ELEMENTARY GRAPH ALGORITHMS

DESIGN AND ANALYSIS OF ALGORITHMS (CSB 252)

SUBMITTED BY : SUMITRA SIVAKUMAR

ROLL NO. : 181210053

WHAT ARE GRAPHS?

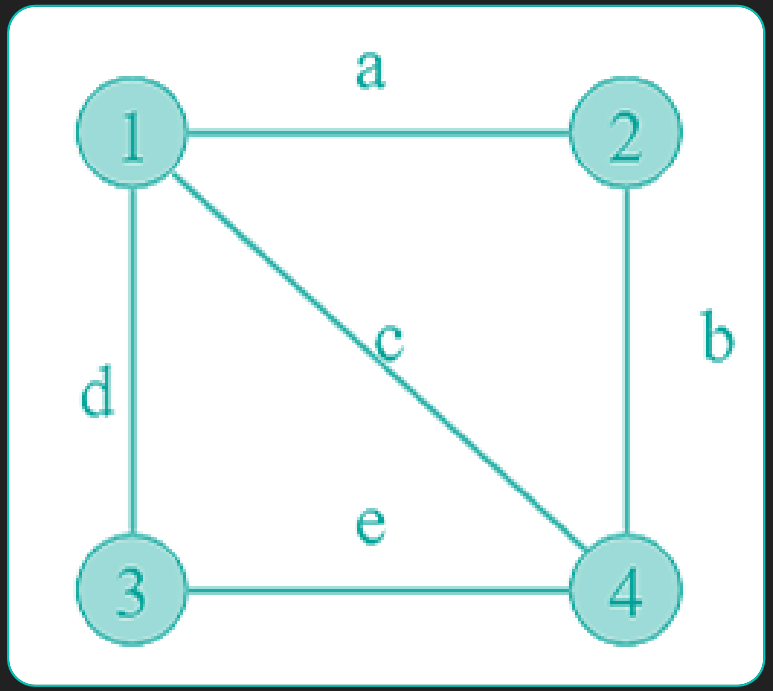
- A Graph is a **non-linear data structure** consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.

- More formally a Graph can be defined as,

A Graph consists of a finite set of vertices(or nodes) and set of edges which connect a pair of nodes.

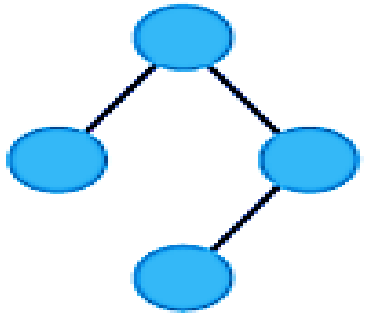
FEW IMPORTANT TERMS

- Simple Graph G = (V , E)
- **Vertex** − Each node of the graph is represented as a vertex.
- V(G) = {1 , 2 , 3 , 4}
- **Edge** represents a path between two vertices or a line between two vertices.
- E = E(G) = {a , b , c , d , e}
- **Adjacency** − Two node or vertices are adjacent if they are connected to each other through an edge.
- {(1,2) , (2,4) , (1,3) , (3,4) , (1,4)}
- **Path** − Path represents a sequence of edges between the two vertices.

