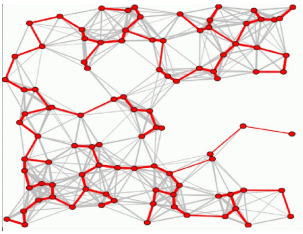
Graph

1. Applications of Minimum Spanning Tree Problem

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

2. Applications of Depth First Search

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

Following are the problems that use DFS as a bulding 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
See this for details.

5) To test if a graph is bipartite
We can augment either BFS or DFS when we first discover a new vertex, color it opposited 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/LEC/LECTUR16/NODE16.HTM
http://en.wikipedia.org/wiki/Depth-first_search
http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/depthSearch.h
http://ww3.algorithmdesign.net/handouts/DFS.pdf

3. Boruvka's algorithm for Minimum Spanning Tree

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 Boruuvka 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:

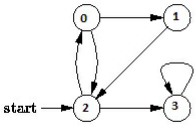
http://en.wikipedia.org/wiki/Bor%C5%AFvka%27s_algorithm

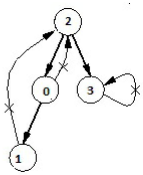
http://theory.cs.uiuc.edu/~jeffe/teaching/algorithms/notes/12-mst.pdf

4. Depth First Traversal for a Graph

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. Depth First Traversal of the following graph is 2, 0, 1, 3





See this post for all applications of Depth First Traversal.
Following is C++ implementation of simple Depth First Traversal. The implementation uses adjacency list representation of graphs. STL's list container is used to store lists of adjacent nodes.

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

using namespace std;

// Graph class represents a directed graph using adjacency list repres
class Graph
{
    int V;    // No. of vertices
    list<int> *adj;    // Pointer to an array containing adjacency lis
    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(int v);    // DFS traversal of the vertices reachable fro
};

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 D
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
    g.DFS(2);

    return 0;
}
```

Output:

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

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 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.

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

class Graph
{
    int V;    // No. of vertices
    list<int> *adj;    // Pointer to an array containing adjacency lis
    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
```

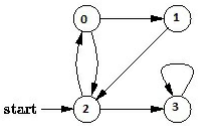
```
};

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 (starting from vertex
    g.DFS();

    return 0;
}
```

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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

5. Breadth First Traversal for a Graph

Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree (See mthod 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. Breadth First Traversal of the following graph is 2, 0, 3, 1.



Following is C++ implementation 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.

```
// 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 represe
class Graph
{
    int V;    // No. of vertices
    list<int> *adj;    // Pointer to an array containing adjacency lis
public:
    Graph(int V);  // Constructor
    void addEdge(int v, int w); // function to add an edge to graph
    void BFS(int s);  // prints BFS traversal from a given source 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 verte
    g.BFS(2);

    return 0;
}
```

Output:

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

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.

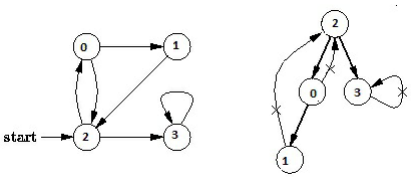
Also see Depth First Traversal

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

6. Detect Cycle in a Directed Graph

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->2->0, 0->1->2->0 and 3->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 forrest as output. To detect cycle, we can check for 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 back edge. We have used recStack[] array to keep track of vertices in the recursion stack.

```
// 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 lis
    bool isCyclicUtil(int v, bool visited[], bool *rs);  // used by is
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 gr
};

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 DFSUytil() in http://www.geeksforge
bool Graph::isCyclicUtil(int v, bool visited[], bool *recStack)
{
    if(visited[v] == false)
    {
        // Mark the current node as visited and part of recursion stac
        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;
            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 http://www.geeksforgeeks.o
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;
}
```

Output:

```
Graph contains cycle
```

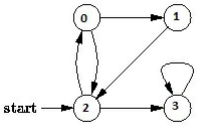
Time Complexity of this method is same as time complexity of DFS traversal which is O(V+E).

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

7. Find if there is a path between two vertices in a directed graph

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 is C++ code that uses BFS for finding reachability of second vertex from first vertex.

```
// Program to check if there is exist a path between two vertices of a
#include<iostream>
#include <list>

using namespace std;

// This class represents a directed graph using adjacency list represe
class Graph
{
    int V;    // No. of vertices
    list<int> *adj;    // Pointer to an array containing adjacency lis
public:
    Graph(int V);  // Constructor
    void addEdge(int v, int w); // function to add an edge to graph
    bool isReachable(int s, int d);  // returns true if there is a pat
};

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 ret
            if (*i == d)
                return true;

            // Else, continue to do BFS
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }

    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;
}
```

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.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

8. Backtracking | Set 6 (Hamiltonian Cycle)

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 graph[V][V] where V is the number of vertices in graph and graph[V][V] is adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.

Output:
An array path[V] that should contain the Hamiltonian Path. path[i] should represent the ith 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.

```
(0)--(1)--(2)
 |   / \   |
 |  /   \  |
```

```
 | /     \ |
(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 is C/C++ implementation of the Backtracking solution.

```
// Program to print Hamiltonian cycle
#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 '
   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 th
        // 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 ham
    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 f
       then return false */
    return false;
}

/* This function solves the Hamiltonian Cycle problem using Backtracki
  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 solutio
  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 i
       a Hamiltonian Cycle, then the path can be started from any poin
       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:"
            " 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;
}
```

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.

9. Dynamic Programming | Set 16 (Floyd Warshall Algorithm)

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
             10
       (0)------->(3)
        |         /|\
      5 |          |
        |          | 1
       \|/         |
       (1)------->(2)
             3
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
```

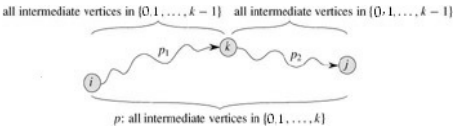
Floyd Warshall Algorithm
We initialize the solution matrix same as the input graph matrix as a first step. Then we

update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and update 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, .. k-1} as intermediate vertices. For every pair (i, j) of 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 is taken from the Cormen book. It shows the above optimal substructure property in the all-pairs shortest path problem.



Following is C implementation of the Floyd Warshall algorithm.

```
// 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[][V]);

// Solves the all-pairs shortest path problem using Floyd Warshall alg
void floydWarshell (int graph[][V])
{
    /* dist[][] will be the output matrix that will finally have the s
      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 betw
      pairs of vertices such that the shortest distances consider only
      vertices in set {0, 1, 2, .. k-1} as intermediate vertices.
      ----> After the end of a iteration, vertex no. k is added to the
      intermediate vertices and the set becomes {0, 1, 2, .. k} */
    for (k = 0; k < V; k++)
    {
        // Pick all vertices as source one by one
```

```
        // 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[][V])
{
    printf ("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
    floydWarshell(graph);
    return 0;
}
```

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 arithmatic overflow.

```
#include<limits.h>

#define INF INT_MAX
.........................
if (dist[i][k] != INF && dist[k][j] != INF && dist[i][k] + dist[k][j]
    dist[i][j] = dist[i][k] + dist[k][j];
.........................
```

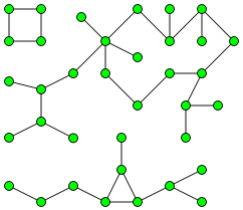
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

10. Find the number of islands

Given a boolean 2D matrix, find the number of islands.

