Chapter 10 Graphs

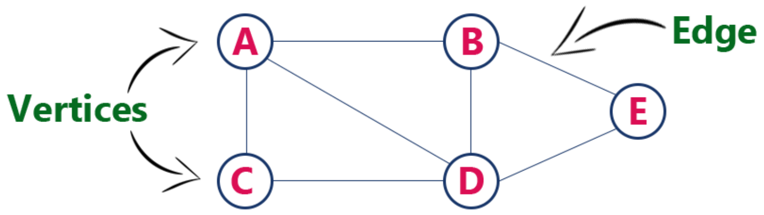
Representation and Application:

Graph is a non-linear data structure. It contains a set of points known as nodes (or vertices) and a set of links known as edges (or Arcs). Here edges are used to connect the vertices.

A graph G can be defined as an ordered set G (V, E) where V (G) represents the finite and non-empty set of vertices and E(G) represents the set of edges which are used to connect these vertices.

Example:

The following is a graph with 5 vertices and 6 edges.
$This\ graph\ G\ can\ be\ defined\ as\ G\ =\ (V, E)$
$Where\ V\ =\ \{A, B, C, D, E\}\ and\ E\ =\ \{(A, B), (A, C)(A, D), (B, D), (C, D), (B, E), (E, D)\}$.



Graph terminologies:

Vertex: An individual data element of a graph is called as Vertex. Vertex is also known as node. In above example graph, A, B, C, D & E are known as vertices.

Edge: An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (starting Vertex, ending Vertex).In above graph, the link between vertices A and B is represented as (A, B).

Adjacent Vertices: Two vertices **u** and **v** in an undirected graph **G** are called adjacent in **G** if **u** and **v** are endpoints of an edge **e** of **G**. Such an edge **e** is called incident with the vertices **u** and **v** and **e** is said to connect **u** and **v**.

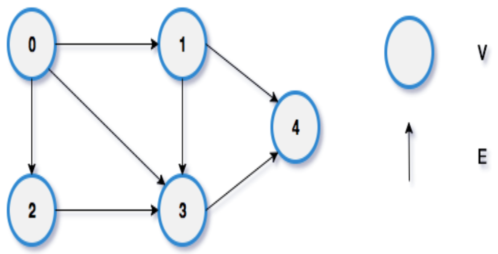
Degree of Vertex: The degree of a vertex in an undirected graph is the number of edges incident with it. A loop at a vertex contributes twice to the degree of that vertex. The degree of the vertex v is denoted by deg(v). A vertex of degree zero is called isolated. A vertex with degree one is called pendant. Example: deg (A) = 3, deg (B) =3 deg (E)=2.

In-degree: The in-degree of a vertex v is the number of edges that are incoming - towards v (head of edge).It is denoted by **deg– (u).**

Out-degree: the out-degree of a vertex v is the number of edges that are outgoing from v (tail of edge).It is denoted by **deg+ (u).**

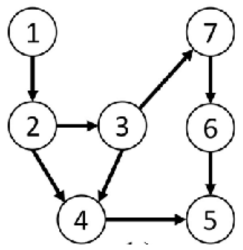
Chapter 10 Graphs

Node (u)	Deg-(u)	Deg+(u)
0	0	3
1	1	2
2	1	1
3	3	1
4	2	0

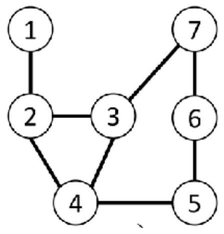


Types of Graph:

- **Directed Graph:**
  - In a directed graph, the connection between two nodes is one-directional.
  - It is a graph in which each edge has a direction to its successor.
  - It is a graph with only directed edges.



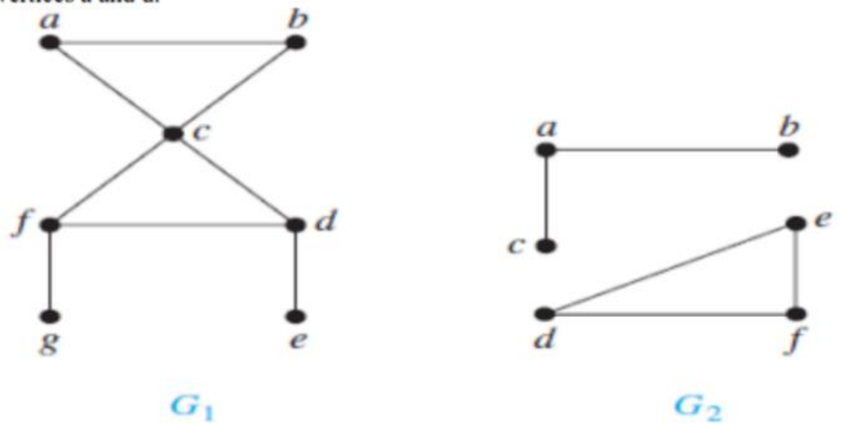
- **Undirected Graph:**
  - In an undirected graph, all connections are bi-directional.
  - It is a graph in which there is no direction on the edges. The flow between two vertices can go in either direction.



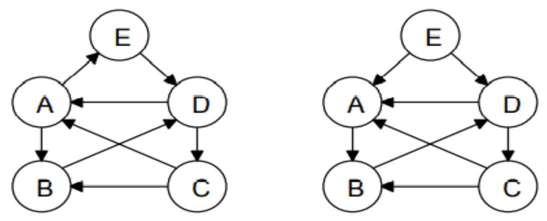
- **Connected Graph**: An undirected graph is called connected if there is a path between every pair of distinct vertices of the graph.

- **Not-Connected Graph**: An undirected graph that is not connected is called disconnected

Example: G1 is the connected graph because for every pair of distinct vertices there is a path between them and G2 is the not-connected graph because there is no path between vertices a and d.



- **Strongly Connected Graph**: A directed graph is strongly connected if there is a path from A to B and from B to A whenever A and B are vertices in the graph.
- **Weakly Connected Graph:** A directed graph is weakly connected if there is a path between every two vertices in the underlying undirected graph. That is, a directed graph is weakly connected if and only if there is always a path between two vertices when the directions of the edges are disregarded.

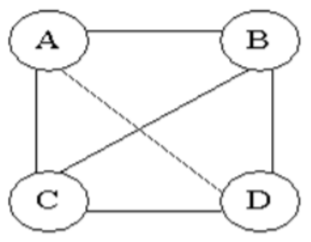


Strongly Connected

Weakly Connected

Complete Graph:

- A graph in which any V node is adjacent to all other nodes present in the graph is known as a complete graph. An undirected graph contains the edges that are equal to $edges = n(n-1)/2$ where n is the number of vertices present in the graph. The following figure shows a complete graph.



A complete graph.

Regular Graph:

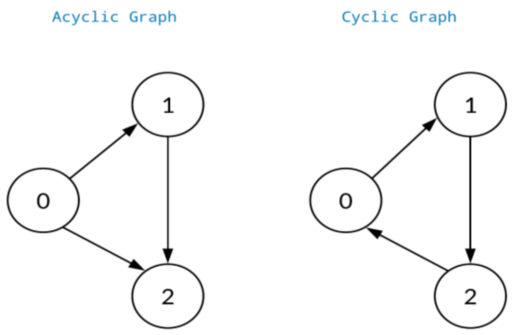
Chapter 10 Graphs

- It is the graph in which nodes are adjacent to each other, i.e., each node is accessible from any other node.

Cycle Graph:

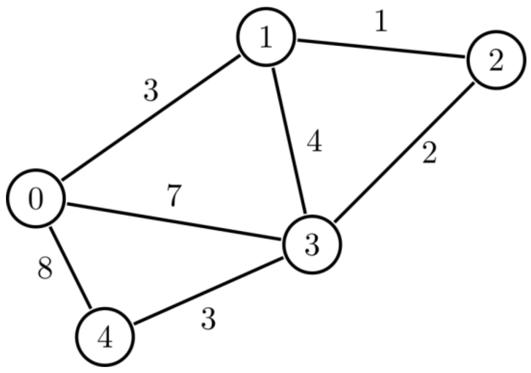
- A graph having cycle is called cycle graph. In this case the first and last nodes are the same. A closed simple path is a cycle.

Acyclic Graph: A graph without cycle is called acyclic graphs.



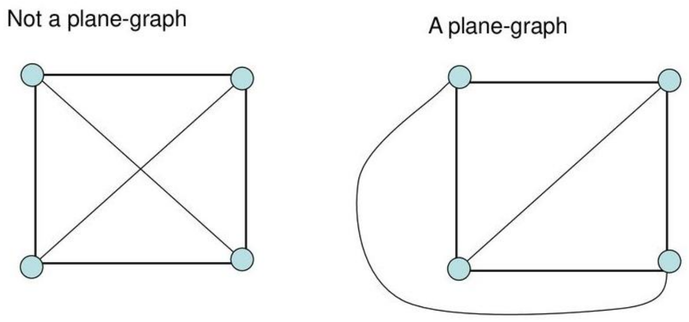
Weighted Graph:

- Graphs that have a number assigned to each edge are called weighted graphs.
- In a weighted graph, each edge has an associated numerical value, called the weight of the edge. Edge weights may represent distances, costs, etc.
- Example: In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



Planar Graph:

- A graph is called planar if it can be drawn in the plane without any edges crossing, i.e., it can be drawn on the plane in such a way that its edges intersect only at their endpoints. In other words, it can be drawn in such a way that no edges cross each other.

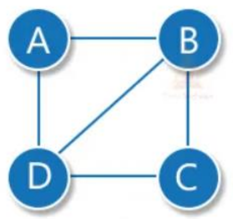


Representation of Graphs:

Chapter 10 Graphs

1. **Adjacency matrix :**

   - In it, we have a matrix of order n*n where n is the number of nodes in the graph. The matrix represents the mapping between various edges and vertices.
   - In the matrix, each row and column represents a vertex. The values determine the presence of edges.
   - Let Aij represents each element of the adjacency matrix. Then,
   - For an undirected graph, the value of Aij is 1 if there exists an edge between i and j. Otherwise, the value of Aij is 0.

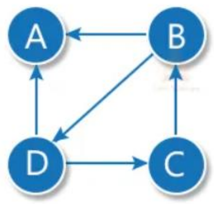


Undirected graph

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 0 & 1 \\ B & 1 & 0 & 1 & 1 \\ C & 0 & 1 & 0 & 1 \\ D & 1 & 1 & 1 & 0 \end{array}$$

Adjacency matrix

   - For a directed graph, the value of Aij is 1 only if there is an edge from i to j i.e. **i** is the initial node and **j** is the terminal node.



Directed graph

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 0 & 0 & 0 \\ B & 1 & 0 & 0 & 1 \\ C & 0 & 1 & 0 & 0 \\ D & 1 & 0 & 1 & 0 \end{array}$$

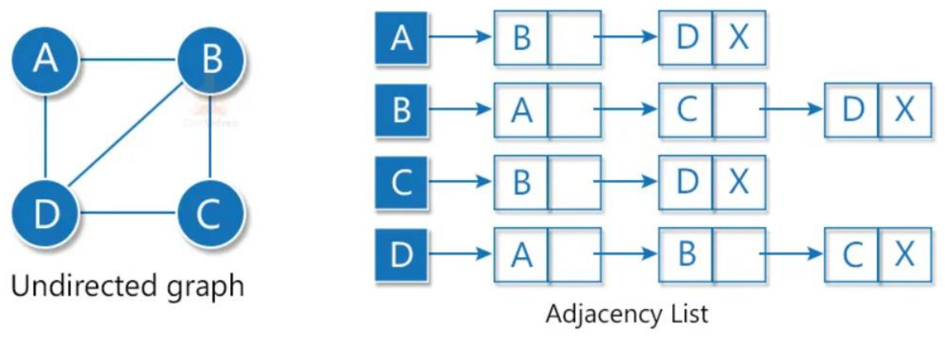
Adjacency matrix

   - The time complexity of the adjacency matrix is O(n^2).