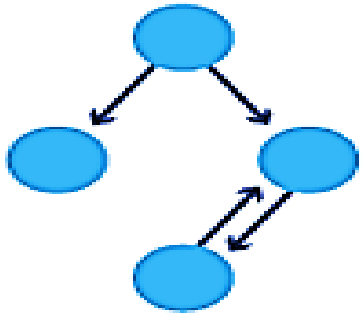


TYPES OF GRAPHS

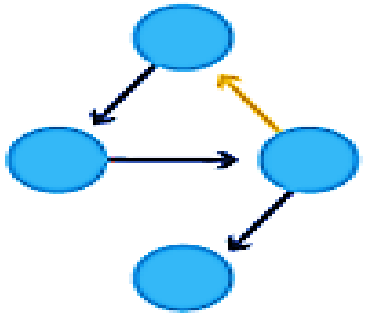
Undirected



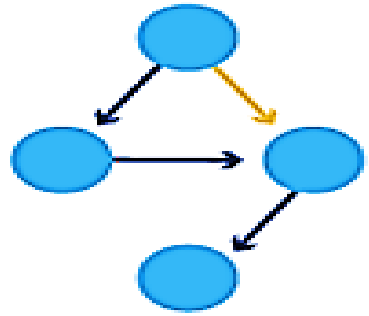
Directed



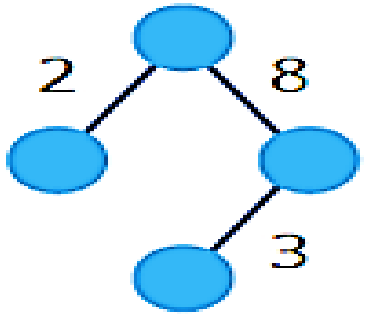
Cyclic



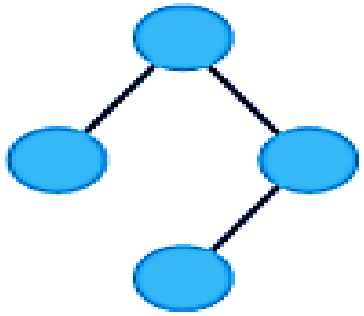
Acyclic



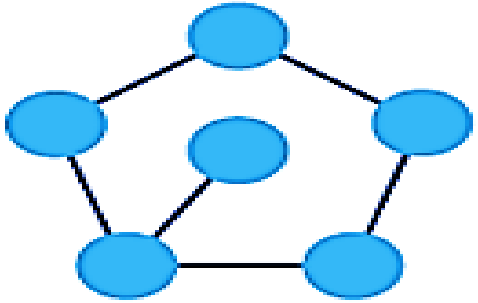
Weighted



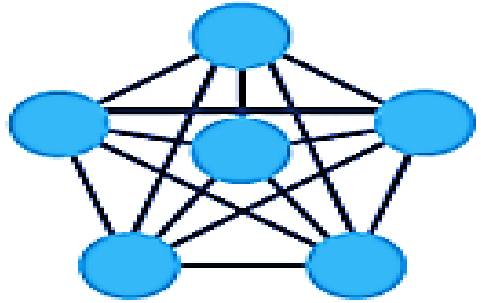
Unweighted



Sparse



Dense



BREADTH-FIRST SEARCH (BFS)

- **Breadth-first search** (**BFS**) is an algorithm for traversing or searching a graph data structure.

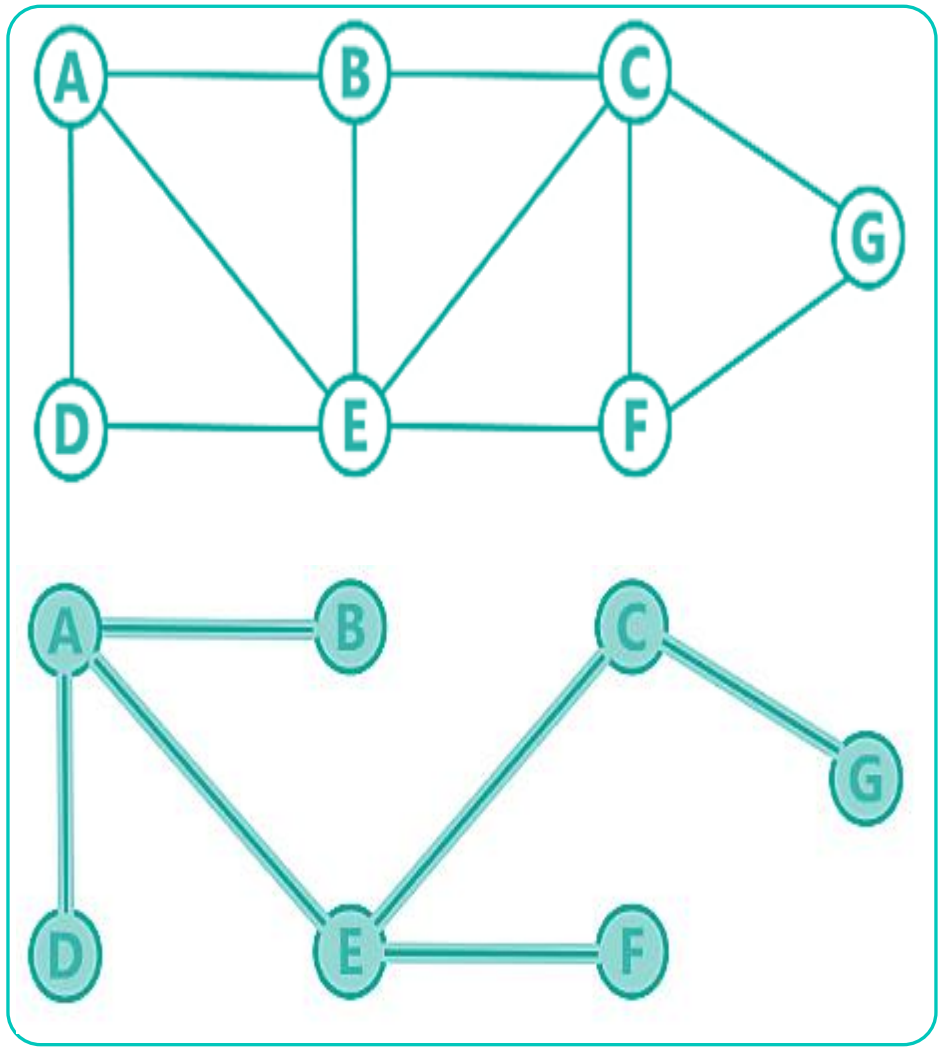
- It starts at the arbitrary node of a graph, sometimes referred to as a 'search key, and **explores all of the neighbour nodes** at the present depth prior to moving on to the nodes at the next depth level.

- Time Complexity of BFS Algorithm is O(|V|+|E|).

