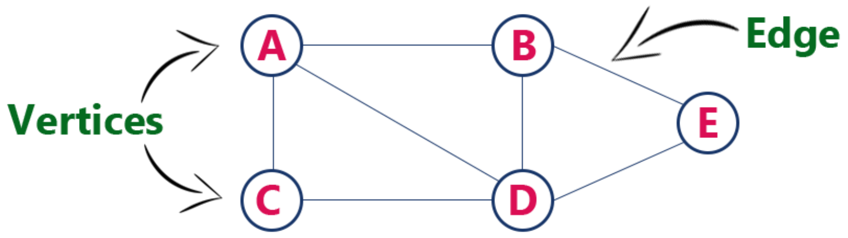
Graph Data Structure

Introduction to Graphs

- 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 is defined as follows...
- Graph is a collection of vertices and arcs in which vertices are connected with arcs
- Graph is a collection of nodes and edges in which nodes are connected with edges
- Generally, a graph G is represented as G = (V , E), where V is set of vertices and E is set of edges.

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 Terminology

We use the following terms in graph data structure...

Vertex

- 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 (startingVertex, endingVertex).
- For example, in above graph the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

  Edges are three types.

Undirected Edge - An undirected egde is a bidirectional edge. If there is undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).

Directed Edge - A directed egde is a unidirectional edge. If there is directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).

Weighted Edge - A weighted egde is a edge with value (cost) on it.

Undirected Graph

- A graph with only undirected edges is said to be undirected graph.

Directed Graph

- A graph with only directed edges is said to be directed graph.

Mixed Graph

- A graph with both undirected and directed edges is said to be mixed graph.

End vertices or Endpoints

- The two vertices joined by edge are called end vertices (or endpoints) of that edge.

Origin

- If a edge is directed, its first endpoint is said to be the origin of it.

Destination

- If a edge is directed, its first endpoint is said to be the origin of it and the other endpoint is said to be the destination of that edge.

Adjacent

- If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, vertices A and B are said to be adjacent if there is an edge between them.

Incident

- Edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.

Outgoing Edge

- A directed edge is said to be outgoing edge on its origin vertex.

Incoming Edge

- A directed edge is said to be incoming edge on its destination vertex.

Degree

- Total number of edges connected to a vertex is said to be degree of that vertex.

Indegree

- ➢ Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

Outdegree

- ➢ Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

Parallel edges or Multiple edges

- ➢ If there are two undirected edges with same end vertices and two directed edges with same origin and destination, such edges are called parallel edges or multiple edges.

Self-loop

- ➢ Edge (undirected or directed) is a self-loop if its two endpoints coincide with each other.

Simple Graph

- ➢ A graph is said to be simple if there are no parallel and self-loop edges.

Path

- ➢ A path is a sequence of alternate vertices and edges that starts at a vertex and ends at other vertex such that each edge is incident to its predecessor and successor vertex.

Graph Representations

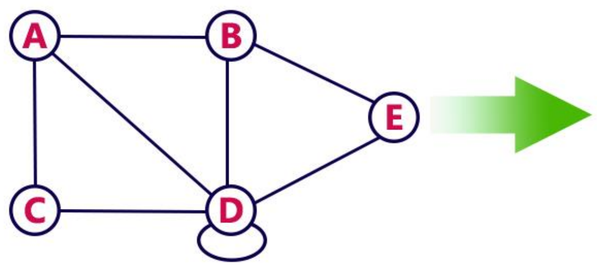
Graph data structure is represented using following representations...

- **Adjacency Matrix**
- **Incidence Matrix**
- **Adjacency List**

Adjacency Matrix

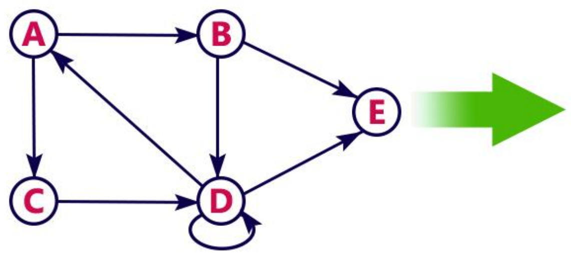
- ➢ In this representation, the graph is represented using a matrix of size total number of vertices by a total number of vertices.
- ➢ That means a graph with 4 vertices is represented using a matrix of size 4X4. In this matrix, both rows and columns represent vertices.
- ➢ This matrix is filled with either 1 or 0. Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex.

  For example, consider the following undirected graph representation.



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

Directed graph representation...

