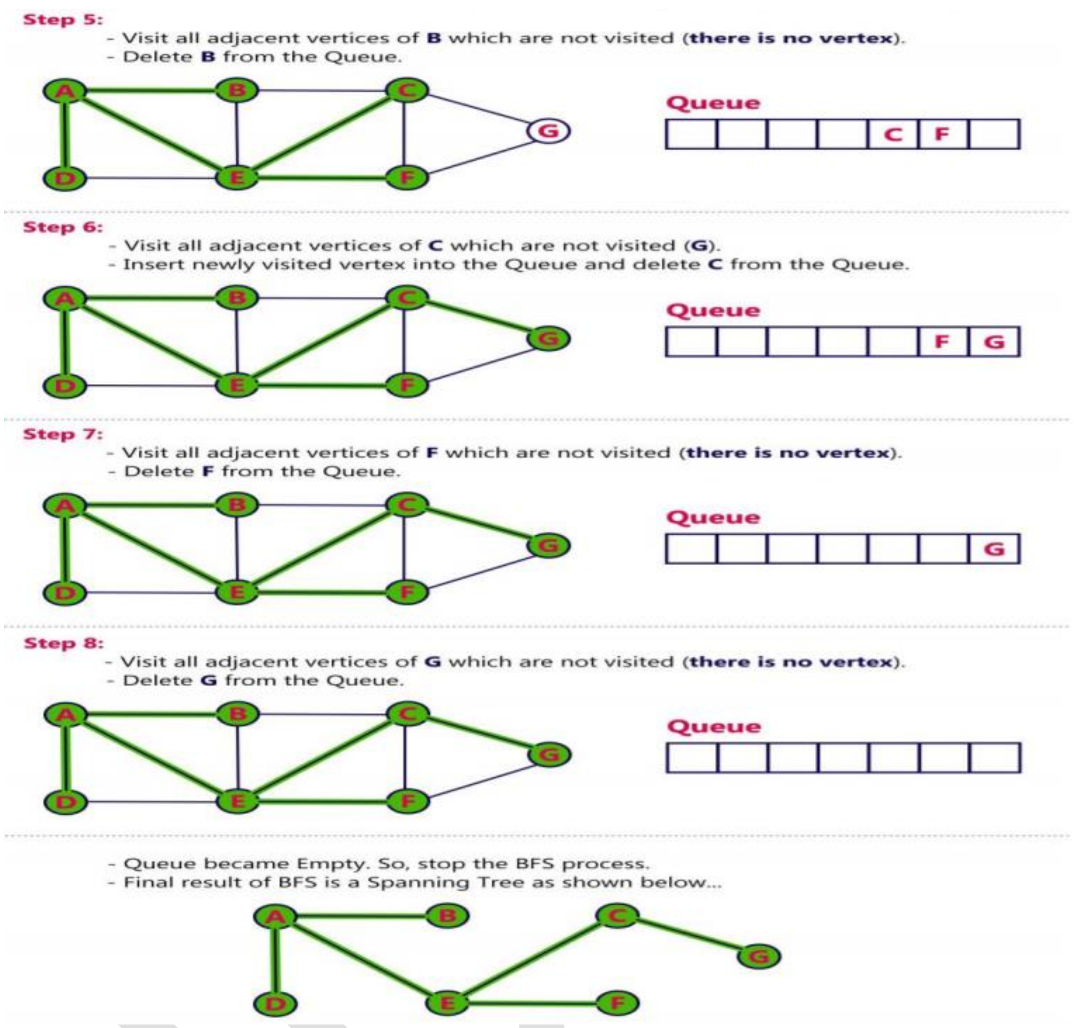


Step 5:
- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

Step 6:
- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

Step 7:
- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

Step 8:
- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...

12.3 Graph Traversals

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.

12.3.1 In-order

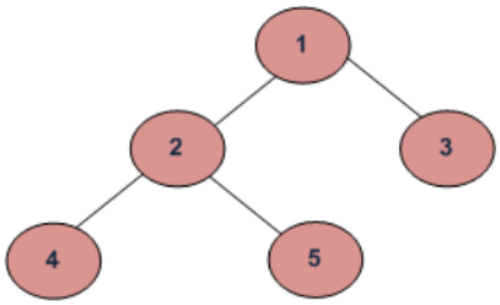
Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Uses of Inorder

In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal s reversed can be used.

Example-1: Inorder traversal for the below-given tree is 4 2 5 1 3.



Example-2: Consider input as given below:
Input:

```
      1
    / \
   3    2
```

Output: 3 1 2

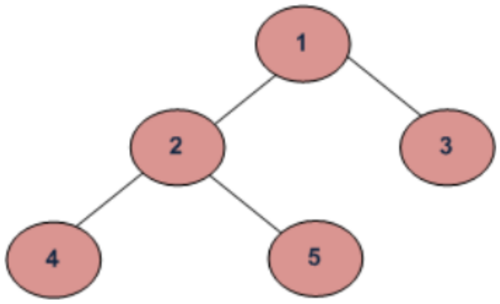
12.3.2 Pre-order

```
Algorithm Preorder(tree)
  1. Visit the root.
  2. Traverse the left subtree, i.e., call Preorder(left-subtree)
  3. Traverse the right subtree, i.e., call Preorder(right-subtree)
```

Uses of Preorder

- Preorder traversal is used to create a copy of the tree.
- Preorder traversal is also used to get prefix expression on of an expression tree.

Example-1: Preorder traversal for the below given figure is 1 2 4 5 3.



Example-2: Consider Input as given below:
Input:

```
       1
      /
    4
  /   \
4       2
```

Output: 1 4 4 2

12.3.3 Post-order

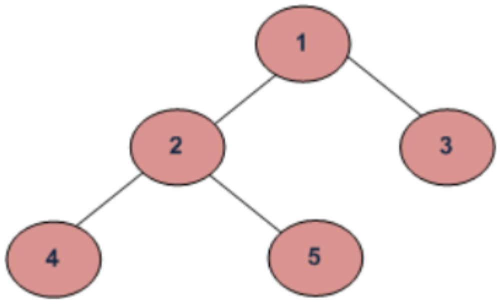
```
Algorithm Postorder(tree)
  1. Traverse the left subtree, i.e., call Postorder(left-subtree)
  2. Traverse the right subtree, i.e., call Postorder(right-subtree)
```

3. Visit the root.

Uses of Postorder

- Postorder traversal is used to delete the tree.
- Postorder traversal is also useful to get the postfix expression of an expression tree.

Example-1: Postorder traversal for the below given figure is 4 5 2 3 1.



