11 Graph Algorithms

Introduction :

Graphs are mathematical structures that represent pairwise relationships between objects. A graph is a flow structure that represents the relationship between various objects. It can be visualized by using the following two basic components :

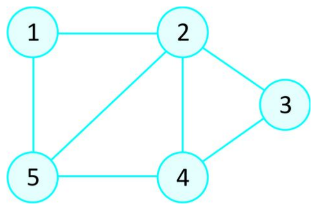
- **Nodes :** These are the most important components in any graph. Nodes are entities whose relationships are expressed using edges. If a graph comprises 2 nodes A and B and an undirected edge between them, then it expresses a bi-directional relationship between the nodes and edge.
- **Edges :** Edges are the components that are used to represent the relationships between various nodes in a graph. An edge between two nodes expresses a one-way or two-way relationship between the nodes.

Some Real Life Applications:

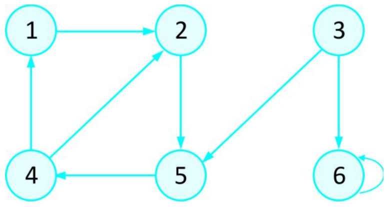
- ✓ **Google Maps :** To find a route based on shortest route/Time.
- **Social Networks :** Connecting with friends on social media, where each user is a vertex, and when users connect they create an edge.
- **Web Search :** Google, to search for webpages, where pages on the internet are linked to each other by hyperlinks, each page is a vertex and the link between two pages is an edge.
- **Recommendation System :** On eCommerce websites relationship graphs are used to show recommendations.

Types of Graphs :

- **Undirected :** An undirected graph is a graph in which all the edges are bi-directional i.e. the edges do not point in any specific direction.



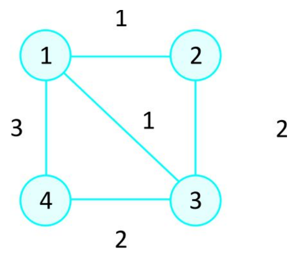
- **Directed :** A directed graph is a graph in which all the edges are uni-directional i.e. the edges point in a single direction.



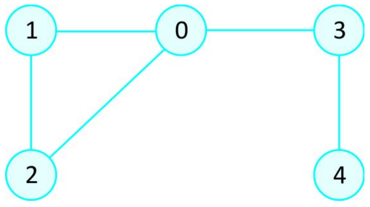
- **Weighted :** In a weighted graph, each edge is assigned a weight or cost. Consider a graph of 4 nodes as in the diagram below. As you can see each edge has a weight/cost assigned to it. If you want to go from vertex 1 to vertex 3, you can take one of the following 3 paths:

  - 1 -> 2 -> 3
  - 1 -> 3
  - 1 -> 4 -> 3

  Therefore the total cost of each path will be as follows: The total cost of 1 -> 2 -> 3 will be (1 + 2) i.e. 3 units - The total cost of 1 -> 3 will be 1 unit - The total cost of 1 -> 4 -> 3 will be (3 + 2) i.e. 5 units



- **Cyclic:** A graph is cyclic if the graph comprises a path that starts from a vertex and ends at the same vertex. That path is called a cycle. An acyclic graph is a graph that has no cycle.

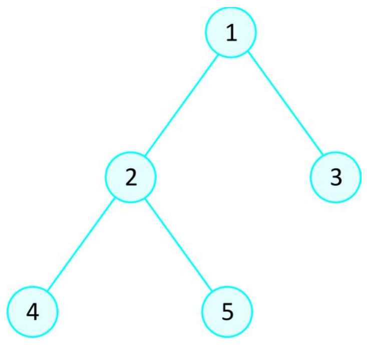


Cycle presents in the above graph (0->1->2)

A tree is an undirected graph in which any two vertices are connected by only one path. A tree is an acyclic graph and has N - 1 edges where N is the number of vertices. Each node in a graph may have one or multiple parent nodes. However, in a tree, each node (except the root node) comprises exactly one parent node.

Note: A root node has no parent.

A tree cannot contain any cycles or self loops, however, the same does not apply to graphs.



(**Tree : There is no any cycle or self loop in this graph**)

Graph Representation

You can represent a graph in many ways. The two most common ways of representing a graph is as follows:

1. Adjacency Matrix

An adjacency matrix is a $V*V$ binary matrix A. Element $A_{i,j}$ is 1 if there is an edge from vertex i to vertex j else $A_{i,j}$ is 0.

Note: A binary matrix is a matrix in which the cells can have only one of two possible values - either a 0 or 1.

The adjacency matrix can also be modified for the weighted graph in which instead of storing 0 or 1 in $A_{i,j}$ the weight or cost of the edge will be stored.

In an undirected graph, if $A_{i,j} = 1$, then $A_{j,i} = 1$.

In a directed graph, if $A_{i,j} = 1$, then may or may not be 1.

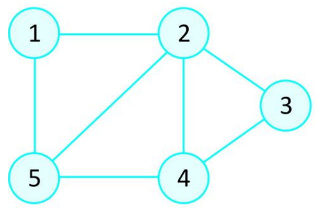
Adjacency matrix provides constant time access (O(1)) to determine if there is an edge between two nodes. Space complexity of the adjacency matrix is $O(V^2)$.

2. Adjacency List

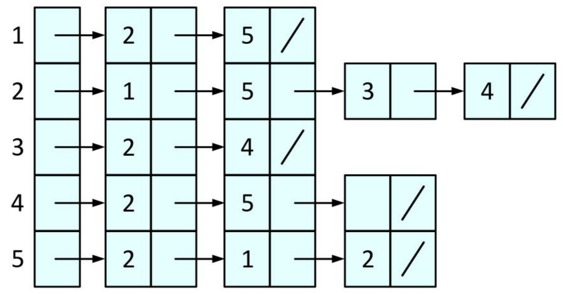
The other way to represent a graph is by using an adjacency list. An adjacency list is an array A of separate lists. Each element of the array A_i is a list, which contains all the vertices that are adjacent to vertex i.

For a weighted graph, the weight or cost of the edge is stored along with the vertex in the list using pairs. In an undirected graph, if vertex j is in list A_i then vertex i will be in list A_j.

The space complexity of adjacency list is O(V + E)because in an adjacency list information is stored only for those edges that actually exist in the graph. In a lot of cases, where a matrix is sparse using an adjacency matrix may not be very useful. This is because using an adjacency matrix will take up a lot of space where most of the elements will be 0, anyway. In such cases, using an adjacency list is better.



(a)



(b)

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \end{array}$$

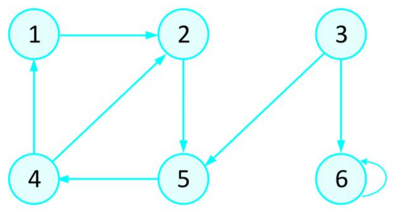
(c)

Two representations of an undirected graph.

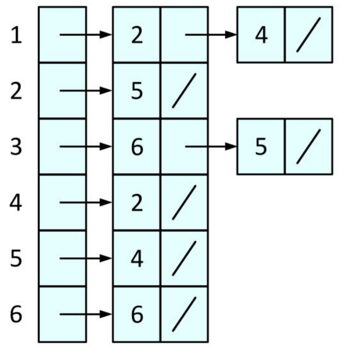
(a) An undirected graph G having five vertices and seven edges

(b) An adjacency-list representation of G

(c) The adjacency-matrix representation of G



(a)



(b)

$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 0 & 1 & 1 \\ 4 & 0 & 1 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 1 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}$$

(c)

Code for Adjacency list representation of a graph :

```
#include<iostream>
#include<list>
using namespace std;
class Graph{
    int V;
    list<int> *l;
public:
    Graph(int v){
        V = v;
        //Array of Linked Lists
        l = new list<int>[V];
    }
    void addEdge(int u,int v,bool bidir=true){
            l[u].push_back(v);
            if(bidir){
                  l[v].push_back(u);
            }
      }
      void printAdjList(){
            for(int i=0;i<V;i++){
                        cout<<i<<"->";
                        //l[i] is a linked list
                        for(int vertex: l[i]){
                              cout<<vertex<<",";
                        }
                        cout<<endl;
            }
      }
};
int main(){
      // Graph has 5 vertices number from 0 to 4
      Graph g(5);
      g.addEdge(0,1);
      g.addEdge(0,4);
      g.addEdge(4,3);
      g.addEdge(1,4);
      g.addEdge(1,2);
      g.addEdge(2,3);
      g.addEdge(1,3);
      g.printAdjList();
return 0;
}
```

Graph Traversal :

Graph traversal means visiting every vertex and edge exactly once in a well-defined order. While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once. The order in which the vertices are visited are important and may depend upon the algorithm or question that you are solving.