WORKING OF BFS

- **Step 1** - Define a Queue of size total number of vertices in the graph.
- **Step 2** - Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.
- **Step 3** - Visit all the non-visited adjacent vertices of the vertex which is at front of the Queue and insert them into the Queue.
- **Step 4** - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
- **Step 5** - Repeat steps 3 and 4 until queue becomes empty.
- **Step 6** - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

QUEUE -



PSEUDO CODE -BFS

```
BFS (G, s):
//here, G is the graph and s is the source or initial node
    Q.enqueue( s )
    while ( Q is not empty)
     //removing that vertex from queue whose neighbour will be visited now
        v  =  Q.dequeue( )
        //this will return the node inserted first in the queue
       //processing all the neighbours of v
       //if the node "w" is already visited, then ignore it
        //if the node "w" is not visited, then insert it into the queue
     for all neighbours w of v in Graph G
            if w is not visited
                    Q.enqueue( w )
                    //Stores w in Q to further visit its neighbour
                    mark w as visited.
```

DEPTH-FIRST SEARCH (DFS)

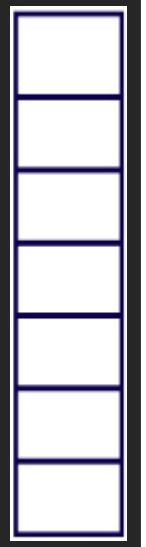
- **Depth-first search** (**DFS**) is an algorithm for traversing or searching a graph data structure.

- The algorithm starts at an arbitrary node as the root node in the case of a graph and **explores as far as possible along each branch** (depth-wise) before backtracking.

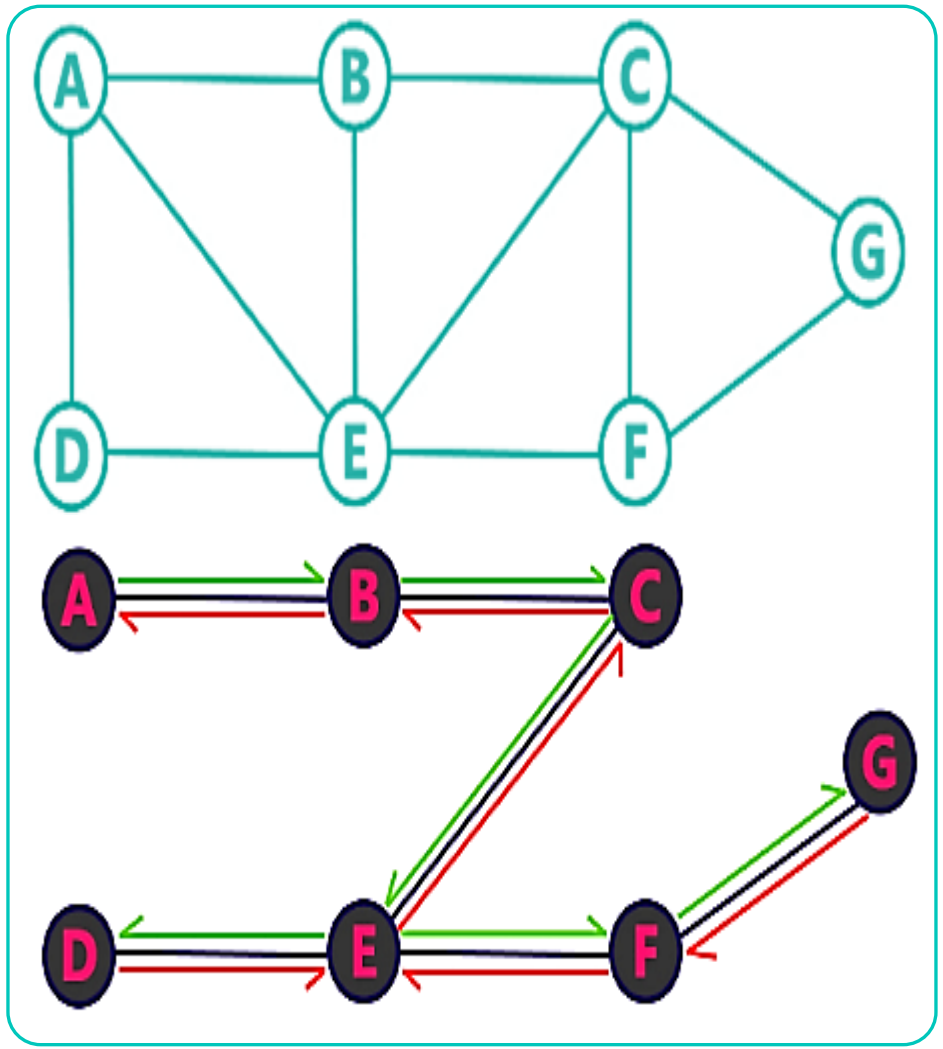
- Time Complexity of BFS Algorithm is O(|V|+|E|).

WORKING OF DFS

- **Step 1** - Define a Stack of size total number of vertices in the graph.
- **Step 2** - Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.
- **Step 3** - Visit any one of the non-visited adjacent vertices of a vertex which is at the top of stack and push it on to the stack.
- **Step 4 -** Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.
- **Step 5** - When there is no new vertex to visit then use back tracking and pop one vertex from the stack.
- **Step 6** - Repeat steps 3, 4 and 5 until stack becomes Empty.
- **Step 7** - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph



STACK

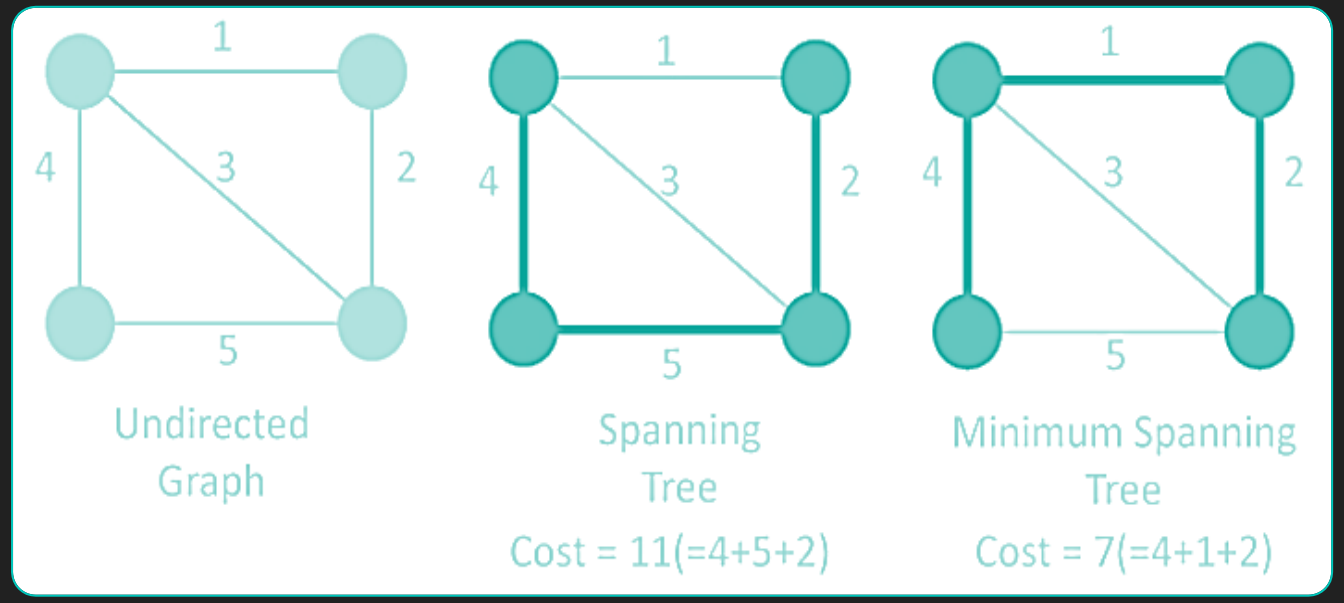


PSEUDO CODE -DFS

```
DFS (G, s):
//here, G is the Graph and s is the source/starting node
    let S be the stack
    S.push( s )
     //pushing source node in stack S
    mark s as visited.
    while ( S is not empty):
        //Pop a node from the stack S
        v  =  S.top( )
        S.pop( )
        //Push all the neighbours of v in stack that are not visited
       for all neighbours w of v in Graph G:
            if w is not visited :
                S.push( w )
            mark w as visited
```

MINIMUM SPANNING TREE

- A **spanning tree** is a sub-graph of an undirected and a connected graph, which includes all the vertices of the graph having a minimum possible number of edges.
- The total number of spanning trees with n vertices that can be created from a complete graph is equal to n^(n-2).
- A **minimum spanning tree** is a spanning tree in which the sum of the weight of the edges is as minimum as possible.
- The minimum spanning tree from a graph is found using the following algorithms:
  - Prim's Algorithm
  - Kruskal's Algorithm