Example-2: Consider input as given below:

Input:

```
      19
    /    \
   10     8
  /   \
 11    13
```

Output: 11 13 10 8 19

12.4 Summary

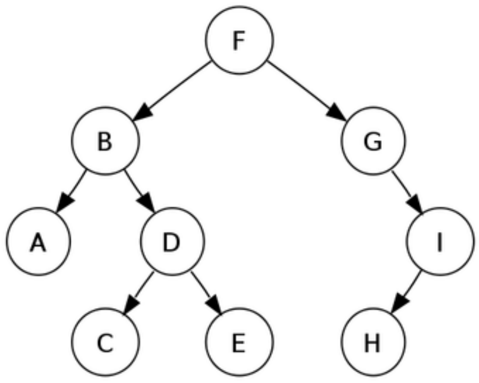
In this chapter, we have seen what graphs are, what are the various terminologies used in graphs. We have also seen the graph operations like Breadth First Search (BFS) & Depth First Search (DFS). We have seen the carious graph traversal methods like Inorder, Preorder and Postorder.

12.5 Review Your Learning

- Can you explain what are graphs?
- Can you explain what is BFS and DFS?
- Are you able to write the graph nodes sequence using Inorder, Preorderand :ostorder traversal methods?
- Can you relate day to day rea problems using graphical notations?

12.6 Questions

1. Draw a directed graph with five vertices and seven edges. Exactly one of the edges should be a loop, and do not have any multiple edges.
2. Draw an undirected graph with five edges and four vertices. The vertices should be called v1, v2, v3 and v4--and there must be a path of length three from v1 to v4. Draw a squiggly line along this path from v1 to v4.
3. Explain Inorder, Preorder and Postorder traversal methods. Writeh sequence for following given graph.



4. Explain BFS and DFA Algorithms with examples.
5. Draw the directed graph that corresponds to this adjacency matrix:

	0	1	2	3
0	True	False	True	False
1	True	False	False	False
2	False	False	False	True
3	True	False	True	False

12.7 Further Reading

- https://lpuguidecom.files.wordpress.com/2017/04/fundamentals-of-data-structures-ellis-horowitz-sartaj-sahni.pdf
- https://www.guru99.com/data-structure-algorithms-books.html
- https://edutechlearners.com/data-structures-with-c-by-schaum-series-pdf/

12.8 References

1. http://www.musaliarcollege.com/e-Books/CSE/Data%20structures%20algorithms%20and%20applications%20in%20C.pdf
2. https://lpuguidecom.files.wordpress.com/2017/04/fundamentals-of-data-structures-ellis-horowitz-sartaj-sahni.pdf
3. https://www.geeksforgeeks.org/

Unit 6: Chapter 13

Graph Algorithms

Graph Algorithms

13.0 Objectives

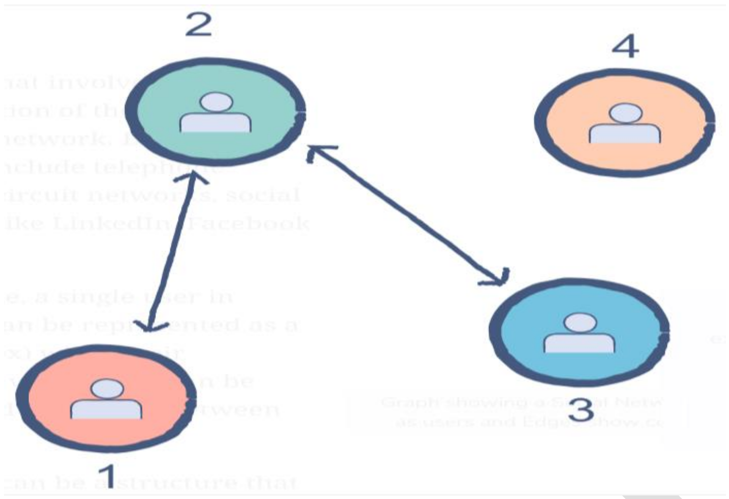
5. Identify Minimum Spanning Tree in a given graph
6. Explain Kruskal's Algorithm
7. Explain Prim's Algorithm
8. Analyse working with Kruskal's Algorithm, Prim's Algorithm
9. Explain Greedy Algorithms.

13.1 Introduction

A graph is a common data structure that consists of a finite set of nodes (or vertices) and a set of edges connecting them.

A pair (x,y) is referred to as an edge, which communicates that the x vertex connects to the y vertex.

In the examples below, circles represent vertices, while lines represent edges.



Graphs are used to solve real-life problems that involve representation of the problem space as a network. Examples of networks include telephone networks, circuit networks, social networks (like LinkedIn, Facebook etc.).

For example, a single user in Facebook can be represented as a node (vertex) while their connection with others can be represented as an edge between nodes.

Each node can be a structure that contains information like user's id, name, gender, etc.

Types of graphs:

A. Undirected Graph:

In an undirected graph, nodes are connected by edges that are all bidirectional. For example, if an edge connects node 1 and 2, we can traverse from node 1 to node 2, and from node 2 to 1.

B. Directed Graph

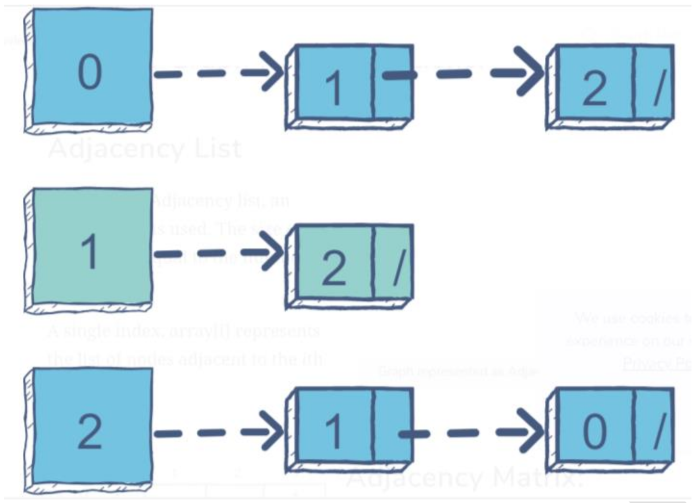
In a directed graph, nodes are connected by directed edges – they only go in one direction. For example, if an edge connects node 1 and 2, but the arrowhead points towards 2, we can only traverse from node 1 to node 2 – not in the opposite direction.