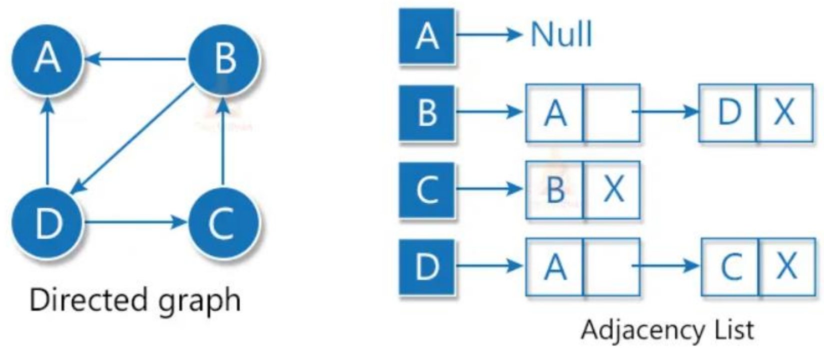
2. **Adjacency list:**

- The adjacency list is an array of linked lists where the array denotes the total vertices and each linked list denotes the vertices connected to a particular node.
- In a linked list, the most important component is the pointer named 'Head' because this single pointer maintains the whole linked list. For linked list representation, we will have total pointers equal to the number of nodes in the graph.
- For an undirected graph, we will link all the edges in the list that are connected to a node as shown:

Chapter 10 Graphs



- In a directed graph, we will link only the initial nodes in the list as shown:



Applications of Graph

- In Computer science graphs are used to represent the flow of computation.
- Google maps uses graphs for building transportation systems, where intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices.
- Computer networks: Local area network, Internet, Web
- Facebook’s Friend suggestion algorithm uses graph theory. Facebook is an example of undirected graph.
- In World Wide Web, web pages are considered to be the vertices. There is an edge from a page u to other page v if there is a link of page v on page u. This is an example of Directed graph. It was the basic idea behind Google Page Ranking Algorithm.

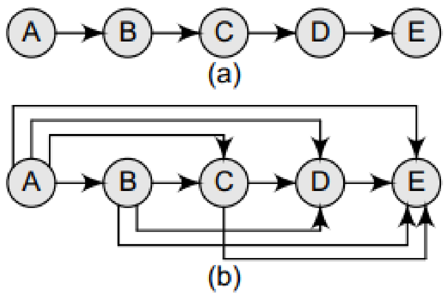
Chapter 10 Graphs

Transitive closure:

- Transitive Closure it the reachability matrix to reach from vertex u to vertex v of a graph. One graph is given, we have to find a vertex v which is reachable from another vertex u, for all vertex pairs (u, v).
- It states if there is a path from vertex a to b then there should be an edge from a to b.
- For finding transitive closure of a graph
- Add an edge from a to c if there exists a path from a to b and b to c
- Repeat this process of adding edge until no new edges are added.
- Hence, it can be defined as, If G=(V,E) in a graph then its transitive closure can be defined as G*=(V,E*) where E*={(Vi,Vj): there exists a path from Vi to Vj in G}

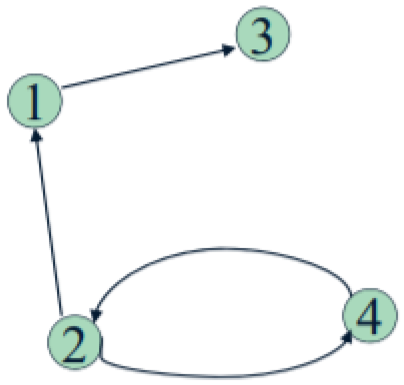
Transitive Closure of a Directed Graph

- Given a directed graph, find out if a vertex j is reachable from another vertex i for all vertex pairs (i, j) in the given graph. Here reachable mean that there is a path from vertex i to j. The reach-ability matrix is called the transitive closure of a graph.
- Transitive closure is also stored as a matrix T, so if T[1][5] = 1, then node 5 can be reached from node 1 in one or more hops.



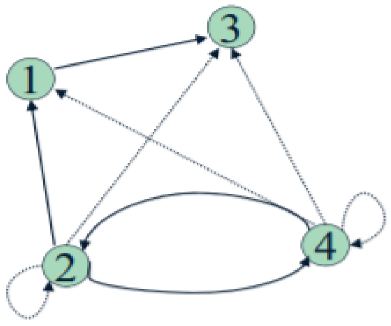
(a) A graph G and its (b) transitive closure G*

- For example, consider below graph



V	**1**	**2**	**3**	**4**
1	0	0	1	0
2	1	0	0	1
3	0	0	0	0
4	0	1	0	0

- Transitive closure of above graphs is



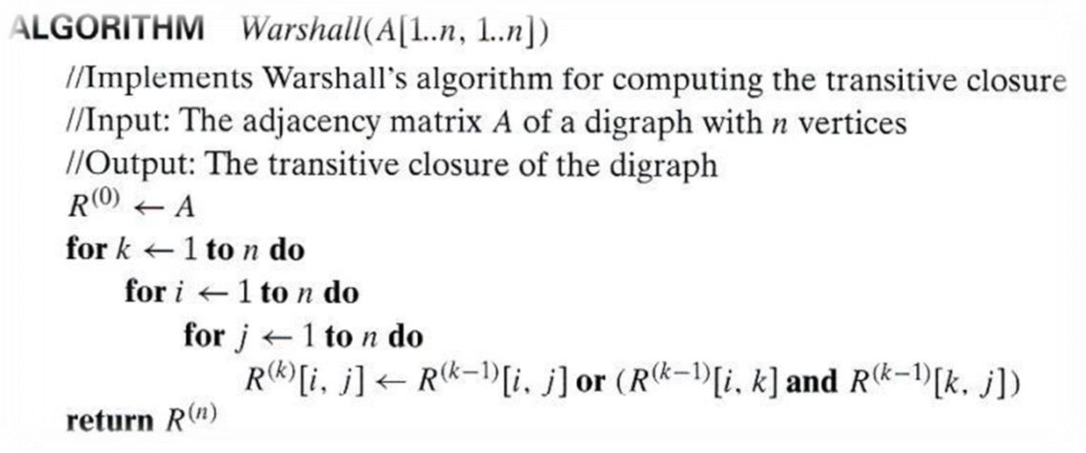
V	1	2	3	4
1	0	0	1	0
2	1	1	1	1
3	0	0	0	0
4	1	1	1	1

Warshall's algorithm for finding transitive closure from diagraph:

- Recurrence relating elements $R^{(k)}$ to elements of $R^{(k-1)}$ is:

$$R^{(k)}[i,j] = R^{(k-1)}[i,j] \text{ or } (R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j])$$

- It implies the following rules for generating $R^{(k)}$ from $R^{(k-1)}$:
  Rule 1 If an element in row i and column j is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$
  Rule 2 If an element in row i and column j is 0 in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ if and only if the element in its row i and column k and the element in its column j and row k are both 1's in $R^{(k-1)}$.

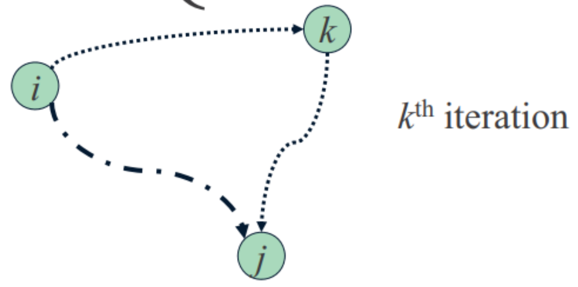


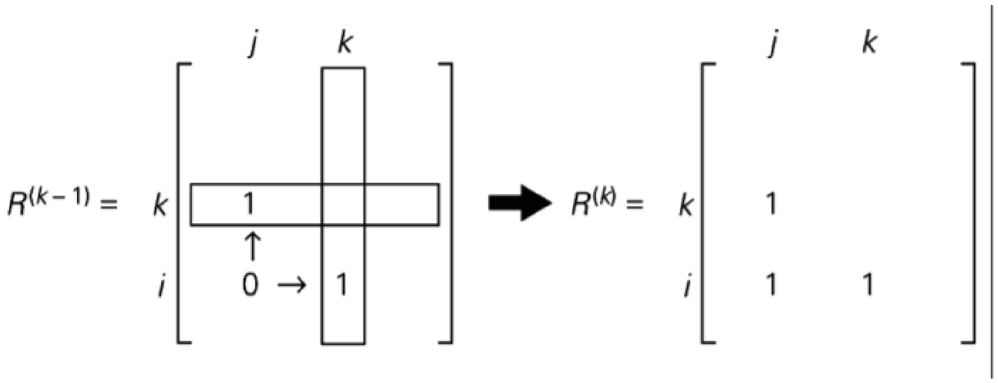
```
ALGORITHM   Warshall(A[1..n, 1..n])
    //Implements Warshall's algorithm for computing the transitive closure
    //Input: The adjacency matrix A of a digraph with n vertices
    //Output: The transitive closure of the digraph
    $R^{(0)} \leftarrow A$
    for $k \leftarrow 1$ to $n$ do
        for $i \leftarrow 1$ to $n$ do
            for $j \leftarrow 1$ to $n$ do
                $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ or $(R^{(k-1)}[i, k]$ and $R^{(k-1)}[k, j])$
    return $R^{(n)}$
```

The main idea of these graphs is described as follows:

- The vertices i, j will be contained a path if
- The graph contains an edge from i to j; or
- The graph contains a path from i to j with the help of vertex 1; or
- The graph contains a path from i to j with the help of vertex 1 and/or vertex 2; or
- The graph contains a path from i to j with the help of any other vertices.

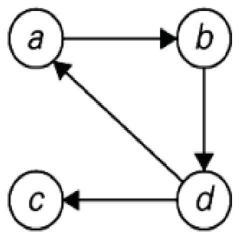
$$R^{(k)}[i,j] = \begin{cases} R^{(k-1)}[i,j] & \text{(path using just } 1,\ldots,k\text{-}1) \\ \text{or} & \\ (R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j]) & \text{(path from } i \text{ to } k \text{ and from } k \text{ to } i \text{ using just } 1,\ldots,k\text{-}1) \end{cases}$$





Example 1:

Fig: Rule for changing zero's in Warshall's Algorithm



(a)

$$A = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array}$$

(b)

$$T = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 1 & 1 & 1 & 1 \\ b & 1 & 1 & 1 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

(c)

Figure. (a) Digraph. (b) Its adjacency matrix. (c) Its transitive closure.

Chapter 10 Graphs

$$\begin{tikzcd} a \arrow[r] & b \arrow[d] \\ c & d \arrow[l] \arrow[ul] \end{tikzcd}$$

$$R^{(0)} = \begin{array}{c|cccc|} & a & b & c & d \\ a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array}$$

Ones reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

$$R^{(1)} = \begin{array}{c|cccc|} & a & b & c & d \\ a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & \mathbf{1} & 1 & 0 \end{array}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex *a* (note a new path from *d* to *b*); boxed row and column are used for getting $R^{(2)}$.

$$R^{(2)} = \begin{array}{c|cccc|} & a & b & c & d \\ a & 0 & 1 & 0 & \mathbf{1} \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & \mathbf{1} \end{array}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., *a* and *b* (note two new paths); boxed row and column are used for getting $R^{(3)}$.

$$R^{(3)} = \begin{array}{c|cccc|} & a & b & c & d \\ a & 0 & 1 & 0 & 1 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., *a, b,* and *c* (no new paths); boxed row and column are used for getting $R^{(4)}$.

$$R^{(4)} = \begin{array}{c|cccc|} & a & b & c & d \\ a & \mathbf{1} & 1 & \mathbf{1} & 1 \\ b & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., *a, b, c,* and *d* (note five new paths).

Example 2:

Apply Warshall's algorithm to find the transitive closure of the digraph defined by the following adjacency matrix.

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Applying Warshall's algorithm yields the following sequence of matrices

$$R^{(0)} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$R^{(1)} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$R^{(2)} = \begin{bmatrix} 0 & 1 & \mathbf{1} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

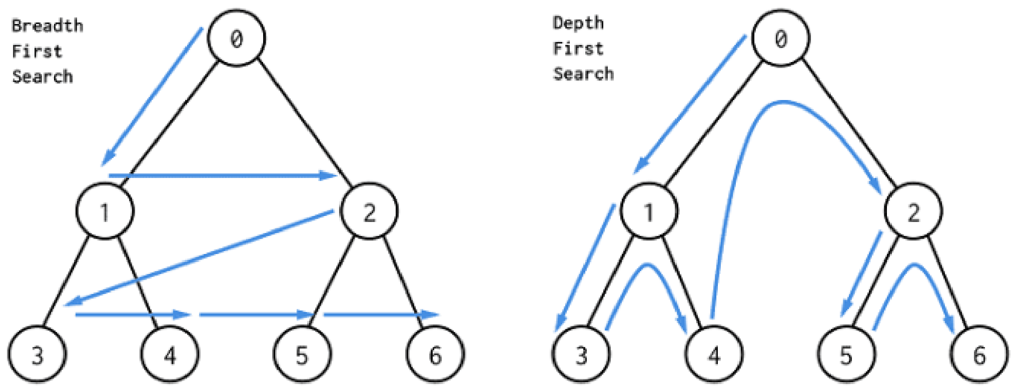
$$R^{(3)} = \begin{bmatrix} 0 & 1 & 1 & \mathbf{1} \\ 0 & 0 & 1 & \mathbf{1} \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$R^{(4)} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} = T$$

Chapter 10 Graphs

Traversing a Graph:

- Graph traversal is a technique used for searching a vertex in a graph.
- The graph traversal is also used to decide the order of vertices is visited in the search process.
- A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.
- There are two graph traversal techniques and they are as follows:
    1. DFS (Depth First Search)
    2. BFS (Breadth First Search)



DFS (Depth First Search):

DFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without loops. We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal. We use the following steps to implement DFS traversal...