This is an variation of the standard problem: “Counting number of connected components in a 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 neighbors. So, unlike standard DFS(), where we recursively call for all adjacent vertices, here we can recursive call for 8 neighbors only. We keep track of the visited 1s so that they are not visited again.

```
// 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 DI
int isSafe(int M[][COL], int row, int col, bool visited[][COL])
{
    return (row >= 0) && (row < ROW) &&     // row number is in range
           (col >= 0) && (col < COL) &&     // column number is in rang
           (M[row][col] && !visited[row][col]); // value is 1 and not y
}

// A utility function to do DFS for a 2D boolean matrix. It only consi
// the 8 neighbors 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 neighb
    // of a given cell
```

```
    // 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 travese 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 i
            {                              // visited yet, then new is
                DFS(M, i, j, visited);     // Visit all cells in this
                ++count;                   // and increment island cou
            }

    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;
}
```

Output:

```
Number of islands is: 5
```

Time complexity: O(ROW x COL)

Reference:
http://en.wikipedia.org/wiki/Connected_component_%28graph_theory%29

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

11. Union-Find Algorithm | Set 1 (Detect Cycle in a an Undirected Graph)

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:

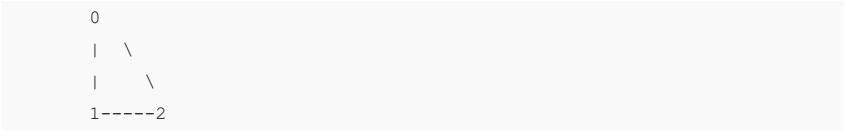
Find: Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.

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 keeps track of the subsets in a 1D array, lets 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,
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 is the code:

```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 Edg

    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);
    parent[xset] = yset;
}

// The main function to check whether a given graph contains cycle or
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 */
    struct Graph* graph = createGraph(3, 3);

    // 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;
}
```

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.

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.

12. Union-Find Algorithm | Set 2 (Union By Rank and Path Compression)

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(Logn) (See this for prrof)

```
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)
    1
 /  |  \
0   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.
              9
         /    |    \
        4     5      6
     /     \        /  \
    0        3     7    8
            /  \
           1    2

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 child of 9 so
that when find() is called next time for 1, 2 or 3, the path to root is
reduced.

               9
         /    /  \    \
        4    5    6     3
     /           /  \   /  \
    0           7    8 1    2
```

The two techniques complement each other. The time complexity of each operations 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 cycle in a graph.

```
// A union by rank and path compression based program to detect cycle
#include <stdio.h>
#include <stdlib.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;
};

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 Edg

    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
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 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;
}
```

Output:

```
Graph contains cycle
```

We will soon be discussing Kruskal's Algorithm as next application of Union-Find Algorithm.

References:
http://en.wikipedia.org/wiki/Disjoint-set_data_structure
IITD Video Lecture

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

13. Greedy Algorithms | Set 2 (Kruskal's Minimum Spanning Tree Algorithm)

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.

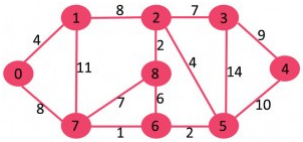
Below are the steps for finding MST using Kruskal's algorithm

```
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.

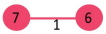


The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

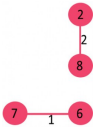
```
After sorting:
Weight   Src    Dest
1          7      6
2          8      2
2          6      5
4          0      1
4          2      5
6          8      6
7          2      3
7          7      8
8          0      7
8          1      2
9          3      4
10         5      4
11         1      7
14         3      5
```

Now pick all edges one by one from sorted list of edges

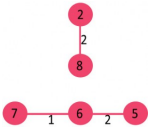
1. *Pick edge 7-6:* No cycle is formed, include it.



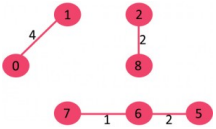
2. *Pick edge 8-2:* No cycle is formed, include it.



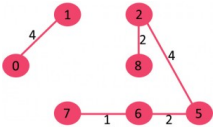
3. *Pick edge 6-5:* No cycle is formed, include it.



4. *Pick edge 0-1:* No cycle is formed, include it.

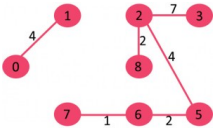


5. *Pick edge 2-5:* No cycle is formed, include it.



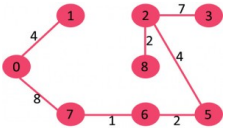
6. *Pick edge 8-6:* Since including this edge results in cycle, discard it.

7. *Pick edge 2-3:* No cycle is formed, include it.



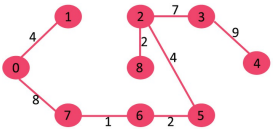
8. *Pick edge 7-8:* Since including this edge results in cycle, discard it.

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.

```
// Kruskal's algortihm to find Minimum Spanning Tree of a given connec
// 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
```

```
    // 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 = (struct Graph*) malloc( sizeof(struct Graph)
    graph->V = V;
    graph->E = E;

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

    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];  // Tnis 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 we
    // If we are not allowed to change the given graph, we can create
    // array of edges
    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);

    // Allocate memory for creating V ssubsets
    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;
}
```

```
Following are the edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
```

Time Complexity: O(ElogE) or O(ElogV). Sorting of edges takes O(ELogE) time. After sorting, we iterate through all edges and apply find-union algorithm. The find and union operations can take atmost O(LogV) time. So overall complexity is O(ELogE + ELogV) time. The value of E can be atmost V^2, so O(LogV) are O(LogE) same. Therefore, overall time complexity is O(ElogE) or O(ElogV)

References:
http://www.ics.uci.edu/~eppstein/161/960206.html

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

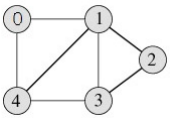
This article is compiled by Aashish Barnwal and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

14. Graph and its representations

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 directed graph(di-graph). The pair of 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. This can be easily viewed by http://graph.facebook.com/barnwal.aashish where barnwal.aashish is the profile name. See this for more applications of graph.

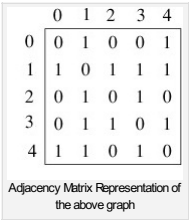
Following is an example undirected graph with 5 vertices.



Following two are the most commonly used representations of 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 x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot 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 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:



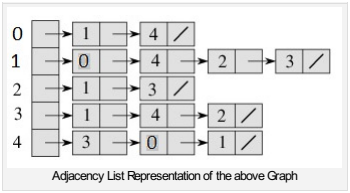
	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Adjacency Matrix Representation of the above graph

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.

Adjacency List:
An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be array[]. An entry array[i] represents the linked 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 stored in nodes of linked lists. Following is adjacency list representation of the above graph.



Adjacency List Representation of the above Graph

Below is C code for adjacency list representation of an undirected graph:

```
// 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 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
```

```
// 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 adjac
    // 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 adjacenncy list representation of g
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 fugure
    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;
}
```

Output:

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


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


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


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


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

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_%28abstract_data_type%29

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.

15. Greedy Algorithms | Set 5 (Prim's Minimum Spanning Tree (MST))

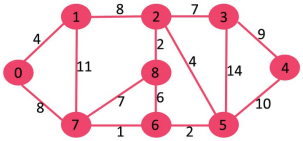
We have discussed Kruskal's algorithm for Minimum Spanning Tree. Like Kruskal's algorithm, Prim's algorithm is also a Greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains 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 verices), 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.

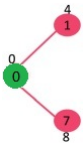
Algorithm
1) Create a set *mstSet* that keeps track of vertices already included in MST.
2) Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
3) While mstSet doesn't include all vertices
….**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.

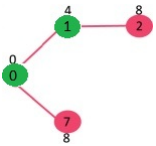
Let us understand with the following example:



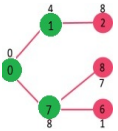
The set *mstSet* is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum key value. The vertex 0 is picked, include it in *mstSet*. So *mstSet* becomes {0}. After including to *mstSet*, update key values 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.



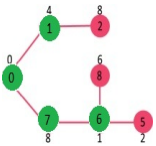
Pick the vertex with minimum key value and not already included in MST (not in mstSET). The vertex 1 is picked and added to mstSet. So mstSet now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.



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).

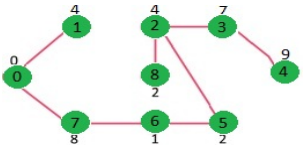


Pick the vertex with minimum key value and not already included in MST (not in mstSET). Vertex 6 is picked. So mstSet now becomes {0, 1, 7, 6}. Update the key values of adjacent vertices of 6. The key value of vertex 5 and 8 are updated.



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.