During a traversal, it is important that you track which vertices have been visited. The most common way of tracking vertices is to mark them.

Breadth First Search (BFS) :

There are many ways to traverse graphs. BFS is the most commonly used approach.

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer

Breadth-first search colors each vertex white, gray, or black. All vertices start out white and may later become gray and then black. A vertex is *discovered* the first time it is encountered during the search, at which time it becomes nonwhite. Gray and black vertices, therefore, have been discovered, but breadth-first search distinguishes between them to ensure that the search proceeds in a breadth-first manner. If (*u*, v) " *E* and vertex *u* is black, then vertex v is either gray or black; that is, all vertices adjacent to black vertices have been discovered. Gray vertices may have some adjacent white vertices; they represent the frontier between discovered and undiscovered vertices.

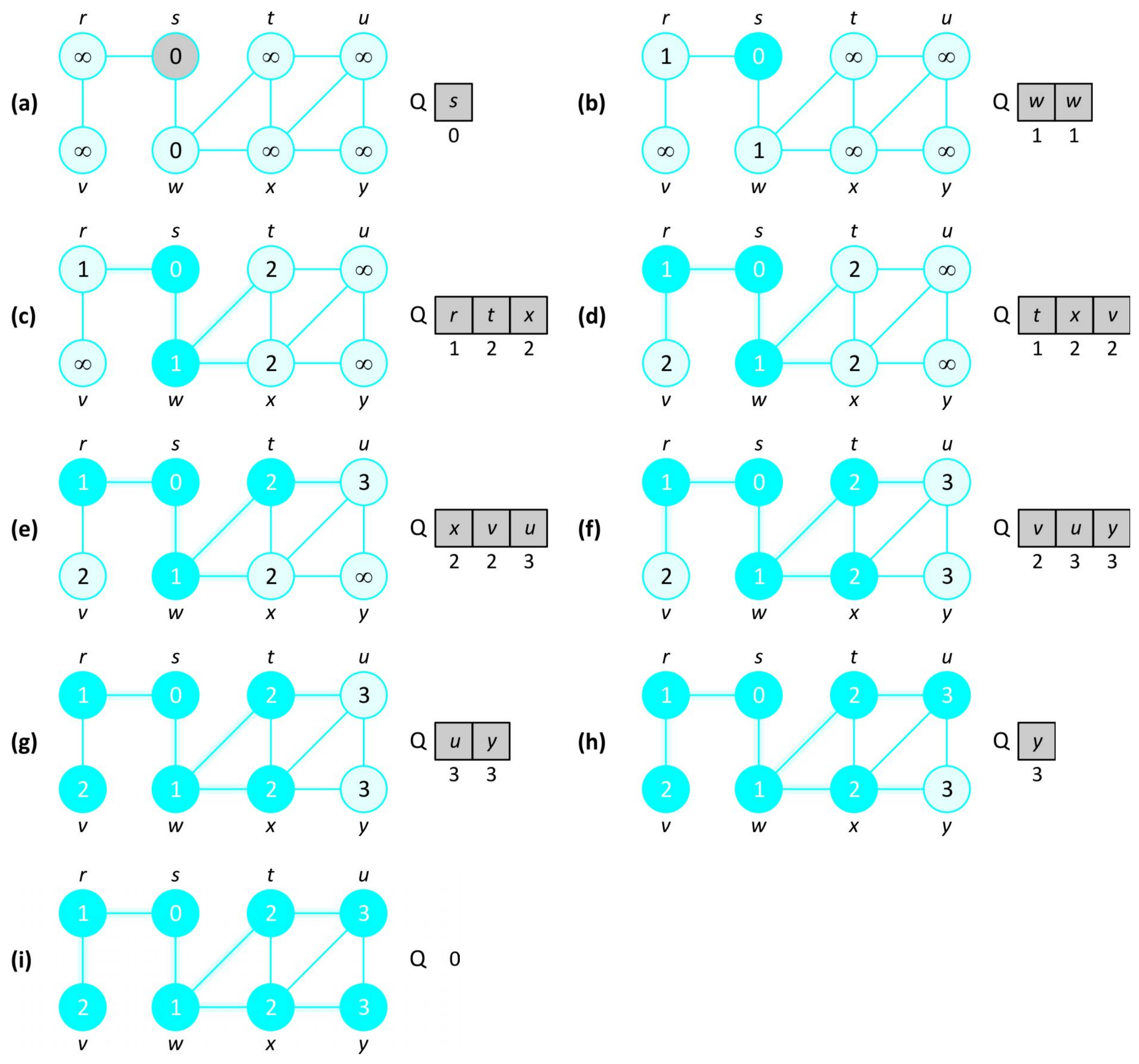
Breadth-first search constructs a breadth-first tree, initially containing only its root, which is the source vertex *s*. Whenever a white vertex v is discovered in the course of scanning the adjacency list of an already discovered vertex *u*, the vertex v and the edge (*u*, v) are added to the tree. We say that *u* is the *predecessor* or *parent* of v in the breadth-first tree.

Algorithm :

```
BFS(G, s)
    for each vertex u ∈ V [G] - {s}
        do color[u] ← WHITE
           d[u] ← ∞
           π[u] ← NIL
    color[s] ← GRAY
    d[s] ← 0
    π[s] ← NIL
    Q ← 0
    ENQUEUE(Q, s)
    while Q ≠ 0
        do u ← DEQUEUE(Q)
            for each ∈ Adj[u]
                do if color[v] = WHITE
                    then color[v] ← GRAY
```

```
                  d[v] ← d[u] + 1
                  π[v] ← u
                  ENQUEUE(Q, v)

    color[u] ← BLACK
```



The operation of BFS on an undirected graph. Tree edges are shown shaded as they are prduced by BFS. Within each vertex vertex u is shown d[u]. The queue Q is shown at the beginning of each iteration of the while loop of lines 10-18. Vertex distances are shown next to vertices in the queue.

Code :

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

template<typename T>
class Graph{

	map<T,list<T> > adjList;

public:
	Graph(){

	}
	void addEdge(T u, T v,bool bidir=true){

		adjList[u].push_back(v);
		if(bidir){
			adjList[v].push_back(u);
		}
	}

	void print(){
```

Time Complexity :

Time complexity and space complexity of above algorithm is O(V + E) and O(V).

Application of BFS

1. **Shortest Path and Minimum Spanning Tree for unweighted graph** In unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.

   Code for finding shortest path using BFS :

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

```
#include<queue>
using namespace std;
template<typename T>
class Graph{
            map<T,list<T> > adjList;
public:
            Graph(){
 }
            void addEdge(T u, T v,bool bidir=true){
                  adjList[u].push_back(v);
            if(bidir){
                  adjList[v].push_back(u);
            }
      }

      void print(){
            //Iterate over the map
            for(auto i:adjList){
                  cout<<i.first<<"->";
            //i.second is LL
            for(T entry:i.second){
                  cout<<entry<<",";
            }
            cout<<endl;
      }
}
voidbfs(T src){
      queue<T> q;
      map<T,int>dist;
      map<T,T> parent;
      for(auto i:adjList){
            dist[i.first] = INT_MAX;
      }
      q.push(src);
```

