer for loop will be executed V number of times and the inner for loop will be executed E number of times, Thus overall time complexity is O(V+E).

The overall time complexity of the algorithm is O(V+E)

Below is C++ implementation of above algorithm. The implementation uses method 2 discussed above for finding indegrees.

C++

```
// A C++ program to print topological sorting of a graph
// using indegrees.
#include<bits/stdc++.h>
using namespace std;

// Class to represent a graph
class Graph
{
    int V;      // No. of vertices'

    // Pointer to an array containing adjacency listsList
    list<int> *adj;

public:
    Graph(int V);    // Constructor

    // function to add an edge to graph
    void addEdge(int u, int v);

    // prints a Topological Sort of the complete graph
    void topologicalSort();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int u, int v)
{
    adj[u].push_back(v);
}

// The function to do Topological Sort.
void Graph::topologicalSort()
{
    // Create a vector to store indegrees of all
    // vertices. Initialize all indegrees as 0.
    vector<int> in_degree(V, 0);
```

```

// Traverse adjacency lists to fill indegrees of
// vertices. This step takes O(V+E) time
for (int u=0; u<V; u++)
{
    list<int>::iterator itr;
    for (itr = adj[u].begin(); itr != adj[u].end(); itr++)
        in_degree[*itr]++;
}

// Create an queue and enqueue all vertices with
// indegree 0
queue<int> q;
for (int i = 0; i < V; i++)
    if (in_degree[i] == 0)
        q.push(i);

// Initialize count of visited vertices
int cnt = 0;

// Create a vector to store result (A topological
// ordering of the vertices)
vector <int> top_order;

// One by one dequeue vertices from queue and enqueue
// adjacents if indegree of adjacent becomes 0
while (!q.empty())
{
    // Extract front of queue (or perform dequeue)
    // and add it to topological order
    int u = q.front();
    q.pop();
    top_order.push_back(u);

    // Iterate through all its neighbouring nodes
    // of dequeued node u and decrease their in-degree
    // by 1
    list<int>::iterator itr;
    for (itr = adj[u].begin(); itr != adj[u].end(); itr++)

        // If in-degree becomes zero, add it to queue
        if (--in_degree[*itr] == 0)
            q.push(*itr);

    cnt++;
}

// Check if there was a cycle

```

```
if (cnt != V)
{
    cout << "There exists a cycle in the graph\n";
    return;
}

// Print topological order
for (int i=0; i<top_order.size(); i++)
    cout << top_order[i] << " ";
cout << endl;
}

// Driver program to test above functions
int main()
{
    // Create a graph given in the above diagram
    Graph g(6);
    g.addEdge(5, 2);
    g.addEdge(5, 0);
    g.addEdge(4, 0);
    g.addEdge(4, 1);
    g.addEdge(2, 3);
    g.addEdge(3, 1);

    cout << "Following is a Topological Sort of\n";
    g.topologicalSort();

    return 0;
}
```

Java

```
// A Java program to print topological sorting of a graph
// using indegrees
import java.util.*;

//Class to represent a graph
class Graph
{
    int V;// No. of vertices

    //An Array of List which contains
    //references to the Adjacency List of
    //each vertex
    List <Integer> adj[];
    public Graph(int V)// Constructor
    {
        this.V = V;
```

```

adj = new ArrayList[V];
for(int i = 0; i < V; i++)
    adj[i]=new ArrayList<Integer>();
}

// function to add an edge to graph
public void addEdge(int u,int v)
{
    adj[u].add(v);
}
// prints a Topological Sort of the complete graph
public void topologicalSort()
{
    // Create a array to store indegrees of all
    // vertices. Initialize all indegrees as 0.
    int indegree[] = new int[V];

    // Traverse adjacency lists to fill indegrees of
    // vertices. This step takes O(V+E) time
    for(int i = 0; i < V; i++)
    {
        ArrayList<Integer> temp = (ArrayList<Integer>) adj[i];
        for(int node : temp)
        {
            indegree[node]++;
        }
    }

    // Create a queue and enqueue all vertices with
    // indegree 0
    Queue<Integer> q = new LinkedList<Integer>();
    for(int i = 0;i < V; i++)
    {
        if(indegree[i]==0)
            q.add(i);
    }

    // Initialize count of visited vertices
    int cnt = 0;

    // Create a vector to store result (A topological
    // ordering of the vertices)
    Vector <Integer> topOrder=new Vector<Integer>();
    while(!q.isEmpty())
    {
        // Extract front of queue (or perform dequeue)
        // and add it to topological order
        int u=q.poll();
        ...
    }
}

```

```
topOrder.add(u);

// Iterate through all its neighbouring nodes
// of dequeued node u and decrease their in-degree
// by 1
for(int node : adj[u])
{
    // If in-degree becomes zero, add it to queue
    if(--indegree[node] == 0)
        q.add(node);
}
cnt++;

}

// Check if there was a cycle
if(cnt != V)
{
    System.out.println("There exists a cycle in the graph");
    return ;
}

// Print topological order
for(int i : topOrder)
{
    System.out.print(i+" ");
}
}

// Driver program to test above functions
class Main
{
    public static void main(String args[])
    {
        // Create a graph given in the above diagram
        Graph g=new Graph(6);
        g.addEdge(5, 2);
        g.addEdge(5, 0);
        g.addEdge(4, 0);
        g.addEdge(4, 1);
        g.addEdge(2, 3);
        g.addEdge(3, 1);
        System.out.println("Following is a Topological Sort");
        g.topologicalSort();

    }
}
```

Python

```
# A Python program to print topological sorting of a graph
# using indegrees
from collections import defaultdict

#Class to represent a graph
class Graph:
    def __init__(self,vertices):
        self.graph = defaultdict(list) #dictionary containing adjacency List
        self.V = vertices #No. of vertices

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # The function to do Topological Sort.
    def topologicalSort(self):

        # Create a vector to store indegrees of all
        # vertices. Initialize all indegrees as 0.
        in_degree = [0]*(self.V)

        # Traverse adjacency lists to fill indegrees of
        # vertices. This step takes O(V+E) time
        for i in self.graph:
            for j in self.graph[i]:
                in_degree[j] += 1

        # Create an queue and enqueue all vertices with
        # indegree 0
        queue = []
        for i in range(self.V):
            if in_degree[i] == 0:
                queue.append(i)

        #Initialize count of visited vertices
        cnt = 0

        # Create a vector to store result (A topological
        # ordering of the vertices)
        top_order = []

        # One by one dequeue vertices from queue and enqueue
        # adjacents if indegree of adjacent becomes 0
        while queue:

            # Extract front of queue (or perform dequeue)
            # and add it to topological order
            node = queue.pop(0)
            top_order.append(node)

            for i in self.graph[node]:
                in_degree[i] -= 1
                if in_degree[i] == 0:
                    queue.append(i)

        # If there was a cycle, topological sort won't
        # happen and will return an empty list
        if cnt != self.V:
            print("There exists a cycle in the graph")
        else:
            print("Topological Sort")
            print(top_order)
```

```
u = queue.pop(0)
top_order.append(u)

# Iterate through all neighbouring nodes
# of dequeued node u and decrease their in-degree
# by 1
for i in self.graph[u]:
    in_degree[i] -= 1
    # If in-degree becomes zero, add it to queue
    if in_degree[i] == 0:
        queue.append(i)

cnt += 1

# Check if there was a cycle
if cnt != self.V:
    print "There exists a cycle in the graph"
else :
    #Print topological order
    print top_order


g= Graph(6)
g.addEdge(5, 2);
g.addEdge(5, 0);
g.addEdge(4, 0);
g.addEdge(4, 1);
g.addEdge(2, 3);
g.addEdge(3, 1);

print "Following is a Topological Sort of the given graph"
g.topologicalSort()

# This code is contributed by Neelam Yadav
```

Output :

```
Following is a Topological Sort
4 5 2 0 3 1
```

This article is contributed by **Chirag Agarwal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

<https://www.geeksforgeeks.org/topological-sorting-indegree-based-solution/>

Chapter 182

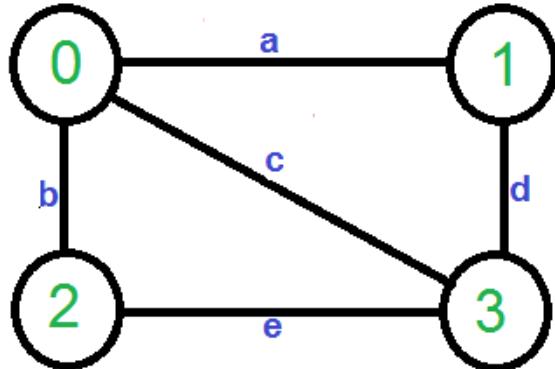
Karger's algorithm for Minimum Cut Set 1 (Introduction and Implementation)

Karger's algorithm for Minimum Cut Set 1 (Introduction and Implementation) - Geeks-forGeeks

Given an undirected and unweighted graph, find the smallest cut (smallest number of edges that disconnects the graph into two components).

The input graph may have parallel edges.

For example consider the following example, the smallest cut has 2 edges.



Min-Cut for above graph is either {a, d} OR {b, e}

A Simple Solution use [Max-Flow based s-t cut algorithm](#) to find minimum cut. Consider every pair of vertices as source 's' and sink 't', and call minimum s-t cut algorithm to find

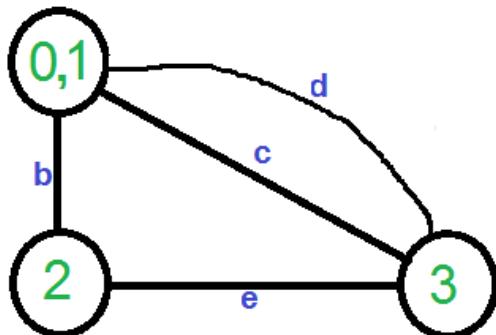
the s-t cut. Return minimum of all s-t cuts. Best possible time complexity of this algorithm is $O(V^5)$ for a graph. [How? there are total possible V^2 pairs and s-t cut algorithm for one pair takes $O(V \cdot E)$ time and $E = O(V^2)$].

Below is simple Karger's Algorithm for this purpose. Below Karger's algorithm can be implemented in $O(E) = O(V^2)$ time.

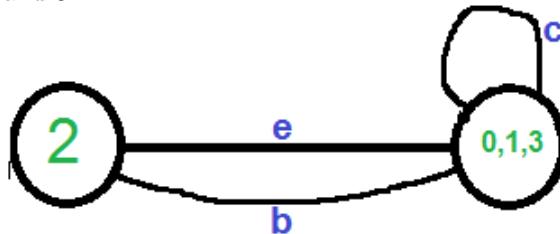
- 1) Initialize contracted graph CG as copy of original graph
- 2) While there are more than 2 vertices.
 - a) Pick a random edge (u, v) in the contracted graph.
 - b) Merge (or contract) u and v into a single vertex (update the contracted graph).
 - c) Remove self-loops
- 3) Return cut represented by two vertices.

Let us understand above algorithm through the example given.

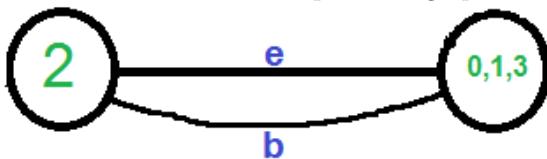
Let the first randomly picked vertex be 'a' which connects vertices 0 and 1. We remove this edge and contract the graph (combine vertices 0 and 1). We get the following graph.



Let the next randomly picked edge be 'd'. We remove this edge and combine vertices (0,1) and 3.

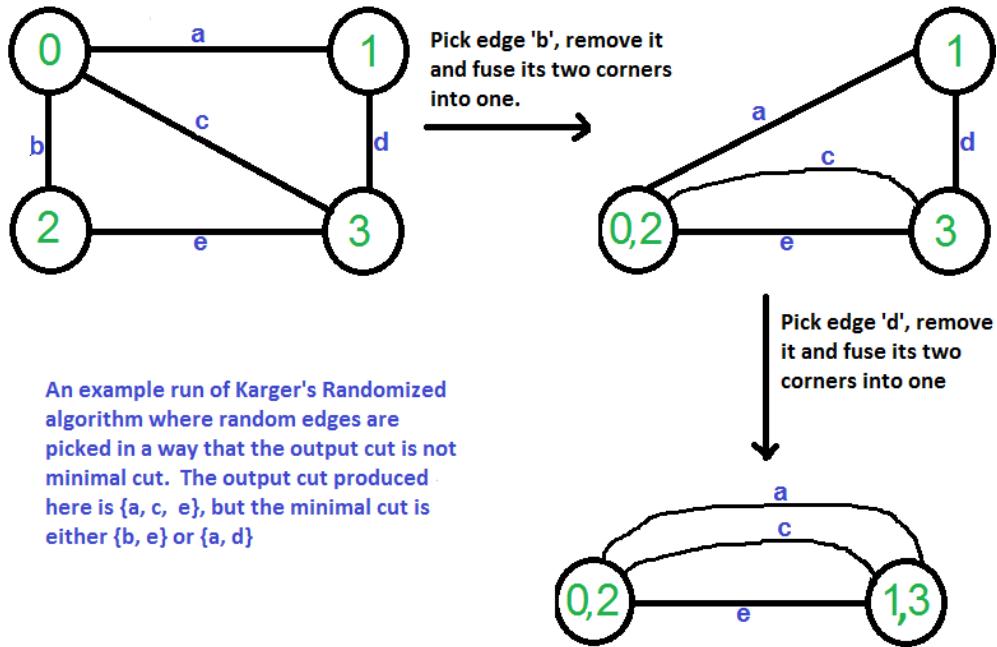


We need to remove self-loops in the graph. So we remove edge 'c'



Now graph has two vertices, so we stop. The number of edges in the resultant graph is the cut produced by Karger's algorithm.

Karger's algorithm is a Monte Carlo algorithm and cut produced by it may not be minimum. For example, the following diagram shows that a different order of picking random edges produces a min-cut of size 3.



Below is C++ implementation of above algorithm. The input graph is represented as a collection of edges and [union-find data structure](#) is used to keep track of components.

```

// Karger's algorithm to find Minimum Cut in an
// undirected, unweighted and connected graph.
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// a structure to represent a unweighted edge in graph
struct Edge
{
    int src, dest;
};

// a structure to represent a connected, undirected
// and unweighted graph as a collection of edges.
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;
}

```

```
// graph is represented as an array of edges.  
// Since the graph is undirected, the edge  
// from src to dest is also edge from dest  
// to src. Both are counted as 1 edge here.  
Edge* edge;  
};  
  
// A structure to represent a subset for union-find  
struct subset  
{  
    int parent;  
    int rank;  
};  
  
// Function prototypes for union-find (These functions are defined  
// after kargerMinCut() )  
int find(struct subset subsets[], int i);  
void Union(struct subset subsets[], int x, int y);  
  
// A very basic implementation of Karger's randomized  
// algorithm for finding the minimum cut. Please note  
// that Karger's algorithm is a Monte Carlo Randomized algo  
// and the cut returned by the algorithm may not be  
// minimum always  
int kargerMinCut(struct Graph* graph)  
{  
    // Get data of given graph  
    int V = graph->V, E = graph->E;  
    Edge *edge = graph->edge;  
  
    // Allocate memory for creating V subsets.  
    struct subset *subsets = new subset[V];  
  
    // Create V subsets with single elements  
    for (int v = 0; v < V; ++v)  
    {  
        subsets[v].parent = v;  
        subsets[v].rank = 0;  
    }  
  
    // Initially there are V vertices in  
    // contracted graph  
    int vertices = V;  
  
    // Keep contracting vertices until there are  
    // 2 vertices.  
    while (vertices > 2)
```

```
{  
    // Pick a random edge  
    int i = rand() % E;  
  
    // Find vertices (or sets) of two corners  
    // of current edge  
    int subset1 = find(subsets, edge[i].src);  
    int subset2 = find(subsets, edge[i].dest);  
  
    // If two corners belong to same subset,  
    // then no point considering this edge  
    if (subset1 == subset2)  
        continue;  
  
    // Else contract the edge (or combine the  
    // corners of edge into one vertex)  
    else  
    {  
        printf("Contracting edge %d-%d\n",  
               edge[i].src, edge[i].dest);  
        vertices--;  
        Union(subsets, subset1, subset2);  
    }  
}  
  
// Now we have two vertices (or subsets) left in  
// the contracted graph, so count the edges between  
// two components and return the count.  
int cutedges = 0;  
for (int i=0; i<E; i++)  
{  
    int subset1 = find(subsets, edge[i].src);  
    int subset2 = find(subsets, edge[i].dest);  
    if (subset1 != subset2)  
        cutedges++;  
}  
  
return cutedges;  
}  
  
// A utility function to find set of an element i  
// (uses path compression technique)  
int find(struct subset subsets[], int i)  
{  
    // find root and make root as parent of i  
    // (path compression)  
    if (subsets[i].parent != i)  
        subsets[i].parent =
```

```
        find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(struct subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high
    // rank tree (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and
    // increment its rank by one
    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}

// Driver program to test above functions
int main()
{
    /* Let us create following unweighted graph
       0-----1
       | \     |
       |   \   |
       |     \| |
       2-----3 */
    int V = 4; // Number of vertices in graph
```

```
int E = 5; // Number of edges in graph
struct Graph* graph = createGraph(V, E);

// add edge 0-1
graph->edge[0].src = 0;
graph->edge[0].dest = 1;

// add edge 0-2
graph->edge[1].src = 0;
graph->edge[1].dest = 2;

// add edge 0-3
graph->edge[2].src = 0;
graph->edge[2].dest = 3;

// add edge 1-3
graph->edge[3].src = 1;
graph->edge[3].dest = 3;

// add edge 2-3
graph->edge[4].src = 2;
graph->edge[4].dest = 3;

// Use a different seed value for every run.
srand(time(NULL));

printf("\nCut found by Karger's randomized algo is %d\n",
      kargerMinCut(graph));

return 0;
}
```

Output:

```
Contracting edge 0-2
Contracting edge 0-3

Cut found by Karger's randomized algo is 2
```

Note that the above program is based on outcome of a random function and may produce different output.

In this post, we have discussed simple Karger's algorithm and have seen that the algorithm doesn't always produce min-cut. The above algorithm produces min-cut with probability greater or equal to that $1/(n^2)$. See next post on [Analysis and Applications of Karger's Algorithm](#), applications, proof of this probability and improvements are discussed.

References:

http://en.wikipedia.org/wiki/Karger%27s_algorithm

<https://www.youtube.com/watch?v=P0l8jMDQTEQ>

<https://www.cs.princeton.edu/courses/archive/fall13/cos521/lecnotes/lec2final.pdf>

<http://web.stanford.edu/class/archive/cs/cs161/cs161.1138/lectures/11/Small11.pdf>

Source

<https://www.geeksforgeeks.org/kargers-algorithm-for-minimum-cut-set-1-introduction-and-implementation/>

Chapter 183

Karp's minimum mean (or average) weight cycle algorithm

Karp's minimum mean (or average) weight cycle algorithm - GeeksforGeeks

Given a directed and [strongly connected graph](#) with non negative edge weights. We define mean weight of a cycle as the summation of all the edge weights of the cycle divided by the no. of edges. Our task is to find the minimum mean weight among all the directed cycles of the graph.

Examples:

Input : Below Graph

Output : 1.66667

Method to find the smallest mean weight value cycle efficiently

Step 1: Choose first vertex as source.

Step 2: Compute the shortest path to all other vertices
on a path consisting of k edges $0 \leq k \leq V$
where V is number of vertices.

This is a simple dp problem which can be computed
by the recursive solution
 $dp[k][v] = \min(dp[k][v], dp[k-1][u] + \text{weight}(u,v))$
where v is the destination and the edge(u,v) should
belong to E

Step 3: For each vertex calculate $\max(dp[n][v] - dp[k][v]) / (n-k)$

where $0 \leq k \leq n-1$

Step 4: The minimum of the values calculated above is the required answer.

Please refer solution of problem 9.2 [here](#) for proof that above steps find minimum average weight.

```
// C++ program to find minimum average
// weight of a cycle in connected and
// directed graph.
#include<bits/stdc++.h>
using namespace std;

const int V = 4;

// a struct to represent edges
struct edge
{
    int from, weight;
};

// vector to store edges
vector <edge> edges[V];

void addedge(int u,int v,int w)
{
    edges[v].push_back({u, w});
}

// calculates the shortest path
void shortestpath(int dp[][][V])
{
    // initializing all distances as -1
    for (int i=0; i<=V; i++)
        for (int j=0; j<V; j++)
            dp[i][j] = -1;

    // shortest distance from first vertex
    // to in itself consisting of 0 edges
    dp[0][0] = 0;

    // filling up the dp table
    for (int i=1; i<=V; i++)
    {
        for (int j=0; j<V; j++)
        {
            for (int k=0; k<edges[j].size(); k++)
                if (dp[i][j] < 0 || dp[i][j] > edges[j][k].weight)
                    dp[i][j] = edges[j][k].weight;
                else
                    dp[i][j] = min(dp[i][j], edges[j][k].weight);
        }
    }
}
```

```

{
    if (dp[i-1][edges[j][k].from] != -1)
    {
        int curr_wt = dp[i-1][edges[j][k].from] +
                      edges[j][k].weight;
        if (dp[i][j] == -1)
            dp[i][j] = curr_wt;
        else
            dp[i][j] = min(dp[i][j], curr_wt);
    }
}
}

// Returns minimum value of average weight of a
// cycle in graph.
double minAvgWeight()
{
    int dp[V+1][V];
    shortestpath(dp);

    // array to store the avg values
    double avg[V];
    for (int i=0; i<V; i++)
        avg[i] = -1;

    // Compute average values for all vertices using
    // weights of shortest paths store in dp.
    for (int i=0; i<V; i++)
    {
        if (dp[V][i] != -1)
        {
            for (int j=0; j<V; j++)
                if (dp[j][i] != -1)
                    avg[i] = max(avg[i],
                                  ((double)dp[V][i]-dp[j][i])/(V-j));
        }
    }

    // Find minimum value in avg[]
    double result = avg[0];
    for (int i=0; i<V; i++)
        if (avg[i] != -1 && avg[i] < result)
            result = avg[i];

    return result;
}

```

```
// Driver function
int main()
{
    addedge(0, 1, 1);
    addedge(0, 2, 10);
    addedge(1, 2, 3);
    addedge(2, 3, 2);
    addedge(3, 1, 0);
    addedge(3, 0, 8);

    cout << minAvgWeight();

    return 0;
}
```

Output:

1.66667

Here the graph with no cycle will return value as -1.

Reference:

<https://courses.csail.mit.edu/6.046/fall01/handouts/ps9sol.pdf>
<https://www.hackerearth.com/practice/notes/karp-minimum-mean-weighted-cycle/>
Introduction to Algorithms Third Edition page 681 by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein

Source

<https://www.geeksforgeeks.org/karps-minimum-mean-average-weight-cycle-algorithm/>

Chapter 184

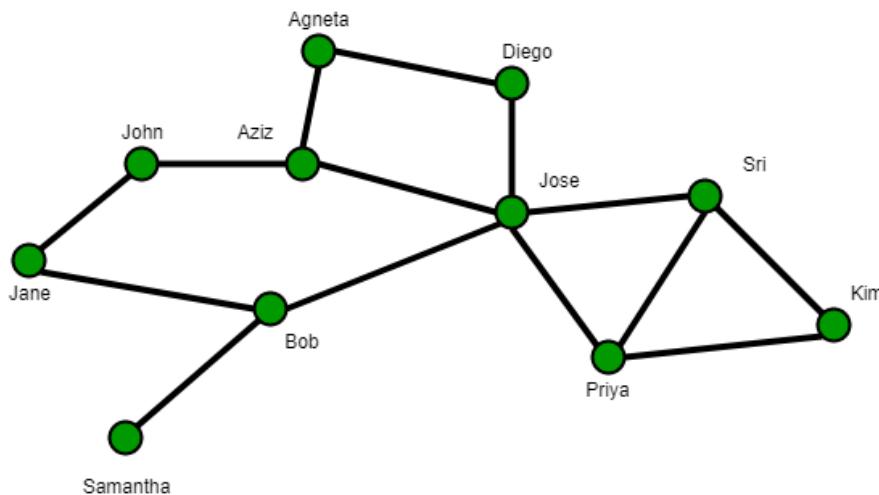
Katz Centrality (Centrality Measure)

Katz Centrality (Centrality Measure) - GeeksforGeeks

In graph theory, the Katz centrality of a node is a measure of centrality in a network. It was introduced by Leo Katz in 1953 and is used to measure the relative degree of influence of an actor (or node) within a social network. Unlike typical centrality measures which consider only the shortest path (the geodesic) between a pair of actors, Katz centrality measures influence by taking into account the total number of walks between a pair of actors.

It is similar to Google's PageRank and to the eigenvector centrality.

Measuring Katz centrality



A simple social network: the nodes represent people or actors and the edges between nodes represent some relationship between actors

Katz centrality computes the relative influence of a node within a network by measuring

the number of the immediate neighbors (first degree nodes) and also all other nodes in the network that connect to the node under consideration through these immediate neighbors. Connections made with distant neighbors are, however, penalized by an attenuation factor α . Each path or connection between a pair of nodes is assigned a weight determined by α and the distance between nodes as α^k .

For example, in the figure on the right, assume that John's centrality is being measured and that $\alpha = 0.5$. The weight assigned to each link that connects John with his

immediate neighbors Jane and Bob will be $\alpha^1 = 0.5$. Since Jose connects to John indirectly through Bob, the weight assigned to this connection (composed of two links)

will be $\alpha^2 = 0.25$. Similarly, the weight assigned to the connection between

Agneta and John through Aziz and Jane will be $\alpha^3 = 0.125$ and the weight assigned to the connection between Agneta and John through Diego, Jose and Bob will be $\alpha^4 = 0.0625$.

Mathematical formulation

Let A be the adjacency matrix of a network under consideration. Elements a_{ij} of A are variables that take a value 1 if a node i is connected to node j and 0 otherwise. The powers of A indicate the presence (or absence) of links between two nodes through intermediaries.

For instance, in matrix A^2 , if element $a_{2,12} = 1$, it indicates that node 2 and node 12 are connected through some first and second degree neighbors of node 2. If \vec{C}_{Katz} denotes Katz centrality of a node i , then mathematically:

$$\vec{C}_{\text{Katz}} = \frac{\alpha}{1-\alpha} \vec{A}^T \vec{C}_{\text{Katz}}$$

Note that the above definition uses the fact that the element at location (i, j) of the adjacency matrix A raised to the power k (i.e. a_{ij}^k) reflects the total number of k degree connections between nodes i and j . The value of the attenuation factor α has to be chosen such that it is smaller than the reciprocal of the absolute value of the largest eigenvalue of the adjacency matrix A . In this case the following expression can be used to calculate Katz centrality:

$$\vec{C}_{\text{Katz}} = ((I - \alpha A^T)^{-1} - I) \vec{I}$$

Here I is the identity matrix, \vec{I} is an identity vector of size n (n is the number of nodes)

consisting of ones. \mathbf{A}^T denotes the transposed matrix of \mathbf{A} and $(\mathbf{I} - \alpha \mathbf{A}^T)^{-1}$ denotes matrix inversion of the term $(\mathbf{I} - \alpha \mathbf{A}^T)$.

Following is the code for the calculation of the Katz Centrality of the graph and its various nodes.

```
def katz_centrality(G, alpha=0.1, beta=1.0,
                    max_iter=1000, tol=1.0e-6,
                    nstart=None, normalized=True,
                    weight = 'weight'):
    """Compute the Katz centrality for the nodes
       of the graph G.
```

Katz centrality computes the centrality for a node based on the centrality of its neighbors. It is a generalization of the eigenvector centrality. The Katz centrality for node `i` is

```
.. math::

x_i = \alpha \sum_{j} A_{ij} x_j + \beta,
```

where `A` is the adjacency matrix of the graph G with eigenvalues `λ`.

The parameter `β` controls the initial centrality and

```
.. math::

\alpha < \frac{1}{\lambda_{\max}}.
```

Katz centrality computes the relative influence of a node within a network by measuring the number of the immediate neighbors (first degree nodes) and also all other nodes in the network that connect to the node under consideration through these immediate neighbors.

Extra weight can be provided to immediate neighbors through the parameter :math:`\beta`. Connections made with distant neighbors are, however, penalized by an attenuation factor `α` which should be strictly less than the inverse largest eigenvalue of the adjacency matrix in order for the Katz centrality to be computed correctly.

Parameters

G : graph
A NetworkX graph

alpha : float
Attenuation factor

beta : scalar or dictionary, optional (default=1.0)
Weight attributed to the immediate neighborhood.
If not a scalar, the dictionary must have an value
for every node.

max_iter : integer, optional (default=1000)
Maximum number of iterations in power method.

tol : float, optional (default=1.0e-6)
Error tolerance used to check convergence in
power method iteration.

nstart : dictionary, optional
Starting value of Katz iteration for each node.

normalized : bool, optional (default=True)
If True normalize the resulting values.

weight : None or string, optional
If None, all edge weights are considered equal.
Otherwise holds the name of the edge attribute
used as weight.

Returns

nodes : dictionary
Dictionary of nodes with Katz centrality as
the value.

Raises

NetworkXError
If the parameter `beta` is not a scalar but
lacks a value for at least one node

Notes

This algorithm it uses the power method to find the eigenvector corresponding to the largest eigenvalue of the adjacency matrix of G. The constant alpha should be strictly less than the inverse of largest eigenvalue of the adjacency matrix for the algorithm to converge. The iteration will stop after max_iter iterations or an error tolerance of number_of_nodes(G)*tol has been reached.

When $\alpha = 1/\lambda_{\max}$ and $\beta=0$, Katz centrality is the same as eigenvector centrality.

For directed graphs this finds "left" eigenvectors which corresponds to the in-edges in the graph. For out-edges Katz centrality first reverse the graph with G.reverse().

"""

```
from math import sqrt

if len(G) == 0:
    return {}

nnodes = G.number_of_nodes()

if nstart is None:

    # choose starting vector with entries of 0
    x = dict([(n,0) for n in G])
else:
    x = nstart

try:
    b = dict.fromkeys(G,float(beta))
except (TypeError,ValueError,AttributeError):
    b = beta
    if set(beta) != set(G):
        raise nx.NetworkXError('beta dictionary '
                               'must have a value for every node')

# make up to max_iter iterations
for i in range(max_iter):
    xlast = x
    x = dict.fromkeys(xlast, 0)
```

```

# do the multiplication y^T = Alpha * x^T A - Beta
for n in x:
    for nbr in G[n]:
        x[nbr] += xlast[n] * G[n][nbr].get(weight, 1)
for n in x:
    x[n] = alpha*x[n] + b[n]

# check convergence
err = sum([abs(x[n]-xlast[n]) for n in x])
if err < nnodes*tol:
    if normalized:

        # normalize vector
        try:
            s = 1.0/sqrt(sum(v**2 for v in x.values()))

            # this should never be zero?
            except ZeroDivisionError:
                s = 1.0
        else:
            s = 1
        for n in x:
            x[n] *= s
    return x

raise nx.NetworkXError('Power iteration failed to converge in '
                      '%d iterations.' % max_iter)

```

The above function is invoked using the networkx library and once the library is installed, you can eventually use it and the following code has to be written in python for the implementation of the katz centrality of a node.

```

>>> import networkx as nx
>>> import math
>>> G = nx.path_graph(4)
>>> phi = (1+math.sqrt(5))/2.0 # largest eigenvalue of adj matrix
>>> centrality = nx.katz_centrality(G,1/phi-0.01)
>>> for n,c in sorted(centrality.items()):
...     print("%d %0.2f"%(n,c))

```

The output of the above code is:

```

0 0.37
1 0.60
2 0.60
3 0.37

```

The above result is a dictionary depicting the value of katz centrality of each node. The above is an extension of my article series on the centrality measures. Keep networking!!!

References

<http://networkx.readthedocs.io/en/networkx-1.10/index.html>
https://en.wikipedia.org/wiki/Katz_centrality

Source

<https://www.geeksforgeeks.org/katz-centrality-centrality-measure/>

Chapter 185

Kruskal's Algorithm (Simple Implementation for Adjacency Matrix)

Kruskal's Algorithm (Simple Implementation for Adjacency Matrix) - GeeksforGeeks

Below are the steps for finding MST using [Kruskal's algorithm](#)

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are $(V-1)$ edges in the spanning tree.

We have discussed one implementation of Kruskal's algorithm in [previous post](#). In this post, a simpler implementation for adjacency matrix is discussed.

```
// Simple C++ implementation for Kruskal's
// algorithm
#include <bits/stdc++.h>
using namespace std;

#define V 5
int parent[V];

// Find set of vertex i
int find(int i)
{
    while (parent[i] != i)
        i = parent[i];
    return i;
}
```

```
}  
  
// Does union of i and j. It returns  
// false if i and j are already in same  
// set.  
void union1(int i, int j)  
{  
    int a = find(i);  
    int b = find(j);  
    parent[a] = b;  
}  
  
// Finds MST using Kruskal's algorithm  
void kruskalMST(int cost[][][V])  
{  
    int mincost = 0; // Cost of min MST.  
  
    // Initialize sets of disjoint sets.  
    for (int i = 0; i < V; i++)  
        parent[i] = i;  
  
    // Include minimum weight edges one by one  
    int edge_count = 0;  
    while (edge_count < V - 1) {  
        int min = INT_MAX, a = -1, b = -1;  
        for (int i = 0; i < V; i++) {  
            for (int j = 0; j < V; j++) {  
                if (find(i) != find(j) && cost[i][j] < min) {  
                    min = cost[i][j];  
                    a = i;  
                    b = j;  
                }  
            }  
        }  
  
        union1(a, b);  
        printf("Edge %d:(%d, %d) cost:%d \n",
              edge_count++, a, b, min);
        mincost += min;
    }
    printf("\n Minimum cost= %d \n", mincost);
}  
  
// driver program to test above function
int main()
{
    /* Let us create the following graph
       2      3
```

```
(0)--(1)--(2)
|   / \   |
6| 8/    \5 |7
| /       \ |
(3)-----(4)
         9      */
int cost[] [V] = {
    { INT_MAX, 2, INT_MAX, 6, INT_MAX },
    { 2, INT_MAX, 3, 8, 5 },
    { INT_MAX, 3, INT_MAX, INT_MAX, 7 },
    { 6, 8, INT_MAX, INT_MAX, 9 },
    { INT_MAX, 5, 7, 9, INT_MAX },
};

// Print the solution
kruskalMST(cost);

return 0;
}
```

Output:

```
Edge 0:(0, 1) cost:2
Edge 1:(1, 2) cost:3
Edge 2:(1, 4) cost:5
Edge 3:(0, 3) cost:6
```

```
Minimum cost= 16
```

Note that the above solution is not efficient. The idea is to provide a simple implementation for adjacency matrix representations. Please see below for efficient implementations.

[Kruskal's Minimum Spanning Tree Algorithm Greedy Algo-2](#)

[Kruskal's Minimum Spanning Tree using STL in C++](#)

Source

<https://www.geeksforgeeks.org/kruskals-algorithm-simple-implementation-for-adjacency-matrix/>

Chapter 186

Kruskal's Minimum Spanning Tree Algorithm Greedy Algo-2

Kruskal's Minimum Spanning Tree Algorithm Greedy Algo-2 - GeeksforGeeks

What is Minimum Spanning Tree?

Given a connected and undirected graph, a *spanning tree* of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A *minimum spanning tree (MST)* or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

How many edges does a minimum spanning tree has?

A minimum spanning tree has $(V - 1)$ edges where V is the number of vertices in the given graph.

What are the applications of Minimum Spanning Tree?

See [this](#)for applications of MST.

Below are the steps for finding MST using Kruskal's algorithm

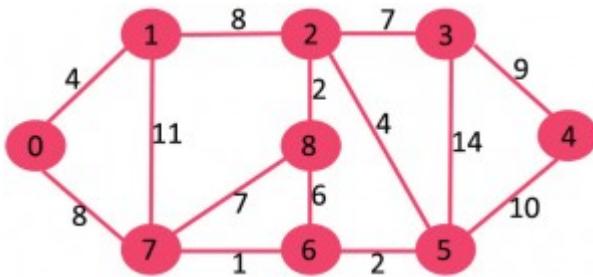
1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are $(V-1)$ edges in the spanning tree.

The step#2 uses [Union-Find algorithm](#) to detect cycle. So we recommend to read following post as a prerequisite.

[Union-Find Algorithm Set 1 \(Detect Cycle in a Graph\)](#)

[Union-Find Algorithm Set 2 \(Union By Rank and Path Compression\)](#)

The algorithm is a Greedy Algorithm. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far. Let us understand it with an example: Consider the below input graph.



The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having $(9 - 1) = 8$ edges.

After sorting:

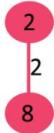
Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

Now pick all edges one by one from sorted list of edges

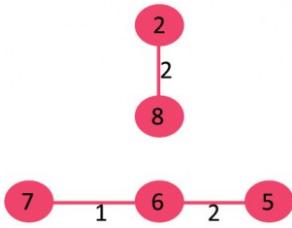
1. Pick edge 7-6: No cycle is formed, include it.



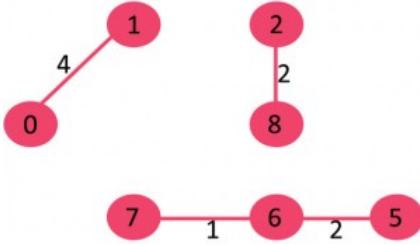
2. Pick edge 8-2: No cycle is formed, include it.



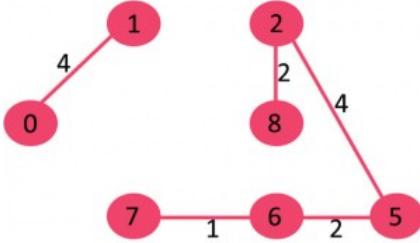
3. Pick edge 6-5: No cycle is formed, include it.



4. Pick edge 0-1: No cycle is formed, include it.

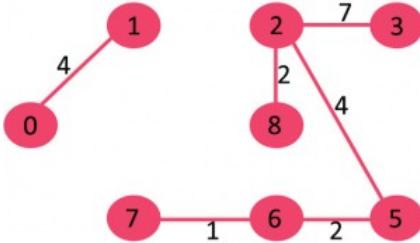


5. Pick edge 2-5: No cycle is formed, include it.



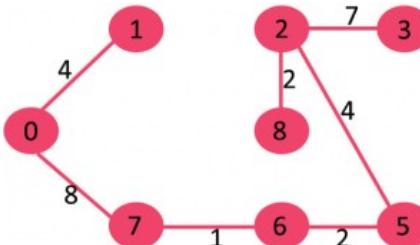
6. Pick edge 8-6: Since including this edge results in cycle, discard it.

7. Pick edge 2-3: No cycle is formed, include it.



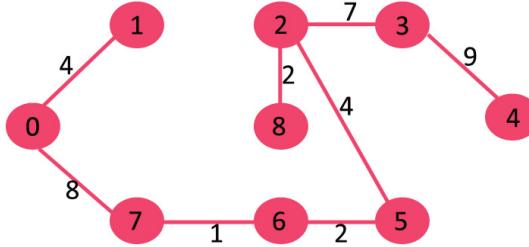
8. Pick edge 7-8: Since including this edge results in cycle, discard it.

9. Pick edge 0-7: No cycle is formed, include it.



10. Pick edge 1-2: Since including this edge results in cycle, discard it.

11. Pick edge 3-4: No cycle is formed, include it.



Since the number of edges included equals $(V - 1)$, the algorithm stops here.

C/C++

```
// C++ program for Kruskal's algorithm to find Minimum Spanning Tree
// of a given connected, undirected and weighted graph
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// a structure to represent a weighted edge in graph
struct Edge
{
    int src, dest, weight;
};

// a structure to represent a connected, undirected
// and weighted graph
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    // Since the graph is undirected, the edge
    // from src to dest is also edge from dest
    // to src. Both are counted as 1 edge here.
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
```

```
graph->edge = new Edge[E];  
  
    return graph;  
}  
  
// A structure to represent a subset for union-find  
struct subset  
{  
    int parent;  
    int rank;  
};  
  
// A utility function to find set of an element i  
// (uses path compression technique)  
int find(struct subset subsets[], int i)  
{  
    // find root and make root as parent of i  
    // (path compression)  
    if (subsets[i].parent != i)  
        subsets[i].parent = find(subsets, subsets[i].parent);  
  
    return subsets[i].parent;  
}  
  
// A function that does union of two sets of x and y  
// (uses union by rank)  
void Union(struct subset subsets[], int x, int y)  
{  
    int xroot = find(subsets, x);  
    int yroot = find(subsets, y);  
  
    // Attach smaller rank tree under root of high  
    // rank tree (Union by Rank)  
    if (subsets[xroot].rank < subsets[yroot].rank)  
        subsets[xroot].parent = yroot;  
    else if (subsets[xroot].rank > subsets[yroot].rank)  
        subsets[yroot].parent = xroot;  
  
    // If ranks are same, then make one as root and  
    // increment its rank by one  
    else  
    {  
        subsets[yroot].parent = xroot;  
        subsets[xroot].rank++;  
    }  
}  
  
// Compare two edges according to their weights.
```

```

// Used in qsort() for sorting an array of edges
int myComp(const void* a, const void* b)
{
    struct Edge* a1 = (struct Edge*)a;
    struct Edge* b1 = (struct Edge*)b;
    return a1->weight > b1->weight;
}

// The main function to construct MST using Kruskal's algorithm
void KruskalMST(struct Graph* graph)
{
    int V = graph->V;
    struct Edge result[V]; // This will store the resultant MST
    int e = 0; // An index variable, used for result[]
    int i = 0; // An index variable, used for sorted edges

    // Step 1: Sort all the edges in non-decreasing
    // order of their weight. If we are not allowed to
    // change the given graph, we can create a copy of
    // array of edges
    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);

    // Allocate memory for creating V subsets
    struct subset *subsets =
        (struct subset*) malloc( V * sizeof(struct subset) );

    // Create V subsets with single elements
    for (int v = 0; v < V; ++v)
    {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Number of edges to be taken is equal to V-1
    while (e < V - 1)
    {
        // Step 2: Pick the smallest edge. And increment
        // the index for next iteration
        struct Edge next_edge = graph->edge[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        // If including this edge does't cause cycle,
        // include it in result and increment the index
        // of result for next edge
        if (x != y)
        {

```

```

        result[e++] = next_edge;
        Union(subsets, x, y);
    }
    // Else discard the next_edge
}

// print the contents of result[] to display the
// built MST
printf("Following are the edges in the constructed MST\n");
for (i = 0; i < e; ++i)
    printf("%d -- %d == %d\n", result[i].src, result[i].dest,
                           result[i].weight);
return;
}

// Driver program to test above functions
int main()
{
    /* Let us create following weighted graph
       10
       0-----1
       |   \
       |   5\   |15
       |       \ |
       2-----3
       4           */
    int V = 4; // Number of vertices in graph
    int E = 5; // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = 10;

    // add edge 0-2
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 6;

    // add edge 0-3
    graph->edge[2].src = 0;
    graph->edge[2].dest = 3;
    graph->edge[2].weight = 5;

    // add edge 1-3
    graph->edge[3].src = 1;
}

```

```
graph->edge[3].dest = 3;
graph->edge[3].weight = 15;

// add edge 2-3
graph->edge[4].src = 2;
graph->edge[4].dest = 3;
graph->edge[4].weight = 4;

KruskalMST(graph);

return 0;
}

Java

// Java program for Kruskal's algorithm to find Minimum
// Spanning Tree of a given connected, undirected and
// weighted graph
import java.util.*;
import java.lang.*;
import java.io.*;

class Graph
{
    // A class to represent a graph edge
    class Edge implements Comparable<Edge>
    {
        int src, dest, weight;

        // Comparator function used for sorting edges
        // based on their weight
        public int compareTo(Edge compareEdge)
        {
            return this.weight - compareEdge.weight;
        }
    };

    // A class to represent a subset for union-find
    class subset
    {
        int parent, rank;
    };

    int V, E;      // V-> no. of vertices & E->no.of edges
    Edge edge[]; // collection of all edges

    // Creates a graph with V vertices and E edges
    Graph(int v, int e)
```

```

{
    V = v;
    E = e;
    edge = new Edge[E];
    for (int i=0; i<e; ++i)
        edge[i] = new Edge();
}

// A utility function to find set of an element i
// (uses path compression technique)
int find(subset subsets[], int i)
{
    // find root and make root as parent of i (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high rank tree
    // (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and increment
    // its rank by one
    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// The main function to construct MST using Kruskal's algorithm
void KruskalMST()
{
    Edge result[] = new Edge[V]; // This will store the resultant MST
    int e = 0; // An index variable, used for result[]
    int i = 0; // An index variable, used for sorted edges
}

```

```
for (i=0; i<V; ++i)
    result[i] = new Edge();

// Step 1: Sort all the edges in non-decreasing order of their
// weight. If we are not allowed to change the given graph, we
// can create a copy of array of edges
Arrays.sort(edge);

// Allocate memory for creating V subsets
subset subsets[] = new subset[V];
for(i=0; i<V; ++i)
    subsets[i]=new subset();

// Create V subsets with single elements
for (int v = 0; v < V; ++v)
{
    subsets[v].parent = v;
    subsets[v].rank = 0;
}

i = 0; // Index used to pick next edge

// Number of edges to be taken is equal to V-1
while (e < V - 1)
{
    // Step 2: Pick the smallest edge. And increment
    // the index for next iteration
    Edge next_edge = new Edge();
    next_edge = edge[i++];

    int x = find(subsets, next_edge.src);
    int y = find(subsets, next_edge.dest);

    // If including this edge does't cause cycle,
    // include it in result and increment the index
    // of result for next edge
    if (x != y)
    {
        result[e++] = next_edge;
        Union(subsets, x, y);
    }
    // Else discard the next_edge
}

// print the contents of result[] to display
// the built MST
System.out.println("Following are the edges in " +
                    "the constructed MST");
```

```
for (i = 0; i < e; ++i)
    System.out.println(result[i].src+" -- " +
                       result[i].dest+" == "+ result[i].weight);
}

// Driver Program
public static void main (String[] args)
{

/* Let us create following weighted graph
   10
   0-----1
   | \     |
   6|   5\   |15
   |   \   |
   2-----3
   4           */
int V = 4; // Number of vertices in graph
int E = 5; // Number of edges in graph
Graph graph = new Graph(V, E);

// add edge 0-1
graph.edge[0].src = 0;
graph.edge[0].dest = 1;
graph.edge[0].weight = 10;

// add edge 0-2
graph.edge[1].src = 0;
graph.edge[1].dest = 2;
graph.edge[1].weight = 6;

// add edge 0-3
graph.edge[2].src = 0;
graph.edge[2].dest = 3;
graph.edge[2].weight = 5;

// add edge 1-3
graph.edge[3].src = 1;
graph.edge[3].dest = 3;
graph.edge[3].weight = 15;

// add edge 2-3
graph.edge[4].src = 2;
graph.edge[4].dest = 3;
graph.edge[4].weight = 4;

graph.KruskalMST();
}
```

```
}
```

//This code is contributed by Aakash Hasija

Python

```
# Python program for Kruskal's algorithm to find
# Minimum Spanning Tree of a given connected,
# undirected and weighted graph

from collections import defaultdict

#Class to represent a graph
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices
        self.graph = [] # default dictionary
                           # to store graph

        # function to add an edge to graph
    def addEdge(self,u,v,w):
        self.graph.append([u,v,w])

    # A utility function to find set of an element i
    # (uses path compression technique)
    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    # A function that does union of two sets of x and y
    # (uses union by rank)
    def union(self, parent, rank, x, y):
        xroot = self.find(parent, x)
        yroot = self.find(parent, y)

        # Attach smaller rank tree under root of
        # high rank tree (Union by Rank)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot

        # If ranks are same, then make one as root
        # and increment its rank by one
        else :
            parent[yroot] = xroot
```

```

rank[xroot] += 1

# The main function to construct MST using Kruskal's
# algorithm
def KruskalMST(self):

    result = [] #This will store the resultant MST

    i = 0 # An index variable, used for sorted edges
    e = 0 # An index variable, used for result[]

    # Step 1: Sort all the edges in non-decreasing
    # order of their
    # weight. If we are not allowed to change the
    # given graph, we can create a copy of graph
    self.graph = sorted(self.graph,key=lambda item: item[2])

    parent = [] ; rank = []

    # Create V subsets with single elements
    for node in range(self.V):
        parent.append(node)
        rank.append(0)

    # Number of edges to be taken is equal to V-1
    while e < self.V - 1 :

        # Step 2: Pick the smallest edge and increment
        # the index for next iteration
        u,v,w = self.graph[i]
        i = i + 1
        x = self.find(parent, u)
        y = self.find(parent ,v)

        # If including this edge does't cause cycle,
        # include it in result and increment the index
        # of result for next edge
        if x != y:
            e = e + 1
            result.append([u,v,w])
            self.union(parent, rank, x, y)
        # Else discard the edge

    # print the contents of result[] to display the built MST
    print "Following are the edges in the constructed MST"
    for u,v,weight in result:
        #print str(u) + " -- " + str(v) + " == " + str(weight)
        print ("%d -- %d == %d" % (u,v,weight))

```

```
# Driver code
g = Graph(4)
g.addEdge(0, 1, 10)
g.addEdge(0, 2, 6)
g.addEdge(0, 3, 5)
g.addEdge(1, 3, 15)
g.addEdge(2, 3, 4)

g.KruskalMST()

#This code is contributed by Neelam Yadav
```

Following are the edges in the constructed MST

```
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
```

Time Complexity: $O(E \log E)$ or $O(E \log V)$. Sorting of edges takes $O(E \log E)$ time. After sorting, we iterate through all edges and apply find-union algorithm. The find and union operations can take atmost $O(\log V)$ time. So overall complexity is $O(E \log E + E \log V)$ time. The value of E can be atmost $O(V^2)$, so $O(\log V)$ are $O(\log E)$ same. Therefore, overall time complexity is $O(E \log E)$ or $O(E \log V)$

References:

<http://www.ics.uci.edu/~eppstein/161/960206.html>
http://en.wikipedia.org/wiki/Minimum_spanning_tree

This article is compiled by [Aashish Barnwal](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

<https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/>

Chapter 187

Kruskal's Minimum Spanning Tree using STL in C++

Kruskal's Minimum Spanning Tree using STL in C++ - GeeksforGeeks

Given an undirected, connected and weighted graph, find **Minimum Spanning Tree (MST)** of the graph using Kruskal's algorithm.

Input : Graph as an array of edges

Output : Edges of MST are

```
6 - 7  
2 - 8  
5 - 6  
0 - 1  
2 - 5  
2 - 3  
0 - 7  
3 - 4
```

Weight of MST is 37

Note : There are two possible MSTs, the other MST includes edge 1-2 in place of 0-7.

We have discussed below Kruskal's MST implementations.

[Greedy Algorithms Set 2 \(Kruskal's Minimum Spanning Tree Algorithm\)](#)

Below are the steps for finding MST using Kruskal's algorithm

1. Sort all the edges in non-decreasing order of their weight.

2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far.
If cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.

Here are some key points which will be useful for us in implementing the Kruskal's algorithm using STL.

1. Use a vector of edges which consist of all the edges in the graph and each item of a vector will contain 3 parameters: source, destination and the cost of an edge between the source and destination.

```
vector<pair<int, pair<int, int> > > edges;
```

Here in the outer pair (i.e pair<int,pair<int,int> >) the first element corresponds to the cost of a edge while the second element is itself a pair, and it contains two vertices of edge.

2. Use the inbuilt `std::sort` to sort the edges in the non-decreasing order; by default the sort function sort in non-decreasing order.
3. We use the [Union Find Algorithm](#) to check if it the current edge forms a cycle if it is added in the current MST. If yes discard it, else include it (union).

Pseudo Code:

```
// Initialize result
mst_weight = 0

// Create V single item sets
for each vertex v
    parent[v] = v;
    rank[v] = 0;

Sort all edges into non decreasing
order by weight w

for each (u, v) taken from the sorted list E
    do if FIND-SET(u) != FIND-SET(v)
        print edge(u, v)
        mst_weight += weight of edge(u, v)
        UNION(u, v)
```

Below is C++ implementation of above algorithm.

```
// C++ program for Kruskal's algorithm to find Minimum
// Spanning Tree of a given connected, undirected and
```

```
// weighted graph
#include<bits/stdc++.h>
using namespace std;

// Creating shortcut for an integer pair
typedef pair<int, int> iPair;

// Structure to represent a graph
struct Graph
{
    int V, E;
    vector< pair<int, iPair> > edges;

    // Constructor
    Graph(int V, int E)
    {
        this->V = V;
        this->E = E;
    }

    // Utility function to add an edge
    void addEdge(int u, int v, int w)
    {
        edges.push_back({w, {u, v}});
    }

    // Function to find MST using Kruskal's
    // MST algorithm
    int kruskalMST();
};

// To represent Disjoint Sets
struct DisjointSets
{
    int *parent, *rnk;
    int n;

    // Constructor.
    DisjointSets(int n)
    {
        // Allocate memory
        this->n = n;
        parent = new int[n+1];
        rnk = new int[n+1];

        // Initially, all vertices are in
        // different sets and have rank 0.
        for (int i = 0; i <= n; i++)
    }
```

```
{  
    rnk[i] = 0;  
  
    //every element is parent of itself  
    parent[i] = i;  
}  
}  
  
// Find the parent of a node 'u'  
// Path Compression  
int find(int u)  
{  
    /* Make the parent of the nodes in the path  
       from u--> parent[u] point to parent[u] */  
    if (u != parent[u])  
        parent[u] = find(parent[u]);  
    return parent[u];  
}  
  
// Union by rank  
void merge(int x, int y)  
{  
    x = find(x), y = find(y);  
  
    /* Make tree with smaller height  
       a subtree of the other tree */  
    if (rnk[x] > rnk[y])  
        parent[y] = x;  
    else // If rnk[x] <= rnk[y]  
        parent[x] = y;  
  
    if (rnk[x] == rnk[y])  
        rnk[y]++;
}  
};  
  
/* Functions returns weight of the MST*/  
  
int Graph::kruskalMST()  
{  
    int mst_wt = 0; // Initialize result  
  
    // Sort edges in increasing order on basis of cost  
    sort(edges.begin(), edges.end());  
  
    // Create disjoint sets  
    DisjointSets ds(V);
```

```
// Iterate through all sorted edges
vector< pair<int, iPair> >::iterator it;
for (it=edges.begin(); it!=edges.end(); it++)
{
    int u = it->second.first;
    int v = it->second.second;

    int set_u = ds.find(u);
    int set_v = ds.find(v);

    // Check if the selected edge is creating
    // a cycle or not (Cycle is created if u
    // and v belong to same set)
    if (set_u != set_v)
    {
        // Current edge will be in the MST
        // so print it
        cout << u << " - " << v << endl;

        // Update MST weight
        mst_wt += it->first;

        // Merge two sets
        ds.merge(set_u, set_v);
    }
}

return mst_wt;
}

// Driver program to test above functions
int main()
{
    /* Let us create above shown weighted
       and unidirected graph */
    int V = 9, E = 14;
    Graph g(V, E);

    // making above shown graph
    g.addEdge(0, 1, 4);
    g.addEdge(0, 7, 8);
    g.addEdge(1, 2, 8);
    g.addEdge(1, 7, 11);
    g.addEdge(2, 3, 7);
    g.addEdge(2, 8, 2);
    g.addEdge(2, 5, 4);
    g.addEdge(3, 4, 9);
    g.addEdge(3, 5, 14);
```

```
g.addEdge(4, 5, 10);
g.addEdge(5, 6, 2);
g.addEdge(6, 7, 1);
g.addEdge(6, 8, 6);
g.addEdge(7, 8, 7);

cout << "Edges of MST are \n";
int mst_wt = g.kruskalMST();

cout << "\nWeight of MST is " << mst_wt;

return 0;
}
```

Output :

```
Edges of MST are
6 - 7
2 - 8
5 - 6
0 - 1
2 - 5
2 - 3
0 - 7
3 - 4
```

```
Weight of MST is 37
```

Optimization:

The above code can be optimized to stop the main loop of Kruskal when number of selected edges become $V-1$. We know that MST has $V-1$ edges and there is no point iterating after $V-1$ edges are selected. We have not added this optimization to keep code simple.

References:

[Introduction to Algorithms by Cormen Leiserson Rivest and Stein\(CLRS\) 3](#)

Time complexity and step by step illustration are discussed in [previous post on Kruskal's algorithm.](#)

Source

<https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-using-stl-in-c/>

Chapter 188

Largest connected component on a grid

Largest connected component on a grid - GeeksforGeeks

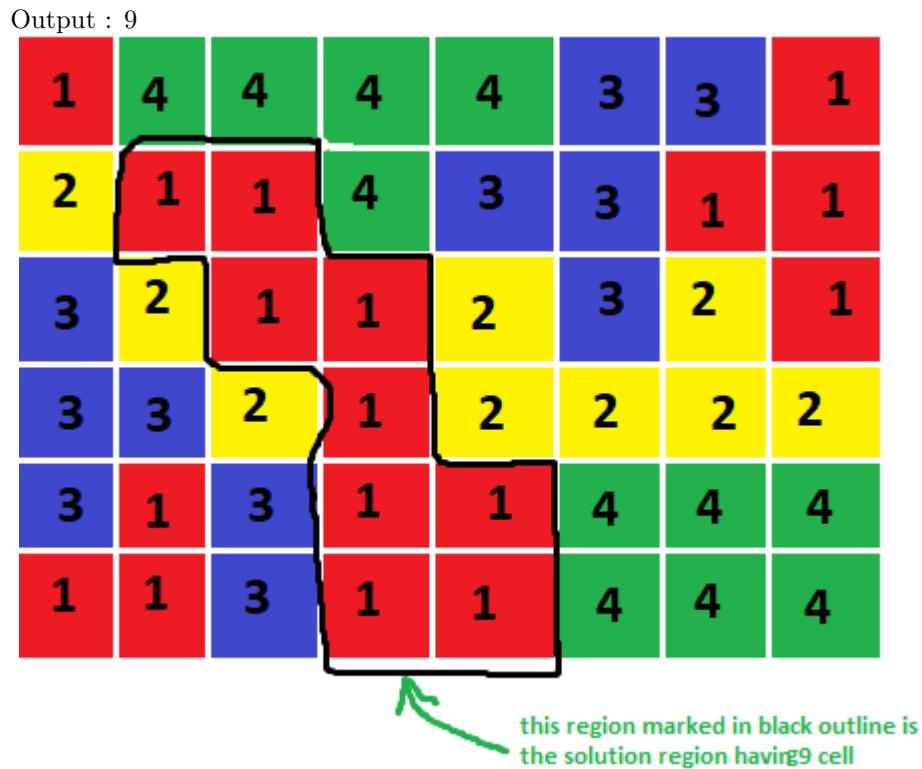
Given a grid with different colors in a different cell, each color represented by a different number. The task is to find out the largest connected component on the grid. Largest component grid refers to a maximum set of cells such that you can move from any cell to any other cell in this set by only moving between side-adjacent cells from the set.

Examples:

Input :

1	4	4	4	4	4	3	3	1
2	1	1	4	3	3	1	1	1
3	2	1	1	2	3	2	1	1
3	3	2	1	2	2	2	2	2
3	1	3	1	1	4	4	4	4
1	1	3	1	1	4	4	4	4

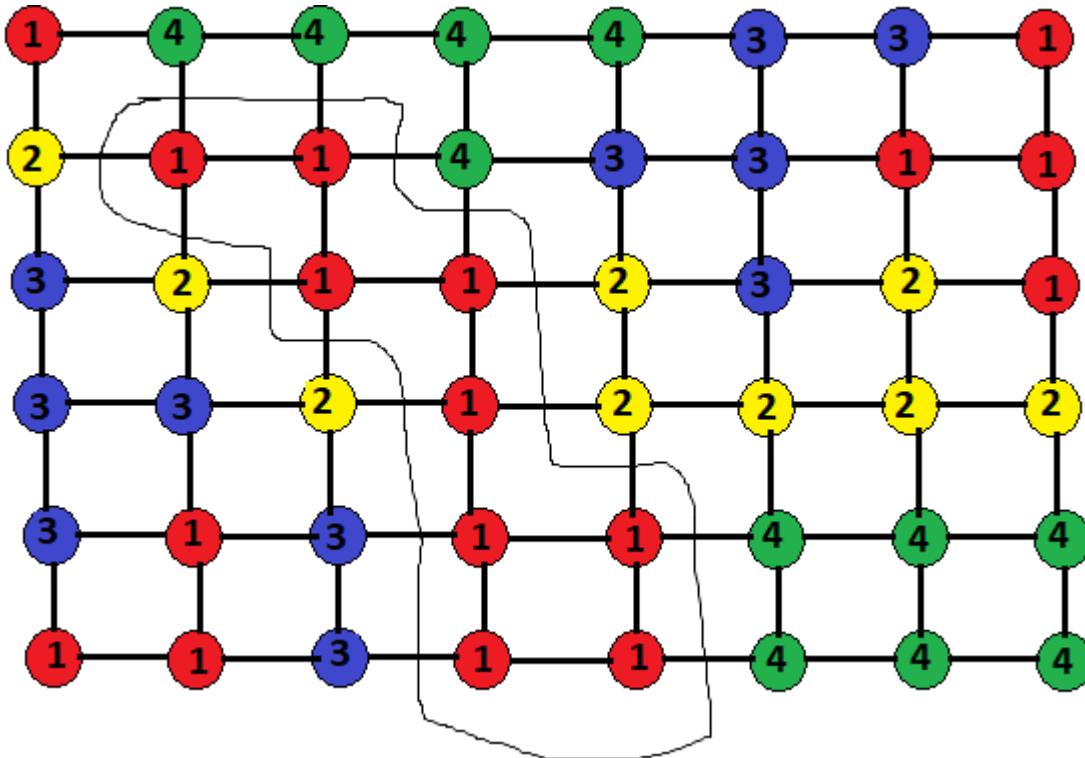
Grid of different colors



Largest connected component of grid

Approach :

The approach is to visualize the given grid as a graph with each cell representing a separate node of the graph and each node connected to four other nodes which are to immediately up, down, left, and right of that grid. Now doing a **BFS** search for every node of the graph, find *all the nodes connected to the current node with same color value as the current node*. Here is the graph for above example :



Graph representation of grid

At every cell (i, j) , a BFS can be done. The possible moves from a cell will be either to **right, left, top or bottom**. Move to only those cells which are in range and are of the same color. If the same nodes have been visited previously, then the largest component value of the grid is stored in **result[]** array. Using memoization, reduce the number of BFS on any cell. **visited[]** array is used to mark if the cell has been visited previously and count stores the count of the connected component when a BFS is done for every cell. Store the maximum of the count and print the resultant grid using **result[]** array.

Below is the illustration of the above approach:

C++

```
// CPP program to print the largest
// connected component in a grid
#include <bits/stdc++.h>
using namespace std;

const int n = 6;
const int m = 8;

// stores information about which cell
// are already visited in a particular BFS
```

```

int visited[n][m];

// result stores the final result grid
int result[n][m];

// stores the count of cells in the largest
// connected component
int COUNT;

// Function checks if a cell is valid i.e it
// is inside the grid and equal to the key
bool is_valid(int x, int y, int key, int input[n][m])
{
    if (x < n && y < m && x >= 0 && y >= 0) {
        if (visited[x][y] == false && input[x][y] == key)
            return true;
        else
            return false;
    }
    else
        return false;
}

// BFS to find all cells in
// connection with key = input[i][j]
void BFS(int x, int y, int i, int j, int input[n][m])
{
    // terminating case for BFS
    if (x != y)
        return;

    visited[i][j] = 1;
    COUNT++;

    // x_move and y_move arrays
    // are the possible movements
    // in x or y direction
    int x_move[] = { 0, 0, 1, -1 };
    int y_move[] = { 1, -1, 0, 0 };

    // checks all four points connected with input[i][j]
    for (int u = 0; u < 4; u++)
        if (is_valid(i + y_move[u], j + x_move[u], x, input))
            BFS(x, y, i + y_move[u], j + x_move[u], input);
}

// called every time before a BFS
// so that visited array is reset to zero

```

```
void reset_visited()
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            visited[i][j] = 0;
}

// If a larger connected component
// is found this function is called
// to store information about that component.
void reset_result(int key, int input[n][m])
{
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (visited[i][j] && input[i][j] == key)
                result[i][j] = visited[i][j];
            else
                result[i][j] = 0;
        }
    }
}

// function to print the result
void print_result(int res)
{
    cout << "The largest connected "
        << "component of the grid is :" << res << "\n";

    // prints the largest component
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (result[i][j])
                cout << result[i][j] << " ";
            else
                cout << ". ";
        }
        cout << "\n";
    }
}

// function to calculate the largest connected
// component
void computeLargestConnectedGrid(int input[n][m])
{
    int current_max = INT_MIN;

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            reset_visited();
```

```
COUNT = 0;

// checking cell to the right
if (j + 1 < m)
    BFS(input[i][j], input[i][j + 1], i, j, input);

// updating result
if (COUNT >= current_max) {
    current_max = COUNT;
    reset_result(input[i][j], input);
}
reset_visited();
COUNT = 0;

// checking cell downwards
if (i + 1 < n)
    BFS(input[i][j], input[i + 1][j], i, j, input);

// updating result
if (COUNT >= current_max) {
    current_max = COUNT;
    reset_result(input[i][j], input);
}
}

print_result(current_max);
}

// Drivers Code
int main()
{
    int input[n][m] = { { 1, 4, 4, 4, 4, 3, 3, 1 },
                        { 2, 1, 1, 4, 3, 3, 1, 1 },
                        { 3, 2, 1, 1, 2, 3, 2, 1 },
                        { 3, 3, 2, 1, 2, 2, 2, 2 },
                        { 3, 1, 3, 1, 1, 4, 4, 4 },
                        { 1, 1, 3, 1, 1, 4, 4, 4 } };

    // function to compute the largest
    // connected component in the grid
    computeLargestConnectedGrid(input);
    return 0;
}
```

Java

```
// Java program to print the largest
// connected component in a grid
import java.util.*;
```

```
import java.lang.*;
import java.io.*;

class GFG
{
static final int n = 6;
static final int m = 8;

// stores information about which cell
// are already visited in a particular BFS
static final int visited[][] = new int [n][m];

// result stores the final result grid
static final int result[][] = new int [n][m];

// stores the count of
// cells in the largest
// connected component
static int COUNT;

// Function checks if a cell
// is valid i.e it is inside
// the grid and equal to the key
static boolean is_valid(int x, int y,
                      int key,
                      int input[][])
{
    if (x < n && y < m &&
        x >= 0 && y >= 0)
    {
        if (visited[x][y] == 0 &&
            input[x][y] == key)
            return true;
        else
            return false;
    }
    else
        return false;
}

// BFS to find all cells in
// connection with key = input[i][j]
static void BFS(int x, int y, int i,
               int j, int input[][])
{
    // terminating case for BFS
    if (x != y)
        return;
```

```

visited[i][j] = 1;
COUNT++;

// x_move and y_move arrays
// are the possible movements
// in x or y direction
int x_move[] = { 0, 0, 1, -1 };
int y_move[] = { 1, -1, 0, 0 };

// checks all four points
// connected with input[i][j]
for (int u = 0; u < 4; u++)
    if ((is_valid(i + y_move[u],
                  j + x_move[u], x, input)) == true)
        BFS(x, y, i + y_move[u],
             j + x_move[u], input);
}

// called every time before
// a BFS so that visited
// array is reset to zero
static void reset_visited()
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            visited[i][j] = 0;
}

// If a larger connected component
// is found this function is
// called to store information
// about that component.
static void reset_result(int key,
                         int input[][])
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            if (visited[i][j] == 1 &&
                input[i][j] == key)
                result[i][j] = visited[i][j];
            else
                result[i][j] = 0;
        }
    }
}

```

```
// function to print the result
static void print_result(int res)
{
    System.out.println ("The largest connected " +
                        "component of the grid is :" +
                        res );

    // prints the largest component
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            if (result[i][j] != 0)
                System.out.print(result[i][j] + " ");
            else
                System.out.print(". ");
        }
        System.out.println();
    }
}

// function to calculate the
// largest connected component
static void computeLargestConnectedGrid(int input[][])
{
    int current_max = Integer.MIN_VALUE;

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            reset_visited();
            COUNT = 0;

            // checking cell to the right
            if (j + 1 < m)
                BFS(input[i][j], input[i][j + 1],
                     i, j, input);

            // updating result
            if (COUNT >= current_max)
            {
                current_max = COUNT;
                reset_result(input[i][j], input);
            }
            reset_visited();
            COUNT = 0;
        }
    }
}
```

```
// checking cell downwards
if (i + 1 < n)
    BFS(input[i][j],
        input[i + 1][j], i, j, input);

// updating result
if (COUNT >= current_max)
{
    current_max = COUNT;
    reset_result(input[i][j], input);
}
}

print_result(current_max);
}

// Driver Code
public static void main(String args[])
{
    int input[][] = {{1, 4, 4, 4, 4, 3, 3, 1},
                    {2, 1, 1, 4, 3, 3, 1, 1},
                    {3, 2, 1, 1, 2, 3, 2, 1},
                    {3, 3, 2, 1, 2, 2, 2, 2},
                    {3, 1, 3, 1, 1, 4, 4, 4},
                    {1, 1, 3, 1, 1, 4, 4, 4}};

    // function to compute the largest
    // connected component in the grid
    computeLargestConnectedGrid(input);
}
}

// This code is contributed by Subhadeep
```

Output:

The largest connected component of the grid is :9

```
. . . . .
. 1 1 . . .
. . 1 1 . . .
. . . 1 . . .
. . . 1 1 . .
. . . 1 1 . .
```

Improved By : [tufan_gupta2000](#)

Source

<https://www.geeksforgeeks.org/largest-connected-component-on-a-grid/>

Chapter 189

Largest subset of Graph vertices with edges of 2 or more colors

Largest subset of Graph vertices with edges of 2 or more colors - GeeksforGeeks

Given an undirected complete graph with N nodes or vertices. Edges of the graph are colored, find the largest subset of vertices with edges of 2 or more colors. We are given graph as adjacency matrix $C[][]$ where $C[i][j]$ is color of edge from vertex i to vertex j . Since graph is undirected, values $C[i][j]$ of $C[j][i]$ are same.

We define $C[i][i]$ to be zero, although there is no such edge present. i.e. the graph does not contain self-loops.

Examples:

Example 1:

```
Input : C[] []= {{0, 1, 2},  
                 {1, 0, 3},  
                 {2, 3, 0}}  
  
Output : 3
```

Example 2:

```
Input : C[] []= {{0, 1, 1},  
                 {1, 0, 3},  
                 {1, 3, 0}}  
  
Output : 0
```

Since graph is complete, each edge can be one of $n*(n-1)/2 + 1$ different colors. These colors are labeled from 0 to $n*(n-1)/2$, inclusive. But not all these $n*(n-1)/2 + 1$ colors need to be used. i.e., it is possible that two different edges could have the same color.

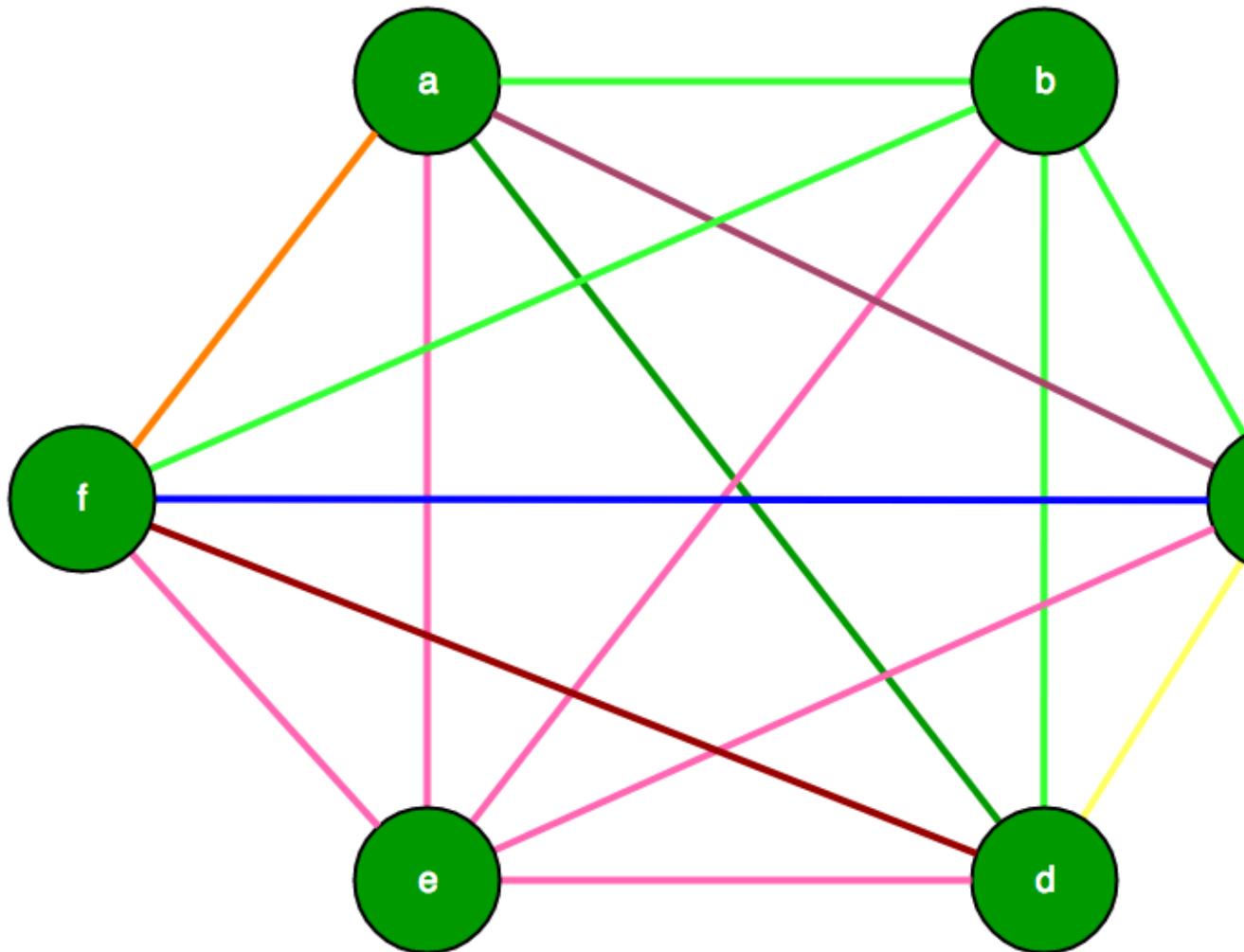
Let's call a vertex "bad" if all its neighbors are of the same color. Obviously, we can't have any such bad vertex in our subset, so remove such bad vertex from the graph. This might

introduce some more bad vertices, but we can keep repeating this process until we find a subset free of bad vertices. So, at last, we should remain through a graph which does not have any bad vertex means every vertex of our subset has at least two different color edges with other adjacent vertices.