Types of Graph Representations:

A. Adjacency List

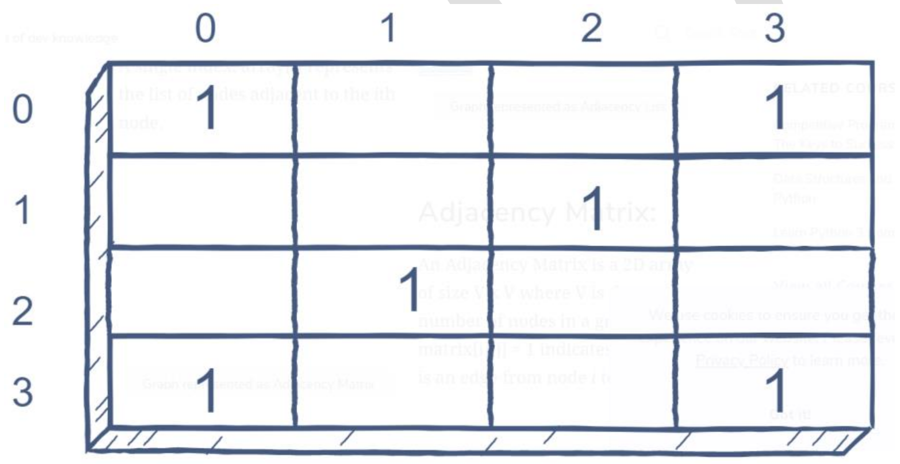
To create an Adjacency list, an array of lists is used. The size of the array is equal to the number of nodes.

A single index, array[i] represents the list of nodes adjacent to the ith node.



B. Adjacency Matrix:

An Adjacency Matrix is a 2D array of size V x V where V is the number of nodes in a graph. A slot matrix[i][j] = 1 indicates that there is an edge from node i to node j.



$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 1 & & & 1 \\ 1 & & & 1 & \\ 2 & & 1 & & \\ 3 & 1 & & & 1 \end{array}$$

13.2 Minimum Spanning Tree

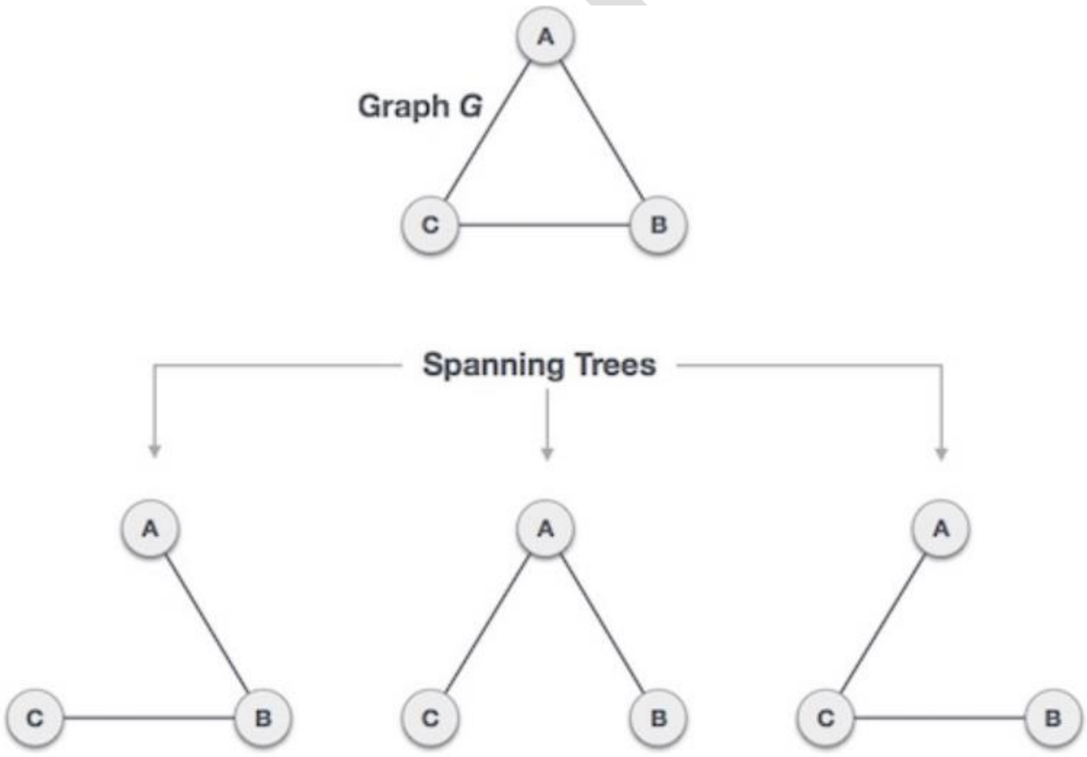
13.2.1 Spanning Tree

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.

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).

Following figure shows the original undirected graph and its various possible spanning trees.



We found three spanning trees off one complete graph. A complete undirected graph can have maximum n^{n-2} number of spanning trees, where n is the number of nodes. In the above addressed example, n is 3, hence 33−2 = 3 spanning trees are possible.

General Properties of Spanning Tree

As one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G −

1. A connected graph G can have more than one spanning tree.

2. All possible spanning trees of graph G, have the same number of edges and vertices.
3. The spanning tree does not have any cycle (loops).
4. Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected.
5. Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is maximally acyclic.

Mathematical Properties of Spanning Tree

1. Spanning tree has n-1 edges, where n is the number of nodes (vertices).
2. From a complete graph, by removing maximum e - n + 1 edges, we can construct a spanning tree.
3. A complete graph can have maximum nn-2 number of spanning trees.
4. Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

Application of Spanning Tree

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are −

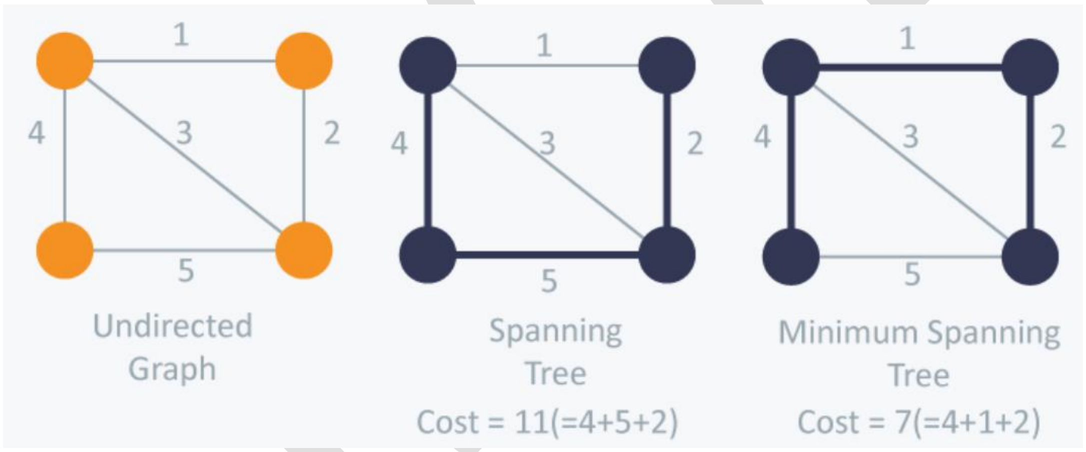
- Civil Network Planning
- Computer Network Routing Protocol
- Cluster Analysis

13.2.2 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:

1. Kruskal’s Algorithm
2. Prim’s Algorithm

These both algorithms are Greedy Algorithms.