```
                    dist[src] = 0;
                    parent[src] = src;
                    while(!q.empty()){
                            T node = q.front();
                            cout<<node<<" ";
                            q.pop();
                            // For the neigbours of the current node, find out the nodes which
                    are not visited
                            for(intneigbour :adjList[node]){
                                    if(dist[neigbour]==INT_MAX){
                                            q.push(neigbour);
                                            dist[neigbour] = dist[node]  + 1;
                                            parent[neigbour] = node;
                                    }
                            }
                    }
                    //Print the distance to all the nodes
                    for(auto i:adjList){
                            T node = i.first;
                            cout<<"Dist of "<<node<<" from "<<src<<" is "<<dist[node]<<endl;
                    }
            }
};
intmain(){
        Graph<int> g;
        g.addEdge(0,1);
        g.addEdge(1,2);
        g.addEdge(0,4);
        g.addEdge(2,4);
        g.addEdge(2,3);
        g.addEdge(3,5);
        g.addEdge(3,4);
        g.bfs(0);
}
```

2. **Peer to Peer Networks :** In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.
3. **Crawlers in Search Engines :** Crawlers build index using Breadth First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of built tree can be limited.
4. **Social Networking Websites :** In social networks, we can find people within a given distance 'k' from a person using Breadth First Search till 'k' levels.
5. **GPS Navigation systems :** Breadth First Search is used to find all neighboring locations.
6. **Broadcasting in Network :** In networks, a broadcasted packet follows Breadth First Search to reach all nodes.
7. **In Garbage Collection :** Breadth First Search is used in copying garbage collection using Cheney's algorithm. Refer this and for details. Breadth First Search is preferred over Depth First Search because of better locality of reference:
8. **Cycle detection in undirected graph :** In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. In directed graph, only depth first search can be used.
9. **Ford–Fulkerson algorithm :** In Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worst case time complexity to O(VE2).
10. **To test if a graph is Bipartite :** We can either use Breadth First or Depth First Traversal.
11. **Path Finding :** We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.
12. **Finding all nodes within one connected component :** We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

Depth First Search (DFS)

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:

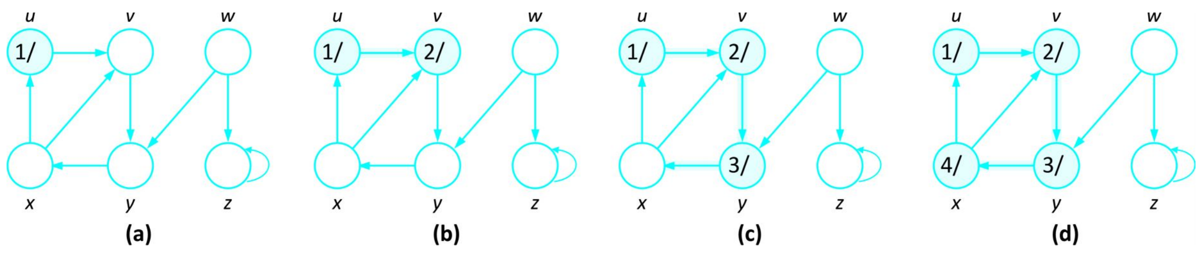
- Pick a starting node and push all its adjacent nodes into a stack.
- Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.

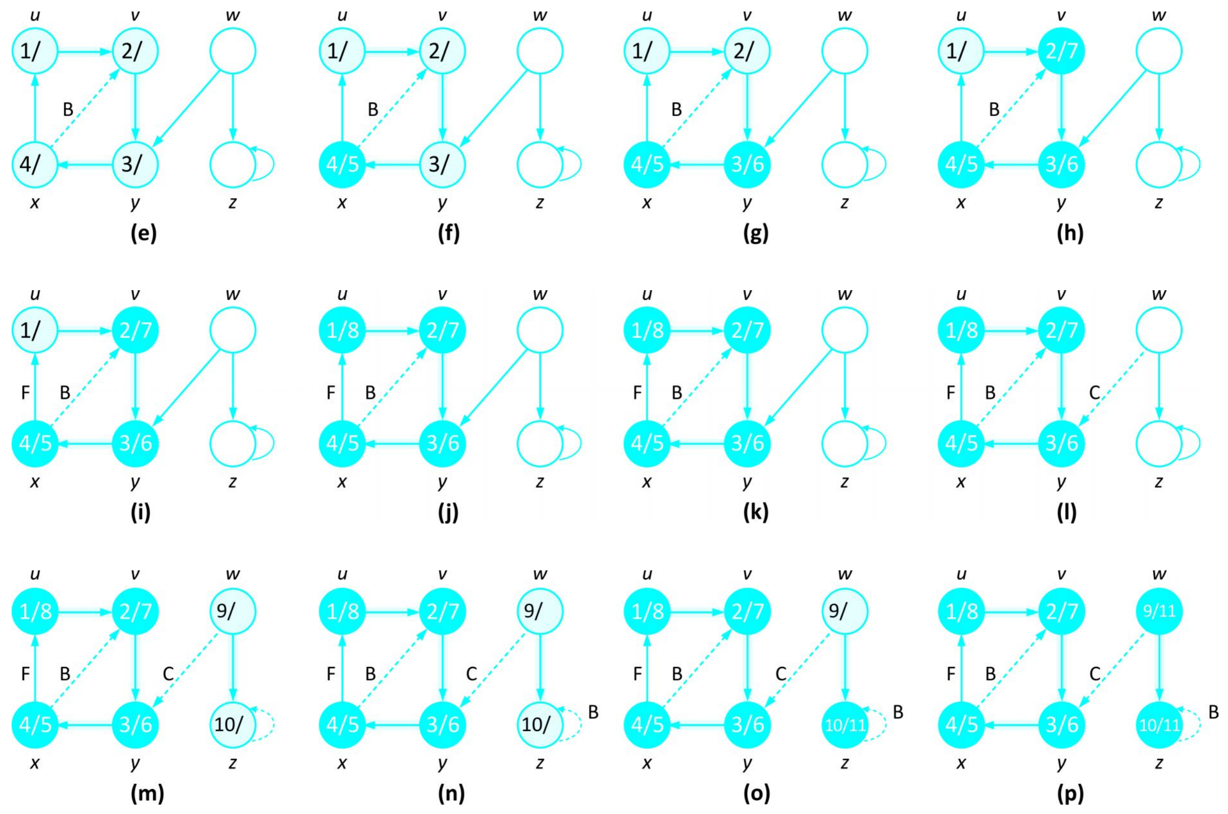
- Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

  As in breadth-first search, vertices are colored during the search to indicate their state. Each vertex is initially white, is grayed when it is ***discovered*** in the search, and is blackened when it is ***finished***, that is, when its adjacency list has been examined completely. This technique guarantees that each vertex ends up in exactly one depth-first tree, so that these trees are disjoint.

Algorithm :

```
DFS(G)
for each vertex u ∈ V[G]
    do color [u] ← WHITE
       p[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
    do if color[u] = WHITE
           then DFS-VISIT(u)
DFS-VISIT(u)
color[u] ← GRAY        White vertex u has just been discovered
time ← time + 1
d[u] ← time
for each v Î Adj[u]          Explore edge (u, v)
    do if color[v] = WHITE
          then π[v] ← u
                  DFS-VISIT(v)
color[u] BLACK         Blacken u; it is finished
f[u] ← time ← time + 1
```





The progress of the depth-first-search algorithm DFS on a directed path. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C or F according to whether they are back, cross, or forward edges. Vertices are timestamped by discovery time/finishing time.

Code :

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

template<typename T>
class Graph{

    map<T,list<T> > adjList;

public:
    Graph(){

    }
    void addEdge(T u, T v,bool bidir=true){
         adjList[u].push_back(v);
```

```
            if(bidir){
                  adjList[v].push_back(u);
            }
      }

      void print(){

            //Iterate over the map
            for(auto i:adjList){
                  cout<<i.first<<"->";

                  //i.second is LL
                  for(T entry:i.second){
                        cout<<entry<<",";
                  }
                  cout<<endl;
            }
      }
      void dfsHelper(T node,map<T,bool> &visited){
            //Whenever to come to a node, mark it visited
            visited[node] = true;
            cout<<node<<" ";
            //Try to find out a node which is neigbour of current node and not
yet visited
            for(T neighbour: adjList[node]){
                  if(!visited[neighbour]){
                        dfsHelper(neighbour,visited);
                  }
            }
      }
      void dfs(T src){
            map<T,bool> visited;
            dfsHelper(src,visited);
      }
};
int main(){

      Graph<int> g;
      g.addEdge(0,1);
      g.addEdge(1,2);
      g.addEdge(0,4);
```

```
     g.addEdge(2,4);
     g.addEdge(2,3);
     g.addEdge(3,4);
     g.addEdge(3,5);
     g.dfs(0);


return 0;
}
```

Time Complexity :

Time complexity of above algorithm is O(V + E).

Application of DFS :

1. For an unweighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.
2. **Detecting cycle in a graph :** A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edge.

Code for detect a cycle in a graph :

```
#include<iostream>
#include<map>
#include<list>
using namespace std;
template<typename T>
class Graph{
     map<T,list<T> > adjList;
public:
     Graph(){
     }
     void addEdge(T u, T v,bool bidir=true){
          adjList[u].push_back(v);
          if(bidir){
               adjList[v].push_back(u);
          }
     }
     bool isCyclicHelper(T node,map<T,bool> &visited,map<T,bool> &inStack){
```

```
                    //Processing the current node - Visited, InStack
                    visited[node] = true;
                    inStack[node] = true;
```

3. **Path Finding :** We can specialize the DFS algorithm to find a path between two given vertices u and z.
   (i) Call DFS(G, u) with u as the start vertex.
   (ii) Use a stack S to keep track of the path between the start vertex and the current vertex.
   (iii) As soon as destination vertex z is encountered, return the path as the contents of the stack.