```
// 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>

// 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   Weight\n");
   for (int i = 1; i < V; i++)
      printf("%d - %d    %d \n", parent[i], i, graph[i][parent[i]]);
}
```

```
// Function to construct and print MST for a graph represented using a
// matrix representation
void primMST(int graph[V][V])
{
     int parent[V]; // Array to store constructed MST
     int key[V];   // Key values used to pick minimum weight edge in c
     bool mstSet[V];  // To represent set of vertices not yet included

     // 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 fi
    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
       // the picked vertex. Consider only those vertices which are n
       // 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[
            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;
}
```

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. We will soon be discussing O(E Log V) algorithm as a separate post.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

16. Greedy Algorithms | Set 6 (Prim's MST for Adjacency List Representation)

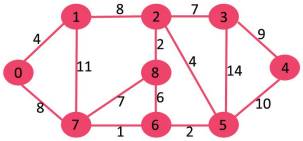
We recommend to read following two posts as a prerequisite of this post.

1. Greedy Algorithms | Set 5 (Prim's Minimum Spanning Tree (MST))
2. Graph and its representations

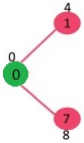
We have discussed Prim's algorithm and its implementation for adjacency matrix representation of graphs. The time complexity for the matrix representation is O(V^2). In this post, O(ELogV) algorithm for adjacency list representation is discussed.
As discussed in the previous post, in Prim's algorithm, 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(LogV) 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.

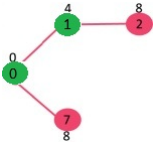
Let us understand the above algorithm with the following example:



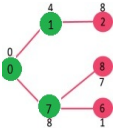
Initially, key value of first vertex is 0 and INF (infinite) for all other vertices. So vertex 0 is extracted from Min Heap and key values of vertices adjacent to 0 (1 and 7) are updated. Min Heap contains all vertices except vertex 0.
The vertices in green color are the vertices included in MST.



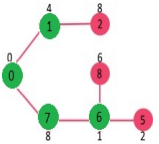
Since key value of vertex 1 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 1 are updated (Key is updated if the a vertex is not in Min Heap and previous key value is greater than the weight of edge from 1 to the adjacent). Min Heap contains all vertices except vertex 0 and 1.



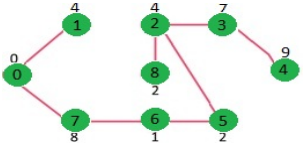
Since key value of vertex 7 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 7 are updated (Key is updated if the a vertex is not in Min Heap and previous key value is greater than the weight of edge from 7 to the adjacent). Min Heap contains 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++ program for Prim's MST for adjacency list representation of
```

```c
// C / C++ program for Prim's MST for adjacency list representation of

#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 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
// 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 adjac
    // 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 MinHea
    return minHeap;
}

// A utility function to swap two nodes of min heap. Needed for min he
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];
```

```
    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]->k
    {
        // 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) /

        // 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

    // 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;
}
```

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(LogV) time. So overall time complexity is O(E+V)*O(LogV) which is O((E+V)*LogV) = O(ELogV) (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.

17. Greedy Algorithms | Set 7 (Dijkstra's shortest path algorithm)

Given a graph and a source vertex in 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 minimum distance from 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.

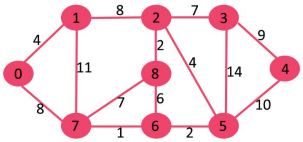
3) While *sptSet* doesn't include all vertices

….**a)** Pick a vertex u which is not there in *sptSet*and has minimum distance value.

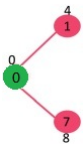
….**b)** Include u to *sptSet*.

….**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.

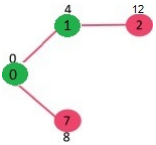
Let us understand with the following example:



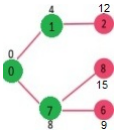
The set *sptSet*is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*, update 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 color.



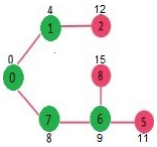
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



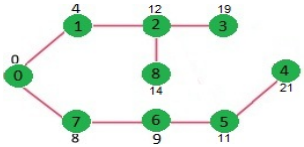
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



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.

```
// A C / C++ program for Dijkstra's single source shortest path algori
// 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,
// 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\t %d\n", i, dist[i]);
}

// Funtion that implements Dijkstra's single source shortest path algo
// 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 shor
                      // distance from src to i

     bool sptSet[V]; // sptSet[i] will true if vertex i is included in
                     // path tree or shortest distance from src to i i

     // 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 no
       // 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 ver
       for (int v = 0; v < V; v++)

         // Update dist[v] only if is not in sptSet, there is an edge
         // u to v, and total weight of path from src to  v through u
         // 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, 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;
}
```

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 dijekstra 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) 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. We will soon be discussing O(E Log V) algorithm as a separate post.

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.
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

18. Greedy Algorithms | Set 8 (Dijkstra's Algorithm for Adjacency List Representation)

We recommend to read following two posts as a prerequisite of this post.

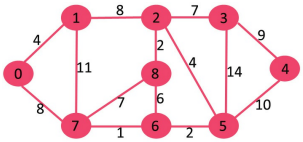
1. Greedy Algorithms | Set 7 (Dijkstra's shortest path algorithm)
2. Graph and its representations

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(ELogV) algorithm for adjacency list representation is discussed.

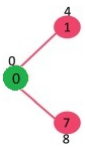
As discussed in the previous post, in Dijkstra's algorithm, two sets are maintained, one set contains list of vertices already included in SPT (Shortest Path Tree), other set contains vertices not yet included. With adjacency list representation, all vertices of a graph can be traversed in O(V+E) time 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(LogV) 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.

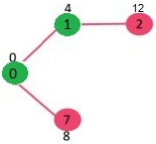
Let us understand with the following example. Let the given source vertex be 0



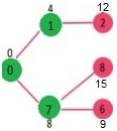
Initially, distance value of source vertex is 0 and INF (infinite) for all other vertices. So source vertex is extracted from Min Heap and distance values of vertices adjacent to 0 (1 and 7) are updated. Min Heap contains all vertices except vertex 0.
The vertices in green color are the vertices for which minimum distances are finalized and are not in Min Heap



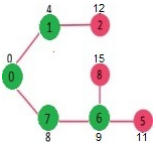
Since distance value of vertex 1 is minimum among all nodes in Min Heap, it is extracted from Min Heap and distance values of vertices adjacent to 1 are updated (distance is updated if the a vertex is not in Min Heap and distance through 1 is shorter than the previous distance). Min Heap contains all vertices except vertex 0 and 1.



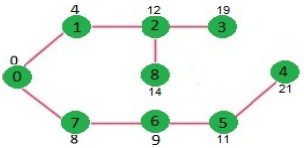
Pick the vertex with minimum distance value from min heap. Vertex 7 is picked. So min heap now contains all vertices except 0, 1 and 7. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



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++ program for Dijkstra's shortest path algorithm for adjacenc
// 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 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
// 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 adjac
    // 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 MinHea
    return minHeap;
}

// A utility function to swap two nodes of min heap. Needed for min he
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 decreasy 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 hepified.
    // This is a O(Logn) loop
    while (i && minHeap->array[i]->dist < minHeap->array[(i - 1) / 2]-
    {
        // 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) /
```

```
        // 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 calulates distances of shortest paths from s
// 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

    // minHeap represents set E
    struct MinHeap* minHeap = createMinHeap(V);

    // Initialize min heap with all vertices. dist value of all vertic
    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 firs
    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 dis
            // through u is less than its previously calculated distan
            if (isInMinHeap(minHeap, v) && dist[u] != INT_MAX &&
                                          pCrawl->weight + dist[u] < d
            {
                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 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);

    dijkstra(graph, 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: 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(LogV) time. So overall time complexity is O(E+V)*O(LogV) which is O((E+V)*LogV) = O(ELogV)
Note that the above code uses Binary Heap for Priority Queue implementation. Time complexity can be reduced to O(E + VLogV) using Fibonacci Heap. The reason is, Fibonacci Heap takes O(1) time for decrease-key operation while Binary Heap takes O(Logn) 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 dijekstra 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.

References:
Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.
Algorithms by Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

19. Dynamic Programming | Set 23 (Bellman–Ford Algorithm)

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. Dijksra's algorithm is a Greedy algorithm and time complexity is O(VLogV) (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.

