

Graph theory

In progress...

By Mourad NOUAILI

References:

MIT OCW:

https://www.youtube.com/playlist?list=PLUl4u3cNGP61Oq3tWYp6V_F-5jb5L2iHb

www.stackoverflow.com

www.geeksforgeeks.org

Table of Contents

Directed (Oriented) graph (Fig. #1).....	4
Undirected (Non-oriented) graph (Fig. #2).....	4
Algorithmic representation of a graph.....	5
Adjacency matrix.....	6
Undirected graph.....	6
Mathematic formula.....	6
Examples.....	7
Directed graph.....	8
Mathematic formula.....	8
Example.....	8
Adjacency list.....	9
Example.....	9
Activity #1:.....	10
Activity #2:.....	12
Search in a graph.....	14
BFS.....	14
flowchart of the function BFS.....	16
Application: hackerrank's problem.....	17
DFS.....	18
Analysis.....	28
Other codes.....	29
Application: hackerrank's problem.....	31
Motivation.....	32
algorithms.....	32
Some definitions.....	32
Path & shortest path.....	32
Weighted graphs.....	32
How to get all shortest paths from a source vertex.....	33
Relaxation.....	33
Relaxation is safe.....	33
Example: getting all shortest paths.....	34
Negative weights.....	42
Motivation.....	42
Negative cycles.....	43
General structure of shortest path algorithm (no negative cycles).....	44
Optimum structure propriety.....	45
Proof.....	45
Directed acyclic graphs (D.A.Gs.).....	46
Example.....	46
Dijkstra Algorithm.....	49
Pseudo-code.....	49
Dijkstra complexity.....	49
proof of the correctness of the Dijkstra algorithm.....	50
Lemma:.....	50
Application: hackerrank's problem.....	52
speeding up Dijkstra.....	53

Bellman-Ford algorithm.....	55
Observation 1.....	63
Observation 2.....	63
Conclusion.....	63
Editorial by amititkgp.....	67
C++.....	67
Editorial the_lazy_duck.....	70
Find astronauts of the same country.....	70
Compute in how many ways we can pick a pair of astronauts belonging to different countries	70
Solution with formula (André Nicolas on https://math.stackexchange.com/questions/1418652/how-many-ways-to-select-distinct-pairs-from-k-disjoint-sets).....	75

A graph is a set of points named *nodes* or *vertices* (sometimes *cells*) linked by lines or arrows named *edges*.

Directed (Oriented) graph (Fig. #1)

The edges are represented by arrows.

A **directed graph** (or **digraph**) is a graph that is a set of vertices connected by edges, where the edges have a direction associated with them.

Undirected (Non-oriented) graph (Fig. #2)

The edges are represented by lines.

An *undirected graph* is a graph in which edges have no orientation. The edge (x, y) is identical to the edge (y, x)

In mathematics, The set of vertices is denoted by V , and the set of edges is denoted by E . Then the graph can be represented by $G = (V, E)$

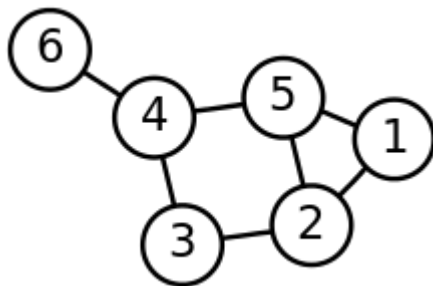


Fig. #2

$G=(V,E)$
 $V = \{1,2,3,4,5,6\}$
 $E = \{\{1,2\}, \{1,5\}, \{2,3\}, \{2,5\}, \{5,4\}, \{3,4\}, \{4,6\}\}$

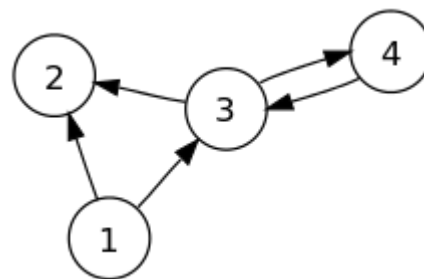


Fig. #1

$G=(V,E)$
 $V=\{1,2,3,4\}$
 $E = \{(1,2), (1,3), (3,2), (3,4), (4,3)\}$

Algorithmic representation of a graph

In algorithmic, we can represent a graph by two ways :

- An adjacency matrix.
- An adjacency list.

Adjacency matrix

Every graph $G = (V, E)$ can be represented by a [matrix](#). The relationship between the vertices and the edges is called incidence relationship, in the [incidence matrix](#) of the graph. The adjacency relationship is established when two vertices are linked by an edge, we can define the adjacency matrix like this :

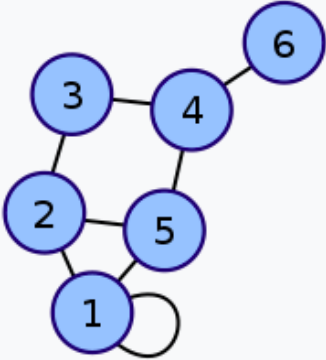
Undirected graph

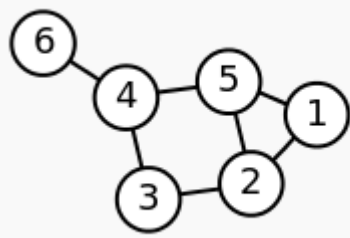
The convention followed here (for undirected graphs) is that each edge adds 1 to the appropriate cell in the matrix, and each loop adds 2. This allows the degree of a vertex to be easily found by taking the sum of the values in either its respective row or column in the adjacency matrix.

Mathematic formula

$$a_{ij} = \begin{cases} 2 & \text{if } (v_i, v_j) \in E \text{ and } i=j \\ 1 & \text{if } (v_i, v_j) \in E \text{ and } i \neq j \\ 0 & \text{else} \end{cases}$$

Examples

Labeled graph	Adjacency matrix
	$\begin{pmatrix} 2 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$ <p>Coordinates are 1-6.</p>

	$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$
---	--

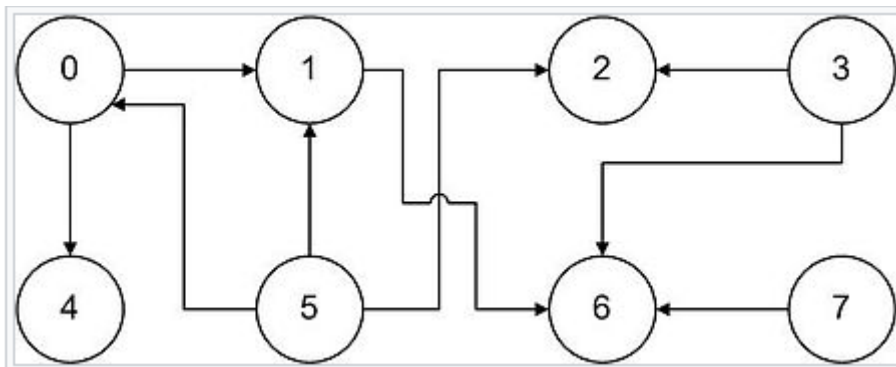
Directed graph

In directed graphs, the [in-degree](#) of a vertex can be computed by summing the entries of the corresponding column, and the out-degree can be computed by summing the entries of the corresponding row.

Mathematic formula

$$a_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{else} \end{cases}$$

Example



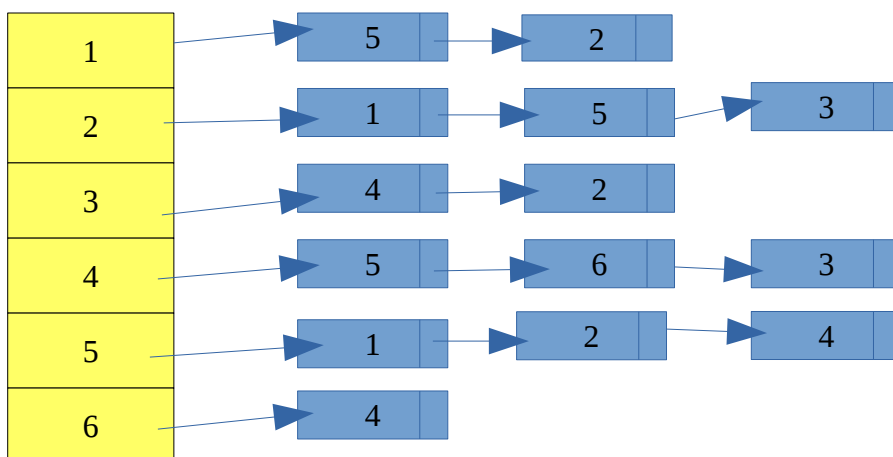
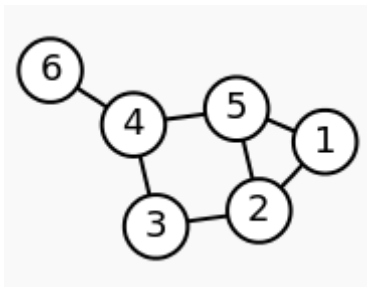
$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

Adjacency list

It's an array of $|V|$ linked lists, where each list describes the set of neighbors of a vertex in the graph.

$$adjList[u] = \{ v \in V \mid (u, v) \in E \}$$

Example



Activity #1:

Write a C++ program that display the adjacency matrix of a undirected graph.

INPUT

First line two spaced integers: ***nbVertices nbEdges*** .

In the second line, ***nbEdges*** lines, in each line, there are two spaced integers which represent the tow linked vertices by an edge.

OUT PUT :

The adjacency matrix.

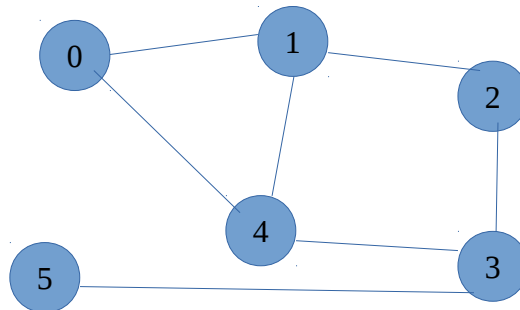
Example :

INPUT :

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

OUTPUT :

```
0 1 0 0 1 0
1 0 1 0 1 0
0 1 0 1 0 0
0 0 1 0 1 1
1 1 0 1 0 0
0 0 0 1 0 0
```



Code C++ :

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
void createAdjMatrix(vector<vector<int> > & matrix , int nbEdges) {  
    for (int edge = 0 ; edge < nbEdges ; ++edge) {  
        int vertex1 , vertex2;  
        cin >> vertex1 >> vertex2;  
        if (vertex1 == vertex2)  
            matrix[vertex1][vertex2] = 2;  
        else {  
            matrix[vertex1][vertex2] = 1;  
            matrix[vertex2][vertex1] = 1;  
        }  
    }  
}
```

```
void displayAdjMatrix(vector<vector<int> > matrix , int nbVertices) {  
    for (int line = 0 ; line < nbVertices ; ++line) {  
        for (int column = 0 ; column < nbVertices ; ++column)  
            cout << matrix[line][column] << ' ';  
        cout << "\n";  
    }  
}
```

```
int main() {  
    int nbVertices , nbEdges;  
    cin >> nbVertices >> nbEdges;  
  
    vector<vector<int> > adjMatrix(nbVertices , vector<int> (nbVertices));  
    createAdjMatrix(adjMatrix,nbEdges);  
    displayAdjMatrix(adjMatrix,nbVertices);  
  
    return 0;  
}
```

Activity #2:

Same stuff in activity #1, but we want to display the adjacency list.

Example :

INPUT :

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

OUTPUT :

```
0-->1 4
1-->0 2 4
2-->1 3
3-->4 2 5
4-->0 1 3
5-->3
```

Code C++ :

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
void createAdjList(vector<list<int> > & adjList , int nbEdges) {  
    for (int edge = 0 ; edge < nbEdges ; ++edge) {  
        int vertex1 , vertex2;  
        cin >> vertex1 >> vertex2;  
        adjList[vertex1].push_back(vertex2);  
        adjList[vertex2].push_back(vertex1);  
    }  
}
```

```
void displayAdjList(vector<list<int> > adjList , int nbVertices) {  
    for (int vertex = 0 ; vertex < nbVertices ; ++vertex) {  
        cout << vertex <<"-->";  
        list<int>::iterator it;  
        for (it = adjList[vertex].begin() ; it != adjList[vertex].end() ; ++it)  
            cout << *it << ' ';  
        cout << '\n';  
    }  
}
```

```
int main() {  
    int nbVertices , nbEdges;  
    cin >> nbVertices >> nbEdges;  
  
    vector<list<int> > adjList(nbVertices);  
    createAdjList(adjList,nbEdges);  
    displayAdjList(adjList,nbVertices);  
  
    return 0;  
}
```

Search in a graph

One of common problem is how to explore a graph in order, for example, to search to shortest path between two nodes, solve a Rubik's cube, web crawling, find friends in a social network, ...

There are two main algorithms:

- Breadth First Search: BFS
- Deep First Search: DFS

BFS

MIT OCW: <https://www.youtube.com/watch?v=s-CYnVz-uh4>

BFS explore a graph layer by layer. For each vertex, explore its neighbors. If a vertex is visited, no need to explore it again.