13.3 Graph Algorithms

13.3.1 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:

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

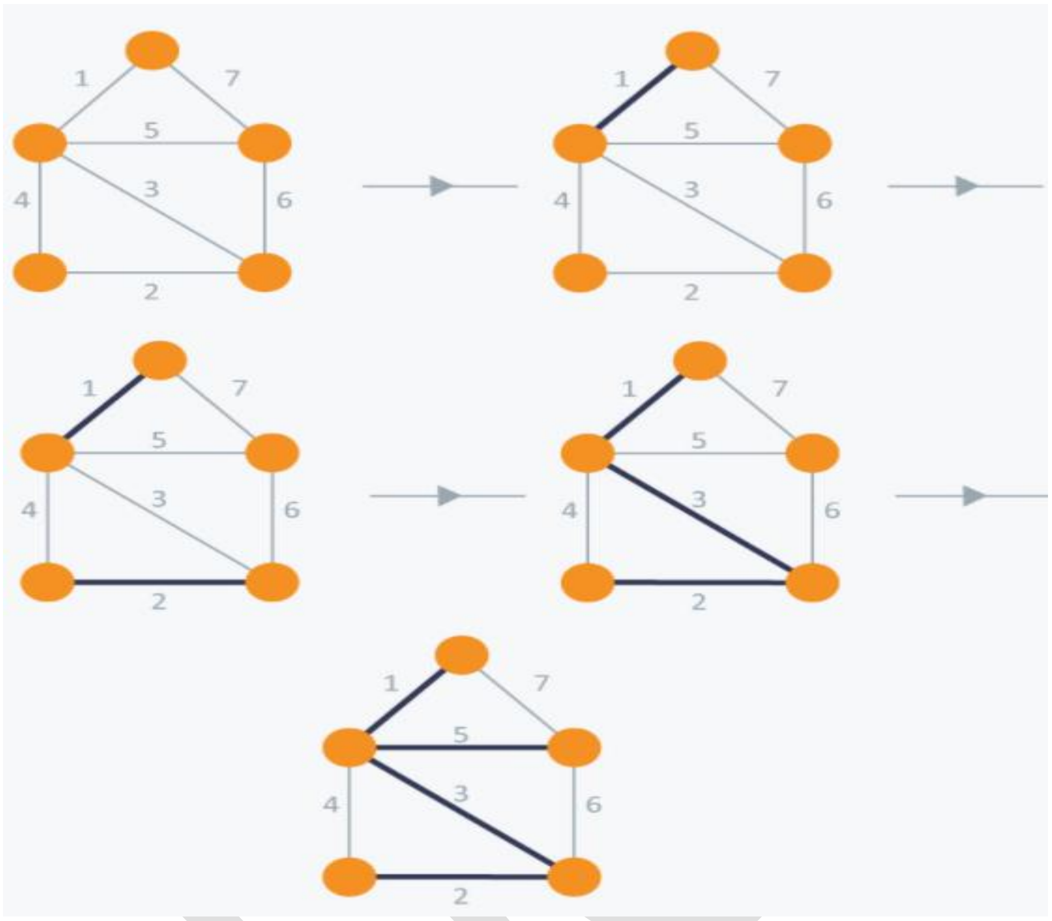
So now the question is how to check if 2 vertices are connected or not ?

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".**

DisjointSets:

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

Consider following example:



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 i.e., the edges with weight 1. After that we will select the second lowest weighted edge i.e., edge with weight 2. Notice these two edges are totally disjoint. Now, the next edge will be the third lowest weighted edge i.e., edge with weight 3, which connects the two disjoint pieces of the graph. Now, we are not allowed to pick the edge with weight 4, that will create a cycle and we can't have any cycles. So we will select the fifth lowest weighted edge i.e., edge with weight 5. Now the other two edges will create cycles so we will ignore them. In the end, we end up with a minimum spanning tree with total cost 11 (= 1 + 2 + 3 + 5).

Program:

```
#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 longkruskal(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;

}
```

TimeComplexity:

Time Complexity:

In Kruskal's algorithm, most time consuming operation is sorting because the total complexity of the Disjoint-Set operations will be O(E log V), which is the overall Time Complexity of the algorithm.

13.3.2 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.

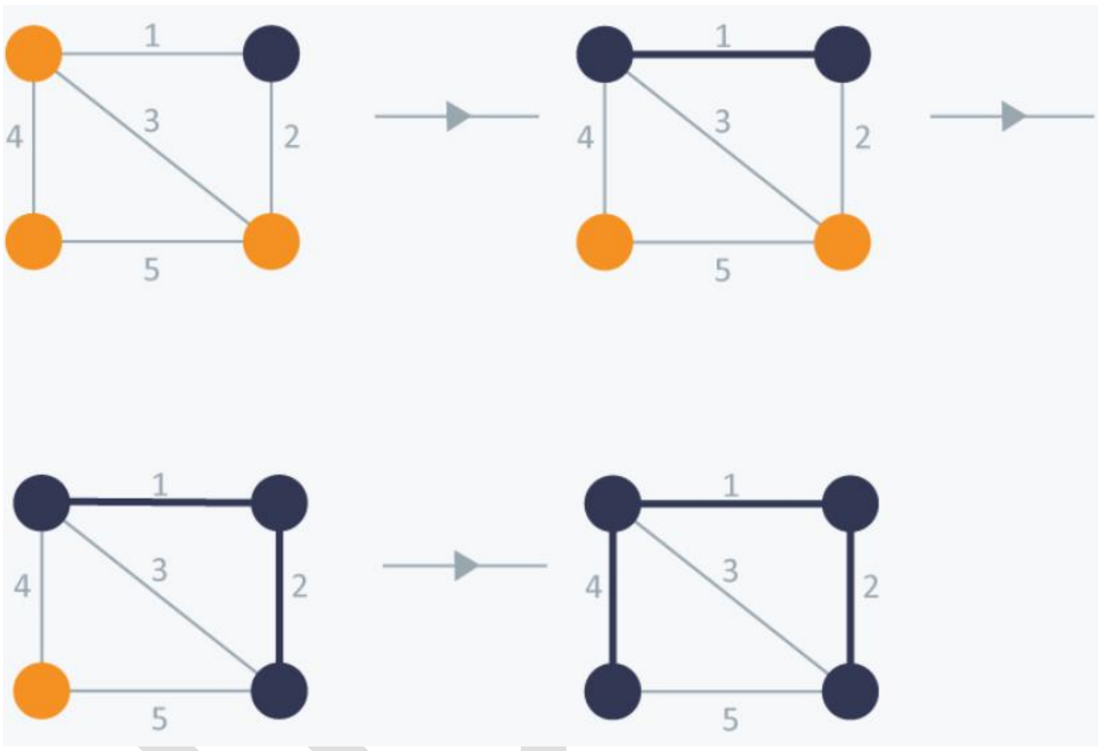
Algorithm Steps:

1. 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.
2. 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.