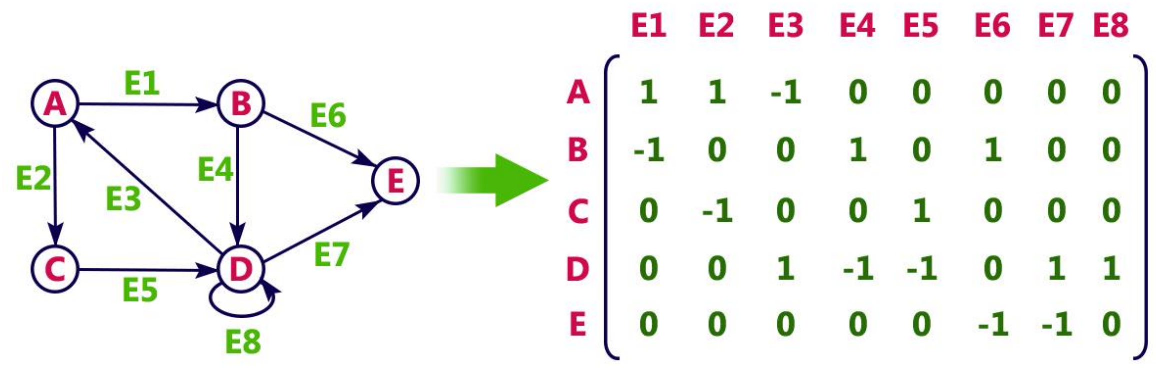


$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 0 & 0 \\ B & 0 & 0 & 0 & 1 & 1 \\ C & 0 & 0 & 0 & 1 & 0 \\ D & 1 & 0 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 \end{array}$$

Incidence Matrix

- In this representation, the graph is represented using a matrix of size total number of vertices by a total number of edges.
- That means graph with 4 vertices and 6 edges is represented using a matrix of size 4X6. In this matrix, rows represent vertices and columns represents edges.
- This matrix is filled with 0 or 1 or -1. Here, 0 represents that the row edge is not connected to column vertex, 1 represents that the row edge is connected as the outgoing edge to column vertex and -1 represents that the row edge is connected as the incoming edge to column vertex.

  For example, consider the following directed graph representation...

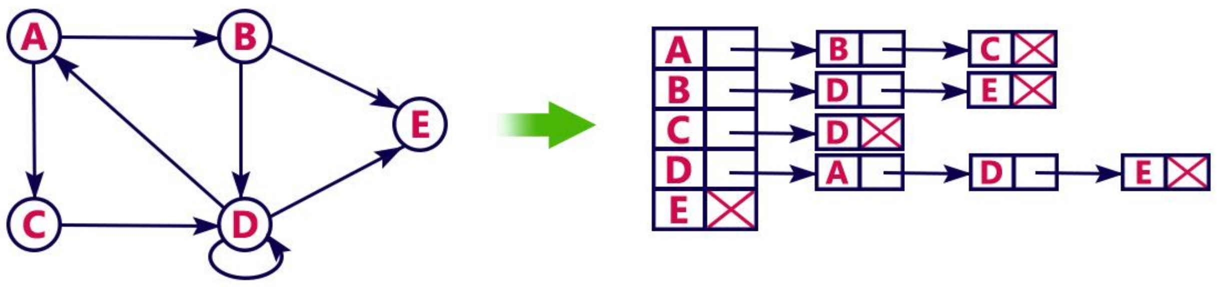


$$
\begin{array}{c|cccccccc}
 & E1 & E2 & E3 & E4 & E5 & E6 & E7 & E8 \\
\hline
A & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\
B & -1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
C & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\
D & 0 & 0 & 1 & -1 & -1 & 0 & 1 & 1 \\
E & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0
\end{array}
$$

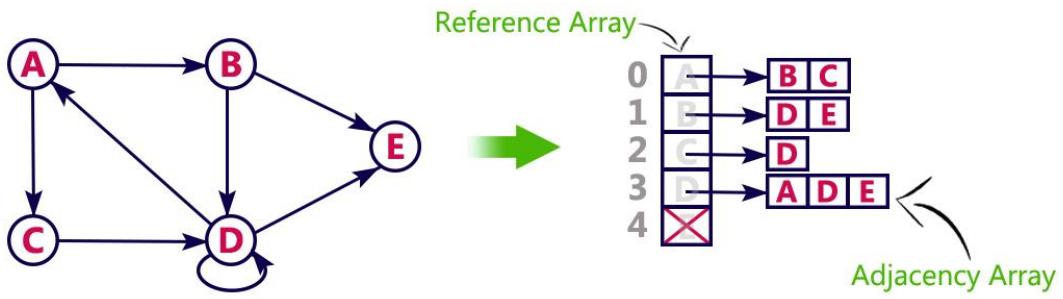
Adjacency List

- ➢ In this representation, every vertex of a graph contains list of its adjacent vertices.