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.

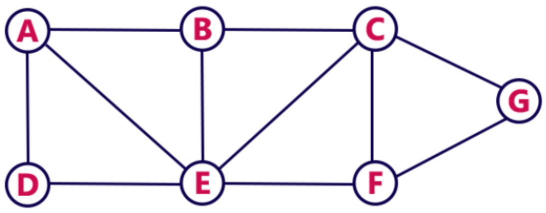
Step 5 - When there is no new vertex to visit then use back tracking and pop one vertex from the stack.

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

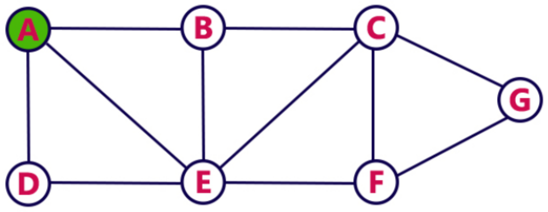
Example 1:

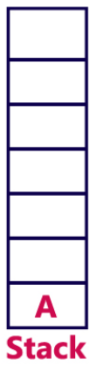
Consider the following example graph to perform DFS traversal



Step 1:

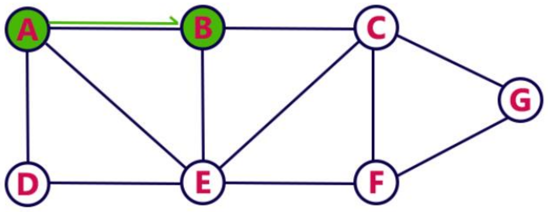
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.

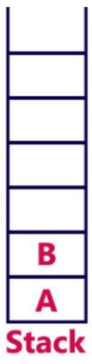




Step 2:

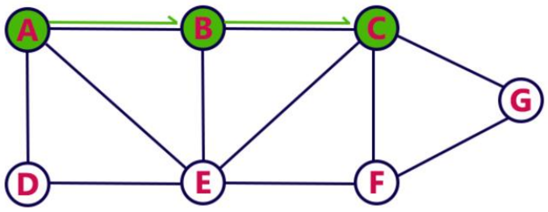
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

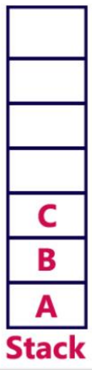




Step 3:

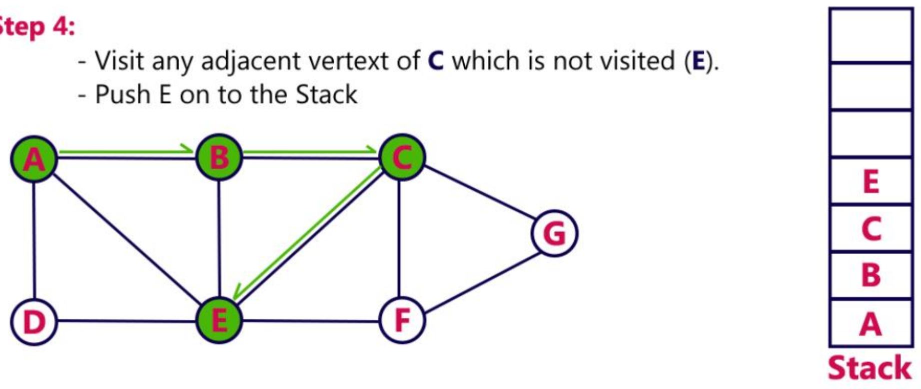
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.





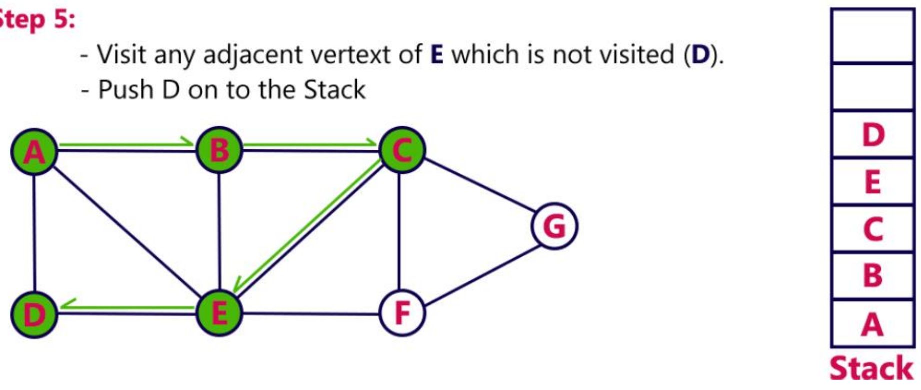
Step 4:

- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack



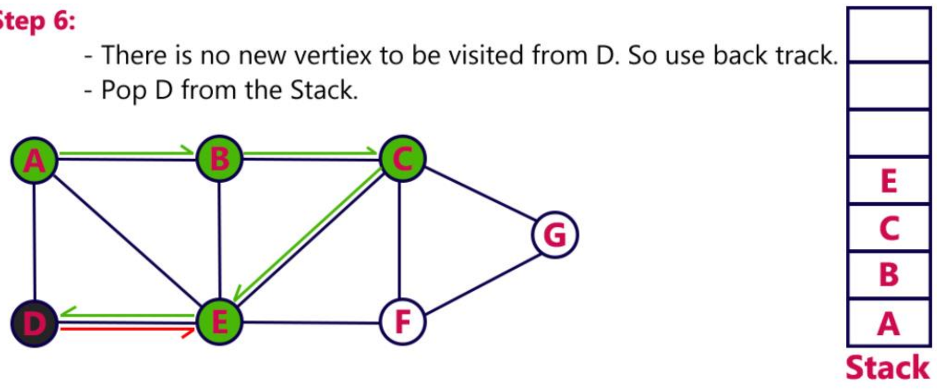
Step 5:

- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack



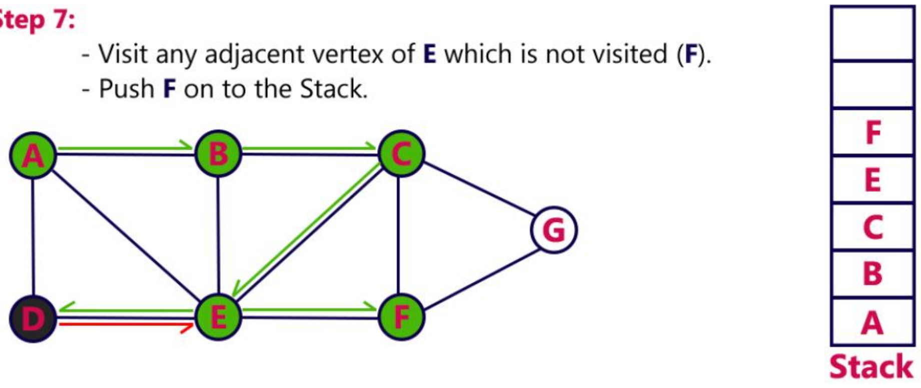
Step 6:

- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.



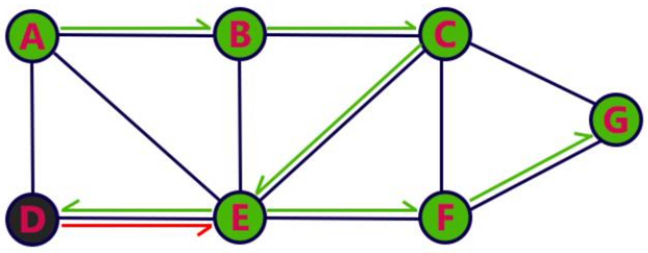
Step 7:

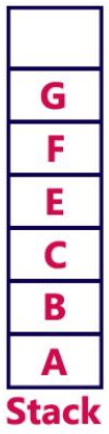
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.



Step 8:

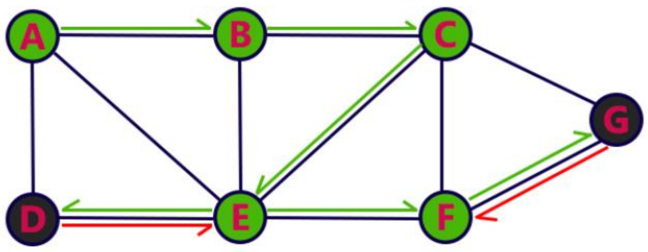
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

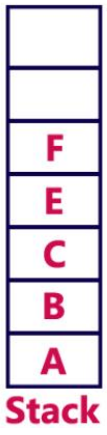




Step 9:

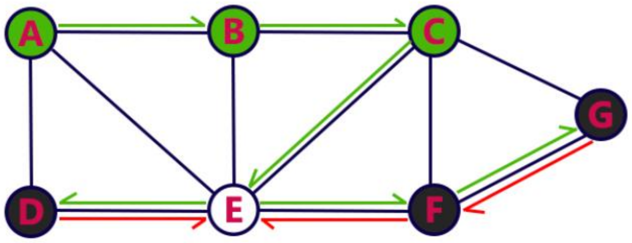
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

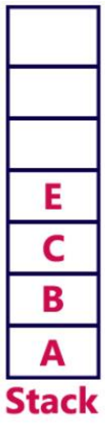




Step 10:

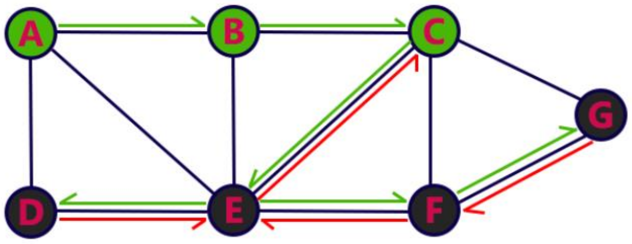
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

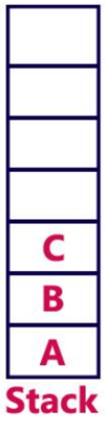




Step 11:

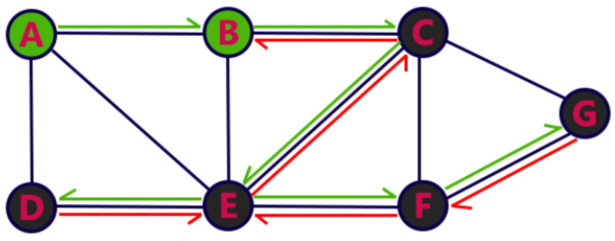
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

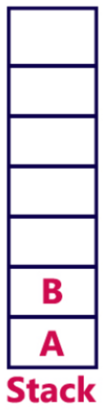




Step 12:

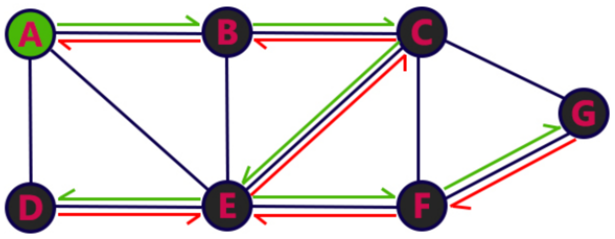
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

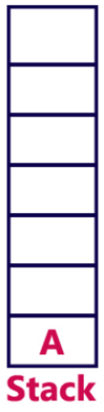




Step 13:

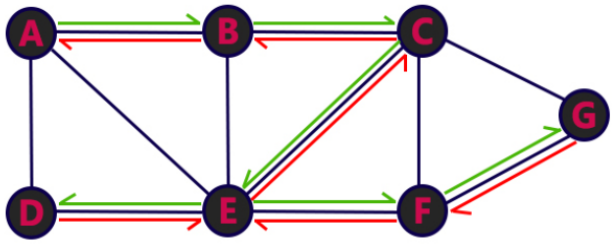
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

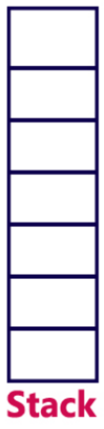




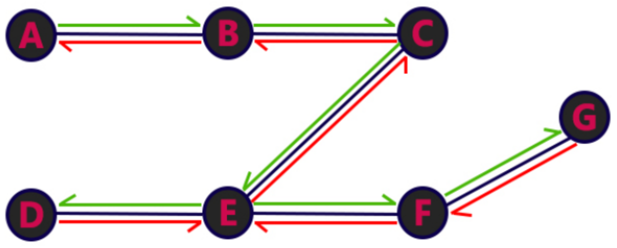
Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.