3. 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.

Consider the example below:



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. So we will simply choose the edge with weight 1. In the next iteration we have three options, edges with weight 2, 3 and 4. So, we will select the edge with weight 2 and mark the vertex. Now again we have three options, edges with weight 3, 4 and 5. But we can't choose edge with weight 3 as it is creating a cycle. So we will select the edge with weight 4 and we end up with the minimum spanning tree of total cost 7 (= 1 + 2 +4).

Program:

```
#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 longprim(int x)

{

priority_queue<PII, vector<PII>, greater<PII>>Q;

   int y;

   long longminimumCost = 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;
```

}

Time Complexity:

The time complexity of the Prim's Algorithm is O((V+E)logV) because each vertex is inserted in the priority queue only once and insertion in priority queue take logarithmic time.

13.3 Summary

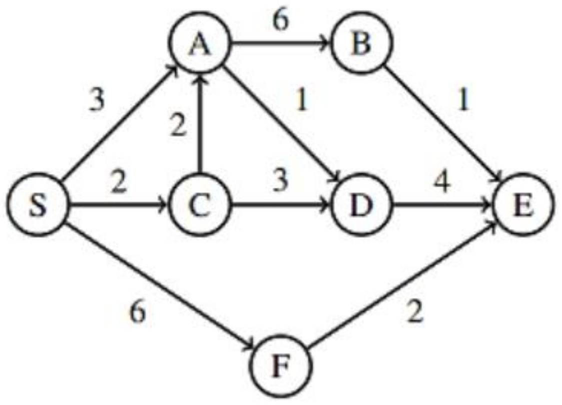
In this chapter, we have studied Minimum Spanning Tree (MST) and its use in various algorithms like Kruskal's Algorithm, Prim's Algorithm. We have also seen all-pair shortest path algorithms like Floyd Warshall's Algorithm, Dijkstra's Algorithm.

13.4 Review Your Learning

1. Are you able to explain what is Minimum Spanning Tree (MST)?
2. Are you able to explain Greedy algorithms working on graph data structures?
3. Are you able to analyse the condition when to use Prim's and Kruskal's Algorithm?
4. Are you able to find shortest path using graph algorithms?
5. Can you explain the applications of Graph data structures in day-to-day life?

13.5 Questions

1. Explain Minimum Spanning Tree (MST). Explain its use in various algorithms which works on graphs.
2. Explain applications of Graph data structures in day-to-day life?
3. Explain various graph algorithms to find shortest path using Greedy Approach?
4. Find shortest path using Prim's Algorithm on graph given below:



5. Explain difference between Greedy and Dynamic programming approach. Explain the examples of graph algorithms in each categories.

13.6 Further Reading

1. https://runestone.academy/runestone/books/published/pythonds/Graphs/DijkstrasAlgorithm.html
2. https://www.oreilly.com/library/view/data-structures-and/9781118771334/18_chap14.html
3. https://medium.com/javarevisited/10-best-books-for-data-structure-and-algorithms-for-beginners-in-java-c-c-and-python-5e3d9b478eb1
4. https://examradar.com/data-structure-graph-mcq-based-online-test-3/
5. https://www.tutorialspoint.com/data_structures_algorithms/data_structures_algorithms_online_quiz.htm

13.7 References

1. http://www.nitjsr.ac.in/course_assignment/CS01CS1302A%20Book%20Fundamentals%20of%20Data%20Structure%20(1982)%20by%20Ellis%20Horowitz%20and%20Sartaj%20Sahni.pdf
2. https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/
3. https://www.geeksforgeeks.org/prims-mst-for-adjacency-list-representation-greedy-algo-6/
4. https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/
5. http://wccclab.cs.nchu.edu.tw/www/images/Data_Structure_105/chapter6.pdf

Unit 6: Chapter 14

Dynamic Graph Algorithms

Dynamic Graph Algorithms

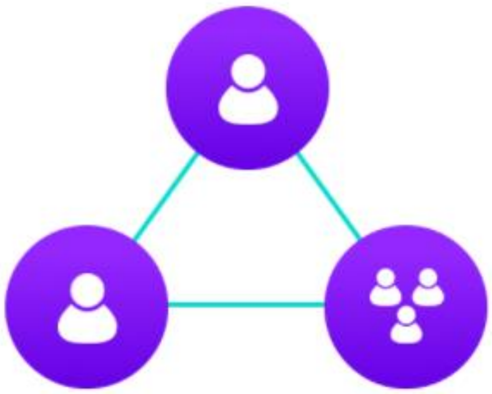
14.0 Objectives

10. Analyse working with Kruskal's Algorithm, Prim's Algorithm, Warshall's Algorithm
11. Explain Shortest Path Algorithm using Dijkstra's Algorithm
12. Explain difference between single source shortest path and all pair shortest path algorithms.
13. Explain difference between greedy and dynamic algorithm categories.
14. Analyse the applications of graphs in day-to-day life

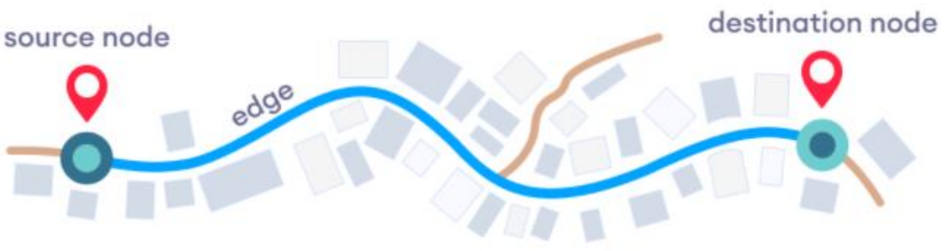
14.1 Introduction

A Graph is a network of interconnected items. Each item is known as a node and the connection between them is known as the edge.

You probably use social media like Facebook, LinkedIn, Instagram, and so on. Social media is a great example of a graph being used. Social media uses graphs to store information about each user. Here, every user is a node just like in Graph. And, if one user, let's call him Jack, becomes friends with another user, Rose, then there exists an edge (connection) between Jack and Rose. Likewise, the more we are connected with people, the nodes and edges of the graph keep on increasing.



Similarly, Google Map is another example where Graphs are used. In the case of the Google Map, every location is considered as nodes, and roads between locations are considered as edges. And, when one has to move from one location to another, the Google Map uses various Graph-based algorithms to find the shortest path. We will discuss this later in this blog.



Need for Dynamic Graph Algorithms:

The goal of a dynamic graph algorithm is to support query and update operations as quickly as possible (usually much faster than recomputing from scratch). Graphs subject to insertions only, or deletions only, but not both. Graphs subject to intermixed sequences of insertions and deletions.