  For example, consider the following directed graph representation implemented using linked list...



This representation can also be implemented using an array as follows..



Graph Traversal

- Graph traversal is a technique used for a searching 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.

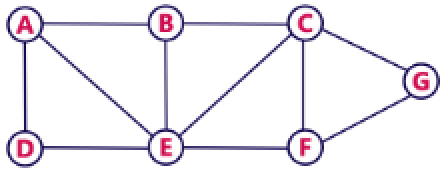
Step 7 - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

---

Back tracking is coming back to the vertex from which we reached the current vertex.

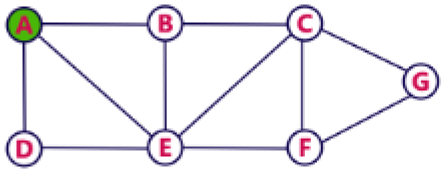
Example

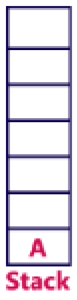
Consider the following example graph to perform DFS traversal



Step 1:

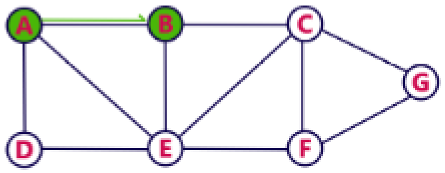
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.

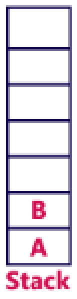




Step 2:

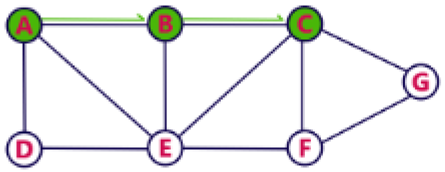
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

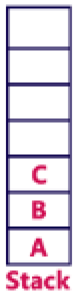




Step 3:

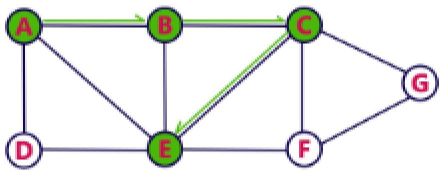
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

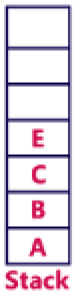




Step 4:

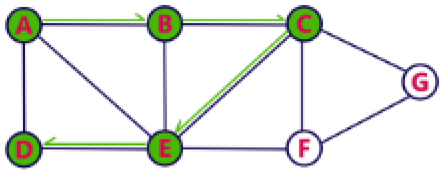
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

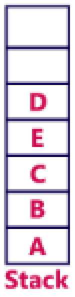




Step 5:

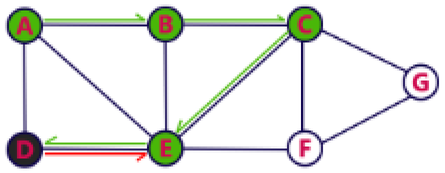
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

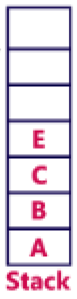




Step 6:

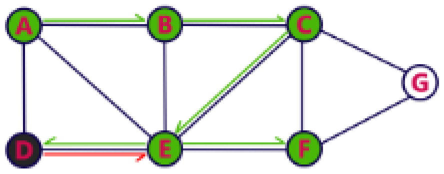
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

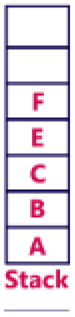




Step 7:

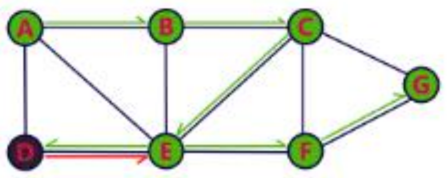
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

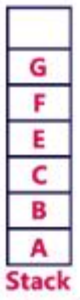




Step 8:

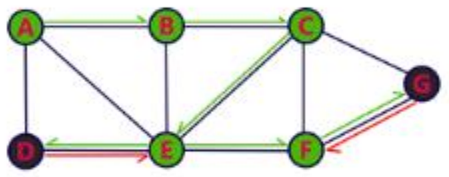
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