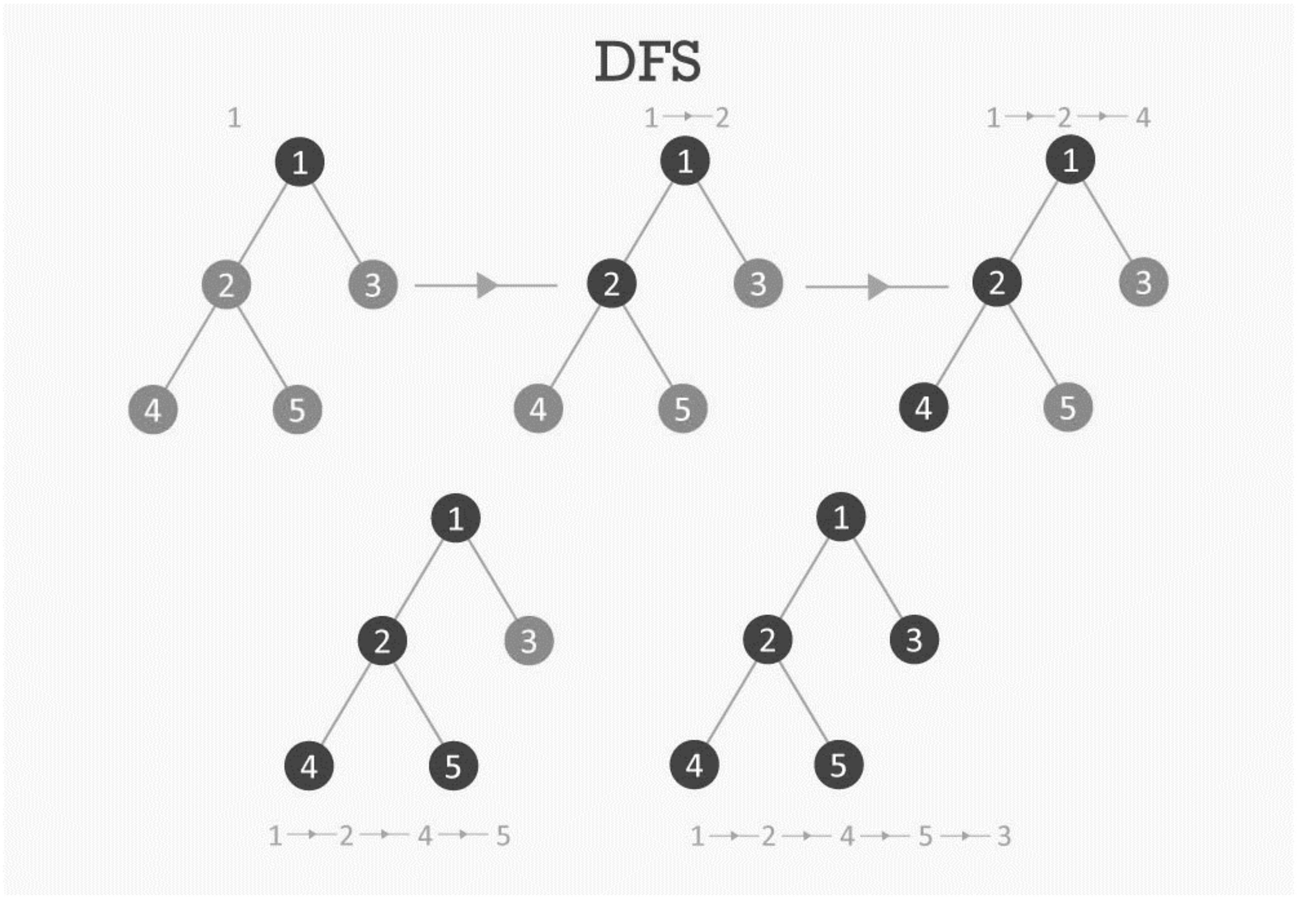




- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



Example 2:



Chapter 10 Graphs

BFS (Breadth First Search)

BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.

We use the following steps to implement BFS traversal...

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.

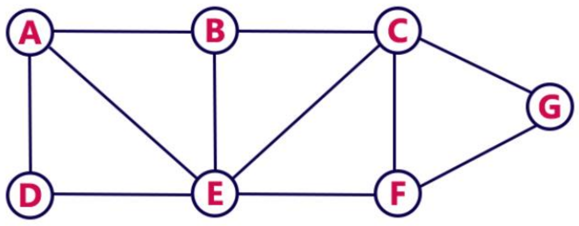
Step 4 - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.

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

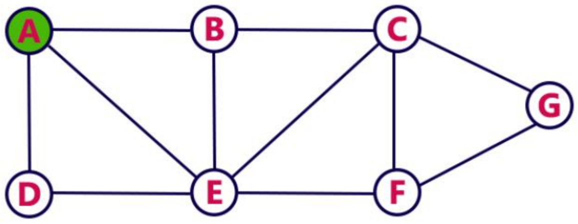
Example 1:

Consider the following example graph to perform BFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

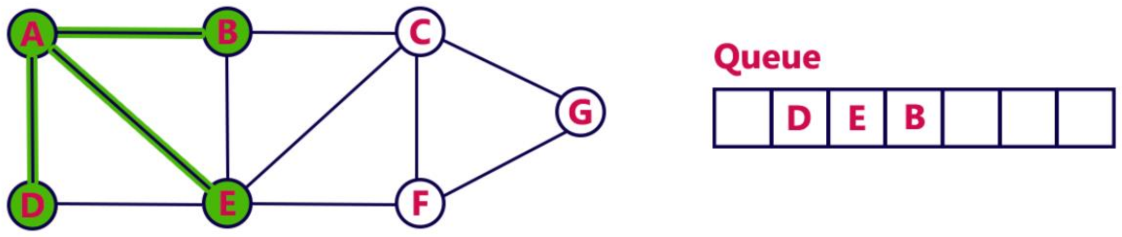


Queue

A						

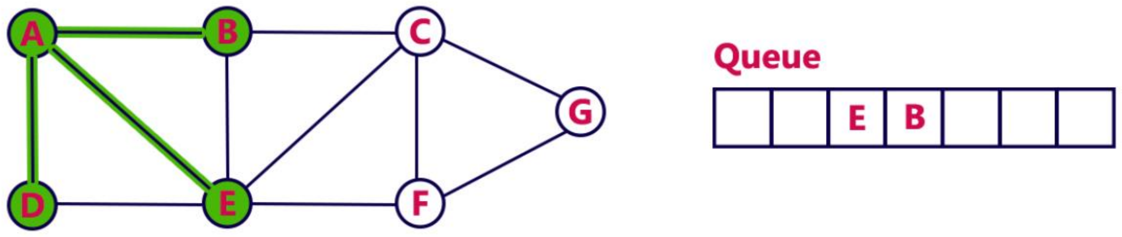
Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..



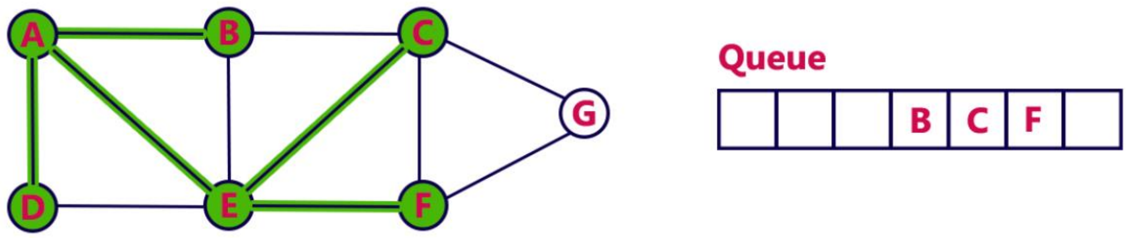
Step 3:

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



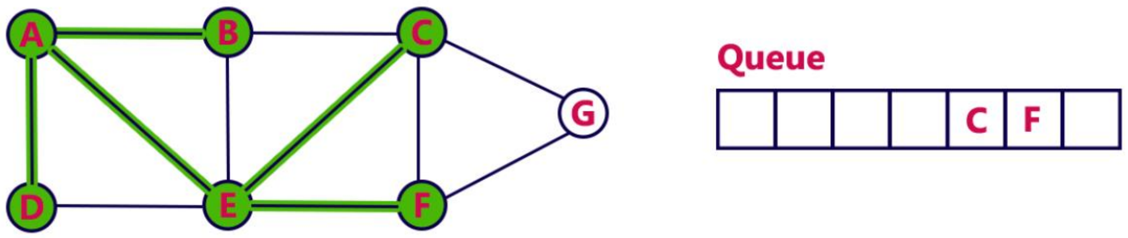
Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.



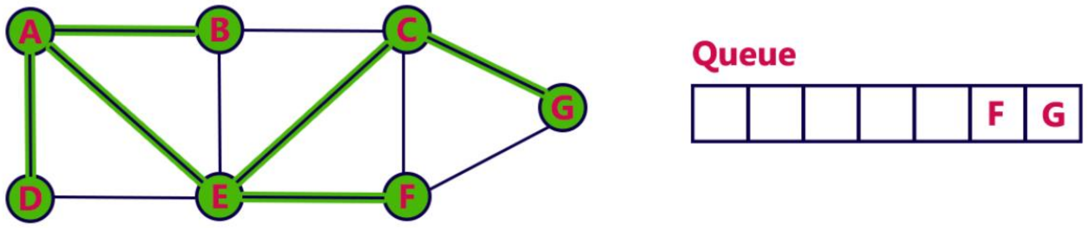
Step 5:

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



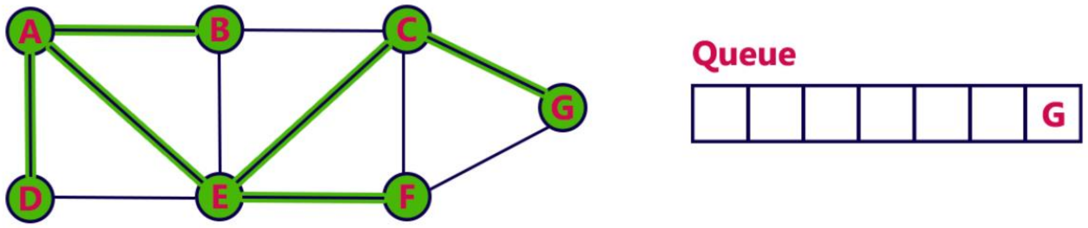
Step 6:

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



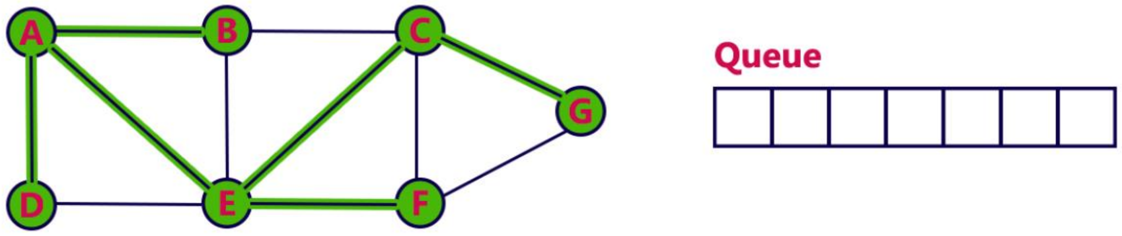
Step 7:

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

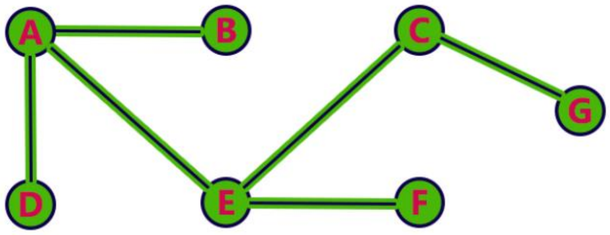


Step 8:

- 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...