4. **Topological Sorting** : In the field of computer science, a **topological sort** or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering. For instance, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another; in this application, a topological ordering is just a valid sequence for the tasks. A topological ordering is possible if and only if the graph has no directed cycles, that is, if it is a directed acyclic graph (DAG). Any DAG has at least one topological ordering.

Code for printing Topological sorting of DAG :

```
#include<iostream>
#include<map>
#include<list>
#include<queue>
using namespace std;
template<typename T>
class Graph{
      map<T,list<T> > adjList;
 public:
      Graph(){
      }
      void addEdge(T u, T v,bool bidir=true){
             adjList[u].push_back(v);
             if(bidir){
                    adjList[v].push_back(u);
             }
      }
      void topologicalSort(){
            queue<T> q;
            map<T,bool> visited;
            map<T,int> indegree;
            for(auto i:adjList){
                        //i is pair of node and its list
                        T node = i.first;
                        visited[node] = false;
                        indegree[node] = 0;
            }
            //Init the indegrees of all nodes
            for(auto i:adjList){
                  T u = i.first;
                  for(T v: adjList[u]){
                  indegree[v]++;
                  }
            }
            //Find out all the nodes with 0 indegree
            for(auto i:adjList){
                  T node = i.first;
                  if(indegree[node]==0){
                        q.push(node);
                  }
            }
            //Start with algorithm
            while(!q.empty()){
                  T node = q.front();
                  q.pop();
                  cout<<node<<"—>";
                  for(T neighbour:adjList[node]){
                        indegree[neighbour]—;
                  if(indegree[neighbour]==0){
                        q.push(neighbour);
                  }
            }
```

```
                    }
          }
};

intmain(){
          Graph<string> g;
          g.addEdge("English","ProgrammingLogic",false);
          g.addEdge("Maths","Programming Logic",false);
          g.addEdge("Programming Logic","HTML",false);
          g.addEdge("Programming Logic","Python",false);
          g.addEdge("Programming Logic","Java",false);
          g.addEdge("Programming Logic","JS",false);
          g.addEdge("Python","WebDev",false);
          g.addEdge("HTML","CSS",false);
          g.addEdge("CSS","JS",false);
          g.addEdge("JS","WebDev",false);
          g.addEdge("Java","WebDev",false);
          g.addEdge("Python","WebDev",false);
          g.topologicalSort();

return0;
}
```

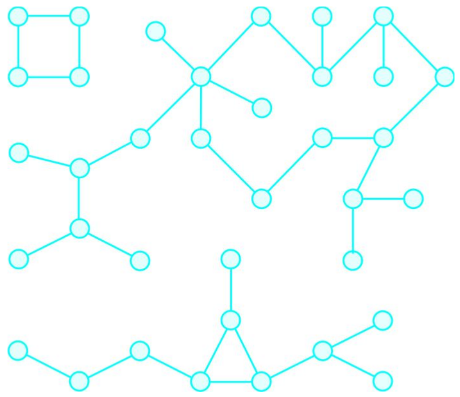
5. **To test if a graph is bipartite :** We can augment either BFS or DFS when we first discover a new vertex, color it opposited its parents, and for each other edge, check it doesn't link two vertices of the same color. The first vertex in any connected component can be red or black.

6. **Finding Strongly Connected Components of a graph :** A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex.

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

8. **For finding Connected Components :** In graph theory, a connected component (or just component) of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph.

Example **:**



In the above picture, there are three connected components.

Code for finding connected components :

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

template<typename T>
class Graph{
      map<T,list<T> > adjList;
public:
      Graph(){
      }
      void addEdge(T u, T v,bool bidir=true){
              adjList[u].push_back(v);
              if(bidir){
                       adjList[v].push_back(u);
              }
       }
       voidprint(){
            //Iterate over the map
            for(auto i:adjList){
                  cout<<i.first<<"->";
                  //i.second is LL
            for(T entry:i.second){
                  cout<<entry<<“,”;
            }
            cout<<endl;
      }
}
void dfsHelper(T node,map<T,bool> &visited){
      //Whenever to come to a node, mark it visited
      visited[node] = true;
      cout<<node<<“ “;
      //Try to find out a node which is neigbour of current node and not yet visited
      for(T neighbour: adjList[node]){
            if(!visited[neighbour]){
                  dfsHelper(neighbour,visited);
            }
      }
}
void dfs(T src){
      map<T,bool> visited;
      int component = 1;
      dfsHelper(src,visited);
      cout<<endl;
      for(auto i:adjList){
            T city = i.first;
            if(!visited[city]){
                  dfsHelper(city,visited);
                  component++;
            }
      }
cout<<endl;
            cout<<“The current graph had “<<component<<“ components”;
      }
};
int main(){
      Graph<string> g;
```

```
        g.addEdge("Amritsar","Jaipur");
        g.addEdge("Amritsar","Delhi");
        g.addEdge("Delhi","Jaipur");
        g.addEdge("Mumbai","Jaipur");
        g.addEdge("Mumbai","Bhopal");
        g.addEdge("Delhi","Bhopal");
        g.addEdge("Mumbai","Bangalore");
        g.addEdge("Agra","Delhi");
        g.addEdge("Andaman","Nicobar");
        g.dfs("Amritsar");
return 0;
}
```

Minimum Spanning Tree

What is a Spanning Tree?

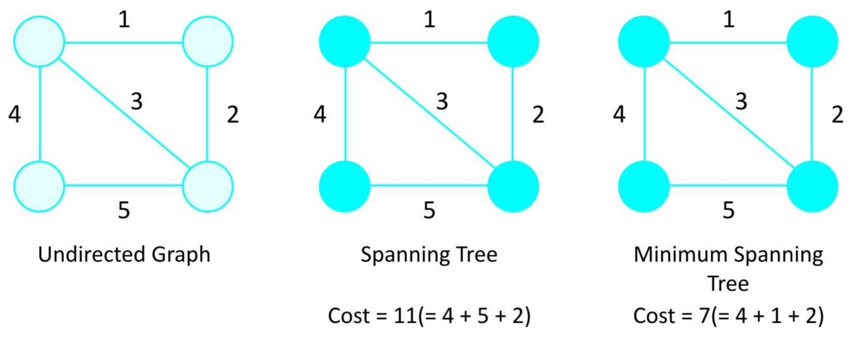
Given an undirected and connected graph G=(V,E) , a spanning tree of the graph G is a tree that spans G (that is, it includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G)

What is a Minimum Spanning Tree?

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.

Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum-cost weighted perfect matching. Other practical applications are:

1. Cluster Analysis **2.** Handwriting recognition **3.** Image segmentation



There are two famous algorithms for finding the Minimum Spanning Tree:

Kruskal’s Algorithm

Kruskal’s Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal’s algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

Algorithm Steps

- Sort the graph edges with respect to their weights.
- Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which doesn’t form a cycle , edges which connect only disconnected components.

So now the question is how to check if 2 vertices are connected or not ?

This could be done using DFS which starts from the first vertex, then check if the second vertex is visited or not. But DFS will make time complexity large as it has an order of O(V+E) where V is the number of vertices, E is the number of edges. So the best solution is **“Disjoint Sets”**:

Disjoint sets are sets whose intersection is the empty set so it means that they don’t have any element in common.

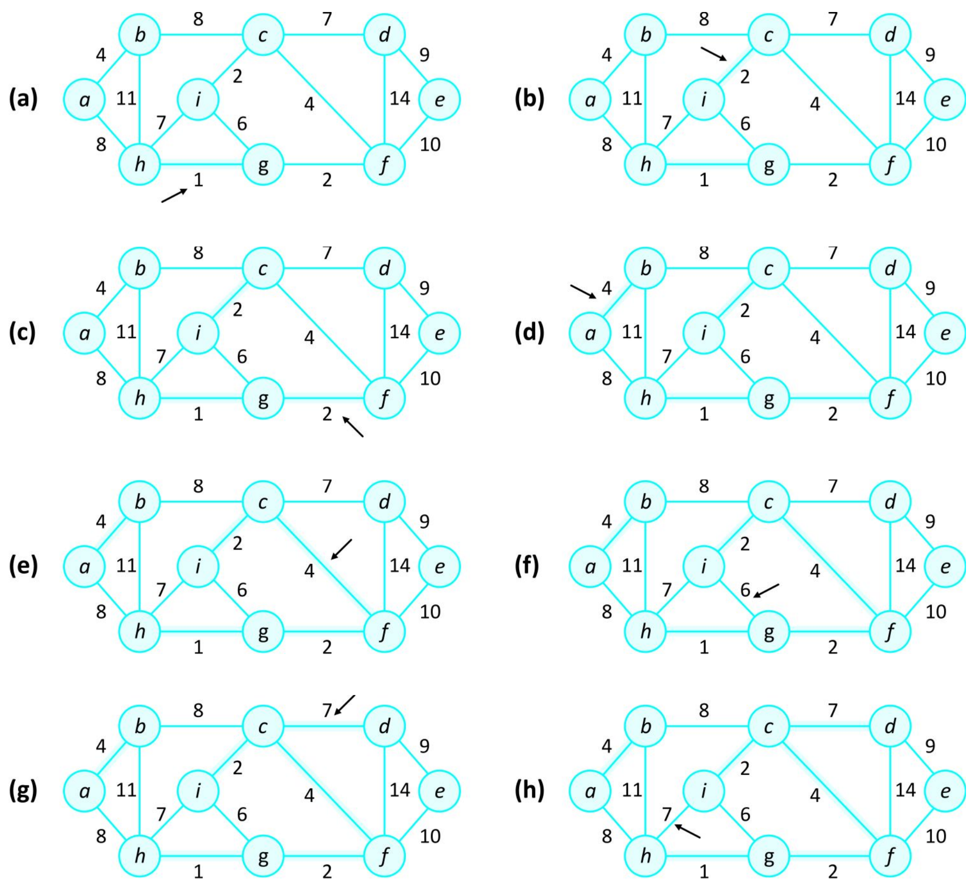
In Kruskal’s algorithm, at each iteration we will select the edge with the lowest weight. So, we will start with the lowest weighted edge first.

Note : Kruskal’s algorithm is a greedy algorithm, because at each step it adds to the forest an edge of least possible weight.

Algorithm :