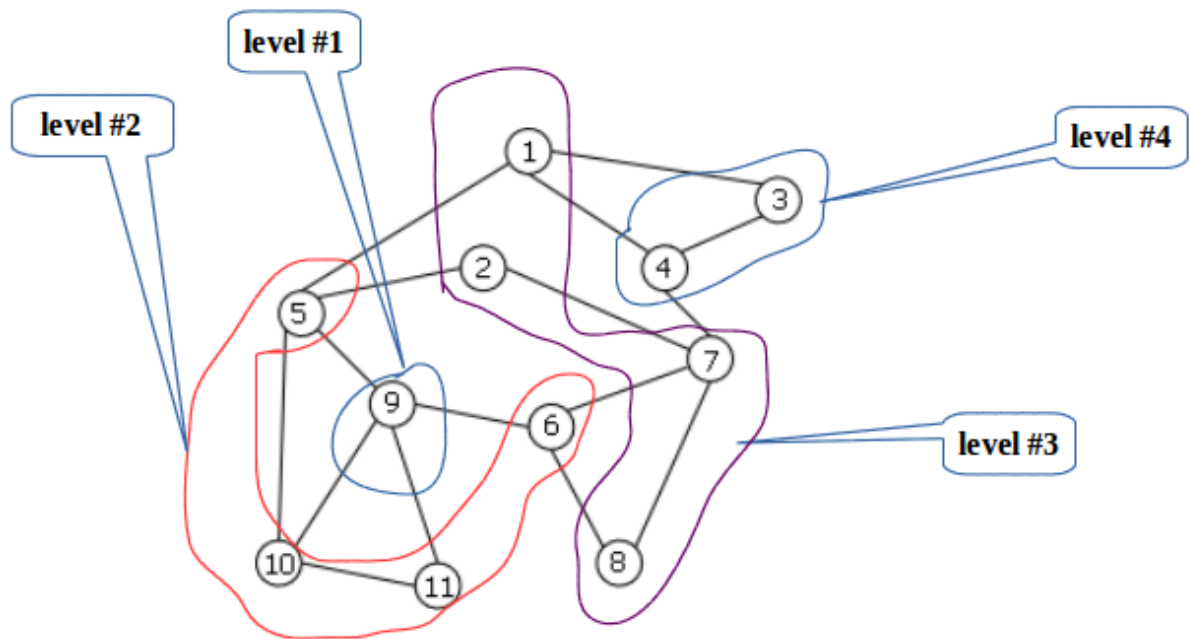
Let's take an example:

- Level #1: The starting point is the vertex number 9.
- Level #2: All the neighbors of 9 (6, 11, 10, 5).
- Level #3: All the neighbors of all vertices from level #2 except the 9:
 - 6's neighbors: 7, 8
 - 11's neighbors: 9, 10 (both are visited)
 - 10's neighbors: 11, 9, 5 (all of them are visited)
 - 5's neighbors: 2, 1

==> The level #3 vertices are: 7, 8, 2, 1

- Level #4: All the neighbors of all vertices from level #3 except those were visited :
 - 7's neighbors: 4
 - 8's neighbors: none
 - 2's neighbors: none
 - 1's neighbors: 3

==> The level #4 vertices are: 4, 3



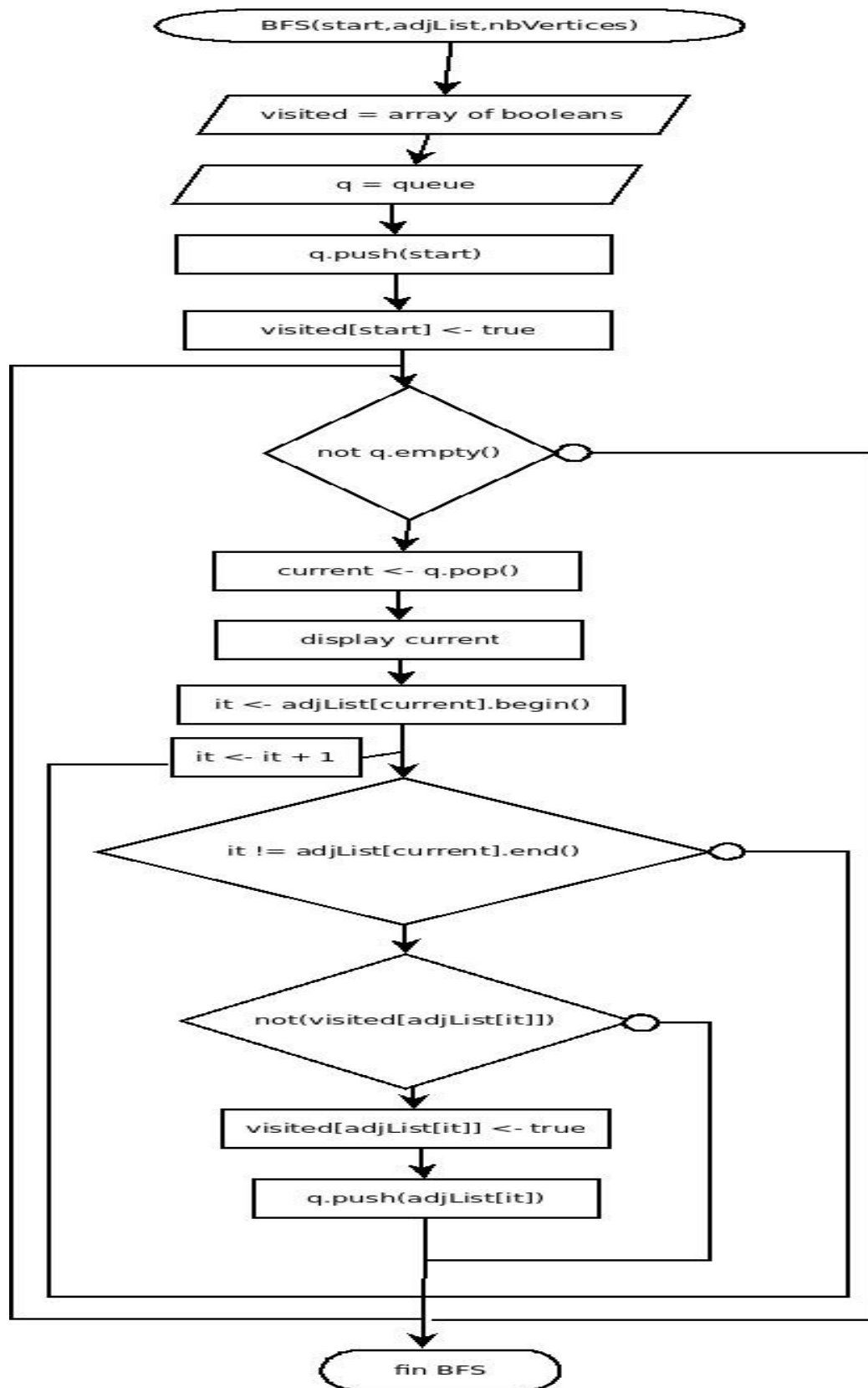
BFS results:

9 5 10 11 6 8 7 1 2 4 3

9 5 6 10 11 1 2 7 8 3 4

....

flowchart of the function BFS



Function BFS's C++ Code : $O(V+E)$

```
void bfs (int start , vector<list<int> > adjList , int nbVertices) {  
    vector<bool> visited(nbVertices);  
    queue<int> q;  
    q.push(start);  
    visited[start] = true;  
    while (!q.empty()) {  
        int current = q.front();  
        cout << current+1 <<' ';  
        q.pop();  
        for (list<int>::iterator it = adjList[current].begin() ; it != adjList[current].end() ; ++it) {  
            if (!visited[*it]) {  
                visited[*it] = true;  
                q.push(*it);  
            }  
        }  
    }  
}
```

Application: hackerrank's problem

Problem name: Breadth First Search: Shortest Reach.

link: <https://www.hackerrank.com/challenges/bfsshortreach/problem>

Editorial: [Breadth First Search: Shortest Reach editorial.](#)

DFS

To solve a maze, for example, you can't use a BFS. Because, you have to do a backtracking when you reach an end. You have to know which edge is visited.

MIT OCW: <https://www.youtube.com/watch?v=AfSk24UTFS8>

Pseudo-code (MIT OCW)

```
parent = {s: None}
dfs-visit(V, adjList, s):
    for v in adjList[s]:
        if v not in parent:
            parent[v] = s
            dfs-visit(V, adjList, v)
```

Visit all vertices reachable for a given source vertex *s*

C++ code:

```
void dfs_visit(vector<list<int> > adjList, int s){
    for (auto it: adjList[s]) {
        int v = it;
        map<int, int>::iterator it1 = parent.find(v);
        if (it1 == parent.end()){
            parent[v] = s;
            dfs_visit(adjList, v);
        }
    }
}
```


A complete C++ code

```
#include <bits/stdc++.h>

using namespace std;

map<int, int> parent;

void createAdjList(vector<list<int> > & adjList , int nbEdges) {
    for (int edge = 0 ; edge < nbEdges ; ++edge) {
        int vertex1 , vertex2;
        cin >> vertex1 >> vertex2;
        vertex1--;
        vertex2--;
        adjList[vertex1].push_back(vertex2);
        adjList[vertex2].push_back(vertex1);
    }
}

void dfs_visit(vector<list<int> > adjList, int s){
    for (auto it: adjList[s]) {
        int v = it;
        map<int, int>::iterator it1 = parent.find(v);
        if (it1 == parent.end()){
            parent[v] = s;
            dfs_visit(adjList, v);
        }
    }
}
```

```

int main() {
    int V, E;
    cin >> V >> E;

    vector<list<int> > adjList(V);
    createAdjList(adjList, E);

    int s;
    cin >> s;

    s--;

    parent.insert({s, NAN});

    dfs_visit(adjList, s);

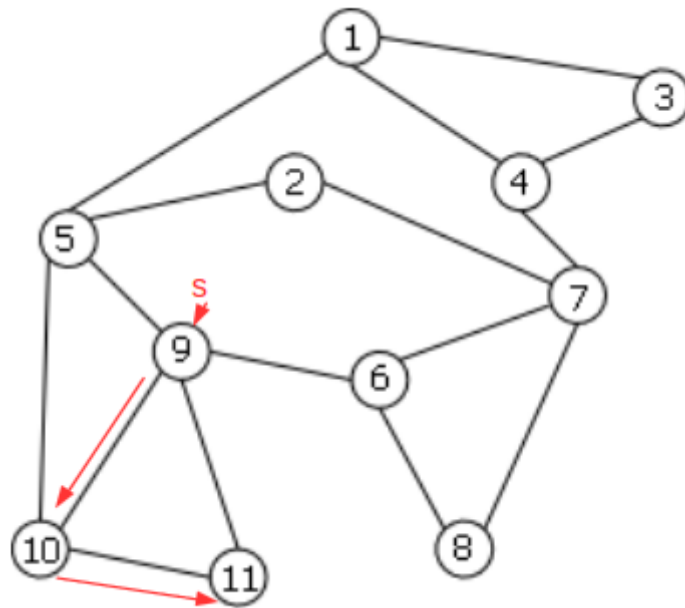
    for (auto it: parent)
        cout << it.first + 1 << ": " << it.second + 1 << "\n";

    return 0;
}

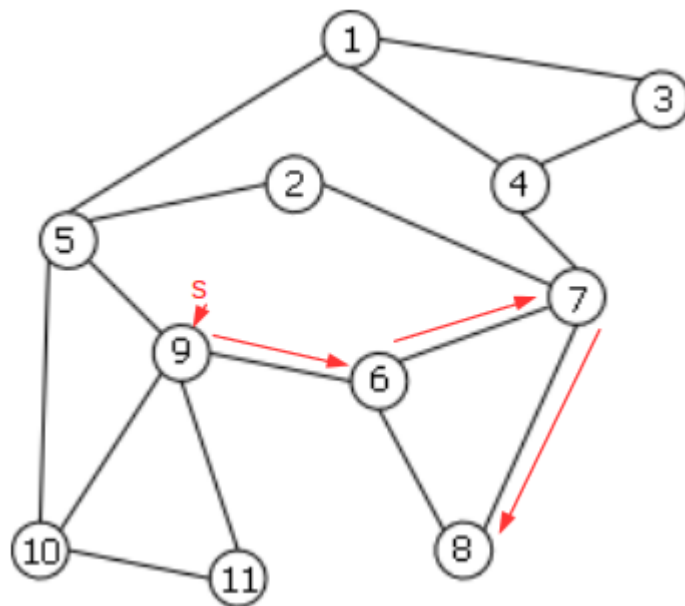
```

INPUT	OUTPUT	The graph
11 15	1: 5	
1 3	2: 7	
1 4	3: 1	
3 4	4: 3	
3 4	5: 2	
1 5	6: 9	
2 5	7: 6	
2 7	8: 7	
4 7	9: -2147483647	
7 6	10: 9	
6 9	11: 10	
5 9		
7 8		
6 8		
9 10		
9 11		
10 11		
9		

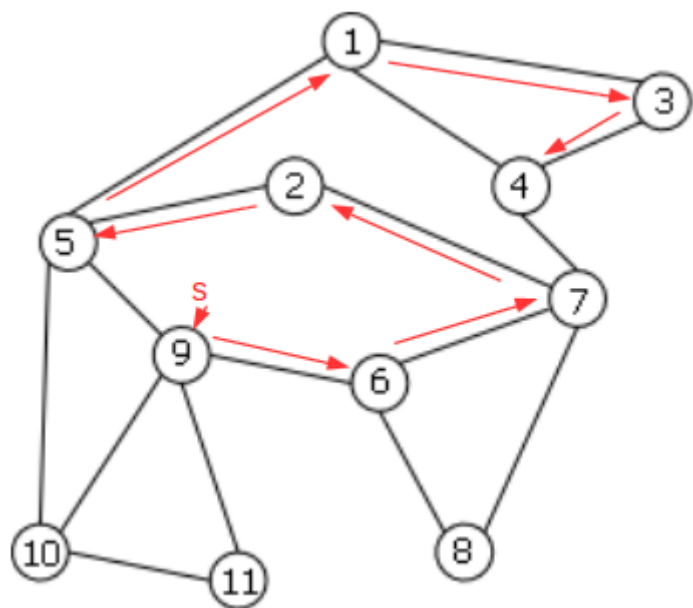
Starting from 9, we have 3 ways (there are others):
 $9 \rightarrow 10 \rightarrow 11$



$9 \rightarrow 6 \rightarrow 7 \rightarrow 8$



$9 \rightarrow 6 \rightarrow 7 \rightarrow 2 \rightarrow 5 \rightarrow 1 \rightarrow 3 \rightarrow 4$



Pseudo-code (MIT OCW)

```
dfs(V, adjList):  
  for s in V:  
    parent={}  
    if s not in parent:  
      parent[s] = None  
      dfs-visit(V, adjList, s)
```

Visit all vertices reachable for a all vertices.

dfs function C++ code

```
void dfs(int V, vector<list<int> > adjList){  
  for (int s = 0 ; s < V ; ++s){  
    cout << "Vertex #" << s+1 << ":\n";  
    parent.clear();  
    map<int, int>::iterator it1 = parent.find(s);  
    if (it1 == parent.end()){  
      parent[s] = NAN;  
      dfs_visit(adjList, s);  
    }  
    for (auto it: parent)  
      cout << it.first + 1 << ": " << it.second + 1 << "\n";  
    cout << "*****\n";  
  }  
}
```

Code

```
#include <bits/stdc++.h>

using namespace std;

map<int, int> parent;

void createAdjList(vector<list<int> > & adjList , int nbEdges) {
    for (int edge = 0 ; edge < nbEdges ; ++edge) {
        int vertex1 , vertex2;
        cin >> vertex1 >> vertex2;
        vertex1--;
        vertex2--;
        adjList[vertex1].push_back(vertex2);
        adjList[vertex2].push_back(vertex1);
    }
}

void dfs_visit(vector<list<int> > adjList, int s){
    for (auto it: adjList[s]) {
        int v = it;
        map<int, int>::iterator it1 = parent.find(v);
        if (it1 == parent.end()){
            parent[v] = s;
            dfs_visit(adjList, v);
        }
    }
}

void dfs(int V, vector<list<int> > adjList){
    for (int s = 0 ; s < V ; ++s){
        cout << "Vertex #" << s+1 <<":\n";
        parent.clear();
        map<int, int>::iterator it1 = parent.find(s);
        if (it1 == parent.end()){
            parent[s] = NAN;
            dfs_visit(adjList, s);
        }
        for (auto it: parent)
            cout << it.first + 1 << ": " << it.second + 1 << "\n";
        cout << "*****\n";
    }
}
```

```

int main() {
    int V, E;
    cin >> V >> E;

    vector<list<int> > adjList(V);
    createAdjList(adjList, E);

    dfs(V, adjList);

    return 0;
}

```

OUTPUT

Vertex #1:

```

1: 1
2: 7
3: 1
4: 3
5: 2
6: 9
7: 4
8: 6
9: 5
10: 9
11: 10

```

Vertex #2:

```

1: 5
2: 1
3: 1
4: 3
5: 2
6: 7
7: 4
8: 6
9: 6
10: 9
11: 10

```

Vertex #3:

```

1: 3

```

2: 7
3: 1
4: 1
5: 2
6: 9
7: 4
8: 6
9: 5
10: 9
11: 10

Vertex #4:

1: 4
2: 5
3: 1
4: 1
5: 1
6: 7
7: 2
8: 6
9: 6
10: 9
11: 10

Vertex #5:

1: 5
2: 7
3: 1
4: 3
5: 1
6: 7
7: 4
8: 6
9: 6
10: 9
11: 10

Vertex #6:

1: 5
2: 7

3: 1
4: 3
5: 2
6: 1
7: 6
8: 7
9: 5
10: 9
11: 10

Vertex #7:

1: 5
2: 7
3: 1
4: 3
5: 2
6: 9
7: 1
8: 6
9: 5
10: 9
11: 10

Vertex #8:

1: 5
2: 7
3: 1
4: 3
5: 2
6: 9
7: 8
8: 1
9: 5
10: 9
11: 10

Vertex #9:

1: 5
2: 7
3: 1

4: 3
 5: 2
 6: 9
 7: 6
 8: 7
 9: 1
 10: 9
 11: 10

Vertex #10:

1: 5
 2: 7
 3: 1
 4: 3
 5: 2
 6: 9
 7: 6
 8: 7
 9: 10
 10: 1
 11: 9

Vertex #11:

1: 5
 2: 7
 3: 1
 4: 3
 5: 2
 6: 9
 7: 6
 8: 7
 9: 11
 10: 9
 11: 1

Analysis

- Visit each vertex once in DFS alone $O(V)$
- dfs-visit called once per vertex v (visit $\text{adjList}[v]$) $\Rightarrow O\left(\sum_{v \in V} \text{adjList}[v]\right) = O(E)$

The whole time complexity is: $\theta(V+E)$

Other codes

Code that print all reachable vertices from a start vertex

```
#include <bits/stdc++.h>

using namespace std;

int V, E;
list<int> *adjList = NULL;

void createAdjList(int nbEdges) {
    for (int edge = 0 ; edge < nbEdges ; ++edge) {
        int vertex1 , vertex2;
        cin >> vertex1 >> vertex2;

        vertex1--;
        vertex2--;

        adjList[vertex1].push_back(vertex2);
        adjList[vertex2].push_back(vertex1);
    }
}

void dfs_visit(int v, bool visited[]){
    //Mark the current node as visited and
    // print it
    visited[v] = true;
    cout << v+1 << " ";
    // Recur for all the vertices adjacent
    // to this vertex
    list<int>::iterator i;
    for (i = adjList[v].begin(); i != adjList[v].end(); ++i)
        if (!visited[*i])
            dfs_visit(*i, visited);
}

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

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

```

int main() {

    cin >> V >> E;

    adjList = new list<int>[V];

    createAdjList(E);

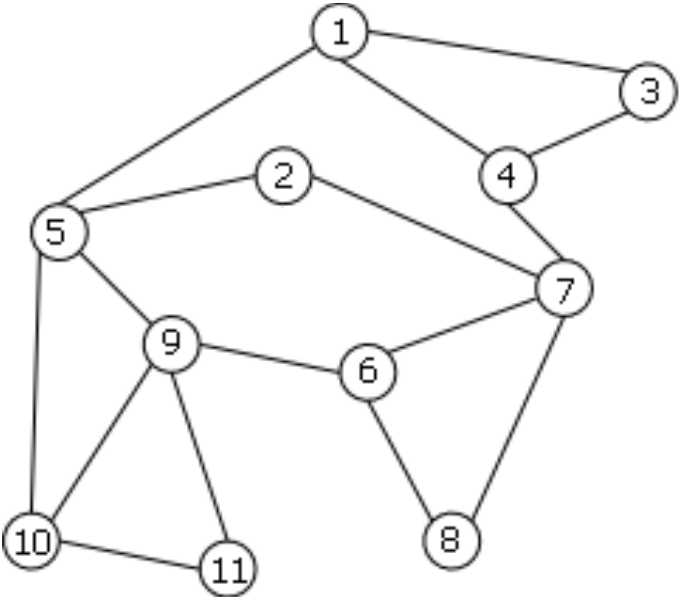
    int start;

    scanf("%d", &start);
    dfs(--start);

    printf("\n");

    return 0;
}

```

Display all reachable vertices from vertex 9		
INPUT	OUTPUT	The graph
11 15 1 3 1 4 3 4 1 5 2 5 2 7 4 7 7 6 6 9 5 9 7 8 6 8 9 10 9 11 10 11 9	9 6 7 2 5 1 3 4 8 10 11	

Code C++ iterative version :

same principle as BFS using a stack instead a queue.

```
void dfs-visit (int start , vector<list<int> > adjList , int nbVertices) {  
    vector<bool> visited(nbVertices);  
    stack<int> s;  
    s.push(start);  
    visited[start] = true;  
    while (!s.empty()) {  
        int current = s.top();  
        cout << current+1 << ' ' ;  
        s.pop();  
        for (list<int>::iterator it = adjList[current].begin() ; it != adjList[current].end() ; ++it) {  
            if (!visited[*it]) {  
                visited[*it] = true;  
                s.push(*it);  
            }  
        }  
    }  
}
```

recursive version :