Example:

Input :

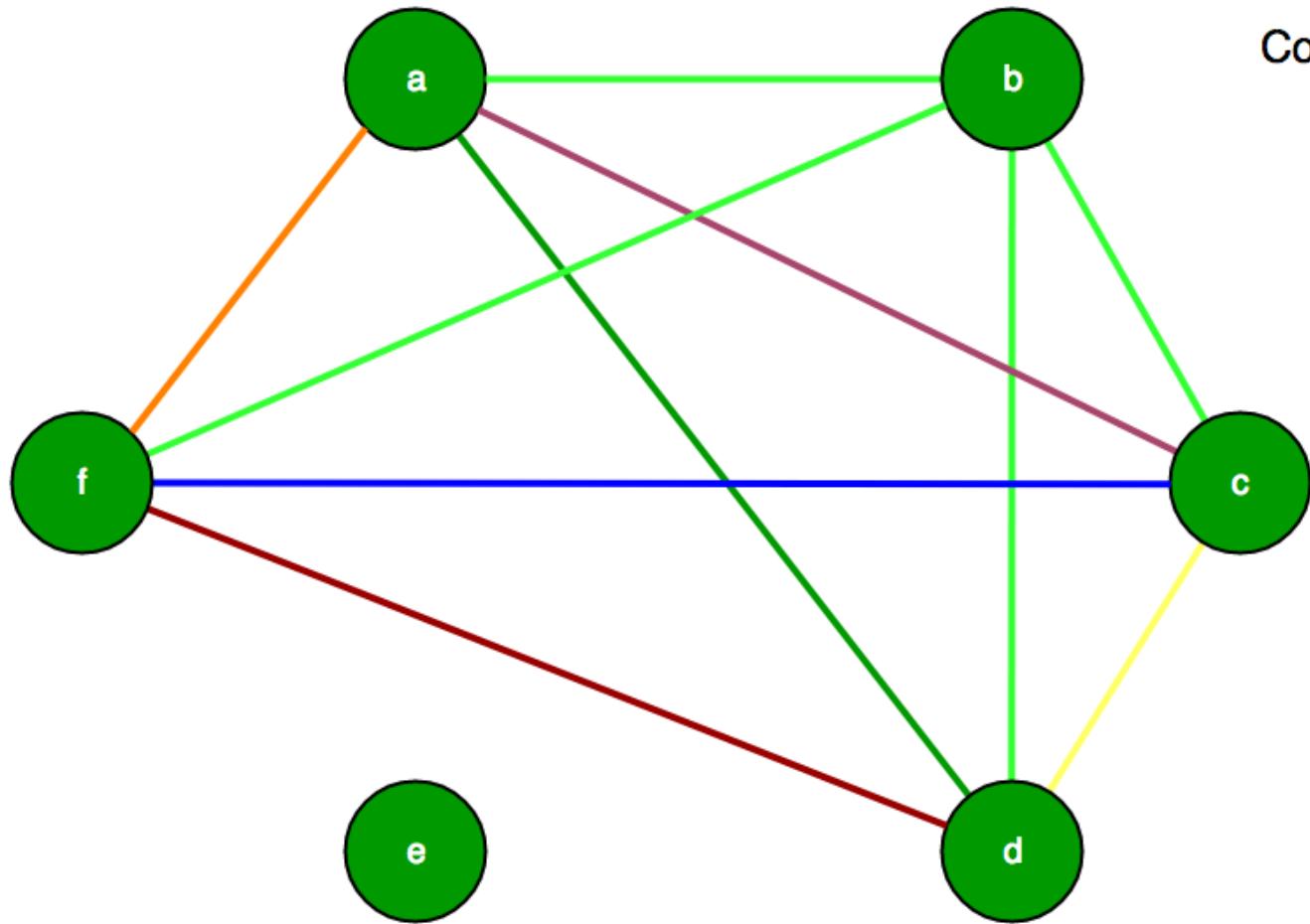
```
let C[6][6]:  
{{0, 9, 2, 4, 7, 8},  
{9, 0, 9, 9, 7, 9},  
{2, 9, 0, 3, 7, 6},  
{4, 9, 3, 0, 7, 1},  
{7, 7, 7, 7, 0, 7},  
{8, 9, 6, 1, 7, 0}};
```



Step I: First of all, we can see that row 5(node 'e') contains only 7 means node 'e' is connected through edges having color code 7 so it does not have more than one color edge so we have to remove 5 from subset. Now, our graph will contain only 5 vertex and are as:

$C[5][5]$:

$\{\{0, 9, 2, 4, 8\},$
 $\{9, 0, 9, 9, 9\},$
 $\{2, 9, 0, 3, 6\},$
 $\{4, 9, 3, 0, 1\},$
 $\{8, 9, 6, 1, 0\}\};$



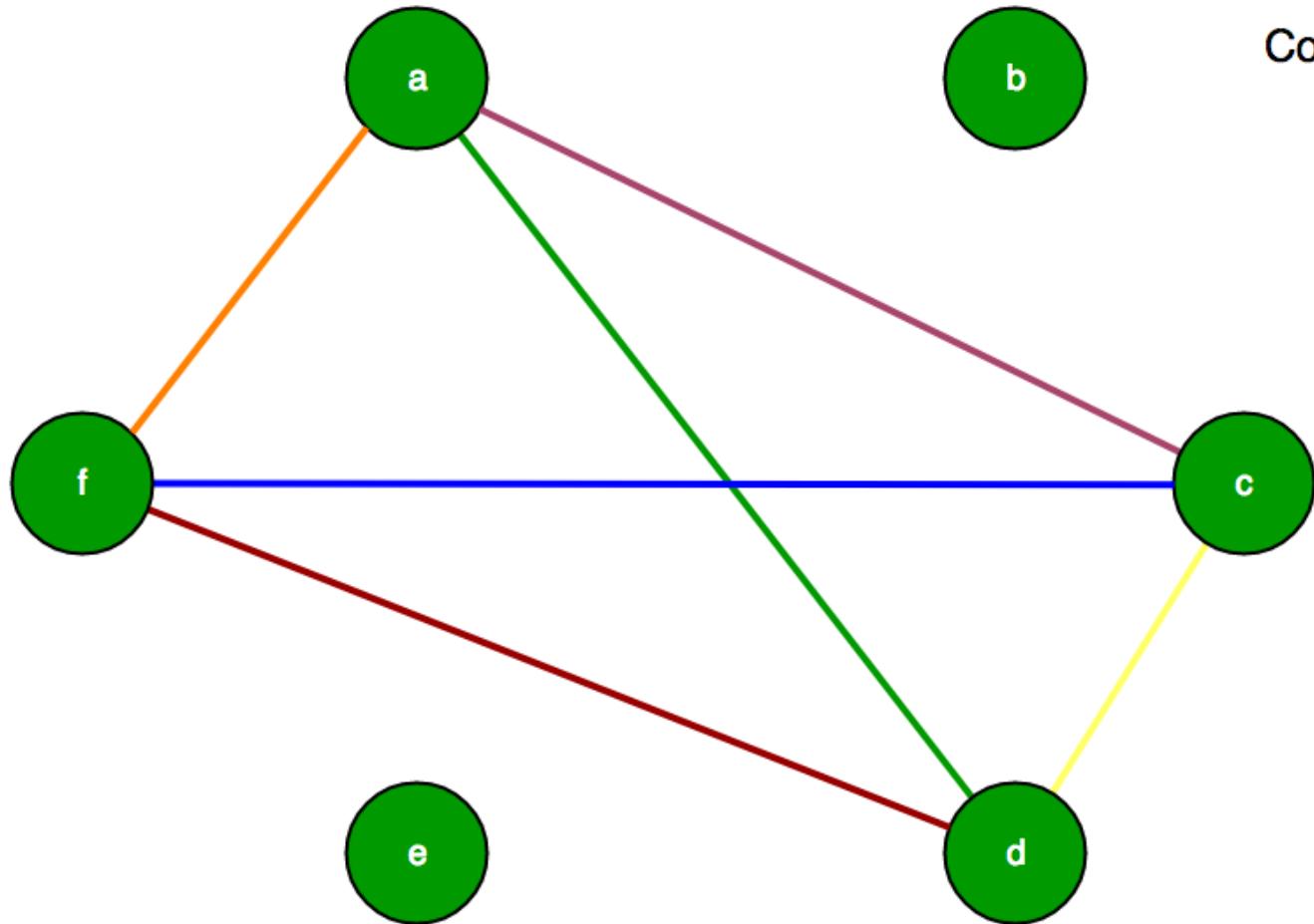
Graph after disconnecting node 'e'

Step II: Further, we can see that row 2 (node 'b') also doesn't contain more than 1 color edge, so we should remove row 2 and column 2 also. Which result in our new graph as:

$C[4][4]$:

$\{\{0, 2, 4, 8\},$

```
{2, 0, 3, 6},  
{4, 3, 0, 1},  
{8, 6, 1, 0}};
```



Final graph having 4 nodes connected

Step III: Now, we can see that each vertex has more than 1 different color edge. So, the total number of vertices in the subset is 4.

```
// C++ program to find size of subset of graph vertex  
// such that each vertex has more than 1 color edges  
#include <bits/stdc++.h>  
using namespace std;  
  
// Number of vertices  
const int N = 6;
```

```
// function to calculate max subset size
int subsetGraph(int C[][][N])
{
    // set for number of vertices
    set<int> vertices;
    for (int i = 0; i < N; ++i)
        vertices.insert(i);

    // loop for deletion of vertex from set
    while (!vertices.empty())
    {
        // if subset has only 1 vertex return 0
        if (vertices.size() == 1)
            return 1;

        // for each vertex iterate and keep removing
        // a vertex while we find a vertex with all
        // edges of same color.
        bool someone_removed = false;
        for (int x : vertices)
        {
            // note down different color values
            // for each vertex
            set<int> values;
            for (int y : vertices)
                if (y != x)
                    values.insert(C[x][y]);

            // if only one color is found
            // erase that vertex (bad vertex)
            if (values.size() == 1)
            {
                vertices.erase(x);
                someone_removed = true;
                break;
            }
        }

        // If no vertex was removed in the
        // above loop.
        if (!someone_removed)
            break;
    }

    return (vertices.size());
}
```

```
// Driver program
int main()
{
    int C[][][N] = {{0, 9, 2, 4, 7, 8},
                    {9, 0, 9, 9, 7, 9},
                    {2, 9, 0, 3, 7, 6},
                    {4, 9, 3, 0, 7, 1},
                    {7, 7, 7, 7, 0, 7},
                    {8, 9, 6, 1, 7, 0}
    };
    cout << subsetGraph(C);
    return 0;
}
```

Output:

4

Source

<https://www.geeksforgeeks.org/largest-subset-graph-vertices-edges-2-colors/>

Chapter 190

Level Ancestor Problem

Level Ancestor Problem - GeeksforGeeks

The [level ancestor problem](#) is the problem of preprocessing a given rooted tree T into a data structure that can determine the ancestor of a given node at a given depth from the root of the tree. Here **depth** of any node in a tree is the number of edges on the shortest path from the root of the tree to the node.

Given tree is represented as **un-directed connected graph** having n nodes and **$n-1$ edges**.

The idea to solve the above query is to use **Jump Pointer Algorithm** and pre-processes the tree in $O(n \log n)$ time and answer level ancestor queries in $O(\log n)$ time. In jump pointer, there is a pointer from node N to N 's j -th ancestor, for $j = 1, 2, 4, 8, \dots$, and so on. We refer to these pointers as $\text{Jump}_N[i]$, where $\text{Jump}_u[i] = \text{LA}(N, \text{depth}(N) - 2^i)$.

When the algorithm is asked to process a query, we repeatedly jump up the tree using these jump pointers. The number of jumps will be at most $\log n$ and therefore queries can be answered in $O(\log n)$ time.

So we store **2^i th ancestor** of each node and also find the depth of each node from the root of the tree.

Now our task reduces to find the $(\text{depth}(N) - 2^i)$ th ancestor of node N . Let's denote X as $(\text{depth}(N) - 2^i)$ and let b bits of the X are **set bits** (1) denoted by $s_1, s_2, s_3, \dots, s_b$.
$$X = 2^{(s_1)} + 2^{(s_2)} + \dots + 2^{(s_b)}$$

Now the problem is how to find 2^j ancestors of each node and depth of each node from the root of the tree?

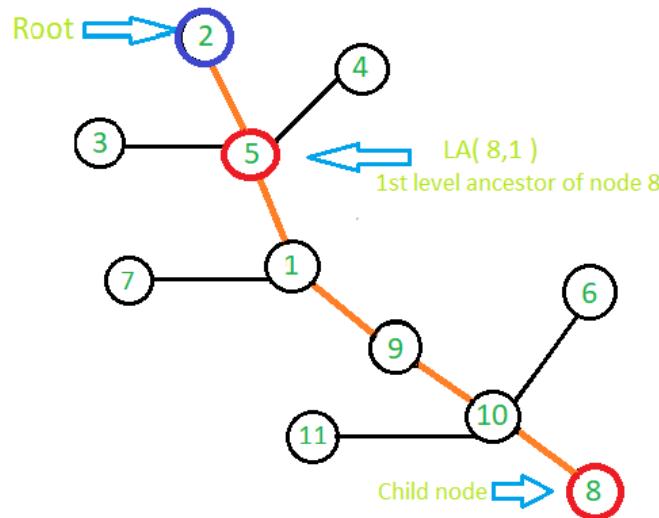
Initially, we know the 2^0 th ancestor of each node is its parent. We can recursively compute 2^j -th ancestor. We know 2^j -th ancestor is 2^{j-1} -th ancestor of 2^{j-1} -th ancestor. To calculate the depth of each node we use the ancestor matrix. If we found the root node present in the array of the k th element at j th index then the depth of that node is simply 2^j but if root node doesn't present in the array of ancestors of the k th element than the depth of k th element is $2^{(\text{index of last non zero ancestor at } k\text{th row})} + \text{depth of ancestor present at last index of } k\text{th row}$.

Below is the algorithm to fill the ancestor matrix and depth of each node using dynamic programming. Here, we denote root node as **R** and initially assume the ancestor of root node as **0**. We also initialize depth array with **-1** means the depth of the current node is not set and we need to find its depth. If the depth of the current node is not equal to **-1** means we have already computed its depth.

```
we know the first ancestor of each node so we take j>=1,
For j>=1
```

```
ancstr[k][j] = 2jth ancestor of k
              = 2j-1th ancestor of (2j-1th ancestor of k)
              = ancstr[ancstr[i][j-1][j-1]
                if ancstr[k][j] == R && depth[k] == -1
                  depth[k] = 2j
                else if ancstr[k][j] == -1 && depth[k] == -1
                  depth[k] = 2(j-1) + depth[ ancstr[k][j-1] ]
```

Let's understand this algorithm with below diagram.



In the given figure we need to compute 1st level ancestor of the node with value **8**. First, we make ancestor matrix which stores 2^{th} ancestor of nodes. Now, 2^0 ancestor of node **8** is **10** and similarly 2^0 ancestor of node **10** is **9** and for node **9** it is **1** and for node **1** it is **5**. Based on the above algorithm 1st level ancestor of node **8** is (**depth(8)-1**)**th** ancestor of node **8**. We have pre computed depth of each node and depth of **8** is **5** so we finally need to find **(5-1) = 4th** ancestor of node **8** which is equal to **2^1 th ancestor of [** 2^1 **ancestor of**

node 8] and 2^1 th ancestor of node 8 is 2^0 th ancestor of [2^0 th ancestor of node 8]. So, 2^0 th ancestor of [2^0 th ancestor of node 8] is node with value 9 and 2^1 th ancestor of node 9 is node with value 5. Thus in this way we can compute all query in $O(\log n)$ time complexity.

```

// CPP program to implement Level Ancestor Algorithm
#include <bits/stdc++.h>
using namespace std;
int R = 0;

// n -> it represent total number of nodes
// len -> it is the maximum length of array to hold
//         ancestor of each node. In worst case,
// the highest value of ancestor a node can have is n-1.
//  $2^{\lceil \log_2 n \rceil} \leq n-1$ 
// len =  $O(\log n)$ 
int getLen(int n)
{
    return (int)(log(n) / log(2)) + 1;
}

// ancstr represents 2D matrix to hold ancestor of node.
// Here we pass reference of 2D matrix so that the change
// made occur directly to the original matrix
// depth[] stores depth of each node
// len is same as defined above
// n is total nodes in graph
// R represent root node
void setancestor(vector<vector<int>>& ancstr,
                  vector<int>& depth, int* node, int len, int n)
{
    // depth of root node is set to 0
    depth[R] = 0;

    // if depth of a node is -1 it means its depth
    // is not set otherwise we have computed its depth
    for (int j = 1; j <= len; j++) {
        for (int i = 0; i < n; i++) {
            ancstr[node[i]][j] = ancstr[ancstr[node[i]][j - 1]][j - 1];

            // if ancestor of current node is R its height is
            // previously not set, then its height is  $2^j$ 
            if (ancstr[node[i]][j] == R && depth[node[i]] == -1) {

                // set the depth of ith node
                depth[node[i]] = pow(2, j);
            }
        }
    }
}

```

```

// if ancestor of current node is 0 means it
// does not have root node at its 2th power
// on its path so its depth is 2^(index of
// last non zero ancestor means j-1) + depth
// of 2^(j-1) th ancestor
else if (ancstr[node[i]][j] == 0 &&
         node[i] != R && depth[node[i]] == -1) {
    depth[node[i]] = pow(2, j - 1) +
        depth[ancstr[node[i]][j - 1]];
}
}

}

// c -> it represent child
// p -> it represent ancestor
// i -> it represent node number
// p=0 means the node is root node
// R represent root node
// here also we pass reference of 2D matrix and depth
// vector so that the change made occur directly to
// the original matrix and original vector
void constructGraph(vector<vector<int> >& ancstr,
                     int* node, vector<int>& depth, int* isNode,
                     int c, int p, int i)
{
    // enter the node in node array
    // it stores all the nodes in the graph
    node[i] = c;

    // to confirm that no child node have 2 ancestors
    if (isNode == 0) {
        isNode = 1;

        // make ancestor of x as y
        ancstr[0] = p;

        // if its first ancestor is root than its depth is 1
        if (R == p) {
            depth = 1;
        }
    }
    return;
}

// this function will delete leaf node
// x is node to be deleted
void removeNode(vector<vector<int> >& ancstr,

```

```

        int* isNode, int len, int x)
{
    if (isNode[x] == 0)
        cout << "node does not present in graph " << endl;
    else {
        isNode[x] = 0;

        // make all ancestor of node x as 0
        for (int j = 0; j <= len; j++) {
            ancstr[x][j] = 0;
        }
    }
    return;
}

// x -> it represent new node to be inserted
// p -> it represent ancestor of new node
void addNode(vector<vector<int> &> ancstr,
             vector<int> & depth, int* isNode, int len,
             int x, int p)
{
    if (isNode[x] == 1) {
        cout << " Node is already present in array " << endl;
        return;
    }
    if (isNode[p] == 0) {
        cout << " ancestor not does not present in an array " << endl;
        return;
    }

    isNode[x] = 1;
    ancstr[x][0] = p;

    // depth of new node is 1 + depth of its ancestor
    depth[x] = depth[p] + 1;
    int j = 0;

    // while we don't reach root node
    while (ancstr[x][j] != 0) {
        ancstr[x][j + 1] = ancstr[ancstr[x][j]][j];
        j++;
    }

    // remaining array will fill with 0 after
    // we find root of tree
    while (j <= len) {
        ancstr[x][j] = 0;
        j++;
    }
}

```

```

    }
    return;
}

// LA function to find Lth level ancestor of node x
void LA(vector<vector<int> >& ancstr, vector<int> depth,
        int* isNode, int x, int L)
{
    int j = 0;
    int temp = x;

    // to check if node is present in graph or not
    if (isNode[x] == 0) {
        cout << "Node is not present in graph " << endl;
        return;
    }

    // we change L as depth of node x -
    int k = depth[x] - L;
    // int q = k;
    // in this loop we decrease the value of k by k/2 and
    // increment j by 1 after each iteration, and check for set bit
    // if we get set bit then we update x with jth ancestor of x
    // as k becomes less than or equal to zero means we
    // reach to kth level ancestor
    while (k > 0) {

        // to check if last bit is 1 or not
        if (k & 1) {
            x = ancstr[x][j];
        }

        // use of shift operator to make k = k/2
        // after every iteration
        k = k >> 1;
        j++;
    }
    cout << L << "th level ancestor of node "
        << temp << " is = " << x << endl;
}

int main()
{
    // n represent number of nodes
    int n = 12;
}

```

```
// initialization of ancestor matrix
// suppose max range of node is up to 1000
// if there are 1000 nodes than also length
// of ancestor matrix will not exceed 10
vector<vector<int>> ancestor(1000, vector<int>(10));

// this vector is used to store depth of each node.
vector<int> depth(1000);

// fill function is used to initialize depth with -1
fill(depth.begin(), depth.end(), -1);

// node array is used to store all nodes
int* node = new int[1000];

// isNode is an array to check whether a
// node is present in graph or not
int* isNode = new int[1000];

// memset function to initialize isNode array with 0
memset(isNode, 0, 1000 * sizeof(int));

// function to calculate len
// len -> it is the maximum length of array to
// hold ancestor of each node.
int len = getLen(n);

// R stores root node
R = 2;

// construction of graph
// here 0 represent that the node is root node
constructGraph(ancestor, node, depth, isNode, 2, 0, 0);
constructGraph(ancestor, node, depth, isNode, 5, 2, 1);
constructGraph(ancestor, node, depth, isNode, 3, 5, 2);
constructGraph(ancestor, node, depth, isNode, 4, 5, 3);
constructGraph(ancestor, node, depth, isNode, 1, 5, 4);
constructGraph(ancestor, node, depth, isNode, 7, 1, 5);
constructGraph(ancestor, node, depth, isNode, 9, 1, 6);
constructGraph(ancestor, node, depth, isNode, 10, 9, 7);
constructGraph(ancestor, node, depth, isNode, 11, 10, 8);
constructGraph(ancestor, node, depth, isNode, 6, 10, 9);
constructGraph(ancestor, node, depth, isNode, 8, 10, 10);

// function to pre compute ancestor matrix
setancestor(ancestor, depth, node, len, n);

// query to get 1st level ancestor of node 8
```

```
LA(ancestor, depth, isNode, 8, 1);

// add node 12 and its ancestor is 8
addNode(ancestor, depth, isNode, len, 12, 8);

// query to get 2nd level ancestor of node 12
LA(ancestor, depth, isNode, 12, 2);

// delete node 12
removeNode(ancestor, isNode, len, 12);

// query to get 5th level ancestor of node
// 12 after deletion of node
LA(ancestor, depth, isNode, 12, 1);

return 0;
}
```

Output:

```
1th level ancestor of node 8 is = 5
2th level ancestor of node 12 is = 1
Node is not present in graph
```

Source

<https://www.geeksforgeeks.org/level-ancestor-problem/>

Chapter 191

Level of Each node in a Tree from source node (using BFS)

Level of Each node in a Tree from source node (using BFS) - GeeksforGeeks

[BFS\(Breadth First Search\)](#) is a graph traversal technique where a node and its neighbors are visited first and then the neighbors of neighbors. In simple terms it traverses level wise from the source. First it traverses level 1 nodes (direct neighbors of source node) and then level 2 nodes (neighbors of neighbors of source node) and so on.

Now, suppose if we have to know at which level all the nodes are at (from source node). Then BFS can be used to determine the level of each node.

Examples:

Input :

Output :	Node	Level
	0	0
	1	1
	2	1
	3	2
	4	2
	5	2
	6	2
	7	3

Explanation :

```
// CPP Program to determine level of each node
// and print level
#include <iostream>
```

```
#include <queue>
#include <vector>
using namespace std;

// function to determine level of each node starting
// from x using BFS
void printLevels(vector<int> graph[], int V, int x)
{
    // array to store level of each node
    int level[V];
    bool marked[V];

    // create a queue
    queue<int> que;

    // enqueue element x
    que.push(x);

    // initialize level of source node to 0
    level[x] = 0;

    // marked it as visited
    marked[x] = true;

    // do until queue is empty
    while (!que.empty()) {

        // get the first element of queue
        x = que.front();

        // dequeue element
        que.pop();

        // traverse neighbors of node x
        for (int i = 0; i < graph[x].size(); i++) {
            // b is neighbor of node x
            int b = graph[x][i];

            // if b is not marked already
            if (!marked[b]) {

                // enqueue b in queue
                que.push(b);

                // level of b is level of x + 1
                level[b] = level[x] + 1;

                // mark b
            }
        }
    }
}
```

```
        marked[b] = true;
    }
}

// display all nodes and their levels
cout << "Nodes"
<< "    "
<< "Level" << endl;
for (int i = 0; i < V; i++)
    cout << " " << i << " --> " << level[i] << endl;
}

// Dirver Code
int main()
{
    // adjacency graph for tree
    int V = 8;
    vector<int> graph[V];

    graph[0].push_back(1);
    graph[0].push_back(2);
    graph[1].push_back(3);
    graph[1].push_back(4);
    graph[1].push_back(5);
    graph[2].push_back(5);
    graph[2].push_back(6);
    graph[6].push_back(7);

    // call levels function with source as 0
    printLevels(graph, V, 0);

    return 0;
}
```

Output:

Nodes	Level
0	--> 0
1	--> 1
2	--> 1
3	--> 2
4	--> 2
5	--> 2
6	--> 2
7	--> 3

Source

<https://www.geeksforgeeks.org/level-node-tree-source-node-using-bfs/>

Chapter 192

Longest path between any pair of vertices

Longest path between any pair of vertices - GeeksforGeeks

We are given a map of cities connected with each other via cable lines such that there is no cycle between any two cities. We need to find the maximum length of cable between any two cities for given city map.

```
Input : n = 6
        1 2 3 // Cable length from 1 to 2 (or 2 to 1) is 3
        2 3 4
        2 6 2
        6 4 6
        6 5 5
Output: maximum length of cable = 12
```

Method 1 (Simple DFS)

We create undirected graph for given city map and do **DFS** from every city to find maximum length of cable. While traversing, we look for total cable length to reach the current city and if it's adjacent city is not visited then call **DFS** for it but if all adjacent cities are visited for current node, then update the value of `max_length` if previous value of `max_length` is less than current value of total cable length.

```
// C++ program to find the longest cable length
// between any two cities.
#include<bits/stdc++.h>
using namespace std;

// visited[] array to make nodes visited
```

```
// src is starting node for DFS traversal
// prev_len is sum of cable length till current node
// max_len is pointer which stores the maximum length
// of cable value after DFS traversal
void DFS(vector< pair<int,int> > graph[], int src,
         int prev_len, int *max_len,
         vector <bool> &visited)
{
    // Mark the src node visited
    visited[src] = 1;

    // curr_len is for length of cable from src
    // city to its adjacent city
    int curr_len = 0;

    // Adjacent is pair type which stores
    // destination city and cable length
    pair < int, int > adjacent;

    // Traverse all adjacent
    for (int i=0; i<graph[src].size(); i++)
    {
        // Adjacent element
        adjacent = graph[src][i];

        // If node or city is not visited
        if (!visited[adjacent.first])
        {
            // Total length of cable from src city
            // to its adjacent
            curr_len = prev_len + adjacent.second;

            // Call DFS for adjacent city
            DFS(graph, adjacent.first, curr_len,
                 max_len, visited);
        }

        // If total cable length till now greater than
        // previous length then update it
        if ((*max_len) < curr_len)
            *max_len = curr_len;

        // make curr_len = 0 for next adjacent
        curr_len = 0;
    }
}

// n is number of cities or nodes in graph
```

```
// cable_lines is total cable_lines among the cities
// or edges in graph
int longestCable(vector<pair<int,int> > graph[],
                  int n)
{
    // maximum length of cable among the connected
    // cities
    int max_len = INT_MIN;

    // call DFS for each city to find maximum
    // length of cable
    for (int i=1; i<=n; i++)
    {
        // initialize visited array with 0
        vector< bool > visited(n+1, false);

        // Call DFS for src vertex i
        DFS(graph, i, 0, &max_len, visited);
    }

    return max_len;
}

// driver program to test the input
int main()
{
    // n is number of cities
    int n = 6;

    vector< pair<int,int> > graph[n+1];

    // create undirected graph
    // first edge
    graph[1].push_back(make_pair(2, 3));
    graph[2].push_back(make_pair(1, 3));

    // second edge
    graph[2].push_back(make_pair(3, 4));
    graph[3].push_back(make_pair(2, 4));

    // third edge
    graph[2].push_back(make_pair(6, 2));
    graph[6].push_back(make_pair(2, 2));

    // fourth edge
    graph[4].push_back(make_pair(6, 6));
    graph[6].push_back(make_pair(4, 6));
```

```
// fifth edge
graph[5].push_back(make_pair(6, 5));
graph[6].push_back(make_pair(5, 5));

cout << "Maximum length of cable = "
<< longestCable(graph, n);

return 0;
}
```

Output:

```
Maximum length of cable = 12
```

Time Complexity : $O(V * (V + E))$

Method 2 (Efficient : Works only if Graph is Directed)

We can solve above problem in $O(V+E)$ time if the given graph is directed instead of undirected graph. Below are steps.

- 1) Create a distance array dist[] and initialize all entries of it as minus infinite
- 2) Order all vertices in **toplogical order**.
- 3) Do following for every vertex u in topological order.
....Do following for every adjacent vertex v of u
.....if ($dist[v] < dist[u] + weight(u, v)$)
..... $dist[v] = dist[u] + weight(u, v)$
- 4) Return maximum value from dist[]

Since there is no negative weight, processing vertices in topological order would always produce an array of longest paths dist[] such that $dist[u]$ indicates longest path ending at vertex 'u'.

The implementation of above approach can be easily adopted from [here](#). The differences here are, there are no negative weight edges and we need overall longest path (not longest paths from a source vertex). Finally we return maximum of all values in dist[].

Time Complexity : $O(V + E)$

This article is contributed by **Shashank Mishra (Gullu)**. This article is reviewed by team GeeksForGeeks. If you have any better approach for this problem then please share.

Source

<https://www.geeksforgeeks.org/longest-path-between-any-pair-of-vertices/>

Chapter 193

Longest Path in a Directed Acyclic Graph Set 2

Longest Path in a Directed Acyclic Graph Set 2 - GeeksforGeeks

Given a Weighted Directed Acyclic Graph (DAG) and a source vertex in it, find the longest distances from source vertex to all other vertices in the given graph.

We have already discussed how we can find [Longest Path in Directed Acyclic Graph\(DAG\)](#) in Set 1. In this post, we will discuss another interesting solution to find longest path of DAG that uses algorithm for finding [Shortest Path in a DAG](#).

The idea is to **negate the weights of the path and find the shortest path in the graph**. A longest path between two given vertices s and t in a weighted graph G is the same thing as a shortest path in a graph G' derived from G by changing every weight to its negation. Therefore, if shortest paths can be found in G', then longest paths can also be found in G.

Below is the step by step process of finding longest paths –

We change weight of every edge of given graph to its negation and initialize distances to all vertices as infinite and distance to source as 0, then we find a topological sorting of the graph which represents a linear ordering of the graph. When we consider a vertex u in topological order, it is guaranteed that we have considered every incoming edge to it. i.e. We have already found shortest path to that vertex and we can use that info to update shorter path of all its adjacent vertices. Once we have topological order, we one by one process all vertices in topological order. For every vertex being processed, we update distances of its adjacent vertex using shortest distance of current vertex from source vertex and its edge weight. i.e.

```
for every adjacent vertex v of every vertex u in topological order
    if (dist[v] > dist[u] + weight(u, v))
        dist[v] = dist[u] + weight(u, v)
```

Once we have found all shortest paths from the source vertex, longest paths will be just negation of shortest paths.

Below is its C++ implementation –

```
// A C++ program to find single source longest distances
// in a DAG
#include <bits/stdc++.h>
using namespace std;

// Graph is represented using adjacency list. Every node of
// adjacency list contains vertex number of the vertex to
// which edge connects. It also contains weight of the edge
class AdjListNode
{
    int v;
    int weight;
public:
    AdjListNode(int _v, int _w)
    {
        v = _v;
        weight = _w;
    }
    int getV()
    {
        return v;
    }
    int getWeight()
    {
        return weight;
    }
};

// Graph class represents a directed graph using adjacency
// list representation
class Graph
{
    int V; // No. of vertices

    // Pointer to an array containing adjacency lists
    list<AdjListNode*>* adj;

    // This function uses DFS
    void longestPathUtil(int, vector<bool> &, stack<int> &);

public:
    Graph(int); // Constructor
    ~Graph(); // Destructor

    // function to add an edge to graph
```

```
void addEdge(int, int, int);

void longestPath(int);
};

Graph::Graph(int V) // Constructor
{
    this->V = V;
    adj = new list<AdjListNode>[V];
}

Graph::~Graph() // Destructor
{
    delete[] adj;
}

void Graph::addEdge(int u, int v, int weight)
{
    AdjListNode node(v, weight);
    adj[u].push_back(node); // Add v to u's list
}

// A recursive function used by longestPath. See below
// link for details.
// https://www.geeksforgeeks.org/topological-sorting/
void Graph::longestPathUtil(int v, vector<bool> &visited,
                           stack<int> &Stack)
{
    // Mark the current node as visited
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    for (AdjListNode node : adj[v])
    {
        if (!visited[node.getV()])
            longestPathUtil(node.getV(), visited, Stack);
    }

    // Push current vertex to stack which stores topological
    // sort
    Stack.push(v);
}

// The function do Topological Sort and finds longest
// distances from given source vertex
void Graph::longestPath(int s)
{
    // Initialize distances to all vertices as infinite and
```

```

// distance to source as 0
int dist[V];
for (int i = 0; i < V; i++)
    dist[i] = INT_MAX;
dist[s] = 0;

stack<int> Stack;

// Mark all the vertices as not visited
vector<bool> visited(V, false);

for (int i = 0; i < V; i++)
    if (visited[i] == false)
        longestPathUtil(i, visited, Stack);

// Process vertices in topological order
while (!Stack.empty())
{
    // Get the next vertex from topological order
    int u = Stack.top();
    Stack.pop();

    if (dist[u] != INT_MAX)
    {
        // Update distances of all adjacent vertices
        // (edge from u -> v exists)
        for (AdjListNode v : adj[u])
        {
            // consider negative weight of edges and
            // find shortest path
            if (dist[v.getV()] > dist[u] + v.getWeight() * -1)
                dist[v.getV()] = dist[u] + v.getWeight() * -1;
        }
    }
}

// Print the calculated longest distances
for (int i = 0; i < V; i++)
{
    if (dist[i] == INT_MAX)
        cout << "INT_MIN ";
    else
        cout << (dist[i] * -1) << " ";
}
}

// Driver code
int main()

```

```
{  
    Graph g(6);  
  
    g.addEdge(0, 1, 5);  
    g.addEdge(0, 2, 3);  
    g.addEdge(1, 3, 6);  
    g.addEdge(1, 2, 2);  
    g.addEdge(2, 4, 4);  
    g.addEdge(2, 5, 2);  
    g.addEdge(2, 3, 7);  
    g.addEdge(3, 5, 1);  
    g.addEdge(3, 4, -1);  
    g.addEdge(4, 5, -2);  
  
    int s = 1;  
  
    cout << "Following are longest distances from "  
         << "source vertex " << s << "\n";  
    g.longestPath(s);  
  
    return 0;  
}
```

Output:

```
Following are longest distances from source vertex 1  
INT_MIN 0 2 9 8 10
```

Time Complexity: Time complexity of topological sorting is $O(V + E)$. After finding topological order, the algorithm process all vertices and for every vertex, it runs a loop for all adjacent vertices. As total adjacent vertices in a graph is $O(E)$, the inner loop runs $O(V + E)$ times. Therefore, overall time complexity of this algorithm is $O(V + E)$.

Source

<https://www.geeksforgeeks.org/longest-path-directed-acyclic-graph-set-2/>

Chapter 194

m Coloring Problem Backtracking-5

m Coloring Problem Backtracking-5 - GeeksforGeeks

Given an undirected graph and a number m, determine if the graph can be colored with at most m colors such that no two adjacent vertices of the graph are colored with same color. Here coloring of a graph means assignment of colors to all vertices.

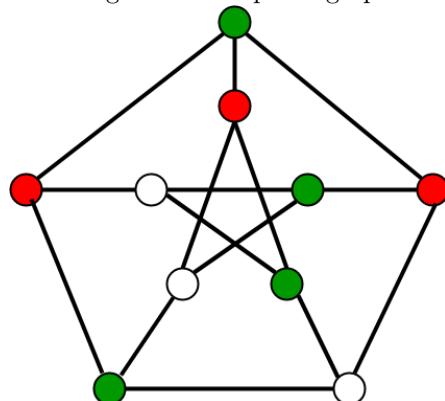
Input:

- 1) A 2D array $\text{graph}[V][V]$ where V is the number of vertices in graph and $\text{graph}[V][V]$ is adjacency matrix representation of the graph. A value $\text{graph}[i][j]$ is 1 if there is a direct edge from i to j, otherwise $\text{graph}[i][j]$ is 0.
- 2) An integer m which is maximum number of colors that can be used.

Output:

An array $\text{color}[V]$ that should have numbers from 1 to m. $\text{color}[i]$ should represent the color assigned to the ith vertex. The code should also return false if the graph cannot be colored with m colors.

Following is an example of graph that can be colored with 3 different colors.



Naive Algorithm

Generate all possible configurations of colors and print a configuration that satisfies the given constraints.

```
while there are untried conflagrations
{
    generate the next configuration
    if no adjacent vertices are colored with same color
    {
        print this configuration;
    }
}
```

There will be V^m configurations of colors.

Backtracking Algorithm

The idea is to assign colors one by one to different vertices, starting from the vertex 0. Before assigning a color, we check for safety by considering already assigned colors to the adjacent vertices. If we find a color assignment which is safe, we mark the color assignment as part of solution. If we do not find a color due to clashes then we backtrack and return false.

Implementation of Backtracking solution

C/C++

```
#include<stdio.h>

// Number of vertices in the graph
#define V 4

void printSolution(int color[]);

/* A utility function to check if the current color assignment
is safe for vertex v i.e. checks whether the edge exists or not
(i.e., graph[v][i]==1). If exist then checks whether the color to
be filled in the new vertex(c is sent in the parameter) is already
used by its adjacent vertices(i-->adj vertices) or not (i.e., color[i]==c) */
bool isSafe (int v, bool graph[V][V], int color[], int c)
{
    for (int i = 0; i < V; i++)
        if (graph[v][i] && c == color[i])
            return false;
    return true;
}

/* A recursive utility function to solve m coloring problem */
bool graphColoringUtil(bool graph[V][V], int m, int color[], int v)
```

```

{
    /* base case: If all vertices are assigned a color then
       return true */
    if (v == V)
        return true;

    /* Consider this vertex v and try different colors */
    for (int c = 1; c <= m; c++)
    {
        /* Check if assignment of color c to v is fine*/
        if (isSafe(v, graph, color, c))
        {
            color[v] = c;

            /* recur to assign colors to rest of the vertices */
            if (graphColoringUtil (graph, m, color, v+1) == true)
                return true;

            /* If assigning color c doesn't lead to a solution
               then remove it */
            color[v] = 0;
        }
    }

    /* If no color can be assigned to this vertex then return false */
    return false;
}

/* This function solves the m Coloring problem using Backtracking.
   It mainly uses graphColoringUtil() to solve the problem. It returns
   false if the m colors cannot be assigned, otherwise return true and
   prints assignments of colors to all vertices. Please note that there
   may be more than one solutions, this function prints one of the
   feasible solutions.*/
bool graphColoring(bool graph[V][V], int m)
{
    // Initialize all color values as 0. This initialization is needed
    // correct functioning of isSafe()
    int *color = new int[V];
    for (int i = 0; i < V; i++)
        color[i] = 0;

    // Call graphColoringUtil() for vertex 0
    if (graphColoringUtil(graph, m, color, 0) == false)
    {
        printf("Solution does not exist");
        return false;
    }
}

```

```
// Print the solution
printSolution(color);
return true;
}

/* A utility function to print solution */
void printSolution(int color[])
{
    printf("Solution Exists:\n"
           " Following are the assigned colors \n");
    for (int i = 0; i < V; i++)
        printf(" %d ", color[i]);
    printf("\n");
}

// driver program to test above function
int main()
{
    /* Create following graph and test whether it is 3 colorable
       (3)---(2)
          |   / |
          |   / |
          | /   |
       (0)---(1)
    */
    bool graph[V][V] = {{0, 1, 1, 1},
                        {1, 0, 1, 0},
                        {1, 1, 0, 1},
                        {1, 0, 1, 0},
    };
    int m = 3; // Number of colors
    graphColoring (graph, m);
    return 0;
}
```

Java

```
/* Java program for solution of M Coloring problem
   using backtracking */
public class mColoringProblem {
    final int V = 4;
    int color[];

    /* A utility function to check if the current
       color assignment is safe for vertex v */
    boolean isSafe(int v, int graph[][] , int color[],
                  int c)
```

```

{
    for (int i = 0; i < V; i++)
        if (graph[v][i] == 1 && c == color[i])
            return false;
    return true;
}

/* A recursive utility function to solve m
   coloring problem */
boolean graphColoringUtil(int graph[][] , int m,
                           int color[], int v)
{
    /* base case: If all vertices are assigned
       a color then return true */
    if (v == V)
        return true;

    /* Consider this vertex v and try different
       colors */
    for (int c = 1; c <= m; c++)
    {
        /* Check if assignment of color c to v
           is fine*/
        if (isSafe(v, graph, color, c))
        {
            color[v] = c;

            /* recur to assign colors to rest
               of the vertices */
            if (graphColoringUtil(graph, m,
                                  color, v + 1))
                return true;

            /* If assigning color c doesn't lead
               to a solution then remove it */
            color[v] = 0;
        }
    }

    /* If no color can be assigned to this vertex
       then return false */
    return false;
}

/* This function solves the m Coloring problem using
   Backtracking. It mainly uses graphColoringUtil()
   to solve the problem. It returns false if the m
   colors cannot be assigned, otherwise return true
}

```

```

and prints assignments of colors to all vertices.
Please note that there may be more than one
solutions, this function prints one of the
feasible solutions.*/
boolean graphColoring(int graph[][] , int m)
{
    // Initialize all color values as 0. This
    // initialization is needed correct functioning
    // of isSafe()
    color = new int[V];
    for (int i = 0; i < V; i++)
        color[i] = 0;

    // Call graphColoringUtil() for vertex 0
    if (!graphColoringUtil(graph, m, color, 0))
    {
        System.out.println("Solution does not exist");
        return false;
    }

    // Print the solution
    printSolution(color);
    return true;
}

/* A utility function to print solution */
void printSolution(int color[])
{
    System.out.println("Solution Exists: Following" +
                       " are the assigned colors");
    for (int i = 0; i < V; i++)
        System.out.print(" " + color[i] + " ");
    System.out.println();
}

// driver program to test above function
public static void main(String args[])
{
    mColoringProblem Coloring = new mColoringProblem();
    /* Create following graph and test whether it is
       3 colorable
       (3)---(2)
          |   /   |
          |   /   |
          |   /   |
       (0)---(1)
    */
    int graph[][] = {{0, 1, 1, 1},

```

```
{1, 0, 1, 0},  
{1, 1, 0, 1},  
{1, 0, 1, 0},  
};  
int m = 3; // Number of colors  
Coloring.graphColoring(graph, m);  
}  
}  
// This code is contributed by Abhishek Shankhadhar
```

Python

```
# Python program for solution of M Coloring  
# problem using backtracking  
  
class Graph():  
  
    def __init__(self, vertices):  
        self.V = vertices  
        self.graph = [[0 for column in range(vertices)]\  
                     for row in range(vertices)]  
  
        # A utility function to check if the current color assignment  
        # is safe for vertex v  
    def isSafe(self, v, colour, c):  
        for i in range(self.V):  
            if self.graph[v][i] == 1 and colour[i] == c:  
                return False  
        return True  
  
        # A recursive utility function to solve m  
        # coloring problem  
    def graphColourUtil(self, m, colour, v):  
        if v == self.V:  
            return True  
  
        for c in range(1, m+1):  
            if self.isSafe(v, colour, c) == True:  
                colour[v] = c  
                if self.graphColourUtil(m, colour, v+1) == True:  
                    return True  
                colour[v] = 0  
  
    def graphColouring(self, m):  
        colour = [0] * self.V  
        if self.graphColourUtil(m, colour, 0) == False:  
            return False
```

```
# Print the solution
print "Solution exist and Following are the assigned colours:"
for c in colour:
    print c,
return True

# Driver Code
g = Graph(4)
g.graph = [[0,1,1,1], [1,0,1,0], [1,1,0,1], [1,0,1,0]]
m=3
g.graphColouring(m)

# This code is contributed by Divyanshu Mehta
```

Output:

```
Solution Exists: Following are the assigned colors
1 2 3 2
```

References:

http://en.wikipedia.org/wiki/Graph_coloring

Improved By : SarathChandra1

Source

<https://www.geeksforgeeks.org/m-coloring-problem-backtracking-5/>

Chapter 195

Magical Indices in an array

Magical Indices in an array - GeeksforGeeks

Given an array A of integers. Index i of A is said to be connected to index j if $j = (i + A[i]) \% n + 1$ (Assume 1-based indexing). Start traversing array from index i and jump to its next connected index. If on traversing array in the described order, index i is again visited then index i is a magical index. Count the number of magical indexes in the array. Assume that array A consists of non-negative integers.

Examples :

```
Input : A = {1, 1, 1, 1}
Output : 4
Possible traversals:
1 -> 3 -> 1
2 -> 4 -> 2
3 -> 1 -> 3
4 -> 2 -> 4
Clearly all the indices are magical
```

```
Input : A = {0, 0, 0, 2}
Output : 2
Possible traversals:
1 -> 2 -> 3 -> 4 -> 3...
2 -> 3 -> 4 -> 3...
3 -> 4 -> 3
4 -> 3 ->4
Magical indices = 3, 4
```

Approach: The problem is of counting number of nodes in all the cycles present in the graph. Each index represents a single node of the graph. Each node has a single directed edge as described in the problem statement. This graph has a special property: On starting

a traversal from any vertex, a cycle is always detected. This property will be helpful in reducing the time complexity of the solution.

Read this post on how to detect cycle in a directed graph: [Detect Cycle in directed graph](#)

Let the traversal begins from node i. Node i will be called parent node of this traversal and this parent node will be assigned to all the nodes visited during traversal. While traversing the graph if we discover a node that is already visited and parent node of that visited node is same as parent node of the traversal then a new cycle is detected. To count number of nodes in this cycle, start another dfs from this node until this same node is not visited again. This procedure is repeated for every node i of the graph. In worst case every node will be traversed at most 3 times. Hence solution has linear time complexity.

The stepwise algorithm is:

1. For each node in the graph:
 if node i is not visited then:
 for every adjacent node j:
 if node j is not visited:
 par[j] = i
 else:
 if par[j]==i
 cycle detected
 count nodes in cycle
2. return count

Implementation:

C++

```
// C++ program to find number of magical
// indices in the given array.
#include <bits/stdc++.h>
using namespace std;

#define mp make_pair
#define pb push_back
#define mod 1000000007

// Function to count number of magical indices.
int solve(int A[], int n)
{
    int i, cnt = 0, j;

    // Array to store parent node of traversal.
    int parent[n + 1];

    // Array to determine whether current node
    // is already counted in the cycle.
```

```

int vis[n + 1];

// Initialize the arrays.
memset(parent, -1, sizeof(parent));
memset(vis, 0, sizeof(vis));

for (i = 0; i < n; i++) {
    j = i;

    // Check if current node is already
    // traversed or not. If node is not
    // traversed yet then parent value
    // will be -1.
    if (parent[j] == -1) {

        // Traverse the graph until an
        // already visited node is not
        // found.
        while (parent[j] == -1) {
            parent[j] = i;
            j = (j + A[j] + 1) % n;
        }

        // Check parent value to ensure
        // a cycle is present.
        if (parent[j] == i) {

            // Count number of nodes in
            // the cycle.
            while (!vis[j]) {
                vis[j] = 1;
                cnt++;
                j = (j + A[j] + 1) % n;
            }
        }
    }
}

return cnt;
}

int main()
{
    int A[] = { 0, 0, 0, 2 };
    int n = sizeof(A) / sizeof(A[0]);
    cout << solve(A, n);
    return 0;
}

```

Java

```
// Java program to find number of magical
// indices in the given array.
import java.io.*;
import java.util.*;

public class GFG {

    // Function to count number of magical
    // indices.
    static int solve(int []A, int n)
    {
        int i, cnt = 0, j;

        // Array to store parent node of
        // traversal.
        int []parent = new int[n + 1];

        // Array to determine whether current node
        // is already counted in the cycle.
        int []vis = new int[n + 1];

        // Initialize the arrays.
        for (i = 0; i < n+1; i++) {
            parent[i] = -1;
            vis[i] = 0;
        }

        for (i = 0; i < n; i++) {
            j = i;

            // Check if current node is already
            // traversed or not. If node is not
            // traversed yet then parent value
            // will be -1.
            if (parent[j] == -1) {

                // Traverse the graph until an
                // already visited node is not
                // found.
                while (parent[j] == -1) {
                    parent[j] = i;
                    j = (j + A[j] + 1) % n;
                }

                // Check parent value to ensure
            }
        }
    }
}
```

```
// a cycle is present.
if (parent[j] == i) {

    // Count number of nodes in
    // the cycle.
    while (vis[j]==0) {
        vis[j] = 1;
        cnt++;
        j = (j + A[j] + 1) % n;
    }
}

return cnt;
}

// Driver code
public static void main(String args[])
{
    int []A = { 0, 0, 0, 2 };
    int n = A.length;
    System.out.print(solve(A, n));
}
}

// This code is contributed by Manish Shaw
// (manishshaw1)
```

Python3

```
# Python3 program to find number of magical
# indices in the given array.

# Function to count number of magical
# indices.
def solve(A, n) :

    cnt = 0

    # Array to store parent node of
    # traversal.
    parent = [None] * (n + 1)

    # Array to determine whether current node
    # is already counted in the cycle.
    vis = [None] * (n + 1)
```

```
# Initialize the arrays.
for i in range(0, n+1):
    parent[i] = -1
    vis[i] = 0

for i in range(0, n):
    j = i

    # Check if current node is already
    # traversed or not. If node is not
    # traversed yet then parent value
    # will be -1.
    if (parent[j] == -1) :

        # Traverse the graph until an
        # already visited node is not
        # found.
        while (parent[j] == -1) :
            parent[j] = i
            j = (j + A[j] + 1) % n

        # Check parent value to ensure
        # a cycle is present.
        if (parent[j] == i) :

            # Count number of nodes in
            # the cycle.
            while (vis[j]==0) :
                vis[j] = 1
                cnt = cnt + 1
                j = (j + A[j] + 1) % n
return cnt

# Driver code
A = [ 0, 0, 0, 2 ]
n = len(A)
print (solve(A, n))

# This code is contributed by Manish Shaw
# (manishshaw1)
```

C#

```
// C# program to find number of magical
// indices in the given array.
using System;
using System.Collections.Generic;
using System.Linq;
```

```
class GFG {

    // Function to count number of magical
    // indices.
    static int solve(int []A, int n)
    {
        int i, cnt = 0, j;

        // Array to store parent node of
        // traversal.
        int []parent = new int[n + 1];

        // Array to determine whether current node
        // is already counted in the cycle.
        int []vis = new int[n + 1];

        // Initialize the arrays.
        for (i = 0; i < n+1; i++) {
            parent[i] = -1;
            vis[i] = 0;
        }

        for (i = 0; i < n; i++) {
            j = i;

            // Check if current node is already
            // traversed or not. If node is not
            // traversed yet then parent value
            // will be -1.
            if (parent[j] == -1) {

                // Traverse the graph until an
                // already visited node is not
                // found.
                while (parent[j] == -1) {
                    parent[j] = i;
                    j = (j + A[j] + 1) % n;
                }

                // Check parent value to ensure
                // a cycle is present.
                if (parent[j] == i) {

                    // Count number of nodes in
                    // the cycle.
                    while (vis[j]==0) {
```

```
        vis[j] = 1;
        cnt++;
        j = (j + A[j] + 1) % n;
    }
}
}

return cnt;
}

// Driver code
public static void Main()
{
    int []A = { 0, 0, 0, 2 };
    int n = A.Length;
    Console.WriteLine(solve(A, n));
}
}

// This code is contributed by Manish Shaw
// (manishshaw1)
```

PHP

```
<?php
// PHP program to find
// number of magical
// indices in the given
// array.

// Function to count number
// of magical indices.
function solve($A, $n)
{
    $i = 0;
    $cnt = 0;
    $j = 0;

    // Array to store parent
    // node of traversal.
    $parent = array();

    // Array to determine
    // whether current node
    // is already counted
    // in the cycle.
    $vis = array();
```

```
// Initialize the arrays.  
for ($i = 0; $i < $n + 1; $i++)  
{  
    $parent[$i] = -1;  
    $vis[$i] = 0;  
}  
  
for ($i = 0; $i < $n; $i++)  
{  
    $j = $i;  
  
    // Check if current node is  
    // already traversed or not.  
    // If node is not traversed  
    // yet then parent value will  
    // be -1.  
    if ($parent[$j] == -1)  
    {  
  
        // Traverse the graph until  
        // an already visited node  
        // is not found.  
        while ($parent[$j] == -1)  
        {  
            $parent[$j] = $i;  
            $j = ($j + $A[$j] + 1) % $n;  
        }  
  
        // Check parent value to ensure  
        // a cycle is present.  
        if ($parent[$j] == $i)  
        {  
  
            // Count number of  
            // nodes in the cycle.  
            while ($vis[$j] == 0)  
            {  
                $vis[$j] = 1;  
                $cnt++;  
                $j = ($j + $A[$j] + 1) % $n;  
            }  
        }  
    }  
}  
  
return $cnt;
```

```
}

// Driver code
$A = array( 0, 0, 0, 2 );
$n = count($A);
echo (solve($A, $n));

// This code is contributed by
// Manish Shaw (manishshaw1)
?>
```

Output :

2

Time Complexity: O(n)

Space Complexity: O(n)

Improved By : [manishshaw1](#)

Source

<https://www.geeksforgeeks.org/magical-indices-array/>

Chapter 196

Mathematics Graph theory practice questions

Mathematics Graph theory practice questions - GeeksforGeeks

Problem 1 – There are 25 telephones in Geeksland. Is it possible to connect them with wires so that each telephone is connected with exactly 7 others.

Solution – Let us suppose that such an arrangement is possible. This can be viewed as a graph in which telephones are represented using vertices and wires using the edges. Now we have 25 vertices in this graph. The degree of each vertex in the graph is 7.

From [handshaking lemma](#), we know.

$$\begin{aligned}\text{sum of degrees of all vertices} &= 2 * (\text{number of edges}) \\ \text{number of edges} &= (\text{sum of degrees of all vertices}) / 2\end{aligned}$$

We need to understand that an edge connects two vertices. So the sum of degrees of all the vertices is equal to twice the number of edges.

Therefore,

$$\begin{aligned}25 * 7 &= 2 * (\text{number of edges}) \\ \text{number of edges} &= 25 * 7 / 2 = 87.5\end{aligned}$$

This is not an integer. As a result we can conclude that our supposition is wrong and such an arrangement is not possible.

Problem 2 – The figure below shows an arrangement of knights on a 3*3 grid.

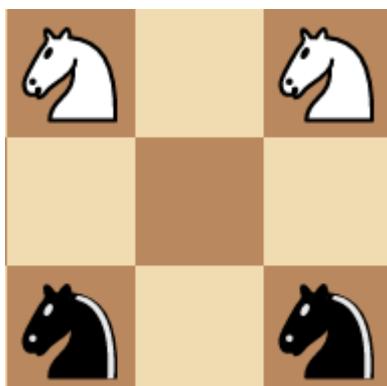


Figure – initial state

Is it possible to reach the final state as shown below using valid knight's moves ?

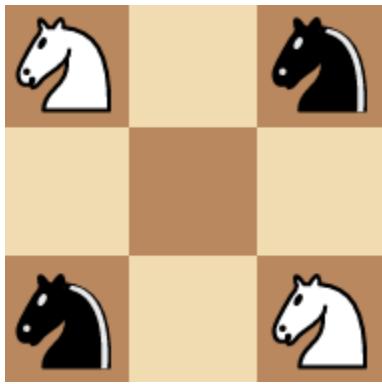


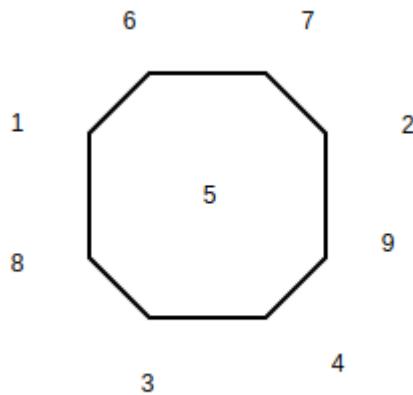
Figure – final state

Solution – NO. You might think you need to be a good chess player in order to crack the above question. However the above question can be solved using graphs. But what kind of a graph should we draw? Let each of the 9 vertices be represented by a number as shown below.

1	4	7
2	5	8
3	6	9

Now we consider each square of the grid as a vertex in our graph. There exists an edge between two vertices in our graph if a valid knight's move is possible between the corresponding squares in the graph. For example – If we consider square 1. The reachable squares with valid knight's moves are 6 and 8. We can say that vertex 1 is connected to vertices 6 and 8 in our graph.

Similarly we can draw the entire graph as shown below. Clearly vertex 5 can't be reached from any of the squares. Hence none of the edges connect to vertex 5.



We use a hollow circle to depict a white knight in our graph and a filled circle to depict a black knight. Hence the initial state of the graph can be represented as :

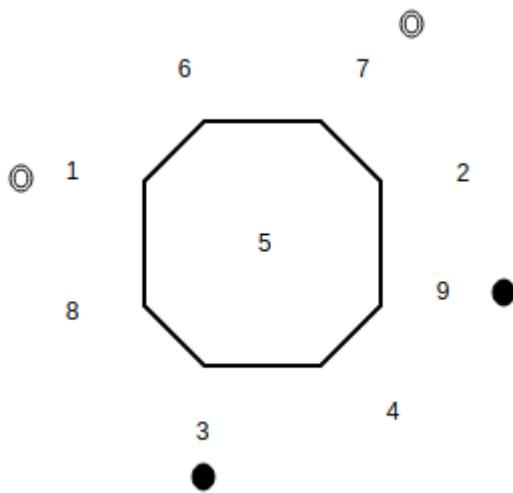


Figure – initial state

The final state is represented as :

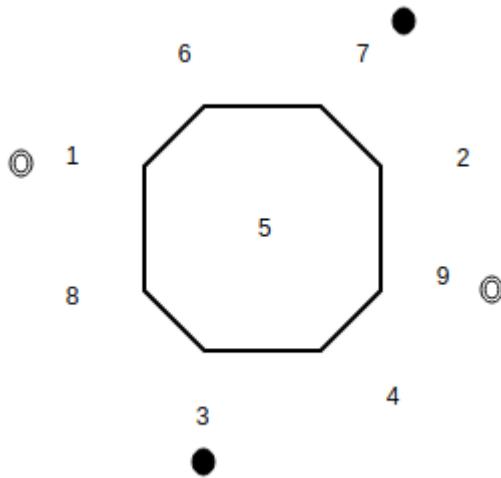


Figure – final state

Note that in order to achieve the final state there needs to exist a path where two knights (a black knight and a white knight cross-over). We can only move the knights in a clockwise or counter-clockwise manner on the graph (If two vertices are connected on the graph: it means that a corresponding knight's move exists on the grid). However the order in which knights appear on the graph cannot be changed. There is no possible way for a knight to cross over (Two knights cannot exist on one vertex) the other in order to achieve the final state. Hence, we can conclude that no matter what the final arrangement is not possible.

Problem 3: There are 9 line segments drawn in a plane. Is it possible that each line

segment intersects exactly 3 others?

Solution: This problem seems very difficult initially. We could think of solving it using graphs. But how do we draw the graph. If we try to approach this problem by using line segments as edges of a graph, we seem to reach nowhere (This sounds confusing initially). Here we need to consider a graph where each line segment is represented as a vertex. Now two vertices of this graph are connected if the corresponding line segments intersect.

Now this graph has 9 vertices. The degree of each vertex is 3.

We know that for a graph

Sum of degrees of all vertices = $2 * \text{Number of Edges in the graph}$

Since the sum of degrees of vertices in the above problem is $9 * 3 = 27$ i.e odd, such an arrangement is not possible.

Improved By : [PavanBansal](#)

Source

<https://www.geeksforgeeks.org/graph-theory-practice-questions/>

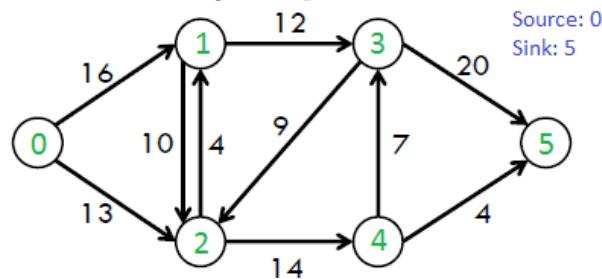
Chapter 197

Max Flow Problem Introduction

Max Flow Problem Introduction - GeeksforGeeks

Maximum flow problems involve finding a feasible flow through a single-source, single-sink flow network that is maximum.

Let's take an image to explain how above definition wants to say.



Each edge is labeled with a capacity, the maximum amount of stuff that it can carry. The goal is to figure out how much stuff can be pushed from the vertex s(source) to the vertex t(sink).

maximum flow possible is : **23**

Following are different approaches to solve the problem :

1. Naive Greedy Algorithm Approach (May not produce optimal or correct result)

Greedy approach to the maximum flow problem is to start with the all-zero flow and greedily produce flows with ever-higher value. The natural way to proceed from one to the next is to send more flow on some path from s to t

How Greedy approach work to find the maximum flow :

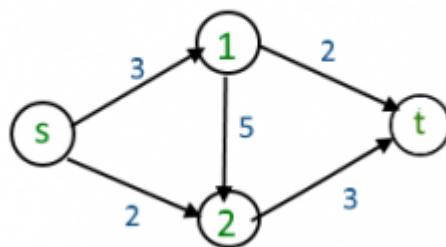
E number of edge

$f(e)$ flow of edge
 $C(e)$ capacity of edge

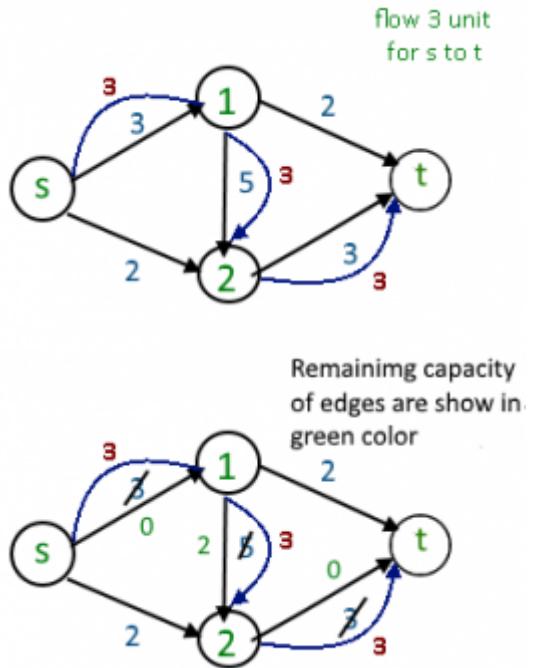
- 1) Initialize : $\text{max_flow} = 0$
 $f(e) = 0$ for every edge 'e' in E
- 2) Repeat search for an s-t path P while it exists.
 - a) Find if there is a path from s to t using BFS or DFS. A path exists if $f(e) < C(e)$ for every edge e on the path.
 - b) If no path found, return max_flow .
 - c) Else find minimum edge value for path P


```
// Our flow is limited by least remaining
// capacity edge on path P.
(i) flow = min(C(e)- f(e)) for path P ]
    max_flow += flow
(ii) For all edge e of path increment flow
    f(e) += flow
```
- 3) Return max_flow

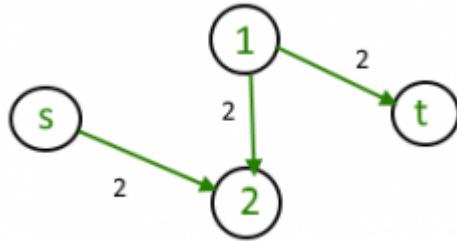
Note that the path search just needs to determine whether or not there is an s-t path in the subgraph of edges e with $f(e) < C(e)$. This is easily done in linear time using BFS or DFS.



There is a path from source (s) to sink(t) [$s \rightarrow 1 \rightarrow 2 \rightarrow t$] with maximum flow 3 unit (path show in blue color)



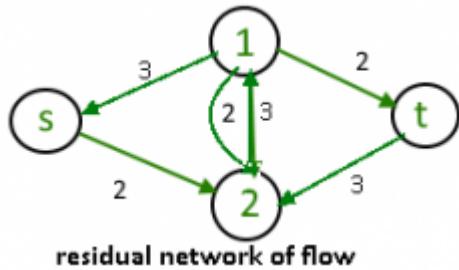
After removing all useless edge from graph it's look like



For above graph there is no path from source to sink so maximum flow : 3 unit But maximum flow is 5 unit. to over come form this issue we use residual Graph.

2. Residual Graphs

The idea is to extend the naive greedy algorithm by allowing “undo” operations. For example, from the point where this algorithm gets stuck in above image, we’d like to route two more units of flow along the edge $(s, 2)$, then backward along the edge $(1, 2)$, undoing 2 of the 3 units we routed the previous iteration, and finally along the edge $(1,t)$.



backward edge : ($f(e)$) and forward edge : ($C(e) - f(e)$)

We need a way of formally specifying the allowable “undo” operations. This motivates the following simple but important definition, of a residual network. The idea is that, given a graph G and a flow f in it, we form a new flow network G_f that has the same vertex set of G and that has two edges for each edge of G . An edge $e = (1,2)$ of G that carries flow $f(e)$ and has capacity $C(e)$ (for above image) spawns a “forward edge” of G_f with capacity $C(e)-f(e)$ (the room remaining) and a “backward edge” $(2,1)$ of G_f with capacity $f(e)$ (the amount of previously routed flow that can be undone). source(s)- sink(t) paths with $f(e) < C(e)$ for all edges, as searched for by the naive greedy algorithm, correspond to the special case of $s-t$ paths of G_f that comprise only forward edges.

The idea of residual graph is used [The Ford-Fulkerson](#) and Dinic’s algorithms

Source :

<http://theory.stanford.edu/~tim/w16/l/l1.pdf>

Source

<https://www.geeksforgeeks.org/max-flow-problem-introduction/>

Chapter 198

Maximum edges that can be added to DAG so that it remains DAG

Maximum edges that can be added to DAG so that it remains DAG - GeeksforGeeks

A DAG is given to us, we need to find maximum number of edges that can be added to this DAG, after which new graph still remain a DAG that means the reformed graph should have maximal number of edges, adding even single edge will create a cycle in graph.

The Output for above example should be following edges in any order.
4-2, 4-5, 4-3, 5-3, 5-1, 2-0, 2-1, 0-3, 0-1

As shown in above example, we have added all the edges in one direction only to save ourselves from making a cycle. This is the trick to solve this question. We sort all our nodes in [topological order](#) and create edges from node to all nodes to the right if not there already. How can we say that, it is not possible to add any more edge? the reason is we have added all possible edges from left to right and if we want to add more edge we need to make that from right to left, but adding edge from right to left we surely create a cycle because its counter part left to right edge is already been added to graph and creating cycle is not what we needed.

So solution proceeds as follows, we consider the nodes in topological order and if any edge is not there from left to right, we will create it.

Below is the solution, we have printed all the edges that can be added to given DAG without making any cycle.

```
// C++ program to find maximum edges after adding
```

```
// which graph still remains a DAG
#include <bits/stdc++.h>
using namespace std;

class Graph
{
    int V;      // No. of vertices

    // Pointer to a list containing adjacency list
    list<int> *adj;

    // Vector to store indegree of vertices
    vector<int> indegree;

    // function returns a topological sort
    vector<int> topologicalSort();

public:
    Graph(int V);    // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // Prints all edges that can be added without making any cycle
    void maximumEdgeAddtion();
};

// Constructor of graph
Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];

    // Initialising all indegree with 0
    for (int i = 0; i < V; i++)
        indegree.push_back(0);
}

// Utility function to add edge
void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.

    // increasing inner degree of w by 1
    indegree[w]++;
}

// Main function to print maximum edges that can be added
```

```
vector<int> Graph::topologicalSort()
{
    vector<int> topological;
    queue<int> q;

    // In starting push all node with indegree 0
    for (int i = 0; i < V; i++)
        if (indegree[i] == 0)
            q.push(i);

    while (!q.empty())
    {
        int t = q.front();
        q.pop();

        // push the node into topological vector
        topological.push_back(t);

        // reducing indegree of adjacent vertices
        for (list<int>:: iterator j = adj[t].begin();
                j != adj[t].end(); j++)
        {
            indegree[*j]--;
            // if indegree becomes 0, just push
            // into queue
            if (indegree[*j] == 0)
                q.push(*j);
        }
    }
    return topological;
}

// The function prints all edges that can be
// added without making any cycle
// It uses recursive topologicalSort()
void Graph::maximumEdgeAddtion()
{
    bool *visited = new bool[V];
    vector<int> topo = topologicalSort();

    // looping for all nodes
    for (int i = 0; i < topo.size(); i++)
    {
        int t = topo[i];
```

```
// In below loop we mark the adjacent node of t
for (list<int>::iterator j = adj[t].begin();
     j != adj[t].end(); j++)
    visited[*j] = true;

// In below loop unmarked nodes are printed
for (int j = i + 1; j < topo.size(); j++)
{
    // if not marked, then we can make an edge
    // between t and j
    if (!visited[topo[j]])
        cout << t << "-" << topo[j] << " ";

    visited[topo[j]] = false;
}
}

// Driver code to test above methods
int main()
{
    // Create a graph given in the above diagram
    Graph g(6);
    g.addEdge(5, 2);
    g.addEdge(5, 0);
    g.addEdge(4, 0);
    g.addEdge(4, 1);
    g.addEdge(2, 3);
    g.addEdge(3, 1);

    g.maximumEdgeAddtion();
    return 0;
}
```

Output:

4-5, 4-2, 4-3, 5-3, 5-1, 2-0, 2-1, 0-3, 0-1

Source

<https://www.geeksforgeeks.org/maximum-edges-can-added-dag-remains-dag/>

Chapter 199

Maximum number of edges to be added to a tree so that it stays a Bipartite graph

Maximum number of edges to be added to a tree so that it stays a Bipartite graph - Geeks-forGeeks

A tree is always a [Bipartite Graph](#) as we can always break into two disjoint sets with alternate levels. In other words we always color it with two colors such that alternate levels have same color. The task is to compute the maximum no. of edges that can be added to the tree so that it remains Bipartite Graph.

Examples:

Input : Tree edges as vertex pairs
1 2
1 3
Output : 0
Explanation :
The only edge we can add is from node 2 to 3.
But edge 2, 3 will result in odd cycle, hence
violation of Bipartite Graph property.

Input : Tree edges as vertex pairs
1 2
1 3
2 4
3 5
Output : 2
Explanation : On colouring the graph, {1, 4, 5}
and {2, 3} form two different sets. Since, 1 is

connected from both 2 and 3, we are left with edges 4 and 5. Since, 4 is already connected to 2 and 5 to 3, only options remain {4, 3} and {5, 2}.

- 1) Do a simple [DFS](#) (or [BFS](#)) traversal of graph and color it with two colors.
- 2) While coloring also keep track of counts of nodes colored with the two colors. Let the two counts be `count_color0` and `count_color1`.
- 3) Now we know maximum edges a bipartite graph can have are `count_color0 x count_color1`.
- 4) We also know tree with n nodes has n-1 edges.
- 5) So our answer is `count_color0 x count_color1 - (n-1)`.

Below is the CPP implementation :

```
// CPP code to print maximum edges such that
// Tree remains a Bipartite graph
#include <bits/stdc++.h>
using namespace std;

// To store counts of nodes with two colors
long long count_color[2];

void dfs(vector<int> adj[], int node, int parent, int color)
{
    // Increment count of nodes with current
    // color
    count_color[color]++;
    
    // Traversing adjacent nodes
    for (int i = 0; i < adj[node].size(); i++) {
        
        // Not recurring for the parent node
        if (adj[node][i] != parent)
            dfs(adj, adj[node][i], node, !color);
    }
}

// Finds maximum number of edges that can be added
// without violating Bipartite property.
int findMaxEdges(vector<int> adj[], int n)
{
    // Do a DFS to count number of nodes
    // of each color
    dfs(adj, 1, 0, 0);

    return count_color[0] * count_color[1] - (n - 1);
}
```

```
// Driver code
int main()
{
    int n = 5;
    vector<int> adj[n + 1];
    adj[1].push_back(2);
    adj[1].push_back(3);
    adj[2].push_back(4);
    adj[3].push_back(5);
    cout << findMaxEdges(adj, n);
    return 0;
}
```

Output:

2

Time Complexity : $O(n)$

Source

<https://www.geeksforgeeks.org/maximum-number-edges-added-tree-stays-bipartite-graph/>

Chapter 200

Maximum product of two non-intersecting paths in a tree

Maximum product of two non-intersecting paths in a tree - GeeksforGeeks

Given an undirected connected tree with N nodes (and N-1 edges), we need to find two paths in this tree such that they are non-intersecting and the product of their length is maximum.

Examples:

In first tree two paths which are non-intersecting and have highest product are, 1-2 and 3-4, so answer is $1*1 = 1$

In second tree two paths which are non-intersecting and has highest product are, 1-3-5 and 7-8-6-2 (or 4-8-6-2), so answer is $3*2 = 6$

We can solve this problem by depth first search of tree by proceeding as follows, Since tree is connected and paths are non-intersecting, If we take any pair of such paths there must be a third path, connecting these two and If we remove an edge from the third path then tree will be divided into two components — one containing the first path, and the other containing the second path. This observation suggests us the algorithm: iterate over the edges; for each edge remove it, find the length of the path in both connected components and multiply the lengths of these paths. The length of the path in a tree can be found by modified depth first search where we will call for maximum path at each neighbor and we will add two maximum lengths returned, which will be the maximum path length at subtree rooted at current node.

Implementation Details:

Input is a tree, but there is no specified root in it as we have only collection of edges. The tree is represented as undirected graph. We traverse adjacency list. For every edge, we find maximum length paths on both sides of it (after removing the edge). We keep track of maximum product caused by an edge removal.

```
// C++ program to find maximum product of two
// non-intersecting paths
#include <bits/stdc++.h>
using namespace std;

/* Returns maximum length path in subtree rooted
   at u after removing edge connecting u and v */
int dfs(vector<int> g[], int& curMax, int u, int v)
{
    // To find lengths of first and second maximum
    // in subtrees. currMax is to store overall
    // maximum.
    int max1 = 0, max2 = 0, total = 0;

    // loop through all neighbors of u
    for (int i = 0; i < g[u].size(); i++)
    {
        // if neighbor is v, then skip it
        if (g[u][i] == v)
            continue;

        // call recursively with current neighbor as root
        total = max(total, dfs(g, curMax, g[u][i], u));

        // get max from one side and update
        if (curMax > max1)
        {
            max2 = max1;
            max1 = curMax;
        }
        else
            max2 = max(max2, curMax);
    }

    // store total length by adding max
    // and second max
    total = max(total, max1 + max2);

    // update current max by adding 1, i.e.
    // current node is included
    curMax = max1 + 1;
    return total;
}

// method returns maximum product of length of
// two non-intersecting paths
int maxProductOfTwoPaths(vector<int> g[], int N)
{
```

```

int res = INT_MIN;
int path1, path2;

// one by one removing all edges and calling
// dfs on both subtrees
for (int i = 0; i < N; i++)
{
    for (int j = 0; j < g[i].size(); j++)
    {
        // calling dfs on subtree rooted at
        // g[i][j], excluding edge from g[i][j]
        // to i.
        int curMax = 0;
        path1 = dfs(g, curMax, g[i][j], i);

        // calling dfs on subtree rooted at
        // i, edge from i to g[i][j]
        curMax = 0;
        path2 = dfs(g, curMax, i, g[i][j]);

        res = max(res, path1 * path2);
    }
}
return res;
}

// Utility function to add an undirected edge (u,v)
void addEdge(vector<int> g[], int u, int v)
{
    g[u].push_back(v);
    g[v].push_back(u);
}

// Driver code to test above methods
int main()
{
    int edges[] [2] = {{1, 8}, {2, 6}, {3, 1},
                      {5, 3}, {7, 8}, {8, 4},
                      {8, 6}};
    int N = sizeof(edges)/sizeof(edges[0]);

    // there are N edges, so +1 for nodes and +1
    // for 1-based indexing
    vector<int> g[N + 2];
    for (int i = 0; i < N; i++)
        addEdge(g, edges[i][0], edges[i][1]);

    cout << maxProductOfTwoPaths(g, N) << endl;
}

```

```
    return 0;  
}
```

Output:

6

Source

<https://www.geeksforgeeks.org/maximum-product-of-two-non-intersecting-paths-in-a-tree/>

Chapter 201

Minimize the number of weakly connected nodes

Minimize the number of weakly connected nodes - GeeksforGeeks

Given an **undirected graph**, task is to find the minimum number of weakly connected nodes after converting this graph into directed one.

Weakly Connected Nodes : Nodes which are having 0 indegree(number of incoming edges).

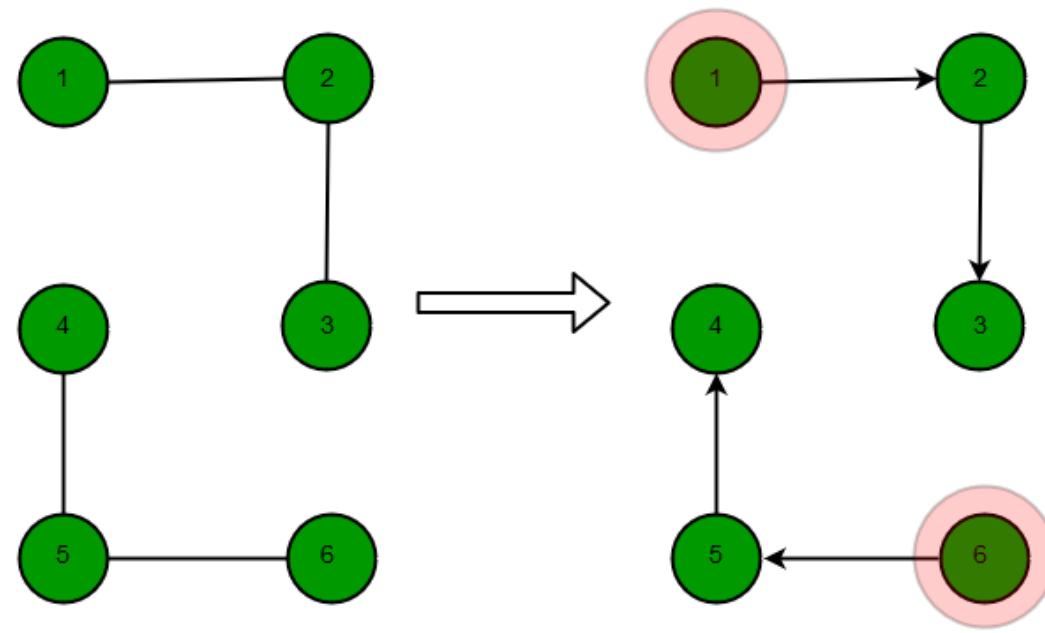
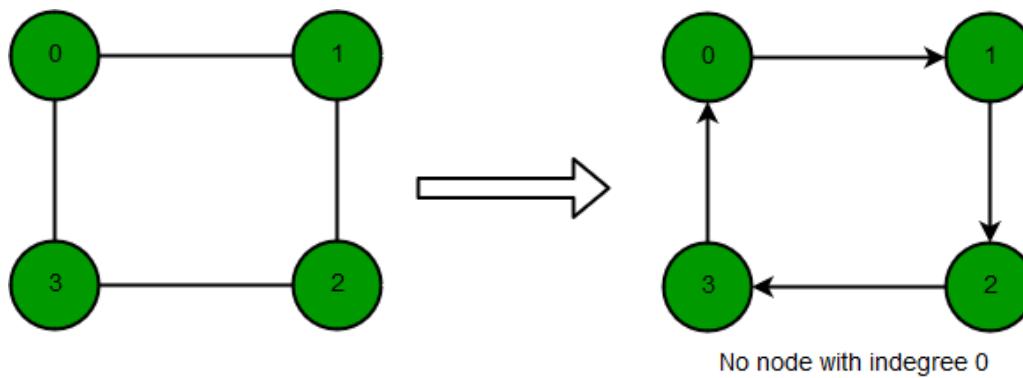
Prerequisite : [BFS traversal](#)

Examples :

```
Input : 4 4
        0 1
        1 2
        2 3
        3 0
Output : 0 disconnected components
```

```
Input : 6 5
        1 2
        2 3
        4 5
        4 6
        5 6
Output : 1 disconnected components
```

Explanation :



Approach : We find a node which helps in traversing maximum nodes in a single walk. To cover all possible paths, [DFS graph traversal](#) technique is used for this.

Do the above steps to traverse the graph. Now, iterate through graph again and check which nodes are having 0 indegree.

```
// CPP code to minimize the number
// of weakly connected nodes
#include <bits/stdc++.h>
using namespace std;
```

```
// Set of nodes which are traversed
// in each launch of the DFS
set<int> node;
vector<int> Graph[10001];

// Function traversing the graph using DFS
// approach and updating the set of nodes
void dfs(bool visit[], int src)
{
    visit[src] = true;
    node.insert(src);
    int len = Graph[src].size();
    for (int i = 0; i < len; i++)
        if (!visit[Graph[src][i]])
            dfs(visit, Graph[src][i]);
}

// building a undirected graph
void buildGraph(int x[], int y[], int len){

    for (int i = 0; i < len; i++)
    {
        int p = x[i];
        int q = y[i];
        Graph[p].push_back(q);
        Graph[q].push_back(p);
    }
}

// computes the minimum number of disconnected
// components when a bi-directed graph is
// converted to a undirected graph
int compute(int n)
{
    // Declaring and initializing
    // a visited array
    bool visit[n + 5];
    memset(visit, false, sizeof(visit));
    int number_of_nodes = 0;

    // We check if each node is
    // visited once or not
    for (int i = 0; i < n; i++)
    {
        // We only launch DFS from a
        // node iff it is unvisited.
        if (!visit[i]) {
```

```
// Clearing the set of nodes
// on every relaunch of DFS
node.clear();

// relaunching DFS from an
// unvisited node.
dfs(visit, i);

// iterating over the node set to count the
// number of nodes visited after making the
// graph directed and storing it in the
// variable count. If count / 2 == number
// of nodes - 1, then increment count by 1.
int count = 0;
for (auto it = node.begin(); it != node.end(); ++it)
    count += Graph[*it].size();

count /= 2;
if (count == node.size() - 1)
    number_of_nodes++;
}

return number_of_nodes;
}

//Driver function
int main()
{
    int n = 6, m = 4;
    int x[m + 5] = {1, 1, 4, 4};
    int y[m+5] = {2, 3, 5, 6};

    /*For given x and y above, graph is as below :
     1-----2           4-----5
     |           |
     |           |
     |           |
     3           6

    // Note : This code will work for
    // connected graph also as :
    1-----2
    |       |
    |       |
    |       |
    3-----4----5
*/
}
```

```
// Building graph in the form of a adjacency list  
buildGraph(x, y, n);  
cout << compute(n) << " weakly connected nodes";  
  
return 0;  
}
```

Output:

```
2 weakly connected nodes
```

Source

<https://www.geeksforgeeks.org/convert-undirected-graph-directed-graph-minimize-disconnected-component/>

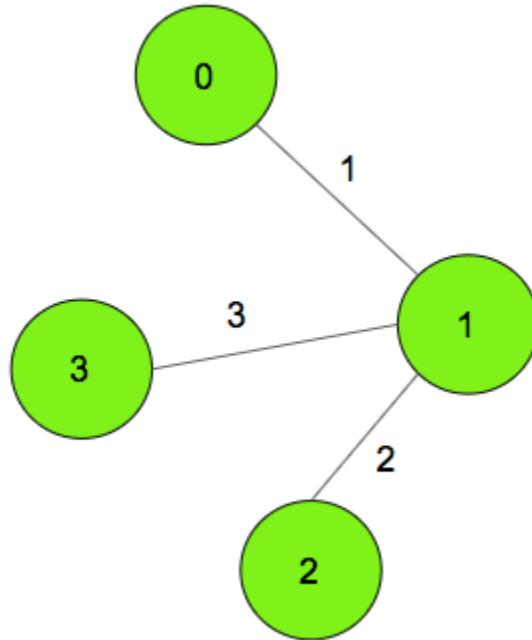
Chapter 202

Minimum cost path from source node to destination node via an intermediate node

Minimum cost path from source node to destination node via an intermediate node - Geeks-forGeeks

Given an undirected weighted graph. The task is to find the minimum cost of the path from source node to the destination node via an intermediate node.