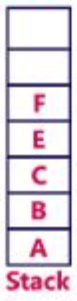




Step 9:

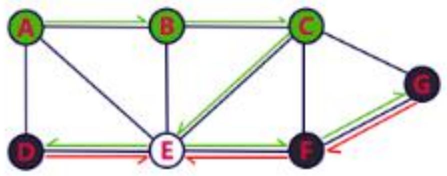
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

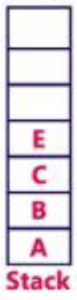




Step 10:

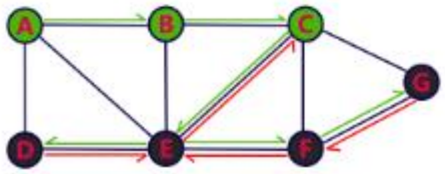
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

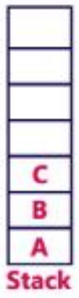




Step 11:

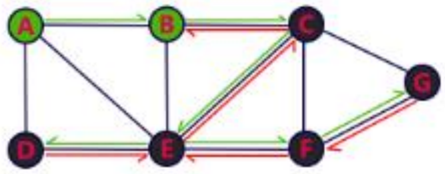
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

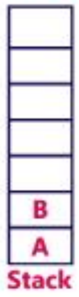




Step 12:

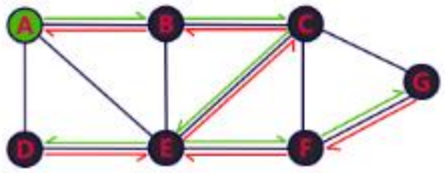
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

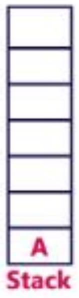




Step 13:

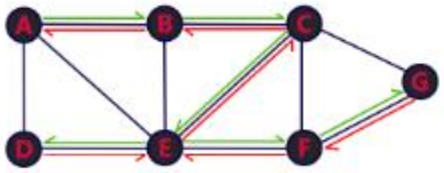
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

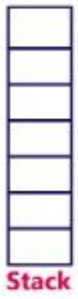




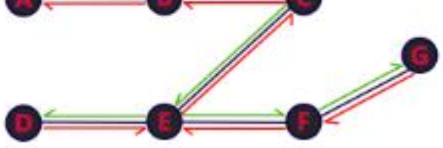
Step 14:

- 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.



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.

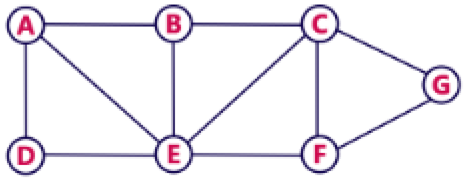
Step 4 - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.

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

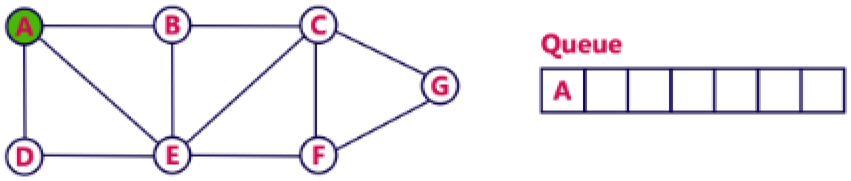
Example

Consider the following example graph to perform BFS traversal



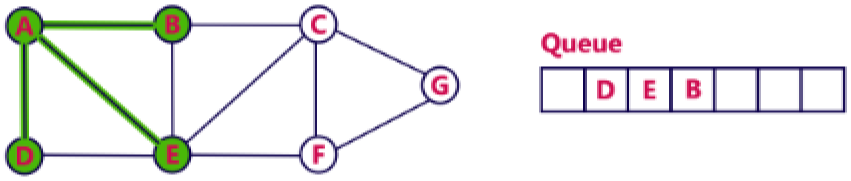
Step 1:

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



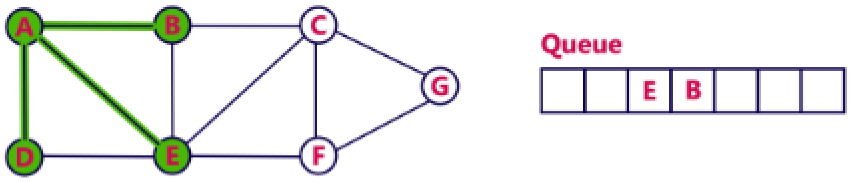
Step 2:

- 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..