```
void dfs-visit (int start) {  
    visited[start] = true;  
    cout << start+1 << ' ' ;  
    for (list<int>::iterator it = adjList[start].begin() ; it != adjList[start].end() ; ++it) {  
        if (!visited[*it]) {  
            visited[*it] = true;  
            dfs-visit(*it);  
        }  
    }  
}
```

Application: hackerrank's problem

Problem name: Roads and Libraries.

link: <https://www.hackerrank.com/challenges/torque-and-development/problem>

Editorial: [Roads and Libraries editorial.](#)

Problem name: Journey to the Moon.

link: <https://www.hackerrank.com/challenges/journey-to-the-moon/problem>

Editorial: [Journey to the moon editorial](#)

Single source, many targets shortest path

Motivation

Find the fastest way or the shortest way to get from a source or a start point to a destination or an end point.

The graph must be weighted, in order to have the possibility to compute the shortest path.

In a graph $G(V, E, w)$, $W: E \rightarrow \mathbb{R}$

V : vertices

E : edges

W : weights

algorithms

- Dijkstra:
 - the graph must non-negative weighted edges.
 - Time complexity: $O(V \log V + E)$, in a complete graph $E = O(V^2)$
- Bellman-Ford:
 - Works on positive or negative weighted edges.
 - Time complexity: $O(VE)$

Some definitions

Path & shortest path

A path $p = \langle v_0, v_1, \dots, v_k \rangle$

p is a path, if $(v_i, v_{i+1}) \in E$ for $0 \leq i \leq k-1$

The weight of the path $p = w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$

To find the shortest path, it's just to find a path p that have a minimum weight.

Weighted graphs

$v_0 \xrightarrow{p} v_k$: from v_0 using a path p to v_k .

(v_0) is a single vertex path from v_0 to v_0 of weight 0.

Shortest path from u to v as:

$$\delta(u, v) = \begin{cases} \min \{w(p), u \xrightarrow{p} v\} \exists \text{ any such path} \\ \infty \text{ otherwise} \end{cases}$$

$\delta(u, v)$ is the length of a shortest path.

$d(v)$ is the current weight of v , which means the length of the current path from source to v .

$\pi(v)$: predecessor vertex on the best current path of v , from s to v

$\delta(u, v)$ is:

- The minimum over all the paths $w(p)$ such that p is the path from u to v , if exists any such path.
- Infinity, otherwise.

How to get all shortest paths from a source vertex

Initially, from a the point source, we have $\delta(s, s) = 0$ and $\delta(s, v_i) = \infty$, $0 \leq i \leq \#V$, because we don't know yet the weight from s to v_i .

Then, when going to each vertex reachable from the source, we gonna reduce the $d()$'s values: **Edge relaxation**.

If the vertex is not reachable its value stay infinity.

Relaxation

Reduce $d()$'s values down to $\delta()$'s values. When all vertices have converted to $\delta()$ s, then the algorithm is done.

relax (u, v, w):

if $d[v] > d[u] + w(u, v)$:

$d[v] = d[u] + w(u, v)$

$\pi[v] = u$

Relaxation is safe

By doing the relaxation process, we are sure that we gonna reach the minimum path.

Lemma: The relaxation operation maintains the invariant that $d[v] \geq \delta(s, v)$ for all $v \in V$.

proof: by induction on the number of steps.

Base case

Initially we have $d[v] = \infty$, and $d[s] = 0$. $d[v] \geq \delta(s, v)$ for all $v \in V$ is correct.

Inductive step

Induction hypothesis (I.H.): $d[u] \geq \delta(s, u)$ is correct for n iterations.

Prove that the I.H. will be correct for the next iteration.

By triangle-inequality:

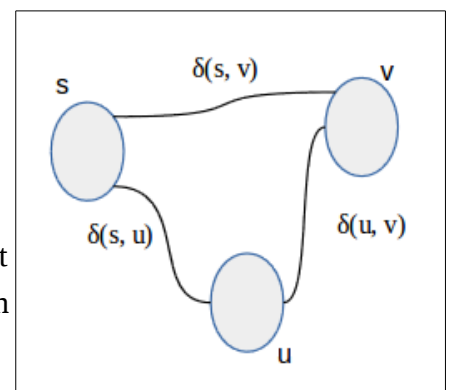
In the triangle inequality, we can substitute:

- $\delta(s, u)$ by $d[u]$ since $d[u] \geq \delta(s, u) \Rightarrow \delta(s, v) \leq d[u] + \delta(u, v)$
- $\delta(u, v)$ by $w(u, v)$ since $w(u, v) \geq \delta(u, v)$, $w(u, v)$ is the weight of edge (u, v) it can be never smaller that the the shortest path from u to v ($\delta(u, v)$)

$\Rightarrow \delta(s, v) \leq \delta(s, u) + \delta(u, v) \Rightarrow \delta(s, v) \leq d[u] + w(u, v)$

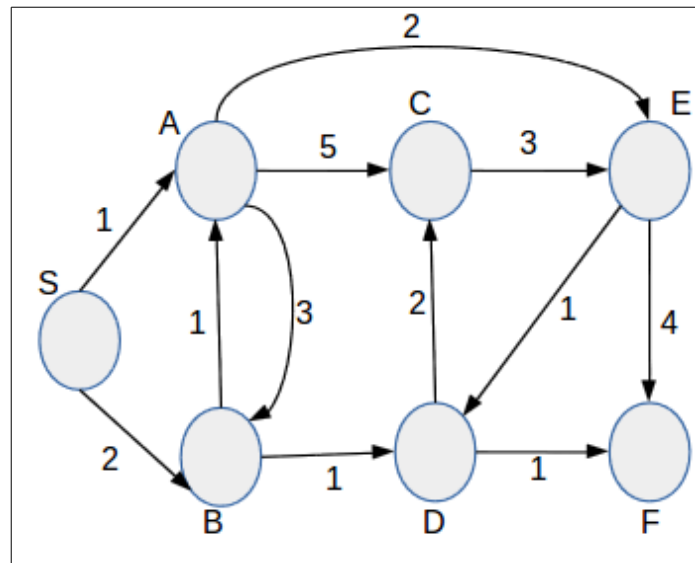
$d[u] + w(u, v)$ is the relaxation of edge $(u, v, w) \Rightarrow d[u] + w(u, v) = d[v]$

$\Rightarrow \delta(s, v) \leq d[v]$



Example: getting all shortest paths

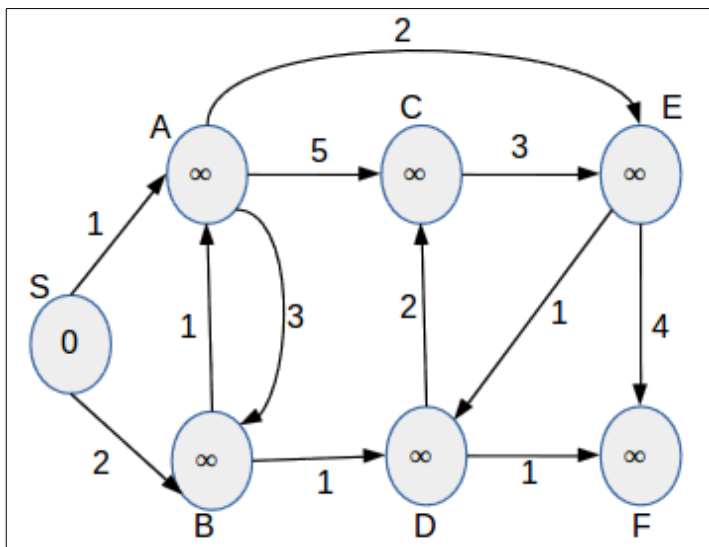
Let's take this graph:



- Initially:

$d(S) = 0, d(A) = \infty, d(B) = \infty, d(C) = \infty, d(D) = \infty, d(E) = \infty, d(F) = \infty$

$\pi(S) = \text{null}, \pi(A) = \text{null}, \pi(B) = \text{null}, \pi(C) = \text{null}, \pi(D) = \text{null}, \pi(E) = \text{null}, \pi(F) = \text{null}$



d:

0	∞	∞	∞	∞	∞	∞
S	A	B	C	D	E	F

π :

NULL	NULL	NULL	NULL	NULL	NULL	NULL
S	A	B	C	D	E	F

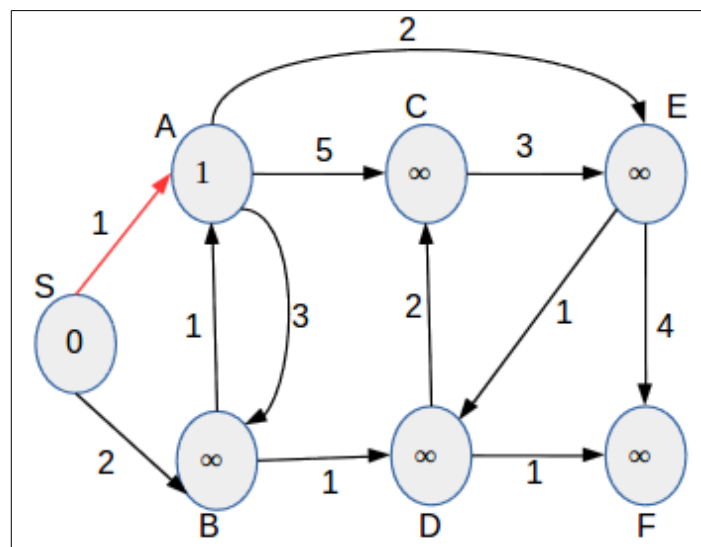
- from S:
 - Going to A, relax (S, A):
 $d[A] > d[S] + w(S, A) \Rightarrow \min \{1, \infty\} = 1 \Rightarrow d[A] = 1, \pi[A] = S$

d:

0	1	∞	∞	∞	∞	∞
S	A	B	C	D	E	F

π :

NULL	S	NULL	NULL	NULL	NULL	NULL
S	A	B	C	D	E	F



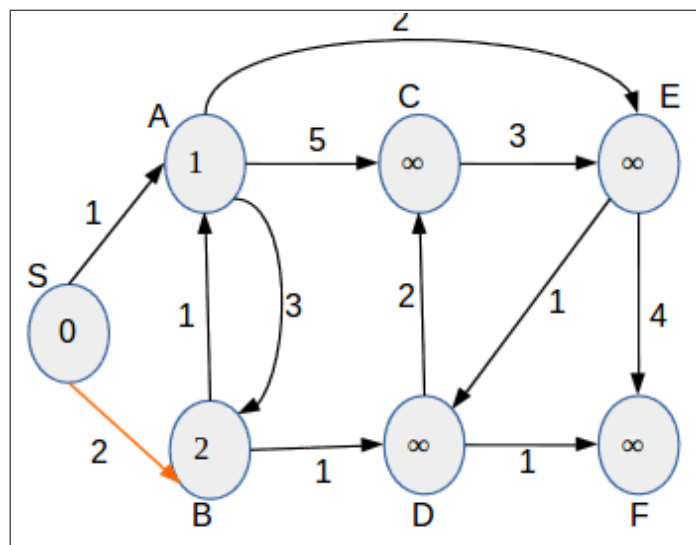
- Going to B, , relax (S, B):
 $d[B] > d[S] + w(S, B) \Rightarrow \min \{2, \infty\} = 2 \Rightarrow d[B] = 2, \pi[B] = S$

d:

0	1	2	∞	∞	∞	∞
S	A	B	C	D	E	F

π :

NULL	S	S	NULL	NULL	NULL	NULL
S	A	B	C	D	E	F



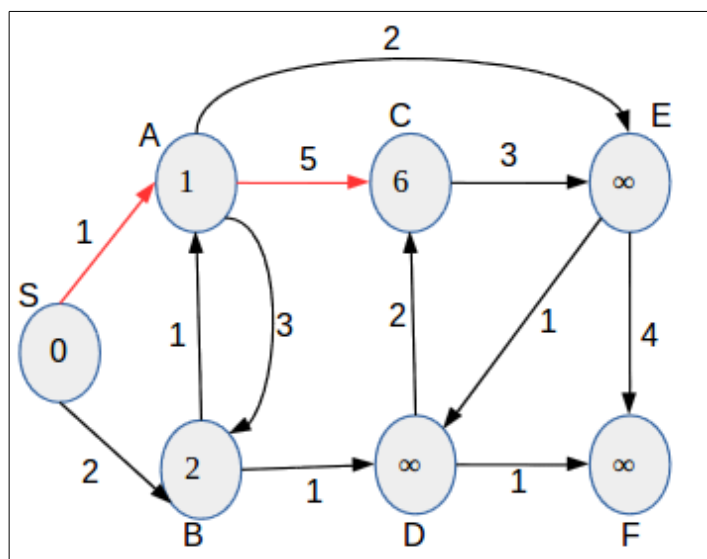
- Going to C (throw A), relax(A, C):
 $d[C] > d[A] + w(A, C) \Rightarrow \min \{6, \infty\} = 6 \Rightarrow d[C] = 6, \pi[C] = A$

d:

0	1	2	6	∞	∞	∞
S	A	B	C	D	E	F

π :

NULL	S	S	A	NULL	NULL	NULL
S	A	B	C	D	E	F



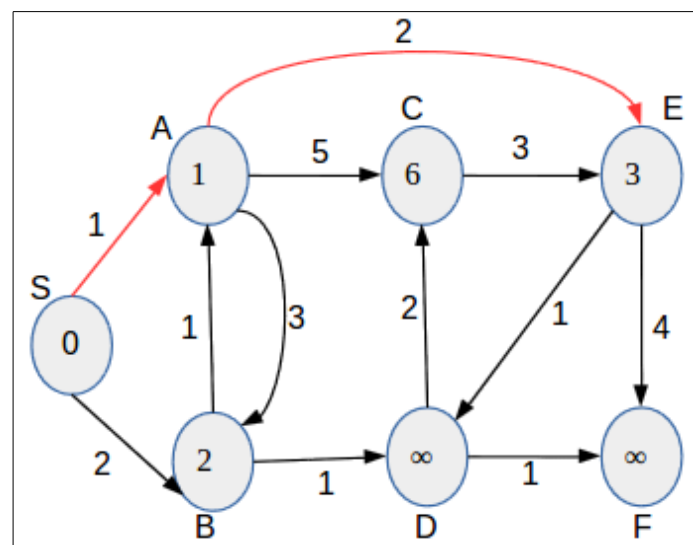
- Going to E (throw A) , relax(A, E):
 $d[E] > d[A] + w(A, E) \Rightarrow \min \{3, \infty\} = 3 \Rightarrow d[E] = 3, \pi[E] = A$

d:

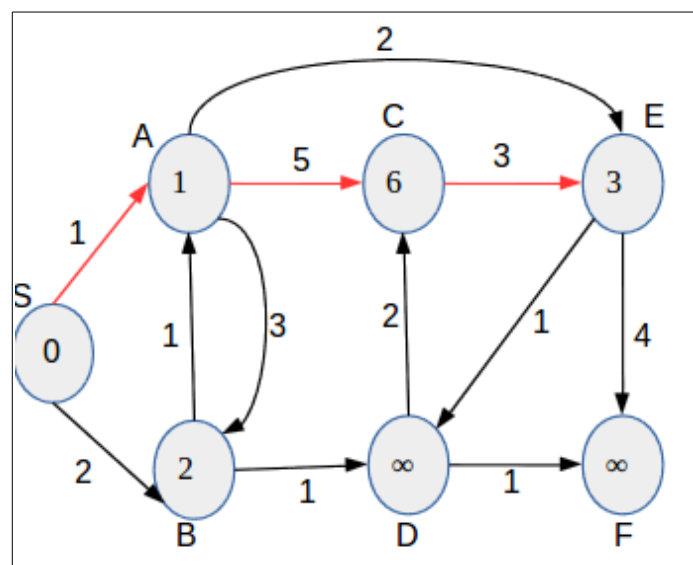
0	1	2	6	∞	3	∞
S	A	B	C	D	E	F

π :

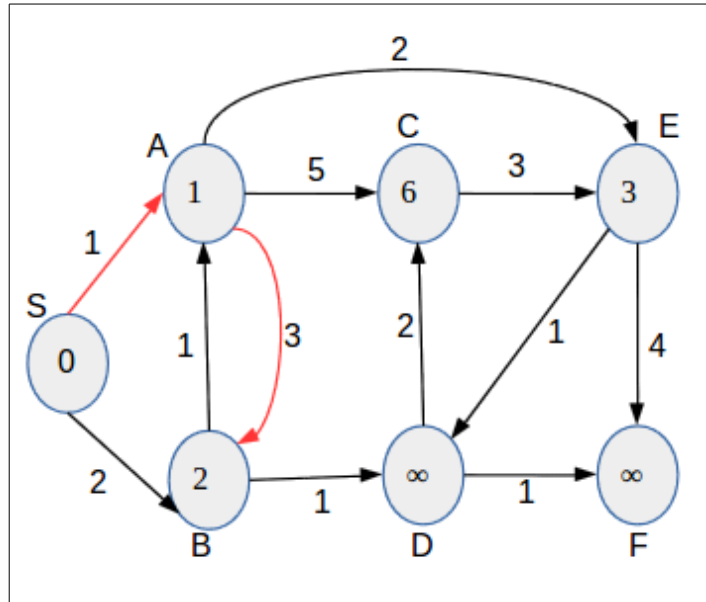
NULL	S	S	A	NULL	A	NULL
S	A	B	C	D	E	F



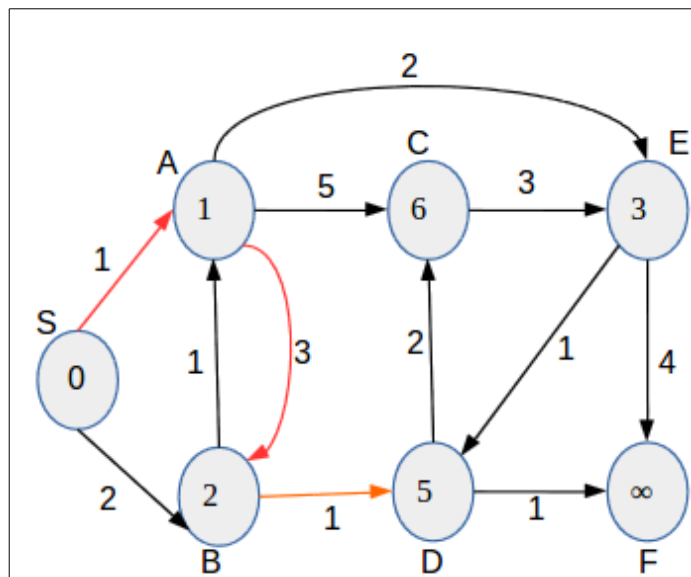
- Going to E (throw A and C), relax (C, E):
 $d[E] > d[C] + w(C, E) \Rightarrow \min \{9, 3\} = 3 \Rightarrow d[E] \text{ stay } 3, \pi[E] \text{ stay } A$
 In other words, this path is not taken:



- Going to B (throw A), relax(A, B):
 $d[B] > d[A] + w(A, B) \Rightarrow \min \{2, 4\} = 2 \Rightarrow d[B] \text{ stay } 2, \pi[B] \text{ stay } S.$
 This path is not taken:



- Going to D (throw A and B), relax(B, D):
 $d[D] > d[B] + w(B, D) \Rightarrow \min \{\infty, 5\} = 5 \Rightarrow d[D] = 5, \pi[D] = B.$



d:

0	1	2	6	5	3	∞
<i>S</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	<i>F</i>

π :

NULL	S	S	A	B	A	NULL
<i>S</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	<i>F</i>

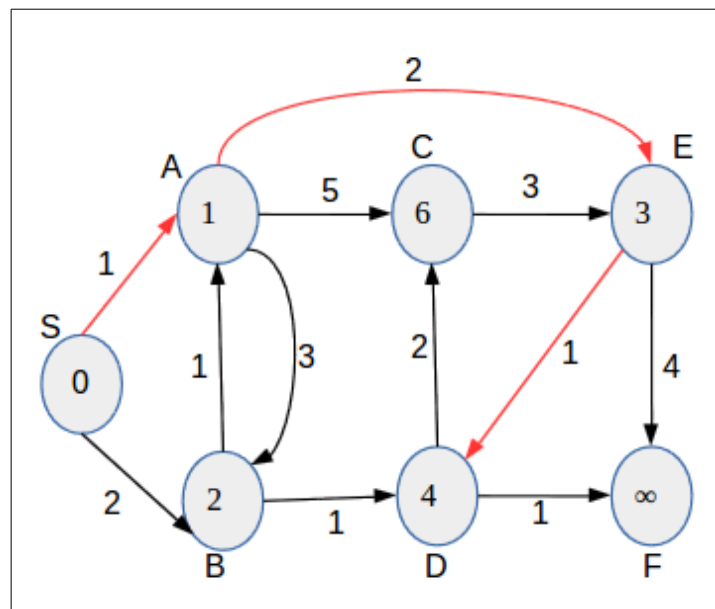
- Going to D (throw A, C and E), relax(E, D):
 $d[D] > d[E] + w(E, D) \Rightarrow \min \{5, 4\} = 4 \Rightarrow d[D] = 4, \pi[D] = E.$

d:

0	1	2	6	4	3	∞
<i>S</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	<i>F</i>

π :

NULL	S	S	A	E	A	NULL
<i>S</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	<i>F</i>



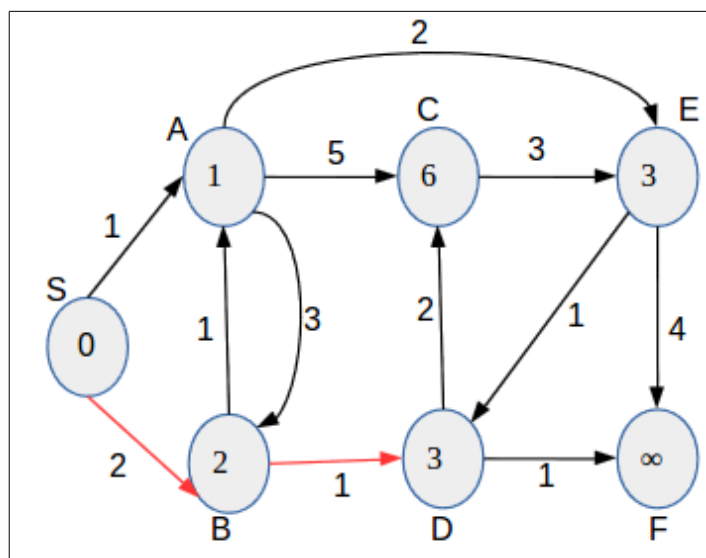
- Going to D (trow B), relax(B, D):
 $d[D] > d[B] + w(B, D) \Rightarrow \min \{4, 3\} = 3 \Rightarrow d[D] = 3, \pi[D] = B.$

d:

0	1	2	6	3	3	∞
<i>S</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	<i>F</i>

π :

NULL	S	S	A	B	A	NULL
<i>S</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	<i>F</i>



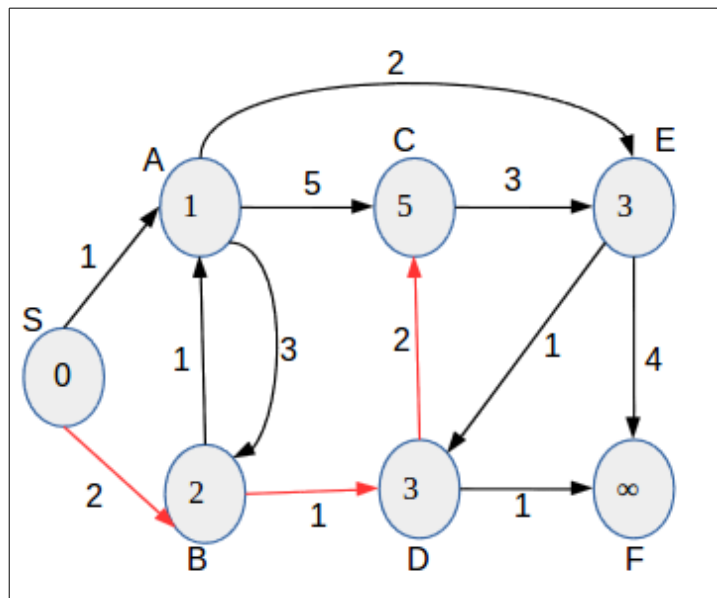
- Going to C (trow B and D), relax(D, C):
 $d[C] > d[D] + w(D, C) \Rightarrow \min \{6, 5\} = 5 \Rightarrow d[C] = 5, \pi[C] = D.$

d:

0	1	2	5	3	3	∞
<i>S</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	<i>F</i>

π :

NULL	S	S	D	B	A	NULL
<i>S</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	<i>F</i>



And so on, until we get all δ (s)

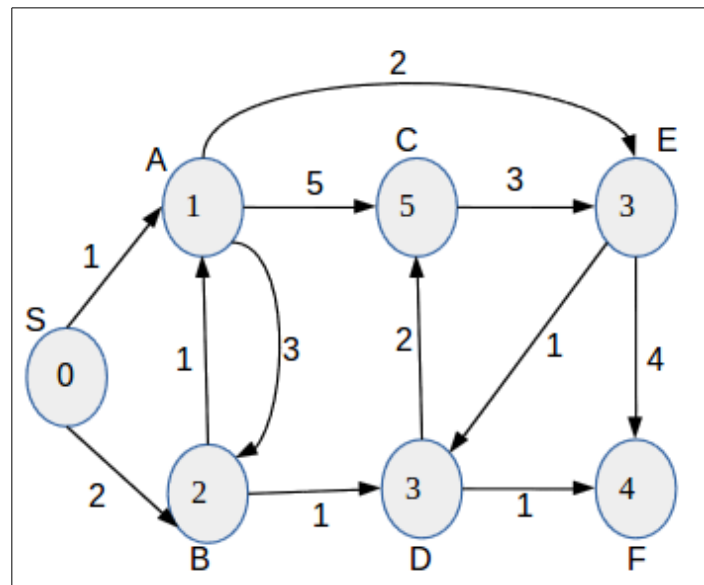
d:

0	1	2	5	3	3	4
<i>S</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	<i>F</i>

π :

NULL	S	S	D	B	A	D
<i>S</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	<i>F</i>

$\delta(S, A) = 1$
 $\delta(S, B) = 2$
 $\delta(S, C) = 5$
 $\delta(S, D) = 3$
 $\delta(S, E) = 3$
 $\delta(S, F) = 4$



To reconstruct a path,

just check the π structure.

For example, what's the shortest path from S to F:

path = "F"

$\pi[F] = D \Rightarrow$ path = "FD"

$\pi[D] = B \Rightarrow$ path = "FDB"

$\pi[B] = S \Rightarrow$ path = "FDBS"

$\pi[B] = \text{NULL}$, we finish building and the path is "SBDF" of length $d[F] = 4$.

Negative weights

Motivation

In some cases, we need a graph with negative weights. For example: social networks (#likes, #dislikes), ...

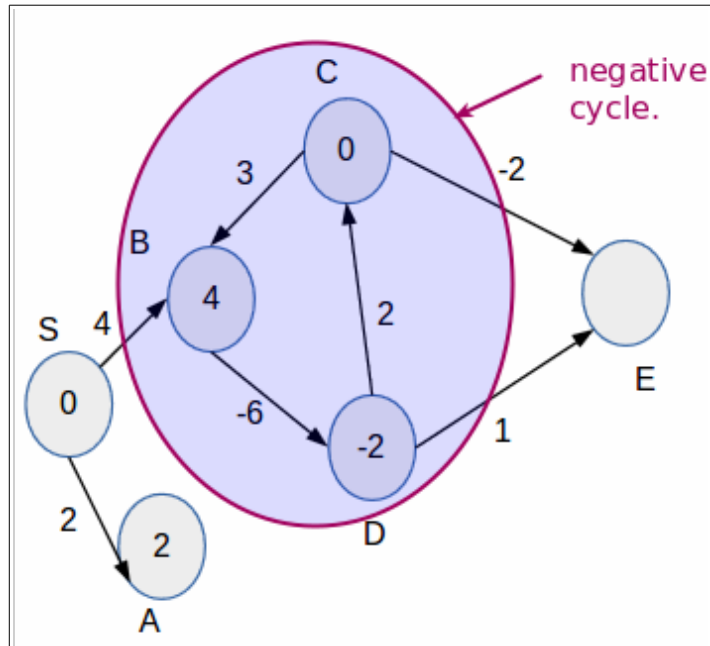
To make easy, try, if possible, to shift all weights to positive without compromising the problem specification. In order to use an $O(V \log V + E)$ algorithm (Dijkstra) instead of $O(VE)$ (Bellman-Ford).

Negative cycles

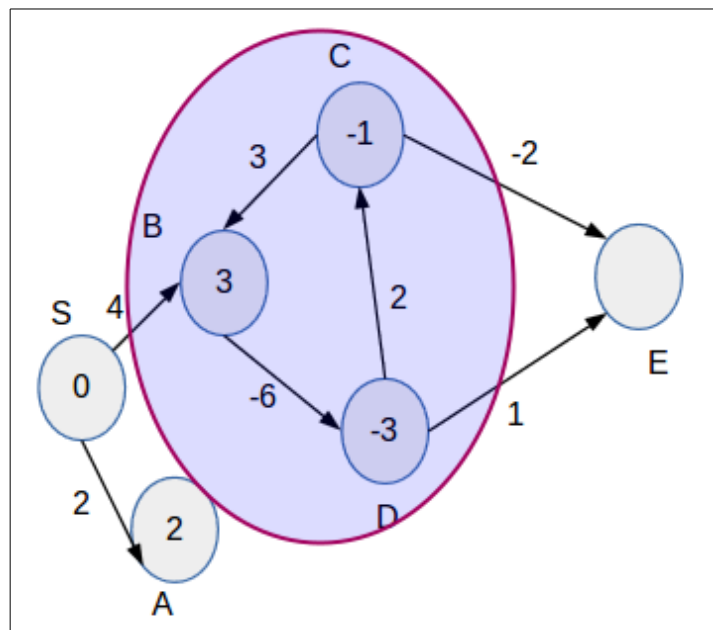
When looking for the shortest paths from a single source, your algorithm could have a bug once dealing with negative cycle as show in the example bellow:

$\delta(S, A) = 2$ (No problem with that edge)

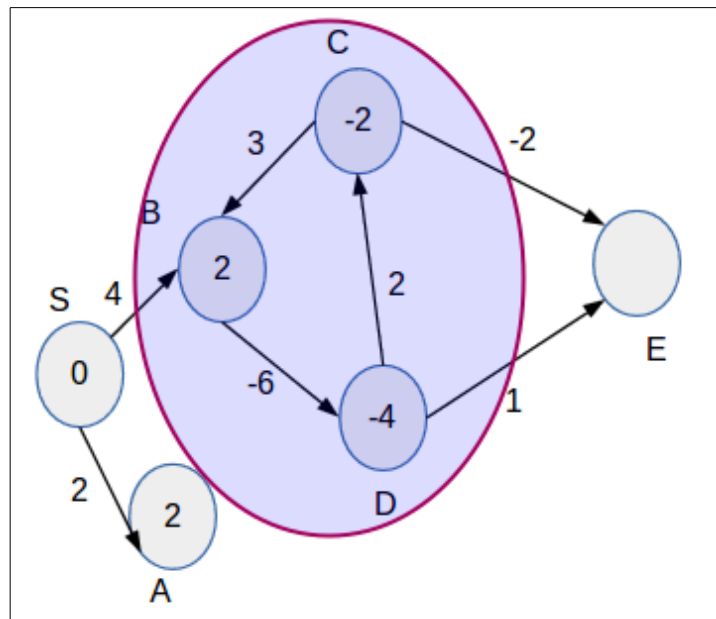
$d[B] = 4$
 $d[D] = -2$
 $d[C] = 0$



$d[B] = 3$
 $d[D] = -3$
 $d[C] = -1$



$d[B] = 2$
 $d[D] = -4$
 $d[C] = -2$



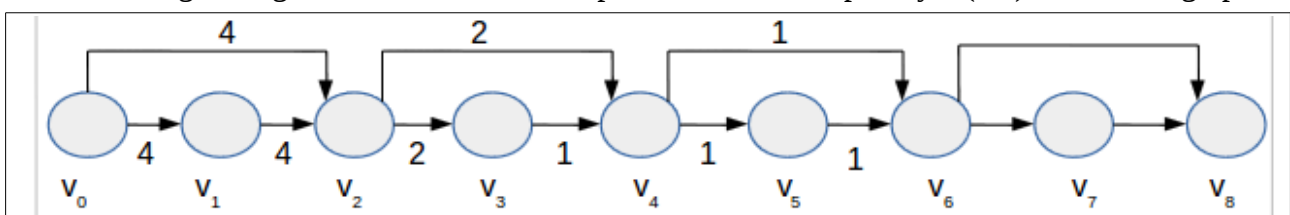
As you see, with negative cycles, the algorithm could do an infinite loop. Negative cycles makes the computation of the shortest paths lengths indeterminable. If the graph have negative weights, it could have negative cycles too. We must use an algorithm that handle with that (like Bellmen-Ford, that mark the vertices of negative cycles indeterminable)

General structure of shortest path algorithm (no negative cycles)