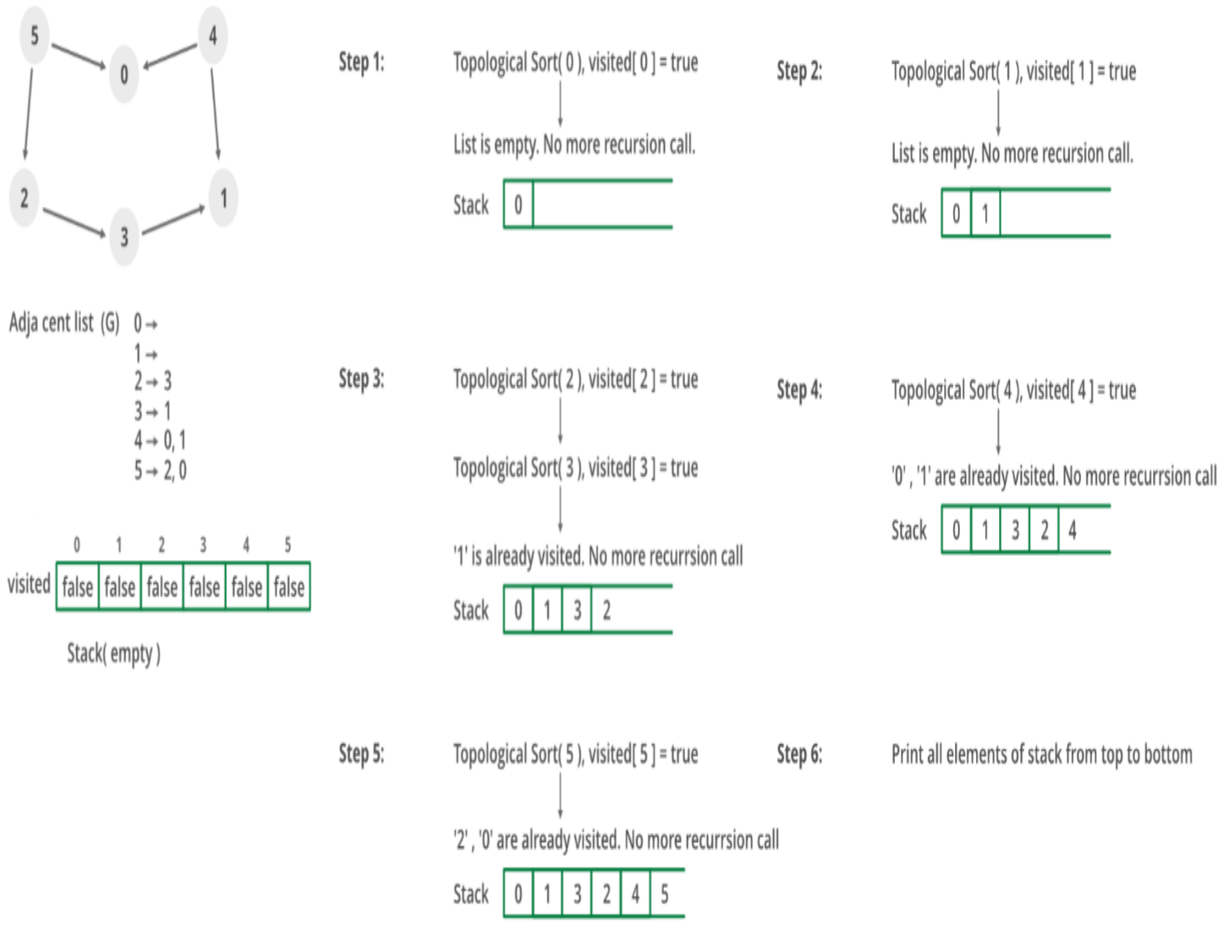
Chapter 10 Graphs

Topological Sort

- ***Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u v, vertex u comes before v in the ordering***. Topological sorting for a graph is not possible if the graph is not a DAG.

Topological Sort using Depth First Search (DFS)

- Use temporary stack to store the vertex.
- Maintain a visited [] to keep track of already visited vertices.
- In DFS we print the vertex and make recursive call to the adjacent vertices but here we will make the recursive call to the adjacent vertices and then push the vertex to stack.
- Observe closely the previous step, it will ensure that vertex will be pushed to stack only when all of its adjacent vertices (descendants) are pushed into stack.
- Finally print the stack.
- For disconnected graph, Iterate through all the vertices, during iteration, at a time consider each vertex as source (if not already visited).



Adja cent list (G) 0 →
1 →
2 → 3
3 → 1
4 → 0, 1
5 → 2, 0

	0	1	2	3	4	5
visited	false	false	false	false	false	false

Stack(empty)

Step 1: Topological Sort(0), visited[0] = true
List is empty. No more recursion call.
Stack | 0 |

Step 2: Topological Sort(1), visited[1] = true
List is empty. No more recursion call.
Stack | 0 | 1 |

Step 3: Topological Sort(2), visited[2] = true
Topological Sort(3), visited[3] = true
'1' is already visited. No more recurrsion call
Stack | 0 | 1 | 3 | 2 |

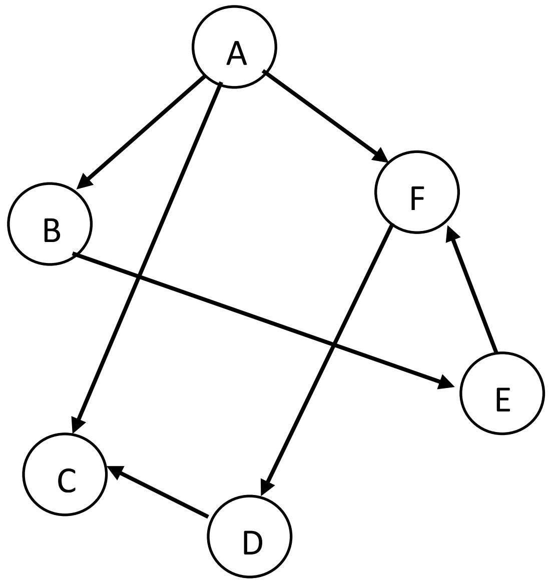
Step 4: Topological Sort(4), visited[4] = true
'0' , '1' are already visited. No more recurrsion call
Stack | 0 | 1 | 3 | 2 | 4 |

Step 5: Topological Sort(5), visited[5] = true
'2' , '0' are already visited. No more recurrsion call
Stack | 0 | 1 | 3 | 2 | 4 | 5 |

Step 6: Print all elements of stack from top to bottom

- For example, a topological sorting of the following graph is “5 4 2 3 1 0”.
- There can be more than one topological sorting for a graph.

- For example, another topological sorting of the following graph is "4 5 2 3 1 0".

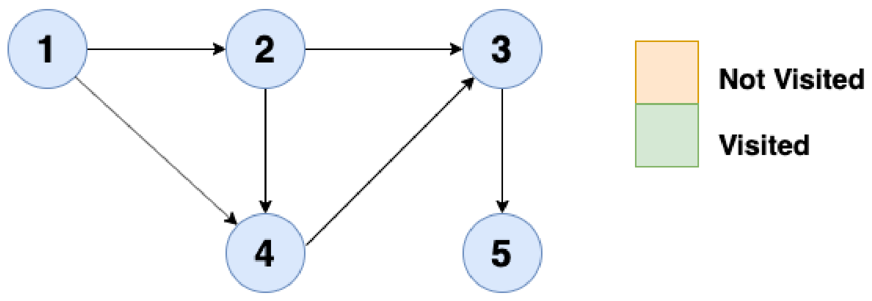


- For example, a topological sorting of the following graph is "A B E F D C ".
- There can be more than one topological sorting for a graph.

Topological Sort using Breadth First Search (BFS)/ Kahn's Algorithm

For that, we will maintain an array **T[ ],** which will store the ordering of the vertices in topological order. We will store the number of edges that are coming into a vertex in an array **in_degree[N],** where the ***i-th*** element will store the number of edges coming into the vertex ***i.*** We will also store whether a certain vertex has been visited or not in **visited[N].** We will follow the below steps:

- First, take out the vertex whose in_degree is 0. That means there is no edge that is coming into that vertex.
- We will append the vertices in the Queue and mark these vertices as visited.
- Now we will traverse through the queue and in each step we will dequeue () the front element in the Queue and push it into the **T.**
- Now, we will put out all the edges that are originated from the front vertex which means we will decrease the in_degree of the vertices which has an edge with the front vertex.
- Similarly, for those vertices whose in_degree is 0, we will push it in Queue and also mark that vertex as visited.



Step 1

Queue = [1]

in_degree[ ]

0	1	2	2	1
1	2	3	4	5

T = [ ]

Step 2

Queue = [2]

in_degree[ ]

0	0	2	1	1
1	2	3	4	5

T = [1]

Step 3

Queue = [4]

in_degree[ ]

0	0	1	0	1
1	2	3	4	5

T = [1, 2]

Step 4

Queue = [3]

in_degree[ ]

0	0	0	0	1
1	2	3	4	5

T = [1, 2, 4]

Step 5

Queue = [5]

in_degree[ ]

0	0	0	0	0
1	2	3	4	5

T = [1, 2, 4, 3]

Step 6

Queue = [ ]

in_degree[ ]

0	0	0	0	0
1	2	3	4	5

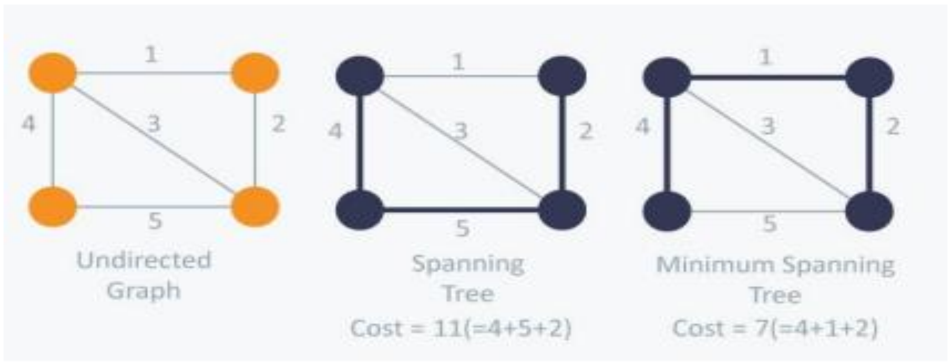
T = [1, 2, 4, 3, 5]

Chapter 10 Graphs

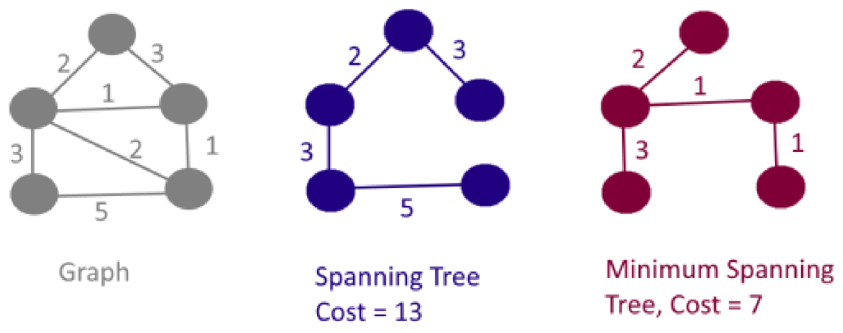
Minimum Spanning Tree:

- Given a ***connected and undirected graph***, a spanning tree of that graph is a subgraph that is a tree and ***connects all the vertices together and the graph doesn't have any nodes which loop back to itself.***
- A single graph can have many different spanning trees.
- A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree.
- The weight of a spanning tree is the sum of weights given to each edge of the spanning tree
- In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the 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 **has (V – 1) edges** where V is the number of vertices in the given graph.

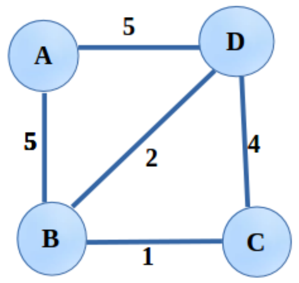
Example 1:

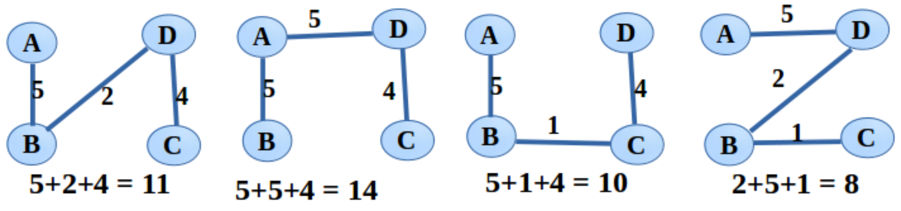


Example 2:



Example 3:





Kruskal's algorithm:

Below are the steps for finding MST using Kruskal's algorithm

Step 1: Remove all loops and parallel edges. In case of parallel edges, keep the one which has the least cost associated and remove all others.