```
MST-KRUSKAL(G, w)
A ← 0
For each vertex v ∈ V[G]
    Do MAKE-SET(v)
sort the edges of E into nondecreasing order by weight w
for each edge (u, v) ∈ E, taken in nondecreasing order by weight
    do if FIND-SET(u)≠ FIND-SET(v)
           then A ← A ∪ {(u, v)}
                UNION(u, v)
return A
```

Here **A** is the set which contains all the edges of minimum spanning tree.



The execution of Kruskal's algorithm on the graph from figure. Shaded edges belong to the forest A being grown. The edges are considered by the algorithm in sorted order by weight. An arrow points to the edge consideration at each step of the algorithm. If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.

Code :

```
#include <iostream>
#include <vector>
#include <utility>
#include <algorithm>

using namespace std;
const int MAX = 1e4 + 5;
int id[MAX], nodes, edges;
pair <long long, pair<int, int> > p[MAX];

void initialize()
{
   for(int i = 0;i < MAX;++i)
```

```
        id[i] = i;
}

int root(int x)
{
    while(id[x] != x)
    {
         id[x] = id[id[x]];
         x = id[x];
    }
    return x;
}

void union1(int x, int y)
{
    int p = root(x);
    int q = root(y);
    id[p] = id[q];
}

long long kruskal(pair<long long, pair<int, int> > p[])
{
    int x, y;
    long long cost, minimumCost = 0;
    for(int i = 0;i < edges;++i)
    {
          // Selecting edges one by one in increasing order from the beginning
         x = p[i].second.first;
         y = p[i].second.second;
         cost = p[i].first;
         // Check if the selected edge is creating a cycle or not
         if(root(x) != root(y))
         {
              minimumCost += cost;
              union1(x, y);
         }
    }
    return minimumCost;
}

int main()
{
    int x, y;
    long long weight, cost, minimumCost;
    initialize();
    cin >> nodes >> edges;
    for(int i = 0;i < edges;++i)
    {
        cin >> x >> y >> weight;
        p[i] = make_pair(weight, make_pair(x, y));
    }
    // Sort the edges in the ascending order
    sort(p, p + edges);
    minimumCost = kruskal(p);
    cout << minimumCost << endl;
    return 0;
}
```

Time Complexity :

The total running time complexity of kruskal algorithm is O(V log E).

Prim's Algorithm :

Prim's Algorithm also use Greedy approach to find the minimum spanning tree. In Prim's Algorithm we grow the spanning tree from a starting position. Unlike an edge in Kruskal's, we add vertex to the growing spanning tree in Prim's.

Algorithm Steps :

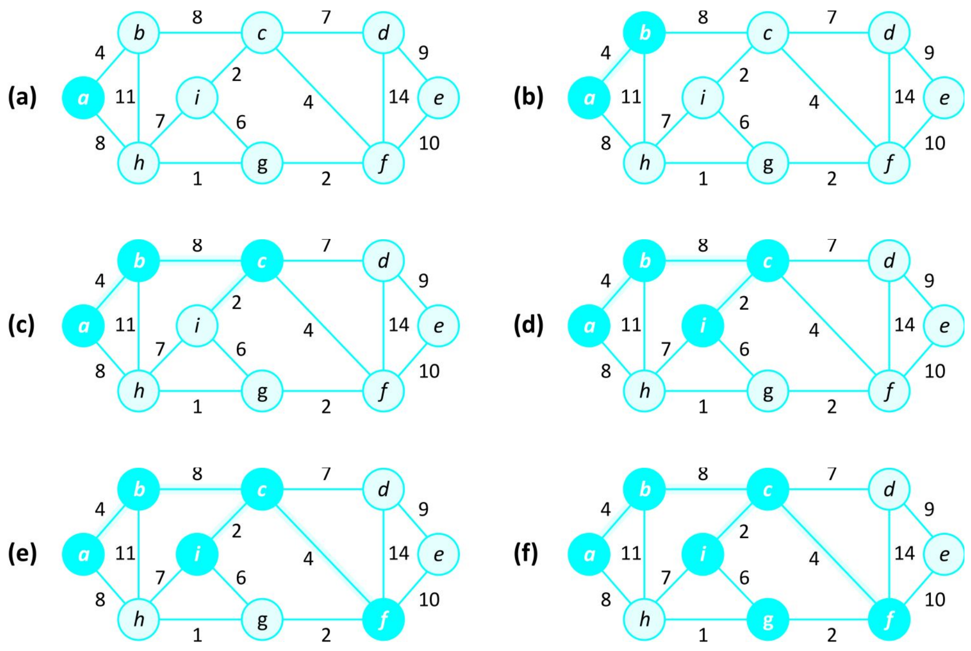
- Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
- Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.

- Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.

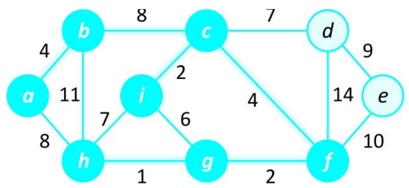
In Prim's Algorithm, we will start with an arbitrary node (it doesn't matter which one) and mark it. In each iteration we will mark a new vertex that is adjacent to the one that we have already marked. As a greedy algorithm, Prim's algorithm will select the cheapest edge and mark the vertex.