```

Initialize for all  $u \in V$ :
     $d[v] \leftarrow \infty, \pi[v] \leftarrow \text{nil}$ 
 $d[s] \leftarrow 0$ 
repeat:
    select some edge( $u, v$ ) [Some how]
    relax edge( $u, v, w$ )
until all edges have  $d[v] \leq d[u] + w(u, v)$ 
  
```

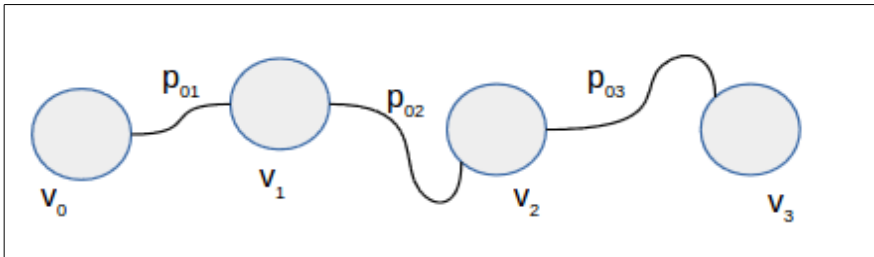
This is a brute force algorithm, which is very bad. **We must be careful of how to select**, if we select wrong the algorithm could have an exponential time complexity $O(2^{n/2})$, like is this graph:



also , this algorithm will not terminate if there is a negative weight cycle reachable from the source.

Optimum structure propriety

Subpaths of a shortest path are shortest paths.



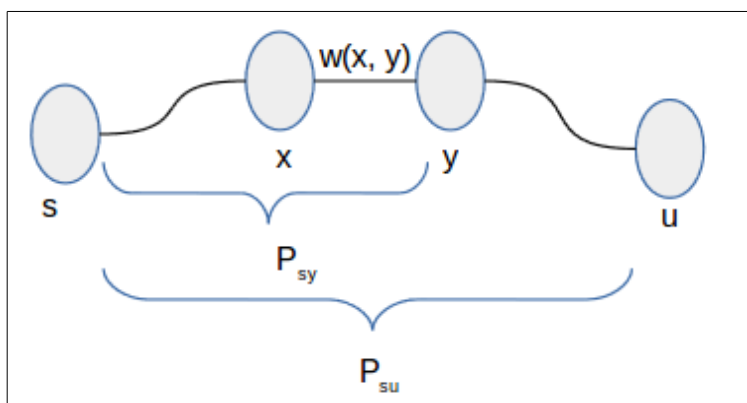
If the path that concatenate the subpaths p_{01} , p_{02} and p_{03} is a shortest path, then each p_{01} , p_{02} and p_{03} are shortest paths.

Proof

Let P_{su} is the shortest path from s to u passing by a vertex x and a vertex y , such as y is adjacent to x .

P_{sy} is the path from s to y . We must prove that P_{sy} is the shortest path from s to y .

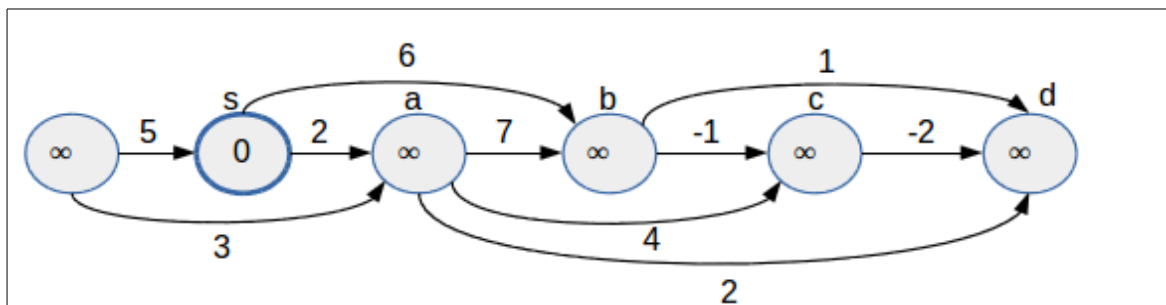
P_{su} is the shortest path. That means when reaching the vertex x , the edge (x, y) will be relaxed. As we proved that the relaxation is safe, the $d[y]$ will be reduced until $d[y] = \delta(s, y)$ (Because $P_{sy} \in P_{su}$, and P_{su} is the shortest path)



Directed acyclic graphs (D.A.Gs.)

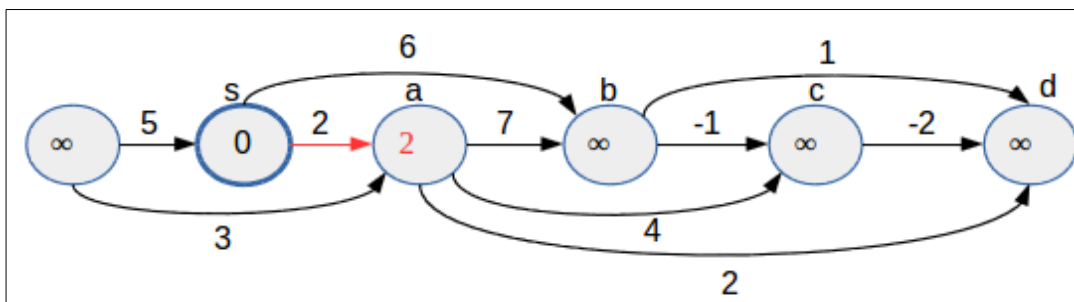
- can have negative edges, but not negative cycles.
- Topological sort the DAG. Path from u to v implies that u is before v in the linear ordering.
- One pass over vertices in topologically sorted order relaxing each edge that leaves each vertex.
- Time complexity is $O(V+E)$

Example



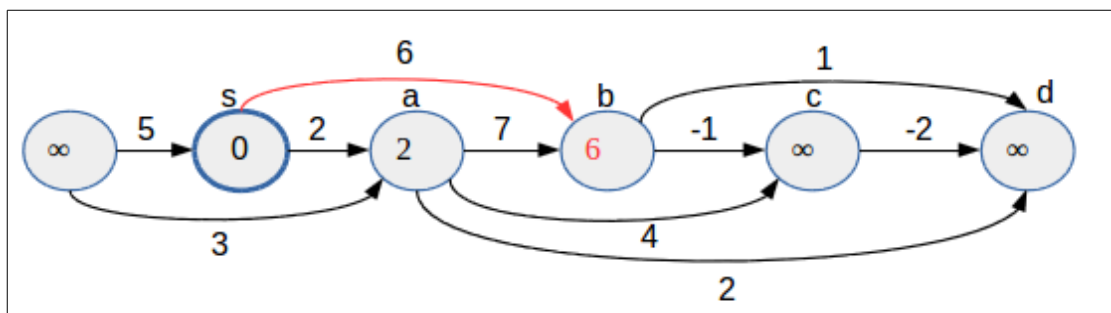
relax(s, a, 2):

$$d[a] \geq d[s] + w(s, a) \Rightarrow \infty \geq 0 + 2 \Rightarrow d[a] \leftarrow 2$$



relax(s, b, 6):

$$d[b] \geq d[s] + w(s, b) \Rightarrow \infty \geq 0 + 6 \Rightarrow d[b] \leftarrow 6$$



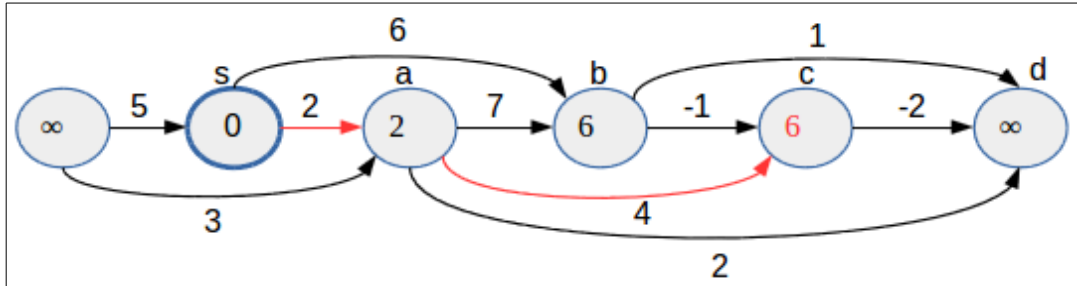
s is done, go to a

realx(a, b, 7):

$d[b] \geq d[a] + w(a, b) \Rightarrow 6 \geq 2 + 7 \Rightarrow$ nothing to do.

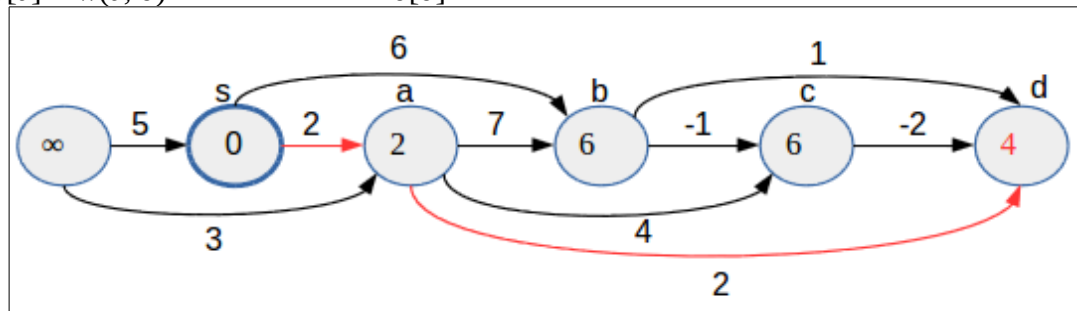
realx(a, c, 4):

$d[c] \geq d[a] + w(a, c) \Rightarrow \infty \geq 2 + 4 \Rightarrow d[c] \leftarrow 6$



realx(a, d, 2):

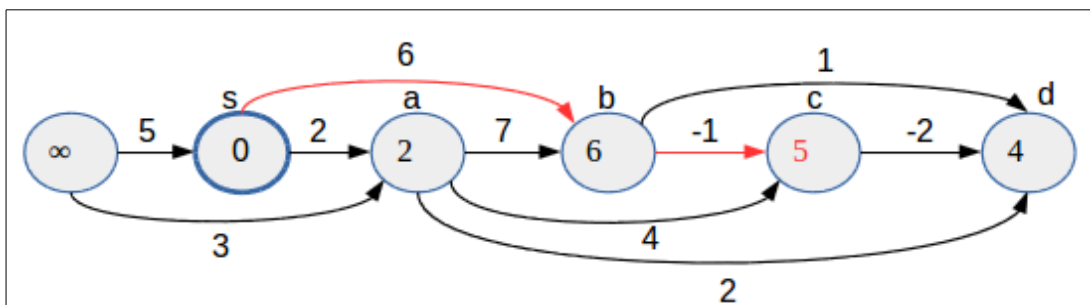
$d[d] \geq d[a] + w(a, d) \Rightarrow \infty \geq 2 + 2 \Rightarrow d[d] \leftarrow 4$



a is done, going to b

realx(b, c, -1):

$d[c] \geq d[b] + w(b, c) \Rightarrow 6 \geq 6 + (-1) \Rightarrow d[c] \leftarrow 5$



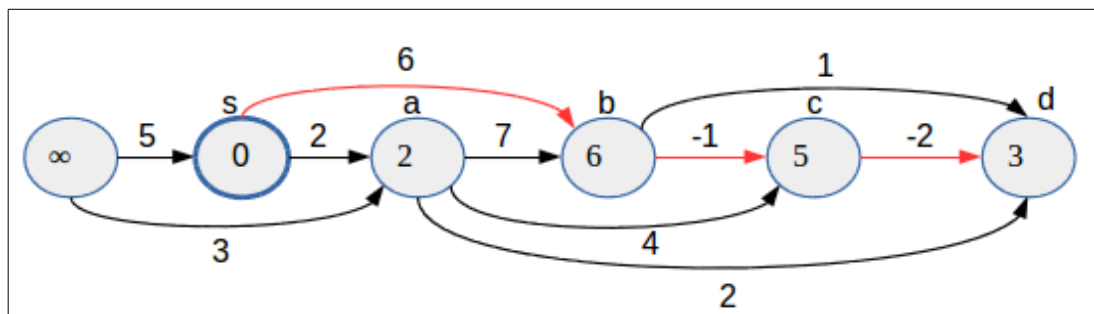
realx(b, d, -1):

$d[d] \geq d[b] + w(b, d) \Rightarrow 4 \geq 6 + 1 \Rightarrow$ nothing to do.

b is done, going to *c*

realx(c, d, -1):

$d[d] \geq d[c] + w(c, d) \Rightarrow 4 \geq 5 + (-2) \Rightarrow d[d] \leftarrow 3$



c is done, going to *d* (No vertices to go from *d*)

$\delta(s, s) = 0, \delta(s, a) = 2, \delta(s, b) = 6, \delta(s, c) = 5, \delta(s, d) = 3$

Dijkstra Algorithm

For each edge $(u, v) \in E$, assume that $w(u, v) \geq 0$, maintain a set S of vertices whose final shortest path weights have been determined. Repeatedly select $u \in V - S$ with minimum shortest path estimate, add u to S , relax all edges out of u .

G : the graph

W : the weights

s : the source

S : set of vertices of shortest paths

Q : is a **priority queue** that initially contains the all vertices of the graph.

Pseudo-code