14.2 Dynamic Graph Algorithms

Let us say that we have a machine, and to determine its state at time t, we have certain quantities called state variables. There will be certain times when we have to make a decision which affects the state of the system, which may or may not be known to us in advance. These decisions or changes are equivalent to transformations of state variables. The results of the previous decisions help us in choosing the future ones.

What do we conclude from this? We need to break up a problem into a series of overlapping sub-problems and build up solutions to larger and larger sub-problems. If you are given a problem, which can be broken down into smaller sub-problems, and these smaller sub-problems can still be broken into smaller ones - and if you manage to find out that there are some overlapping sub-problems, then you've encountered a DP problem.

Some famous Dynamic Programming algorithms are:

- Unix diff for comparing two files
- Bellman-Ford for shortest path routing in networks
- TeX the ancestor of LaTeX
- WASP - Winning and Score Predictor

The core idea of Dynamic Programming is to avoid repeated work by remembering partial results and this concept finds it application in a lot of real life situations.

In programming, Dynamic Programming is a powerful technique that allows one to solve different types of problems in time O(n2) or O(n3) for which a naive approach would take exponential time.

Dynamic Programming and Recursion:

Dynamic programming is basically, recursion plus using common sense. What it means is that recursion allows you to express the value of a function in terms of other values of that function. Where the common sense tells you that if you implement your function in a way that the recursive calls are done in advance, and stored for easy access, it will make your program faster. This is what we call Memorization - it is memorizing the results of some specific states, which can then be later accessed to solve other sub-problems.

The intuition behind dynamic programming is that we trade space for time, i.e. to say that instead of calculating all the states taking a lot of time but no space, we take up space to store the results of all the sub-problems to save time later.

Let's try to understand this by taking an example of Fibonacci numbers.

Fibonacci (n) = 1; if n = 0

Fibonacci (n) = 1; if n = 1

Fibonacci (n) = Fibonacci(n-1) + Fibonacci(n-2)

So, the first few numbers in this series will be: 1, 1, 2, 3, 5, 8, 13, 21... and so on!

Majority of the Dynamic Programming problems can be categorized into two types:

1. Optimization problems.
2. Combinatorial problems.

The optimization problems expect you to select a feasible solution, so that the value of the required function is minimized or maximized. Combinatorial problems expect you to figure out the number of ways to do something, or the probability of some event happening.

Every Dynamic Programming problem has a schema to be followed:

- Show that the problem can be broken down into optimal sub-problems.
- Recursively define the value of the solution by expressing it in terms of optimal solutions for smaller sub-problems.
- Compute the value of the optimal solution in bottom-up fashion.
- Construct an optimal solution from the computed information.

Bottom up vs. Top Down:

Bottom Up - I'm going to learn programming. Then, I will start practicing. Then, I will start taking part in contests. Then, I'll practice even more and try to improve. After working hard like crazy, I'll be an amazing coder.

Top Down - I will be an amazing coder. How? I will work hard like crazy. How? I'll practice more and try to improve. How? I'll start taking part in contests. Then? I'll practicing. How? I'm going to learn programming.

Dynamic Programming in Graph Data Structures:

Dynamic programming is "an algorithmic technique which is usually based on a recurrent formula and one (or some) starting states." When it's applied to graphs, we can solve for the shortest paths with one source or shortest paths for every pair.

Examples: Bellman-Ford Algorithm, Floyd Warshall's Algorithm, Dijkstra's Algorithm

14.2.1 Floyd Warshall's Algorithm

The Floyd Warshall Algorithm is for solving the All-Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph. Floyd Warshall Algorithm is an example of dynamic programming approach.

Floyd Warshall's Algorithm has the following main advantages-

1. It is extremely simple.
2. It is easy to implement.

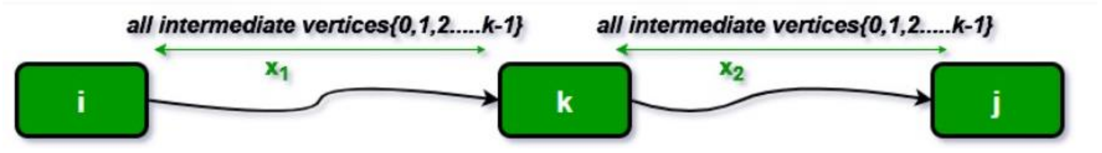
Algorithm Steps:

1. Initialize the solution matrix same as the input graph matrix as a first step.
2. Then update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and updates all shortest paths

which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.

- k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.
- k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j] if dist[i][j] >dist[i][k] + dist[k][j]

The following figure shows the above optimal substructure property in the all-pairs shortest path problem.


all intermediate vertices{0,1,2.....k-1}
all intermediate vertices{0,1,2.....k-1}
x1
x2
i
k
j

Program:

```
// C++ Program for Floyd Warshall Algorithm

#include <bits/stdc++.h>

using namespace std;


// Number of vertices in the graph

#define V 4
```

```
/* Define Infinite as a large enough value.This value will be used for vertices not connected to each other */

#define INF 99999


// A function to print the solution matrix

void printSolution(int dist[][V]);


// Solves the all-pairs shortest path problem using Floyd Warshall algorithm

void floydWarshall (int graph[][V])

{

        /* dist[][] will be the output matrix that will finally have the shortest distances between every pair of vertices */

        int dist[V][V], i, j, k;


        /* Initialize the solution matrix same as input graph matrix. Or we can say the initial values of shortest distances are based on shortest paths considering no intermediate vertex. */

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

                for (j = 0; j < V; j++)

                        dist[i][j] = graph[i][j];

        /* Add all vertices one by one to the set of intermediate vertices. ---> Before start of an iteration, we have shortest distances between all pairs of vertices such that the     shortest distances consider only the vertices in set {0, 1, 2, .. k-1} as intermediate vertices. ----> After the end of an iteration, vertex no. k is added to the set of intermediate vertices and the set becomes {0, 1, 2, .. k} */

        for (k = 0; k < V; k++)
```

```
          {
                    // Pick all vertices as source one by one
                    for (i = 0; i< V; i++)
                    {
                              // Pick all vertices as destination for the above picked source
                              for (j = 0; j < V; j++)
                              {
                                        // If vertex k is on the shortest path from i to j, then
update the
                                        // value of dist[i][j]
                                        if (dist[i][k] + dist[k][j] <dist[i][j])
                                                  dist[i][j] = dist[i][k] + dist[k][j];
                              }
                    }
          }
          // Print the shortest distance matrix
          printSolution(dist);
}

/* A utility function to print solution */
void printSolution(int dist[][V])
{
          cout<<"The following matrix shows the shortest distances"
                              " between every pair of vertices \n";
```

```
        for (int i = 0; i< V; i++)
        {
                for (int j = 0; j < V; j++)
                {
                        if (dist[i][j] == INF)
                                cout<<"INF"<<"        ";
                        else
                                cout<<dist[i][j]<<"      ";
                }
                cout<<endl;
        }
}
// Driver code
int main()
{
        /* Let us create the weighted graph */
        int graph[V][V] = { {0, 5, INF, 10},
                                                {INF, 0, 3, INF},
                                                {INF, INF, 0, 1},
                                                {INF, INF, INF, 0}
                                        };
        // Print the solution
        floydWarshall(graph);
        return 0;
```