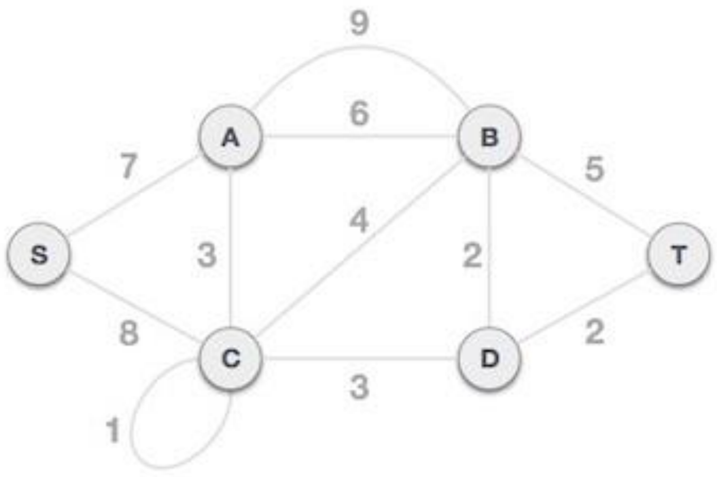
Step 2: Arrange all edges in their increasing order of weight

Step 3: Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.

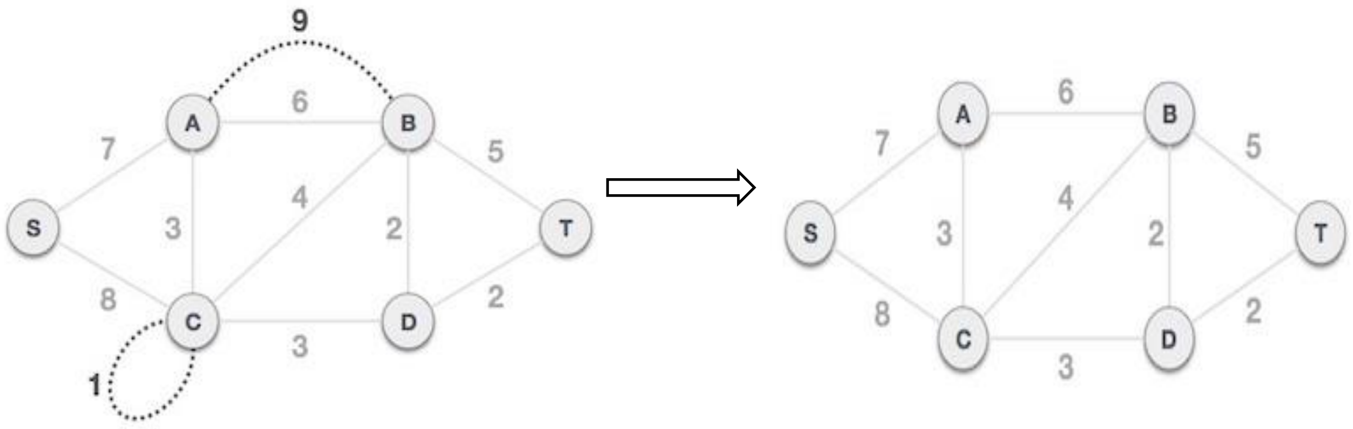
Step 4: Repeat step 3 until there are (V-1) edges in the spanning tree.

Chapter 10 Graphs

Example 1:



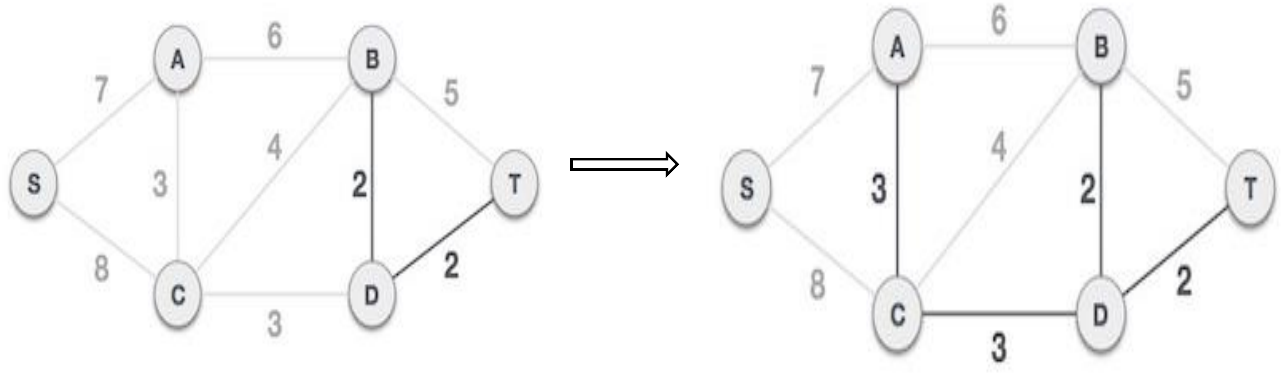
- Remove all loops and parallel edges. In case of parallel edges, keep the one which has the least cost associated and remove all others.



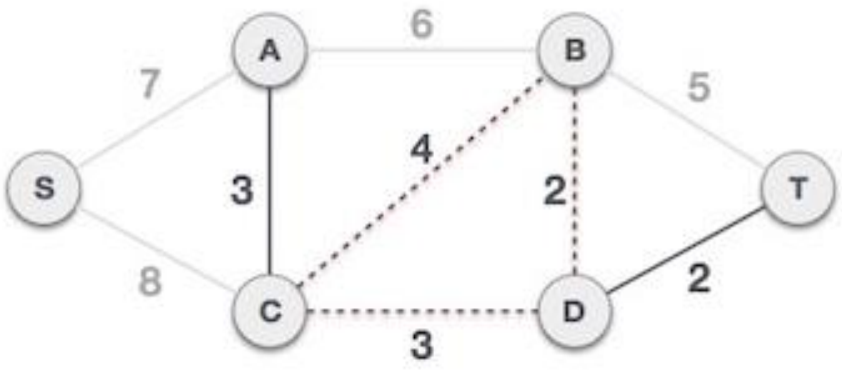
- Arrange all edges in their increasing order of weight. Create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

- Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.



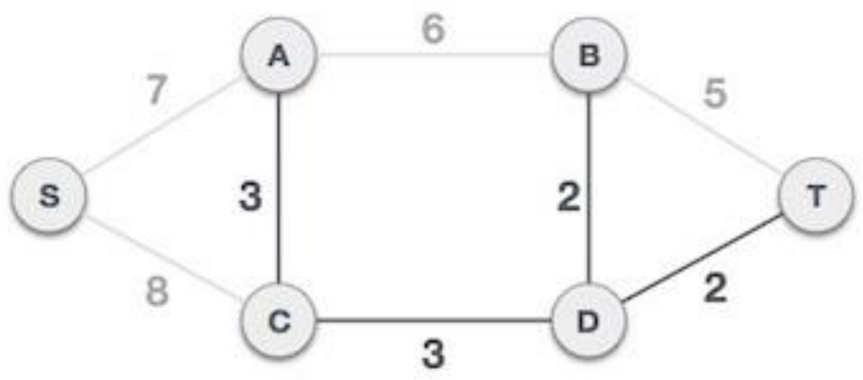
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. −We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



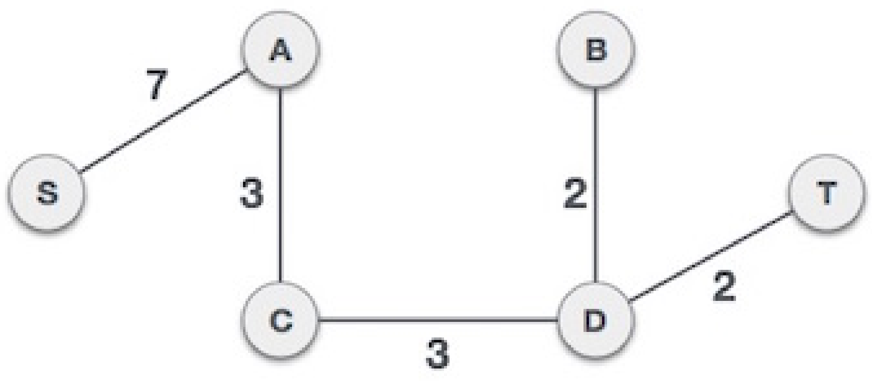
We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on

By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

Prim's Algorithm is a greedy algorithm that is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.



Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.



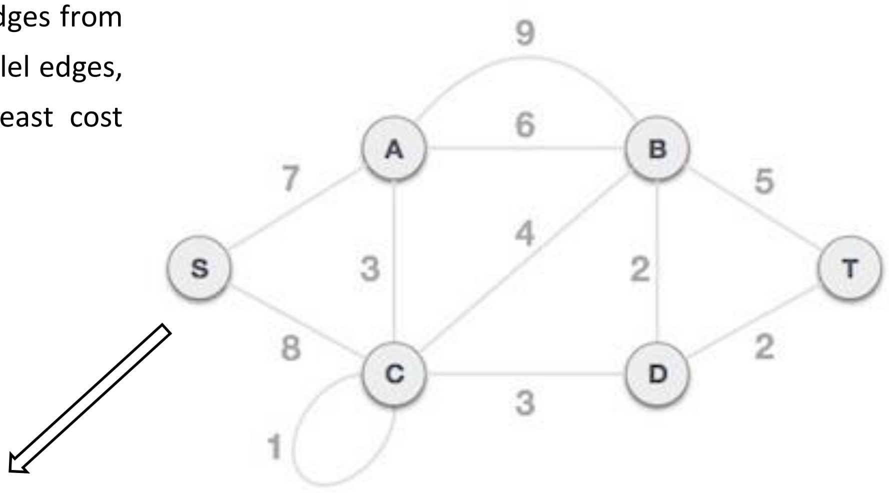
The steps to implement the prim's algorithm are given as follows -

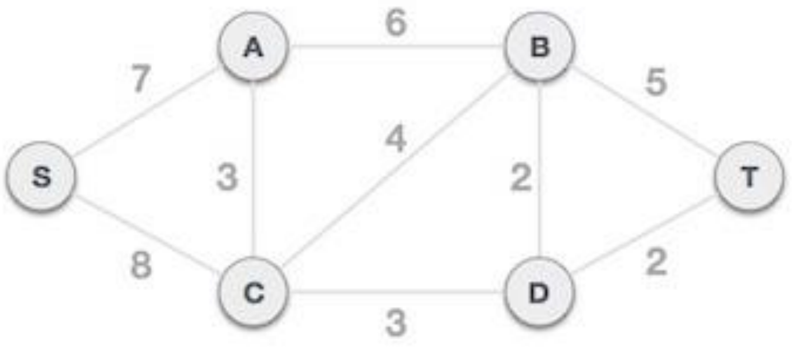
- First, remove all loops and parallel edges and we have to initialize an MST with the randomly chosen vertex.

- Now, we have to find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree.
- Repeat step 2 until the minimum spanning tree is formed.