```
Dijkstra ( $G, W, s$ ):
     $d[s] \leftarrow 0$ 
    for all  $u \in V \setminus \{s\}$ :
         $d[u] = \infty$ 
     $S \leftarrow \emptyset$ 
     $Q \leftarrow V$ 
    while  $Q \neq \emptyset$ :
         $(d[u], u) \leftarrow \text{extract-min}(Q)$  //Extract the vertex that have the minimum  $d()$  value.
        delete  $(d[u], u)$  from  $Q$ 
         $S \leftarrow S \cup \{u\}$ 
        for each vertex  $v \in \text{adj}[u]$ :
            //relax( $u, v, w$ ) : decrease key (reduce  $d()$  values)
            if  $d[v] > d[u] + w(u, v)$ :
                 $d[v] = d[u] + w(u, v)$ 
                 $\pi[v] = u$ 
                push  $(d[v], v)$  in the  $Q$ 
```

Dijkstra complexity

Priority queue

- $\theta(V)$ inserts in the priority queue.
- $\theta(V)$ extract-min operations.
- $\theta(E)$ decrease key operations.

Array:

- $\theta(V)$ time for extract min.
- $\theta(1)$ decrease key operations.
- Total = $\theta(V \cdot V + E \cdot 1) = \theta(V^2)$

binary min heap:

- $\theta(\log V)$ time for extract min.
- $\theta(\log V)$ decrease key operations.
- Total = $\theta(V \log V + E \log V)$

Fibonacci heap:

- $\theta(\log V)$ time for extract min.
- $\theta(1)$ amortized for decrease key operations.
- Amortized cost: Total = $\theta(V \log V + E)$

proof of the correctness of the Dijkstra algorithm

In general, to get the shortest paths from a source vertex s , we must reduce all $d()$ values to $\delta()$ values.

At the end of the Dijkstra algorithm, we must show that for each vertex $v \in S$, $d[v] = \delta(s, v)$

Lemma:

for each vertex $x \in S$, $d[x] = \delta(s, x)$

- **proof by induction:**

- base case:

At the beginning, $S = \{s\}$

$d[s] = \delta(s, s) = 0$, which is correct.

- Inductive step:

- Inductive hypothesis:

Suppose that: for each vertex $x \in S$, $d[x] = \delta(s, x)$ is correct for n iterations.

- Proof for $(n+1)$ iterations:

Let u the last vertex added to S .

$S' = S \cup \{u\}$

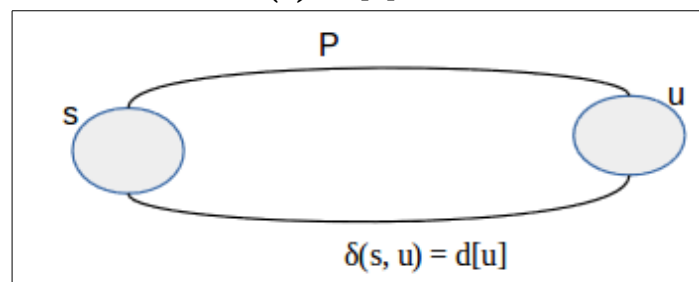
We have to proof that for each vertex $x \in S'$, $d[x] = \delta(s, x)$

For all $x \in S \setminus \{u\}$, the inductive hypothesis gives that $d[x] = \delta(s, x)$.

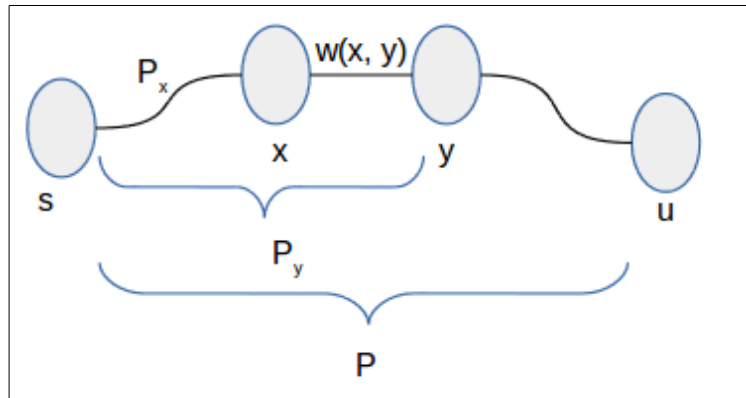
So, we need only to proof that $d[u] = \delta(s, u)$

(1) Assume that there is another shortest path P from s to u :

which means that: $w(P) < d[u]$

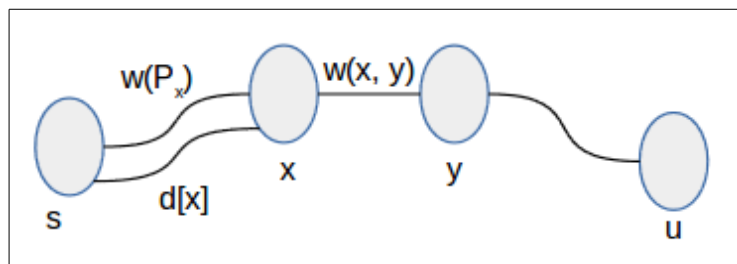


(2) In that shortest path P , we have a shortest path P_x , and a vertex y adjacent to x :



$$\text{so, } w(P_x) + w(x, y) < w(P)$$

(3) The length of the shortest path s to x is $d[x] \implies d[x] \leq w(P_x)$



(4) y is adjacent to x , $d[y]$ must be updated by the algorithm:

$$d[y] \leq d[x] + w(x, y)$$

(5) since u is picked by the algorithm, we are sure that $d[u]$ is the smallest distance:

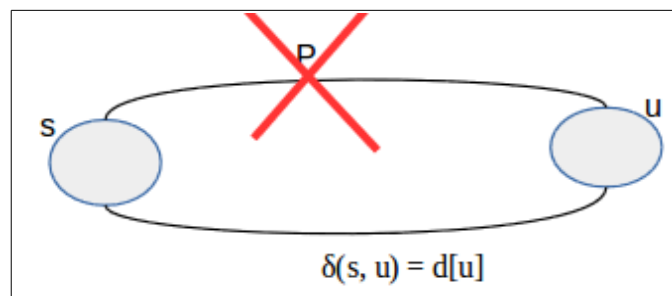
$$d[u] < d[y]$$

we have:

$$\left. \begin{array}{l} d[y] > d[u] \\ d[y] \leq d[x] + w(x, y) \end{array} \right\} d[u] \leq d[x] + w(x, y)$$

$$\left. \begin{array}{l} d[u] \leq d[x] + w(x, y) \\ d[u] + w(x, y) \leq w(P_x) \end{array} \right\} w(P_x) \geq d[u]$$

$$\left. \begin{array}{l} w(P_x) \geq d[u] \\ w(P_x) + w(x, y) < w(P) \end{array} \right\} \begin{array}{l} d[u] + w(x, y) < w(P) \implies d[u] < w(P) \\ \text{Contradiction with the first formula.} \\ \textbf{Impossible no such path.} \end{array}$$



We get : $d[u] = \delta(s, u)$

Application: hackerrank's problem

Problem's name: Dijkstra: Shortest Reach 2

link: <https://www.hackerrank.com/challenges/dijkstrashortreach/problem>

Editorial: [Dijkstra: Shortest Reach 2 editorial](#)

speeding up Dijkstra

As you see in the [application's editorial below](#), the Dijkstra algorithm gives a TLE in the code's submission.

The reason that, after the relaxation of the vertex v , we push it in the priority queue with the new $d[v]$ even if it exists. So the algorithm will pick that vertex v twice (may be more). First the algorithm will pick v that has the smallest $d[v]$, after that, it will pick v , many times, those have distances greater than the smallest computed distance ($d[v]$). Which does not make any sense, because the v 's neighbors (of the 2nd, 3rd, ... pick) will be never relaxed.

To solve this issue and speed up the single source, many targets, Dijkstra algorithm, is by checking if the vertex u is visited or not.

Dijkstra (G, W, s):

$d[s] \leftarrow 0$

for all $u \in V \setminus \{s\}$:

$d[u] = \infty$

$S \leftarrow \emptyset$

$Q \leftarrow V$

for all $u \in V$:

$visited[u] \leftarrow false$

while $Q \neq \emptyset$:

$(d[u], u) \leftarrow \text{extract-min}(Q)$ //Extract the vertex that have the minimum $d()$ value.

delete $(d[u], u)$ from Q

if $visited[u]$:

continue

$visited[u] \leftarrow true$

$S \leftarrow S \cup \{u\}$

for each vertex $v \in adj[u]$:

//relax(u, v, w) : decrease key (reduce $d()$ values)

if $d[v] > d[u] + w(u, v)$:

$d[v] = d[u] + w(u, v)$

$\pi[v] = u$

push $(d[v], v)$ in the Q

Another solution is to maintain $d[v]$ in a structure like a map (don't forget that map must behave like a priority queue, I notice it by *heap+map* to remember that it's a map that behave like a min heap) instead of pushing a vertex that already exists with another $d[]$ value.

Dijkstra (G, W, s):