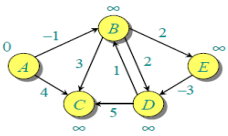
2) This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.
…..**a)** Do following for each edge u-v
……If dist[v] > dist[u] + weight of edge uv, then update dist[v]
…….dist[v] = dist[u] + weight of edge uv

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

How does this work? Like other Dynamic Programming Problems, the algorithm calculate shortest paths in bottom-up manner. It first calculates the shortest distances for the shortest paths which have at-most one edge in the path. Then, it calculates shortest paths with at-nost 2 edges, and so on. After the ith 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)

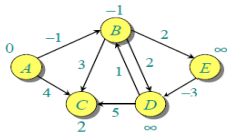
Example
Let us understand the algorithm with following example graph. The images are taken from this source.

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.*



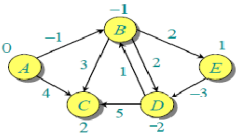
A	B	C	D	E
0	∞	∞	∞	∞

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.



A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞

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).



A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞
0	−1	2	∞	1
0	−1	2	1	1
0	−1	2	−2	1

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:

```
// A C / C++ program for Bellman-Ford's single source shortest path algorithm.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.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 = (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 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;
}
```

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%E2%80%93Ford_algorithm
http://www.cs.arizona.edu/classes/cs445/spring07/ShortestPath2.prn.pdf

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

20. Transitive closure of a graph

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.
The graph is given in the form of adjacency matrix say 'graph[V][V]' where graph[i][j] is 1 if there is an edge from vertex i to vertex j or i is equal to j, otherwise graph[i][j] is 0.

Floyd Warshall Algorithm can be used, we can calculate the distance matrix dist[V][V] using Floyd Warshall, if dist[i][j] is infinite, then j is not reachable from i, otherwise j is reachable and value of 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 reach[V][V] (we save space). The value 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)

```
// 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[][V]);

// Prints transitive closure of graph[][] using Floyd Warshall algorit
void transitiveClosure(int graph[][V])
{
    /* reach[][] will be the output matrix that will finally have the
      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 fo
      pairs of vertices such that the reachability values consider onl
      vertices in set {0, 1, 2, .. k-1} as intermediate vertices.
      ----> After the end of a iteration, vertex no. k is added to the
      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] || (reach[i][k] && reach[k][
            }
        }
    }

    // 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
    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;
}
```

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.

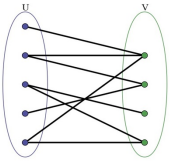
References:
Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

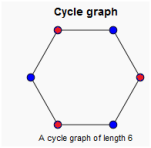
21. Check whether a given graph is Bipartite or not

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

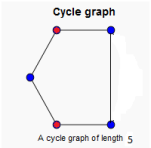
belongs to V and v to U. We can also say that there is no edge that connects vertices of same set.



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.



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



Algorithm to check if a graph is Bipartite:
One approach is to check whether the graph is 2-colorable or not using backtracking algorithm m coloring problem.
Following is a simple algorithm to find out whether a given graph is Birpartite 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++ 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 fals
bool isBipartite(int G[][V], int src)
{
    // Create a color array to store colors assigned to all veritces.
    // number is used as index in this array. The value '-1' of  color
    // is used to indicate that no color is assigned to vertex 'i'.  T
```

```
    // 1 is used to indicate first color is assigned and value 0 indic
    // 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 vert
    // 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();

         // 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 col
            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 colore
            // 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 wit
    // 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;
}
```

Output:

```
Yes
```

Refer this for C implementation of the same.

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?
2. The above algorithm works if the graph is strongly connected. Extend above code to work for graph with more than one component.

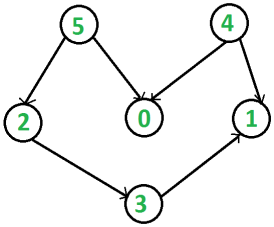
References:
http://en.wikipedia.org/wiki/Graph_coloring
http://en.wikipedia.org/wiki/Bipartite_graph

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

22. Topological Sorting

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 in-coming 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 above 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 is C++ implementation 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.

```
// 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> &Sta
{
    // 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 topologicalS
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;
}
```

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].

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

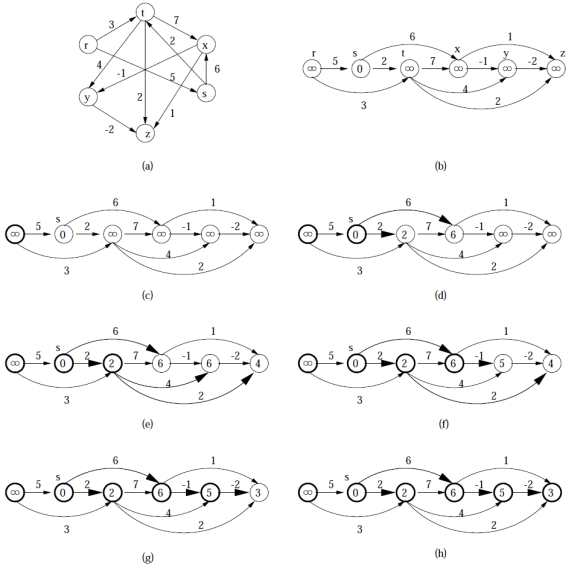
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

23. Shortest Path in Directed Acyclic Graph

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 + VLogV) 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 dist[] = {INF, INF, ….} and dist[s] = 0 where s is the source vertex.
2) Create a toplogical order of all vertices.
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)

```
// A C++ program to find single source shortest paths for Directed Acy
#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
// contains vertex number of the vertex to which edge connects. It als
// 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 detai
// http://www.geeksforgeeks.org/topological-sorting/
void Graph::topologicalSortUtil(int v, bool visited[], stack<int> &Sta
{
    // 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 recu
// 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
    // 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 <<" \
    g.shortestPath(s);

    return 0;
}
```

Output:

```
Following are shortest distances from source 1
INF 0 2 6 5 3
```

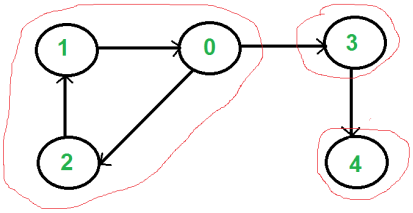
Time Complexity: Time complexity of topological sorting is O(V+E). After finding topological order, the algorithm process all vertices and for every vertex, it runs a loop for all adjacent vertices. Total adjacent vertices in a graph is O(E). So the inner loop runs O(V+E) times. Therefore, overall time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

References:
http://www.utdallas.edu/~sizheng/CS4349.d/l-notes.d/L17.pdf

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

24. Strongly Connected Components

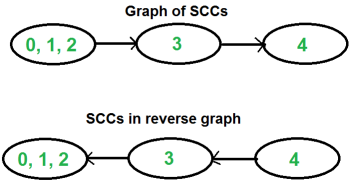
A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (**SCC**) of a directed graph is a maximal strongly connected subgraph. For example, there are 3 SCCs in the following graph.



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.
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'. Take v as source and do DFS (call DFSUtil(v)). The DFS starting from v prints strongly connected component of v.

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 that 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. 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.

```
// 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 tim
    // 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 comp
    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 to S
    Stack.push(v);
}

// The main function that finds and prints all strongly connected compo
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 grap
    g.printSCCs();

    return 0;
}
```

Output:

```
Following are strongly connected components in given graph
0 1 2
3
4
```

Time Complexity: The above algorithm calls DFS, fins 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 simple 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

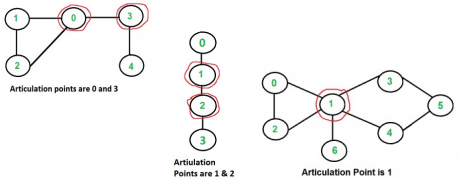
You may also like to see Tarjan's Algorithm to find Strongly Connected Components.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

25. Articulation Points (or Cut Vertices) in a Graph

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.



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

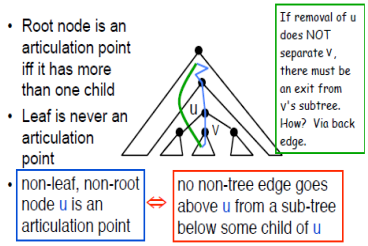
causes disconnected graph. Following are steps of simple approach for connected graph.

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. (Source Ref 2)



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 is C++ implementation of Tarjan's algorithm for finding articulation points.

