

Strivers A2Z DSA Course/Sheet

 takeuforward.org/strivers-a2z-dsa-course/strivers-a2z-dsa-course-sheet-2

June 20, 2022



▼ Step 15: Graphs [Concepts & Problems]

(0/54)

▼ Step 15.1: Learning

(0/6)

Action	Problem [Articles, Codes]	PL-1	PL-2	Solution	Notes	Company
	Graph and Types					
	Graph Representation C++					
	Graph Representation Java					
	Connected Components Logic Explan...					
	BFS					
	DFS					

▼ Step 15.2: Problems on BFS/DFS

(0/14)

Action	Problem [Articles, Codes]	PL-1	Solution	PL-2	Notes	Company
	Number of provinces (leetcode)					
	Connected Components Problem in Mat...					
	Rotten Oranges					
	Flood fill					
	Cycle Detection in unirected Graph					
	...					
	Cycle Detection in undirected Graph...					
	0/1 Matrix (Bfs Problem)					
	Surrounded Regions (dfs)					
	Number of Enclaves [flood fill impl...]					
	Word ladder - 1					
	Word ladder - 2					
	Number of Distinct Islands [dfs mul...]					
	Bipartite Graph (DFS)					
	Cycle Detection in Directed Graph (...)					

▼ Step 15.3: Topo Sort and Problems

(0/7)

Action	Problem [Articles, Codes]	PL-1	Solution	PL-2	Notes	Company
	Topo Sort					
	Kahn's Algorithm					

Action	Problem [Articles, Codes]	PL-1	Solution	PL-2	Notes	Company
	Cycle Detection in Directed Graph (...)					
	Course Schedule - I					
	Course Schedule - II					
	Find eventual safe states					
	Alien dictionary					

▼ Step 15.4: Shortest Path Algorithms and Problems
(0/13)

Action	Problem [Articles, Codes]	PL-1	Solution	PL-2	Notes	Company
	Shortest Path in UG with unit weigh...					
	Shortest Path in DAG					
	Djisktra's Algorithm					
	Why priority Queue is used in Djisk...					
	Shortest path in a binary maze					
	Path with minimum effort					
	Cheapest flights within k stops					
	Network Delay time					
	Number of ways to arrive at destina...					
	Minimum steps to reach end from sta...					
	Bellman Ford Algorithm					
	Floyd Warshal Algorithm					
	Find the city with the smallest num...					

▼ Step 15.5: Minimum Spanning Tree/Disjoint Set and Problems

(0/11)

Action	Problem [Articles, Codes]	PL-1	Solution	PL-2	Notes	Company
	Minimum Spanning Tree					
	Prim's Algorithm					
	Disjoint Set [Union by Rank]					
	Disjoint Set [Union by Size]					
	Kruskal's Algorithm					
	Number of operations to make network...					
	Most stones removed with same rows ...					
	Accounts merge					
	Number of island II					
	Making a Large Island					
	Swim in rising water					

▼ Step 15.6: Other Algorithms

(0/3)

Action	Problem [Articles, Codes]	PL-1	Solution	PL-2	Notes	Company
	Bridges in Graph					
	Articulation Point					
	Kosaraju's Algorithm					

► : Strings

Hurrah!! You are ready for your placement after some months of hard work! All the best, keep striving...

Striver

Share the course/sheet with your friends, created with love for takeUforward fam!

If you find any mistakes in the sheet, it can be a wrong link as well, please fill out the google form [here](#), our team will check it on a weekly basis, thanks.

Introduction to Graph

 takeuforward.org/graph/introduction-to-graph

August 4, 2022

What is a graph data structure?

There are two types of data structures

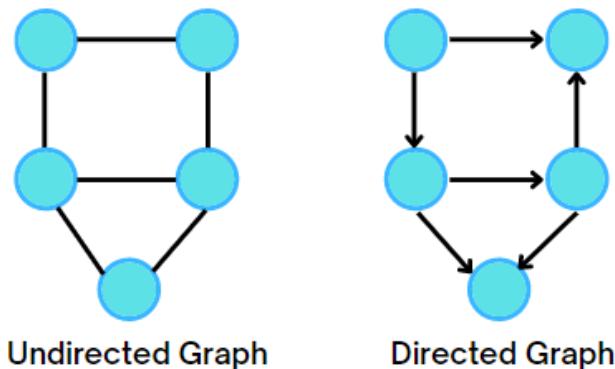
1. Linear
2. Non – linear

We are aware of linear data structures such as arrays, stacks, queues, and linked lists. They are called linear because data elements are arranged in a linear or sequential manner.

The only non-linear data structure that we've seen so far is Tree. In fact, a tree is a special type of graph with some restrictions. Graphs are data structures that have a wide-ranging application in real life. These include analysis of electrical circuits, finding the shortest routes between two places, building navigation systems like Google Maps, even social media using graphs to store data about each user, etc. To understand and use the graph data structure, let's get familiar with the definitions and terms associated with graphs.

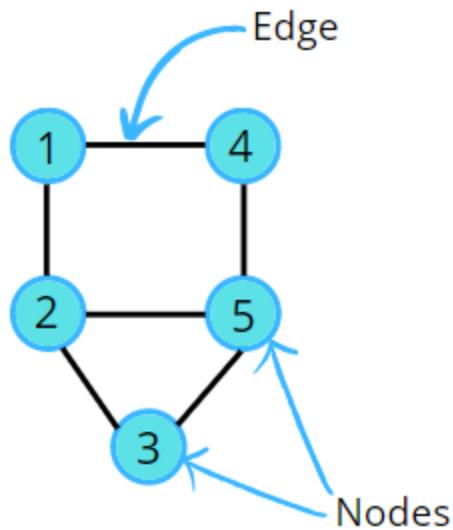
Definitions and Terminology

A graph is a non-linear data structure consisting of nodes that have data and are connected to other nodes through edges.



Nodes are circles represented by numbers. Nodes are also referred to as vertices. They store the data. The numbering of the nodes can be done in any order, no specific order needs to be followed.

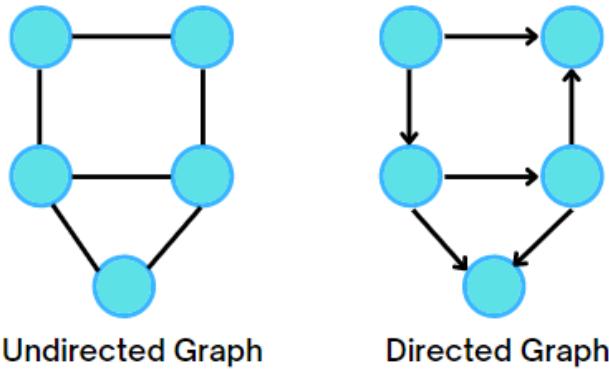
In the following example, the number of nodes or vertices = 5



Two nodes are connected by a horizontal line called **Edge**. Edge can be directed or undirected. Basically, pairs of vertices are called edges.

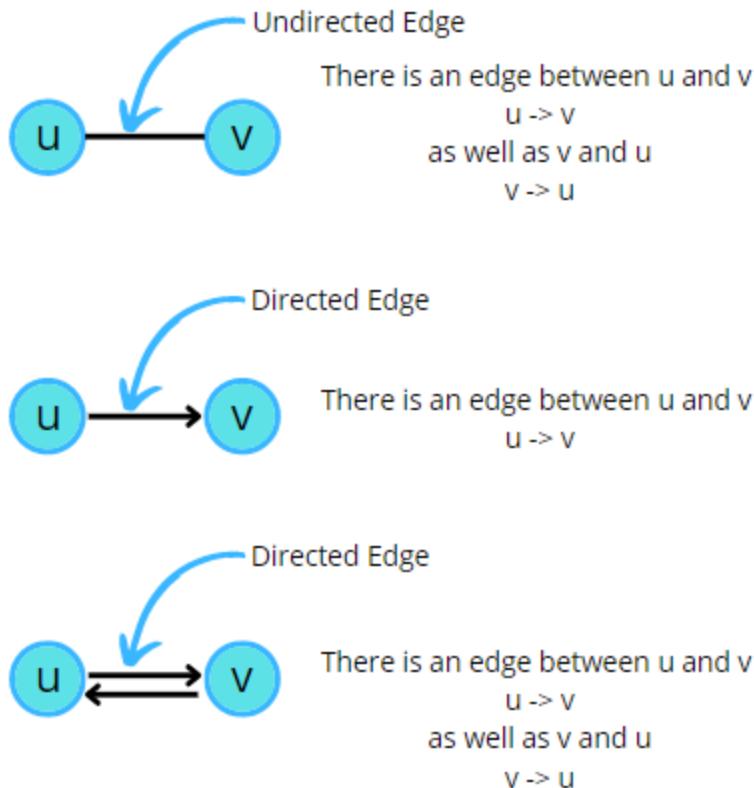
In the above example, the edge can go from 1 to 4 or from 4 to 1, i.e. a bidirectional edge can be in both directions, hence called an **undirected edge**. Thus, the pairs (1,4) and (4,1) represent the same edge.

Types of Graphs



1. An **undirected graph** is a graph where edges are bidirectional, with no direction associated with them, i.e, there will be an undirected edge. In an undirected graph, the pair of vertices representing any edge is unordered. Thus, the pairs (u, v) and (v, u) represent the same edge.
2. A **directed graph** is a graph where all the edges are directed from one vertex to another, i.e, there will be a directed edge. It contains an ordered pair of vertices. It implies each edge is represented by a directed pair $\langle u, v \rangle$. Therefore, $\langle u, v \rangle$ and $\langle v, u \rangle$ represent two different edges.

There can be multi-directed edges, hence bidirectional edges, as shown in the example below.



Structure of Graph

Does every graph have a cycle?

The answer is No! Let us consider the following examples to understand this.

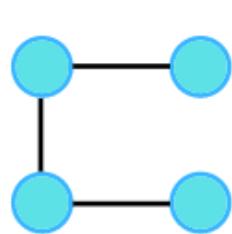


Fig. 1

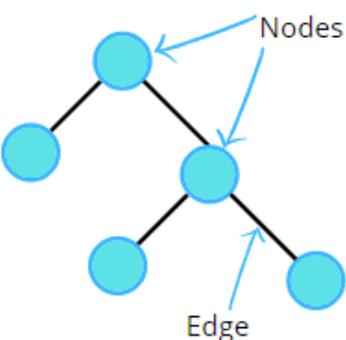
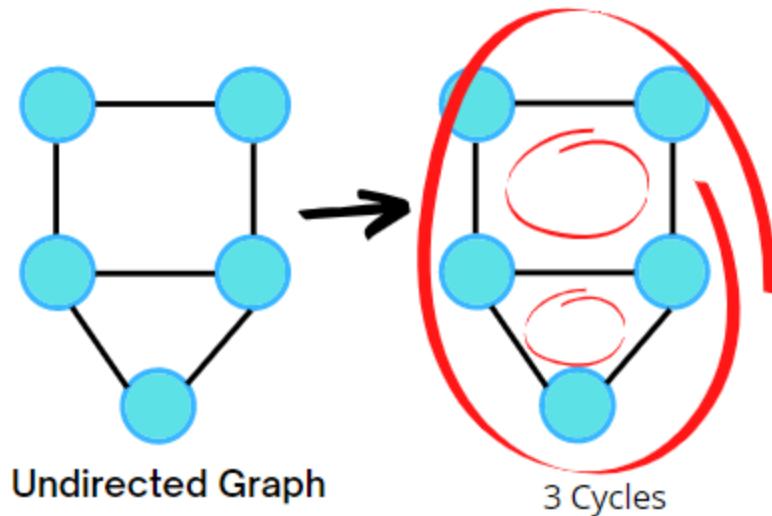


Fig. 2

Fig. 1 does not form a cycle but still, it is a graph.

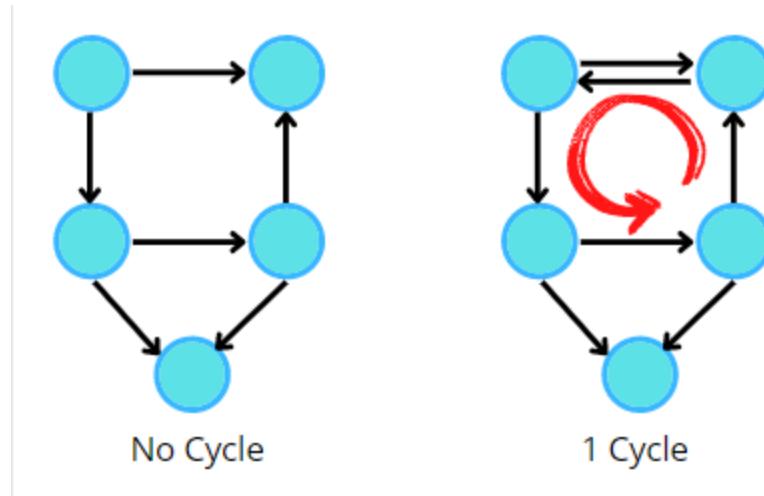
Fig. 2 is an example of a binary tree. It can also be called a graph because it follows all the rules. We've nodes and edges, and this is the minimal condition to be called a graph.

So a graph does not necessarily mean to be an enclosed structure, it can be an open structure as well. A graph is said to have a cycle if it starts from a node and ends at the same node. There can be multiple cycles in a graph.



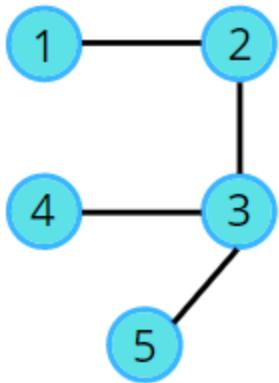
If there is at least one cycle present in the graph then it is called an **Undirected Cyclic Graph**.

In the following examples of directed graphs, the first directed graph is not cyclic as we can't start from a node and end at the same node. Hence it is called **Directed Acyclic Graph**, commonly called **DAG**.



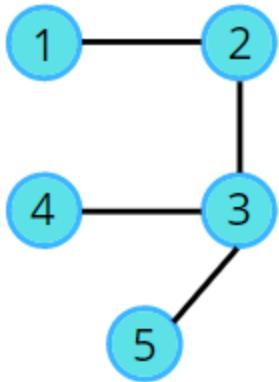
If we just add an edge to the directed graph, then at least one cycle is present in the graph, hence it becomes **Directed Cyclic Graph**.

Path in a Graph



The path contains a lot of nodes and each of them is reachable.

Consider the given graph,



1 2 3 5 is a path.

1 2 3 2 1 is not a path, because a node can't appear twice in a path.

1 3 5 is not a path, as adjacent nodes must have an edge and there is no edge between 1 and 3.

Degree of Graph

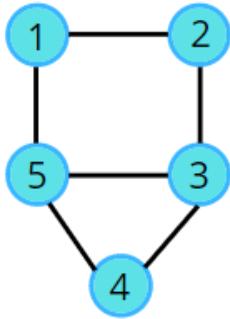
It is the number of edges that go inside or outside that node.

For **undirected graphs**, the degree is the number of edges attached to a node.

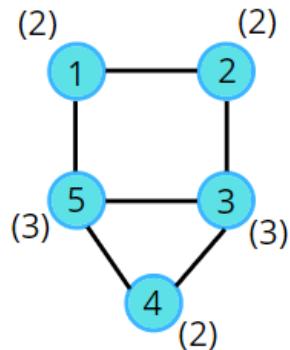
Example,

$$D(3) = 3$$

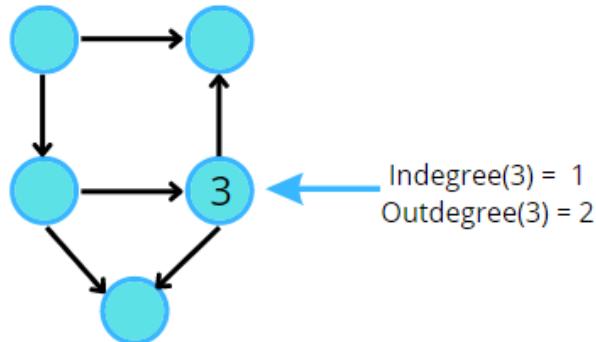
$$D(4) = 2$$



Property: It states that the total degree of a graph is equal to twice the number of edges. This is because every edge is associated/ connected to two nodes.



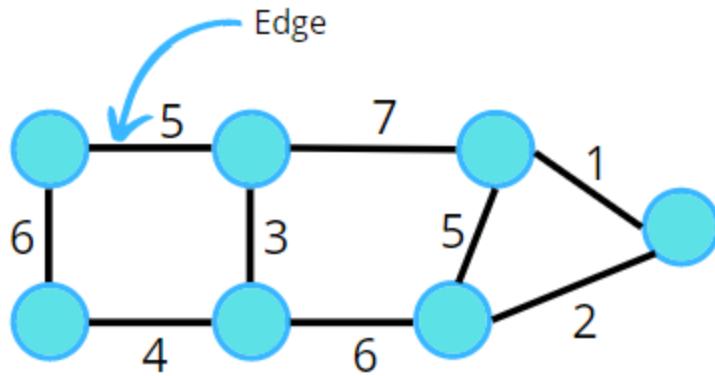
Total Degree of a graph = $2 \times E$
 Example, $(2+2+3+2+3) = 2 \times 6 \Rightarrow 12 = 12$



For **directed graphs**, we've Indegree and Outdegree. **The indegree** of a node is the number of incoming edges. **The outdegree** of a node is the number of outgoing edges.

Edge Weight

A graph may have weights assigned on its edges. It is often referred to as the cost of the edge.



If weights are not assigned then we assume the unit weight, i.e, 1. In applications, weight may be a measure of the cost of a route. For example, if vertices A and B represent towns in a road network, then weight on edge AB may represent the cost of moving from A to B, or vice versa.

*Special thanks to **Vanshika Singh Gour** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article. If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com*



Watch Video At: https://youtu.be/M3_pLsDdeuU

Graph Representation in C++

In this article, we are going to study the different ways of representing a graph in memory, but before that first, let us understand how to take the input of the graph.

Input Format

In the question, they will mention whether it is a directed or undirected graph. The first line contains two space-separated integers n and m denoting the number of nodes and the number of edges respectively. Next m lines contain two integers u and v representing an edge between u and v . In the case of an undirected graph if there is an edge between u and v , it means there is an edge between v and u as well. Now the question arises if there is any boundation on the number of edges, i.e., the value of m ? The answer is NO. If we add more edges, then the value of m will increase.

Graph Representations

After understanding the input format, let us try to understand how the graph can be stored. The two most commonly used representations for graphs are

1. Adjacency Matrix
2. Adjacency Lists

Adjacency Matrix

An adjacency matrix of a graph is a two-dimensional array of size $n \times n$, where n is the number of nodes in the graph, with the property that $a[i][j] = 1$ if the edge (v_i, v_j) is in the set of edges, and $a[i][j] = 0$ if there is no such edge.

Consider the example of the following undirected graph,

Input:

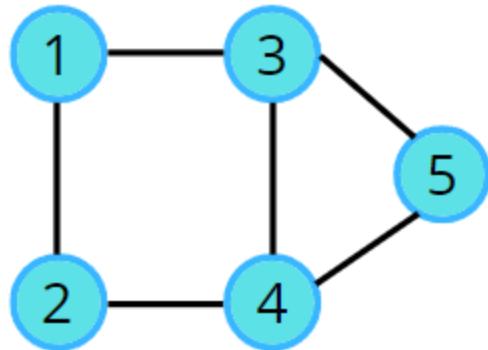
```
5 6  
1 2  
1 3  
2 4  
3 4  
3 5  
4 5
```

Explanation:

Number of nodes, $n = 5$

Number of edges, $m = 6$

Next m lines represent the edges.



We need to store these edges so that future algorithms can be performed. Are the nodes zero-based or one-based? In this case, the nodes follow one-based indexing as the last node is 5 and the total number of nodes is also 5. Now, define an adjacency matrix of size $(n+1) \times (n+1)$, i.e., $\text{adj}[n+1][n+1]$. If there is an edge between 1 and 2, mark 1 at $(1,2)$ and $(2,1)$ as there is an edge between 2 and 1 as well (in the case of an undirected graph). Similarly, follow for other edges.

	0	1	2	3	4	5
0						
1			1			
2		1				
3						
4						
5						

1 2
 1 3
 2 4
 3 4
 3 5
 4 5

All the edges are marked in the adjacency matrix, remaining spaces in the matrix are marked as zero or left as it is.

	0	1	2	3	4	5
0						
1			1	1		
2		1			1	
3		1			1	1
4			1	1		1
5		○		1	1	

This matrix will tell if there is an edge between two particular nodes. For example, there is an edge between 5 and 3 as 1 is at (5,3) but there is no edge between 5 and 1 as the space is empty (or can be filled with 0) at position (5,1) in the adjacency matrix.

The space needed to represent a graph using its adjacency matrix is n^2 locations. Space complexity = $(n \times n)$, It is a costly method as n^2 locations are consumed.

Code:

C++ Code

```

using namespace std;

int main()
{
    int n, m;
    cin >> n >> m;
    // adjacency matrix for undirected graph
    // time complexity: O(n)
    int adj[n+1][n+1];
    for(int i = 0; i < m; i++)
    {
        int u, v;
        cin >> u >> v;
        adj[u][v] = 1;
        adj[v][u] = 1 // this statement will be removed in case of directed graph
    }
    return 0;
}

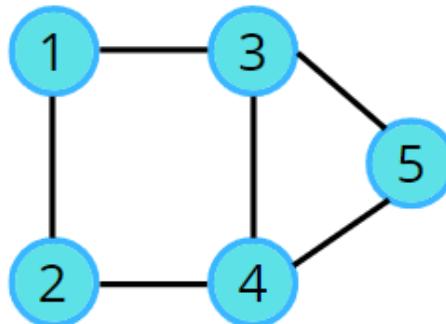
```

Adjacency Lists

In the previous storing method, we saw it was taking n^2 space to store the graph, this is where the adjacency list comes into the picture, it takes a very less amount of space.

This is a node-based representation. In this representation, we associate with each node a list of nodes adjacent to it. Normally an array is used to store the nodes. The array provides random access to the adjacency list for any particular node.

Consider the example of the following **undirected graph**,



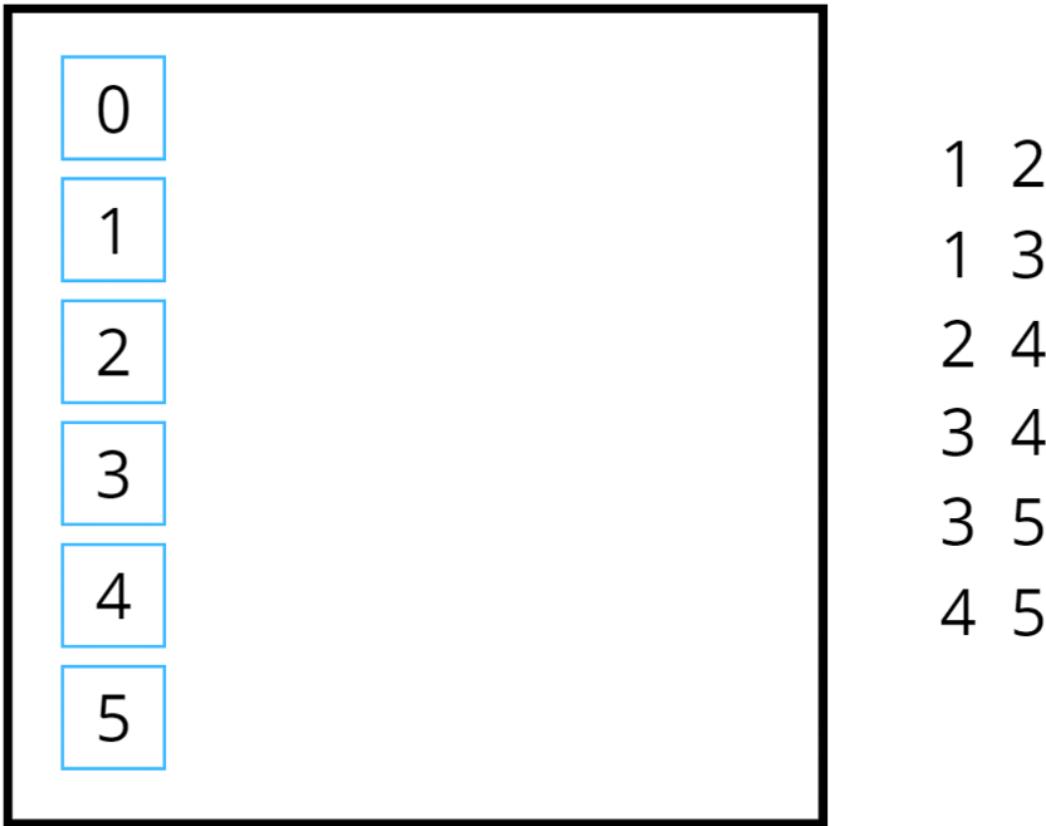
To create an adjacency list, we will create an array of size $n+1$ where n is the number of nodes. This array will contain a list, so in C++ list is nothing but the vector of integers.

```
vector <int> adj[n+1];
```

Now every index is containing an empty vector/ list. With respect to the example, 6 indexes contain empty vectors.

What is the motive of the list?

In the example, we can clearly see that node 4 has nodes 2, 3, and 5 as its adjacent neighbors. So, to store its immediate neighbors in any order, we use the list.



Hence, we stored all the neighbors in the particular indexes. In this representation, for an undirected graph, each edge data appears twice. For example, nodes 1 and 2 are adjacent hence node 2 appears in the list of node 1, and node 1 appears in the list of node 2. So, the space needed to represent an undirected graph using its adjacency list is $2 \times E$ locations, where E denotes the number of edges.

Space complexity = $O(2xE)$

This representation is much better than the adjacency matrix, as matrix representation consumes n^2 locations, and most of them are unused.

Code:

C++ Code

```

#include <iostream>

using namespace std;

int main()
{
    int n, m;
    cin >> n >> m;
    // adjacency list for undirected graph
    // time complexity: O(2E)
    vector<int> adj[n+1];
    for(int i = 0; i < m; i++)
    {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    return 0;
}

```

For **directed graphs**, if there is an edge between u and v it means the edge only goes from u to v, i.e., v is the neighbor of u, but vice versa is not true. The space needed to represent a directed graph using its adjacency list is E locations, where E denotes the number of edges, as here each edge data appears only once.

Space complexity = $O(E)$

Code:

C++ Code

```

#include <iostream>

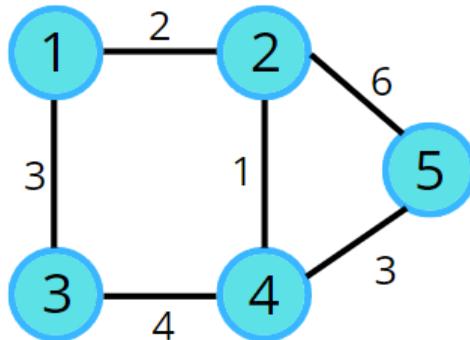
using namespace std;

int main()
{
    int n, m;
    cin >> n >> m;
    // adjacency list for directed graph
    // time complexity: O(E)
    vector<int> adj[n+1];
    for(int i = 0; i < m; i++)
    {
        int u, v;
        // u -> v
        cin >> u >> v;
        adj[u].push_back(v);
    }
    return 0;
}

```

Weighted Graph Representation

As of now, we were considering graphs with unit weight edges (i.e., if there is an edge between two nodes then the weight on the edge is unit weight), now what if there are weights on its edges as shown in the following example?



For the **adjacency matrix**, it is much simpler.

Undirected Graph

```

int u, v, wt;
cin >> u >> v >> wt;
adj[u][v] = wt;
adj[v][u] = wt;

```

Directed Graph

```

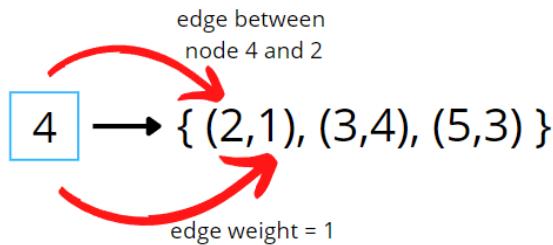
int u, v, wt;
cin >> u >> v >> wt;
adj[u][v] = wt;

```

But how are we going to implement it in the **adjacency list**?

Earlier in the adjacency list, we were storing a list of integers in each index, but for weighted graphs, we will store pairs (node, edge weight) in it.

```
vector< pair <int,int> > adjList[n+1];
```



Special thanks to Vanshika Singh Gour for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article. If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com



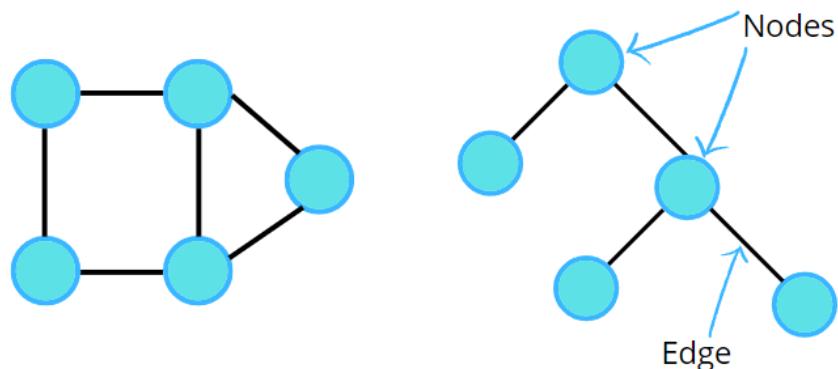
Watch Video At: <https://youtu.be/3oI-34aPMWM>

Connected Components in Graphs

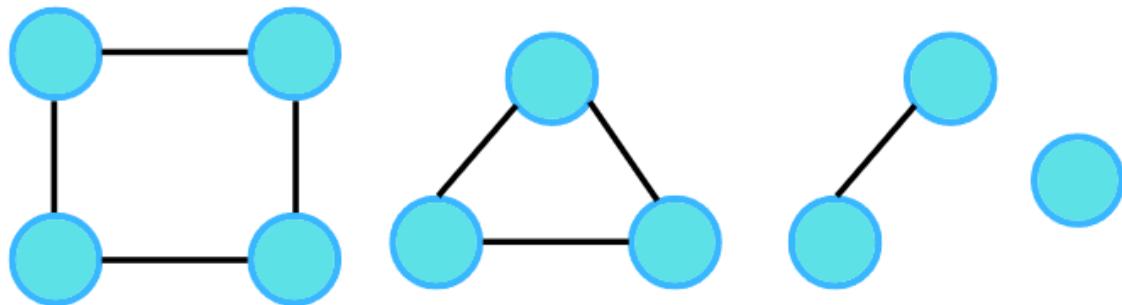
 takeuforward.org/graph/connected-components-in-graphs

August 6, 2022

So far we've seen different types of graphs. Graphs can be connected or can be like a binary tree (as we know all trees are graphs with some restrictions) as shown in the following figure.



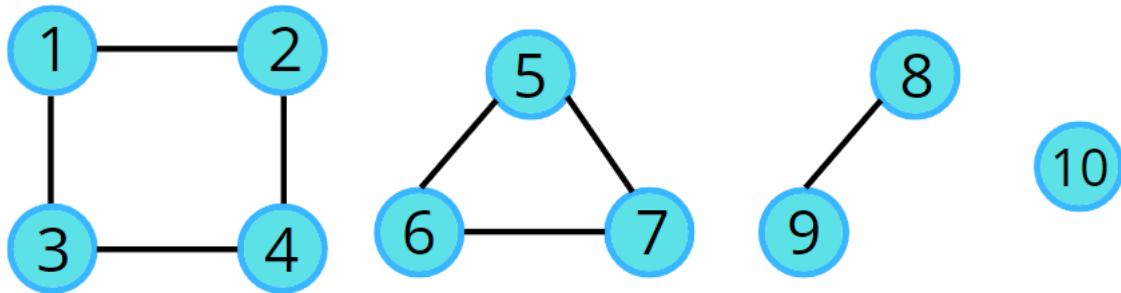
But what would you call the following figure?



The most common answer would be these are 4 different graphs as they are not connected.

But is it possible to call them a single graph? To answer this, let us consider the question given:

Given an undirected graph with 10 nodes and 8 edges. The edges are (1,2), (1,3), (2,4), (4,3), (5,6), (5,7), (6,7), (8,9). The graph that can be formed with the given information is as follows:



Apparently, it's a graph, which is in 4 pieces, the last one being a single node. In this case, we can say, the graph has been broken down into 4 different **connected components**. So next time if you see two different parts of a graph and they are not connected, then do not say that it cannot be a single graph. In the above example, they can be 4 different graphs but according to the given question and the input, we can call them parts of a single graph.

Graph Traversal

In the upcoming topics, we'll be learning about a lot of algorithms. Now, assume a traversal algorithm. Any traversal algorithm will always use a **visited array**.

0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0

For the same example, we will create an array of size 11 ($n+1$) starting with the zeroth index. Initialize this visited array to zero, indicating that all the nodes are unvisited. Then follow the following algorithm. If a node is not visited, then call the traversal algorithm.

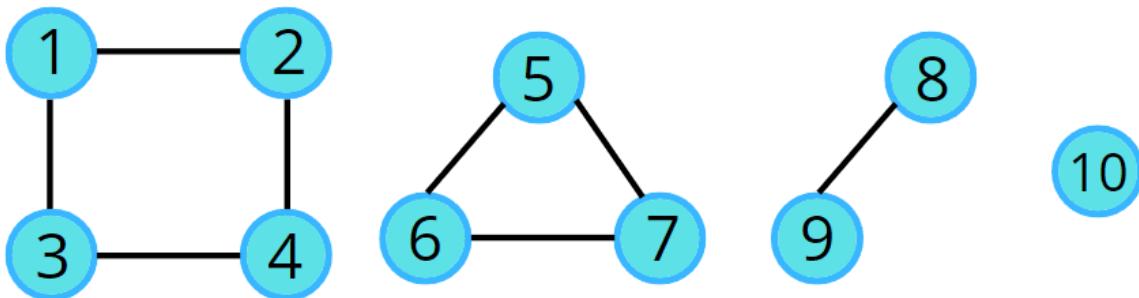
```

for i:= 1 to 10
    if( !visisted[ i ] )
        traversal( i );
    
```

Why can't we just call traversal(1)?

We cannot just call `traversal(node)` because a graph can have multiple components and traversal algorithms are designed in such a way that they will traverse the entire connected portion of the graph. For example, `traversal(1)` will traverse only the connected nodes, i.e., nodes 2, 3, and 4, but not the connected components.

Consider the following illustration to understand how a traversal algorithm will traverse the connected components.



0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0

```
for i:= 1 to 10
    if( !visited[ i ] )
        traversal( i );
```

Special thanks to Vanshika Singh Gour for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article. If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com



Watch Video At: https://youtu.be/lea-WI_uWXY

Breadth First Search (BFS): Level Order Traversal

 takeuforward.org/graph/breadth-first-search-bfs-level-order-traversal

August 7, 2022

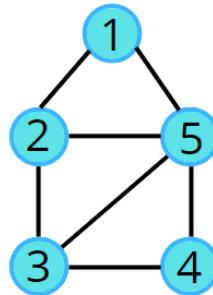
Problem Statement: Given an undirected graph, return a vector of all nodes by traversing the graph using breadth-first search (BFS).

Pre-req: Graph Representation, Queue STL

Examples:

Example 1:

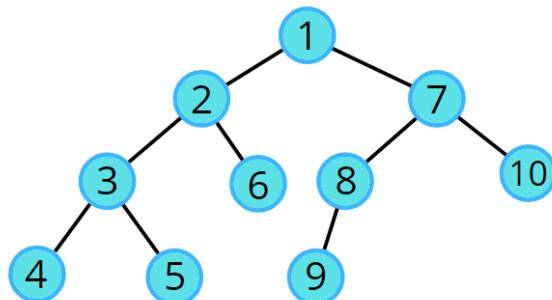
Input:



Output: 1 2 5 3 4

Example 2:

Input:



Output: 1 2 7 3 6 8 10 4 5 9

Solution

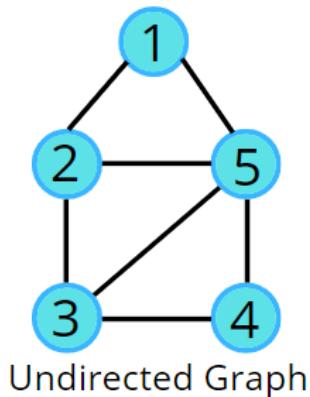
Disclaimer: Don't jump directly to the solution, try it out yourself first.

Approach:

Initial Configuration:

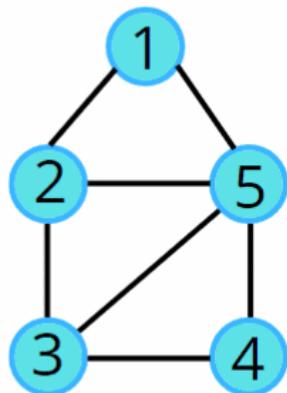
- Queue data structure: follows FIFO, and will always contain the starting.
 - Visited array: an array initialized to 0
1. In BFS, we start with a “starting” node, mark it as visited, and push it into the queue data structure.
 2. In every iteration, we pop out the node ‘v’ and put it in the solution vector, as we are traversing this node.
 3. All the unvisited adjacent nodes from ‘v’ are visited next and are pushed into the queue. The list of adjacent neighbors of the node can be accessed from the adjacency list.
 4. Repeat steps 2 and 3 until the queue becomes empty, and this way you can easily traverse all the nodes in the graph.

In this way, all the nodes are traversed in a breadthwise manner.



0	→	{ }
1	→	{ 2, 5 }
2	→	{ 1, 5, 3 }
3	→	{ 2, 4, 5 }
4	→	{ 3, 5 }
5	→	{ 1, 2, 3, 4 }

Adjacency List



0	1	2	3	4	5
0	0	0	0	0	0

Visited Array

Print:



Queue

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
    // Function to return Breadth First Traversal of given graph.
    vector<int> bfsOfGraph(int V, vector<int> adj[]) {
        int vis[V] = {0};
        vis[0] = 1;
        queue<int> q;
        // push the initial starting node
        q.push(0);
        vector<int> bfs;
        // iterate till the queue is empty
        while(!q.empty()) {
            // get the topmost element in the queue
            int node = q.front();
            q.pop();
            bfs.push_back(node);
            // traverse for all its neighbours
            for(auto it : adj[node]) {
                // if the neighbour has previously not been visited,
                // store in Q and mark as visited
                if(!vis[it]) {
                    vis[it] = 1;
                    q.push(it);
                }
            }
        }
        return bfs;
    }

    void addEdge(vector <int> adj[], int u, int v) {
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    void printAns(vector <int> &ans) {
        for (int i = 0; i < ans.size(); i++) {
            cout << ans[i] << " ";
        }
    }
}

int main()
{
    vector <int> adj[6];

    addEdge(adj, 0, 1);
    addEdge(adj, 1, 2);
    addEdge(adj, 1, 3);
    addEdge(adj, 0, 4);
}

```

```
Solution obj;
vector <int> ans = obj.bfsOfGraph(5, adj);
printAns(ans);

return 0;
}
```

Output: 0 1 4 2 3

Time Complexity: $O(N) + O(2E)$, Where N = Nodes, 2E is for total degrees as we traverse all adjacent nodes.

Space Complexity: $O(3N) \sim O(N)$, Space for queue data structure visited array and an adjacency list

Special thanks to [Vanshika Singh Gour](#) for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, [please check out this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com

Depth First Search (DFS)

 takeuforward.org/data-structure/depth-first-search-dfs

August 10, 2022

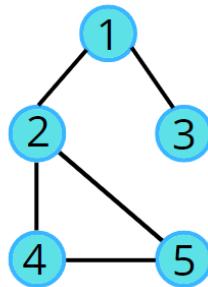
Problem Statement: Given an undirected graph, return a vector of all nodes by traversing the graph using depth-first search (DFS).

Pre-req: Recursion, Graph Representation

Examples:

Example 1:

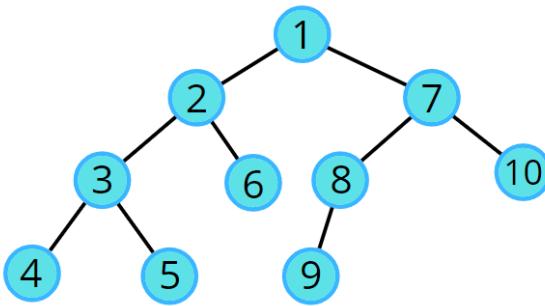
Input:



Output: 1 2 4 5 3

Example 2:

Input:



Output: 1 2 3 4 5 6 7 8 9 10

Solution

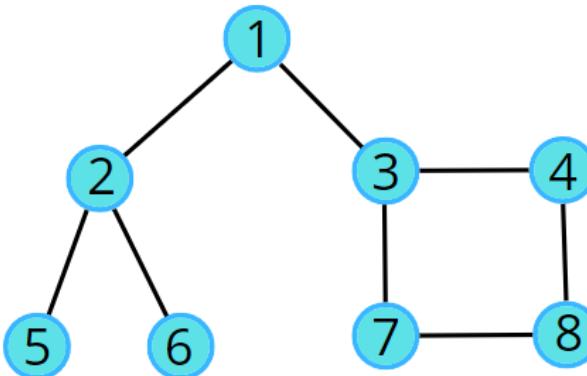
Disclaimer: Don't jump directly to the solution, try it out yourself first.

Approach:

DFS is a traversal technique which involves the idea of recursion and backtracking. DFS goes in-depth, i.e., traverses all nodes by going ahead, and when there are no further nodes to traverse in the current path, then it backtracks on the same path and traverses other unvisited nodes.

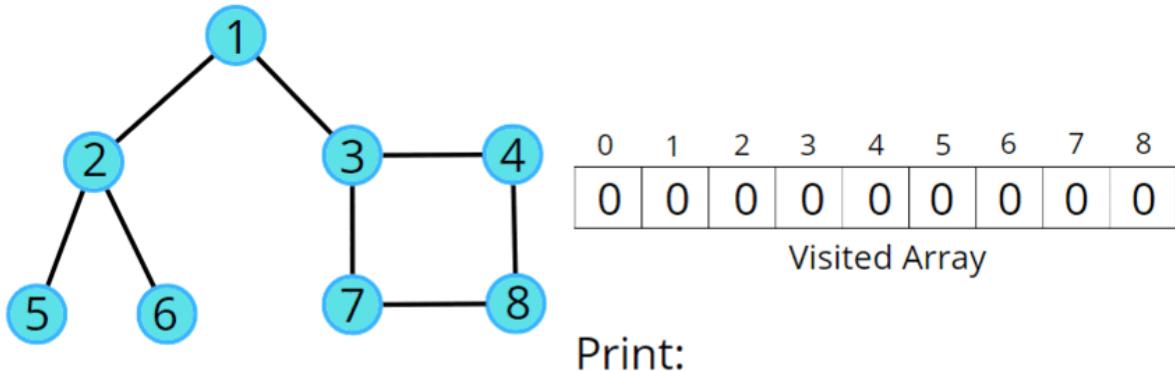
1. In DFS, we start with a node 'v', mark it as visited and store it in the solution vector. It is unexplored as its adjacent nodes are not visited.
2. We run through all the adjacent nodes, and call the recursive dfs function to explore the node 'v' which has not been visited previously. This leads to the exploration of another node 'u' which is its adjacent node and is not visited.
3. The adjacency list stores the list of neighbours for any node. Pick the neighbour list of node 'v' and run a for loop on the list of neighbours (say nodes 'u' and 'w' are in the list). We go in-depth with each node. When node 'u' is explored completely then it backtracks and explores node 'w'.
4. This traversal terminates when all the nodes are completely explored.

In this way, all the nodes are traversed in a depthwise manner.



0	→	{ }
1	→	{ 2, 3 }
2	→	{ 1, 5, 6 }
3	→	{ 1, 4, 7 }
4	→	{ 3, 8 }
5	→	{ 2 }
6	→	{ 2 }
7	→	{ 3, 8 }
8	→	{ 4, 7 }

Adjacency List



```
dfs(node)
{
    visited[node] = 1
    List.add(node)
    for ( auto it: adj[node])
    {
        if(!visited[it])
            dfs(it)
    }
}
```

Note: For a better understanding of the dry run please check the video listed below.

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
    void dfs(int node, vector<int> adj[], int vis[], vector<int> &ls) {
        vis[node] = 1;
        ls.push_back(node);
        // traverse all its neighbours
        for(auto it : adj[node]) {
            // if the neighbour is not visited
            if(!vis[it]) {
                dfs(it, adj, vis, ls);
            }
        }
    }
public:
    // Function to return a list containing the DFS traversal of the graph.
    vector<int> dfsOfGraph(int V, vector<int> adj[]) {
        int vis[V] = {0};
        int start = 0;
        // create a list to store dfs
        vector<int> ls;
        // call dfs for starting node
        dfs(start, adj, vis, ls);
        return ls;
    }
};

void addEdge(vector <int> adj[], int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
}

void printAns(vector <int> &ans) {
    for (int i = 0; i < ans.size(); i++) {
        cout << ans[i] << " ";
    }
}

int main()
{
    vector <int> adj[5];

    addEdge(adj, 0, 2);
    addEdge(adj, 2, 4);
    addEdge(adj, 0, 1);
    addEdge(adj, 0, 3);

    Solution obj;
    vector <int> ans = obj.dfsOfGraph(5, adj);
    printAns(ans);
}

```

```
    return 0;  
}
```

Output: 0 2 4 1 3

Time Complexity: For an undirected graph, $O(N) + O(2E)$, For a directed graph, $O(N) + O(E)$, Because for every node we are calling the recursive function once, the time taken is $O(N)$ and $2E$ is for total degrees as we traverse for all adjacent nodes.

Space Complexity: $O(3N) \sim O(N)$, Space for dfs stack space, visited array and an adjacency list.

Special thanks to [Vanshika Singh Gour](#) for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, [please check out this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com

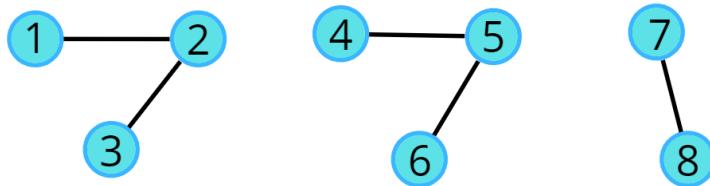
Number of Provinces

Problem Statement: Given an undirected graph with V vertices. We say two vertices u and v belong to a single province if there is a path from u to v or v to u . Your task is to find the number of provinces.

Pre-req: Connected Components, Graph traversal techniques

Examples:

Input :



Output : 3

Solution

***Disclaimer:** Don't jump directly to the solution, try it out yourself first.*

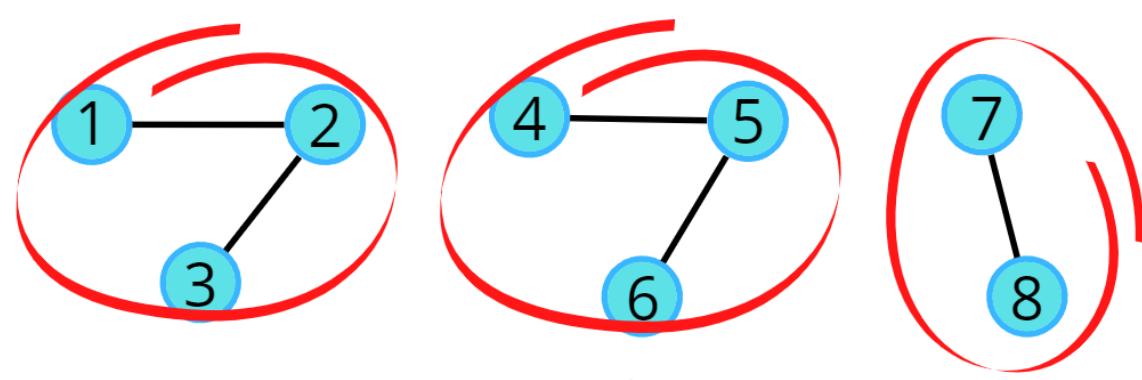
Approach:

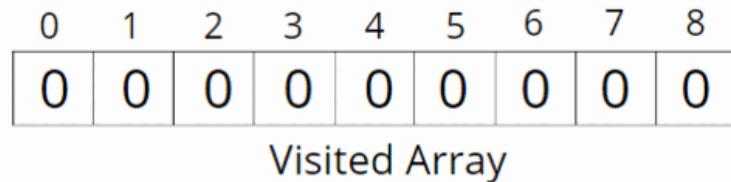
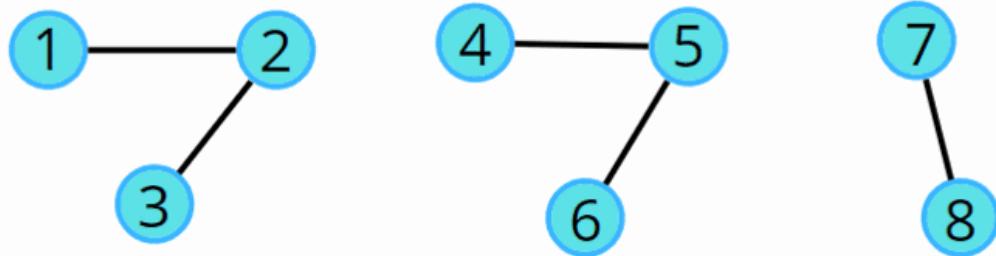
A province is a group of directly or indirectly connected cities and no other cities outside of the group. Considering the above example, we can go from 1 to 2 as well as to 3, from every other node in a province we can go to each other. As we cannot go from 2 to 4 so it is not a province. We know about both the traversals, Breadth First Search (BFS) and Depth First Search (DFS). We can use any of the traversals to solve this problem because a traversal algorithm visits all the nodes in a graph. In any traversal technique, we have one starting node and it traverses all the nodes in the graph. Suppose there is an ' N ' number of provinces so we need to call the traversal algorithm ' N ' times, i.e., there will be ' N ' starting nodes. So, we just need to figure out the number of starting nodes.

The algorithm steps are as follows:

- We need a visited array initialized to 0, representing the nodes that are not visited.
- Run the for loop looping from 0 to N , and call the DFS for the first unvisited node.