```
    heap+map[s]  $\leftarrow$  0,  
    for all  $u \in V \setminus \{s\}$ :  
        heap+map[u] =  $\infty$   
    S  $\leftarrow \emptyset$   
    while heap+map  $\neq \emptyset$ :  
        u  $\leftarrow$  extract-min(heap+map) //Extract the vertex that have the  
                                         //minimum heap+map() value.  
        d[u] = heap+map[u]  
  
        delete heap+map[u]  
        S  $\leftarrow S \cup \{u\}$   
        for each vertex v  $\in$  adj[u]:  
            if v  $\in$  heap+map:  
                continue  
            //relax(u, v, w) : decrease key (reduce d() values)  
            if heap+map[v] > d[u] + w(u, v):  
                heap+map[v] = d[u] + w(u, v)  
                 $\pi[v] = u$ 
```

Bellman-Ford algorithm

Initialize for all $u \in V$:

$d[v] \leftarrow \infty, \pi[v] \leftarrow \text{nil}$

$d[s] \leftarrow 0$

repeat:

 select some edge(u, v) [Some how]

 relax edge(u, v, w)

until all edges have $d[v] \leq d[u] + w(u, v)$

With the generic structure algorithm, we are confronted to two problems:

1. complexity could be exponential time (even for positive edges weights)
2. will not terminate if there is a negative weight cycle reachable from the source.

Dijkstra algorithm fix the first problem.

Breadth First Search: Shortest Reach's editorial

We want to print the shortest distance to each node from a starting point s .

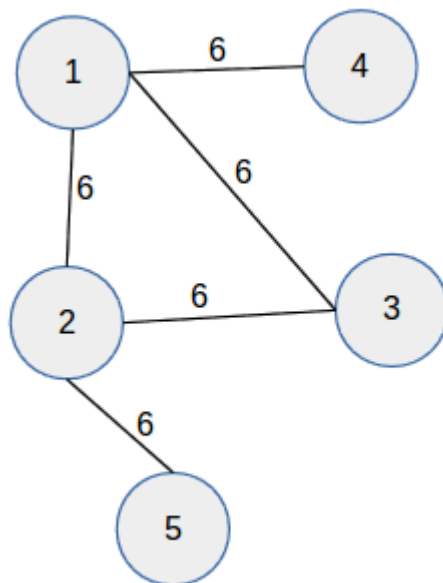
By doing a BFS, we can get the shortest distances very easy.

The weight for all edges is always 6. To compute the shortest distance between s and each other vertex v , we can compute the minimum number of edges between them.

In a BFS when jumping from a level to higher one, add the distance of the current vertex + 1 to all non-visited neighbors. So, we need an array to store the distances.

The current vertex is the vertex which we check its neighbors.

Example: Let a graph $G = (V, E)$, $V = \{1, 2, 3, 4, 5\}$, $E = \{\{1, 4\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 5\}\}$



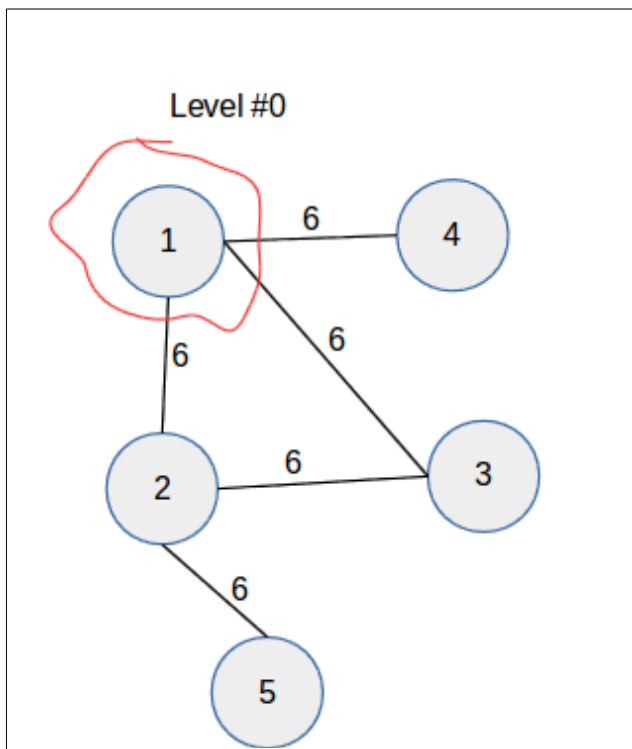
Initiate all distances to 0:

dist:

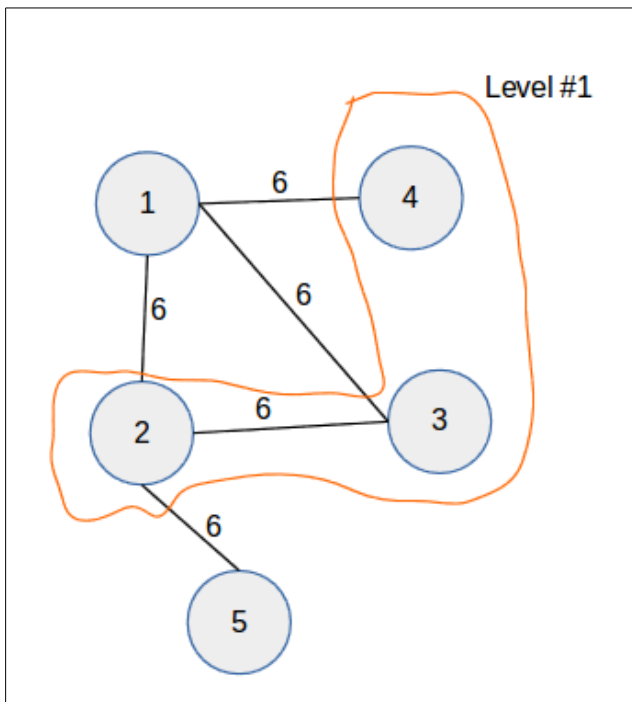
0	0	0	0	0
1	2	3	4	5

BFS

- level #0: just knowing s (s = 1)



- level #1: check all the neighbors of the vertex 1

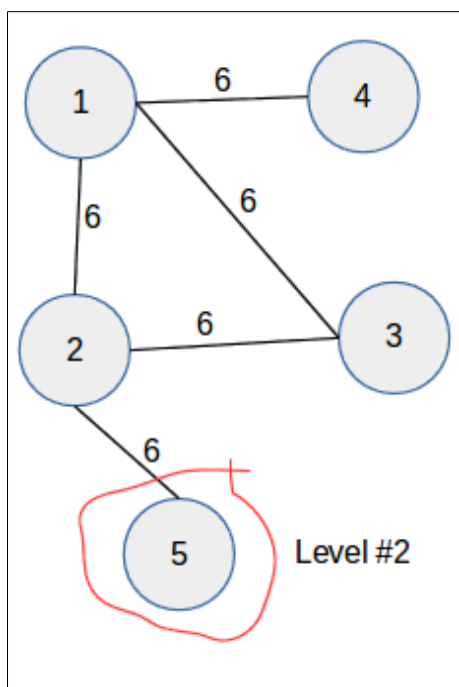


dist:

0	1	1	1	0
1	2	3	4	5

$$\text{dist}[2] = \text{dist}[3] = \text{dist}[4] = \text{dist}[1] + 1 = 0 + 1 = 1$$

- level #2: check all the neighbors of the vertex 2



dist:

0	1	1	1	2
1	2	3	4	5

$$\text{dist}[5] = \text{dist}[2] + 1 = 1 + 1 = 2$$

- For the other levels, all vertices were visited. So, for the example, the program print: 6 6 6 12, which means the shortest distances between 1 to 2, 1 to 3, 1 to 4 and 1 to 5 are respectively 6, 6, 6 and 12 .

C++ code

```
#include <bits/stdc++.h>

using namespace std;

void createAdjList(vector<list<int> > & adjList , int nbEdges) {
    for (int edge = 0 ; edge < nbEdges ; ++edge) {
        int vertex1 , vertex2;
        cin >> vertex1 >> vertex2;
        adjList[vertex1-1].push_back(vertex2-1);
        adjList[vertex2-1].push_back(vertex1-1);
    }
}

void bfs (int start , vector<list<int> > adjList , int nbVertices , vector<int> & dist) {
    vector<bool> visited(nbVertices);
    queue<int> q;
    q.push(start);
    visited[start] = true;
    while (!q.empty()) {
        int current = q.front();
        q.pop();
        for (list<int>::iterator it = adjList[current].begin() ; it != adjList[current].end() ; ++it) {
            if (!visited[*it]) {
                visited[*it] = true;
                q.push(*it);
                dist[*it] = dist[current] + 1;
            }
        }
    }
}

void displayDistances(vector<int> & dist , int nbVertices , int start) {
    for (int i = 0 ; i < nbVertices ; ++i)
        if (i != start-1) {
            if (dist[i] != 0)
                cout << 6 * dist[i] << ' ';
            else
                cout << "-1 ";
        }
}
```

```

int main() {
    int testCases;
    cin >> testCases;

    while (testCases--) {
        int nbVertices , nbEdges;
        cin >> nbVertices >> nbEdges;

        vector<list<int> > adjList(nbVertices);
        createAdjList(adjList,nbEdges);

        int start;
        cin >> start;

        vector<int> dist(nbVertices);

        bfs(start-1,adjList,nbVertices,dist);

        displayDistances(dist,nbVertices,start);
        cout << "\n";
    }

    return 0;
}

```

Roads and Libraries editorial

<https://www.hackerrank.com/challenges/torque-and-development/problem>

<https://www.hackerrank.com/challenges/torque-and-development/editorial>

Editorial by [torquecode](#)

Observation 1

We can't build new roads in HackerLand; we can only repair existing ones; however, when repairing roads, we need to be cognizant that there is no guarantee that each and every city is connected, meaning the graph may be disconnected and groups of cities will form. For example, a group of 3 cities might be connected to each other, but not connected to 2 other cities that are only connected to one another. We'll call each of these groups components, and in order to solve the whole problem we'll need to solve the components individually.

Observation 2

Each component needs at least one library. Without a library in one of the component's cities, there is no way for the cities in the component to access a library.

Conclusion

With these observations taken into account, there are two ways to assemble a component:

- A library must exist in at least one city, so $ct-1$ roads must be repaired (where ct is the number of cities in the component).
- A library must exist in every city in a component, meaning that no roads need to be repaired. We choose this option when the cost of building a library is less than the cost of repairing a road.

The minimum cost for each component will either be $c_{lib} + (ct-1) \cdot c_{road}$ when we repair roads, or $ct \cdot c_{lib}$ when we build a library in each city. Choosing the option that is smallest and summing it with all of the other smallest options for each component yields the value of the cheapest solution. If the cost for repairing a road and building a library are the same, the two approaches will be equal (meaning both options are equally valid).

C++ Code

```

#include <bits/stdc++.h>

using namespace std;

typedef unsigned long long int ULLI;

ULLI vpc;
bool *visited = NULL;

void createAdjList(vector<ULLI> *adjList , ULLI nbEdges) {
    for (ULLI edge = 0 ; edge < nbEdges ; ++edge) {
        ULLI vertex1 , vertex2;
        cin >> vertex1 >> vertex2;
        vertex1--;
        vertex2--;
        adjList[vertex1].push_back(vertex2);
        adjList[vertex2].push_back(vertex1);
    }
}

void dfs_visit(vector<ULLI> *adjList, ULLI s){
    vector <ULLI> l = adjList[s];
    for (auto it: l){
        ULLI v = it;

        if (visited[v]) continue;
        visited[v] = true;

        vpc++;
        dfs_visit(adjList, v);
    }
}

ULLI dfs(ULLI V, vector<ULLI> *adjList, ULLI cLib, ULLI cRoad){
    ULLI roadsCoast = 0;

```

```

ULLI libsCoast = 0;
for (ULLI s = 0 ; s < V ; ++s){
    if (visited[s]) continue;
    visited[s] = true;
    vpc = 0;
    vpc++;

    dfs_visit(adjList, s);

    roadsCoast += (vpc - 1) * cRoad;
    libsCoast += cLib;
}

return roadsCoast + libsCoast;
}

int main() {
    ULLI q;

    scanf("%llu", &q);

    for (ULLI i = 1 ; i <= q ; ++i){

        ULLI n, m, cLib, cRoad;

        scanf("%llu%llu%llu%llu", &n, &m, &cLib, &cRoad);

        vector<ULLI> *adjList = new vector<ULLI>[n];
        createAdjList(adjList, m);

        if (cRoad >= cLib)
            printf("%llu\n", n * cLib);
        else {
            visited = new bool[n];
            fill(visited, visited + n , false);

            printf("%llu\n", dfs(n, adjList, cLib, cRoad));
        }
    }
    return 0;
}

```


Journey to the Moon

<https://www.hackerrank.com/challenges/journey-to-the-moon/problem>

<https://www.hackerrank.com/challenges/journey-to-the-moon/editorial>

Editorial by [amititkqp](#)

This problem can be thought of as a graph problem. The very first step is to compute how many different countries are there. For this, we apply Depth First Search to calculate how many different connected components are present in the graph where the vertices are represented by the people and the people from the same country form one connected component. After we get how many connected components are present, say M , we just need to calculate the number of ways of selecting two persons from two different connected component. Let us assume that component i contains M_i people. So, for the number of ways selecting two persons from different components, we subtract the number of ways of selecting two persons from the same component from the total numbers of ways of selecting two persons i.e.

Ways = $N \text{ choose } 2 - (\sum (M_i \text{ Choose } 2) \text{ for } i = 1 \text{ to } M)$

C++

```
//Animesh Sinha

#include <iostream>
#include <list>
#include <vector>
#include <stdio.h>
#include <iterator>
#include <cmath>

#define MAX 100000

using namespace std;

list<int> *ad;
int *visited;
int vertices;
```

```
void DFS(int u)
{
    visited[u] = 1;

    vertices++;

    list<int>::iterator it;

    for(it=ad[u].begin();it!=ad[u].end();it++)
    {
        if(visited[*it] == 0)
        {
            visited[*it] = 1;
            DFS(*it);
        }
    }
}
```

```

int main()
{
    int i,m,u,v,numComponents=0,allv=0,temp=2,count=0;
    long long int n;
    int eachC[MAX];

    cin >> n >> m;

    if(n == 1)
    {
        cout << "0\n";
        return 0;
    }

    ad = new list<int>[n];

    list<int>::iterator it;

    for(i=0;i<m;i++)
    {
        cin >> u >> v;

        ad[u].push_back(v);
        ad[v].push_back(u);
    }

    visited = new int[n];

    for(i=0;i<n;i++)
    {
        visited[i] = 0;
    }

    for(i=0;i<n;i++)
    {
        if(visited[i] == 0)
        {
            vertices = 0;
            DFS(i);
            eachC[numComponents] = vertices;
            numComponents++;
        }
    }

    long long int totalWays = n*(n-1) / 2;
    long long int sameWays = 0;