Note: If an edge is traveled twice, only once weight is calculated as cost.



Examples:

Input: source = 0, destination = 2, intermediate = 3;

Output: 6

The minimum cost path 0->1->3->1->2

The edge (1-3) occurs twice in the path, but its weight is added only once to the answer.

Input: source = 0, destination = 2, intermediate = 1;

Output: 3

The minimum cost path is 0->1>2

Approach: Let suppose take a path P1 from Source to intermediate, and a path P2 from intermediate to destination. There can be some common edges among these 2 paths. Hence, the optimal path will always have the following form: for any node U, the walk consists of edges on the shortest path from Source to U, from intermediate to U, and from destination to U. Hence, if $\text{dist}(a, b)$ is the cost of shortest path between node a and b, the required minimum cost path will be $\min\{ \text{dist}(\text{Source}, U) + \text{dist}(\text{intermediate}, U) + \text{dist}(\text{destination}, U) \}$ for all U. The Minimum distance of all nodes from Source, intermediate, and destination can be found by doing [Dijkstra's Shortest Path algorithm](#) from these 3 nodes.

Below is the implementation of the above approach.

```
// CPP program to find minimum distance between  
// source and destination node and visiting
```

```
// of intermediate node is compulsory
#include <bits/stdc++.h>
using namespace std;
#define MAXN 100005

// to strore maped values of graph
vector<pair<int, int> > v[MAXN];

// to store distance of
// all nodes from the source node
int dist[MAXN];

// Dijkstra's algorithm to find
// shortest path from source to node
void dijkstra(int source, int n)
{
    // set all the vertices
    // distances as infinity
    for (int i = 0; i < n; i++)
        dist[i] = INT_MAX;

    // set all vertex as unvisited
    bool vis[n];
    memset(vis, false, sizeof vis);

    // make distance from source
    // vertex to source vertex is zero
    dist = 0;

    // // multiset do the job
    // as a min-priority queue
    multiset<pair<int, int> > s;

    // insert the source node with distance = 0
    s.insert({ 0, source });

    while (!s.empty()) {
        pair<int, int> p = *s.begin();
        // pop the vertex with the minimum distance
        s.erase(s.begin());

        int x = p.second;
        int wei = p.first;

        // check if the popped vertex
        // is visited before
        if (vis[x])
            continue;

        vis[x] = true;

        // update the distances of adjacent vertices
        for (auto it : v[x]) {
            int adjNode = it.first;
            int adjWei = it.second;

            if (dist[adjNode] > dist[x] + adjWei) {
                dist[adjNode] = dist[x] + adjWei;
                s.insert({ dist[adjNode], adjNode });
            }
        }
    }
}
```

```
vis[x] = true;

for (int i = 0; i < v[x].size(); i++) {
    int e = v[x][i].first;
    int w = v[x][i].second;

    // check if the next vertex
    // distance could be minimized
    if (dist[x] + w < dist[e]) {

        dist[e] = dist[x] + w;

        // insert the next vertex
        // with the updated distance
        s.insert({ dist[e], e });
    }
}
}

// function to add edges in graph
void add_edge(int s, int t, int weight)
{
    v[s].push_back({ t, weight });
    v[t].push_back({ s, weight });
}

// function to find the minimum shortest path
int solve(int source, int destination,
          int intermediate, int n)
{
    int ans = INT_MAX;

    dijkstra(source, n);

    // store distance from source to
    // all other vertices
    int dsouce[n];
    for (int i = 0; i < n; i++)
        dsouce[i] = dist[i];

    dijkstra(destination, n);
    // store distance from destination
    // to all other vertices
    int ddestination[n];
    for (int i = 0; i < n; i++)
        ddestination[i] = dist[i];
```

```
dijkstra(intermediate, n);
// store distance from intermediate
// to all other vertices
int dintermediate[n];
for (int i = 0; i < n; i++)
    dintermediate[i] = dist[i];

// find required answer
for (int i = 0; i < n; i++)
    ans = min(ans, dsouce[i] + ddestination[i]
               + dintermediate[i]);

return ans;
}

// Driver code
int main()
{

    int n = 4;
    int source = 0, destination = 2, intermediate = 3;

    // add edges in graph
    add_edge(0, 1, 1);
    add_edge(1, 2, 2);
    add_edge(1, 3, 3);

    // function call for minimum shortest path
    cout << solve(source, destination, intermediate, n);

    return 0;
}
```

Output:

6

Time complexity: $O((N + M) * \log N)$, where N is number of nodes, M is number of edges.

Auxiliary Space: $O(N+M)$

Source

<https://www.geeksforgeeks.org/minimum-cost-path-from-source-node-to-destination-node-via-an-intermediate-node/>

Chapter 203

Minimum Cost Path with Left, Right, Bottom and Up moves allowed

Minimum Cost Path with Left, Right, Bottom and Up moves allowed - GeeksforGeeks

Given a two dimensional grid, each cell of which contains integer cost which represents a cost to traverse through that cell, we need to find a path from top left cell to bottom right cell by which total cost incurred is minimum.

Note : It is assumed that negative cost cycles do not exist in input matrix.

This problem is extension of below problem.

[Min Cost Path with right and bottom moves allowed.](#)

In previous problem only going right and bottom was allowed but in this problem we are allowed to go bottom, up, right and left i.e. in all 4 direction.

Examples:

```
A cost grid is given in below diagram, minimum
cost to reach bottom right from top left
is 327 (= 31 + 10 + 13 + 47 + 65 + 12 + 18 +
6 + 33 + 11 + 20 + 41 + 20)
```

The chosen least cost path is shown in green.

It is not possible to solve this problem using dynamic programming similar to previous problem because here current state depends not only on right and bottom cells but also on left and upper cells. We solve this problem using [dijkstra's algorithm](#). Each cell of grid represents a vertex and neighbor cells adjacent vertices. We do not make an explicit graph

from these cells instead we will use matrix as it is in our dijkstra's algorithm. In below code [Dijkstra' algorithm's implementation using C++ STL](#) is used. The code implemented below is changed to cope with matrix represented implicit graph. Please also see use of dx and dy arrays in below code, these arrays are taken for simplifying the process of visiting neighbor vertices of each cell.

```
// C++ program to get least cost path in a grid from
// top-left to bottom-right
#include <bits/stdc++.h>
using namespace std;

#define ROW 5
#define COL 5

// structure for information of each cell
struct cell
{
    int x, y;
    int distance;
    cell(int x, int y, int distance) :
        x(x), y(y), distance(distance) {}
};

// Utility method for comparing two cells
bool operator<(const cell& a, const cell& b)
{
    if (a.distance == b.distance)
    {
        if (a.x != b.x)
            return (a.x < b.x);
        else
            return (a.y < b.y);
    }
    return (a.distance < b.distance);
}

// Utility method to check whether a point is
// inside the grid or not
bool isInsideGrid(int i, int j)
{
    return (i >= 0 && i < COL && j >= 0 && j < ROW);
}

// Method returns minimum cost to reach bottom
// right from top left
int shortest(int grid[ROW][COL], int row, int col)
{
    int dis[row][col];
```

```

// initializing distance array by INT_MAX
for (int i = 0; i < row; i++)
    for (int j = 0; j < col; j++)
        dis[i][j] = INT_MAX;

// direction arrays for simplification of getting
// neighbour
int dx[] = {-1, 0, 1, 0};
int dy[] = {0, 1, 0, -1};

set<cell> st;

// insert (0, 0) cell with 0 distance
st.insert(cell(0, 0, 0));

// initialize distance of (0, 0) with its grid value
dis[0][0] = grid[0][0];

// loop for standard dijkstra's algorithm
while (!st.empty())
{
    // get the cell with minimum distance and delete
    // it from the set
    cell k = *st.begin();
    st.erase(st.begin());

    // looping through all neighbours
    for (int i = 0; i < 4; i++)
    {
        int x = k.x + dx[i];
        int y = k.y + dy[i];

        // if not inside boundry, ignore them
        if (!isInsideGrid(x, y))
            continue;

        // If distance from current cell is smaller, then
        // update distance of neighbour cell
        if (dis[x][y] > dis[k.x][k.y] + grid[x][y])
        {
            // If cell is already there in set, then
            // remove its previous entry
            if (dis[x][y] != INT_MAX)
                st.erase(st.find(cell(x, y, dis[x][y])));

            // update the distance and insert new updated
            // cell in set
        }
    }
}

```

```
        dis[x][y] = dis[k.x][k.y] + grid[x][y];
        st.insert(cell(x, y, dis[x][y]));
    }
}

// uncomment below code to print distance
// of each cell from (0, 0)
/*
for (int i = 0; i < row; i++, cout << endl)
    for (int j = 0; j < col; j++)
        cout << dis[i][j] << " ";
*/
// dis[row - 1][col - 1] will represent final
// distance of bottom right cell from top left cell
return dis[row - 1][col - 1];
}

// Driver code to test above methods
int main()
{
    int grid[ROW][COL] =
    {
        31, 100, 65, 12, 18,
        10, 13, 47, 157, 6,
        100, 113, 174, 11, 33,
        88, 124, 41, 20, 140,
        99, 32, 111, 41, 20
    };

    cout << shortest(grid, ROW, COL) << endl;
    return 0;
}
```

Output:

327

Source

<https://www.geeksforgeeks.org/minimum-cost-path-left-right-bottom-moves-allowed/>

Chapter 204

Minimum cost to connect all cities

Minimum cost to connect all cities - GeeksforGeeks

There are n cities and there are roads in between some of the cities. Somehow all the roads are damaged simultaneously. We have to repair the roads to connect the cities again. There is a fixed cost to repair a particular road. Find out the minimum cost to connect all the cities by repairing roads. Input is in matrix(city) form, if city[i][j] = 0 then there is not any road between city i and city j, if city[i][j] = a > 0 then the cost to rebuild the path between city i and city j is a. Print out the minimum cost to connect all the cities.
It is sure that all the cities were connected before the roads were damaged.

Examples:

```
Input : {{0, 1, 2, 3, 4},  
         {1, 0, 5, 0, 7},  
         {2, 5, 0, 6, 0},  
         {3, 0, 6, 0, 0},  
         {4, 7, 0, 0, 0}};  
Output : 10
```

```
Input : {{0, 1, 1, 100, 0, 0},  
         {1, 0, 1, 0, 0, 0},  
         {1, 1, 0, 0, 0, 0},  
         {100, 0, 0, 0, 2, 2},  
         {0, 0, 0, 2, 0, 2},  
         {0, 0, 0, 2, 2, 0}};  
Output : 106
```

Method: Here we have to connect all the cities by path which will cost us least. The way to do that is to find out the Minimum Spanning Tree([MST](#)) of the map of the cities(i.e.

each city is a node of the graph and all the damaged roads between cities are edges). And the total cost is the addition of the path edge values in the Minimum Spanning Tree.

Prerequisite: MST Prim's Algorithm

C++

\

```
// C++ code to find out minimum cost
// path to connect all the cities
#include <iostream>
#include <limits>
#include <vector>

using namespace std;

// Function to find out minimum valued node
// among the nodes which are not yet included in MST
int minnode(int n, int keyval[], bool mstset[]) {
    int mini = numeric_limits<int>::max();
    int mini_index;

    // Loop through all the values of the nodes
    // which are not yet included in MST and find
    // the minimum valued one.
    for (int i = 0; i < n; i++) {
        if (mstset[i] == false && keyval[i] < mini) {
            mini = keyval[i], mini_index = i;
        }
    }
    return mini_index;
}

// Function to find out the MST and
// the cost of the MST.
void findcost(int n, vector<vector<int>> city) {

    // Array to store the parent node of a
    // particular node.
    int parent[n];

    // Array to store key value of each node.
    int keyval[n];

    // Boolean Array to hold bool values whether
    // a node is included in MST or not.
    bool mstset[n];
}
```

```
// Set all the key values to infinite and
// none of the nodes is included in MST.
for (int i = 0; i < n; i++) {
    keyval[i] = numeric_limits<int>::max();
    mstset[i] = false;
}

// Start to find the MST from node 0.
// Parent of node 0 is none so set -1.
// key value or minimum cost to reach
// 0th node from 0th node is 0.
parent[0] = -1;
keyval[0] = 0;

// Find the rest n-1 nodes of MST.
for (int i = 0; i < n - 1; i++) {

    // First find out the minimum node
    // among the nodes which are not yet
    // included in MST.
    int u = minnode(n, keyval, mstset);

    // Now the uth node is included in MST.
    mstset[u] = true;

    // Update the values of neighbor
    // nodes of u which are not yet
    // included in MST.
    for (int v = 0; v < n; v++) {

        if (city[u][v] && mstset[v] == false &&
            city[u][v] < keyval[v]) {
            keyval[v] = city[u][v];
            parent[v] = u;
        }
    }
}

// Find out the cost by adding
// the edge values of MST.
int cost = 0;
for (int i = 1; i < n; i++)
    cost += city[parent[i]][i];
cout << cost << endl;
}

// Utility Program:
int main() {
```

```
// Input 1
int n1 = 5;
vector<vector<int>> city1 = {{0, 1, 2, 3, 4},
                                {1, 0, 5, 0, 7},
                                {2, 5, 0, 6, 0},
                                {3, 0, 6, 0, 0},
                                {4, 7, 0, 0, 0}};
findcost(n1, city1);

// Input 2
int n2 = 6;
vector<vector<int>> city2 = {{0, 1, 1, 100, 0, 0},
                                {1, 0, 1, 0, 0, 0},
                                {1, 1, 0, 0, 0, 0},
                                {100, 0, 0, 0, 2, 2},
                                {0, 0, 0, 2, 0, 2},
                                {0, 0, 0, 2, 2, 0}};
findcost(n2, city2);

return 0;
}
```

Output:

```
10
106
```

Complexity: The outer loop(i.e. the loop to add new node to MST) runs n times and in each iteration of the loop it takes $O(n)$ time to find the minnode and $O(n)$ time to update the neighboring nodes of u-th node. Hence the overall complexity is $O(n^2)$

Source

<https://www.geeksforgeeks.org/minimum-cost-connect-cities/>

Chapter 205

Minimum cost to connect weighted nodes represented as array

Minimum cost to connect weighted nodes represented as array - GeeksforGeeks

Given an array of N elements(nodes), where every element is weight of that node. Connecting two nodes will take a cost of product of their weights. You have to connect every node with every other node(directly or indirectly). Output the minimum cost required.

Examples:

```
Input : a[] = {6, 2, 1, 5}
Output : 13
Explanation :
Here, we connect the nodes as follows:
connect a[0] and a[2], cost = 6*1 = 6,
connect a[2] and a[1], cost = 1*2 = 2,
connect a[2] and a[3], cost = 1*5 = 5.
every node is reachable from every other node:
Total cost = 6+2+5 = 13.
```

```
Input : a[] = {5, 10}
Output : 50
Explanation : connections:
connect a[0] and a[1], cost = 5*10 = 50,
Minimum cost = 50.
```

We need to make some observations that we have to make a connected graph with $N-1$ edges. As the output will be sum of products of two numbers, we have to minimize the product of

every term in that sum equation. How can we do it? Clearly, choose the minimum element in the array and connect it with every other. In this way we can reach every other node from a particular node.

Let, minimum element = $a[i]$, (let it's index be 0)

Minimum Cost

So, answer is *product of minimum element and sum of all the elements except minimum element.*

C++

```
// cpp code for Minimum Cost Required to connect weighted nodes
#include <bits/stdc++.h>
using namespace std;
int minimum_cost(int a[], int n)
{
    int mn = INT_MAX;
    int sum = 0;
    for (int i = 0; i < n; i++) {

        // To find the minimum element
        mn = min(a[i], mn);

        // sum of all the elements
        sum += a[i];
    }

    return mn * (sum - mn);
}

// Driver code
int main()
{
    int a[] = { 4, 3, 2, 5 };
    int n = sizeof(a) / sizeof(a[0]);
    cout << minimum_cost(a, n) << endl;
    return 0;
}
```

Java

```
// Java code for Minimum Cost Required to
// connect weighted nodes

import java.io.*;

class GFG {

    static int minimum_cost(int a[], int n)
```

```
{  
  
    int mn = Integer.MAX_VALUE;  
    int sum = 0;  
  
    for (int i = 0; i < n; i++) {  
  
        // To find the minimum element  
        mn = Math.min(a[i], mn);  
  
        // sum of all the elements  
        sum += a[i];  
    }  
  
    return mn * (sum - mn);  
}  
  
// Driver code  
public static void main(String[] args)  
{  
    int a[] = { 4, 3, 2, 5 };  
    int n = a.length;  
  
    System.out.println(minimum_cost(a, n));  
}  
}  
  
// This code is contributed by vt_m.
```

C#

```
// C# code for Minimum Cost Required  
// to connect weighted nodes  
using System;  
  
class GFG {  
  
    // Function to calculate minimum cost  
    static int minimum_cost(int []a, int n)  
    {  
  
        int mn = int.MaxValue;  
        int sum = 0;  
  
        for (int i = 0; i < n; i++)  
        {  
  
            // To find the minimum element
```

```
mn = Math.Min(a[i], mn);

// sum of all the elements
sum += a[i];
}

return mn * (sum - mn);
}

// Driver code
public static void Main()
{
    int []a = {4, 3, 2, 5};
    int n = a.Length;

    Console.WriteLine(minimum_cost(a, n));
}
}

// This code is contributed by vt_m.
```

Output:

24

Time complexity: O(n).

Improved By : [vt_m](#)

Source

<https://www.geeksforgeeks.org/minimum-cost-connect-weighted-nodes-represented-array/>

Chapter 206

Minimum edge reversals to make a root

Minimum edge reversals to make a root - GeeksforGeeks

Given a directed tree with V vertices and $V-1$ edges, we need to choose such a root (from given nodes from where we can reach to every other node) with a minimum number of edge reversal.

Examples:

In above tree, if we choose node 3 as our root then we need to reverse minimum number of 3 edges to reach every other node, changed tree is shown on the right side.

We can solve this problem using **DFS**. we start dfs at any random node of given tree and at each node we store its distance from starting node assuming all edges as undirected and we also store number of edges which need to be reversed in the path from starting node to current node, let's denote such edges as back edges so back edges are those which point towards the node in a path. With this dfs, we also calculate total number of edge reversals in the tree. After this computation, at each node we can calculate 'number of edge reversal to reach every other node' as follows,

Let total number of reversals in tree when some node is chosen as starting node for dfs is R then **if we want to reach every other node from node i we need to reverse all back edges from path node i to starting node and we also need to reverse all other back edges other than node i to starting node path.** First part will be (distance of node i from starting node – back edges count at node i) because we want to reverse edges in path from node i to starting node it will be total edges (i.e. distance) minus back edges from starting node to node i (i.e. back edge count at node i). The second part will be (total

edge reversal or total back edges of tree R – back edge count of node i). After calculating this value at each node we will choose minimum of them as our result.

In below code, in the given edge direction weight 0 is added and in reverse direction weight 1 is added which is used to count reversal edges in dfs method.

```

// C++ program to find min edge reversal to
// make every node reachable from root
#include <bits/stdc++.h>
using namespace std;

// method to dfs in tree and populates disRev values
int dfs(vector< pair<int, int> > g[],
         pair<int, int> disRev[], bool visit[], int u)
{
    // visit current node
    visit[u] = true;
    int totalRev = 0;

    // looping over all neighbors
    for (int i = 0; i < g[u].size(); i++)
    {
        int v = g[u][i].first;
        if (!visit[v])
        {
            // distance of v will be one more than distance of u
            disRev[v].first = disRev[u].first + 1;

            // initialize back edge count same as
            // parent node's count
            disRev[v].second = disRev[u].second;

            // if there is a reverse edge from u to i,
            // then only update
            if (g[u][i].second)
            {
                disRev[v].second = disRev[u].second + 1;
                totalRev++;
            }
            totalRev += dfs(g, disRev, visit, v);
        }
    }

    // return total reversal in subtree rooted at u
    return totalRev;
}

// method prints root and minimum number of edge reversal
void printMinEdgeReverseForRootNode(int edges[][][2], int e)

```

```
{  
    // number of nodes are one more than number of edges  
    int V = e + 1;  
  
    // data structure to store directed tree  
    vector< pair<int, int> > g[V];  
  
    // disRev stores two values - distance and back  
    // edge count from root node  
    pair<int, int> disRev[V];  
  
    bool visit[V];  
  
    int u, v;  
    for (int i = 0; i < e; i++)  
    {  
        u = edges[i][0];  
        v = edges[i][1];  
  
        // add 0 weight in direction of u to v  
        g[u].push_back(make_pair(v, 0));  
  
        // add 1 weight in reverse direction  
        g[v].push_back(make_pair(u, 1));  
    }  
  
    // initialize all variables  
    for (int i = 0; i < V; i++)  
    {  
        visit[i] = false;  
        disRev[i].first = disRev[i].second = 0;  
    }  
  
    int root = 0;  
  
    // dfs populates disRev data structure and  
    // store total reverse edge counts  
    int totalRev = dfs(g, disRev, visit, root);  
  
    // Uncomment below lines to print each node's  
    // distance and edge reversal count from root node  
    /*  
    for (int i = 0; i < V; i++)  
    {  
        cout << i << " : " << disRev[i].first  
            << " " << disRev[i].second << endl;  
    }  
    */
```

```
int res = INT_MAX;

// loop over all nodes to choose minimum edge reversal
for (int i = 0; i < V; i++)
{
    // (reversal in path to i) + (reversal
    // in all other tree parts)
    int edgesToRev = (totalRev - disRev[i].second) +
                    (disRev[i].first - disRev[i].second);

    // choose minimum among all values
    if (edgesToRev < res)
    {
        res = edgesToRev;
        root = i;
    }
}

// print the designated root and total
// edge reversal made
cout << root << " " << res << endl;
}

// Driver code to test above methods
int main()
{
    int edges[] [2] =
    {
        {0, 1},
        {2, 1},
        {3, 2},
        {3, 4},
        {5, 4},
        {5, 6},
        {7, 6}
    };
    int e = sizeof(edges) / sizeof(edges[0]);

    printMinEdgeReverseForRootNode(edges, e);
    return 0;
}
```

Output:

Source

<https://www.geeksforgeeks.org/minimum-edge-reversals-to-make-a-root/>

Chapter 207

Minimum edges required to add to make Euler Circuit

Minimum edges required to add to make Euler Circuit - GeeksforGeeks

Given a undirected graph of **n** nodes and **m** edges. The task is to find minimum edges required to make [Euler Circuit](#) in the given graph.

Examples:

```
Input : n = 3,  
        m = 2  
        Edges[] = {{1, 2}, {2, 3}}  
Output : 1
```

By connecting 1 to 3, we can create a Euler Circuit.

For a Euler Circuit to exist in the graph we require that every node should have even degree because then there exists an edge that can be used to exit the node after entering it.

Now, there can be two case:

1. There is one connected component in the graph

In this case, if all the nodes in the graph is of even degree then we say that the graph already have a Euler Circuit and we don't need to add any edge in it. But if there is any node with odd degree we need to add edges.

There can be even number of odd degree vertices in the graph. This can be easily proved by the fact that the sum of degrees from the even degrees node and degrees from odd degrees node should match the total degrees that is always even as every edge contributes two to this sum. Now, if we pair up random odd degree nodes in the graph and add an edge between them we can make all nodes to have even degree and thus make an Euler Circuit exist.

2. There are disconnected components in the graph

We first mark component as odd and even. Odd components are those which have at least

one odd degree node in them. Take all the even components and select a random vertex from every component and line them up linearly. Now we add an edge between adjacent vertices. So we have connected the even components and made an equivalent odd components that has two nodes with odd degree.

Now to deal with odd components i.e components with at least one odd degree node. We can connect all these odd components using edges whose number is equal to the number of disconnected components. This can be done by placing the components in the cyclic order and picking two odd degree nodes from every component and using these to connect to the components on either side. Now we have a single connected component for which we have discussed.

Below is C++ implementation of this approach:

```
// C++ program to find minimum edge required
// to make Euler Circuit
#include <bits/stdc++.h>
using namespace std;

// Depth-First Search to find a connected
// component
void dfs(vector<int> g[], int vis[], int odd[],
          int deg[], int comp, int v)
{
    vis[v] = 1;

    if (deg[v]%2 == 1)
        odd[comp]++;
}

for (int u : g[v])
    if (vis[u] == 0)
        dfs(g, vis, odd, deg, comp, u);
}

// Return minimum edge required to make Euler
// Circuit
int minEdge(int n, int m, int s[], int d[])
{
    // g : to store adjacency list
    //      representation of graph.
    // e : to store list of even degree vertices
    // o : to store list of odd degree vertices
    vector<int> g[n+1], e, o;

    int deg[n+1]; // Degrees of vertices
    int vis[n+1]; // To store visited in DFS
    int odd[n+1]; // Number of odd nodes in components
    memset(deg, 0, sizeof(deg));
    memset(vis, 0, sizeof(vis));
    memset(odd, 0, sizeof(odd));
```

```

for (int i = 0; i < m; i++)
{
    g[s[i]].push_back(d[i]);
    g[d[i]].push_back(s[i]);
    deg[s[i]]++;
    deg[d[i]]++;
}

// 'ans' is result and 'comp' is component id
int ans = 0, comp = 0;
for (int i = 1; i <= n; i++)
{
    if (vis[i]==0)
    {
        comp++;
        dfs(g, vis, odd, deg, comp, i);

        // Checking if connected component
        // is odd.
        if (odd[comp] == 0)
            e.push_back(comp);

        // Checking if connected component
        // is even.
        else
            o.push_back(comp);
    }
}

// If whole graph is a single connected
// component with even degree.
if (o.size() == 0 && e.size() == 1)
    return 0;

// If all connected component is even
if (o.size() == 0)
    return e.size();

// If graph have atleast one even connected
// component
if (e.size() != 0)
    ans += e.size();

// For all the odd connected component.
for (int i : o)
    ans += odd[i]/2;

```

```
    return ans;
}

// Driven Program
int main()
{
    int n = 3, m = 2;
    int source[] = { 1, 2 };
    int destination[] = { 2, 3 };

    cout << minEdge(n, m, source, destination) << endl;
    return 0;
}
```

Output:

1

Source

<https://www.geeksforgeeks.org/minimum-edges-required-to-add-to-make-euler-circuit/>

Chapter 208

Minimum edges to reverse to make path from a source to a destination

Minimum edges to reverse to make path from a source to a destination - GeeksforGeeks

Given a directed graph and a source node and destination node, we need to find how many edges we need to reverse in order to make at least 1 path from source node to destination node.

Examples:

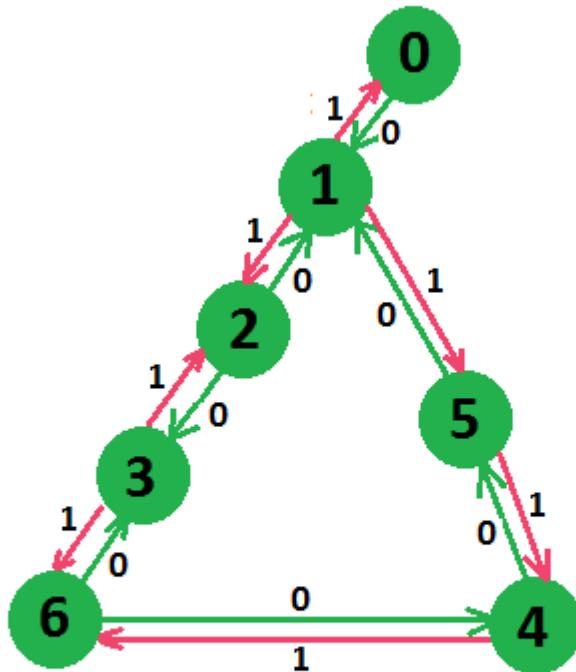
In above graph there were two paths from node 0 to node 6,

0 → 1 → 2 → 3 → 6

0 → 1 → 5 → 4 → 6

But for first path only two edges need to be reversed, so answer will be 2 only.

This problem can be solved assuming a different version of the given graph. In this version we make a reverse edge corresponding to every edge and we assign that a weight 1 and assign a weight 0 to original edge. After this modification above graph looks something like below,



Modified graph

Now we can see that we have modified the graph in such a way that, if we move towards original edge, no cost is incurred, but if we move toward reverse edge 1 cost is added. So if we apply [Dijkstra's shortest path](#) on this modified graph from given source, then that will give us minimum cost to reach from source to destination i.e. minimum edge reversal from source to destination.

Below is the code based on above concept.

```
// Program to find minimum edge reversal to get
// atleast one path from source to destination
#include <bits/stdc++.h>
using namespace std;
#define INF 0x3f3f3f3f

// This class represents a directed graph using
// adjacency list representation
class Graph
{
    int V;      // No. of vertices
```

```
// In a weighted graph, we need to store vertex
// and weight pair for every edge
list< pair<int, int> > *adj;

public:
    Graph(int V); // Constructor

    // function to add an edge to graph
    void addEdge(int u, int v, int w);

    // returns shortest path from s
    vector<int> shortestPath(int s);
};

// Allocates memory for adjacency list
Graph::Graph(int V)
{
    this->V = V;
    adj = new list< pair<int, int> >[V];
}

// method adds a directed edge from u to v with weight w
void Graph::addEdge(int u, int v, int w)
{
    adj[u].push_back(make_pair(v, w));
}

// Prints shortest paths from src to all other vertices
vector<int> Graph::shortestPath(int src)
{
    // Create a set to store vertices that are being
    // preroocessed
    set< pair<int, int> > setds;

    // Create a vector for distances and initialize all
    // distances as infinite (INF)
    vector<int> dist(V, INF);

    // Insert source itself in Set and initialize its
    // distance as 0.
    setds.insert(make_pair(0, src));
    dist[src] = 0;

    /* Looping till all shortest distance are finalized
       then setds will become empty */
    while (!setds.empty())
    {
```

```

// The first vertex in Set is the minimum distance
// vertex, extract it from set.
pair<int, int> tmp = *(setds.begin());
setds.erase(setds.begin());

// vertex label is stored in second of pair (it
// has to be done this way to keep the vertices
// sorted distance (distance must be first item
// in pair)
int u = tmp.second;

// 'i' is used to get all adjacent vertices of a vertex
list< pair<int, int> >::iterator i;
for (i = adj[u].begin(); i != adj[u].end(); ++i)
{
    // Get vertex label and weight of current adjacent
    // of u.
    int v = (*i).first;
    int weight = (*i).second;

    // If there is shorter path to v through u.
    if (dist[v] > dist[u] + weight)
    {
        /* If distance of v is not INF then it must be in
           our set, so removing it and inserting again
           with updated less distance.
           Note : We extract only those vertices from Set
           for which distance is finalized. So for them,
           we would never reach here. */
        if (dist[v] != INF)
            setds.erase(setds.find(make_pair(dist[v], v)));

        // Updating distance of v
        dist[v] = dist[u] + weight;
        setds.insert(make_pair(dist[v], v));
    }
}
return dist;
}

/* method adds reverse edge of each original edge
   in the graph. It gives reverse edge a weight = 1
   and all original edges a weight of 0. Now, the
   length of the shortest path will give us the answer.
   If shortest path is p: it means we used p reverse
   edges in the shortest path. */
Graph modelGraphWithEdgeWeight(int edge[][][2], int E, int V)

```

```
{  
    Graph g(V);  
    for (int i = 0; i < E; i++)  
    {  
        // original edge : weight 0  
        g.addEdge(edge[i][0], edge[i][1], 0);  
  
        // reverse edge : weight 1  
        g.addEdge(edge[i][1], edge[i][0], 1);  
    }  
    return g;  
}  
  
// Method returns minimum number of edges to be  
// reversed to reach from src to dest  
int getMinEdgeReversal(int edge[][2], int E, int V,  
                      int src, int dest)  
{  
    // get modified graph with edge weight  
    Graph g = modelGraphWithEdgeWeight(edge, E, V);  
  
    // get shortest path vector  
    vector<int> dist = g.shortestPath(src);  
  
    // If distance of destination is still INF,  
    // not possible  
    if (dist[dest] == INF)  
        return -1;  
    else  
        return dist[dest];  
}  
  
// Driver code to test above method  
int main()  
{  
    int V = 7;  
    int edge[][2] = {{0, 1}, {2, 1}, {2, 3}, {5, 1},  
                    {4, 5}, {6, 4}, {6, 3}};  
    int E = sizeof(edge) / sizeof(edge[0]);  
  
    int minEdgeToReverse =  
        getMinEdgeReversal(edge, E, V, 0, 6);  
    if (minEdgeToReverse != -1)  
        cout << minEdgeToReverse << endl;  
    else  
        cout << "Not possible" << endl;  
    return 0  
}
```

Output:

2

Source

<https://www.geeksforgeeks.org/minimum-edges-reverse-make-path-source-destination/>

Chapter 209

Minimum initial vertices to traverse whole matrix with given conditions

Minimum initial vertices to traverse whole matrix with given conditions - GeeksforGeeks

We are given a matrix that contains different values in its each cell. Our aim is to find the minimal set of positions in the matrix such that entire matrix can be traversed starting from the positions in the set.

We can traverse the matrix under below conditions:

- We can move only to those neighbors that contain value less than or to equal to the current cell's value. A neighbor of cell is defined as the cell that shares a side with the given cell.

Examples:

```
Input : 1 2 3  
        2 3 1  
        1 1 1  
Output : 1 1  
        0 2
```

If we start from 1, 1 we can cover 6 vertices in the order 1, 1 → 1, 0 → 2, 0 → 2, 1 → 2, 2 → 1, 2. We cannot cover the entire matrix with this vertex. Remaining vertices can be covered 0, 2 → 0, 1 → 0, 0.

```
Input : 3 3  
        1 1
```

```
Output : 0 1
If we start from 0, 1, we can traverse
the entire matrix from this single vertex
in this order 0, 0 -> 0, 1 -> 1, 1 -> 1, 0.
Traversing the matrix in this order
satisfies all the conditions stated above.
```

From the above examples, we can easily identify that in order to use minimum number of positions we have to start from the positions having highest cell value. Therefore we pick the positions that contain the highest value in the matrix. We take the vertices having highest value in separate array. We perform **DFS** on every vertex starting from the highest value. If we encounter any unvisited vertex during dfs then we have to include this vertex in our set. When all the cells have been processed then the set contains the required vertices.

How does this work?

We need to visit all vertices and to reach largest values we must start with them. If two largest values are not adjacent, then both of them must be picked. If two largest values are adjacent, then any of them can be picked as moving to equal value neighbors is allowed.

```
// CPP program to find minimum initial
// vertices to reach whole matrix.
#include <bits/stdc++.h>
using namespace std;

const int MAX = 100;

// (n, m) is current source cell from which
// we need to do DFS. N and M are total no.
// of rows and columns.
void dfs(int n, int m, bool visit[][][MAX],
         int adj[][][MAX], int N, int M)
{
    // Marking the vertex as visited
    visit[n][m] = 1;

    // If below neighbor is valid and has
    // value less than or equal to current
    // cell's value
    if (n + 1 < N &&
        adj[n][m] >= adj[n + 1][m] &&
        !visit[n + 1][m])
        dfs(n + 1, m, visit, adj, N, M);

    // If right neighbor is valid and has
    // value less than or equal to current
    // cell's value
    if (m + 1 < M &&
        adj[n][m] >= adj[n][m + 1] &&
```

```

        !visit[n][m + 1])
        dfs(n, m + 1, visit, adj, N, M);

    // If above neighbor is valid and has
    // value less than or equal to current
    // cell's value
    if (n - 1 >= 0 &&
        adj[n][m] >= adj[n - 1][m] &&
        !visit[n - 1][m])
        dfs(n - 1, m, visit, adj, N, M);

    // If left neighbor is valid and has
    // value less than or equal to current
    // cell's value
    if (m - 1 >= 0 &&
        adj[n][m] >= adj[n][m - 1] &&
        !visit[n][m - 1])
        dfs(n, m - 1, visit, adj, N, M);
    }

void printMinSources(int adj[][][MAX], int N, int M)
{
    // Storing the cell value and cell indices
    // in a vector.
    vector<pair<long int, pair<int, int>> x;
    for (int i = 0; i < N; i++)
        for (int j = 0; j < M; j++)
            x.push_back(make_pair(adj[i][j],
                                  make_pair(i, j)));

    // Sorting the newly created array according
    // to cell values
    sort(x.begin(), x.end());

    // Create a visited array for DFS and
    // initialize it as false.
    bool visit[N][MAX];
    memset(visit, false, sizeof(visit));

    // Applying dfs for each vertex with
    // highest value
    for (int i = x.size() - 1; i >= 0; i--)
    {
        // If the given vertex is not visited
        // then include it in the set
        if (!visit[x[i].second.first][x[i].second.second])
        {

```

```
cout << x[i].second.first << " "
    << x[i].second.second << endl;
dfs(x[i].second.first, x[i].second.second,
    visit, adj, N, M);
}
}
}

// Driver code
int main()
{
    int N = 2, M = 2;

    int adj[N][MAX] = {{3, 3},
                       {1, 1}};
    printMinSources(adj, N, M);
    return 0;
}
```

Output:

0 1

Source

<https://www.geeksforgeeks.org/minimum-initial-vertices-traverse-whole-matrix-given-conditions/>

Chapter 210

Minimum number of edges between two vertices of a Graph

Minimum number of edges between two vertices of a Graph - GeeksforGeeks

You are given a undirected graph $G(V, E)$ with N vertices and M edges. We need to find the minimum number of edges between a given pair of vertices (u, v) .

Examples:

Input : For given graph G. Find minimum number of edges between (1, 5).

Output : 2

Explanation: (1, 2) and (2, 5) are the only edges resulting into shortest path between 1 and 5.

The idea is to perform BFS from one of given input vertex(u). At the time of BFS maintain an array of $distance[n]$ and initialize it to zero for all vertices. Now, suppose during BFS, vertex x is popped from queue and we are pushing all adjacent non-visited vertices(i) back into queue at the same time we should update $distance[i] = distance[x] + 1$; Finally $distance[v]$ gives the minimum number of edges between u and v .

Algorithm:

```
int minEdgeBFS(int u, int v, int n)
{
    // visited[n] for keeping track of visited
    // node in BFS
    bool visited[n] = {0};
```

```
// Initialize distances as 0
int distance[n] = {0};

// queue to do BFS.
queue Q;
distance[u] = 0;

Q.push(u);
visited[u] = true;
while (!Q.empty())
{
    int x = Q.front();
    Q.pop();

    for (int i=0; i<edges[x].size(); i++)
    {
        if (visited[edges[x][i]])
            continue;

        // update distance for i
        distance[edges[x][i]] = distance[x] + 1;
        Q.push(edges[x][i]);
        visited[edges[x][i]] = 1;
    }
}
return distance[v];
}
```

C++

```
// C++ program to find minimum edge
// between given two vertex of Graph
#include<bits/stdc++.h>
using namespace std;

// function for finding minimum no. of edge
// using BFS
int minEdgeBFS(vector <int> edges[], int u,
                int v, int n)
{
    // visited[n] for keeping track of visited
    // node in BFS
    vector<bool> visited(n, 0);

    // Initialize distances as 0
    vector<int> distance(n, 0);

    // queue to do BFS.
```

```
queue <int> Q;
distance[u] = 0;

Q.push(u);
visited[u] = true;
while (!Q.empty())
{
    int x = Q.front();
    Q.pop();

    for (int i=0; i<edges[x].size(); i++)
    {
        if (visited[edges[x][i]])
            continue;

        // update distance for i
        distance[edges[x][i]] = distance[x] + 1;
        Q.push(edges[x][i]);
        visited[edges[x][i]] = 1;
    }
}
return distance[v];
}

// function for addition of edge
void addEdge(vector <int> edges[], int u, int v)
{
    edges[u].push_back(v);
    edges[v].push_back(u);
}

// Driver function
int main()
{
    // To store adjacency list of graph
    int n = 9;
    vector <int> edges[9];
    addEdge(edges, 0, 1);
    addEdge(edges, 0, 7);
    addEdge(edges, 1, 7);
    addEdge(edges, 1, 2);
    addEdge(edges, 2, 3);
    addEdge(edges, 2, 5);
    addEdge(edges, 2, 8);
    addEdge(edges, 3, 4);
    addEdge(edges, 3, 5);
    addEdge(edges, 4, 5);
    addEdge(edges, 5, 6);
```

```
    addEdge(edges, 6, 7);
    addEdge(edges, 7, 8);
    int u = 0;
    int v = 5;
    cout << minEdgeBFS(edges, u, v, n);
    return 0;
}
```

Java

```
// Java program to find minimum edge
// between given two vertex of Graph

import java.util.LinkedList;
import java.util.Queue;
import java.util.Vector;

class Test
{
    // Method for finding minimum no. of edge
    // using BFS
    static int minEdgeBFS(Vector <Integer> edges[], int u,
                          int v, int n)
    {
        // visited[n] for keeping track of visited
        // node in BFS
        Vector<Boolean> visited = new Vector<Boolean>(n);

        for (int i = 0; i < n; i++) {
            visited.addElement(false);
        }

        // Initialize distances as 0
        Vector<Integer> distance = new Vector<Integer>(n);

        for (int i = 0; i < n; i++) {
            distance.addElement(0);
        }

        // queue to do BFS.
        Queue<Integer> Q = new LinkedList<>();
        distance.setElementAt(0, u);

        Q.add(u);
        visited.setElementAt(true, u);
        while (!Q.isEmpty())
        {
            int x = Q.peek();
```

```
Q.poll();

for (int i=0; i<edges[x].size(); i++)
{
    if (visited.elementAt(edges[x].get(i)))
        continue;

    // update distance for i
    distance.setElementAt(distance.get(x) + 1,edges[x].get(i));
    Q.add(edges[x].get(i));
    visited.setElementAt(true,edges[x].get(i));
}
}

return distance.get(v);
}

// method for addition of edge
static void addEdge(Vector <Integer> edges[], int u, int v)
{
    edges[u].add(v);
    edges[v].add(u);
}

// Driver method
public static void main(String args[])
{
    // To store adjacency list of graph
    int n = 9;
    Vector <Integer> edges[] = new Vector[9];

    for (int i = 0; i < edges.length; i++) {
        edges[i] = new Vector<>();
    }

    addEdge(edges, 0, 1);
    addEdge(edges, 0, 7);
    addEdge(edges, 1, 7);
    addEdge(edges, 1, 2);
    addEdge(edges, 2, 3);
    addEdge(edges, 2, 5);
    addEdge(edges, 2, 8);
    addEdge(edges, 3, 4);
    addEdge(edges, 3, 5);
    addEdge(edges, 4, 5);
    addEdge(edges, 5, 6);
    addEdge(edges, 6, 7);
    addEdge(edges, 7, 8);
    int u = 0;
```

```
        int v = 5;
        System.out.println(minEdgeBFS(edges, u, v, n));
    }
}
// This code is contributed by Gaurav Miglani
```

Output:

3

Source

<https://www.geeksforgeeks.org/minimum-number-of-edges-between-two-vertices-of-a-graph/>

Chapter 211

Minimum number of operation required to convert number x into y

Minimum number of operation required to convert number x into y - GeeksforGeeks

Given a initial number x and two operations which are given below:

1. Multiply number by 2.
2. Subtract 1 from the number.

The task is to find out minimum number of operation required to convert number x into y using only above two operations. We can apply these operations any number of times.

Constraints:

$1 \leq x, y \leq 10000$

Example:

```
Input : x = 4, y = 7
Output : 2
We can transform x into y using following
two operations.
1. 4*2 = 8
2. 8-1 = 7
```

```
Input : x = 2, y = 5
Output : 4
We can transform x into y using following
four operations.
```

```
1. 2*2 = 4
2. 4-1 = 3
3. 3*2 = 6
4. 6-1 = 5
Answer = 4
Note that other sequences of two operations
would take more operations.
```

The idea is to use [BFS](#) for this. We run a BFS and create nodes by multiplying with 2 and subtracting by 1, thus we can obtain all possible numbers reachable from starting number.

Important Points :

- 1) When we subtract 1 from a number and if it becomes < 0 i.e. Negative then there is no reason to create next node from it (As per input constraints, numbers x and y are positive).
- 2) Also, if we have already created a number then there is no reason to create it again. i.e. we maintain a visited array.

C++

```
// C++ program to find minimum number of steps needed
// to convert a number x into y with two operations
// allowed : (1) multiplication with 2 (2) subtraction
// with 1.
#include<bits/stdc++.h>
using namespace std;

// A node of BFS traversal
struct node
{
    int val;
    int level;
};

// Returns minimum number of operations
// needed to convert x into y using BFS
int minOperations(int x, int y)
{
    // To keep track of visited numbers
    // in BFS.
    set<int> visit;

    // Create a queue and enqueue x into it.
    queue<node> q;
    node n = {x, 0};
    q.push(n);

    // Do BFS starting from x
    while (!q.empty())
```

```
{  
    // Remove an item from queue  
    node t = q.front();  
    q.pop();  
  
    // If the removed item is target  
    // number y, return its level  
    if (t.val == y)  
        return t.level;  
  
    // Mark dequeued number as visited  
    visit.insert(t.val);  
  
    // If we can reach y in one more step  
    if (t.val*2 == y || t.val-1 == y)  
        return t.level+1;  
  
    // Insert children of t if not visited  
    // already  
    if (visit.find(t.val*2) == visit.end())  
    {  
        n.val = t.val*2;  
        n.level = t.level+1;  
        q.push(n);  
    }  
    if (t.val-1>=0 && visit.find(t.val-1) == visit.end())  
    {  
        n.val = t.val-1;  
        n.level = t.level+1;  
        q.push(n);  
    }  
}  
}  
  
// Driver code  
int main()  
{  
    int x = 4, y = 7;  
    cout << minOperations(x, y);  
    return 0;  
}
```

Java

```
// Java program to find minimum  
// number of steps needed to  
// convert a number x into y  
// with two operations allowed :
```

```
// (1) multiplication with 2
// (2) subtraction with 1.

import java.util.HashSet;
import java.util.LinkedList;
import java.util.Set;

class GFG
{
    int val;
    int steps;

    public GFG(int val, int steps)
    {
        this.val = val;
        this.steps = steps;
    }
}

public class GeeksForGeeks
{
    private static int minOperations(int src,
                                    int target)
    {

        Set<GFG> visited = new HashSet<>(1000);
        LinkedList<GFG> queue = new LinkedList<GFG>();

        GFG node = new GFG(src, 0);

        queue.offer(node);
        visited.add(node);

        while (!queue.isEmpty())
        {
            GFG temp = queue.poll();
            visited.add(temp);

            if (temp.val == target)
            {
                return temp.steps;
            }

            int mul = temp.val * 2;
            int sub = temp.val - 1;

            // given constraints
            if (mul > 0 && mul < 1000)
```

```
{  
    GFG nodeMul = new GFG(mul, temp.steps + 1);  
    queue.offer(nodeMul);  
}  
if (sub > 0 && sub < 1000)  
{  
    GFG nodeSub = new GFG(sub, temp.steps + 1);  
    queue.offer(nodeSub);  
}  
}  
return -1;  
}  
  
// Driver code  
public static void main(String[] args)  
{  
    // int x = 2, y = 5;  
    int x = 4, y = 7;  
    GFG src = new GFG(x, y);  
    System.out.println(minOperations(x, y));  
}  
}  
  
// This code is contributed by Rahul
```

Output :

2

Improved By : [rahulpopat](#)

Source

<https://www.geeksforgeeks.org/minimum-number-operation-required-convert-number-x-y/>

Chapter 212

Minimum number of swaps required to sort an array

Minimum number of swaps required to sort an array - GeeksforGeeks

Given an array of **n** distinct elements, find the minimum number of swaps required to sort the array.

Examples:

Input : {4, 3, 2, 1}

Output : 2

Explanation : Swap index 0 with 3 and 1 with 2 to form the sorted array {1, 2, 3, 4}.

Input : {1, 5, 4, 3, 2}

Output : 2

This can be easily done by visualizing the problem as a graph. We will have **n** nodes and an edge directed from node **i** to node **j** if the element at **i**'th index must be present at **j**'th index in the sorted array.

Graph for {4, 3, 2, 1}

The graph will now contain many non-intersecting cycles. Now a cycle with 2 nodes will only require 1 swap to reach the correct ordering, similarly a cycle with 3 nodes will only require 2 swap to do so.

Graph for {4, 5, 2, 1, 5}

Hence,

$$\text{ans} = \sum_{i=1}^k (\text{cycle_size} - 1)$$

where k is the number of cycles

Below is the C++ implementation of the idea.

C++

```
// C++ program to find minimum number of swaps
// required to sort an array
#include<bits/stdc++.h>

using namespace std;

// Function returns the minimum number of swaps
// required to sort the array
int minSwaps(int arr[], int n)
{
    // Create an array of pairs where first
    // element is array element and second element
    // is position of first element
    pair<int, int> arrPos[n];
    for (int i = 0; i < n; i++)
    {
        arrPos[i].first = arr[i];
        arrPos[i].second = i;
    }

    // Sort the array by array element values to
    // get right position of every element as second
    // element of pair.
    sort(arrPos, arrPos + n);

    // To keep track of visited elements. Initialize
    // all elements as not visited or false.
    vector<bool> vis(n, false);

    // Initialize result
    int ans = 0;

    // Traverse array elements
    for (int i = 0; i < n; i++)
    {
        // already swapped and corrected or
        // already present at correct pos
        if (vis[i] || arrPos[i].second == i)
            continue;

        int correct_pos = arrPos[i].second;
```

```
// find out the number of node in
// this cycle and add in ans
int cycle_size = 0;
int j = i;
while (!vis[j])
{
    vis[j] = 1;

    // move to next node
    j = arrPos[j].second;
    cycle_size++;
}

// Update answer by adding current cycle.
if(cycle_size > 0)
{
    ans += (cycle_size - 1);
}
}

// Return result
return ans;
}

// Driver program to test the above function
int main()
{
    int arr[] = {1, 5, 4, 3, 2};
    int n = (sizeof(arr) / sizeof(int));
    cout << minSwaps(arr, n);
    return 0;
}
```

Java

```
// Java program to find minimum number of swaps
// required to sort an array
import javafx.util.Pair;
import java.util.ArrayList;
import java.util.*;

class GfG
{
    // Function returns the minimum number of swaps
    // required to sort the array
    public static int minSwaps(int[] arr)
    {
        int n = arr.length;
```

```
// Create two arrays and use as pairs where first
// array is element and second array
// is position of first element
ArrayList <Pair <Integer, Integer> > arrpos =
    new ArrayList <Pair <Integer, Integer> > ();
for (int i = 0; i < n; i++)
    arrpos.add(new Pair <Integer, Integer> (arr[i], i));

// Sort the array by array element values to
// get right position of every element as the
// elements of second array.
arrpos.sort(new Comparator<Pair<Integer, Integer>>()
{
    @Override
    public int compare(Pair<Integer, Integer> o1,
                       Pair<Integer, Integer> o2)
    {
        if (o1.getKey() > o2.getKey())
            return -1;

        // We can change this to make it then look at the
        // words alphabetical order
        else if (o1.getKey().equals(o2.getKey()))
            return 0;

        else
            return 1;
    }
});

// To keep track of visited elements. Initialize
// all elements as not visited or false.
Boolean[] vis = new Boolean[n];
Arrays.fill(vis, false);

// Initialize result
int ans = 0;

// Traverse array elements
for (int i = 0; i < n; i++)
{
    // already swapped and corrected or
    // already present at correct pos
    if (vis[i] || arrpos.get(i).getValue() == i)
        continue;

    // find out the number of node in
```

```
// this cycle and add in ans
int cycle_size = 0;
int j = i;
while (!vis[j])
{
    vis[j] = true;

    // move to next node
    j = arrpos.get(j).getValue();
    cycle_size++;
}

// Update answer by adding current cycle.
if(cycle_size > 0)
{
    ans += (cycle_size - 1);
}
}

// Return result
return ans;
}

// Driver class
class MinSwaps
{
    // Driver program to test the above function
    public static void main(String[] args)
    {
        int []a = {1, 5, 4, 3, 2};
        GfG g = new GfG();
        System.out.println(g.minSwaps(a));
    }
}
// This code is contributed by Saksham Seth

[ /sourcecode]
```

Python3

```
# Python3 program to find minimum number
# of swaps required to sort an array

# Function returns the minimum
# number of swaps required to sort the array
def minSwaps(arr):
    n = len(arr)
```

```
# Create two arrays and use
# as pairs where first array
# is element and second array
# is position of first element
arrpos = [*enumerate(arr)]

# Sort the array by array element
# values to get right position of
# every element as the elements
# of second array.
arrpos.sort(key = lambda it:it[1])

# To keep track of visited elements.
# Initialize all elements as not
# visited or false.
vis = {k:False for k in range(n)}

# Initialize result
ans = 0
for i in range(n):

    # already swapped or
    # already present at
    # correct position
    if vis[i] or arrpos[i][0] == i:
        continue

    # find number of nodes
    # in this cycle and
    # add it to ans
    cycle_size = 0
    j = i
    while not vis[j]:

        # mark node as visited
        vis[j] = True

        # move to next node
        j = arrpos[j][0]
        cycle_size += 1

    # update answer by adding
    # current cycle
    if cycle_size > 0:
        ans += (cycle_size - 1)
# return answer
return ans
```

```
# Driver Code
arr = [1, 5, 4, 3, 2]
print(minSwaps(arr))

# This code is contributed
# by Dharan Aditya
```

Output:

2

Time Complexity: $O(n \log n)$

Auxiliary Space: $O(n)$

Related Article :

[Number of swaps to sort when only adjacent swapping allowed](#)

Reference:

<http://stackoverflow.com/questions/15152322/compute-the-minimal-number-of-swaps-to-order-a-sequence/15152602#15152602>

Improved By : [dharan1011](#), PREM UKKOJI

Source

<https://www.geeksforgeeks.org/minimum-number-swaps-required-sort-array/>

Chapter 213

Minimum Product Spanning Tree

Minimum Product Spanning Tree - GeeksforGeeks

Given a connected and undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A minimum product spanning tree for a weighted, connected and undirected graph is a spanning tree with weight product less than or equal to the weight product of every other spanning tree. The weight product of a spanning tree is the product of weights corresponding to each edge of the spanning tree. All weights of the given graph will be positive for simplicity.

Examples:

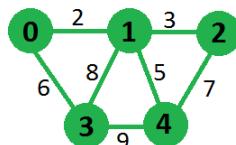
Minimum Product that we can obtain is
180 for above graph by choosing edges
0-1, 1-2, 0-3 and 1-4

This problem can be solved using standard minimum spanning tree algorithms like [krushkal](#) and [prim](#)'s algorithm, but we need to modify our graph to use these algorithms. Minimum spanning tree algorithms tries to minimize total sum of weights, here we need to minimize total product of weights. We can use property of [logarithms](#) to overcome this problem.
As we know,

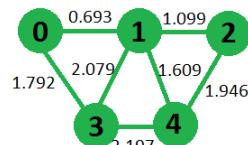
$$\log(w_1 * w_2 * w_3 * \dots * w_N) = \\ \log(w_1) + \log(w_2) + \log(w_3) + \dots + \log(w_N)$$

We can replace each weight of graph by its log value, then we apply any minimum spanning tree algorithm which will try to minimize sum of $\log(w_i)$ which in-turn minimizes weight product.

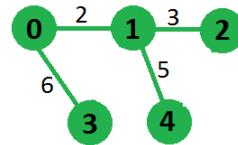
For example graph, steps are shown in below diagram,



Graph with edge weights



Graph with log of edge weights



MST for log graph with actual edges

In below code first we have constructed the log graph from given input graph, then that graph is given as input to prim's MST algorithm, which will minimize the total sum of weights of tree. Since weight of modified graph are logarithms of actual input graph, we actually minimize the product of weights of spanning tree.

```

// A C/C++ program for getting minimum product
// spanning tree The program is for adjacency matrix
// representation of the graph
#include <bits/stdc++.h>

// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with minimum
// key value, from the set of vertices not yet included
// in MST
int minKey(int key[], bool mstSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

// A utility function to print the constructed MST
// stored in parent[] and print Minimum Obtainable
// product
int printMST(int parent[], int n, int graph[V][V])
{
    printf("Edge    Weight\n");
    
```

```

int minProduct = 1;
for (int i = 1; i < V; i++)
{
    printf("%d - %d      %d \n",
           parent[i], i, graph[i][parent[i]]);

    minProduct *= graph[i][parent[i]];
}
printf("Minimum Obtainable product is %d\n",
       minProduct);
}

// Function to construct and print MST for a graph
// represented using adjacency matrix representation
// inputGraph is sent for printing actual edges and
// logGraph is sent for actual MST operations
void primMST(int inputGraph[V][V], double logGraph[V][V])
{
    int parent[V]; // Array to store constructed MST
    int key[V];   // Key values used to pick minimum
                  // weight edge in cut
    bool mstSet[V]; // To represent set of vertices not
                    // yet included in MST

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    // Always include first 1st vertex in MST.
    key[0] = 0;      // Make key 0 so that this vertex is
                    // picked as first vertex
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (int count = 0; count < V-1; count++)
    {
        // Pick the minimum key vertex from the set of
        // vertices not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set
        mstSet[u] = true;

        // Update key value and parent index of the
        // adjacent vertices of the picked vertex.
        // Consider only those vertices which are not yet
        // included in MST
        for (int v = 0; v < V; v++)

```

```

        // logGraph[u][v] is non zero only for
        // adjacent vertices of m mstSet[v] is false
        // for vertices not yet included in MST
        // Update the key only if logGraph[u][v] is
        // smaller than key[v]
        if (logGraph[u][v] > 0 &&
            mstSet[v] == false &&
            logGraph[u][v] < key[v])

            parent[v] = u, key[v] = logGraph[u][v];
        }

        // print the constructed MST
        printMST(parent, V, inputGraph);
    }

    // Method to get minimum product spanning tree
    void minimumProductMST(int graph[V][V])
    {
        double logGraph[V][V];

        // Constructing logGraph from original graph
        for (int i = 0; i < V; i++)
        {
            for (int j = 0; j < V; j++)
            {
                if (graph[i][j] > 0)
                    logGraph[i][j] = log(graph[i][j]);
                else
                    logGraph[i][j] = 0;
            }
        }

        // Applying standard Prim's MST algorithm on
        // Log graph.
        primMST(graph, logGraph);
    }

    // driver program to test above function
    int main()
    {
        /* Let us create the following graph
           2   3
           (0)--(1)--(2)
           |   / \   |
           6| 8/   \5 |7
           | /       \ |
    
```

```
(3)----- (4)
         9      */
int graph[V][V] = {{0, 2, 0, 6, 0},
{2, 0, 3, 8, 5},
{0, 3, 0, 0, 7},
{6, 8, 0, 0, 9},
{0, 5, 7, 9, 0},
};

// Print the solution
minimumProductMST(graph);

return 0;
}
```

Output:

Edge	Weight
0 - 1	2
1 - 2	3
0 - 3	6
1 - 4	5

Minimum Obtainable product is 180

Source

<https://www.geeksforgeeks.org/minimum-product-spanning-tree/>

Chapter 214

Minimum steps to reach a destination

Minimum steps to reach a destination - GeeksforGeeks

Given a number line from $-\infty$ to $+\infty$. You start at 0 and can go either to the left or to the right. The condition is that in i 'th move, you take i steps.

- a) Find if you can reach a given number x
- b) Find the most optimal way to reach a given number x , if we can indeed reach it. For example, 3 can be reached in 2 steps, $(0, 1)$ $(1, 3)$ and 4 can be reached in 3 steps $(0, -1)$, $(-1, 1)$ $(1, 4)$.

Source: [Flipkart Interview Question](#)

The important think to note is we can reach any destination as it is always possible to make a move of length 1. At any step i , we can move forward i , then backward $i + 1$.

Below is a recursive solution suggested by Arpit Thapar [here](#).

- 1) Since distance of $+ 5$ and $- 5$ from 0 is same, hence we find answer for absolute value of destination.
- 2) We use a recursive function which takes as arguments:
 - i) Source Vertex
 - ii) Value of last step taken
 - iii) Destination
- 3) If at any point source vertex = destination; return number of steps.
- 4) Otherwise we can go in both of the possible directions. Take the minimum of steps in both cases.

From any vertex we can go to :
(current source + last step +1) and
(current source – last step -1)

- 5) If at any point, absolute value of our position exceeds the absolute value of our destination then it is intuitive that the shortest path is not possible from here. Hence we make the value

of steps INT_MAX, so that when i take the minimum of both possibilities, this one gets eliminated.

If we don't use this last step, the program enters into an INFINITE recursion and gives RUN TIME ERROR.

Below is the implementation of above idea. Note that the solution only counts steps.

C++

```
// C++ program to count number of
// steps to reach a point
#include<bits/stdc++.h>
using namespace std;

// Function to count number of steps
// required to reach a destination

// source -> source vertex
// step -> value of last step taken
// dest -> destination vertex
int steps(int source, int step, int dest)
{
    // base cases
    if (abs(source) > (dest))
        return INT_MAX;
    if (source == dest) return step;

    // at each point we can go either way

    // if we go on positive side
    int pos = steps(source + step + 1,
                    step + 1, dest);

    // if we go on negative side
    int neg = steps(source - step - 1,
                    step + 1, dest);

    // minimum of both cases
    return min(pos, neg);
}

// Driver code
int main()
{
    int dest = 11;
    cout << "No. of steps required to reach "
         << dest << " is "
         << steps(0, 0, dest);
    return 0;
}
```

Java

```
// Java program to count number of
// steps to reach a point
import java.io.*;

class GFG
{

    // Function to count number of steps
    // required to reach a destination

    // source -> source vertex
    // step -> value of last step taken
    // dest -> destination vertex
    static int steps(int source, int step,
                     int dest)
    {
        // base cases
        if (Math.abs(source) > (dest))
            return Integer.MAX_VALUE;

        if (source == dest)
            return step;

        // at each point we can go either way

        // if we go on positive side
        int pos = steps(source + step + 1,
                        step + 1, dest);

        // if we go on negative side
        int neg = steps(source - step - 1,
                        step + 1, dest);

        // minimum of both cases
        return Math.min(pos, neg);
    }

    // Driver Code
    public static void main(String[] args)
    {
        int dest = 11;
        System.out.println("No. of steps required"+
                           " to reach " + dest +
                           " is " + steps(0, 0, dest));
    }
}
```

```
// This code is contributed by Prerna Saini
```

Python3

```
# python program to count number of
# steps to reach a point
import sys

# Function to count number of steps
# required to reach a destination

# source -> source vertex
# step -> value of last step taken
# dest -> destination vertex
def steps(source, step, dest):

    #base cases
    if (abs(source) > (dest)) :
        return sys.maxsize

    if (source == dest):
        return step

    # at each point we can go
    # either way

    # if we go on positive side
    pos = steps(source + step + 1,
                step + 1, dest)

    # if we go on negative side
    neg = steps(source - step - 1,
                step + 1, dest)

    # minimum of both cases
    return min(pos, neg)

# Driver Code
dest = 11;
print("No. of steps required",
      " to reach " ,dest ,
      " is " , steps(0, 0, dest));

# This code is contributed by Sam007.
```

C#

```
// C# program to count number of
// steps to reach a point
using System;

class GFG
{
    // Function to count number of steps
    // required to reach a destination

    // source -> source vertex
    // step -> value of last step taken
    // dest -> destination vertex
    static int steps(int source, int step,
                     int dest)
    {
        // base cases
        if (Math.Abs(source) > (dest))
            return int.MaxValue;

        if (source == dest)
            return step;

        // at each point we can go either way

        // if we go on positive side
        int pos = steps(source + step + 1,
                        step + 1, dest);

        // if we go on negative side
        int neg = steps(source - step - 1,
                        step + 1, dest);

        // minimum of both cases
        return Math.Min(pos, neg);
    }

    // Driver Code
    public static void Main()
    {
        int dest = 11;
        Console.WriteLine("No. of steps required"+
                          " to reach " + dest +
                          " is " + steps(0, 0, dest));
    }
}
```

```
// This code is contributed by Sam007
```

PHP

```
<?php
// PHP program to count number
// of steps to reach a point

// Function to count number
// of steps required to reach
// a destination

// source -> source vertex
// step -> value of last step taken
// dest -> destination vertex
function steps($source, $step, $dest)
{
    // base cases
    if (abs($source) > ($dest))
        return PHP_INT_MAX;
    if ($source == $dest)
        return $step;

    // at each point we
    // can go either way

    // if we go on positive side
    $pos = steps($source + $step + 1,
                 $step + 1, $dest);

    // if we go on negative side
    $neg = steps($source - $step - 1,
                 $step + 1, $dest);

    // minimum of both cases
    return min($pos, $neg);
}

// Driver code
$dest = 11;
echo "No. of steps required to reach ",
     $dest, " is ", steps(0, 0, $dest);

// This code is contributed by aj_36
?>
```

Output :

No. of steps required to reach 11 is 5

Thanks to Arpit Thapar for providing above algorithm and implementation.

Optimized Solution : Find minimum moves to reach target on an infinite line

This article is contributed by Abhay. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Improved By : [Sam007](#), [jit_t](#)

Source

<https://www.geeksforgeeks.org/minimum-steps-to-reach-a-destination/>

Chapter 215

Minimum steps to reach end of array under constraints

Minimum steps to reach end of array under constraints - GeeksforGeeks

Given an array containing one digit numbers only, assuming we are standing at first index, we need to reach to end of array using minimum number of steps where in one step, we can jump to neighbor indices or can jump to a position with same value.

In other words, if we are at index i , then in one step you can reach to, $\text{arr}[i-1]$ or $\text{arr}[i+1]$ or $\text{arr}[K]$ such that $\text{arr}[K] = \text{arr}[i]$ (value of $\text{arr}[K]$ is same as $\text{arr}[i]$)

Examples:

```
Input : arr[] = {5, 4, 2, 5, 0}
Output : 2
Explanation : Total 2 step required.
We start from 5(0), in first step jump to next 5
and in second step we move to value 0 (End of arr[]).
```

```
Input : arr[] = [0, 1, 2, 3, 4, 5, 6, 7, 5, 4,
                 3, 6, 0, 1, 2, 3, 4, 5, 7]
Output : 5
Explanation : Total 5 step required.
0(0) -> 0(12) -> 6(11) -> 6(6) -> 7(7) ->
(18)
(inside parenthesis indices are shown)
```

This problem can be solved using [BFS](#). We can consider the given array as unweighted graph where every vertex has two edges to next and previous array elements and more edges to array elements with same values. Now for fast processing of third type of edges, we keep 10 [vectors](#) which store all indices where digits 0 to 9 are present. In above example, vector

corresponding to 0 will store [0, 12], 2 indices where 0 has occurred in given array. Another Boolean array is used, so that we don't visit same index more than once. As we are using BFS and BFS proceeds level by level, optimal minimum steps are guaranteed.

```
// C++ program to find minimum jumps to reach end
// of array
#include <bits/stdc++.h>
using namespace std;

// Method returns minimum step to reach end of array
int getMinStepToReachEnd(int arr[], int N)
{
    // visit boolean array checks whether current index
    // is previously visited
    bool visit[N];

    // distance array stores distance of current
    // index from starting index
    int distance[N];

    // digit vector stores indicies where a
    // particular number resides
    vector<int> digit[10];

    // In starting all index are unvisited
    memset(visit, false, sizeof(visit));

    // storing indices of each number in digit vector
    for (int i = 1; i < N; i++)
        digit[arr[i]].push_back(i);

    // for starting index distance will be zero
    distance[0] = 0;
    visit[0] = true;

    // Creating a queue and inserting index 0.
    queue<int> q;
    q.push(0);

    // loop untill queue in not empty
    while(!q.empty())
    {
        // Get an item from queue, q.
        int idx = q.front();           q.pop();

        // If we reached to last index break from loop
        if (idx == N-1)
            break;
    }
}
```

```
// Find value of dequeued index
int d = arr[idx];

// looping for all indices with value as d.
for (int i = 0; i<digit[d].size(); i++)
{
    int nextidx = digit[d][i];
    if (!visit[nextidx])
    {
        visit[nextidx] = true;
        q.push(nextidx);

        // update the distance of this nextidx
        distance[nextidx] = distance[idx] + 1;
    }
}

// clear all indices for digit d, because all
// of them are processed
digit[d].clear();

// checking condition for previous index
if (idx-1 >= 0 && !visit[idx - 1])
{
    visit[idx - 1] = true;
    q.push(idx - 1);
    distance[idx - 1] = distance[idx] + 1;
}

// checking condition for next index
if (idx + 1 < N && !visit[idx + 1])
{
    visit[idx + 1] = true;
    q.push(idx + 1);
    distance[idx + 1] = distance[idx] + 1;
}

// N-1th position has the final result
return distance[N - 1];
}

// driver code to test above methods
int main()
{
    int arr[] = {0, 1, 2, 3, 4, 5, 6, 7, 5,
                4, 3, 6, 0, 1, 2, 3, 4, 5, 7};
```

```
int N = sizeof(arr) / sizeof(int);
cout << getMinStepToReachEnd(arr, N);
return 0;
}
```

Output:

5

Source

<https://www.geeksforgeeks.org/minimum-steps-reach-end-array-constraints/>

Chapter 216

Minimum steps to reach target by a Knight Set 1

Minimum steps to reach target by a Knight Set 1 - GeeksforGeeks

Given a square chessboard of $N \times N$ size, the position of Knight and position of a target is given. We need to find out minimum steps a Knight will take to reach the target position.

Examples:

In above diagram Knight takes 3 step to reach from (4, 5) to (1, 1) (4, 5) \rightarrow (5, 3) \rightarrow (3, 2) \rightarrow (1, 1) as shown in diagram

This problem can be seen as shortest path in unweighted graph. Therefore we use [BFS](#) to solve this problem. We try all 8 possible positions where a Knight can reach from its position. If reachable position is not already visited and is inside the board, we push this state into queue with distance 1 more than its parent state. Finally we return distance of target position, when it gets pop out from queue.

Below code implements BFS for searching through cells, where each cell contains its coordinate and distance from starting node. In worst case, below code visits all cells of board, making worst-case time complexity as $O(N^2)$

```
// C++ program to find minimum steps to reach to
// specific cell in minimum moves by Knight
#include <bits/stdc++.h>
using namespace std;

// structure for storing a cell's data
```

```
struct cell
{
    int x, y;
    int dis;
    cell() {}
    cell(int x, int y, int dis) : x(x), y(y), dis(dis) {}

}

// Utility method returns true if (x, y) lies
// inside Board
bool isInside(int x, int y, int N)
{
    if (x >= 1 && x <= N && y >= 1 && y <= N)
        return true;
    return false;
}

// Method returns minimum step to reach target position
int minStepToReachTarget(int knightPos[], int targetPos[],
                        int N)
{
    // x and y direction, where a knight can move
    int dx[] = {-2, -1, 1, 2, -2, -1, 1, 2};
    int dy[] = {-1, -2, -2, -1, 1, 2, 2, 1};

    // queue for storing states of knight in board
    queue<cell> q;

    // push starting position of knight with 0 distance
    q.push(cell(knightPos[0], knightPos[1], 0));

    cell t;
    int x, y;
    bool visit[N + 1][N + 1];

    // make all cell unvisited
    for (int i = 1; i <= N; i++)
        for (int j = 1; j <= N; j++)
            visit[i][j] = false;

    // visit starting state
    visit[knightPos[0]][knightPos[1]] = true;

    // loop untill we have one element in queue
    while (!q.empty())
    {
        t = q.front();
        q.pop();
        ... // Process current cell and enqueue neighbors
    }
}
```

```
// if current cell is equal to target cell,  
// return its distance  
if (t.x == targetPos[0] && t.y == targetPos[1])  
    return t.dis;  
  
// loop for all reachable states  
for (int i = 0; i < 8; i++)  
{  
    x = t.x + dx[i];  
    y = t.y + dy[i];  
  
    // If reachable state is not yet visited and  
    // inside board, push that state into queue  
    if (isInside(x, y, N) && !visit[x][y]) {  
        visit[x][y] = true;  
        q.push(cell(x, y, t.dis + 1));  
    }  
}  
}  
  
// Driver code to test above methods  
int main()  
{  
    int N = 30;  
    int knightPos[] = {1, 1};  
    int targetPos[] = {30, 30};  
    cout << minStepToReachTarget(knightPos, targetPos, N);  
    return 0;  
}
```

Output:

20

Source

<https://www.geeksforgeeks.org/minimum-steps-reach-target-knight/>

Chapter 217

Move weighting scale alternate under given constraints

Move weighting scale alternate under given constraints - GeeksforGeeks

Given a weighting scale and an array of different positive weights where we have an infinite supply of each weight. Our task is to put weights on left and right pans of scale one by one in such a way that pans move to that side where weight is put i.e. each time, pans of scale moves to alternate sides.

- We are given another integer ‘steps’, times which we need to perform this operation.
- Another constraint is that we can’t put same weight consecutively i.e. if weight w is taken then in next step while putting the weight on opposite pan we can’t take w again.

Let weight array is [7, 11] and steps = 3
then 7, 11, 7 is the sequence in which
weights should be kept in order to move
scale alternatively.

Let another weight array is [2, 3, 5, 6]
and steps = 10 then, 3, 2, 3, 5, 6, 5, 3,
2, 3 is the sequence in which weights should
be kept in order to move scale alternatively.

This problem can be solved by doing [DFS](#) among scale states.

1. We traverse among various DFS states for the solution where each DFS state will correspond to actual difference value between left and right pans and current step count.

2. Instead of storing weights of both pans, we just store the difference residue value and each time chosen weight value should be greater than this difference and shouldn't be equal to previously chosen value of weight.
3. If it is, then we call the DFS method recursively with new difference value and one more step.

Please see below code for better understanding,

```
// C++ program to print weights for alternating
// the weighting scale
#include <bits/stdc++.h>
using namespace std;

// DFS method to traverse among states of weighting scales
bool dfs(int residue, int curStep, int wt[], int arr[],
          int N, int steps)
{
    // If we reach to more than required steps,
    // return true
    if (curStep > steps)
        return true;

    // Try all possible weights and choose one which
    // returns 1 afterwards
    for (int i = 0; i < N; i++)
    {
        /* Try this weight only if it is greater than
           current residue and not same as previous chosen
           weight */
        if (arr[i] > residue && arr[i] != wt[curStep - 1])
        {
            // assign this weight to array and recur for
            // next state
            wt[curStep] = arr[i];
            if (dfs(arr[i] - residue, curStep + 1, wt,
                    arr, N, steps))
                return true;
        }
    }

    // if any weight is not possible, return false
    return false;
}

// method prints weights for alternating scale and if
// not possible prints 'not possible'
void printWeightsOnScale(int arr[], int N, int steps)
```

```
{  
    int wt[steps];  
  
    // call dfs with current residue as 0 and current  
    // steps as 0  
    if (dfs(0, 0, wt, arr, N, steps))  
    {  
        for (int i = 0; i < steps; i++)  
            cout << wt[i] << " ";  
        cout << endl;  
    }  
    else  
        cout << "Not possible\n";  
}  
  
// Driver code to test above methods  
int main()  
{  
    int arr[] = {2, 3, 5, 6};  
    int N = sizeof(arr) / sizeof(int);  
  
    int steps = 10;  
    printWeightsOnScale(arr, N, steps);  
    return 0;  
}
```

Output:

2 3 2 3 5 6 5 3 2 3

Source

<https://www.geeksforgeeks.org/move-weighting-scale-alternate-given-constraints/>

Chapter 218

Multi Source Shortest Path in Unweighted Graph

Multi Source Shortest Path in Unweighted Graph - GeeksforGeeks

Suppose there are n towns connected by m bidirectional roads. There are s towns among them with a police station. We want to find out the distance of each town from the nearest police station. If the town itself has one the distance is 0.

Example:

```
Input :  
Number of Vertices = 6  
Number of Edges = 9  
Towns with Police Station : 1, 5  
Edges:  
1 2  
1 6  
2 6  
2 3  
3 6  
5 4  
6 5  
3 4  
5 3
```

```
Output :  
1 0  
2 1  
3 1  
4 1  
5 0
```

6 1

Naive Approach: We can loop through the vertices and from each vertex run a BFS to find the closest town with police station from that vertex. This will take $O(V.E)$.

Naive approach implementation using BFS from each vertex:

```
// cpp program to demonstrate distance to
// nearest source problem using BFS
// from each vertex
#include <bits/stdc++.h>
using namespace std;
#define N 100000 + 1
#define inf 1000000

// This array stores the distances of the
// vertices from the nearest source
int dist[N];

// a hash array where source[i] = 1
// means vertex i is a source
int source[N];

// The BFS Queue
// The pairs are of the form (vertex, distance
// from current source)
deque<pair<int, int> > BFSQueue;

// visited array for remembering visited vertices
int visited[N];

// The BFS function
void BFS(vector<int> graph[], int start)
{
    // clearing the queue
    while (!BFSQueue.empty())
        BFSQueue.pop_back();

    // push_back starting vertices
    BFSQueue.push_back({ start, 0 });

    while (!BFSQueue.empty()) {

        int s = BFSQueue.front().first;
        int d = BFSQueue.front().second;
        visited[s] = 1;
        BFSQueue.pop_front();

        for (int v : graph[s]) {
            if (visited[v] == 0) {
                BFSQueue.push_back({ v, d + 1 });
                dist[v] = d + 1;
            }
        }
    }
}
```

```

// stop at the first source we reach during BFS
if (source[s] == 1) {
    dist[start] = d;
    return;
}

// Pushing the adjacent unvisited vertices
// with distance from current source = this
// vertex's distance + 1
for (int i = 0; i < graph[s].size(); i++)
    if (visited[graph[s][i]] == 0)
        BFSQueue.push_back({ graph[s][i], d + 1 });
}
}

// This function calculates the distance of each
// vertex from nearest source
void nearestTown(vector<int> graph[], int n,
                  int sources[], int S)
{

    // resetting the source hash array
    for (int i = 1; i <= n; i++)
        source[i] = 0;
    for (int i = 0; i <= S - 1; i++)
        source[sources[i]] = 1;

    // loop through all the vertices and run
    // a BFS from each vertex to find the distance
    // to nearest town from it
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++)
            visited[j] = 0;
        BFS(graph, i);
    }

    // Printing the distances
    for (int i = 1; i <= n; i++)
        cout << i << " " << dist[i] << endl;
}

void addEdge(vector<int> graph[], int u, int v)
{
    graph[u].push_back(v);
    graph[v].push_back(u);
}

// Driver Code

```

```

int main()
{
    // Number of vertices
    int n = 6;

    vector<int> graph[n + 1];

    // Edges
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 6);
    addEdge(graph, 2, 6);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 6);
    addEdge(graph, 5, 4);
    addEdge(graph, 6, 5);
    addEdge(graph, 3, 4);
    addEdge(graph, 5, 3);

    // Sources
    int sources[] = { 1, 5 };

    int S = sizeof(sources) / sizeof(sources[0]);

    nearestTown(graph, n, sources, S);

    return 0;
}

```

Output:

```

1 0
2 1
3 1
4 1
5 0
6 1

```

Time Complexity: $O(V.E)$

Efficient Method A better method is to use the [Djikstra's algorithm](#) in a modified way. Let's consider one of the sources as the original source and the other sources to be vertices with 0 cost paths from the original source. Thus we push all the sources into the Djikstra Queue with distance = 0, and the rest of the vertices with distance = infinity. The minimum distance of each vertex from the original source now calculated using the Djikstra's Algorithm are now essentially the distances from the nearest source.