Example 1:

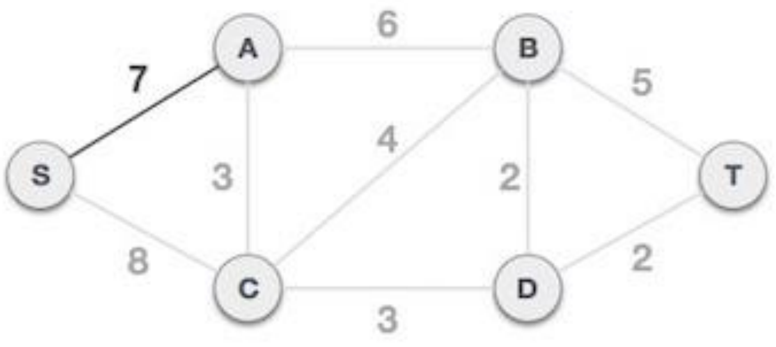
Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all other



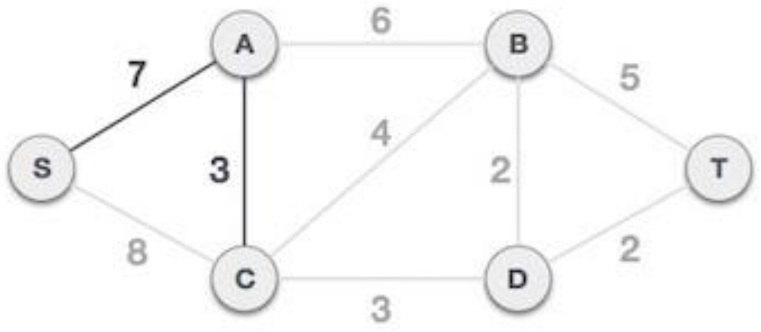


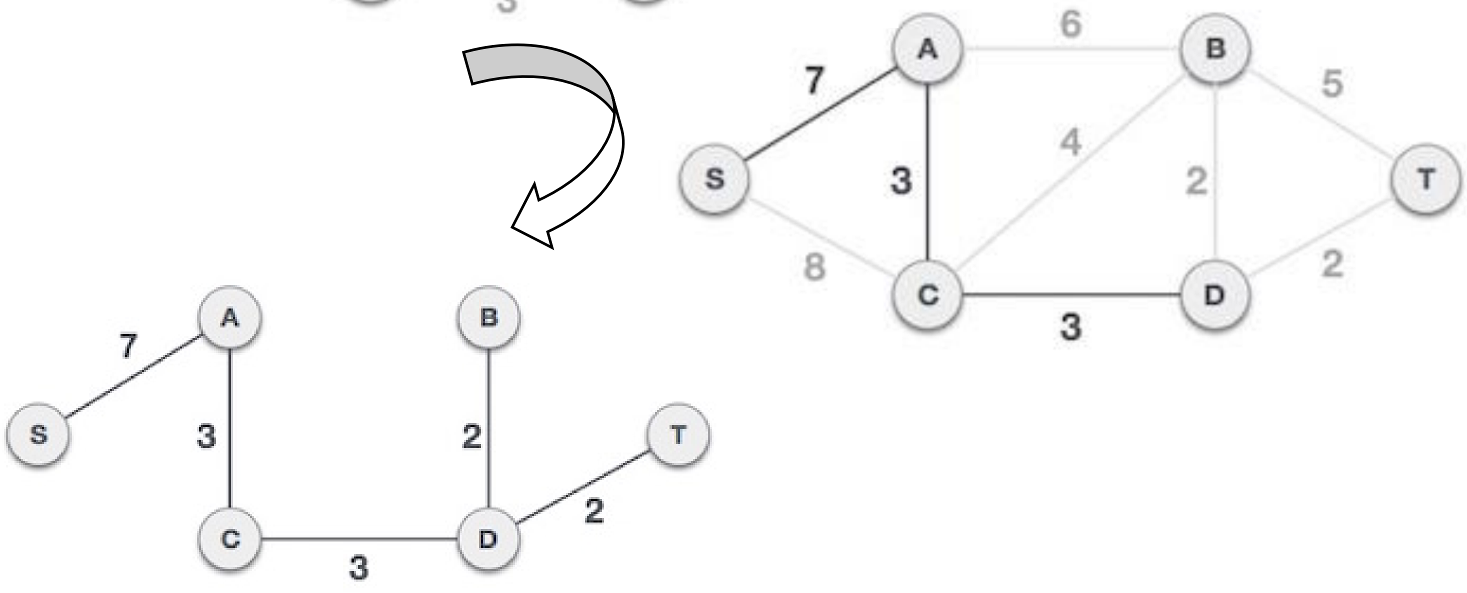
Choose any arbitrary node as root node

In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any node can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.



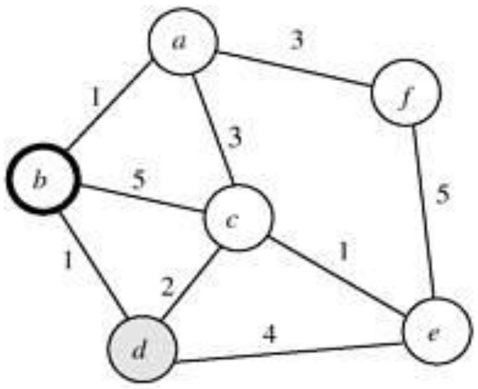
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree i.e., S-7-A-3-C.





Example 2:

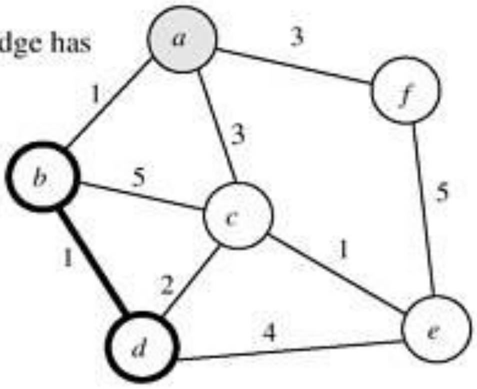
(a) Original graph



	a	b	c	d	e	f
d[]	1	0	5	1	∞	∞

$$\begin{array}{c|cccccc}
a & 0 & 1 & 3 & \infty & \infty & 3 \\
b & 1 & 0 & 5 & 1 & \infty & \infty \\
c & 3 & 5 & 0 & 2 & 1 & \infty \\
d & \infty & 1 & 2 & 0 & 4 & \infty \\
e & \infty & \infty & 1 & 4 & 0 & 5 \\
f & 2 & \infty & \infty & \infty & 5 & 0
\end{array}$$

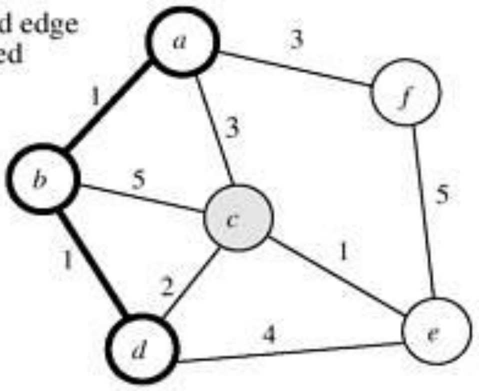
(b) After the first edge has been selected



	a	b	c	d	e	f
d[]	1	0	2	1	4	∞

$$\begin{array}{c|cccccc}
a & 0 & 1 & 3 & \infty & \infty & 3 \\
b & 1 & 0 & 5 & 1 & \infty & \infty \\
c & 3 & 5 & 0 & 2 & 1 & \infty \\
d & \infty & 1 & 2 & 0 & 4 & \infty \\
e & \infty & \infty & 1 & 4 & 0 & 5 \\
f & 2 & \infty & \infty & \infty & 5 & 0
\end{array}$$

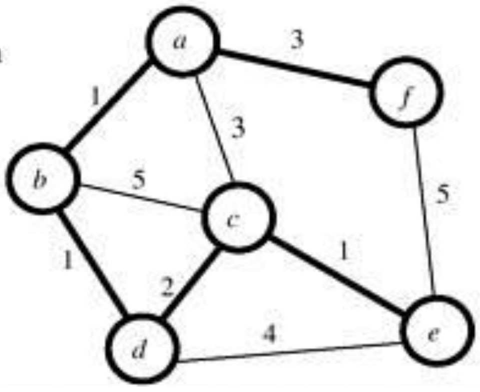
(c) After the second edge has been selected



	a	b	c	d	e	f
d[]	1	0	2	1	4	3

$$\begin{array}{c|cccccc}
a & 0 & 1 & 3 & \infty & \infty & 3 \\
b & 1 & 0 & 5 & 1 & \infty & \infty \\
c & 3 & 5 & 0 & 2 & 1 & \infty \\
d & \infty & 1 & 2 & 0 & 4 & \infty \\
e & \infty & \infty & 1 & 4 & 0 & 5 \\
f & 2 & \infty & \infty & \infty & 5 & 0
\end{array}$$

(d) Final minimum spanning tree



	a	b	c	d	e	f
d[]	1	0	2	1	1	3

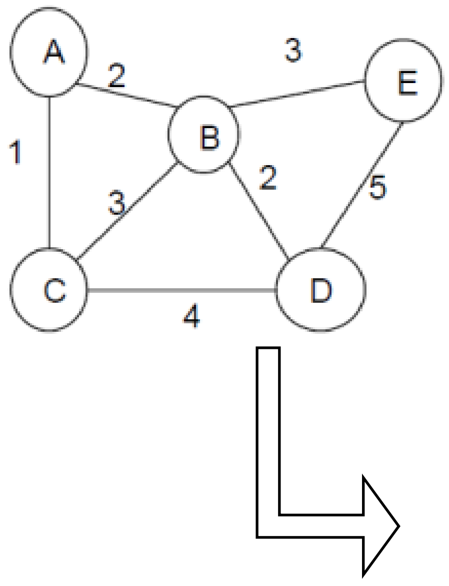
$$\begin{array}{c|cccccc}
a & 0 & 1 & 3 & \infty & \infty & 3 \\
b & 1 & 0 & 5 & 1 & \infty & \infty \\
c & 3 & 5 & 0 & 2 & 1 & \infty \\
d & \infty & 1 & 2 & 0 & 4 & \infty \\
e & \infty & \infty & 1 & 4 & 0 & 5 \\
f & 2 & \infty & \infty & \infty & 5 & 0
\end{array}$$

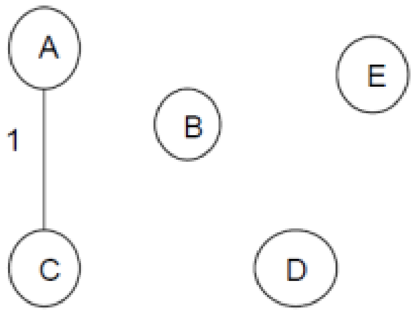
Round Robin Algorithm:

This method provides better performance when the number of edges is low. Initially each node is considered to be a partial tree. Each partial tree is maintained in a queue Q.

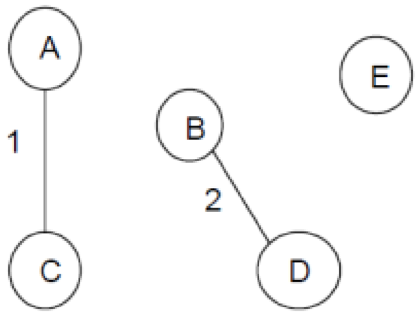
- Priority queue is associated with each partial tree, which contains all the arcs ordered by their weights.
- The algorithm proceeds by removing a partial tree, T1, from the front of Q, finding the minimum weight arc a in T1; deleting from Q, the tree T2, at the other end of arc a; combining T1 and T2 into a single new tree T3 and at the same time combining priority queues of T1 and T2 and adding T3 at the rear of priority queue.
- This continues until Q contains a single tree, the minimum spanning tree.

Q	Priority queue
{A}	1, 2
{B}	2, 2, 3, 3
{C}	1, 3, 4
{D}	2, 4, 5
{E}	3, 5

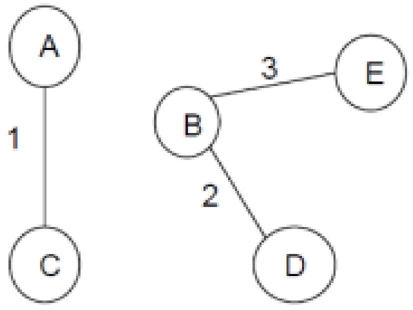




Q	Priority queue
{B}	2, 2, 3, 3
{D}	2, 4, 5
{E}	3, 5
{A, C}	2, 3, 4

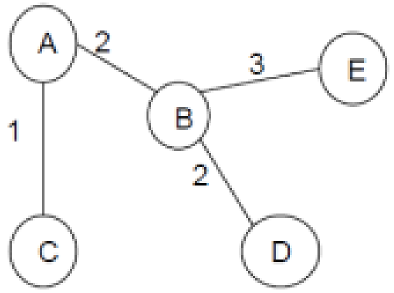


Q	Priority queue
{E}	3, 5
{A, C}	2, 3, 4
{B, D}	2, 3, 3, 4, 5



Q	Priority queue
{A, C}	2, 3, 4
{E, B, D}	2, 3, 4, 5, 5

Q	Priority queue
{A, C, E, B, D}	3, 3, 4, 4, 5, 5



Only 1 partial tree is left in the queue, which is the required minimum spanning tree.

Chapter 10 Graphs

Shortest path algorithm

Greedy Algorithm:

A greedy algorithm is a simple, intuitive algorithm that is used in optimization problems. ***The algorithm makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem.***

It follows local optimal choice of each stage with intend of finding global optimum.

Greedy algorithms are quite successful in some problems, such as Huffman encoding which is used to compress data, or Dijkstra's algorithm, which is used to find the shortest path through a graph.