```
// A C++ program to find articulation points in a given undirected gra
#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 traver
// 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 s
    // 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 aadjacent 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 chilren.
            if (parent[u] == NIL && children > 1)
               ap[u] = true;

            // (2) If u is not root and low value of one of its child
            // 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) array
    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;
}
```

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.

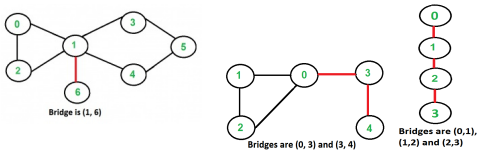
References:
https://www.cs.washington.edu/education/courses/421/04su/slides/artic.pdf
http://www.slideshare.net/TraianRebedea/algorithm-design-and-complexity-course-8
http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250_Weiss/L25-Connectivity.htm

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

26. Bridges in a graph

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 connected 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 exit any other alternative to reach u or an ancestor of u from subtree rooted with v. As discussed in the previous post, the value low[v] indicates earliest visited vertex reachable from subtree rooted with v. *The condition for an edge (u, v) to be a bridge is, "low[v] > disc[u]".*

Following is C++ implementation of above approach.

```
// 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 p
public:
    Graph(int V);   // Constructor
    void addEdge(int v, int w);   // function 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 traver
// 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 st
    // 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 aadjacent 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.
```

```
        // 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
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;
}
```

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.

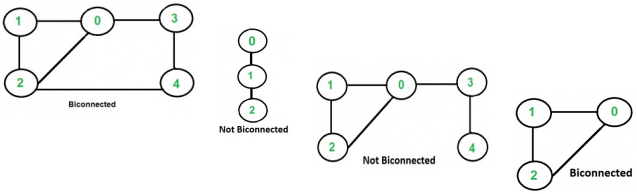
References:
https://www.cs.washington.edu/education/courses/421/04su/slides/artic.pdf
http://www.slideshare.net/TraianRebedea/algorithm-design-and-complexity-course-8
http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250_Weiss/L25-Connectivity.htm
http://www.youtube.com/watch?v=bmyyxNyZKzI

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

27. Biconnected graph

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.

```
// A C++ program to find articulation points in a given undirected gra
#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 par
public:
    Graph(int V);   // Constructor
    void addEdge(int v, int w);   // function to add an edge to graph
    bool isBC();    // returns true if graph is Biconnected, else retu
};

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/Me9
// 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 p
{
    // A static variable is used for simplicity, we can avoid use of s
    // 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 aadjacent 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 poi
            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 chilren.
            if (parent[u] == NIL && children > 1)
               return true;

            // (2) If u is not root and low value of one of its child
            // 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, otherw
// 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) array
    for (int i = 0; i < V; i++)
    {
        parent[i] = NIL;
        visited[i] = false;
    }

    // Call the recursive helper function to find if there is an artic
    // point in given graph. We do DFS traversal starring from vertex
    if (isBCUtil(0, visited, disc, low, parent) == true)
        return false;

    // Now check whether the given graph is connected or not. An undir
    // graph is connected if all vertices are reachable from any start
    // 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;
}
```

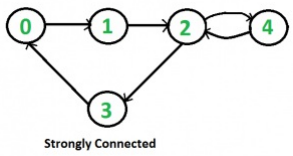
Time Complexity: The above function is a simple DFS with additional arrays. So time complexity is same as DFS which is O(V+E) for adjacency list representation of graph.

References:
http://www.cs.purdue.edu/homes/ayg/CS251/slides/chap9d.pdf

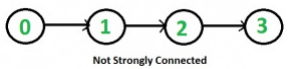
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

28. Connectivity in a directed graph

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 C++ implementation of above algorithm.

```
// Program to check if a given directed graph is strongly connected or
#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()
{
    // St1p 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.
    // Staring 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;
}
```

Output:

```
Yes
No
```

Time Complexity: Time complexity of above implementation is sane as Depth First Search which is O(V+E) if the graph is represented using adjacency matrix representation.

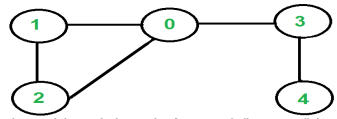
Exercise:
Can we use BFS instead of DFS in above algorithm?

References:
http://www.ieor.berkeley.edu/~hochbaum/files/ieor266-2012.pdf

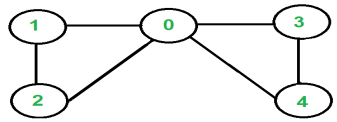
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

29. Eulerian path and circuit for undirected graph

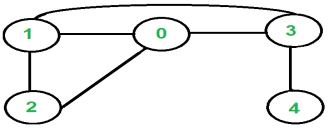
Eulerian Path is a path in graph that visits every edge exactly once. Eulerian Circuit is an Eulerian Path which starts and ends on the same vertex.



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 zero or 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.

```
// 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);
```

```
    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 veritces
    // with zero degree
    Graph g5(3);
    test(g5);

    return 0;
}
```

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)

We will soon be covering following topics on Eulerian Path and Circuit
1) Eulerian Path and Circuit for a Directed Graphs.
2) How to print a Eulerian Path or Circuit?

References:
http://en.wikipedia.org/wiki/Eulerian_path

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

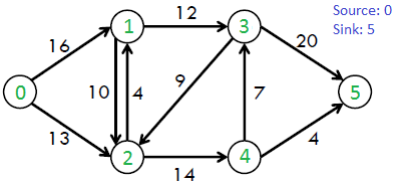
30. Ford-Fulkerson Algorithm for Maximum Flow Problem

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:

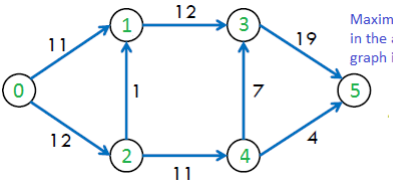
a) Flow on an edge doesn't exceed the given capacity of the edge.

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.




```
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(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(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 to 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 video for example).

http://www.youtube.com/watch?v=-8MwfgB-lyM

Following is C++ implementation of Ford-Fulkerson algorithm. To keep things simple, graph is represented as a 2D matrix.

```
// 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];
    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;
```

```
                visited[v] = true;
            }
        }
    }

    // If we reached sink in BFS starting from source, then return
    // true, else false
    return (visited[t] == true);
}

// Returns tne 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
    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 tere is path from source to sink
    while (bfs(rGraph, s, t, parent))
    {
        // Find minimum residual capacity of the edhes 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,

    return 0;
}
```

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.

Exercise:

Modify the above implementation so that it that 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

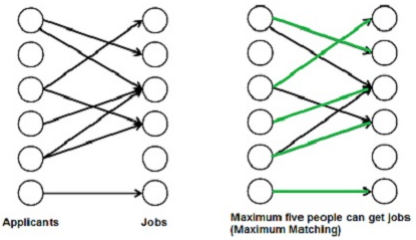
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

31. Maximum Bipartite Matching

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 other words, a matching is maximum 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.

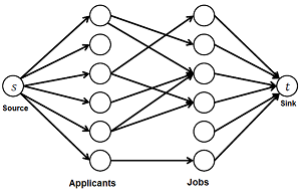
Why do we care?
There are many real world problems that can be formed as Bipartite Matching. For example, consider the following problem:
There are M job applicants and N jobs. Each applicant has a subset of jobs that he/she is interested in. Each job opening can only accept one applicant and a job applicant can be appointed for only one job. Find an assignment of jobs to applicants in such that as many applicants as possible get jobs.



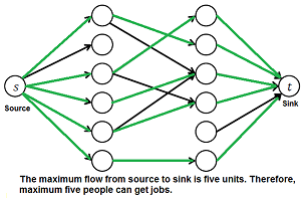
We strongly recommend to read the following post first.

Ford-Fulkerson Algorithm for Maximum Flow Problem

Maximum Bipartite Matching and Max Flow Problem
Maximum **B**ipartite **M**atching (**MBP**) problem can be solved by converting it into a flow network (See this video to know how did we arrive 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.