KRUSKAL'S ALGORITHM

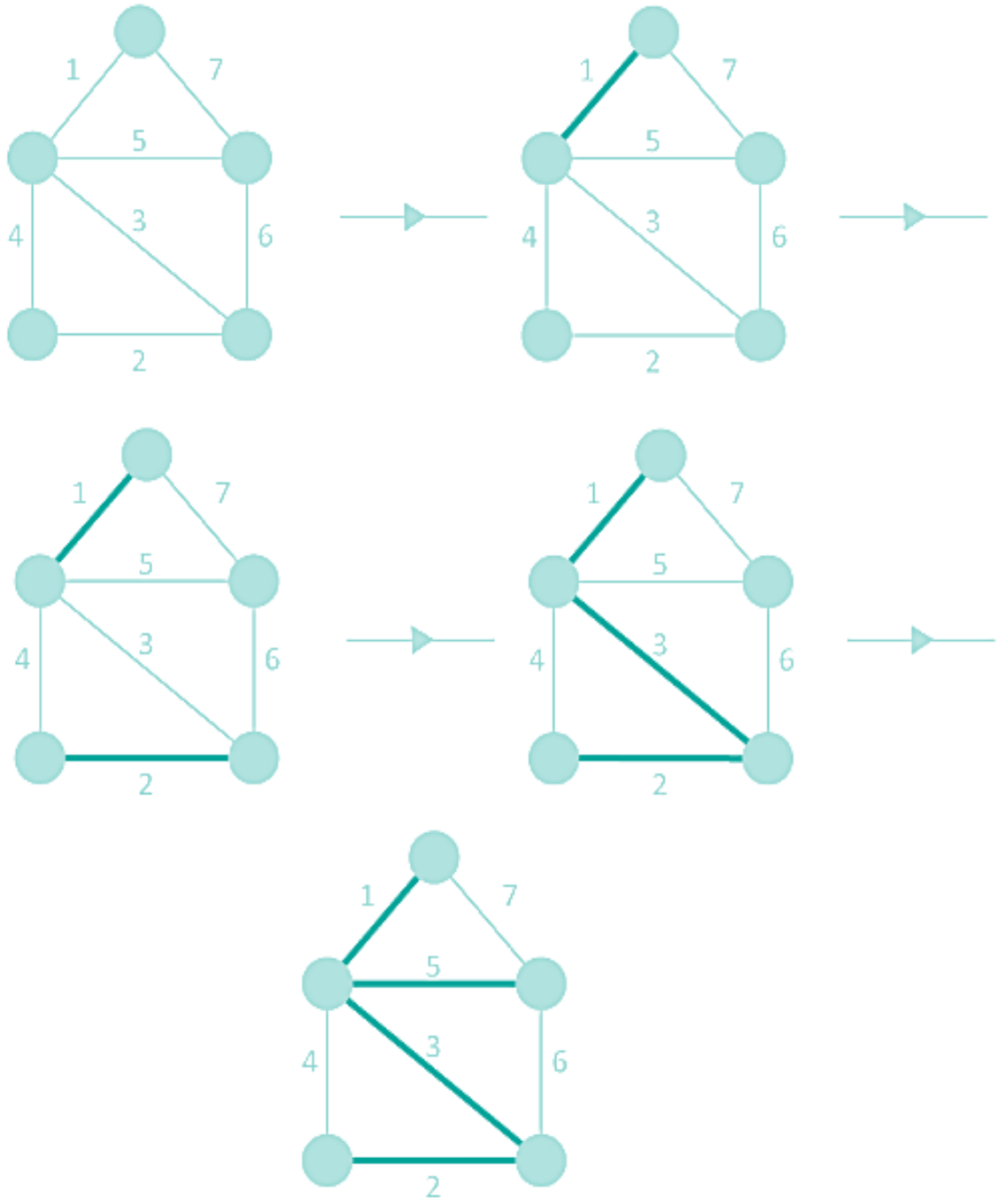
Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree.

It follows **greedy approach** as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

WOKRING OF KRUSKAL'S ALGORITHM

The steps for implementing Kruskal's algorithm are as follows:

- **Step 1 -** Sort all the edges from low weight to high.
- **Step 2 -** Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
- **Step 3 -** Keep adding edges until we reach all vertices.



PRIM'S ALGORITHM

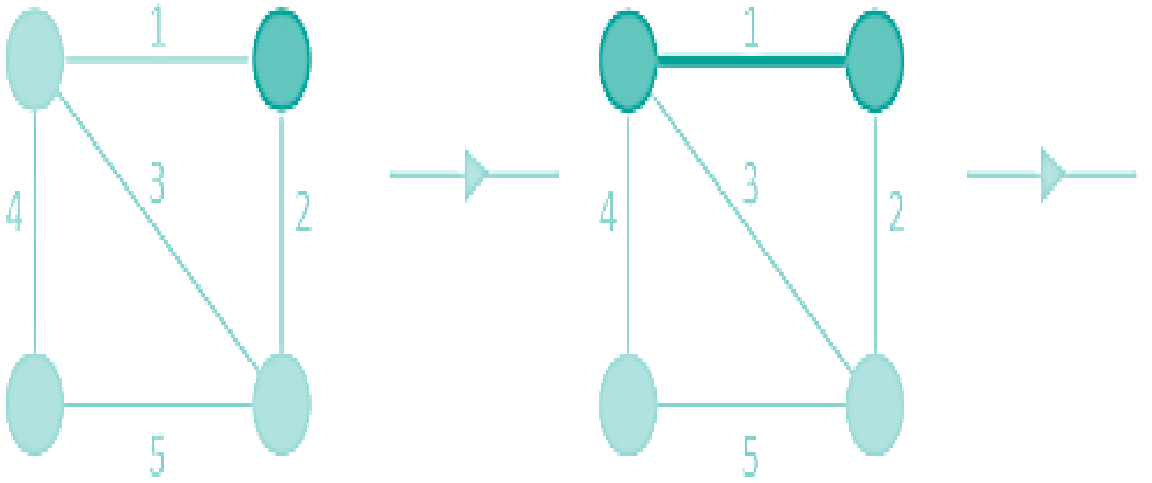
In Prim's Algorithm we grow the spanning tree from a starting position. Unlike an **edge** in Kruskal's, we add **vertex** to the growing spanning tree in Prim's.

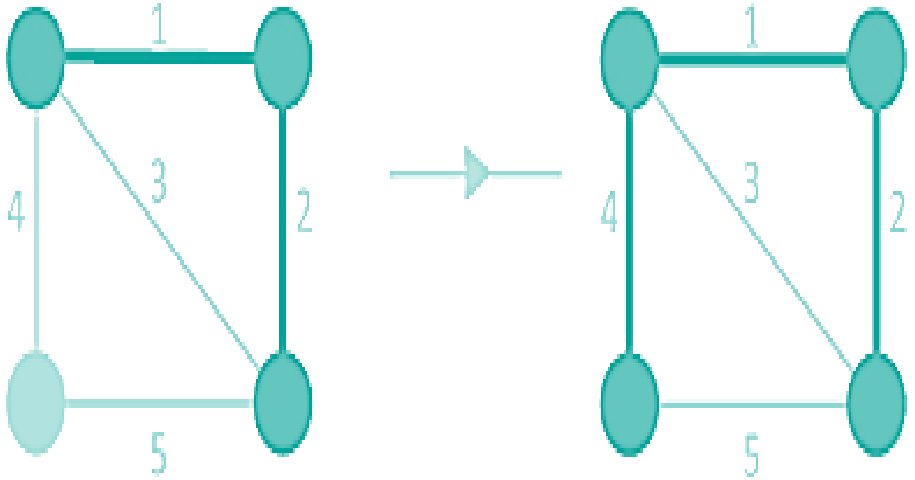
Prim's Algorithm also uses Greedy approach to find the minimum spanning tree.