}

Time Complexity-

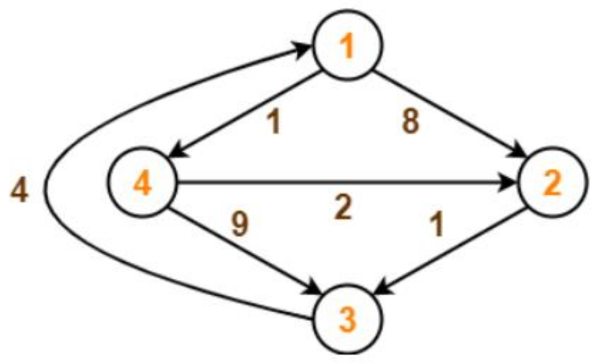
- Floyd Warshall Algorithm consists of three loops over all the nodes.
- The inner most loop consists of only constant complexity operations.
- Hence, the asymptotic complexity of Floyd Warshall algorithm is O(n3).
- Here, n is the number of nodes in the given graph.

When Floyd Warshall Algorithm Is Used?

- Floyd Warshall Algorithm is best suited for dense graphs.
- This is because its complexity depends only on the number of vertices in the given graph.
- For sparse graphs, Johnson's Algorithm is more suitable.

Example-1:

Consider the following directed weighted graph. Using Floyd Warshall Algorithm, find the shortest path distance between every pair of vertices.



Step-1:

- Remove all the self-loops and parallel edges (keeping the lowest weight edge) from the graph.
- In the given graph, there are neither self-edges nor parallel edges.

Step-2:

Write the initial distance matrix.

- It represents the distance between every pair of vertices in the form of given weights.
- For diagonal elements (representing self-loops), distance value = 0.
- For vertices having a direct edge between them, distance value = weight of that edge.
- For vertices having no direct edge between them, distance value = ∞.

Initial distance matrix for the given graph is-

$$D_0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & \infty & 0 & \infty \\ 4 & \infty & 2 & 9 & 0 \end{array}$$

Step-3:

Using Floyd Warshall Algorithm, write the following 4 matrices-

$$D_1 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & 12 & 0 & 5 \\ 4 & \infty & 2 & 9 & 0 \end{array}$$

$$D_2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & 9 & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & 12 & 0 & 5 \\ 4 & \infty & 2 & 3 & 0 \end{array}$$

$$D_3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & 9 & 1 \\ 2 & 5 & 0 & 1 & 6 \\ 3 & 4 & 12 & 0 & 5 \\ 4 & 7 & 2 & 3 & 0 \end{array}$$

$$D_4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 4 & 1 \\ 2 & 5 & 0 & 1 & 6 \\ 3 & 4 & 7 & 0 & 5 \\ 4 & 7 & 2 & 3 & 0 \end{array}$$

Please note:

- In the above problem, there are 4 vertices in the given graph.
- So, there will be total 4 matrices of order 4 x 4 in the solution excluding the initial distance matrix.
- Diagonal elements of each matrix will always be 0.

The last matrix D4 represents the shortest path distance between every pair of vertices.

14.2.2 Dijkstra's Algorithms

Dijkstra's algorithm, published in 1959 and named after its creator Dutch computer scientist Edsger Dijkstra, can be applied on a weighted graph. The graph can either be directed or undirected. One stipulation to using the algorithm is that the graph needs to have a nonnegative weight on every edge.

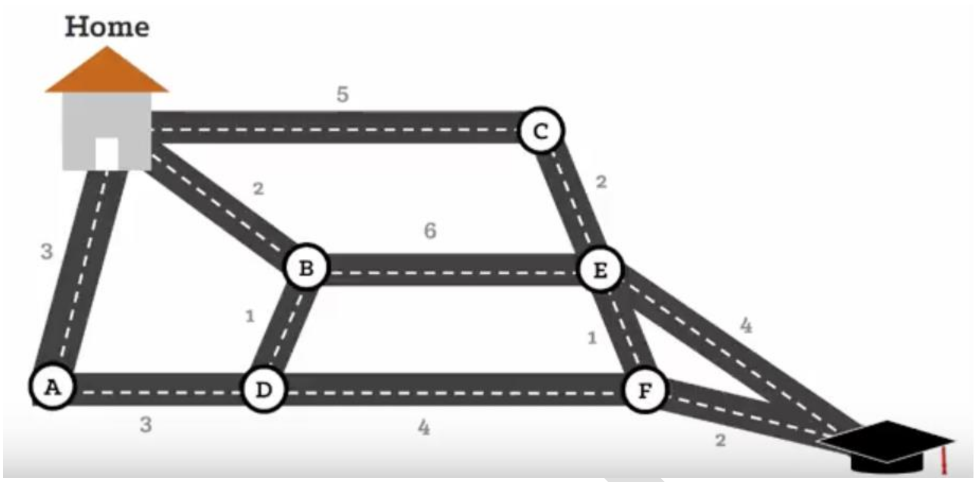
If you are given a directed or undirected weighted graph with n vertices and m edges. The weights of all edges are non-negative. You are also given a starting vertex s. To find the lengths of the shortest paths from a starting vertex s to all other vertices, and output the shortest paths themselves, we use Dijkstra's Algorithm.

This problem is also called **single-source shortest paths problem**.

One algorithm for finding the shortest path from a starting node to a target node in a weighted graph is Dijkstra's algorithm. The algorithm creates a tree of shortest paths from the starting vertex, the source, to all other points in the graph.

Suppose a student wants to go from home to school in the shortest possible way. She knows some roads are heavily congested and difficult to use. In Dijkstra's algorithm, this means the edge has a large weight--the shortest path tree found by the algorithm will try to avoid edges with larger weights. If the student looks up directions using a map service, it is likely they may use Dijkstra's algorithm, as well as others.