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 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().

```
// 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])
        {
            seen[v] = true; // Mark v as visited

            // If job 'v' is not assigned to an applicant OR
            // previously assigned applicant for job v (which is matchR
            // has an alternate job available.
            // Since v is marked as visited in the above line, matchR[
```

```
            // in the following recursive call will not get job 'v' ag
            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));

    int result = 0; // Count of jobs assigned to applicants
    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 program to test above functions
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;
}
```

Output:

```
Maximum number of applicants that can get job is 5
```

References:

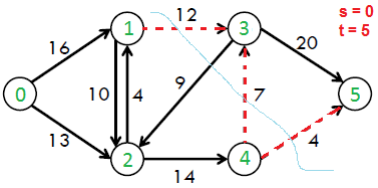
http://www.cs.cornell.edu/~wdtseng/icpc/notes/graph_part5.pdf
http://www.youtube.com/watch?v=NlQqmEXuiC8
http://en.wikipedia.org/wiki/Maximum_matching
http://www.stanford.edu/class/cs97si/08-network-flow-problems.pdf
http://www.cs.princeton.edu/courses/archive/spring13/cos423/lectures/07NetworkFlowII-2×2.pdf
http://www.ise.ncsu.edu/fangroup/or766.dir/or766_ch7.pdf

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

32. Find minimum s-t cut in a flow network

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 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 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 minimum cut.

1) Run Ford-Fulkerson algorithm and consider the final residual graph.

2) Find the set of vertices that are reachable from 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++ 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 (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 fu
// marks visited[i] as true if i is reachable from s.  The initial val
```

```
// visited[] must be false. We can also use BFS to find reachable vert
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 e
    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 tere is path from source to sink
    while (bfs(rGraph, s, t, parent))
    {
        // Find minimum residual capacity of the edhes 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
```

```
// 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;
}
```

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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

33. Fleury's Algorithm for printing Eulerian Path or Circuit

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.
http://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).

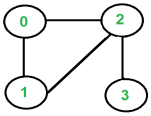
1. Make sure the graph has either 0 or 2 odd vertices.

2. If there are 0 odd vertices, start anywhere. If there are 2 odd vertices, start at one of them.

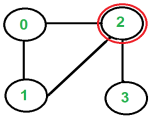
3. Follow edges one at a time. If you have a choice between a bridge and a non-bridge, *always choose the non-bridge*.

4. Stop when you run out of edges.

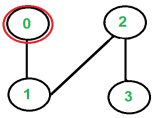
The idea is, ***"don't burn bridges"*** so that we can come back to a vertex and traverse remaining edges. For example let us consider the following graph.



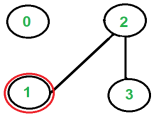
There are two vertices with odd degree, '2' and '3', we can start path from any of them. Let us start tour from vertex '2'.



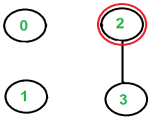
There are three edges going out from vertex '2', which one to pick? We don't pick the edge '2-3′ because that is a bridge (we won't be able to come back to '3'). We can pick any of the remaining two edge. Let us say we pick '2-0′. We remove this edge and move to vertex '0'.



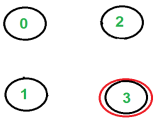
There is only one edge from vertex '0', so we pick it, remove it and move to vertex '1'. Euler tour becomes '2-0 0-1′.



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.

```
// A C++ program print Eulerian Trail in a given Eulerian or Semi-Eule
#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(
  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
  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 adjcent 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;
}
```

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, we will soon be covering them as separate posts.

References:
http://www.math.ku.edu/~jmartin/courses/math105-F11/Lectures/chapter5-part2.pdf
http://en.wikipedia.org/wiki/Eulerian_path#Fleury.27s_algorithm

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

34. Longest Path in a Directed Acyclic Graph

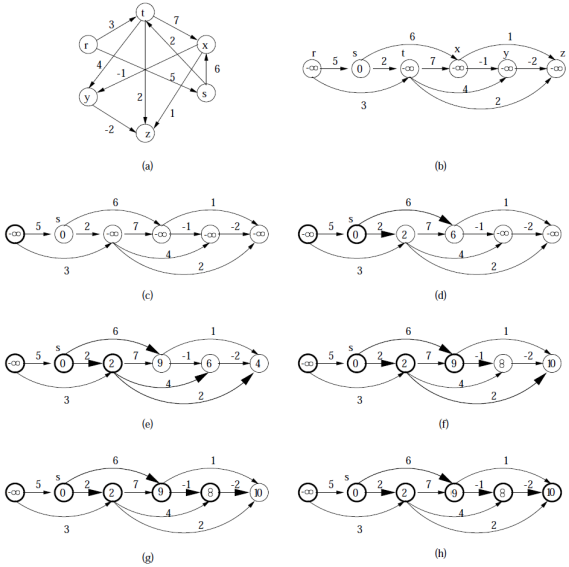
Given a Weighted **D**irected **A**cyclic **G**raph (DAG) and a source vertex s in it, find the longest distances from s to all other vertices in the given graph.

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 Tological 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 dist[] = {NINF, NINF, ….} and dist[s] = 0 where s is the source vertex. Here NINF means negative infinite.
2) Create a toplogical order of all vertices.
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)

Following is C++ implementation of the above algorithm.

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

```
#include <list>
#include <stack>
#include <limits.h>
#define NINF INT_MIN
using namespace std;

// Graph is represented using adjacency list. Every node of adjacency
// contains vertex number of the vertex to which edge connects. It als
// 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 detail
// http://www.geeksforgeeks.org/topological-sorting/
void Graph::topologicalSortUtil(int v, bool visited[], stack<int> &Sta
{
    // 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;
```

```
        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
    // 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
    g.longestPath(s);

    return 0;
}
```

Output:

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

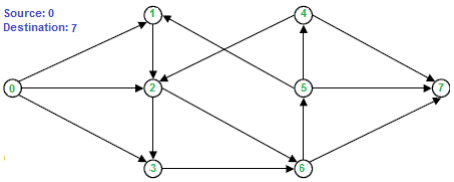
Time Complexity: Time complexity of topological sorting is O(V+E). After finding topological order, the algorithm process all vertices and for every vertex, it runs a loop for all adjacent vertices. Total adjacent vertices in a graph is O(E). So the inner loop runs O(V+E) times. Therefore, overall time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

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

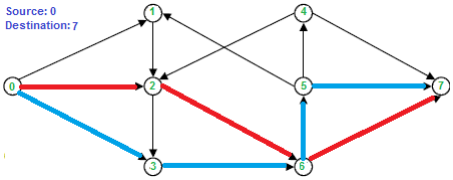
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

35. Find maximum number of edge disjoint paths between two vertices

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 0-3-6-5-7.



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++ 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;
    parent[s] = -1;

    // Standard BFS Loop
```

```
    // 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 tne maximum number of edge-disjoint paths from s to t.
// This function is copy of forFulkerson() discussed at http://goo.gl/
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 i
    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 tere 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;
}
```

Output:

```
There can be maximum 2 edge-disjoint paths from 0 to 7
```

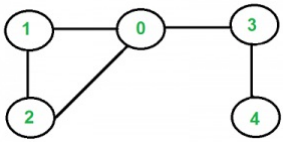
Time Complexity: Same as time complexity of Edmonds-Karp implementation of Ford-Fulkerson (See time complexity discussed here)

References:
http://www.win.tue.nl/~nikhil/courses/2012/2WO08/max-flow-applications-4up.pdf

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

36. Detect cycle in an undirected graph

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(ELogV). 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.

```
// 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 lis
    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;
}
```

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)

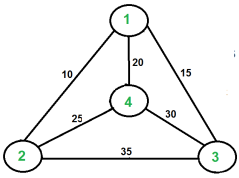
Exercise: Can we use BFS to detect cycle in an undirected graph in O(V+E) time? What about directed graphs?