WORKING OF PRIM'S ALGORITHM

The steps for implementing Prim's algorithm are as follows:

- **Step 1 -**Initialize the minimum spanning tree with a vertex chosen at random.
- **Step 2 -** Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree. If adding the vertices creates a cycle, then reject this edge.
- **Step 3 -** Keep repeating step 2 until we get a minimum spanning tree





DIJKSTRA'S ALGORITHM

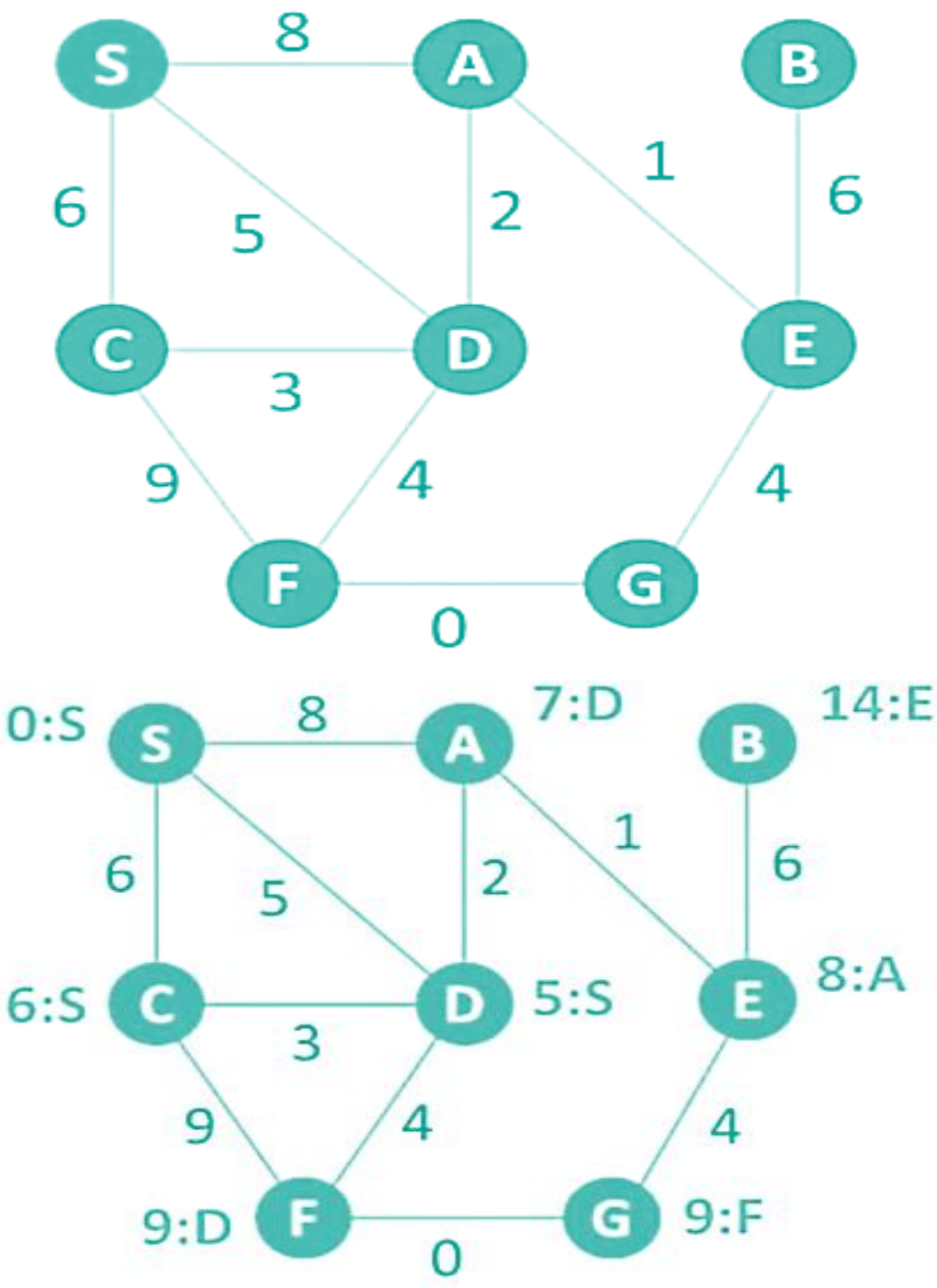
SINGLE-SOURCE SHORTEST PATH ALGORITHM

Dijkstra's algorithm (or **Dijkstra's Shortest Path First algorithm algorithm**) is an algorithm for finding the shortest paths between nodes in a graph.