```
MST-PRISM(G, w, r)
   for each u ∈ V[G]
       do key[u] ← ∞
          p[u] ← NIL
   key[r] ← 0
   Q ← V[G]
   while Q ≠ 0
       do u ← EXTRACT-MIN(Q)
          for each v ∈ Adj[u]
              do if v ∈ Q and w(u, v) < key[v]
                   then π[v] ← u
                        key[v] ← w(u, v)
```

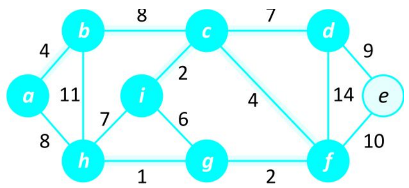
The prims algorithm works as shown in below graph.



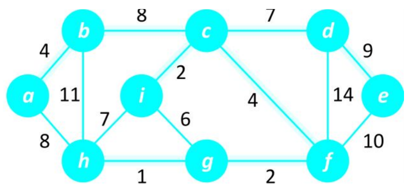
(g)



(h)



(i)



Code :

```
#include <iostream>
#include <vector>
#include <queue>
#include <functional>
#include <utility>
using namespace std;
const int MAX = 1e4 + 5;
typedef pair<long long, int> PII;
bool marked[MAX];
vector <PII> adj[MAX];

long long prim(int x)
{
    priority_queue<PII, vector<PII>, greater<PII> > Q;
   int y;
   long long minimumCost = 0;
   PII p;
   Q.push(make_pair(0, x));
   while(!Q.empty())
   {
        // Select the edge with minimum weight
        p = Q.top();
        Q.pop();
        x = p.second;
        // Checking for cycle
        if(marked[x] == true)
            continue;
        minimumCost += p.first;
        marked[x] = true;
        for(int i = 0;i < adj[x].size();++i)
        {
            y = adj[x][i].second;
            if(marked[y] == false)
                Q.push(adj[x][i]);
        }
    }
    return minimumCost;
}
int main()
{
    int nodes, edges, x, y;
    long long weight, minimumCost;
    cin >> nodes >> edges;
    for(int i = 0;i < edges;++i)
    {
        cin >> x >> y >> weight;
         adj[x].push_back(make_pair(weight, y));
         adj[y].push_back(make_pair(weight, x));
    }
    // Selecting 1 as the starting node
    minimumCost = prim(1);
    cout << minimumCost << endl;
    return 0;
}
```

Shortest Path Algorithms

The shortest path problem is about finding a path between 2 vertices in a graph such that the total sum of the edges weights is minimum.

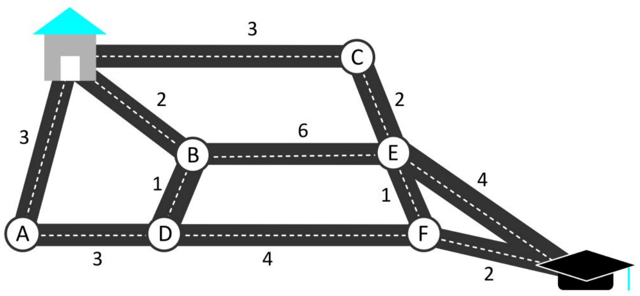
This problem could be solved easily using (BFS) if all edge weights were (1), but here weights can take any value. There are some algorithm discussed below which work to find Shortest path between two vertices.

1. Single Source Shortest Path Algorithm :

In this kind of problem, we need to find the shortest path of single vertices to all other vertices.

Example **:**

Find the shortest path from home to school in the following graph:



A weighted graph representing roads from home to school

The shortest path, which could be fund using Dijkstra's algorithm, is

Home → B → D → F → School

There are two algorithm works to find Single source shortest path from a given graph.

A. Dijkstra's Algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are nonnegative.

Algorithm Steps:

- Set all vertices distances = infinity except for the source vertex, set the source distance = 0.
- Push the source vertex in a min-priority queue in the form (distance , vertex), as the comparison in the min-priority queue will be according to vertices distances.
- Pop the vertex with the minimum distance from the priority queue (at first the popped vertex = source).
- Update the distances of the connected vertices to the popped vertex in case of "current vertex distance + edge weight < next vertex distance", then push the vertex with the new distance to the priority queue.
- If the popped vertex is visited before, just continue without using it.
- Apply the same algorithm again until the priority queue is empty.

Pseudo Code :

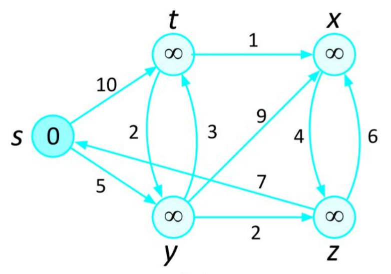
```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S ← 0
Q ← V[G]
while Q ≠ 0
```

```
        do u ← EXTRACT-MIN(Q)
            S ← S ∪ {u}
            for each vertex v ∈ Adj[u]
                 do RELAX(u, v, w)
```

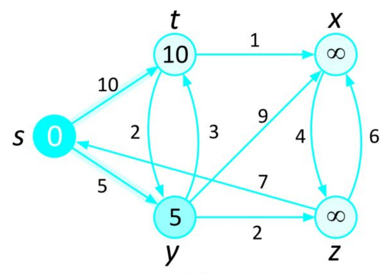
For relaxing an edge of given graph, Algorithm works likes this :

```
 RELAX(u, v, w)
    if d[v] > d[u] + w(u, v)
        then d[v] ← d[u] + w(u, v)
              π[v] ← u
```

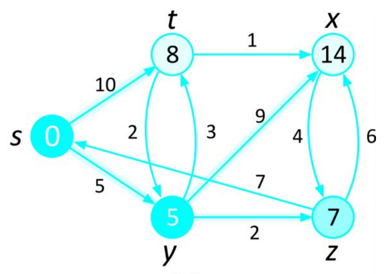
Example **:**



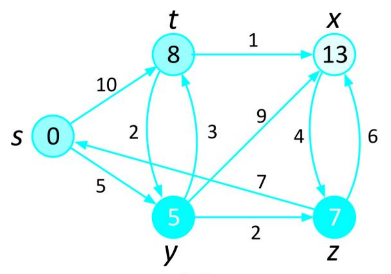
(a)



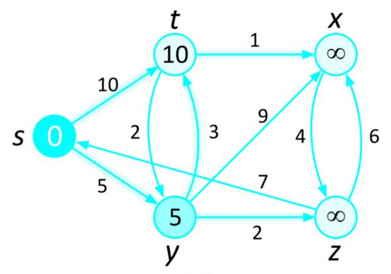
(b)



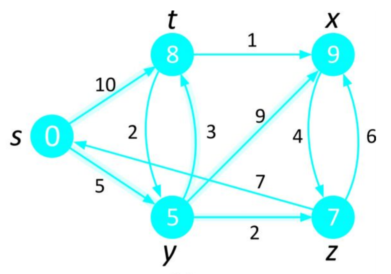
(b)



(d)



(e)



(f)

The execution of Dijkstra's algorithm. The source *s* is the leftmost vertex. The shortest-path estimates are shown within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set *S*, and white vertices are in the min-priority queue *Q* = *V* – *S*. **(a)** The situation just before the first iteration of the **while** loop. The shaded vertex has the minimum *d* value and is chosen as vertex *u* . **(b)–(f)** The situation after each successive iteration of the **while** loop. The shaded vertex in each part is chosen as vertex *u* in line 5 of the next iteration. The *d* and À values shown in part (f) are the final values.

Code :

```
#include<bits/stdc++.h>
using namespace std;
template<typename T>
class Graph{
    unordered_map<T, list<pair<T,int> > > m;
```

```
public:
    void addEdge(T u,T v,int dist,bool bidir=true){
         m[u].push_back(make_pair(v,dist));
        if(bidir){
             m[v].push_back(make_pair(u,dist));
        }
    }
    void printAdj(){
        //Let try to print the adj list
        //Iterate over all the key value pairs in the map
        for(auto j:m){
            cout<<j.first<<"->";
            //Iterater over the list of cities
            for(auto l: j.second){
                     cout<<"("<<l.first<<","<<l.second<<")";
            }
            cout<<endl;
        }
    }
    void dijsktraSSSP(T src){
        unordered_map<T,int> dist;
        //Set all distance to infinity
        for(auto j:m){
            dist[j.first] = INT_MAX;
        }
       //Make a set to find a out node with the minimum distance
        set<pair<int, T> > s;
       dist[src] = 0;
        s.insert(make_pair(0,src));
        while(!s.empty()){
            //Find the pair at the front.
            auto p =   *(s.begin());
            T node = p.second;
            int nodeDist = p.first;
            s.erase(s.begin());
            //Iterate over neighbours/children of the current node
             for(auto childPair: m[node]){
                  if(nodeDist + childPair.second < dist[childPair.first]){
                      //In the set updation of a particular is not possible
                       // we have to remove the old pair, and insert the new pair
to   simulation updation
                      T dest = childPair.first;
                       auto f = s.find( make_pair(dist[dest],dest));
                      if(f!=s.end()){
                          s.erase(f);
                     }
                      //Insert the new pair
                       dist[dest] = nodeDist + childPair.second;
                       s.insert(make_pair(dist[dest],dest));
                 }
            }
        }
         //Lets print distance to all other node from src
        for(auto d:dist){
             cout<<d.first<<“ is located at distance of  “<<d.second<<endl;
        }
    }
};
int main(){
   Graph<int> g;
    g.addEdge(1,2,1);
    g.addEdge(1,3,4);
    g.addEdge(2,3,1);
    g.addEdge(3,4,2);
    g.addEdge(1,4,7);
    //g.printAdj();
  //  g.dijsktraSSSP(1);
   Graph<string> india;
    india.addEdge(“Amritsar”,”Delhi”,1);
    india.addEdge(“Amritsar”,”Jaipur”,4);
    india.addEdge(“Jaipur”,”Delhi”,2);
    india.addEdge("Jaipur","Mumbai",8);
    india.addEdge("Bhopal","Agra",2);
    india.addEdge("Mumbai","Bhopal",3);
    india.addEdge("Agra","Delhi",1);
   //india.printAdj();
    india.dijsktraSSSP("Amritsar");