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

37. Travelling Salesman Problem | Set 1 (Naive and Dynamic Programming)

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 Hamiltoninan 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 know 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: $\Omega(n!)$

Dynamic Programming:
Let the given set of vertices be {1, 2, 3, 4,….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 cost(i), the cost of corresponding Cycle would be cost(i) + dist(i, 1) where dist(i, 1) is the distance from i to 1. Finally, we return the minimum of all [cost(i) + dist(i, 1)] values. This looks simple so far. Now the question is how to get cost(i)?
To calculate 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.

References:
http://www.lsi.upc.edu/~mjserna/docencia/algofib/P07/dynprog.pdf
http://www.cs.berkeley.edu/~vazirani/algorithms/chap6.pdf

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

38. Travelling Salesman Problem | Set 2 (Approximate using MST)

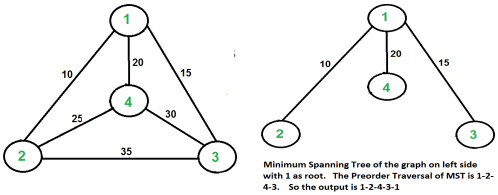
We introduced Travelling Salesman Problem and discussed Naive and Dynamic Programming Solutions for the problem in the previous post,. Both of the solutions are infeasible. In fact, there is no polynomial time solution available for this problem as the problem is a known NP-Hard problem. There are approximate 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., dis(i, j) is always less than or equal to dis(i, k) + 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 **M**inimum **S**panning **T**ree (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-approximateness.
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 prreorder 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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

39. Johnson's algorithm for All-pairs shortest paths

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*VLogV) 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], .. 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]".

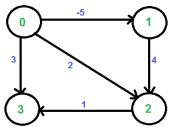
4) Remove the added vertex s and run Dijkstra's algorithm for every vertex.

How does the transformation ensure nonnegative weight edges?
The following property is always true about h[] values as they are shortest distances.

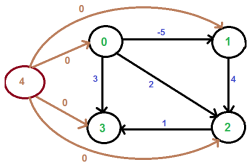
```
h[v] <= h[u] + w(u, v)
```

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] <= h[u] + w(u, v)".

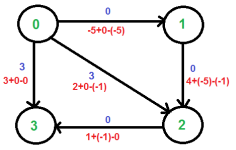
Example:
Let us consider the following graph.



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].



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(VLogV). 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=TV2Z6nbo1ic
http://en.wikipedia.org/wiki/Johnson%27s_algorithm
http://www.youtube.com/watch?v=Sygq1e0xWnM

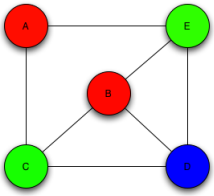
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

40. Graph Coloring | Set 1 (Introduction and Applications)

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 am exam schedule for a university. We have list 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 mean 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) Suduku: Suduku 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 colowing 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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

41. Graph Coloring | Set 2 (Greedy Algorithm)

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 is C++ implementation of the above Greedy Algorithm.

```
// 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
```

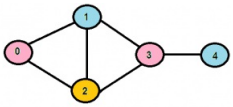
```
        // Reset the values back to false for the next iteration
        for (i = adj[u].begin(); i != adj[u].end(); ++i)
            if (result[*i] != -1)
                available[result[*i]] = false;
    }

    // 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;
}
```

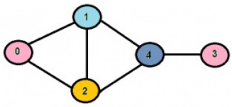
Output:

```
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.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

42. Some interesting shortest path questions | Set 1

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 + VLogV) 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 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/
http://geeksquiz.com/graph-shortest-paths/

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

43. Channel Assignment Problem

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 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 ? |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'
            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 parallely in 1
// time slot from sender to receiver
```

```
// 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

44. Given a sorted dictionary of an alien language, find order of characters

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'
```

We strongly recommend to minimize the browser and try this yourself first.

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 C++ implementation of the above algorithm.

```
// 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> &Sta
{
    // 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 topologicalS
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 characters from a sorted
```

```
// 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 'aplha' 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;
}
```

Output:

```
c a b
```

Time Complexity: The first step to create a graph takes O(n + alhpa) time where n is number of given words and alpha is number of characters in given alphabet. The second step is also topological sorting. Note that there would be *alpha* vertices and at-most (n-1) edges in the graph. The time complexity of topological sorting is O(V+E) which is O(n + aplha) here. So overall time complexity is O(n + aplha) + O(n + aplha) which is O(n + aplha).

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.

45. Given an array of strings, find if the strings can be chained to form a circle

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
```

We strongly recommend to minimize the browser and try this yourself first.

The idea is to create a directed graph of all characters and then find if their is an eulerian circuit in the graph or not. 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 is C++ implementation of the above algorithm.

```
// 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];
```

```
    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
// http://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.
    // Staring 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;
}
```

Output:

```
Can be chained
Can't be chained
```

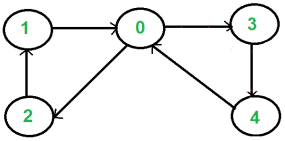
This article is contributed by **Piyush Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

46. Euler Circuit in a Directed Graph

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 an 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 is C++ implementation of above approach.

```
// 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
// http://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.
    // Staring 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;
}
```

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.

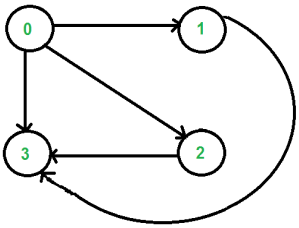
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

47. Count all possible walks from a source to a destination with exactly k edges

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 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}



We strongly recommend to minimize the browser and try this yourself first.

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 C++ implementation of this simple solution.

```
// 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])  // 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;
}
```

Output:

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++ 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 && 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 th
                if (e > 1)
                {
                    for (int a = 0; a < V; a++) // adjacent of source
                        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;
}
```

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 $O(V^3Logk)$ time. The count of walks of length k from u to v is the [u][v]'th entry in $(graph[V][V])^k$. We can calculate power of by doing O(Logk) multiplication by using the divide and conquer technique to calculate power. A multiplication between two matrices of size V x V takes $O(V^3)$ time. Therefore overall time complexity of this method is $O(V^3Logk)$.

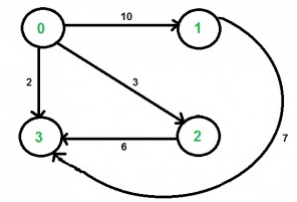
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

48. Shortest path with exactly k edges in a directed and weighted graph

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 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 is C++ implementation of this simple solution.

```
// 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 inifinite 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;
}
```

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.

```
// 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 inifinite 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 stor
    // 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 tha
                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;
}
```

Output:

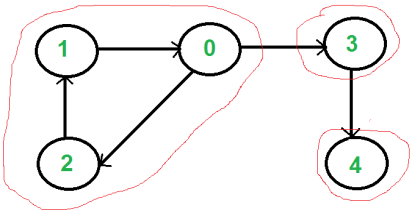
```
Weight of the shortest path is 9
```

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

49. Tarjan's Algorithm to find Strongly Connected Components

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, 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 disc[u] = low[u].

Disc and Low Values
(click on image to see it properly)

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 it's 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 it's 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].
low[u] = min(low[u], 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].
low[u] = min(low[u], 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 C++ implementation of Tarjan's algorithm to print all SCCs.

```
// 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 of '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;
}
```

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

50. Snake and Ladder Problem

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 on right side (taken from here), the minimum number of dice throws required to reach cell 30 from cell 1 is 3. Following are 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.

We strongly recommend to minimize the browser and try this yourself first.
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 C++ 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++ 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, I
    return 0;
}
```

Output:

```
Min Dice throws required is 3
```

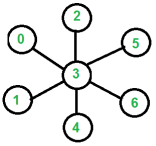
Time complexity of the above solution is O(N) as every cell is added and removed only once from queue. And a typical enqueue or dequeue operation takes O(1) time.

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.

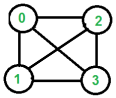
51. Vertex Cover Problem | Set 1 (Introduction and Approximate Algorithm)

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***.

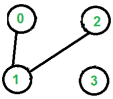
Following are some examples.



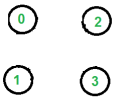
Minimum Vertex Cover is {3}



Minimum Vertex Cover is {0, 1, 2} or {0, 1, 3} or {1, 2, 3}



Minimum Vertex Cover is {1}



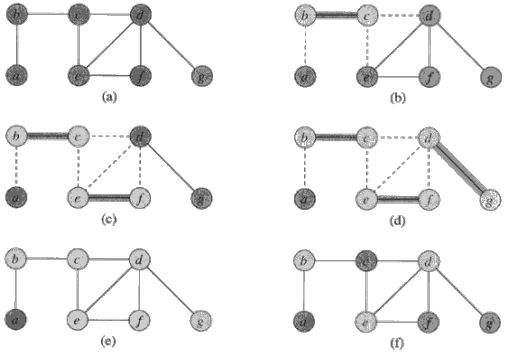
Minimum Vertex Cover is empty {}

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
```

Following diagram taken from CLRS book shows execution of above approximate algorithm.