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

WORKING OF DIJKSTRA'S ALGORITHM

- **Step 1 -** Set all vertices distances = infinity except for the source vertex, set the source distance = 0.
- **Step 2 -**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.
- **Step 3 -** Pop the vertex with the minimum distance from the priority queue (at first the popped vertex = source).
- **Step 4 -** 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.
- **Step 5 -** If the popped vertex is visited before, just continue without using it.
- **Step 6 -** Apply the same algorithm again until the priority queue is empty.



PSEUDO CODE – DIJKSTRA'S ALGORITHM

```
function Dijkstra(Graph, source):
    create vertex set Q
     for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0
    while Q is not empty:
        u ← vertex in Q with min dist[u]
        remove u from Q
        for each neighbor v of u:
        // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
            dist[v] ← alt
              prev[v] ← u
    return dist[], prev[]
```

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 is similar to Dijkstra's algorithm but it can work with graphs in which edges can have **negative weights**.

Time Complexity of Bellman Ford algorithm is relatively high **O(V·E)**

WORKING OF BELLMAN FORD'S ALGORITHM

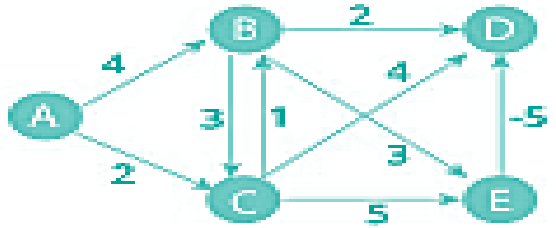
- **Step 1** - Initialize distance from the source to all vertices as infinite and distance to the source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.
- **Step 2 -** Calculate shortest distances. Do the following |V|-1 times where |V| is the number of vertices in given graph.

  a) Do following for each edge u-v
  If dist[v] > dist[u] + weight of edge uv, then update dist[v]
  dist[v] = dist[u] + weight of edge uv
- **Step 3 -** Report if there is a negative weight cycle in graph. Do following for each edge u-v

  If dist[v] > dist[u] + weight of edge uv, then "Graph contains negative weight cycle".

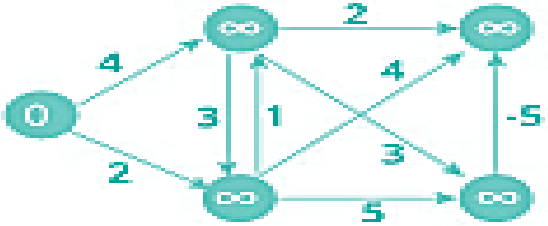
1

Start with a weighted graph



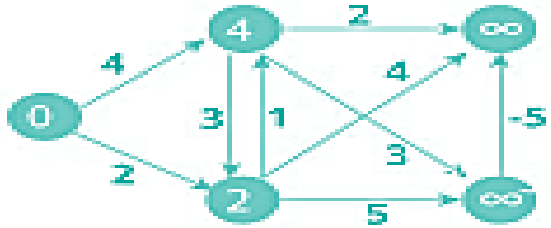
2

Choose a starting vertex and assign infinity path values to all other vertices



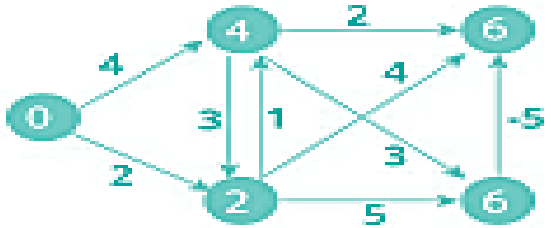
3

Visit each edge and relax the path distances if they are inaccurate



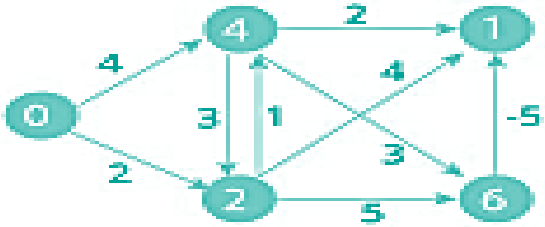
4

We need to do this V times because in the worst case, a vertex's path length might need to be readjusted V times



5

Notice how the vertex at the top right corner had its path length adjusted



6

After all the vertices have their path lengths, we check if a negative cycle is present.

A	B	C	D	E
0	∞	∞	∞	∞
0	4	2	∞	∞
0	3	2	6	6
0	3	2	1	6
0	3	2	1	6

PSEUDO CODE – BELLMAN FORD'S ALGORITHM

```
function BellmanFord(list vertices, list edges, vertex
source) is ::distance[], predecessor[]
    // Step 1: initialize graph
    for each vertex v in vertices do
        distance[v] := inf
        predecessor[v] := null
        distance[source] := 0
    // Step 2: relax edges repeatedly
    for i from 1 to size(vertices)−1 do
        for each edge (u, v) with weight w in
        edges do
            if distance[u] + w < distance[v]
            then
                distance[v] := distance[u] + w
                predecessor[v] := u
    // Step 3: check for negative-weight cycles
    for each edge (u, v) with weight w in edges do
        if distance[u] + w < distance[v] then
            error "Graph contains a negative-
            weight cycle"

    return distance[], predecessor[]
```

FLOYD-WARSHALL ALGORITHM

ALL-PAIR SHORTEST PATH ALGORITHM

Floyd-Warshall Algorithm is an algorithm for finding the shortest path between **all the pairs of vertices** in a weighted graph.