- DFS function call will make sure that it starts the DFS call from that unvisited node, and visits all the nodes that are in that province, and at the same time, it will also mark them as visited.
- Since the nodes traveled in a traversal will be marked as visited, they will no further be called for any further DFS traversal.
- Keep repeating these steps, for every node that you find unvisited, and visit the entire province.
- Add a counter variable to count the number of times the DFS function is called, as in this way we can count the total number of starting nodes, which will give us the number of provinces.





```

for ( i = 1; i <= V; i++ )
{
    if(visited[i] == 0)
    {
        dfs(i); // bfs(i)
    }
}
  
```

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
    // dfs traversal function
    void dfs(int node, vector<int> adjLs[], int vis[]) {
        // mark the node as visited
        vis[node] = 1;
        for(auto it: adjLs[node]) {
            if(!vis[it]) {
                dfs(it, adjLs, vis);
            }
        }
    }
public:
    int numProvinces(vector<vector<int>> adj, int V) {
        vector<int> adjLs[V];

        // to change adjacency matrix to list
        for(int i = 0;i<V;i++) {
            for(int j = 0;j<V;j++) {
                // self nodes are not considered
                if(adj[i][j] == 1 && i != j) {
                    adjLs[i].push_back(j);
                    adjLs[j].push_back(i);
                }
            }
        }
        int vis[V] = {0};
        int cnt = 0;
        for(int i = 0;i<V;i++) {
            // if the node is not visited
            if(!vis[i]) {
                // counter to count the number of provinces
                cnt++;
                dfs(i, adjLs, vis);
            }
        }
        return cnt;
    }
};

int main() {

    vector<vector<int>> adj
    {
        {1, 0, 1},
        {0, 1, 0},
        {1, 0, 1}
    };
}

```

```
Solution ob;
cout << ob.numProvinces(adj, 3) << endl;

return 0;
}
```

Output: 2

Time Complexity: $O(N) + O(V+2E)$, Where $O(N)$ is for outer loop and inner loop runs in total a single DFS over entire graph, and we know DFS takes a time of $O(V+2E)$.

Space Complexity: $O(N) + O(N)$, Space for recursion stack space and visited array.

Special thanks to Vanshika Singh Gour for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article. If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com



Watch Video At: <https://youtu.be/ACzkVtewUYA>

Rotten Oranges : Min time to rot all oranges : BFS

 takeuforward.org/data-structure/rotten-oranges-min-time-to-rot-all-oranges-bfs

December 2, 2021

Problem Statement: You will be given an $m \times n$ grid, where each cell has the following values :

1. 2 – represents a rotten orange
2. 1 – represents a Fresh orange
3. 0 – represents an Empty Cell

Every minute, if a Fresh Orange is adjacent to a Rotten Orange in 4-direction (upward, downwards, right, and left) it becomes Rotten.

Return the minimum number of minutes required such that none of the cells has a Fresh Orange. If it's not possible, return -1.

Examples:

Example 1:

Input: grid - [[2,1,1] , [0,1,1] , [1,0,1]]

Output: -1

Explanation:

Minute - 0

2	1	1
0	1	1
1	0	1

Minute - 1

2	2	1
0	1	1
1	0	1

Minute - 2

2	2	2
0	2	1
1	0	1

Minute - 3

2	2	2
0	2	2
1	0	1

Minute - 4

2	2	2
0	2	2
1	0	2

Example 2:

Input: grid - [[2,1,1] , [1,1,0] , [0,1,1]]

Output: 4

Explanation:

Minute - 0		
2	1	1
1	1	0
0	1	1

Minute - 1		
2	2	1
2	1	0
0	1	1

Minute - 2		
2	2	2
2	2	0
0	1	1

Minute - 3		
2	2	2
2	2	0
0	2	1

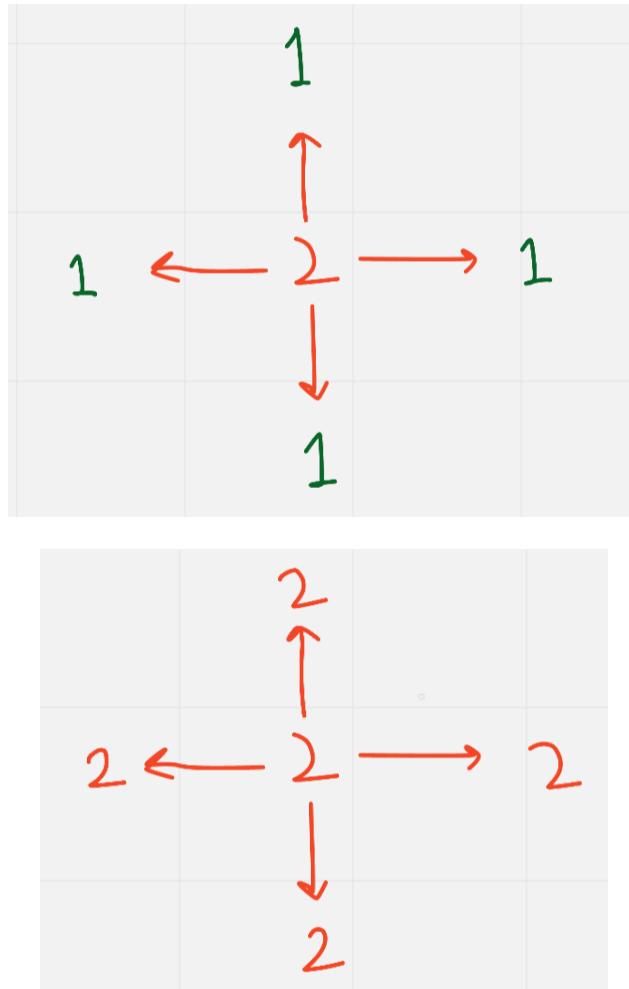
Minute - 4		
2	2	2
2	2	0
0	2	2

Solution

***Disclaimer:** Don't jump directly to the solution, try it out yourself first.*

Intuition:

The idea is that for each rotten orange, we will find how many fresh oranges there are in its 4 directions. If we find any fresh orange we will make it into a rotten orange. One rotten orange can rotten up to 4 fresh oranges present in its 4 directions. For this problem, we will be using the BFS (Breadth-First Search) technique.



Approach:

-> First of all we will create a Queue data structure to store coordinate of Rotten Oranges

We will also have variables as:

1. **Total_oranges** – It will store total number of oranges in the grid (Rotten + Fresh)
2. **Count** – It will store the total number of oranges rotten by us .
3. **Total_time** – total time taken to rotten.

-> After this, we will traverse the whole grid and count the total number of oranges in the grid and store it in Total_oranges. Then we will also push the rotten oranges in the Queue data structure as well.

-> Now while our queue is not empty, we will pick up each Rotten Orange and check in all its 4 directions whether a Fresh orange is present or not. If it is present we will make it rotten and push it in our queue data structure and pop out the Rotten Orange which we took up as its work is done now.

-> Also we will keep track of the count of rotten oranges we are getting.

-> If we rotten some oranges, then obviously our queue will not be empty. In that case, we will increase our total time. This goes on until our queue becomes empty.

-> After it becomes empty, We will check whether the total number of oranges initially is equal to the current count of oranges. If yes, we will return the **total time taken**, else will return -1 because some fresh oranges are still left and can't be made rotten.

Code:

- C++ Code
- Java Code

```

#include<bits/stdc++.h>
using namespace std;
int orangesRotting(vector<vector<int>>& grid) {
    if(grid.empty()) return 0;
    int m = grid.size(), n = grid[0].size(), days = 0, tot = 0, cnt = 0;
    queue<pair<int, int>> rotten;
    for(int i = 0; i < m; ++i){
        for(int j = 0; j < n; ++j){
            if(grid[i][j] != 0) tot++;
            if(grid[i][j] == 2) rotten.push({i, j});
        }
    }

    int dx[4] = {0, 0, 1, -1};
    int dy[4] = {1, -1, 0, 0};

    while(!rotten.empty()){
        int k = rotten.size();
        cnt += k;
        while(k--){
            int x = rotten.front().first, y = rotten.front().second;
            rotten.pop();
            for(int i = 0; i < 4; ++i){
                int nx = x + dx[i], ny = y + dy[i];
                if(nx < 0 || ny < 0 || nx >= m || ny >= n || grid[nx][ny] != 1)
                    continue;
                grid[nx][ny] = 2;
                rotten.push({nx, ny});
            }
        }
        if(!rotten.empty()) days++;
    }

    return tot == cnt ? days : -1;
}

int main()
{
    vector<vector<int>> v{ {2,1,1} , {1,1,0} , {0,1,1} } ;
    int rotting = orangesRotting(v);
    cout<<"Minimum Number of Minutes Required "<<rotting<<endl;
}

```

Output:

Minimum Number of Minutes Required 4

Time Complexity: O (n x n) x 4

Reason: Worst-case – We will be making each fresh orange rotten in the grid and for each rotten orange will check in 4 directions

Space Complexity: O (n x n)

Reason: worst-case – If all oranges are Rotten, we will end up pushing all rotten oranges into the Queue data structure

Special thanks to [Shreyas Vishwakarma](#) for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article

Flood Fill Algorithm – Graphs

 takeuforward.org/graph/flood-fill-algorithm-graphs

August 12, 2022

Problem Statement: An image is represented by a 2-D array of integers, each integer representing the pixel value of the image. Given a coordinate (sr, sc) representing the starting pixel (row and column) of the flood fill, and a pixel value newColor, “flood fill” the image.

To perform a “flood fill”, consider the starting pixel, plus any pixels connected 4-directionally to the starting pixel of the same colour as the starting pixel, plus any pixels connected 4-directionally to those pixels (also with the same colour as the starting pixel), and so on. Replace the colour of all of the aforementioned pixels with the newColor.

Pre-req: Connected Components, Graph traversal techniques

Example 1:

Input :

1	1	1
1	1	0
1	0	1

sr = 1, sc = 1, newColor = 2

Output :

2	2	2
2	2	0
2	0	1

Explanation:

0	1	2
0	1	1
1	1	1
2	1	0

initial color = 1
new Color = 2



From the centre of the image
(with position $(sr, sc) = (1, 1)$), all pixels connected by a path of the same colour as the starting pixel are colored with the new colour.

Note the bottom corner is not colored 2,
because it is not 4-directionally connected to
the starting pixel.

Example 2:

Input:

1	1	1
2	2	0
2	2	2

$sr = 2, sc = 0, newColor = 3$

Output:

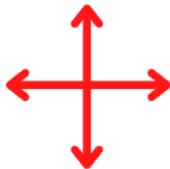
1	1	1
3	3	0
3	3	3

Solution

***Disclaimer:** Don't jump directly to the solution, try it out yourself first.*

Approach:

To perform a “flood fill”, consider the starting pixel, plus any pixels connected 4-directionally to the starting pixel of the same colour as the starting pixel, plus any pixels connected 4-directionally to those pixels (also with the same colour as the starting pixel), and so on.



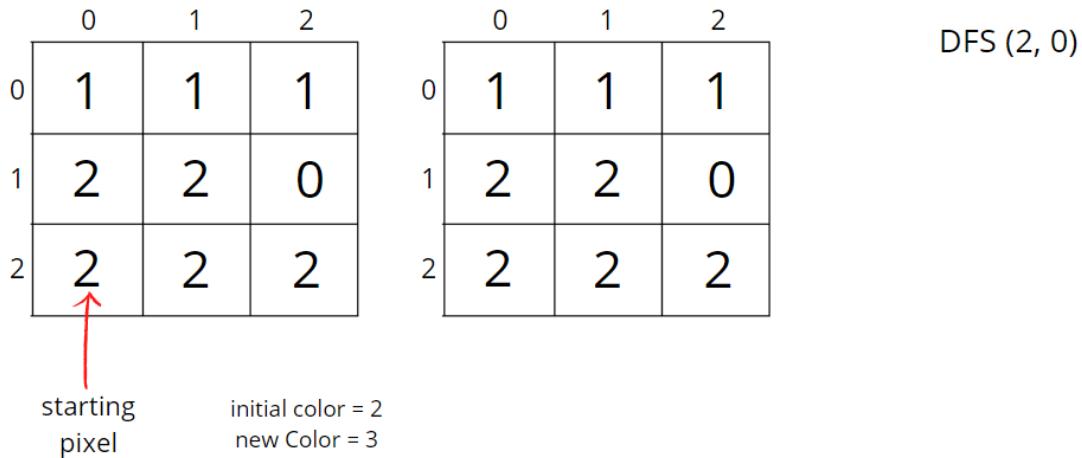
We know about both the traversals, Breadth First Search (BFS) and Depth First Search (DFS). We can follow BFS also, where we start at a given point and colour level wise, i.e., we go simultaneously to all its neighbours. We can use any of the traversals to solve this problem, in our case we will be using DFS just to explore multiple approaches.

The algorithm steps are as follows:

- Initial DFS call will start with the starting pixel (sr , sc) and make sure to store the initial colour.
- Now, either we can use the same matrix to replace the colour of all of the aforementioned pixels with the newColor or create a replica of the given matrix. It is advised to use another matrix because we work on the data and not tamper with it. So we will create a copy of the input matrix.
- Check for the neighbours of the respective pixel that has the same initial colour and has not been visited or coloured. DFS call goes first in the depth on either of the neighbours.
- We go to all 4 directions and check for **unvisited** neighbours with the same initial colour. To travel 4 directions we will use nested loops, you can find the implementation details in the code.
- DFS function call will make sure that it starts the DFS call from that unvisited neighbour, and colours all the pixels that have the same initial colour, and at the same time it will also mark them as visited.

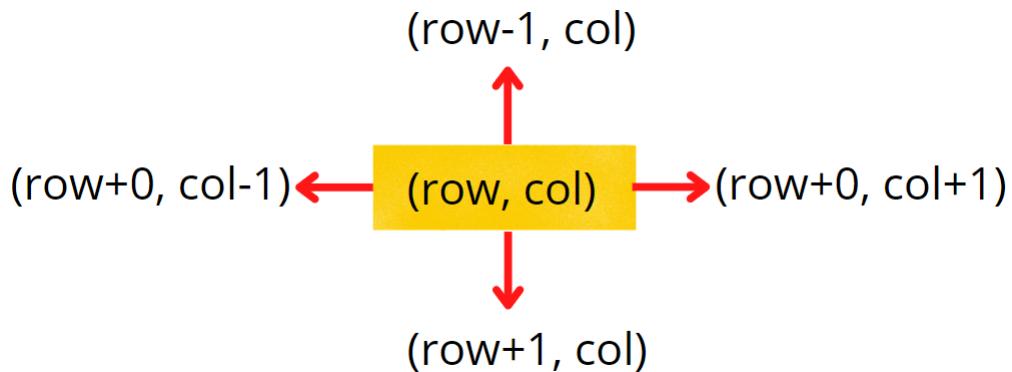
In this way, “flood fill” will be performed. It doesn’t matter how we are colouring the pixels, we just want to colour all of the aforementioned pixels with the newColor. So, we can use any of the traversal techniques.

Consider the following example to understand how DFS traverses the pixels and colours them accordingly.



How to set boundaries for 4 directions?

The 4 neighbours will have following indexes:



Now, either we can apply 4 conditions or follow the following method.

From the above image, it is clear that delta change in row is -1, +0, +1, +0. Similarly, the delta change in column is 0, +1, +0, -1. So we can apply the same logic to find the neighbours of a particular pixel ($\langle row, column \rangle$).

Code:

- C++ Code
- Java Code

```

#include<bits/stdc++.h>
using namespace std;

class Solution {
private:
    void dfs(int row, int col, vector<vector<int>>&ans,
              vector<vector<int>>& image, int newColor, int delRow[], int delCol[],
              int iniColor) {
        // color with new color
        ans[row][col] = newColor;
        int n = image.size();
        int m = image[0].size();
        // there are exactly 4 neighbours
        for(int i = 0;i<4;i++) {
            int nrow = row + delRow[i];
            int ncol = col + delCol[i];
            // check for valid coordinate
            // then check for same initial color and unvisited pixel
            if(nrow>=0 && nrow<n && ncol>=0 && ncol < m &&
               image[nrow][ncol] == iniColor && ans[nrow][ncol] != newColor) {
                dfs(nrow, ncol, ans, image, newColor, delRow, delCol, iniColor);
            }
        }
    }
public:
    vector<vector<int>> floodFill(vector<vector<int>>& image,
                                     int sr, int sc, int newColor) {
        // get initial color
        int iniColor = image[sr][sc];
        vector<vector<int>> ans = image;
        // delta row and delta column for neighbours
        int delRow[] = {-1, 0, +1, 0};
        int delCol[] = {0, +1, 0, -1};
        dfs(sr, sc, ans, image, newColor, delRow, delCol, iniColor);
        return ans;
    }
};

int main(){
    vector<vector<int>>image{
        {1,1,1},
        {1,1,0},
        {1,0,1}
    };

    // sr = 1, sc = 1, newColor = 2
    Solution obj;
    vector<vector<int>> ans = obj.floodFill(image, 1, 1, 2);
    for(auto i: ans){
        for(auto j: i)
            cout << j << " ";
    }
}

```

```
        cout << "\n";
    }

    return 0;
}
```

Output:

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

Time Complexity: $O(N \times M + N \times M \times 4) \sim O(N \times M)$

For the worst case, all of the pixels will have the same colour, so DFS function will be called for $(N \times M)$ nodes and for every node we are traversing for 4 neighbours, so it will take $O(N \times M \times 4)$ time.

Space Complexity: $O(N \times M) + O(N \times M)$

$O(N \times M)$ for copied input array and recursive stack space takes up $N \times M$ locations at max.

Special thanks to Vanshika Singh Gour for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article. If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com

Detect Cycle in an Undirected Graph (using BFS)

 takeuforward.org/data-structure/detect-cycle-in-an-undirected-graph-using-bfs

August 27, 2022

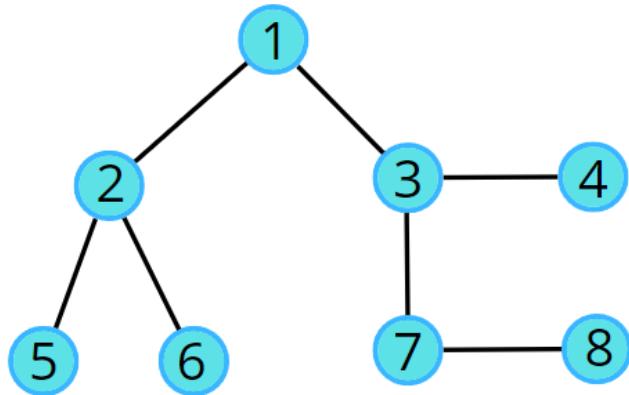
Problem Statement: Given an undirected graph with V vertices and E edges, check whether it contains any cycle or not.

Examples:

Example 1:

Input:

$V = 8, E = 7$



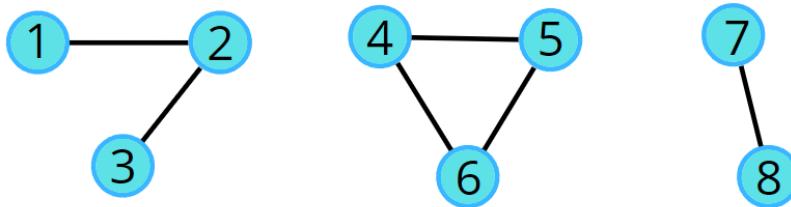
Output: 0

Explanation: No cycle in the given graph.

Example 2:

Input:

$V = 8, E = 6$



Output: 1

Explanation: $4 \rightarrow 5 \rightarrow 6 \rightarrow 4$ is a cycle.

Solution

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Intuition:

The cycle in a graph starts from a node and ends at the same node. So we can think of two algorithms to do this, in this article we will be reading about the BFS, and in the next, we will be learning how to use DFS to check.

Breadth First Search, BFS is a traversal technique where we visit the nodes level-wise, i.e., it visits the same level nodes simultaneously, and then moves to the next level.

The intuition is that we start from a node, and start doing BFS level-wise, if somewhere down the line, we visit a single node twice, it means we came via two paths to end up at the same node. It implies there is a cycle in the graph because we know that we start from different directions but can arrive at the same node only if the graph is connected or contains a cycle, otherwise we would never come to the same node again.

Approach:

Initial configuration:

- **Queue:** Define a queue and insert the source node along with parent data (<source node, parent>). For example, (2, 1) means 2 is the source node and 1 is its parent node.
- **Visited array:** an array initialized to 0 indicating unvisited nodes.

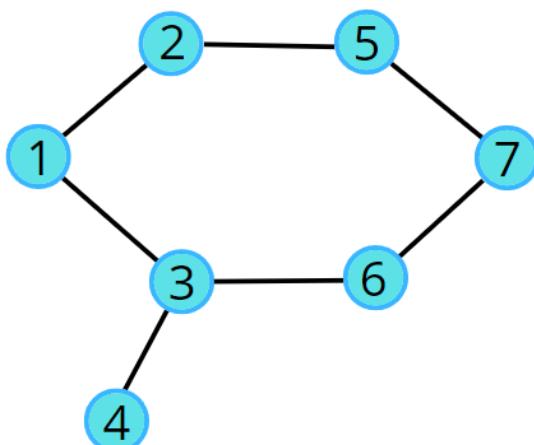
The algorithm steps are as follows:

- For BFS traversal, we need a queue data structure and a visited array.
- Push the pair of the source node and its parent data (<source, parent>) in the queue, and mark the node as visited. The parent will be needed so that we don't do a backward traversal in the graph, we just move frontwards.
- Start the BFS traversal, pop out an element from the queue every time and travel to all its unvisited neighbors using an adjacency list.
- Repeat the steps either until the queue becomes empty, or a node appears to be already visited which is not the parent, even though we traveled in different directions during the traversal, indicating there is a cycle.
- If the queue becomes empty and no such node is found then there is no cycle in the graph.

A graph can have connected components as well. In such cases, if any component forms a cycle then the graph is said to have a cycle. We can follow the algorithm for the same:

```
// check for connected components in a graph
for ( i = 1; i <= n; i++)
{
    if(!vis[i])
    {
        if(detectCycle(i) == true)
            return true;
    }
}
return false;
```

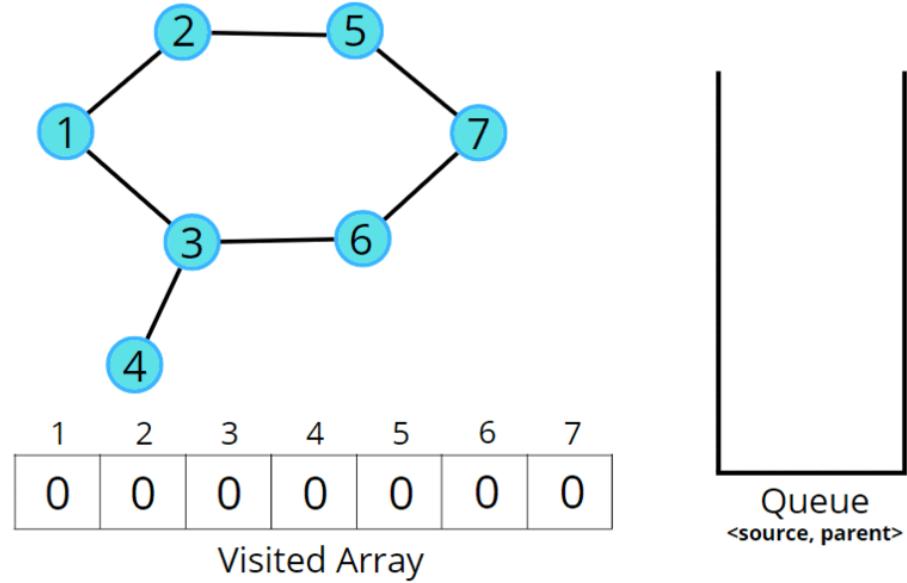
Consider the following graph and its adjacency list.



1	→ { 2, 3 }
2	→ { 1, 5 }
3	→ { 1, 4, 6 }
4	→ { 3 }
5	→ { 2, 7 }
6	→ { 3, 7 }
7	→ { 5, 6 }

Adjacency List

Consider the following illustration to understand the process of detecting a cycle using BFS traversal.



Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
    bool detect(int src, vector<int> adj[], int vis[]) {
        vis[src] = 1;
        // store <source node, parent node>
        queue<pair<int, int>> q;
        q.push({src, -1});
        // traverse until queue is not empty
        while(!q.empty()) {
            int node = q.front().first;
            int parent = q.front().second;
            q.pop();

            // go to all adjacent nodes
            for(auto adjacentNode: adj[node]) {
                // if adjacent node is unvisited
                if(!vis[adjacentNode]) {
                    vis[adjacentNode] = 1;
                    q.push({adjacentNode, node});
                }
                // if adjacent node is visited and is not it's own parent node
                else if(parent != adjacentNode) {
                    // yes it is a cycle
                    return true;
                }
            }
        }
        // there's no cycle
        return false;
    }
public:
    // Function to detect cycle in an undirected graph.
    bool isCycle(int V, vector<int> adj[]) {
        // initialise them as unvisited
        int vis[V] = {0};
        for(int i = 0;i<V;i++) {
            if(!vis[i]) {
                if(detect(i, adj, vis)) return true;
            }
        }
        return false;
    }
};

int main() {

    // V = 4, E = 2
    vector<int> adj[4] = {{}, {2}, {1, 3}, {2}};
    Solution obj;

```

```
bool ans = obj.isCycle(4, adj);
if (ans)
    cout << "1\n";
else
    cout << "0\n";
return 0;
}
```

Output: 0

Time Complexity: $O(N + 2E) + O(N)$, Where N = Nodes, 2E is for total degrees as we traverse all adjacent nodes. In the case of connected components of a graph, it will take another $O(N)$ time.

Space Complexity: $O(N) + O(N) \sim O(N)$, Space for queue data structure and visited array.

Special thanks to Vanshika Singh Gour for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article. If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com

Detect Cycle in an Undirected Graph (using DFS)

 takeuforward.org/data-structure/detect-cycle-in-an-undirected-graph-using-dfs

August 29, 2022

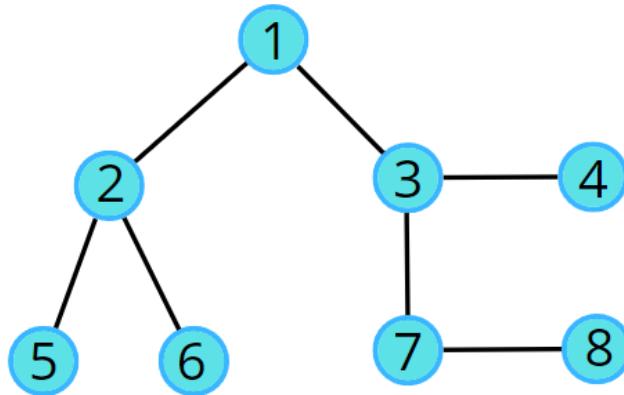
Problem Statement: Given an undirected graph with V vertices and E edges, check whether it contains any cycle or not.

Examples:

Example 1:

Input:

$V = 8, E = 7$



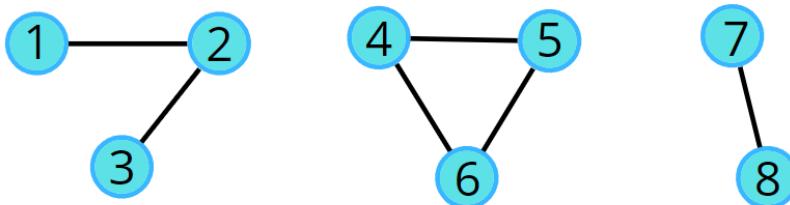
Output: No

Explanation: No cycle in the given graph.

Example 2:

Input:

$V = 8, E = 6$



Output: Yes

Explanation:

$4 \rightarrow 5 \rightarrow 6 \rightarrow 4$ is a cycle.

Solution

Disclaimer: Don't jump directly to the solution, try it out yourself first.

Intuition:

The cycle in a graph starts from a node and ends at the same node. DFS is a traversal technique that involves the idea of recursion and backtracking. DFS goes in-depth, i.e., traverses all nodes by going ahead, and when there are no further nodes to traverse in the current path, then it backtracks on the same path and traverses other unvisited nodes. The intuition is that we start from a source and go in-depth, and reach any node that has been previously visited in the past; it means there's a cycle.

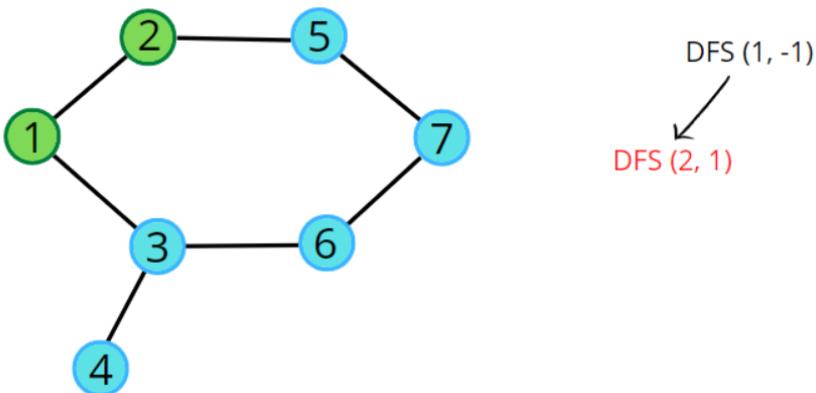
Approach:

The algorithm steps are as follows:

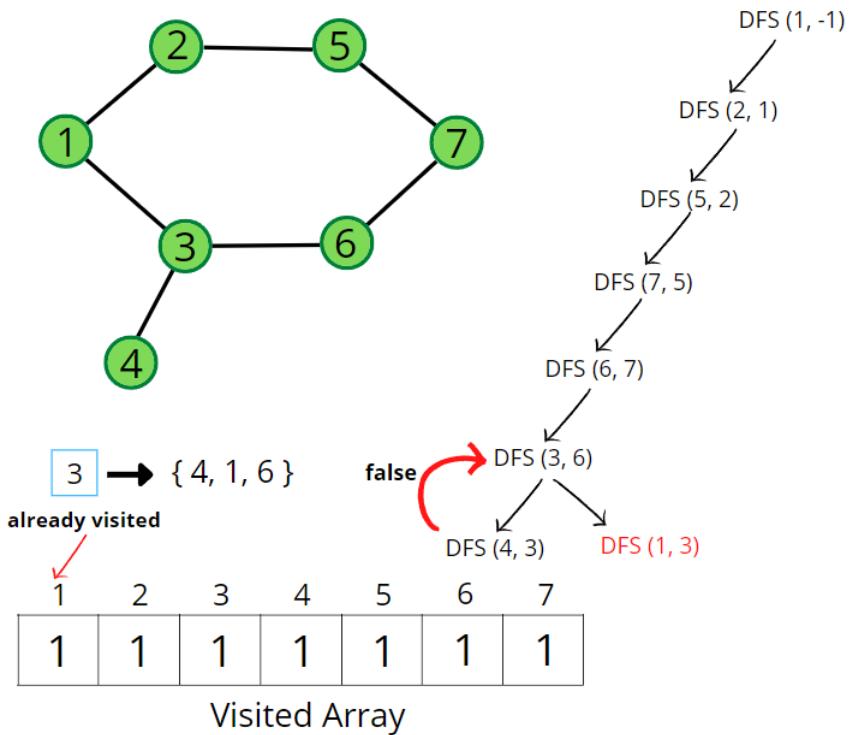
- In the DFS function call make sure to store the parent data along with the source node, create a visited array, and initialize to 0. The parent is stored so that while checking for re-visited nodes, we don't check for parents.
- We run through all the unvisited adjacent nodes using an adjacency list and call the recursive dfs function. Also, mark the node as visited.
- If we come across a node that is already marked as visited and is **not a parent node**, then keep on returning true indicating the presence of a cycle; otherwise return false after all the adjacent nodes have been checked and we did not find a cycle.

NOTE: We can call it a cycle only if the already visited node is a non-parent node because we cannot say we came to a node that was previously the parent node.

For example, node 2 has two adjacent nodes 1 and 5. 1 is already visited but it is the parent node ($\text{DFS}(2, 1)$), So this cannot be called a cycle.



Node 3 has three adjacent nodes, where 4 and 6 are already visited but node 1 is not visited by node 3, but it's already marked as visited and is a non-parent node ($\text{DFS}(3, 6)$), indicating the presence of cycle.



Pseudocode:

```

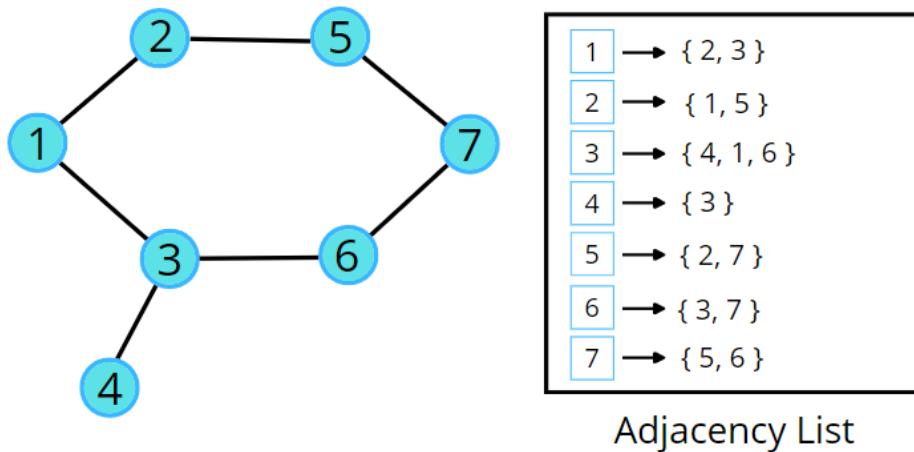
dfs( node, parent )
{
    vis[node] = 1;
    // visit the neighbours
    for( auto it: adj [node])
    {
        if(vis[i] == 0)
        {
            if(dfs(it, node) == true)
                return true;
        }
        // already visited but is not parent node
        else if(it != parent)
            return true;
    }
    // not a cycle
    return false;
}

```

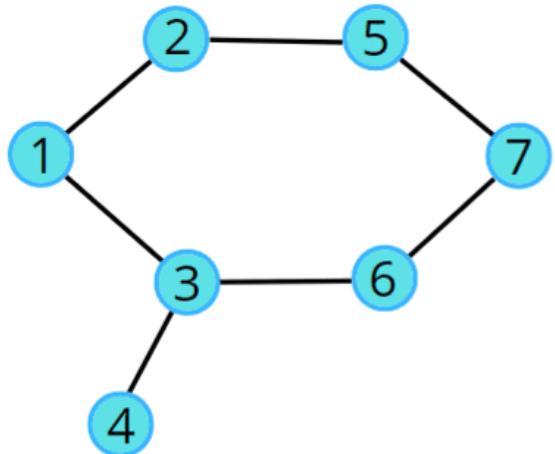
A graph can have connected components as well. In such cases, if any component forms a cycle then the graph is said to have a cycle. We can follow the algorithm for the same:

```
// check for connected components in a graph
for ( i = 1; i<= n; i++ )
{
    if( !vis[i] )
    {
        if( dfs(i) == true)
            return true;
    }
}
return false;
```

Consider the following graph and its adjacency list.



Consider the following illustration to understand the process of detecting a cycle using DFS traversal.



1	2	3	4	5	6	7
0	0	0	0	0	0	0

Visited Array

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
    bool dfs(int node, int parent, int vis[], vector<int> adj[]) {
        vis[node] = 1;
        // visit adjacent nodes
        for(auto adjacentNode: adj[node]) {
            // unvisited adjacent node
            if(!vis[adjacentNode]) {
                if(dfs(adjacentNode, node, vis, adj) == true)
                    return true;
            }
            // visited node but not a parent node
            else if(adjacentNode != parent) return true;
        }
        return false;
    }
public:
    // Function to detect cycle in an undirected graph.
    bool isCycle(int V, vector<int> adj[]) {
        int vis[V] = {0};
        // for graph with connected components
        for(int i = 0;i<V;i++) {
            if(!vis[i]) {
                if(dfs(i, -1, vis, adj) == true) return true;
            }
        }
        return false;
    }
};

};


```

```

int main() {

    // V = 4, E = 2
    vector<int> adj[4] = {{}, {2}, {1, 3}, {2}};
    Solution obj;
    bool ans = obj.isCycle(4, adj);
    if (ans)
        cout << "1\n";
    else
        cout << "0\n";
    return 0;
}

```

Output: 0

Time Complexity: $O(N + 2E) + O(N)$, Where N = Nodes, 2E is for total degrees as we traverse all adjacent nodes. In the case of connected components of a graph, it will take another $O(N)$ time.

Space Complexity: $O(N) + O(N) \sim O(N)$, Space for recursive stack space and visited array.

Special thanks to Vanshika Singh Gour for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article. If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com



Watch Video At: <https://youtu.be/zQ3zgFypzX4>

Distance of Nearest Cell having 1

 takeuforward.org/graph/distance-of-nearest-cell-having-1

September 3, 2022

Problem Statement: Given a binary grid of $N \times M$. Find the distance of the nearest 1 in the grid for each cell.

The distance is calculated as $|i_1 - i_2| + |j_1 - j_2|$, where i_1, j_1 are the row number and column number of the current cell, and i_2, j_2 are the row number and column number of the nearest cell having value 1.

Examples:

Example 1:

Input:

1	0	1
1	1	0
1	0	0

Output:

0	1	0
0	0	1
0	1	2

Explanation:

0's at $(0,1)$, $(1,2)$, $(2,1)$ and $(2,2)$ are at a distance of 1, 1, 1 and 2 from 1's at $(0,0)$, $(0,2)$, $(2,0)$ and $(1,1)$ respectively.

Example 2:

Input:

0	0	0
0	1	0
1	0	1

Output:

2	1	2
1	0	1
0	1	0

Solution

***Disclaimer:** Don't jump directly to the solution, try it out yourself first.*

Intuition:

Breadth First Search, BFS, is a traversal technique where we visit the nodes level-wise, i.e., it visits the same level nodes simultaneously, and then moves to the next level.

The intuition is that BFS will take a step from cells containing 1 and will reach out to all zeros that are at a distance of one. Apparently, we can say that the nearest 1 to the 0s is at a distance of one. Again if we take another step, we will reach the next set of zeros, for these zeros 1 is at a distance of two. If we continue the same, till we can go, we can reach all the 0's possible.

We will choose the BFS algorithm as it moves step by step and we want all of them to traverse in a single step together so that we can have a minimum count with us.

Approach:

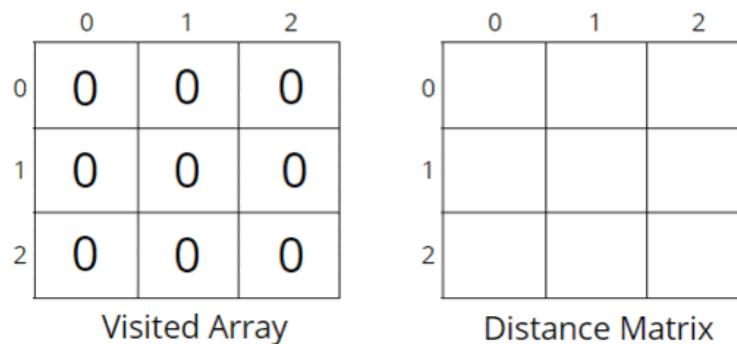
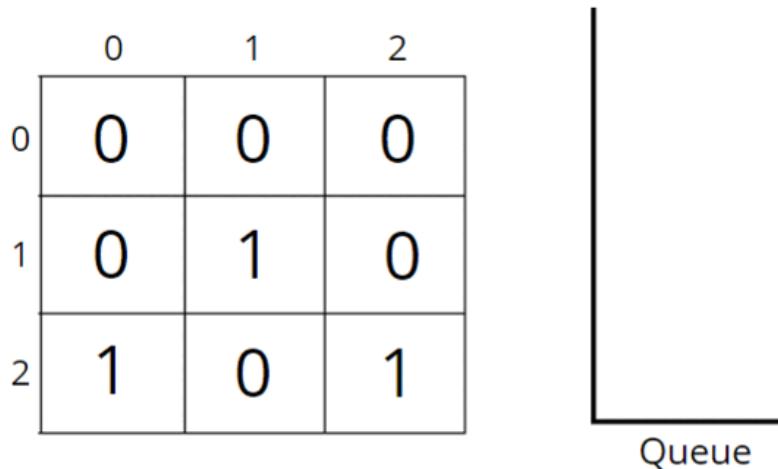
Initial configuration:

- **Queue:** Define a queue and insert the pair of starting nodes' coordinates along with the steps (<coordinates, step>). For example, ((2, 1), 2) means cell (2, 1) is the source node and the nearest 1 can be found at a distance of 2 from the node.
- **Visited array:** an array initialized to 0 indicating unvisited nodes.
- **Distance matrix:** stores the distance of the nearest cell having 1 for every particular cell.

The algorithm steps are as follows:

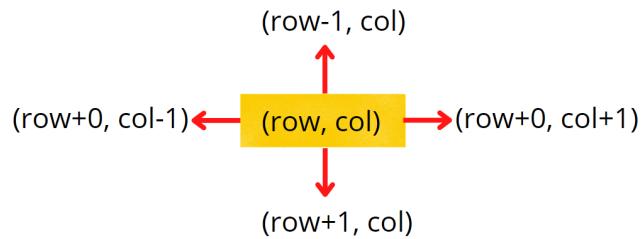
- Push the pair of starting points and its steps (<coordinates, step>) in the queue, and mark the cell as visited.
- Start the BFS traversal, pop out an element from the queue every time, and travel to all its unvisited neighbors having 0.
- For every neighboring unvisited 0, we can mark the distance to be +1 of the current node distance and store it in the distance 2D array, and at the same time insert <{row, col}, steps+1> into the queue.
- Repeat the steps until the queue becomes empty and then return the distance matrix where we have stored the steps.

Consider the following illustration to understand how BFS traverses the cells and calculates the distance of the nearest 1 in the grid.



How do set boundaries for 4 directions?

The 4 neighbors will have the following indexes:



Now, either we can apply 4 conditions or follow the following method.

From the above image, it is clear that the delta change in the row is $-1, +0, +1, +0$. Similarly, the delta change in the column is $0, +1, +0, -1$. So we can apply the same logic to find the neighbors of a particular pixel ($\langle row, column \rangle$).

Code:

- C++ Code
- Java Code

```

#include<bits/stdc++.h>
using namespace std;

class Solution
{
public:
    //Function to find the distance of nearest 1 in the grid for each cell.
    vector<vector<int>>nearest(vector<vector<int>>grid)
    {
        int n = grid.size();
        int m = grid[0].size();
        // visited and distance matrix
        vector<vector<int>> vis(n, vector<int>(m, 0));
        vector<vector<int>> dist(n, vector<int>(m, 0));
        // <coordinates, steps>
        queue<pair<pair<int,int>, int>> q;
        // traverse the matrix
        for(int i = 0;i<n;i++) {
            for(int j = 0;j<m;j++) {
                // start BFS if cell contains 1
                if(grid[i][j] == 1) {
                    q.push({{i,j}, 0});
                    vis[i][j] = 1;
                }
                else {
                    // mark unvisited
                    vis[i][j] = 0;
                }
            }
        }

        int delrow[] = {-1, 0, +1, 0};
        int delcol[] = {0, +1, 0, -1};

        // traverse till queue becomes empty
        while(!q.empty()) {
            int row = q.front().first.first;
            int col = q.front().first.second;
            int steps = q.front().second;
            q.pop();
            dist[row][col] = steps;
            // for all 4 neighbours
            for(int i = 0;i<4;i++) {
                int nrow = row + delrow[i];
                int ncol = col + delcol[i];
                // check for valid unvisited cell
                if(nrow >= 0 && nrow < n && ncol >= 0 && ncol < m
                && vis[nrow][ncol] == 0) {
                    vis[nrow][ncol] = 1;
                    q.push({{nrow, ncol}, steps+1});
                }
            }
        }
    }
}

```

```

        }
        // return distance matrix
        return dist;
    }

};

int main(){
    vector<vector<int>>grid{
        {0,1,1,0},
        {1,1,0,0},
        {0,0,1,1}
    };

    Solution obj;
    vector<vector<int>> ans = obj.nearest(grid);

    for(auto i: ans){
        for(auto j: i)
            cout << j << " ";
        cout << "\n";
    }

    return 0;
}

```

Output:

```

1 0 0 1
0 0 1 1
1 1 0 0

```

Time Complexity: $O(N \times M + N \times M \times 4) \sim O(N \times M)$

For the worst case, the BFS function will be called for $(N \times M)$ nodes, and for every node, we are traversing for 4 neighbors, so it will take $O(N \times M \times 4)$ time.

Space Complexity: $O(N \times M) + O(N \times M) + O(N \times M) \sim O(N \times M)$

$O(N \times M)$ for the visited array, distance matrix, and queue space takes up $N \times M$ locations at max.

Special thanks to [Vanshika Singh Gour](#) for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, [please check out this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com



Watch Video At: <https://youtu.be/edXdVwkYHF8>

Surrounded Regions | Replace O's with X's

 takeuforward.org/graph/surrounded-regions-replace-os-with-xs/

September 6, 2022

Problem Statement: Given a matrix mat of size $N \times M$ where every element is either 'O' or 'X'. Replace all 'O' with 'X' that is surrounded by 'X'. An 'O' (or a set of 'O') is considered to be surrounded by 'X' if there are 'X' at locations just below, just above, just left, and just right of it.

Examples:

Example 1:

Input: $n = 5, m = 4$

X	X	X	X
X	O	X	X
X	O	O	X
X	O	X	X
X	X	O	O

Output:

X	X	X	X
X	X	X	X
X	X	X	X
X	X	X	X
X	X	O	O

Example 2:

Input: n = 5, m = 4

X	X	X	X
X	X	X	X
X	O	O	X
X	O	O	X
X	X	X	X

Output:

X	X	X	X
X	X	X	X
X	X	X	X
X	X	X	X
X	X	X	X

Solution

***Disclaimer:** Don't jump directly to the solution, try it out yourself first.*

Intuition:

The boundary elements in the matrix cannot be replaced with 'X' as they are not surrounded by 'X' from all 4 directions. This means if 'O' (or a set of 'O') is connected to a boundary 'O' then it can't be replaced with 'X'.

The intuition is that we start from boundary elements having 'O' and go through its neighboring Os in 4 directions and mark them as visited to avoid replacing them with 'X'.

Approach:

We can follow either of the traversal techniques as long as we are starting with a boundary element and marking all those Os connected to it. We will be solving it using DFS traversal, but you can apply BFS traversal as well.

DFS is a traversal technique that involves the idea of recursion.. DFS goes in-depth, i.e., traverses all nodes by going ahead, and when there are no further nodes to traverse in the current path, then it backtracks on the same path and traverses other unvisited nodes.

The algorithm steps are as follows:

- Create a corresponding visited matrix and initialize it to 0.
- Start with boundary elements, once 'O' is found, call the DFS function for that element and mark it as visited. In order to traverse for boundary elements, you can traverse through the first row, last row, first column, and last column.

- DFS function call will run through all the unvisited neighboring ‘O’s in all 4 directions and mark them as visited so that they are not converted to ‘X’ in the future. The DFS function will not be called for the already visited elements to save time, as they have already been traversed.
- When all the boundaries are traversed and corresponding sets of ‘O’s are marked as visited, they cannot be replaced with ‘X’. All the other remaining unvisited ‘O’s are replaced with ‘X’. This can be done in the same input matrix as the problem talks about replacing the values, otherwise tampering with data is not advised.

Consider the following illustration to understand how DFS traverses the matrix and replaces O's with X's.

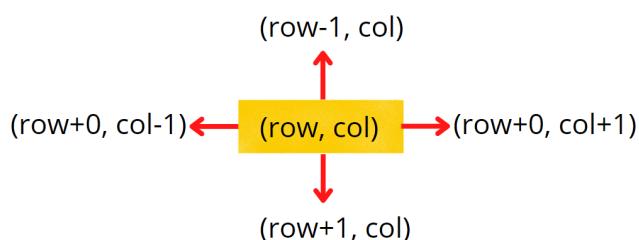
	0	1	2	3	4
0	X	X	X	X	X
1	X	O	O	X	O
2	X	X	O	X	O
3	X	O	X	O	X
4	O	O	X	X	X

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

Visited Matrix

How do set boundaries for 4 directions?

The 4 neighbors will have the following indexes:



Now, either we can apply 4 conditions or follow the following method.

From the above image, it is clear that the delta change in the row is -1, +0, +1, +0. Similarly, the delta change in the column is 0, +1, +0, -1. So we can apply the same logic to find the neighbors of a particular pixel (<row, column>).

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution{
private:
    void dfs(int row, int col, vector<vector<int>> &vis,
    vector<vector<char>> &mat, int delrow[], int delcol[]) {
        vis[row][col] = 1;
        int n = mat.size();
        int m = mat[0].size();

        // check for top, right, bottom, left
        for(int i = 0;i<4;i++) {
            int nrow = row + delrow[i];
            int ncol = col + delcol[i];
            // check for valid coordinates and unvisited 0s
            if(nrow >=0 && nrow <n && ncol >= 0 && ncol < m
            && !vis[nrow][ncol] && mat[nrow][ncol] == '0') {
                dfs(nrow, ncol, vis, mat, delrow, delcol);
            }
        }
    }
public:
    vector<vector<char>> fill(int n, int m,
    vector<vector<char>> mat)
    {
        int delrow[] = {-1, 0, +1, 0};
        int delcol[] = {0, 1, 0, -1};
        vector<vector<int>> vis(n, vector<int>(m,0));
        // traverse first row and last row
        for(int j = 0 ; j<m;j++) {
            // check for unvisited 0s in the boundary rows
            // first row
            if(!vis[0][j] && mat[0][j] == '0') {
                dfs(0, j, vis, mat, delrow, delcol);
            }

            // last row
            if(!vis[n-1][j] && mat[n-1][j] == '0') {
                dfs(n-1,j,vis,mat, delrow, delcol);
            }
        }

        for(int i = 0;i<n;i++) {
            // check for unvisited 0s in the boundary columns
            // first column
            if(!vis[i][0] && mat[i][0] == '0') {
                dfs(i, 0, vis, mat, delrow, delcol);
            }

            // last column
            if(!vis[i][m-1] && mat[i][m-1] == '0') {

```

```

        dfs(i, m-1, vis, mat, delrow, delcol);
    }
}

// if unvisited 0 then convert to X
for(int i = 0;i<n;i++) {
    for(int j= 0 ;j<m;j++) {
        if(!vis[i][j] && mat[i][j] == '0')
            mat[i][j] = 'X';
    }
}

return mat;
}
};

int main(){

vector<vector<char>> mat{
    {'X', 'X', 'X', 'X'},
    {'X', 'O', 'X', 'X'},
    {'X', 'O', 'O', 'X'},
    {'X', 'O', 'X', 'X'},
    {'X', 'X', 'O', 'O'}
};

Solution ob;
// n = 5, m = 4
vector<vector<char>> ans = ob.fill(5, 4, mat);
for(int i = 0;i < 5;i++) {
    for(int j = 0;j < 4;j++) {
        cout<<ans[i][j]<<" ";
    }
    cout<<"\n";
}
return 0;
}
}

```

Output:

```

XXXX
XXXX
XXXX
XXXX
XXOO

```

Time Complexity: $O(N) + O(M) + O(N \times M \times 4) \sim O(N \times M)$, For the worst case, every element will be marked as 'O' in the matrix, and the DFS function will be called for $(N \times M)$ nodes and for every node, we are traversing for 4 neighbors, so it will take $O(N \times M \times 4)$ time. Also, we are running loops for boundary elements so it will take $O(N) + O(M)$.

Space Complexity ~ $O(N \times M)$, $O(N \times M)$ for the visited array, and auxiliary stack space takes up $N \times M$ locations at max.

Special thanks to Vanshika Singh Gour for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article. If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com



Watch Video At: <https://youtu.be/BtdgAys4yMk>

Number of Enclaves

Problem Statement: You are given an $N \times M$ binary matrix grid, where 0 represents a sea cell and 1 represents a land cell. A move consists of walking from one land cell to another adjacent (4-directionally) land cell or walking off the boundary of the grid. Find the number of land cells in the grid for which we cannot walk off the boundary of the grid in any number of moves.

Examples:

Example 1:

Input:

0	0	0	0
1	0	1	0
0	1	1	0
0	0	0	0

Output: 3

Explanation: The highlighted cells represent the land cells.

0	0	0	0
1	0	1	0
0	1	1	0
0	0	0	0

Example 2:

Input:

0	0	0	1
0	1	1	0
0	1	1	0
0	0	0	1
0	1	1	0

Output: 4

Explanation: The highlighted cells represent the land cells.

Solution

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Intuition:

The land cells present in the boundary cannot be counted in the answer as we will walk off the boundary of the grid. Also, land cells connected to the boundary land cell can never be the answer.

The intuition is that we need to figure out the boundary land cells, go through their connected land cells and mark them as visited. The sum of all the remaining land cells will be the answer.

Approach:

We can follow either of the traversal techniques as long as we are starting with a boundary element and marking all those 1s connected to it. We will be solving it using BFS traversal, but you can apply DFS traversal as well, we have applied DFS traversal to solve a similar problem in the previous article.

Breadth First Search, BFS is a traversal technique where we visit the nodes level-wise, i.e., it visits the same level nodes simultaneously, and then moves to the next level.

Initial configuration:

- **Queue:** Define a queue and insert the coordinates of the cell <row, column> which are in the boundary and are marked as 1. The boundary cells will always have row $i = 0$ or row $i = n-1$ or col $j = 0$ or col $j = m-1$.
- **Visited array:** an array initialized to 0 indicating unvisited cells, apart from the ones in the boundary which are already in the queue ds.

The algorithm steps are as follows:

- For BFS traversal, we need a queue data structure and a visited array. Create a corresponding visited array.
- Push the coordinates of boundary nodes in the queue and mark them as visited.
- Start the BFS traversal, pop out an element from the queue every time and travel to all its ***unvisited neighboring land cells*** in the 4 directions. For every unvisited node, push it {row, col} into the Q and mark it as visited to avoid multiple traversals in the future.
- Repeat the steps until the queue becomes empty. When all the boundaries are traversed and corresponding sets of 1s are marked as visited, use a counter variable to count the number of remaining unvisited land cells.
- Return the value of the counter as it indicates the number of land cells that cannot cross the boundary.

Consider the following illustration to understand how BFS traverses the matrix and finds the number of land cells in the grid for which we cannot walk off the boundary of the grid in any number of moves.

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

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

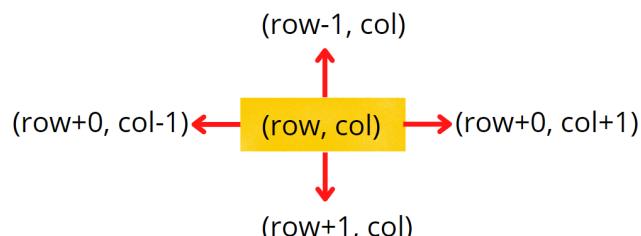
Visited Matrix



Queue

How do set boundaries for 4 directions?

The 4 neighbors will have the following indexes:



Now, either we can apply 4 conditions or follow the following method. From the above image, it is clear that the delta change in the row is -1, +0, +1, +0. Similarly, the delta change in the column is 0, +1, +0, -1. So we can apply the same logic to find the neighbors of a particular pixel ($\langle \text{row}, \text{column} \rangle$).

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
    int numberOfEnclaves(vector<vector<int>> &grid) {
        queue<pair<int,int>> q;
        int n = grid.size();
        int m = grid[0].size();
        int vis[n][m] = {0};
        // traverse boundary elements
        for(int i = 0;i<n;i++) {
            for(int j = 0;j<m;j++) {
                // first row, first col, last row, last col
                if(i == 0 || j == 0 || i == n-1 || j == m-1) {
                    // if it is a land then store it in queue
                    if(grid[i][j] == 1) {
                        q.push({i, j});
                        vis[i][j] = 1;
                    }
                }
            }
        }

        int delrow[] = {-1, 0, +1, 0};
        int delcol[] = {0, +1, +0, -1};

        while(!q.empty()) {
            int row = q.front().first;
            int col = q.front().second;
            q.pop();

            // traverses all 4 directions
            for(int i = 0;i<4;i++) {
                int nrow = row + delrow[i];
                int ncol = col + delcol[i];
                // check for valid coordinates and for land cell
                if(nrow >=0 && nrow <n && ncol >=0 && ncol < m
                && vis[nrow][ncol] == 0 && grid[nrow][ncol] == 1) {
                    q.push({nrow, ncol});
                    vis[nrow][ncol] = 1;
                }
            }
        }

        int cnt = 0;
        for(int i = 0;i<n;i++) {
            for(int j = 0;j<m;j++) {
                // check for unvisited land cell
                if(grid[i][j] == 1 & vis[i][j] == 0)
                    cnt++;
            }
        }
    }
}

```

```

        }
    }
    return cnt;
}
};

int main() {

    vector<vector<int>> grid{
        {0, 0, 0, 0},
        {1, 0, 1, 0},
        {0, 1, 1, 0},
        {0, 0, 0, 0}};

    Solution obj;
    cout << obj.numberOfEnclaves(grid) << endl;
}

```

Output: 3

Time Complexity: $O(N \times M \times 4) \sim O(N \times M)$, For the worst case, assuming all the pieces as land, the BFS function will be called for $(N \times M)$ nodes and for every node, we are traversing for 4 neighbors, so it will take $O(N \times M \times 4)$ time.

Space Complexity $\sim O(N \times M)$, $O(N \times M)$ for the visited array, and queue space takes up $N \times M$ locations at max.

Special thanks to [Vanshika Singh Gour](#) for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, [please check out this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com

Word Ladder – I : G-29

 takeuforward.org/graph/word-ladder-i-g-29

October 21, 2022

Given are the two distinct words **startWord** and **targetWord**, and a list denoting **wordList** of unique words of equal lengths. Find the length of the shortest transformation sequence from startWord to targetWord.

In this problem statement, we need to keep the following conditions in mind:

- A word can only consist of lowercase characters.
- Only one letter can be changed in each transformation.
- Each transformed word must exist in the wordList including the targetWord.
- startWord may or may not be part of the wordList

Note: If there's no possible way to transform the sequence from startWord to targetWord return **0**.

Example 1:

Input:

```
wordList = {"des", "der", "dfr", "dgt", "dfs"}  
startWord = "der", targetWord = "dfs"
```

Output:

3

Explanation:

The length of the smallest transformation sequence from "der" to "dfs" is 3 i.e. "der" -> (replace 'e' by 'f') -> "dfr" -> (replace 'r' by 's') -> "dfs". So, it takes 3 different strings for us to reach the targetWord. Each of these strings are present in the wordList.

Example 2:

Input:

```
wordList = {"geek", "gefk"}  
startWord = "gedk", targetWord= "geek"
```

Output:

2

Explanation:

The length of the smallest transformation sequence from "gedk" to "geek" is 2 i.e. "gedk" -> (replace 'd' by 'e') -> "geek" So, it takes 2 different strings for us to reach the targetWord. Each of these strings are present in the wordList.

Solution

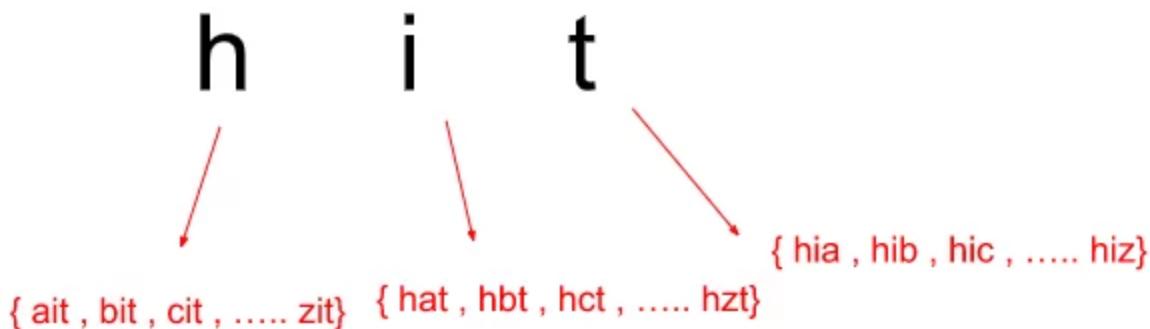
Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Problem Link

Note: In case any image/dry run is not clear please refer to the video attached at the bottom.

Approach:

Let's first understand the Brute force approach to this problem. In Brute force, we just simply replace the startingWord **character by character** and then check whether the transformed word is present in the wordList. If a word is present in the wordList, we try replacing another character in that word by again following similar steps as above, in order to attain the targetWord. We do this for all the characters in the startWord and then eventually return the minimum length of transforming the startWord to targetWord.



Now, to make this algorithm a little less time-consuming and easier, we implement this using a BFS traversal technique.

Initial configuration:

- **Queue:** Define a queue data structure to store the BFS traversal.
- **Hash set:** Create a hash set to store the elements present in the word list to carry out the search and delete operations in O(1) time.

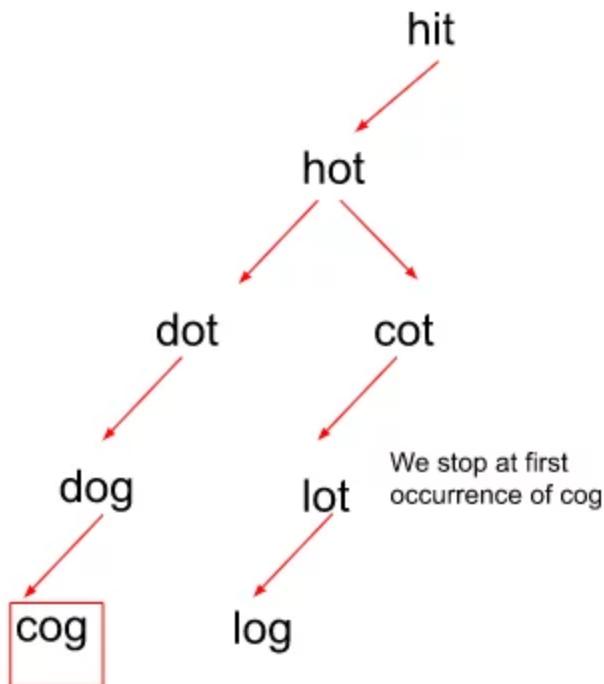
The Algorithm involves the following steps:

- Firstly, we start by creating a queue data structure in order to store the word and the length of the sequence to reach that word as a pair. We store them in form of {word, steps}.
- Then, we push the startWord into the queue with length as '1' indicating that this is the word from which the sequence needs to start from.
- We now create a hash set wherein, we put all the elements in wordList to keep a check on if we've visited that word before or not. In order to mark a word as visited here, we simply delete the word from the hash set. There is no point in visiting someone multiple times during the algorithm.

- Now, we pop the first element out of the queue and carry out the BFS traversal where, for each word popped out of the queue, we try to replace every character with ‘a’ – ‘z’, and we get a transformed word. We check if the transformed word is present in the wordList or not.
- If the word is present, we push it in the queue and increase the count of the sequence by 1 and if not, we simply move on to replacing the original character with the next character.
- Remember, we also need to delete the word from the wordList if it matches with the transformed word to ensure that we do not reach the same point again in the transformation which would only increase our sequence length.
- Now, we pop the next element out of the queue ds and if at any point in time, the transformed word becomes the same as the targetWord, we return the count of the steps taken to reach that word. Here, we’re only concerned about the first occurrence of the targetWord because after that it would only lead to an increase in the sequence length which is for sure not **minimum**.
- If a transformation sequence is not possible, return 0.

Intuition:

The intuition behind using the BFS traversal technique for this particular problem is that if we notice carefully, we go on replacing the characters one by one which seems just like we’re moving level-wise in order to reach the destination i.e. the targetWord.



In level-order traversal, when we reach the destination, we stop the traversal. Similar to that, when we reach our targetWord, we terminate the algorithm and return the counted steps.

We no longer continue the algorithm after that because that would only **increase** the step count to reach the targetWord.

Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution
{
public:
    int wordLadderLength(string startWord, string targetWord,
                         vector<string> &wordList)
    {
        // Creating a queue ds of type {word,transitions to reach 'word'}.
        queue<pair<string, int>> q;

        // BFS traversal with pushing values in queue
        // when after a transformation, a word is found in wordList.
        q.push({startWord, 1});

        // Push all values of wordList into a set
        // to make deletion from it easier and in less time complexity.
        unordered_set<string> st(wordList.begin(), wordList.end());
        st.erase(startWord);
        while (!q.empty())
        {
            string word = q.front().first;
            int steps = q.front().second;
            q.pop();

            // we return the steps as soon as
            // the first occurrence of targetWord is found.
            if (word == targetWord)
                return steps;

            for (int i = 0; i < word.size(); i++)
            {
                // Now, replace each character of 'word' with char
                // from a-z then check if 'word' exists in wordList.
                char original = word[i];
                for (char ch = 'a'; ch <= 'z'; ch++)
                {
                    word[i] = ch;
                    // check if it exists in the set and push it in the queue.
                    if (st.find(word) != st.end())
                    {
                        st.erase(word);
                        q.push({word, steps + 1});
                    }
                }
                word[i] = original;
            }
        }
        // If there is no transformation sequence possible
        return 0;
    }
}

```

```

};

int main()
{
    vector<string> wordList = {"des", "der", "dfr", "dgt", "dfs"};
    string startWord = "der", targetWord = "dfs";

    Solution obj;

    int ans = obj.wordLadderLength(startWord, targetWord, wordList);

    cout << ans;
    cout << endl;
    return 0;
}

```

Output:

3

Time Complexity: $O(N * M * 26)$

Where N = size of wordList Array and M = word length of words present in the wordList..

Note that, hashing operations in an **unordered set** takes $O(1)$ time, but if you want to use **set** here, then the time complexity would increase by a factor of $\log(N)$ as hashing operations in a set take $O(\log(N))$ time.

Space Complexity: $O(N)$ { for creating an unordered set and copying all values from wordList into it }

Where N = size of wordList Array.

*Special thanks to **Priyanshi Goel** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article. If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com*



Watch Video At: <https://youtu.be/tRPda0rcf8E>

G-30 : Word Ladder-II

 takeuforward.org/graph/g-30-word-ladder-ii

November 9, 2022

Given two distinct words **startWord** and **targetWord**, and a list denoting **wordList** of unique words of equal lengths. Find all shortest transformation sequence(s) from **startWord** to **targetWord**. You can return them in any order possible.

In this problem statement, we need to keep the following conditions in mind:

- A word can only consist of lowercase characters.
- Only one letter can be changed in each transformation.
- Each transformed word must exist in the **wordList** including the **targetWord**.
- **startWord** may or may not be part of the **wordList**.
- Return an empty list if there is no such transformation sequence.