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 is C++ implementation of above approximate algorithm.

```
// 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;
}
```

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(LogV) (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

52. Biconnected Components

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.

```
// A C++ program to find biconnected components in a given undirected g
#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
```

```
    // 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 of '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])
        {
            low[u]  = min(low[u], disc[v]);
            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;
}
```

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
```

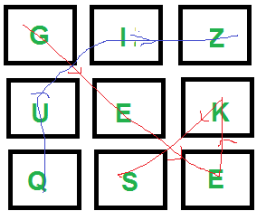
This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

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

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 charactersNote 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
```



We strongly recommend to minimize your browser and try this yourself first.

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 givem 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
```

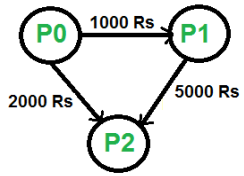
Note that the above solution may print same word multiple times. For example, if we add “SEEK” to dictionary, it is printed multiple times. To avoid this, we can use hashing to keep track of all printed words.

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.

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

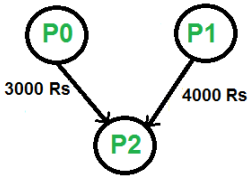
Given a number of friends who have to give or take some amount of money from one another. Design an algorithm by which the total cash flow among all the friends is minimized.

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



P1 pays 4000 Rs to P2
P0 pays 3000 Rs to P2

We strongly recommend to minimize your browser and try this yourself first.

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 Pi 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 be 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 Pd and maximum creditor be Pc.

3) Find the minimum of maxDebit and maxCredit. Let minimum of two be x. Debit 'x' from Pd and credit this amount to Pc

4) If x is equal to maxCredit, then remove Pc from set of persons and recur for remaining (n-1) persons.

5) If x is equal to maxDebit, then remove Pd 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 C++ implementation of above algorithm.

```
// 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[]
```

```
// 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[mxDebit] 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;
}
```

Output:

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

Algorithmic Paradigm: Greedy

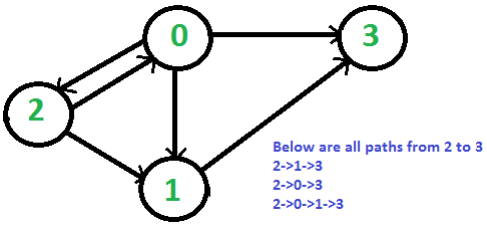
Time Complexity: O(N^2) where N is the number of persons.

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

55. Print all paths from a given source to a destination

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.



We strongly recommend to minimize your browser and try this yourself first

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 C++ implementation of above idea.

```
// 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[]
```

```
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;
}
```

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

56. Optimal read list for given number of days

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.

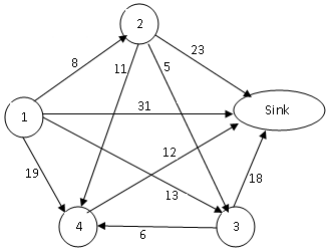
```
Example 1:
Input:  Number of Days to Finish book = 2
        Number of pages in chapters = {10, 5, 5}
Output: Day 1:  Chapter 1
        Day 2:  Chapters 2 and 3

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 2:  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 “abs(8-10) + abs(5+6-10) + abs(12-10)” which is 5. The value 5 is the optimal value of sum of differences.

We strongly recommend to minimize the browser and try this yourself first.

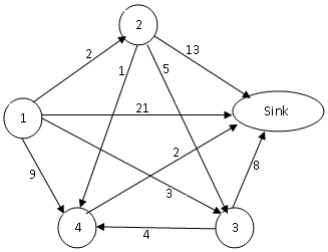
Consider the example 2 above where a book has 4 chapters with pages 8, 5, 6 and 12. User wishes to finish it in 3 days. The graphical representation of the above scenario is,



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 |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
```

```
// 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
// 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[DAYS+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 funtion 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.

57. Applications of Breadth First Traversal

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 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 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 Bread 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 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 algorithm 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

58. Find the minimum cost to reach destination using a train

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 C++ implementation of above recursive formula.

```
// 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;
}
```

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 subrpoblems 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., dist[0] = 0

2) The min cost for station 1 is cost[0][1], i.e., dist[1] = cost[0][1]

3) The min cost for station 2 is minimum of following two.
   a) dist[0] + cost[0][2]
   b) dist[1] + cost[1][2]

3) The min cost for station 3 is minimum of following three.
   a) dist[0] + cost[0][3]
   b) dist[1] + cost[1][3]
   c) dist[2] + cost[2][3]

Similarly, dist[4], dist[5], ... dist[N-1] are calculated.

Below is C++ implementation of above idea.

```
// 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[][N])
{
    // 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;
}
```

Output:

```
The Minimum cost to reach station 4 is 65
```

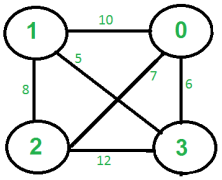
This article is contributed by **Udit Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

59. K Centers Problem | Set 1 (Greedy Approximate Algorithm)

Given n cities and distances between every pair of cities, select k cities to place

warehouses (or ATMs) such that the maximum distance of a city to a warehouse (or ATM) 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.



k = 2

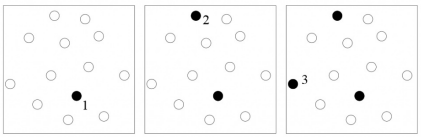
The two ATMs should be placed in cities 2 and 3. The maximum distance of a city from an ATM becomes 6 in this optimal placement (We can not get the maximum distance less than 7)

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 that 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 c1, c2, c3, … ci 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
Min[dist(p, c1), dist(p, c2), dist(p, c3), …. dist(p, ci)]

The following diagram taken from here illustrates above algorithm.



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 distanced from 2 to already considered centers
Min[dist(2, 0), dist(2, 1)] = Min[7, 8] = 7

Minimum of all distanced from 3 to already considered centers
Min[dist(3, 0), dist(3, 1)] = 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*OPT.

The proof can be done using contradiction.

a) Assume that the distance from the furthest point to all centers is > 2·OPT.

b) This means that distances between all centers are also > 2·OPT.

c) We have k + 1 points with distances > 2·OPT between every pair.

d) Each point has a center of the optimal solution with distance ? 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·OPT (triangle inequality) which is a contradiction.

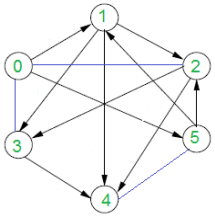
Source:
http://algo2.iti.kit.edu/vanstee/courses/kcenter.pdf

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

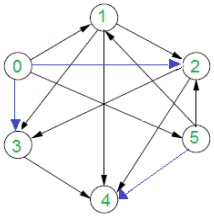
60. Assign directions to edges so that the directed graph remains acyclic

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.



Given Graph

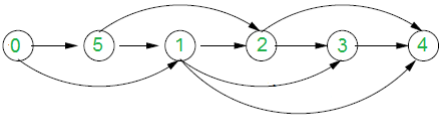


Graph after adding directions to undirected edges such that the graph remains acyclic.

We strongly recommend to minimize your browser and try this yourself first.

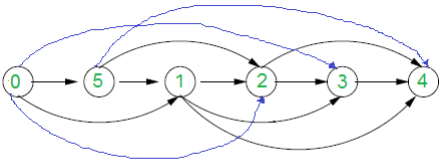
The idea is to use Topological Sorting. Following are two steps used in the algorithm.

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.