Explanation: The C++ implementation uses a [set of pairs](#) (distance from the source, vertex) sorted according to the distance from the source. Initially, the set contains the sources with distance = 0 and all the other vertices with distance = infinity.

On each step, we will go to the vertex with minimum distance(d) from source, i.e, the first element of the set (the source itself in the first step with distance = 0). We go through all it's adjacent vertices and if the distance of any vertex is $> d + 1$ we replace its entry in the set with the new distance. Then we remove the current vertex from the set. We continue this until the set is empty.

The idea is there cannot be a shorter path to the vertex at the front of the set than the current one since any other path will be a sum of a longer path (\geq it's length) and a non-negative path length (unless we are considering negative edges).

Since all the sources have a distance = 0, in the beginning, the adjacent non-source vertices will get a distance = 1. All vertices will get distance = distance from their nearest source.

Implementation of Efficient Approach:

```
// cpp program to demonstrate
// multi-source BFS
#include <bits/stdc++.h>
using namespace std;
#define N 100000 + 1
#define inf 1000000

// This array stores the distances of the vertices
// from the nearest source
int dist[N];

// This Set contains the vertices not yet visited in
// increasing order of distance from the nearest source
// calculated till now
set<pair<int, int> > Q;

// Util function for Multi-Source BFS
void multiSourceBFSUtil(vector<int> graph[], int s)
{
    set<pair<int, int> >::iterator it;
    int i;
    for (i = 0; i < graph[s].size(); i++) {
        int v = graph[s][i];
        if (dist[s] + 1 < dist[v]) {

            // If a shorter path to a vertex is
            // found than the currently stored
            // distance replace it in the Q
            it = Q.find({ dist[v], v });
            Q.erase(it);
            dist[v] = dist[s] + 1;
            Q.insert({ dist[v], v });
        }
    }
}

// Stop when the Q is empty -> All
```

```

// vertices have been visited. And we only
// visit a vertex when we are sure that a
// shorter path to that vertex is not
// possible
if (Q.size() == 0)
    return;

// Go to the first vertex in Q
// and remove it from the Q
it = Q.begin();
int next = it->second;
Q.erase(it);

multiSourceBFSUtil(graph, next);
}

// This function calculates the distance of
// each vertex from nearest source
void multiSourceBFS(vector<int> graph[], int n,
                    int sources[], int S)
{
    // a hash array where source[i] = 1
    // means vertex i is a source
    int source[n + 1];

    for (int i = 1; i <= n; i++)
        source[i] = 0;
    for (int i = 0; i <= S - 1; i++)
        source[sources[i]] = 1;

    for (int i = 1; i <= n; i++) {
        if (source[i]) {
            dist[i] = 0;
            Q.insert({ 0, i });
        }
        else {
            dist[i] = inf;
            Q.insert({ inf, i });
        }
    }
}

set<pair<int, int> >::iterator itr;

// Get the vertex with lowest distance,
itr = Q.begin();

// currently one of the souces with distance = 0
int start = itr->second;

```

```
multiSourceBFSUtil(graph, start);

// Printing the distances
for (int i = 1; i <= n; i++)
    cout << i << " " << dist[i] << endl;
}

void addEdge(vector<int> graph[], int u, int v)
{
    graph[u].push_back(v);
    graph[v].push_back(u);
}

// Driver Code
int main()
{
    // Number of vertices
    int n = 6;

    vector<int> graph[n + 1];

    // Edges
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 6);
    addEdge(graph, 2, 6);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 6);
    addEdge(graph, 5, 4);
    addEdge(graph, 6, 5);
    addEdge(graph, 3, 4);
    addEdge(graph, 5, 3);

    // Sources
    int sources[] = { 1, 5 };

    int S = sizeof(sources) / sizeof(sources[0]);

    multiSourceBFS(graph, n, sources, S);

    return 0;
}
```

Output:

```
1 0
2 1
```

3 1
4 1
5 0
6 1

Time Complexity: $O(E \cdot \log V)$

Source

<https://www.geeksforgeeks.org/multi-source-shortest-path-in-unweighted-graph/>

Chapter 219

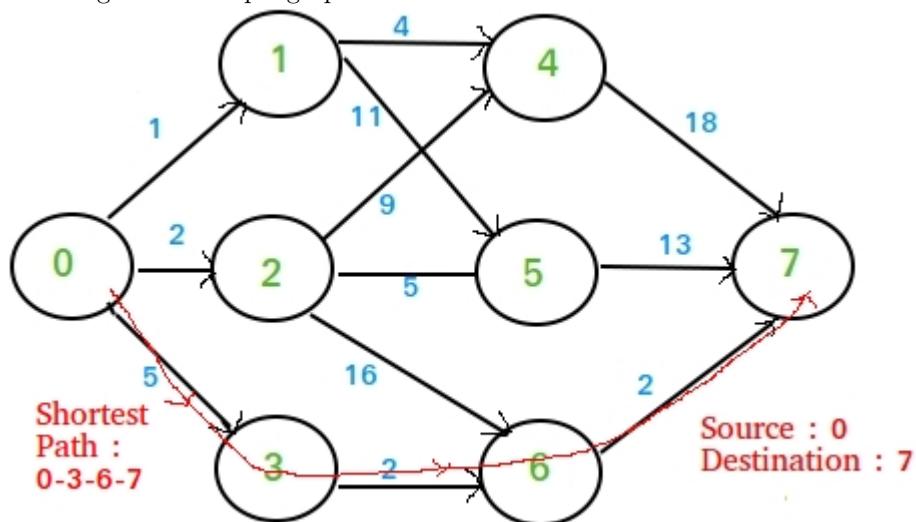
Multistage Graph (Shortest Path)

Multistage Graph (Shortest Path) - GeeksforGeeks

A **Multistage graph** is a directed graph in which the nodes can be divided into a set of stages such that all edges are from a stage to next stage only (In other words there is no edge between vertices of same stage and from a vertex of current stage to previous stage).

We are given a multistage graph, a source and a destination, we need to find shortest path from source to destination. By convention, we consider source at stage 1 and destination as last stage.

Following is an example graph we will consider in this article :-



Now there are various strategies we can apply :-

- The **Brute force** method of finding all possible paths between Source and Destination and then finding the minimum. That's the **WORST** possible strategy.
- **Dijkstra's Algorithm** of Single Source shortest paths. This method will find shortest paths from source to all other nodes which is not required in this case. So it will take a lot of time and it doesn't even use the **SPECIAL** feature that this **MULTI-STAGE** graph has.
- **Simple Greedy Method** – At each node, choose the shortest outgoing path. If we apply this approach to the example graph give above we get the solution as $1 + 4 + 18 = 23$. But a quick look at the graph will show much shorter paths available than 23. So the greedy method fails !
- The best option is Dynamic Programming. So we need to find **Optimal Substructure, Recursive Equations and Overlapping Sub-problems**.

Optimal Substructure and Recursive Equation :-

We define the notation :- $M(x, y)$ as the minimum cost to T (target node) from Stage x , Node y .

Shortest distance from stage 1, node 0 to destination, i.e., 7 is $M(1, 0)$.

```
// From 0, we can go to 1 or 2 or 3 to
// reach 7.
M(1, 0) = min(1 + M(2, 1),
                2 + M(2, 2),
                5 + M(2, 3))
```

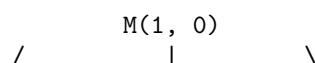
This means that our problem of $0 \rightarrow 7$ is now sub-divided into 3 sub-problems :-

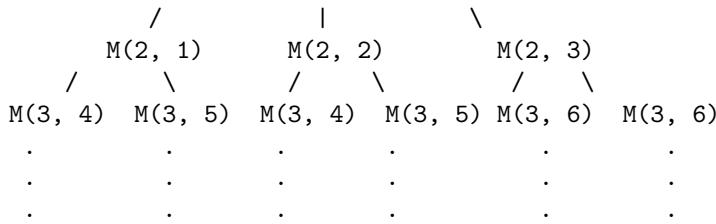
So if we have total ' n ' stages and target as T , then the stopping condition will be :-
 $M(n-1, i) = i \rightarrow T + M(n, T) = i \rightarrow T$

Recursion Tree and Overlapping Sub-Problems:-

So, the hierarchy of $M(x, y)$ evaluations will look something like this :-

In $M(i, j)$, i is stage number and j is node number





So, here we have drawn a very small part of the Recursion Tree and we can already see Overlapping Sub-Problems. We can largely reduce the number of $M(x, y)$ evaluations using Dynamic Programming.

Implementation details:

The below implementation assumes that nodes are numbered from 0 to $N-1$ from first stage (source) to last stage (destination). We also assume that the input graph is multistage.

```

// CPP program to find shortest distance
// in a multistage graph.
#include<bits/stdc++.h>
using namespace std;

#define N 8
#define INF INT_MAX

// Returns shortest distance from 0 to
// N-1.
int shortestDist(int graph[N][N]) {

    // dist[i] is going to store shortest
    // distance from node i to node N-1.
    int dist[N];

    dist[N-1] = 0;

    // Calculating shortest path for
    // rest of the nodes
    for (int i = N-2 ; i >= 0 ; i--)
    {

        // Initialize distance from i to
        // destination (N-1)
        dist[i] = INF;

        // Check all nodes of next stages
        // to find shortest distance from
        // i to N-1.
        for (int j = i ; j < N ; j++)
        {
  
```

```
// Reject if no edge exists
if (graph[i][j] == INF)
    continue;

// We apply recursive equation to
// distance to target through j.
// and compare with minimum distance
// so far.
dist[i] = min(dist[i], graph[i][j] +
                dist[j]);
}

}

return dist[0];
}

// Driver code
int main()
{
    // Graph stored in the form of an
    // adjacency Matrix
    int graph[N][N] =
        {{INF, 1, 2, 5, INF, INF, INF, INF},
         {INF, INF, INF, INF, 4, 11, INF, INF},
         {INF, INF, INF, INF, 9, 5, 16, INF},
         {INF, INF, INF, INF, INF, INF, 2, INF},
         {INF, INF, INF, INF, INF, INF, INF, 18},
         {INF, INF, INF, INF, INF, INF, INF, 13},
         {INF, INF, INF, INF, INF, INF, INF, 2}};

    cout << shortestDist(graph);
    return 0;
}
```

Output:

9

Time Complexity : $O(n^2)$

Source

<https://www.geeksforgeeks.org/multistage-graph-shortest-path/>

Chapter 220

NetworkX : Python software package for study of complex networks

NetworkX : Python software package for study of complex networks - GeeksforGeeks

NetworkX is a Python language software package for the creation, manipulation, and study of the structure, dynamics, and function of complex networks. It is used to study large complex networks represented in form of graphs with nodes and edges. Using networkx we can load and store complex networks. We can generate many types of random and classic networks, analyze network structure, build network models, design new network algorithms and draw networks.

Installation of the package:

```
pip install networkx
```

Creating Nodes

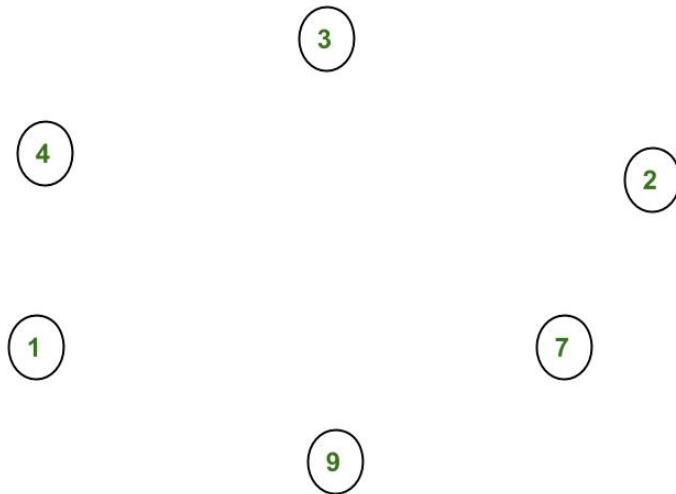
Add one node at a time:

```
G.add_node(1)
```

Add a list of nodes:

```
G.add_nodes_from([2,3])
```

Let us create nodes in the graph G. After adding nodes 1, 2, 3, 4, 7, 9



Creating Edges

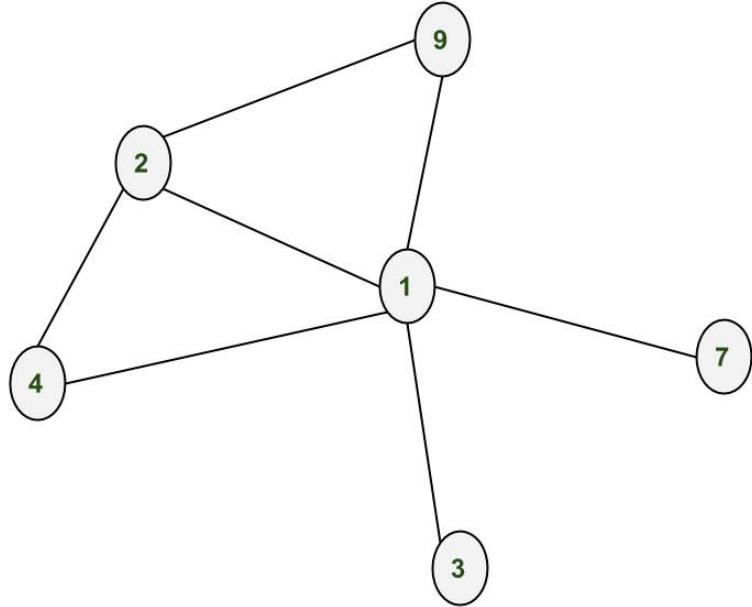
Adding one edge at a time:

```
G.add_edge(1,2)
G.add_edge(3,1)
G.add_edge(2,4)
G.add_edge(4,1)
G.add_edge(9,1)
```

Adding a list of edges:

```
G.add_edges_from([(1,2),(1,3)])
```

After adding edges (1,2), (3,1), (2,4), (4,1), (9,1), (1,7), (2,9)

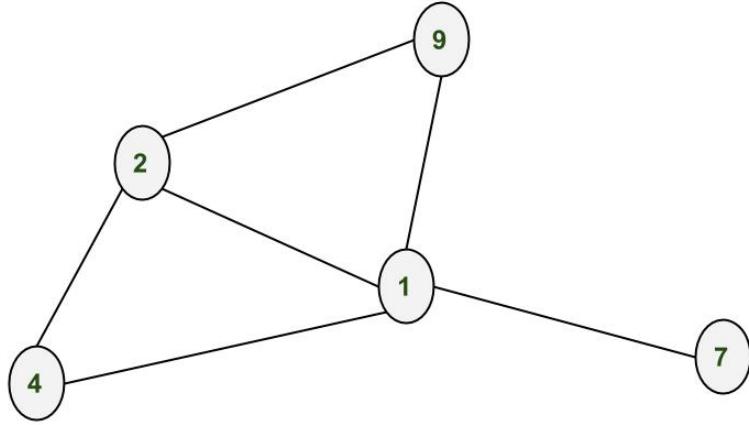


Removing Nodes and Edges

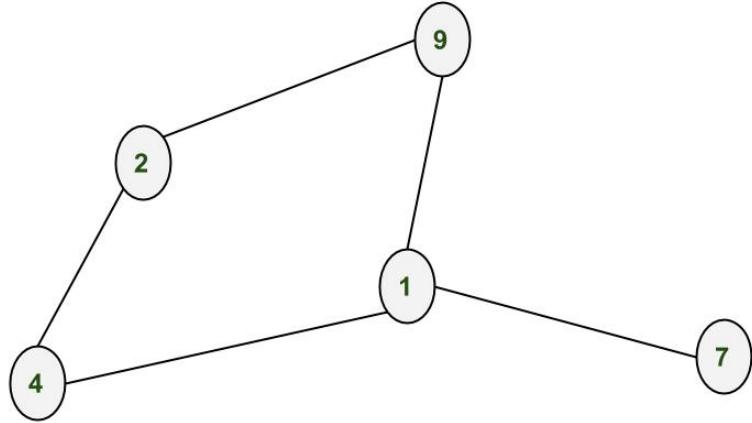
One can demolish the graph using any of these functions:

`Graph.remove_node()`, `Graph.remove_nodes_from()`,
`Graph.remove_edge()` and `Graph.remove_edges_from()`

After removing node 3



After removing edge (1,2)



```
# Python program to create an undirected
# graph and add nodes and edges to a graph

# To import package
import networkx

# To create an empty undirected graph
G = networkx.Graph()

# To add a node
G.add_node(1)
G.add_node(2)
G.add_node(3)
G.add_node(4)
G.add_node(7)
G.add_node(9)

# To add an edge
# Note graph is undirected
# Hence order of nodes in edge doesn't matter
G.add_edge(1,2)
G.add_edge(3,1)
```

```
G.add_edge(2,4)
G.add_edge(4,1)
G.add_edge(9,1)
G.add_edge(1,7)
G.add_edge(2,9)

# To get all the nodes of a graph
node_list = G.nodes()
print("#1")
print(node_list)

# To get all the edges of a graph
edge_list = G.edges()
print("#2")
print(edge_list)

# To remove a node of a graph
G.remove_node(3)
node_list = G.nodes()
print("#3")
print(node_list)

# To remove an edge of a graph
G.remove_edge(1,2)
edge_list = G.edges()
print("#4")
print(edge_list)

# To find number of nodes
n = G.number_of_nodes()
print("#5")
print(n)

# To find number of edges
m = G.number_of_edges()
print("#6")
print(m)

# To find degree of a node
# d will store degree of node 2
d = G.degree(2)
print("#7")
print(d)

# To find all the neighbor of a node
neighbor_list = G.neighbors(2)
print("#8")
print(neighbor_list)
```

```
#To delete all the nodes and edges  
G.clear()
```

Output :

```
#1  
[1, 2, 3, 4, 7, 9]  
#2  
[(1, 9), (1, 2), (1, 3), (1, 4), (1, 7), (2, 4), (2, 9)]  
#3  
[1, 2, 4, 7, 9]  
#4  
[(1, 9), (1, 4), (1, 7), (2, 4), (2, 9)]  
#5  
5  
#6  
5  
#7  
2  
#8  
[4, 9]
```

In the next post, we'll be discussing how to create weighted graphs, directed graphs, multi graphs. How to draw graphs. In later posts we'll see how to use inbuilt functions like Depth fist search aka dfs, breadth first search aka BFS, dijkstra's shortest path algorithm.

Reference : [Networxx at Github](#)

Source

<https://www.geeksforgeeks.org/networkx-python-software-package-study-complex-networks/>

Chapter 221

Number of cyclic elements in an array where we can jump according to value

Number of cyclic elements in an array where we can jump according to value - GeeksforGeeks

Given a array arr[] of n integers. For every value arr[i], we can move to arr[i] + 1 clockwise considering array elements in cycle. We need to count cyclic elements in the array. An element is cyclic if starting from it and moving to arr[i] + 1 leads to same element.

Examples:

```
Input : arr[] = {1, 1, 1, 1}
Output : 4
All 4 elements are cyclic elements.
1 -> 3 -> 1
2 -> 4 -> 2
3 -> 1 -> 3
4 -> 2 -> 4
```

```
Input : arr[] = {3, 0, 0, 0}
Output : 1
There is one cyclic point 1,
1 -> 1
The path covered starting from 2 is
2 -> 3 -> 4 -> 1 -> 1.
```

```
The path covered starting from 3 is
2 -> 3 -> 4 -> 1 -> 1.
```

```
The path covered starting from 4 is
```

4 -> 1 -> 1

One **simple solution** is to check all elements one by one. We follow simple path starting from every element arr[i], we go to arr[i] + 1. If we come back to a visited element other than arr[i], we do not count arr[i]. Time complexity of this solution is O(n²)

An **efficient solution** is based on below steps.

- 1) Create a directed graph using array indexes as nodes. We add an edge from i to node (arr[i] + 1)%n.
- 2) Once a graph is created we find all [strongly connected components using Kosaraju's Algorithm](#)
- 3) We finally return sum of counts of nodes in individual strongly connected component.

```
// CPP program to count cyclic points
// in an array using Kosaraju's Algorithm
#include <bits/stdc++.h>
using namespace std;

// Most of the code is taken from below link
// https://www.geeksforgeeks.org/strongly-connected-components/
class Graph {
    int V;
    list<int>* adj;
    void fillOrder(int v, bool visited[], stack<int>& Stack);
    int DFSUtil(int v, bool visited[]);
public:
    Graph(int V);
    void addEdge(int v, int w);
    int countSCCNodes();
    Graph getTranspose();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

// Counts number of nodes reachable
// from v
int Graph::DFSUtil(int v, bool visited[])
{
    visited[v] = true;
    int ans = 1;
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            ans += DFSUtil(*i, visited);
    return ans;
}
```

```
        if (!visited[*i])
            ans += DFSUtil(*i, visited);
    return ans;
}

Graph Graph::getTranspose()
{
    Graph g(V);
    for (int v = 0; v < V; v++) {
        list<int>::iterator i;
        for (i = adj[v].begin(); i != adj[v].end(); ++i)
            g.adj[*i].push_back(v);
    }
    return g;
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
}

void Graph::fillOrder(int v, bool visited[], stack<int>& Stack)
{
    visited[v] = true;
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            fillOrder(*i, visited, Stack);
    Stack.push(v);
}

// This function mainly returns total count of
// nodes in individual SCCs using Kosaraju's
// algorithm.
int Graph::countSCCNodes()
{
    int res = 0;
    stack<int> Stack;
    bool* visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            fillOrder(i, visited, Stack);
    Graph gr = getTranspose();
    for (int i = 0; i < V; i++)
        visited[i] = false;
```

```
while (Stack.empty() == false) {
    int v = Stack.top();
    Stack.pop();
    if (visited[v] == false) {
        int ans = gr.DFSUtil(v, visited);
        if (ans > 1)
            res += ans;
    }
}
return res;
}

// Returns count of cyclic elements in arr[]
int countCyclic(int arr[], int n)
{
    int res = 0;

    // Create a graph of array elements
    Graph g(n + 1);

    for (int i = 1; i <= n; i++) {
        int x = arr[i-1];

        // If i + arr[i-1] jumps beyond last
        // element, we take mod considering
        // cyclic array
        int v = (x + i) % n + 1;

        // If there is a self loop, we
        // increment count of cyclic points.
        if (i == v)
            res++;

        g.addEdge(i, v);
    }

    // Add nodes of strongly connected components
    // of size more than 1.
    res += g.countSCCNodes();

    return res;
}

// Driver code
int main()
{
    int arr[] = {1, 1, 1, 1};
    int n = sizeof(arr)/sizeof(arr[0]);
```

Chapter 221. Number of cyclic elements in an array where we can jump according to value

```
cout << countCyclic(arr, n);
return 0;
}
```

Output:

4

Time Complexity : $O(n)$

Auxiliary space : $O(n)$ Note that there are only $O(n)$ edges.

Source

<https://www.geeksforgeeks.org/number-of-cyclic-elements-in-an-array-where-we-can-jump-according-to-value/>

Chapter 222

Number of groups formed in a graph of friends

Number of groups formed in a graph of friends - GeeksforGeeks

Given n friends and their friendship relations, find the total number of groups that exist. And the number of ways of new groups that can be formed consisting of people from every existing group.

If no relation is given for any person then that person has no group and singularly forms a group. If a is a friend of b and b is a friend of c, then a b and c form a group.

Examples:

```
Input : Number of people = 6
        Relations : 1 - 2, 3 - 4
                     and 5 - 6
Output: Number of existing Groups = 3
        Number of new groups that can
        be formed = 8
Explanation: The existing groups are
(1, 2), (3, 4), (5, 6). The new 8 groups
that can be formed by considering a
member of every group are (1, 3, 5),
(1, 3, 6), (1, 4, 5), (1, 4, 6), (2,
3, 5), (2, 3, 6), (2, 4, 5) and (2, 4,
6).
```

```
Input: Number of people = 6
      Relations : 1 - 2 and 2 - 3
Output: Number of existing Groups = 2
      Number of new groups that can
      be formed = 3
```

Explanation: The existing groups are (1, 2, 3) and (4). The new groups that can be formed by considering a member of every group are (1, 4), (2, 4), (3, 4).

To count number of groups, we need to simply count [connected components in the given undirected graph](#). Counting connected components can be easily done using [DFS](#) or [BFS](#). Since this is an undirected graph, the number of times a Depth First Search starts from an unvisited vertex for every friend is equal to the number of groups formed.

To count number of ways in which we form new groups can be done using simply formula which is $(N_1)*(N_2)*\dots*(N_n)$ where N_i is the no of people in i -th group.

```
// CPP program to count number of existing
// groups and number of new groups that can
// be formed.
#include <bits/stdc++.h>
using namespace std;

class Graph {
    int V; // No. of vertices

    // Pointer to an array containing
    // adjacency lists
    list<int*>* adj;

    int countUtil(int v, bool visited[]);
public:
    Graph(int V); // Constructor

    // function to add an edge to graph
    void addRelation(int v, int w);
    void countGroups();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

// Adds a relation as a two way edge of
// undirected graph.
void Graph::addRelation(int v, int w)
{
    // Since indexing is 0 based, reducing
    // edge numbers by 1.
    v--;
    adj[v].push_back(w);
    adj[w].push_back(v);
}

int Graph::countUtil(int v, bool visited[])
{
    if (visited[v])
        return 0;
    visited[v] = true;
    int count = 1;
    for (list<int*>::iterator i = adj[v].begin(); i != adj[v].end(); ++i)
        count += countUtil(**i, visited);
    return count;
}

int Graph::countGroups()
{
    int count = 0;
    for (int i = 0; i < V; i++)
        count += countUtil(i, new bool[V]);
    return count;
}
```

```
w--;
adj[v].push_back(w);
adj[w].push_back(v);
}

// Returns count of not visited nodes reachable
// from v using DFS.
int Graph::countUtil(int v, bool visited[])
{
    int count = 1;
    visited[v] = true;
    for (auto i=adj[v].begin(); i!=adj[v].end(); ++i)
        if (!visited[*i])
            count = count + countUtil(*i, visited);
    return count;
}

// A DFS based function to Count number of
// existing groups and number of new groups
// that can be formed using a member of
// every group.
void Graph::countGroups()
{
    // Mark all the vertices as not visited
    bool* visited = new bool[V];
    memset(visited, 0, V*sizeof(int));

    int existing_groups = 0, new_groups = 1;
    for (int i = 0; i < V; i++)
    {
        // If not in any group.
        if (visited[i] == false)
        {
            existing_groups++;

            // Number of new groups that
            // can be formed.
            new_groups = new_groups *
                countUtil(i, visited);
        }
    }

    if (existing_groups == 1)
        new_groups = 0;

    cout << "No. of existing groups are "
        << existing_groups << endl;
    cout << "No. of new groups that can be"
```

```
        " formed are " << new_groups
        << endl;
}

// Driver code
int main()
{
    int n = 6;

    // Create a graph given in the above diagram
    Graph g(n); // total 6 people
    g.addRelation(1, 2); // 1 and 2 are friends
    g.addRelation(3, 4); // 3 and 4 are friends
    g.addRelation(5, 6); // 5 and 6 are friends

    g.countGroups();

    return 0;
}
```

Output:

```
No. of existing groups are 3
No. of new groups that can be formed are 8
```

Time complexity: $O(N + R)$ where N is the number of people and R is the number of relations.

Source

<https://www.geeksforgeeks.org/number-groups-formed-graph-friends/>

Chapter 223

Number of loops of size k starting from a specific node

Number of loops of size k starting from a specific node - GeeksforGeeks

Given two positive integer n , k . Consider an undirected complete connected graph of n nodes in a complete connected graph. The task is to calculate the number of ways in which one can start from any node and return to it by visiting K nodes.

Examples:

Input : $n = 3$, $k = 3$
Output : 2

Input : $n = 4$, $k = 2$
Output : 3

Lets $f(n, k)$ be a function which return number of ways in which one can start from any node and return to it by visiting K nodes.

If we start and end from one node, then we have $K - 1$ choices to make for the intermediate nodes since we have already chosen one node in the beginning. For each intermediate choice, you have $n - 1$ options. So, this will yield $(n - 1)^{k - 1}$ but then we have to remove all the choices cause smaller loops, so we subtract $f(n, k - 1)$.

So, recurrence relation becomes,

$f(n, k) = (n - 1)^{k - 1} - f(n, k - 1)$ with base case $f(n, 2) = n - 1$.

On expanding,

$f(n, k) = (n - 1)^{k - 1} - (n - 1)^{k - 2} + (n - 1)^{k - 3} \dots (n - 1)$ (say eqn 1)

Dividing $f(n, k)$ by $(n - 1)$,

$f(n, k)/(n - 1) = (n - 1)^{k - 2} - (n - 1)^{k - 3} + (n - 1)^{k - 4} \dots 1$ (say eqn 2)

On adding eqn 1 and eqn 2,

$$f(n, k) + f(n, k)/(n - 1) = (n - 1)^{k - 1} + (-1)^k$$

$$f(n, k) * ((n - 1) + 1)/(n - 1) = (n - 1)^{k - 1} + (-1)^k$$

$$\frac{f(n, k) + f(n, k)/(n - 1)}{f(n, k) * ((n - 1) + 1)/(n - 1)} = \frac{(n - 1)^{k - 1} + (-1)^k}{(n - 1)^{k - 1} + (-1)^k}$$

Below is the implementation of this approach:

C++

```
// C++ Program to find number of cycles of length
// k in a graph with n nodes.
#include <bits/stdc++.h>
using namespace std;

// Return the Number of ways from a
// node to make a loop of size K in undirected
// complete connected graph of N nodes
int num0fways(int n, int k)
{
    int p = 1;

    if (k % 2)
        p = -1;

    return (pow(n - 1, k) + p * (n - 1)) / n;
}

// Driven Program
int main()
{
    int n = 4, k = 2;
    cout << num0fways(n, k) << endl;
    return 0;
}
```

Java

```
// Java Program to find number of
// cycles of length k in a graph
// with n nodes.
public class GFG {

    // Return the Number of ways
    // from a node to make a loop
    // of size K in undirected
    // complete connected graph of
```

```
// N nodes
static int num0fways(int n, int k)
{
    int p = 1;

    if (k % 2 != 0)
        p = -1;

    return (int)(Math.pow(n - 1, k)
                 + p * (n - 1)) / n;
}

// Driver code
public static void main(String args[])
{
    int n = 4, k = 2;

    System.out.println(num0fways(n, k));
}
}

// This code is contributed by Sam007.
```

Python3

```
# python Program to find number of
# cycles of length k in a graph
# with n nodes.

# Return the Number of ways from a
# node to make a loop of size K in
# undirected complete connected
# graph of N nodes
def num0fways(n,k):

    p = 1

    if (k % 2):
        p = -1

    return (pow(n - 1, k) +
            p * (n - 1)) / n

# Driver code
n = 4
k = 2
print (num0fways(n, k))
```

```
# This code is contributed by Sam007.
```

C#

```
// C# Program to find number of cycles
// of length k in a graph with n nodes.
using System;

class GFG {

    // Return the Number of ways from
    // a node to make a loop of size
    // K in undirected complete
    // connected graph of N nodes
    static int numOfways(int n, int k)
    {
        int p = 1;

        if (k % 2 != 0)
            p = -1;

        return (int)(Math.Pow(n - 1, k)
                    + p * (n - 1)) / n;
    }

    // Driver code
    static void Main()
    {
        int n = 4, k = 2;

        Console.WriteLine( numOfways(n, k) );
    }
}

// This code is contributed by Sam007.
```

PHP

```
<?php
// PHP Program to find number
// of cycles of length
// k in a graph with n nodes.

// Return the Number of ways from a
// node to make a loop of size K
// in undirected complete connected
// graph of N nodes
```

```
function num0fways( $n, $k)
{
$p = 1;

if ($k % 2)
$p = -1;

return (pow($n - 1, $k) +
$p * ($n - 1)) / $n;
}

// Driver Code
$n = 4;
$k = 2;
echo num0fways($n, $k);

// This code is contributed by vt_m.
?>
```

Output:

3

Improved By : [Sam007](#), [vt_m](#)

Source

<https://www.geeksforgeeks.org/number-ways-node-make-loop-size-k-undirected-complete-connected-graph-n-nodes/>

Chapter 224

Number of pair of positions in matrix which are not accessible

Number of pair of positions in matrix which are not accessible - GeeksforGeeks

Given a positive integer N . Consider a matrix of $N \times N$. No cell can be accessible from any other cell, except the given pair cell in the form of $(x_1, y_1), (x_2, y_2)$ i.e there is a path (accessible) between (x_2, y_2) to (x_1, y_1) . The task is to find the count of pairs $(a_1, b_1), (a_2, b_2)$ such that cell (a_2, b_2) is not accessible from (a_1, b_1) .

Examples:

```
Input : N = 2
Allowed path 1: (1, 1) (1, 2)
Allowed path 2: (1, 2) (2, 2)
Output : 6
Cell (2, 1) is not accessible from any cell
and no cell is accessible from it.
```

```
(1, 1) - (2, 1)
(1, 2) - (2, 1)
(2, 2) - (2, 1)
(2, 1) - (1, 1)
(2, 1) - (1, 2)
(2, 1) - (2, 2)
```

Consider each cell as a node, numbered from 1 to N^2 . Each cell (x, y) can be mapped to number using $(x - 1)N + y$. Now, consider each given allowed path as an edge between nodes. This will form a disjoint set of the connected component. Now, using [Depth First Traversal](#) or [Breadth First Traversal](#), we can easily find the number of nodes or size of a connected component, say x . Now, count of non-accessible paths are $x*(N^2 - x)$. This way we can find non-accessible paths for each connected path.

Below is C++ implementation of this approach:

```
// C++ program to count number of pair of positions
// in matrix which are not accessible
#include<bits/stdc++.h>
using namespace std;

// Counts number of vertices connected in a component
// containing x. Stores the count in k.
void dfs(vector<int> graph[], bool visited[],
          int x, int *k)
{
    for (int i = 0; i < graph[x].size(); i++)
    {
        if (!visited[graph[x][i]])
        {
            // Incrementing the number of node in
            // a connected component.
            (*k)++;

            visited[graph[x][i]] = true;
            dfs(graph, visited, graph[x][i], k);
        }
    }
}

// Return the number of count of non-accessible cells.
int countNonAccessible(vector<int> graph[], int N)
{
    bool visited[N*N + N];
    memset(visited, false, sizeof(visited));

    int ans = 0;
    for (int i = 1; i <= N*N; i++)
    {
        if (!visited[i])
        {
            visited[i] = true;

            // Initialize count of connected
            // vertices found by DFS starting
            // from i.
            int k = 1;
            dfs(graph, visited, i, &k);

            // Update result
            ans += k * (N*N - k);
        }
    }
}
```

```
        }
        return ans;
    }

// Inserting the edge between edge.
void insertpath(vector<int> graph[], int N, int x1,
                int y1, int x2, int y2)
{
    // Mapping the cell coordinate into node number.
    int a = (x1 - 1) * N + y1;
    int b = (x2 - 1) * N + y2;

    // Inserting the edge.
    graph[a].push_back(b);
    graph[b].push_back(a);
}

// Driven Program
int main()
{
    int N = 2;

    vector<int> graph[N*N + 1];

    insertpath(graph, N, 1, 1, 1, 2);
    insertpath(graph, N, 1, 2, 2, 2);

    cout << countNonAccessible(graph, N) << endl;
    return 0;
}
```

Output:

6

Time Complexity : $O(N^2)$.

Source

<https://www.geeksforgeeks.org/number-pair-positions-matrix-not-accessible/>

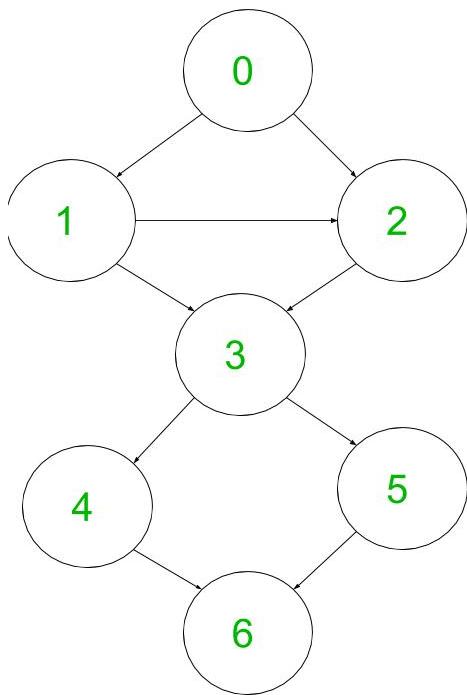
Chapter 225

Number of shortest paths in an unweighted and directed graph

Number of shortest paths in an unweighted and directed graph - GeeksforGeeks

Given an unweighted directed graph, can be cyclic or acyclic. Print the number of shortest paths from a given vertex to each of the vertices. For example consider the below graph. There is one shortest path vertex 0 to vertex 0 (from each vertex there is a single shortest path to itself), one shortest path between vertex 0 to vertex 2 ($0 \rightarrow 2$), and there are 4 different shortest paths from vertex 0 to vertex 6:

1. $0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 6$
2. $0 \rightarrow 1 \rightarrow 3 \rightarrow 5 \rightarrow 6$
3. $0 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6$
4. $0 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6$



The idea is to use [BFS](#). We use two arrays called `dist[]` and `paths[]`, `dist[]` represents the shortest distances from source vertex, and `paths[]` represents the number of different shortest paths from the source vertex to each of the vertices. Initially all the elements in `dist[]` are infinity except source vertex which is equal to 0, since the distance to source vertex from itself is 0, and all the elements in `paths[]` are 0 except source vertex which is equal to 1, since each vertex has a single shortest path to itself. after that, we start traversing the graph using BFS manner.

Then, for every neighbor Y of each vertex X do:

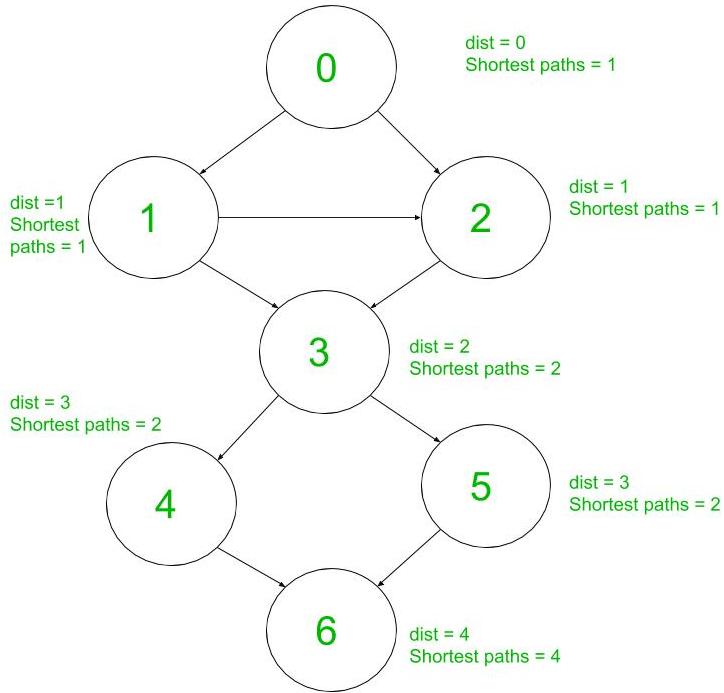
1) if $\text{dist}[Y] > \text{dist}[X] + 1$ decrease the $\text{dist}[Y]$ to $\text{dist}[X] + 1$ and assign the number of paths of vertex X to number of paths of vertex Y.

2) else if $\text{dist}[Y] = \text{dist}[X] + 1$, then add the number of paths of vertex X to the number of paths of vertex Y.

For example:

Let's take a look at the below graph. The source vertex is 0. Suppose we traverse on vertex 2, we check all its neighbors, which is only 3. since vertex 3 was already visited when we were traversed vertex 1, $\text{dist}[3] = 2$ and $\text{paths}[3] = 1$. The second condition is true, so it means that additional shortest paths have been found, so we add to the number of paths of vertex 3, the number of paths of vertex 2.

The equal condition happens when we traverse on vertex 5:



```

// CPP program to count number of shortest
// paths from a given source to every other
// vertex using BFS.
#include <bits/stdc++.h>
using namespace std;

// Traverses graph in BFS manner. It fills
// dist[] and paths[]
void BFS(vector<int> adj[], int src, int dist[],
         int paths[], int n)
{
    bool visited[n];
    for (int i = 0; i < n; i++)
        visited[i] = false;
    dist[src] = 0;
    paths[src] = 1;

    queue <int> q;
    q.push(src);
    visited[src] = true;
    while (!q.empty())
    {
        int curr = q.front();
        q.pop();
        for (int i = 0; i < adj[curr].size(); i++)
            if (!visited[adj[curr][i]])
            {
                dist[adj[curr][i]] = dist[curr] + 1;
                paths[adj[curr][i]] = paths[curr];
                q.push(adj[curr][i]);
                visited[adj[curr][i]] = true;
            }
    }
}
  
```

```

// For all neighbors of current vertex do:
for (auto x : adj[curr])
{
    // if the current vertex is not yet
    // visited, then push it to the queue.
    if (visited[x] == false)
    {
        q.push(x);
        visited[x] = true;
    }

    // check if there is a better path.
    if (dist[x] > dist[curr] + 1)
    {
        dist[x] = dist[curr] + 1;
        paths[x] = paths[curr];
    }

    // additional shortest paths found
    else if (dist[x] == dist[curr] + 1)
        paths[x] += paths[curr];
}
}

// function to find number of different
// shortest paths from given vertex s.
// n is number of vertices.
void findShortestPaths(vector<int> adj[],
                      int s, int n)
{
    int dist[n], paths[n];
    for (int i = 0; i < n; i++)
        dist[i] = INT_MAX;
    for (int i = 0; i < n; i++)
        paths[i] = 0;
    BFS(adj, s, dist, paths, n);
    cout << "Numbers of shortest Paths are: ";
    for (int i = 0; i < n; i++)
        cout << paths[i] << " ";
}

// A utility function to add an edge in a
// directed graph.
void addEdge(vector<int> adj[], int u, int v)
{
    adj[u].push_back(v);
}

```

```
}  
  
// Driver code  
int main()  
{  
    int n = 7; // Number of vertices  
    vector <int> adj[n];  
    addEdge(adj, 0, 1);  
    addEdge(adj, 0, 2);  
    addEdge(adj, 1, 2);  
    addEdge(adj, 1, 3);  
    addEdge(adj, 2, 3);  
    addEdge(adj, 3, 4);  
    addEdge(adj, 3, 5);  
    addEdge(adj, 4, 6);  
    addEdge(adj, 5, 6);  
    findShortestPaths(adj, 0, 7);  
    return 0;  
}
```

Output:

Numbers of shortest Paths are: 1 1 1 2 2 2 4

Time Complexity : $O(V + E)$

Improved By : [Shlomi Elhaiani](#)

Source

<https://www.geeksforgeeks.org/number-shortest-paths-unweighted-directed-graph/>

Chapter 226

Number of single cycle components in an undirected graph

Number of single cycle components in an undirected graph - GeeksforGeeks

Given a set of ‘n’ vertices and ‘m’ edges of an undirected simple graph (no parallel edges and no self-loop), find the number of single-cycle-components present in the graph. A single-cyclic-component is a graph of n nodes containing a single cycle through all nodes of the component.

Example:

Let us consider the following graph with 15 vertices.

```
Input: V = 15, E = 14
      1 10 // edge 1
      1 5  // edge 2
      5 10 // edge 3
      2 9  // ..
      9 15 // ..
      2 15 // ..
      2 12 // ..
      12 15 // ..
      13 8  // ..
      6 14  // ..
      14 3  // ..
      3 7  // ..
      7 11 // edge 13
      11 6 // edge 14
```

Output :2

In the above-mentioned example, the two single-cyclic-components are composed of vertices (1, 10, 5) and (6, 11, 7, 3, 14) respectively.

Now we can easily see that a single-cycle-component is a connected component where every vertex has the degree as two.

Therefore, in order to solve this problem we first identify all the [connected components of the disconnected graph](#). For this, we use depth-first search algorithm. For the DFS algorithm to work, it is required to maintain an array ‘found’ to keep an account of all the vertices that have been discovered by the recursive function DFS. Once all the elements of a particular connected component are discovered (like vertices(9, 2, 15, 12) form a connected graph component), we check if all the vertices in the component are having the degree equal to two. If yes, we increase the counter variable ‘count’ which denotes the number of single-cycle-components found in the given graph. To keep an account of the component we are presently dealing with, we may use a vector array ‘curr_graph’ as well.

```
// CPP program to find single cycle components
// in a graph.
#include <bits/stdc++.h>
using namespace std;

const int N = 100000;

// degree of all the vertices
int degree[N];

// to keep track of all the vertices covered
// till now
bool found[N];

// all the vertices in a particular
// connected component of the graph
vector<int> curr_graph;

// adjacency list
vector<int> adj_list[N];

// depth-first traversal to identify all the
// nodes in a particular connected graph
// component
void DFS(int v)
{
    found[v] = true;
    curr_graph.push_back(v);
    for (int it : adj_list[v])
        if (!found[it])
```

```
        DFS(it);
    }

// function to add an edge in the graph
void addEdge(vector<int> adj_list[N], int src,
             int dest)
{
    // for index decrement both src and dest.
    src--, dest--;
    adj_list[src].push_back(dest);
    adj_list[dest].push_back(src);
    degree[src]++;
    degree[dest]++;
}

int countSingleCycles(int n, int m)
{
    // count of cycle graph components
    int count = 0;
    for (int i = 0; i < n; ++i) {
        if (!found[i]) {
            curr_graph.clear();
            DFS(i);

            // traversing the nodes of the
            // current graph component
            int flag = 1;
            for (int v : curr_graph) {
                if (degree[v] == 2)
                    continue;
                else {
                    flag = 0;
                    break;
                }
            }
            if (flag == 1) {
                count++;
            }
        }
    }
    return(count);
}

int main()
{
    // n->number of vertices
    // m->number of edges
    int n = 15, m = 14;
```

```
addEdge(adj_list, 1, 10);
addEdge(adj_list, 1, 5);
addEdge(adj_list, 5, 10);
addEdge(adj_list, 2, 9);
addEdge(adj_list, 9, 15);
addEdge(adj_list, 2, 15);
addEdge(adj_list, 2, 12);
addEdge(adj_list, 12, 15);
addEdge(adj_list, 13, 8);
addEdge(adj_list, 6, 14);
addEdge(adj_list, 14, 3);
addEdge(adj_list, 3, 7);
addEdge(adj_list, 7, 11);
addEdge(adj_list, 11, 6);

cout << countSingleCycles(n, m);

return 0;
}
```

Output:

2

Hence, total number of cycle graph component is found.

Improved By : [PiyushKumar](#)

Source

<https://www.geeksforgeeks.org/number-of-simple-cyclic-components-in-an-undirected-graph/>

Chapter 227

Number of sink nodes in a graph

Number of sink nodes in a graph - GeeksforGeeks

Given a Directed Acyclic Graph of **n** nodes (numbered from 1 to n) and **m** edges. The task is to find the number of sink nodes. A sink node is a node such that no edge emerges out of it.

Examples:

```
Input : n = 4, m = 2
        Edges[] = {{2, 3}, {4, 3}}
Output : 2
```

Only node 1 and node 3 are sink nodes.

```
Input : n = 4, m = 2
        Edges[] = {{3, 2}, {3, 4}}
Output : 3
```

The idea is to iterate through all the edges. And for each edge, mark the source node from which the edge emerged out. Now, for each node check if it is marked or not. And count the unmarked nodes.

Algorithm:

1. Make any array A[] of size equal to the number of nodes and initialize to 1.
2. Traverse all the edges one by one, say, $u \rightarrow v$.

- (i) Mark A[u] as 1.
3. Now traverse whole array A[] and count number of unmarked nodes.

Below is C++ implementation of this approach:

```
// C++ program to count number if sink nodes
#include<bits/stdc++.h>
using namespace std;

// Return the number of Sink NOdes.
int countSink(int n, int m, int edgeFrom[],
              int edgeTo[])
{
    // Array for marking the non-sink node.
    int mark[n];
    memset(mark, 0, sizeof mark);

    // Marking the non-sink node.
    for (int i = 0; i < m; i++)
        mark[edgeFrom[i]] = 1;

    // Counting the sink nodes.
    int count = 0;
    for (int i = 1; i <= n ; i++)
        if (!mark[i])
            count++;

    return count;
}

// Driven Program
int main()
{
    int n = 4, m = 2;
    int edgeFrom[] = { 2, 4 };
    int edgeTo[] = { 3, 3 };

    cout << countSink(n, m, edgeFrom, edgeTo) << endl;

    return 0;
}
```

Output:

Time Complexity: $O(m + n)$ where n is number of nodes and m is number of edges.

Related Article:

[The Celebrity Problem](#)

Source

<https://www.geeksforgeeks.org/number-sink-nodes-graph/>

Chapter 228

Number of Transpositions in a Permutation

Number of Transpositions in a Permutation - GeeksforGeeks

Permutation A permutation is an arrangement of elements. A permutation of n elements can be represented by an arrangement of the numbers $1, 2, \dots, n$ in some order. e.g. $5, 1, 4, 2, 3$.

Cycle notation A permutation can be represented as a composition of permutation cycles. A permutation cycle is a set of elements in a permutation which trade places with one another.

For e.g.

$$P = \{ 5, 1, 4, 2, 3 \}:$$

Here, 5 goes to 1, 1 goes to 2 and so on (according to their indices position):

$$5 \rightarrow 1$$

$$1 \rightarrow 2$$

$$2 \rightarrow 4$$

$$4 \rightarrow 3$$

$$3 \rightarrow 5$$

Thus it can be represented as a single cycle: $(5, 1, 2, 4, 3)$.

Now consider the permutation: $\{ 5, 1, 4, 3, 2 \}$. Here

$$5 \rightarrow 1$$

$$1 \rightarrow 2$$

$2 \rightarrow 5$ this closes 1 cycle.

The other cycle is

$$4 \rightarrow 3$$

$$3 \rightarrow 4$$

In cycle notation it will be represented as $(5, 1, 2) (4, 3)$.

Transpositions

Now all cycles can be decomposed into a composition of 2 cycles (transpositions). The

number of transpositions in a permutation is important as it gives the minimum number of 2 element swaps required to get this particular arrangement from the identity arrangement: 1, 2, 3, ... n. The parity of the number of such 2 cycles represents whether the permutation is even or odd.

For e.g.

The cycle (5, 1, 2, 4, 3) can be written as (5, 3)(5, 4)(5, 2)(5, 1). 4 transpositions (even).

Similarly,

(5, 1, 2) -> (5, 2)(5, 1)

(5, 1, 2)(4, 3) -> (5, 2)(5, 1)(4, 3). 3 transpositions (odd).

It is clear from the examples that the **number of transpositions from a cycle = length of the cycle – 1**.

Problem

Given a permutation of n numbers $P_1, P_2, P_3, \dots, P_n$. Calculate the number of transpositions in it.

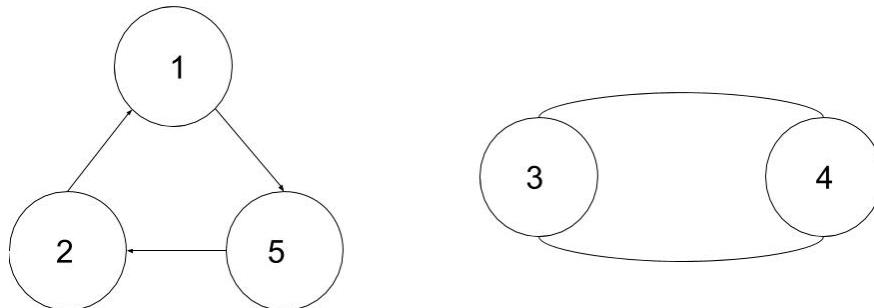
Example:

Input: 5 1 4 3 2

Output: 3

Approach: The permutation can be easily represented as a directed graph where the number of connected components gives the number of cycles. And (the size of each component – 1) gives the number of transpositions from that cycle.

Example Permutation : { 5, 1, 4, 3, 2 } -> (5, 1, 2)(4, 3)



Transpositions = 2

Total = 3

Transpositions = 1

Below is the implementation of above approach.

C++

```
// CPP Program to find the number of
// transpositions in a permutation
#include <bits/stdc++.h>
using namespace std;

#define N 1000001

int visited[N];

// This array stores which element goes to which position
int goesTo[N];

// For eg. in { 5, 1, 4, 3, 2 }
// goesTo[1] = 2
// goesTo[2] = 5
// goesTo[3] = 4
// goesTo[4] = 3
// goesTo[5] = 1

// This function returns the size of a component cycle
int dfs(int i)
{
    // If it is already visited
    if (visited[i] == 1)
        return 0;

    visited[i] = 1;
    int x = dfs(goesTo[i]);
    return (x + 1);
}

// This function returns the number
// of transpositions in the permutation
int noOfTranspositions(int P[], int n)
{
    // Initializing visited[] array
    for (int i = 1; i <= n; i++)
        visited[i] = 0;

    // building the goesTo[] array
    for (int i = 0; i < n; i++)
        goesTo[P[i]] = i + 1;

    int transpositions = 0;

    for (int i = 1; i <= n; i++) {
        if (visited[i] == 0) {
            int ans = dfs(i);
```

```
        transpositions += ans - 1;
    }
}
return transpositions;
}

// Driver Code
int main()
{
    int permutation[] = { 5, 1, 4, 3, 2 };
    int n = sizeof(permutation) / sizeof(permutation[0]);

    cout << noOfTranspositions(permutation, n);
    return 0;
}
```

Java

```
// Java Program to find the number of
// transpositions in a permutation
import java.io.*;

class GFG {

    static int N = 1000001;

    static int visited[] = new int[N];

    // This array stores which element
    // goes to which position
    static int goesTo[] = new int[N];

    // For eg. in { 5, 1, 4, 3, 2 }
    // goesTo[1] = 2
    // goesTo[2] = 5
    // goesTo[3] = 4
    // goesTo[4] = 3
    // goesTo[5] = 1

    // This function returns the size
    // of a component cycle
    static int dfs(int i)
    {

        // If it is already visited
        if (visited[i] == 1)
            return 0;
```

```
visited[i] = 1;
int x = dfs(goesTo[i]);
return (x + 1);
}

// This function returns the number
// of transpositions in the
// permutation
static int noOfTranspositions(int P[],
                               int n)
{
    // Initializing visited[] array
    for (int i = 1; i <= n; i++)
        visited[i] = 0;

    // building the goesTo[] array
    for (int i = 0; i < n; i++)
        goesTo[P[i]] = i + 1;

    int transpositions = 0;

    for (int i = 1; i <= n; i++) {
        if (visited[i] == 0) {
            int ans = dfs(i);
            transpositions += ans - 1;
        }
    }
    return transpositions;
}

// Driver Code
public static void main (String[] args)
{
    int permutation[] = { 5, 1, 4, 3, 2 };
    int n = permutation.length ;

    System.out.println(
        noOfTranspositions(permutation, n));
}
}

// This code is contributed by anuj_67.
```

C#

```
// C# Program to find the number of
// transpositions in a permutation
using System;
```

```
class GFG {

    static int N = 1000001;

    static int []visited = new int[N];

    // This array stores which element
    // goes to which position
    static int []goesTo= new int[N];

    // For eg. in { 5, 1, 4, 3, 2 }
    // goesTo[1] = 2
    // goesTo[2] = 5
    // goesTo[3] = 4
    // goesTo[4] = 3
    // goesTo[5] = 1

    // This function returns the size
    // of a component cycle
    static int dfs(int i)
    {

        // If it is already visited
        if (visited[i] == 1)
            return 0;

        visited[i] = 1;
        int x = dfs(goesTo[i]);
        return (x + 1);
    }

    // This functio returns the number
    // of transpositions in the
    // permutation
    static int noOfTranspositions(int []P,
                                  int n)
    {
        // Initializing visited[] array
        for (int i = 1; i <= n; i++)
            visited[i] = 0;

        // building the goesTo[] array
        for (int i = 0; i < n; i++)
            goesTo[P[i]] = i + 1;

        int transpositions = 0;
```

```
for (int i = 1; i <= n; i++) {
    if (visited[i] == 0) {
        int ans = dfs(i);
        transpositions += ans - 1;
    }
}
return transpositions;
}

// Driver Code
public static void Main ()
{
    int []permutation = { 5, 1, 4, 3, 2 };
    int n = permutation.Length ;

    Console.WriteLine(
        noOfTranspositions(permutation, n));
}
}

// This code is contributed by anuj_67.
```

Output:

3

Time Complexity : $O(n)$

Auxiliary space : $O(n)$

Improved By : [vt_m](#)

Source

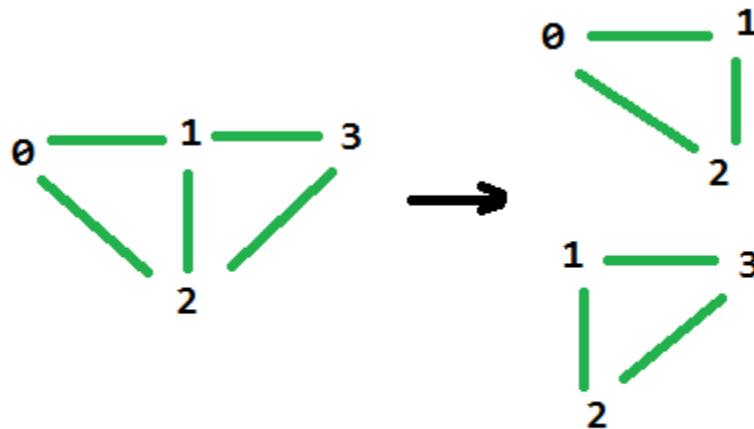
<https://www.geeksforgeeks.org/number-of-transpositions-in-a-permutation/>

Chapter 229

Number of Triangles in an Undirected Graph

Number of Triangles in an Undirected Graph - GeeksforGeeks

Given an Undirected simple graph, We need to find how many triangles it can have. For example below graph have 2 triangles in it.



Graph with 2 triangles

Let $A[]$ be adjacency matrix representation of graph. If we calculate A^3 , then the number of triangle in Undirected Graph is equal to $\text{trace}(A^3) / 6$. Where $\text{trace}(A)$ is the sum of the elements on the main diagonal of matrix A.

Trace of a graph represented as adjacency matrix $A[V][V]$ is,
 $\text{trace}(A[V][V]) = A[0][0] + A[1][1] + \dots + A[V-1][V-1]$

Count of triangles = $\text{trace}(A^3) / 6$

Below is implementation of above formula.

C++

```
// A C++ program for finding
// number of triangles in an
// Undirected Graph. The program
// is for adjacency matrix
// representation of the graph
#include <bits/stdc++.h>
using namespace std;

// Number of vertices in the graph
#define V 4

// Utility function for matrix
// multiplication
void multiply(int A[][][V], int B[][][V], int C[][][V])
{
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            C[i][j] = 0;
            for (int k = 0; k < V; k++)
                C[i][j] += A[i][k]*B[k][j];
        }
    }
}

// Utility function to calculate
// trace of a matrix (sum of
// diagonal elements)
int getTrace(int graph[][][V])
{
    int trace = 0;
    for (int i = 0; i < V; i++)
        trace += graph[i][i];
    return trace;
}

// Utility function for calculating
// number of triangles in graph
int triangleInGraph(int graph[][][V])
{
    // To Store graph^2
    int aux2[V][V];

    // To Store graph^3
```

```
int aux3[V][V];

// Initialising aux
// matrices with 0
for (int i = 0; i < V; ++i)
    for (int j = 0; j < V; ++j)
        aux2[i][j] = aux3[i][j] = 0;

// aux2 is graph^2 now printMatrix(aux2);
multiply(graph, graph, aux2);

// after this multiplication aux3 is
// graph^3 printMatrix(aux3);
multiply(graph, aux2, aux3);

int trace = getTrace(aux3);
return trace / 6;
}

// driver code
int main()
{
    int graph[V][V] = {{0, 1, 1, 0},
                        {1, 0, 1, 1},
                        {1, 1, 0, 1},
                        {0, 1, 1, 0}};
    printf("Total number of Triangle in Graph : %d\n",
           triangleInGraph(graph));
    return 0;
}
```

Java

```
// Java program to find number
// of triangles in an Undirected
// Graph. The program is for
// adjacency matrix representation
// of the graph
import java.io.*;

class Directed
{
    // Number of vertices in
    // the graph
    int V = 4;
```

```
// Utility function for
// matrix multiplication
void multiply(int A[][] , int B[][] ,
              int C[][])
{
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            C[i][j] = 0;
            for (int k = 0; k < V;
                 k++)
            {
                C[i][j] += A[i][k]*
                           B[k][j];
            }
        }
    }
}

// Utility function to calculate
// trace of a matrix (sum of
// diagonal elements)
int getTrace(int graph[][])
{
    int trace = 0;

    for (int i = 0; i < V; i++)
    {
        trace += graph[i][i];
    }
    return trace;
}

// Utility function for
// calculating number of
// triangles in graph
int triangleInGraph(int graph[][])
{
    // To Store graph^2
    int[][] aux2 = new int[V][V];

    // To Store graph^3
    int[][] aux3 = new int[V][V];

    // Initialising aux matrices
    // with 0
```

```

        for (int i = 0; i < V; ++i)
        {
            for (int j = 0; j < V; ++j)
            {
                aux2[i][j] = aux3[i][j] = 0;
            }
        }

        // aux2 is graph^2 now
        // printMatrix(aux2)
        multiply(graph, graph, aux2);

        // after this multiplication aux3
        // is graph^3 printMatrix(aux3)
        multiply(graph, aux2, aux3);

        int trace = getTrace(aux3);

        return trace / 6;
    }

    // Driver code
    public static void main(String args[])
    {
        Directed obj = new Directed();

        int graph[][] = { {0, 1, 1, 0},
                          {1, 0, 1, 1},
                          {1, 1, 0, 1},
                          {0, 1, 1, 0}
                        };

        System.out.println("Total number of Triangle in Graph : "+
                           obj.triangleInGraph(graph));
    }
}

// This code is contributed by Anshika Goyal.

```

C#

```

// C# program to find number
// of triangles in an Undirected
// Graph. The program is for
// adjacency matrix representation
// of the graph
using System;

```

```
class GFG
{
// Number of vertices
// in the graph
int V = 4;

// Utility function for
// matrix multiplication
void multiply(int [,]A, int [,]B,
              int [,]C)
{
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            C[i, j] = 0;
            for (int k = 0; k < V;
                 k++)
            {
                C[i, j] += A[i, k]*
                            B[k, j];
            }
        }
    }
}

// Utility function to
// calculate trace of
// a matrix (sum of
// diagonal elements)
int getTrace(int [,]graph)
{
    int trace = 0;

    for (int i = 0; i < V; i++)
    {
        trace += graph[i, i];
    }
    return trace;
}

// Utility function for
// calculating number of
// triangles in graph
int triangleInGraph(int [,]graph)
{
    // To Store graph^2
    int[,] aux2 = new int[V, V];
```

```
// To Store graph^3
int[,] aux3 = new int[V, V];

// Initialising aux matrices
// with 0
for (int i = 0; i < V; ++i)
{
    for (int j = 0; j < V; ++j)
    {
        aux2[i, j] = aux3[i, j] = 0;
    }
}

// aux2 is graph^2 now
// printMatrix(aux2)
multiply(graph, graph, aux2);

// after this multiplication aux3
// is graph^3 printMatrix(aux3)
multiply(graph, aux2, aux3);

int trace = getTrace(aux3);

return trace / 6;
}

// Driver code
public static void Main()
{
    GFG obj = new GFG();

    int [,]graph = {{0, 1, 1, 0},
                    {1, 0, 1, 1},
                    {1, 1, 0, 1},
                    {0, 1, 1, 0}};

    Console.WriteLine("Total number of " +
                      "Triangle in Graph : "+
                      obj.triangleInGraph(graph));
}
}

// This code is contributed by anuj_67.
```

Output:

```
Total number of Triangle in Graph : 2
```

How does this work?

If we compute A^n for an adjacency matrix representation of graph, then a value $A^n[i][j]$ represents number of distinct walks between vertex i to j in graph. In A^3 , we get all distinct paths of length 3 between every pair of vertices.

A triangle is a cyclic path of length three, i.e. begins and ends at same vertex. So $A^3[i][i]$ represents a triangle beginning and ending with vertex i . Since a triangle has three vertices and it is counted for every vertex, we need to divide result by 3. Furthermore, since the graph is undirected, every triangle twice as $i-p-q-j$ and $i-q-p-j$, so we divide by 2 also. Therefore, number of triangles is $\text{trace}(A^3) / 6$.

Time Complexity:

The time complexity of above algorithm is $O(V^3)$ where V is number of vertices in the graph, we can improve the performance to $O(V^{2.8074})$ using [Strassen's matrix multiplication](#) algorithm.

References:

<http://www.d.umn.edu/math/Technical%20Reports/Technical%20Reports%202007-/TR%202012/yang.pdf>

[Number of Triangles in Directed and Undirected Graphs](#)

This article is contributed by Utkarsh Trivedi. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Improved By : [vt_m](#)

Source

<https://www.geeksforgeeks.org/number-of-triangles-in-a-undirected-graph/>

Chapter 230

Number of Triangles in Directed and Undirected Graphs

Number of Triangles in Directed and Undirected Graphs - GeeksforGeeks

Given a Graph, count number of triangles in it. The graph is can be directed or undirected.

Example:

```
Input: digraph[V] [V] = { {0, 0, 1, 0},
                          {1, 0, 0, 1},
                          {0, 1, 0, 0},
                          {0, 0, 1, 0}
                        };
Output: 2
Give adjacency matrix represents following
directed graph.
```

We have discussed a [method based on graph trace](#) that works for undirected graphs. In this post a new method is discussed with that is simpler and works for both directed and undirected graphs.

The idea is to use three nested loops to consider every triplet (i, j, k) and check for the above condition (there is an edge from i to j, j to k and k to i)

However in an **undirected graph**, the triplet (i, j, k) can be permuted to give six combination (See [previous post](#) for details). Hence we divide the total count by 6 to get the actual number of triangles.

In case of **directed graph**, the number of permutation would be 3 (as order of nodes becomes relevant). Hence in this case the total number of triangles will be obtained by dividing total count by 3. For example consider the directed graph given below

Following is the implementation.

C/C++

```
// C++ program to count triangles
// in a graph. The program is for
// adjacency matrix representation
// of the graph.
#include<bits/stdc++.h>

// Number of vertices in the graph
#define V 4

using namespace std;

// function to calculate the
// number of triangles in a
// simple directed/undirected
// graph. isDirected is true if
// the graph is directed, its
// false otherwise
int countTriangle(int graph[V][V],
                  bool isDirected)
{
    // Initialize result
    int count_Triangle = 0;

    // Consider every possible
    // triplet of edges in graph
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            for (int k = 0; k < V; k++)
            {
                // check the triplet if
                // it satisfies the condition
                if (graph[i][j] && graph[j][k]
                    && graph[k][i])
                    count_Triangle++;
            }
        }
    }

    // if graph is directed ,
    // division is done by 3,
    // else division by 6 is done
    isDirected? count_Triangle /= 3 :
               count_Triangle /= 6;

    return count_Triangle;
}
```

```
//driver function to check the program
int main()
{
    // Create adjacency matrix
    // of an undirected graph
    int graph[][] [V] = { {0, 1, 1, 0},
                         {1, 0, 1, 1},
                         {1, 1, 0, 1},
                         {0, 1, 1, 0}
                       };

    // Create adjacency matrix
    // of a directed graph
    int digraph[][] [V] = { {0, 0, 1, 0},
                           {1, 0, 0, 1},
                           {0, 1, 0, 0},
                           {0, 0, 1, 0}
                         };

    cout << "The Number of triangles in undirected graph : "
         << countTriangle(graph, false);
    cout << "\n\nThe Number of triangles in directed graph : "
         << countTriangle(digraph, true);

    return 0;
}
```

Java

```
// Java program to count triangles
// in a graph. The program is
// for adjacency matrix
// representation of the graph.
import java.io.*;

class GFG {

    // Number of vertices in the graph
    int V = 4;

    // function to calculate the number
    // of triangles in a simple
    // directed/undirected graph. isDirected
    // is true if the graph is directed,
    // its false otherwise.
    int countTriangle(int graph[] [],
                      boolean isDirected)
```

```
{  
    // Initialize result  
    int count_Triangle = 0;  
  
    // Consider every possible  
    // triplet of edges in graph  
    for (int i = 0; i < V; i++)  
    {  
        for (int j = 0; j < V; j++)  
        {  
            for (int k=0; k<V; k++)  
            {  
                // check the triplet if it  
                // satisfies the condition  
                if (graph[i][j] == 1 &&  
                    graph[j][k] == 1 &&  
                    graph[k][i] == 1)  
                    count_Triangle++;  
            }  
        }  
    }  
  
    // if graph is directed , division  
    // is done by 3 else division  
    // by 6 is done  
    if(isDirected == true)  
    {  
        count_Triangle /= 3;  
    }  
    else  
    {  
        count_Triangle /= 6;  
    }  
    return count_Triangle;  
}  
  
// driver code  
public static void main(String args[])  
{  
  
    // Create adjacency matrix  
    // of an undirected graph  
    int graph[][] = {{0, 1, 1, 0},  
                    {1, 0, 1, 1},  
                    {1, 1, 0, 1},  
                    {0, 1, 1, 0}};  
}
```

```

// Create adjacency matrix
// of a directed graph
int digraph[][] = { {0, 0, 1, 0},
                    {1, 0, 0, 1},
                    {0, 1, 0, 0},
                    {0, 0, 1, 0}
                };

System.out.println("The Number of triangles "+
                    "in undirected graph : " +
                    countTriangle(graph, false));

System.out.println("\n\nThe Number of triangles"+
                    " in directed graph : " +
                    countTriangle(digraph, true));

}

}

// This code is contributed by Anshika Goyal.

```

Python

```

# Python program to count triangles in a graph. The program is
# for adjacency matrix representation of the graph.

# function to calculate the number of triangles in a simple
# directed/undirected graph.
# isDirected is true if the graph is directed, its false otherwise
def countTriangle(g, isDirected):
    nodes = len(g)
    count_Triangle = 0 #Initialize result
    # Consider every possible triplet of edges in graph
    for i in range(nodes):
        for j in range(nodes):
            for k in range(nodes):
                # check the triplet if it satisfies the condition
                if( i!=j and i !=k and j !=k and
                    g[i][j] and g[j][k] and g[k][i]):
                    count_Triangle += 1
    # if graph is directed , division is done by 3
    # else division by 6 is done
    return count_Triangle/3 if isDirected else count_Triangle/6

# Create adjacency matrix of an undirected graph
graph = [[0, 1, 1, 0],
          [1, 0, 1, 1],

```

```
[1, 1, 0, 1],  
[0, 1, 1, 0 ]]  
# Create adjacency matrix of a directed graph  
digraph = [[0, 0, 1, 0],  
           [1, 0, 0, 1],  
           [0, 1, 0, 0],  
           [0, 0, 1, 0 ]]  
  
print ("The Number of triangles in undirected graph : %d" %countTriangle(graph, False))  
  
print ("The Number of triangles in directed graph : %d" %countTriangle(digraph, True))  
  
# This code is contributed by Neelam Yadav
```

C#

```
// C# program to count triangles in a graph.  
// The program is for adjacency matrix  
// representation of the graph.  
using System;  
  
class GFG {  
  
    // Number of vertices in the graph  
    const int V = 4;  
  
    // function to calculate the  
    // number of triangles in a  
    // simple directed/undirected  
    // graph. isDirected is true if  
    // the graph is directed, its  
    // false otherwise  
    static int countTriangle(int[,] graph,  
                           bool isDirected)  
    {  
        // Initialize result  
        int count_Triangle = 0;  
  
        // Consider every possible  
        // triplet of edges in graph  
        for (int i = 0; i < V; i++)  
        {  
            for (int j = 0; j < V; j++)  
            {  
                for (int k = 0; k < V; k++)  
                {  
                    // check the triplet if  
                    // it satisfies the condition
```

```
        if (graph[i,j] != 0 &&
            graph[j,k] != 0 &&
            graph[k,i] != 0 )
            count_Triangle++;
    }
}

// if graph is directed ,
// division is done by 3,
// else division by 6 is done
if(isDirected != false)
    count_Triangle =
        count_Triangle / 3 ;
else
    count_Triangle =
        count_Triangle / 6;

return count_Triangle;
}

// Driver function to check the program
static void Main()
{
    // Create adjacency matrix
    // of an undirected graph
    int[,] graph = new int[4,4] {
        {0, 1, 1, 0},
        {1, 0, 1, 1},
        {1, 1, 0, 1},
        {0, 1, 1, 0}
    };

    // Create adjacency matrix
    // of a directed graph
    int[,] digraph = new int [4,4] {
        {0, 0, 1, 0},
        {1, 0, 0, 1},
        {0, 1, 0, 0},
        {0, 0, 1, 0}
    };

    Console.WriteLine("The Number of triangles"
        + " in undirected graph : "
        + countTriangle(graph, false));
}
```

```
Console.WriteLine("\n\nThe Number of "
+ "triangles in directed graph : "
+ countTriangle(digraph, true));
}
}

// This code is contributed by anuj_67
```

PHP

```
<?php
// PHP program to count triangles
// in a graph. The program is for
// adjacency matrix representation
// of the graph.

// Number of vertices in the graph
$V = 4;

// function to calculate the
// number of triangles in a
// simple directed/undirected
// graph. isDirected is true if
// the graph is directed, its
// false otherwise
function countTriangle($graph,
                      $isDirected)
{
    global $V;

    // Initialize result
    $count_Triangle = 0;

    // Consider every possible
    // triplet of edges in graph
    for($i = 0; $i < $V; $i++)
    {
        for($j = 0; $j < $V; $j++)
        {
            for($k = 0; $k < $V; $k++)
            {

                // check the triplet if
                // it satisfies the condition
                if ($graph[$i][$j] and $graph[$j][$k]
                    and $graph[$k][$i])
                    $count_Triangle++;
            }
        }
    }
}
```

```
        }
    }
}

// if graph is directed ,
// division is done by 3,
// else division by 6 is done
$isDirected? $count_Triangle /= 3 :
    $count_Triangle /= 6;

return $count_Triangle;
}

// Driver Code
// Create adjacency matrix
// of an undirected graph
$graph = array(array(0, 1, 1, 0),
               array(1, 0, 1, 1),
               array(1, 1, 0, 1),
               array(0, 1, 1, 0));

// Create adjacency matrix
// of a directed graph
$digraph = array(array(0, 0, 1, 0),
                 array(1, 0, 0, 1),
                 array(0, 1, 0, 0),
                 array(0, 0, 1, 0));

echo "The Number of triangles in undirected graph : "
     , countTriangle($graph, false);
echo "\nThe Number of triangles in directed graph : "
     , countTriangle($digraph, true);

// This code is contributed by anuj_67
?>
```

Output:

```
The Number of triangles in undirected graph : 2
The Number of triangles in directed graph : 2
```

Comparison of this approach with [previous approach](#):
Advantages:

- No need to calculate Trace.
- Matrix- multiplication is not required.

- Auxiliary matrices are not required hence optimized in space.
- Works for directed graphs.

Disadvantages:

- The time complexity is $O(n^3)$ and can't be reduced any further.

Improved By : [vt_m](#)

Source

<https://www.geeksforgeeks.org/number-of-triangles-in-directed-and-undirected-graphs/>

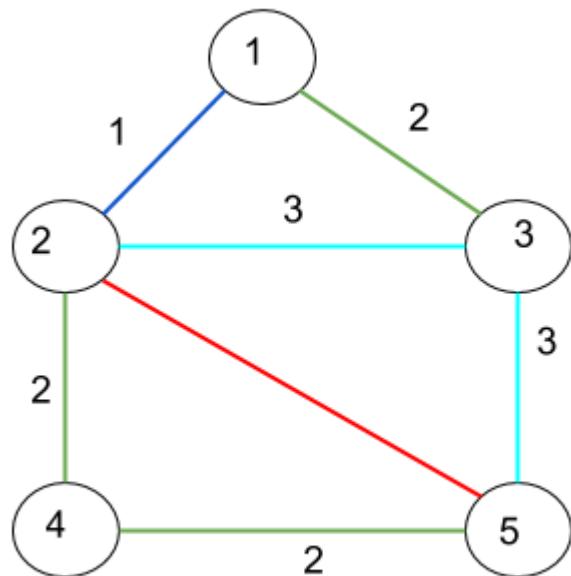
Chapter 231

Number of Unicolored Paths between two nodes

Number of Unicolored Paths between two nodes - GeeksforGeeks

Given an undirected colored graph([edges are colored](#)), with a source vertex ‘s’ and a destination vertex ‘d’, print number of paths which from given ‘s’ to ‘d’ such that the path is UniColored(edges in path having same color).

The edges are colored, here Colors are represented with numbers. At maximum, number of different colors will be number of edges.



```
Input : Graph
    u, v, color
    1, 2, 1
    1, 3, 2
    2, 3, 3
    2, 4, 2
    2, 5, 4
    3, 5, 3
    4, 5, 2
source = 2    destination = 5

Output : 3
Explanation : There are three paths from 2 to 5
2 -> 5 with color red
2 -> 3 -> 5 with color sky blue
2 -> 4 -> 5 with color green
```

Algorithm :

1. Do dfs traversal on the neighbour nodes of source node.
2. The color between source node and neighbour nodes is known, if the DFS traversal also have same color, proceed, else stop going on that path.
3. After reaching destination node, increment count by 1.