return 0;
}
```

Time Complexity : Time Complexity of Dijkstra's Algorithm is O(V^2) but with min-priority queue it drops down to O(V + E logV).

B. Bellman Ford's Algorithm

Bellman Ford's algorithm is used to find the shortest paths from the source vertex to all other vertices in a weighted graph. It depends on the following concept: Shortest path contains at most n – 1 edges, because the shortest path couldn't have a cycle.

So why shortest path shouldn't have a cycle ?

There is no need to pass a vertex again, because the shortest path to all other vertices could be found without the need for a second visit for any vertices.

Algorithm Steps:

- The outer loop traverses from 0 to n – 1.
- Loop over all edges, check if the next node distance > current node distance + edge weight, in this case update the next node distance to "current node distance + edge weight".

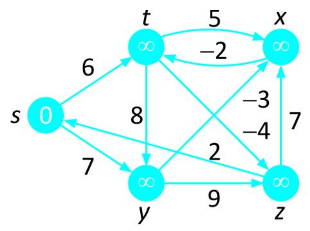
This algorithm depends on the relaxation principle where the shortest distance for all vertices is gradually replaced by more accurate values until eventually reaching the optimum solution. In the beginning all vertices have a distance of "Infinity", but only the distance of the source vertex = 0, then update all the connected vertices with the new distances (source vertex distance + edge weights), then apply the same concept for the new vertices with new distances and so on.

Pseudo Code :

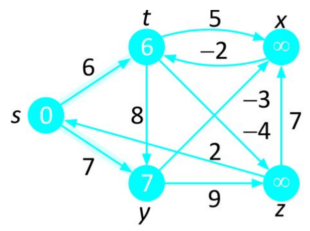
```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i ← 1 to |V[G]| - 1
      do for each edge (u, v) ∈ E[G]
              do RELAX(u, v, w)
for each edge (u, v) ∈ E[G]
```

```
        do if d[v] > d[u] + w(u, v)
                then return FALSE
return TRUE
```

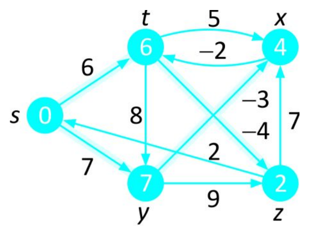
Example :



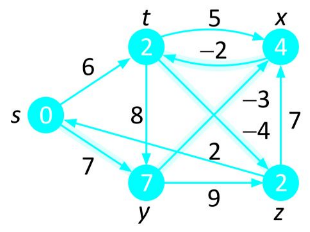
(a)



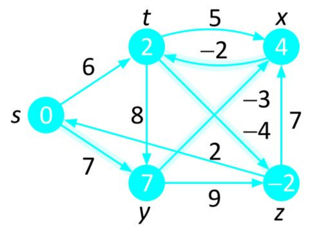
(b)



(c)



(d)



(e)

All-Pairs Shortest Paths :

The all-pairs shortest path problem is the determination of the shortest graph distances between every pair of vertices in a given graph. The problem can be solved using n applications of Dijkstra's algorithm at each vertex if graph doesn't contains negative weight. We use two algorithms for finding All-pair shortest path of a graph.

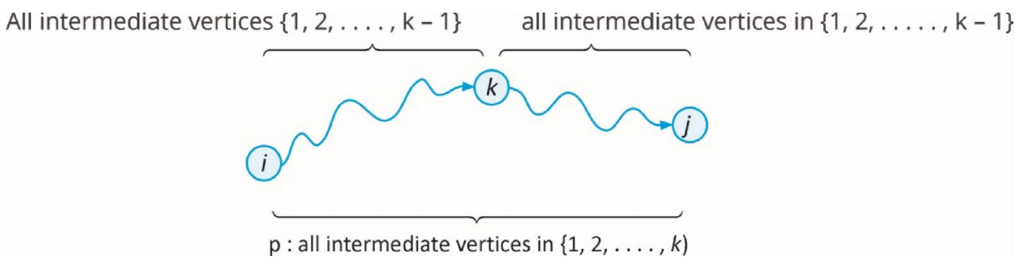
1. Floyd-Warshall's Algorithm **2.** Johnson's Algorithm

Floyd-Warshall's Algorithm

Here we use a different dynamic-programming formulation to solve the all-pairs shortest-paths problem on a directed graph $G = (V, E)$. The resulting algorithm, known as the ***Floyd-Warshall algorithm***. Floyd–Warshall's Algorithm is used to find the shortest paths between between all pairs of vertices in a graph, where each edge in the graph has a weight which is positive or negative. The biggest advantage of using this algorithm is that all the shortest distances between any 2 vertices could be calculated in $O(V^3)$, where V is the number of vertices in a graph.

The Floyd-Warshall algorithm is based on the following observation. Under our assumption that the vertices of G are $V = \{1, 2, . . . , n\}$, let us consider a subset $\{1,2,...,k\}$ of vertices for some k. For any pair of vertices i, $j – V$, consider all paths from i to j whose intermediate vertices are all drawn from $\{1, 2, . . . , k\}$, and let p be a minimum-weight path from among them. The Floyd- Warshall algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set $\{1, 2, . . . , k – 1\}$. The relationship depends on whether or not k is an intermediate vertex of path p. There would two possibilities :

1. If *k* is not an intermediate vertex of path *p*, then all intermediate vertices of path *p* are in the set {1,2,...,*k* – 1}. Thus, a shortest path from vertex *i* to vertex *j* with all intermediate vertices in the set {1,2,...,*k* –1} is also a shortest path from *i* to *j* with all intermediate vertices in the set {1, 2, . . . , *k*}.
2. If *k* is an intermediate vertex of path *p*, then we break *p* down into *i* -> *k* -> *j* as *p*1 is a shortest path from *i* to *k* *and p*2 is a shortest path from *k* to *j. Since* *p*1 is a shortest path from *i* to *k* with all intermediate vertices in the set {1, 2, . . . , *k*}. Because vertex *k* is not an intermediate vertex of path *p*1, we see that *p*1 is a shortest path from *i* to *k* with all intermediate vertices in the set {1, 2, . . . , *k* – 1}. Similarly, *p*2 is a shortest path from vertex *k* to vertex *j* with all intermediate vertices in the set {1,2,...,*k* – 1}.


All intermediate vertices {1, 2, ... , k – 1}
all intermediate vertices in {1, 2, ... , k – 1}
k
i
j
p : all intermediate vertices in {1, 2, ... , k)

Path P is a shortest path from vertices *i* to vertex j, and k is the highest=numbered intermediate vertex of p. Path p_1, the portion of path p from vertex i to vertex k, has all intermediate vertices in the set {1, 2, ... , k – 1}. The same holds for path p_2 from vertex k to vertex j.

Let $d_{ij}^{(k)}$ be the weight of shortest path from vertex *i* to vertex *j* for which all intermediate vertices are in the set {1, 2, . . . , *k*}. A recursive definition following the above discussion is given by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if} \quad k = 0 \\ \min\left(d_{ij}^{(k-1)},\, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if} \quad k \geq 1 \end{cases}$$

The Algorithm Steps:

For a graph with V vertices:

- Initialize the shortest paths between any 2 vertices with Infinity.
- Find all pair shortest paths that use 0 intermediate vertices, then find the shortest paths that use 1 intermediate vertex and so on.. until using all V vertices as intermediate nodes.
- Minimize the shortest paths between any 2 pairs in the previous operation.
- For any 2 vertices (i , j) , one should actually minimize the distances between this pair using the first K nodes, so the shortest path will be:

Min (dist[i][k] + dist[k][j] , dist[i][j])

dist[i][k] represents the shortest path that only uses the first K vertices, dist[k][j] represents the shortest path between the pair k , j. As the shortest path will be a concatenation of the shortest path from i to k, then from k to j.

Constructing a shortest path

For construction of the shortest path, we can use the concept of predecessor matrix π to construct the path. We can compute the predecessor matrix π "on-line" just as the Floyd-Warshall algorithm computes the matrices $D(k)$. Specifically, we compute a sequence of matrices $\pi^{(k)}$ where $\pi^{(k)}$ is defined to be the predecessor of vertex *j* on a shortest path from vertex *i* with all intermediate vertices in the set {1,2,...,*k*}. We can give a recursive formulation of $\pi^{(k)}$ as

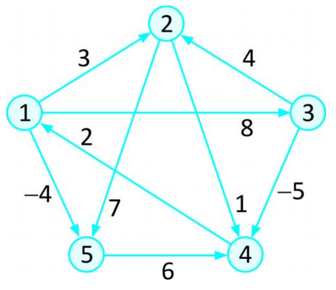
For k = 0 :

$$\pi_{ij}^{(0)} = \begin{cases} \text{NILL} & \text{if} \quad i = j \text{ or } w_{ij} = \infty \\ i & \text{if} \quad i \neq j \text{ and } w_{ij} < \infty \end{cases}$$

And For 1<=k<=V

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if} \quad d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ i & \text{if} \quad d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

Consider the below graph and find distance (D) and parents (π) matrix for this graph



Computed distance (D) and parents (π) matrix for the above graph :

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Implemented Code :