Note: Please watch the [previous video](#) of this series before moving on to this particular problem as this is just an extension of the problem [Word Ladder-I](#) that is being discussed previously. The approach used for this problem would be similar to the approach used in that question.

Examples:

Example 1:

Input:

```
startWord = "der", targetWord = "dfs",
wordList = {"des", "der", "dfr", "dgt", "dfs"}
```

Output:

```
[ [ "der", "dfr", "dfs" ], [ "der", "des", "dfs" ] ]
```

Explanation:

The length of the smallest transformation sequence here is 3.

Following are the only two shortest ways to get to the **targetWord** from the **startWord** :

"der" -> (replace 'r' by 's') -> "des" -> (replace 'e' by 'f') -> "dfs".

"der" -> (replace 'e' by 'f') -> "dfr" -> (replace 'r' by 's') -> "dfs".

Example 2:

Input:

```
startWord = "gedk", targetWord= "geek"
wordList = {"geek", "gefk"}
```

Output:

```
[ [ "gedk", "geek" ] ]
```

Explanation:

The length of the smallest transformation sequence here is 2.

Following is the only shortest way to get to the **targetWord** from the **startWord** :

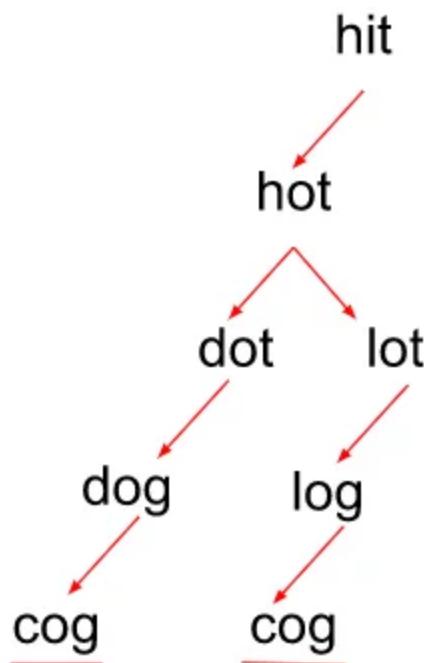
"gedk" -> (replace 'd' by 'e') -> "geek".

Disclaimer: Don't jump directly to the solution, try it out yourself first.

Note: In case any image/dry run is not clear please refer to the video attached at the bottom.

Intuition:

The intuition behind using the BFS traversal technique for these kinds of problems is that if we notice carefully, we go on replacing the characters one by one which seems just like we're moving level-wise in order to reach the destination i.e. the targetWord. Here, in the example below we can notice there are two possible paths in order to reach the targetWord.



Contrary to the previous problem, here we do not stop the traversal on the first occurrence of the targetWord, but rather continue it for as many occurrences of the word as possible as we need **all** the shortest possible sequences in order to reach the destination word. The only trick here is that we **do not** have to delete a particular word immediately from the wordList even if during the replacement of characters it matches with the transformed word. Instead, we delete it after the traversal for a particular level when completed which allows us to explore all possible paths. This allows us to discover multiple sequences in order to reach the targetWord involving similar words.

From the above figure, we can configure that there can be 2 shortest possible sequences in order to reach the word 'cog'.

Approach:

This problem uses the BFS traversal technique for finding out all the shortest possible transformation sequences by exploring all possible ways in which we can reach the targetWord.

Initial configuration:

- **Queue:** Define a queue data structure to store the level-wise formed sequences. The queue will be storing a List of strings, which will be representing the path till now. The last word in the list will be the last converted word.
- **Hash set:** Create a hash set to store the elements present in the word list to carry out the search and delete operations in O(1) time.
- **Vector:** Define a 1D vector ‘usedOnLevel’ to store the words which are currently being used for transformation on a particular level and a 2D vector ‘ans’ for storing all the shortest sequences of transformation.

The Algorithm for this problem involves the following steps:

- Firstly, we start by creating a hash set to store all the elements present in the wordList which would make the search and delete operations faster for us to implement.
- Next, we create a Queue data structure for storing the successive sequences/ path in the form of a vector which on transformation would lead us to the target word.
- Now, we add the startWord to the queue as a List and also push it into the usedOnLevel vector to denote that this word is currently being used for transformation in this particular level.
- Pop the first element out of the queue and carry out the BFS traversal, where for each word that popped out from the back of the sequence present at the top of the queue, we check for all of its characters by replacing them with ‘a’ – ‘z’ if they are present in the wordList or not. In case a word is present in the wordList, we simply first push it onto the usedOnLevel vector and do not delete it from the wordList immediately.
- Now, push that word into the vector containing the previous sequence and add it to the queue. So we will get a new path, but we need to explore other paths as well, so pop the word out of the list to explore other paths.
- After completion of traversal on a particular level, we can now delete all the words that were currently being used on that level from the usedOnLevel vector which ensures that these words won’t be used again in the future, as using them in the later stages will mean that it won’t be the shortest path anymore.
- If at any point in time we find out that the last word in the sequence present at the top of the queue is equal to the target word, we simply push the sequence into the resultant vector if the resultant vector ‘ans’ is empty.
- If the vector is not empty, we check if the current sequence length is equal to the first element added in the ans vector or not. This has to be checked because we need the shortest possible transformation sequences.
- In case, there is no transformation sequence possible, we return an empty 2D vector.

Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution
{
public:
    vector<vector<string>> findSequences(string beginWord, string endWord,
                                            vector<string> &wordList)
    {
        // Push all values of wordList into a set
        // to make deletion from it easier and in less time complexity.
        unordered_set<string> st(wordList.begin(), wordList.end());

        // Creating a queue ds which stores the words in a sequence which is
        // required to reach the targetWord after successive transformations.
        queue<vector<string>> q;

        // BFS traversal with pushing the new formed sequence in queue
        // when after a transformation, a word is found in wordList.

        q.push({beginWord});

        // A vector defined to store the words being currently used
        // on a level during BFS.
        vector<string> usedOnLevel;
        usedOnLevel.push_back(beginWord);
        int level = 0;

        // A vector to store the resultant transformation sequence.
        vector<vector<string>> ans;
        while (!q.empty())
        {
            vector<string> vec = q.front();
            q.pop();

            // Now, erase all words that have been
            // used in the previous levels to transform
            if (vec.size() > level)
            {
                level++;
                for (auto it : usedOnLevel)
                {
                    st.erase(it);
                }
            }

            string word = vec.back();

            // store the answers if the end word matches with targetWord.
            if (word == endWord)
            {
                // the first sequence where we reached end

```

```

        if (ans.size() == 0)
        {
            ans.push_back(vec);
        }
        else if (ans[0].size() == vec.size())
        {
            ans.push_back(vec);
        }
    }
    for (int i = 0; i < word.size(); i++)
    {
        // Now, replace each character of 'word' with char
        // from a-z then check if 'word' exists in wordList.
        char original = word[i];
        for (char c = 'a'; c <= 'z'; c++)
        {
            word[i] = c;
            if (st.count(word) > 0)
            {
                // Check if the word is present in the wordList and
                // push the word along with the new sequence in the queue.
                vec.push_back(word);
                q.push(vec);
                // mark as visited on the level
                usedOnLevel.push_back(word);
                vec.pop_back();
            }
        }
        word[i] = original;
    }
}
return ans;
}

// A comparator function to sort the answer.
bool comp(vector<string> a, vector<string> b)
{
    string x = "", y = "";
    for (string i : a)
        x += i;
    for (string i : b)
        y += i;

    return x < y;
}

int main()
{
    vector<string> wordList = {"des", "der", "dfr", "dgt", "dfs"};
    string startWord = "der", targetWord = "dfs";
}

```

```

Solution obj;
vector<vector<string>> ans = obj.findSequences(startWord, targetWord, wordList);

// If no transformation sequence is possible.
if (ans.size() == 0)
    cout << -1 << endl;
else
{
    sort(ans.begin(), ans.end(), comp);
    for (int i = 0; i < ans.size(); i++)
    {
        for (int j = 0; j < ans[i].size(); j++)
        {
            cout << ans[i][j] << " ";
        }
        cout << endl;
    }
}

return 0;
}

```

Output:

der des dfs

der dfr dfs

Time Complexity and Space Complexity: It cannot be predicted for this particular algorithm because there can be multiple sequences of transformation from startWord to targetWord depending upon the example, so we cannot define a fixed range of time or space in which this program would run for all the test cases.

Note: This approach/code will give TLE when solved on the Leetcode platform due to the strict time constraints being put up there. So, you need to optimize it to a greater extent in order to pass all the test cases for LeetCode. For the optimized approach to this question please check out the [next video](#).

Special thanks to [Priyanshi Goel](#) for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, [please check out this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com



Watch Video At: <https://youtu.be/DREutrv2XD0>

Number of Islands

Problem Statement: Given a grid of size NxM (N is the number of rows and M is the number of columns in the grid) consisting of '0's (Water) and '1's(Land). Find the number of islands.

Note: An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically or diagonally i.e., in all 8 directions.

Pre-req: Connected Components, Graph traversal techniques

Examples:

Example 1:

Input:

0	1	1	0
0	1	1	0
0	0	1	0
0	0	0	0
1	1	0	1

Output: 3

Explanation:

0	1	1	0
0	1	1	0
0	0	1	0
0	0	0	0
1	1	0	1

There are 3 islands as the different components are surrounded by water (i.e. 0), and there is no land connectivity in either of the 8 directions hence separating them into 3 islands.

Example 2:

Input:

0	1	1	0	1	0	0
0	0	1	1	0	1	0

Output: 1

Explanation:

0	1	1	0	1	0	0
0	0	1	1	0	1	0

All lands are connected. So, only 1 island is present.

Solution

Disclaimer: Don't jump directly to the solution, try it out yourself first.

Intuition:

An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically or diagonally i.e., in all 8 directions.



The question arises how can this problem be solved using a graph? Try to think of all the elements as a node or a vertex, we can observe they are connected in some way as all 8 directions connectivity is allowed. If we start a traversal algorithm, from a particular 1 (land) it will make sure it traverses the nearest 1 (land). So, one traversal with a starting point will cover an island. If we do 3 traversals then we will have 3 starting nodes, anyone can be considered as a starting node in an island, and make sure it visits everyone. In the following example, if we take 3 starting nodes we will be able to touch all the pieces of land. Hence, one starting node makes sure it touches all the connected lands. The basic idea is that “*one starting node represents one island*”. So, we just need to figure out the number of starting points.

Approach:

In any traversal technique, we have one starting node and it traverses all the nodes in the graph. We know about both the traversals, Breadth First Search (BFS) and Depth First Search (DFS). We can use any of the traversals to solve this problem, in our case we will be using BFS.

The algorithm steps are as follows:

- The pairs of row and column (<row, column>) will represent the node numbers.
- For BFS traversal, we need a queue data structure and a visited array. Create a replica of the given array, i.e., create another array of the same size and call it a visited array. We can use the same matrix, but we will avoid alteration of the original data.
- In the queue, insert a vertex (pair of <row, column>) and mark it as visited.
- While BFS traversal, pop out an element from the queue and travel to all its neighbours. In a graph, we store the list of neighbours in an adjacency list but here we know the neighbours are in 8 directions.
- We go in all 8 directions and check for unvisited land neighbours. To travel in 8 directions we will use nested loops, you can find the implementation details in the code.
- BFS function call will make sure that it starts the BFS call from that unvisited land, and visits all the nodes that are on that island, and at the same time, it will also mark them as visited.
- Since the nodes travelled in a traversal will be marked as visited, they will no further be called for any further BFS traversal.
- Keep repeating these steps, for every land that you find unvisited, and visit the entire island.

- Add a counter variable to count the number of times the BFS function is called, as in this way we can count the total number of starting nodes, which will give us the number of islands.

In the following example there are 3 islands, i.e., we will have 3 starting nodes.

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

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

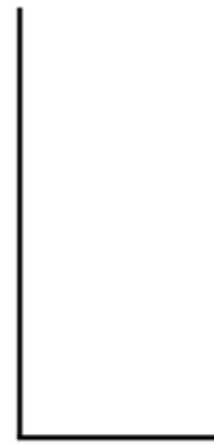
Let us understand how BFS traverses the nodes and covers an island.

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

starting node:

	0	1	2	3
0				
1				
2				
3				
4				

Visited Array



Queue

The above illustration illustrates the traversal of the starting node (0, 1). Similarly, we will have BFS traversal for (4, 0) and (4, 3) starting nodes. So, 3 starting nodes represent 3 islands.

How to decide the starting points?

Start from (0, 0), if you get water (i.e., 0) move to the next index, otherwise, if you find land (i.e., 1) then call BFS traversal for that pair (<row, column>), which will be a starting node. Repeat the step, and call BFS traversal only for *unvisited land pairs* (<row, column>).

Pseudo Code:

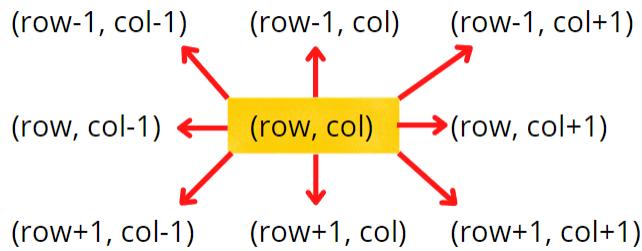
```

for ( row := 0 to n)
{
    for ( col := 0 to m)
    {
        if (! vis[row][col])
        {
            bfs(row, col);
            // count number of starting points
            cnt++;
        }
    }
}

```

How do set boundaries for 8 directions?

The 8 neighbours will have the following indexes:



Now, either we can apply 8 conditions or follow the following method. From the above image, it is clear that the row can be row-1, row, or row+1, i.e., the delta row varies from -1 to 1. Similarly, the delta column varies from -1 to 1. So we can apply the same logic to find the neighbours of a particular pair (<row, column>).

```

for (deltarow := -1 to +1)
{
    for (deltacol := -1 to +1)
    {
        neighbor = row + deltarow;
        neighbor = col + deltacol;
    }
}

```

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
    void bfs(int row, int col, vector<vector<int>> &vis, vector<vector<char>>&grid) {
        // mark it visited
        vis[row][col] = 1;
        queue<pair<int,int>> q;
        // push the node in queue
        q.push({row, col});
        int n = grid.size();
        int m = grid[0].size();

        // until the queue becomes empty
        while(!q.empty()) {
            int row = q.front().first;
            int col = q.front().second;
            q.pop();

            // traverse in the neighbours and mark them if its a land
            for(int delrow = -1; delrow<=1;delrow++) {
                for(int delcol = -1; delcol <= 1; delcol++) {
                    int nrow = row + delrow;
                    int ncol = col + delcol;
                    // neighbour row and column is valid, and is an unvisited land
                    if(nrow >= 0 && nrow < n && ncol >= 0 && ncol < m
                    && grid[nrow][ncol] == '1' && !vis[nrow][ncol]) {
                        vis[nrow][ncol] = 1;
                        q.push({nrow, ncol});
                    }
                }
            }
        }
    }
public:
    // Function to find the number of islands.
    int numIslands(vector<vector<char>>& grid) {
        int n = grid.size();
        int m = grid[0].size();
        // create visited array and initialise to 0
        vector<vector<int>> vis(n, vector<int>(m, 0));
        int cnt = 0;
        for(int row = 0; row < n ; row++) {
            for(int col = 0; col < m ;col++) {
                // if not visited and is a land
                if(!vis[row][col] && grid[row][col] == '1') {
                    cnt++;
                    bfs(row, col, vis, grid);
                }
            }
        }
    }
}

```

```

        return cnt;
    }

};

int main() {
    // n: row, m: column
    vector<vector<char>> grid
    {
        {'0', '1', '1', '1', '0', '0', '0'},
        {'0', '0', '1', '1', '0', '1', '0'}
    };

    Solution obj;
    cout << obj.numIslands(grid) << endl;

    return 0;
}

```

Output: 2

Time Complexity ~ $O(N^2 + NxMx9)$, N^2 for the nested loops, and $NxMx9$ for the overall DFS of the matrix, that will happen throughout if all the cells are filled with 1.

Space Complexity: $O(N^2)$ for visited array max queue space $O(N^2)$, If all are marked as 1 then the maximum queue space will be N^2 .

Special thanks to [Vanshika Singh Gour](#) for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, [please check out this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com



Watch Video At: <https://youtu.be/muncqIKJrH0>

Bipartite Graph | DFS Implementation

 takeuforward.org/graph/bipartite-graph-dfs-implementation

September 6, 2022

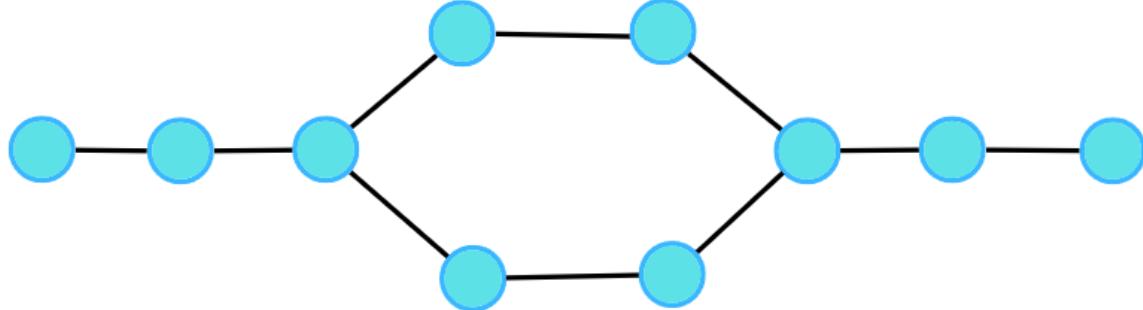
Problem Statement: Given an adjacency list of a graph adj of V no. of vertices having 0 based index. Check whether the graph is bipartite or not.

If we are able to colour a graph with two colours such that no adjacent nodes have the same colour, it is called a bipartite graph.

Examples:

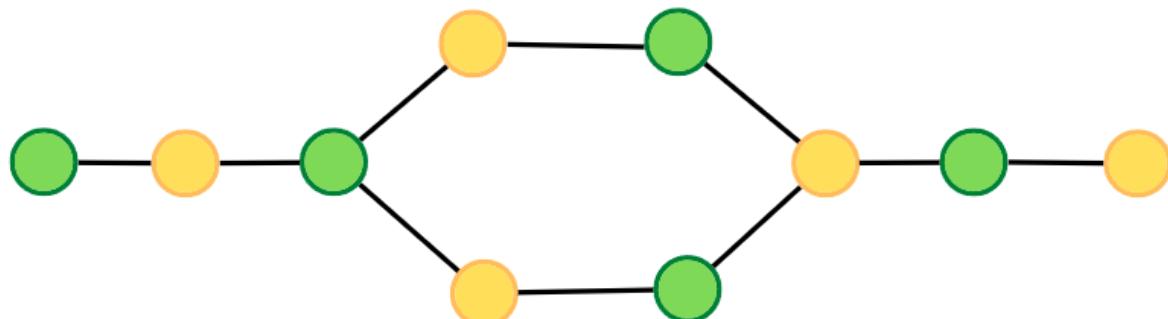
Example 1:

Input:



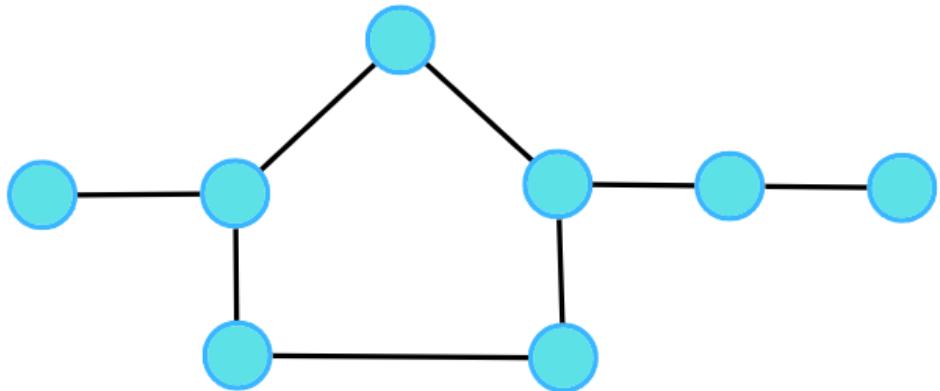
Output: 1

Explanation:



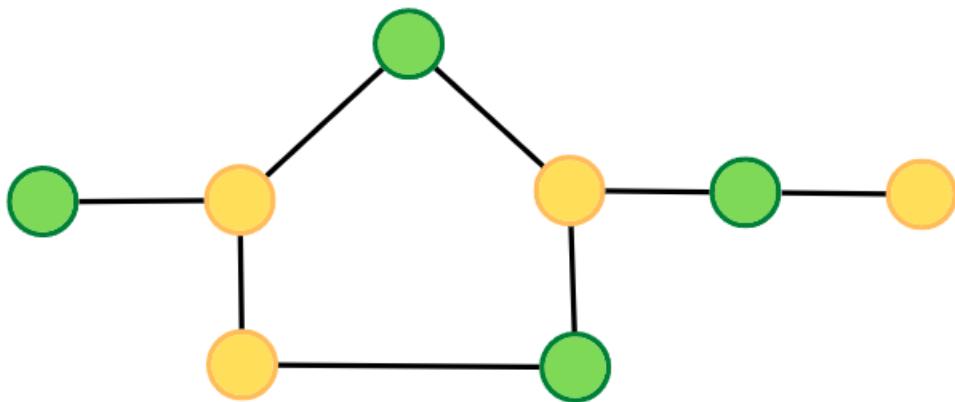
Example 2:

Input:



Output: 0

Explanation:



Solution

***Disclaimer:** Don't jump directly to the solution, try it out yourself first.*

Intuition:

A bipartite graph is a graph which can be coloured using 2 colours such that no adjacent nodes have the same colour. Any linear graph with no cycle is always a bipartite graph. With a cycle, any graph with an even cycle length can also be a bipartite graph. So, any graph with an odd cycle length can never be a bipartite graph.

The intuition is the brute force of filling colours using any traversal technique, just make sure no two adjacent nodes have the same colour. If at any moment of traversal, we find the adjacent nodes to have the same colour, it means that there is an odd cycle, or it cannot be a bipartite graph.

Approach:

We can follow either of the traversal techniques. In this article, we will be solving it using DFS traversal.

DFS is a traversal technique which involves the idea of recursion and backtracking. DFS goes in-depth, i.e., traverses all nodes by going ahead, and when there are no further nodes to traverse in the current path, then it backtracks on the same path and traverses other unvisited nodes.

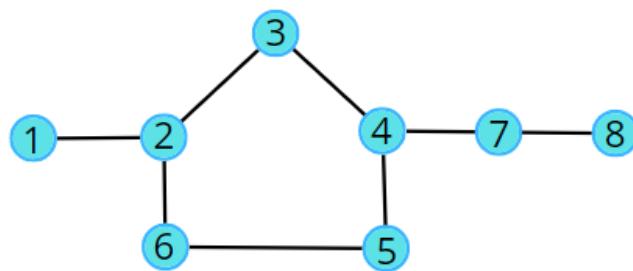
We will be defining the DFS traversal below, but this check has to be done for every component, for that we can use the simple for loop concept that we have learnt, to call the traversals for unvisited nodes.

```
// check for connected components in a graph
for ( i = 1; i<= n; i++ )
{
    if(!vis[i])
    {
        if( dfs(i) == true)
            return true;
    }
}
return false;
```

The algorithm steps are as follows:

- For DFS traversal, we need a start node and a visited array but in this case, instead of a visited array, we will take a colour array where all the nodes are initialised to -1 indicating they are not coloured yet.
- In the DFS function call, make sure to pass the value of the assigned colour, and assign the same in the colour array. We will try to colour with 0 and 1, but you can choose other colours as well. We will start with the colour 0, you can start with 1 as well, just make sure for the adjacent node, it should be opposite of what the current node has.
- In DFS traversal, we travel in-depth to all its uncoloured neighbours using the adjacency list. For every uncoloured node, initialise it with the opposite colour to that of the current node.
- If at any moment, we get an adjacent node from the adjacency list which is already coloured and has the same colour as the current node, we can say it is not possible to colour it, hence it cannot be bipartite. Thereby return a false indicating the given graph is not bipartite; otherwise, keep on returning true.

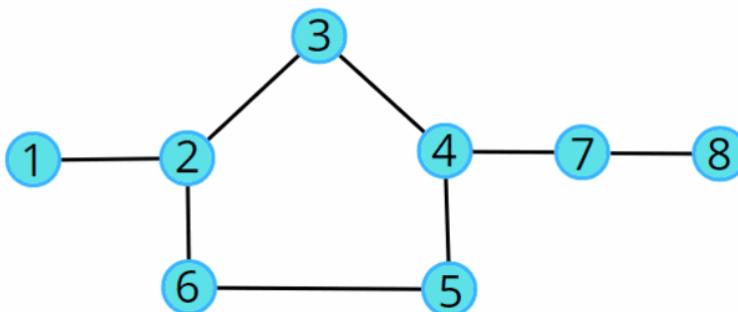
Consider the following graph and its adjacency list.



1	→ { 2 }
2	→ { 1, 3, 6 }
3	→ { 2, 4 }
4	→ { 3, 7, 5 }
5	→ { 4, 6 }
6	→ { 2, 5 }
7	→ { 4, 8 }
8	→ { 7 }

Adjacency List

Consider the following illustration to understand the colouring of the nodes using DFS traversal.



1	2	3	4	5	6	7	8
-1	-1	-1	-1	-1	-1	-1	-1

Colour Array

Code:

- C++ Code
- Java Code

```

#include<bits/stdc++.h>
using namespace std;

class Solution {
private:
    bool dfs(int node, int col, int color[], vector<int> adj[]) {
        color[node] = col;

        // traverse adjacent nodes
        for(auto it : adj[node]) {
            // if uncoloured
            if(color[it] == -1) {
                if(dfs(it, !col, color, adj) == false) return false;
            }
            // if previously coloured and have the same colour
            else if(color[it] == col) {
                return false;
            }
        }

        return true;
    }
public:
    bool isBipartite(int V, vector<int>adj[]){
        int color[V];
        for(int i = 0;i<V;i++) color[i] = -1;

        // for connected components
        for(int i = 0;i<V;i++) {
            if(color[i] == -1) {
                if(dfs(i, 0, color, adj) == false)
                    return false;
            }
        }
        return true;
    }
};

void addEdge(vector <int> adj[], int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
}

int main(){
    // V = 4, E = 4
    vector<int>adj[4];

    addEdge(adj, 0, 2);
    addEdge(adj, 0, 3);
    addEdge(adj, 2, 3);
    addEdge(adj, 3, 1);
}

```

```
Solution obj;
bool ans = obj.isBipartite(4, adj);
if(ans)cout << "1\n";
else cout << "0\n";

return 0;
}
```

Output: 0

Time Complexity: $O(V + 2E)$, Where V = Vertices, $2E$ is for total degrees as we traverse all adjacent nodes.

Space Complexity: $O(3V) \sim O(V)$, Space for DFS stack space, colour array and an adjacency list.

Special thanks to [Vanshika Singh Gour](#) for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article. If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com

Detect cycle in a directed graph (using DFS) : G 19

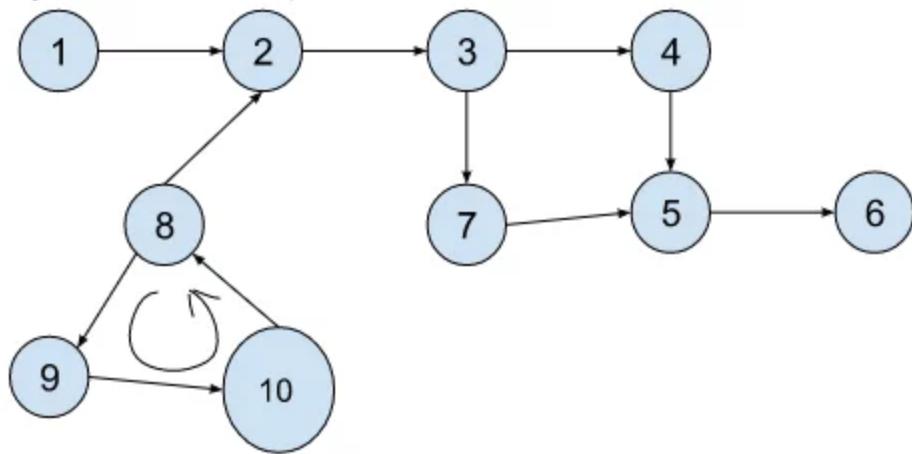
 takeuforward.org/data-structure/detect-cycle-in-a-directed-graph-using-dfs-g-19

October 11, 2022

Problem Statement: Given a directed graph with V vertices and E edges, check whether it contains any cycle or not.

Example 1:

Input: $N = 10, E = 11$

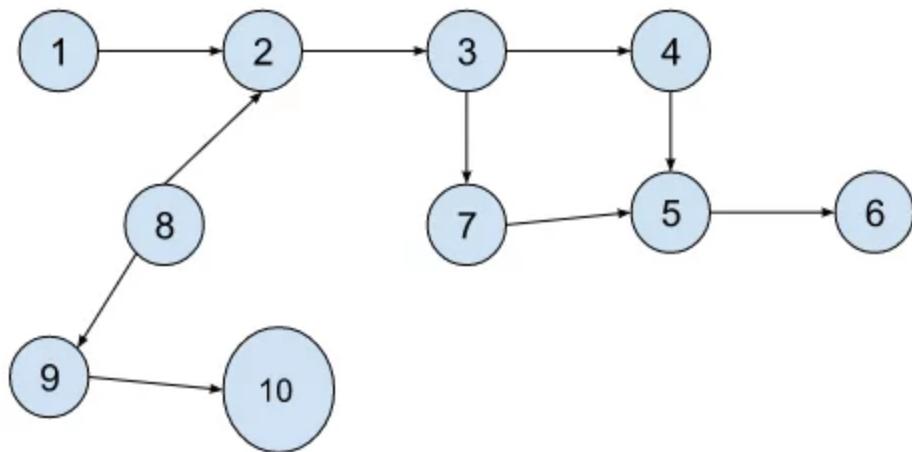


Output: true

Explanation: $8 \rightarrow 9 \rightarrow 10$ is a cycle.

Example 2:

Input Format: N = 10, E = 10



Result: false

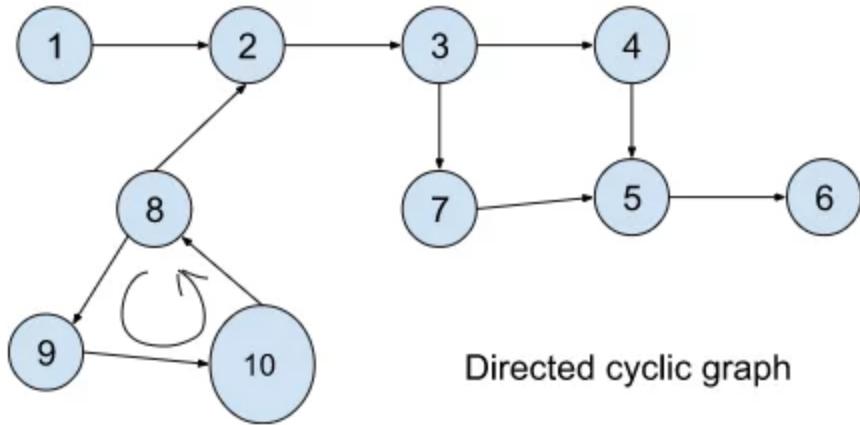
Explanation: No cycle detected.

Solution

Disclaimer: Don't jump directly to the solution, try it out yourself first.

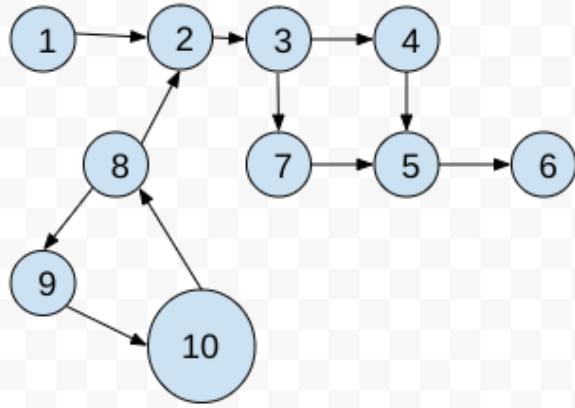
Intuition:

In a **Directed Cyclic Graph**, during traversal, if we end up at a node, which we have visited previously in the path, that means we came around a circle and ended up at this node, which determines that it has a cycle. Previously, we have learned a similar technique to **detect cycles in an Undirected Graph (using DFS)**. In that method, the algorithm returns true, if it finds an adjacent node that is previously visited and not a parent of the current node. But the same algorithm will not work in this case. Let's understand why this happens considering the below graph.



- You can also look at the GIF below, in case you fail to understand the below points.
- Let's start DFS from node 1. It will follow the path $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6$, all the nodes including 5 will be visited as marked.
- As there are no further nodes after node 6, DFS will backtrack to node 3 and will follow the path: $3 \rightarrow 7 \rightarrow 5 \rightarrow 6$. It followed this path because this path was left to be explored.
- Reaching node 7, the adjacent node 5 can be found previously visited, but ideally, it should not have been visited, as we did not visit this node in a continuous path. At this point, the algorithm will conclude that this is a cycle and will return true but this is not a cycle as node 5 has been visited twice following two different paths.
- This would have been true if the nodes are connected to undirected edges. But as we are dealing with directed edges this algorithm fails to detect a cycle.
- Due to the above reason, we need to think of an algorithm, which keeps a track of visited nodes, in the traversal only.

The process will be similar as illustrated.

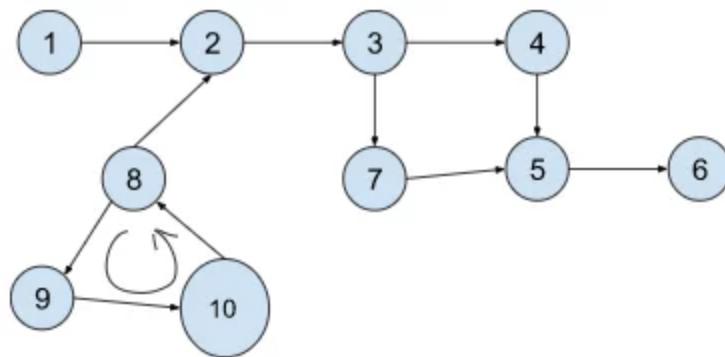


So the intuition is to reach a previously visited node again on the same path. If it can be done, we conclude that the graph has a cycle.

Note: If a directed graph contains a cycle, the node has to be visited again on the same path and not through different paths.

Approach:

Consider the following graph:

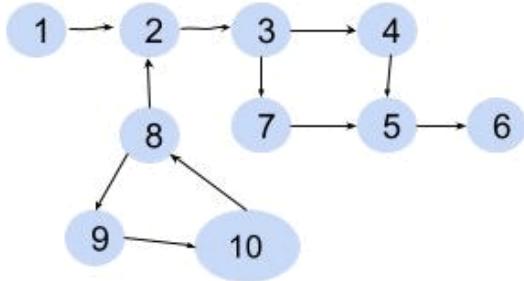


We will be solving it using DFS traversal. DFS goes in-depth, i.e., traverses all nodes by going ahead, and when there are no further nodes to traverse in the current path, then it backtracks on the same path and traverses other unvisited nodes.

The algorithm steps are as follows:

1. We will traverse the graph component-wise using the DFS technique.
2. Make sure to carry two visited arrays in the DFS call. One is a visited array(vis) and the other is a path-visited(pathVis) array. The visited array keeps a track of visited nodes, and the path-visited keeps a track of visited nodes in the current traversal only.
3. While making a DFS call, at first we will mark the node as visited in both the arrays and then will traverse through its adjacent nodes. Now, there may be either of the three cases:
 1. Case 1: If the adjacent node is not visited, we will make a new DFS call recursively with that particular node.
 2. Case 2: If the adjacent node is visited and also on the same path(i.e marked visited in the pathVis array), we will return true, because it means it has a cycle, thereby the pathVis was true. Returning true will mean the end of the function call, as once we have got a cycle, there is no need to check for further adjacent nodes.
 3. Case 3: If the adjacent node is visited but not on the same path(i.e not marked in the pathVis array), we will continue to the next adjacent node, as it would have been marked as visited in some other path, and not on the current one.
4. Finally, if there are no further nodes to visit, we will unmark the current node in the pathVis array and just return false. Then we will backtrack to the previous node with the returned value. The point to remember is, while we enter we mark both the pathVis and vis as true, but at the end of traversal to all adjacent nodes, we just make sure we unmark the pathVis and still keep the vis marked as true, as it will avoid future extra traversal calls.

The following illustration will be useful in understanding the algorithm:



Starting DFS from node 1

Both the arrays are initialized to 0

	1	2	3	4	5	6	7	8	9	10
vis	0	0	0	0	0	0	0	0	0	0
pathVis	0	0	0	0	0	0	0	0	0	0



Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
    bool dfsCheck(int node, vector<int> adj[], int vis[], int pathVis[]) {
        vis[node] = 1;
        pathVis[node] = 1;

        // traverse for adjacent nodes
        for (auto it : adj[node]) {
            // when the node is not visited
            if (!vis[it]) {
                if (dfsCheck(it, adj, vis, pathVis) == true)
                    return true;
            }
            // if the node has been previously visited
            // but it has to be visited on the same path
            else if (pathVis[it]) {
                return true;
            }
        }
        pathVis[node] = 0;
        return false;
    }
public:
    // Function to detect cycle in a directed graph.
    bool isCyclic(int V, vector<int> adj[]) {
        int vis[V] = {0};
        int pathVis[V] = {0};

        for (int i = 0; i < V; i++) {
            if (!vis[i]) {
                if (dfsCheck(i, adj, vis, pathVis) == true) return
true;
            }
        }
        return false;
    }
};

int main() {

    // V = 11, E = 11;
    vector<int> adj[11] = {{}, {2}, {3}, {4, 7}, {5}, {6}, {}, {5}, {9}, {10},
{8}};
    int V = 11;
    Solution obj;
    bool ans = obj.isCyclic(V, adj);
}

```

```

    if (ans)
        cout << "True\n";
    else
        cout << "False\n";

    return 0;
}

```

Output: True

Time Complexity: $O(V+E)+O(V)$, where V = no. of nodes and E = no. of edges. There can be at most V components. So, another $O(V)$ time complexity.

Space Complexity: $O(2N) + O(N) \sim O(2N)$: $O(2N)$ for two visited arrays and $O(N)$ for recursive stack space.

*Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out [this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com*



Watch Video At: <https://youtu.be/9twcmtQj4DU>

Topological Sort Algorithm | DFS: G-21

 takeuforward.org/data-structure/topological-sort-algorithm-dfs-g-21

October 17, 2022

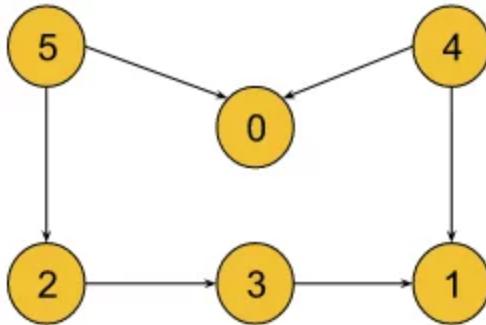


Problem Statement: Given a Directed Acyclic Graph (DAG) with V vertices and E edges, Find any Topological Sorting of that Graph.

Note: In topological sorting, node u will always appear before node v if there is a directed edge from node u towards node v ($u \rightarrow v$).

Example 1:

Input: V = 6, E = 6



Output: 5, 4, 2, 3, 1, 0

Explanation: A graph may have multiple topological sortings.

The result is one of them. The necessary conditions for the ordering are:

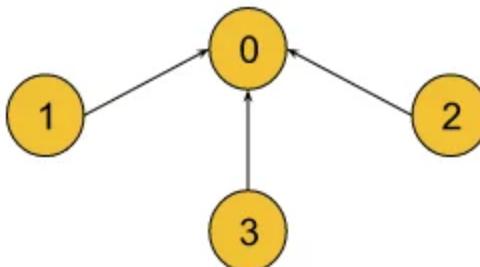
- According to edge $5 \rightarrow 0$, node 5 must appear before node 0 in the ordering.
- According to edge $4 \rightarrow 0$, node 4 must appear before node 0 in the ordering.
- According to edge $5 \rightarrow 2$, node 5 must appear before node 2 in the ordering.
- According to edge $2 \rightarrow 3$, node 2 must appear before node 3 in the ordering.
- According to edge $3 \rightarrow 1$, node 3 must appear before node 1 in the ordering.
- According to edge $4 \rightarrow 1$, node 4 must appear before node 1 in the ordering.

The above result satisfies all the necessary conditions.

[4, 5, 2, 3, 1, 0] is also one such topological sorting that satisfies all the conditions.

Example 2:

Input: V = 4, E = 3



Result: 3, 2, 1, 0

Explanation: The necessary conditions for the ordering are:

- For edge $1 \rightarrow 0$ node 1 must appear before node 0.
- For edge $2 \rightarrow 0$ node 1 must appear before node 0.
- For edge $3 \rightarrow 0$ node 1 must appear before node 0.

[2, 3, 1, 0] is also another topological sorting for the graph.

Solution

Disclaimer: Don't jump directly to the solution, try it out yourself first.

Problem link

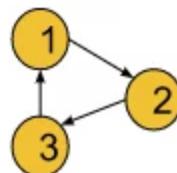
Topological sorting only exists in Directed Acyclic Graph (DAG). If the nodes of a graph are connected through directed edges and the graph does not contain a cycle, it is called a **directed acyclic graph(DAG)**.

The **topological sorting** of a directed acyclic graph is nothing but the linear ordering of vertices such that if there is an edge between node u and v($u \rightarrow v$), node u appears before v in that ordering.

Now, let's understand **Why topological sort only exists in DAG:**

Case 1 (If the edges are undirected): If there is an undirected edge between node u and v, it signifies that there is an edge from node u to v($u \rightarrow v$) as well as there is an edge from node v to u($v \rightarrow u$). But according to the definition of topological sorting, it is practically impossible to write such ordering where **u appears before v** and **v appears before u** simultaneously. So, it is only possible for directed edges.

Case 2 (If the directed graph contains a cycle): The following directed graph contains a cycle:



If we try to get topological sorting of this cyclic graph, for edge $1 \rightarrow 2$, node 1 must appear before 2, for edge $2 \rightarrow 3$, node 2 must appear before 3, and for edge $3 \rightarrow 1$, node 3 must appear before 1 in the linear ordering. But such ordering is not possible as there exists a cyclic dependency in the graph. Thereby, topological sorting is only possible for a directed acyclic graph.

Approach:

We will be solving it using the DFS traversal technique. DFS goes in-depth, i.e., traverses all nodes by going ahead, and when there are no further nodes to traverse in the current path, then it backtracks on the same path and traverses other unvisited nodes.

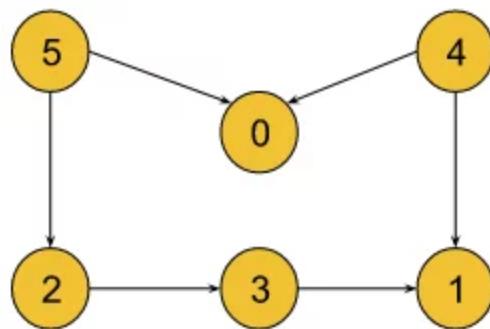
The algorithm steps are as follows:

1. We must traverse all components of the graph.

2. Make sure to carry a visited array(all elements are initialized to 0) and a stack data structure, where we are going to store the nodes after completing the DFS call.
3. In the DFS call, first, the current node is marked as visited. Then DFS call is made for all its adjacent nodes.
4. After visiting all its adjacent nodes, DFS will backtrack to the previous node and meanwhile, the current node is pushed into the stack.
5. Finally, we will get the stack containing one of the topological sortings of the graph.

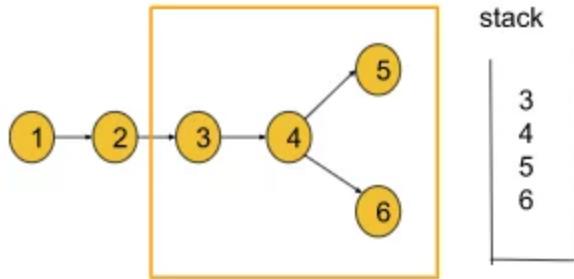
Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Let's quickly understand the algorithm considering the following graph:



- DFS will start from **node 0** and mark it as visited. But it has no adjacent nodes. So the DFS will return putting node 0 into the stack(stack: {0}).
- Then DFS will again start from **node 1** and mark it as visited, but it also has no adjacent nodes. So node 1 is pushed into the stack(stack: {1, 0}) and the DFS call will be over.
- Then DFS will start from **node 2** and mark it as visited as well. It will again call DFS for its adjacent node 3 and mark 3 as visited. After visiting node 3, it will find out that only adjacent node 1 is previously visited.
- So it will backtrack to node 2, putting node 3 first and then node 2 into the stack (stack: {2, 3, 1, 0}).
- Again, DFS will start from **node 4** and mark it as visited. It will find all its adjacent nodes 0 and 1 have been previously visited. So, node 4 will be pushed into the stack(stack: {4, 2, 3, 1, 0}).
- Lastly, DFS will start from **node 5** and mark it as visited. Again, it will find all its adjacent nodes 0 and 2 are previously visited. So, it will return putting node 5 into the stack(stack: {5, 4, 2, 3, 1, 0}).
- Finally, the stack will contain {5, 4, 2, 3, 1, 0}, which is one of the topological sortings of the graph.

Let's understand how the linear orderings are maintained considering the following simple graph:



The linear ordering for the above graph can be 1, 2, 3, 4, 5, 6 or 1, 2, 3, 4, 6, 5. If we closely observe this algorithm, it is designed in such a way that when the DFS call for a node is completed, the node is always kept in the stack. So for example, if the DFS call for 3 is over, we must have the nodes 3, 4, 5, and 6 linearly ordered in the stack. And this is true for every other node. Thus the linear ordering is always maintained for each node of the graph.

Intuition:

Since we are inserting the nodes into the stack after the completion of the traversal, we are making sure, there will be no one who appears afterward but may come before in the ordering as everyone during the traversal would have been inserted into the stack.

Note: *Points to remember, that node will be marked as visited immediately after making the DFS call and before returning from the DFS call, the node will be pushed into the stack.*

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
    void dfs(int node, int vis[], stack<int> &st,
             vector<int> adj[]) {
        vis[node] = 1;
        for (auto it : adj[node]) {
            if (!vis[it]) dfs(it, vis, st, adj);
        }
        st.push(node);
    }
public:
    //Function to return list containing vertices in Topological order.
    vector<int> topoSort(int V, vector<int> adj[])
    {
        int vis[V] = {0};
        stack<int> st;
        for (int i = 0; i < V; i++) {
            if (!vis[i]) {
                dfs(i, vis, st, adj);
            }
        }

        vector<int> ans;
        while (!st.empty()) {
            ans.push_back(st.top());
            st.pop();
        }
        return ans;
    }
};


```

```

int main() {

    //V = 6;
    vector<int> adj[6] = {{}, {}, {3}, {1}, {0, 1}, {0, 2}};
    int V = 6;
    Solution obj;
    vector<int> ans = obj.topoSort(V, adj);

    for (auto node : ans) {
        cout << node << " ";
    }
    cout << endl;

    return 0;
}

```

Output: 5 4 2 3 1 0

Time Complexity: $O(V+E)+O(V)$, where V = no. of nodes and E = no. of edges. There can be at most V components. So, another $O(V)$ time complexity.

Space Complexity: $O(2N) + O(N) \sim O(2N)$: $O(2N)$ for the visited array and the stack carried during DFS calls and $O(N)$ for recursive stack space, where N = no. of nodes.

*Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, [please check out this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com*



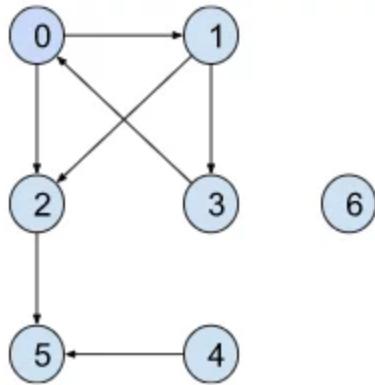
Watch Video At: <https://youtu.be/5IZ0iJMrUMk>

Find Eventual Safe States – DFS: G-20

Problem Statement: A directed graph of V vertices and E edges is given in the form of an adjacency list adj . Each node of the graph is labeled with a distinct integer in the range 0 to $V - 1$. A node is a terminal node if there are no outgoing edges. A node is a safe node if every possible path starting from that node leads to a terminal node. You have to return an array containing all the safe nodes of the graph. The answer should be sorted in ascending order.

Example 1:

Input Format: N = 7, E = 7



Result: {2 4 5 6}

Explanation: Here terminal nodes are 5 and 6 as they have no outgoing edges.

From node 0: two paths are there $0 \rightarrow 2 \rightarrow 5$ and $0 \rightarrow 1 \rightarrow 3 \rightarrow 0$.

The second path does not end at a terminal node. So it is not a safe node.

From node 1: two paths exist: $1 \rightarrow 3 \rightarrow 0 \rightarrow 1$ and $1 \rightarrow 2 \rightarrow 5$.

But the first one does not end at a terminal node. So it is not a safe node.

From node 2: only one path: $2 \rightarrow 5$ and 5 is a terminal node.

So it is a **safe node**.

From node 3: two paths: $3 \rightarrow 0 \rightarrow 1 \rightarrow 3$ and $3 \rightarrow 0 \rightarrow 2 \rightarrow 5$

but the first path does not end at a terminal node.

So it is not a **safe node**.

From node 4: Only one path: $4 \rightarrow 5$ and 5 is a terminal node.

So it is also a **safe node**.

From node 5: It is a terminal node.

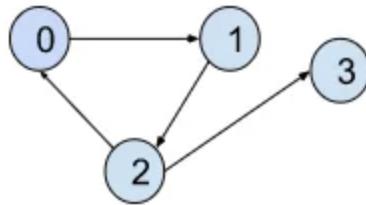
So it is a **safe node** as well.

From node 6: It is a terminal node.

So it is a **safe node** as well.

Example 2:

Input Format: N = 4, E = 4



Result: {3}

Explanation: Node 3 itself is a terminal node and it is a **safe node** as well. But all the paths from other nodes do not lead to a terminal node. So they are excluded from the answer.

Solution

Disclaimer: Don't jump directly to the solution, try it out yourself first.

[Problem Link](#).

Solution:

A **terminal node** is a node without any outgoing edges(i.e outdegree = 0). Now a node is considered to be a **safe node** if all possible paths starting from it lead to a terminal node. Here we need to find out all safe nodes and return them sorted in ascending order.

If we closely observe, all possible paths starting from a node are going to end at some terminal node unless there exists a cycle and the paths return back to themselves. Let's understand it considering the below graph:



- In the above graph, there exists a cycle i.e $0 \rightarrow 1 \rightarrow 3 \rightarrow 0$, and node 7 is connected to the cycle with an incoming edge towards the cycle.
- Some paths starting from these nodes are definitely going to end somewhere in the cycle and not at any terminal node. So, these nodes are not safe nodes.
- Though node 2 is connected to the cycle, the edge is directed outwards the cycle and all the paths starting from it lead to the terminal node 5. So, it is a safe node and the rest of the nodes (4, 5, 6) are safe nodes as well.

So, the intuition is to figure out the nodes which are either a part of a cycle or incoming to the cycle. We can do this easily using the cycle detection technique that was used previously to **detect cycles in a directed graph (using DFS)**.

Note: Points to remember that any node which is a part of a cycle or leads to the cycle through an incoming edge towards the cycle, cannot be a safe node. Apart from these types of nodes, every node is a safe node.

Approach:

We will be solving it using DFS traversal. DFS goes in-depth, i.e., traverses all nodes by going ahead, and when there are no further nodes to traverse in the current path, then it backtracks on the same path and traverses other unvisited nodes.

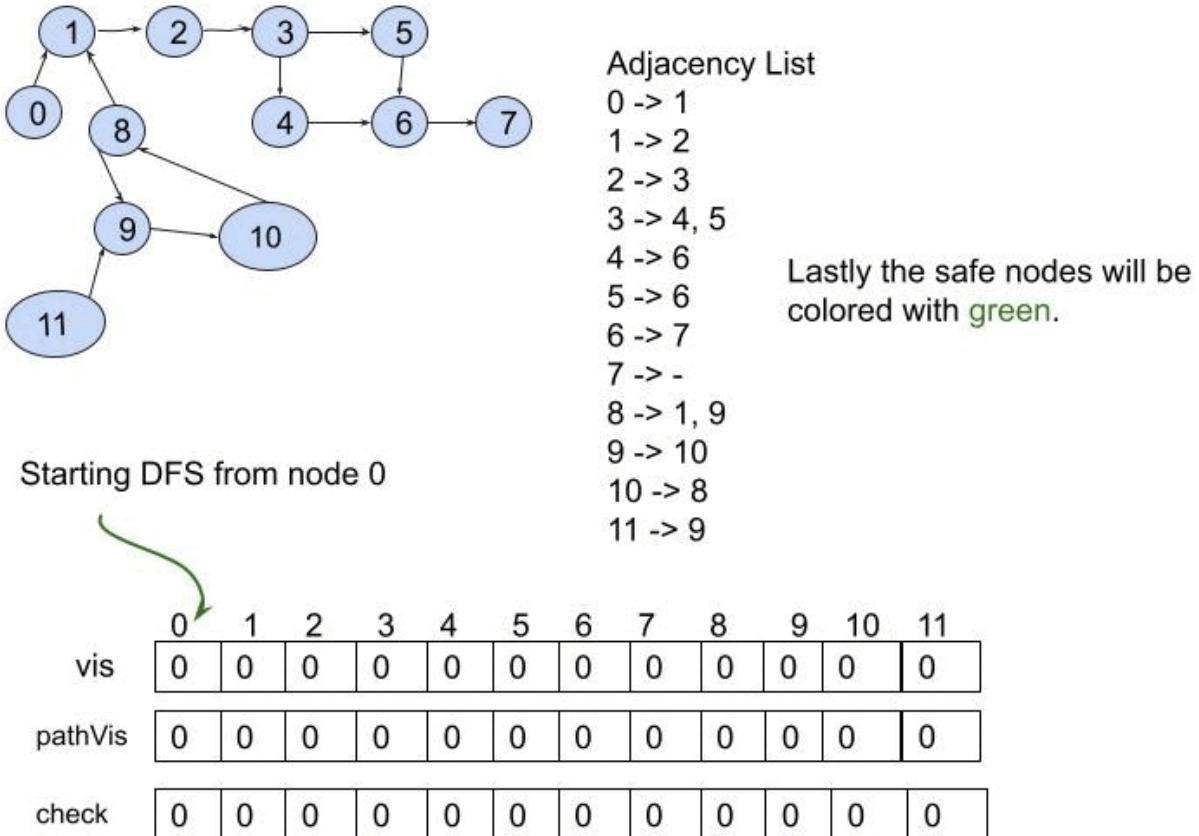
The algorithm steps are as follows:

- We must traverse all components of the graph.
- Make sure to carry two visited arrays in the DFS call. One is a visited array(vis) and the other is a path-visited(pathVis) array. The visited array keeps a track of visited nodes, and the path-visited keeps a track of visited nodes in the current traversal only.
- Along with that, we will be carrying ***an extra array(check) to mark the safe nodes***.

4. While making a DFS call, at first we will mark the node as visited in both the arrays and then will traverse through its adjacent nodes. Now, there may be either of the three cases:
 1. **Case 1:** If the adjacent node is not visited, we will ***make a new DFS call recursively*** with that particular node.
 2. **Case 2:** If the adjacent node is visited and also on the same path(i.e marked visited in the pathVis array), we will ***return true***, because it means it has a cycle, thereby the pathVis was true. Returning true will mean the end of the function call, as once we have got a cycle, there is no need to check for further adjacent nodes.
 3. **Case 3:** If the adjacent node is visited but not on the same path(i.e not marked in the pathVis array), we will continue to the next adjacent node, as it would have been marked as visited in some other path, and not on the current one.
5. Finally, if there are no further nodes to visit, we will mark the node as safe in the check array and unmark the current node in the pathVis array and just return false. Then we will backtrack to the previous node with the returned value.

The Point to remember is, while we enter we mark both the pathVis and vis as true, but at the end of traversal to all adjacent nodes, we just make sure to mark the current node safe and unmark the pathVis and still keep the vis marked as true, as it will avoid future extra traversal calls.

The following illustration will be useful in understanding the algorithm:



Let's briefly understand the algorithm, using the above illustration.

- First, the DFS will start from node 0 and mark all the nodes visited and path-visited following the path: 0->1->2->3->5->6->7.
- As there are no further nodes after 7 to visit, it will backtrack to node 3 and meanwhile, it will unmark nodes 7, 6, and 4 from the pathVis array and also mark them as safe nodes.
- Now, from node 3, it again follows the path: 3->5->6->7. But after visiting node 5, it will find node 6 as visited but not path-visited and so it will not move further. And while returning it will unmark node 5 in the pathVis array and mark it a safe node as well.
- From node 3 it will backtrack to node 0 and unmark the nodes 3, 2, 1, and 0 in the pathVis array and also mark them as safe nodes.
- Then, the DFS call will again start from node 8 following path 8->9->10. Now after reaching node 10, it finds node 8 previously visited as well as path-visited(i.e node 8 has been previously visited on the same path) and concludes that 8->9->10->8 is a cycle. So, nodes 8, 9, and 10 are not safe nodes.
- After that, the DFS call will start from node 11 which is incoming towards cycle 8->9->10->8. So it is not a safe node as well.
- Finally, the safe nodes will be 0, 1, 2, 3, 4, 5, 6, and 7 colored green in the illustration.