NOTE : Number of Colors will always be less than number of edges.

```
// C++ code to find unicolored paths
#include <bits/stdc++.h>
using namespace std;

const int MAX_V = 100;

int color[MAX_V];
bool vis[MAX_V];

// Graph class represents a undirected graph
// using adjacency list representation
class Graph
{
    // vertices, edges, adjancy list
    int V;
    int E;
    vector<pair<int, int>> adj[MAX_V];

    // function used by UniColorPaths
    // DFS traversal o from x to y
    void dfs(int x, int y, int z);
```

```
// Constructor
public:
    Graph(int V, int E);

    // function to add an edge to graph
    void addEdge(int v, int w, int z);

    // finds paths between a and b having
    // same color edges
    int UniColorPaths(int a, int b);
};

Graph::Graph(int V, int E)
{
    this -> V = V;
    this -> E = E;
}

void Graph::addEdge(int a, int b, int c)
{
    adj[a].push_back({b, c}); // Add b to a's list.
    adj[b].push_back({a, c}); // Add c to b's list.
}

void Graph::dfs(int x, int y, int col)
{
    if (vis[x])
        return;
    vis[x] = 1;

    // mark this as a possible color to reach s to d
    if (x == y)
    {
        color[col] = 1;
        return;
    }

    // if the next edge is also of same color
    for (int i = 0; i < int(adj[x].size()); i++)
        if (adj[x][i].second == col)
            dfs(adj[x][i].first, y, col);
}

// function that finds paths between a and b
// such that all edges are same colored
// It uses recursive dfs()
int Graph::UniColorPaths(int a, int b)
{
```

```
// dfs on nodes directly connected to source
for (int i = 0; i < int(adj[a].size()); i++)
{
    dfs(a, b, adj[a][i].second);

    // to visit again visited nodes
    memset(vis, 0, sizeof(vis));
}

int cur = 0;
for (int i = 0; i <= E; i++)
    cur += color[i];

return (cur);
}

// driver code
int main()
{
    // Create a graph given in the above diagram
    Graph g(5, 7);
    g.addEdge(1, 2, 1);
    g.addEdge(1, 3, 2);
    g.addEdge(2, 3, 3);
    g.addEdge(2, 4, 2);
    g.addEdge(2, 5, 4);
    g.addEdge(3, 5, 3);
    g.addEdge(4, 5, 2);

    int s = 2; // source
    int d = 5; // destination

    cout << "Number of unicolored paths : ";
    cout << g.UniColorPaths(s, d) << endl;
    return 0;
}
```

Output:

Number of unicolored paths : 3

Time Complexity : $O(E * (E + V))$

Source

<https://www.geeksforgeeks.org/number-unicolored-paths-two-nodes/>

Chapter 232

Path in a Rectangle with Circles

Path in a Rectangle with Circles - GeeksforGeeks

There is a $m \times n$ rectangular matrix whose top-left(start) location is $(1, 1)$ and bottom-right(end) location is (m, n) . There are k circles each with radius r . Find if there is any path from start to end without touching any circle.

The input contains values of m , n , k , r and two array of integers X and Y , each of length k . $(X[i], Y[i])$ is the centre of i th circle.

Source : [Directi Interview](#)

```
Input1 : m = 5, n = 5, k = 2, r = 1,
         X = {1, 3}, Y = {3, 3}
Output1 : Possible
```

Here is a path from start to end point.

```
Input2 : m = 5, n = 5, k = 2, r = 1,
         X = {1, 1}, Y = {2, 3}.
Output2 : Not Possible
```

Approach : Check if the centre of a cell (i, j) of the rectangle comes within any of the circles then do not traverse through that cell and mark that as ‘blocked’. Mark rest of the cells initially as ‘unvisited’. Then use [BFS](#) to find out shortest path of each cell from starting position. If the end cell is visited then we will return “Possible” otherwise “Not Possible”.

Algorithm :

1. Take an array of size $m \times n$. Initialize all the cells to 0.
2. For each cell of the rectangle check whether it comes within any circle or not (by calculating the distance of that cell from each circle). If it comes within any circle then change the value of that cell to -1(‘blocked’).

3. Now, apply BFS from the starting cell and if a cell can be reached then change the value of that cell to 1.
4. If the value of the ending cell is 1, then return ‘Possible’, otherwise return ‘Not Possible’.

C++

```

// C++ program to find out path in
// a rectangle containing circles.
#include <iostream>
#include <math.h>
#include <vector>

using namespace std;

// Function to find out if there is
// any possible path or not.
bool isPossible(int m, int n, int k, int r,
                vector<int> X, vector<int> Y)
{
    // Take an array of m*n size and
    // initialize each element to 0.
    int rect[m][n] = {0};

    // Now using Pythagorean theorem find if a
    // cell touches or within any circle or not.
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            for (int p = 0; p < k; p++) {
                if (sqrt((pow((X[p] - 1 - i), 2) +
                           pow((Y[p] - 1 - j), 2))) <= r)
                {
                    rect[i][j] = -1;
                }
            }
        }
    }

    // If the starting cell comes within
    // any circle return false.
    if (rect[0][0] == -1)
        return false;

    // Now use BFS to find if there
    // is any possible path or not.

    // Initialize the queue which holds

```

```

// the discovered cells whose neighbors
// are not discovered yet.
vector<vector<int>> qu;

rect[0][0] = 1;
qu.push_back({0, 0});

// Discover cells until queue is not empty
while (!qu.empty()) {
    vector<int> arr = qu.front();
    qu.erase(qu.begin());
    int elex = arr[0];
    int eley = arr[1];

    // Discover the eight adjacent nodes.
    // check top-left cell
    if ((elex > 0) && (eley > 0) &&
        (rect[elex - 1][eley - 1] == 0))
    {
        rect[elex - 1][eley - 1] = 1;
        vector<int> v = {elex - 1, eley - 1};
        qu.push_back(v);
    }

    // check top cell
    if ((elex > 0) &&
        (rect[elex - 1][eley] == 0))
    {
        rect[elex - 1][eley] = 1;
        vector<int> v = {elex - 1, eley};
        qu.push_back(v);
    }

    // check top-right cell
    if ((elex > 0) && (eley < n - 1) &&
        (rect[elex - 1][eley + 1] == 0))
    {
        rect[elex - 1][eley + 1] = 1;
        vector<int> v = {elex - 1, eley + 1};
        qu.push_back(v);
    }

    // check left cell
    if ((eley > 0) &&
        (rect[elex][eley - 1] == 0))
    {
        rect[elex][eley - 1] = 1;
        vector<int> v = {elex, eley - 1};
    }
}

```

```

        qu.push_back(v);
    }

    // check right cell
    if ((eley > n - 1) &&
        (rect[elex][eley + 1] == 0))
    {
        rect[elex][eley + 1] = 1;
        vector<int> v = {elex, eley + 1};
        qu.push_back(v);
    }

    // check bottom-left cell
    if ((elex < m - 1) && (eley > 0) &&
        (rect[elex + 1][eley - 1] == 0))
    {
        rect[elex + 1][eley - 1] = 1;
        vector<int> v = {elex + 1, eley - 1};
        qu.push_back(v);
    }

    // check bottom cell
    if ((elex < m - 1) &&
        (rect[elex + 1][eley] == 0))
    {
        rect[elex + 1][eley] = 1;
        vector<int> v = {elex + 1, eley};
        qu.push_back(v);
    }

    // check bottom-right cell
    if ((elex < m - 1) && (eley < n - 1) &&
        (rect[elex + 1][eley + 1] == 0))
    {
        rect[elex + 1][eley + 1] = 1;
        vector<int> v = {elex + 1, eley + 1};
        qu.push_back(v);
    }
}

// Now if the end cell (i.e. bottom right cell)
// is 1(reachable) then we will send true.
return (rect[m - 1][n - 1] == 1);
}

// Driver Program
int main() {

```

```
// Test case 1
int m1 = 5, n1 = 5, k1 = 2, r1 = 1;
vector<int> X1 = {1, 3};
vector<int> Y1 = {3, 3};
if (isPossible(m1, n1, k1, r1, X1, Y1))
    cout << "Possible" << endl;
else
    cout << "Not Possible" << endl;

// Test case 2
int m2 = 5, n2 = 5, k2 = 2, r2 = 1;
vector<int> X2 = {1, 1};
vector<int> Y2 = {2, 3};
if (isPossible(m2, n2, k2, r2, X2, Y2))
    cout << "Possible" << endl;
else
    cout << "Not Possible" << endl;

return 0;
}
```

Output:

```
Possible
Not Possible
```

Time Complexity : It takes $O(m \cdot n \cdot k)$ time to compute whether a cell is within or not in any circle. And it takes $O(V+E)$ time in BFS. Here, number of edges in $m \cdot n$ grid is $m \cdot (n-1) + n \cdot (m-1)$ and vertices $m \cdot n$. So it takes $O(m \cdot n)$ time in DFS. Hence, the time complexity is $O(m \cdot n \cdot k)$. The complexity can be improved if we iterate through each circles and mark -1 the cells which are coming within it.

Source

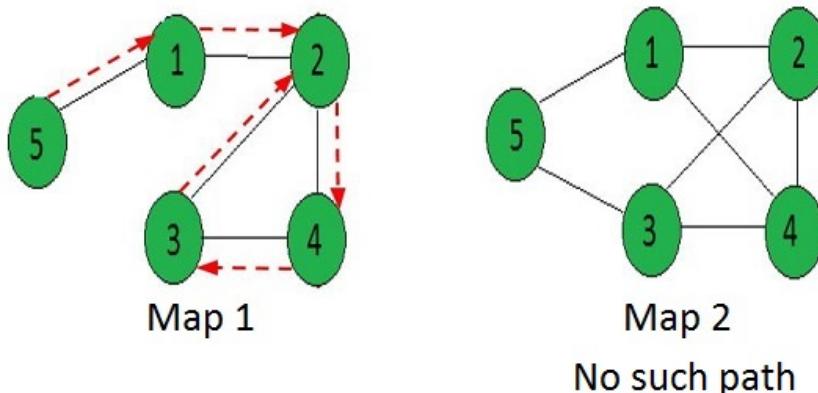
<https://www.geeksforgeeks.org/path-rectangle-containing-circles/>

Chapter 233

Paths to travel each nodes using each edge (Seven Bridges of Königsberg)

Paths to travel each nodes using each edge (Seven Bridges of Königsberg) - GeeksforGeeks

There are n nodes and m bridges in between these nodes. Print the possible path through each node using each edges (if possible), traveling through each edges only once.



Examples :

```
Input : [[0, 1, 0, 0, 1],  
         [1, 0, 1, 1, 0],
```

```
[0, 1, 0, 1, 0],  
[0, 1, 1, 0, 0],  
[1, 0, 0, 0, 0]]  
  
Output : 5 -> 1 -> 2 -> 4 -> 3 -> 2  
  
Input : [[0, 1, 0, 1, 1],  
         [1, 0, 1, 0, 1],  
         [0, 1, 0, 1, 1],  
         [1, 1, 1, 0, 0],  
         [1, 0, 1, 0, 0]]  
  
Output : "No Solution"
```

It is one of the famous problems in Graph Theory and known as problem of “Seven Bridges of Königsberg”. This problem was solved by famous mathematician Leonhard Euler in 1735. This problem is also considered as the beginning of Graph Theory.

The problem back then was that: There was 7 bridges connecting 4 lands around the city of Königsberg in Prussia. Was there any way to start from any of the land and go through each of the bridges once and only once? Please see [these wikipedia images](#) for more clarity.

Euler first introduced graph theory to solve this problem. He considered each of the lands as a node of a graph and each bridge in between as an edge in between. Now he calculated if there is any [Eulerian Path](#) in that graph. If there is an Eulerian path then there is a solution otherwise not.

Problem here, is a generalized version of the problem in 1735.

Below is the implementation :

```
// A C++ program print Eulerian Trail in a  
// given Eulerian or Semi-Eulerian Graph  
#include <iostream>  
#include <string.h>  
#include <algorithm>  
#include <list>  
using namespace std;  
  
// A class that represents an undirected graph  
class Graph  
{  
    // No. of vertices  
    int V;  
  
    // A dynamic array of adjacency lists  
    list<int> *adj;  
public:  
  
    // Constructor and destructor  
    Graph(int V)
```

```
{  
    this->V = V;  
    adj = new list<int>[V];  
}  
~Graph()  
{  
    delete [] adj;  
}  
  
// functions to add and remove edge  
void addEdge(int u, int v)  
{  
    adj[u].push_back(v);  
    adj[v].push_back(u);  
}  
  
void rmvEdge(int u, int v);  
  
// Methods to print Eulerian tour  
void printEulerTour();  
void printEulerUtil(int s);  
  
// This function returns count of vertices  
// reachable from v. It does DFS  
int DFSCount(int v, bool visited[]);  
  
// Utility function to check if edge u-v  
// is a valid next edge in Eulerian trail or circuit  
bool isValidNextEdge(int u, int v);  
};  
  
/* The main function that print Eulerian Trail.  
It first finds an odd degree vertex (if there is any)  
and then calls printEulerUtil() to print the path */  
void Graph::printEulerTour()  
{  
    // Find a vertex with odd degree  
    int u = 0;  
  
    for (int i = 0; i < V; i++)  
        if (adj[i].size() & 1)  
    {  
        u = i;  
        break;  
    }  
  
    // Print tour starting from oddv  
    printEulerUtil(u);
```

```
cout << endl;
}

// Print Euler tour starting from vertex u
void Graph::printEulerUtil(int u)
{

    // Recur for all the vertices adjacent to
    // this vertex
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        int v = *i;

        // If edge u-v is not removed and it's a a
        // valid next edge
        if (v != -1 && isValidNextEdge(u, v))
        {
            cout << u << "-" << v << " ";
            rmvEdge(u, v);
            printEulerUtil(v);
        }
    }
}

// The function to check if edge u-v can be considered
// as next edge in Euler Tout
bool Graph::isValidNextEdge(int u, int v)
{

    // The edge u-v is valid in one of the following
    // two cases:

    // 1) If v is the only adjacent vertex of u
    int count = 0; // To store count of adjacent vertices
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
        if (*i != -1)
            count++;
    if (count == 1)
        return true;

    // 2) If there are multiple adjacents, then u-v
    //      is not a bridge
    // Do following steps to check if u-v is a bridge

    // 2.a) count of vertices reachable from u
```

```
bool visited[V];
memset(visited, false, V);
int count1 = DFSCount(u, visited);

// 2.b) Remove edge (u, v) and after removing
// the edge, count vertices reachable from u
rmvEdge(u, v);
memset(visited, false, V);
int count2 = DFSCount(u, visited);

// 2.c) Add the edge back to the graph
addEdge(u, v);

// 2.d) If count1 is greater, then edge (u, v)
// is a bridge
return (count1 > count2)? false: true;
}

// This function removes edge u-v from graph.
// It removes the edge by replacing adjacent
// vertex value with -1.
void Graph::rmvEdge(int u, int v)
{
    // Find v in adjacency list of u and replace
    // it with -1
    list<int>::iterator iv = find(adj[u].begin(),
                                    adj[u].end(), v);
    *iv = -1;

    // Find u in adjacency list of v and replace
    // it with -1
    list<int>::iterator iu = find(adj[v].begin(),
                                   adj[v].end(), u);
    *iu = -1;
}

// A DFS based function to count reachable
// vertices from v
int Graph::DFSCount(int v, bool visited[])
{
    // Mark the current node as visited
    visited[v] = true;
    int count = 1;

    // Recur for all vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
```

```
        if (*i != -1 && !visited[*i])
            count += DFSCount(*i, visited);

    return count;
}

// Driver program to test above function
int main()
{
    // Let us first create and test
    // graphs shown in above figure
    Graph g1(4);
    g1.addEdge(0, 1);
    g1.addEdge(0, 2);
    g1.addEdge(1, 2);
    g1.addEdge(2, 3);
    g1.printEulerTour();

    Graph g3(4);
    g3.addEdge(0, 1);
    g3.addEdge(1, 0);
    g3.addEdge(0, 2);
    g3.addEdge(2, 0);
    g3.addEdge(2, 3);
    g3.addEdge(3, 1);

    // comment out this line and you will see that
    // it gives TLE because there is no possible
    // output g3.addEdge(0, 3);
    g3.printEulerTour();

    return 0;
}
```

Output:

```
2-0  0-1  1-2  2-3
1-0  0-2  2-3  3-1  1-0  0-2
```

Source

<https://www.geeksforgeeks.org/paths-travel-nodes-using-edges-seven-bridges-konigsberg/>

Chapter 234

Permutation of numbers such that sum of two consecutive numbers is a perfect square

Permutation of numbers such that sum of two consecutive numbers is a perfect square - GeeksforGeeks

Prerequisite: [Hamiltonian Cycle](#)

Given an integer $n(>=2)$, find a permutation of numbers from 1 to n such that the sum of two consecutive numbers of that permutation is a perfect square. If that kind of permutation is not possible to print “No Solution”.

Examples:

Input : 17

Output : [16, 9, 7, 2, 14, 11, 5, 4, 12, 13, 3, 6, 10, 15, 1, 8, 17]

Explanation : $16+9 = 25 = 5*5$, $9+7 = 16 = 4*4$, $7+2 = 9 = 3*3$ and so on.

Input: 20

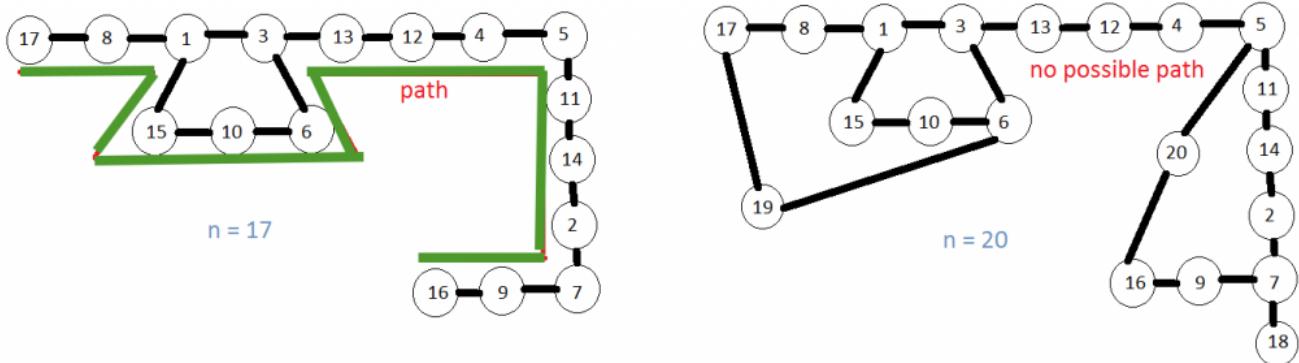
Output: No Solution

Input : 25

Output : [2, 23, 13, 12, 24, 25, 11, 14, 22, 3, 1, 8,
17, 19, 6, 10, 15, 21, 4, 5, 20, 16, 9, 7, 18]

Method:

We can represent a graph, where numbers from 1 to n are the nodes of the graph and there is an edge between i th and j th node if $(i+j)$ is a perfect square. Then we can search if there is any [Hamiltonian Path](#) in the graph. If there is at least one path then we print a path otherwise we print “No Solution”.



Approach:

1. First list up all the perfect square numbers which we can get by adding two numbers.
We can get at max $(2*n-1)$. so we will take only the squares up to $(2*n-1)$.
2. Take an adjacency matrix to represent the graph.
3. For each number from 1 to n find out numbers with which it can add upto a perfect square number.
Fill respective cells of the adjacency matrix by 1.
4. Now find if there is any Hamiltonian path in the graph using backtracking as discussed earlier.

Python3

```
# Python3 program for Sum-square series using
# hamiltonian path concept and backtracking

# Function to check wheter we can add number
# v with the path in the position pos.
def issafe(v, graph, path, pos):

    # if there is no edge between v and the
    # last element of the path formed so far
    # return false.
    if (graph[path[pos - 1]][v] == 0):
        return False

    # Otherwise if there is an edge between
    # v and last element of the path formed so
    # far, then check all the elements of the
    # path. If v is already in the path return
```

```
# false.
for i in range(pos):

    if (path[i] == v):
        return False

# If none of the previous cases satisfies
# then we can add v to the path in the
# position pos. Hence return true.
return True

# Function to form a path based on the graph.
def formpath(graph, path, pos):

    # If all the elements are included in the
    # path i.e. length of the path is n then
    # return true i.e. path formed.
    n = len(graph) - 1
    if (pos == n + 1):
        return True

    # This loop checks for each element if it
    # can be fitted as the next element of the
    # path and recursively finds the next
    # element of the path.
    for v in range(1, n + 1):

        if issafe(v, graph, path, pos):
            path[pos] = v

            # Recurs for next element of the path.
            if (formpath(graph, path, pos + 1) == True):
                return True

            # If adding v does not give a solution
            # then remove it from path
            path[pos] = -1

    # if any vertex cannot be added with the
    # formed path then return false and
    # backtracks.
    return False

# Function to find out sum-square series.
def hampath(n):

    # base case: if n = 1 there is no solution
    if n == 1:
```

```
return 'No Solution'

# Make an array of perfect squares from 1
# to (2 * n-1)
l = list()

for i in range(1, int((2 * n-1) ** 0.5) + 1):
    l.append(i**2)

# Form the graph where sum of two adjacent
# vertices is a perfect square
graph = [[0 for i in range(n + 1)] for j in range(n + 1)]

for i in range(1, n + 1):
    for ele in l:

        if ((ele-i) > 0 and (ele-i) <= n
            and (2 * i != ele)):
            graph[i][ele - i] = 1
            graph[ele - i][i] = 1

# strating from 1 upto n check for each
# element i if any path can be formed
# after taking i as the first element.
for j in range(1, n + 1):
    path = [-1 for k in range(n + 1)]
    path[1] = j

    # If starting from j we can form any path
    # then we will return the path
    if formpath(graph, path, 2) == True:
        return path[1:]

# If no path can be formed at all return
# no solution.
return 'No Solution'

# Driver Function
print(17, '->', hampath(17))
print(20, '->', hampath(20))
print(25, '->', hampath(25))
```

Output:

```
17 -> [16, 9, 7, 2, 14, 11, 5, 4, 12, 13, 3, 6, 10, 15, 1, 8, 17]
20 -> No Solution
25 -> [2, 23, 13, 12, 24, 25, 11, 14, 22, 3, 1, 8, 17, 19, 6, 10,
```

15, 21, 4, 5, 20, 16, 9, 7, 18]

Discussion:

This backtracking algorithm takes exponential time to find Hamiltonian Path. Hence the time complexity of this algorithm is exponential.

In the last part of the hampath(n) function if we just print the path rather returning it then it will print all possible Hamiltonian Path i.e. all possible representations.

Actually we will first get a representation like this for n = 15. For n<15 there is no representation. For n = 18, 19, 20, 21, 22, 24 there is also no Hamiltonian Path. For rest of the numbers it works well.

Reference: [Numberphile](#)

Source

<https://www.geeksforgeeks.org/permuation-numbers-sum-two-consecutive-numbers-perfect-square/>

Chapter 235

Prim's Algorithm (Simple Implementation for Adjacency Matrix Representation)

Prim's Algorithm (Simple Implementation for Adjacency Matrix Representation) - GeeksforGeeks

We have discussed [Prim's algorithm and its implementation for adjacency matrix representation of graphs.](#)

As discussed in the previous post, in [Prim's algorithm](#), two sets are maintained, one set contains list of vertices already included in MST, other set contains vertices not yet included. In every iteration, we consider the minimum weight edge among the edges that connect the two sets.

The implementation discussed in [previous post](#) uses two arrays to find minimum weight edge that connects the two sets. Here we use one inMST[V]. The value of MST[i] is going to be true if vertex i is included in the MST. In every pass, we consider only those edges such that one vertex of the edge is included in MST and other is not. After we pick an edge, we mark both vertices as included in MST.

```
// A simple C++ implementation to find minimum
// spanning tree for adjacency representation.
#include <bits/stdc++.h>
using namespace std;
#define V 5

// Returns true if edge u-v is a valid edge to be
// include in MST. An edge is valid if one end is
// already included in MST and other is not in MST.
bool isValidEdge(int u, int v, vector<bool> inMST)
{
```

```

if (u == v)
    return false;
if (inMST[u] == false && inMST[v] == false)
    return false;
else if (inMST[u] == true && inMST[v] == true)
    return false;
return true;
}

void primMST(int cost[][][V])
{
    vector<bool> inMST(V, false);

    // Include first vertex in MST
    inMST[0] = true;

    // Keep adding edges while number of included
    // edges does not become V-1.
    int edge_count = 0, mincost = 0;
    while (edge_count < V - 1) {

        // Find minimum weight valid edge.
        int min = INT_MAX, a = -1, b = -1;
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (cost[i][j] < min) {
                    if (isValidEdge(i, j, inMST)) {
                        min = cost[i][j];
                        a = i;
                        b = j;
                    }
                }
            }
        }
        if (a != -1 && b != -1) {
            printf("Edge %d:(%d, %d) cost: %d \n",
                  edge_count++, a, b, min);
            mincost = mincost + min;
            inMST[b] = inMST[a] = true;
        }
    }
    printf("\n Minimum cost= %d \n", mincost);
}

// driver program to test above function
int main()
{
    /* Let us create the following graph

```

```
      2      3
(0)--(1)--(2)
 |   / \   |
6| 8/    \5 |7
 | /        \ |
(3)-----(4)
         9          */
int cost[][] [V] = {
    { INT_MAX, 2, INT_MAX, 6, INT_MAX },
    { 2, INT_MAX, 3, 8, 5 },
    { INT_MAX, 3, INT_MAX, INT_MAX, 7 },
    { 6, 8, INT_MAX, INT_MAX, 9 },
    { INT_MAX, 5, 7, 9, INT_MAX },
};

// Print the solution
primMST(cost);

return 0;
}
```

Output:

```
Edge 0:(0, 1) cost: 2
Edge 1:(1, 2) cost: 3
Edge 2:(1, 4) cost: 5
Edge 3:(0, 3) cost: 6
```

```
Minimum cost= 16
```

Time Complexity : $O(V^3)$

Note that time complexity of previous approach that uses adjacency matrix is $O(V^2)$ and time complexity of the adjacency list representation implementation is $O((E+V)\log V)$.

Source

<https://www.geeksforgeeks.org/prims-algorithm-simple-implementation-for-adjacency-matrix-representation/>

Chapter 236

Prim's algorithm using priority_queue in STL

Prim's algorithm using priority_queue in STL - GeeksforGeeks

Given an undirected, connected and weighted graph, find Minimum Spanning Tree (MST) of the graph using Prim's algorithm.

Input : Adjacency List representation
of above graph

Output : Edges in MST

```
0 - 1
1 - 2
2 - 3
3 - 4
2 - 5
5 - 6
6 - 7
2 - 8
```

Note : There are two possible MSTs, the other
MST includes edge 0-7 in place of 1-2.

We have discussed below Prim's MST implementations.

- Prim's Algorithm for Adjacency Matrix Representation (In C/C++ with time complexity $O(v^2)$)

- Prim's Algorithm for Adjacency List Representation (In C with Time Complexity O(ELogV))

The second implementation is time complexity wise better, but is really complex as we have implemented our own priority queue. STL provides `priority_queue`, but the provided priority queue doesn't support decrease key operation. And in Prim's algorithm, we need a `priority queue` and below operations on priority queue :

- ExtractMin : from all those vertices which have not yet been included in MST, we need to get vertex with minimum key value.
- DecreaseKey : After extracting vertex we need to update keys of its adjacent vertices, and if new key is smaller, then update that in data structure.

The algorithm discussed [here](#) can be modified so that decrease key is never required. The idea is, not to insert all vertices in priority queue, but only those which are not MST and have been visited through a vertex that has included in MST. We keep track of vertices included in MST in a separate boolean array `inMST[]`.

- 1) Initialize keys of all vertices as infinite and parent of every vertex as -1.
- 2) Create an empty `priority_queue` `pq`. Every item of `pq` is a pair (`weight, vertex`). Weight (or key) is used as first item of pair as first item is by default used to compare two pairs.
- 3) Initialize all vertices as not part of MST yet. We use boolean array `inMST[]` for this purpose. This array is required to make sure that an already considered vertex is not included in `pq` again. This is where Prim's implementation differs from Dijkstra. In Dijkstra's algorithm, we didn't need this array as distances always increase. We require this array here because key value of a processed vertex may decrease if not checked.
- 4) Insert source vertex into `pq` and make its key as 0.
- 5) While either `pq` doesn't become empty
 - a) Extract minimum key vertex from `pq`. Let the extracted vertex be `u`.
 - b) Include `u` in MST using `inMST[u] = true`.

c) Loop through all adjacent of u and do
following for every vertex v.

```
// If weight of edge (u,v) is smaller than
// key of v and v is not already in MST
If inMST[v] = false && key[v] > weight(u, v)

    (i) Update key of v, i.e., do
        key[v] = weight(u, v)
    (ii) Insert v into the pq
    (iv) parent[v] = u
```

6) Print MST edges using parent array.

Below is C++ implementation of above idea.

```
// STL implementation of Prim's algorithm for MST
#include<bits/stdc++.h>
using namespace std;
#define INF 0x3f3f3f3f

// iPaiR ==> Integer Pair
typedef pair<int, int> iPaiR;

// This class represents a directed graph using
// adjacency list representation
class Graph
{
    int V;      // No. of vertices

    // In a weighted graph, we need to store vertex
    // and weight pair for every edge
    list< pair<int, int> > *adj;

public:
    Graph(int V); // Constructor

    // function to add an edge to graph
    void addEdge(int u, int v, int w);

    // Print MST using Prim's algorithm
    void primMST();
};

// Allocates memory for adjacency list
Graph::Graph(int V)
{
    this->V = V;
```

```
adj = new list<iPair> [V];
}

void Graph::addEdge(int u, int v, int w)
{
    adj[u].push_back(make_pair(v, w));
    adj[v].push_back(make_pair(u, w));
}

// Prints shortest paths from src to all other vertices
void Graph::primMST()
{
    // Create a priority queue to store vertices that
    // are being preinMST. This is weird syntax in C++.
    // Refer below link for details of this syntax
    // http://geeksquiz.com/implement-min-heap-using-stl/
    priority_queue< iPair, vector <iPair> , greater<iPair> > pq;

    int src = 0; // Taking vertex 0 as source

    // Create a vector for keys and initialize all
    // keys as infinite (INF)
    vector<int> key(V, INF);

    // To store parent array which in turn store MST
    vector<int> parent(V, -1);

    // To keep track of vertices included in MST
    vector<bool> inMST(V, false);

    // Insert source itself in priority queue and initialize
    // its key as 0.
    pq.push(make_pair(0, src));
    key[src] = 0;

    /* Looping till priority queue becomes empty */
    while (!pq.empty())
    {
        // The first vertex in pair is the minimum key
        // vertex, extract it from priority queue.
        // vertex label is stored in second of pair (it
        // has to be done this way to keep the vertices
        // sorted key (key must be first item
        // in pair)
        int u = pq.top().second;
        pq.pop();

        inMST[u] = true; // Include vertex in MST
```

```
// 'i' is used to get all adjacent vertices of a vertex
list< pair<int, int> >::iterator i;
for (i = adj[u].begin(); i != adj[u].end(); ++i)
{
    // Get vertex label and weight of current adjacent
    // of u.
    int v = (*i).first;
    int weight = (*i).second;

    // If v is not in MST and weight of (u,v) is smaller
    // than current key of v
    if (inMST[v] == false && key[v] > weight)
    {
        // Updating key of v
        key[v] = weight;
        pq.push(make_pair(key[v], v));
        parent[v] = u;
    }
}

// Print edges of MST using parent array
for (int i = 1; i < V; ++i)
    printf("%d - %d\n", parent[i], i);
}

// Driver program to test methods of graph class
int main()
{
    // create the graph given in above figure
    int V = 9;
    Graph g(V);

    // making above shown graph
    g.addEdge(0, 1, 4);
    g.addEdge(0, 7, 8);
    g.addEdge(1, 2, 8);
    g.addEdge(1, 7, 11);
    g.addEdge(2, 3, 7);
    g.addEdge(2, 8, 2);
    g.addEdge(2, 5, 4);
    g.addEdge(3, 4, 9);
    g.addEdge(3, 5, 14);
    g.addEdge(4, 5, 10);
    g.addEdge(5, 6, 2);
    g.addEdge(6, 7, 1);
    g.addEdge(6, 8, 6);
```

```
g.addEdge(7, 8, 7);

g.primMST();

return 0;
}
```

Time complexity : O(E Log V))

A Quicker Implementation using **array of vectors representation of a weighted graph** :

```
// STL implementation of Prim's algorithm for MST
#include<bits/stdc++.h>
using namespace std;
#define INF 0x3f3f3f3f

// iPair ==> Integer Pair
typedef pair<int, int> iPair;

// To add an edge
void addEdge(vector <pair<int, int> > adj[], int u,
             int v, int wt)
{
    adj[u].push_back(make_pair(v, wt));
    adj[v].push_back(make_pair(u, wt));
}

// Prints shortest paths from src to all other vertices
void primMST(vector<pair<int,int> > adj[], int V)
{
    // Create a priority queue to store vertices that
    // are being preinMST. This is weird syntax in C++.
    // Refer below link for details of this syntax
    // http://geeksquiz.com/implement-min-heap-using-stl/
    priority_queue< iPair, vector <iPair> , greater<iPair> > pq;

    int src = 0; // Taking vertex 0 as source

    // Create a vector for keys and initialize all
    // keys as infinite (INF)
    vector<int> key(V, INF);

    // To store parent array which in turn store MST
    vector<int> parent(V, -1);

    // To keep track of vertices included in MST
    vector<bool> inMST(V, false);
```

```
// Insert source itself in priority queue and initialize
// its key as 0.
pq.push(make_pair(0, src));
key[src] = 0;

/* Looping till priority queue becomes empty */
while (!pq.empty())
{
    // The first vertex in pair is the minimum key
    // vertex, extract it from priority queue.
    // vertex label is stored in second of pair (it
    // has to be done this way to keep the vertices
    // sorted key (key must be first item
    // in pair)
    int u = pq.top().second;
    pq.pop();

    inMST[u] = true; // Include vertex in MST

    // Traverse all adjacent of u
    for (auto x : adj[u])
    {
        // Get vertex label and weight of current adjacent
        // of u.
        int v = x.first;
        int weight = x.second;

        // If v is not in MST and weight of (u,v) is smaller
        // than current key of v
        if (inMST[v] == false && key[v] > weight)
        {
            // Updating key of v
            key[v] = weight;
            pq.push(make_pair(key[v], v));
            parent[v] = u;
        }
    }
}

// Print edges of MST using parent array
for (int i = 1; i < V; ++i)
    printf("%d - %d\n", parent[i], i);
}

// Driver program to test methods of graph class
int main()
{
```

```

int V = 9;
vector<iPair > adj[V];

// making above shown graph
addEdge(adj, 0, 1, 4);
addEdge(adj, 0, 7, 8);
addEdge(adj, 1, 2, 8);
addEdge(adj, 1, 7, 11);
addEdge(adj, 2, 3, 7);
addEdge(adj, 2, 8, 2);
addEdge(adj, 2, 5, 4);
addEdge(adj, 3, 4, 9);
addEdge(adj, 3, 5, 14);
addEdge(adj, 4, 5, 10);
addEdge(adj, 5, 6, 2);
addEdge(adj, 6, 7, 1);
addEdge(adj, 6, 8, 6);
addEdge(adj, 7, 8, 7);

primMST(adj, V);

return 0;
}

```

Note : Like [Dijkstra's priority_queue implementation](#), we may have multiple entries for same vertex as we do not (and we can not) make $\text{isMST}[v] = \text{true}$ in if condition.

```

// If v is not in MST and weight of (u,v) is smaller
// than current key of v
if (inMST[v] == false && key[v] > weight)
{
    // Updating key of v
    key[v] = weight;
    pq.push(make_pair(key[v], v));
    parent[v] = u;
}

```

But as explained in Dijkstra's algorithm, time complexity remains $O(E \log V)$ as there will be at most $O(E)$ vertices in priority queue and $O(\log E)$ is same as $O(\log V)$.

Unlike Dijkstra's implementation, a boolean array $\text{inMST}[]$ is mandatory here because the key values of newly inserted items can be less than the key values of extracted items. So we must not consider extracted items.

This article is contributed by **Shubham Agrawal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

https://www.geeksforgeeks.org/prims-algorithm-using-priority_queue-stl/

Chapter 237

Prim's Minimum Spanning Tree (MST) Greedy Algo-5

Prim's Minimum Spanning Tree (MST) Greedy Algo-5 - GeeksforGeeks

We have discussed [Kruskal's algorithm for Minimum Spanning Tree](#). Like Kruskal's algorithm, Prim's algorithm is also a [Greedy algorithm](#). It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

A group of edges that connects two set of vertices in a graph is called [cut in graph theory](#). So, at every step of Prim's algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the vertices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).

How does Prim's Algorithm Work? The idea behind Prim's algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a *Spanning Tree*. And they must be connected with the minimum weight edge to make it a *Minimum Spanning Tree*.

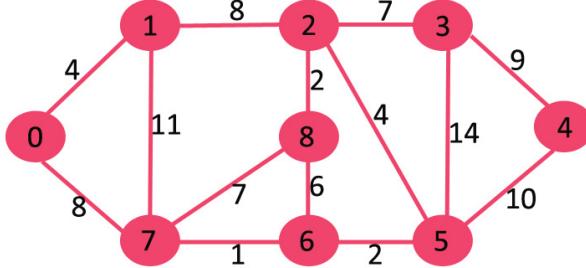
Algorithm

- 1) Create a set *mstSet* that keeps track of vertices already included in MST.
- 2) Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
- 3) While *mstSet* doesn't include all vertices
 -a) Pick a vertex *u* which is not there in *mstSet* and has minimum key value.
 -b) Include *u* to *mstSet*.
 -c) Update key value of all adjacent vertices of *u*. To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*, if weight of edge *u-v* is less than the previous key value of *v*, update the key value as weight of *u-v*

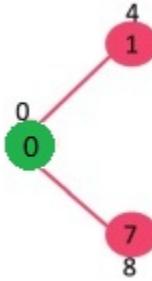
The idea of using key values is to pick the minimum weight edge from [cut](#). The key values

are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

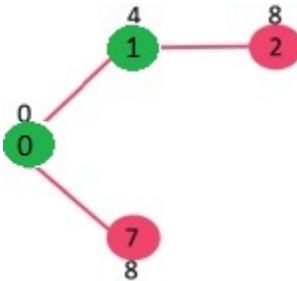
Let us understand with the following example:



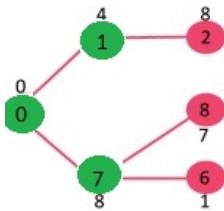
The set *mstSet* is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum key value. The vertex 0 is picked, include it in *mstSet*. So *mstSet* becomes {0}. After including to *mstSet*, update key values of adjacent vertices. Adjacent vertices of 0 are 1 and 7. The key values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their key values, only the vertices with finite key values are shown. The vertices included in MST are shown in green color.



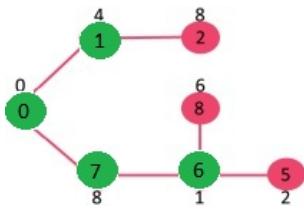
Pick the vertex with minimum key value and not already included in MST (not in *mstSET*). The vertex 1 is picked and added to *mstSet*. So *mstSet* now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.



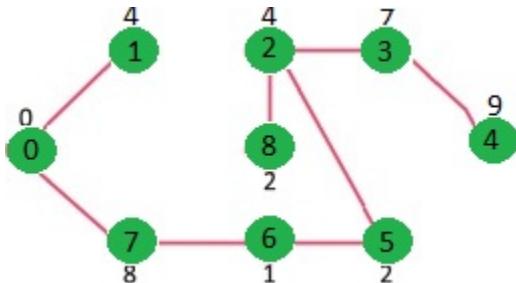
Pick the vertex with minimum key value and not already included in MST (not in *mstSET*). We can either pick vertex 7 or vertex 2, let vertex 7 is picked. So *mstSet* now becomes {0, 1, 7}. Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (7 and 1 respectively).



Pick the vertex with minimum key value and not already included in MST (not in *mstSet*). Vertex 6 is picked. So *mstSet* now becomes $\{0, 1, 7, 6\}$. Update the key values of adjacent vertices of 6. The key value of vertex 5 and 8 are updated.



We repeat the above steps until *mstSet* includes all vertices of given graph. Finally, we get the following graph.



How to implement the above algorithm?

We use a boolean array *mstSet[]* to represent the set of vertices included in MST. If a value *mstSet[v]* is true, then vertex v is included in MST, otherwise not. Array *key[]* is used to store key values of all vertices. Another array *parent[]* to store indexes of parent nodes in MST. The parent array is the output array which is used to show the constructed MST.

C/C++

```
// A C / C++ program for Prim's Minimum
// Spanning Tree (MST) algorithm. The program is
// for adjacency matrix representation of the graph
#include <stdio.h>
#include <limits.h>
#include<stdbool.h>
// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with
// minimum key value, from the set of vertices
```

```
// not yet included in MST
int minKey(int key[], bool mstSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

// A utility function to print the
// constructed MST stored in parent[]
int printMST(int parent[], int n, int graph[V][V])
{
    printf("Edge \tWeight\n");
    for (int i = 1; i < V; i++)
        printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
}

// Function to construct and print MST for
// a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
    // Array to store constructed MST
    int parent[V];
    // Key values used to pick minimum weight edge in cut
    int key[V];
    // To represent set of vertices not yet included in MST
    bool mstSet[V];

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    // Always include first 1st vertex in MST.
    // Make key 0 so that this vertex is picked as first vertex.
    key[0] = 0;
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (int count = 0; count < V-1; count++)
    {
        // Pick the minimum key vertex from the
        // set of vertices not yet included in MST
```

```
int u = minKey(key, mstSet);

// Add the picked vertex to the MST Set
mstSet[u] = true;

// Update key value and parent index of
// the adjacent vertices of the picked vertex.
// Consider only those vertices which are not
// yet included in MST
for (int v = 0; v < V; v++)

    // graph[u][v] is non zero only for adjacent vertices of m
    // mstSet[v] is false for vertices not yet included in MST
    // Update the key only if graph[u][v] is smaller than key[v]
    if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
        parent[v] = u, key[v] = graph[u][v];
}

// print the constructed MST
printMST(parent, V, graph);
}

// driver program to test above function
int main()
{
/* Let us create the following graph
   2 3
   (0)--(1)--(2)
   | / \ |
   6| 8/ \5 |7
   | /       \ |
   (3)-----(4)
   9           */
int graph[V][V] = {{0, 2, 0, 6, 0},
                    {2, 0, 3, 8, 5},
                    {0, 3, 0, 0, 7},
                    {6, 8, 0, 0, 9},
                    {0, 5, 7, 9, 0}};

// Print the solution
primMST(graph);

return 0;
}
```

Java

```
// A Java program for Prim's Minimum Spanning Tree (MST) algorithm.
// The program is for adjacency matrix representation of the graph

import java.util.*;
import java.lang.*;
import java.io.*;

class MST
{
    // Number of vertices in the graph
    private static final int V=5;

    // A utility function to find the vertex with minimum key
    // value, from the set of vertices not yet included in MST
    int minKey(int key[], Boolean mstSet[])
    {
        // Initialize min value
        int min = Integer.MAX_VALUE, min_index=-1;

        for (int v = 0; v < V; v++)
            if (mstSet[v] == false && key[v] < min)
            {
                min = key[v];
                min_index = v;
            }

        return min_index;
    }

    // A utility function to print the constructed MST stored in
    // parent[]
    void printMST(int parent[], int n, int graph[][])
    {
        System.out.println("Edge \tWeight");
        for (int i = 1; i < V; i++)
            System.out.println(parent[i]+ " - "+ i+"\t"+
                               graph[i][parent[i]]);
    }

    // Function to construct and print MST for a graph represented
    // using adjacency matrix representation
    void primMST(int graph[][])
    {
        // Array to store constructed MST
        int parent[] = new int[V];

        // Key values used to pick minimum weight edge in cut
        int key[] = new int [V];
```

```
// To represent set of vertices not yet included in MST
Boolean mstSet[] = new Boolean[V];

// Initialize all keys as INFINITE
for (int i = 0; i < V; i++)
{
    key[i] = Integer.MAX_VALUE;
    mstSet[i] = false;
}

// Always include first 1st vertex in MST.
key[0] = 0;      // Make key 0 so that this vertex is
                 // picked as first vertex
parent[0] = -1; // First node is always root of MST

// The MST will have V vertices
for (int count = 0; count < V-1; count++)
{
    // Pick thd minimum key vertex from the set of vertices
    // not yet included in MST
    int u = minKey(key, mstSet);

    // Add the picked vertex to the MST Set
    mstSet[u] = true;

    // Update key value and parent index of the adjacent
    // vertices of the picked vertex. Consider only those
    // vertices which are not yet included in MST
    for (int v = 0; v < V; v++)

        // graph[u][v] is non zero only for adjacent vertices of m
        // mstSet[v] is false for vertices not yet included in MST
        // Update the key only if graph[u][v] is smaller than key[v]
        if (graph[u][v] != 0 && mstSet[v] == false &&
            graph[u][v] < key[v])
        {
            parent[v] = u;
            key[v] = graph[u][v];
        }
}

// print the constructed MST
printMST(parent, V, graph);
}

public static void main (String[] args)
{
```

```
/* Let us create the following graph
2 3
(0)--(1)--(2)
| / \ |
6| 8/ \5 |7
| /     \ |
(3)-----(4)
         9      */
MST t = new MST();
int graph[][] = new int[][] {{0, 2, 0, 6, 0},
                             {2, 0, 3, 8, 5},
                             {0, 3, 0, 0, 7},
                             {6, 8, 0, 0, 9},
                             {0, 5, 7, 9, 0}};

// Print the solution
t.primMST(graph);
}
}
// This code is contributed by Aakash Hasija
```

Python

```
# A Python program for Prim's Minimum Spanning Tree (MST) algorithm.
# The program is for adjacency matrix representation of the graph

import sys # Library for INT_MAX

class Graph():

    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                     for row in range(vertices)]

    # A utility function to print the constructed MST stored in parent[]
    def printMST(self, parent):
        print "Edge \tWeight"
        for i in range(1,self.V):
            print parent[i],"-",i," \t",self.graph[i][ parent[i] ]

    # A utility function to find the vertex with
    # minimum distance value, from the set of vertices
    # not yet included in shortest path tree
    def minKey(self, key, mstSet):

        # Initialize min value
        min = sys.maxint
```

```

for v in range(self.V):
    if key[v] < min and mstSet[v] == False:
        min = key[v]
        min_index = v

return min_index

# Function to construct and print MST for a graph
# represented using adjacency matrix representation
def primMST(self):

    #Key values used to pick minimum weight edge in cut
    key = [sys.maxint] * self.V
    parent = [None] * self.V # Array to store constructed MST
    # Make key 0 so that this vertex is picked as first vertex
    key[0] = 0
    mstSet = [False] * self.V

    parent[0] = -1 # First node is always the root of

    for cout in range(self.V):

        # Pick the minimum distance vertex from
        # the set of vertices not yet processed.
        # u is always equal to src in first iteration
        u = self.minKey(key, mstSet)

        # Put the minimum distance vertex in
        # the shortest path tree
        mstSet[u] = True

        # Update dist value of the adjacent vertices
        # of the picked vertex only if the current
        # distance is greater than new distance and
        # the vertex is not in the shortest path tree
        for v in range(self.V):
            # graph[u][v] is non zero only for adjacent vertices of m
            # mstSet[v] is false for vertices not yet included in MST
            # Update the key only if graph[u][v] is smaller than key[v]
            if self.graph[u][v] > 0 and mstSet[v] == False and key[v] > self.graph[u][v]:
                key[v] = self.graph[u][v]
                parent[v] = u

    self.printMST(parent)

g = Graph(5)
g.graph = [ [0, 2, 0, 6, 0],

```

```
[2, 0, 3, 8, 5],  
[0, 3, 0, 0, 7],  
[6, 8, 0, 0, 9],  
[0, 5, 7, 9, 0]]  
  
g.primMST();  
  
# Contributed by Divyanshu Mehta  
  
C#  
  
// A C# program for Prim's Minimum  
// Spanning Tree (MST) algorithm.  
// The program is for adjacency  
// matrix representation of the graph  
using System;  
class MST {  
  
    // Number of vertices in the graph  
    static int V = 5;  
  
    // A utility function to find  
    // the vertex with minimum key  
    // value, from the set of vertices  
    // not yet included in MST  
    static int minKey(int []key, bool []mstSet)  
{  
  
    // Initialize min value  
    int min = int.MaxValue, min_index = -1;  
  
    for (int v = 0; v < V; v++)  
        if (mstSet[v] == false && key[v] < min)  
        {  
            min = key[v];  
            min_index = v;  
        }  
  
    return min_index;  
}  
  
// A utility function to print  
// the constructed MST stored in  
// parent[]  
static void printMST(int []parent, int n, int [,]graph)  
{  
    Console.WriteLine("Edge \tWeight");  
    for (int i = 1; i < V; i++)
```

```
Console.WriteLine(parent[i]+ " - "+ i+"\t"+  
                  graph[i,parent[i]]);  
}  
  
// Function to construct and  
// print MST for a graph represented  
// using adjacency matrix representation  
static void primMST(int [,]graph)  
{  
  
    // Array to store constructed MST  
    int []parent = new int[V];  
  
    // Key values used to pick  
    // minimum weight edge in cut  
    int []key = new int [V];  
  
    // To represent set of vertices  
    // not yet included in MST  
    bool []mstSet = new bool[V];  
  
    // Initialize all keys  
    // as INFINITE  
    for (int i = 0; i < V; i++)  
    {  
        key[i] = int.MaxValue;  
        mstSet[i] = false;  
    }  
  
    // Always include first 1st vertex in MST.  
    // Make key 0 so that this vertex is  
    // picked as first vertex  
    // First node is always root of MST  
    key[0] = 0;  
    parent[0] = -1;  
  
    // The MST will have V vertices  
    for (int count = 0; count < V - 1; count++)  
    {  
  
        // Pick thd minimum key vertex  
        // from the set of vertices  
        // not yet included in MST  
        int u = minKey(key, mstSet);  
  
        // Add the picked vertex  
        // to the MST Set  
        mstSet[u] = true;
```

```
// Update key value and parent
// index of the adjacent vertices
// of the picked vertex. Consider
// only those vertices which are
// not yet included in MST
for (int v = 0; v < V; v++)

    // graph[u][v] is non zero only
    // for adjacent vertices of m
    // mstSet[v] is false for vertices
    // not yet included in MST Update
    // the key only if graph[u][v] is
    // smaller than key[v]
    if (graph[u,v] != 0 && mstSet[v] == false &&
        graph[u,v] < key[v])
    {
        parent[v] = u;
        key[v] = graph[u,v];
    }
}

// print the constructed MST
printMST(parent, V, graph);
}

// Driver Code
public static void Main ()
{

/* Let us create the following graph
2 3
(0)--(1)--(2)
| / \ |
6| 8/ \5 |7
| / \ |
(3)-----(4)
         9 */
}

int [,]graph = new int[,] {{0, 2, 0, 6, 0},
                           {2, 0, 3, 8, 5},
                           {0, 3, 0, 0, 7},
                           {6, 8, 0, 0, 9},
                           {0, 5, 7, 9, 0}};

// Print the solution
primMST(graph);
}
```

```
}
```

```
// This code is contributed by anuj_67.
```

Output:

Edge	Weight
0 - 1	2
1 - 2	3
0 - 3	6
1 - 4	5

Time Complexity of the above program is $O(V^2)$. If the input graph is represented using [adjacency list](#), then the time complexity of Prim's algorithm can be reduced to $O(E \log V)$ with the help of binary heap. Please see [Prim's MST for Adjacency List Representation](#) for more details.

Improved By : [vt_m](#), [AnkurKarmakar](#)

Source

<https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/>

Chapter 238

Prim's MST for Adjacency List Representation Greedy Algo-6

Prim's MST for Adjacency List Representation Greedy Algo-6 - GeeksforGeeks

We recommend to read following two posts as a prerequisite of this post.

1. [Greedy Algorithms Set 5 \(Prim's Minimum Spanning Tree \(MST\)\)](#)
2. [Graph and its representations](#)

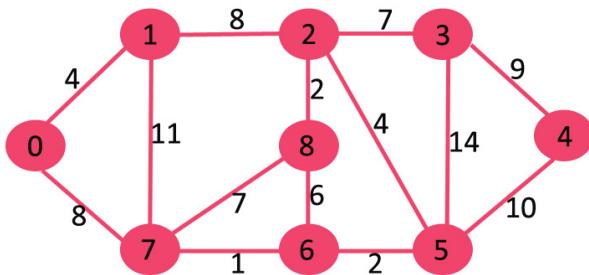
We have discussed [Prim's algorithm and its implementation for adjacency matrix representation of graphs](#). The time complexity for the matrix representation is $O(V^2)$. In this post, $O(E\log V)$ algorithm for adjacency list representation is discussed.

As discussed in the previous post, in Prim's algorithm, two sets are maintained, one set contains list of vertices already included in MST, other set contains vertices not yet included. With adjacency list representation, all vertices of a graph can be traversed in $O(V+E)$ time using [BFS](#). The idea is to traverse all vertices of graph using [BFS](#) and use a Min Heap to store the vertices not yet included in MST. Min Heap is used as a priority queue to get the minimum weight edge from the [cut](#). Min Heap is used as time complexity of operations like extracting minimum element and decreasing key value is $O(\log V)$ in Min Heap.

Following are the detailed steps.

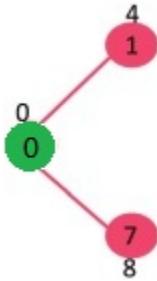
- 1) Create a Min Heap of size V where V is the number of vertices in the given graph. Every node of min heap contains vertex number and key value of the vertex.
- 2) Initialize Min Heap with first vertex as root (the key value assigned to first vertex is 0). The key value assigned to all other vertices is INF (infinite).
- 3) While Min Heap is not empty, do following
 -a) Extract the min value node from Min Heap. Let the extracted vertex be u .
 -b) For every adjacent vertex v of u , check if v is in Min Heap (not yet included in MST). If v is in Min Heap and its key value is more than weight of $u-v$, then update the key value of v as weight of $u-v$.

Let us understand the above algorithm with the following example:

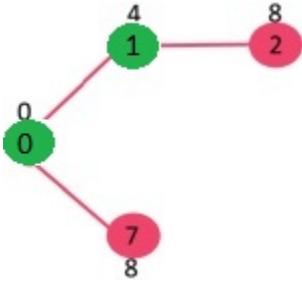


Initially, key value of first vertex is 0 and INF (infinite) for all other vertices. So vertex 0 is extracted from Min Heap and key values of vertices adjacent to 0 (1 and 7) are updated. Min Heap contains all vertices except vertex 0.

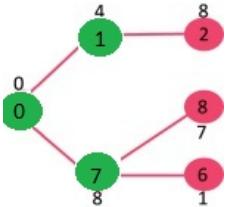
The vertices in green color are the vertices included in MST.



Since key value of vertex 1 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 1 are updated (Key is updated if the a vertex is not in Min Heap and previous key value is greater than the weight of edge from 1 to the adjacent). Min Heap contains all vertices except vertex 0 and 1.

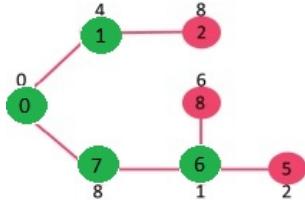


Since key value of vertex 7 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 7 are updated (Key is updated if the a vertex is not in Min Heap and previous key value is greater than the weight of edge from 7 to the adjacent). Min Heap contains all vertices except vertex 0, 1 and 7.

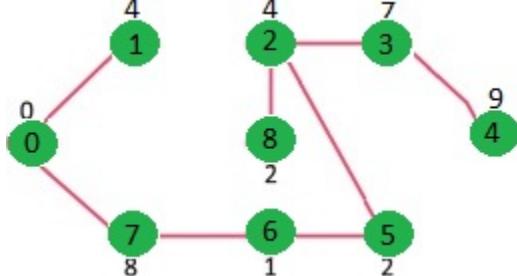


Since key value of vertex 6 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 6 are updated (Key is updated if the a

vertex is not in Min Heap and previous key value is greater than the weight of edge from 6 to the adjacent). Min Heap contains all vertices except vertex 0, 1, 7 and 6.



The above steps are repeated for rest of the nodes in Min Heap till Min Heap becomes empty



C++

```
// C / C++ program for Prim's MST for adjacency list representation of graph

#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

// A structure to represent a node in adjacency list
struct AdjListNode {
    int dest;
    int weight;
    struct AdjListNode* next;
};

// A structure to represent an adjacency list
struct AdjList {
    struct AdjListNode* head; // pointer to head node of list
};

// A structure to represent a graph. A graph is an array of adjacency lists.
// Size of array will be V (number of vertices in graph)
struct Graph {
    int V;
    struct AdjList* array;
};

// A utility function to create a new adjacency list node
struct AdjListNode* newAdjListNode(int dest, int weight)
```

```
{  
    struct AdjListNode* newNode = (struct AdjListNode*)malloc(sizeof(struct AdjListNode));  
    newNode->dest = dest;  
    newNode->weight = weight;  
    newNode->next = NULL;  
    return newNode;  
}  
  
// A utility function that creates a graph of V vertices  
struct Graph* createGraph(int V)  
{  
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));  
    graph->V = V;  
  
    // Create an array of adjacency lists. Size of array will be V  
    graph->array = (struct AdjList*)malloc(V * sizeof(struct AdjList));  
  
    // Initialize each adjacency list as empty by making head as NULL  
    for (int i = 0; i < V; ++i)  
        graph->array[i].head = NULL;  
  
    return graph;  
}  
  
// Adds an edge to an undirected graph  
void addEdge(struct Graph* graph, int src, int dest, int weight)  
{  
    // Add an edge from src to dest. A new node is added to the adjacency  
    // list of src. The node is added at the begining  
    struct AdjListNode* newNode = newAdjListNode(dest, weight);  
    newNode->next = graph->array[src].head;  
    graph->array[src].head = newNode;  
  
    // Since graph is undirected, add an edge from dest to src also  
    newNode = newAdjListNode(src, weight);  
    newNode->next = graph->array[dest].head;  
    graph->array[dest].head = newNode;  
}  
  
// Structure to represent a min heap node  
struct MinHeapNode {  
    int v;  
    int key;  
};  
  
// Structure to represent a min heap  
struct MinHeap {  
    int size; // Number of heap nodes present currently
```

```

int capacity; // Capacity of min heap
int* pos; // This is needed for decreaseKey()
struct MinHeapNode** array;
};

// A utility function to create a new Min Heap Node
struct MinHeapNode* newMinHeapNode(int v, int key)
{
    struct MinHeapNode* minHeapNode = (struct MinHeapNode*)malloc(sizeof(struct MinHeapNode));
    minHeapNode->v = v;
    minHeapNode->key = key;
    return minHeapNode;
}

// A utilit function to create a Min Heap
struct MinHeap* createMinHeap(int capacity)
{
    struct MinHeap* minHeap = (struct MinHeap*)malloc(sizeof(struct MinHeap));
    minHeap->pos = (int*)malloc(capacity * sizeof(int));
    minHeap->size = 0;
    minHeap->capacity = capacity;
    minHeap->array = (struct MinHeapNode**)malloc(capacity * sizeof(struct MinHeapNode*));
    return minHeap;
}

// A utility function to swap two nodes of min heap. Needed for min heapify
void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b)
{
    struct MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}

// A standard function to heapify at given idx
// This function also updates position of nodes when they are swapped.
// Position is needed for decreaseKey()
void minHeapify(struct MinHeap* minHeap, int idx)
{
    int smallest, left, right;
    smallest = idx;
    left = 2 * idx + 1;
    right = 2 * idx + 2;

    if (left < minHeap->size && minHeap->array[left]->key < minHeap->array[smallest]->key)
        smallest = left;

    if (right < minHeap->size && minHeap->array[right]->key < minHeap->array[smallest]->key)
        smallest = right;
}

```

```
if (smallest != idx) {
    // The nodes to be swapped in min heap
    MinHeapNode* smallestNode = minHeap->array[smallest];
    MinHeapNode* idxNode = minHeap->array[idx];

    // Swap positions
    minHeap->pos[smallestNode->v] = idx;
    minHeap->pos[idxNode->v] = smallest;

    // Swap nodes
    swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

    minHeapify(minHeap, smallest);
}
}

// A utility function to check if the given minHeap is ampty or not
int isEmpty(struct MinHeap* minHeap)
{
    return minHeap->size == 0;
}

// Standard function to extract minimum node from heap
struct MinHeapNode* extractMin(struct MinHeap* minHeap)
{
    if (isEmpty(minHeap))
        return NULL;

    // Store the root node
    struct MinHeapNode* root = minHeap->array[0];

    // Replace root node with last node
    struct MinHeapNode* lastNode = minHeap->array[minHeap->size - 1];
    minHeap->array[0] = lastNode;

    // Update position of last node
    minHeap->pos[root->v] = minHeap->size - 1;
    minHeap->pos[lastNode->v] = 0;

    // Reduce heap size and heapify root
    --minHeap->size;
    minHeapify(minHeap, 0);

    return root;
}

// Function to decreasy key value of a given vertex v. This function
```

```

// uses pos[] of min heap to get the current index of node in min heap
void decreaseKey(struct MinHeap* minHeap, int v, int key)
{
    // Get the index of v in heap array
    int i = minHeap->pos[v];

    // Get the node and update its key value
    minHeap->array[i]->key = key;

    // Travel up while the complete tree is not hepified.
    // This is a O(Logn) loop
    while (i && minHeap->array[i]->key < minHeap->array[(i - 1) / 2]->key) {
        // Swap this node with its parent
        minHeap->pos[minHeap->array[i]->v] = (i - 1) / 2;
        minHeap->pos[minHeap->array[(i - 1) / 2]->v] = i;
        swapMinHeapNode(&minHeap->array[i], &minHeap->array[(i - 1) / 2]);

        // move to parent index
        i = (i - 1) / 2;
    }
}

// A utility function to check if a given vertex
// 'v' is in min heap or not
bool isInMinHeap(struct MinHeap* minHeap, int v)
{
    if (minHeap->pos[v] < minHeap->size)
        return true;
    return false;
}

// A utility function used to print the constructed MST
void printArr(int arr[], int n)
{
    for (int i = 1; i < n; ++i)
        printf("%d - %d\n", arr[i], i);
}

// The main function that constructs Minimum Spanning Tree (MST)
// using Prim's algorithm
void PrimMST(struct Graph* graph)
{
    int V = graph->V; // Get the number of vertices in graph
    int parent[V]; // Array to store constructed MST
    int key[V]; // Key values used to pick minimum weight edge in cut

    // minHeap represents set E
    struct MinHeap* minHeap = createMinHeap(V);

```

```
// Initialize min heap with all vertices. Key value of
// all vertices (except 0th vertex) is initially infinite
for (int v = 1; v < V; ++v) {
    parent[v] = -1;
    key[v] = INT_MAX;
    minHeap->array[v] = newMinHeapNode(v, key[v]);
    minHeap->pos[v] = v;
}

// Make key value of 0th vertex as 0 so that it
// is extracted first
key[0] = 0;
minHeap->array[0] = newMinHeapNode(0, key[0]);
minHeap->pos[0] = 0;

// Initially size of min heap is equal to V
minHeap->size = V;

// In the followin loop, min heap contains all nodes
// not yet added to MST.
while (!isEmpty(minHeap)) {
    // Extract the vertex with minimum key value
    struct MinHeapNode* minHeapNode = extractMin(minHeap);
    int u = minHeapNode->v; // Store the extracted vertex number

    // Traverse through all adjacent vertices of u (the extracted
    // vertex) and update their key values
    struct AdjListNode* pCrawl = graph->array[u].head;
    while (pCrawl != NULL) {
        int v = pCrawl->dest;

        // If v is not yet included in MST and weight of u-v is
        // less than key value of v, then update key value and
        // parent of v
        if (isInMinHeap(minHeap, v) && pCrawl->weight < key[v]) {
            key[v] = pCrawl->weight;
            parent[v] = u;
            decreaseKey(minHeap, v, key[v]);
        }
        pCrawl = pCrawl->next;
    }
}

// print edges of MST
printArr(parent, V);
}
```

```
// Driver program to test above functions
int main()
{
    // Let us create the graph given in above fugure
    int V = 9;
    struct Graph* graph = createGraph(V);
    addEdge(graph, 0, 1, 4);
    addEdge(graph, 0, 7, 8);
    addEdge(graph, 1, 2, 8);
    addEdge(graph, 1, 7, 11);
    addEdge(graph, 2, 3, 7);
    addEdge(graph, 2, 8, 2);
    addEdge(graph, 2, 5, 4);
    addEdge(graph, 3, 4, 9);
    addEdge(graph, 3, 5, 14);
    addEdge(graph, 4, 5, 10);
    addEdge(graph, 5, 6, 2);
    addEdge(graph, 6, 7, 1);
    addEdge(graph, 6, 8, 6);
    addEdge(graph, 7, 8, 7);

    PrimMST(graph);

    return 0;
}
```

Java

```
// Java program for Prim's MST for
// adjacency list representation of graph
import java.util.LinkedList;
import java.util.PriorityQueue;
import java.util.Comparator;

public class prims {
    class node1 {

        // Stores destination vertex in adjacency list
        int dest;

        // Stores weight of a vertex in adjacency list
        int weight;

        // Constructor
        node1(int a, int b)
        {
            dest = a;
            weight = b;
        }
    }
}
```

```
        }
    }
static class Graph {

    // Number of vertices in the graph
    int V;

    // List of adjacent nodes of a given vertex
    LinkedList<node1>[] adj;

    // Constructor
    Graph(int e)
    {
        V = e;
        adj = new LinkedList[V];
        for (int o = 0; o < V; o++)
            adj[o] = new LinkedList<>();
    }
}

// class to represent a node in PriorityQueue
// Stores a vertex and its corresponding
// key value
class node {
    int vertex;
    int key;
}

// Comparator class created for PriorityQueue
// returns 1 if node0.key > node1.key
// returns 0 if node0.key < node1.key and
// returns -1 otherwise
class comparator implements Comparator<node> {

    @Override
    public int compare(node node0, node node1)
    {
        return node0.key - node1.key;
    }
}

// method to add an edge
// between two vertices
void addEdge(Graph graph, int src, int dest, int weight)
{
    node1 node0 = new node1(dest, weight);
    node1 node = new node1(src, weight);
```

```
graph.adj[src].addLast(node0);
graph.adj[dest].addLast(node);
}

// method used to find the mst
void prims_mst(Graph graph)
{

    // Whether a vertex is in PriorityQueue or not
    Boolean[] mstset = new Boolean[graph.V];
    node[] e = new node[graph.V];

    // Stores the parents of a vertex
    int[] parent = new int[graph.V];

    for (int o = 0; o < graph.V; o++)
        e[o] = new node();

    for (int o = 0; o < graph.V; o++) {

        // Initialize to false
        mstset[o] = false;

        // Initialize key values to infinity
        e[o].key = Integer.MAX_VALUE;
        e[o].vertex = o;
        parent[o] = -1;
    }

    // Include the source vertex in mstset
    mstset[0] = true;

    // Set key value to 0
    // so that it is extracted first
    // out of PriorityQueue
    e[0].key = 0;

    // PriorityQueue
    PriorityQueue<node> queue = new PriorityQueue<>(graph.V, new comparator());

    for (int o = 0; o < graph.V; o++)
        queue.add(e[o]);

    // Loops until the PriorityQueue is not empty
    while (!queue.isEmpty()) {

        // Extracts a node with min key value
        node node0 = queue.poll();
    }
}
```

```
// Include that node into mstset
mstset[node0.vertex] = true;

// For all adjacent vertex of the extracted vertex V
for (node1 iterator : graph.adj[node0.vertex]) {

    // If V is in PriorityQueue
    if (mstset[iterator.dest] == false) {
        // If the key value of the adjacent vertex is
        // more than the extracted key
        // update the key value of adjacent vertex
        // to update first remove and add the updated vertex
        if (e[iterator.dest].key > iterator.weight) {
            queue.remove(e[iterator.dest]);
            e[iterator.dest].key = iterator.weight;
            queue.add(e[iterator.dest]);
            parent[iterator.dest] = node0.vertex;
        }
    }
}

// Prints the vertex pair of mst
for (int o = 1; o < graph.V; o++)
    System.out.println(parent[o] + " "
                       + "-"
                       + " " + o);
}

public static void main(String[] args)
{
    int V = 9;

    Graph graph = new Graph(V);

    prims e = new prims();

    e.addEdge(graph, 0, 1, 4);
    e.addEdge(graph, 0, 7, 8);
    e.addEdge(graph, 1, 2, 8);
    e.addEdge(graph, 1, 7, 11);
    e.addEdge(graph, 2, 3, 7);
    e.addEdge(graph, 2, 8, 2);
    e.addEdge(graph, 2, 5, 4);
    e.addEdge(graph, 3, 4, 9);
    e.addEdge(graph, 3, 5, 14);
    e.addEdge(graph, 4, 5, 10);
```

```
e.addEdge(graph, 5, 6, 2);
e.addEdge(graph, 6, 7, 1);
e.addEdge(graph, 6, 8, 6);
e.addEdge(graph, 7, 8, 7);

// Method invoked
e.prims_mst(graph);
}

}

// This code is contributed by Vikash Kumar Dubey
```

Python

```
# A Python program for Prims's MST for
# adjacency list representation of graph

from collections import defaultdict
import sys

class Heap():

    def __init__(self):
        self.array = []
        self.size = 0
        self.pos = []

    def newMinHeapNode(self, v, dist):
        minHeapNode = [v, dist]
        return minHeapNode

    # A utility function to swap two nodes of
    # min heap. Needed for min heapify
    def swapMinHeapNode(self, a, b):
        t = self.array[a]
        self.array[a] = self.array[b]
        self.array[b] = t

    # A standard function to heapify at given idx
    # This function also updates position of nodes
    # when they are swapped. Position is needed
    # for decreaseKey()
    def minHeapify(self, idx):
        smallest = idx
        left = 2 * idx + 1
        right = 2 * idx + 2

        if left < self.size and self.array[left][1] < \
           self.array[smallest][1]:
```

```
smallest = left

if right < self.size and self.array[right][1] < \
    self.array[smallest][1]:
    smallest = right

# The nodes to be swapped in min heap
# if idx is not smallest
if smallest != idx:

    # Swap positions
    self.pos[ self.array[smallest][0] ] = idx
    self.pos[ self.array[idx][0] ] = smallest

    # Swap nodes
    self.swapMinHeapNode(smallest, idx)

    self.minHeapify(smallest)

# Standard function to extract minimum node from heap
def extractMin(self):

    # Return NULL wif heap is empty
    if self.isEmpty() == True:
        return

    # Store the root node
    root = self.array[0]

    # Replace root node with last node
    lastNode = self.array[self.size - 1]
    self.array[0] = lastNode

    # Update position of last node
    self.pos[lastNode[0]] = 0
    self.pos[root[0]] = self.size - 1

    # Reduce heap size and heapify root
    self.size -= 1
    self.minHeapify(0)

    return root

def isEmpty(self):
    return True if self.size == 0 else False

def decreaseKey(self, v, dist):
```

```
# Get the index of v in heap array
i = self.pos[v]

# Get the node and update its dist value
self.array[i][1] = dist

# Travel up while the complete tree is not
# hepified. This is a O(Logn) loop
while i > 0 and self.array[i][1] < \
        self.array[(i - 1) / 2][1]:

    # Swap this node with its parent
    self.pos[ self.array[i][0] ] = (i-1)/2
    self.pos[ self.array[(i-1)/2][0] ] = i
    self.swapMinHeapNode(i, (i - 1)/2)

    # move to parent index
    i = (i - 1) / 2;

# A utility function to check if a given vertex
# 'v' is in min heap or not
def isInMinHeap(self, v):

    if self.pos[v] < self.size:
        return True
    return False

def printArr(parent, n):
    for i in range(1, n):
        print "% d - % d" % (parent[i], i)

class Graph():

    def __init__(self, V):
        self.V = V
        self.graph = defaultdict(list)

    # Adds an edge to an undirected graph
    def addEdge(self, src, dest, weight):

        # Add an edge from src to dest. A new node is
        # added to the adjacency list of src. The node
        # is added at the begining. The first element of
        # the node has the destination and the second
        # elements has the weight
```

```
newNode = [dest, weight]
self.graph[src].insert(0, newNode)

# Since graph is undirected, add an edge from
# dest to src also
newNode = [src, weight]
self.graph[dest].insert(0, newNode)

# The main function that prints the Minimum
# Spanning Tree(MST) using the Prim's Algorithm.
# It is a O(ELogV) function
def PrimMST(self):
    # Get the number of vertices in graph
    V = self.V

    # key values used to pick minimum weight edge in cut
    key = []

    # List to store constructed MST
    parent = []

    # minHeap represents set E
    minHeap = Heap()

    # Initialize min heap with all vertices. Key values of all
    # vertices (except the 0th vertex) is initially infinite
    for v in range(V):
        parent.append(-1)
        key.append(sys.maxint)
        minHeap.array.append(minHeap.newMinHeapNode(v, key[v]))
        minHeap.pos.append(v)

    # Make key value of 0th vertex as 0 so
    # that it is extracted first
    minHeap.pos[0] = 0
    key[0] = 0
    minHeap.decreaseKey(0, key[0])

    # Initially size of min heap is equal to V
    minHeap.size = V;

    # In the following loop, min heap contains all nodes
    # not yet added in the MST.
    while minHeap.isEmpty() == False:

        # Extract the vertex with minimum distance value
        newHeapNode = minHeap.extractMin()
        u = newHeapNode[0]
```

```
# Traverse through all adjacent vertices of u
# (the extracted vertex) and update their
# distance values
for pCrawl in self.graph[u]:
    v = pCrawl[0]

    # If shortest distance to v is not finalized
    # yet, and distance to v through u is less than
    # its previously calculated distance
    if minHeap.isInMinHeap(v) and pCrawl[1] < key[v]:
        key[v] = pCrawl[1]
        parent[v] = u

        # update distance value in min heap also
        minHeap.decreaseKey(v, key[v])

printArr(parent, V)

# Driver program to test the above functions
graph = Graph(9)
graph.addEdge(0, 1, 4)
graph.addEdge(0, 7, 8)
graph.addEdge(1, 2, 8)
graph.addEdge(1, 7, 11)
graph.addEdge(2, 3, 7)
graph.addEdge(2, 8, 2)
graph.addEdge(2, 5, 4)
graph.addEdge(3, 4, 9)
graph.addEdge(3, 5, 14)
graph.addEdge(4, 5, 10)
graph.addEdge(5, 6, 2)
graph.addEdge(6, 7, 1)
graph.addEdge(6, 8, 6)
graph.addEdge(7, 8, 7)
graph.PrimMST()

# This code is contributed by Divyanshu Mehta
```

Output:

```
0 - 1
5 - 2
2 - 3
3 - 4
6 - 5
```

7 - 6
0 - 7
2 - 8

Time Complexity: The time complexity of the above code/algorithm looks $O(V^2)$ as there are two nested while loops. If we take a closer look, we can observe that the statements in inner loop are executed $O(V+E)$ times (similar to BFS). The inner loop has decreaseKey() operation which takes $O(\log V)$ time. So overall time complexity is $O(E+V)*O(\log V)$ which is $O((E+V)*\log V) = O(E\log V)$ (For a connected graph, $V = O(E)$)

References:

Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.

http://en.wikipedia.org/wiki/Prim's_algorithm

This article is compiled by [Aashish Barnwal](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Improved By : [VikashDubey](#), [Sumon Nath](#)

Source

<https://www.geeksforgeeks.org/prims-mst-for-adjacency-list-representation-greedy-algo-6/>

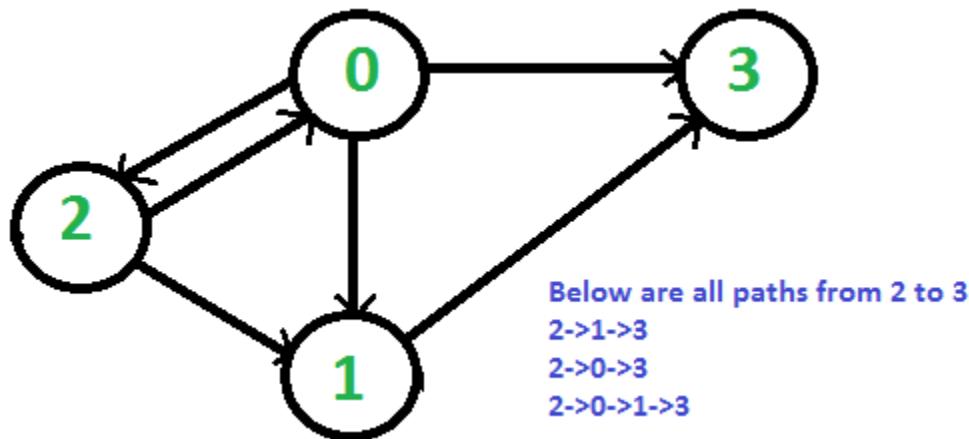
Chapter 239

Print all paths from a given source to a destination using BFS

Print all paths from a given source to a destination using BFS - GeeksforGeeks

Given a directed graph, a source vertex 'src' and a destination vertex 'dst', print all paths from given 'src' to 'dst'.

Consider the following directed graph. Let the src be 2 and dst be 3. There are 3 different paths from 2 to 3.



We have already discussed [Print all paths from a given source to a destination using DFS](#).

Below is BFS based solution.