Chapter 10 Graphs

Dijkstra's shortest path algorithm

Given a graph and a source vertex in the graph, find shortest paths from source to all vertices in the given graph.

Algorithm

1. Create a set sptSet (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

2. Assign a distance value to all vertices in the input graph. Initialize all distance values as

INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

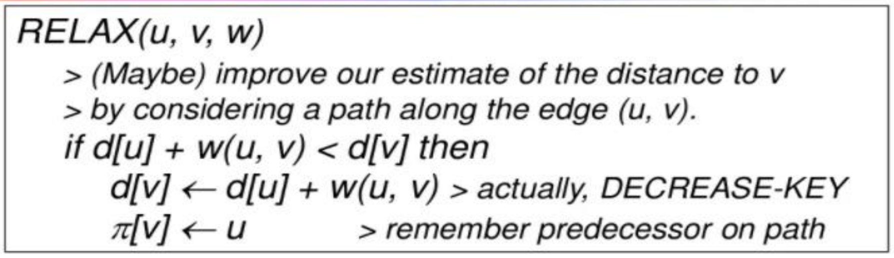
3. While sptSet doesn't include all vertices

   a) Pick a vertex u which is not there in sptSet and has minimum distance value.

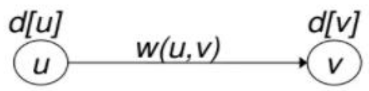
   b) Include u to sptSet.

   c) Update distance value of all adjacent vertices of u.

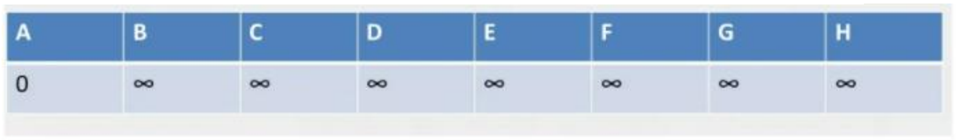
   To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.



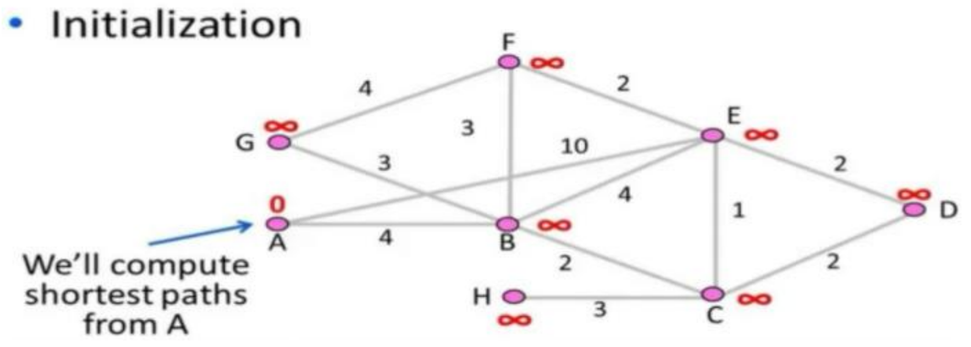
```
RELAX(u, v, w)
    > (Maybe) improve our estimate of the distance to v
    > by considering a path along the edge (u, v).
    if $d[u] + w(u, v) < d[v]$ then
        $d[v] \leftarrow d[u] + w(u, v)$   > actually, DECREASE-KEY
        $\pi[v] \leftarrow u$               > remember predecessor on path
```



Example:

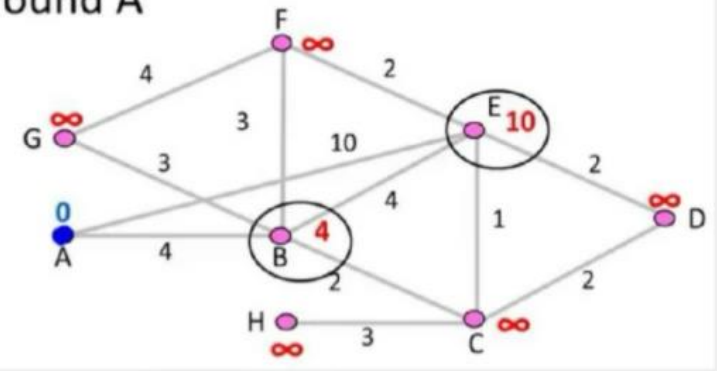


A	B	C	D	E	F	G	H
0	∞	∞	∞	∞	∞	∞	∞



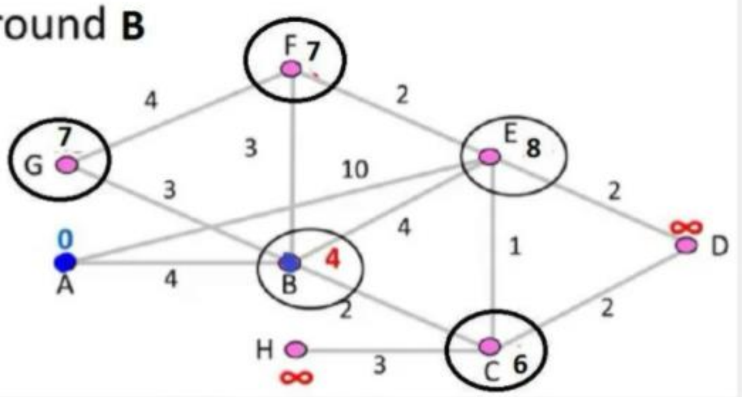
A	B	C	D	E	F	G	H
0	4	∞	∞	10	∞	∞	∞

- Relax around A



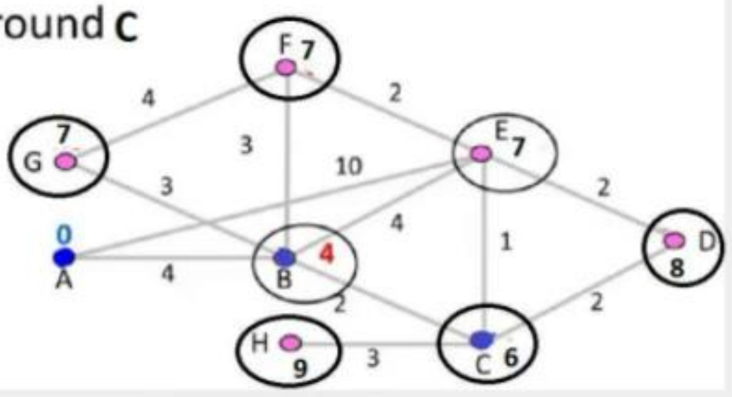
A	B	C	D	E	F	G	H
0	4	6	∞	8	7	7	∞

- Relax around **B**



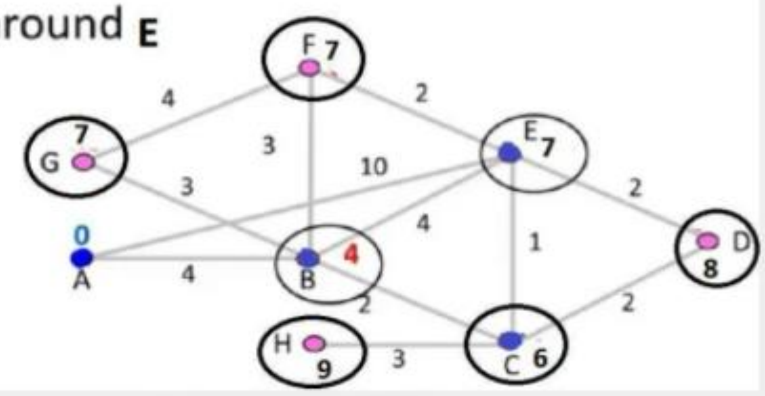
A	B	C	D	E	F	G	H
0	4	6	8	7	7	7	9

- Relax around **C**



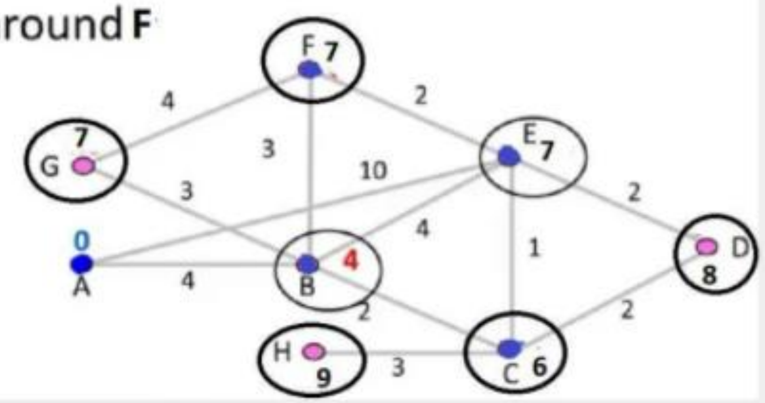
A	B	C	D	E	F	G	H
0	4	6	8	7	7	7	9

- Relax around **E**



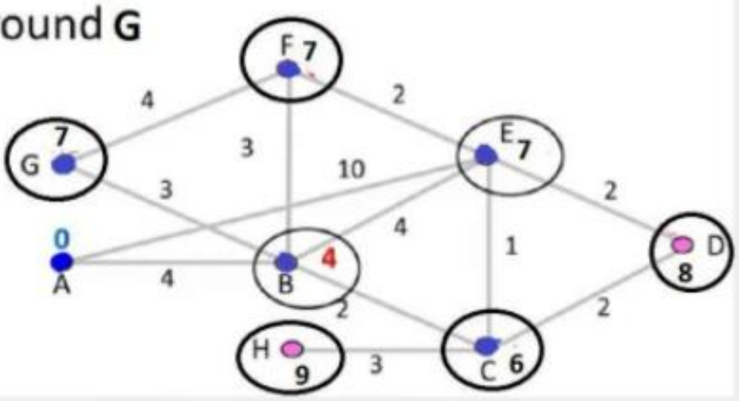
A	B	C	D	E	F	G	H
0	4	6	8	7	7	7	9

- Relax around **F**



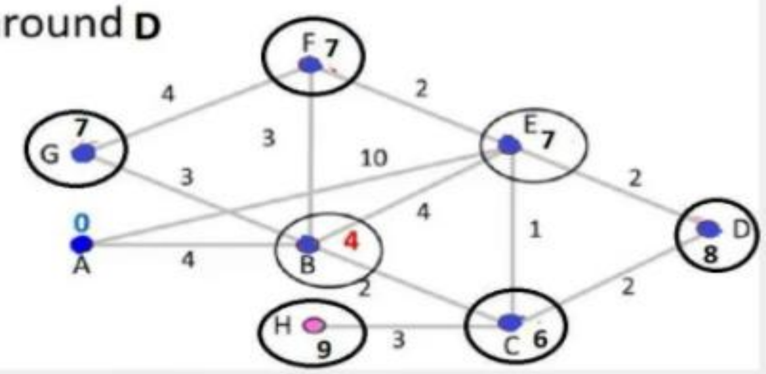
A	B	C	D	E	F	G	H
0	4	6	8	7	7	7	9

- Relax around **G**



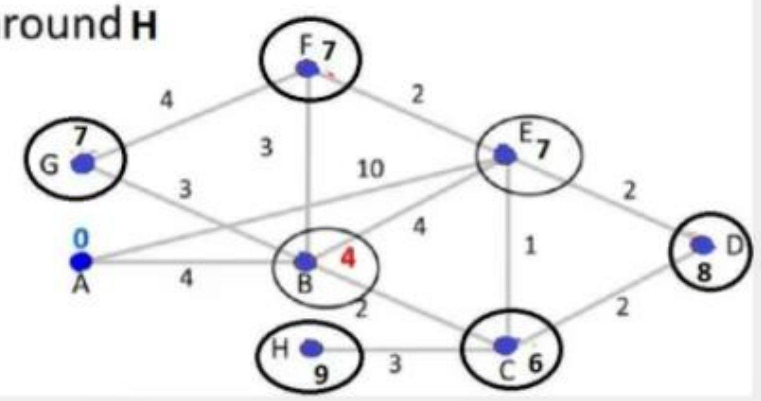
A	B	C	D	E	F	G	H
0	4	6	8	7	7	7	9

- Relax around **D**



A	B	C	D	E	F	G	H
0	4	6	8	7	7	7	9

- Relax around **H**



- So , the shortest path from source node A to all other nodes are –
  - A – B = 4
  - A – C = 6
  - A – D = 8
  - A – E = 7
  - A – F = 7
  - A – G = 7
  - A – H = 9