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

// User function Template for C++

class Solution {
private:
    bool dfsCheck(int node, vector<int> adj[], int vis[],
    int pathVis[], 
    int check[]) {
        vis[node] = 1;
        pathVis[node] = 1;
        check[node] = 0;
        // traverse for adjacent nodes
        for (auto it : adj[node]) {
            // when the node is not visited
            if (!vis[it]) {
                if (dfsCheck(it, adj, vis, pathVis, check) == true) {
                    check[node] = 0;
                    return true;
                }
            }
            // if the node has been previously visited
            // but it has to be visited on the same path
            else if (pathVis[it]) {
                check[node] = 0;
                return true;
            }
        }
        check[node] = 1;
        pathVis[node] = 0;
        return false;
    }
public:
    vector<int> eventualSafeNodes(int V, vector<int> adj[]) {
        int vis[V] = {0};
        int pathVis[V] = {0};
        int check[V] = {0};
        vector<int> safeNodes;
        for (int i = 0; i < V; i++) {
            if (!vis[i]) {
                dfsCheck(i, adj, vis, pathVis, check);
            }
        }
        for (int i = 0; i < V; i++) {
            if (check[i] == 1) safeNodes.push_back(i);
        }
        return safeNodes;
    }
};


```

```

int main() {

    //V = 12;
    vector<int> adj[12] = {{1}, {2}, {3}, {4, 5}, {6}, {7}, {}, {1, 9},
{10},
    {8},{9}};
    int V = 12;
    Solution obj;
    vector<int> safeNodes = obj.eventualSafeNodes(V, adj);

    for (auto node : safeNodes) {
        cout << node << " ";
    }
    cout << endl;

    return 0;
}

```

Output: 0 1 2 3 4 5 6 7

Time Complexity: $O(V+E)+O(V)$, where V = no. of nodes and E = no. of edges. There can be at most V components. So, another $O(V)$ time complexity.

Space Complexity: $O(3N) + O(N) \sim O(3N)$: $O(3N)$ for three arrays required during dfs calls and $O(N)$ for recursive stack space.

*Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, [please check out this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com*

Kahn's Algorithm | Topological Sort Algorithm | BFS: G-22

 takeuforward.org/data-structure/kahns-algorithm-topological-sort-algorithm-bfs-g-22

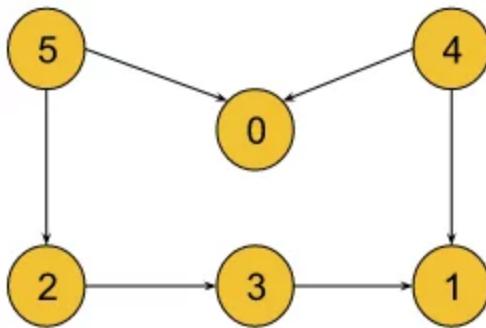
October 17, 2022

Problem Statement: Given a Directed Acyclic Graph (DAG) with V vertices and E edges, Find any Topological Sorting of that Graph.

Note: In topological sorting, node u will always appear before node v if there is a directed edge from node u towards node v ($u \rightarrow v$).

Example 1:

Input Format: V = 6, E = 6



Result: 5, 4, 0, 2, 3, 1

Explanation: A graph may have multiple topological sortings.

The result is one of them. The necessary conditions for the ordering are:

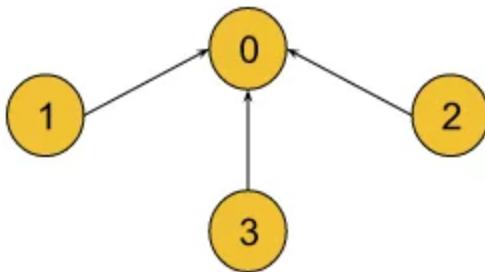
- According to edge $5 \rightarrow 0$, node 5 must appear before node 0 in the ordering.
- According to edge $4 \rightarrow 0$, node 4 must appear before node 0 in the ordering.
- According to edge $5 \rightarrow 2$, node 5 must appear before node 2 in the ordering.
- According to edge $2 \rightarrow 3$, node 2 must appear before node 3 in the ordering.
- According to edge $3 \rightarrow 1$, node 3 must appear before node 1 in the ordering.
- According to edge $4 \rightarrow 1$, node 4 must appear before node 1 in the ordering.

The above result satisfies all the necessary conditions.

[4, 5, 2, 3, 1, 0] and [4, 5, 0, 2, 3, 1] are also such topological sortings that satisfy all the conditions.

Example 2:

Input Format: V = 4, E = 3



Result: 1, 2, 3, 0

Explanation: The necessary conditions for the ordering are:

- For edge 1 → 0 node 1 must appear before node 0.
- For edge 2 → 0 node 1 must appear before node 0.
- For edge 3 → 0 node 1 must appear before node 0.

[2, 3, 1, 0] is also another topological sorting for the graph.

Solution

Disclaimer: Don't jump directly to the solution, try it out yourself first.

[Problem link](#).

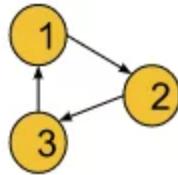
Topological sorting only exists in Directed Acyclic Graph (DAG). If the nodes of a graph are connected through directed edges and the graph does not contain a cycle, it is called a **directed acyclic graph(DAG)**.

The **topological sorting** of a directed acyclic graph is nothing but the linear ordering of vertices such that if there is an edge between node u and v($u \rightarrow v$), node u appears before v in that ordering.

Now, let's understand **Why topological sort only exists in DAG:**

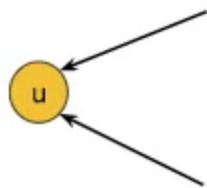
Case 1 (If the edges are undirected): If there is an undirected edge between node u and v, it signifies that there is an edge from node u to v($u \rightarrow v$) as well as there is an edge from node v to u($v \rightarrow u$). But according to the definition of topological sorting, it is practically impossible to write such ordering where **u appears before v** and **v appears before u** simultaneously. So, it is only possible for directed edges.

Case 2 (If the directed graph contains a cycle): The following directed graph contains a cycle:



If we try to get topological sorting of this cyclic graph, for edge $1 \rightarrow 2$, node 1 must appear before 2, for edge $2 \rightarrow 3$, node 2 must appear before 3, and for edge $3 \rightarrow 1$, node 3 must appear before 1 in the linear ordering. But such ordering is not possible as there exists a cyclic dependency in the graph. Thereby, topological sorting is only possible for a directed acyclic graph.

Therefore, the intuition is to find the ordering in which the nodes are connected in a directed acyclic graph. For this, we will use a slightly modified version of BFS where we will be requiring a queue data structure(First In First Out data structure) and an array that will store the **indegree** of each node. **The indegree** of a node is the number of directed edges incoming towards it.



For example, the indegree of node u is 2, as there are 2 incoming edges towards node u .

Approach:

Previously, we solved this question using the [DFS traversal technique](#). But in this article, we will apply the [BFS\(Breadth First Search\)](#) traversal technique. Breadth First Search or BFS is a traversal technique where we visit the nodes level-wise, i.e., it visits the same level nodes simultaneously, and then moves to the next level.

Initial Configuration:

Indegree Array: Initially all elements are set to 0. Then, We will count the incoming edges for a node and store it in this array. For example, if indegree of node 3 is 2, $\text{indegree}[3] = 2$.

Queue: As we will use BFS, a queue is required. Initially, the node with indegree 0 will be pushed into the queue.

Answer array: Initially empty and is used to store the linear ordering.

The algorithm steps are as follows:

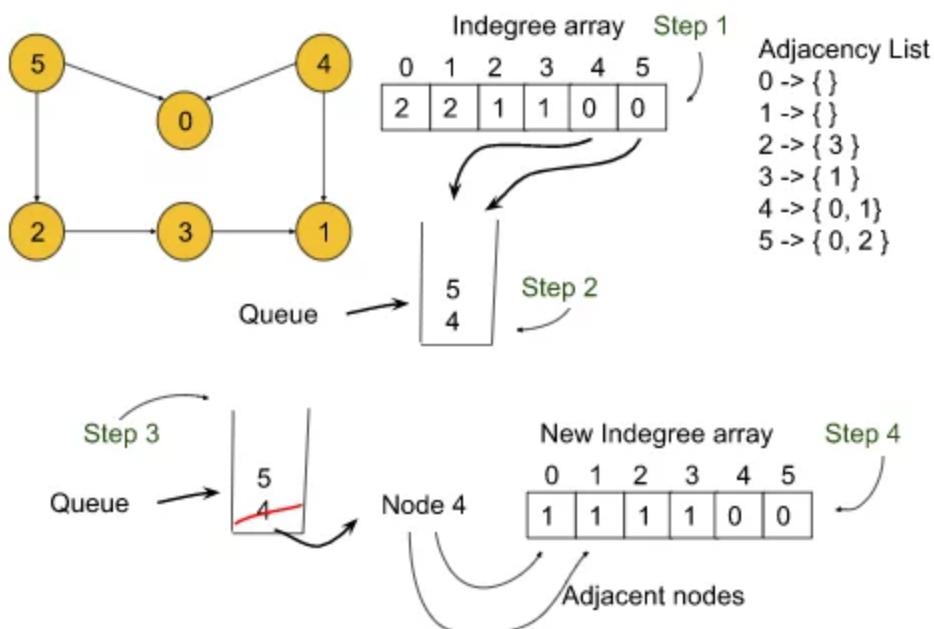
1. First, we will calculate the indegree of each node and store it in the indegree array. We can iterate through the given adj list, and simply for every node $u \rightarrow v$, we can increase the indegree of v by 1 in the indegree array.
2. Initially, there will be always at least a single node whose indegree is 0. So, we will push the node(s) with indegree 0 into the queue.
3. Then, we will pop a node from the queue including the node in our answer array, and for all its adjacent nodes, we will decrease the indegree of that node by one. For example, if node u that has been popped out from the queue has an edge towards node $v(u \rightarrow v)$, we will decrease $\text{indegree}[v]$ by 1.
4. After that, if for any node the indegree becomes 0, we will push that node again into the queue.
5. We will repeat steps 3 and 4 until the queue is completely empty. Finally, completing the BFS we will get the linear ordering of the nodes in the answer array.

Let's understand **how to find the indegree(s)**:

By visiting the adjacency list, we can find out the indegrees for each node. For example, if node 3 is an adjacent node of node 2, we will just increase $\text{indegree}[3]$ by 1 as the adjacency list suggests that node 3 has an incoming edge from node 2.

Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Let's quickly understand the algorithm using the below graph:



- First, we will calculate the indegrees for all 6 nodes. For node 0, it will be 2, for node 1, it will also be 2 and similarly, we will calculate for other nodes. The indegree array will look like this: {2, 2, 1, 1, 0, 0}.
- Next, the queue will be pushed with nodes 4 and 5 as their indegrees are 0. Then we will start the BFS.
- First, **node 4** will be popped out and kept in the answer array, and for all its adjacent nodes 0 and 1, the indegrees will be decreased by 1(**indegree array: {1, 1, 1, 1, 0, 0}**). No nodes will be pushed into the queue, as there are no other nodes with indegree 0.
- Now, similarly, **node 5** will be popped out and indegree[0] and indegree[2] will decrease by 1 keeping node 5 in the answer array. Now, we will push nodes 0 and 2 into the queue as their indegree has become 0(**indegree array: {0, 1, 0, 1, 0, 0}**).
- Then **node 0** will be popped out and as node 0 has no adjacent nodes it will be simply kept in the answer array.
- Next, **node 2** will be popped out and kept in the answer array while decreasing indegree[3] by 1(**indegree array: {0, 1, 0, 0, 0, 0}**). Now indegree[3] is 0 and so node 3 is pushed into the queue.
- Next, **node 3** is popped out and kept in the answer array and indegree[1] is decreased by 1 as node 1 is the adjacent node of node 3(**indegree array: {0, 0, 0, 0, 0, 0}**). Now indegree[1] is 0 and so node 1 is pushed into the queue.
- Lastly, **node 1** will be popped out and kept in the answer array, as node 1 has no adjacent nodes. Now, the BFS is completed for the graph.
- Finally, the answer array will look like: **{4, 5, 0, 2, 3, 1}**.

Note: Points to remember when a node is popped out, indegrees for all its adjacent nodes are decreased by one and if any of them becomes 0, we push that node into the queue. Meanwhile, we include the current node in the answer immediately after it is popped out of the queue.

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
    //Function to return list containing vertices in Topological order.
    vector<int> topoSort(int V, vector<int> adj[])
    {
        int indegree[V] = {0};
        for (int i = 0; i < V; i++) {
            for (auto it : adj[i]) {
                indegree[it]++;
            }
        }

        queue<int> q;
        for (int i = 0; i < V; i++) {
            if (indegree[i] == 0) {
                q.push(i);
            }
        }
        vector<int> topo;
        while (!q.empty()) {
            int node = q.front();
            q.pop();
            topo.push_back(node);
            // node is in your topo sort
            // so please remove it from the indegree

            for (auto it : adj[node]) {
                indegree[it]--;
                if (indegree[it] == 0) q.push(it);
            }
        }

        return topo;
    }
};

int main() {

    //V = 6;
    vector<int> adj[6] = {{}, {}, {3}, {1}, {0, 1}, {0, 2}};
    int V = 6;
    Solution obj;
    vector<int> ans = obj.topoSort(V, adj);

    for (auto node : ans) {
        cout << node << " ";
    }
    cout << endl;
}

```

```
    return 0;  
}
```

Output: 4 5 0 2 3 1

Time Complexity: $O(V+E)$, where V = no. of nodes and E = no. of edges. This is a simple BFS algorithm.

Space Complexity: $O(N) + O(N) \sim O(2N)$, $O(N)$ for the indegree array, and $O(N)$ for the queue data structure used in BFS(where N = no.of nodes).

*Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, [please check out this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com*



Watch Video At: <https://youtu.be/73sneFXuTEg>

Detect a Cycle in Directed Graph | Topological Sort | Kahn's Algorithm | G-23

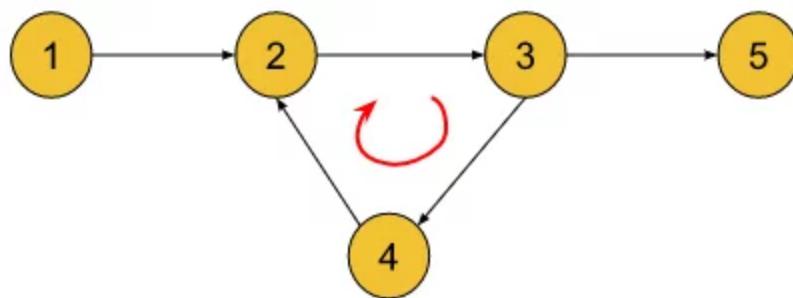
 takeuforward.org/data-structure/detect-a-cycle-in-directed-graph-topological-sort-kahns-algorithm-g-23

October 21, 2022

Problem Statement: Given a Directed Graph with V vertices and E edges, check whether it contains any cycle or not.

Example 1:

Input Format: $V = 5, E = 5$

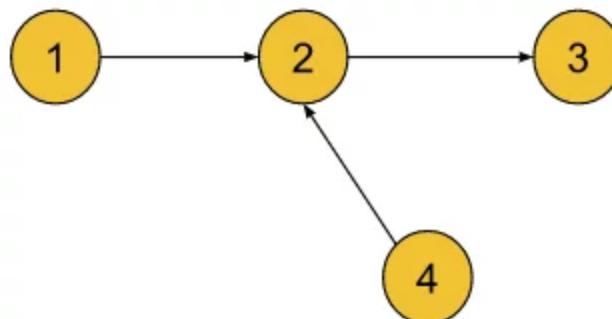


Result: True

Explanation: $2 \rightarrow 3 \rightarrow 4 \rightarrow 2$ is a cycle.

Example 2:

Input Format: $V = 4, E = 3$



Result: False

Explanation: There is no cycle in the graph.
This is a directed acyclic graph.

Disclaimer: Don't jump directly to the solution, try it out yourself first.

[Problem link](#).

Solution:

Previously, we learned how to detect cycles in a directed graph using the DFS traversal technique. Now, we will use the BFS traversal for the same purpose.

While using DFS, the algorithm is designed with two visited arrays(vis and pathVis) in such a way that after making a DFS call the node is marked in both arrays, and while backtracking the node is unmarked from the path-visited (pathVis) array. And when we find a node marked in both arrays, we conclude that there exists a cycle. A point to note here is that we involve backtracking in the logic of DFS while finding the cycle, which is tough to replicate if we are trying to solve it iteratively.

Now, we intend to design such an algorithm using BFS which can check the cycle. To fulfill that purpose, we are going to use Kahn's Algorithm(Topological Sorting Using BFS).

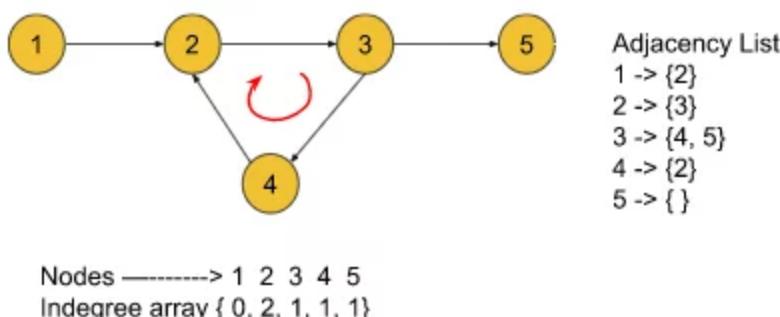
Intuition:

- Since we know topological sorting is only possible for **directed acyclic graphs(DAGs)** if we apply Kahn's algorithm in a directed cyclic graph(A directed graph that contains a cycle), it will fail to find the topological sorting(i.e. The final sorting will not contain all the nodes or vertices).
- So, finally, we will check the sorting to see if it contains all V vertices or not. If the result does not include all V vertices, we can conclude that there is a cycle.

Note: The intuition is to check the size of the final topological sorting if it equals V(no. of vertices or nodes) or not.

Let's quickly understand **why Kahn's Algorithm will fail for a directed cyclic graph**.

If a directed graph contains a cycle, the indegree of the nodes that are parts of that cycle will never be 0 due to the cyclic dependency. But in that algorithm, we push a node into the queue only if its in-degree becomes 0. So, those nodes of the cycle will never be pushed into the queue as well as included in the topological sorting. And here it violates the rules of topological sorting as topological sorting is a linear ordering of all V vertices (i.e. All V vertices must be present in that ordering). Consider the below graph:



- First, we will put node 1 into the queue as its in-degree is 0.
- Now, after popping node 1 out of the queue, we will reduce the indegree[2] by 1. But in this step, there is no other node with in-degree 0. So, the algorithm is over. But the topological sorting only contains node 1.

Approach:

We will apply the BFS(Breadth First Search) traversal technique. Breadth First Search or BFS is a traversal technique where we visit the nodes level-wise, i.e., it visits the same level nodes simultaneously, and then moves to the next level.

Initial Configuration:

Indegree Array: Initially all elements are set to 0. Then, We will count the incoming edges for a node and store it in this array. For example, if indegree of node 3 is 2, $\text{indegree}[3] = 2$.

Queue: As we will use BFS, a queue is required. Initially, the node with indegree 0 will be pushed into the queue.

Answer array(Optional): Initially empty and is used to store the linear ordering.

The algorithm steps are as follows:

1. First, we will calculate the in-degree of each node and store it in the in-degree array. We can iterate through the given adj list, and simply for every node $u \rightarrow v$, we can increase the in-degree of v by 1 in the in-degree array.
2. Initially, there will be always at least a single node whose indegree is 0. So, we will push the node(s) with in-degree 0 into the queue.
3. Then, we will pop a node from the queue including the node in our answer array, and for all its adjacent nodes, we will decrease the in-degree of that node by one. For example, if node u that has been popped out from the queue has an edge towards node $v (u \rightarrow v)$, we will decrease $\text{indegree}[v]$ by 1.
4. After that, if for any node the in-degree becomes 0, we will push that node again into the queue.
5. We will repeat steps 3 and 4 until the queue is completely empty. Now, completing the BFS we will get the linear ordering of the nodes in the answer array.
6. Finally, we will check the length of the answer array. If it equals V (no. of nodes) then the algorithm will return false otherwise it will return true.

Space Optimization:

In this particular algorithm, we are only concerned about the length of the topological sorting and not the exact nodes it contains. So in step 3, after popping a node out of the queue, **instead of putting it into an array we can carry a counter variable and**

increment it. After completing the BFS this counter variable will give the length of the topological sorting. So, we need not use the extra answer.

Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
    // Function to detect cycle in a directed graph.
    bool isCyclic(int V, vector<int> adj[]) {
        int indegree[V] = {0};
        for (int i = 0; i < V; i++) {
            for (auto it : adj[i]) {
                indegree[it]++;
            }
        }

        queue<int> q;
        for (int i = 0; i < V; i++) {
            if (indegree[i] == 0) {
                q.push(i);
            }
        }

        int cnt = 0;
        // O(v + e)
        while (!q.empty()) {
            int node = q.front();
            q.pop();
            cnt++;
            // node is in your topo sort
            // so please remove it from the indegree

            for (auto it : adj[node]) {
                indegree[it]--;
                if (indegree[it] == 0) q.push(it);
            }
        }

        if (cnt == V) return false;
        return true;
    }
};

int main() {

    //V = 6;
    vector<int> adj[6] = {{}, {2}, {3}, {4, 5}, {2}, {}};
    int V = 6;
    Solution obj;
    bool ans = obj.isCyclic(V, adj);
    if (ans) cout << "True";
    else cout << "False";
    cout << endl;
}

```

```
    return 0;  
}
```

Output: True

Time Complexity: $O(V+E)$, where V = no. of nodes and E = no. of edges. This is a simple BFS algorithm.

Space Complexity: $O(N) + O(N) \sim O(2N)$, $O(N)$ for the in-degree array, and $O(N)$ for the queue data structure used in BFS(where N = no.of nodes).

*Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, [please check out this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com*



Watch Video At: <https://youtu.be/iTBal90lpDQ>

Course Schedule I and II | Pre-requisite Tasks | Topological Sort: G-24

 takeuforward.org/data-structure/course-schedule-i-and-ii-pre-requisite-tasks-topological-sort-g-24

November 9, 2022



Problem Statement I (Course Schedule): There are a total of n tasks you have to pick, labeled from 0 to $n-1$. Some tasks may have prerequisites tasks, for example, to pick task 0 you have to first finish tasks 1, which is expressed as a pair: $[0, 1]$

Given the total number of n tasks and a list of prerequisite pairs of size m . Find the order of tasks you should pick to finish all tasks.

Note: There may be multiple correct orders, you need to return one of them. If it is impossible to finish all tasks, return an empty array.

Problem Statement II (Pre-requisite Tasks): There are a total of N tasks, labeled from 0 to $N-1$. Some tasks may have prerequisites, for example, to do task 0 you have to first complete task 1, which is expressed as a pair: $[0, 1]$

Given the total number of tasks N and a list of prerequisite pairs P , find if it is possible to finish all tasks.

Note: These two questions are linked. The second question asks if it is possible to finish all the tasks and the first question states to return the ordering of the tasks if it is possible to perform all the tasks, otherwise return an empty array.

Examples:

Example 1:

Input: N = 4, P = 3, array[] = {{1, 0}, {2, 1}, {3, 2}}

Output: Yes

Explanation: It is possible to finish all the tasks in the order : 3 2 1 0.

First, we will finish task 3. Then we will finish task 2, task 1, and task 0.

Example 2:

Input: N = 4, P = 4, array[] = {{1, 2}, {4, 3}, {2, 4}, {4, 1}}

Output: No

Explanation: It is impossible to finish all the tasks. Let's analyze the pairs:

For pair {1, 2} -> we need to finish task 1 first and then task 2. (order : 1 2).

For pair{4, 3} -> we need to finish task 4 first and then task 3. (order: 4 3).

For pair {2, 4} -> we need to finish task 2 first and then task 4. (order: 1 2 4 3).

But for pair {4, 1} -> we need to finish task 4 first and then task 1 but this pair contradicts the previous pair. So, it is not possible to finish all the tasks.

Solution

Disclaimer: Don't jump directly to the solution, try it out yourself first.

[Problem link 1.](#)

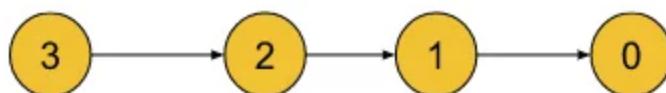
[Problem link 2.](#)

Solution:

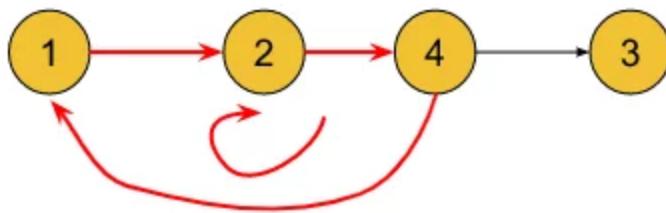
The solutions will be similar for both questions as we need to check for one, and in the other, we need to print the order. The questions state that the given pairs signify the dependencies of tasks. For example, the pair $\{u, v\}$ signifies that to perform task v, first we need to finish task u. Now, if we closely observe, we can think of a directed edge between u and v ($u \rightarrow v$) where u and v are two nodes. Now, if we can think of each task as a node and every pair as a directed edge between those two nodes, the whole problem becomes a **graph problem**.

Now, let's analyze the examples from the graph point of view:

For example 1, the number of tasks(considered as nodes) is 4, and pairs(considered as edges) are 3. If we draw the graph accordingly, the following illustration is produced:



And For example 2, the number of tasks(considered as nodes) is 4, and pairs(considered as edges) are 4. If we draw the graph accordingly, the following illustration is produced:



Analyzing the graphs, we can conclude that performing all the tasks from the first set is possible because the first graph does not contain any cycle but as the second graph contains a cycle, performing all the tasks from the second set is impossible(there exists a cyclic dependency between the tasks). So, first, we need to identify a graph as a **directed acyclic graph** and if it is so we need to find out the linear ordering of the nodes as well(*second part for the question: Course schedule*).

Now, we have successfully reduced the problem to one of our known concepts: **Detect cycle in a directed graph**. We will solve this problem using the **Topological Sort Algorithm or Kahn's Algorithm**.

Topological sorting only exists in Directed Acyclic Graph (DAG). If the nodes of a graph are connected through directed edges and the graph does not contain a cycle, it is called a **directed acyclic graph(DAG)**.

The **topological sorting** of a directed acyclic graph is nothing but the linear ordering of vertices such that if there is an edge between node u and v($u \rightarrow v$), node u appears before v in that ordering.

For the second problem, we can also apply the algorithm used in the detection of cycles in a directed graph (using DFS) where we used to find out if the graph has a cycle or not. But to solve the first question we have to figure out the linear ordering of the task as well. So, we will use the topological sort algorithm for both questions.

Intuition:

For problem I, the intuition is to find the linear ordering in which the tasks will be performed if it is possible to perform all the tasks otherwise, to return an empty array.

For problem II, the intuition is to find if it is possible to perform all the tasks(i.e. The graph contains a cycle or not).

Approach:

We will apply the BFS(Breadth First Search) traversal technique. Breadth First Search or BFS is a traversal technique where we visit the nodes level-wise, i.e., it visits the same level nodes simultaneously, and then moves to the next level.

Initial Configuration:

Indegree Array: Initially all elements are set to 0. Then, We will count the incoming edges for a node and store it in this array. For example, if the indegree of node 3 is 2, $\text{indegree}[3] = 2$.

Queue: As we will use BFS, a queue is required. Initially, the node with indegree 0 will be pushed into the queue.

Answer array: Initially empty and is used to store the linear ordering.

The algorithm steps are as follows:

1. First, we will form the adjacency list of the graph using the pairs. For example, for the pair $\{u, v\}$ we will add node v as an adjacent node of u in the list.
2. Then, we will calculate the in-degree of each node and store it in the indegree array. We can iterate through the given adj list, and simply for every node $u \rightarrow v$, we can increase the indegree of v by 1 in the indegree array.
3. Initially, there will be always at least a single node whose indegree is 0. So, we will push the node(s) with indegree 0 into the queue.
4. Then, we will pop a node from the queue including the node in our answer array, and for all its adjacent nodes, we will decrease the indegree of that node by one. For example, if node u that has been popped out from the queue has an edge towards node v ($u \rightarrow v$), we will decrease $\text{indegree}[v]$ by 1.
5. After that, if for any node the indegree becomes 0, we will push that node again into the queue.
6. We will repeat steps 3 and 4 until the queue is completely empty. Now, completing the BFS we will get the linear ordering of the nodes in the answer array.
7. **For the first problem(Course Schedule):** We will return the answer array if the length of the ordering equals the number of tasks. Otherwise, we will return an empty array.
For the Second problem(Prerequisite tasks): We will return true if the length of the ordering equals the number of tasks. otherwise, we will return false.

Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Code (Course Schedule):

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution
{
public:
    vector<int> findOrder(int V, int m, vector<vector<int>> prerequisites)
    {
        vector<int> adj[V];
        for (auto it : prerequisites) {
            adj[it[1]].push_back(it[0]);
        }

        int indegree[V] = {0};
        for (int i = 0; i < V; i++) {
            for (auto it : adj[i]) {
                indegree[it]++;
            }
        }

        queue<int> q;
        for (int i = 0; i < V; i++) {
            if (indegree[i] == 0) {
                q.push(i);
            }
        }

        vector<int> topo;
        while (!q.empty()) {
            int node = q.front();
            q.pop();
            topo.push_back(node);
            // node is in your topo sort
            // so please remove it from the indegree

            for (auto it : adj[node]) {
                indegree[it]--;
                if (indegree[it] == 0) q.push(it);
            }
        }

        if (topo.size() == V) return topo;
        return {};
    }
};

int main() {
    int N = 4;

```

```
int M = 3;

vector<vector<int>> prerequisites(3);
prerequisites[0].push_back(0);
prerequisites[0].push_back(1);

prerequisites[1].push_back(1);
prerequisites[1].push_back(2);

prerequisites[2].push_back(2);
prerequisites[2].push_back(3);

Solution obj;
vector<int> ans = obj.findOrder(N, M, prerequisites);

for (auto task : ans) {
    cout << task << " ";
}
cout << endl;
return 0;
}
```

Output: 3 2 1 0

Code (Pre-requisite Tasks):

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
    bool isPossible(int V, vector<pair<int, int> >& prerequisites) {
        vector<int> adj[V];
        for (auto it : prerequisites) {
            adj[it.first].push_back(it.second);
        }

        int indegree[V] = {0};
        for (int i = 0; i < V; i++) {
            for (auto it : adj[i]) {
                indegree[it]++;
            }
        }

        queue<int> q;
        for (int i = 0; i < V; i++) {
            if (indegree[i] == 0) {
                q.push(i);
            }
        }
        vector<int> topo;
        while (!q.empty()) {
            int node = q.front();
            q.pop();
            topo.push_back(node);
            // node is in your topo sort
            // so please remove it from the indegree

            for (auto it : adj[node]) {
                indegree[it]--;
                if (indegree[it] == 0) q.push(it);
            }
        }

        if (topo.size() == V) return true;
        return false;
    }
};

int main() {
    vector<pair<int, int>> prerequisites;
    int N = 4;
    prerequisites.push_back({1, 0});

```

```
prerequisites.push_back({2, 1});
prerequisites.push_back({3, 2});

Solution obj;
bool ans = obj.isPossible(N, prerequisites);

if (ans) cout << "YES";
else cout << "NO";
cout << endl;

return 0;
}
```

Output: YES

Time Complexity: $O(V+E)$, where V = no. of nodes and E = no. of edges. This is a simple BFS algorithm.

Space Complexity: $O(N) + O(N) \sim O(2N)$, $O(N)$ for the indegree array, and $O(N)$ for the queue data structure used in BFS (where N = no. of nodes). Extra $O(N)$ for storing the topological sorting. Total $\sim O(3N)$.

*Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out [this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com*

Find Eventual Safe States – BFS – Topological Sort: G-25

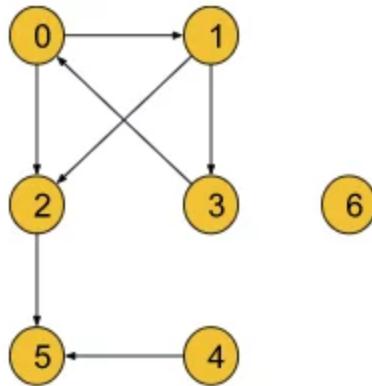
 takeuforward.org/data-structure/find-eventual-safe-states-bfs-topological-sort-g-25

November 9, 2022

Problem Statement: A directed graph of V vertices and E edges is given in the form of an adjacency list adj . Each node of the graph is labeled with a distinct integer in the range 0 to $V - 1$. A node is a terminal node if there are no outgoing edges. A node is a safe node if every possible path starting from that node leads to a terminal node. You have to return an array containing all the safe nodes of the graph. The answer should be sorted in ascending order.

Example 1:

Input Format: $N = 7, E = 7$



Result: {2 4 5 6}

Explanation: Here terminal nodes are 5 and 6 as they have no outgoing edges.

From node 0: two paths are there $0 \rightarrow 2 \rightarrow 5$ and $0 \rightarrow 1 \rightarrow 3 \rightarrow 0$.
The second path does not end at a terminal node.
So it is not a safe node.

From node 1: two paths exist: $1 \rightarrow 3 \rightarrow 0 \rightarrow 1$ and $1 \rightarrow 2 \rightarrow 5$.
But the first one does not end at a terminal node.
So it is not a safe node.

From node 2: only one path: $2 \rightarrow 5$ and 5 is a terminal node. So it is a **safe node**.

From node 3: two paths: $3 \rightarrow 0 \rightarrow 1 \rightarrow 3$ and $3 \rightarrow 0 \rightarrow 2 \rightarrow 5$ but the first path does not end at a terminal node.
So it is not a **safe node**.

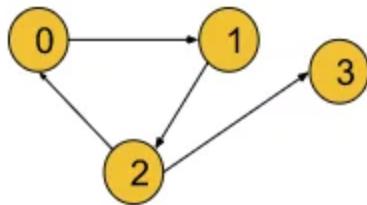
From node 4: Only one path: $4 \rightarrow 5$ and 5 is a terminal node.
So it is also a **safe node**.

From node 5: It is a terminal node. So it is a **safe node** as well.

From node 6: It is a terminal node.
So it is a **safe node** as well.

Example 2:

Input Format: N = 4, E = 4



Result: {3}

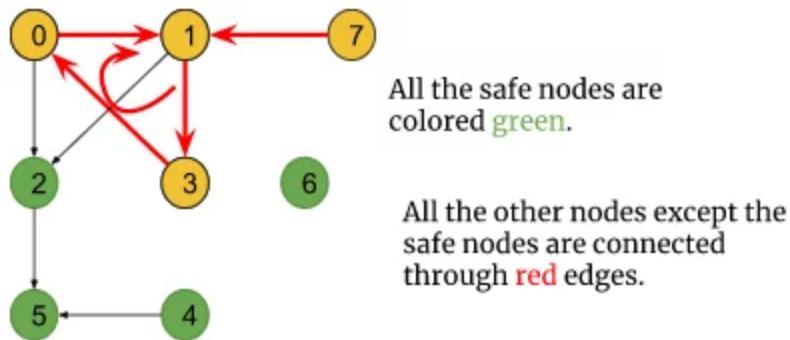
Explanation: Node 3 itself is a terminal node and it is a **safe node** as well. But all the paths from other nodes do not lead to a terminal node. So they are excluded from the answer.

Disclaimer: Don't jump directly to the solution, try it out yourself first.

[Problem link](#).

Solution:

A **terminal node** is a node without any outgoing edges(i.e outdegree = 0). Now a node is considered to be a **safe node** if all possible paths starting from it lead to a terminal node. Here we need to find out all safe nodes and return them sorted in ascending order. If we closely observe, all possible paths starting from a node are going to end at some terminal node unless there exists a cycle and the paths return back to themselves. Let's understand it considering the below graph:



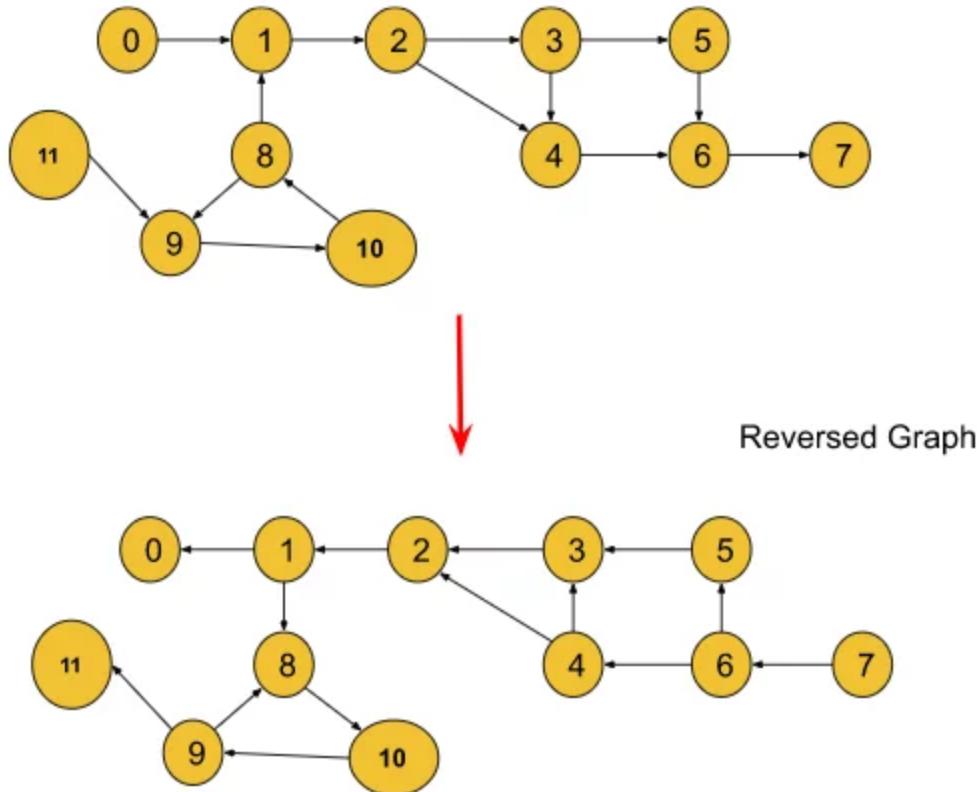
- In the above graph, there exists a cycle i.e $0 \rightarrow 1 \rightarrow 3 \rightarrow 0$, and node 7 is connected to the cycle with an incoming edge towards the cycle.
- Some paths starting from these nodes are definitely going to end somewhere in the cycle and not at any terminal node. So, these nodes are not safe nodes.

- Though node 2 is connected to the cycle, the edge is directed outwards the cycle and all the paths starting from it lead to the terminal node 5. So, it is a safe node and the rest of the nodes (4, 5, 6) are safe nodes as well.

So, the intuition is to figure out the nodes which are neither a part of a cycle nor connected to the cycle. We have previously solved this problem using the DFS traversal technique. Now, we are going to solve it using the BFS traversal technique especially using the topological sort algorithm. The topological sort algorithm will automatically exclude the nodes which are either forming a cycle or connected to a cycle. **Note:** *Points to remember that any node which is a part of a cycle or leads to the cycle through an incoming edge towards the cycle, cannot be a safe node. Apart from these types of nodes, every node is a safe node.*

Approach:

The node with outdegree 0 is considered to be a terminal node but the topological sort algorithm deals with the indegrees of the nodes. So, to use the topological sort algorithm, we will reverse every edge of the graph. Now, the nodes with indegree 0 become the terminal nodes. After this step, we will just follow the topological sort algorithm as it is.



We will apply the BFS(Breadth First Search) traversal technique. Breadth First Search or BFS is a traversal technique where we visit the nodes level-wise, i.e., it visits the same level nodes simultaneously, and then moves to the next level.

Initial Configuration:

Indegree Array: Initially all elements are set to 0. Then, We will count the incoming edges for a node and store it in this array. For example, if indegree of node 3 is 2, $\text{indegree}[3] = 2$. If you don't know how to find indegrees, you can refer to the step 2 in the algorithm.

Queue: As we will use BFS, a queue is required. Initially, the node with indegree 0 will be pushed into the queue.

safeNodes array: Initially empty and is used to store the safe nodes.

The algorithm steps are as follows:

1. First, we will reverse all the edges of the graph so that we can apply the topological sort algorithm afterward.
2. Then, we will calculate the indegree of each node and store it in the indegree array. We can iterate through the given adj list, and simply for every node $u \rightarrow v$, we can increase the indegree of v by 1 in the indegree array.
3. Then, we will push the node(s) with indegree 0 into the queue.
4. Then, we will pop a node from the queue including the node in our safeNodes array, and for all its adjacent nodes, we will decrease the indegree of that node by one. For example, if node u that has been popped out from the queue has an edge towards node $v (u \rightarrow v)$, we will decrease $\text{indegree}[v]$ by 1.
5. After that, if for any node the indegree becomes 0, we will push that node again into the queue.
6. We will repeat steps 3 and 4 until the queue is completely empty. Finally, completing the BFS we will get the linear ordering of the nodes in the safeNodes array.
7. Finally, the safeNodes array will only consist of the nodes that are neither a part of any cycle nor connected to any cycle. Then we will sort the final safeNodes array as the question requires the answer in sorted order.

Note: *If you wish to see the dry run of the above approach, you can watch the video attached to this article.*

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
    vector<int> eventualSafeNodes(int V, vector<int> adj[]) {
        vector<int> adjRev[V];
        int indegree[V] = {0};
        for (int i = 0; i < V; i++) {
            // i -> it
            // it -> i
            for (auto it : adj[i]) {
                adjRev[it].push_back(i);
                indegree[i]++;
            }
        }
        queue<int> q;
        vector<int> safeNodes;
        for (int i = 0; i < V; i++) {
            if (indegree[i] == 0) {
                q.push(i);
            }
        }
        while (!q.empty()) {
            int node = q.front();
            q.pop();
            safeNodes.push_back(node);
            for (auto it : adjRev[node]) {
                indegree[it]--;
                if (indegree[it] == 0) q.push(it);
            }
        }
        sort(safeNodes.begin(), safeNodes.end());
        return safeNodes;
    }
};

int main() {
    vector<int> adj[12] = {{1}, {2}, {3, 4}, {4, 5}, {6}, {6}, {7}, {}, {1, 9}, {10}, {8}, {9}};
    int V = 12;
    Solution obj;
    vector<int> safeNodes = obj.eventualSafeNodes(V, adj);

    for (auto node : safeNodes) {
        cout << node << " ";
    }
}

```

```
    }
    cout << endl;

    return 0;
}
```

Output : 0 1 2 3 4 5 6 7

Time Complexity: $O(V+E)+O(N*\log N)$, where V = no. of nodes and E = no. of edges. This is a simple BFS algorithm. Extra $O(N*\log N)$ for sorting the `safeNodes` array, where N is the number of safe nodes.

Space Complexity: $O(N) + O(N) + O(N) \sim O(3N)$, $O(N)$ for the indegree array, $O(N)$ for the queue data structure used in BFS(where N = no.of nodes), and extra $O(N)$ for the adjacency list to store the graph in a reversed order.

*Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, [please check out this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com*

Alien Dictionary – Topological Sort: G-26

Problem Statement: Given a sorted dictionary of an alien language having N words and k starting alphabets of a standard dictionary. Find the order of characters in the alien language.

Note: Many orders may be possible for a particular test case, thus you may return any valid order.

Examples:

Example 1:

Input: N = 5, K = 4

dict = {"baa", "abcd", "abca", "cab", "cad"}

Output: b d a c

Explanation:

We will analyze every consecutive pair to find out the order of the characters.

The pair “baa” and “abcd” suggests ‘b’ appears before ‘a’ in the alien dictionary.

The pair “abcd” and “abca” suggests ‘d’ appears before ‘a’ in the alien dictionary.

The pair “abca” and “cab” suggests ‘a’ appears before ‘c’ in the alien dictionary.

The pair “cab” and “cad” suggests ‘b’ appears before ‘d’ in the alien dictionary.

So, ['b', 'd', 'a', 'c'] is a valid ordering.

Example 2:

Input: N = 3, K = 3

dict = {"caa", "aaa", "aab"}

Output: c a b

Explanation: Similarly, if we analyze the consecutive pair for this example, we will figure out ['c', 'a', 'b'] is a valid ordering.

Disclaimer: Don't jump directly to the solution, try it out yourself first.

Problem Link.

Solution:

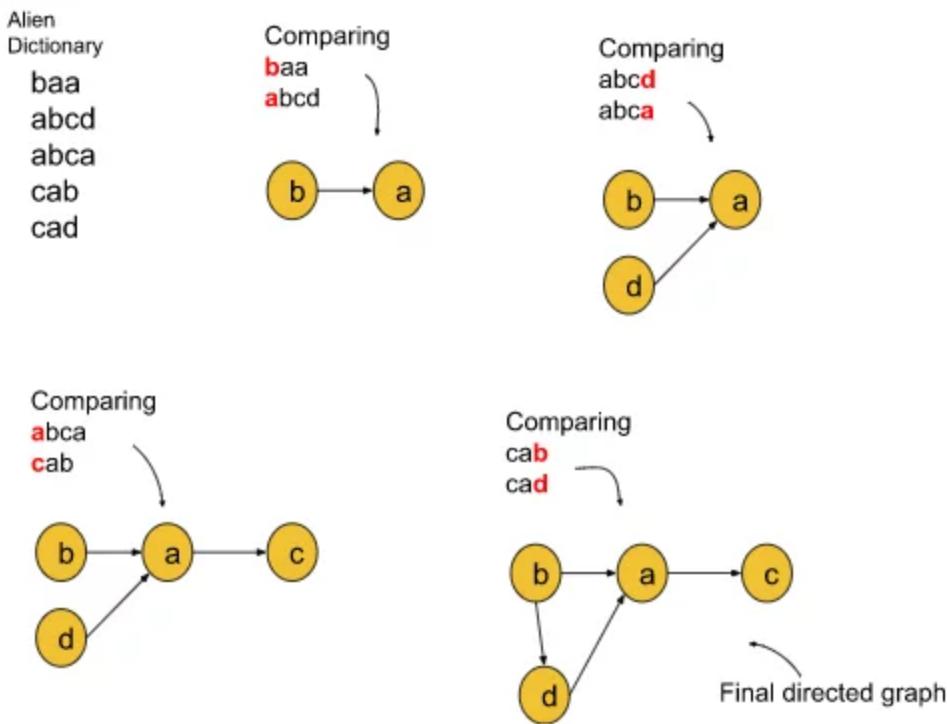
Let's consider the first example where **N** = 5, **K** = 4 and **dict** = {"baa", "abcd", "abca", "cab", "cad"}. So, here we need to find out the correct ordering of the first 4 letters of the alphabet(i.e. ‘a’, ‘b’, ‘c’, ‘d’). If we consider the first 2 words and try to figure out why “baa” appears before “abcd”, we can clearly observe that they are differentiated by the first letter i.e. ‘b’ and ‘a’. That is why, we can conclude that in the alien dictionary, **‘b’ appears before ‘a’(i.e. ‘b’ is smaller than ‘a’)**. We can correspond this differentiating factor to a **directed graph** like the following:



Let's understand *why we need not check “baa” and “abca” (the 1st and the 3rd word) next:*

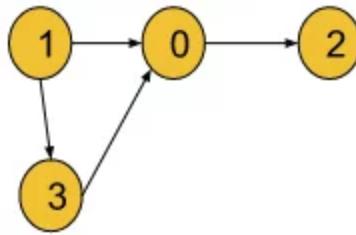
Until now, we have figured out why “baa” appears before “abcd”. So, by convention, if “abcd” is appearing before “abca” and “baa” is appearing before “abcd”, **“baa” will obviously appear before “abca”**. That is why we will check the pair of “abcd” and “abca” next rather than checking “baa” with any other words and this flow will be continued for the rest of the words.

Note: Points to remember that we need not check every pair of words rather we will just check the consecutive pair of words in the dictionary. Comparing each pair of consecutive words in the dictionary, we can construct a directed graph like the following:



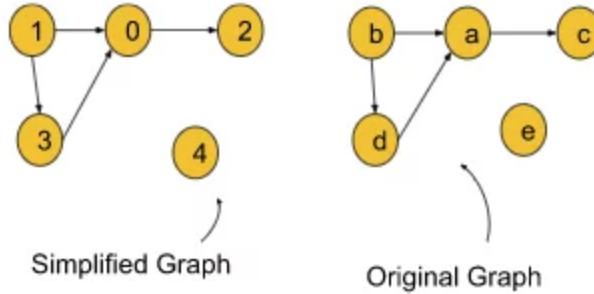
Now, we have successfully reduced the problem to a known **directed graph problem**. If we look at the problem from the graph point of view, we just need to find out the **linear ordering of the nodes of the directed graph**. And we can do this easily using the topological sort algorithm which we have previously learned.

To further simplify the problem, we will denote the alphabet with numbers like: ‘a’ with 0, ‘b’ with 1, ‘c’ with 2, and so on. For example, if the letter is ‘z’, we will denote it using 25. Finally, the directed graph will look like the following illustration:



Note: The intuition is to check every consecutive pair of words and find out the differentiating factor. With these factors, we will form a directed graph, and the whole problem boils down to a topological sort of problem.

Edge Case: The problem arises when the value of K becomes 5 and there is no word in the dictionary containing the letter ‘e’. In this case, we will add a separate node with the value ‘e’ in the graph and it will be considered a component of the directed graph like the following, and the same algorithm will work fine for multiple components.



Note: If the value of K is greater than the number of unique characters appearing in the dictionary, then the extra characters will be considered the different components of the directed graph formed.

The follow-up question for the interview:

When is the ordering not possible?

There are two such cases when ordering is not possible:

- **If every character matches and the largest word appears before the shortest word:** For example, if “abcd” appears before “abc”, we can say the ordering is not possible.
- **If there exists a cyclic dependency between the characters:** For example, in the dictionary: **dict: {“abc”, “bat”, “ade”}** there exists a cyclic dependency between ‘a’ and ‘b’ because the dictionary states ‘a’ < ‘b’ < ‘a’, which is not possible.

Approach:

We will apply the BFS(Breadth First Search) traversal technique. Breadth First Search or BFS is a traversal technique where we visit the nodes level-wise, i.e., it visits the same level nodes simultaneously, and then moves to the next level.

Initial Configuration:

Adjacency List: Initially it will be empty and we will create this adjacency list comparing the consecutive pair of words.

Indegree Array: Initially all elements are set to 0. Then, We will count the incoming edges for a node and store it in this array. For example, if the indegree of node 3 is 2, $\text{indegree}[3] = 2$.

Queue: As we will use BFS, a queue is required. Initially, the node with indegree 0 will be pushed into the queue.

Answer array(topo): Initially empty and is used to store the linear ordering.

The algorithm steps are as follows:

1. First, we need to create the adjacency list for the graph. The steps are the following:
 1. We will run a loop from the starting index to the **second last index** because we will check the i^{th} element and the $(i+1)^{\text{th}}$ element.
 2. Inside the loop, we will pick two words (the word at the current index(**s1**) and the word at the next index(**s2**)). For comparing them, we will again run a loop up to the length of the smallest string.
 3. Inside that second loop, if at any index we find inequality (**s1[i] != s2[i]**), we will add them to the adjacency list (**s1[i] —> s2[i]**) in terms of numbers(subtracting 'a' from them), and then we will immediately come out of the loop.
 4. This is only because we want the first differentiating character. Finally, we will get the adjacency list.
 5. In short, we need to find the differentiating character for adjacent strings and create the graph.
2. Once the graph is created, simply perform a topo sort, whose steps are given below.
3. Then, we will calculate the indegree of each node and store it in the indegree array. We can iterate through the given adj list, and simply for every node $u \rightarrow v$, we can increase the indegree of v by 1 in the indegree array.
4. Initially, there will be always at least a single node whose indegree is 0. So, we will push the node(s) with indegree 0 into the queue.
5. Then, we will pop a node from the queue including the node in our answer array, and for all its adjacent nodes, we will decrease the indegree of that node by one. For example, if node u that has been popped out from the queue has an edge towards node $v(u \rightarrow v)$, we will decrease $\text{indegree}[v]$ by 1.

6. After that, if for any node the indegree becomes 0, we will push that node again into the queue.
7. We will repeat steps 3 and 4 until the queue is completely empty. Finally, completing the BFS we will get the linear ordering of the nodes in the answer array.
8. For the final answer, we will iterate on the answer array and add each element in terms of character(addng ‘a’ to each of them) to the final string. Then we will return the string as our final answer.

Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
    // works for multiple components
private:
    vector<int> topoSort(int V, vector<int> adj[])
    {
        int indegree[V] = {0};
        for (int i = 0; i < V; i++) {
            for (auto it : adj[i]) {
                indegree[it]++;
            }
        }

        queue<int> q;
        for (int i = 0; i < V; i++) {
            if (indegree[i] == 0) {
                q.push(i);
            }
        }
        vector<int> topo;
        while (!q.empty()) {
            int node = q.front();
            q.pop();
            topo.push_back(node);
            // node is in your topo sort
            // so please remove it from the indegree

            for (auto it : adj[node]) {
                indegree[it]--;
                if (indegree[it] == 0) q.push(it);
            }
        }

        return topo;
    }
public:
    string findOrder(string dict[], int N, int K) {
        vector<int>adj[K];
        for (int i = 0; i < N - 1; i++) {
            string s1 = dict[i];
            string s2 = dict[i + 1];
            int len = min(s1.size(), s2.size());
            for (int ptr = 0; ptr < len; ptr++) {
                if (s1[ptr] != s2[ptr]) {
                    adj[s1[ptr] - 'a'].push_back(s2[ptr] - 'a');
                    break;
                }
            }
        }
    }
}

```

```

    }

    vector<int> topo = topoSort(K, adj);
    string ans = "";
    for (auto it : topo) {
        ans = ans + char(it + 'a');
    }
    return ans;
}

};

int main() {

    int N = 5, K = 4;
    string dict[] = {"baa", "abcd", "abca", "cab", "cad"};
    Solution obj;
    string ans = obj.findOrder(dict, N, K);

    for (auto ch : ans)
        cout << ch << ' ';
    cout << endl;

    return 0;
}

```

Output: b d a c

Time Complexity: $O(N*len)+O(K+E)$, where N is the number of words in the dictionary, 'len' is the length up to the index where the first inequality occurs, K = no. of nodes, and E = no. of edges.

Space Complexity: $O(K) + O(K)+O(K)+O(K) \sim O(4K)$, O(K) for the indegree array, and O(K) for the queue data structure used in BFS(where K = no.of nodes), O(K) for the answer array and O(K) for the adjacency list used in the algorithm.

*Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, [please check out this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com*

Shortest Path in Directed Acyclic Graph Topological Sort: G-27

 takeuforward.org/data-structure/shortest-path-in-directed-acyclic-graph-topological-sort-g-27

October 14, 2022

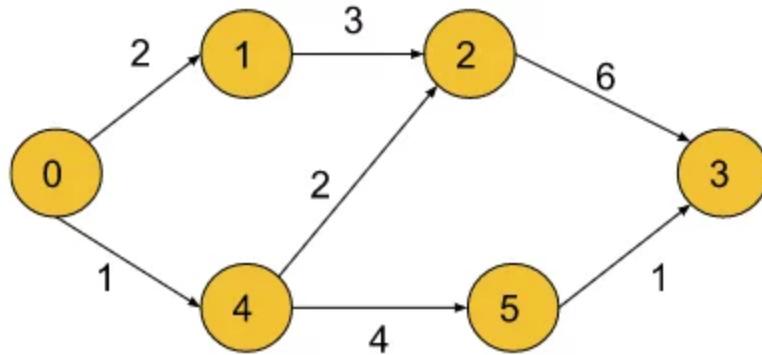
Given a DAG, find the shortest path from the source to all other nodes in this DAG. In this problem statement, we have assumed the source vertex to be '0'. You will be given the weighted edges of the graph.

Note: What is a DAG (Directed Acyclic Graph)?

A Directed Graph (containing one-sided edges) having no cycles is said to be a Directed Acyclic Graph.

Examples:

Example 1:



Input: $n = 6$, $m = 7$

edges = [[0, 1, 2], [0, 4, 1], [4, 5, 4], [4, 2, 2], [1, 2, 3], [2, 3, 6], [5, 3, 1]]

Output: 0 2 3 6 1 5

Explanation: The above output list shows the **shortest path** to all the nodes from the source vertex (0),

Dist[0] = 0

Dist[1] = 2

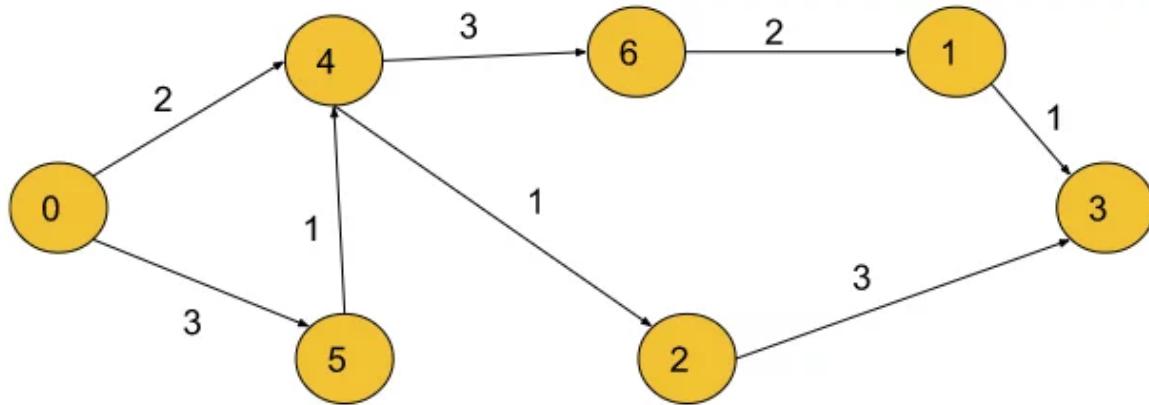
Dist[2] = 3

Dist[3] = 6

Dist[4] = 1

Dist[5] = 5

Example 2:



Input: n = 7, m= 8

Edges =[[0,4,2],[0,5,3],[5,4,1],[4,6,3],[4,2,1],[6,1,2],[2,3,3],[1,3,1]]

Output: 0 7 3 6 2 3 5

Explanation:

The above output list shows the **shortest path** to all the nodes from the source vertex (0),

Dist[0] = 0

Dist[1] = 7

Dist[2] = 3

Dist[3] = 6

Dist[4] = 2

Dist[5] = 3

Dist[6] = 5

Solution

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Intuition:

Finding the shortest path to a vertex is easy if you already know the shortest paths to all the vertices that can precede it. Processing the vertices in topological order ensures that by the time you get to a vertex, you've already processed all the vertices that can precede it which reduces the computation time significantly. In this approach, we traverse the nodes sequentially according to their reachability from the source.

Dijkstra's algorithm is necessary for graphs that can contain cycles because they can't be topologically sorted. In other cases, the topological sort would work fine as we start from the first node, and then move on to the others in a directed manner.

Approach:

We will calculate the shortest path in a directed acyclic graph by using topological sort. Topological sort can be implemented in two ways- BFS and DFS. Here, we will be implementing using the DFS technique. Depth First Search, DFS is a traversal technique where we visit a node and then continue visiting its adjacent nodes until we reach the end point, i.e., it keeps on moving in the depth of a particular node and then backtracks when no further adjacent nodes are available.

Initial configuration:

- **Adjacency List:** Create an adjacency list of the form vector of pairs of size 'N', where each index denotes a node 'u' and contains a vector that consists of pairs denoting the adjacent nodes 'v' and the distance to that adjacent node from initial node 'u'.
- Visited Array: Create a visited array and mark all the indices as unvisited (0) initially.
- Stack: Define a stack data structure to store the topological sort.
- Distance Array: Initialise this array by Max integer value and then update the value for each node successively while calculating the shortest distance between the source and the current node.

The shortest path in a directed acyclic graph can be calculated by the following steps:

- Perform topological sort on the graph using BFS/DFS and store it in a stack. In order to get a hang of how the **Topological Sort** works, you can refer to [this article](#) for the same.

- Now, iterate on the topo sort. We can keep the generated topo sort in the stack only, and do an iteration on it, it reduces the extra space which would have been required to store it. Make sure for the source node, we will assign $\text{dist}[\text{src}] = 0$.
- For every node that comes out of the stack which contains our topo sort, we can traverse for all its adjacent nodes, and relax them.
- In order to relax them, we simply do a simple comparison of $\text{dist}[\text{node}] + \text{wt}$ and $\text{dist}[\text{adjNode}]$. Here $\text{dist}[\text{node}]$ means the distance taken to reach the current node, and it is the edge weight between the node and the adjNode .
- If ($\text{dist}[\text{node}] + \text{wt} < \text{dist}[\text{adjNode}]$), then we will go ahead and update the distance of the $\text{dist}[\text{adjNode}]$ to the new found better path.
- Once all the nodes have been iterated, the $\text{dist}[]$ array will store the shortest paths and we can then return it.

Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Code:

- C++ Code
- Java Code

```

#include<bits/stdc++.h>

using namespace std;

class Solution {
private:
    void topoSort(int node, vector < pair < int, int >> adj[],
        int vis[], stack < int > & st) {
        //This is the function to implement Topological sort.
        vis[node] = 1;
        for (auto it: adj[node]) {
            int v = it.first;
            if (!vis[v]) {
                topoSort(v, adj, vis, st);
            }
        }
        st.push(node);
    }
public:
    vector < int > shortestPath(int N, int M, vector < vector < int >> & edges) {

        //We create a graph first in the form of an adjacency list.
        vector < pair < int, int >> adj[N];
        for (int i = 0; i < M; i++) {
            int u = edges[i][0];
            int v = edges[i][1];
            int wt = edges[i][2];
            adj[u].push_back({v, wt});
        }
        // A visited array is created with initially
        // all the nodes marked as unvisited (0).
        int vis[N] = {
            0
        };
        //Now, we perform topo sort using DFS technique
        //and store the result in the stack st.
        stack < int > st;
        for (int i = 0; i < N; i++) {
            if (!vis[i]) {
                topoSort(i, adj, vis, st);
            }
        }
        //Further, we declare a vector 'dist' in which we update the value of the
        //nodes'
        //distance from the source vertex after relaxation of a particular node.

        vector < int > dist(N);
        for (int i = 0; i < N; i++) {
            dist[i] = 1e9;
        }

        dist[0] = 0;
    }
}

```

```

while (!st.empty()) {
    int node = st.top();
    st.pop();

    for (auto it: adj[node]) {
        int v = it.first;
        int wt = it.second;

        if (dist[node] + wt < dist[v]) {
            dist[v] = wt + dist[node];
        }
    }
}

for (int i = 0; i < N; i++) {
    if (dist[i] == 1e9) dist[i] = -1;
}
return dist;
}
};

int main() {

int N = 6, M = 7;

vector<vector<int>> edges= {{0,1,2},{0,4,1},{4,5,4},{4,2,2},{1,2,3},{2,3,6},
{5,3,1}};
Solution obj;
vector < int > ans = obj.shortestPath(N, M, edges);

for (int i = 0; i < ans.size(); i++) {

    cout << ans[i] << " ";
}

return 0;
}

```

Output:

0 2 3 6 1 5

Time Complexity: $O(N+M)$ {for the topological sort} + $O(N+M)$ {for relaxation of vertices, each node and its adjacent nodes get traversed} $\sim O(N+M)$.

Where N = number of vertices and M = number of edges.

Space Complexity: $O(N)$ {for the stack storing the topological sort} + $O(N)$ {for storing the shortest distance for each node} + $O(N)$ {for the visited array} + $O(N+2M)$ {for the adjacency list} $\sim O(N+M)$.

Where N= number of vertices and M= number of edges.

*Special thanks to **Priyanshi Goel** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article. If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com*



Watch Video At: <https://youtu.be/ZUFQfFaU-8U>

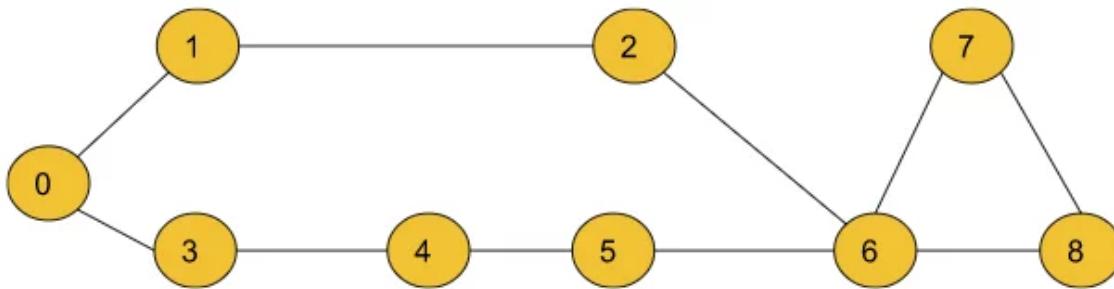
Shortest Path in Undirected Graph with unit distance: G-28

 takeuforward.org/data-structure/shortest-path-in-undirected-graph-with-unit-distance-g-28

October 17, 2022

Given an Undirected Graph having **unit weight**, find the shortest path from the source to all other nodes in this graph. In this problem statement, we have assumed the source vertex to be '0'. If a vertex is unreachable from the source node, then return -1 for that vertex.

Example 1:



Input:

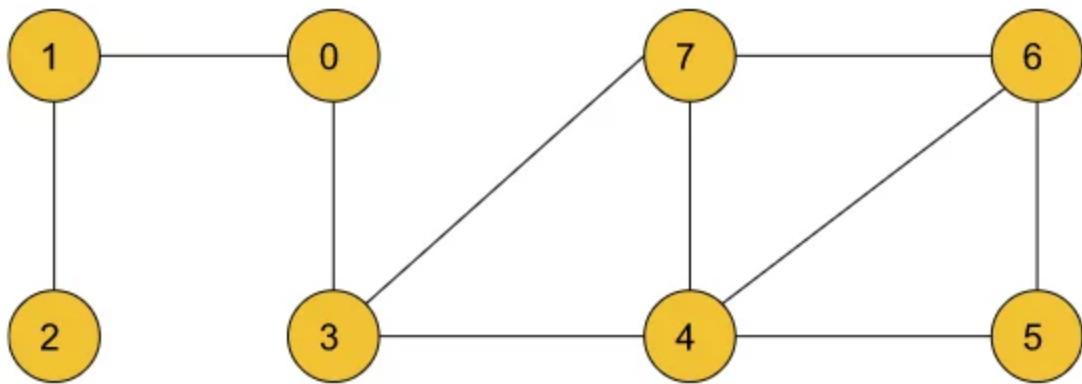
```
n = 9, m = 10  
edges = [[0,1],[0,3],[3,4],[4 ,5],[5, 6],[1,2],[2,6],[6,7],[7,8],[6,8]]  
src=0
```

Output: 0 1 2 1 2 3 3 4 4

Explanation:

The above output array shows the shortest path to all the nodes from the source vertex (0), $\text{Dist}[0] = 0$, $\text{Dist}[1] = 1$, $\text{Dist}[2] = 2$, ..., $\text{Dist}[8] = 4$. Where $\text{Dist}[\text{node}]$ is the shortest path between src and the node. For a node, if the value of $\text{Dist}[\text{node}] = -1$, then we conclude that the node is unreachable from the src node.

Example 2:



Input:

```
n = 8, m = 10
Edges =[[1,0],[2,1],[0,3],[3,7],[3,4],[7,4],[7,6],[4,5],[4,6],[6,5]]
src=0
```

Output: 0 1 2 1 2 3 3 2

Explanation:

The above output list shows the shortest path to all the nodes from the source vertex (0), Dist[0] = 0, Dist[1] = 1, Dist[2] = 2, Dist[7] = 2

Solution

Disclaimer: Don't jump directly to the solution, try it out yourself first.

Approach:

According to intuition, we will calculate the shortest path in an undirected graph having unit weights by using the Breadth First Search. BFS is a traversal technique where we visit the nodes level-wise, i.e., it visits the same level nodes simultaneously, and then moves to the next level.

Initial configuration:

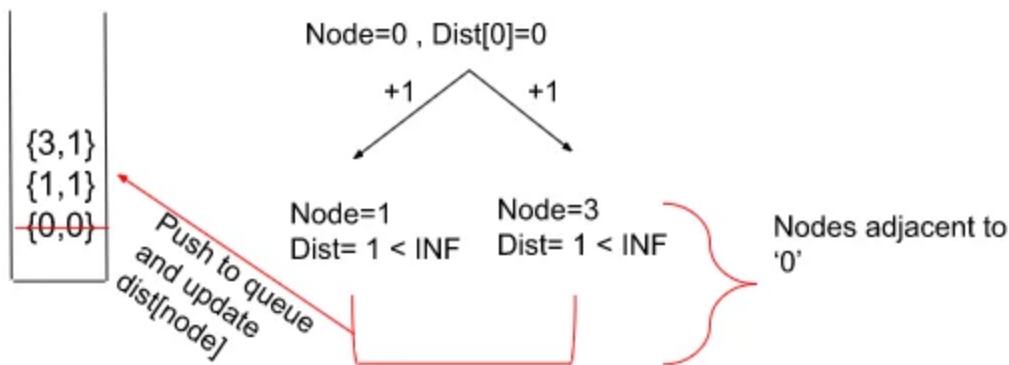
- **Adjacency List:** Create an adjacency list of the form vector of pairs of size ‘N’, where each index denotes a node ‘u’ and contains a vector that consists of pairs denoting the adjacent nodes ‘v’ and the distance to that adjacent node from initial node ‘u’.
- **Queue:** Define a queue data structure to store the BFS traversal.
- **Distance Array:** Initialise this array by Max Integer value and then update the value for each node successively while calculating the shortest distance between the source and the current node.

- **Resultant Array:** initialised with -1, this array stores the updated shortest distances from the source node after completion of the algorithm. The index which remains as -1 is said to be unreachable from the source node. This is required to return the answer according to the question.

The shortest path in an undirected graph can be calculated by the following steps:

- Firstly, we convert the graph into an adjacency list which displays the adjacent nodes for each index represented by a node.
- Now, we create a **dist** array of size N initialized with a very large number which can never be the answer to indicate that initially, all the nodes are untraversed.
- Then, perform the standard BFS traversal.
- In every iteration, pick up the front() node, and then traverse for its adjacent nodes. For every adjacent node, we will relax the distance to the adjacent node if ($\text{dist}[\text{node}] + 1 < \text{dist}[\text{adjNode}]$). Here $\text{dist}[\text{node}]$ means the distance taken to reach the current node, and '1' is the edge weight between the node and the adjNode. We will relax the edges if this distance is shorter than the previously taken distance. Every time a distance is updated for the adjacent node, we push that into the Queue with the increased distance.

Let us understand it using an example below,

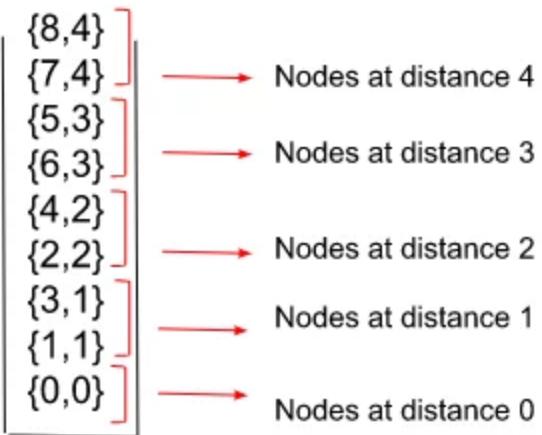


Where nodes '1' and '3' are adjacent to node '0'.

- Once all the nodes have been iterated, the **dist[]** array will store the shortest paths.
- Create a resultant array and initialize it by -1 and put all the distances which are updated in the resultant array. If anyone still holds the Large Integer value which we assigned at the start, it means it is not reachable, and we don't update our resultant array. The node which still remains marked as -1 is unreachable from the source node.

Intuition:

For finding the shortest path in an undirected graph with unit weight, the technique we use is the Breadth-First Search (BFS). Now, the question arises why do we use the BFS technique in finding the shortest path here when we could've easily used other standard graph shortest path algorithms to implement the same? If we start traversal from the src node, we move to other adjacent nodes, everyone is at a distance of 1, so everyone goes into the queue, then subsequently we get the next set of nodes at 1 more distance, making the distance to 2, and if you look at the queue closely, it will look something like below. Queue here acts as a sorted Queue, hence we don't need any sorted ds which we generally require in the other graph algorithms.



Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Code:

- C++ Code
- Java Code

```

#include<bits/stdc++.h>
using namespace std;

class Solution {
public:
    vector<int> shortestPath(vector<vector<int>>& edges, int N, int M, int src){
        //Create an adjacency list of size N for storing the undirected graph.
        vector<int> adj[N];
        for(auto it : edges) {
            adj[it[0]].push_back(it[1]);
            adj[it[1]].push_back(it[0]);
        }

        //A dist array of size N initialised with a large number to
        //indicate that initially all the nodes are untraversed.

        int dist[N];
        for(int i = 0;i<N;i++) dist[i] = 1e9;
        // BFS Implementation.
        dist[src] = 0;
        queue<int> q;
        q.push(src);
        while(!q.empty()) {
            int node = q.front();
            q.pop();
            for(auto it : adj[node]) {
                if(dist[node] + 1 < dist[it]) {
                    dist[it] = 1 + dist[node];
                    q.push(it);
                }
            }
        }
        // Updated shortest distances are stored in the resultant array 'ans'.
        // Unreachable nodes are marked as -1.
        vector<int> ans(N, -1);
        for(int i = 0;i<N;i++) {
            if(dist[i] != 1e9) {
                ans[i] = dist[i];
            }
        }
        return ans;
    }
};

int main(){
    int N=9, M=10;
    vector<vector<int>> edges= {{0,1},{0,3},{3,4},{4,5},{5,6},{1,2},{2,6},{6,7},{7,8},{6,8}};

    Solution obj;
    vector<int> ans = obj.shortestPath(edges,N,M,0);
}

```

```
for(int i=0;i<ans.size();i++){  
    cout<<ans[i]<<" ";  
}  
  
return 0;  
}
```

Output:

0 1 2 1 2 3 3 4 4

Time Complexity: $O(M)$ { for creating the adjacency list from given list ‘edges’} + $O(N + 2M)$ { for the BFS Algorithm} + $O(N)$ { for adding the final values of the shortest path in the resultant array} ~ $O(N+2M)$.

Where N= number of vertices and M= number of edges.

Space Complexity: $O(N)$ {for the stack storing the BFS} + $O(N)$ {for the resultant array} + $O(N)$ {for the dist array storing updated shortest paths} + $O(N+2M)$ {for the adjacency list} ~ $O(N+M)$.

Where N= number of vertices and M= number of edges.

Special thanks to Priyanshi Goel for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article. If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com



Watch Video At: <https://youtu.be/C4gxoTaI71U>

Dijkstra's Algorithm – Using Set : G-33

 takeuforward.org/data-structure/dijkstras-algorithm-using-set-g-33

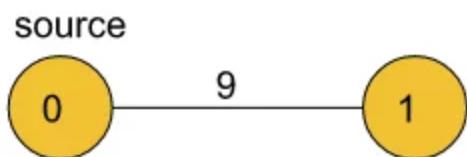
November 23, 2022

Given a weighted, undirected, and connected graph of V vertices and an adjacency list adj where $\text{adj}[i]$ is a list of lists containing two integers where the first integer of each list j denotes there is an **edge** between i and j , second integers corresponds to the weight of that edge. You are given the source vertex S and You have to Find the shortest distance of all the vertex from the source vertex S . You have to return a list of integers denoting the shortest distance between **each node** and Source vertex **S**.

Note: The Graph doesn't contain any negative weight cycle

Example 1:

Input :



```
V = 2
adj [] = {{{1, 9}}, {{0, 9}}}
S = 0
```

Output :

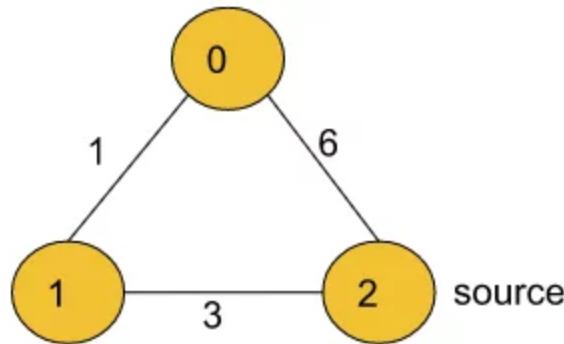
```
0 9
```

Explanation :

The source vertex is 0. Hence, the shortest distance of node 0 from the source is 0 and the shortest distance of node 1 from source will be 9.

Example 2:

Input:



```
V = 3, E = 3  
adj = {{{1, 1}, {2, 6}}, {{2, 3}, {0, 1}}, {{1, 3}, {0, 6}}}  
S = 2
```

Output:

```
4 3 0
```

Explanation:

For nodes 2 to 0, we can follow the path 2-1-0.

This has a distance of $1+3 = 4$, whereas the path 2-0 has a distance of 6. So, the Shortest path from 2 to 0 is 4.

The shortest distance from 0 to 1 is 1.

Solution

Disclaimer: Don't jump directly to the solution, try it out yourself first.

Problem Link

Note: In case any image/dry run is not clear please refer to the video attached at the bottom.

Approach:

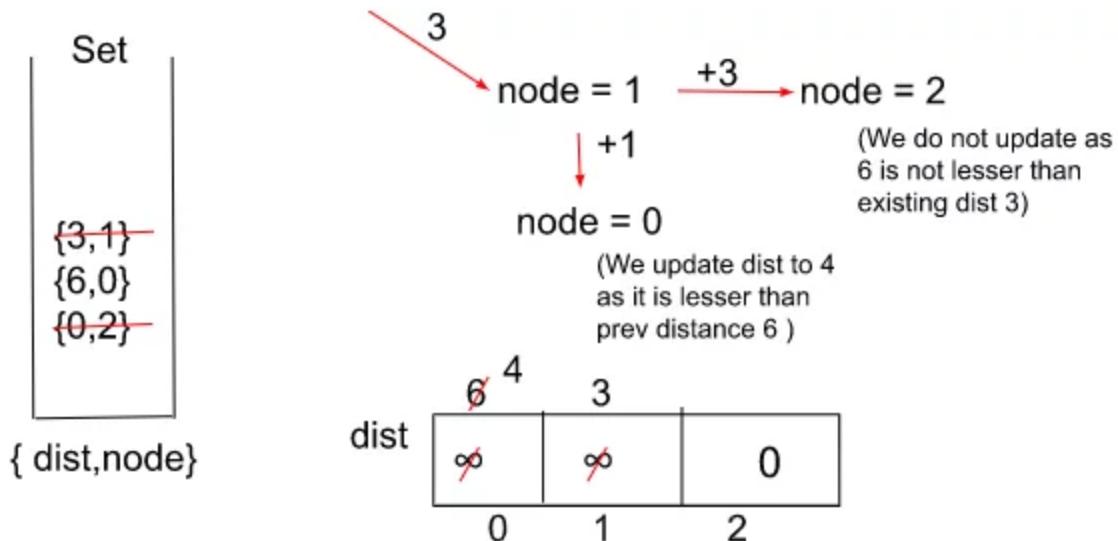
We'll be using Set in this approach for finding the shortest distances from the source node to every other node through Dijkstra's Algorithm.

Initial configuration:

- **Source Node:** Before starting off with the Algorithm, we need to define a source node from which the shortest distances to every other node would be calculated.
- **Set:** Define a Set that would contain pairs of the type {dist, node}, where 'dist' indicates the currently updated value of the shortest distance from the source to the 'node'.
- **Dist Array:** Define a dist array initialized with a large integer number at the start indicating that the nodes are unvisited initially. This array stores the shortest distances to all the nodes from the source node.

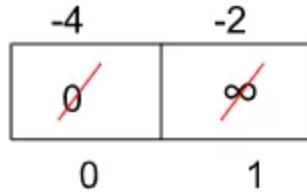
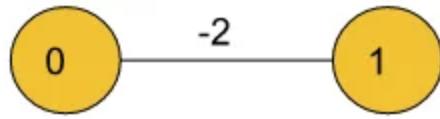
The Algorithm consists of the following steps :

- We would be using a set and a distance array of size V (where ' V ' are the number of nodes in the graph) initialized with infinity (indicating that at present none of the nodes are reachable from the source node) and initialize the distance to source node as 0.
- We push the source node to the set along with its distance which is also 0.
- Now, we start erasing the elements from the set and look out for their adjacent nodes one by one. If the current reachable distance is better than the previous distance indicated by the distance array, we update the distance and insert it in the set.
- A node with a lower distance would be first erased from the set as opposed to a node with a higher distance. By following step 3, until our set becomes empty, we would get the minimum distance from the source node to all other nodes. We can then return the distance array.
- The only difference between using a Priority Queue and a Set is that in a set we can check if there exists a pair with the same node but a greater distance than the current inserted node as there will be no point in keeping that node into the set if we come across a much better value than that. So, we simply delete the element with a greater distance value for the same node.
- Here's a quick demonstration of the algorithm :



Note: Dijkstra's Algorithm is not valid for negative weights or negative cycles.

We can understand that by looking at the illustration below :



Here, we initially mark the source node '0' as 0 and node '1' as INFINITY (as it is unvisited). Now, when we start applying the above algorithm on this we notice that both the nodes are updated each time we come to them again. This is due to the negative weight of '-2' which makes the total distance to the node always lesser than the previous value. Therefore, due to inaccurate results, we assume the graph to always contain positive weights while using Dijkstra's Algorithm.

Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Intuition:

The above problem implements a BFS kind of approach using the set data structure. The only thing that we need to take care of is that for all the paths possible between a pair of nodes, we need to store the path with the minimum cost between them. That is, say we have a node that has been reached by two paths, one with a cost of 7 and another with a cost of say 9. It is obvious that the path with a cost of 7 would be more optimal than the path with a cost of 9. A set data structure in C++ always stores the elements in increasing order i.e., when we erase from a set, the smallest valued elements get erased first.

Code:

C++ Code

```

#include<bits/stdc++.h>
using namespace std;

class Solution
{
public:
    //Function to find the shortest distance of all the vertices
    //from the source vertex S.
    vector <int> dijkstra(int V, vector<vector<int>> adj[], int S)
    {
        // Create a set ds for storing the nodes as a pair {dist,node}
        // where dist is the distance from source to the node.
        // set stores the nodes in ascending order of the distances
        set<pair<int,int>> st;

        // Initialising dist list with a large number to
        // indicate the nodes are unvisited initially.
        // This list contains distance from source to the nodes.
        vector<int> dist(V, 1e9);

        st.insert({0, S});

        // Source initialised with dist=0
        dist[S] = 0;

        // Now, erase the minimum distance node first from the set
        // and traverse for all its adjacent nodes.
        while(!st.empty()) {
            auto it = *(st.begin());
            int node = it.second;
            int dis = it.first;
            st.erase(it);

            // Check for all adjacent nodes of the erased
            // element whether the prev dist is larger than current or not.
            for(auto it : adj[node]) {
                int adjNode = it[0];
                int edgW = it[1];

                if(dis + edgW < dist[adjNode]) {
                    // erase if it was visited previously at
                    // a greater cost.
                    if(dist[adjNode] != 1e9)
                        st.erase({dist[adjNode], adjNode});

                    // If current distance is smaller,
                    // push it into the queue
                    dist[adjNode] = dis + edgW;
                    st.insert({dist[adjNode], adjNode});
                }
            }
        }
    }
}

```

```

        // Return the list containing shortest distances
        // from source to all the nodes.
        return dist;
    }
};

int main()
{
    // Driver code.
    int V = 3, E = 3, S = 2;
    vector<vector<int>> adj[V];
    vector<vector<int>> edges;
    vector<int> v1{1, 1}, v2{2, 6}, v3{2, 3}, v4{0, 1}, v5{1, 3}, v6{0, 6};
    int i = 0;
    adj[0].push_back(v1);
    adj[0].push_back(v2);
    adj[1].push_back(v3);
    adj[1].push_back(v4);
    adj[2].push_back(v5);
    adj[2].push_back(v6);

    Solution obj;
    vector<int> res = obj.dijkstra(V, adj, S);

    for (int i = 0; i < V; i++)
    {
        cout << res[i] << " ";
    }
    cout << endl;
    return 0;
}

```

Output:

4 3 0

Note: This set approach cannot be implemented in **JAVA** by using **TreeSet** or **HashSet**. For implementing Dijkstra's Algorithm in **JAVA**, we would use a priority queue only.

Time Complexity : $O(E \log(V))$

Where E = Number of edges and V = Number of Nodes.

Space Complexity : $O(|E| + |V|)$

Where E = Number of edges and V = Number of Nodes.

*Special thanks to **Priyanshi Goel** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, [please check out this article.](#) If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com*

Dijkstra's Algorithm – Using Priority Queue : G-32

 takeuforward.org/data-structure/dijkstras-algorithm-using-priority-queue-g-32

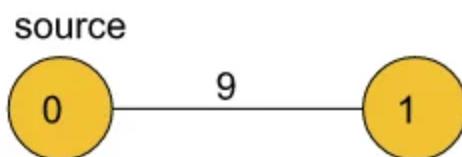
November 23, 2022

Given a weighted, undirected, and connected graph of V vertices and an adjacency list adj where $\text{adj}[i]$ is a list of lists containing two integers where the **first** integer of each list j denotes there is an **edge** between i and j , second integers corresponds to the **weight** of that edge. You are given the source vertex **S** and You have to Find the shortest distance of all the vertex from the source vertex **S**. You have to return a list of integers denoting the shortest distance between **each node** and the Source vertex **S**.

Note: The Graph doesn't contain any negative weight cycle.

Examples:

Example 1:



Input:

$V = 2$

$\text{adj} [] = \{\{\{1, 9\}\}, \{\{0, 9\}\}\}$

$S = 0$

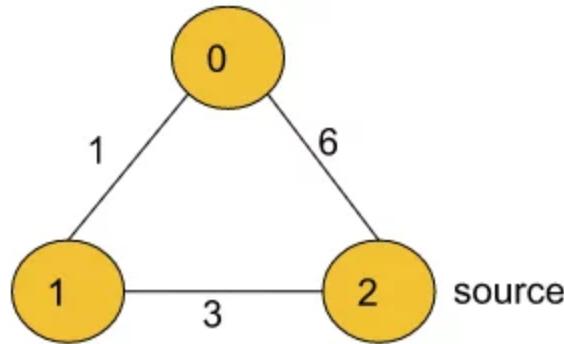
Output:

0 9

Explanation:

The source vertex is 0. Hence, the shortest distance of node 0 from the source is 0 and the shortest distance of node 1 from source will be 9.

Example 2:



Input:

$V = 3, E = 3$

$\text{adj} = \{\{\{1, 1\}, \{2, 6\}\}, \{\{2, 3\}, \{0, 1\}\}, \{\{1, 3\}, \{0, 6\}\}\}$

$S = 2$

Output:

4 3 0

Explanation:

For nodes 2 to 0, we can follow the path 2-1-0. This has a distance of $1+3 = 4$, whereas the path 2-0 has a distance of 6. So, the Shortest path from 2 to 0 is 4.

The shortest distance from 0 to 1 is 1.

Solution

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Problem Link

Note: In case any image/dry run is not clear please refer to the video attached at the bottom.

Approach:

We'll be using Priority Queue in this approach for finding the shortest distances from the source node to every other node through Dijkstra's Algorithm.

Initial configuration:

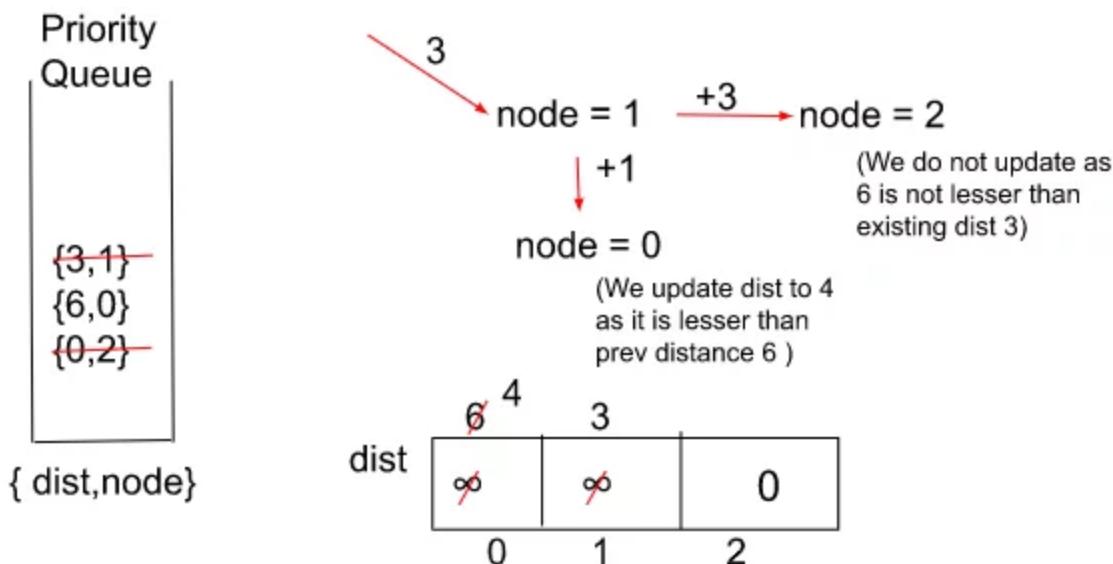
- **Source Node:** Before starting off with the Algorithm, we need to define a source node from which the shortest distances to every other node would be calculated.

- **Priority Queue:** Define a Priority Queue which would contain pairs of the type {dist, node}, where 'dist' indicates the currently updated value of the shortest distance from the source to the 'node'.
- **Dist Array:** Define a dist array initialized with a large integer number at the start indicating that the nodes are unvisited initially. This array stores the shortest distances to all the nodes from the source node.

The Algorithm consists of the following steps :

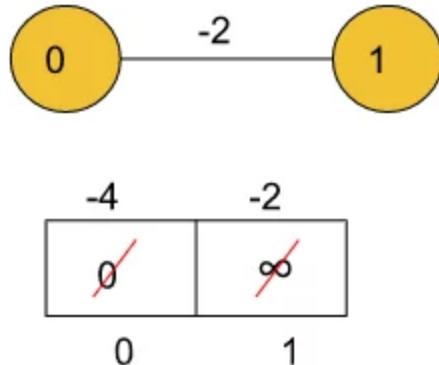
- We would be using a min-heap and a distance array of size V (where 'V' are the number of nodes in the graph) initialized with infinity (indicating that at present none of the nodes are reachable from the source node) and initialize the distance to source node as 0.
- We push the source node to the queue along with its distance which is also 0.
- For every node at the top of the queue, we pop the element out and look out for its adjacent nodes. If the current reachable distance is better than the previous distance indicated by the distance array, we update the distance and push it into the queue.

A node with a lower distance would be at the top of the priority queue as opposed to a node with a higher distance because we are using a min-heap. By following step 3, until our queue becomes empty, we would get the minimum distance from the source node to all other nodes. We can then return the distance array. Here's a quick demonstration of the algorithm :



Note: Dijkstra's Algorithm is not valid for negative weights or negative cycles.

We can understand that by looking at the illustration below :



Here, we initially mark the source node '0' as 0 and node '1' as INFINITY (as it is unvisited). Now, when we start applying the above algorithm on this we notice that both the nodes are updated each time we come to them again. This is due to the negative weight of '-2' which makes the total distance to the node always lesser than the previous value. Therefore, due to inaccurate results, we assume the graph to always contain positive weights while using Dijkstra's Algorithm.

Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Intuition:

The above problem seems familiar to finding the shortest distance in the case of unit edge weights for undirected graphs. Hence, our first guess could be a BFS kind of approach. The only thing that we need to take care of is that for all the paths possible between a pair of nodes, we need to store the path with the minimum cost between them. That is, say we have a node that has been reached by two paths, one with a cost of 7 and another with a cost of say 9. It is obvious that the path with a cost of 7 would be more optimal than the path with a cost of 9.

For knowing the intuition behind why Priority Queue is being used for storing distances of nodes from the source, please watch [this video](#) of the series to get a clear understanding of why instead a Queue is not being used in this approach.

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution
{
public:
    // Function to find the shortest distance of all the vertices
    // from the source vertex S.
    vector<int> dijkstra(int V, vector<vector<int>> adj[], int s)
    {

        // Create a priority queue for storing the nodes as a pair {dist,node}
        // where dist is the distance from source to the node.
        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

        // Initialising distTo list with a large number to
        // indicate the nodes are unvisited initially.
        // This list contains distance from source to the nodes.
        vector<int> distTo(V, INT_MAX);

        // Source initialised with dist=0.
        distTo[S] = 0;
        pq.push({0, S});

        // Now, pop the minimum distance node first from the min-heap
        // and traverse for all its adjacent nodes.
        while (!pq.empty())
        {
            int node = pq.top().second;
            int dis = pq.top().first;
            pq.pop();

            // Check for all adjacent nodes of the popped out
            // element whether the prev dist is larger than current or not.
            for (auto it : adj[node])
            {
                int v = it[0];
                int w = it[1];
                if (dis + w < distTo[v])
                {
                    distTo[v] = dis + w;

                    // If current distance is smaller,
                    // push it into the queue.
                    pq.push({dis + w, v});
                }
            }
        }

        // Return the list containing shortest distances
        // from source to all the nodes.
        return distTo;
    }
};

```

```

    }

};

int main()
{
    // Driver code.
    int V = 3, E = 3, S = 2;
    vector<vector<int>> adj[V];
    vector<vector<int>> edges;
    vector<int> v1{1, 1}, v2{2, 6}, v3{2, 3}, v4{0, 1}, v5{1, 3}, v6{0, 6};
    int i = 0;
    adj[0].push_back(v1);
    adj[0].push_back(v2);
    adj[1].push_back(v3);
    adj[1].push_back(v4);
    adj[2].push_back(v5);
    adj[2].push_back(v6);

    Solution obj;
    vector<int> res = obj.dijkstra(V, adj, S);

    for (int i = 0; i < V; i++)
    {
        cout << res[i] << " ";
    }
    cout << endl;
    return 0;
}

```

Output:

4 3 0

Time Complexity: $O(E \log(V))$, Where E = Number of edges and V = Number of Nodes.

Space Complexity: $O(|E| + |V|)$, Where E = Number of edges and V = Number of Nodes.

Special thanks to [Priyanshi Goel](#) for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, [please check out this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com



Watch Video At: <https://youtu.be/V6H1qAeB-I4>

G-36: Shortest Distance in a Binary Maze

 takeuforward.org/data-structure/g-36-shortest-distance-in-a-binary-maze

January 5, 2023

Problem Statement:

Given an $n * m$ matrix grid where each element can either be **0** or **1**. You need to find the shortest distance between a given source cell to a destination cell. The path can only be created out of a cell if its value is 1.

If the path is not possible between the source cell and the destination cell, then return **-1**.

Note: You can move into an adjacent cell if that adjacent cell is filled with element 1. Two cells are adjacent if they share a side. In other words, you can move in one of four directions, Up, Down, Left, and Right.

Examples:

Example 1:

```
Input:  
grid[][] = {{1, 1, 1, 1},  
            {1, 1, 0, 1},  
            {1, 1, 1, 1},  
            {1, 1, 0, 0},  
            {1, 0, 0, 1}}  
source = {0, 1}  
destination = {2, 2}  
Output:  
3
```

Explanation:

```
1 1 1 1  
1 1 0 1  
1 1 1 1  
1 1 0 0  
1 0 0 1
```

The highlighted part in the above matrix denotes the shortest path from source to destination cell.

Example 2:

```
Input:  
grid[][] = {{1, 1, 1, 1, 1},  
            {1, 1, 1, 1, 1},  
            {1, 1, 1, 1, 0},  
            {1, 0, 1, 0, 1}}  
source = {0, 0}  
destination = {3, 4}  
Output:  
-1
```

Explanation:

Since, there is no path possible between the source cell and the destination cell, hence we return -1.

Solution

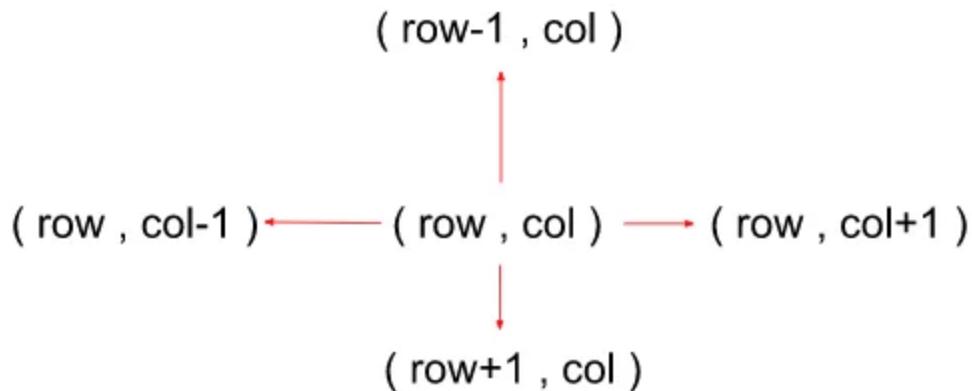
Disclaimer: Don't jump directly to the solution, try it out yourself first.

Problem Link

Note: In case any image/dry run is not clear please refer to the video attached at the bottom.

Approach:

We'll solve this problem by Dijkstra's Algorithm using a simple queue. Since, there is no adjacency list for this particular problem we can say that the adjacent cell for a coordinate is that cell which is either on the top, bottom, left, or right of the current cell i.e, a cell can have a maximum of 4 cells adjacent to it.



Initial configuration:

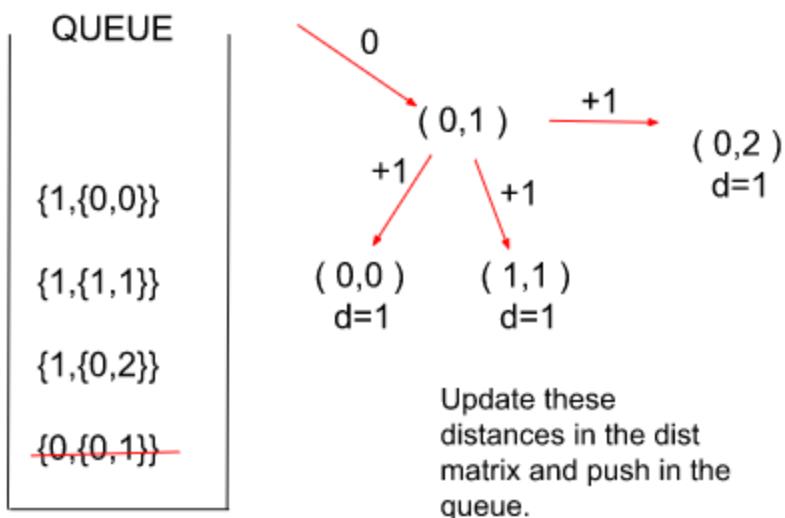
- **Source Node and Destination Node:** Before starting off with the Algorithm, we need to define a source node and a destination node, between which we need the shortest possible distance.
- **Queue:** Define a Queue which would contain pairs of the type {dist, pair of coordinates of cell }, where 'dist' indicates the currently updated value of the shortest distance from the source to the cell.
- **Distance Matrix:** Define a distance matrix that would contain the distance from the source cell to that particular cell. If a cell is marked as 'infinity' then it is treated as unreachable/unvisited.

The Algorithm consists of the following steps :

- Start by creating a queue that stores the distance-node pairs in the form {dist, coordinates of cell pair} and a dist matrix with each cell initialized with a very large number (to indicate that they're unvisited initially) and the source cell marked as '0'.
- We push the source cell to the queue along with its distance which is also 0.
- Pop the element at the front of the queue and look out for its adjacent nodes (left, right, bottom, and top cell). Also, for each cell, check the validity of the cell if it lies within the limits of the matrix or not.
- If the current reachable distance to a cell from the source is better than the previous distance indicated by the distance matrix, we update the distance and push it into the queue along with cell coordinates.
- A cell with a lower distance would be at the front of the queue as opposed to a node with a higher distance. We repeat the above two steps until the queue becomes empty or until we encounter the destination node.

- Return the calculated distance and stop the algorithm from reaching the destination node. If the queue becomes empty and we don't encounter the destination node, return '-1' indicating there's no path from source to destination.
- Here's a quick demonstration of the algorithm :

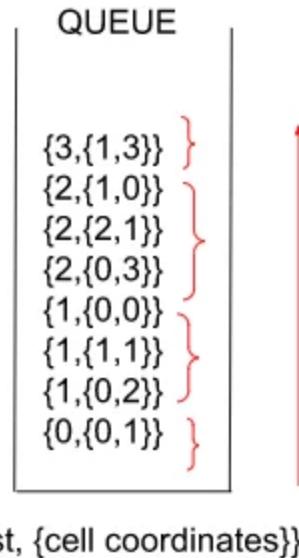
grid[][]				dist[][]			
1	s 1	1	1	∞	0	∞	1
1	1	0	1	∞	∞	∞	∞
1	1	d 1	1	∞	∞	∞	∞
1	1	0	0	∞	∞	∞	∞
1	0	0	0	∞	∞	∞	∞



{dist, {cell coordinates}}

Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Intuition: Here in this problem, instead of a graph we have a 2D binary matrix in which we have to reach a destination cell from a source cell. So, we can see that this problem is easily approachable by Dijkstra's Algorithm. Now, here we use a **queue** instead of a **priority queue** for storing the distance-node pairs. Let's understand through an illustration why a queue is better here:



We can see clearly in the above illustration that the distances are increasing monotonically (because of constant edge weights). Since greater distance comes at the top automatically, so we do not need the priority queue as the pop operation will always pop the smaller distance which is at the front of the queue. This helps us to eliminate an additional $\log(N)$ of time needed to perform insertion-deletion operations in a priority queue.

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution
{
public:
    int shortestPath(vector<vector<int>> &grid, pair<int, int> source,
                    pair<int, int> destination)
    {
        // Edge Case: if the source is only the destination.
        if (source.first == destination.first &&
            source.second == destination.second)
            return 0;

        // Create a queue for storing cells with their distances from source
        // in the form {dist,{cell coordinates pair}}.
        queue<pair<int, pair<int, int>> q;
        int n = grid.size();
        int m = grid[0].size();

        // Create a distance matrix with initially all the cells marked as
        // unvisited and the source cell as 0.
        vector<vector<int>> dist(n, vector<int>(m, 1e9));
        dist[source.first][source.second] = 0;
        q.push({0, {source.first, source.second}});

        // The following delta rows and delts columns array are created such that
        // each index represents each adjacent node that a cell may have
        // in a direction.
        int dr[] = {-1, 0, 1, 0};
        int dc[] = {0, 1, 0, -1};

        // Iterate through the maze by popping the elements out of the queue
        // and pushing whenever a shorter distance to a cell is found.
        while (!q.empty())
        {
            auto it = q.front();
            q.pop();
            int dis = it.first;
            int r = it.second.first;
            int c = it.second.second;

            // Through this loop, we check the 4 direction adjacent nodes
            // for a shorter path to destination.
            for (int i = 0; i < 4; i++)
            {
                int newr = r + dr[i];
                int newc = c + dc[i];

                // Checking the validity of the cell and updating if dist is shorter.
                if (newr >= 0 && newr < n && newc >= 0 && newc < m && grid[newr]
[newc]

```

```

== 1 && dis + 1 < dist[newr][newc])
{
    dist[newr][newc] = 1 + dis;

    // Return the distance until the point when
    // we encounter the destination cell.
    if (newr == destination.first &&
        newc == destination.second)
        return dis + 1;
    q.push({1 + dis, {newr, newc}});
}
}

// If no path is found from source to destination.
return -1;
}

};

int main()
{
    // Driver Code.

    pair<int, int> source, destination;
    source.first = 0;
    source.second = 1;
    destination.first = 2;
    destination.second = 2;

    vector<vector<int>> grid = {{1, 1, 1, 1},
                                {1, 1, 0, 1},
                                {1, 1, 1, 1},
                                {1, 1, 0, 0},
                                {1, 0, 0, 1}};

    Solution obj;

    int res = obj.shortestPath(grid, source, destination);

    cout << res;
    cout << endl;

    return 0;
}

```

Output :

3

Time Complexity: O(4*N*M) { N*M are the total cells, for each of which we also check 4 adjacent nodes for the shortest path length}, Where N = No. of rows of the binary maze and M = No. of columns of the binary maze.

Space Complexity: O(N*M), Where N = No. of rows of the binary maze and M = No. of columns of the binary maze.

Special thanks to Priyanshi Goel for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article. If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com



Watch Video At: <https://youtu.be/U5Mw4eyUmw4>

G-37: Path With Minimum Effort

 takeuforward.org/data-structure/g-37-path-with-minimum-effort

January 5, 2023

You are a hiker preparing for an upcoming hike. You are given heights, a 2D array of size rows x columns, where heights[row][col] represents the height of the cell (row, col). You are situated in the top-left cell, (0, 0), and you hope to travel to the bottom-right cell, (rows-1, columns-1) (i.e., **0-indexed**). You can move **up**, **down**, **left**, or **right**, and you wish to find a route that requires the **minimum effort**.

A route's **effort** is the **maximum absolute difference** in heights between two consecutive cells of the route.

Examples:

Example 1:

Input:

```
heights = [[1,2,2],[3,8,2],[5,3,5]]
```

Output:

```
2
```

Explanation:

The route of [1,3,5,3,5] has a maximum absolute difference of 2 in consecutive cells. This is better than the route of [1,2,2,2,5], where the maximum absolute difference is 3.

Example 2:

Input:

```
heights = [[1,2,1,1,1],[1,2,1,2,1],[1,2,1,2,1],[1,1,1,2,1]]
```

Output:

```
0
```

Explanation:

The route of [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1] has a maximum absolute difference of 0 in consecutive cells. This is better than the route of [1,1,1,1,1,1,2,1], where the maximum absolute difference is 1.

Solution

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Problem Link

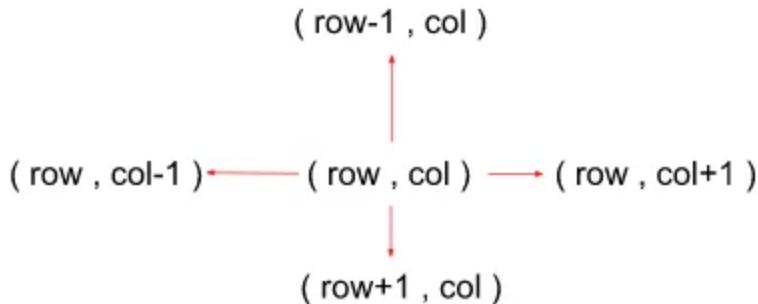
Note: If any image/dry run is unclear, please refer to the video attached at the bottom.

Approach:

Brute Force: We can figure out the effort for all the paths and return the minimum effort among them.

Optimised (Using Dijkstra) :

In this particular problem, since there is no adjacency list we can say that the adjacent cell for a coordinate is that cell which is either on the top, bottom, left, or right of the current cell i.e, a cell can have a maximum of 4 cells adjacent to it and can only move in these directions.



Initial configuration:

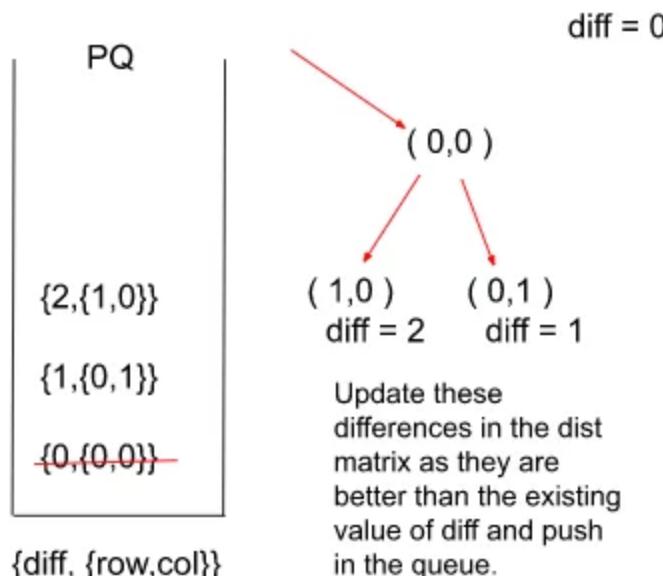
- **Queue:** Define a Queue which would contain pairs of the type $\{\text{diff}, (\text{row}, \text{col})\}$, where ‘dist’ indicates the currently updated value of difference from source to the cell.
- **Distance Matrix:** Define a distance matrix that would contain the minimum difference from the source cell to that particular cell. If a cell is marked as ‘infinity’ then it is treated as unreachable/unvisited.

The Algorithm consists of the following steps :

- Start by creating a queue that stores the distance-node pairs in the form $\{\text{dist}, (\text{row}, \text{col})\}$ and a dist matrix with each cell initialized with a very large number (to indicate that they're unvisited initially) and the source cell marked as ‘0’.
- We push the source cell to the queue along with its distance which is also 0.
- Pop the element at the front of the queue and look out for its adjacent nodes (left, right, bottom, and top cell). Also, for each cell, check the validity of the cell if it lies within the limits of the matrix or not.
- If the current difference value of a cell from its parent is better than the previous difference indicated by the distance matrix, we update the difference in the matrix and push it into the queue along with cell coordinates.
- A cell with a lower difference value would be at the front of the queue as opposed to a node with a higher difference. The only difference between this problem and Dijkstra's Standard problem is that there we used to update the value of the distance of a node from the source and here we update the absolute **difference** of a node from its parent.

- We repeat the above three steps until the queue becomes empty or until we encounter the destination node.
- Return the calculated difference and stop the algorithm from reaching the destination node. If the queue becomes empty and we don't encounter the destination node, return '0' indicating there's no path from source to destination.
- Here's a quick demonstration of the Algorithm's 1st iteration (all the further iterations would be done in a similar way) :

heights[][], dist[][]		
s 1	2	2
3	8	2
5	3	d 5



Note: Updating the value of difference will only yield us the **effort** for the path traversed.

Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Intuition:

In this problem, we need to minimize the **effort** of moving from the source cell (0,0) to the destination cell ($n - 1, m - 1$). The effort can be calculated as the maximum value of the difference between the node and its next node in the path from the source to the destination. Among all the possible paths, we have to **minimize** this effort. So, for these types of

minimum path problems, there's one standard algorithm that always comes to our mind and that is Dijkstra's Algorithm which would be used in solving this problem also. We update the distance every time we encounter a value of difference less than the previous value. This way, whenever we reach the destination we finally return the value of difference which is also the **minimum effort**.

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution
{
public:
    int MinimumEffort(vector<vector<int>> &heights)
    {

        // Create a priority queue containing pairs of cells
        // and their respective distance from the source cell in the
        // form {diff, {row of cell, col of cell}}.
        priority_queue<pair<int, pair<int, int>>,
            vector<pair<int, pair<int, int>>>,
            greater<pair<int, pair<int, int>>>
        pq;

        int n = heights.size();
        int m = heights[0].size();

        // Create a distance matrix with initially all the cells marked as
        // unvisited and the dist for source cell (0,0) as 0.
        vector<vector<int>> dist(n, vector<int>(m, 1e9));
        dist[0][0] = 0;
        pq.push({0, {0, 0}});

        // The following delta rows and delts columns array are created such that
        // each index represents each adjacent node that a cell may have
        // in a direction.
        int dr[] = {-1, 0, 1, 0};
        int dc[] = {0, 1, 0, -1};

        // Iterate through the matrix by popping the elements out of the queue
        // and pushing whenever a shorter distance to a cell is found.
        while (!pq.empty())
        {
            auto it = pq.top();
            pq.pop();
            int diff = it.first;
            int row = it.second.first;
            int col = it.second.second;

            // Check if we have reached the destination cell,
            // return the current value of difference (which will be min).
            if (row == n - 1 && col == m - 1)
                return diff;

            for (int i = 0; i < 4; i++)
            {
                // row - 1, col
                // row, col + 1
                // row - 1, col

```

```

        // row, col - 1
        int newr = row + dr[i];
        int newc = col + dc[i];

        // Checking validity of the cell.
        if (newr >= 0 && newc >= 0 && newr < n && newc < m)
        {
            // Effort can be calculated as the max value of differences
            // between the heights of the node and its adjacent nodes.
            int newEffort = max(abs(heights[row][col] - heights[newr][newc]),
diff);

            // If the calculated effort is less than the prev value
            // we update as we need the min effort.
            if (newEffort < dist[newr][newc])
            {
                dist[newr][newc] = newEffort;
                pq.push({newEffort, {newr, newc}});
            }
        }
    }

    return 0; // if unreachable
}

int main()
{
    // Driver Code.

    vector<vector<int>> heights = {{1, 2, 2}, {3, 8, 2}, {5, 3, 5}};

    Solution obj;

    int ans = obj.MinimumEffort(heights);

    cout << ans;
    cout << endl;

    return 0;
}

```

Output :

2

Time Complexity: $O(4 \cdot N \cdot M \cdot \log(N \cdot M))$ { $N \cdot M$ are the total cells, for each of which we also check 4 adjacent nodes for the minimum effort and additional $\log(N \cdot M)$ for insertion-deletion operations in a priority queue }

Where, N = No. of rows of the binary maze and M = No. of columns of the binary maze.

Space Complexity: $O(N \times M)$ { Distance matrix containing $N \times M$ cells + priority queue in the worst case containing all the nodes ($N \times M$) }.

Where, N = No. of rows of the binary maze and M = No. of columns of the binary maze.

Special thanks to Priyanshi Goel for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article. If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com

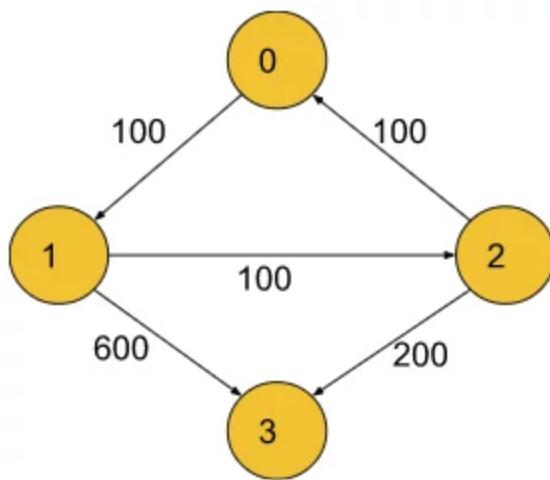
G-38: Cheapest Flights Within K Stops

 takeuforward.org/data-structure/g-38-cheapest-flights-within-k-stops

January 10, 2023

There are n cities and m edges connected by some number of flights. You are given an array of flights where $\text{flights}[i] = [\text{from}_i, \text{to}_i, \text{price}_i]$ indicates that there is a flight from city from_i to city to_i with cost price. You have also given three integers src , dst , and k , **and return the cheapest price from src to dst with at most k stops**. If there is no such route, return -1.

Example 1:



Input:

```
n = 4
flights = [[0,1,100],[1,2,100],[2,0,100],[1,3,600],[2,3,200]]
src = 0
dst = 3
k = 1
```

Output:

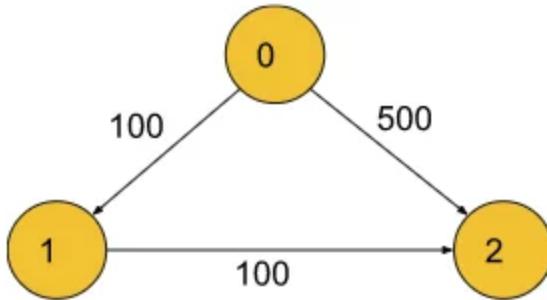
```
700
```

Explanation:

The optimal path with at most 1 stops from city 0 to 3 is marked in red and has cost $100 + 600 = 700$.

Note that the path through cities $[0,1,2,3]$ is cheaper but is invalid because it uses 2 stops.

Example 2:



Input:

```
n = 3
flights = [[0,1,100],[1,2,100],[0,2,500]]
```

```
src = 0
```

```
dst = 2
```

```
k = 1
```

Output:

```
200
```

Explanation:

The graph is shown above.

The optimal path with at most 1 stops from city 0 to 2 is marked in red and has cost $100 + 100 = 200$.

Disclaimer: Don't jump directly to the solution, try it out yourself first.

Problem Link

Note: In case any image/dry run is not clear please refer to the video attached at the bottom.

Intuition:

Since in this problem, we have to calculate the minimum cost to reach the destination from the source but with a restriction on the number of stops, we would be using Dijkstra's Algorithm. Now, one must wonder that based on what parameter we should add elements to the priority queue.

Now, if we store the elements in the priority queue with the priority given to the minimum distance first, then after a few iterations we would realize that the Algorithm will halt when the number of stops would exceed. This may result in a wrong answer as it would not allow us to explore those paths which have more cost but fewer stops than the current answer.

To tackle this issue, we store the elements in terms of the minimum number of **stops** in the priority queue so that when the algorithm halts, we can get the minimum cost within limits.