Algorithm :

create a queue which will store path(s) of type vector
initialise the queue with first path starting from src

Now run a loop till queue is not empty
 get the frontmost path from queue
 check if the lastnode of this path is destination
 if true then print the path
 run a loop for all the vertices connected to the
 current vertex i.e. lastnode extracted from path
 if the vertex is not visited in current path
 a) create a new path from earlier path and
 append this vertex
 b) insert this new path to queue

```
// CPP program to print all paths of source to
// destination in given graph
#include <bits/stdc++.h>
using namespace std;

// utility function for printing
// the found path in graph
void printpath(vector<int>& path)
{
    int size = path.size();
    for (int i = 0; i < size; i++)
        cout << path[i] << " ";
    cout << endl;
}

// utility function to check if current
// vertex is already present in path
int isNotVisited(int x, vector<int>& path)
{
    int size = path.size();
    for (int i = 0; i < size; i++)
        if (path[i] == x)
            return 0;
    return 1;
}

// utility function for finding paths in graph
// from source to destination
void findpaths(vector<vector<int> >&g, int src,
              int dst, int v)
{
    // create a queue which stores
    // the paths
    queue<vector<int> > q;
```

```
// path vector to store the current path
vector<int> path;
path.push_back(src);
q.push(path);
while (!q.empty()) {
    path = q.front();
    q.pop();
    int last = path[path.size() - 1];

    // if last vertex is the desired destination
    // then print the path
    if (last == dst)
        printpath(path);

    // traverse to all the nodes connected to
    // current vertex and push new path to queue
    for (int i = 0; i < g[last].size(); i++) {
        if (isNotVisited(g[last][i], path)) {
            vector<int> newpath(path);
            newpath.push_back(g[last][i]);
            q.push(newpath);
        }
    }
}

// driver program
int main()
{
    vector<vector<int> > g;
    // number of vertices
    int v = 4;
    g.resize(4);

    // construct a graph
    g[0].push_back(3);
    g[0].push_back(1);
    g[0].push_back(2);
    g[1].push_back(3);
    g[2].push_back(0);
    g[2].push_back(1);

    int src = 2, dst = 3;
    cout << "path from src " << src
        << " to dst " << dst << " are \n";

    // function for finding the paths
```

```
    findpaths(g, src, dst, v);  
  
    return 0;  
}
```

Output:

```
path from src 2 to dst 3 are  
2 0 3  
2 1 3  
2 0 1 3
```

Source

<https://www.geeksforgeeks.org/print-paths-given-source-destination-using-bfs/>

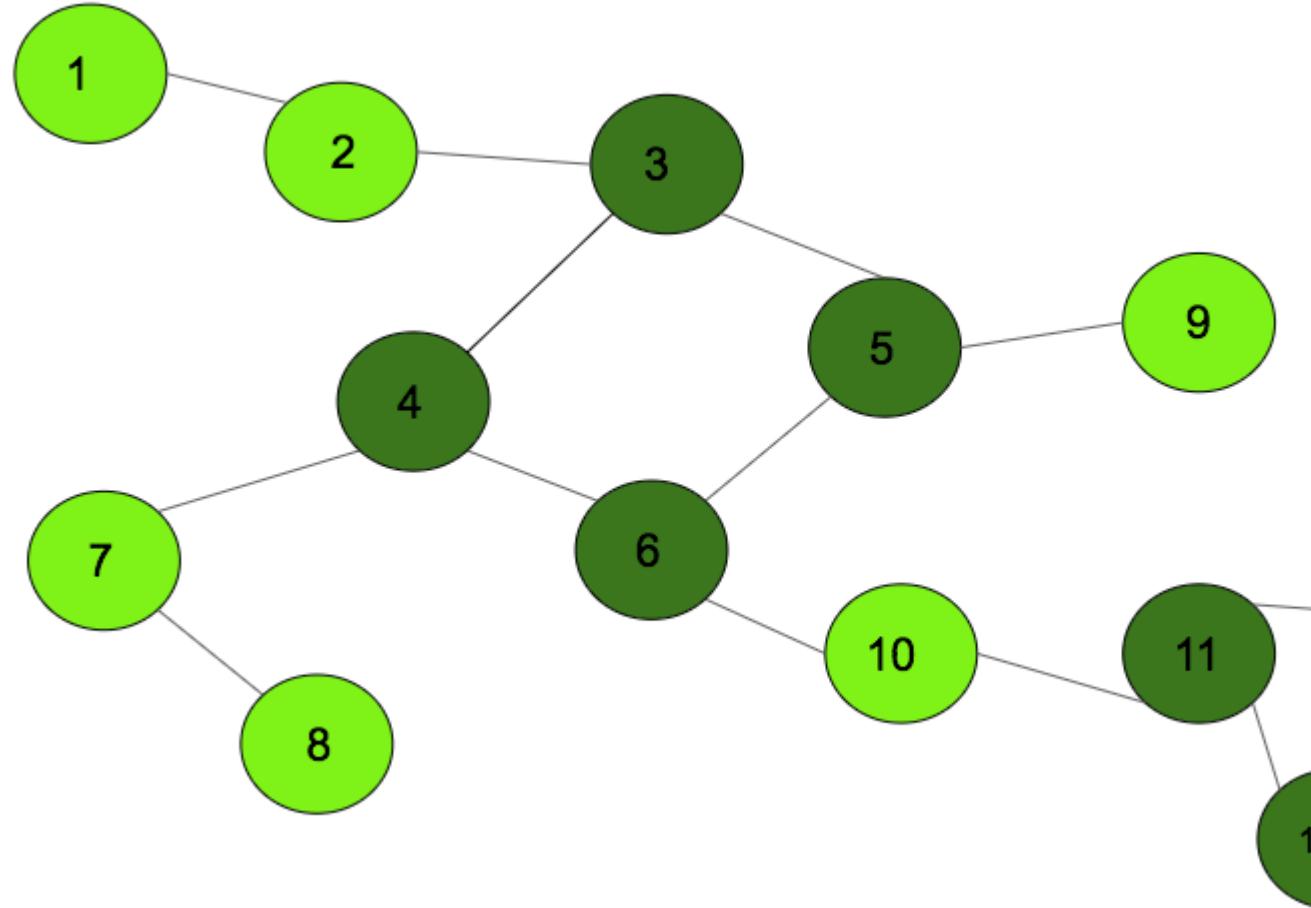
Chapter 240

Print all the cycles in an undirected graph

Print all the cycles in an undirected graph - GeeksforGeeks

Given an undirected graph, print all the vertices that form cycles in it.

Pre-requisite: [Detect Cycle in a directed graph using colors](#)



In the above diagram, the cycles have been marked with dark green color. The output for the above will be

1st cycle: 3 5 4 6

2nd cycle: 11 12 13

Approach: Using the [graph coloring method](#), mark all the vertex of the different cycles with unique numbers. Once the graph traversal is completed, push all the similar marked numbers to an adjacency list and print the adjacency list accordingly. Given below is the algorithm:

- Insert the edges into an adjacency list.
- Call the DFS function which uses the coloring method to mark the vertex.
- Whenever there is a partially visited vertex, **backtrack** till the current vertex is reached and mark all of them with cycle numbers. Once all the vertexes are marked, increase the cycle number.

- Once Dfs is completed, iterate for the edges and push the same marked number edges to another adjacency list.
- Iterate in the another adjacency list and print the vertex cycle-number wise.

Below is the implementation of the above approach:

```
// C++ program to print all the cycles
// in an undirected graph
#include <bits/stdc++.h>
using namespace std;
const int N = 100000;

// variables to be used
// in both functions
vector<int> graph[N];
vector<int> cycles[N];

// Function to mark the vertex with
// different colors for different cycles
void dfs_cycle(int u, int p, int color[],
               int mark[], int par[], int& cyclenumber)
{

    // already (completely) visited vertex.
    if (color[u] == 2) {
        return;
    }

    // seen vertex, but was not completely visited -> cycle detected.
    // backtrack based on parents to find the complete cycle.
    if (color[u] == 1) {

        cyclenumber++;
        int cur = p;
        mark[cur] = cyclenumber;

        // backtrack the vertex which are
        // in the current cycle thaths found
        while (cur != u) {
            cur = par[cur];
            mark[cur] = cyclenumber;
        }
        return;
    }
    par[u] = p;

    // partially visisted.
```

```
color[u] = 1;

// simple dfs on graph
for (int v : graph[u]) {

    // if it has not been visited previously
    if (v == par[u]) {
        continue;
    }
    dfs_cycle(v, u, color, mark, par, cyclenumber);
}

// completely visited.
color[u] = 2;
}

// add the edges to the graph
void addEdge(int u, int v)
{
    graph[u].push_back(v);
    graph[v].push_back(u);
}

// Function to print the cycles
void printCycles(int edges, int mark[], int& cyclenumber)
{

    // push the edges that into the
    // cycle adjacency list
    for (int i = 1; i <= edges; i++) {
        if (mark[i] != 0)
            cycles[mark[i]].push_back(i);
    }

    // print all the vertex with same cycle
    for (int i = 1; i <= cyclenumber; i++) {
        // Print the i-th cycle
        cout << "Cycle Number " << i << ": ";
        for (int x : cycles[i])
            cout << x << " ";
        cout << endl;
    }
}

// Driver Code
int main()
{
```

```
// add edges
addEdge(1, 2);
addEdge(2, 3);
addEdge(3, 4);
addEdge(4, 6);
addEdge(4, 7);
addEdge(5, 6);
addEdge(3, 5);
addEdge(7, 8);
addEdge(6, 10);
addEdge(5, 9);
addEdge(10, 11);
addEdge(11, 12);
addEdge(11, 13);
addEdge(12, 13);

// arrays required to color the
// graph, store the parent of node
int color[N];
int par[N];

// mark with unique numbers
int mark[N];

// store the numbers of cycle
int cyclenumber = 0;
int edges = 13;

// call DFS to mark the cycles
dfs_cycle(1, 0, color, mark, par, cyclenumber);

// function to print the cycles
printCycles(edges, mark, cyclenumber);
}
```

Output:

```
Cycle Number 1: 3 4 5 6
Cycle Number 2: 11 12 13
```

Time Complexity: $O(N + M)$, where N is number of vertex and M is the number of edges.

Auxiliary Space: $O(N + M)$

Source

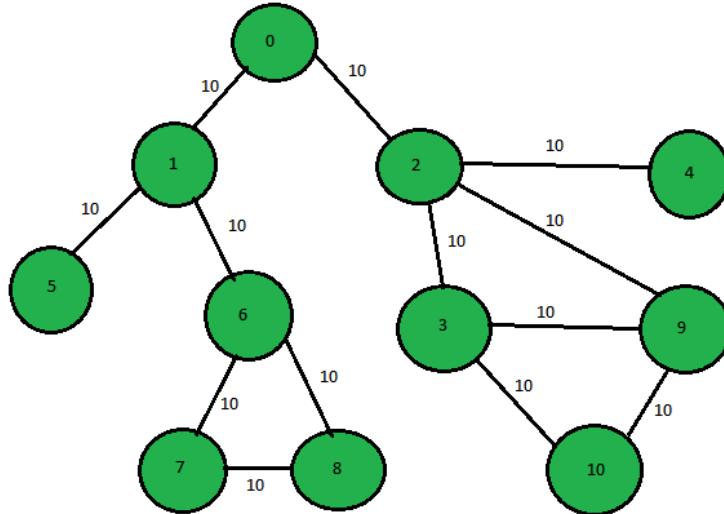
<https://www.geeksforgeeks.org/print-all-the-cycles-in-an-undirected-graph/>

Chapter 241

Print the DFS traversal step-wise (Backtracking also)

Print the DFS traversal step-wise (Backtracking also) - GeeksforGeeks

Given a [graph](#), the task is to print the DFS traversal of a graph which includes the every step including the backtracking.

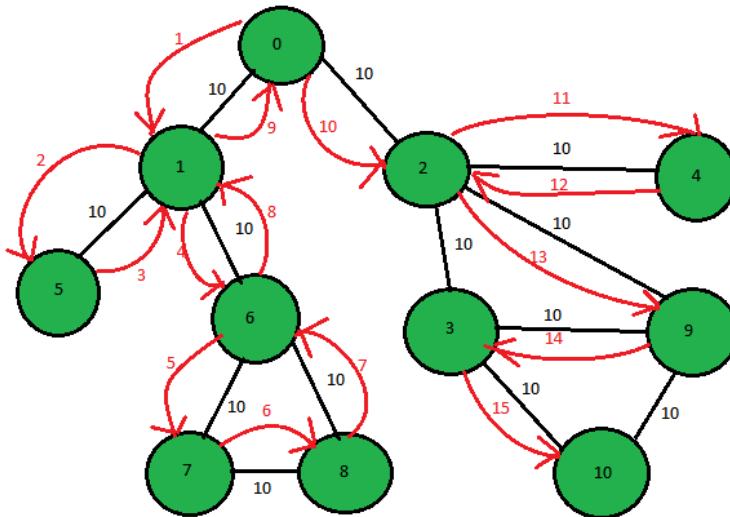


1st step:- 0 → 1
2nd step:- 1 → 5
3rd step:- 5 → 1 (backtracking step)
4th step:- 1 → 6...
and so on till all the nodes are visited.

Dfs step-wise(including backtracking) is:

0 1 5 1 6 7 8 7 6 1 0 2 4 2 9 3 10

Note: In this above diagram the weight between the edges has just been added, it does not have any role in DFS-traversal



Approach: **DFS with Backtracking** will be used here. First, visit every node using DFS simultaneously and keep track of the previously used edge and the parent node. If a node comes whose all the adjacent node has been visited, backtrack using the last used edge and print the nodes. Continue the steps and at every step, the parent node will become the present node. Continue the above steps to find the complete DFS traversal of the graph.

Below is the implementation of the above approach:

```
// C++ program to print the complete
// DFS-traversal of graph
// using back-tracking
#include <bits/stdc++.h>
using namespace std;
const int N = 1000;
vector<int> adj[N];

// Function to print the complete DFS-traversal
void dfsUtil(int u, int node, bool visited[],
             vector<pair<int, int> > road_used, int parent, int it)
{
    int c = 0;
```

```
// Check if all th node is visited or not
// and count unvisited nodes
for (int i = 0; i < node; i++)
    if (visited[i])
        c++;

// If all the node is visited return;
if (c == node)
    return;

// Mark not visited node as visited
visited[u] = true;

// Track the current edge
road_used.push_back({ parent, u });

// Print the node
cout << u << " ";

// Check for not visited node and proceed with it.
for (int x : adj[u]) {

    // call the DFs function if not visited
    if (!visited[x])
        dfsUtil(x, node, visited, road_used, u, it + 1);
}

// Backtrack through the last
// visited nodes
for (auto y : road_used)
    if (y.second == u)
        dfsUtil(y.first, node, visited,
                road_used, u, it + 1);
}

// Function to call the DFS function
// which prints the DFS-travesal stepwise
void dfs(int node)
{

    // Create a array of visited ndoe
    bool visited[node];

    // Vector to track last visited road
    vector<pair<int, int> > road_used;

    // Initialize all the node with false
    for (int i = 0; i < node; i++)
```

```
visited[i] = false;

// call the function
dfsUtil(0, node, visited, road_used, -1, 0);
}

// Function to insert edges in Graph
void insertEdge(int u, int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}

// Driver Code
int main()
{
    // number of nodes and edges in the graph
    int node = 11, edge = 13;

    // Function call to create the graph
    insertEdge(0, 1);
    insertEdge(0, 2);
    insertEdge(1, 5);
    insertEdge(1, 6);
    insertEdge(2, 4);
    insertEdge(2, 9);
    insertEdge(6, 7);
    insertEdge(6, 8);
    insertEdge(7, 8);
    insertEdge(2, 3);
    insertEdge(3, 9);
    insertEdge(3, 10);
    insertEdge(9, 10);

    // Call the function to print
    dfs(node);

    return 0;
}
```

Output:

```
0 1 5 1 6 7 8 7 6 1 0 2 4 2 9 3 10
```

Source

<https://www.geeksforgeeks.org/print-the-dfs-traversal-step-wise-backtracking-also/>

Chapter 242

Printing Paths in Dijkstra's Shortest Path Algorithm

Printing Paths in Dijkstra's Shortest Path Algorithm - GeeksforGeeks

Given a graph and a source vertex in graph, find shortest paths from source to all vertices in the given graph.

We have discussed Dijkstra's Shortest Path algorithm in below posts.

- Dijkstra's shortest path for adjacency matrix representation
- Dijkstra's shortest path for adjacency list representation

The implementations discussed above only find shortest distances, but do not print paths. In this post printing of paths is discussed.

For example, consider below graph and source as 0,

Output should be		
Vertex	Distance	Path
0 → 1	4	0 1
0 → 2	12	0 1 2
0 → 3	19	0 1 2 3
0 → 4	21	0 7 6 5 4
0 → 5	11	0 7 6 5
0 → 6	9	0 7 6
0 → 7	8	0 7
0 → 8	14	0 1 2 8

The idea is to create a separate array `parent[]`. Value of `parent[v]` for a vertex v stores parent vertex of v in shortest path tree. Parent of root (or source vertex) is -1. Whenever we find shorter path through a vertex u, we make u as parent of current vertex.

Once we have parent array constructed, we can print path using below recursive function.

```
void printPath(int parent[], int j)
{
    // Base Case : If j is source
    if (parent[j]==-1)
        return;

    printPath(parent, parent[j]);
    printf("%d ", j);
}
```

Below is the complete implementation
C/C++

```
// C program for Dijkstra's single
// source shortest path algorithm.
// The program is for adjacency matrix
// representation of the graph.
#include <stdio.h>
#include <limits.h>

// Number of vertices
// in the graph
#define V 9

// A utility function to find the
// vertex with minimum distance
// value, from the set of vertices
// not yet included in shortest
// path tree
int minDistance(int dist[],
                bool sptSet[])
{

    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (sptSet[v] == false &&
            dist[v] <= min)
            min = dist[v], min_index = v;
```

```
        return min_index;
    }

// Function to print shortest
// path from source to j
// using parent array
void printPath(int parent[], int j)
{

    // Base Case : If j is source
    if (parent[j] == -1)
        return;

    printPath(parent, parent[j]);
    printf("%d ", j);
}

// A utility function to print
// the constructed distance
// array
int printSolution(int dist[], int n,
                  int parent[])
{
    int src = 0;
    printf("Vertex\t Distance\tPath");
    for (int i = 1; i < V; i++)
    {
        printf("\n%d -> %d \t\t %d\t\t %d ", src, i, dist[i], src);
        printPath(parent, i);
    }
}

// Function that implements Dijkstra's
// single source shortest path
// algorithm for a graph represented
// using adjacency matrix representation
void dijkstra(int graph[V][V], int src)
{
    // The output array. dist[i]
    // will hold the shortest
    // distance from src to i
    int dist[V];

    // sptSet[i] will true if vertex
```

```
// i is included / in shortest
// path tree or shortest distance
// from src to i is finalized
bool sptSet[V];

// Parent array to store
// shortest path tree
int parent[V];

// Initialize all distances as
// INFINITE and sptSet[] as false
for (int i = 0; i < V; i++)
{
    parent[0] = -1;
    dist[i] = INT_MAX;
    sptSet[i] = false;
}

// Distance of source vertex
// from itself is always 0
dist[src] = 0;

// Find shortest path
// for all vertices
for (int count = 0; count < V - 1; count++)
{
    // Pick the minimum distance
    // vertex from the set of
    // vertices not yet processed.
    // u is always equal to src
    // in first iteration.
    int u = minDistance(dist, sptSet);

    // Mark the picked vertex
    // as processed
    sptSet[u] = true;

    // Update dist value of the
    // adjacent vertices of the
    // picked vertex.
    for (int v = 0; v < V; v++)

        // Update dist[v] only if is
        // not in sptSet, there is
        // an edge from u to v, and
        // total weight of path from
        // src to v through u is smaller
        // than current value of
```

```
// dist[v]
if (!sptSet[v] && graph[u][v] &&
    dist[u] + graph[u][v] < dist[v])
{
    parent[v] = u;
    dist[v] = dist[u] + graph[u][v];
}
}

// print the constructed
// distance array
printSolution(dist, V, parent);
}

// Driver Code
int main()
{
    // Let us create the example
    // graph discussed above
    int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},
                        {4, 0, 8, 0, 0, 0, 0, 11, 0},
                        {0, 8, 0, 7, 0, 4, 0, 0, 2},
                        {0, 0, 7, 0, 9, 14, 0, 0, 0},
                        {0, 0, 0, 9, 0, 10, 0, 0, 0},
                        {0, 0, 4, 0, 10, 0, 2, 0, 0},
                        {0, 0, 0, 14, 0, 2, 0, 1, 6},
                        {8, 11, 0, 0, 0, 0, 1, 0, 7},
                        {0, 0, 2, 0, 0, 0, 6, 7, 0}
                    };
    dijkstra(graph, 0);
    return 0;
}
```

Java

```
// A Java program for Dijkstra's
// single source shortest path
// algorithm. The program is for
// adjacency matrix representation
// of the graph.

class DijkstrasAlgorithm {

    private static final int NO_PARENT = -1;

    // Function that implements Dijkstra's
    // single source shortest path
```

```
// algorithm for a graph represented
// using adjacency matrix
// representation
private static void dijkstra(int[][] adjacencyMatrix,
                             int startVertex)
{
    int nVertices = adjacencyMatrix[0].length;

    // shortestDistances[i] will hold the
    // shortest distance from src to i
    int[] shortestDistances = new int[nVertices];

    // added[i] will true if vertex i is
    // included / in shortest path tree
    // or shortest distance from src to
    // i is finalized
    boolean[] added = new boolean[nVertices];

    // Initialize all distances as
    // INFINITE and added[] as false
    for (int vertexIndex = 0; vertexIndex < nVertices;
         vertexIndex++)
    {
        shortestDistances[vertexIndex] = Integer.MAX_VALUE;
        added[vertexIndex] = false;
    }

    // Distance of source vertex from
    // itself is always 0
    shortestDistances[startVertex] = 0;

    // Parent array to store shortest
    // path tree
    int[] parents = new int[nVertices];

    // The starting vertex does not
    // have a parent
    parents[startVertex] = NO_PARENT;

    // Find shortest path for all
    // vertices
    for (int i = 1; i < nVertices; i++)
    {

        // Pick the minimum distance vertex
        // from the set of vertices not yet
        // processed. nearestVertex is
        // always equal to startNode in
```

```

// first iteration.
int nearestVertex = -1;
int shortestDistance = Integer.MAX_VALUE;
for (int vertexIndex = 0;
     vertexIndex < nVertices;
     vertexIndex++)
{
    if (!added[vertexIndex] &&
        shortestDistances[vertexIndex] <
        shortestDistance)
    {
        nearestVertex = vertexIndex;
        shortestDistance = shortestDistances[vertexIndex];
    }
}

// Mark the picked vertex as
// processed
added[nearestVertex] = true;

// Update dist value of the
// adjacent vertices of the
// picked vertex.
for (int vertexIndex = 0;
     vertexIndex < nVertices;
     vertexIndex++)
{
    int edgeDistance = adjacencyMatrix[nearestVertex][vertexIndex];

    if (edgeDistance > 0
        && ((shortestDistance + edgeDistance) <
              shortestDistances[vertexIndex]))
    {
        parents[vertexIndex] = nearestVertex;
        shortestDistances[vertexIndex] = shortestDistance +
                                         edgeDistance;
    }
}

printSolution(startVertex, shortestDistances, parents);
}

// A utility function to print
// the constructed distances
// array and shortest paths
private static void printSolution(int startVertex,
                                  int[] distances,

```

```
int[] parents
{
    int nVertices = distances.length;
    System.out.print("Vertex\t Distance\tPath");

    for (int vertexIndex = 0;
         vertexIndex < nVertices;
         vertexIndex++)
    {
        if (vertexIndex != startVertex)
        {
            System.out.print("\n" + startVertex + " -> ");
            System.out.print(vertexIndex + " \t\t");
            System.out.print(distances[vertexIndex] + "\t\t");
            printPath(vertexIndex, parents);
        }
    }
}

// Function to print shortest path
// from source to currentVertex
// using parents array
private static void printPath(int currentVertex,
                             int[] parents)
{

    // Base case : Source node has
    // been processed
    if (currentVertex == NO_PARENT)
    {
        return;
    }
    printPath(parents[currentVertex], parents);
    System.out.print(currentVertex + " ");
}

// Driver Code
public static void main(String[] args)
{
    int[][] adjacencyMatrix = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                               { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                               { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                               { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                               { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                               { 0, 0, 4, 0, 10, 0, 2, 0, 0 },
                               { 0, 0, 0, 14, 0, 2, 0, 1, 6 },
                               { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                               { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };
}
```

```
        dijkstra(adjacencyMatrix, 0);
    }
}

// This code is contributed by Harikrishnan Rajan
```

Python

```
# Python program for Dijkstra's
# single source shortest
# path algorithm. The program
# is for adjacency matrix
# representation of the graph

from collections import defaultdict

#Class to represent a graph
class Graph:

    # A utility function to find the
    # vertex with minimum dist value, from
    # the set of vertices still in queue
    def minDistance(self,dist,queue):
        # Initialize min value and min_index as -1
        minimum = float("Inf")
        min_index = -1

        # from the dist array,pick one which
        # has min value and is till in queue
        for i in range(len(dist)):
            if dist[i] < minimum and i in queue:
                minimum = dist[i]
                min_index = i
        return min_index

    # Function to print shortest path
    # from source to j
    # using parent array
    def printPath(self, parent, j):

        #Base Case : If j is source
        if parent[j] == -1 :
            print j,
            return
        self.printPath(parent , parent[j])
        print j,
```



```

# Update dist value and parent
# index of the adjacent vertices of
# the picked vertex. Consider only
# those vertices which are still in
# queue
for i in range(col):
    '''Update dist[i] only if it is in queue, there is
    an edge from u to i, and total weight of path from
    src to i through u is smaller than current value of
    dist[i]'''
    if graph[u][i] and i in queue:
        if dist[u] + graph[u][i] < dist[i]:
            dist[i] = dist[u] + graph[u][i]
            parent[i] = u

# print the constructed distance array
self.printSolution(dist,parent)

g= Graph()

graph = [[0, 4, 0, 0, 0, 0, 0, 8, 0],
          [4, 0, 8, 0, 0, 0, 0, 11, 0],
          [0, 8, 0, 7, 0, 4, 0, 0, 2],
          [0, 0, 7, 0, 9, 14, 0, 0, 0],
          [0, 0, 0, 9, 0, 10, 0, 0, 0],
          [0, 0, 4, 14, 10, 0, 2, 0, 0],
          [0, 0, 0, 0, 0, 2, 0, 1, 6],
          [8, 11, 0, 0, 0, 0, 1, 0, 7],
          [0, 0, 2, 0, 0, 0, 6, 7, 0]
         ]

#print the solution
g.dijkstra(graph,0)

#This code is contributed by Neelam Yadav

```

Output:

Vertex	Distance	Path
0 -> 1	4	0 1
0 -> 2	12	0 1 2
0 -> 3	19	0 1 2 3
0 -> 4	21	0 7 6 5 4
0 -> 5	11	0 7 6 5

0 -> 6	9	0 7 6
0 -> 7	8	0 7
0 -> 8	14	0 1 2 8

This article is contributed by **Aditya Goel**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Improved By : [rhari](#)

Source

<https://www.geeksforgeeks.org/printing-paths-dijkstras-shortest-path-algorithm/>

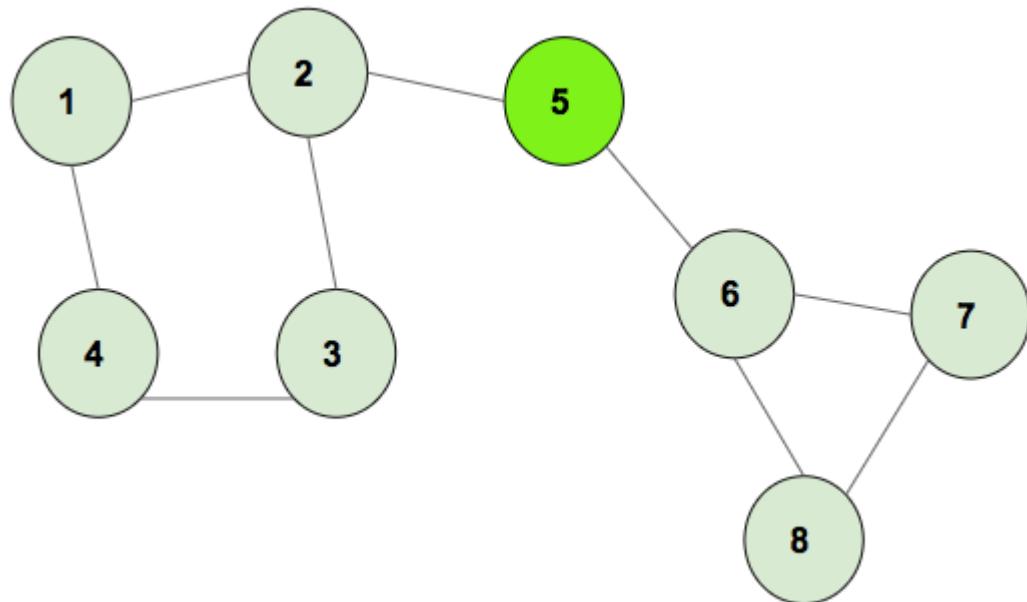
Chapter 243

Product of lengths of all cycles in an undirected graph

Product of lengths of all cycles in an undirected graph - GeeksforGeeks

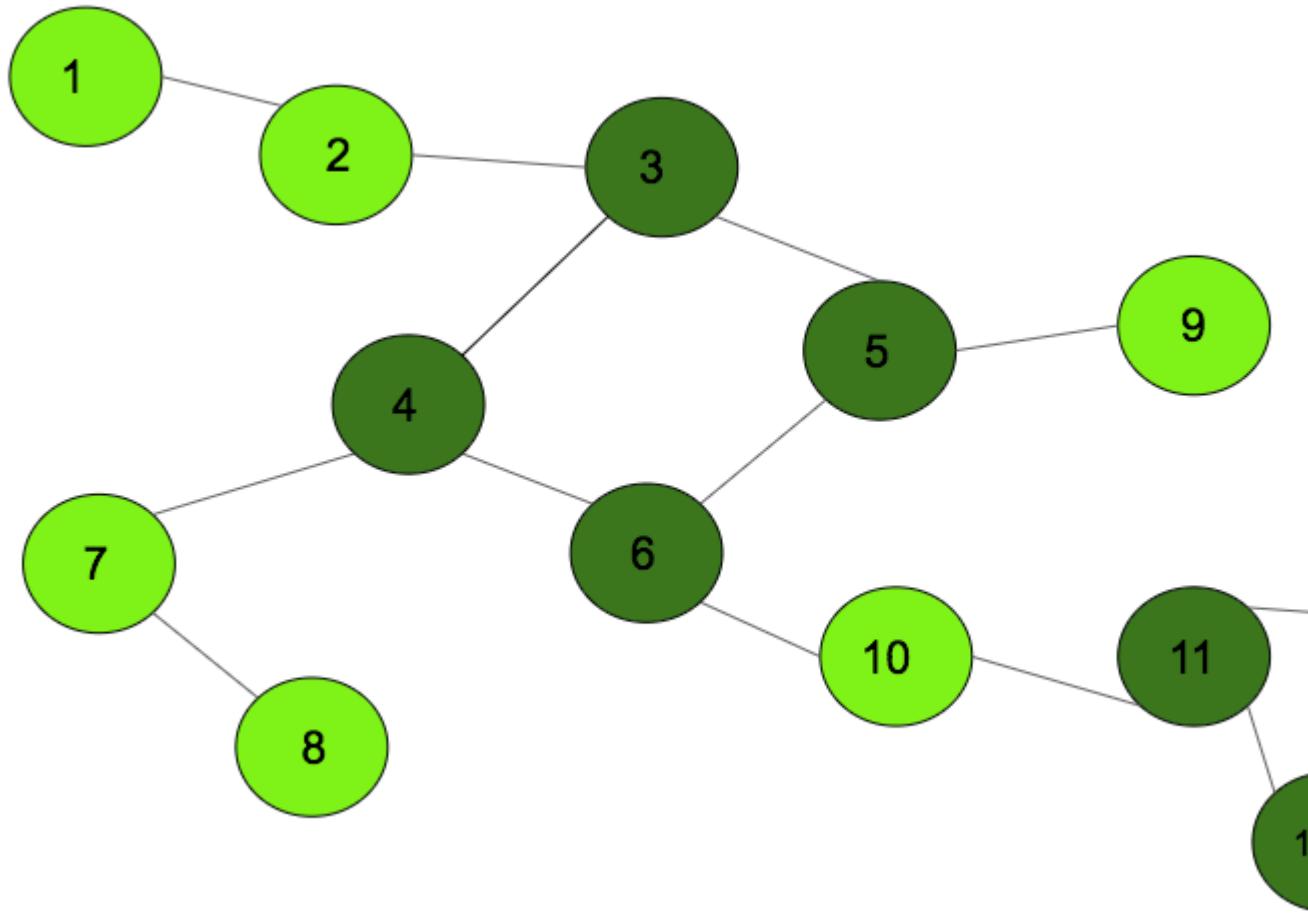
Given an undirected and unweighted graph. The task is to find the product of the lengths of all cycles formed in it.

Example 1:



The above graph has two cycles of length 4 and 3, the product of cycle lengths is 12.

Example 2:



The above graph has two cycles of length 4 and 3, the product of cycle lengths is 12.

Approach: Using the [graph coloring method](#), mark all the vertex of the different cycles with unique numbers. Once the graph traversal is completed, push all the similar marked numbers to an adjacency list and print the adjacency list accordingly. Given below is the algorithm:

- Insert the edges into an adjacency list.
- Call the DFS function which uses the coloring method to mark the vertex.
- Whenever there is a partially visited vertex, backtrack till the current vertex is reached and mark all of them with cycle numbers. Once all the vertexes are marked, increase the cycle number.

- Once Dfs is completed, iterate for the edges and push the same marked number edges to another adjacency list.
- Iterate in the another adjacency list, and keep the count of number of vertex in a cycle using map and cycle numbers
- Iterate for cycle numbers, and multiply the lengths to get the final product which will be the answer.

Below is the implementation of the above approach:

```

// C++ program to find the
// product of lengths of cycle
#include <bits/stdc++.h>
using namespace std;
const int N = 100000;

// variables to be used
// in both functions
vector<int> graph[N];

// Function to mark the vertex with
// different colors for different cycles
void dfs_cycle(int u, int p, int color[],
               int mark[], int par[], int& cyclenumber)
{

    // already (completely) visited vertex.
    if (color[u] == 2) {
        return;
    }

    // seen vertex, but was not completely
    // visited -> cycle detected.
    // backtrack based on parents to find
    // the complete cycle.
    if (color[u] == 1) {

        cyclenumber++;
        int cur = p;
        mark[cur] = cyclenumber;

        // backtrack the vertex which are
        // in the current cycle thots found
        while (cur != u) {
            cur = par[cur];
            mark[cur] = cyclenumber;
        }
    }
}

```

```

        return;
    }
    par[u] = p;

    // partially visited.
    color[u] = 1;

    // simple dfs on graph
    for (int v : graph[u]) {

        // if it has not been visited previously
        if (v == par[u]) {
            continue;
        }
        dfs_cycle(v, u, color, mark, par, cyclenumber);
    }

    // completely visited.
    color[u] = 2;
}

// add the edges to the graph
void addEdge(int u, int v)
{
    graph[u].push_back(v);
    graph[v].push_back(u);
}

// Function to print the cycles
int productLength(int edges, int mark[], int& cyclenumber)
{
    unordered_map<int, int> mp;

    // push the edges that into the
    // cycle adjacency list
    for (int i = 1; i <= edges; i++) {
        if (mark[i] != 0)
            mp[mark[i]]++;
    }
    int cnt = 1;

    // product all the length of cycles
    for (int i = 1; i <= cyclenumber; i++) {
        cnt = cnt * mp[i];
    }
    if (cyclenumber == 0)
        cnt = 0;
}

```

```
    return cnt;
}

// Driver Code
int main()
{

    // add edges
    addEdge(1, 2);
    addEdge(2, 3);
    addEdge(3, 4);
    addEdge(4, 6);
    addEdge(4, 7);
    addEdge(5, 6);
    addEdge(3, 5);
    addEdge(7, 8);
    addEdge(6, 10);
    addEdge(5, 9);
    addEdge(10, 11);
    addEdge(11, 12);
    addEdge(11, 13);
    addEdge(12, 13);

    // arrays required to color the
    // graph, store the parent of node
    int color[N];
    int par[N];

    // mark with unique numbers
    int mark[N];

    // store the numbers of cycle
    int cyclenumber = 0;
    int edges = 13;

    // call DFS to mark the cycles
    dfs_cycle(1, 0, color, mark, par, cyclenumber);

    // function to print the cycles
    cout << productLength(edges, mark, cyclenumber);

    return 0;
}
```

Output:

Time Complexity: $O(N)$, where N is the number of nodes in the graph.

Source

<https://www.geeksforgeeks.org/product-of-lengths-of-all-cycles-in-an-undirected-graph/>

Chapter 244

Proof that Hamiltonian Path is NP-Complete

Proof that Hamiltonian Path is NP-Complete - GeeksforGeeks

Prerequisite : [NP-Completeness](#)

The class of languages for which membership can be *decided* quickly fall in the class of **P** and The class of languages for which membership can be *verified* quickly fall in the class of **NP**(stands for problem solved in Non-deterministic Turing Machine in polynomial time). In straight words, every NP problem has its own polynomial-time verifier. A verifier for a language A is an algorithm V, where

```
A = {w | V accepts (w, c) for some string c}  
where c is certificate or proof that w is a member of A.
```

We are interested in NP-Complete problems. NP-Complete problem is defined as follows:

- (1)The problem itself is in NP class.
- (2)All other problems in NP class can be polynomial time reducible to that.
(B is polynomial time reducible to C is denoted as)

If the 2nd condition is only satisfied then the problem is called NP-Hard.
But it is not possible to reduce every NP problem into another NP problem to show its NP-Completeness all the time. That is why if we want to show a problem is NP-Complete we just show that the problem is in NP and any NP-Complete problem is reducible to that then we are done, i.e. if B is NP-Complete and $B \leq P C$ for C in NP, then C is NP-Complete.

We have to show Hamiltonian Path is NP-Complete. **Hamiltonian Path** or **HAMPATH** in a directed graph G is a directed path that goes through each node exactly once. We Consider the problem of testing whether a directed graph contain a Hamiltonian path connecting two specified nodes, i.e.

HAMPATH = {(G, s, t) | G is directed graph with a Hamiltonian path from s to t}

To prove HAMPATH is NP-Complete we have to prove that HAMPATH is in NP. To prove HAMPATH is in NP we must have a polynomial-time verifier. Even though we don't have a fast polynomial time algorithm to determine whether a graph contains a HAMPATH or not, if such a path is discovered somehow (maybe with exponential time brute force searching) we could easily-work it out whether the path is HAMPATH or not, in polynomial time. Here the certificate will be a Hamiltonian path from s to t itself in G if exists. So HAMPATH is in NP proved.

So, now we have to show that every problem to NP class is polynomial time reducible to HAMPATH to show its NP-Completeness. Rather we shall show 3SAT (A NP-Complete problem proved previously from SAT(Circuit Satisfiability Problem)) is polynomial time reducible to HAMPATH. We will convert a given cnf (Conjunctive Normal Form) form to a graph where gadgets (structure to simulate variables and clauses) will mimic the variables and clauses (several literals or variables connected with \vee). We have to prove now

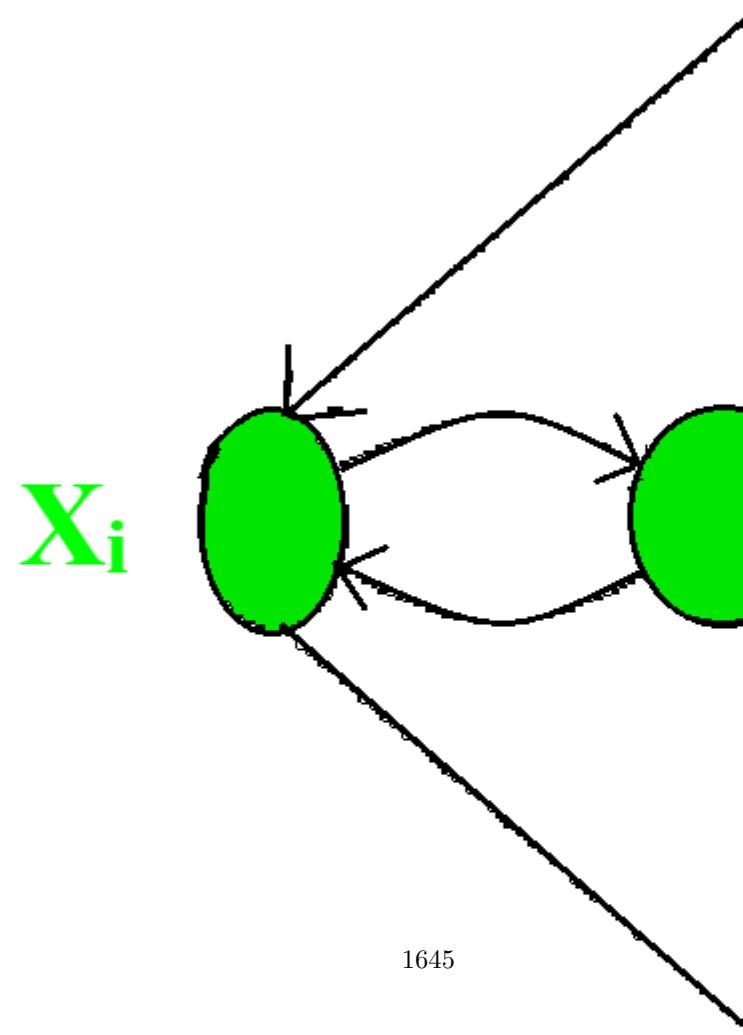
3SAT \leq_p HAMPATH

For each 3-cnf formula Φ we will show how to build graph G with s and t, where a Hamiltonian path exists between s and t iff Φ is satisfiable.

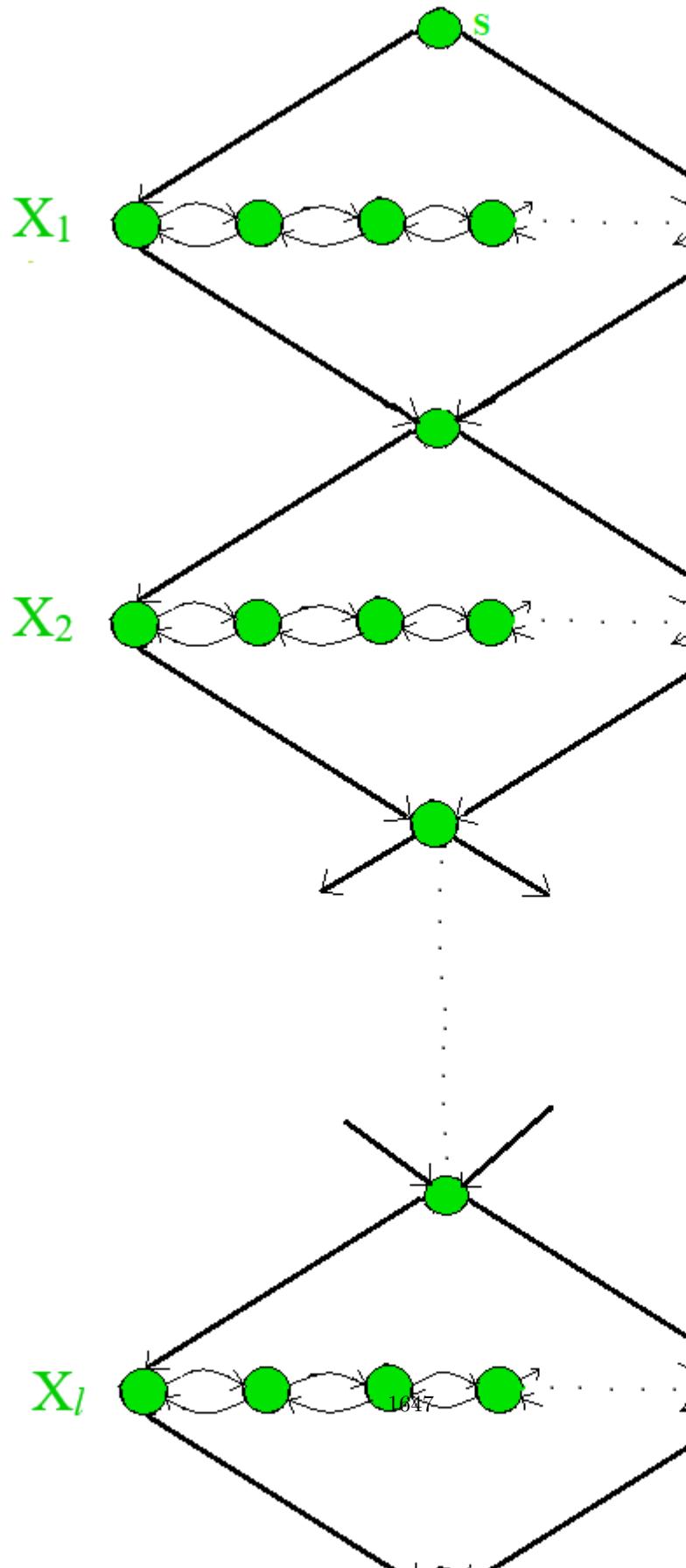
We start with a 3-cnf formula Φ containing k clauses,

where each a_i, b_i, c_i is a literal x_i or \bar{x}_i . Let x_1, x_2, \dots, x_l be the l variables of Φ . Now we show how to convert Φ to a graph G. The graph G that we construct has various parts to represent the variables and clauses that appear in Φ .

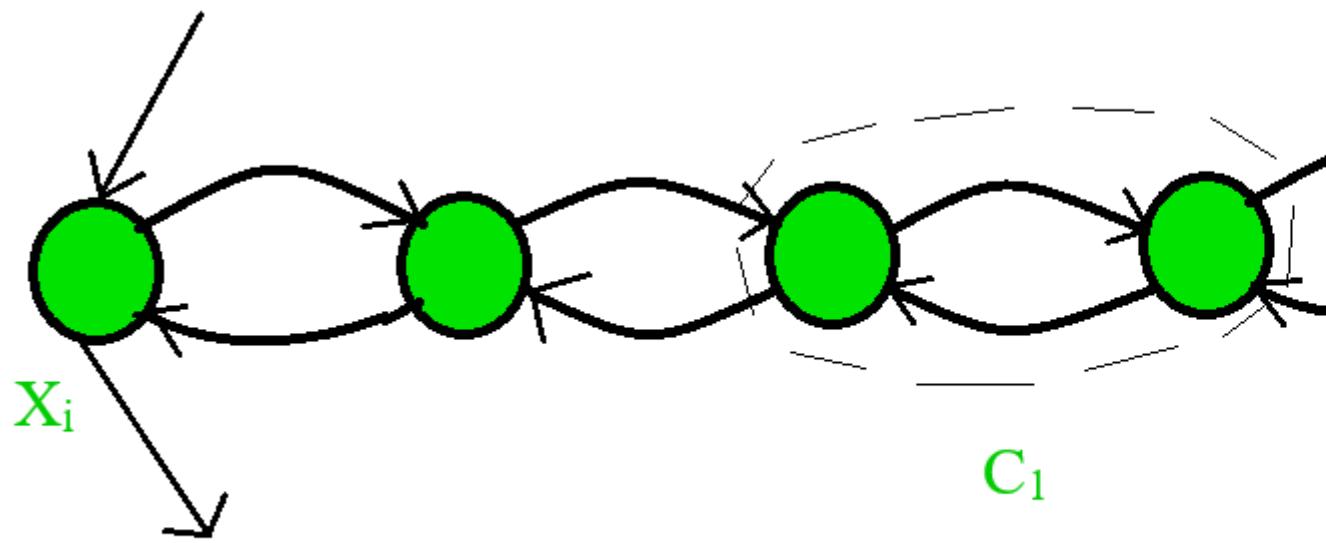
We represent each variable x_i with a diamond-shaped structure that contains a horizontal row of nodes as shown in following figure. We specify the number of nodes that appear in the horizontal row later.



The following figure depicts the global structure of G. It shows all the elements of G and their relationships, except the edges that represent the relationship of the variables to the clauses that contain them.

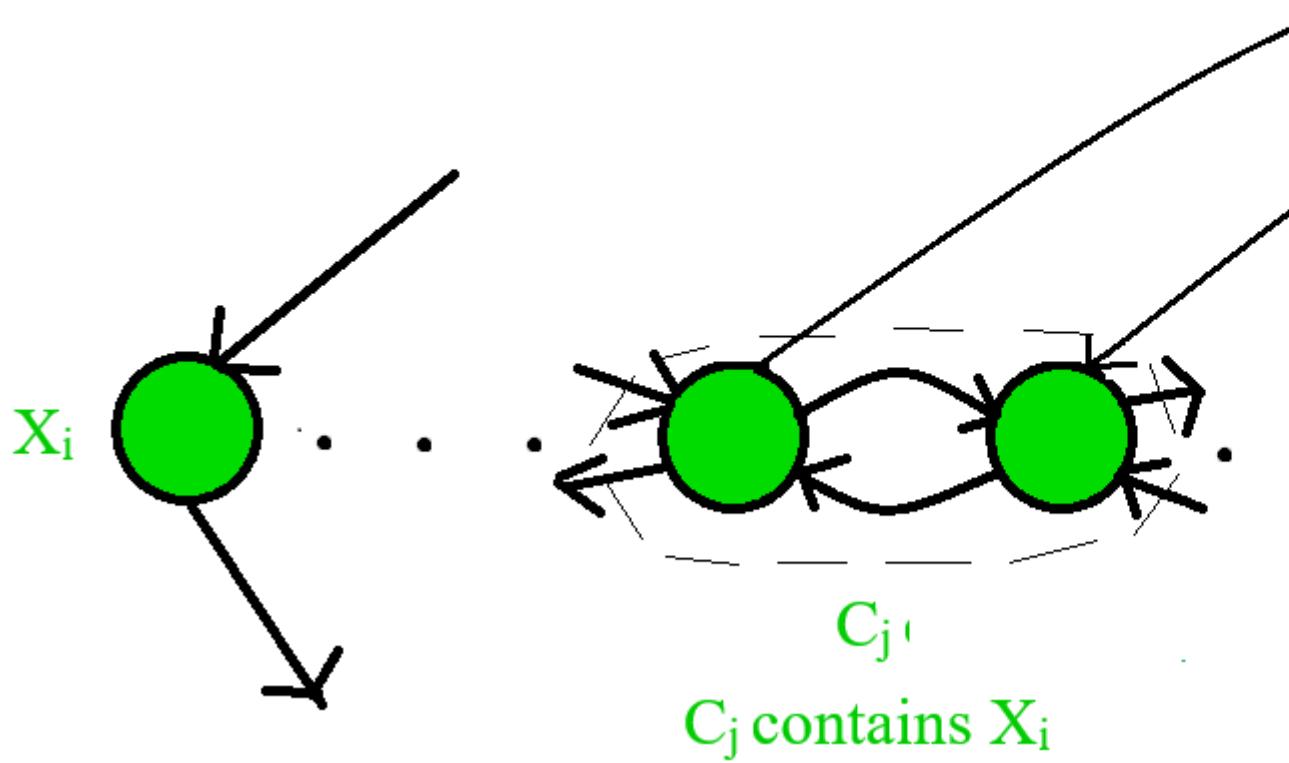


Each diamond structure contains a horizontal row of nodes connected by edges running in both directions. The horizontal row contains $2k$ nodes (as 2 nodes for each clause) plus, $k-1$ extra nodes in between every two nodes for a clause plus, 2 nodes on the ends belonging to the diamonds; total $3k+1$ nodes. Following diagram gives a clear picture.

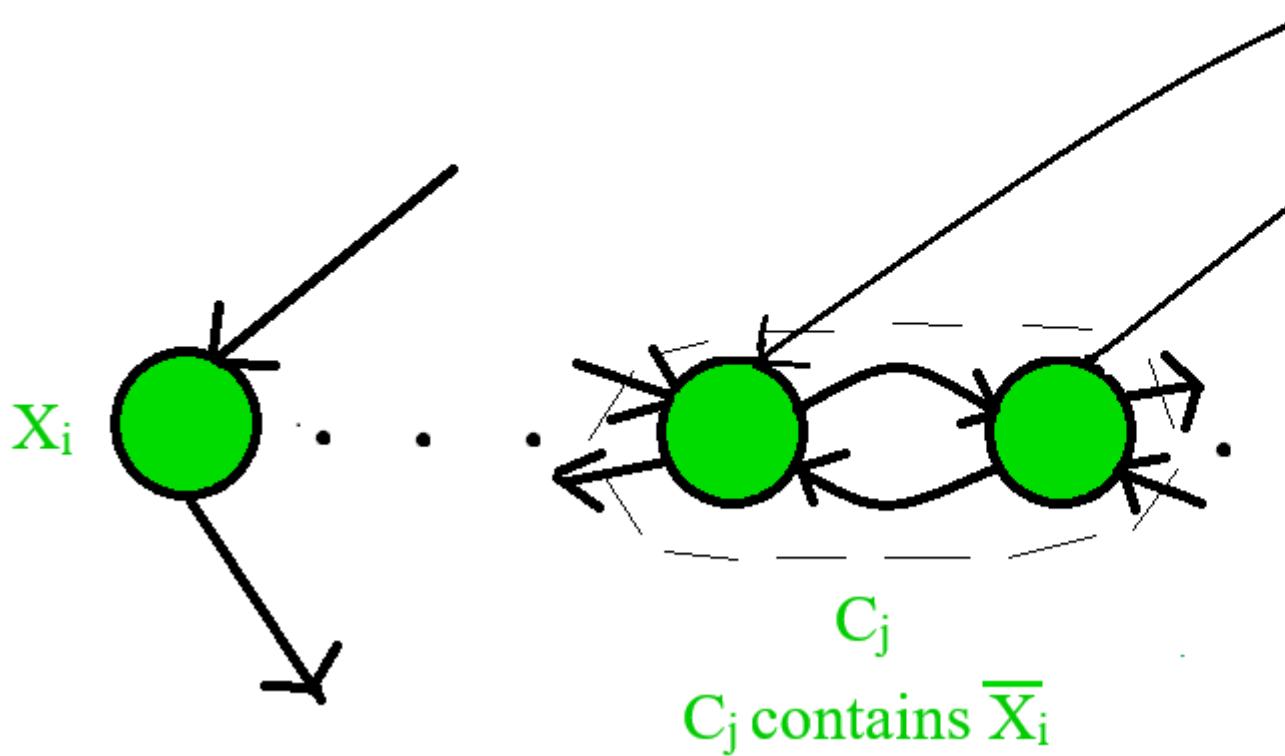


Horizontal nodes i

If variable x_i appears in clause $\neg j$, we add the following two edges from the j th pair in the i th diamond to the j th clause node.

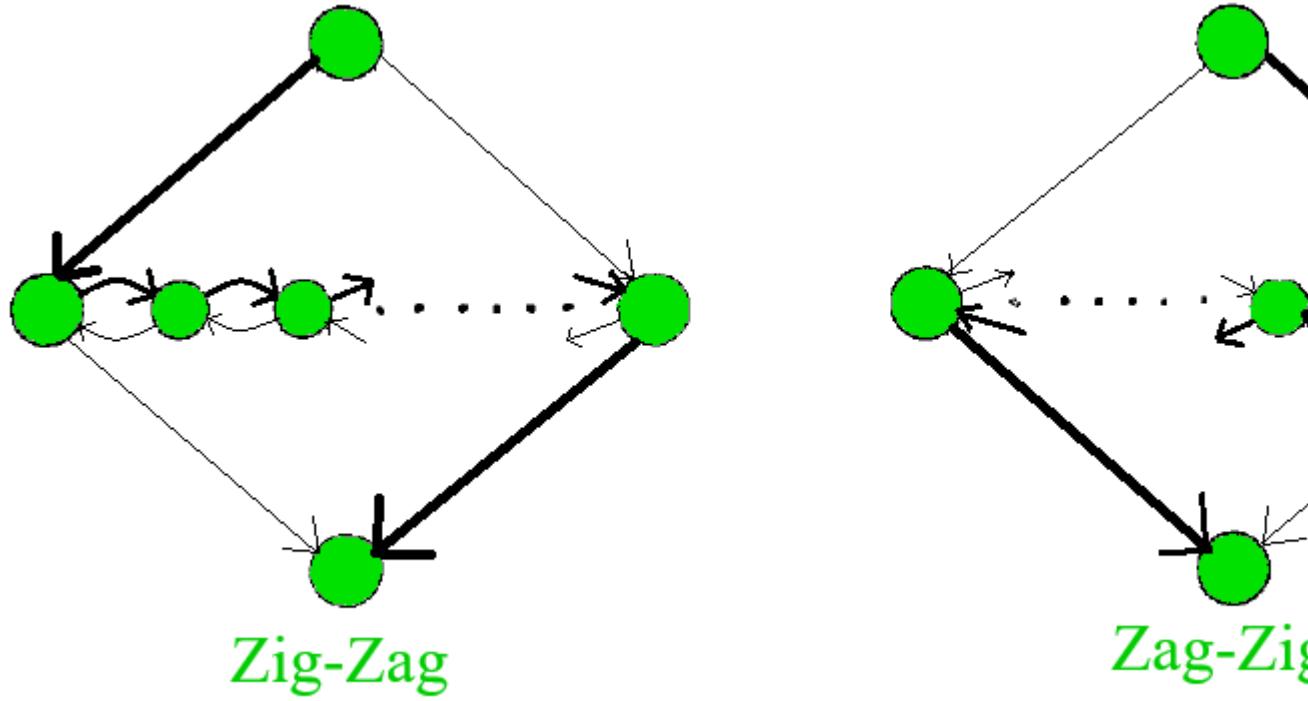


If $\overline{x_2}$ appears in clause c_j , we add two edges from the j th pair in the i th diamond to the j th clause node, as in the following figure.



After we add all the edges corresponding to each occurrence of φ_i or φ_j in each clause, the construction of G is complete. To show its correctness we will prove if φ is satisfiable, a Hamiltonian path exists from s to t and conversely, if such a path exists φ is satisfiable.

Suppose that φ is satisfiable. To demonstrate a Hamiltonian path from s to t , we first ignore the clause nodes. The path begins at s , goes through each diamond in turn and ends up at t . To hit horizontal nodes in a diamond, the path either zig-zags from left to right or zag-zigs from right to left; the satisfying assignment to φ determines whether φ_i is assigned TRUE or FALSE respectively. We show both cases in the following figure.



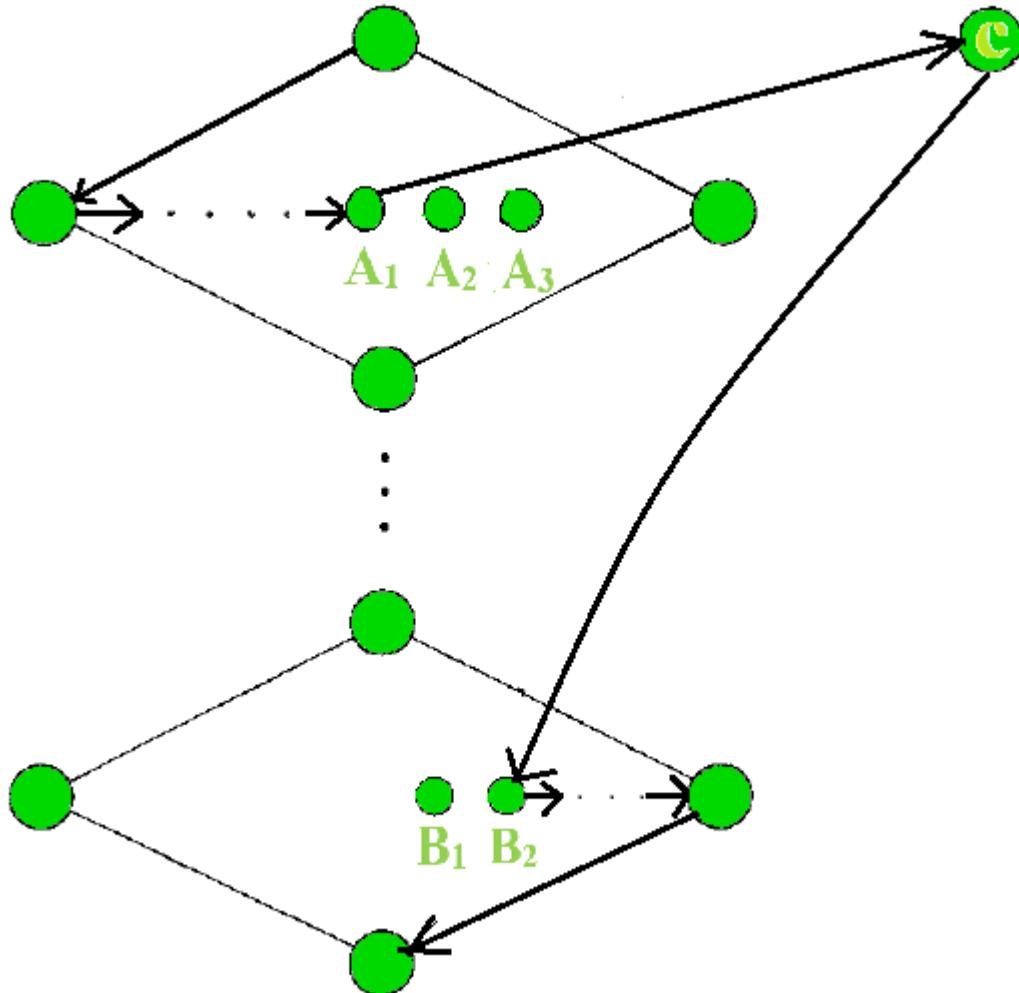
So far this path covers all the nodes in G except the clause nodes. We can easily include them by adding detours at the horizontal nodes. In each clause, we select one of the literals assigned TRUE, by satisfying assignment.

If we selected φ_i in clause $\neg j$, we can detour at the j th pair in the i th diamond. Doing so is possible because φ_i must be TRUE, so the path zig-zags from left to right through the corresponding diamond. Hence the edges to the $\neg j$ node are in the correct order to allow a detour and return.

Similarly, if we selected φ_i in clause $\neg j$, we can detour at the j th pair in the i th diamond because φ_i must be FALSE, so the path zag-zigs from right to left through the corresponding diamond. Hence the edges to the $\neg j$ node are again in the correct order to allow a detour and return. Every time one detour is taken if each literal in clause provides an option for detour. Thus each node is visited exactly once and thus Hamiltonian Path is constructed.

For the reverse direction, if G has a Hamiltonian path from s to t , we demonstrate a satisfying assignment for φ . If the Hamiltonian path is normal i.e. it goes through diamonds in order from top to bottom node except the detour for the closure nodes; we can easily obtain the satisfying assignment. If the path zig-zags in diamond we assign the variables as TRUE and if it zag-zigs then assign it FALSE. Because every node appears on the path by observing how the detour is taken we may determine corresponding TRUE variables.

All that remains to be shown that Hamiltonian path must be normal means the path enters a clause from one diamond but returns to another like in the following figure.



The path goes from node a_1 to c ; but instead of returning to a_2 in the same diamond, it returns to b_2 in the different diamond. If that occurs then either a_3 or b_3 must be a separator node. If a_3 were a separator node, the only edges entering in a_3 would be from a_1 or a_2 . If b_3 were a separator node then a_1 and b_3 would be in same clause, then edges that enters b_3 from a_1 , a_2 and c . In either case path cannot enter b_3 from a_3 because a_3 is only available node that b_3 points at, so path must exit b_3 via b_2 . Hence Hamiltonian path must be normal. This reduction obviously operates in polynomial time and hence the proof is complete that HAMPATH is NP-Complete.

Images reference: https://tr.m.wikipedia.org/wiki/Hamilton_path

Source

<https://www.geeksforgeeks.org/proof-hamiltonian-path-np-complete/>

Chapter 245

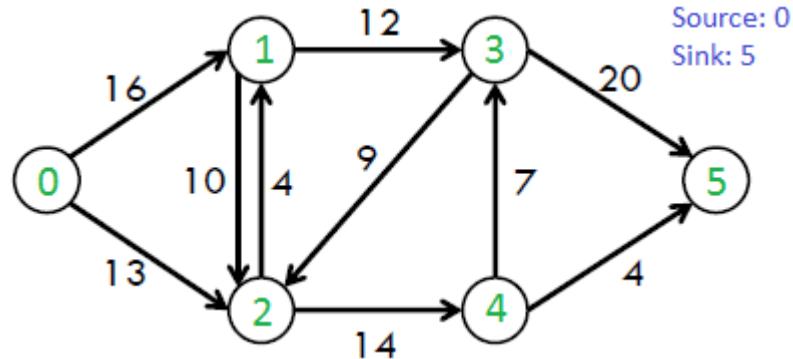
Push Relabel Algorithm Set 1 (Introduction and Illustration)

Push Relabel Algorithm Set 1 (Introduction and Illustration) - GeeksforGeeks

Given a graph which represents a flow network where every edge has a capacity. Also given two vertices *source* 's' and *sink* 't' in the graph, find the maximum possible flow from s to t with following constraints:

- a) Flow on an edge doesn't exceed the given capacity of the edge.
- b) Incoming flow is equal to outgoing flow for every vertex except s and t.

For example, consider the following graph from CLRS book.



The maximum possible flow in the above graph is 23.

We have discussed [Ford Fulkerson Algorithm](#) that uses augmenting path for computing maximum flow.

Push-Relabel Algorithm

Push-Relabel approach is the more efficient than Ford-Fulkerson algorithm. In this post, Goldberg's "generic" maximum-flow algorithm is discussed that runs in $O(V^2E)$ time. This

time complexity is better than $O(E^2V)$ which is time complexity of Edmond-Karp algorithm (a BFS based implementation of Ford-Fulkerson). There exist a push-relabel approach based algorithm that works in $O(V^3)$ which is even better than the one discussed here.

Similarities with Ford Fulkerson

- Like Ford-Fulkerson, Push-Relabel also works on Residual Graph (Residual Graph of a flow network is a graph which indicates additional possible flow. If there is a path from source to sink in residual graph, then it is possible to add flow).

Differences with Ford Fulkerson

- Push-relabel algorithm works in a more localized. Rather than examining the entire residual network to find an augmenting path, push-relabel algorithms work on one vertex at a time (Source : CLRS Book).
- In Ford-Fulkerson, net difference between total outflow and total inflow for every vertex (Except source and sink) is maintained 0. Push-Relabel algorithm allows inflow to exceed the outflow before reaching the final flow. In final flow, the net difference is 0 for all except source and sink.
- Time complexity wise more efficient.

The intuition behind the push-relabel algorithm (considering a fluid flow problem) is that we consider edges as water pipes and nodes are joints. The source is considered to be at the highest level and it sends water to all adjacent nodes. Once a node has excess water, it **pushes** water to a smaller height node. If water gets locally trapped at a vertex, the vertex is **Relabeled** which means its height is increased.

Following are some useful facts to consider before we proceed to algorithm.

- Each vertex has associated to it a height variable and a Excess Flow. **Height** is used to determine whether a vertex can push flow to an adjacent or not (A vertex can push flow only to a smaller height vertex). **Excess flow** is the difference of total flow coming into the vertex minus the total flow going out of the vertex.

$$\text{Excess Flow of } u = \text{Total Inflow to } u - \text{Total Outflow from } u$$

- Like Ford Fulkerson. each edge has associated to it a **flow** (which indicates current flow) and a **capacity**

Following are abstract steps of complete algorithm.

Push-Relabel Algorithm

- 1) Initialize PreFlow : Initialize Flows and Heights
- 2) While it is possible to perform a Push() or Relabel() on a vertex

```
// Or while there is a vertex that has excess flow
    Do Push() or Relabel()

// At this point all vertices have Excess Flow as 0 (Except source
// and sink)
3) Return flow.
```

There are three main operations in Push-Relabel Algorithm

1. **Initialize PreFlow()** It initializes heights and flows of all vertices.

```
Preflow()
1) Initialize height and flow of every vertex as 0.
2) Initialize height of source vertex equal to total
    number of vertices in graph.
3) Initialize flow of every edge as 0.
4) For all vertices adjacent to source s, flow and
    excess flow is equal to capacity initially.
```

2. **Push()** is used to make the flow from a node which has excess flow. If a vertex has excess flow and there is an adjacent with smaller height (in residual graph), we push the flow from the vertex to the adjacent with lower height. The amount of pushed flow through the pipe (edge) is equal to the minimum of excess flow and capacity of edge.
3. **Relabel()** operation is used when a vertex has excess flow and none of its adjacent is at lower height. We basically increase height of the vertex so that we can perform push(). To increase height, we pick the minimum height adjacent (in residual graph, i.e., an adjacent to whom we can add flow) and add 1 to it.

Note that above operations are performed on residual graph (like [Ford-Fulkerson](#)).

Illustration:

Before we proceed to below example, we need to make sure that we understand residual graph (See [this](#) for more details of residual graph). Residual graph is different from graphs shown.

Whenever we push or add a flow from a vertex u to v, we do following updates in residual graph :

- 1) We subtract the flow from capacity of edge from u to v. If capacity of an edge becomes 0, then the edge no longer exists in residual graph.
- 2) We add flow to the capacity of edge from v to u.

For example, consider two vertices u and v.

In original graph

$3/10$

$u \xrightarrow{\quad} v$

3 is current flow from u to v and
10 is capacity of edge from u to v.

In residual Graph, there are two edges corresponding to one edge shown above.

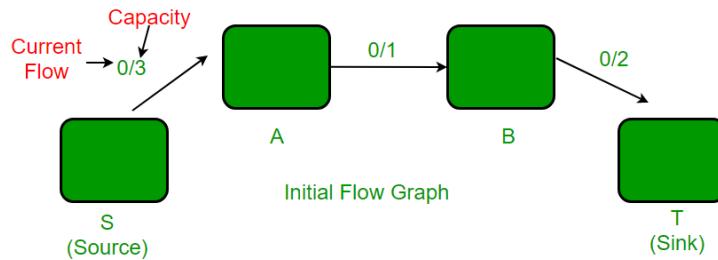
7

$u \xrightarrow{\quad} v$

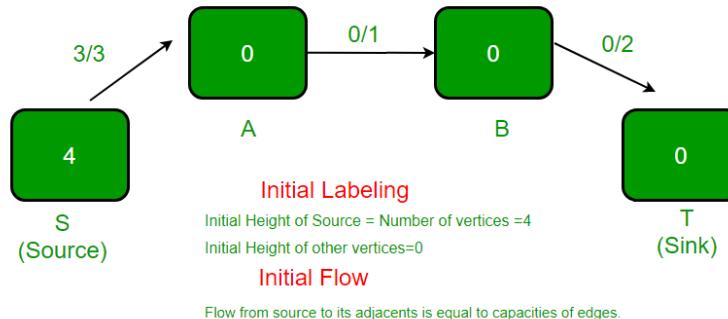
3

$u \xleftarrow{\quad} v$

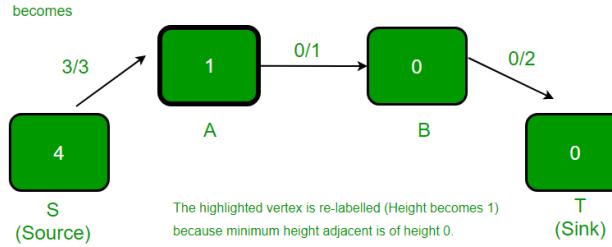
- Initial given flow graph.



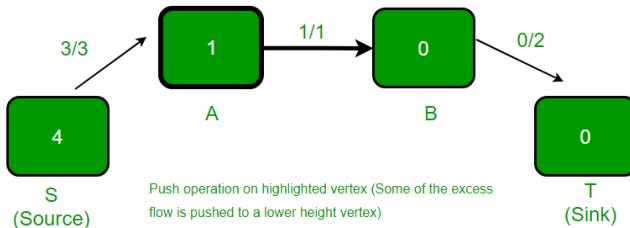
- After PreFlow operation. In residual graph, there is an edge from A to S with capacity 3 and no edge from S to A.



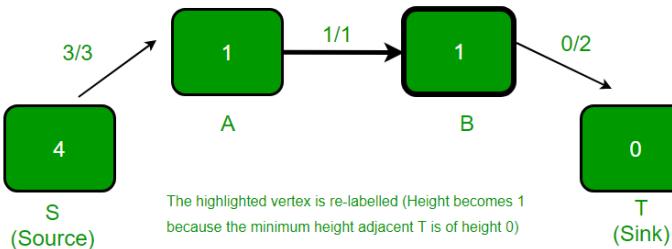
- The highlighted vertex is relabeled (height becomes 1) as it has excess flow and there is no adjacent with smaller height. The new height is equal to minimum of heights of adjacent plus 1. In residual graph, there are two adjacent of vertex A, one is S and other is B. Height of S is 4 and height of B is 0. Minimum of these two heights is 0. We take the minimum and add 1 to it.



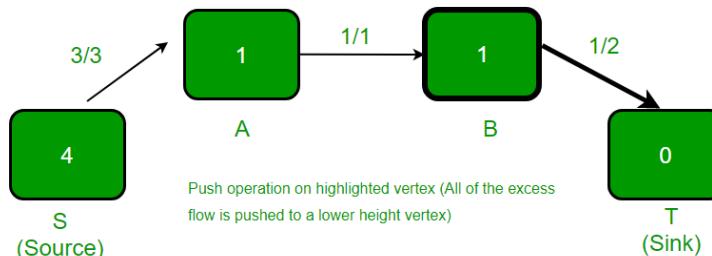
4. The highlighted vertex has excess flow and there is an adjacent with lower height, so push() happens. Excess flow of vertex A is 2 and capacity of edge (A, B) is 1. Therefore, the amount of pushed flow is 1 (minimum of two values).



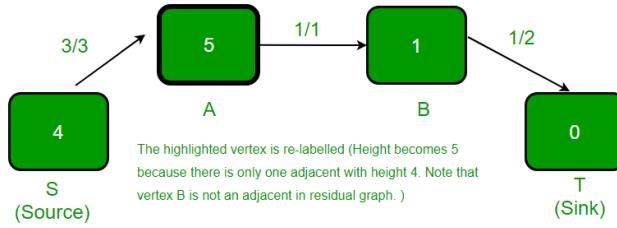
5. The highlighted vertex is relabeled (height becomes 1) as it has excess flow and there is no adjacent with smaller height.



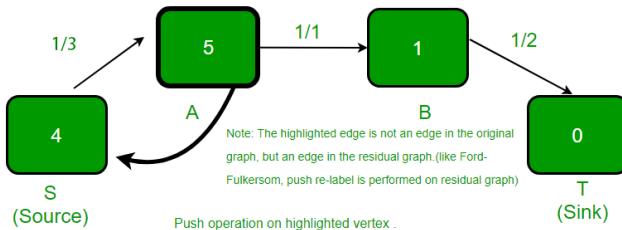
6. The highlighted vertex has excess flow and there is an adjacent with lower height, so flow() is pushed from B to T.



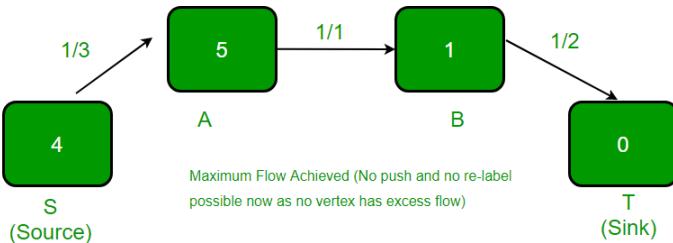
7. The highlighted vertex is relabeled (height becomes 5) as it has excess flow and there is no adjacent with smaller height.



8. The highlighted vertex has excess flow and there is an adjacent with lower height, so push() happens.



9. The highlighted vertex is relabeled (height is increased by 1) as it has excess flow and there is no adjacent with smaller height.



The above example is taken from [here](#).

[Push Relabel Algorithm Set 2 \(Implementation\)](#)

Source

<https://www.geeksforgeeks.org/push-relabel-algorithm-set-1-introduction-and-illustration/>

Chapter 246

Push Relabel Algorithm Set 2 (Implementation)

Push Relabel Algorithm Set 2 (Implementation) - GeeksforGeeks

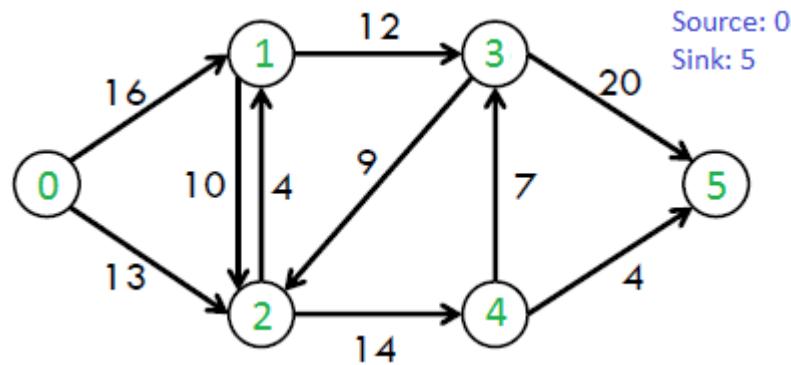
We strongly recommend to refer below article before moving on to this article.

[Push Relabel Algorithm Set 1 \(Introduction and Illustration\)](#)

Problem Statement : Given a graph which represents a flow network where every edge has a capacity. Also given two vertices *source* ‘s’ and *sink* ‘t’ in the graph, find the maximum possible flow from s to t with following constraints:

- a) Flow on an edge doesn’t exceed the given capacity of the edge.
- b) Incoming flow is equal to outgoing flow for every vertex except s and t.

For example, consider the following graph from CLRS book.



The maximum possible flow in the above graph is 23.

Push-Relabel Algorithm

- 1) Initialize PreFlow : Initialize Flows and Heights

```

2) While it is possible to perform a Push() or Relabel() on a vertex
   // Or while there is a vertex that has excess flow
      Do Push() or Relabel()

   // At this point all vertices have Excess Flow as 0 (Except source
   // and sink)
3) Return flow.

```

Below are main operations performed in Push Relabel algorithm.

There are three main operations in Push-Relabel Algorithm

1. **Initialize PreFlow()** It initializes heights and flows of all vertices.

```

Preflow()
1) Initialize height and flow of every vertex as 0.
2) Initialize height of source vertex equal to total
   number of vertices in graph.
3) Initialize flow of every edge as 0.
4) For all vertices adjacent to source s, flow and
   excess flow is equal to capacity initially.

```

2. **Push()** is used to make the flow from a node which has excess flow. If a vertex has excess flow and there is an adjacent with smaller height (in residual graph), we push the flow from the vertex to the adjacent with lower height. The amount of pushed flow through the pipe (edge) is equal to the minimum of excess flow and capacity of edge.
3. **Relabel()** operation is used when a vertex has excess flow and none of its adjacent is at lower height. We basically increase height of the vertex so that we can perform push(). To increase height, we pick the minimum height adjacent (in residual graph, i.e., an adjacent to whom we can add flow) and add 1 to it.