    for(i=0;i<numComponents;i++)

```

```
{
    sameWays = sameWays + (eachC[i]*(eachC[i]-1) / 2);
}
cout << (totalWays - sameWays) << endl;
return 0;
}
```

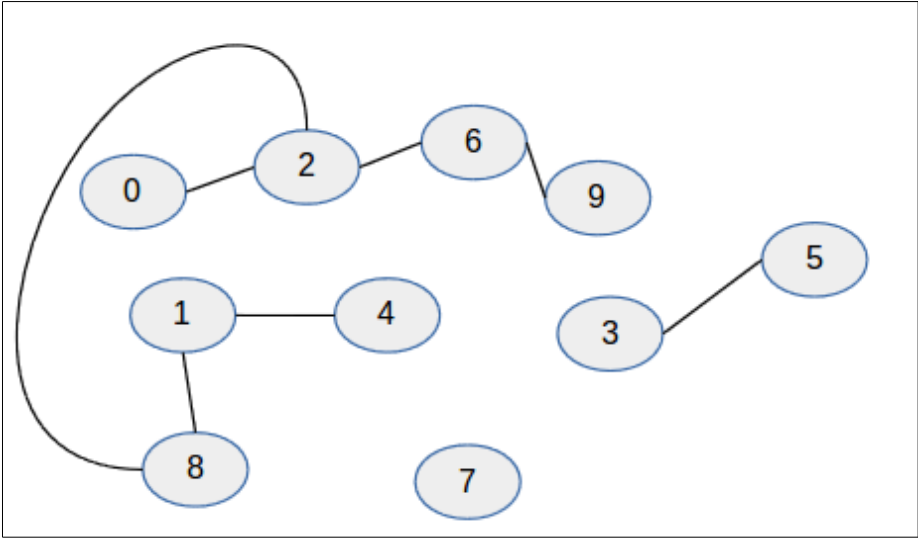
Editorial [the lazy duck](#)

Find astronauts of the same country

Draw a graph where nodes are astronauts and edges represents the relationship "same country".

From each node *i*, run a DFS to find all reachable vertices from *i*. We must pay attention to do not DFSing a visited node.

let's take an example:

Example	graph
INPUT	
10 7	
0 2	
1 8	
1 4	
2 8	
2 6	
3 5	
6 9	

Running a DFS on vertex 0, gives: 0 2 6 9 1 8 4. No need to run a DFS on edges 1, 2, 4, 6, 8 and 9. we have to run DFS on 3, which gives: 3 5 and vertex 7, will gives 7.

so we get 3 sets of astronauts from the same countries: {0, 2, 6, 9, 1, 8, 4}, {3, 5} and {7}

Compute in how many ways we can pick a pair of astronauts belonging to different countries

The question demands the concept of using **disjoint sets**. (find and union with rank and path compression).

However, this method below using DFS also works (which I think i pretty similar to the editorial) :

We need to find the distinct sets (i.e. if x and y are from same country, they belong to same set) to get the answer.

Let's say set A has a elements, set B has b elements.

By doing the Cartesian product of $A \times B$, such $A \times B = \{(x, y) \mid x \in A \text{ and } y \in B\}$

The Cartesian product $A \times B$ is the set of all ordered pairs (x, y) where $x \in A$ and $y \in B$

(https://en.wikipedia.org/wiki/Cartesian_product)

So the number of pairs is $a \times b$

Similarly, let's calculate answer for 4 sets – A, B, C and D with a, b, c, d elements respectively.

Lets say somehow I find set A has a elements.

Answer = 0 (Since I don't have another country to pair with)

Now, I find set B with b elements.

Answer = axb ;(1)

Then, I find set C with c elements.

Answer = $(axb) + (axc) + (bxc)$ [because we can select a pair from A and B , or A and C or B and C]

But this can be written as

Answer = $(axb) + (a+b)xc$ (2)

Now I find set D with d elements, and I've examined all people.

Answer = $(axb) + (axc) + (axd) + (bxc) + (bxd) + (cxd)$

Or

Answer = $(axb) + (a+b)xc + (a+b+c)xd$ (3)

Do you see the pattern? That means when we find a new set, the new answer is the

old answer + the sum of old values x new value.

Now, how do we find the people of same country?

Make each person a node. Draw an edge for every input (i.e. they belong to same country)

Now, run DFS starting from every node (if the node is not already visited). DFS should return the nodes in that set.

Example	graph	sets	DFS and answer progression
INPUT 10 7 0 2 1 8 1 4 2 8 2 6 3 5 6 9		3 sets $\{0,1,2,4,6,8,9\}$ $\{3,5\}$ $\{7\}$	DFS from 0 returns "7" answer = 0 DFS from 1 should not start [since we have visited 1 while dfs(0)] DFS from 2 should not start ... DFS from 3 returns "2" answer = $7 * 2 = 14$ DFS from 7 returns "1" answer = $14 + (7+2)*1$ answer = 23

Accepted C++ code: $O(V+1)$

```
#include <bits/stdc++.h>

using namespace std;

int V, E;

list<int> *adjList = NULL;

bool *visited = NULL;

int cnt;

void createAdjList(int nbEdges) {
    for (int edge = 0 ; edge < nbEdges ; ++edge) {
        int vertex1 , vertex2;
        cin >> vertex1 >> vertex2;

        adjList[vertex1].push_back(vertex2);
        adjList[vertex2].push_back(vertex1);
    }
}

void dfs_visit(int v, bool visited[]){
    visited[v] = true;

    //Compute the number of elements of each set.
    cnt++;

    list<int>::iterator i;

    for (i = adjList[v].begin(); i != adjList[v].end(); ++i)
        if (!visited[*i])
            dfs_visit(*i, visited);
}

void dfs(int v){
    cnt = 0;
    dfs_visit(v, visited);
}
```

```
int main() {  
  
    cin >> V >> E;  
  
    adjList = new list<int>[V];  
  
    createAdjList(E);  
  
    visited = new bool[V];  
  
    for (int i = 0; i < V; ++i)  
        visited[i] = false;  
  
    long long int ans = 0;  
    long long int s = 0;  
    for (int i = 0 ; i < V ; ++i) {  
        if (visited[i]) continue;  
        visited[i] = true;  
        dfs(i);  
        ans += s * cnt;  
        s += cnt;  
    }  
  
    printf("%lld\n", ans);  
  
    return 0;  
}
```


Solution with formula ([André Nicolas on https://math.stackexchange.com/questions/1418652/how-many-ways-to-select-distinct-pairs-from-k-disjoint-sets](https://math.stackexchange.com/questions/1418652/how-many-ways-to-select-distinct-pairs-from-k-disjoint-sets))

Let's say set A has a elements, set B has b elements.

By doing the Cartesian product of $A \times B$, such $A \times B = \{(x, y) \mid x \in A \text{ and } y \in B\}$

The Cartesian product $A \times B$ is the set of all [ordered pairs](https://en.wikipedia.org/wiki/Cartesian_product) (x, y) where $x \in A$ and $y \in B$ (https://en.wikipedia.org/wiki/Cartesian_product)

So the number of pairs is $a \cdot b$

$$\text{answer} = ab$$

$$\Leftrightarrow 2\text{answer} = 2ab = a^2 + 2ab + b^2 - (a^2 + b^2)$$

$$\text{so, } 2\text{answer} = (a + b)^2 - (a^2 + b^2) \Leftrightarrow \text{answer} = \frac{(a+b)^2 - (a^2 + b^2)}{2}$$

Suppose that we have n groups, of sizes a_1 to a_n . Except when n is smallish, the following formula for the number answer of ways to choose a pair of people from different groups is more efficient:

$$\text{answer} = \frac{(a_1 + a_2 + a_3 + \dots + a_n)^2 - (a_1^2 + a_2^2 + a_3^2 + \dots + a_n^2)}{2}$$

Accepted C++ code : O(V+1)

```
#include <bits/stdc++.h>

using namespace std;

int V, E;

list<int> *adjList = NULL;

bool *visited = NULL;

int cnt;
```

```

void createAdjList(int nbEdges) {
    for (int edge = 0 ; edge < nbEdges ; ++edge) {
        int vertex1 , vertex2;
        cin >> vertex1 >> vertex2;

        adjList[vertex1].push_back(vertex2);
        adjList[vertex2].push_back(vertex1);
    }
}

void dfs_visit(int v, bool visited[]){
    visited[v] = true;

    //Compute the number of elements of each set.
    cnt++;

    list<int>::iterator i;

    for (i = adjList[v].begin(); i != adjList[v].end(); ++i)
        if (!visited[*i])
            dfs_visit(*i, visited);
}

void dfs(int v){
    cnt = 0;
    dfs_visit(v, visited);
}

```

```

int main() {

    cin >> V >> E;

    adjList = new list<int>[V];

    createAdjList(E);

    visited = new bool[V];

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

    long long int ans = 0;
    long long int s = 0;
    long long int sp = 0;
    for (int i = 0 ; i < V ; ++i) {
        if (visited[i]) continue;
        visited[i] = true;
        dfs(i);
        s += cnt;
        sp += cnt * cnt;
    }
    ans = (s * s - sp) / 2;
    printf("%lld\n", ans);

    return 0;
}

```


Dijkstra: Shortest Reach 2

(Direct application of the Dijkstra algorithm)

<https://www.hackerrank.com/challenges/dijkstrashortreach/problem>

<https://www.hackerrank.com/challenges/dijkstrashortreach/editorial>

A good explanation: <https://www.youtube.com/watch?v=lAXZGERcDf4>

TLE solution with STL priority_queue as a min heap

```
#include <bits/stdc++.h>

using namespace std;

const int INF = 1e9;

class Edge{
public:
    pair<int, int> edge;
public:
    Edge(pair<int, int> e ) {
        edge = e;
    }
};

void createAdjList(vector<list<Edge> > & adjList , int nbEdges) {
    for (int edge = 0 ; edge < nbEdges ; ++edge) {
        int vertex1 , vertex2 , weight;
        scanf("%d%d%d", &vertex1, &vertex2, &weight);
        Edge e1(make_pair(vertex2-1,weight));
        Edge e2(make_pair(vertex1-1,weight));
        adjList[vertex1-1].push_back(e1);
        adjList[vertex2-1].push_back(e2);
    }
}

void dijkstra (int start , vector<list<Edge> > adjList , int nbVertices , int *dist) {

    dist[start] = 0;

    priority_queue<pair<int, int>, vector<pair<int, int> >, greater<pair<int, int> > > q;
    q.push({0, start});

    while (!q.empty()) {
        int current = q.top().second;
        q.pop();

        for (auto it: adjList[current]) {
```

```

    pair<int, int> e = it.edge;
    int v = e.first;
    int w = e.second;
    if (dist[v] > dist[current] + w ) {
        dist[v] = dist[current] + w;
        q.push({dist[v], v});
    }
}
}
}

void displayDistances(int *dist , int nbVertices , int start) {
    for (int i = 0 ; i < nbVertices ; ++i) {
        if (i == start) continue;
        printf("%d ", (dist[i] != INF ? dist[i] : -1));
    }
}

```

```

int main() {
    int testCases;
    scanf("%d", &testCases);

    while (testCases--) {
        int nbVertices , nbEdges;
        scanf("%d%d", &nbVertices, &nbEdges);

        vector<list<Edge> > adjList(nbVertices);
        createAdjList(adjList, nbEdges);

        int start;
        scanf("%d", &start);

        int *dist = new int[nbVertices];
        fill(dist, dist + nbVertices, INF);

        dijkstra(start-1, adjList, nbVertices, dist);

        displayDistances(dist, nbVertices, start-1);

        cout << "\n";
    }

    return 0;
}

```

Dijkstra: Shortest Reach 2

 by pranav9413

Problem

Submissions

Leaderboard

Discussions

Editorial

Submitted 18 minutes ago • Score: 46.36

Status: **Terminated due to timeout**



Test Case #0



Test Case #1



Test Case #2



Test Case #3



Test Case #4



Test Case #5



Test Case #6



Test Case #7

Accepted solution with STL priority_queue as a min heap

```
#include <bits/stdc++.h>

using namespace std;

const int INF = 1e9;

class Edge{
public:
    pair<int, int> edge;
public:
    Edge(pair<int, int> e ) {
        edge = e;
    }
};

void createAdjList(vector<list<Edge> > & adjList , int nbEdges) {
    for (int edge = 0 ; edge < nbEdges ; ++edge) {
        int vertex1 , vertex2 , weight;
        scanf("%d%d%d", &vertex1, &vertex2, &weight);
        Edge e1(make_pair(vertex2-1,weight));
        Edge e2(make_pair(vertex1-1,weight));
        adjList[vertex1-1].push_back(e1);
        adjList[vertex2-1].push_back(e2);
    }
}
```



```

void dijkstra (int start , vector<list<Edge> > adjList , int nbVertices , int *dist) {
    bool visited[nbVertices];
    fill(visited, visited + nbVertices, false);

    dist[start] = 0;

    priority_queue<pair<int, int>, vector<pair<int, int> >, greater<pair<int, int> > > q;
    q.push({0, start});

    while (!q.empty()) {
        int current = q.top().second;
        q.pop();

        if (visited[current]) continue;

        visited[current] = true;

        for (auto it: adjList[current]) {
            pair<int, int> e = it.edge;
            int v = e.first;
            int w = e.second;
            if (dist[v] > dist[current] + w ) {
                dist[v] = dist[current] + w;
                q.push({dist[v], v});
            }
        }
    }
}

void displayDistances(int *dist , int nbVertices , int start) {
    for (int i = 0 ; i < nbVertices ; ++i) {
        if (i == start) continue;
        printf("%d ", (dist[i] != INF ? dist[i] : -1));
    }
}

```

```

int main() {
    int testCases;
    scanf("%d", &testCases);

    while (testCases--) {
        int nbVertices , nbEdges;
        scanf("%d%d", &nbVertices, &nbEdges);

        vector<list<Edge> > adjList(nbVertices);
        createAdjList(adjList, nbEdges);

        int start;
        scanf("%d", &start);

        int *dist = new int[nbVertices];
        fill(dist, dist + nbVertices, INF);

        dijkstra(start-1, adjList, nbVertices, dist);


        displayDistances(dist, nbVertices, start-1);

        cout << '\n';
    }

    return 0;
}

```

Dijkstra: Shortest Reach 2

 by [pranav9413](#)

Problem

Submissions

Leaderboard

Discussions

Editorial

Submitted 17 minutes ago • Score: 60.00

Status: **Accepted**

✓	Test Case #0	✓	Test Case #1	✓	Test Case #2
✓	Test Case #3	✓	Test Case #4	✓	Test Case #5
✓	Test Case #6	✓	Test Case #7		

Accepted solution with heap+map (you have to define the *extract-min* function)

```
#include <bits/stdc++.h>
#include <map>
#include <algorithm>

using namespace std;

const int INF = 1e9;

class Edge {
public:
    pair<int, int> edge;
public:
    Edge(pair<int, int> e) {
        edge = e;
    }
};

void createAdjList(vector<list<Edge>> & adjList , int nbEdges) {
    for (int edge = 0 ; edge < nbEdges ; ++edge) {
        int vertex1 , vertex2 , weight;
        scanf("%d%d%d", &vertex1, &vertex2, &weight);
        vertex1--;
        vertex2--;
        Edge e1({vertex2, weight});
        Edge e2({vertex1, weight});
        adjList[vertex1].push_back(e1);
        adjList[vertex2].push_back(e2);
    }
}

void initHeap(map<int, int> & h , int nbVertices) {
    for (int i = 0 ; i < nbVertices ; ++i)
        h.insert(pair<int, int>(i, INF));
}

template <typename T>
typename T::iterator min_map_element(T& m) {
    return min_element(m.begin(), m.end(), [](typename T::value_type& l, typename T::value_type& r) -> bool {
        return l.second < r.second;
    });
}
```

```

void dijkstra(int *dist,
             vector<list<Edge>> & adjList,
             map<int,int> h) {

    while(!h.empty()) {
        int current = min_map_element(h)->first;
        dist[current] = h[current];
        h.erase (current);

        for (auto it: adjList[current]){
            pair<int, int> e = it.edge;
            int v = e.first;
            int w = e.second;

            map<int, int>::iterator it1 = h.find(v);
            if (it1 == h.end()) continue;

            if (h[v] > dist[current] + w) {
                h[v] = dist[current] + w;
            }
        }
    }
}

```

```

int main() {
    int testCases;
    scanf("%d", &testCases);
    while (testCases--) {
        int nbVertices , nbEdges;
        scanf("%d%d", &nbVertices, &nbEdges);

        vector<list<Edge>> adjList(nbVertices);
        createAdjList(adjList, nbEdges);

        /*int i = -1;
        for (auto it1: adjList) {
            list<Edge> l = it1;
            i++;
            cout << i << " ==> ";
            for (auto it2: l)
                cout << it2.edge.first << "/" << it2.edge.second << ' ';
            cout << "\n";
        }*/

        int start;
        scanf("%d", &start);

        start--;

        map<int,int> heapMap;

        initHeap(heapMap, nbVertices);

        heapMap[start] = 0;

        /*for (auto it: heapMap)
            printf("%d ==> %d\n", it.first, it.second);*/

        int *dist = new int[nbVertices];
        dijkstra(dist, adjList, heapMap);

        for (int i = 0 ; i < nbVertices ; ++i) {
            if (i == start) continue;
            printf("%d ", dist[i] != INF ? dist[i] : -1);
        }

        delete dist;

        printf("\n");
    }
    return 0;
}

```