This algorithm works for both the directed and undirected weighted graphs.

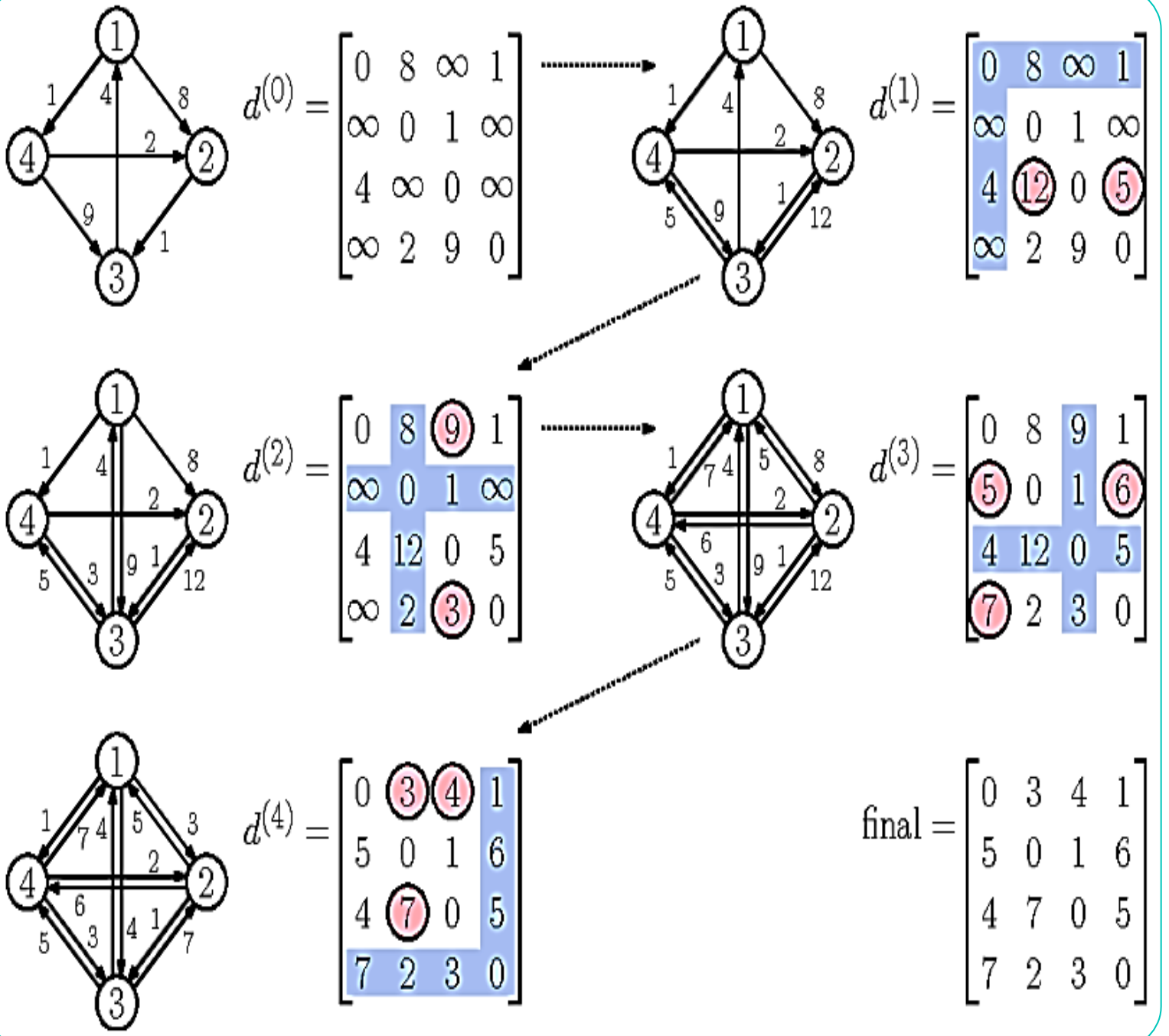
But, it does not work for the graphs with negative cycles.

Time Complexity of Floyd–Warshall's Algorithm is **O(V3),** where V is the number of vertices in a graph.

WORKING OF FLOYD-WARSHALL ALGORITHM

- **Step 1 -** Initialize the shortest paths between any 2 vertices with Infinity.
- **Step 2 -** 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 N vertices as intermediate nodes.
- **Step 3 -** Minimize the shortest paths between any 2 pairs in the previous operation.
- **Step 4 -** 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])



PSEUDO CODE – FLOYD WARSHALL ALGORITHM

```
for i = 1 to N
   for j = 1 to N
      if there is an edge from i to j
         dist[0][i][j] = the length of the edge from i to j
      else
         dist[0][i][j] = INFINITY

for k = 1 to N
   for i = 1 to N
      for j = 1 to N
         dist[k][i][j] = min(dist[k-1][i][j], dist[k-1][i][k]
         + dist[k-1][k][j])
```

THANK YOU