Also, a point to note here is that do we really need a priority queue for carrying out the algorithm? The answer for that is **No** because when we are storing everything in terms of a number of stops, the stops are increasing monotonically which means that the number of

sops is increasing by 1 and when we pop an element out of the queue, we are always popping the element with a lesser number of stops first. Replacing the priority queue with a simple queue will let us eliminate an extra $\log(N)$ of the complexity of insertion-deletion in a priority queue which would in turn make our algorithm a lot faster.

Approach:

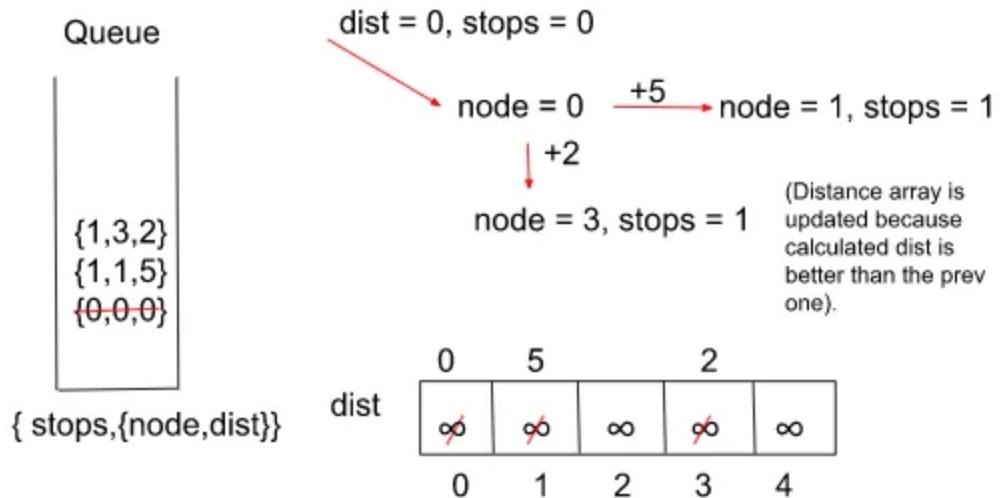
Initial configuration:

- **Queue:** Define a Queue that would contain pairs of the type {stops, {node,dist} }, where 'dist' indicates the currently updated value of the distance from the source to the 'node' and 'stops' contains the number of nodes one has to traverse in order to reach node from src.
- **Distance Array:** Define a distance array that would contain the minimum cost/distance from the source cell to a particular cell. If a cell is marked as 'infinity' then it is treated as unreachable/unvisited.
- **Source and Destination:** Define the source and the destination from where the flights have to run.

The Algorithm consists of the following steps :

- Start by creating an adjacency list, a queue that stores the distance-node and stops pairs in the form {stops,{node,dist}} and a dist array with each node initialized with a very large number (to indicate that they're unvisited initially) and the source node marked as '0'.
- We push the source cell to the queue along with its distance which is also 0 and the stops are marked as '0' initially because we've just started.
- Pop the element at the front of the queue and look out for its adjacent nodes.
- If the current dist value of a node is better than the previous distance indicated by the distance array and the number of stops until now is less than K, we update the distance in the array and push it to the queue. Also, increase the stop count by 1.
- We repeat the above three steps until the queue becomes empty. Note that we **do not** stop the algorithm from just reaching the destination node as it may give incorrect results.
- Return the calculated distance/cost after we reach the required number of stops. If the queue becomes empty and we don't encounter the destination node, return '-1' indicating there's no path from source to destination.

Here's a quick demonstration of the Algorithm's 1st iteration for **example 1** stated above (all the further iterations would be done in a similar way) :



Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution
{
public:
    int CheapestFLight(int n, vector<vector<int>> &flights,
                        int src, int dst, int K)
    {
        // Create the adjacency list to depict airports and flights in
        // the form of a graph.
        vector<pair<int, int>> adj[n];
        for (auto it : flights)
        {
            adj[it[0]].push_back({it[1], it[2]});
        }

        // Create a queue which stores the node and their distances from the
        // source in the form of {stops, {node, dist}} with 'stops' indicating
        // the no. of nodes between src and current node.
        queue<pair<int, pair<int, int>>> q;

        q.push({0, {src, 0}});

        // Distance array to store the updated distances from the source.
        vector<int> dist(n, 1e9);
        dist[src] = 0;

        // Iterate through the graph using a queue like in Dijkstra with
        // popping out the element with min stops first.
        while (!q.empty())
        {
            auto it = q.front();
            q.pop();
            int stops = it.first;
            int node = it.second.first;
            int cost = it.second.second;

            // We stop the process as soon as the limit for the stops reaches.
            if (stops > K)
                continue;
            for (auto iter : adj[node])
            {
                int adjNode = iter.first;
                int edW = iter.second;

                // We only update the queue if the new calculated dist is
                // less than the prev and the stops are also within limits.
                if (cost + edW < dist[adjNode] && stops <= K)
                {
                    dist[adjNode] = cost + edW;
                    q.push({stops + 1, {adjNode, cost + edW}});
                }
            }
        }
    }
};

```

```

        }
    }
}

// If the destination node is unreachable return '-1'
// else return the calculated dist from src to dst.
if (dist[dst] == 1e9)
    return -1;
return dist[dst];
}
};

int main()
{
    // Driver Code.
    int n = 4, src = 0, dst = 3, K = 1;

    vector<vector<int>> flights = {{0, 1, 100}, {1, 2, 100}, {2, 0, 100}, {1, 3, 600},
{2, 3, 200}};

    Solution obj;

    int ans = obj.CheapestFlight(n, flights, src, dst, K);

    cout << ans;
    cout << endl;

    return 0;
}

```

Output :

700

Time Complexity: $O(N)$ { Additional $\log(N)$ of time eliminated here because we're using a simple **queue** rather than a **priority queue** which is usually used in Dijkstra's Algorithm }.

Where N = Number of flights / Number of edges.

Space Complexity: $O(|E| + |V|)$ { for the adjacency list, priority queue, and the dist array }.

Where E = Number of edges (`flights.size()`) and V = Number of Airports.

Special thanks to Priyanshi Goel for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article. If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com



Watch Video At: <https://youtu.be/9XybHVqTHcQ>

G-39: Minimum Multiplications to Reach End

 takeuforward.org/graph/g-39-minimum-multiplications-to-reach-end/

January 10, 2023

Given **start**, **end**, and an array **arr** of **n** numbers. At each step, the **start** is multiplied by any number in the array and then a mod operation with **100000** is done to get the new start. Your task is to find the minimum steps in which **the end** can be achieved starting from **the start**. If it is not possible to reach the **end**, then return **-1**.

Example 2:

Input:

```
arr[] = {2, 5, 7}
start = 3
end = 30
```

Output:

```
2
```

Explanation:

Step 1: $3 * 2 = 6 \ % \ 100000 = 6$

Step 2: $6 * 5 = 30 \ % \ 100000 = 30$

Therefore, in minimum 2 multiplications we reach the

end number which is treated as a destination

node of a graph here.

Example 2:

Input:

```
arr[] = {3, 4, 65}
start = 7
end = 66175
```

Output:

```
4
```

Explanation:

Step 1: $7 * 3 = 21 \ % \ 100000 = 21$

Step 2: $21 * 3 = 63 \ % \ 100000 = 63$

Step 3: $63 * 65 = 4095 \ % \ 100000 = 4095$

Step 4: $4095 * 65 = 266175 \ % \ 100000 = 66175$

Therefore, in minimum 4 multiplications we reach the end number which is treated as a destination node of a graph here.

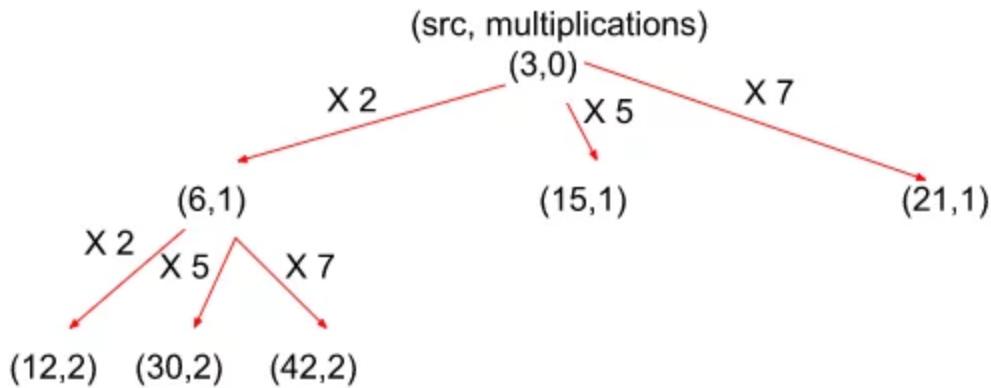
Solution

Disclaimer: Don't jump directly to the solution, try it out yourself first.

Problem Link

Note: In case any image/dry run is not clear please refer to the video attached at the bottom.

Intuition: Since we need to find the **minimum** number of multiplications in order to attain the end number from the start number, the one standard algorithm that we can think of for finding the shortest/minimum paths is of course Dijkstra's Algorithm. But a question may arise in our minds: how can we depict this problem in the form of a graph? So let us understand this through an illustration :



As per the above image, we can clearly make out that the numbers after multiplication with the start number can be considered as nodes of a graph ranging from 0 to 99999 and the edges from each node will be the size of the given array i.e. the number of ways in which we can multiply a number. We update the distance array whenever we find a lesser number of multiplications in order to reach a node. In this way, whenever we reach the end number, the multiplications needed to reach it would always be **minimum**.

Approach:

This problem can be visualized as a graph problem as we need to find the minimum number of steps to reach an end number from the start following a number of multiplications. We would be solving it using Dijkstra's Algorithm.

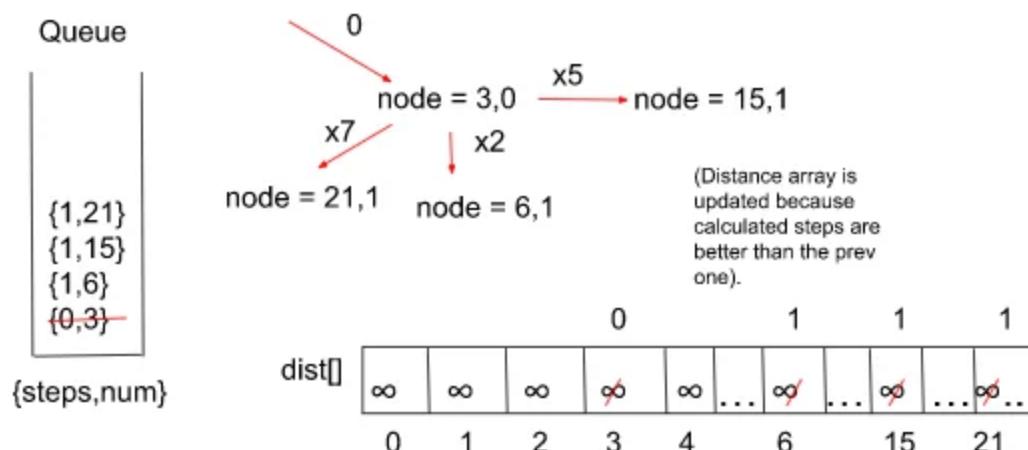
Initial configuration:

- **Queue:** Define a Queue which would contain pairs of the type {steps, num }, where 'steps' indicates the currently updated value of no. of steps taken to reach from source to the current 'num'.
- **Distance Array:** Define a distance array that would contain the minimum no. of multiplications/distance from the start number to the current number. If a cell is marked as 'infinity' then it is treated as unreachable/unattained.
- **Start and End:** Define the start and the end value which we have to reach through a series of multiplications.

The Algorithm consists of the following steps :

- Start by creating a queue that stores the step-num pairs in the form {steps, num} and a dist array with each node initialized with a very large number (to indicate that they've not been attained initially). The size of the 'dist' array is set to 100000 because it is the maximum number of distinct numbers that can be generated.
- We push the start number to the queue along with its steps marked as '0' initially because we've just started the algorithm.
- Pop the element from the front of the queue and look out for its adjacent nodes (here, adjacent nodes can be regarded as the numbers which we get when we multiply the start number by each element from the arr).
- If the current dist value for a number is better than the previous distance indicated by the distance array, we update the distance/steps in the array and push it to the queue.
- We repeat the above three steps until the queue becomes empty or we reach the end number.
- Return the calculated number of steps after we reach the end number. If the queue becomes empty and we don't encounter the required number, return '-1' indicating that the following number is unattainable by multiplying the start number any number of times.

Here's a quick demonstration of the Algorithm's 1st iteration for **example 1** stated above (all the further iterations would be done in a similar way) :



Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution
{
public:
    int minimumMultiplications(vector<int> &arr,
                               int start, int end)
    {
        // Create a queue for storing the numbers as a result of multiplication
        // of the numbers in the array and the start number.
        queue<pair<int, int>> q;
        q.push({start, 0});

        // Create a dist array to store the no. of multiplications to reach
        // a particular number from the start number.
        vector<int> dist(100000, 1e9);
        dist[start] = 0;
        int mod = 100000;

        // Multiply the start no. with each of numbers in the arr
        // until we get the end no.
        while (!q.empty())
        {
            int node = q.front().first;
            int steps = q.front().second;
            q.pop();

            for (auto it : arr)
            {
                int num = (it * node) % mod;

                // If the no. of multiplications are less than before
                // in order to reach a number, we update the dist array.
                if (steps + 1 < dist[num])
                {
                    dist[num] = steps + 1;

                    // Whenever we reach the end number
                    // return the calculated steps
                    if (num == end)
                        return steps + 1;
                    q.push({num, steps + 1});
                }
            }
        }
        // If the end no. is unattainable.
        return -1;
    }
};

int main()

```

```

{
    // Driver Code.
    int start = 3, end = 30;

    vector<int> arr = {2, 5, 7};

    Solution obj;

    int ans = obj.minimumMultiplications(arr, start, end);

    cout << ans;
    cout << endl;

    return 0;
}

```

Output :

2

Time Complexity : $O(100000 * N)$

Where '100000' are the total possible numbers generated by multiplication (hypothetical) and N = size of the array with numbers of which each node could be multiplied.

Space Complexity : $O(100000 * N)$

Where '100000' are the total possible numbers generated by multiplication (hypothetical) and N = size of the array with numbers of which each node could be multiplied. $100000 * N$ is the max possible queue size. The space complexity of the dist array is constant.

Special thanks to Priyanshi Goel for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article. If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com



Watch Video At: https://youtu.be/_BvEJ3VIDWw

G-40: Number of Ways to Arrive at Destination

 takeuforward.org/data-structure/g-40-number-of-ways-to-arrive-at-destination

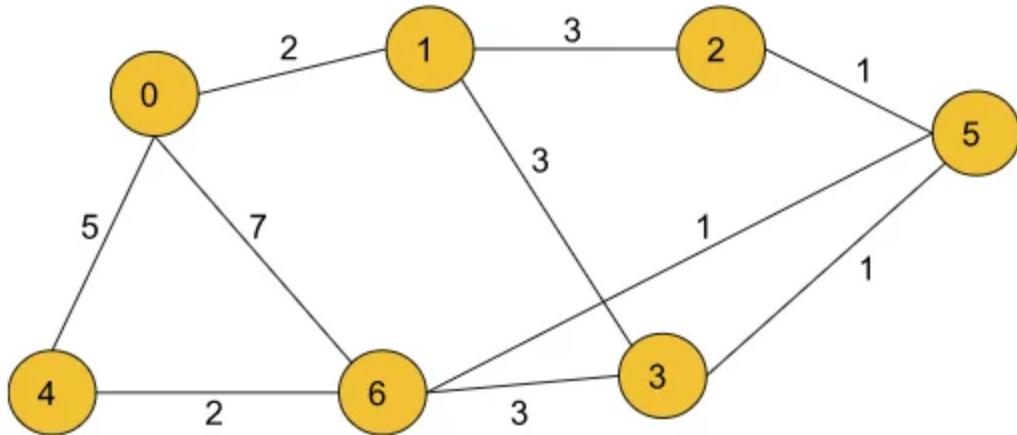
January 10, 2023

You are in a city that consists of n intersections numbered from 0 to $n - 1$ with **bi-directional** roads between some intersections. The inputs are generated such that you can reach any intersection from any other intersection and that there is at most one road between any two intersections.

You are given an integer n and a 2D integer array ‘roads’ where $\text{roads}[i] = [u_i, v_i, \text{time}_i]$ means that there is a road between intersections u_i and v_i that takes time_i minutes to travel. You want to know in how many ways you can travel from intersection 0 to intersection $n - 1$ in the **shortest amount of time**.

Return *the number of ways you can arrive at your destination in the shortest amount of time*. Since the answer may be large, return it modulo $10^9 + 7$.

Example 1:



Input:

$n=7, m=10$

edges= [[0,6,7],[0,1,2],[1,2,3],[1,3,3],[6,3,3],[3,5,1],[6,5,1],[2,5,1],[0,4,5],[4,6,2]]

Output:

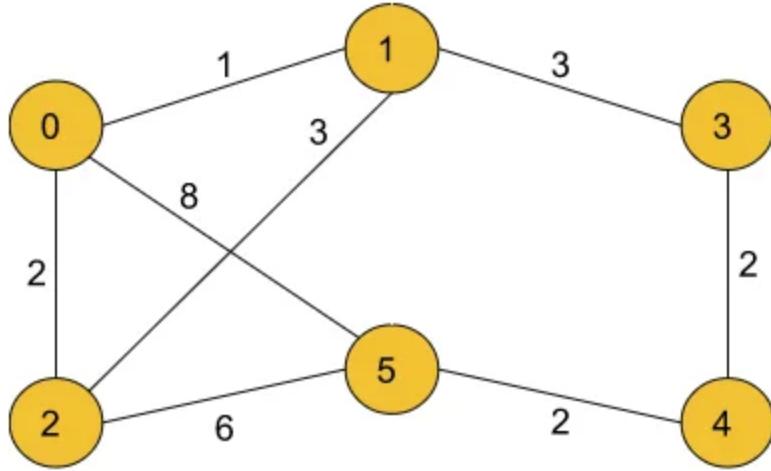
4

Explanation:

The four ways to get there in 7 minutes (which is the shortest calculated time) are:

- 0 6
- 0 4 6
- 0 1 2 5 6
- 0 1 3 5 6

Example 2:



Input:

n=6, m=8

edges= [[0,5,8],[0,2,2],[0,1,1],[1,3,3],[1,2,3],[2,5,6],[3,4,2],[4,5,2]]

Output:

3

Explanation:

The three ways to get there in 8 minutes (which is the shortest calculated time) are:

- 0 5
- 0 2 5
- 0 1 3 4 5

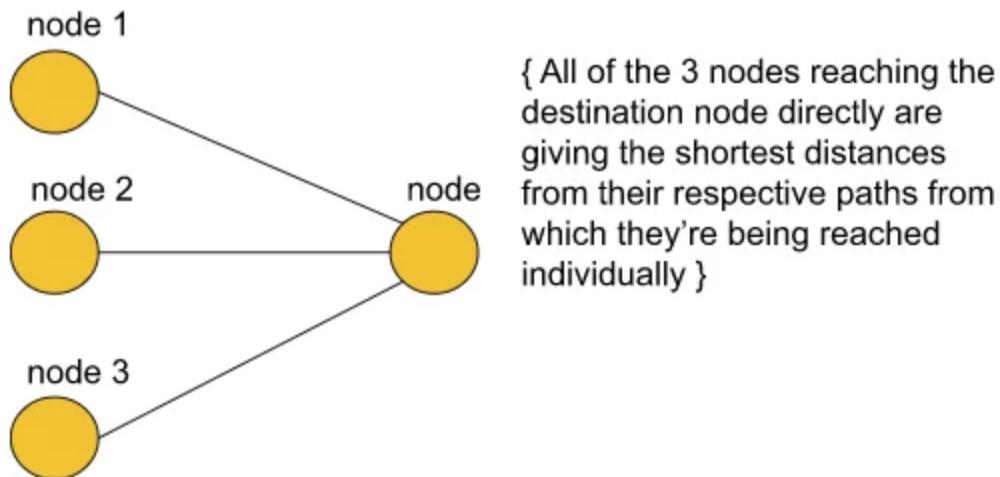
Solution

Disclaimer: Don't jump directly to the solution, try it out yourself first.

Problem Link

Note: In case any image/dry run is not clear please refer to the video attached at the bottom.

Intuition: Since there can be many paths to reach a destination node from the given source node, in this problem, we have to find all those paths that are the **shortest** in order to reach our destination. For an easier understanding of this particular problem, we can say that we can divide the problem into partitions such as illustrated below :



From the above picture, we may assume that there will be 3 shortest paths to the destination node. But that may not be the case every time. Let us understand how – We assume the total number of ways in which the destination node is reachable by the shortest possible distance be $\text{ways}[\text{node}]$ where ‘node’ depicts the destination node and node1, node2 and node3 are the three nodes which act as intermediate nodes that provide shortest paths to the destination. We can say :

$$\text{ways}[\text{node}] = \text{ways}[\text{node1}] + \text{ways}[\text{node2}] + \text{ways}[\text{node3}]$$

Where, $\text{ways}[\text{node1}]$, $\text{ways}[\text{node2}]$, and $\text{ways}[\text{node3}]$ are the number of shortest paths possible to node1, node2, and node3 respectively from the source node, the sum of which is the total possible shortest paths and that can be hence greater than 3.

Approach:

This problem is based on Dijkstra’s Algorithm where we count all the possible shortest paths from the source to the destination node.

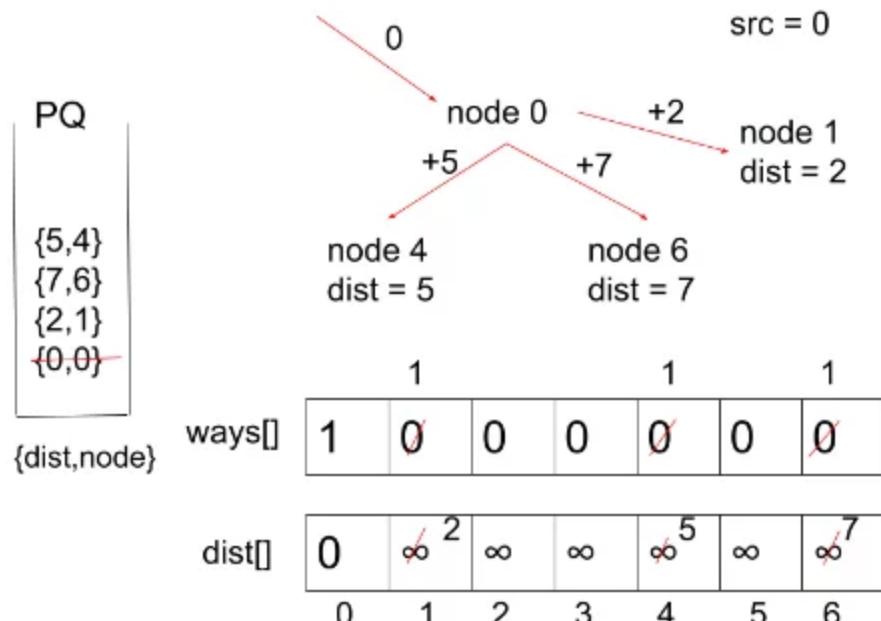
Initial configuration:

- **Priority Queue:** Define a Priority Queue which would contain pairs of the type {dist, node }, where ‘dist’ indicates the currently updated value of the shortest dist taken to reach from source to the current ‘node’.
- **Distance Array:** Define a distance array that would contain the minimum distance from the start node to the current node. If a cell is marked as ‘infinity’ then it is treated as unreachable/ unvisited.
- **Source Node:** Define the start node from where we have to calculate the total number of shortest paths.
- **Ways Array:** Define a ways array which would contain the number of possible shortest ways/paths for each node. Eventually, we would want to return $\text{ways}[n-1]$ where n= Number of nodes.

The Algorithm consists of the following steps :

- Start by creating an adjacency list, a priority queue that stores the dist-node pairs in the form {dist, node} and a dist array with each node initialized with a very large number (to indicate that the nodes have not been visited initially).
- In addition to the standard configuration of Dijkstra's algorithm, we have one more array in this problem by the name 'ways' which is initialized to '0' for every node when they're unvisited (so the number of ways is 0).
- Now, we push the start node to the queue along with its distance marked as '0' and ways marked as '1' initially because we've just started the algorithm.
- Pop the element from the front of the queue and look out for its adjacent nodes.
- If the current dist value for a number is better than the previous distance indicated by the distance array, we update the distance in the array and push it to the queue. Now, here side by side we also keep the number of ways to the 'node' the same as before.
- If the current dist value is the **same** as the previously stored dist value at the same index, increment the number of ways by 1 at that index.
- We repeat the above steps until the queue becomes empty or till we reach the destination.
- Return the ways[n-1] modulo 10^9+7 when the queue becomes empty.

Here's a quick demonstration of the Algorithm's 1st iteration for **example 1** stated above (all the further iterations would be done in a similar way) :



Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution
{
public:
    int countPaths(int n, vector<vector<int>> &roads)
    {
        // Creating an adjacency list for the given graph.
        vector<pair<int, int>> adj[n];
        for (auto it : roads)
        {
            adj[it[0]].push_back({it[1], it[2]});
            adj[it[1]].push_back({it[0], it[2]});
        }

        // Defining a priority queue (min heap).
        priority_queue<pair<int, int>,
                      vector<pair<int, int>>, greater<pair<int, int>>> pq;

        // Initializing the dist array and the ways array
        // along with their first indices.
        vector<int> dist(n, INT_MAX), ways(n, 0);
        dist[0] = 0;
        ways[0] = 1;
        pq.push({0, 0});

        // Define modulo value
        int mod = (int)(1e9 + 7);

        // Iterate through the graph with the help of priority queue
        // just as we do in Dijkstra's Algorithm.
        while (!pq.empty())
        {
            int dis = pq.top().first;
            int node = pq.top().second;
            pq.pop();

            for (auto it : adj[node])
            {
                int adjNode = it.first;
                int edW = it.second;

                // This 'if' condition signifies that this is the first
                // time we're coming with this short distance, so we push
                // in PQ and keep the no. of ways the same.
                if (dis + edW < dist[adjNode])
                {
                    dist[adjNode] = dis + edW;
                    pq.push({dis + edW, adjNode});
                    ways[adjNode] = ways[node];
                }
            }
        }
    }
};

```

```

        // If we again encounter a node with the same short distance
        // as before, we simply increment the no. of ways.
        else if (dis + edw == dist[adjNode])
        {
            ways[adjNode] = (ways[adjNode] + ways[node]) % mod;
        }
    }
    // Finally, we return the no. of ways to reach
    // (n-1)th node modulo 10^9+7.
    return ways[n - 1] % mod;
}
};

int main()
{
    // Driver Code.
    int n = 7;

    vector<vector<int>> edges = {{0, 6, 7}, {0, 1, 2}, {1, 2, 3}, {1, 3, 3}, {6, 3,
3},
{3, 5, 1}, {6, 5, 1}, {2, 5, 1}, {0, 4, 5}, {4, 6, 2}};

    Solution obj;

    int ans = obj.countPaths(n, edges);

    cout << ans;
    cout << endl;

    return 0;
}

```

Output :

4

Time Complexity: $O(E \log(V))$ { As we are using simple Dijkstra's algorithm here, the time complexity will be of the order $E \log(V)$ }

Where E = Number of edges and V = No. of vertices.

Space Complexity : $O(N)$ { for dist array + ways array + approximate complexity for priority queue }

Where, N = Number of nodes.

*Special thanks to **Priyanshi Goel** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, [please check out this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com*



Watch Video At: https://youtu.be/_-0mx0SmYxA

Bellman Ford Algorithm: G-41

 takeuforward.org/data-structure/bellman-ford-algorithm-g-41

November 23, 2022

Problem Statement: Given a weighted, directed and connected graph of V vertices and E edges, Find the shortest distance of all the vertices from the source vertex S.

Note: If the Graph contains a negative cycle then return an array consisting of only -1.

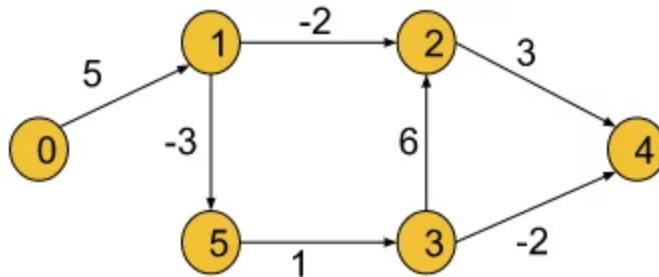
Example 1:

Input Format:

V = 6,

E = [[3, 2, 6], [5, 3, 1], [0, 1, 5], [1, 5, -3], [1, 2, -2], [3, 4, -2], [2, 4, 3]],

S = 0

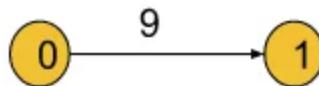


Result: 0 5 3 3 1 2

Explanation: Shortest distance of all nodes from the source node is returned.

Example 2:

Input Format: V = 2, E = [[0,1,9]], S = 0



Result: 0 9

Explanation: Shortest distance of all nodes from the source node is returned.

Solution

Disclaimer: Don't jump directly to the solution, try it out yourself first. [Problem link](#).

Solution:

The bellman-Ford algorithm helps to find the shortest distance from the source node to all other nodes. But, we have already learned **Dijkstra's algorithm** (Dijkstra's algorithm article link) to fulfill the same purpose. Now, the question is **how this algorithm is different from Dijkstra's algorithm**.

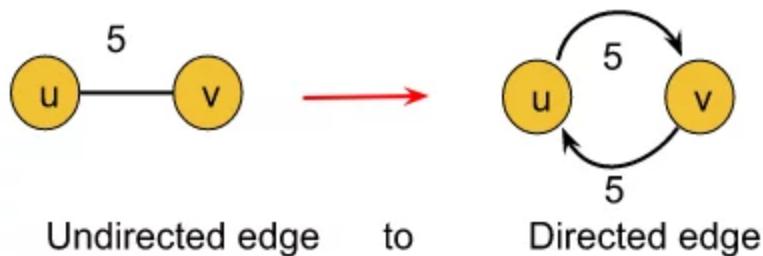
While learning Dijkstra's algorithm, we came across the following two situations, where Dijkstra's algorithm failed:

- **If the graph contains negative edges.**
- **If the graph has a negative cycle (In this case Dijkstra's algorithm fails to minimize the distance, keeps on running, and goes into an infinite loop. As a result it gives TLE error).**

Negative Cycle: A cycle is called a negative cycle if the sum of all its weights becomes negative. The following illustration is an example of a negative cycle:



Bellman-Ford's algorithm successfully solves these problems. **It works fine with negative edges** as well as **it is able to detect if the graph contains a negative cycle**. But this algorithm is only applicable for **directed graphs**. In order to apply this algorithm to an undirected graph, we just need to convert the undirected edges into directed edges like the following:



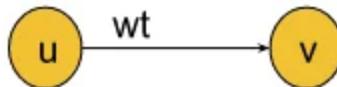
Explanation: An undirected edge between nodes u and v necessarily means that there are two opposite-directed edges, one towards node u and the other towards node v. So the above conversion is valid.

After converting the undirected graph into a directed graph following the above method, we can use the Bellman-Ford algorithm as it is.

Intuition:

In this algorithm, the edges can be given in any order. The intuition is to relax all the edges for $N-1$ ($N = \text{no. of nodes}$) times sequentially. After $N-1$ iterations, we should have minimized the distance to every node.

Let's understand what the relaxation of edges means using an example.

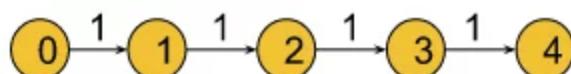


Let's consider the above graph with $\text{dist}[u]$, $\text{dist}[v]$, and wt . Here, wt is the weight of the edge and $\text{dist}[u]$ signifies the shortest distance to reach node u found until now. Similarly, $\text{dist}[v]$ (maybe infinite) signifies the shortest distance to reach node v found until now. If the distance to reach v through u (i.e. $\text{dist}[u] + \text{wt}$) is smaller than $\text{dist}[v]$, we will update the value of $\text{dist}[v]$ with $(\text{dist}[u] + \text{wt})$. This process of updating the distance is called the relaxation of edges.

We will apply the above process(i.e. minimizing the distance to reach every node) $N-1$ times in the Bellman-Ford algorithm.

Two follow-up questions about the algorithm: Why do we need exact $N-1$ iterations?

Let's try to first understand this using an example:



Given order of the edges:

Checking in each iteration	u	v	wt
$\text{dist}[3] + 1 < \text{dist}[4]$	3	4	1
$\text{dist}[2] + 1 < \text{dist}[3]$	2	3	1
$\text{dist}[1] + 1 < \text{dist}[2]$	1	2	1
$\text{dist}[0] + 1 < \text{dist}[1]$	0	1	1

- In the above graph, the algorithm will minimize the distance of the i^{th} node in the i^{th} iteration like $\text{dist}[1]$ will be updated in the 1st iteration, $\text{dist}[2]$ will be updated in the 2nd iteration, and so on. So we will need a total of 4 iterations(i.e. $N-1$ iterations) to minimize all the distances as $\text{dist}[0]$ is already set to 0.

Note: Points to remember since, in a graph of N nodes we will take at most $N-1$ edges to reach from the first to the last node, we need exact $N-1$ iterations. It is impossible to draw a graph that takes more than $N-1$ edges to reach any node.

- **How to detect a negative cycle in the graph?**

- We know if we keep on rotating inside a negative cycle, the path weight will be decreased in every iteration. But according to our intuition, we should have minimized all the distances within N-1 iterations(that means, after N-1 iterations no relaxation of edges is possible).
- In order to check the existence of a negative cycle, we will relax the edges one more time after the completion of N-1 iterations. And if in that Nth iteration, it is found that further relaxation of any edge is possible, we can conclude that the graph has a negative cycle. Thus, the Bellman-Ford algorithm detects negative cycles.

Approach:

Initial Configuration:

distance array(dist[]): The dist[] array will be initialized with infinity, except for the source node as dist[src] will be initialized to 0.

The algorithm steps will be the following:

1. First, we will initialize the source node in the distance array to 0 and the rest of the nodes to infinity.
2. Then we will run a loop for N-1 times.
3. Inside that loop, we will try to relax every given edge.
For example, one of the given edge information is like (u, v, wt), where u = starting node of the edge, v = ending node, and wt = edge weight. For all edges like this we will be checking if node u is reachable and if the distance to reach v through u is less than the distance to v found until now(i.e. **dist[u] and dist[u]+ wt < dist[v]**).
4. After repeating the 3rd step for N-1 times, we will apply the same step one more time to check if the negative cycle exists. If we found further relaxation is possible, we will conclude the graph has a negative cycle and from this step, we will return a distance array of -1(i.e. minimization of distances is not possible).
5. Otherwise, we will return the distance array which contains all the minimized distances.

Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
    /* Function to implement Bellman Ford
     * edges: vector of vectors which represents the graph
     * S: source vertex to start traversing graph with
     * V: number of vertices
     */
    vector<int> bellman_ford(int V, vector<vector<int>>& edges, int S) {
        vector<int> dist(V, 1e8);
        dist[S] = 0;
        for (int i = 0; i < V - 1; i++) {
            for (auto it : edges) {
                int u = it[0];
                int v = it[1];
                int wt = it[2];
                if (dist[u] != 1e8 && dist[u] + wt < dist[v]) {
                    dist[v] = dist[u] + wt;
                }
            }
        }
        // Nth relaxation to check negative cycle
        for (auto it : edges) {
            int u = it[0];
            int v = it[1];
            int wt = it[2];
            if (dist[u] != 1e8 && dist[u] + wt < dist[v]) {
                return {-1};
            }
        }
    }

    return dist;
}
};

int main() {

    int V = 6;
    vector<vector<int>> edges(7, vector<int>(3));
    edges[0] = {3, 2, 6};
    edges[1] = {5, 3, 1};
    edges[2] = {0, 1, 5};
    edges[3] = {1, 5, -3};
    edges[4] = {1, 2, -2};
    edges[5] = {3, 4, -2};
    edges[6] = {2, 4, 3};

    int S = 0;
}

```

```
Solution obj;
vector<int> dist = obj.bellman_ford(V, edges, S);
for (auto d : dist) {
    cout << d << " ";
}
cout << endl;

return 0;
}
```

Output: 0 5 3 3 1 2

Time Complexity: O(V*E), where V = no. of vertices and E = no. of Edges.

Space Complexity: O(V) for the distance array which stores the minimized distances.

*Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, [please check out this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com*



Watch Video At: <https://youtu.be/0vVofAhAYjc>

Floyd Warshall Algorithm: G-42

 takeuforward.org/data-structure/floyd-warshall-algorithm-g-42

November 23, 2022

Problem Statement: The problem is to find the shortest distances between every pair of vertices in a given edge-weighted directed graph. The graph is represented as an adjacency matrix of size $n \times n$. Matrix[i][j] denotes the weight of the edge from i to j. If Matrix[i][j]=-1, it means there is no edge from i to j.

Do it in place.

Example 1:

Input Format:

```
matrix[][] = { {0, 2, -1, -1},  
              {1, 0, 3, -1}, {-1, -1, 0, -1}, {3, 5, 4, 0} }
```

Result:

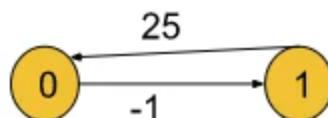
```
0 2 5 -1  
1 0 3 -1  
-1 -1 0 -1  
3 5 4 0
```

Explanation: In this example, the final matrix is storing the shortest distances. For example, matrix[i][j] is storing the shortest distance from node i to j.

Example 2:

Input Format:

```
matrix[][] = {{0, 25},  
              {-1, 0}}
```



Result:

```
0 25  
-1 0
```

Explanation: In this example, the shortest distance is already given (if it exists).

Solution

Disclaimer: Don't jump directly to the solution, try it out yourself first. [Problem link](#).

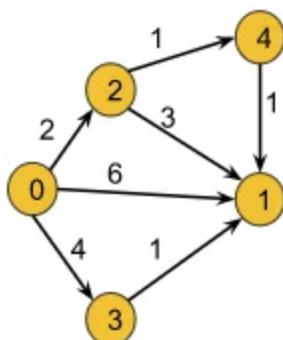
Solution:

In this article, we will be discussing Floyd Warshall Algorithm which is very much different from the two which we have previously learned: Dijkstra's Shortest Path algorithm and the Bellman-Ford algorithm.

Dijkstra's Shortest Path algorithm and Bellman-Ford algorithm are **single-source shortest path algorithms** where we are given a single source node and are asked to find out the shortest path to every other node from that given source. But in the Floyd Warshall algorithm, we need to figure out the shortest distance from each node to every other node.

Basically, the Floyd Warshall algorithm is a **multi-source shortest path algorithm** and it **helps to detect negative cycles as well**. The **shortest path** between node u and v necessarily means the path(from u to v) for which the sum of the edge weights is minimum.

In Floyd Warshall's algorithm, we need to check every possible path going via each possible node. And after checking every possible path, we will figure out the shortest path(a kind of brute force approach to find the shortest path). Let's understand it using the following illustration:



Here we are finding the shortest path from i to j going via every possible k.

For example consider the pair of nodes 0 and 1:
So to find the shortest distance between 0 and 1 we will check the following paths:
 $(0 \rightarrow 1) = 6$
 $(0 \rightarrow 2) + (2 \rightarrow 1) = 5$
 $(0 \rightarrow 3) + (3 \rightarrow 1) = 5$
 $(0 \rightarrow 4) + (4 \rightarrow 1) = 4$
 \downarrow
 $(0 \rightarrow 2) + (2 \rightarrow 4) = 3$ (path to reach from 0 to 4)
Therefore, $\text{dist}[0][1] = 4$ (as 4 is the minimum cost)

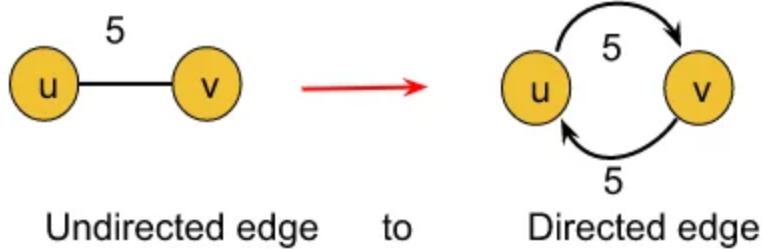
Here we can consider source node as i, destination node as j and the nodes via which we are reaching from i to j as k. Then the value of k varies from 0 to 4 in this case.

From the above example we can derive the following formula:

$\text{matrix}[i][j] = \min(\text{matrix}[i][j], \text{matrix}[i][k] + \text{matrix}[k][j])$, where i = source node, j = destination node, and k = the node via which we are reaching from i to j.

Here we will calculate $\text{dist}[i][j]$ for every possible node k ($k = 0, 1, \dots, V$, where $V = \text{no. of nodes}$), and will select the minimum value as our result.

In order to apply this algorithm to an undirected graph, we just need to convert the undirected edges into directed edges like the following:



Explanation: An undirected edge between nodes u and v necessarily means that there are two opposite-directed edges, one towards node u and the other towards node v. So the above conversion is valid.

Note:

- Until now, to store a graph we have used the adjacency list. But in this algorithm, we are going to use the **adjacency matrix** method.
- One additional point to remember is that the cost of reaching a node from itself must always be 0 i.e. $dist[i][i] = 0$, where i = current node.

Intuition:

The intuition is to check all possible paths between every possible pair of nodes and to choose the shortest one. Checking all possible paths means going via each and every possible node.

The follow-up questions for interviews:

How to detect a negative cycle using the Floyd Warshall algorithm?

Negative Cycle: A cycle is called a negative cycle if the sum of all its weights becomes negative. The following illustration is an example of a negative cycle:



- We have previously said that the cost of reaching a node from itself must be 0. But in the above graph, if we try to reach node 0 from itself we can follow the path: $0 \rightarrow 1 \rightarrow 2 \rightarrow 0$. In this case, the cost to reach node 0 from itself becomes -3 which is less than 0. This is only possible if the graph contains a negative cycle.
- So, if we find that the cost of reaching any node from itself is less than 0, we can conclude that the graph has a negative cycle.

- **What will happen if we will apply Dijkstra's algorithm for this purpose?**
 - **If the graph has a negative cycle:** We cannot apply Dijkstra's algorithm to the graph which contains a negative cycle. It will give TLE error in that case.
 - **If the graph does not contain a negative cycle:** In this case, we will apply Dijkstra's algorithm for every possible node to make it work like a multi-source shortest path algorithm like Floyd Warshall. The time complexity of **Floyd Warshall** is $O(V^3)$ (Which we will discuss later in this article) whereas if we apply **Dijkstra's algorithm** for the same purpose the time complexity reduces to $O(V^*(E \log V))$ (where v = no. of vertices).

Approach:

The algorithm is not much intuitive as the other ones'. It is more of a brute force, where all combinations of paths have been tried to get the shortest paths. Nothing to panic much with the intuition, it is a simple brute force approach on all paths. Focus on the three 'for' loops.

Formula:

matrix[i][j] =min(matrix[i][j], matrix[i][k]+matrix[k][j]), where i = source node, j = destination node and k = the node via which we are reaching from i to j.

The algorithm steps are as follows:

Initial Configuration:

Adjacency Matrix: The adjacency matrix should store the edge weights for the given edges and the rest of the cells must be initialized with infinity().

1. After having set the adjacency matrix accordingly, we will run a loop from 0 to $V-1$ (V = no. of vertices). In the k^{th} iteration, this loop will help us to check the path via node k for every possible pair of nodes. Basically, this loop will change the value of k in the formula.
2. Inside the loop, there will be two nested loops for generating every possible pair of nodes (Nothing but to visit each cell of a 2D matrix using the nested loop). Among these two loops, the first loop will change the value of i and the second one will change the value of j in the formula.
3. Inside these nested loops, we will apply the above formula to calculate the shortest distance between the pair of nodes.
4. Finally, the adjacency matrix will store all the shortest paths. For example, $\text{matrix}[i][j]$ will store the shortest path from node i to node j .

If we want to check for a negative cycle:

After completing the steps(outside those three loops), we will run a loop and check if any cell having the row and column the same($i = j$) contains a value less than 0.

Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
    void shortest_distance(vector<vector<int>>&matrix) {
        int n = matrix.size();
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (matrix[i][j] == -1) {
                    matrix[i][j] = 1e9;
                }
                if (i == j) matrix[i][j] = 0;
            }
        }

        for (int k = 0; k < n; k++) {
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    matrix[i][j] = min(matrix[i][j],
                                        matrix[i][k] + matrix[k]
[j]);
                }
            }
        }

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (matrix[i][j] == 1e9) {
                    matrix[i][j] = -1;
                }
            }
        }
    }
};

int main() {

    int V = 4;
    vector<vector<int>> matrix(V, vector<int>(V, -1));
    matrix[0][1] = 2;
    matrix[1][0] = 1;
    matrix[1][2] = 3;
    matrix[3][0] = 3;
    matrix[3][1] = 5;
    matrix[3][2] = 4;
}

```

```

Solution obj;
obj.shortest_distance(matrix);

for (auto row : matrix) {
    for (auto cell : row) {
        cout << cell << " ";
    }
    cout << endl;
}

return 0;
}

```

Output:

0 2 5 -1
 1 0 3 -1
 -1 -1 0 -1
 3 5 4 0

Time Complexity: $O(V^3)$, as we have three nested loops each running for V times, where V = no. of vertices.

Space Complexity: $O(V^2)$, where V = no. of vertices. This space complexity is due to storing the adjacency matrix of the given graph.

*Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, [please check out this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com*



Watch Video At: <https://youtu.be/YbY8cVwWAvw>

Find the City With the Smallest Number of Neighbours at a Threshold Distance: G-43

 takeuforward.org/data-structure/find-the-city-with-the-smallest-number-of-neighbours-at-a-threshold-distance-g-43

November 23, 2022



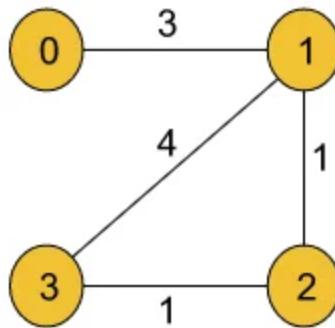
Problem Statement: There are n cities numbered from 0 to $n-1$. Given the array edges where $\text{edges}[i] = [\text{from}_i, \text{to}_i, \text{weight}_i]$ represents a bidirectional and weighted edge between cities from_i and to_i , and given the integer distance Threshold. You need to find out a city with the smallest number of cities that are reachable through some path and whose distance is at most Threshold Distance, If there are multiple such cities, our answer will be the city with the greatest number.

Note: that the distance of a path, connecting cities i and j are equal to the sum of the edges' weights along that path.

Example 1:

Input Format:

N=4, M=4,
edges = [[0,1,3],[1,2,1],[1,3,4],[2,3,1]],
distanceThreshold = 4

**Result:** 3**Explanation:**

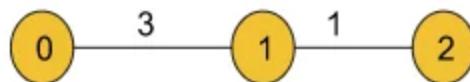
The adjacent cities for each city at a distanceThreshold are =

City 0 →[City 1, City 2]
City 1 →[City 0, City 2, City 3]
City 2 →[City 0, City 1, City 3]
City 3 →[City 1, City 2]

Here, City 0 and City 3 have a minimum number of cities i.e. 2 within distanceThreshold. So, the result will be the city with the largest number. So, the answer is City 3.

Example 2:**Input Format:**

N=3, M=2, edges = [[0,1,3],[1,2,1]], distanceThreshold = 2

**Result:** 2**Explanation:**

City 1 → City 2,
City 2 → City 1
Hence, 2 is the answer.

Solution

Disclaimer: Don't jump directly to the solution, try it out yourself first.

In order to solve this problem, we will use the **Floyd Warshall algorithm** (article link of Floyd Warshall). We know Floyd Warshall algorithm helps us to generate a 2D matrix, that stores the shortest distances from each node to every other node. In the generated 2D matrix, each

cell matrix[i][j] represents the shortest distance from node i to node j.

After generating the 2D matrix(that contains the shortest paths) using the Floyd Warshall algorithm, for each node, we will count the number of nodes with a distance lesser or equal to the distanceThreshold by iterating each row of that matrix. Finally, we will choose the node with the minimum number of adjacent cities(whose distance is at the most distanceThreshold) and with the largest value.

Note: This 2D matrix can also be generated using Dijkstra's algorithm. As Dijkstra's algorithm is a single-source shortest-path algorithm, we need to calculate the shortest distances for one single node at a time. So, to create the 2D matrix we need to apply Dijkstra's algorithm to each of the V nodes separately.

Intuition:

*For each node, the job is to find the shortest distances to every other node and count the number of adjacent cities(Let's say: **cntCity**) whose distance is at the most distanceThreshold. Finally, the task is to choose the node with the largest value and the minimum '**cntCity**' value.*

Approach:

Initial Configuration:

Adjacency Matrix(dist): All the cells of the matrix are initially set to infinity(∞).

cntCity: Initially set to V(no. of nodes) i.e. the maximum value possible. It will store the minimum number of cities whose distance is at most distanceThreshold.

cityNo: Initially set to -1. It will store the answer i.e. the node with the largest value and the minimum '**cntCity**' value.

The algorithm steps are as follows:

1. First, we will iterate over the edges, and set the value of $dist[from_i][to_i]$ and $dist[to_i][from_i]$ to weight_i as the edges are bidirectional.
2. After having set the adjacency matrix accordingly, we will run a loop from 0 to V-1(V = no. of vertices). In the kth iteration, this loop will help us to check the path via node k for every possible pair of nodes. Basically, this loop will change the value of k in the formula(given in step 4).
3. Inside the loop, there will be two nested loops for generating every possible pair of nodes(Nothing but to visit each cell of a 2D matrix using the nested loop). Among these two loops, the first loop will change the value of i and the second one will change the value of j in the formula(given in step 4).

4. Inside these nested loops, we will apply the following formula to calculate the shortest distance between the pair of nodes: **$\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k]+\text{dist}[k][j])$** , where **i = source node, j = destination node, and k = the node via which we are reaching from i to j.**
5. The adjacency matrix will store all the shortest paths for each node. For example, $\text{dist}[i][j]$ will store the shortest path from node i to node j.
6. After that, we will count the nodes(**cnt**) with a distance lesser or equal to **distanceThreshold** and check if it is lesser than the current value of **cntCity**.
7. If it is lesser, we will update **cntCity** with the count of nodes and **cityNo** with the value of the current city.
8. Finally, we will return **cityNo** as our answer.

Note: *If we need to find out the number of adjacent cities as well, we need to return cntCity-1. This is because we have included the node itself in the group of adjacent nodes whose distance is at the most distanceThreshold.* **Note:** *If you wish to see the dry run of the above approach, you can watch the video attached to this article.*

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
public:
    int findCity(int n, int m, vector<vector<int>>& edges,
                int distanceThreshold) {
        vector<vector<int>> dist(n, vector<int> (n, INT_MAX));
        for (auto it : edges) {
            dist[it[0]][it[1]] = it[2];
            dist[it[1]][it[0]] = it[2];
        }
        for (int i = 0; i < n; i++) dist[i][i] = 0;
        for (int k = 0; k < n; k++) {
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    if (dist[i][k] == INT_MAX || dist[k][j] == INT_MAX)
                        continue;
                    dist[i][j] = min(dist[i][j], dist[i][k] +
dist[k][j]);
                }
            }
        }
        int cntCity = n;
        int cityNo = -1;
        for (int city = 0; city < n; city++) {
            int cnt = 0;
            for (int adjCity = 0; adjCity < n; adjCity++) {
                if (dist[city][adjCity] <= distanceThreshold)
                    cnt++;
            }

            if (cnt <= cntCity) {
                cntCity = cnt;
                cityNo = city;
            }
        }
        return cityNo;
    }
};

int main() {

    int n = 4;
    int m = 4;
    vector<vector<int>> edges = {{0, 1, 3}, {1, 2, 1}, {1, 3, 4}, {2, 3, 1}};
    int distanceThreshold = 4;
}

```

```
Solution obj;
int cityNo = obj.findCity(n, m, edges, distanceThreshold);
cout << "The answer is node: " << cityNo << endl;

return 0;
}
```

Output: The answer is node: 3

Time Complexity: $O(V^3)$, as we have three nested loops each running for V times, where V = no. of vertices.

Space Complexity: $O(V^2)$, where V = no. of vertices. This space complexity is due to storing the adjacency matrix of the given graph.

*Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, [please check out this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com*



Watch Video At: <https://youtu.be/PwMVNSJ5SLI>

Minimum Spanning Tree – Theory: G-44

 takeuforward.org/data-structure/minimum-spanning-tree-theory-g-44

November 23, 2022

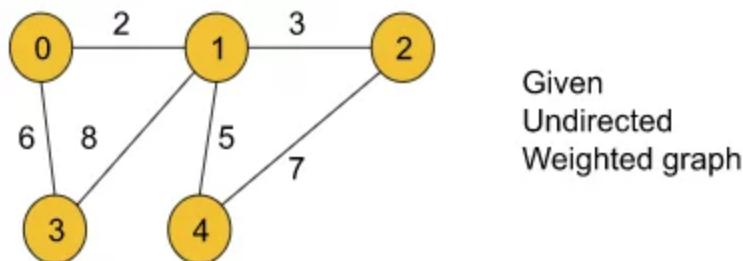
In this article, we will be discussing the **minimum spanning tree**. So, to understand the minimum spanning tree, we first need to discuss **what a spanning tree is**.

Let's further discuss this below:

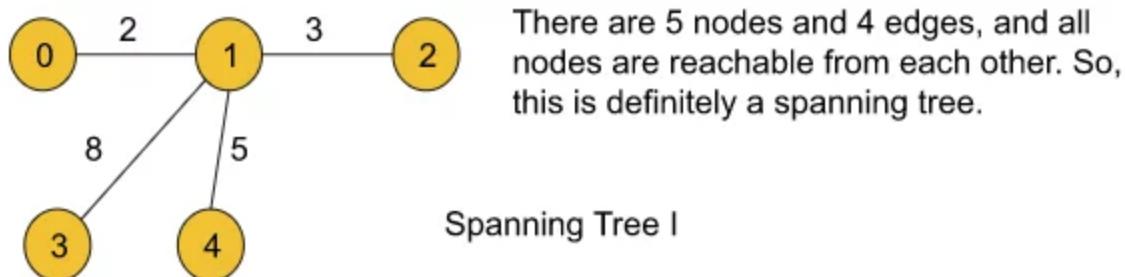
Spanning Tree:

A spanning tree is a tree in which we have N nodes(i.e. All the nodes present in the original graph) and $N-1$ edges and all nodes are reachable from each other.

Let's understand this using an example. Assume we are given an undirected weighted graph with N nodes and M edges. Here in this example, we have taken N as 5 and M as 6.

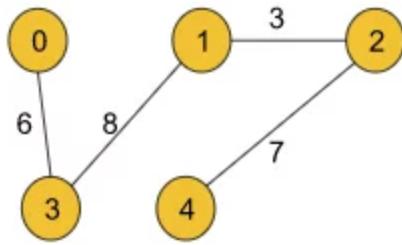


For the above graph, if we try to draw a spanning tree, the following illustration will be one:



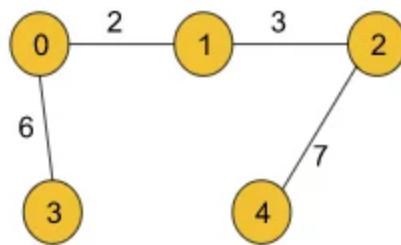
Sum of edge weights = 18

We can draw more spanning trees for the given graph. Two of them are like the following:



Spanning Tree II

Sum of edge weights = 24



Spanning Tree III

Sum of edge weights = 18

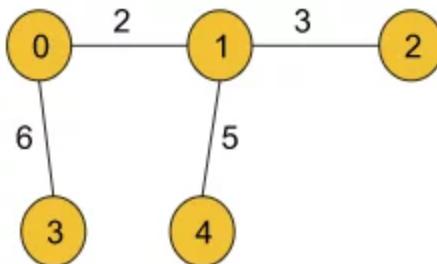
Note: Point to remember is that a graph may have more than one spanning trees.

All the above spanning trees contain some edge weights. For each of them, if we add the edge weights we can get the sum for that particular tree. Now, let's try to figure out the minimum spanning tree:

Minimum Spanning Tree:

Among all possible spanning trees of a graph, the minimum spanning tree is the one for which the sum of all the edge weights is the minimum.

Let's understand the definition using the given graph drawn above. Until now, for the given graph we have drawn three spanning trees with the sum of edge weights 18, 24, and 18. If we can draw all possible spanning trees, we will find that the following spanning tree with the minimum sum of edge weights 16 is the **minimum spanning tree** for the given graph:

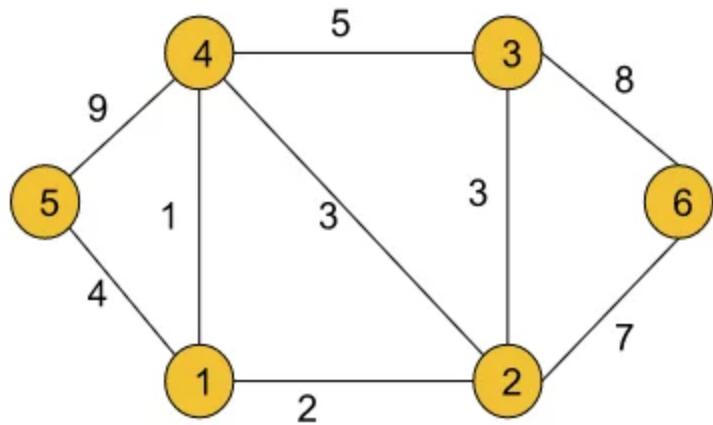


Minimum spanning tree
Sum of edge weights = 16

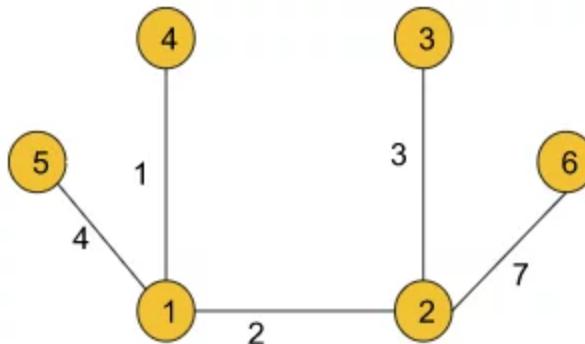
Note: There may exist multiple minimum spanning trees for a graph like a graph may have multiple spanning trees.