Implementation:

The following implementation uses below structure for representing a flow network.

```

struct Vertex
{
    int h;        // Height of node
    int e_flow; // Excess Flow
}

```

```

struct Edge
{
    int u, v; // Edge is from u to v
}

```

```
    int flow; // Current flow
    int capacity;
}

class Graph
{
    Edge edge[]; // Array of edges
    Vertex ver[]; // Array of vertices
}
```

The below code uses given graph itself as a flow network and residual graph. We have not created a separate graph for residual graph and have used the same graph for simplicity.

```
// C++ program to implement push-relabel algorithm for
// getting maximum flow of graph
#include <bits/stdc++.h>
using namespace std;

struct Edge
{
    // To store current flow and capacity of edge
    int flow, capacity;

    // An edge u--->v has start vertex as u and end
    // vertex as v.
    int u, v;

    Edge(int flow, int capacity, int u, int v)
    {
        this->flow = flow;
        this->capacity = capacity;
        this->u = u;
        this->v = v;
    }
};

// Represent a Vertex
struct Vertex
{
    int h, e_flow;

    Vertex(int h, int e_flow)
    {
        this->h = h;
        this->e_flow = e_flow;
    }
};
```

```
};

// To represent a flow network
class Graph
{
    int V;      // No. of vertices
    vector<Vertex> ver;
    vector<Edge> edge;

    // Function to push excess flow from u
    bool push(int u);

    // Function to relabel a vertex u
    void relabel(int u);

    // This function is called to initialize
    // preflow
    void preflow(int s);

    // Function to reverse edge
    void updateReverseEdgeFlow(int i, int flow);

public:
    Graph(int V); // Constructor

    // function to add an edge to graph
    void addEdge(int u, int v, int w);

    // returns maximum flow from s to t
    int getMaxFlow(int s, int t);
};

Graph::Graph(int V)
{
    this->V = V;

    // all vertices are initialized with 0 height
    // and 0 excess flow
    for (int i = 0; i < V; i++)
        ver.push_back(Vertex(0, 0));
}

void Graph::addEdge(int u, int v, int capacity)
{
    // flow is initialized with 0 for all edge
    edge.push_back(Edge(0, capacity, u, v));
}
```

```

void Graph::preflow(int s)
{
    // Making h of source Vertex equal to no. of vertices
    // Height of other vertices is 0.
    ver[s].h = ver.size();

    //
    for (int i = 0; i < edge.size(); i++)
    {
        // If current edge goes from source
        if (edge[i].u == s)
        {
            // Flow is equal to capacity
            edge[i].flow = edge[i].capacity;

            // Initialize excess flow for adjacent v
            ver[edge[i].v].e_flow += edge[i].flow;

            // Add an edge from v to s in residual graph with
            // capacity equal to 0
            edge.push_back(Edge(-edge[i].flow, 0, edge[i].v, s));
        }
    }
}

// returns index of overflowing Vertex
int overFlowVertex(vector<Vertex>& ver)
{
    for (int i = 1; i < ver.size() - 1; i++)
        if (ver[i].e_flow > 0)
            return i;

    // -1 if no overflowing Vertex
    return -1;
}

// Update reverse flow for flow added on ith Edge
void Graph::updateReverseEdgeFlow(int i, int flow)
{
    int u = edge[i].v, v = edge[i].u;

    for (int j = 0; j < edge.size(); j++)
    {
        if (edge[j].v == v && edge[j].u == u)
        {
            edge[j].flow -= flow;
            return;
        }
    }
}

```

```
}

// adding reverse Edge in residual graph
Edge e = Edge(0, flow, u, v);
edge.push_back(e);
}

// To push flow from overflowing vertex u
bool Graph::push(int u)
{
    // Traverse through all edges to find an adjacent (of u)
    // to which flow can be pushed
    for (int i = 0; i < edge.size(); i++)
    {
        // Checks u of current edge is same as given
        // overflowing vertex
        if (edge[i].u == u)
        {
            // if flow is equal to capacity then no push
            // is possible
            if (edge[i].flow == edge[i].capacity)
                continue;

            // Push is only possible if height of adjacent
            // is smaller than height of overflowing vertex
            if (ver[u].h > ver[edge[i].v].h)
            {
                // Flow to be pushed is equal to minimum of
                // remaining flow on edge and excess flow.
                int flow = min(edge[i].capacity - edge[i].flow,
                               ver[u].e_flow);

                // Reduce excess flow for overflowing vertex
                ver[u].e_flow -= flow;

                // Increase excess flow for adjacent
                ver[edge[i].v].e_flow += flow;

                // Add residual flow (With capacity 0 and negative
                // flow)
                edge[i].flow += flow;

                updateReverseEdgeFlow(i, flow);
            }
        }
    }
}
```

```

        return false;
    }

// function to relabel vertex u
void Graph::relabel(int u)
{
    // Initialize minimum height of an adjacent
    int mh = INT_MAX;

    // Find the adjacent with minimum height
    for (int i = 0; i < edge.size(); i++)
    {
        if (edge[i].u == u)
        {
            // if flow is equal to capacity then no
            // relabeling
            if (edge[i].flow == edge[i].capacity)
                continue;

            // Update minimum height
            if (ver[edge[i].v].h < mh)
            {
                mh = ver[edge[i].v].h;

                // updating height of u
                ver[u].h = mh + 1;
            }
        }
    }
}

// main function for printing maximum flow of graph
int Graph::getMaxFlow(int s, int t)
{
    preflow(s);

    // loop untill none of the Vertex is in overflow
    while (overFlowVertex(ver) != -1)
    {
        int u = overFlowVertex(ver);
        if (!push(u))
            relabel(u);
    }

    // ver.back() returns last Vertex, whose
    // e_flow will be final maximum flow
    return ver.back().e_flow;
}

```

```
// Driver program to test above functions
int main()
{
    int V = 6;
    Graph g(V);

    // Creating above shown flow network
    g.addEdge(0, 1, 16);
    g.addEdge(0, 2, 13);
    g.addEdge(1, 2, 10);
    g.addEdge(2, 1, 4);
    g.addEdge(1, 3, 12);
    g.addEdge(2, 4, 14);
    g.addEdge(3, 2, 9);
    g.addEdge(3, 5, 20);
    g.addEdge(4, 3, 7);
    g.addEdge(4, 5, 4);

    // Initialize source and sink
    int s = 0, t = 5;

    cout << "Maximum flow is " << g.getMaxFlow(s, t);
    return 0;
}
```

Output

```
Maximum flow is 23
```

The code in this article is contributed by **Siddharth Lalwani** and **Utkarsh Trivedi**.

Source

<https://www.geeksforgeeks.org/push-relabel-algorithm-set-2-implementation/>

Chapter 247

Remove all outgoing edges except edge with minimum weight

Remove all outgoing edges except edge with minimum weight - GeeksforGeeks

Given a directed graph having n nodes. For each node, delete all the outgoing edges except the outgoing edge with minimum weight. Apply this deletion operation for every node and then print the final graph remained where each node of the graph has at most one outgoing edge and that too with minimum weight.

Note: Here, graph is stored as Adjacency Matrix for ease.

Examples :

Input : Adjacency Matrix of input graph :

	1	2	3	4

1	0	3	2	5
2	0	2	4	7
3	1	2	0	3
4	5	2	1	3

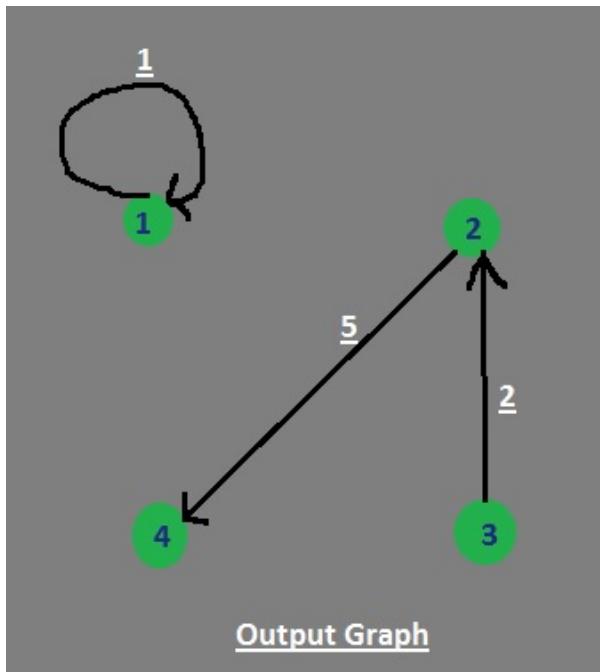
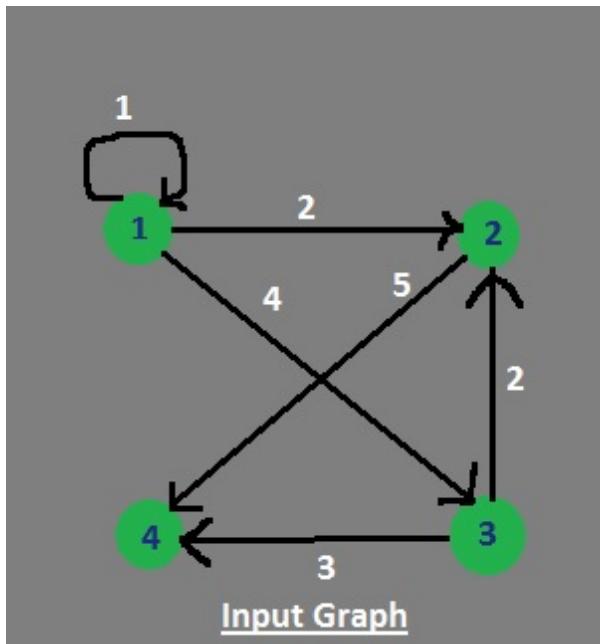
Output : Adjacency Matrix of output graph :

	1	2	3	4

1	0	0	2	0
2	0	2	0	0
3	1	0	0	0
4	0	0	1	0

For every row of the adjacency matrix of graph keep the minimum element (except zero) and make rest of all zero. Do this for every row of the input matrix. Finally, print the resultant Matrix.

Example :



C++

```
// CPP program for minimizing graph
#include <bits/stdc++.h>

using namespace std;

// Utility function for
// finding min of a row
int minFn(int arr[])
{
    int min = INT_MAX;

    for (int i = 0; i < 4; i++)
        if (min > arr[i])
            min = arr[i];
    return min;
}

// Utility function for minimizing graph
void minimizeGraph(int arr[][4])
{
    int min;

    // Set empty edges to INT_MAX
    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 4; j++)
            if (arr[i][j] == 0)
                arr[i][j] = INT_MAX;

    // Finding minimum of each row
    // and deleting rest of edges
    for (int i = 0; i < 4; i++) {

        // Find minimum element of row
        min = minFn(arr[i]);

        for (int j = 0; j < 4; j++) {
            // If edge value is not min
            // set it to zero, also
            // edge value INT_MAX denotes that
            // initially edge value was zero
            if (!(arr[i][j] == min) || (arr[i][j] == INT_MAX))
                arr[i][j] = 0;
            else
                min = 0;
        }
    }

    // Print result;
}
```

```
for (int i = 0; i < 4; i++) {  
    for (int j = 0; j < 4; j++)  
        cout << arr[i][j] << " ";  
    cout << "\n";  
}  
}  
  
// Driver Program  
int main()  
{  
    // Input Graph  
    int arr[4][4] = { 1, 2, 4, 0,  
                    0, 0, 0, 5,  
                    0, 2, 0, 3,  
                    0, 0, 0, 0 };  
  
    minimizeGraph(arr);  
  
    return 0;  
}
```

Java

```
// Java program for  
// minimizing graph  
import java.io.*;  
import java.util.*;  
import java.lang.*;  
  
class GFG  
{  
  
    // Utility function for  
    // finding min of a row  
    static int minFn(int arr[])  
    {  
        int min = Integer.MAX_VALUE;  
  
        for (int i = 0;  
             i < arr.length; i++)  
            if (min > arr[i])  
                min = arr[i];  
        return min;  
    }  
  
    // Utility function  
    // for minimizing graph  
    static void minimizeGraph(int arr[] [])
```

```
{  
    int min;  
  
    // Set empty edges  
    // to INT_MAX  
    for (int i = 0;  
        i < arr.length; i++)  
        for (int j = 0;  
            j < arr.length; j++)  
            if (arr[i][j] == 0)  
                arr[i][j] = Integer.MAX_VALUE;  
  
    // Finding minimum of each  
    // row and deleting rest  
    // of edges  
    for (int i = 0;  
        i < arr.length; i++)  
    {  
  
        // Find minimum  
        // element of row  
        min = minFn(arr[i]);  
  
        for (int j = 0;  
            j < arr.length; j++)  
        {  
            // If edge value is not  
            // min set it to zero,  
            // also edge value INT_MAX  
            // denotes that initially  
            // edge value was zero  
            if ((arr[i][j] != min) ||  
                (arr[i][j] == Integer.MAX_VALUE))  
                arr[i][j] = 0;  
            else  
                min = 0;  
        }  
    }  
  
    // Print result;  
    for (int i = 0;  
        i < arr.length; i++)  
    {  
        for (int j = 0;  
            j < arr.length; j++)  
            System.out.print(arr[i][j] + " ");  
        System.out.print("\n");  
    }  
}
```

```
}  
  
// Driver Code  
public static void main(String[] args)  
{  
    // Input Graph  
    int arr[][] = {{1, 2, 4, 0},  
                  {0, 0, 0, 5},  
                  {0, 2, 0, 3},  
                  {0, 0, 0, 0}};  
  
    minimizeGraph(arr);  
}  
}
```

Output:

```
1 0 0 0  
0 0 0 5  
0 2 0 0  
0 0 0 0
```

Time Complexity: $O(n^2)$

Source

<https://www.geeksforgeeks.org/minimum-weighted-graph/>

Chapter 248

Reverse Delete Algorithm for Minimum Spanning Tree

Reverse Delete Algorithm for Minimum Spanning Tree - GeeksforGeeks

Reverse Delete algorithm is closely related to [Kruskal's algorithm](#). In Kruskal's algorithm what we do is : Sort edges by increasing order of their weights. After sorting, we one by one pick edges in increasing order. We include current picked edge if by including this in spanning tree not form any cycle until there are $V-1$ edges in spanning tree, where V = number of vertices.

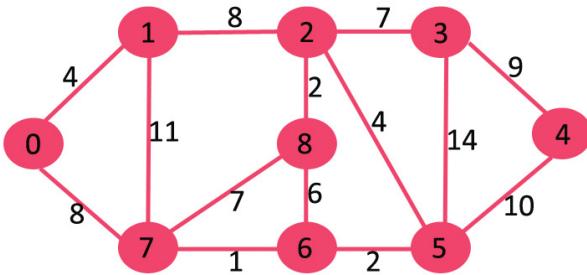
In Reverse Delete algorithm, we sort all edges in **decreasing** order of their weights. After sorting, we one by one pick edges in decreasing order. We **include current picked edge if excluding current edge causes disconnection in current graph**. The main idea is delete edge if its deletion does not lead to disconnection of graph.

The Algorithm

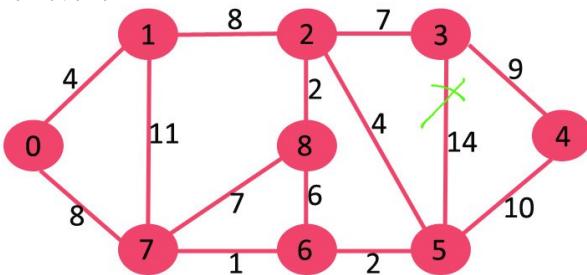
- 1) Sort all edges of graph in non-increasing order of edge weights.
- 2) Initialize MST as original graph and remove extra edges using step 3.
- 3) Pick highest weight edge from remaining edges and check if deleting the edge disconnects the graph or not.
 - If disconnects, then we don't delete the edge.
 - Else we delete the edge and continue.

Illustration:

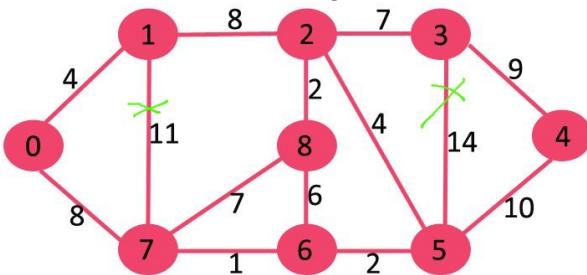
Let us understand with the following example:



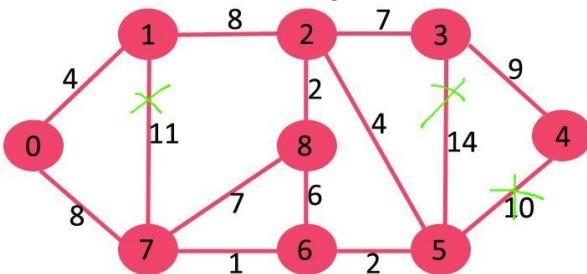
If we delete highest weight edge of weight 14, graph doesn't become disconnected, so we remove it.



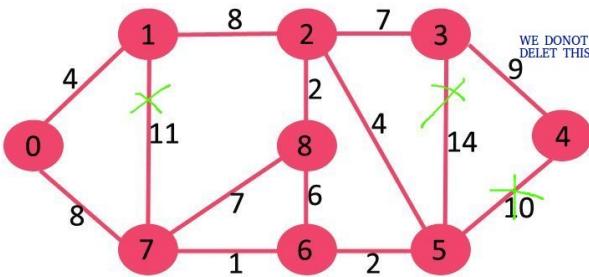
Next we delete 11 as deleting it doesn't disconnect the graph.



Next we delete 10 as deleting it doesn't disconnect the graph.



Next is 9. We cannot delete 9 as deleting it causes disconnection.



We continue this way and following edges remain in final MST.

Edges in MST

- (3, 4)
- (0, 7)
- (2, 3)
- (2, 5)
- (0, 1)
- (5, 6)
- (2, 8)
- (6, 7)

Note : In case of same weight edges, we can pick any edge of the same weight edges.

Below is C++ implementation of above steps.

```

// C++ program to find Minimum Spanning Tree
// of a graph using Reverse Delete Algorithm
#include<bits/stdc++.h>
using namespace std;

// Creating shortcut for an integer pair
typedef pair<int, int> iPaiR;

// Graph class represents a directed graph
// using adjacency list representation
class Graph
{
    int V;      // No. of vertices
    list<int> *adj;
    vector< pair<int, iPaiR> > edges;
    void DFS(int v, bool visited[]);

public:
    Graph(int V);    // Constructor

    // function to add an edge to graph

```

```
void addEdge(int u, int v, int w);

// Returns true if graph is connected
bool isConnected();

void reverseDeleteMST();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int u, int v, int w)
{
    adj[u].push_back(v); // Add w to v's list.
    adj[v].push_back(u); // Add w to v's list.
    edges.push_back({w, {u, v}});
}

void Graph::DFS(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;

    // Recur for all the vertices adjacent to
    // this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFS(*i, visited);
}

// Returns true if given graph is connected, else false
bool Graph::isConnected()
{
    bool visited[V];
    memset(visited, false, sizeof(visited));

    // Find all reachable vertices from first vertex
    DFS(0, visited);

    // If set of reachable vertices includes all,
    // return true.
    for (int i=1; i<V; i++)
        if (visited[i] == false)
            return false;
}
```

```
    return true;
}

// This function assumes that edge (u, v)
// exists in graph or not,
void Graph::reverseDeleteMST()
{
    // Sort edges in increasing order on basis of cost
    sort(edges.begin(), edges.end());

    int mst_wt = 0; // Initialize weight of MST

    cout << "Edges in MST\n";

    // Iterate through all sorted edges in
    // decreasing order of weights
    for (int i=edges.size()-1; i>=0; i--)
    {
        int u = edges[i].second.first;
        int v = edges[i].second.second;

        // Remove edge from undirected graph
        adj[u].remove(v);
        adj[v].remove(u);

        // Adding the edge back if removing it
        // causes disconnection. In this case this
        // edge becomes part of MST.
        if (isConnected() == false)
        {
            adj[u].push_back(v);
            adj[v].push_back(u);

            // This edge is part of MST
            cout << "(" << u << ", " << v << ") \n";
            mst_wt += edges[i].first;
        }
    }

    cout << "Total weight of MST is " << mst_wt;
}

// Driver code
int main()
{
    // create the graph given in above figure
    int V = 9;
```

```
Graph g(V);

// making above shown graph
g.addEdge(0, 1, 4);
g.addEdge(0, 7, 8);
g.addEdge(1, 2, 8);
g.addEdge(1, 7, 11);
g.addEdge(2, 3, 7);
g.addEdge(2, 8, 2);
g.addEdge(2, 5, 4);
g.addEdge(3, 4, 9);
g.addEdge(3, 5, 14);
g.addEdge(4, 5, 10);
g.addEdge(5, 6, 2);
g.addEdge(6, 7, 1);
g.addEdge(6, 8, 6);
g.addEdge(7, 8, 7);

g.reverseDeleteMST();
return 0;
}
```

Output :

```
Edges in MST
(3, 4)
(0, 7)
(2, 3)
(2, 5)
(0, 1)
(5, 6)
(2, 8)
(6, 7)
Total weight of MST is 37
```

Notes :

1. The above implementation is a simple/naive implementation of Reverse Delete algorithm and can be optimized to $O(E \log V (\log \log V)^3)$ [Source : [Wiki](#)]. But this optimized time complexity is still less than [Prim](#) and [Kruskal](#) Algorithms for MST.
2. The above implementation modifies the original graph. We can create a copy of the graph if original graph must be retained.

References:

https://en.wikipedia.org/wiki/Reverse-delete_algorithm

Source

<https://www.geeksforgeeks.org/reverse-delete-algorithm-minimum-spanning-tree/>

Chapter 249

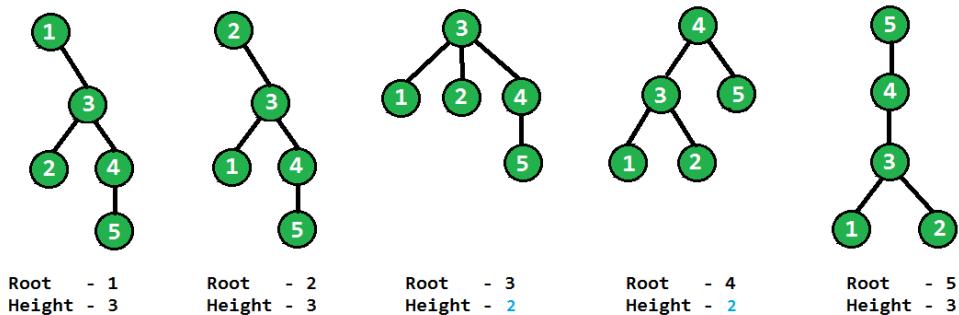
Roots of a tree which give minimum height

Roots of a tree which give minimum height - GeeksforGeeks

Given an undirected graph, which has tree characteristics. It is possible to choose any node as root, the task is to find those nodes only which minimize the height of tree.

Example:

In below diagram all node are made as root one by one, we can see that when 3 and 4 are root, height of tree is minimum(2) so {3, 4} is our answer.



We can solve this problem by first thinking about 1-D solution, that is if a longest graph is given, then the node which will minimize the height will be mid node if total node count is odd or mid two node if total node count is even. This solution can be reached by following approach – Start two pointers from both end of the path and move one step each time, until pointers meet or one step away, at the end pointers will be at those nodes which will minimize the height because we have divided the nodes evenly so height will be minimum. Same approach can be applied to a general tree also. Start pointers from all leaf nodes and move one step inside each time, keep combining pointers which overlap while moving, at the end only one pointer will remain on some vertex or two pointers will remain at one distance away. Those node represent the root of the vertex which will minimize the height of the

tree.

So we can have only one root or at max two root for minimum height depending on tree structure as explained above. For implementation we will not use actual pointers instead we'll follow BFS like approach, In starting all leaf node are pushed into the queue, then they are removed from tree, next new leaf node are pushed in queue, this procedure keeps on going until we have only 1 or 2 node in our tree, which represent the result.

As we are accessing each node once, total time complexity of solution is $O(n)$.

C++

```
// C++ program to find root which gives minimum height to tree
#include <bits/stdc++.h>
using namespace std;

// This class represents a undirected graph using adjacency list
// representation
class Graph
{
public:
    int V; // No. of vertices

    // Pointer to an array containing adjacency lists
    list<int> *adj;

    // Vector which stores degree of all vertices
    vector<int> degree;

    Graph(int V); // Constructor
    void addEdge(int v, int w); // To add an edge

    // function to get roots which give minimum height
    vector<int> rootForMinimumHeight();
};

// Constructor of graph, initializes adjacency list and
// degree vector
Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
    for (int i = 0; i < V; i++)
        degree.push_back(0);
}

// addEdge method adds vertex to adjacency list and increases
// degree by 1
void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list
```

```

        adj[w].push_back(v);      // Add v to w's list
        degree[v]++;             // increment degree of v by 1
        degree[w]++;             // increment degree of w by 1
    }

// Method to return roots which gives minimum height to tree
vector<int> Graph::rootForMinimumHeight()
{
    queue<int> q;

    // first enqueue all leaf nodes in queue
    for (int i = 0; i < V; i++)
        if (degree[i] == 1)
            q.push(i);

    // loop until total vertex remains less than 2
    while (V > 2)
    {
        for (int i = 0; i < q.size(); i++)
        {
            int t = q.front();
            q.pop();
            V--;

            // for each neighbour, decrease its degree and
            // if it become leaf, insert into queue
            for (auto j = adj[t].begin(); j != adj[t].end(); j++)
            {
                degree[*j]--;
                if (degree[*j] == 1)
                    q.push(*j);
            }
        }
    }

    // copying the result from queue to result vector
    vector<int> res;
    while (!q.empty())
    {
        res.push_back(q.front());
        q.pop();
    }
    return res;
}

// Driver code to test above methods
int main()
{

```

```
Graph g(6);
g.addEdge(0, 3);
g.addEdge(1, 3);
g.addEdge(2, 3);
g.addEdge(4, 3);
g.addEdge(5, 4);

vector<int> res = g.rootForMinimumHeight();
for (int i = 0; i < res.size(); i++)
    cout << res[i] << " ";
cout << endl;
}
```

Python

```
# Python program to find root which gives minimum
# height to tree

# This class represents a undirected graph using
# adjacency list representation
class Graph:

    # Constructor of graph, initialize adjacency list
    # and degree vector
    def __init__(self, V, addEdge, rootForMinimumHeight):
        self.V = V
        self.adj = dict( (i, []) for i in range(V))
        self.degree = list()
        for i in range(V):
            self.degree.append(0)

    # The below lines allows us define methods outside
    # of class definition
    # Check http://bit.ly/2e5HfrW for better explanation
    Graph.addEdge = addEdge
    Graph.rootForMinimumHeight = rootForMinimumHeight

    # addEdge method adds vertex to adjacency list and
    # increases degree by 1
    def addEdge(self, v, w):
        self.adj[v].append(w) # Adds w to v's list
        self.adj[w].append(v) # Adds v to w's list
        self.degree[v] += 1      # increment degree of v by 1
        self.degree[w] += 1      # increment degree of w by 1

    # Method to return roots which gives minimum height to tree
```

```
def rootForMinimumHeight(self):  
  
    from Queue import Queue  
    q = Queue()  
  
    # First enqueue all leaf nodes in queue  
    for i in range(self.V):  
        if self.degree[i] == 1:  
            q.put(i)  
  
    # loop until total vertex remains less than 2  
    while(self.V > 2):  
        for i in range(q.qsize()):  
            t = q.get()  
            self.V -=1  
  
            # for each neighbour, decrease its degree and  
            # if it become leaf, insert into queue  
            for j in self.adj[t]:  
                self.degree[j] -= 1  
                if self.degree[j] == 1:  
                    q.put(j)  
  
    # Copying the result from queue to result vector  
    res = list()  
    while(q.qsize() > 0):  
        res.append(q.get())  
  
    return res  
  
# Driver code to test above methods  
g = Graph(6, addEdge, rootForMinimumHeight )  
g.addEdge(0, 3)  
g.addEdge(1, 3)  
g.addEdge(2, 3)  
g.addEdge(4, 3)  
g.addEdge(5, 4)  
res = g.rootForMinimumHeight()  
for i in res:  
    print i,  
  
# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

Output:

3 4

Source

<https://www.geeksforgeeks.org/roots-tree-gives-minimum-height/>

Chapter 250

Shortest path in a Binary Maze

Shortest path in a Binary Maze - GeeksforGeeks

Given a MxN matrix where each element can either be 0 or 1. We need to find the shortest path between a given source cell to a destination cell. The path can only be created out of a cell if its value is 1.

Expected time complexity is O(MN).

For example –

Input:

```
mat [ROW] [COL] = {{1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1 },
                    {1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1 },
                    {1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1 },
                    {0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1 },
                    {1, 1, 1, 0, 1, 1, 1, 0, 1, 0 },
                    {1, 0, 1, 1, 1, 0, 1, 0, 0, 0 },
                    {1, 0, 0, 0, 0, 0, 0, 0, 0, 1 },
                    {1, 0, 1, 1, 1, 0, 1, 1, 1, 1 },
                    {1, 1, 0, 0, 0, 0, 1, 0, 0, 1 }};
```

Source = {0, 0};
Destination = {3, 4};

Output:

```
Shortest Path is 11
```

The idea is inspired from [Lee algorithm](#) and uses BFS.

1. We start from the source cell and calls BFS procedure.
2. We maintain a queue to store the coordinates of the matrix and initialize it with the source cell.

3. We also maintain a Boolean array visited of same size as our input matrix and initialize all its elements to false.
 - (a) We LOOP till queue is not empty
 - (b) Dequeue front cell from the queue
 - (c) Return if the destination coordinates have reached.
 - (d) For each of its four adjacent cells, if the value is 1 and they are not visited yet, we enqueue it in the queue and also mark them as visited.

Below is C++ implementation of the idea –

```

// C++ program to find the shortest path between
// a given source cell to a destination cell.
#include <bits/stdc++.h>
using namespace std;
#define ROW 9
#define COL 10

//to store matrix cell coordinates
struct Point
{
    int x;
    int y;
};

// An Data Structure for queue used in BFS
struct queueNode
{
    Point pt; // The coordinates of a cell
    int dist; // cell's distance of from the source
};

// check whether given cell (row, col) is a valid
// cell or not.
bool isValid(int row, int col)
{
    // return true if row number and column number
    // is in range
    return (row >= 0) && (row < ROW) &&
           (col >= 0) && (col < COL);
}

// These arrays are used to get row and column
// numbers of 4 neighbours of a given cell
int rowNum[] = {-1, 0, 0, 1};
int colNum[] = {0, -1, 1, 0};

```

```
// function to find the shortest path between
// a given source cell to a destination cell.
int BFS(int mat[][] [COL], Point src, Point dest)
{
    // check source and destination cell
    // of the matrix have value 1
    if (!mat[src.x] [src.y] || !mat[dest.x] [dest.y])
        return INT_MAX;

    bool visited[ROW] [COL];
    memset(visited, false, sizeof visited);

    // Mark the source cell as visited
    visited[src.x] [src.y] = true;

    // Create a queue for BFS
    queue<queueNode> q;

    // distance of source cell is 0
    queueNode s = {src, 0};
    q.push(s); // Enqueue source cell

    // Do a BFS starting from source cell
    while (!q.empty())
    {
        queueNode curr = q.front();
        Point pt = curr.pt;

        // If we have reached the destination cell,
        // we are done
        if (pt.x == dest.x && pt.y == dest.y)
            return curr.dist;

        // Otherwise dequeue the front cell in the queue
        // and enqueue its adjacent cells
        q.pop();

        for (int i = 0; i < 4; i++)
        {
            int row = pt.x + rowNum[i];
            int col = pt.y + colNum[i];

            // if adjacent cell is valid, has path and
            // not visited yet, enqueue it.
            if (isValid(row, col) && mat[row] [col] &&
                !visited[row] [col])
            {
                // mark cell as visited and enqueue it
                visited[row] [col] = true;
                queueNode newNode = {Point{row, col}, curr.dist + 1};
                q.push(newNode);
            }
        }
    }
}
```

```

        visited[row][col] = true;
        queueNode Adjcell = { {row, col},
                               curr.dist + 1 };
        q.push(Adjcell);
    }
}

//return -1 if destination cannot be reached
return INT_MAX;
}

// Driver program to test above function
int main()
{
    int mat[ROW][COL] =
    {
        { 1, 0, 1, 1, 1, 1, 0, 1, 1, 1 },
        { 1, 0, 1, 0, 1, 1, 1, 0, 1, 1 },
        { 1, 1, 1, 0, 1, 1, 0, 1, 0, 1 },
        { 0, 0, 0, 0, 1, 0, 0, 0, 0, 1 },
        { 1, 1, 1, 0, 1, 1, 1, 0, 1, 0 },
        { 1, 0, 1, 1, 1, 1, 0, 1, 0, 0 },
        { 1, 0, 0, 0, 0, 0, 0, 0, 0, 1 },
        { 1, 0, 1, 1, 1, 0, 1, 1, 1 },
        { 1, 1, 0, 0, 0, 1, 0, 0, 1 }
    };

    Point source = {0, 0};
    Point dest = {3, 4};

    int dist = BFS(mat, source, dest);

    if (dist != INT_MAX)
        cout << "Shortest Path is " << dist ;
    else
        cout << "Shortest Path doesn't exist";

    return 0;
}

```

Output :

Shortest Path is 11

This article is contributed by **Aditya Goel**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Improved By : [Chinmay Singh](#)

Source

<https://www.geeksforgeeks.org/shortest-path-in-a-binary-maze/>

Chapter 251

Shortest Path in a weighted Graph where weight of an edge is 1 or 2

Shortest Path in a weighted Graph where weight of an edge is 1 or 2 - GeeksforGeeks

Given a directed graph where every edge has weight as either 1 or 2, find the shortest path from a given source vertex ‘s’ to a given destination vertex ‘t’. Expected time complexity is $O(V+E)$.

A **Simple Solution** is to use [Dijkstra's shortest path algorithm](#), we can get a shortest path in $O(E + V\log V)$ time.

How to do it in $O(V+E)$ time? The idea is to use [BFS](#). One important observation about BFS is, the path used in BFS always has least number of edges between any two vertices. So if all edges are of same weight, we can use BFS to find the shortest path. For this problem, we can modify the graph and split all edges of weight 2 into two edges of weight 1 each. In the modified graph, we can use BFS to find the shortest path.

How many new intermediate vertices are needed? We need to add a new intermediate vertex for every source vertex. The reason is simple, if we add a intermediate vertex x between u and v and if we add same vertex between y and z, then new paths u to z and y to v are added to graph which might have note been there in original graph. Therefore in a graph with V vertices, we need V extra vertices.

Below is C++ implementation of above idea. In the below implementation $2*V$ vertices are created in a graph and for every edge (u, v) , we split it into two edges $(u, u+V)$ and $(u+V, v)$. This way we make sure that a different intermediate vertex is added for every source vertex.

C/C++

```
// Program to shortest path from a given source vertex 's' to
// a given destination vertex 't'. Expected time complexity
// is O(V+E).
```

```
#include<bits/stdc++.h>
using namespace std;

// This class represents a directed graph using adjacency
// list representation
class Graph
{
    int V;      // No. of vertices
    list<int> *adj;    // adjacency lists
public:
    Graph(int V); // Constructor
    void addEdge(int v, int w, int weight); // adds an edge

    // finds shortest path from source vertex 's' to
    // destination vertex 'd'.
    int findShortestPath(int s, int d);

    // print shortest path from a source vertex 's' to
    // destination vertex 'd'.
    int printShortestPath(int parent[], int s, int d);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[2*V];
}

void Graph::addEdge(int v, int w, int weight)
{
    // split all edges of weight 2 into two
    // edges of weight 1 each. The intermediate
    // vertex number is maximum vertex number + 1,
    // that is V.
    if (weight==2)
    {
        adj[v].push_back(v+V);
        adj[v+V].push_back(w);
    }
    else // Weight is 1
        adj[v].push_back(w); // Add w to v's list.
}

// To print the shortest path stored in parent[]
int Graph::printShortestPath(int parent[], int s, int d)
{
    static int level = 0;
```

```
// If we reached root of shortest path tree
if (parent[s] == -1)
{
    cout << "Shortest Path between " << s << " and "
        << d << " is " << s << " ";
    return level;
}

printShortestPath(parent, parent[s], d);

level++;
if (s < V)
    cout << s << " ";

return level;
}

// This function mainly does BFS and prints the
// shortest path from src to dest. It is assumed
// that weight of every edge is 1
int Graph::findShortestPath(int src, int dest)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[2*V];
    int *parent = new int[2*V];

    // Initialize parent[] and visited[]
    for (int i = 0; i < 2*V; i++)
    {
        visited[i] = false;
        parent[i] = -1;
    }

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[src] = true;
    queue.push_back(src);

    // 'i' will be used to get all adjacent vertices of a vertex
    list<int>::iterator i;

    while (!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        int s = queue.front();
```

```
if (s == dest)
    return printShortestPath(parent, s, dest);

queue.pop_front();

// Get all adjacent vertices of the dequeued vertex s
// If a adjacent has not been visited, then mark it
// visited and enqueue it
for (i = adj[s].begin(); i != adj[s].end(); ++i)
{
    if (!visited[*i])
    {
        visited[*i] = true;
        queue.push_back(*i);
        parent[*i] = s;
    }
}
}

// Driver program to test methods of graph class
int main()
{
    // Create a graph given in the above diagram
    int V = 4;
    Graph g(V);
    g.addEdge(0, 1, 2);
    g.addEdge(0, 2, 2);
    g.addEdge(1, 2, 1);
    g.addEdge(1, 3, 1);
    g.addEdge(2, 0, 1);
    g.addEdge(2, 3, 2);
    g.addEdge(3, 3, 2);

    int src = 0, dest = 3;
    cout << "\nShortest Distance between " << src
        << " and " << dest << " is "
        << g.findShortestPath(src, dest);

    return 0;
}
```

Python

```
''' Program to shortest path from a given source vertex s to
a given destination vertex t. Expected time complexity
is O(V+E)'''
from collections import defaultdict
```

```
#This class represents a directed graph using adjacency list representation
class Graph:

    def __init__(self,vertices):
        self.V = vertices #No. of vertices
        self.V_org = vertices
        self.graph = defaultdict(list) # default dictionary to store graph

    # function to add an edge to graph
    def addEdge(self,u,v,w):
        if w == 1:
            self.graph[u].append(v)
        else:
            '''split all edges of weight 2 into two
            edges of weight 1 each. The intermediate
            vertex number is maximum vertex number + 1,
            that is V.'''
            self.graph[u].append(self.V)
            self.graph[self.V].append(v)
            self.V = self.V + 1

    # To print the shortest path stored in parent[]
    def printPath(self, parent, j):
        Path_len = 1
        if parent[j] == -1 and j < self.V_org : #Base Case : If j is source
            print j,
            return 0 # when parent[-1] then path length = 0
        l = self.printPath(parent , parent[j])

        #increment path length
        Path_len = l + Path_len

        # print node only if its less than original node length.
        # i.e do not print any new node that has been added later
        if j < self.V_org :
            print j,

        return Path_len

    ''' This function mainly does BFS and prints the
        shortest path from src to dest. It is assumed
        that weight of every edge is 1'''
    def findShortestPath(self,src, dest):

        # Mark all the vertices as not visited
        # Initialize parent[] and visited[]
        visited =[False]*(self.V)
```

```
parent = [-1]*(self.V)

# Create a queue for BFS
queue=[]

# Mark the source node as visited and enqueue it
queue.append(src)
visited[src] = True

while queue :

    # Dequeue a vertex from queue
    s = queue.pop(0)

    # if s = dest then print the path and return
    if s == dest:
        return self.printPath(parent, s)

    # Get all adjacent vertices of the dequeued vertex s
    # If a adjacent has not been visited, then mark it
    # visited and enqueue it
    for i in self.graph[s]:
        if visited[i] == False:
            queue.append(i)
            visited[i] = True
            parent[i] = s

# Create a graph given in the above diagram
g = Graph(4)
g.addEdge(0, 1, 2)
g.addEdge(0, 2, 2)
g.addEdge(1, 2, 1)
g.addEdge(1, 3, 1)
g.addEdge(2, 0, 1)
g.addEdge(2, 3, 2)
g.addEdge(3, 3, 2)

src = 0; dest = 3
print ("Shortest Path between %d and %d is " %(src, dest)),
l = g.findShortestPath(src, dest)
print ("\nShortest Distance between %d and %d is %d " %(src, dest, l)),

#This code is contributed by Neelam Yadav
```

Output :

Shortest Path between 0 and 3 is 0 1 3

Shortest Distance between 0 and 3 is 3

How is this approach $O(V+E)$? In worst case, all edges are of weight 2 and we need to do $O(E)$ operations to split all edges and $2V$ vertices, so the time complexity becomes $O(E) + O(V+E)$ which is $O(V+E)$.

This article is contributed by **Aditya Goel**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

<https://www.geeksforgeeks.org/shortest-path-weighted-graph-weight-edge-1-2/>

Chapter 252

Shortest path in an unweighted graph

Shortest path in an unweighted graph - GeeksforGeeks

Given a unweighted graph, a source and a destination, we need to find shortest path from source to destination in the graph in most optimal way.

unweighted graph of 8 vertices

Input: source vertex = 0 and destination vertex is = 7.

Output: Shortest path length is:2

Path is::

0 3 7

Input: source vertex is = 2 and destination vertex is = 6.

Output: Shortest path length is:5

Path is::

2 1 0 3 4 6

One solution is to solve in $O(VE)$ time using [Bellman–Ford](#). If there are no negative weight cycles, then we can solve in $O(E + V\log V)$ time using [Dijkstra's algorithm](#).

Since the graph is unweighted, we can solve this problem in $O(V + E)$ time. The idea is to use a modified version of [Breadth-first search](#) in which we keep storing the predecessor of a given vertex while doing the breadth first search. This algorithm will work even when negative weight cycles are present in the graph.

We first initialize an array $\text{dist}[0, 1, \dots, v-1]$ such that $\text{dist}[i]$ stores the distance of vertex i from the source vertex and array $\text{pred}[0, 1, \dots, v-1]$ such that $\text{pred}[i]$ represents the immediate predecessor of the vertex i in the breadth first search starting from the source.

Now we get the length of the path from source to any other vertex in $O(1)$ time from array d , and for printing the path from source to any vertex we can use array p and that will take $O(V)$ time in worst case as V is the size of array P . So most of the time of algorithm is spent in doing the Breadth first search from given source which we know takes $O(V+E)$ time. Thus time complexity of our algorithm is $O(V+E)$.

Take the following unweighted graph as an example:

Following is the complete algorithm for finding the shortest path:

```

// CPP code for printing shortest path between
// two vertices of unweighted graph
#include <bits/stdc++.h>
using namespace std;

// utility function to form edge between two vertices
// source and dest
void add_edge(vector<int> adj[], int src, int dest)
{
    adj[src].push_back(dest);
    adj[dest].push_back(src);
}

// a modified version of BFS that stores predecessor
// of each vertex in array p
// and its distance from source in array d
bool BFS(vector<int> adj[], int src, int dest, int v,
          int pred[], int dist[])
{
    // a queue to maintain queue of vertices whose
    // adjacency list is to be scanned as per normal
    // DFS algorithm
    list<int> queue;

    // boolean array visited[] which stores the
    // information whether ith vertex is reached
    // at least once in the Breadth first search
    bool visited[v];

    // initially all vertices are unvisited
    // so v[i] for all i is false
    // and as no path is yet constructed
    // dist[i] for all i set to infinity
    for (int i = 0; i < v; i++) {
        visited[i] = false;
        dist[i] = INT_MAX;
        pred[i] = -1;
    }
}

```

```

// now source is first to be visited and
// distance from source to itself should be 0
visited[src] = true;
dist[src] = 0;
queue.push_back(src);

// standard BFS algorithm
while (!queue.empty()) {
    int u = queue.front();
    queue.pop_front();
    for (int i = 0; i < adj[u].size(); i++) {
        if (visited[adj[u][i]] == false) {
            visited[adj[u][i]] = true;
            dist[adj[u][i]] = dist[u] + 1;
            pred[adj[u][i]] = u;
            queue.push_back(adj[u][i]);

            // We stop BFS when we find
            // destination.
            if (adj[u][i] == dest)
                return true;
        }
    }
}

return false;
}

// utility function to print the shortest distance
// between source vertex and destination vertex
void printShortestDistance(vector<int> adj[], int s,
                           int dest, int v)
{
    // predecessor[i] array stores predecessor of
    // i and distance array stores distance of i
    // from s
    int pred[v], dist[v];

    if (BFS(adj, s, dest, v, pred, dist) == false)
    {
        cout << "Given source and destination"
        << " are not connected";
        return;
    }

    // vector path stores the shortest path
    vector<int> path;
    int crawl = dest;

```

```
path.push_back(crawl);
while (pred[crawl] != -1) {
    path.push_back(pred[crawl]);
    crawl = pred[crawl];
}

// distance from source is in distance array
cout << "Shortest path length is : "
     << dist[dest];

// printing path from source to destination
cout << "\nPath is::\n";
for (int i = path.size() - 1; i >= 0; i--)
    cout << path[i] << " ";
}

// Driver program to test above functions
int main()
{
    // no. of vertices
    int v = 8;

    // array of vectors is used to store the graph
    // in the form of an adjacency list
    vector<int> adj[v];

    // Creating graph given in the above diagram.
    // add_edge function takes adjacency list, source
    // and destination vertex as argument and forms
    // an edge between them.
    add_edge(adj, 0, 1);
    add_edge(adj, 0, 3);
    add_edge(adj, 1, 2);
    add_edge(adj, 3, 4);
    add_edge(adj, 3, 7);
    add_edge(adj, 4, 5);
    add_edge(adj, 4, 6);
    add_edge(adj, 4, 7);
    add_edge(adj, 5, 6);
    add_edge(adj, 6, 7);
    int source = 0, dest = 7;
    printShortestDistance(adj, source, dest, v);
    return 0;
}
```

Output:

```
Shortest path length is : 2
Path is:::
0 3 7
```

Time Complexity : $O(V + E)$
Auxiliary Space : $O(V)$

Source

<https://www.geeksforgeeks.org/shortest-path-unweighted-graph/>

Chapter 253

Shortest path to reach one prime to other by changing single digit at a time

Shortest path to reach one prime to other by changing single digit at a time - GeeksforGeeks

Given two four digit prime numbers, suppose 1033 and 8179, we need to find the shortest path from 1033 to 8179 by altering only single digit at a time such that every number that we get after changing a digit is prime. For example a solution is 1033, 1733, 3733, 3739, 3779, 8779, 8179

Examples:

```
Input : 1033 8179
Output :6
```

```
Input : 1373 8017
Output : 7
```

```
Input : 1033 1033
Output : 0
```

The question can be solved by [BFS](#) and it is a pretty interesting to solve as a starting problem for beginners. We first find out all 4 digit prime numbers till 9999 using technique of [Sieve of Eratosthenes](#). And then using those numbers formed the [graph using adjacency list](#). After forming the adjacency list, we used simple BFS to solve the problem.

```
// CPP program to reach a prime number from
// another by changing single digits and
// using only prime numbers.
```

```
#include <bits/stdc++.h>

using namespace std;

class graph {
    int V;
    list<int*>* l;
public:
    graph(int V)
    {
        this->V = V;
        l = new list<int>[V];
    }
    void addedge(int V1, int V2)
    {
        l[V1].push_back(V2);
        l[V2].push_back(V1);
    }
    int bfs(int in1, int in2);
};

// Finding all 4 digit prime numbers
void SieveOfEratosthenes(vector<int>& v)
{
    // Create a boolean array "prime[0..n]" and initialize
    // all entries it as true. A value in prime[i] will
    // finally be false if i is Not a prime, else true.
    int n = 9999;
    bool prime[n + 1];
    memset(prime, true, sizeof(prime));

    for (int p = 2; p * p <= n; p++) {

        // If prime[p] is not changed, then it is a prime
        if (prime[p] == true) {

            // Update all multiples of p
            for (int i = p * p; i <= n; i += p)
                prime[i] = false;
        }
    }

    // Forming a vector of prime numbers
    for (int p = 1000; p <= n; p++)
        if (prime[p])
            v.push_back(p);
}
```

```
// in1 and in2 are two vertices of graph which are
// actually indexes in pset[]
int graph::bfs(int in1, int in2)
{
    int visited[V];
    memset(visited, 0, sizeof(visited));
    queue<int> que;
    visited[in1] = 1;
    que.push(in1);
    list<int>::iterator i;
    while (!que.empty()) {
        int p = que.front();
        que.pop();
        for (i = l[p].begin(); i != l[p].end(); i++) {
            if (!visited[*i]) {
                visited[*i] = visited[p] + 1;
                que.push(*i);
            }
            if (*i == in2) {
                return visited[*i] - 1;
            }
        }
    }
}

// Returns true if num1 and num2 differ
// by single digit.
bool compare(int num1, int num2)
{
    // To compare the digits
    string s1 = to_string(num1);
    string s2 = to_string(num2);
    int c = 0;
    if (s1[0] != s2[0])
        c++;
    if (s1[1] != s2[1])
        c++;
    if (s1[2] != s2[2])
        c++;
    if (s1[3] != s2[3])
        c++;

    // If the numbers differ only by a single
    // digit return true else false
    return (c == 1);
}

int shortestPath(int num1, int num2)
```

```
{  
    // Generate all 4 digit  
    vector<int> pset;  
    SieveOfEratosthenes(pset);  
  
    // Create a graph where node numbers are indexes  
    // in pset[] and there is an edge between two  
    // nodes only if they differ by single digit.  
    graph g(pset.size());  
    for (int i = 0; i < pset.size(); i++)  
        for (int j = i + 1; j < pset.size(); j++)  
            if (compare(pset[i], pset[j]))  
                g.addedge(i, j);  
  
    // Since graph nodes represent indexes of numbers  
    // in pset[], we find indexes of num1 and num2.  
    int in1, in2;  
    for (int j = 0; j < pset.size(); j++)  
        if (pset[j] == num1)  
            in1 = j;  
    for (int j = 0; j < pset.size(); j++)  
        if (pset[j] == num2)  
            in2 = j;  
  
    return g.bfs(in1, in2);  
}  
  
// Driver code  
int main()  
{  
    int num1 = 1033, num2 = 8179;  
    cout << shortestPath(num1, num2);  
    return 0;  
}
```

Output :

6

Improved By : [goelankita2012](#), [iniesta](#)

Source

<https://www.geeksforgeeks.org/shortest-path-reach-one-prime-changing-single-digit-time/>

Chapter 254

Some interesting shortest path questions Set 1

Some interesting shortest path questions Set 1 - GeeksforGeeks

Question 1: *Given a directed weighted graph. You are also given the shortest path from a source vertex ‘s’ to a destination vertex ‘t’. If weight of every edge is increased by 10 units, does the shortest path remain same in the modified graph?*

The shortest path may change. The reason is, there may be different number of edges in different paths from s to t. For example, let shortest path be of weight 15 and has 5 edges. Let there be another path with 2 edges and total weight 25. The weight of the shortest path is increased by 5×10 and becomes $15 + 50$. Weight of the other path is increased by 2×10 and becomes $25 + 20$. So the shortest path changes to the other path with weight as 45.

Question 2: *This is similar to above question. Does the shortest path change when weights of all edges are multiplied by 10?*

If we multiply all edge weights by 10, the shortest path doesn’t change. The reason is simple, weights of all paths from s to t get multiplied by same amount. The number of edges on a path doesn’t matter. It is like changing unit of weights.

Question 3: *Given a directed graph where every edge has weight as either 1 or 2, find the shortest path from a given source vertex ‘s’ to a given destination vertex ‘t’. Expected time complexity is $O(V+E)$.*

If we apply [Dijkstra’s shortest path algorithm](#), we can get a shortest path in $O(E + V\log V)$ time. How to do it in $O(V+E)$ time? The idea is to use [BFS](#). One important observation about [BFS](#)s, the path used in BFS always has least number of edges between any two vertices. So if all edges are of same weight, we can use BFS to find the shortest path. For this problem, we can modify the graph and split all edges of weight 2 into two edges of weight 1 each. In the modified graph, we can use BFS to find the shortest path. How is this approach $O(V+E)$? In worst case, all edges are of weight 2 and we need to do $O(E)$ operations to split all edges, so the time complexity becomes $O(E) + O(V+E)$ which is $O(V+E)$.

Question 4: *Given a directed acyclic weighted graph, how to find the shortest path from a source s to a destination t in O(V+E) time?*

See: [Shortest Path in Directed Acyclic Graph](#)

More Questions See following links for more questions.

<http://algs4.cs.princeton.edu/44sp/>

<https://www.geeksforgeeks.org/algorithms-gq/graph-shortest-paths-gq/>

Source

<https://www.geeksforgeeks.org/interesting-shortest-path-questions-set-1/>

Chapter 255

Stepping Numbers

Stepping Numbers - GeeksforGeeks

Given two integers ‘n’ and ‘m’, find all the stepping numbers in range [n, m]. A number is called **stepping number** if all adjacent digits have an absolute difference of 1. 321 is a Stepping Number while 421 is not.

Examples :

```
Input : n = 0, m = 21
Output : 0 1 2 3 4 5 6 7 8 9 10 12 21
```

```
Input : n = 10, m = 15
Output : 10, 12
```

Method 1 : Brute force Approach

In this method, a brute force approach is used to iterate through all the integers from n to m and check if it's a stepping number.

C++

```
// A C++ program to find all the Stepping Number in [n, m]
#include<bits/stdc++.h>
using namespace std;

// This function checks if an integer n is a Stepping Number
bool isStepNum(int n)
{
    // Initialize prevDigit with -1
    int prevDigit = -1;

    // Iterate through all digits of n and compare difference
```

```
// between value of previous and current digits
while (n)
{
    // Get Current digit
    int curDigit = n % 10;

    // Single digit is consider as a
    // Stepping Number
    if (prevDigit == -1)
        prevDigit = curDigit;
    else
    {
        // Check if absolute difference between
        // prev digit and current digit is 1
        if (abs(prevDigit - curDigit) != 1)
            return false;
    }
    prevDigit = curDigit;
    n /= 10;
}

return true;
}

// A brute force approach based function to find all
// stepping numbers.
void displaySteppingNumbers(int n, int m)
{
    // Iterate through all the numbers from [N,M]
    // and check if it's a stepping number.
    for (int i=n; i<=m; i++)
        if (isStepNum(i))
            cout << i << " ";
}

// Driver program to test above function
int main()
{
    int n = 0, m = 21;

    // Display Stepping Numbers in
    // the range [n, m]
    displaySteppingNumbers(n, m);

    return 0;
}
```

Java

```
// A Java program to find all the Stepping Number in [n, m]
class Main
{
    // This Method checks if an integer n
    // is a Stepping Number
    public static boolean isStepNum(int n)
    {
        // Initialize prevDigit with -1
        int prevDigit = -1;

        // Iterate through all digits of n and compare
        // difference between value of previous and
        // current digits
        while (n > 0)
        {
            // Get Current digit
            int curDigit = n % 10;

            // Single digit is consider as a
            // Stepping Number
            if (prevDigit != -1)
            {
                // Check if absolute difference between
                // prev digit and current digit is 1
                if (Math.abs(curDigit-prevDigit) != 1)
                    return false;
            }
            n /= 10;
            prevDigit = curDigit;
        }
        return true;
    }

    // A brute force approach based function to find all
    // stepping numbers.
    public static void displaySteppingNumbers(int n,int m)
    {
        // Iterate through all the numbers from [N,M]
        // and check if it is a stepping number.
        for (int i = n; i <= m; i++)
            if (isStepNum(i))
                System.out.print(i+ " ");
    }

    // Driver code
    public static void main(String args[])
    {
        int n = 0, m = 21;
```

```
// Display Stepping Numbers in the range [n,m]
displaySteppingNumbers(n,m);
}
}
```

C#

```
// A C# program to find all
// the Stepping Number in [n, m]
using System;

class GFG
{
    // This Method checks if an
    // integer n is a Stepping Number
    public static bool isStepNum(int n)
    {
        // Initialize prevDigit with -1
        int prevDigit = -1;

        // Iterate through all digits
        // of n and compare difference
        // between value of previous
        // and current digits
        while (n > 0)
        {
            // Get Current digit
            int curDigit = n % 10;

            // Single digit is considered
            // as a Stepping Number
            if (prevDigit != -1)
            {
                // Check if absolute difference
                // between prev digit and current
                // digit is 1
                if (Math.Abs(curDigit -
                            prevDigit) != 1)
                    return false;
            }
            n /= 10;
            prevDigit = curDigit;
        }
        return true;
    }

    // A brute force approach based
```

```
// function to find all stepping numbers.
public static void displaySteppingNumbers(int n,
                                         int m)
{
    // Iterate through all the numbers
    // from [N,M] and check if it is
    // a stepping number.
    for (int i = n; i <= m; i++)
        if (isStepNum(i))
            Console.WriteLine(i + " ");
}

// Driver code
public static void Main()
{
    int n = 0, m = 21;

    // Display Stepping Numbers
    // in the range [n,m]
    displaySteppingNumbers(n, m);
}
}

// This code is contributed by nitin mittal.
```

Output :

```
0 1 2 3 4 5 6 7 8 9 10 12 21
```

Method 2: Using BFS/DFS

The idea is to use a [Breadth First Search/Depth First Search](#) traversal.

How to build the graph?

Every node in the graph represents a stepping number; there will be a directed edge from a node U to V if V can be transformed from U. (U and V are Stepping Numbers) A Stepping Number V can be transformed from U in following manner.

lastDigit refers to the last digit of U (i.e. $U \% 10$)
An adjacent number **V** can be:

- $U * 10 + \text{lastDigit} + 1$ (Neighbor A)
- $U * 10 + \text{lastDigit} - 1$ (Neighbor B)

By applying above operations a new digit is appended to U, it is either $\text{lastDigit}-1$ or $\text{lastDigit}+1$, so that the new number V formed from U is also a Stepping Number.
Therefore, every Node will have at most 2 neighboring Nodes.

Edge Cases: When the last digit of U is **0** or **9**

Case 1: lastDigit is 0 : In this case only digit ‘1’ can be appended.

Case 2: lastDigit is 9 : In this case only digit ‘8’ can be appended.

What will be the source/startng Node?

- Every single digit number is considered as a stepping Number, so bfs traversal for every digit will give all the stepping numbers starting from that digit.
- Do a bfs/dfs traversal for all the numbers from [0,9].

Note: For node 0, no need to explore neighbors during BFS traversal since it will lead to 01, 012, 010 and these will be covered by the BFS traversal starting from node 1.

Example to find all the stepping numbers from 0 to 21

-> 0 is a stepping Number and it is in the range
so display it.
-> 1 is a Stepping Number, find neighbors of 1 i.e.,
10 and 12 and push them into the queue

How to get 10 and 12?

Here U is 1 and last Digit is also 1
 $V = 10 + 0 = 10$ (Adding lastDigit - 1)
 $V = 10 + 2 = 12$ (Adding lastDigit + 1)

Then do the same for 10 and 12 this will result into
101, 123, 121 but these Numbers are out of range.

Now any number transformed from 10 and 12 will result
into a number greater than 21 so no need to explore
their neighbors.

-> 2 is a Stepping Number, find neighbors of 2 i.e.
21, 23.
-> 23 is out of range so it is not considered as a
Stepping Number (Or a neighbor of 2)

The other stepping numbers will be 3, 4, 5, 6, 7, 8, 9.

BFS based Solution:

C++

```
// A C++ program to find all the Stepping Number from N=n
// to m using BFS Approach
#include<bits/stdc++.h>
using namespace std;
```

```
// Prints all stepping numbers reachable from num
// and in range [n, m]
void bfs(int n, int m, int num)
{
    // Queue will contain all the stepping Numbers
    queue<int> q;

    q.push(num);

    while (!q.empty())
    {
        // Get the front element and pop from the queue
        int stepNum = q.front();
        q.pop();

        // If the Stepping Number is in the range
        // [n, m] then display
        if (stepNum <= m && stepNum >= n)
            cout << stepNum << " ";

        // If Stepping Number is 0 or greater than m,
        // need to explore the neighbors
        if (num == 0 || stepNum > m)
            continue;

        // Get the last digit of the currently visited
        // Stepping Number
        int lastDigit = stepNum % 10;

        // There can be 2 cases either digit to be
        // appended is lastDigit + 1 or lastDigit - 1
        int stepNumA = stepNum * 10 + (lastDigit - 1);
        int stepNumB = stepNum * 10 + (lastDigit + 1);

        // If lastDigit is 0 then only possible digit
        // after 0 can be 1 for a Stepping Number
        if (lastDigit == 0)
            q.push(stepNumB);

        //If lastDigit is 9 then only possible
        //digit after 9 can be 8 for a Stepping
        //Number
        else if (lastDigit == 9)
            q.push(stepNumA);

        else
        {
            q.push(stepNumA);
```

```
        q.push(stepNumB);
    }
}

// Prints all stepping numbers in range [n, m]
// using BFS.
void displaySteppingNumbers(int n, int m)
{
    // For every single digit Number 'i'
    // find all the Stepping Numbers
    // starting with i
    for (int i = 0 ; i <= 9 ; i++)
        bfs(n, m, i);
}

//Driver program to test above function
int main()
{
    int n = 0, m = 21;

    // Display Stepping Numbers in the
    // range [n,m]
    displaySteppingNumbers(n,m);

    return 0;
}
```

Java

```
// A Java program to find all the Stepping Number in
// range [n, m]
import java.util.*;

class Main
{
    // Prints all stepping numbers reachable from num
    // and in range [n, m]
    public static void bfs(int n,int m,int num)
    {
        // Queue will contain all the stepping Numbers
        Queue<Integer> q = new LinkedList<Integer> ();

        q.add(num);

        while (!q.isEmpty())
        {
            // Get the front element and pop from
```

```
// the queue
int stepNum = q.poll();

// If the Stepping Number is in
// the range [n,m] then display
if (stepNum <= m && stepNum >= n)
{
    System.out.print(stepNum + " ");
}

// If Stepping Number is 0 or greater
// then m ,need to explore the neighbors
if (stepNum == 0 || stepNum > m)
    continue;

// Get the last digit of the currently
// visited Stepping Number
int lastDigit = stepNum % 10;

// There can be 2 cases either digit
// to be appended is lastDigit + 1 or
// lastDigit - 1
int stepNumA = stepNum * 10 + (lastDigit- 1);
int stepNumB = stepNum * 10 + (lastDigit + 1);

// If lastDigit is 0 then only possible
// digit after 0 can be 1 for a Stepping
// Number
if (lastDigit == 0)
    q.add(stepNumB);

// If lastDigit is 9 then only possible
// digit after 9 can be 8 for a Stepping
// Number
else if (lastDigit == 9)
    q.add(stepNumA);

else
{
    q.add(stepNumA);
    q.add(stepNumB);
}
}

}

// Prints all stepping numbers in range [n, m]
// using BFS.
public static void displaySteppingNumbers(int n,int m)
```

```
{  
    // For every single digit Number 'i'  
    // find all the Stepping Numbers  
    // starting with i  
    for (int i = 0 ; i <= 9 ; i++)  
        bfs(n, m, i);  
}  
  
//Driver code  
public static void main(String args[])  
{  
    int n = 0, m = 21;  
  
    // Display Stepping Numbers in  
    // the range [n,m]  
    displaySteppingNumbers(n,m);  
}  
}
```

DFS based Solution

C++

```
// A C++ program to find all the Stepping Numbers  
// in range [n, m] using DFS Approach  
#include<bits/stdc++.h>  
using namespace std;  
  
// Prints all stepping numbers reachable from num  
// and in range [n, m]  
void dfs(int n, int m, int stepNum)  
{  
    // If Stepping Number is in the  
    // range [n,m] then display  
    if (stepNum <= m && stepNum >= n)  
        cout << stepNum << " ";  
  
    // If Stepping Number is 0 or greater  
    // than m, then return  
    if (stepNum == 0 || stepNum > m)  
        return ;  
  
    // Get the last digit of the currently  
    // visited Stepping Number  
    int lastDigit = stepNum % 10;  
  
    // There can be 2 cases either digit  
    // to be appended is lastDigit + 1 or
```

```
// lastDigit - 1
int stepNumA = stepNum*10 + (lastDigit-1);
int stepNumB = stepNum*10 + (lastDigit+1);

// If lastDigit is 0 then only possible
// digit after 0 can be 1 for a Stepping
// Number
if (lastDigit == 0)
    dfs(n, m, stepNumB);

// If lastDigit is 9 then only possible
// digit after 9 can be 8 for a Stepping
// Number
else if(lastDigit == 9)
    dfs(n, m, stepNumA);
else
{
    dfs(n, m, stepNumA);
    dfs(n, m, stepNumB);
}
}

// Method displays all the stepping
// numbers in range [n, m]
void displaySteppingNumbers(int n, int m)
{
    // For every single digit Number 'i'
    // find all the Stepping Numbers
    // starting with i
    for (int i = 0 ; i <= 9 ; i++)
        dfs(n, m, i);
}

//Driver program to test above function
int main()
{
    int n = 0, m = 21;

    // Display Stepping Numbers in
    // the range [n,m]
    displaySteppingNumbers(n,m);
    return 0;
}
```

Java

```
// A Java program to find all the Stepping Numbers
// in range [n, m] using DFS Approach
```

```
import java.util.*;  
  
class Main  
{  
    // Method display's all the stepping numbers  
    // in range [n, m]  
    public static void dfs(int n,int m,int stepNum)  
    {  
        // If Stepping Number is in the  
        // range [n,m] then display  
        if (stepNum <= m && stepNum >= n)  
            System.out.print(stepNum + " ");  
  
        // If Stepping Number is 0 or greater  
        // than m then return  
        if (stepNum == 0 || stepNum > m)  
            return ;  
  
        // Get the last digit of the currently  
        // visited Stepping Number  
        int lastDigit = stepNum % 10;  
  
        // There can be 2 cases either digit  
        // to be appended is lastDigit + 1 or  
        // lastDigit - 1  
        int stepNumA = stepNum*10 + (lastDigit-1);  
        int stepNumB = stepNum*10 + (lastDigit+1);  
  
        // If lastDigit is 0 then only possible  
        // digit after 0 can be 1 for a Stepping  
        // Number  
        if (lastDigit == 0)  
            dfs(n, m, stepNumB);  
  
        // If lastDigit is 9 then only possible  
        // digit after 9 can be 8 for a Stepping  
        // Number  
        else if(lastDigit == 9)  
            dfs(n, m, stepNumA);  
        else  
        {  
            dfs(n, m, stepNumA);  
            dfs(n, m, stepNumB);  
        }  
    }  
  
    // Prints all stepping numbers in range [n, m]  
    // using DFS.
```

```
public static void displaySteppingNumbers(int n, int m)
{
    // For every single digit Number 'i'
    // find all the Stepping Numbers
    // starting with i
    for (int i = 0 ; i <= 9 ; i++)
        dfs(n, m, i);
}

// Driver code
public static void main(String args[])
{
    int n = 0, m = 21;

    // Display Stepping Numbers in
    // the range [n,m]
    displaySteppingNumbers(n,m);
}
```

Output:

0 1 10 12 2 21 3 4 5 6 7 8 9

Improved By : nitin mittal

Source

<https://www.geeksforgeeks.org/stepping-numbers/>

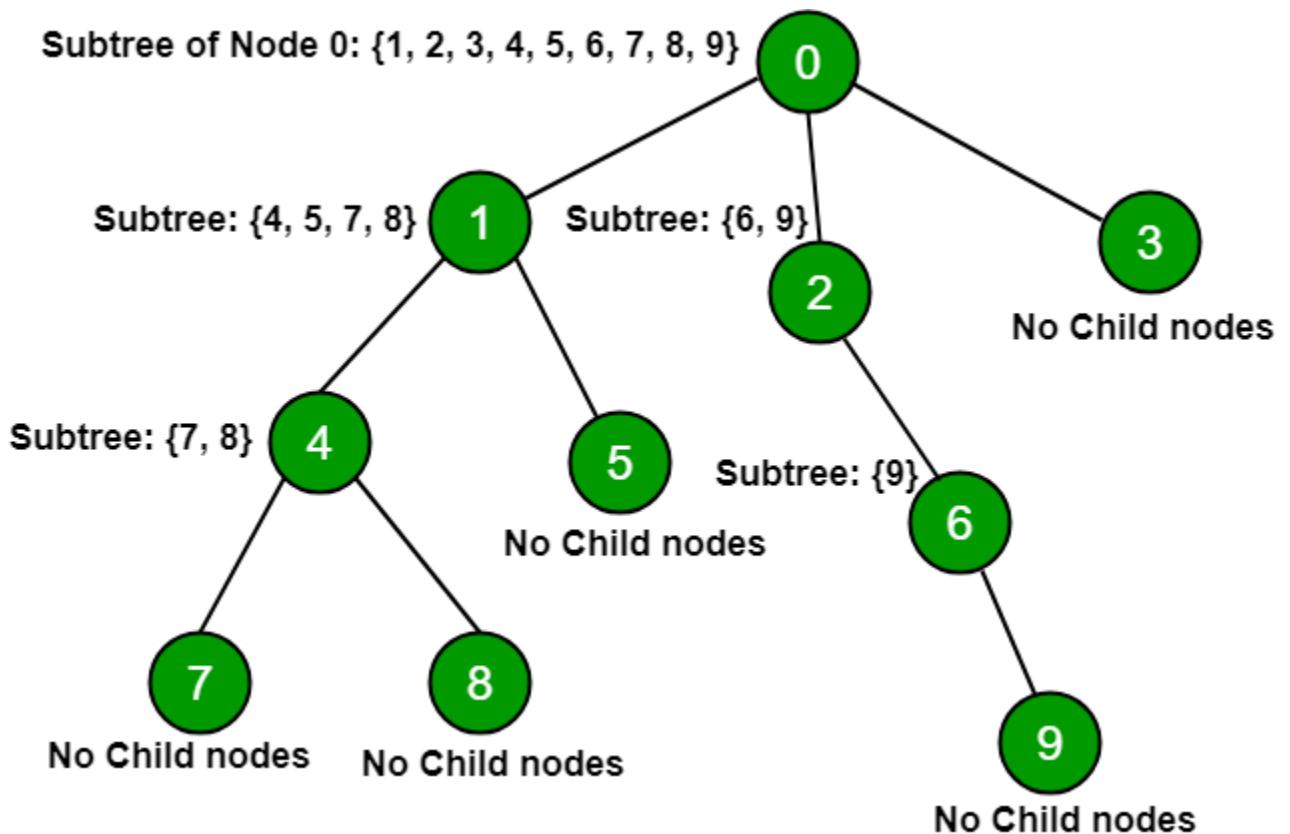
Chapter 256

Subtree of all nodes in a tree using DFS

Subtree of all nodes in a tree using DFS - GeeksforGeeks

Given n nodes of a tree and their connections, print Subtree nodes of every node.

Subtree of a node is defined as a tree which is a child of a node. The name emphasizes that everything which is a descendant of a tree node is a tree too, and is a subset of the larger tree.



Examples :

Input: N = 5

```

0 1
1 2
0 3
3 4

```

Output:

```

Subtree of node 0 is 1 2 3 4
Subtree of node 1 is 2
Subtree of node 3 is 4

```

Input: N = 7

```

0 1
1 2
2 3
0 4
4 5
4 6

```

Output:

```
Subtree of node 0 is 1 2 3 4 5 6
Subtree of node 1 is 2 3
Subtree of node 4 is 5 6
```

Approach: Do DFS traversal for every node and print all the nodes which are reachable from a particular node.

Explanation of below code:

1. When function dfs(0, 0) is called, start[0] = 0, dfs_order.push_back(0), visited[0] = 1 to keep track of dfs order.
2. Now, consider adjacency list (adj[100001]) as considering directional path elements connected to node 0 will be in adjacency list corresponding to node 0.
3. Now, recursively call dfs function till all elements traversed of adj[0].
4. Now, dfs(1, 2) is called, Now start[1] = 1, dfs_order.push_back(1), visited[1] = 1 after adj[1] elements is traversed.
5. Now adj [1] is traversed which contain only node 2 when adj[2] is traversed it contains no element, it will break and end[1]=2.
6. Similarly, all nodes traversed and store dfs_order in array to find subtree of nodes.