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



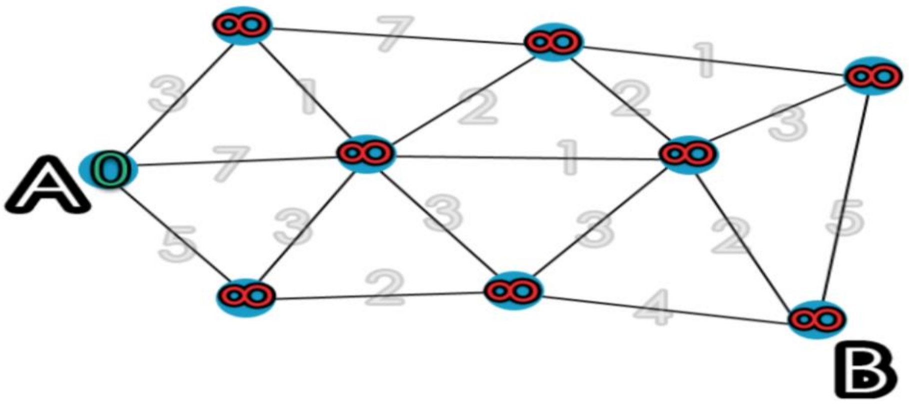
The shortest path, which could be found using Dijkstra's algorithm, is

Home→B→D→F→School

Algorithm:

Dijkstra’s algorithm finds a shortest path tree from a single source node, by building a set of nodes that have minimum distance from the source.

The graph has the following:



- Vertices, or nodes, denoted in the algorithm by vv or uu;
- Weighted edges that connect two nodes: (u,vu,v) denotes an edge, and w(u,v)w(u,v) denotes its weight. In the diagram on the right, the weight for each edge is written in gray.

This is done by initializing three values:

- *dist*, an array of distances from the source node ss to each node in the graph, initialized the following way: *dist*(s) = 0; and for all other nodes v, *dist*(v)=∞. This is done at the beginning because as the algorithm proceeds, the *dist* from the source to each node v in the graph will be recalculated and finalized when the shortest distance to v is found
- Q, a queue of all nodes in the graph. At the end of the algorithm's progress, Q will be empty.
- S, an empty set, to indicate which nodes the algorithm has visited. At the end of the algorithm's run, S will contain all the nodes of the graph.

The algorithm proceeds as follows:

1. While Q is not empty, pop the node v, that is not already in SS, from Q with the smallest *dist* (v). In the first run, source node ss will be chosen because *dist*(s) was initialized to 0. In the next run, the next node with the smallest *dist* value is chosen.
2. Add node v to S, to indicate that v has been visited
3. Update *dist* values of adjacent nodes of the current node v as follows: for each new adjacent node u,
4. if *dist* (v) + weight(u,v) <*dist* (u), there is a new minimal distance found for u, so update *dist*(u) to the new minimal distance value;
5. otherwise, no updates are made to *dist*(u).

The algorithm has visited all nodes in the graph and found the smallest distance to each node. *dist* now contains the shortest path tree from source s.

Note: The weight of an edge (u,v) is taken from the value associated with (u,v) on the graph.

Program:

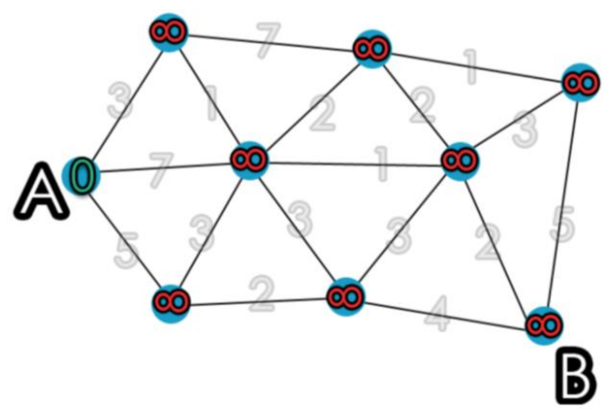
```
function Dijkstra(Graph, source):

dist[source]  := 0                    // Distance from source to source is set to 0
```

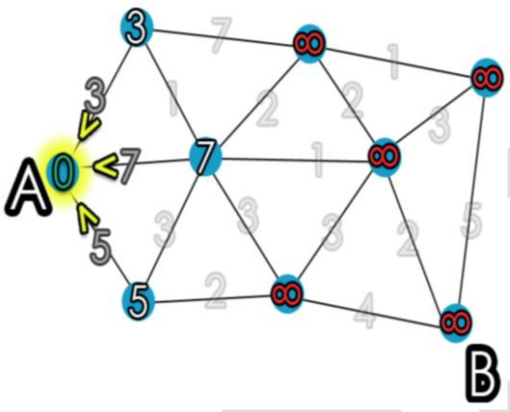
```
     for each vertex v in Graph:          // Initializations
         if v ≠ source
dist[v]  := infinity          // Unknown distance function from source to each node set
                                              // to infinity
         add v to Q                        // All nodes initially in Q
    while Q is not empty:           // The main loop
v := vertex in Q with min dist[v]  // In the first run-through, this vertex is the source node
        remove v from Q
        for each neighbor u of v:            // where neighbor u has not yet been removed from Q.
alt := dist[v] + length(v, u)
            if alt <dist[u]:              // A shorter path to u has been found
                    dist[u]  := alt           // Update distance of u
    return dist[]
  end function
```

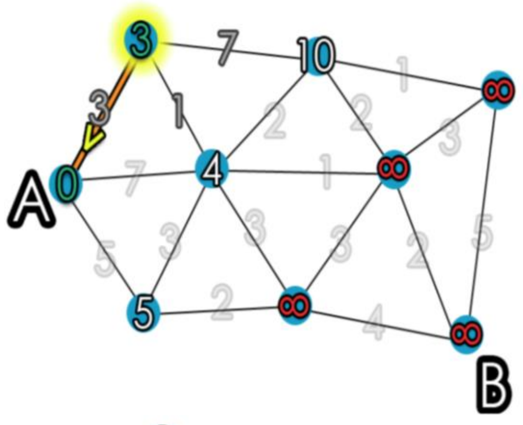
Example:

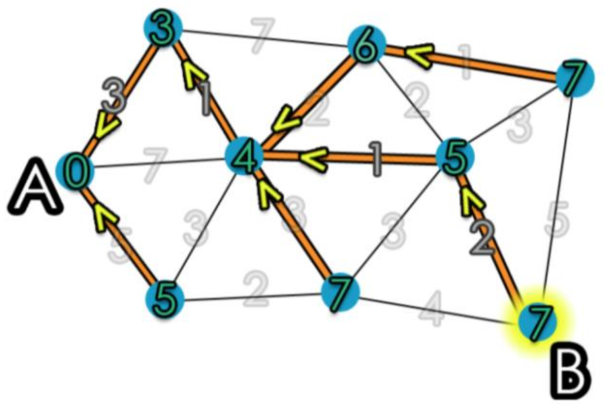
Let us solve above example using following steps through Dijkstra's algorithm:

2. Initialize distances according to the algorithm
3. Pick first node and calculate distances to adjacent nodes.
4. Pick next node with minimal distance; repeat adjacent node distance calculations.
5. Final result of shortest-path tree


A
B


A
B


A
B


A
B