```
#include<bits/stdc++.h>
#include<iostream>
#include<unordered_map>
#define INF 99999
#define V 5
using namespace std;
template<typename T>
class Graph
{
   unordered_map<T, list<pair<T, int> > > m;
   int graph[V][V];
   int parent[V][V];

public:
 //Adjacency list representation of the graph
 void addEdge(T u, T v, int dist,bool bidir = false)
 {
      m[u].push_back(make_pair(v,dist));
     /*if(bidir){
           m[v].push_back(make_pair(u,dist));
      }*/

  }
  //Print Adjacency list
  void printAdj()
  {
      for(auto j:m)
      {
           cout<<j.first<<"->";
           for(auto l: j.second)
           {
                cout<<"("<<l.first<<","<<l.second<<")";
           }
           cout<<endl;
      }

  }
  void matrix_form(int u, int v , int w)
  {
    graph[u-1][v-1] = w;
    parent[u-1][v-1] = u;
    return;
  }

//Adjacency matrix representation of the graph
  void matrix_form2()
  {
      for(int i=0;i<V;i++)
      {
           for(int j=0;j<V;j++)
           {
                if(i==j)
                {
                     graph[i][j]=0;
                     parent[i][j]=0;
                }
```

```
              else
              {
                   graph[i][j]=INF;
                   parent[i][j]=0;
              }
          }
     }
     return;
 }
 //Print Adjacency matrix
  void print_matrix()
 {
      for(int i=0;i<V;i++)
     {
          for(int j=0;j<V;j++)
         {
              if(graph[i][j]==INF)
              cout<<"INF"<<"   ";
              else
              cout<<graph[i][j]<<"    ";
         }
         cout<<endl;
     }

 }

//Print predecessor matrix
 void printParents(int p[][V])
 {

      cout<<"Parents Matrix"<<"\n";
   for (int i = 0; i < V; i++)
   {
       for (int j = 0; j < V; j++)
       {
            if(p[i][j]!=0)
```

```
           cout<<p[i][j]<<"   ";
          else
          {
           cout<<"NIL"<<" ";
           //cout<<p[i][j]<<"   ";
          }
       }
         cout<<endl;
   }
 }
 //All pair shortest path matrix i.e D
 void printSolution(int dist[][V])
{

             cout<<"Following matrix shows the shortest distances"
            " between every pair of vertices"<<"\n";
   for (int i = 0; i < V; i++)
   {
       for (int j = 0; j < V; j++)
       {
            if (dist[i][j] == INF)
                cout<<"INF   ";
            else
                cout<<dist[i][j]<<"   ";
       }
         cout<<endl;
   }
}
 //Print the shortest path , distance and all intermediate vertex
 void print_path(int p[][V], int d[][V])
{
  // cout<<"Hello"<<"\n";
   for(int i=0;i<V;i++)
   {
       for(int j=0;j<V;j++)
       {
```

```
                // cout<<“Hello1”<<“\n”;
                if(i!=j)
                {
                 cout<<“Shortest path from “<<i+1<<“ to “<< j+1<<“ => “;
                 cout<<“[Total Distance : “<<d[i][j]<<“ ( Path : “;
                 //cout<<“Hello1”<<“\n”;
                int k=j;
                int l=0;
                int a[V];
                a[l++] = j+1;
                 while(p[i][k]!=i+1)
                {
                      //cout<<“Hello1”<<“\n”;
                      a[l++]=p[i][k];
                      k=p[i][k]-1;
                }
                a[l]=i+1;
                 //cout<<“Hello1”<<“\n”;
                 for(int r =l;r>0;r—)
                {
                      //cout<<“Hello1”<<“\n”;
                  cout<<a[r]<<“ —> “;
                }
                 cout<<a[0]<<“ )”;
                cout<<endl;

                }
         }
    }
}
 //Floyd Warshall Algorithm
void floydWarshall ()
{
   int dist[V][V], i, j, k;
   int parent2[V][V];
   for (i = 0; i < V; i++)
```

```
         for (j = 0; j < V; j++)
              dist[i][j] = graph[i][j];

    for(int i=0;i<V;i++)
    {
         for(int j=0;j<V;j++)
         {
              parent2[i][j] = parent[i][j];
         }
    }

    for (k = 0; k < V; k++)
    {
         for (i = 0; i < V; i++)
         {
              for (j = 0; j < V; j++)
              {
                    if (dist[i][k] + dist[k][j] < dist[i][j])
                   {
                        dist[i][j] = dist[i][k] + dist[k][j];
                        parent2[i][j] = parent2[k][j];
                   }
              }
         }
    }

   printSolution(dist);
   cout<<“\n\n”;
   printParents(parent2);
   cout<<“\n\n”;
   cout<<“All pair shortest path is given below :”<<“\n”;
   print_path(parent2, dist);
}
};
//Main Function
int main()
```

```
{
   Graph<int> g;
    g.matrix_form2();
  // g.make_parent();
 //Intializing the graph given in above picture
    g.addEdge(1,2,3);
    g.matrix_form(1,2,3);
    g.addEdge(1,3,8);
    g.matrix_form(1,3,8);
    g.addEdge(1,5,-4);
    g.matrix_form(1,5,-4);
    g.addEdge(2,4,1);
    g.matrix_form(2,4,1);
    g.addEdge(2,5,7);
    g.matrix_form(2,5,7);
    g.addEdge(3,2,4);
    g.matrix_form(3,2,4);
    g.addEdge(4,3,-5);
    g.matrix_form(4,3,-5);
    g.addEdge(4,1,2);
    g.matrix_form(4,1,2);
    g.addEdge(5,4,6);
    g.matrix_form(5,4,6);
    cout<<“Graph in the form of adjecency list representation : “<<“\n”;
    g.printAdj();
    cout<<“Graph in the form of matrix representation : “<<“\n”;
    g.print_matrix();
    cout<<“\n\n”;
    g.floydWarshall();


}
```

DO IT YOURSELVES

- ❑ Implement a BFS based algorithm to find the shortest route in **Snakes and Laddders Game**
- ❑ Implement a program to classify each of the graph as **forward edge, back edge** or **cross edge.**
- ❑ Find the shortest path in a weighed graph where each egde is 1 or 2.
- ❑ Read about **Hamiltonian Cycle**. Implement an optimisied **Travelling Salesman Problem** solution using Dynamic Programming. [Refer Tutorial]
- ❑ Read about -

  Articulation Points
  Bridges
  Eulerian Paths and Circuits
- ❑ Read about **Strongly Connected Components** and its algorithms -

  - Kosaraju's algorithm
  - Tarjan's algorithm
- ❑ Solve **Holdiay Accomodation(HOLI)** on Spoj [Refer Tutorial]
- ❑ Implement a **Flood Fill Algorithm**, to color various components of a given pattern. Pattern boundary is represented by '#'
- ❑ Implement an algorithm for '**Splitwise App**', to minimize the cash flow between a group of friends.
- ❑ Solve the Graphs Assignment at **HackerBlocks**

SELF STUDY NOTES

SELF STUDY NOTES