Practice Problem:

Now, in case you have understood the concepts well, you can try to figure out the minimum spanning tree for the following undirected weighted graph:



Answer:



Sum of edge weights = 17

There are a couple of algorithms that help us to find the minimum spanning tree of a graph. One is **Prim's algorithm** and the other is **Kruskal's algorithm**. We will be discussing all of them in the upcoming articles.

Note: If you wish to visualize the above concepts, you can watch the video attached to this article.

Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, [please check out this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com



Watch Video At: <https://youtu.be/ZSPjZuZWCME>

Prim's Algorithm – Minimum Spanning Tree – C++ and Java: G-45

 takeuforward.org/data-structure/prims-algorithm-minimum-spanning-tree-c-and-java-g-45

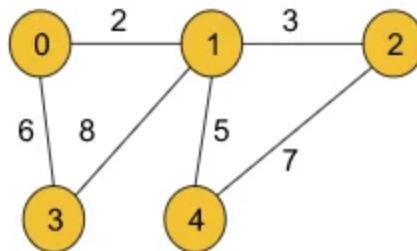
December 15, 2022

Problem Statement: Given a weighted, undirected, and connected graph of V vertices and E edges. The task is to find the sum of weights of the edges of the Minimum Spanning Tree. (*Sometimes it may be asked to find the MST as well, where in the MST the edge-information will be stored in the form {u, v}(u = starting node, v = ending node).*)

Example 1:

Input Format:

V = 5, edges = { {0, 1, 2}, {0, 3, 6}, {1, 2, 3}, {1, 3, 8}, {1, 4, 5}, {4, 2, 7} }

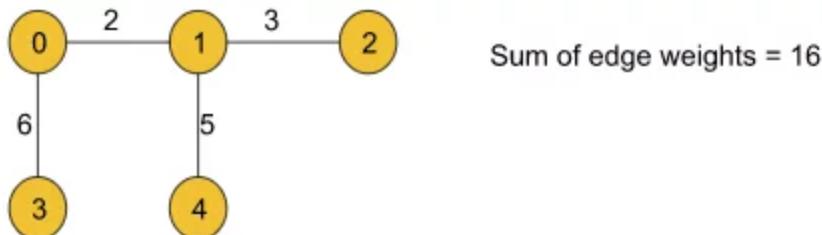


Result: 16

Explanation:

The minimum spanning tree for the given graph is drawn below:

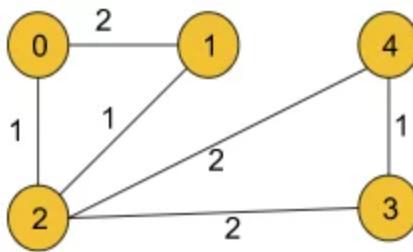
MST = {(0, 1), (0, 3), (1, 2), (1, 4)}



Example 2:

Input Format:

$V = 5$, edges = { {0, 1, 2}, {0, 2, 1}, {1, 2, 1}, {2, 3, 2}, {3, 4, 1}, {4, 2, 2} }

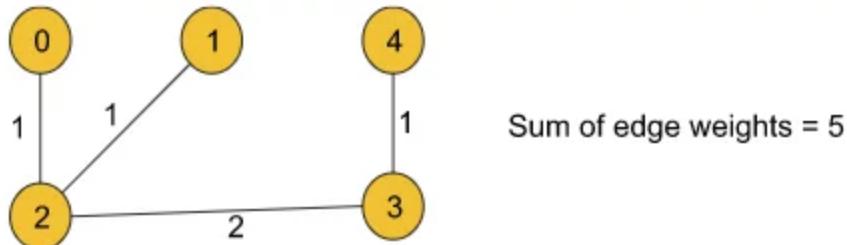


Result: 5

Explanation:

The minimum spanning tree is drawn below:

$$MST = \{(0, 2), (1, 2), (2, 3), (3, 4)\}$$



Solution

Disclaimer: Don't jump directly to the solution, try it out yourself first.

[Problem Link.](#)

In our previous article, we discussed **what a minimum spanning tree is**. Now it's time to discuss **Prim's algorithm** which will help us to find the minimum spanning tree for a given graph.

Approach:

In order to implement Prim's algorithm, we will be requiring an array(**visited array**) and a priority queue that will essentially represent a **min-heap**. We need another array(**MST**) as well if we wish to store the edge information of the minimum spanning tree.

The algorithm steps are as follows:

Priority Queue(Min Heap): The priority queue will be storing the pairs (edge weight, node). We can start from any given node. Here we are going to start from node 0 and so we will initialize the priority queue with (0, 0). If we wish to store the mst of the graph, the priority queue should instead store the triplets (edge weight, adjacent node, parent node) and in that case, we will initialize with (0, 0, -1).

Visited array: All the nodes will be initially marked as unvisited.

sum variable: It will be initialized with 0 and we wish that it will store the sum of the edge weights finally.

MST array(optional): If we wish to store the minimum spanning tree(MST) of the graph, we need this array. This will store the edge information as a pair of starting and ending nodes of a particular edge.

1. We will first push edge weight 0, node value 0, and parent -1 as a triplet into the priority queue to start the algorithm.

Note: *We can start from any node of our choice. Here we have chosen node 0.*

2. Then the top-most element (element with minimum edge weight as it is the min-heap we are using) of the priority queue is popped out.
3. After that, we will check whether the popped-out node is visited or not.

If the node is visited: We will continue to the next element of the priority queue.

If the node is not visited: We will mark the node visited in the **visited array** and add the edge weight to the sum variable. If we wish to store the mst, we should insert the parent node and the current node into the mst array as a pair in this step.

4. Now, we will iterate on all the unvisited adjacent nodes of the current node and will store each of their information in the specified triplet format i.e. (edge weight, node value, and parent node) in the priority queue.
5. We will repeat steps 2, 3, and 4 using a loop until the priority queue becomes empty.
6. Finally, the sum variable should store the sum of all the edge weights of the minimum spanning tree.

Note: *Points to remember if we do not wish to store the mst(minimum spanning tree) for the graph and are only concerned about the sum of all the edge weights of the minimum spanning tree:*

- *First of all, we will not use the triplet format instead, we will just use the pair in the format of (edge weight, node value). Basically, we do not need the parent node.*
- *In step 3, we need not store anything in the mst array and we need not even use the mst array in our whole algorithm as well.*

Intuition:

The intuition of this algorithm is the greedy technique used for every node. If we carefully observe, for every node, we are greedily selecting its unvisited adjacent node with the minimum edge weight(as the priority queue here is a min-heap and the topmost element is the node with the minimum edge weight). Doing so for every node, we can get the sum of all the edge weights of the minimum spanning tree and the spanning tree itself(if we wish to) as well.

Note: *If you wish to see the dry run of the above approach, you can watch the video attached to this article.*

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution
{
public:
    //Function to find sum of weights of edges of the Minimum Spanning Tree.
    int spanningTree(int V, vector<vector<int>> adj[])
    {
        priority_queue<pair<int, int>,
                      vector<pair<int, int> >, greater<pair<int, int>>> pq;

        vector<int> vis(V, 0);
        // {wt, node}
        pq.push({0, 0});
        int sum = 0;
        while (!pq.empty()) {
            auto it = pq.top();
            pq.pop();
            int node = it.second;
            int wt = it.first;

            if (vis[node] == 1) continue;
            // add it to the mst
            vis[node] = 1;
            sum += wt;
            for (auto it : adj[node]) {
                int adjNode = it[0];
                int edW = it[1];
                if (!vis[adjNode]) {
                    pq.push({edW, adjNode});
                }
            }
        }
        return sum;
    }
};

int main() {

    int V = 5;
    vector<vector<int>> edges = {{0, 1, 2}, {0, 2, 1}, {1, 2, 1}, {2, 3, 2}, {3, 4, 1}, {4, 2, 2}};
    vector<vector<int>> adj[V];
    for (auto it : edges) {
        vector<int> tmp(2);
        tmp[0] = it[1];
        tmp[1] = it[2];
        adj[it[0]].push_back(tmp);

        tmp[0] = it[0];

```

```

        tmp[1] = it[2];
        adj[it[1]].push_back(tmp);
    }

    Solution obj;
    int sum = obj.spanningTree(V, adj);
    cout << "The sum of all the edge weights: " << sum << endl;

    return 0;
}

```

Output: The sum of all the edge weights: 5

Time Complexity: $O(E \log E) + O(E \log E) \sim O(E \log E)$, where E = no. of given edges.

The maximum size of the priority queue can be E so after at most E iterations the priority queue will be empty and the loop will end. Inside the loop, there is a pop operation that will take $\log E$ time. This will result in the first $O(E \log E)$ time complexity. Now, inside that loop, for every node, we need to traverse all its adjacent nodes where the number of nodes can be at most E . If we find any node unvisited, we will perform a push operation and for that, we need a $\log E$ time complexity. So this will result in the second $O(E \log E)$.

Space Complexity: $O(E) + O(V)$, where E = no. of edges and V = no. of vertices. $O(E)$ occurs due to the size of the priority queue and $O(V)$ due to the visited array. If we wish to get the mst, we need an extra $O(V-1)$ space to store the edges of the mst.

*Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward.*

If you also wish to share your knowledge with the takeUforward fam, [please check out this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com



Watch Video At: <https://youtu.be/mJcZjjKzeqk>

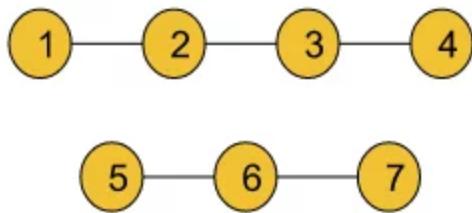
Disjoint Set | Union by Rank | Union by Size | Path Compression: G-46

 takeuforward.org/data-structure/disjoint-set-union-by-rank-union-by-size-path-compression-g-46

December 15, 2022

In this article, we will discuss the **Disjoint Set** data structure which is a very important topic in the entire graph series. Let's first understand **why we need a Disjoint Set data structure using the below question:**

Question: Given two components of an undirected graph



The question is whether node 1 and node 5 are in the same component or not.

Approach:

Now, in order to solve this question we can use either the DFS or BFS traversal technique like if we traverse the components of the graph we can find that node 1 and node 5 are not in the same component. This is actually the **brute force** approach whose time complexity is $O(N+E)$ ($N = \text{no. of nodes}$, $E = \text{no. of edges}$). But **using a Disjoint Set data structure we can solve this same problem in constant time.**

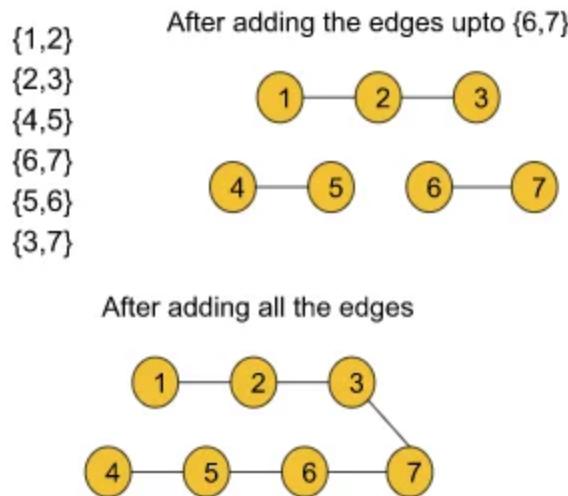
The disjoint Set data structure is generally used for **dynamic graphs**.

Dynamic graph:

A dynamic graph generally refers to a graph that keeps on changing its configuration. Let's deep dive into it using an example:

- Let's consider the edge information for the given graph as: $\{\{1,2\}, \{2,3\}, \{4,5\}, \{6,7\}, \{5,6\}, \{3,7\}\}$. Now if we start adding the edges one by one, in each step the structure of the graph will change. So, after each step, if we perform the same operation on the graph while updating the edges, the result might be different. In this case, the graph will be considered a dynamic graph.

- For example, after adding the first 4 edges if we look at the graph, we will find that node 4 and node 1 belong to different components but after adding all 6 edges if we search for the same we will figure out that node 4 and node 1 belong to the same component.



So, **after any step, if we try to figure out whether two arbitrary nodes u and v belong to the same component or not, Disjoint Set will be able to answer this query in constant time.**

Functionalities of Disjoint Set data structure:

The disjoint set data structure generally provides two types of functionalities:

- Finding the parent for a particular node (***findPar()***)
- Union (in broad terms this method basically adds an edge between two nodes)
 - Union by rank
 - Union by size

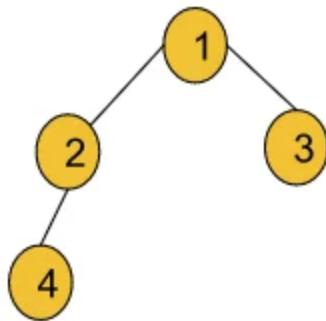
First, we will be discussing Union by rank and then Union by size.

Union by rank:

Before discussing Union by rank we need to discuss some terminologies:

Rank:

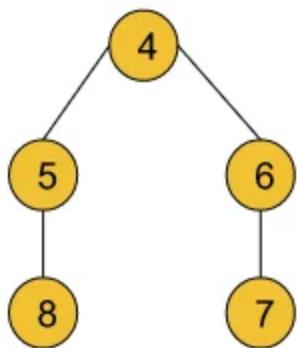
The rank of a node generally refers to the distance (the number of nodes including the leaf node) between the furthest leaf node and the current node. Basically rank includes all the nodes beneath the current node.



Here the rank of node 1 is 2 as the distance between node 1 and the furthest leaf node 4 is 2.

Ultimate parent:

The parent of a node generally refers to the node right above that particular node. But the ultimate parent refers to the topmost node or the root node.



In this graph, the parent of 8 is 5 but the ultimate parent of 8 is 4

Now let's discuss the implementation of the union by rank function. In order to implement Union by rank, we basically need two arrays of size N(no. of nodes). One is the **rank** and the other one is the **parent**. The rank array basically stores the rank of each node and the parent array stores the ultimate parent for each node.

Algorithm:

Initial configuration:

rank array: This array is initialized with zero.

parent array: The array is initialized with the value of nodes i.e. $\text{parent}[i] = i$.

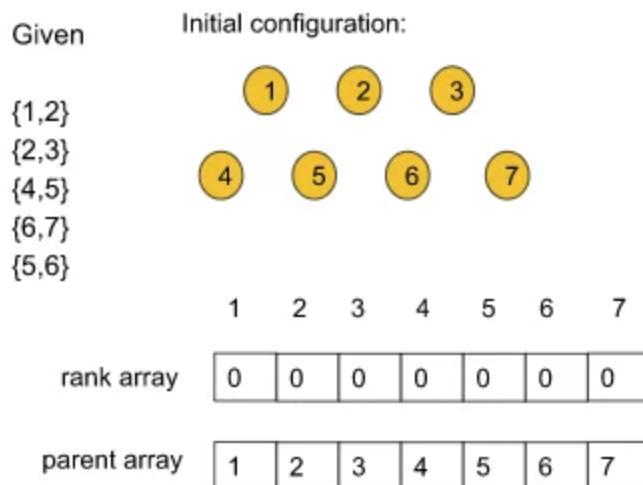
The algorithm steps are as follows:

1. Firstly, the Union function requires two nodes(*let's say u and v*) as arguments. Then we will find the ultimate parent (using the `findPar()` function that is discussed later) of u and v. Let's consider the ultimate parent of u is **pu** and the ultimate parent of v is **pv**.
2. After that, we will find the rank of **pu** and **pv**.

3. Finally, we will connect the ultimate parent with a smaller rank to the other ultimate parent with a larger rank. But if the ranks are equal, we can connect any parent to the other parent and we will increase the rank by one for the parent node to whom we have connected the other one.

Let's understand it further using the below example.

Given the edges of a graph are: $\{\{1,2\}, \{2,3\}, \{4,5\}, \{6,7\}, \{5,6\}\}$



After applying the union by rank function to every edge the graph and the arrays will look like the following:

Observation 1:

If we carefully observe, we are only concerned about the ultimate parent but not the immediate parent.

Let's see **why we need to find the ultimate parents.**

After union by rank operations, if we are asked (refer to the above picture) if node 5 and node 7 belong to the same component or not, the answer must be yes. If we carefully look at their immediate parents, they are not the same but if we consider their ultimate parents they are the same i.e. node 4. So, we can determine the answer by considering the ultimate parent. That is why we need to find the ultimate parent.

So, here comes the **findPar() function** which will help us to find the ultimate parent for a particular node.

findPar() function:

This function actually takes a single node as an argument and finds the ultimate parent for each node.

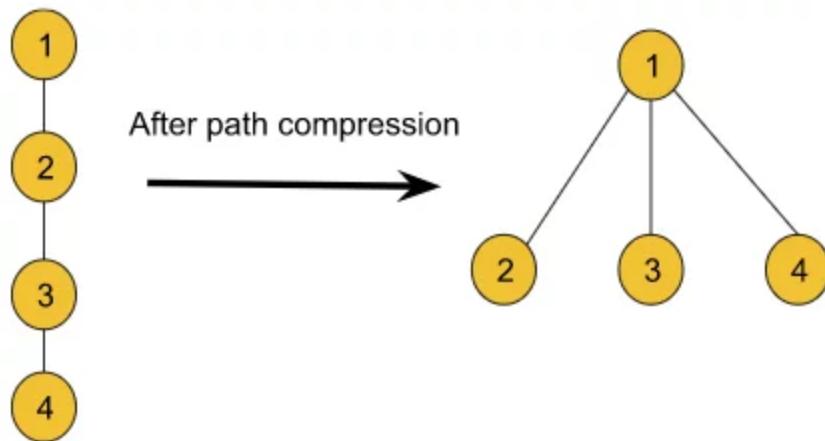
Observation 2:

Now, if we try to find the ultimate parent(typically using recursion) of each query separately, it will end up taking $O(\log N)$ time complexity for each case. But we want the operation to be done in a constant time. This is where the **path compression technique** comes in.

Using the **path compression technique** we can reduce the time complexity nearly to constant time. It is discussed later on why the time complexity actually reduces.

What is path compression?

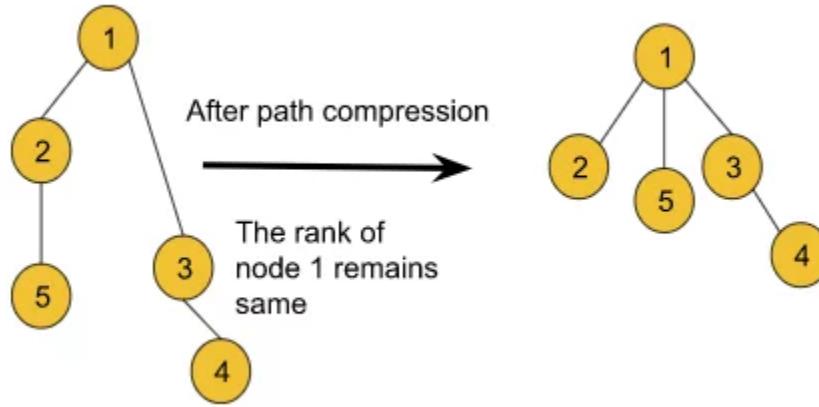
Basically, connecting each node in a particular path to its ultimate parent refers to path compression. Let's understand it using the following illustration:



How the time complexity reduces:

Before path compression, if we had tried to find the ultimate parent for node 4, we had to traverse all the way back to node 1 which is basically the height of size $\log N$. But after path compression, we can easily access the ultimate parent with a single step. Thus the traversal reduces and as a result the time complexity also reduces.

Though using the path compression technique it seems like the rank of the node is also changing, we cannot be sure about it. So, we will not make any changes to the rank array while applying path compression. The following example depicts an example:



Note: We cannot change the ranks while applying path compression.

Overall, `findPar()` method helps to reduce the time complexity of the **union by the rank** method as it can find the ultimate parent within constant time.

Algorithm:

This process is done using the backtracking method.

The algorithm steps are as follows:

1. **Base case:** If the node and the parent of the node become the same, it will return the node.
2. We will call the `findPar()` function for a node until it hits the base case and while backtracking we will update the parent of the current node with the returned value.

Note: The actual time complexity of union by rank and `findPar()` is $O(4)$ which is very small and close to 1. So, we can consider 4 as a constant. Now, this 4 term has a long mathematical derivation which is not required for an interview.

Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Code of Union by rank and `findPar()`:

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;
class DisjointSet {
    vector<int> rank, parent;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] = findUPar(parent[node]);
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[ulp_u]) {
            parent[ulp_v] = ulp_u;
        }
        else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }
};

int main() {
    DisjointSet ds(7);
    ds.unionByRank(1, 2);
    ds.unionByRank(2, 3);
    ds.unionByRank(4, 5);
    ds.unionByRank(6, 7);
    ds.unionByRank(5, 6);
    // if 3 and 7 same or not
    if (ds.findUPar(3) == ds.findUPar(7)) {
        cout << "Same\n";
    }
    else cout << "Not same\n";

    ds.unionByRank(3, 7);

    if (ds.findUPar(3) == ds.findUPar(7)) {
        cout << "Same\n";
    }
}

```

```

    }
else cout << "Not same\n";
return 0;
}

```

Output:

Not same

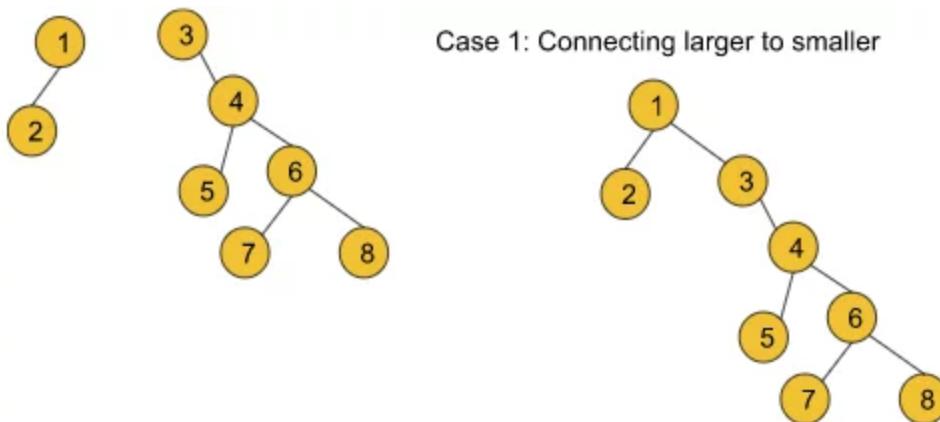
Same

Time Complexity: The actual time complexity is $O(4)$ which is very small and close to 1. So, we can consider 4 as a constant.

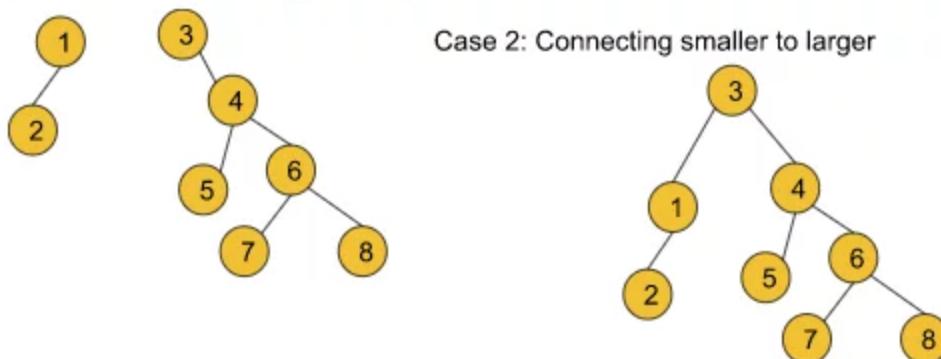
Follow-up question:

In the union by rank method, why do we need to connect the smaller rank to the larger rank?

Let's understand it using the following example:



In this case, the traversal time to find the ultimate parent for nodes 3, 4, 5, 6, 7, and 8 increases and so the path compression time also increases. But if we do the following



the traversal time to find the ultimate parent for only nodes 1 and 2 increases. So the path compression time becomes relatively lesser than in the previous case. So, we can conclude that we should always connect a smaller rank to a larger one with the goal of

- **shrinking the height of the graph.**
- **reducing the time complexity as much as we can.**

Observation 3:

Until now, we have learned union by rank, the findPar() function, and the path compression technique. Now, if we again carefully observe, after applying path compression the rank of the graphs becomes distorted. So, rather than storing the rank, we can just store the size of the components for comparing which component is greater or smaller.

So, here comes the concept of **Union by size**.

Union by size:

This is as same as the Union by rank method except this method uses the size to compare the components while connecting. That is why we need a '**size**' array of size N(no. of nodes) instead of a **rank** array. The size array will be storing the size for each particular node i.e. size[i] will be the size of the component starting from node i.

Typically, the size of a node refers to the number of nodes that are connected to it.

Algorithm:

Initial configuration:

size array: This array is initialized with one.

parent array: The array is initialized with the value of nodes i.e. parent[i] = i.

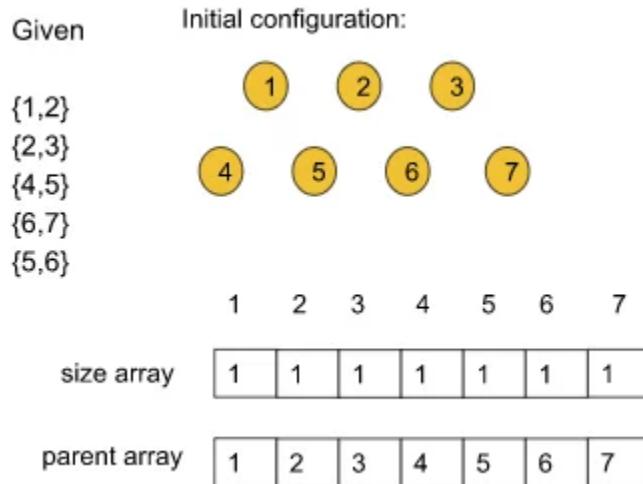
The algorithm steps are as follows:

1. Firstly, the Union function requires two nodes(**let's say u and v**) as arguments. Then we will find the ultimate parent (using the findPar() function discussed earlier) of u and v. Let's consider the ultimate parent of u is **pu** and the ultimate parent of v is **pv**.
2. After that, we will find the size of **pu** and **pv** i.e. size[pu] and size[pv].
3. Finally, we will connect the ultimate parent with a smaller size to the other ultimate parent with a larger size. But if the size of the two is equal, we can connect any parent to the other parent.

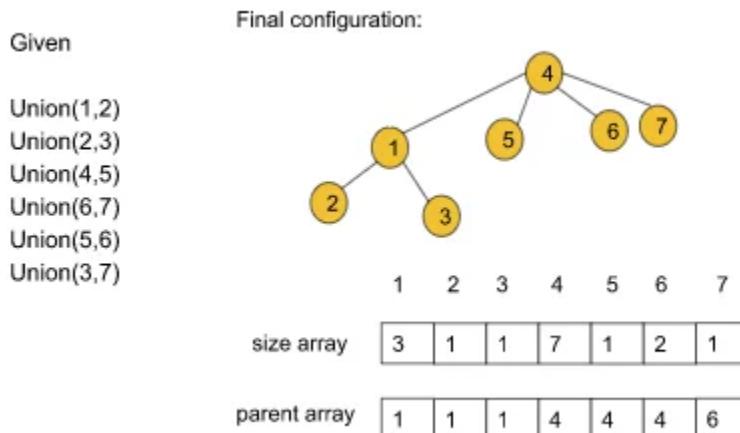
While connecting in both cases we will increase the size of the parent node to whom we have connected by the size of the other parent node which is actually connected.

Let's understand it further using the below example.

Given the edges of a graph are $\{\{1,2\}, \{2,3\}, \{4,5\}, \{6,7\}, \{5,6\}, \{3,7\}\}$



After applying the union by size function to every edge the graph and the arrays will look like the following:



Note: It seems much more intuitive than union by rank as the rank gets distorted after path compression.

Note: The `findPar()` function remains the exact same as we have discussed earlier.

Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Disjoint Set data structure implementation:

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;
class DisjointSet {
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] = findUPar(parent[node]);
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[ulp_u]) {
            parent[ulp_v] = ulp_u;
        }
        else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (size[ulp_u] < size[ulp_v]) {
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];
        }
        else {
            parent[ulp_v] = ulp_u;
            size[ulp_u] += size[ulp_v];
        }
    }
};

int main() {

```

```

DisjointSet ds(7);
ds.unionBySize(1, 2);
ds.unionBySize(2, 3);
ds.unionBySize(4, 5);
ds.unionBySize(6, 7);
ds.unionBySize(5, 6);
// if 3 and 7 same or not
if (ds.findUPar(3) == ds.findUPar(7)) {
    cout << "Same\n";
}
else cout << "Not same\n";

ds.unionBySize(3, 7);

if (ds.findUPar(3) == ds.findUPar(7)) {
    cout << "Same\n";
}
else cout << "Not same\n";
return 0;
}

```

Output:

Not Same

Same

Time Complexity: The time complexity is O(4) which is very small and close to 1. So, we can consider 4 as a constant.

*Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article. If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com*



Watch Video At: <https://youtu.be/aBxjDBC4M1U>

Kruskal's Algorithm – Minimum Spanning Tree : G-47

 takeuforward.org/data-structure/kruskals-algorithm-minimum-spanning-tree-g-47

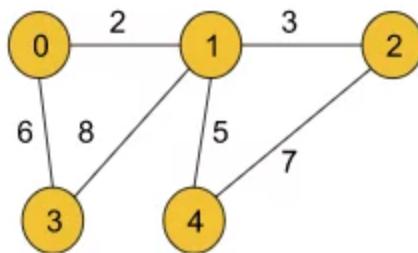
December 20, 2022

Problem Statement: Given a weighted, undirected, and connected graph of V vertices and E edges. The task is to find the sum of weights of the edges of the Minimum Spanning Tree.

Example 1:

Input Format:

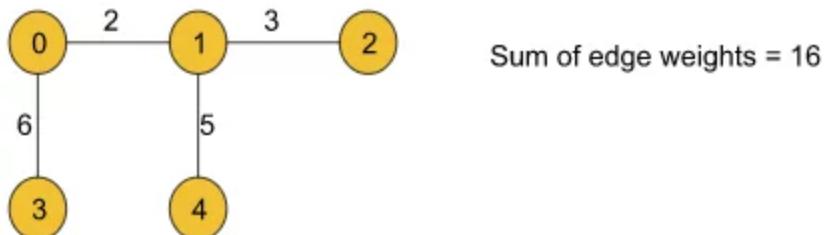
$V = 5$, edges = { {0, 1, 2}, {0, 3, 6}, {1, 2, 3}, {1, 3, 8}, {1, 4, 5}, {4, 2, 7} }



Result: 16

Explanation: The minimum spanning tree for the given graph is drawn below:

MST = {(0, 1), (0, 3), (1, 2), (1, 4)}

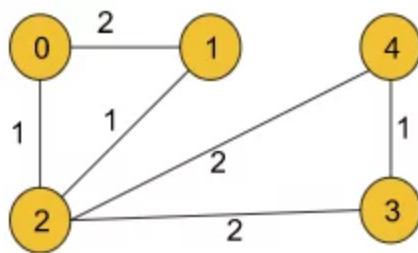


Example 2:

Input Format:

$V = 5$,

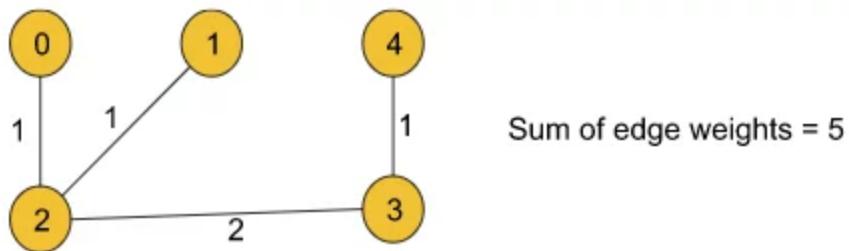
edges = { {0, 1, 2}, {0, 2, 1}, {1, 2, 1}, {2, 3, 2}, {3, 4, 1}, {4, 2, 2} }



Result: 5

Explanation: The minimum spanning tree is drawn below:

MST = {(0, 2), (1, 2), (2, 3), (3, 4)}



Disclaimer: Don't jump directly to the solution, try it out yourself first.

[Problem link](#).

Solution:

In the previous article on the [minimum spanning tree](#), we had already discussed that there are two ways to find the minimum spanning tree for a given weighted and undirected graph. Among those two algorithms, we have already discussed [Prim's algorithm](#).

In this article, we will be discussing another algorithm, named **Kruskal's algorithm**, that is also useful in finding the minimum spanning tree.

Approach:

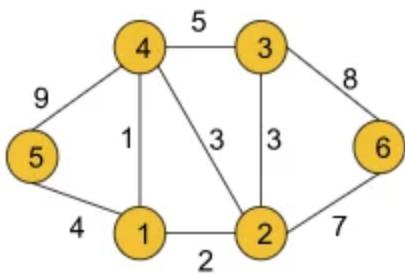
We will be implementing Kruskal's algorithm using the [Disjoint Set data structure](#) that we have previously learned.

Now, we know Disjoint Set provides two methods named **findUPar()** (*This function helps to find the ultimate parent of a particular node*) and **Union** (*This basically helps to add the edges between two nodes*). To know more about these functionalities, do refer to the article on

Disjoint Set.

The algorithm steps are as follows:

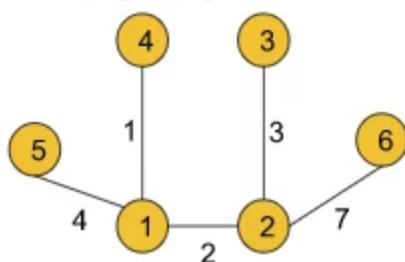
1. First, we need to extract the edge information(*if not given already*) from the given adjacency list in the format of (wt, u, v) where u is the current node, v is the adjacent node and wt is the weight of the edge between node u and v and we will store the tuples in an array.
2. Then the array must be sorted in the ascending order of the weights so that while iterating we can get the edges with the minimum weights first.
3. After that, we will iterate over the edge information, and for each tuple, we will apply the following operation:
 1. First, we will take the two nodes u and v from the tuple and check if the ultimate parents of both nodes are the same or not using the **findUPar()** function provided by the Disjoint Set data structure.
 2. **If the ultimate parents are the same**, we need not do anything to that edge as there already exists a path between the nodes and we will continue to the next tuple.
 3. If the ultimate parents are different, we will add the weight of the edge to our final answer(*i.e. mstWt variable used in the following code*) and apply the **union operation**(*i.e. either unionBySize(u, v) or unionByRank(u, v)*) with the nodes u and v. The union operation is also provided by the Disjoint Set.
4. Finally, we will get our answer (in the mstWt variable as used in the following code) successfully.



The minimum spanning tree with sum of edge weights = 17

Sorted edges according to weights:

(wt	u	v)
1	1	4
2	1	2
3	2	3
3	2	4
4	1	5
5	3	4
7	2	6
8	3	6
9	4	5



Note: Points to remember if the graph is given as an adjacency list we must extract the edge information first. As the graph contains bidirectional edges we can get a single edge twice in our array (For example, (wt, u, v) and (wt, v, u), (5, 1, 2) and (5, 2, 1)). But we should not worry about that as the Disjoint Set data structure will automatically discard the duplicate one.

Note: This algorithm mainly contains the Disjoint Set data structure used to find the minimum spanning tree of a given graph. So, we just need to know the data structure.

Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class DisjointSet {
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] = findUPar(parent[node]);
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[ulp_u]) {
            parent[ulp_v] = ulp_u;
        }
        else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (size[ulp_u] < size[ulp_v]) {
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];
        }
        else {
            parent[ulp_v] = ulp_u;
            size[ulp_u] += size[ulp_v];
        }
    }
}

```

```

};

class Solution
{
public:
    //Function to find sum of weights of edges of the Minimum Spanning Tree.
    int spanningTree(int V, vector<vector<int>> adj[])
    {
        // 1 - 2 wt = 5
        /// 1 - > (2, 5)
        // 2 -> (1, 5)

        // 5, 1, 2
        // 5, 2, 1
        vector<pair<int, pair<int, int>>> edges;
        for (int i = 0; i < V; i++) {
            for (auto it : adj[i]) {
                int adjNode = it[0];
                int wt = it[1];
                int node = i;

                edges.push_back({wt, {node, adjNode}});
            }
        }
        DisjointSet ds(V);
        sort(edges.begin(), edges.end());
        int mstWt = 0;
        for (auto it : edges) {
            int wt = it.first;
            int u = it.second.first;
            int v = it.second.second;

            if (ds.findUPar(u) != ds.findUPar(v)) {
                mstWt += wt;
                ds.unionBySize(u, v);
            }
        }

        return mstWt;
    }
};

int main() {

    int V = 5;
    vector<vector<int>> edges = {{0, 1, 2}, {0, 2, 1}, {1, 2, 1}, {2, 3, 2}, {3, 4, 1}, {4, 2, 2}};
    vector<vector<int>> adj[V];
    for (auto it : edges) {
        vector<int> tmp(2);
        tmp[0] = it[1];
        tmp[1] = it[2];
        adj[it[0]].push_back(tmp);
    }
}

```

```

        tmp[0] = it[0];
        tmp[1] = it[2];
        adj[it[1]].push_back(tmp);
    }

    Solution obj;
    int mstWt = obj.spanningTree(V, adj);
    cout << "The sum of all the edge weights: " << mstWt << endl;
    return 0;
}

```

Output: The sum of all the edge weights: 5

Time Complexity: $O(N+E) + O(E \log E) + O(E^*4\alpha^*2)$ where N = no. of nodes and E = no. of edges. $O(N+E)$ for extracting edge information from the adjacency list. $O(E \log E)$ for sorting the array consists of the edge tuples. Finally, we are using the disjoint set operations inside a loop. The loop will continue to E times. Inside that loop, there are two disjoint set operations like `findUPar()` and `UnionBySize()` each taking 4 and so it will result in 4^*2 . That is why the last term $O(E^*4^*2)$ is added.

Space Complexity: $O(N) + O(N) + O(E)$ where E = no. of edges and N = no. of nodes. $O(E)$ space is taken by the array that we are using to store the edge information. And in the disjoint set data structure, we are using two N -sized arrays i.e. a parent and a size array (as we are using `unionBySize()` function otherwise, a rank array of the same size if `unionByRank()` is used) which result in the first two terms $O(N)$.

*Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out [this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com*



Watch Video At: https://youtu.be/DMnDM_sxVig

Number of Operations to Make Network Connected – DSU: G-49.

 takeuforward.org/data-structure/number-of-operations-to-make-network-connected-dsu-g-49

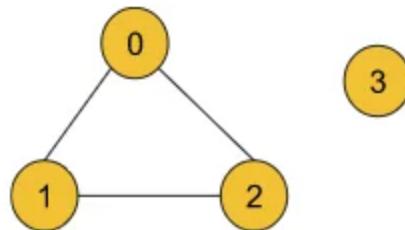
January 3, 2023

Problem Statement: You are given a graph with n vertices and m edges. You can remove one edge from anywhere and add that edge between any two vertices in one operation. Find the minimum number of operations that will be required to make the graph connected. If it is not possible to make the graph connected, return -1.

Pre-requisite: [Disjoint Set data structure](#)

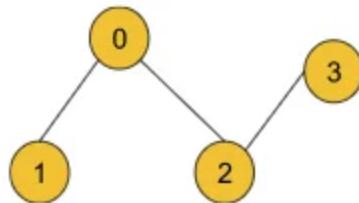
Example 1:

Input Format: $N = 4$, $M = 3$, $\text{Edge}[] = [[0, 1], [0, 2], [1, 2]]$



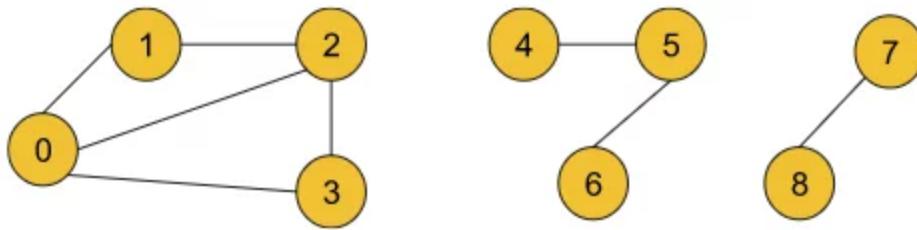
Result: 1

Explanation: We need a minimum of 1 operation to make the two components connected. We can remove the edge $(1, 2)$ and add the edge between node 2 and node 3 like the following:



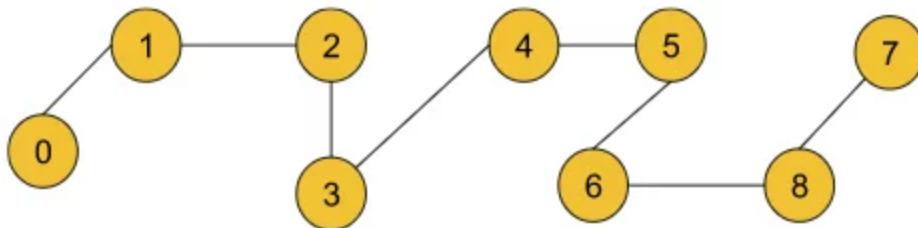
Example 2:

Input Format: N = 9, M = 8, Edge[] = [[0,1],[0,2],[0,3],[1,2],[2,3],[4,5],[5,6],[7,8]]



Result: 2

Explanation: We need a minimum of 2 operations to make the two components connected. We can remove the edge (0,2) and add the edge between node 3 and node 4 and we can remove the edge (0,3) and add it between nodes 6 and 8 like the following:



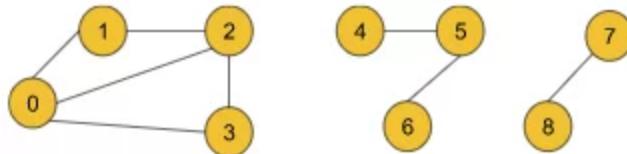
Solution

Disclaimer: Don't jump directly to the solution, try it out yourself first.

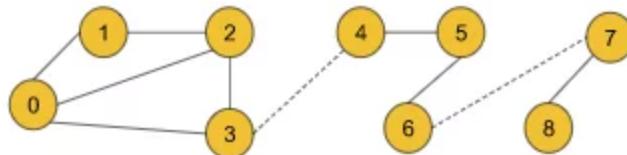
[Problem Link](#).

Solution:

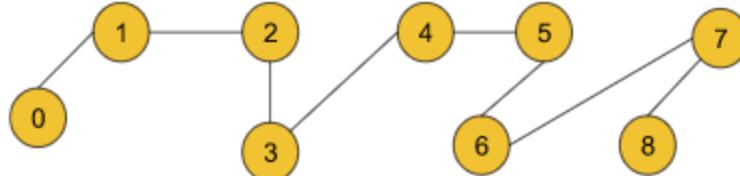
Before discussing the solution, let's understand the question. In the question, it is clearly stated that our objective is to make the given graph connected. Now, we can do this by just connecting the different components of a given graph with some edges like the following:



This graph contains 3 different components. If we just connect node 3 and 4 and node 6 and 7 the graph becomes connected like the following:



But according to the question, we cannot add random edges from outside rather we can remove any given edge and add that to a new position to fulfill the purpose. So, one of the correct ways to connect the above graph will be the following:

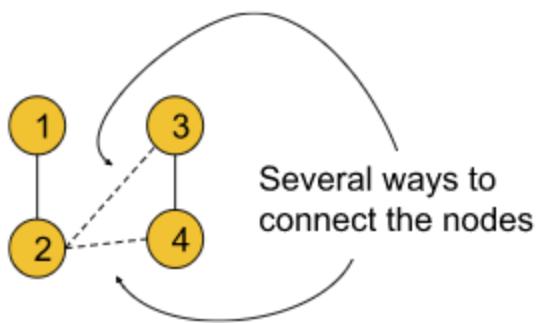


There may be several other possible ways to make the graph connected. In this graph, We have removed the edge (0,2) and added it between node 3 and node 4 and we have also removed the edge (0,3) and added it between nodes 6 and 8 to make the components connected.

Note: *In order to add any edge to the desired position, we must take it out from somewhere inside the graph. We cannot add any edge randomly from outside. So, the intuition is to remove the required minimum number of edges and plant them somewhere in the graph so that the graph becomes connected.*

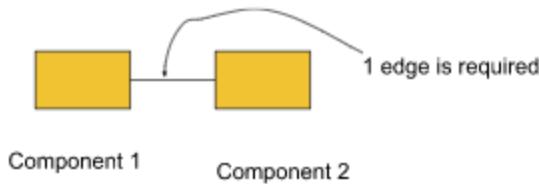
Observation 1: How can we connect components to make the graph connected?

In order to connect two different components of a graph we need to connect any node of the first component to any node of the second component. For example, if we have a graph like the following we can connect them in several ways like connecting nodes 2 and 3 or connecting nodes 2 and 4, and so on.

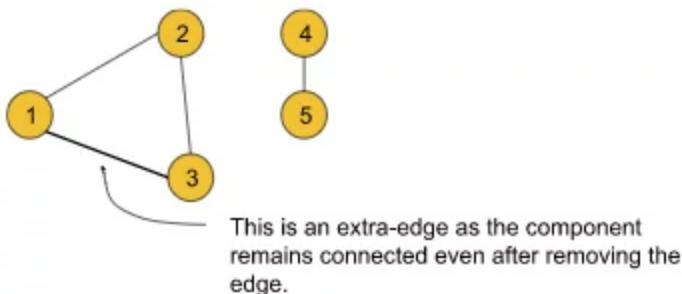


Observation 2:

From the method of connecting the components, discussed above, ***we can conclude that we need a minimum of $nc-1$ edges to make the graph connected if the graph contains nc number of different components.***



For example, the above graph has two different components and so to make it connected we need a minimum of 1 edge. ***Similarly, if a graph contains a single component we need 0 edges to make it connected.*** We need to remove the edges in such a way that the components remain connected even after removing those edges. We can assume these types of edges as ***extra-edges***.



Until now, we have found that we need a minimum of $nc-1$ edges ($nc = \text{no. of components of the graph}$) to make the graph connected. And according to the question, to add these $nc-1$ edges, the graph must contain a minimum of $nc-1$ extra edges.

So, we can conclude that if a graph contains $nc-1$ extra-edges, we can make the graph connected with just $nc-1$ operations (where $nc = \text{no. of components of the graph}$).

Approach:

In order to solve this question we will first find out the number of extra-edges and then we will find out the number of components of the graph. We will be using the [Disjoint Set data structure](#) to do so.

The algorithm steps are the following:

1. First we need to extract all the edge information (**If not already given**) in the form of the pair (u, v) where u = starting node and v = ending node. We should store all the edge information in an array.
2. Then we will iterate through the array selecting every pair and checking the following:
 1. **If the ultimate parent of u and v (checked using the `findPar()` method of the Disjoint set) becomes the same**, we should increase the count of extra-edges by 1.
Because the same ultimate parent means the nodes are already connected and so we can consider the current edge as an extra edge.
 2. But if the ultimate parents are different, then we should apply the union(either `unionBySize()` or `unionByRank()`) method on those two nodes.
3. Thus we will get the count of the extra edges. Now it's time to count the number of components. In order to do so, we will just count the number of the nodes that are the ultimate parent of themselves.
4. We will iterate over all the nodes and for each node, we will check the following:
 1. If the node is the ultimate parent of itself, we will increase the count of components by 1.
 2. Otherwise, we will continue to the next node.

This checking will be done using the parent array inside the Disjoint set.

5. Finally, we will check the count of extra edges and the number of components. If the count of extra-edges is greater or the same, we will return the answer that is (number of components – 1), and otherwise, we will return -1.

Note: *If you wish to see the dry run of the above approach, you can watch the video attached to this article.*

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

//User function Template for C++
class DisjointSet {
public:
    vector<int> rank, parent, size;
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] = findUPar(parent[node]);
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[ulp_u]) {
            parent[ulp_v] = ulp_u;
        }
        else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (size[ulp_u] < size[ulp_v]) {
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];
        }
        else {
            parent[ulp_v] = ulp_u;
            size[ulp_u] += size[ulp_v];
        }
    }
}

```

```

};

class Solution {
public:
    int Solve(int n, vector<vector<int>>& edge) {
        DisjointSet ds(n);
        int cntExtras = 0;
        for (auto it : edge) {
            int u = it[0];
            int v = it[1];
            if (ds.findUPar(u) == ds.findUPar(v)) {
                cntExtras++;
            }
            else {
                ds.unionBySize(u, v);
            }
        }
        int cntC = 0;
        for (int i = 0; i < n; i++) {
            if (ds.parent[i] == i) cntC++;
        }
        int ans = cntC - 1;
        if (cntExtras >= ans) return ans;
        return -1;
    }
};

int main() {

    int V = 9;
    vector<vector<int>> edge = {{0, 1}, {0, 2}, {0, 3}, {1, 2}, {2, 3}, {4, 5}, {5, 6}, {7, 8}};

    Solution obj;
    int ans = obj.Solve(V, edge);
    cout << "The number of operations needed: " << ans << endl;
    return 0;
}

```

Output: The number of operations needed: 2 (for example 2)

Time Complexity: $O(E^*4\alpha)+O(N^*4\alpha)$ where E = no. of edges and N = no. of nodes. The first term is to calculate the number of extra edges and the second term is to count the number of components. 4α is for the disjoint set operation we have used and this term is so small that it can be considered constant.

Space Complexity: $O(2N)$ where N = no. of nodes. $2N$ for the two arrays(parent and size) of size N we have used inside the disjoint set.

*Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, [please check out this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com*



Watch Video At: https://youtu.be/FYrl7iz9_ZU

Most Stones Removed with Same Row or Column – DSU: G-53

 takeuforward.org/data-structure/most-stones-removed-with-same-row-or-column-dsu-g-53

January 5, 2023

Problem Statement: There are n stones at some integer coordinate points on a 2D plane. Each coordinate point may have at most one stone.

You need to remove some stones.

A stone can be removed if it shares either the same row or the same column as another stone that has not been removed.

Given an array of stones of length n where $\text{stones}[i] = [x_i, y_i]$ represents the location of the ith stone, return the maximum possible number of stones that you can remove.

Pre-requisite: [Disjoint Set data structure](#)

Example 1:

Input Format: n=6 stones = [[0, 0], [0, 1], [1, 0], [1, 2], [2, 1], [2, 2]]

S	S	
S		S
	S	S

Result: 5

Explanation: One of the many ways to remove 5 stones is to remove the following stones:

[0,0], [1,0], [0,1], [2,1], [1,2]

Example 2:

Input Format: N = 6, stones = {{0, 0}, {0, 2}, {1, 3}, {3, 1}, {3, 2}, {4, 3}};

S		S	
			S
	S	S	
			S

Result: 4

Explanation: We can remove the following stones:
[0,0], [0,2], [1,3], [3,1]

Disclaimer: Don't jump directly to the solution, try it out yourself first.

[Problem Link](#).

Solution:

Let's first understand the thought process that we will be using to solve this problem. In this problem, it is clearly stated that a stone can be removed if it shares either the same row or the same column as another stone that has not been removed. So, we can assume that these types of stones, sharing either the same row or column, are connected and belong to the same group. If we take example 2:

S		S	
			S
	S	S	
			S

We can easily spot two different groups in this example. The first group includes the stones [0,0], [0,2], [3,2], and [3,1], and the second one includes [1,3] and [4,3].

If we carefully observe, for each group we can remove all the stones leaving one stone intact. So, we can conclude that at most we can remove (size of the group -1) no. of stones from a group as we need to leave one stone untouched for each group.

Now, if we can think of the stones as nodes, the different groups then seem to be the different components of a graph.

Mathematical Explanation of getting the maximum no. of stones:

Let's assume there are n stones in total. And these n stones have formed k different components each containing X_i no. of stones. This indicates the following:

$$\sum_{i=1}^k X_i = X_1 + X_2 + \dots + X_k = n, \text{ where } X_i = \text{no. of stones in } i\text{th component}$$

Now, we have already seen, that we can at most remove $(X_i - 1)$ no. of stones from i th component that contains a total of X_i no. of stones. So,

$$\begin{aligned}\text{total no. of removed stone} &= \sum_{i=1}^k (X_i - 1) = (X_1 - 1) + (X_2 - 1) + \dots + (X_k - 1) \\ &= (X_1 + X_2 + \dots + X_k) - (1 + 1 + \dots, k \text{ times}) = \sum_{i=1}^k X_i - k \\ &= (n - k), \text{ where } n = \text{total no. stones, and } k = \text{total no. of components}\end{aligned}$$

Until now, we have proved that we can remove a maximum of $(n-k)$ no. of stones from the whole 2D plane, where n is the total number of stones and k is the total number of components.

Now, we have reduced the question in such a way that we just need to connect the stones properly to find out the number of different components and we will easily solve the problem.

Here we are getting the thought of connected components. So, we can easily decide to choose the Disjoint Set data structure to solve this problem.

How to connect the cells containing stones to form a component:

In order to connect the cells we will assume that each entire row and column of the 2D plane is a particular node. Now, with each row, we will connect the column no.s in which the stones are located. But column no. may be the same as the row number. To avoid this, we will convert each column no. to (column no. + total no. of rows) and perform the union of row no. and the converted column number i.e. (column no. + total no. of rows) like the following:

S		S	
			S
	S	S	
			S

For the above example, to connect the two stones in the cells [0, 0] and [0, 2] of the first row, we will first take row no. i.e. 0 (because of 0-based indexing) as a node and then convert column no.s 0 to (0+5) and 2 to (2+5). Then, we will perform the union of (0 and 5) and (0

and 7).

Thus we **will connect all the stones that are either in the same row or in the same column** to form different connected components.

Approach:

The algorithm steps are as follows:

1. First, from the stone information, we will find out the maximum row and the maximum column number so that we can get an idea about the size of the 2D plane(i.e. nothing but a matrix).
2. Then, we need to create a disjoint set of sizes (maximum row index+maximum column index). For safety, we may take a size one more than required.
3. Now it's time to connect the cells having a stone. For that we will loop through the given cell information array and for each cell we will extract the row and the column number and do the following:
 1. First, we will convert column no. to (column no. + maximum row index +1).
 2. We will perform the union(**either unionBySize() or unionByRank()**) of the row number and the converted column number.
 3. We will store the row and the converted column number in a map data structure for later use.
4. Now, it's time to calculate the number of components and for that, we will count the number of ultimate parents. Here we will refer to the previously created map.
 1. We just need the nodes in the Disjoint Set that are involved in having a stone. So we have stored the rows and the columns in a map in step 3.3, as they will have stones. Now we just need to check them from the map data structure once for getting the number of ultimate parents.
5. Finally, we will subtract the no. of components(i.e. no. of ultimate parents) from the total no. of stones and we will get our answer.

Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class DisjointSet {
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] = findUPar(parent[node]);
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[ulp_u]) {
            parent[ulp_v] = ulp_u;
        }
        else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (size[ulp_u] < size[ulp_v]) {
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];
        }
        else {
            parent[ulp_v] = ulp_u;
            size[ulp_u] += size[ulp_v];
        }
    }
}

```

```

};

class Solution {
public:
    int maxRemove(vector<vector<int>>& stones, int n) {
        int maxRow = 0;
        int maxCol = 0;
        for (auto it : stones) {
            maxRow = max(maxRow, it[0]);
            maxCol = max(maxCol, it[1]);
        }
        DisjointSet ds(maxRow + maxCol + 1);
        unordered_map<int, int> stoneNodes;
        for (auto it : stones) {
            int nodeRow = it[0];
            int nodeCol = it[1] + maxRow + 1;
            ds.unionBySize(nodeRow, nodeCol);
            stoneNodes[nodeRow] = 1;
            stoneNodes[nodeCol] = 1;
        }

        int cnt = 0;
        for (auto it : stoneNodes) {
            if (ds.findUPar(it.first) == it.first) {
                cnt++;
            }
        }
        return n - cnt;
    }
};

int main() {

    int n = 6;
    vector<vector<int>> stones = {
        {0, 0}, {0, 2},
        {1, 3}, {3, 1},
        {3, 2}, {4, 3}
    };

    Solution obj;
    int ans = obj.maxRemove(stones, n);
    cout << "The maximum number of stones we can remove is: " << ans << endl;
    return 0;
}

```

Output: The maximum number of stones we can remove is: 4 (For example 2)

Time Complexity: O(N), where N = total no. of stones. Here we have just traversed the given stones array several times. And inside those loops, every operation is apparently taking constant time. So, the time complexity is only the time of traversal of the array.

Space Complexity: $O(2^*(\text{max row index} + \text{max column index}))$ for the parent and size array inside the Disjoint Set data structure.

*Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, [please check out this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com*



Watch Video At: <https://youtu.be/OwMNX8SPavM>

Accounts Merge – DSU: G-50

Problem Statement: Given a list of accounts where each element account [i] is a list of strings, where the first element account [i][0] is a name, and the rest of the elements are emails representing emails of the account.

Now, we would like to merge these accounts. Two accounts definitely belong to the same person if there is some common email to both accounts. Note that even if two accounts have the same name, they may belong to different people as people could have the same name. A person can have any number of accounts initially, but all of their accounts definitely have the same name.

After merging the accounts, return the accounts in the following format: the first element of each account is the name, and the rest of the elements are emails in sorted order.

Note: Accounts themselves can be returned in any order.

Pre-requisite: [Disjoint Set data structure](#)

Examples:

Example 1:**Input:** N = 4

```
accounts [ ] =  
[["John", "johnsmith@mail.com", "john_newyork@mail.com"],  
["John", "johnsmith@mail.com", "john00@mail.com"],  
["Mary", "mary@mail.com"],  
["John", "johnnybravo@mail.com"]]
```

Output: [[{"John": "john00@mail.com", "john_newyork@mail.com": "johnsmith@mail.com"}, {"Mary": "mary@mail.com"}, {"John": "johnnybravo@mail.com"}]]

Explanation: The first and the second John are the same person as they have a common email. But the third Mary and fourth John are not the same as they do not have any common email. The result can be in any order but the emails must be in sorted order. The following is also a valid result:

```
[['Mary', 'mary@mail.com'],  
['John', 'johnnybravo@mail.com'],  
['John', 'john00@mail.com', 'john_newyork@mail.com', 'johnsmith@mail.com']]
```

Example 2:**Input:** N = 6

```
accounts [ ] =  
[["John", "j1@com", "j2@com", "j3@com"],  
["John", "j4@com"],  
["Raj", "r1@com", "r2@com"],  
["John", "j1@com", "j5@com"],  
["Raj", "r2@com", "r3@com"],  
["Mary", "m1@com"]]
```

Output: [[{"John": "j1@com", "j2@com", "j3@com", "j5@com", "j4@com"}, {"Raj": "r1@com", "r2@com", "r3@com"}, {"Mary": "m1@com"}]]

Explanation: The first and the fourth John are the same person here as they have a common email. And the third and the fifth Raj are also the same person. So, the same accounts are merged.