```
// C++ code to print subtree of all nodes
#include<bits/stdc++.h>
using namespace std;

// arrays for keeping position
// at each dfs traversal for each node
int start[100001];
int endd[100001];

// Storing dfs order
vector<int>dfs_order;
vector<int>adj[100001];
int visited[100001];

// Recursive function for dfs
// traversal dfsUtil()
void dfs(int a,int &b)
{
    // keep track of node visited
    visited[a]=1;
    b++;
    start[a]=b;
    dfs_order.push_back(a);

    for(int i=0;i<adj[a].size();i++)
        if(visited[adj[a][i]]==0)
            dfs(adj[a][i],b);
}
```

```

for(vector<int>:: iterator it=adj[a].begin();
     it!=adj[a].end();it++)
{
    if(!visited[*it])
    {
        dfs(*it,b);
    }
}
endd[a]=b;
}

// Function to print the subtree nodes
void Print(int n)
{
    for(int i=0;i<n;i++)
    {
        // if node is leaf node
        // start[i] is equals to endd[i]
        if(start[i]!=endd[i])
        {
            cout<<"subtree of node "<<i<<" is ";
            for(int j=start[i]+1;j<=endd[i];j++)
            {
                cout<<dfs_order[j-1]<<" ";
            }
            cout<<endl;
        }
    }
}

// Driver code
int main()
{
    // No of nodes n = 10
    int n =10, c = 0;

    adj[0].push_back(1);
    adj[0].push_back(2);
    adj[0].push_back(3);
    adj[1].push_back(4);
    adj[1].push_back(5);
    adj[4].push_back(7);
    adj[4].push_back(8);
    adj[2].push_back(6);
    adj[6].push_back(9);

    //Calling dfs for node 0
    //Considering root node at 0
}

```

```
dfs(0, c);

// Print child nodes
Print(n);

return 0;

}
```

Source

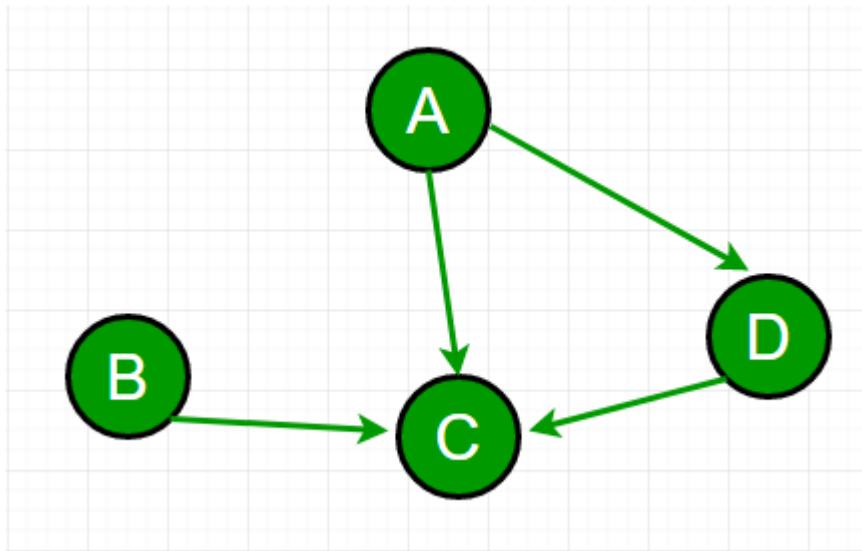
<https://www.geeksforgeeks.org/sub-tree-nodes-tree-using-dfs/>

Chapter 257

Sum of dependencies in a graph

Sum of dependencies in a graph - GeeksforGeeks

Given a directed and connected graph with n nodes. If there is an edge from u to v then u depends on v. Our task was to find out the sum of dependencies for every node.



Example:

For the graph in diagram,

A depends on C and D i.e. 2

B depends on C i.e. 1

D depends on C i.e. 1

And C depends on none.

Hence answer -> $0 + 1 + 1 + 2 = 4$

Asked in : [Flipkart Interview](#)

Idea is to check adjacency list and find how many edges are there from each vertex and

return the total number of edges.

C++

```
// C++ program to find the sum of dependencies
#include <bits/stdc++.h>
using namespace std;

// To add an edge
void addEdge(vector <int> adj[], int u,int v)
{
    adj[u].push_back(v);
}

// find the sum of all dependencies
int findSum(vector<int> adj[], int V)
{
    int sum = 0;

    // just find the size at each vector's index
    for (int u = 0; u < V; u++)
        sum += adj[u].size();

    return sum;
}

// Driver code
int main()
{
    int V = 4;
    vector<int >adj[V];
    addEdge(adj, 0, 2);
    addEdge(adj, 0, 3);
    addEdge(adj, 1, 3);
    addEdge(adj, 2, 3);

    cout << "Sum of dependencies is "
         << findSum(adj, V);
    return 0;
}
```

Java

```
// Java program to find the sum of dependencies

import java.util.Vector;

class Test
```

```
{  
    // To add an edge  
    static void addEdge(Vector <Integer> adj[], int u, int v)  
    {  
        adj[u].addElement((v));  
    }  
  
    // find the sum of all dependencies  
    static int findSum(Vector<Integer> adj[], int V)  
    {  
        int sum = 0;  
  
        // just find the size at each vector's index  
        for (int u = 0; u < V; u++)  
            sum += adj[u].size();  
  
        return sum;  
    }  
  
    // Driver method  
    public static void main(String[] args)  
    {  
        int V = 4;  
        Vector<Integer> adj[] = new Vector[V];  
  
        for (int i = 0; i < adj.length; i++) {  
            adj[i] = new Vector<>();  
        }  
  
        addEdge(adj, 0, 2);  
        addEdge(adj, 0, 3);  
        addEdge(adj, 1, 3);  
        addEdge(adj, 2, 3);  
  
        System.out.println("Sum of dependencies is " +  
                           findSum(adj, V));  
    }  
}  
// This code is contributed by Gaurav Miglani
```

Output:

```
Sum of dependencies is 4
```

Time complexity : O(V) where V is number of vertices in graph.

Source

<https://www.geeksforgeeks.org/sum-dependencies-graph/>

Chapter 258

Sum of the minimum elements in all connected components of an undirected graph

Sum of the minimum elements in all connected components of an undirected graph - Geeks-forGeeks

Given an array A of N numbers where A_i represent the value of the $(i+1)^{\text{th}}$ node. Also given are M pair of edges where u and v represent the nodes that are connected by an edge. The task is to find the sum of the minimum element in all the connected components of the given undirected graph. If a node has no connectivity to any other node, count it as a component with one node.

Examples:

Input: $a[] = \{1, 6, 2, 7, 3, 8, 4, 9, 5, 10\}$ $m = 5$

1 2
3 4
5 6
7 8
9 10

Output: 15

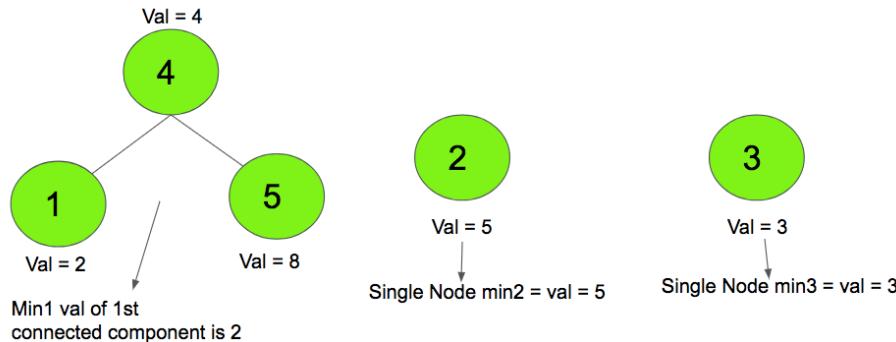
Connected components are: 1–2, 3–4, 5–6, 7–8 and 9–10

Sum of Minimum of all them : $1 + 2 + 3 + 4 + 5 = 15$

Input: $a[] = \{2, 5, 3, 4, 8\}$ $m = 2$

1 4
4 5

Output: 10



Sum of all minimum connected components is:
 $\text{Min1} + \text{Min2} + \text{Min3} = 2 + 5 + 3 = 10$

Approach: Finding connected components for an undirected graph is an easier task. Doing either a [BFS](#) or [DFS](#) starting from every unvisited vertex will give us our connected components. Create a `visited[]` array which has initially all nodes marked as False. Iterate all the nodes, if the node is not visited, call `DFS()` function so that all the nodes connected directly or indirectly to the node are marked as visited. While visiting all the directly or indirectly connected nodes, store the minimum value of all nodes. Create a variable `sum` which stores the summation of the minimum of all these connected components. Once all the nodes are visited, `sum` will have the answer to the problem.

Below is the implementation of the above approach:

```
// C++ program to find the sum
// of the minimum elements in all
// connected components of an undirected graph
#include <bits/stdc++.h>
using namespace std;
const int N = 10000;
vector<int> graph[N];

// Initially all nodes
// marked as unvisited
bool visited[N];

// DFS function that visits all
// connected nodes from a given node
void dfs(int node, int a[], int mini)
{
    // Stores the minimum
    mini = min(mini, a[node]);

    // Marks node as visited
    visited[node] = true;

    // Visits all children of current node
    for (int i : graph[node])
        if (!visited[i])
            dfs(i, a, mini);
}
```

```
// Traversed in all the connected nodes
for (int i : graph[node]) {
    if (!visited[i])
        dfs(i, a, mini);
}
}

// Function to add the edges
void addedge(int u, int v)
{
    graph[u - 1].push_back(v - 1);
    graph[v - 1].push_back(u - 1);
}

// Function that returns the sum of all minimums
// of connected components of graph
int minimumSumConnectedComponents(int a[], int n)
{
    // Initially sum is 0
    int sum = 0;

    // Traverse for all nodes
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            int mini = a[i];
            dfs(i, a, mini);
            sum += mini;
        }
    }
}

// Returns the answer
return sum;
}

// Driver Code
int main()
{
    int a[] = {1, 6, 2, 7, 3, 8, 4, 9, 5, 10};

    // Add edges
    addedge(1, 2);
    addedge(3, 4);
    addedge(5, 6);
    addedge(7, 8);
    addedge(9, 10);

    int n = sizeof(a) / sizeof(a[0]);
```

```
// Calling Function
cout << minimumSumConnectedComponents(a, n);
return 0;
}
```

Output:

15

Source

<https://www.geeksforgeeks.org/sum-of-the-minimum-elements-in-all-connected-components-of-an-undirected-graph/>

Chapter 259

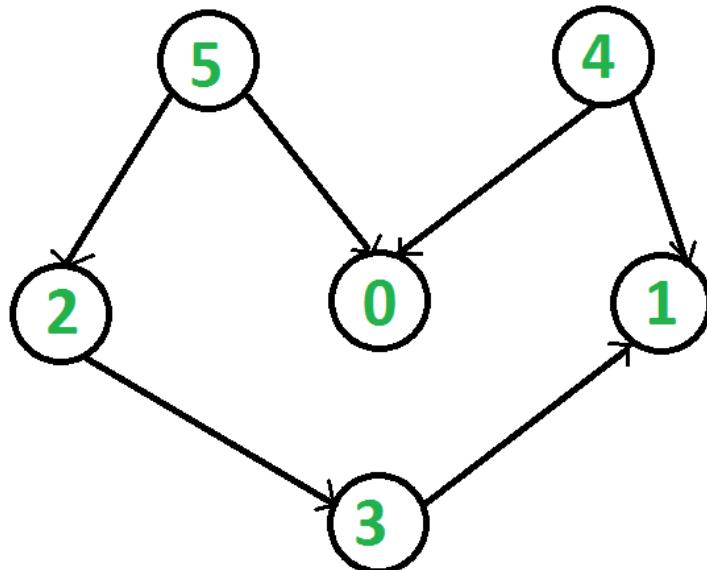
Topological Sort of a graph using departure time of vertex

Topological Sort of a graph using departure time of vertex - GeeksforGeeks

Given a Directed Acyclic Graph (DAG), find Topological Sort of the graph.

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv , vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, a topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 3 1 0”.



Please note that the first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges). For above graph, vertex 4 and 5 have no incoming edges.

We have already discussed a [DFS-based algorithm](#) using stack and [Kahn's Algorithm](#) for Topological Sorting. We have also discussed how to print all topological sorts of the DAG [here](#). In this post, another DFS based approach is discussed for finding Topological sort of a graph by introducing **concept of arrival and departure time of a vertex** in DFS.

What is Arrival Time & Departure Time of Vertices in DFS?

In DFS, **Arrival Time** is the time at which the vertex was explored for the first time and **Departure Time** is the time at which we have explored all the neighbors of the vertex and we are ready to backtrack.

How to find Topological Sort of a graph using departure time?

To find Topological Sort of a graph, we run [DFS](#) starting from all unvisited vertices one by one. For any vertex, before exploring any of its neighbors, we note the arrival time of that vertex and after exploring all the neighbors of the vertex, we note its departure time. Please note only departure time is needed to find Topological Sort of a graph, so we can skip arrival time of vertex. Finally, after we have visited all the vertices of the graph, we print the vertices in order of their decreasing departure time which is our desired Topological Order of Vertices.

Below is C++ implementation of above idea –

```
// A C++ program to print topological sorting of a DAG
#include<bits/stdc++.h>
using namespace std;

// Graph class represents a directed graph using adjacency
// list representation
class Graph
{
    int V; // No. of vertices
    // Pointer to an array containing adjacency lists
    list<int>* adj;
public:
    Graph(int); // Constructor
    ~Graph(); // Destructor

    // function to add an edge to graph
    void addEdge(int, int);

    // The function to do DFS traversal
    void DFS(int, vector<bool> &, vector<int> &, int &);

    // The function to do Topological Sort.
    void topologicalSort();
};
```

```
Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

Graph::~Graph()
{
    delete[] adj;
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);      // Add w to v's list.
}

// The function to do DFS() and stores departure time
// of all vertex
void Graph::DFS(int v, vector<bool> &visited,
                 vector<int> &departure, int &time)
{
    visited[v] = true;
    // time++;      // arrival time of vertex v

    for(int i : adj[v])
        if(!visited[i])
            DFS(i, visited, departure, time);

    // set departure time of vertex v
    departure[++time] = v;
}

// The function to do Topological Sort. It uses DFS().
void Graph::topologicalSort()
{
    // vector to store departure time of vertex.
    vector<int> departure(V, -1);

    // Mark all the vertices as not visited
    vector<bool> visited(V, false);
    int time = -1;

    // perform DFS on all unvisited vertices
    for(int i = 0; i < V; i++)
        if(!visited[i])
            DFS(i, visited, departure, time);
```

```
// Print vertices in topological order
for(int i = V - 1; i >= 0; i--)
    cout << departure[i] << " ";
}

// Driver program to test above functions
int main()
{
    // Create a graph given in the above diagram
    Graph g(6);
    g.addEdge(5, 2);
    g.addEdge(5, 0);
    g.addEdge(4, 0);
    g.addEdge(4, 1);
    g.addEdge(2, 3);
    g.addEdge(3, 1);

    cout << "Topological Sort of the given graph is \n";
    g.topologicalSort();

    return 0;
}
```

Output:

```
Topological Sort of the given graph is
5 4 2 3 1 0
```

Time Complexity of above solution is $O(V + E)$.

Source

<https://www.geeksforgeeks.org/topological-sorting-using-departure-time-of-vertex/>

Chapter 260

Total number of Spanning Trees in a Graph

Total number of Spanning Trees in a Graph - GeeksforGeeks

If a graph is a complete graph with n vertices, then total number of spanning trees is $n^{(n-2)}$ where n is the number of nodes in the graph. In complete graph, the task is equal to counting different labeled trees with n nodes for which have [Cayley's formula](#).

What if graph is not complete?

Follow the given procedure :-

STEP 1: Create Adjacency Matrix for the given graph.

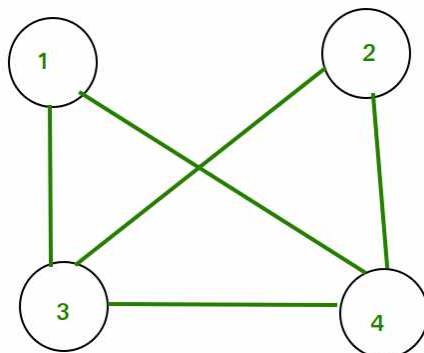
STEP 2: Replace all the diagonal elements with the degree of nodes. For eg. element at (1,1) position of adjacency matrix will be replaced by the degree of node 1, element at (2,2) position of adjacency matrix will be replaced by the degree of node 2, and so on.

STEP 3: Replace all non-diagonal 1's with -1.

STEP 4: Calculate co-factor for any element.

STEP 5: The cofactor that you get is the total number of spanning tree for that graph.

Consider the following graph:



Adjacency Matrix for the above graph will be as follows:

	1	2	3	4
1	0	0	1	1
2	0	0	1	1
3	1	1	0	1
4	1	1	1	0

After applying STEP 2 and STEP 3, adjacency matrix will look like

	1	2	3	4
1	2	0	-1	-1
2	0	2	-1	-1
3	-1	-1	3	-1
4	-1	-1	-1	3

The co-factor for (1, 1) is 8. Hence total no. of spanning tree that can be formed is 8.

NOTE- Co-factor for all the elements will be same. Hence we can compute co-factor for any element of the matrix.

This method is also known as [Kirchhoff's Theorem](#). It can be applied to complete graphs also.

Please refer below link for proof of above procedure.

https://en.wikipedia.org/wiki/Kirchhoff%27s_theorem#Proof_outline

Source

<https://www.geeksforgeeks.org/total-number-spanning-trees-graph/>

Chapter 261

Transitive Closure of a Graph using DFS

Transitive Closure of a Graph using DFS - GeeksforGeeks

Given a directed graph, find out if a vertex v is reachable from another vertex u for all vertex pairs (u, v) in the given graph. Here reachable mean that there is a path from vertex u to v. The reach-ability matrix is called transitive closure of a graph.

For example, consider below graph

Transitive closure of above graphs is

```
1 1 1 1  
1 1 1 1  
1 1 1 1  
0 0 0 1
```

We have discussed a $O(V^3)$ solution for this [here](#). The solution was based [Floyd Warshall Algorithm](#). In this post a $O(V^2)$ algorithm for the same is discussed.

Below are abstract steps of algorithm.

1. Create a matrix $tc[V][V]$ that would finally have transitive closure of given graph. Initialize all entries of $tc[][]$ as 0.
2. Call DFS for every node of graph to mark reachable vertices in $tc[][]$. In recursive calls to DFS, we don't call DFS for an adjacent vertex if it is already marked as reachable in $tc[][]$.

Below is implementation of the above idea. The code uses adjacency list representation of input graph and builds a matrix $tc[V][V]$ such that $tc[u][v]$ would be true if v is reachable from u.

C/C++

```

// C++ program to print transitive closure of a graph
#include<bits/stdc++.h>
using namespace std;

class Graph
{
    int V; // No. of vertices
    bool **tc; // To store transitive closure
    list<int> *adj; // array of adjacency lists
    void DFSUtil(int u, int v);

public:
    Graph(int V); // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w) { adj[v].push_back(w); }

    // prints transitive closure matrix
    void transitiveClosure();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];

    tc = new bool* [V];
    for (int i=0; i<V; i++)
    {
        tc[i] = new bool[V];
        memset(tc[i], false, V*sizeof(bool));
    }
}

// A recursive DFS traversal function that finds
// all reachable vertices for s.
void Graph::DFSUtil(int s, int v)
{
    // Mark reachability from s to t as true.
    tc[s][v] = true;

    // Find all the vertices reachable through v
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (tc[s][*i] == false)
            DFSUtil(s, *i);
}

// The function to find transitive closure. It uses

```

```
// recursive DFSUtil()
void Graph::transitiveClosure()
{
    // Call the recursive helper function to print DFS
    // traversal starting from all vertices one by one
    for (int i = 0; i < V; i++)
        DFSUtil(i, i); // Every vertex is reachable from self.

    for (int i=0; i<V; i++)
    {
        for (int j=0; j<V; j++)
            cout << tc[i][j] << " ";
        cout << endl;
    }
}

// Driver code
int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Transitive closure matrix is \n";
    g.transitiveClosure();

    return 0;
}
```

Java

```
// JAVA program to print transitive
// closure of a graph.

import java.util.ArrayList;
import java.util.Arrays;

// A directed graph using
// adjacency list representation
public class Graph {

    // No. of vertices in graph
    private int vertices;
```

```
// adjacency list
private ArrayList<Integer>[] adjList;

// To store transitive closure
private int[][] tc;

// Constructor
public Graph(int vertices) {

    // initialise vertex count
    this.vertices = vertices;
    this.tc = new int[this.vertices][this.vertices];

    // initialise adjacency list
    initAdjList();
}

// utility method to initialise adjacency list
@SuppressWarnings("unchecked")
private void initAdjList() {

    adjList = new ArrayList[vertices];
    for (int i = 0; i < vertices; i++) {
        adjList[i] = new ArrayList<>();
    }
}

// add edge from u to v
public void addEdge(int u, int v) {

    // Add v to u's list.
    adjList[u].add(v);
}

// The function to find transitive
// closure. It uses
// recursive DFSUtil()
public void transitiveClosure() {

    // Call the recursive helper
    // function to print DFS
    // traversal starting from all
    // vertices one by one
    for (int i = 0; i < vertices; i++) {
        dfsUtil(i, i);
    }
}
```

```
for (int i = 0; i < vertices; i++) {
    System.out.println(Arrays.toString(tc[i]));
}
}

// A recursive DFS traversal
// function that finds
// all reachable vertices for s
private void dfsUtil(int s, int v) {

    // Mark reachability from
    // s to v as true.
    tc[s][v] = 1;

    // Find all the vertices reachable
    // through v
    for (int adj : adjList[v]) {
        if (tc[s][adj]==0) {
            dfsUtil(s, adj);
        }
    }
}

// Driver Code
public static void main(String[] args) {

    // Create a graph given
    // in the above diagram
    Graph g = new Graph(4);

    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);
    System.out.println("Transitive closure " +
                       "matrix is");

    g.transitiveClosure();

}

}

// This code is contributed
// by Himanshu Shekhar
```

Python

```
# Python program to print transitive closure of a graph
from collections import defaultdict

# This class represents a directed graph using adjacency
# list representation
class Graph:

    def __init__(self,vertices):
        # No. of vertices
        self.V= vertices

        # default dictionary to store graph
        self.graph= defaultdict(list)

        # To store transitive closure
        self.tc = [[0 for j in range(self.V)] for i in range(self.V)]

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # A recursive DFS traversal function that finds
    # all reachable vertices for s
    def DFSUtil(self,s,v):

        # Mark reachability from s to v as true.
        self.tc[s][v] = 1

        # Find all the vertices reachable through v
        for i in self.graph[v]:
            if self.tc[s][i]==0:
                self.DFSUtil(s,i)

    # The function to find transitive closure. It uses
    # recursive DFSUtil()
    def transitiveClosure(self):

        # Call the recursive helper function to print DFS
        # traversal starting from all vertices one by one
        for i in range(self.V):
            self.DFSUtil(i, i)
        print self.tc

    # Create a graph given in the above diagram
g = Graph(4)
g.addEdge(0, 1)
```

```
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print "Transitive closure matrix is"
g.transitiveClosure();

# This code is contributed by Neelam Yadav
```

Output:

```
Transitive closure matrix is
1 1 1 1
1 1 1 1
1 1 1 1
0 0 0 1
```

References:

<http://www.cs.princeton.edu/courses/archive/spr03/cs226/lectures/digraph.4up.pdf>

This article is contributed by **Aditya Goel**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

<https://www.geeksforgeeks.org/transitive-closure-of-a-graph-using-dfs/>

Chapter 262

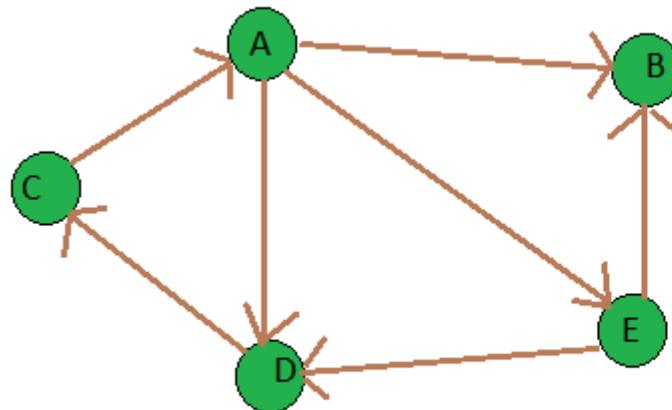
Transpose graph

Transpose graph - GeeksforGeeks

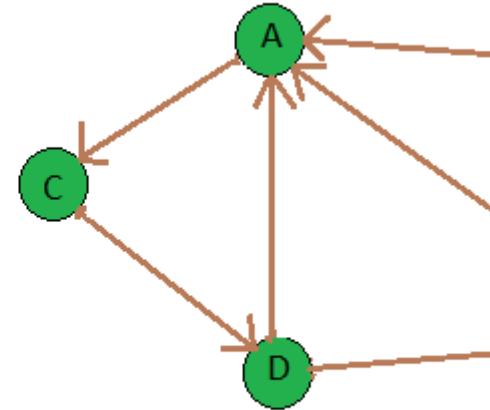
[Transpose](#) of a directed graph G is another directed graph on the same set of vertices with all of the edges reversed compared to the orientation of the corresponding edges in G. That is, if G contains an edge (u, v) then the converse/transpose/reverse of G contains an edge (v, u) and vice versa.

Given a [graph \(represented as adjacency list\)](#), we need to find another graph which is the transpose of the given graph.

Example:



(i)



(ii)

Transpose Graph

Input : figure (i) is the input graph.

Output : figure (ii) is the transpose graph of the given graph.

We traverse the adjacency list and as we find a vertex v in the adjacency list of vertex u which indicates an edge from u to v in main graph, we just add an edge from v to u in the transpose graph i.e. add u in the adjacency list of vertex v of the new graph. Thus traversing lists of all vertices of main graph we can get the transpose graph. Thus the total time complexity of the algorithm is $O(V+E)$ where V is number of vertices of graph and E is the number of edges of the graph.

Note : It is simple to get the transpose of a graph which is stored in adjacency matrix format, you just need to get the [transpose](#) of that matrix.

```
// CPP program to find transpose of a graph.
#include <bits/stdc++.h>
using namespace std;

// function to add an edge from vertex source to vertex dest
void addEdge(vector<int> adj[], int src, int dest)
{
    adj[src].push_back(dest);
}

// function to print adjacency list of a graph
void displayGraph(vector<int> adj[], int v)
{
    for (int i = 0; i < v; i++) {
        cout << i << "--> ";
        for (int j = 0; j < adj[i].size(); j++)
            cout << adj[i][j] << " ";
        cout << "\n";
    }
}

// function to get Transpose of a graph taking adjacency
// list of given graph and that of Transpose graph
void transposeGraph(vector<int> adj[],
                    vector<int> transpose[], int v)
{
    // traverse the adjacency list of given graph and
    // for each edge (u, v) add an edge (v, u) in the
    // transpose graph's adjacency list
    for (int i = 0; i < v; i++)
        for (int j = 0; j < adj[i].size(); j++)
            addEdge(transpose, adj[i][j], i);
}

int main()
{
    int v = 5;
```

```
vector<int> adj[v];
addEdge(adj, 0, 1);
addEdge(adj, 0, 4);
addEdge(adj, 0, 3);
addEdge(adj, 2, 0);
addEdge(adj, 3, 2);
addEdge(adj, 4, 1);
addEdge(adj, 4, 3);

// Finding transpose of graph represented
// by adjacency list adj[]
vector<int> transpose[v];
transposeGraph(adj, transpose, v);

// displaying adjacency list of transpose
// graph i.e. b
displayGraph(transpose, v);

return 0;
}
```

Output:

```
0--> 2
1--> 0  4
2--> 3
3--> 0  4
4--> 0
```

Source

<https://www.geeksforgeeks.orgtranspose-graph/>

Chapter 263

Traveling Salesman Problem (TSP) Implementation

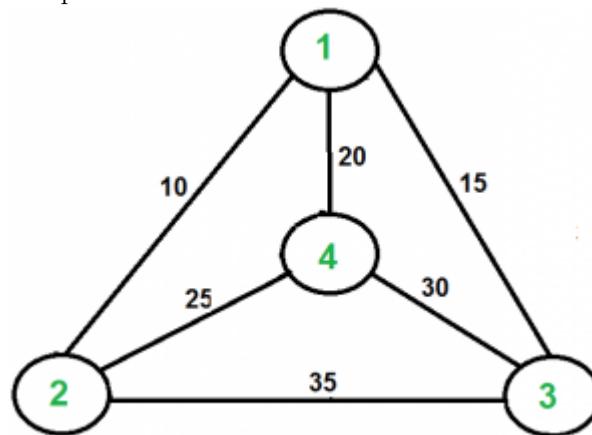
Traveling Salesman Problem (TSP) Implementation - GeeksforGeeks

[Travelling Salesman Problem \(TSP\)](#): Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns back to the starting point.

Note the difference between [Hamiltonian Cycle](#) and TSP. The Hamiltonian cycle problem is to find if there exist a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.

For example, consider the graph shown in figure on right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is $10+25+30+15$ which is 80.

The problem is a famous NP hard problem. There is no polynomial time known solution for this problem.



Examples:

Output of Given Graph:

```
minimum weight Hamiltonian Cycle :  
10 + 20 + 30 + 15 := 80
```

In this post, implementation of simple solution is discussed.

1. Consider city 1 as the starting and ending point. Since route is cyclic, we can consider any point as starting point.
2. Generate all $(n-1)!$ permutations of cities.
3. Calculate cost of every permutation and keep track of minimum cost permutation.
4. Return the permutation with minimum cost.

Below c++ implementation of above idea

```
// CPP program to implement traveling salesman  
// problem using naive approach.  
#include <bits/stdc++.h>  
using namespace std;  
#define V 4  
  
// implementation of traveling Salesman Problem  
int travllingSalesmanProblem(int graph[][] , int s)  
{  
    // store all vertex apart from source vertex  
    vector<int> vertex;  
    for (int i = 0; i < V; i++)  
        if (i != s)  
            vertex.push_back(i);  
  
    // store minimum weight Hamiltonian Cycle.  
    int min_path = INT_MAX;  
    do {  
  
        // store current Path weight(cost)  
        int current_pathweight = 0;  
  
        // compute current path weight  
        int k = s;  
        for (int i = 0; i < vertex.size(); i++) {  
            current_pathweight += graph[k][vertex[i]];  
            k = vertex[i];  
        }  
        current_pathweight += graph[k][s];  
  
        // update minimum
```

```
min_path = min(min_path, current_pathweight);

} while (next_permutation(vertex.begin(), vertex.end()));

return min_path;
}

// driver program to test above function
int main()
{
    // matrix representation of graph
    int graph[][][V] = { { 0, 10, 15, 20 },
                        { 10, 0, 35, 25 },
                        { 15, 35, 0, 30 },
                        { 20, 25, 30, 0 } };
    int s = 0;
    cout << travllingSalesmanProblem(graph, s) << endl;
    return 0;
}
```

Output:

80

Source

<https://www.geeksforgeeks.org/traveling-salesman-problem-tsp-implementation/>

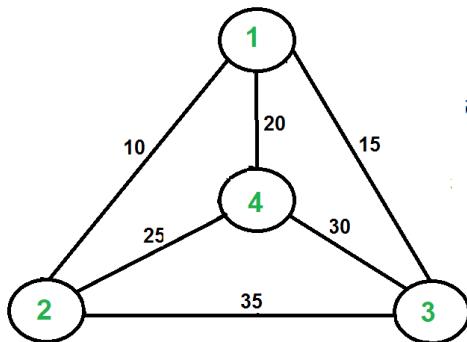
Chapter 264

Travelling Salesman Problem Set 1 (Naive and Dynamic Programming)

Travelling Salesman Problem Set 1 (Naive and Dynamic Programming) - GeeksforGeeks

Travelling Salesman Problem (TSP): Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

Note the difference between [Hamiltonian Cycle](#) and TSP. The Hamiltonian cycle problem is to find if there exist a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.



For example, consider the graph shown in figure on right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is $10+25+30+15$ which is 80.

The problem is a famous [NP hard](#) problem. There is no polynomial time known solution for this problem.

Following are different solutions for the traveling salesman problem.

Naive Solution:

- 1) Consider city 1 as the starting and ending point.
- 2) Generate all $(n-1)!$ Permutations of cities.
- 3) Calculate cost of every permutation and keep track of minimum cost permutation.
- 4) Return the permutation with minimum cost.

Time Complexity: $\Theta(n!)$

Dynamic Programming:

Let the given set of vertices be $\{1, 2, 3, 4, \dots, n\}$. Let us consider 1 as starting and ending point of output. For every other vertex i (other than 1), we find the minimum cost path with 1 as the starting point, i as the ending point and all vertices appearing exactly once. Let the cost of this path be $\text{cost}(i)$, the cost of corresponding Cycle would be $\text{cost}(i) + \text{dist}(i, 1)$ where $\text{dist}(i, 1)$ is the distance from i to 1. Finally, we return the minimum of all $[\text{cost}(i) + \text{dist}(i, 1)]$ values. This looks simple so far. Now the question is how to get $\text{cost}(i)$?

To calculate $\text{cost}(i)$ using Dynamic Programming, we need to have some recursive relation in terms of sub-problems. Let us define a term $C(S, i)$ be the cost of the minimum cost path visiting each vertex in set S exactly once, starting at 1 and ending at i .

We start with all subsets of size 2 and calculate $C(S, i)$ for all subsets where S is the subset, then we calculate $C(S, i)$ for all subsets S of size 3 and so on. Note that 1 must be present in every subset.

```
If size of S is 2, then S must be {1, i},  
C(S, i) = dist(1, i)  
Else if size of S is greater than 2.  
C(S, i) = min { C(S-{i}, j) + dis(j, i)} where j belongs to S, j != i and j != 1.
```

For a set of size n , we consider $n-2$ subsets each of size $n-1$ such that all subsets don't have nth in them.

Using the above recurrence relation, we can write dynamic programming based solution. There are at most $O(n^2 \cdot 2^n)$ subproblems, and each one takes linear time to solve. The total running time is therefore $O(n^2 \cdot 2^n)$. The time complexity is much less than $O(n!)$, but still exponential. Space required is also exponential. So this approach is also infeasible even for slightly higher number of vertices.

We will soon be discussing approximate algorithms for travelling salesman problem.

Next Article: [Traveling Salesman Problem Set 2](#)

References:

<http://www.lsi.upc.edu/~mjserna/docencia/algofib/P07/dynprog.pdf>
<http://www.cs.berkeley.edu/~vazirani/algorithms/chap6.pdf>

Source

<https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/>

Chapter 265

Travelling Salesman Problem Set 2 (Approximate using MST)

Travelling Salesman Problem Set 2 (Approximate using MST) - GeeksforGeeks

We introduced [Travelling Salesman Problem](#) and discussed Naive and Dynamic Programming Solutions for the problem in the [previous post](#). Both of the solutions are infeasible. In fact, there is no polynomial time solution available for this problem as the problem is a known NP-Hard problem. There are approximate algorithms to solve the problem though. The approximate algorithms work only if the problem instance satisfies Triangle-Inequality.

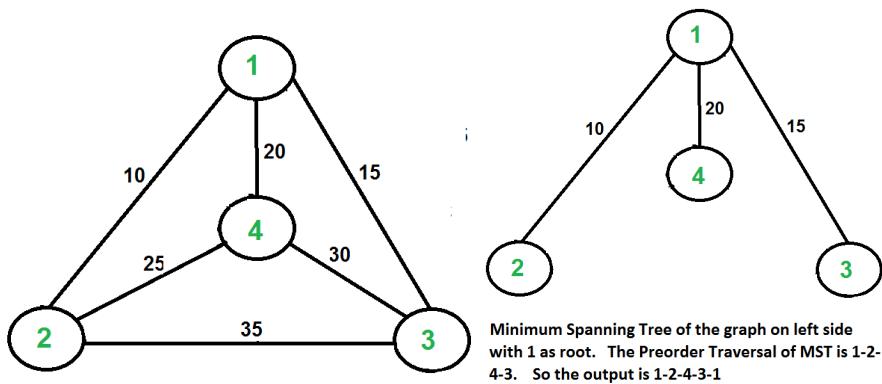
Triangle-Inequality: The least distant path to reach a vertex j from i is always to reach j directly from i , rather than through some other vertex k (or vertices), i.e., $\text{dis}(i, j)$ is always less than or equal to $\text{dis}(i, k) + \text{dist}(k, j)$. The Triangle-Inequality holds in many practical situations.

When the cost function satisfies the triangle inequality, we can design an approximate algorithm for TSP that returns a tour whose cost is never more than twice the cost of an optimal tour. The idea is to use [Minimum Spanning Tree \(MST\)](#). Following is the MST based algorithm.

Algorithm:

- 1) Let 1 be the starting and ending point for salesman.
- 2) Construct MST from with 1 as root using [Prim's Algorithm](#).
- 3) List vertices visited in preorder walk of the constructed MST and add 1 at the end.

Let us consider the following example. The first diagram is the given graph. The second diagram shows MST constructed with 1 as root. The preorder traversal of MST is 1-2-4-3. Adding 1 at the end gives 1-2-4-3-1 which is the output of this algorithm.



In this case, the approximate algorithm produces the optimal tour, but it may not produce optimal tour in all cases.

How is this algorithm 2-approximate? The cost of the output produced by the above algorithm is never more than twice the cost of best possible output. Let us see how is this guaranteed by the above algorithm.

Let us define a term **full walk** to understand this. A full walk is lists all vertices when they are first visited in preorder, it also list vertices when they are returned after a subtree is visited in preorder. The full walk of above tree would be 1-2-1-4-1-3-1.

Following are some important facts that prove the 2-approximateneess.

- 1) The cost of best possible Travelling Salesman tour is never less than the cost of MST. (The definition of **MST**says, it is a minimum cost tree that connects all vertices).
- 2) The total cost of full walk is at most twice the cost of MST (Every edge of MST is visited at-most twice)
- 3) The output of the above algorithm is less than the cost of full walk. In above algorithm, we print preorder walk as output. In preorder walk, two or more edges of full walk are replaced with a single edge. For example, 2-1 and 1-4 are replaced by 1 edge 2-4. So if the graph follows triangle inequality, then this is always true.

From the above three statements, we can conclude that the cost of output produced by the approximate algorithm is never more than twice the cost of best possible solution.

We have discussed a very simple 2-approximate algorithm for the travelling salesman problem. There are other better approximate algorithms for the problem. For example **Christofides algorithm** is 1.5 approximate algorithm. We will soon be discussing these algorithms as separate posts.

References:

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

<http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/AproxAlgor/TSP/tsp.htm>

Improved By : [PranamLashkari](#)

Source

<https://www.geeksforgeeks.org/travelling-salesman-problem-set-2-approximate-using-mst/>

Chapter 266

Two Clique Problem (Check if Graph can be divided in two Cliques)

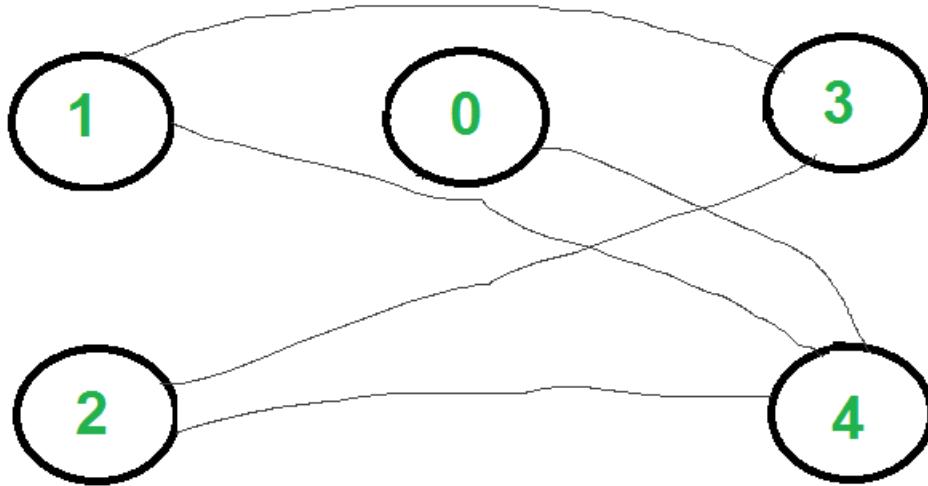
Two Clique Problem (Check if Graph can be divided in two Cliques) - GeeksforGeeks

A Clique is a subgraph of graph such that all vertices in subgraph are completely connected with each other. Given a Graph, find if it can be divided into two Cliques.

```
Input : G[][] = {{0, 1, 1, 0, 0},  
                 {1, 0, 1, 1, 0},  
                 {1, 1, 0, 0, 0},  
                 {0, 1, 0, 0, 1},  
                 {0, 0, 0, 1, 0}};  
Output : Yes
```

This problem looks tricky at first, but has a simple and interesting solution. A graph can be divided in two cliques if its complement graph is [Bipartite](#). So below are two steps to find if graph can be divided in two Cliques or not.

1. Find complement of Graph. Below is complement graph of the above shown graph. In complement, all original edges are removed. And the vertices which did not have an edge between them, now have an edge connecting them.



Complement of above Graph

2. Return true if complement is Bipartite, else false. Above shown graph is Bipartite. Checking whether a Graph is Bipartite or no is discussed [here](#).

How does this work?

If complement is Bipartite, then graph can be divided into two sets U and V such that there is no edge connecting to vertices of same set. This means in original graph, these sets U and V are completely connected. Hence original graph could be divided in two Cliques.

Implementation :

Below is C++ implementation of above steps.

```
// C++ program to find out whether a given graph can be
// converted to two Cliques or not.
#include <bits/stdc++.h>
using namespace std;

const int V = 5;

// This function returns true if subgraph reachable from
// src is Bipartite or not.
bool isBipartiteUtil(int G[][V], int src, int colorArr[])
{
    colorArr[src] = 1;

    // Create a queue (FIFO) of vertex numbers and enqueue
    // source vertex for BFS traversal
    queue<int> q;
    q.push(src);
```

```
// Run while there are vertices in queue (Similar to BFS)
while (!q.empty())
{
    // Dequeue a vertex from queue
    int u = q.front();
    q.pop();

    // Find all non-colored adjacent vertices
    for (int v = 0; v < V; ++v)
    {
        // An edge from u to v exists and destination
        // v is not colored
        if (G[u][v] && colorArr[v] == -1)
        {
            // Assign alternate color to this adjacent
            // v of u
            colorArr[v] = 1 - colorArr[u];
            q.push(v);
        }

        // An edge from u to v exists and destination
        // v is colored with same color as u
        else if (G[u][v] && colorArr[v] == colorArr[u])
            return false;
    }
}

// If we reach here, then all adjacent vertices can
// be colored with alternate color
return true;
}

// Returns true if a Graph G[][] is Bipartite or not. Note
// that G may not be connected.
bool isBipartite(int G[] [V])
{
    // Create a color array to store colors assigned
    // to all veritces. Vertex number is used as index in
    // this array. The value '-1' of colorArr[i]
    // is used to indicate that no color is assigned to
    // vertex 'i'. The value 1 is used to indicate first
    // color is assigned and value 0 indicates
    // second color is assigned.
    int colorArr[V];
    for (int i = 0; i < V; ++i)
        colorArr[i] = -1;

    // One by one check all not yet colored vertices.
```

```
for (int i = 0; i < V; i++)
    if (colorArr[i] == -1)
        if (isBipartiteUtil(G, i, colorArr) == false)
            return false;

    return true;
}

// Returns true if G can be divided into
// two Cliques, else false.
bool canBeDividedinTwoCliques(int G[][])
{
    // Find complement of G[] []
    // All values are complemented except
    // diagonal ones
    int GC[V][V];
    for (int i=0; i<V; i++)
        for (int j=0; j<V; j++)
            GC[i][j] = (i != j)? !G[i][j] : 0;

    // Return true if complement is Bipartite
    // else false.
    return isBipartite(GC);
}

// Driver program to test above function
int main()
{
    int G[][][V] = {{0, 1, 1, 1, 0},
                    {1, 0, 1, 0, 0},
                    {1, 1, 0, 0, 0},
                    {0, 1, 0, 0, 1},
                    {0, 0, 0, 1, 0}};
};

canBeDividedinTwoCliques(G) ? cout << "Yes" :
                                cout << "No";
return 0;
}
```

Output :

Yes

Time complexity of above implementation is $O(V^2)$.

Reference:

<http://math.stackexchange.com/questions/310092/the-two-clique-problem-is-in-p-or-np-p-np-for-hypothesis>

Source

<https://www.geeksforgeeks.org/two-clique-problem-check-graph-can-divided-two-cliques/>

Chapter 267

Undirected graph splitting and its application for number pairs

Undirected graph splitting and its application for number pairs - GeeksforGeeks

Graphs can be used for seemingly unconnected problems. Say the problem of cards which have numbers on both side and you try to create a continuous sequence of numbers using one side. This problem leads to problems in how to split a graph into connected subsets with cycles and connected subsets without cycles.

There are known algorithms to detect if the graph contains a circle. Here we modify it to find all nodes not-connected to circles. We add a Boolean array cyclic initiated as all false, meaning that no node is known to be connected to a circle.

Then we loop through nodes applied a usual algorithm to determine a circle skipping on nodes that are already known to be connected to a circle. Whenever we find a new node connected to a circle we propagate the property to all connected nodes skipping already designated as connected again.

Main function is a modification of DFS algorithm from article “[Detect cycles in undirected graph](#)“:

All nodes connected to a cycle are designated as such in a linear time.

The algorithm was applied to the following problem from the recent [challenge “Germanium”](#), which was solved by about 3.5% of participants.

Suppose we have N cards ($100000 \geq N \geq 1$) and on both sides of the cards there is a number (N are useless and can be discarded. In the same spirit, every card (n, m) : $n \neq N$ allows us to put n upwards and ensured its presence in the sequence, while m is useless anyway.

If we have a card (n, n) : $n < N$, it ensures that n can always be added to a continuous sequence that arrived to $n-1$.

If we have two identical cards (n, m) both n, m can be always added to a sequence if needed, just to put it on the opposite side.

Now we encode cards into a graph in the following way. The graph nodes are supposed to be numbered from 0 to $N-1$. Every node has a set of edges leading to it represented as a vector, initiated as an empty and overall set of edges is a vector of vectors of integers – edges

(N).

(n, m) : n, m N – discarded.

(n, m): n N – add to the graph edge (n-1, n-1)

Now it is easy to understand that every cycle in the graph can be used for all nodes od the graph. A sample: (1, 2) (2, 5), (5, 1) – every number appears on two cards, just pass along the cycle and put any number once up and another time down:

(1, 2) – 1 up, 2 down

(2, 5) – 2 up, 5 down

(5, 1) – 5 up, 1 down

If any number already put on the upper side, every subsequent card that has the number must be out this number down, so another number can be shown on the upper side. In the graph, it means that any number connected by an edge to a number of cycles is free to be shown. The same is true for a card connected to the card connected to the cycle. So, any number/node that connected somehow to any cycle cannot provide any problem. Pairs of identical cards like (1, 2), (1, 2) also produce a small circle in the graph, the same is true for double cards, like 3, 3 it is a cycle of a node connected to itself.

All parts of the graph that are not connected to any cycle can be split in not connected smaller graphs. It is more or less obvious that any such fragment has a problematic number – the biggest in the fragment. We use a similar but much easier algorithm to find such fragments and all maximums in fragments. The least of it is Q – the smallest number that cannot be the end of the continuous coverage from the least numbers, the Q above.

In addition to the cyclic array, we add another array: visitedNonCyclic, meaning that this number already met while passing through a non-cycle fragment.

The maximum in the non-cycle fragment is extracted by recurrent calls to a function:

Please refer findMax() in below code. The graph creation is done in solution() of below code.

After the graph is created the solution just calls to isCyclic and then to:

```
// the main crux of the solution, see full code below
graph.isCyclic();
int res = 1 + graph.GetAnswer();
```

The solution, which was inspired by the rather hard recent “[Germanium 2018 Codility Challenge](#)” and brought the gold medal.

Main Idea:

1. Numbers are big, but we demand a continuous sequence from 1 to the last number, but there are no more than 100 000 numbers. So, the biggest possible is really 100 000, take it as the first hypothesis
2. Due to the same considerations, the biggest number cannot be bigger than N (amount of numbers)
3. We can also pass overall numbers and take the biggest which is NN. The result cannot be bigger than N or NN
4. Taking this into account we can replace all cards like (x, 200 000) into (x, x). Both allow setting x upwards, x is not a problem thus.
5. Cards with both numbers bigger that N or NN just ignored or disposed
6. If there are cards with the same number on both sides it means that the number is always OK, cannot be the smallest problematic, so never the answer.

7. If there are two identical cards like (x, y) and (y, x) it means that both x and y are not problematic and cannot be the answer
8. Now we introduce the main BIG idea that we translate the set of cards into a graph. Vertices of the graph are numbered from 0 to $M-1$, where M is the last possible number (least of N , NN). Every node has some adjacent nodes, so the set of all edges is a vector of vectors of integers
8. Every card like (x, y) with both numbers less or equal to M into edge $x-1, y-1$, so the first node gets an additional adjacent node and vice versa
9. Some vortices will be connected to themselves due to cards like $(3, 3)$. It is a loop case of size 1.
10. Some vortices will be connected by two edges or more if there are identical cards. It is a loop of size 2.
11. Loops of sizes 1 and 2 have numbers that cannot be of any problem. But now we realize that the same is the case of a cycle of any length. For example, cards are 2, 3; 3, 4; 4, 10; 10, 2; All their numbers have no problem, just put 2, 3, 4, 10 to the upper side. The downside will have 3, 4, 10, 2.
12. Now if we have a card x, y and we know that x is guaranteed already in another place, we can put this card x down, y up and ensure that the y is in the resulting sequence.
12. So, if any card is connected to a cycle by an edge, it cannot present a problem since the cycle is all OK, so this number is also OK.
13. Thus if there is an interconnected subset of nodes of a graph which contains a cycle, all vortices cannot present any problem.
So, we can apply one of the known algorithms to find loops like in the article "[Detect cycle in an undirected graph](#)" with an addition of propagation
14. We can also designate all nodes in the graph that are connected to a cycle, `cycled[]`. And we can propagate the property
Just start with some in a cycle, set it as a cycled, then pass over it's adjacent skipping the cycled and calling the same function on the adjacent.
15. Using a combination of the known algorithm to detect cycles in an undirected graph with skipping on `cycled` already and propagating the property "connected to a cycle", we find all nodes connected to cycles. All such nodes are safe, they cannot be a problem
16. But remain nodes not connected to any cycle, their problem can be simplified but cutting into separated graphs which have no common edges or nodes.
17. But what to do with them? It can be a linear branch or branches that cross one another. All nodes are different.
18. With some last effort, one can understand that any such set has only one problematic number – maximum of the set. It can be proven, paying attention that crosses only makes thing better, but the maximum stays still.
19. So we cut all non-cycled into connected separated entities and find the maximum in every one.
20. Then we find the minimum of them and this is the final answer!

Examples:

Input : A = [1, 2, 4, 3] B = [1, 3, 2, 3]

Output : 5.

Because the cards as they are provide 1, 2, 3, 4 but not 5.

Input : A = [4, 2, 1, 6, 5] B = [3, 2, 1, 7, 7],

Output: 4.

Because you can show 3 or 4 by the first card but not both 3 and 4. So, put 3, while 1 and 2 are by the following cards.

Input : A = [2, 3], B = [2, 3]

Output : 1. Because 1 is missing at all.

Complexity:

- expected worst-case time complexity is $O(N)$;
- expected worst-case space complexity is $O(N)$;

The final implementation

```
#include <algorithm>
#include <vector>
using namespace std;

class Graph {
private:
    int V; // No. of vertices
    vector<vector<int> > edges; // edges grouped by nodes
    bool isCyclicUtil(int v, vector<bool>& visited, int parent);
    vector<bool> cyclic;
    vector<int> maxInNonCyclicFragments;
    int findMax(int v);
    vector<bool> visitedNonCyclic;
    void setPropagateCycle(int v);

public:
    Graph(int V); // Constructor
    void addEdge(int v, int w); // to add an edge to graph

    // returns true if there is a cycle and
    // also designates all connected to a cycle
    bool isCyclic();
    int getSize() const { return V; }
    int GetAnswer();
};

Graph::Graph(int V)
{
    this->V = V;
    edges = vector<vector<int> >(V, vector<int>());
    visitedNonCyclic = cyclic = vector<bool>(V, false);
}
```

```
void Graph::addEdge(int v, int w)
{
    edges[v].push_back(w);
    edges[w].push_back(v);
}

void Graph::setPropagateCycle(int v)
{
    if (cyclic[v])
        return;
    cyclic[v] = true;
    for (auto i = edges[v].begin(); i != edges[v].end(); ++i) {
        setPropagateCycle(*i);
    }
}

bool Graph::isCyclicUtil(int v, vector<bool>& visited, int parent)
{
    if (cyclic[v])
        return true;

    // Mark the current node as visited
    visited[v] = true;

    // Recur for all the vertices edgesacent to this vertex
    vector<int>::iterator i;
    for (i = edges[v].begin(); i != edges[v].end(); ++i) {

        // If an edgesacent is not visited, then
        // recur for that edgesacent
        if (!visited[*i]) {
            if (isCyclicUtil(*i, visited, v)) {
                setPropagateCycle(v);
                return true;
            }
        }

        // If an edgesacent is visited and not parent
        // of current vertex, then there is a cycle.
        else if (*i != parent) {
            setPropagateCycle(v);
            return true;
        }
        if (cyclic[*i]) {
            setPropagateCycle(v);
            return true;
        }
    }
}
```

```
    return false;
}

bool Graph::isCyclic()
{
    // Mark all the vortices as not visited
    // and not part of recursion stack
    vector<bool> visited(V, false);

    // Call the recursive helper function
    // to detect cycle in different DFS trees
    bool res = false;
    for (int u = 0; u < V; u++)

        // Don't recur for u if it is already visited{
        if (!visited[u] && !cyclic[u]) {
            if (isCyclicUtil(u, visited, -1)) {
                res = true;

                // there was retur true originally
                visited = vector<bool>(V, false);
            }
        }
    return res;
}

int Graph::findMax(int v)
{
    if (cyclic[v])
        return -1;
    if (visitedNonCyclic.at(v))
        return -1;
    int res = v;
    visitedNonCyclic.at(v) = true;
    for (auto& u2 : edges.at(v)) {
        res = max(res, findMax(u2));
    }
    return res;
}

int Graph::GetAnswer()
{
    // cannot be less than, after extract must add 1
    int res = V;

    for (int u = 0; u < V; u++) {
        maxInNonCyclicFragments.push_back(findMax(u));
    }
}
```

```
for (auto& u : maxInNonCyclicFragments) {
    if (u >= 0)
        res = min(res, u);
}
return res;
}

int solution(vector<int>& A, vector<int>& B)
{
    const int N = (int)A.size();

    const int MAX_AMOUNT = 100001;
    vector<bool> present(MAX_AMOUNT, false);

    for (auto& au : A) {
        if (au <= N) {
            present.at(au) = true;
        }
    }
    for (auto& au : B) {
        if (au <= N) {
            present.at(au) = true;
        }
    }
    int MAX_POSSIBLE = N;
    for (int i = 1; i <= N; i++) {
        if (false == present.at(i)) {
            MAX_POSSIBLE = i - 1;
            break;
        }
    }
}

Graph graph(MAX_POSSIBLE);

for (int i = 0; i < N; i++) {
    if (A.at(i) > MAX_POSSIBLE && B.at(i) > MAX_POSSIBLE) {
        continue;
    }
    int mi = min(A.at(i), B.at(i));
    int ma = max(A.at(i), B.at(i));
    if (A.at(i) > MAX_POSSIBLE || B.at(i) > MAX_POSSIBLE) {
        graph.addEdge(mi - 1, mi - 1);
    }
    else {
        graph.addEdge(mi - 1, ma - 1);
    }
}
graph.isCyclic();
```

```
int res = 1 + graph.GetAnswer();
return res;
}
// Test and driver
#include <iostream>
void test(vector<int>& A, vector<int>& B, int expected,
          bool printAll = false)
{
    int res = solution(A, B);
    if (expected != res || printAll) {
        for (size_t i = 0; i < A.size(); i++) {
            cout << A.at(i) << " ";
        }
        cout << endl;
        for (size_t i = 0; i < B.size(); i++) {
            cout << B.at(i) << " ";
        }
        cout << endl;
        if (expected != res)
            cout << "Error! Expected: " << expected << " ";
        else
            cout << "Expected: " << expected << " ";
    }
    cout << " Result: " << res << endl;
}
int main()
{
    vector<int> VA;
    vector<int> VB;

    int A4[] = { 1, 1, 1, 1, 1 };
    int B4[] = { 2, 3, 4, 5, 6 };
    VA = vector<int>(A4, A4 + 1);
    VB = vector<int>(B4, B4 + 1);
    test(VA, VB, 2, true);

    int A0[] = { 1, 1 };
    int B0[] = { 2, 2 };
    VA = vector<int>(A0, A0 + 2);
    VB = vector<int>(B0, B0 + 2);
    test(VA, VB, 3);

    int A[] = { 1, 2, 4, 3 };
    int B[] = { 1, 3, 2, 3 };
    VA = vector<int>(A, A + 4);
    VB = vector<int>(B, B + 4);
    test(VA, VB, 5);
```

```
int A2[] = { 4, 2, 1, 6, 5 };
int B2[] = { 3, 2, 1, 7, 7 };
VA = vector<int>(A2, A2 + 5);
VB = vector<int>(B2, B2 + 5);
test(VA, VB, 4);

int A3[] = { 2, 3 };
int B3[] = { 2, 3 };
VA = vector<int>(A3, A3 + 2);
VB = vector<int>(B3, B3 + 2);
test(VA, VB, 1);
return 0;
}
```

Output:

```
1
2
Expected: 2    Result: 2
Result: 3
Result: 5
Result: 4
Result: 1
```

Source

<https://www.geeksforgeeks.org/undirected-graph-splitting-and-its-application-for-number-pairs/>

Chapter 268

Union-Find Algorithm (Union By Rank and Find by Optimized Path Compression)

Union-Find Algorithm (Union By Rank and Find by Optimized Path Compression) - Geeks-forGeeks

Check whether a given graph contains a cycle or not.

Example 1:

Input:

Output: Graph contains Cycle.

Example 2:

Input:

Output: Graph does not contain Cycle.

Prerequisites: [Disjoint Set \(Or Union-Find\)](#), [Union By Rank and Path Compression](#)

We have already discussed [union-find to detect cycle](#). Here we discuss find by path compression, where it is slightly modified to work faster than the original method as we are skipping one level each time we are going up the graph. Implementation of find function is iterative, so there is no overhead involved. Time complexity of optimized find function is $O(\log^*(n))$, i.e iterated logarithm, which converges to $O(1)$ for repeated calls.

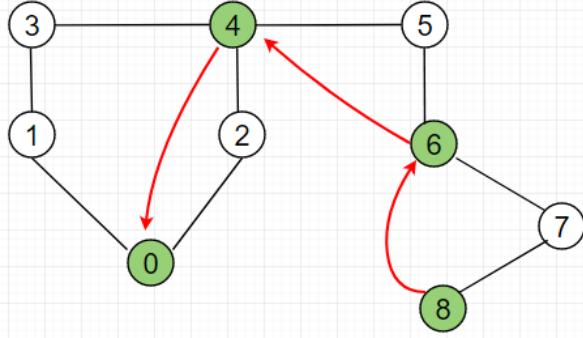
Refer this link for

[Proof of \$\log^*\(n\)\$ complexity of Union-Find](#)

Explanation of find function:

Take Example 1 to understand find function:

(1) call find(8) for **first time** and mappings will be done like this:



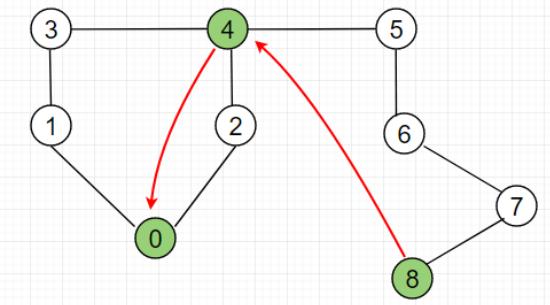
It took 3 mappings for find function to get the root of node 8. Mappings are illustrated below:

From node 8, skipped node 7, Reached node 6.

From node 6, skipped node 5, Reached node 4.

From node 4, skipped node 2, Reached node 0.

(2) call find(8) for **second time** and mappings will be done like this:

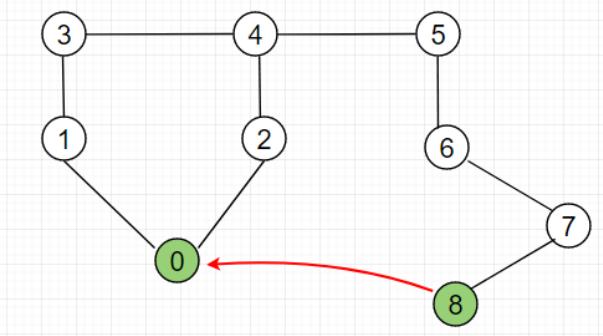


It took 2 mappings for find function to get the root of node 8. Mappings are illustrated below:

From node 8, skipped node 5, node 6 and node 7, Reached node 4.

From node 4, skipped node 2, Reached node 0.

(3) call find(8) for **third time** and mappings will be done like this:



Finally, we see it took only 1 mapping for find function to get the root of node 8. Mappings are illustrated below:

From node 8, skipped node 5, node 6, node 7, node 4, and node 2, Reached node 0.
That is how it converges path from certain mappings to single mapping.

Explanation of example 1:

Initially array size and Arr look like:

Arr[9] = {0, 1, 2, 3, 4, 5, 6, 7, 8}
size[9] = {1, 1, 1, 1, 1, 1, 1, 1, 1}

Consider the edges in the graph, and add them one by one to the disjoint-union set as follows:

Edge 1: 0-1

find(0)=>0, find(1)=>1, both have different root parent

Put these in single connected component as currently they doesn't belong to different connected components.

Arr[1]=0, size[0]=2;

Edge 2: 0-2

find(0)=>0, find(2)=>2, both have different root parent

Arr[2]=0, size[0]=3;

Edge 3: 1-3

find(1)=>0, find(3)=>3, both have different root parent

Arr[3]=0, size[0]=3;

Edge 4: 3-4

find(3)=>1, find(4)=>4, both have different root parent

Arr[4]=0, size[0]=4;

Edge 5: 2-4

find(2)=>0, find(4)=>0, both have same root parent

Hence, There is a cycle in graph.

We stop further checking for cycle in graph.

```
// CPP progxrm to implement Union-Find with union
// by rank and path compression.
#include <bits/stdc++.h>
using namespace std;

const int MAX_VERTEX = 101;

// Arr to represent parent of index i
int Arr[MAX_VERTEX];

// Size to represent the number of nodes
// in subgxrph rooted at index i
int size[MAX_VERTEX];

// set parent of every node to itself and
```

```
// size of node to one
void initialize(int n)
{
    for (int i = 0; i <= n; i++) {
        Arr[i] = i;
        size[i] = 1;
    }
}

// Each time we follow a path, find function
// compresses it further until the path length
// is greater than or equal to 1.
int find(int i)
{
    // while we reach a node whose parent is
    // equal to itself
    while (Arr[i] != i)
    {
        Arr[i] = Arr[Arr[i]]; // Skip one level
        i = Arr[i]; // Move to the new level
    }
    return i;
}

// A function that does union of two nodes x and y
// where xr is root node of x and yr is root node of y
void _union(int xr, int yr)
{
    if (size[xr] < size[yr]) // Make yr parent of xr
    {
        Arr[xr] = Arr[yr];
        size[yr] += size[xr];
    }
    else // Make xr parent of yr
    {
        Arr[yr] = Arr[xr];
        size[xr] += size[yr];
    }
}

// The main function to check whether a given
// gxrph contains cycle or not
int isCycle(vector<int> adj[], int V)
{
    // Iterte through all edges of gxrph, find
    // nodes connecting them.
    // If root nodes of both are same, then there is
    // cycle in gxrph.
```

```
for (int i = 0; i < V; i++) {
    for (int j = 0; j < adj[i].size(); j++) {
        int x = find(i); // find root of i
        int y = find(adj[i][j]); // find root of adj[i][j]

        if (x == y)
            return 1; // If same parent
        _union(x, y); // Make them connect
    }
}
return 0;
}

// Driver program to test above functions
int main()
{
    int V = 3;

    // Initialize the values for array Arr and Size
    initialize(V);

    /* Let us create following graph
       0
       |
       |
       |   \
       |   \
       1---2 */
    vector<int> adj[V]; // Adjacency list for graph

    adj[0].push_back(1);
    adj[0].push_back(2);
    adj[1].push_back(2);

    // call is_cycle to check if it contains cycle
    if (isCycle(adj, V))
        cout << "Graph contains Cycle.\n";
    else
        cout << "Graph does not contain Cycle.\n";

    return 0;
}
```

Output:

Graph contains Cycle.

Time Complexity(Find) : O(log*(n))

Time Complexity(union) : O(1)

Source

<https://www.geeksforgeeks.org/union-find-algorithm-union-by-rank-and-find-by-optimized-path-compression/>

Chapter 269

Union-Find Algorithm Set 2 (Union By Rank and Path Compression)

Union-Find Algorithm Set 2 (Union By Rank and Path Compression) - GeeksforGeeks

In the [previous post](#), we introduced *union find algorithm* and used it to detect cycle in a graph. We used following *union()* and *find()* operations for subsets.

```
// Naive implementation of find
int find(int parent[], int i)
{
    if (parent[i] == -1)
        return i;
    return find(parent, parent[i]);
}

// Naive implementation of union()
void Union(int parent[], int x, int y)
{
    int xset = find(parent, x);
    int yset = find(parent, y);
    parent[xset] = yset;
}
```

The above *union()* and *find()* are naive and the worst case time complexity is linear. The trees created to represent subsets can be skewed and can become like a linked list. Following is an example worst case scenario.

Let there be 4 elements 0, 1, 2, 3

Initially, all elements are single element subsets.
0 1 2 3

```
Do Union(0, 1)
    1   2   3
    /
0
```

```
Do Union(1, 2)
    2   3
    /
1
/
0
```

```
Do Union(2, 3)
    3
    /
2
/
1
/
0
```

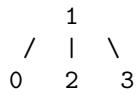
The above operations can be optimized to $O(\log n)$ in worst case. The idea is to always attach smaller depth tree under the root of the deeper tree. This technique is called **union by rank**. The term *rank* is preferred instead of height because if path compression technique (we have discussed it below) is used, then *rank* is not always equal to height. Also, size (in place of height) of trees can also be used as *rank*. Using size as *rank* also yields worst case time complexity as $O(\log n)$ (See [this](#) for proof)

Let us see the above example with union by rank
Initially, all elements are single element subsets.
0 1 2 3

```
Do Union(0, 1)
    1   2   3
    /
0
```

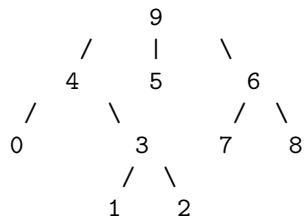
```
Do Union(1, 2)
    1   3
    /   \
0     2
```

Do Union(2, 3)

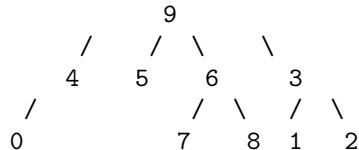


The second optimization to naive method is **Path Compression**. The idea is to flatten the tree when *find()* is called. When *find()* is called for an element x, root of the tree is returned. The *find()* operation traverses up from x to find root. The idea of path compression is to make the found root as parent of x so that we don't have to traverse all intermediate nodes again. If x is root of a subtree, then path (to root) from all nodes under x also compresses.

Let the subset {0, 1, .. 9} be represented as below and *find()* is called for element 3.



When *find()* is called for 3, we traverse up and find 9 as representative of this subset. With path compression, we also make 3 as the child of 9 so that when *find()* is called next time for 1, 2 or 3, the path to root is reduced.



The two techniques complement each other. The time complexity of each operation becomes even smaller than O(Logn). In fact, amortized time complexity effectively becomes small constant.

Following is union by rank and path compression based implementation to find a cycle in a graph.

C++

```

// A union by rank and path compression based program to detect cycle in a graph
#include <stdio.h>
#include <stdlib.h>

// a structure to represent an edge in the graph
struct Edge
{
    int src, dest;
}
  
```

```
};

// a structure to represent a graph
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges
    struct Edge* edge;
};

struct subset
{
    int parent;
    int rank;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = (struct Graph*) malloc( sizeof(struct Graph) );
    graph->V = V;
    graph->E = E;

    graph->edge = (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );

    return graph;
}

// A utility function to find set of an element i
// (uses path compression technique)
int find(struct subset subsets[], int i)
{
    // find root and make root as parent of i (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(struct subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);
```

```

// Attach smaller rank tree under root of high rank tree
// (Union by Rank)
if (subsets[xroot].rank < subsets[yroot].rank)
    subsets[xroot].parent = yroot;
else if (subsets[xroot].rank > subsets[yroot].rank)
    subsets[yroot].parent = xroot;

// If ranks are same, then make one as root and increment
// its rank by one
else
{
    subsets[yroot].parent = xroot;
    subsets[xroot].rank++;
}
}

// The main function to check whether a given graph contains cycle or not
int isCycle( struct Graph* graph )
{
    int V = graph->V;
    int E = graph->E;

    // Allocate memory for creating V sets
    struct subset *subsets =
        (struct subset*) malloc( V * sizeof(struct subset) );

    for (int v = 0; v < V; ++v)
    {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Iterate through all edges of graph, find sets of both
    // vertices of every edge, if sets are same, then there is
    // cycle in graph.
    for(int e = 0; e < E; ++e)
    {
        int x = find(subsets, graph->edge[e].src);
        int y = find(subsets, graph->edge[e].dest);

        if (x == y)
            return 1;

        Union(subsets, x, y);
    }
    return 0;
}

```

```
// Driver program to test above functions
int main()
{
    /* Let us create the following graph
       0
       |
       |
       |   \
       1-----2 */

    int V = 3, E = 3;
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;

    // add edge 1-2
    graph->edge[1].src = 1;
    graph->edge[1].dest = 2;

    // add edge 0-2
    graph->edge[2].src = 0;
    graph->edge[2].dest = 2;

    if (isCycle(graph))
        printf( "Graph contains cycle" );
    else
        printf( "Graph doesn't contain cycle" );

    return 0;
}
```

Java

```
// A union by rank and path compression
// based program to detect cycle in a graph
class Graph
{
    int V, E;
    Edge[] edge;

    Graph(int nV, int nE)
    {
        V = nV;
        E = nE;
        edge = new Edge[E];
        for (int i = 0; i < E; i++)
        {
```

```
        edge[i] = new Edge();
    }
}

// class to represent edge
class Edge
{
    int src, dest;
}

// class to represent Subset
class subset
{
    int parent;
    int rank;
}

// A utility function to find
// set of an element i (uses
// path compression technique)
int find(subset [] subsets , int i)
{
if (subsets[i].parent != i)
    subsets[i].parent = find(subsets,
                           subsets[i].parent);
return subsets[i].parent;
}

// A function that does union
// of two sets of x and y
// (uses union by rank)
void Union(subset [] subsets,
           int x , int y )
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[yroot].rank < subsets[xroot].rank)
        subsets[yroot].parent = xroot;
    else
    {
        subsets[xroot].parent = yroot;
        subsets[yroot].rank++;
    }
}
```

```
// The main function to check whether
// a given graph contains cycle or not
int isCycle(Graph graph)
{
    int V = graph.V;
    int E = graph.E;

    subset [] subsets = new subset[V];
    for (int v = 0; v < V; v++)
    {

        subsets[v] = new subset();
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    for (int e = 0; e < E; e++)
    {
        int x = find(subsets, graph.edge[e].src);
        int y = find(subsets, graph.edge[e].dest);
        if(x == y)
            return 1;
        Union(subsets, x, y);
    }
    return 0;
}

// Driver Code
public static void main(String [] args)
{
/* Let us create the following graph
   0
   | \
   | \
   1----2 */

    int V = 3, E = 3;
    Graph graph = new Graph(V, E);

    // add edge 0-1
    graph.edge[0].src = 0;
    graph.edge[0].dest = 1;

    // add edge 1-2
    graph.edge[1].src = 1;
    graph.edge[1].dest = 2;

    // add edge 0-2
```

```
graph.edge[2].src = 0;
graph.edge[2].dest = 2;

if (graph.isCycle(graph) == 1)
    System.out.println("Graph contains cycle");
else
    System.out.println("Graph doesn't contain cycle");
}
}

// This code is contributed
// by ashwani khemani
```

Output:

Graph contains cycle

Related Articles :

[Union-Find Algorithm Set 1 \(Detect Cycle in a an Undirected Graph\)](#)
[Disjoint Set Data Structures \(Java Implementation\)](#)
[Greedy Algorithms Set 2 \(Kruskal's Minimum Spanning Tree Algorithm\)](#)
[Job Sequencing Problem Set 2 \(Using Disjoint Set\)](#)

References:

http://en.wikipedia.org/wiki/Disjoint-set_data_structure
[IITD Video Lecture](#)

Improved By : [DeepeshThakur](#), [ashwani khemani](#)

Source

<https://www.geeksforgeeks.org/union-find-algorithm-set-2-union-by-rank/>

Chapter 270

Water Connection Problem

Water Connection Problem - GeeksforGeeks

Every house in the colony has at most one pipe going into it and at most one pipe going out of it. Tanks and taps are to be installed in a manner such that every house with one outgoing pipe but no incoming pipe gets a tank installed on its roof and every house with only an incoming pipe and no outgoing pipe gets a tap.'

Given two integers **n** and **p** denoting the number of houses and the number of pipes. The connections of pipe among the houses contain three input values: **a_i**, **b_i**, **d_i** denoting the pipe of diameter **d_i** from house **a_i** to house **b_i**, find out the efficient solution for the network.

The output will contain the number of pairs of tanks and taps **t** installed in first line and the next **t** lines contain three integers: house number of tank, house number of tap and the minimum diameter of pipe between them.

Examples:

```
Input : 4 2
        1 2 60
        3 4 50
```

```
Output :2
```

```
        1 2 60
        3 4 50
```

Explanation:

Connected components are:

1->2 and 3->4

Therefore, our answer is 2 followed by

1 2 60 and 3 4 50.

```
Input :9 6
        7 4 98
        5 9 72
        4 6 10
```

```
2 8 22
9 7 17
3 1 66
Output :3
2 8 22
3 1 66
5 6 10
```

Explanation:

Connected components are 3->1,

5->9->7->4->6 and 2->8.

Therefore, our answer is 3 followed by

2 8 22, 3 1 66, 5 6 10

Approach:

Perform DFS from appropriate houses to find all different connected components. The number of different connected components is our answer t.

The next t lines of the output are the beginning of the connected component, end of the connected component and the minimum diameter from the start to the end of the connected component in each line.

Since, tanks can be installed only on the houses having outgoing pipe and no incoming pipe, therefore these are appropriate houses to start DFS from i.e. perform DFS from such unvisited houses.

Below is the implementation of above approach:

C++

```
// C++ program to find efficient
// solution for the network
#include <bits/stdc++.h>
using namespace std;

// number of houses and number
// of pipes
int n, p;

// Array rd stores the
// ending vertex of pipe
int rd[1100];

// Array wd stores the value
// of diameters between two pipes
int wt[1100];

// Array cd stores the
// starting end of pipe
int cd[1100];

// Vector a, b, c are used
```

```

// to store the final output
vector<int> a;
vector<int> b;
vector<int> c;

int ans;

int dfs(int w)
{
    if (cd[w] == 0)
        return w;
    if (wt[w] < ans)
        ans = wt[w];
    return dfs(cd[w]);
}

// Function performing calculations.
void solve(int arr[][3])
{
    int i = 0;

    while (i < p) {

        int q = arr[i][0], h = arr[i][1],
            t = arr[i][2];

        cd[q] = h;
        wt[q] = t;
        rd[h] = q;
        i++;
    }

    a.clear();
    b.clear();
    c.clear();

    for (int j = 1; j <= n; ++j)

        /*If a pipe has no ending vertex
        but has starting vertex i.e is
        an outgoing pipe then we need
        to start DFS with this vertex.*/
        if (rd[j] == 0 && cd[j]) {
            ans = 1000000000;
            int w = dfs(j);

            // We put the details of component
            // in final output array
        }
    }
}

```

```
a.push_back(j);
b.push_back(w);
c.push_back(ans);
}

cout << a.size() << endl;
for (int j = 0; j < a.size(); ++j)
    cout << a[j] << " " << b[j]
        << " " << c[j] << endl;
}

// driver function
int main()
{
    n = 9, p = 6;

    memset(rd, 0, sizeof(rd));
    memset(cd, 0, sizeof(cd));
    memset(wt, 0, sizeof(wt));

    int arr[][][3] = { { 7, 4, 98 },
                      { 5, 9, 72 },
                      { 4, 6, 10 },
                      { 2, 8, 22 },
                      { 9, 7, 17 },
                      { 3, 1, 66 } };

    solve(arr);
    return 0;
}
```

Java

```
// Java program to find efficient
// solution for the network
import java.util.*;

class GFG {

    // number of houses and number
    // of pipes
    static int n, p;

    // Array rd stores the
    // ending vertex of pipe
    static int rd[] = new int[1100];

    // Array wd stores the value
```

```

// of diameters between two pipes
static int wt[] = new int[1100];

// Array cd stores the
// starting end of pipe
static int cd[] = new int[1100];

// arraylist a, b, c are used
// to store the final output
static List <Integer> a =
    new ArrayList<Integer>();

static List <Integer> b =
    new ArrayList<Integer>();

static List <Integer> c =
    new ArrayList<Integer>();

static int ans;

static int dfs(int w)
{
    if (cd[w] == 0)
        return w;
    if (wt[w] < ans)
        ans = wt[w];

    return dfs(cd[w]);
}

// Function to perform calculations.
static void solve(int arr[][])
{
    int i = 0;

    while (i < p)
    {

        int q = arr[i][0];
        int h = arr[i][1];
        int t = arr[i][2];

        cd[q] = h;
        wt[q] = t;
        rd[h] = q;
        i++;
    }
}

```

```

a=new ArrayList<Integer>();
b=new ArrayList<Integer>();
c=new ArrayList<Integer>();

for (int j = 1; j <= n; ++j)

    /*If a pipe has no ending vertex
     but has starting vertex i.e is
     an outgoing pipe then we need
     to start DFS with this vertex.*/
    if (rd[j] == 0 && cd[j]>0) {
        ans = 1000000000;
        int w = dfs(j);

        // We put the details of
        // component in final output
        // array
        a.add(j);
        b.add(w);
        c.add(ans);
    }

System.out.println(a.size());

for (int j = 0; j < a.size(); ++j)
    System.out.println(a.get(j) + " "
                      + b.get(j) + " " + c.get(j));
}

// main function
public static void main(String args[])
{
    n = 9;
    p = 6;

    // set the value of the araray
    // to zero
    for(int i = 0; i < 1100; i++)
        rd[i] = cd[i] = wt[i] = 0;

    int arr[][] = { { 7, 4, 98 },
                   { 5, 9, 72 },
                   { 4, 6, 10 },
                   { 2, 8, 22 },
                   { 9, 7, 17 },
                   { 3, 1, 66 } };
    solve(arr);
}

```

```
}
```

```
// This code is contributed by Arnab Kundu
```

Output:

```
3
2 8 22
3 1 66
5 6 10
```

Improved By : [andrew1234](#), [ROSHANRK_1995](#)

Source

<https://www.geeksforgeeks.org/water-connection-problem/>

Chapter 271

Water Jug problem using BFS

Water Jug problem using BFS - GeeksforGeeks

You are given a m litre jug and a n litre jug . Both the jugs are initially empty. The jugs don't have markings to allow measuring smaller quantities. You have to use the jugs to measure d litres of water where d is less than n.

(X, Y) corresponds to a state where X refers to amount of water in Jug1 and Y refers to amount of water in Jug2

Determine the path from initial state (x_i, y_i) to final state (x_f, y_f) , where (x_i, y_i) is $(0, 0)$ which indicates both Jugs are initially empty and (x_f, y_f) indicates a state which could be $(0, d)$ or $(d, 0)$.

The operations you can perform are:

1. Empty a Jug, $(X, Y) \rightarrow (0, Y)$ Empty Jug 1
2. Fill a Jug, $(0, 0) \rightarrow (X, 0)$ Fill Jug 1
3. Pour water from one jug to the other until one of the jugs is either empty or full, $(X, Y) \rightarrow (X-d, Y+d)$

Examples:

```
Input : 4 3 2
Output : {(0, 0), (0, 3), (4, 0), (4, 3),
           (3, 0), (1, 3), (3, 3), (4, 2),
           (0, 2)}
```

We have discussed one solution in [The Two Water Jug Puzzle](#)

In this post a [BFS](#) based solution is discussed.

We run breadth first search on the states and these states will be created after applying allowed operations and we also use visited map of pair to keep track of states that should be visited only once in the search. This solution can also be achieved using depth first search.

```
#include <bits/stdc++.h>
#define pii pair<int, int>
#define mp make_pair
using namespace std;

void BFS(int a, int b, int target)
{
    // Map is used to store the states, every
    // state is hashed to binary value to
    // indicate either that state is visited
    // before or not
    map<pii, int> m;
    bool isSolvable = false;
    vector<pii> path;

    queue<pii> q; // queue to maintain states
    q.push({0, 0}); // Initializing with initial state

    while (!q.empty()) {

        pii u = q.front(); // current state

        q.pop(); // pop off used state

        // if this state is already visited
        if (m[{u.first, u.second}] == 1)
            continue;

        // doesn't met jug constraints
        if ((u.first > a || u.second > b ||
             u.first < 0 || u.second < 0))
            continue;

        // filling the vector for constructing
        // the solution path
        path.push_back({u.first, u.second});

        // marking current state as visited
        m[{u.first, u.second}] = 1;

        // if we reach solution state, put ans=1
        if (u.first == target || u.second == target) {
            isSolvable = true;
            if (u.first == target) {
                if (u.second != 0)

                    // fill final state
                    path.push_back({u.first, 0});
            }
        }
    }
}
```

```

    }
else {
    if (u.first != 0)

        // fill final state
        path.push_back({ 0, u.second });

}

// print the solution path
int sz = path.size();
for (int i = 0; i < sz; i++)
    cout << "(" << path[i].first
        << ", " << path[i].second << ")\\n";
break;
}

// if we have not reached final state
// then, start developing intermediate
// states to reach solution state
q.push({ u.first, b }); // fill Jug2
q.push({ a, u.second }); // fill Jug1

for (int ap = 0; ap <= max(a, b); ap++) {

    // pour amount ap from Jug2 to Jug1
    int c = u.first + ap;
    int d = u.second - ap;

    // check if this state is possible or not
    if (c == a || (d == 0 && d >= 0))
        q.push({ c, d });

    // Pour amount ap from Jug 1 to Jug2
    c = u.first - ap;
    d = u.second + ap;

    // check if this state is possible or not
    if ((c == 0 && c >= 0) || d == b)
        q.push({ c, d });
}

q.push({ a, 0 }); // Empty Jug2
q.push({ 0, b }); // Empty Jug1
}

// No, solution exists if ans=0
if (!isSolvable)
    cout << "No solution";

```

```
}

// Driver code
int main()
{
    int Jug1 = 4, Jug2 = 3, target = 2;
    cout << "Path from initial state "
          "to solution state :\n";
    BFS(Jug1, Jug2, target);
    return 0;
}
```

Output:

```
Path from initial state to solution state ::  
(0, 0)  
(0, 3)  
(4, 0)  
(4, 3)  
(3, 0)  
(1, 3)  
(3, 3)  
(4, 2)  
(0, 2)
```

Source

<https://www.geeksforgeeks.org/water-jug-problem-using-bfs/>

Chapter 272

Word Ladder (Length of shortest chain to reach a target word)

Word Ladder (Length of shortest chain to reach a target word) - GeeksforGeeks

Given a dictionary, and two words ‘start’ and ‘target’ (both of same length). Find length of the smallest chain from ‘start’ to ‘target’ if it exists, such that adjacent words in the chain only differ by one character and each word in the chain is a valid word i.e., it exists in the dictionary. It may be assumed that the ‘target’ word exists in dictionary and length of all dictionary words is same.

Example:

```
Input: Dictionary = {POON, PLEE, SAME, POIE, PLEA, PLIE, POIN}
       start = TOON
       target = PLEA
Output: 7
Explanation: TOON - POON - POIN - POIE - PLIE - PLEE - PLEA
```

The idea is to use [BFS](#). We start from the given start word, traverse all words that adjacent (differ by one character) to it and keep doing so until we find the target word or we have traversed all words.

Below is C++ implementation of above idea.

C

```
// C++ program to find length of the shortest chain
// transformation from source to target
#include<bits/stdc++.h>
using namespace std;
```

```
// To check if strings differ by exactly one character
bool isadjacent(string& a, string& b)
{
    int count = 0; // to store count of differences
    int n = a.length();

    // Iterate through all characters and return false
    // if there are more than one mismatching characters
    for (int i = 0; i < n; i++)
    {
        if (a[i] != b[i]) count++;
        if (count > 1) return false;
    }
    return count == 1 ? true : false;
}

// A queue item to store word and minimum chain length
// to reach the word.
struct QItem
{
    string word;
    int len;
};

// Returns length of shortest chain to reach 'target' from 'start'
// using minimum number of adjacent moves. D is dictionary
int shortestChainLen(string& start, string& target, set<string> &D)
{
    // Create a queue for BFS and insert 'start' as source vertex
    queue<QItem> Q;
    QItem item = {start, 1}; // Chain length for start word is 1
    Q.push(item);

    // While queue is not empty
    while (!Q.empty())
    {
        // Take the front word
        QItem curr = Q.front();
        Q.pop();

        // Go through all words of dictionary
        for (set<string>::iterator it = D.begin(); it != D.end(); it++)
        {
            // Process a dictionary word if it is adjacent to current
            // word (or vertex) of BFS
            string temp = *it;
            if (isadjacent(curr.word, temp))
```

```
{  
    // Add the dictionary word to Q  
    item.word = temp;  
    item.len = curr.len + 1;  
    Q.push(item);  
  
    // Remove from dictionary so that this word is not  
    // processed again. This is like marking visited  
    D.erase(temp);  
  
    // If we reached target  
    if (temp == target)  
        return item.len;  
}  
}  
}  
return 0;  
}  
  
// Driver program  
int main()  
{  
    // make dictionary  
    set<string> D;  
    D.insert("poon");  
    D.insert("plee");  
    D.insert("same");  
    D.insert("poie");  
    D.insert("plie");  
    D.insert("poin");  
    D.insert("plea");  
    string start = "toon";  
    string target = "plea";  
    cout << "Length of shortest chain is: "  
        << shortestChainLen(start, target, D);  
    return 0;  
}
```

Python

```
# To check if strings differ by  
# exactly one character  
  
def isadjacent(a, b):  
    count = 0  
    n = len(a)  
  
    # Iterate through all characters and return false
```

```
# if there are more than one mismatching characters
for i in range(n):
    if a[i] != b[i]:
        count += 1
    if count > 1:
        break

return True if count == 1 else False

# A queue item to store word and minimum chain length
# to reach the word.
class QItem():

    def __init__(self, word, len):
        self.word = word
        self.len = len

    # Returns length of shortest chain to reach
    # 'target' from 'start' using minimum number
    # of adjacent moves. D is dictionary
def shortestChainLen(start, target, D):

    # Create a queue for BFS and insert
    # 'start' as source vertex
    Q = []
    item = QItem(start, 1)
    Q.append(item)

    while( len(Q) > 0):

        curr = Q.pop()

        # Go through all words of dictionary
        for it in D:

            # Process a dictionary word if it is
            # adjacent to current word (or vertex) of BFS
            temp = it
            if isadjacent(curr.word, temp) == True:

                # Add the dictionary word to Q
                item.word = temp
                item.len = curr.len + 1
                Q.append(item)

            # Remove from dictionary so that this
            # word is not processed again. This is
            # like marking visited
```

```
D.remove(temp)

# If we reached target
if temp == target:
    return item.len

D = []
D.append("poon")
D.append("plee")
D.append("same")
D.append("poie")
D.append("plie")
D.append("poin")
D.append("plea")
start = "toon"
target = "plea"
print "Length of shortest chain is: %d" \
      % shortestChainLen(start, target, D)

# This code is contributed by Divyanshu Mehta
```

Output:

```
Length of shortest chain is: 7
```

Time Complexity of the above code is $O(n^2m)$ where n is the number of entries originally in the dictionary and m is the size of the string

Thanks to Gaurav Ahirwar and Rajnish Kumar Jha for above solution.

Source

<https://www.geeksforgeeks.org/word-ladder-length-of-shortest-chain-to-reach-a-target-word/>