Solution

Disclaimer: Don't jump directly to the solution, try it out yourself first.

[Problem link](#).

Solution:

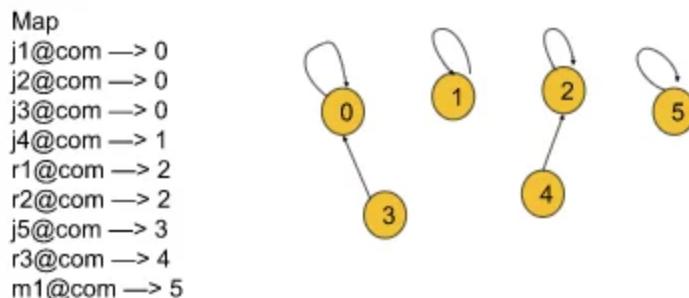
Let's quickly understand the question before moving on to the solution part. According to the question, we can only merge two accounts with the same name only if the accounts contain at least one common email. After merging the accounts accordingly, we should return the

answer where for each account the emails must be in the sorted order. But the order of the accounts does not matter. In order to solve this problem we are going to use the Disjoint Set data structure. Now, let's discuss the approach using the following example:

```
Given: N = 6
accounts [ ] =
[["John", "j1@com", "j2@com", "j3@com"],
 ["John", "j4@com"],
 ["Raj", "r1@com", "r2@com"],
 ["John", "j1@com", "j5@com"],
 ["Raj", "r2@com", "r3@com"],
 ["Mary", "m1@com"]]
```

First, we will try to iterate over every single email and add them with their respective indices(i.e. Index of the accounts the email belongs to) in a map data structure. While doing this, when we will reach out to "j1@com" in the fourth account, we will find that it is already mapped with index 0. This incident means that we are currently in an account that can be merged. So, we will perform the union operation between the current index i.e. 3, and index 0(As in this case, we are following 0-based indexing). It will mean that the ultimate parent of index 3 is index 0. Similarly, this incident will repeat in the case of the third and fifth Raj. So we will perform the union of index 2 and 4.

After completing the above process, the situation will be like the following:



Now, it's time to merge the emails. So, we will iterate over each email and will add them to the ultimate parent of the current account's index. Like, while adding the emails of account 4, we will add them to index 2 as the ultimate parent of 4 is index 2.

Finally, we will sort the emails for each account individually to get our answers in the format specified in the question.

Approach:

Note:

- Here we will perform the disjoint set operations on the indices of the accounts considering them as the nodes.

- As in each account, the first element is the name, we will start iterating from the second element in each account to visit only the emails sequentially.

The algorithm steps are the following:

1. First, we will **create a map data structure**. Then we will store each email with the respective index of the account(the email belongs to) in that map data structure.
2. While doing so, if we encounter an email again(i.e. If any index is previously assigned for the email), we will perform union(**either unionBySize() or unionByRank()**) of the current index and the previously assigned index.
3. After completing step 2, now it's time to **merge the accounts**. For merging, we will iterate over all the emails individually and find the ultimate parent(**using the findUPar() method**) of the assigned index of every email. Then we will add the email of the current account to the index(account index) that is the ultimate parent. Thus the accounts will be merged.
4. Finally, we will **sort the emails for every account separately** and store the final results in the answer array accordingly.

Note: *If you wish to see the dry run of the above approach, you can watch the video attached to this article.*

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;
//User function Template for C++
class DisjointSet {
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] = findUPar(parent[node]);
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[ulp_u]) {
            parent[ulp_v] = ulp_u;
        }
        else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (size[ulp_u] < size[ulp_v]) {
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];
        }
        else {
            parent[ulp_v] = ulp_u;
            size[ulp_u] += size[ulp_v];
        }
    }
};


```

```

class Solution {
public:
    vector<vector<string>> accountsMerge(vector<vector<string>> &details) {
        int n = details.size();
        DisjointSet ds(n);
        sort(details.begin(), details.end());
        unordered_map<string, int> mapMailNode;
        for (int i = 0; i < n; i++) {
            for (int j = 1; j < details[i].size(); j++) {
                string mail = details[i][j];
                if (mapMailNode.find(mail) == mapMailNode.end()) {
                    mapMailNode[mail] = i;
                }
                else {
                    ds.unionBySize(i, mapMailNode[mail]);
                }
            }
        }

        vector<string> mergedMail[n];
        for (auto it : mapMailNode) {
            string mail = it.first;
            int node = ds.findUPar(it.second);
            mergedMail[node].push_back(mail);
        }

        vector<vector<string>> ans;

        for (int i = 0; i < n; i++) {
            if (mergedMail[i].size() == 0) continue;
            sort(mergedMail[i].begin(), mergedMail[i].end());
            vector<string> temp;
            temp.push_back(details[i][0]);
            for (auto it : mergedMail[i]) {
                temp.push_back(it);
            }
            ans.push_back(temp);
        }
        sort(ans.begin(), ans.end());
        return ans;
    }
};

int main() {

    vector<vector<string>> accounts = {{"John", "j1@com", "j2@com", "j3@com"},
                                         {"John", "j4@com"}, {"Raj", "r1@com", "r2@com"}, {"John", "j1@com", "j5@com"}, {"Raj", "r2@com", "r3@com"},

}

```

```

        {"Mary", "m1@com"}
    };

Solution obj;
vector<vector<string>> ans = obj.accountsMerge(accounts);
for (auto acc : ans) {
    cout << acc[0] << ":";  

    int size = acc.size();
    for (int i = 1; i < size; i++) {
        cout << acc[i] << " ";
    }
    cout << endl;
}
return 0;
}

```

Output:

John:j1@com j2@com j3@com j5@com
 John:j4@com
 Mary:m1@com
 Raj:r1@com r2@com r3@com

Time Complexity: $O(N+E) + O(E^4\alpha) + O(N^*(E\log E + E))$ where N = no. of indices or nodes and E = no. of emails. The first term is for visiting all the emails. The second term is for merging the accounts. And the third term is for sorting the emails and storing them in the answer array.

Space Complexity: $O(N) + O(N) + O(2N) \sim O(N)$ where N = no. of nodes/indices. The first and second space is for the ‘mergedMail’ and the ‘ans’ array. The last term is for the parent and size array used inside the Disjoint set data structure.

*Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, [please check out this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com*



Watch Video At: https://youtu.be/FMwpt_aQOGw

Number of Islands – II – Online Queries – DSU: G-51

 takeuforward.org/graph/number-of-islands-ii-online-queries-dsu-g-51

January 3, 2023

Problem Statement: You are given an n, m which means the row and column of the 2D matrix, and an array of size k denoting the number of operations. Matrix elements are 0 if there is water or 1 if there is land. Originally, the 2D matrix is all 0 which means there is no land in the matrix. The array has k operator(s) and each operator has two integers $A[i][0], A[i][1]$ means that you can change the cell matrix $[A[i][0]][A[i][1]]$ from sea to island. Return how many islands are there in the matrix after each operation. You need to return an array of size k .

Note: An island means a group of 1s such that they share a common side.

Pre-requisite: [Disjoint Set data structure](#)

Example 1:

Input Format: $n = 4 m = 5 k = 4 A = \{\{1,1\}, \{0,1\}, \{3,3\}, \{3,4\}\}$ **Output:** 1 1 2 2 **Explanation:**

The following illustration is the representation of the operation:

0. 00000	1. 00000	2. 01000	3. 01000	4. 01000
00000	01000	01000	01000	01000
00000	00000	00000	00000	00000
00000	00000	00000	00010	00011

Final array

Example 2:

Input Format: $n = 4 m = 5 k = 12 A = \{\{0,0\}, \{0,0\}, \{1,1\}, \{1,0\}, \{0,1\}, \{0,3\}, \{1,3\}, \{0,4\}, \{3,2\}, \{2,2\}, \{1,2\}, \{0,2\}\}$ **Output:** 1 1 2 1 1 2 2 2 3 3 1 1 **Explanation:** If we follow the process like in example 1, we will get the above result.

Solution

Disclaimer: Don't jump directly to the solution, try it out yourself first.

[Problem Link](#).

Before moving on to the solution, let's quickly discuss some points about the question. First, we need to remember that an island means a group of 1s such that they share a common side. If we look into it from the matrix view, the statement actually means that two cells with

value 1 are considered a single group if one of them is located in any of the four directions (Up, Down, Left, Right) of the other cell. But two diagonal adjacent cells will not be considered a single group rather they will be counted as different groups. The following illustration will depict the concept:

1	1	0	0	0
0	0	1	0	0
0	0	0	0	0
0	1	0	0	0

Here cells [0,0] and [0,1] are considered a single island as they share a common side but cells [0,1] and [1,2] must be considered two different islands as they do not have any common side.

Now, in the question, it is clearly stated that the operations are given in an array and we should find the number of islands after each operation. This fact actually indicates that after performing each operation the structure of the islands and the sea may change. If we assume the structure as a graph, the graph will be dynamic in nature. And there is also a concept of connecting two different islands if they share a common side.

So, from these observations, we can easily decide to choose the Disjoint Set data structure in order to solve this problem.

These types of problems are considered online query problems where we need to find the result after every query.

Let's discuss the following observations:

Observation 1: What does each operation/query mean?

In each operation/query, an index of a cell will be given and we need to add an island on that particular cell i.e. we need to place the value 1 to that particular cell.

Observation 2: Optimizing the repeating same operations

The same operations may repeat any number of times but it is meaningless to perform all of them every time. So, we will maintain a visited array that will keep track of the cells on which the operations have been already performed. If the operations repeat, by just checking the visited array we can decide not to calculate again, and instead, just take the current answer into our account. Thus we can optimize the number of operations.

Observation 3: How to connect cells to include them in the same group or consider them a single island.

Generally, a cell is represented by two parameters i.e. row and column. But to connect the cells as we have done with nodes, we need to first represent each cell with a single number. So, we will number them from 0 to $n*m-1$ (from left to right) where n = no. of total rows and m = total no. of columns.

For example, if a 5X4 matrix is given we will number the cell in the following way:

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19

Now if we want to connect cells (1,0) and (2,0), we will just perform a union of 5 and 10. The number for representing each cell can be found using the following formula:
number = (row of the current cell*total number of columns)+column of the current cell
for example, for the cell (2, 0) the number is = $(2*5) + 0 = 10$.

Observation 4: How to count the number of islands.

For each operation, if the given cell is not visited, we will first mark the cell visited and increase the counter by 1. Now we will check all four sides of the given cell. If any other islands are found, we will connect the current cell with each of them(If not already connected) decreasing the counter value by 1. While connecting we need to check if the cells are already connected or not. For this, we will first convert the cells' indices into numbers using the above formula and then we will check their ultimate parents. If the parents become the same, we will not connect them as well as we will not make any changes to the counter variable. Thus the number of islands will be calculated.

Approach:

The algorithm steps are as follows:

Initial Configuration:

Visited array: This 2D array should be initialized with 0.

Counter variable: This variable will also be initialized with 0.

Answer array: After performing the algorithm, this array will store the results after performing the queries.

1. First, we will iterate over all the queries selecting each at a time. Now, we can get the row and the column of the cell given in that query.

2. Then, we will check that cell in the visited array, if the cell is previously visited or not.
 1. **If the cell is previously visited**, we will just take the current count into our account storing that count value in our answer array and we will move on to the next query.
 2. **Otherwise**, we will mark the cell as visited in the visited array and increase the value of the counter variable by 1.
 1. Now, it's time to connect the adjacent islands properly. For that, we will check all four adjacent cells of the current cell. If any island is found, we will first check if they(the current cell and the adjacent cell that contains an island) are already connected or not using the **findUPar()** method.
 2. For checking, we will first convert the indices of the current cell and the adjacent cell into the numbers using the specified formula. Then we will check their ultimate parents.
 3. **If the ultimate parents are different**, we will decrease the counter value by 1 and perform the union(**either unionBySize() or unionByRank()**) between those two numbers that represent the cells.
 4. Similarly, checking all four sides and making the required changes in the counter variable, we will put the counter value into our answer array.
3. After performing step 2 for all the queries, we will get our final answer array containing the results for all the queries.

Note: *If you wish to see the dry run of the above approach, you can watch the video attached to this article.*

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

// User function Template for C++
class DisjointSet {
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] = findUPar(parent[node]);
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[ulp_u]) {
            parent[ulp_v] = ulp_u;
        }
        else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (size[ulp_u] < size[ulp_v]) {
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];
        }
        else {
            parent[ulp_v] = ulp_u;
            size[ulp_u] += size[ulp_v];
        }
    }
}

```

```

};

class Solution {
private:
    bool isValid(int adjr, int adjc, int n, int m) {
        return adjr >= 0 && adjr < n && adjc >= 0 && adjc < m;
    }
public:
    vector<int> numOfIslands(int n, int m,
                             vector<vector<int>> &operators) {
        DisjointSet ds(n * m);
        int vis[n][m];
        memset(vis, 0, sizeof vis);
        int cnt = 0;
        vector<int> ans;
        for (auto it : operators) {
            int row = it[0];
            int col = it[1];
            if (vis[row][col] == 1) {
                ans.push_back(cnt);
                continue;
            }
            vis[row][col] = 1;
            cnt++;
            // row - 1, col
            // row , col + 1
            // row + 1, col
            // row, col - 1;
            int dr[] = { -1, 0, 1, 0 };
            int dc[] = { 0, 1, 0, -1 };
            for (int ind = 0; ind < 4; ind++) {
                int adjr = row + dr[ind];
                int adjc = col + dc[ind];
                if (isValid(adjr, adjc, n, m)) {
                    if (vis[adjr][adjc] == 1) {
                        int nodeNo = row * m + col;
                        int adjNodeNo = adjr * m + adjc;
                        if (ds.findUPar(nodeNo) != ds.findUPar(adjNodeNo)) {
                            cnt--;
                            ds.unionBySize(nodeNo, adjNodeNo);
                        }
                    }
                }
            }
            ans.push_back(cnt);
        }
        return ans;
    }
};

int main() {

```

```

int n = 4, m = 5;
vector<vector<int>> operators = {{0, 0}, {0, 0}, {1, 1}, {1, 0}, {0, 1},
    {0, 3}, {1, 3}, {0, 4}, {3, 2}, {2, 2}, {1, 2}, {0, 2}}
};

Solution obj;
vector<int> ans = obj.numOfIslands(n, m, operators);
for (auto res : ans) {
    cout << res << " ";
}
cout << endl;
return 0;
}

```

Output: 1 1 2 1 1 2 2 2 3 3 1 1

Time Complexity: $O(Q^*4\alpha) \sim O(Q)$ where Q = no. of queries. The term 4α is so small that it can be considered constant.

Space Complexity: $O(Q) + O(N*M) + O(N*M)$, where Q = no. of queries, N = total no. of rows, M = total no. of columns. The last two terms are for the parent and the size array used inside the Disjoint set data structure. The first term is to store the answer.

*Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, [please check out this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com*



Watch Video At: <https://youtu.be/Rn6B-Q4SNyA>

Making a Large Island – DSU: G-52

 takeuforward.org/data-structure/making-a-large-island-dsu-g-52

January 5, 2023



Problem Statement: You are given an $n \times n$ binary grid. A grid is said to be binary if every value in the grid is either 1 or 0. You can change at most one cell in the grid from 0 to 1. You need to find the largest group of connected 1's. Two cells are said to be connected if both are adjacent to each other and both have the same value.

Pre-requisite: [Disjoint Set data structure](#)

Example 1:

Input Format: The following grid is given:

1	1	0	1	1	0
1	1	0	1	1	0
1	1	0	1	1	0
0	0	1	0	0	0
0	0	1	1	1	0
0	0	1	1	1	0

Result: 20

Explanation: We can get the largest group of 20 connected 1s if we change the (2,2) to 1. The groups are shown with colored cells.

1	1	0	1	1	0
1	1	0	1	1	0
1	1	1	1	1	0
0	0	1	0	0	0
0	0	1	1	1	0
0	0	1	1	1	0

Example 2:

Input Format: The following grid is given:

1	1	0	1	1	0
1	1	0	1	1	0
0	0	0	0	0	0
0	0	0	0	0	0
1	1	1	1	1	0
1	1	1	1	1	0

Result: 11 **Explanation:** We can get the largest group of 11 connected 1s if we change the (3,0) to 1. The groups are shown with colored cells.

1	1	0	1	1	0
1	1	0	1	1	0
0	0	0	0	0	0
1	0	0	0	0	0
1	1	1	1	1	0
1	1	1	1	1	0

Disclaimer: Don't jump directly to the solution, try it out yourself first.

[Problem Link.](#)

Solution:

Before moving on to the solution, let's quickly discuss some points about the question. First, we need to remember that a group means a group of cells with the value 1 such that they share a common side. If we look into it from the matrix view, the statement actually means that two cells with value 1 are considered a single group if one of them is located in any of the four directions (Up, Down, Left, Right) of the other cell. But two diagonal adjacent cells will not be considered a single group rather they will be counted as different groups. The following illustration will depict the concept:

1	1	0	0	0
0	0	1	0	0
0	0	0	0	0
0	1	0	0	0

Here cells [0,0] and [0,1] are considered a single group as they share a common side but cells [0,1] and [1,2] must be considered two different groups as they do not have any common side.

Now, we need to discuss the approach with which we are trying to solve this question. Here, we are selecting the cells with value 0 one at a time, then placing the value 1 to that selected cell and finally, we are trying to connect the cells to get the largest possible group of connected 1's.

Basically, we are checking the largest group of connected 1's we can get by changing each possible cell with the value 0 one at a time.

So, here is a concept of connecting cells as well as dynamically changing the matrix. We can imagine this matrix as a dynamic graph. So, from these observations, we can easily decide to choose the Disjoint Set data structure to solve this problem.

Let's discuss the following observations:

Observation 1: How to connect cells to include them in the same group.

Generally, a cell is represented by two parameters i.e. row and column. But to connect the cells as we have done with nodes, we need to first represent each cell with a single number. So, we will number them from 0 to $n*m-1$ (from left to right) where n = no. of total rows and m = total no. of columns.

For example, if a 5X4 matrix is given we will number the cell in the following way:

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19

Now if we want to connect cells (1,0) and (2,0), we will just perform a union of 5 and 10. The number for representing each cell can be found using the following formula:

node number = (row of the current cell * total number of columns) + column of the current cell
for example, for the cell (2, 0) the number is = $(2*5) + 0 = 10$.

Observation 2: How to find the cell in which if we invert the value, we will get the largest possible group of connected 1s.

In order to find the cell, we will follow the brute force approach. We will check for every possible cell with a value of 0 one by one and we will try to figure out the largest group we can get after inverting that particular cell to 1 in each case. Among all the answers we will find the cell that creates the largest possible group.

Now, with these two observations, the following is our first approach:

We will first invert a cell from the value 0 to 1 and will check all its four adjacent cells(Up, Down, Left, Right). If any component/group exists, we will just connect the current cell to that adjacent component and add the component's size to our answer. Finally, checking all four cells, we will add an extra 1 to our answer for the current cell being included in the group, and then we will get the total size of the newly created group.

But How to get the size of an existing group/component of connected 1s:

In order to get the size of the existing groups, first, we need to create the existing group by connecting the cells with the value 1. To do so we will do a union of the two node numbers calculated using the above-specified formula if the cells contain 1 and they share a common side. Now after connecting all such cells we will get the different existing components.

Now **to find the size of the components**, we will just find their ultimate parents and refer to the ultimate parent index of the size array inside the Disjoint Set data structure(size[ultimateParent]).

Thus we can calculate the size of the components/groups. But there exists an edge case in this approach.

Edge Case:

Here is the edge case. Let's understand it using the following example.

1	1	0	1	1
1	1	0	1	1
1	1	0	1	1
0	0	1	0	0
0	0	1	1	1
0	0	1	1	1

In this given grid, we will check for every cell with the value 0. When we come to cell (3,3), we will check all four adjacent cells to get the components' sizes. Now it will first add the component of size 7 in our answer while checking the left cell and will again add the same component while checking the downward cell. This is where the answer gets incorrect. **So, to avoid this edge case, instead of adding the component sizes to our answer we will store the ultimate parents in a set data structure.** This process will automatically discard the case of adding duplicate components. After that, to get the size of the ultimate parents we will just refer to the ultimate parent index of the size array inside the Disjoint Set data structure(size[ultimateParent]). Thus we will get the final answer.

Approach:

The algorithm steps are as follows (**step 3 is very important**):

1. Our first objective is to connect all the nodes that have formed groups. In order to do so, we will visit each cell of the grid and check if it contains the value 1.
 1. If the value is 1, we will check all four adjacent cells of the current cell. If we find any adjacent cell with the same value 1, we will perform the union(***either unionBySize() or unionByRank()***) of the two node numbers that represent those two cells i.e. the current cell and the adjacent cell.
 2. Now, step 1 is completed.
2. Then, we will again visit each cell of the grid and check if it contains the value 0.
 1. If the value is 0, we will check all four adjacent cells of the current cell. If we found any cell with value 1, we will just insert the ultimate parent of that cell(using the ***findUPar()*** method) in the set data structure. This process will add the adjacent components to our answer.
 2. After doing so for all the adjacent cells containing 1, we will iterate through the set data structure and add the size of each ultimate parent(*referring to the size array inside the Disjoint Set data structure*) to our answer. Finally, we will add an extra 1 to our answer for the current cell being included in the group.
 3. Now, we will compare to get the maximum answer among all the previous answers we got for the previous cells with the value 0 and the current one.
3. But if the matrix does not contain any cell with 0, step 2 will not be executed. For that reason, we will just run a loop from node number 0 to $n \times n$ and for each node number, we will find the ultimate parent. After that, we will find the sizes of those ultimate parents and will take the size of the largest one.
4. Thus we will get the maximum size of the group of connected 1s stored in our answer.

Note: *If you wish to see the dry run of the above approach, you can watch the video attached to this article.*

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

// User function Template for C++
class DisjointSet {

public:
    vector<int> rank, parent, size;
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] = findUPar(parent[node]);
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[ulp_u]) {
            parent[ulp_v] = ulp_u;
        }
        else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (size[ulp_u] < size[ulp_v]) {
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];
        }
        else {
            parent[ulp_v] = ulp_u;
            size[ulp_u] += size[ulp_v];
        }
    }
}

```

```

    }
};

class Solution {
private:
    bool isValid(int newr, int newc, int n) {
        return newr >= 0 && newr < n && newc >= 0 && newc < n;
    }
public:
    int MaxConnection(vector<vector<int>>& grid) {
        int n = grid.size();
        DisjointSet ds(n * n);
        // step - 1
        for (int row = 0; row < n ; row++) {
            for (int col = 0; col < n ; col++) {
                if (grid[row][col] == 0) continue;
                int dr[] = { -1, 0, 1, 0};
                int dc[] = {0, -1, 0, 1};
                for (int ind = 0; ind < 4; ind++) {
                    int newr = row + dr[ind];
                    int newc = col + dc[ind];
                    if (isValid(newr, newc, n) && grid[newr][newc] == 1) {
                        int nodeNo = row * n + col;
                        int adjNodeNo = newr * n + newc;
                        ds.unionBySize(nodeNo, adjNodeNo);
                    }
                }
            }
        }
        // step 2
        int mx = 0;
        for (int row = 0; row < n; row++) {
            for (int col = 0; col < n; col++) {
                if (grid[row][col] == 1) continue;
                int dr[] = { -1, 0, 1, 0};
                int dc[] = {0, -1, 0, 1};
                set<int> components;
                for (int ind = 0; ind < 4; ind++) {
                    int newr = row + dr[ind];
                    int newc = col + dc[ind];
                    if (isValid(newr, newc, n)) {
                        if (grid[newr][newc] == 1) {
                            components.insert(ds.findUPar(newr * n + newc));
                        }
                    }
                }
                int sizeTotal = 0;
                for (auto it : components) {
                    sizeTotal += ds.size[it];
                }
                mx = max(mx, sizeTotal + 1);
            }
        }
    }
}

```

```

        for (int cellNo = 0; cellNo < n * n; cellNo++) {
            mx = max(mx, ds.size[ds.findUPar(cellNo)]);
        }
        return mx;
    }
};

int main() {

    vector<vector<int>> grid = {
        {1, 1, 0, 1, 1, 0}, {1, 1, 0, 1, 1, 0},
        {1, 1, 0, 1, 1, 0}, {0, 0, 1, 0, 0, 0},
        {0, 0, 1, 1, 1, 0}, {0, 0, 1, 1, 1, 0}
    };

    Solution obj;
    int ans = obj.MaxConnection(grid);
    cout << "The largest group of connected 1s is of size: " << ans << endl;
    return 0;
}

```

Output: The largest group of connected 1s is of size: 20

Time Complexity: $O(N^2) + O(N^2) \sim O(N^2)$ where N = total number of rows of the grid. Inside those nested loops, all the operations are taking apparently constant time. So, $O(N^2)$ for the nested loop only, is the time complexity.

Space Complexity: $O(2*N^2)$ where N = the total number of rows of the grid. This is for the two arrays i.e. parent array and size array of size N^2 inside the Disjoint set.

Bridges in Graph – Using Tarjan's Algorithm of time in and low time: G-55

 takeuforward.org/graph/bridges-in-graph-using-tarjans-algorithm-of-time-in-and-low-time-g-55

January 5, 2023

Problem Statement: There are n servers numbered from 0 to $n - 1$ connected by undirected server-to-server connections forming a network where $\text{connections}[i] = [a_i, b_i]$ represents a connection between servers a_i and b_i . Any server can reach other servers directly or indirectly through the network.

A critical connection is a connection that, if removed, will make some servers unable to reach some other servers.

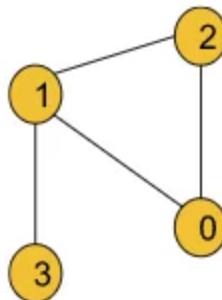
Return all critical connections in the network in any order.

Note: Here servers mean the nodes of the graph. The problem statement is taken from leetcode.

Pre-requisite: [DFS algorithm](#)

Example 1:

Input Format: $N = 4$, $\text{connections} = [[0,1], [1,2], [2,0], [1,3]]$

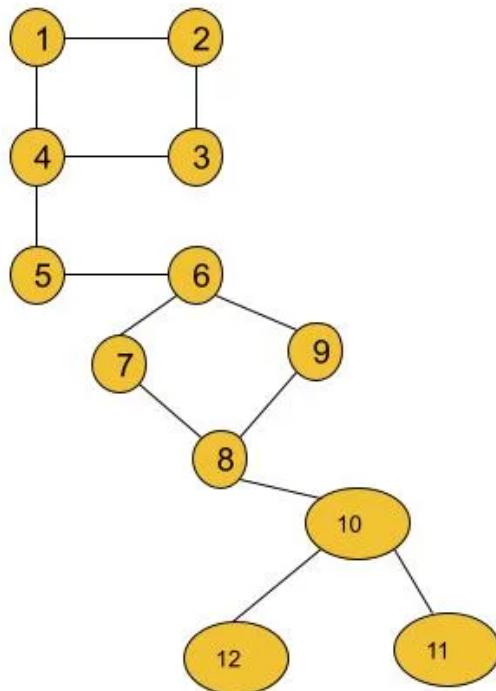


Result: $[[1, 3]]$

Explanation: The edge $[1, 3]$ is the critical edge because if we remove the edge the graph will be divided into 2 components.

Example 2:

Input Format:



Result: [[4, 5], [5, 6], [8, 10]]

Explanation: If we remove any of the three edges, the graph will be divided into 2 or more components.

Solution

Disclaimer: Don't jump directly to the solution, try it out yourself first.

Problem Link.

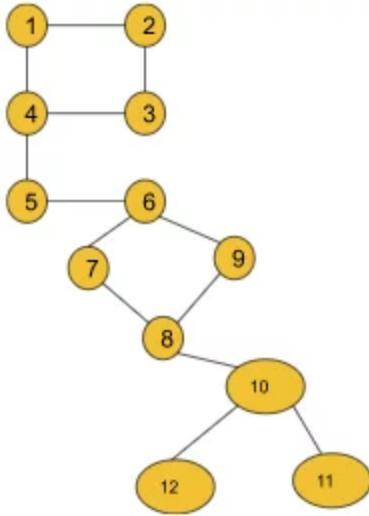
Solution:

Before moving on to the solution, we need to understand the definition of a bridge in a graph.

Bridge:

Any edge in a component of a graph is called a bridge when the component is divided into 2 or more components if we remove that particular edge.

Example:



If in this graph, we remove the edge (5,6), the component gets divided into 2 components. So, it is a bridge. But if we remove the edge (2,3) the component remains connected. So, this is not a bridge. In this graph, we have a total of 3 bridges i.e. (4,5), (5,6), and (10, 8).

In order to find all the bridges of a graph, we will implement some logic over the DFS algorithm. This is more of an algorithm-based approach. So, let's discuss the algorithm in detail. Before that, we will discuss two important concepts of the algorithm i.e. ***time of insertion and lowest time of insertion.***

- **Time of insertion:** During the DFS call, the time when a node is visited, is called its time of insertion. For example, if in the above graph, we start DFS from node 1 it will visit node 1 first then node 2, node 3, node 4, and so on. So, the time of insertion for node 1 will be 1, node 2 will be 2, node 3 will be 3 and it will continue like this. **To store the time of insertion for each node, we will use a time array.**
- **Lowest time of insertion:** In this case, the current node refers to all its adjacent nodes **except the parent** and takes the minimum lowest time of insertion into account. To store this entity for each node, we will use another 'low' array.

The logical modification of the DFS algorithm is discussed below:

After the DFS for any adjacent node gets completed, we will just check if the edge, whose starting point is the current node and ending point is that adjacent node, is a bridge. For that, we will just check if any other path from the current node to the adjacent node exists if we remove that particular edge. If any other alternative path exists, this edge is not a bridge. Otherwise, it can be considered a valid bridge.

Approach:

The algorithm steps are as follows:

1. First, we need to create the adjacency list for the given graph from the edge information(**If not already given**). And we will declare a variable timer(either globally or we can carry it while calling DFS), that will keep track of the time of insertion for each node.
2. Then we will start DFS from node 0(assuming the graph contains a single component otherwise, we will call DFS for every component) with parent -1.
 1. Inside DFS, we will first mark the node visited and then store the time of insertion and the lowest time of insertion properly. The timer may be initialized to 0 or 1.
 2. Now, it's time to visit the adjacent nodes.
 1. **If the adjacent node is the parent itself**, we will just continue to the next node.
 2. **If the adjacent node is not visited**, we will call DFS for the adjacent node with the current node as the parent.
 After the DFS gets completed, we will compare the lowest time of insertion of the current node and the adjacent node and take the minimum one.
 Now, we will check if the lowest time of insertion of the adjacent node is greater than the time of insertion of the current node.
 If it is, then we will store the adjacent node and the current node in our answer array as they are representing the bridge.
 3. **If the adjacent node is already visited**, we will just compare the lowest time of insertion of the current node and the adjacent node and take the minimum one.
 3. Finally, our answer array will store all the bridges.

Note: We are not considering the parent's insertion time during calculating the lowest insertion time as we want to check if any other path from the node to the parent exists excluding the edge we intend to remove.

Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
    int timer = 1;
    void dfs(int node, int parent, vector<int> &vis,
             vector<int> adj[], int tin[], int low[], vector<vector<int>> &bridges) {
        vis[node] = 1;
        tin[node] = low[node] = timer;
        timer++;
        for (auto it : adj[node]) {
            if (it == parent) continue;
            if (vis[it] == 0) {
                dfs(it, node, vis, adj, tin, low, bridges);
                low[node] = min(low[it], low[node]);
                // node --- it
                if (low[it] > tin[node]) {
                    bridges.push_back({it, node});
                }
            } else {
                low[node] = min(low[node], low[it]);
            }
        }
    }
public:
    vector<vector<int>> criticalConnections(int n,
                                              vector<vector<int>>& connections) {
        vector<int> adj[n];
        for (auto it : connections) {
            int u = it[0], v = it[1];
            adj[u].push_back(v);
            adj[v].push_back(u);
        }
        vector<int> vis(n, 0);
        int tin[n];
        int low[n];
        vector<vector<int>> bridges;
        dfs(0, -1, vis, adj, tin, low, bridges);
        return bridges;
    }
};

int main() {

    int n = 4;
    vector<vector<int>> connections = {
        {0, 1}, {1, 2},
        {2, 0}, {1, 3}
    };
}

```

```

Solution obj;
vector<vector<int>> bridges = obj.criticalConnections(n, connections);
for (auto it : bridges) {
    cout << "[" << it[0] << ", " << it[1] << "] ";
}
cout << endl;
return 0;
}

```

Output: [3, 1] (In example 1, [1, 3] and [3, 1] both are accepted.)

Time Complexity: $O(V+2E)$, where V = no. of vertices, E = no. of edges. It is because the algorithm is just a simple DFS traversal.

Space Complexity: $O(V+2E) + O(3V)$, where V = no. of vertices, E = no. of edges. $O(V+2E)$ to store the graph in an adjacency list and $O(3V)$ for the three arrays i.e. tin, low, and vis, each of size V .

*Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, [please check out this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com*



Watch Video At: <https://youtu.be/qrAub5z8FeA>

Articulation Point in Graph: G-56

 takeuforward.org/data-structure/articulation-point-in-graph-g-56

January 5, 2023

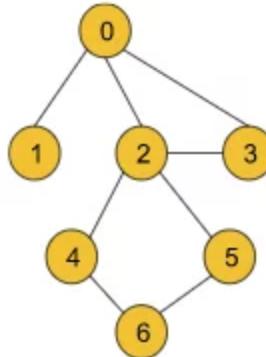
Problem Statement: Given an undirected connected graph with V vertices and adjacency list adj. You are required to find all the vertices removing which (and edges through it) disconnect the graph into 2 or more components.

Note: Indexing is zero-based i.e nodes numbering from (0 to $V-1$). There might be loops present in the graph.

Pre-requisite: [Bridges in Graph](#) problem & [DFS algorithm](#).

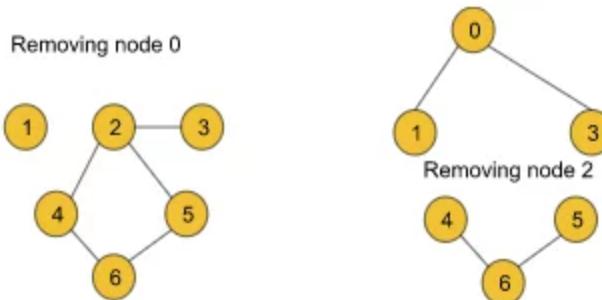
Example 1:

Input Format:



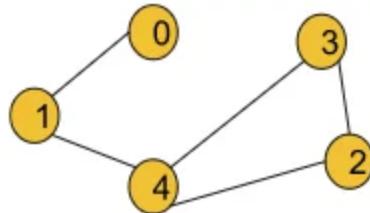
Result: {0, 2}

Explanation: If we remove node 0 or node 2, the graph will be divided into 2 or more components.



Example 2:

Input Format:



Result: {1, 4}

Explanation: If we remove either node 1 or node 4, the graph breaks into multiple components.

Solution

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

[Problem Link](#).

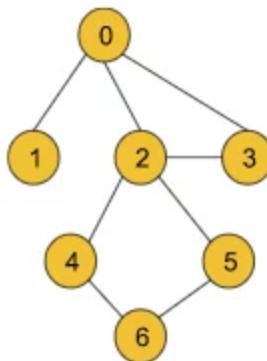
Solution:

Before moving on to the solution, we need to understand the definition of the articulation point of a graph.

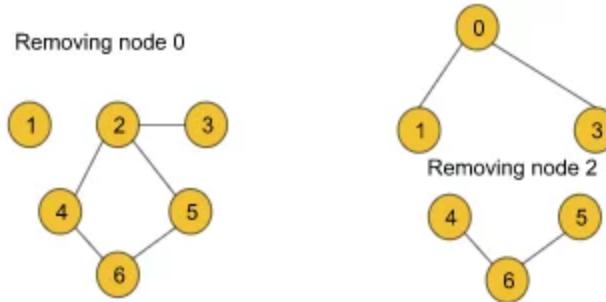
Articulation Point:

Articulation Points of a graph are the nodes on whose removal, the graph breaks into multiple components.

Example:



For the above graph node 0 and node 2 are the articulation points. If we remove either of the two nodes, the graph breaks into multiple components like the following:



But node 3 is not an articulation point as this node's removal does not break the graph into multiple components.

In order to find all the articulation points of a graph, we will implement some logic over the DFS algorithm. This is more of an algorithm-based approach. So, let's discuss the algorithm in detail. Before that, we will discuss the two important concepts of the algorithm i.e. ***time of insertion and lowest time of insertion***.

- **Time of insertion:** During the DFS call, the time when a node is visited, is called its time of insertion. For example, if in the above graph, we start DFS from node 0 it will visit node 1 first then node 2, node 3, and so on. So, the time of insertion for node 0 will be 1, node 1 will be 2, node 2 will be 3 and it will continue like this. **We will use a time array to store the insertion time for each node.**
This definition remains the same as it was during the bridge problem.
- **Lowest time of insertion:** In this case, the current node refers to all its adjacent nodes **except the parent and the visited nodes** and takes the minimum lowest time of insertion into account. To store this entity for each node, we will use another '**low**' array.
The difference in finding the lowest time of insertion in this problem is that in the bridge algorithm, we only excluded the parent node but in this algorithm, we are excluding the visited nodes along with the parent node.

The logical modification of the DFS algorithm is discussed below:

To find out the bridges in the bridge problem, we checked inside the DFS, if there exists any alternative path from the adjacent node to the current node.

But here we cannot do so as in this case, we are trying to remove the current node along with all the edges linked to it. For that reason, here we will check if there exists any path from the adjacent node to the previous node of the current node. ***In addition to that***, we must ensure that the current node we are trying to remove must not be the starting node.

The check conditions for this case will change like the following:

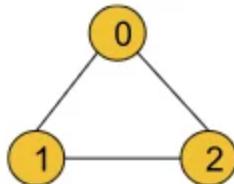
```
if(low[it] > tin[node]) converts to if(low[it] >= tin[node] && parent != -1)
```

For the starting node, we will apply different logic.

The logic for the starting node:

If the node is a starting point we will check the number of children of the node. If the starting node has more than 1 child(The children must not be connected), it will definitely be one of the articulation points.

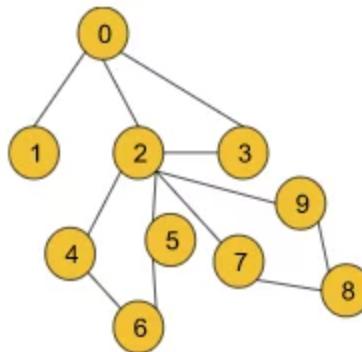
To find the number of children, we will generally count the number of adjacent nodes. But there is a point to notice. In the following graph, the starting node 0 has two adjacent nodes, but it is not an articulation point.



To avoid this edge case, we will increment the number of children only if the adjacent node is not previously visited(*i.e. child++ will be inside the not visited if statement*).

We can get a single node as an articulation point multiple times:

If we carefully observe, we can easily notice that we can get a single node as the articulation point multiple times. For example, consider the following graph:



While checking for node 2, we will get the node as the articulation point once for the first component that contains nodes 4, 5, and 6 and we will again get the same node 2 for the second component that includes the nodes 7, 8, and 9.

To avoid the storing of duplicate nodes, we will store the nodes in a hash array(i.e. mark array used in the code) instead of directly inserting them in a simple array.

Approach:

The algorithm steps are as follows:

1. First, we need to create the adjacency list for the given graph from the edge information(**If not already given**). And we will declare a variable timer(either globally or we can carry it while calling DFS), that will keep track of the time of insertion for each node. The timer may be initialized to 0 or 1 accordingly.
2. Then we will perform DFS for each component. For each component, the starting node will carry -1 as its parent.
 1. Inside DFS, we will first mark the node visited and then store the time of insertion and the lowest time of insertion properly. We will declare a child variable to implement the logic for starting node.
 2. Now, it's time to visit the adjacent nodes.
 1. **If the adjacent node is the parent itself**, we will just continue to the next node.
 2. **If the adjacent node is not visited**, we will call DFS for the adjacent node with the current node as the parent.
 After the DFS gets completed, we will compare the lowest time of insertion of the current node and the adjacent node and take the minimum.
 Now, we will check if the lowest time of insertion of the adjacent node is greater or equal to the time of insertion of the current node and also ensure that the current node is not the starting node(checking parent not equal -1). If the condition matches, then we will mark the current node in our hash array as one of our answers as it is one of the articulation points of the graph.
 Then we will increment the child variable by 1.
 3. **If the adjacent node is visited**, we will just compare the lowest time of insertion of the current node and the time of insertion of the adjacent node and take the minimum.
 3. Finally, we will check if the child value is greater than 1 and if the current node is the starting node. If it is then we will keep the starting node marked in our hash array as the starting node is also an articulation point in this case.
3. Finally, our answer array will store all the bridges.

Note: We are not considering the parent and the visited nodes during calculating the lowest insertion time as they may be the articulation points of the graph which means they may be the nodes we intend to remove.

Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

//User function Template for C++

class Solution {
private:
    int timer = 1;
    void dfs(int node, int parent, vector<int> &vis, int tin[], int low[],
             vector<int> &mark, vector<int> adj[]) {
        vis[node] = 1;
        tin[node] = low[node] = timer;
        timer++;
        int child = 0;
        for (auto it : adj[node]) {
            if (it == parent) continue;
            if (!vis[it]) {
                dfs(it, node, vis, tin, low, mark, adj);
                low[node] = min(low[node], low[it]);
                if (low[it] >= tin[node] && parent != -1) {
                    mark[node] = 1;
                }
                child++;
            }
            else {
                low[node] = min(low[node], tin[it]);
            }
        }
        if (child > 1 && parent == -1) {
            mark[node] = 1;
        }
    }
public:
    vector<int> articulationPoints(int n, vector<int> adj[]) {
        vector<int> vis(n, 0);
        int tin[n];
        int low[n];
        vector<int> mark(n, 0);
        for (int i = 0; i < n; i++) {
            if (!vis[i]) {
                dfs(i, -1, vis, tin, low, mark, adj);
            }
        }
        vector<int> ans;
        for (int i = 0; i < n; i++) {
            if (mark[i] == 1) {
                ans.push_back(i);
            }
        }
        if (ans.size() == 0) return { -1 };
        return ans;
    }
}

```

```

};

int main() {

    int n = 5;
    vector<vector<int>> edges = {
        {0, 1}, {1, 4},
        {2, 4}, {2, 3}, {3, 4}
    };

    vector<int> adj[n];
    for (auto it : edges) {
        int u = it[0], v = it[1];
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    Solution obj;
    vector<int> nodes = obj.articulationPoints(n, adj);
    for (auto node : nodes) {
        cout << node << " ";
    }
    cout << endl;
    return 0;
}

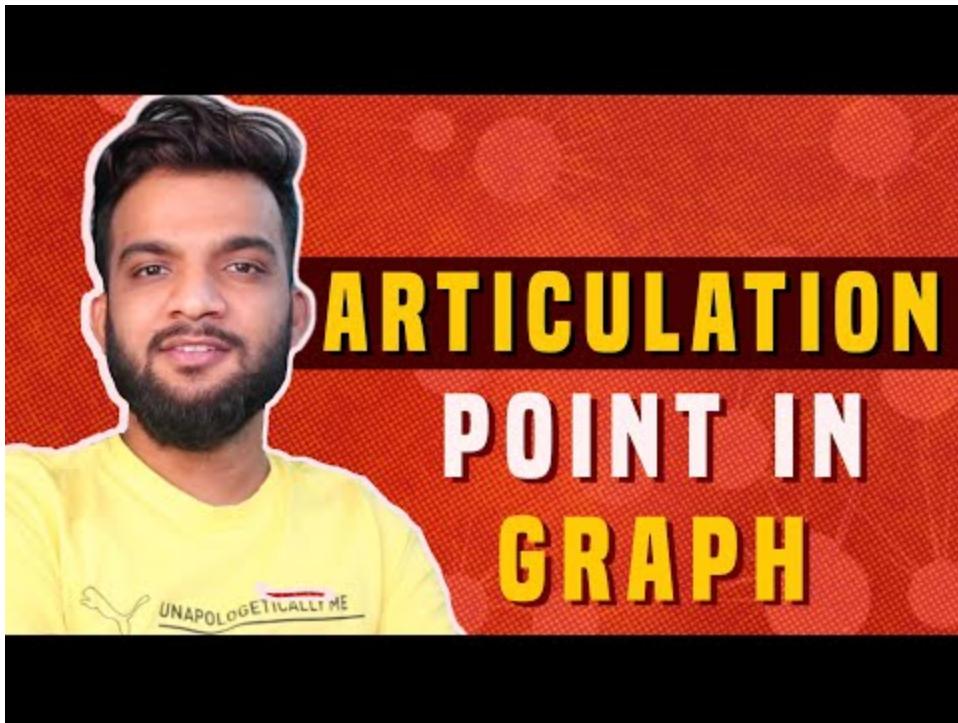
```

Output: 1 4 (Example 2)

Time Complexity: $O(V+2E)$, where V = no. of vertices, E = no. of edges. It is because the algorithm is just a simple DFS traversal.

Space Complexity: $O(3V)$, where V = no. of vertices. $O(3V)$ is for the three arrays i.e. tin, low, and vis, each of size V .

*Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out [this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com*



Watch Video At: <https://youtu.be/j1QDfU21iZk>

Strongly Connected Components – Kosaraju's Algorithm: G-54

 takeuforward.org/graph/strongly-connected-components-kosarajus-algorithm-g-54

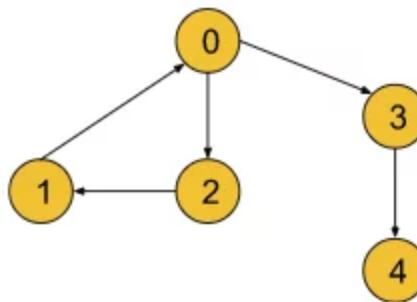
January 5, 2023

Problem Statement: Given a Directed Graph with V vertices (Numbered from 0 to V-1) and E edges, Find the number of strongly connected components in the graph.

Pre-requisite: [DFS algorithm](#)

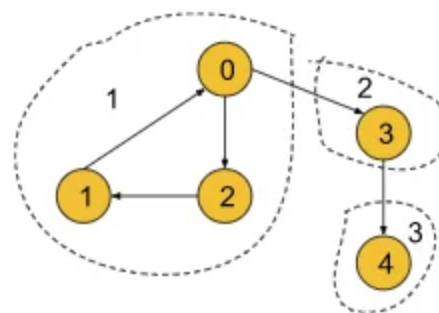
Example 1:

Input Format:



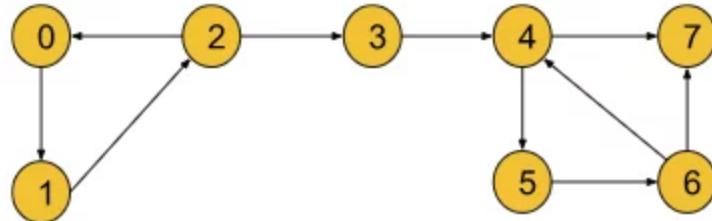
Result: 3

Explanation: Three strongly connected components are marked below:



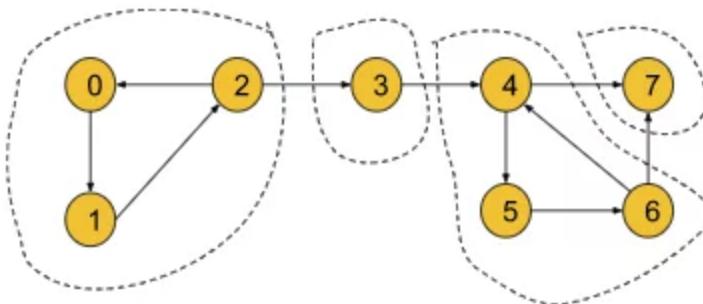
Example 2:

Input Format:



Result: 4

Explanation: Four strongly connected components are marked below:



Solution

Disclaimer: Don't jump directly to the solution, try it out yourself first. [Problem Link](#).

Solution:

In this article, we are going to discuss strongly connected components(SCC) and Kosaraju's algorithm. In an interview, we can expect two types of questions from this topic:

- **Find the number of strongly connected components of a given graph.**
- **Print the strongly connected components of a given graph.**

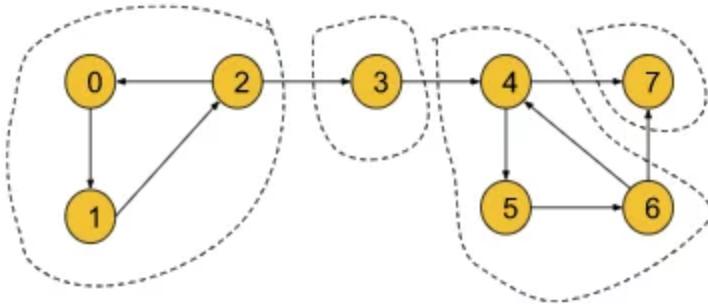
In this article, we are going to discuss the logic part in detail and once the logic part is clear, these two types of questions can be easily solved.

Strongly connected components(SCC) are only valid for directed graphs.

Strongly Connected Component(SCC):

A component is called a Strongly Connected Component(SCC) only if for every possible pair of vertices (u, v) inside that component, u is reachable from v and v is reachable from u .

In the following directed graph, the SCCs have been marked:



If we take 1st SCC in the above graph, we can observe that each node is reachable from any of the other nodes. For example, if take the pair (0, 1) from the 1st SCC, we can see that 0 is reachable from 1 and 1 is also reachable from 0. Similarly, this is true for all other pairs of nodes in the SCC like (0,2), and (1,2). But if we take node 3 with the component, we can notice that for pair (2,3) 3 is reachable from 2 but 2 is not reachable from 3. So, the first SCC only includes vertices 0, 1, and 2.

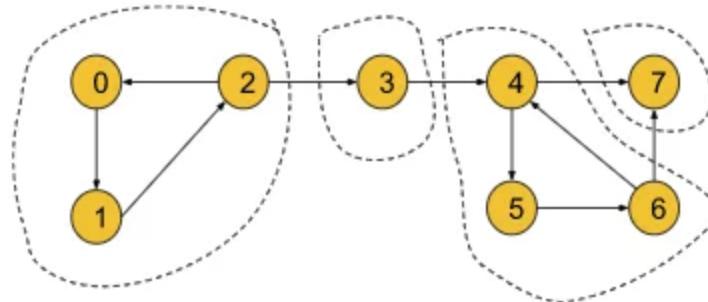
By definition, **a component containing a single vertex is always a strongly connected component**. For that vertex 3 in the above graph is itself a strongly connected component.

By applying this logic, we can conclude that the above graph contains 4 strongly connected components like (0,1,2), (3), (4,5,6), and (7).

Kosaraju's Algorithm:

To find the strongly connected components of a given directed graph, we are going to use Kosaraju's Algorithm.

Before understanding the algorithm, we are going to discuss the thought process behind it. If we start DFS from node 0 for the following graph, we will end up visiting all the nodes. So, it is impossible to differentiate between different SCCs.



Now, we need to think in a different way. We can convert the above graph into the following illustration:



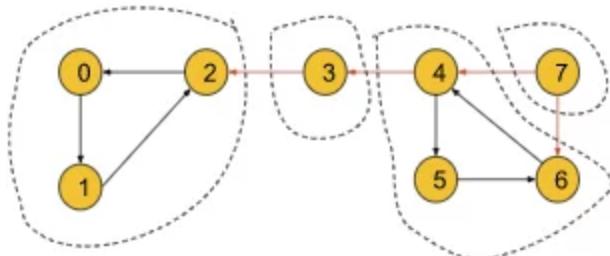
SCC = Strongly Connected Component

By definition, within each SCC, every node is reachable. So, if we start DFS from a node of SCC1 we can visit all the nodes in SCC1 and via edge e1 we can reach SCC2. Similarly, we can travel from SCC2 to SCC3 via e2 and SCC3 to SCC4 via e3. Thus all the nodes of the graph become reachable.

But if we reverse the edges e1, e2, and e3, the graph will look like the following:

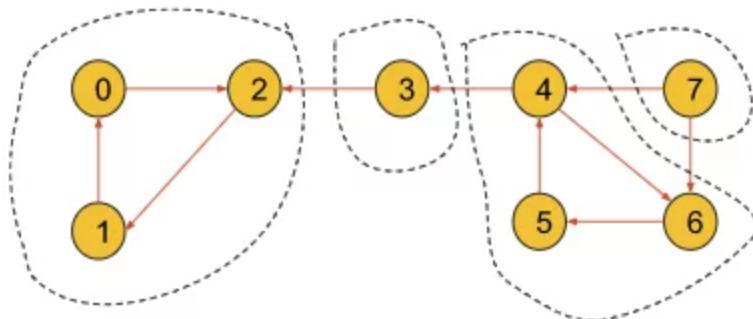


And the original graph will be:



Now in this graph, if we start DFS from node 0 it will visit only the nodes of SCC1. Similarly, if we start from node 3 it will visit only the nodes of SCC2. Thus, by reversing the SCC-connecting edges, the adjacent SCCs become unreachable. Now, the DFS will work in such a way, that in one DFS call we can only visit the nodes of a particular SCC. So, ***the number of DFS calls will represent the number of SCCs.***

Until now, we have successfully found out the process of getting the number of SCCs. But here, comes a new problem i.e. if we do not know the SCCs, how the edges will be reversed? To solve this problem, we will simply try to reverse all the edges of the graph like the following:



If we carefully observe, the nodes within an SCC are reachable from each one to everyone even if we reverse the edges of the SCC. So, the SCCs will have no effect on reversing the edges. Thus we can fulfill our intention of reversing the SCC-connecting edge without affecting the SCCs.

Now, the question might be like, if node 0 is located in SCC4 and we start DFS from node 0, again we will visit all the SCCs at once even after reversing the edges. This is where ***the starting time and the finishing time*** concept will come in.

Now, we have a clear intuition about reversing edges before we move on to the starting and the finishing time concept in the algorithm part.

Algorithm:

The algorithm steps are as follows:

1. Sort all the nodes according to their finishing time:

To sort all the nodes according to their finishing time, we will start DFS from node 0 and while backtracking in the DFS call we will store the nodes in a stack data structure. The nodes in the last SCC will finish first and will be stored in the last of the stack. After the DFS gets completed for all the nodes, the stack will be storing all the nodes in the sorted order of their finishing time.

2. Reverse all the edges of the entire graph:

Now, we will create another adjacency list and store the information of the graph in a reversed manner.

3. Perform the DFS and count the no. of different DFS calls to get the no. of SCC:

Now, we will start DFS from the node which is on the top of the stack and continue until the stack becomes empty. For each individual DFS call, we will increment the counter variable by 1. We will get the number of SCCs by just counting the number of individual DFS calls as in each individual DFS call, all the nodes of a particular SCC get visited.

4. Finally, we will get the number of SCCs in the counter variable. If we want to store the SCCs as well, we need to store the nodes in some array during each individual DFS call in step 3.

Note:

- The first step is to know, from which node we should start the DFS call.
- The second step is to make adjacent SCCs unreachable and to limit the DFS traversal in such a way, that in each DFS call, all the nodes of a particular SCC get visited.
- The third step is to get the numbers of the SCCs. In this step, we can also store the nodes of each SCC if we want to do so.

Note: The sorting of the nodes according to their finishing time is very important. By performing this step, we will get to know where we should start our DFS calls. The top-most element of the stack will finish last and it will surely belong to the SCC1. So, the sorting step is important for the algorithm.

Note: If you wish to see the dry run of the above approach, you can watch the video attached to this article.

Code:

- C++ Code
- Java Code

```

#include <bits/stdc++.h>
using namespace std;

class Solution
{
private:
    void dfs(int node, vector<int> &vis, vector<int> adj[],
             stack<int> &st) {
        vis[node] = 1;
        for (auto it : adj[node]) {
            if (!vis[it]) {
                dfs(it, vis, adj, st);
            }
        }
        st.push(node);
    }
private:
    void dfs3(int node, vector<int> &vis, vector<int> adjT[]) {
        vis[node] = 1;
        for (auto it : adjT[node]) {
            if (!vis[it]) {
                dfs3(it, vis, adjT);
            }
        }
    }
public:
    //Function to find number of strongly connected components in the graph.
    int kosaraju(int V, vector<int> adj[])
    {
        vector<int> vis(V, 0);
        stack<int> st;
        for (int i = 0; i < V; i++) {
            if (!vis[i]) {
                dfs(i, vis, adj, st);
            }
        }

        vector<int> adjT[V];
        for (int i = 0; i < V; i++) {
            vis[i] = 0;
            for (auto it : adj[i]) {
                // i -> it
                // it -> i
                adjT[it].push_back(i);
            }
        }
        int scc = 0;
        while (!st.empty()) {

```

```

        int node = st.top();
        st.pop();
        if (!vis[node]) {
            scc++;
            dfs3(node, vis, adjT);
        }
    }
    return scc;
}

int main() {

    int n = 5;
    int edges[5][2] = {
        {1, 0}, {0, 2},
        {2, 1}, {0, 3},
        {3, 4}
    };
    vector<int> adj[n];
    for (int i = 0; i < n; i++) {
        adj[edges[i][0]].push_back(edges[i][1]);
    }
    Solution obj;
    int ans = obj.kosaraju(n, adj);
    cout << "The number of strongly connected components is: " << ans << endl;
    return 0;
}

```

Output: The number of strongly connected components is: 3 (For example 1)

Time Complexity: $O(V+E) + O(V+E) + O(V+E) \sim O(V+E)$, where V = no. of vertices, E = no. of edges. The first step is a simple DFS, so the first term is $O(V+E)$. The second step of reversing the graph and the third step, containing DFS again, will take $O(V+E)$ each.

Space Complexity: $O(V)+O(V)+O(V+E)$, where V = no. of vertices, E = no. of edges. Two $O(V)$ for the visited array and the stack we have used. $O(V+E)$ space for the reversed adjacent list.

Special thanks to **KRITIDIPTA GHOSH** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, [please check out this article](#). If you want to suggest any improvement/correction in this article please mail us at write4tuf@gmail.com



Watch Video At: <https://youtu.be/R6uoSjZ2imo>