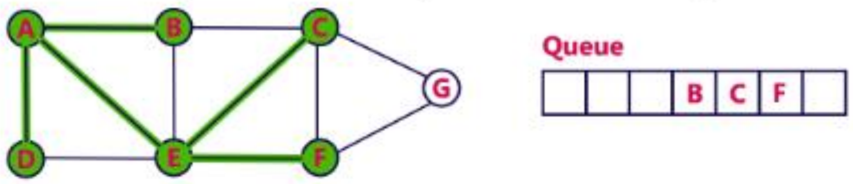
Step 3:

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



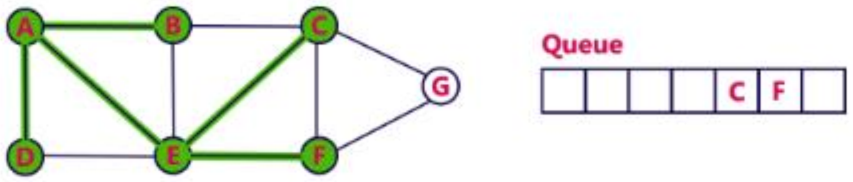
Step 4:

- 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.



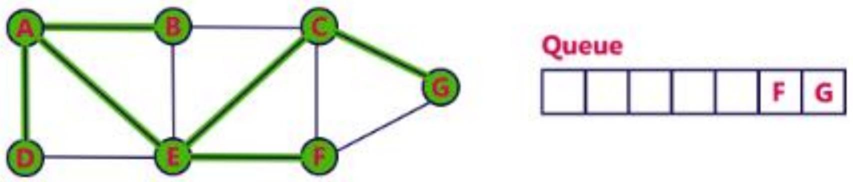
Step 5:

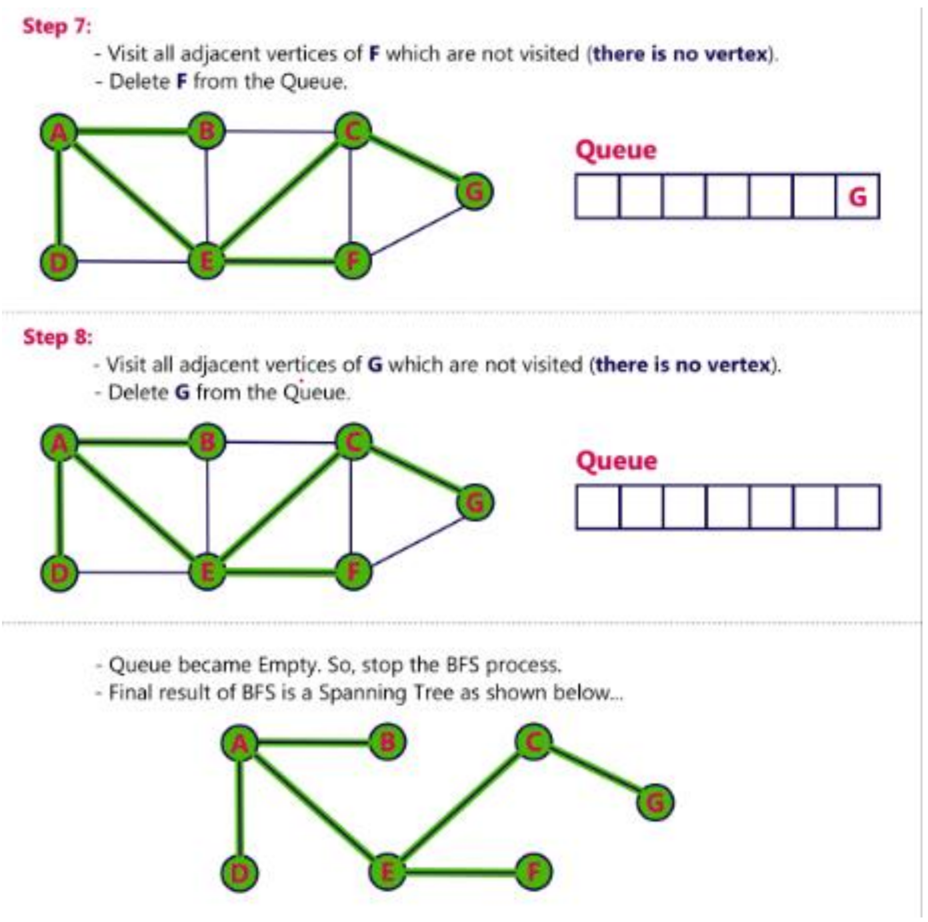
- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.



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...

Comparison between Graph and Tree: -

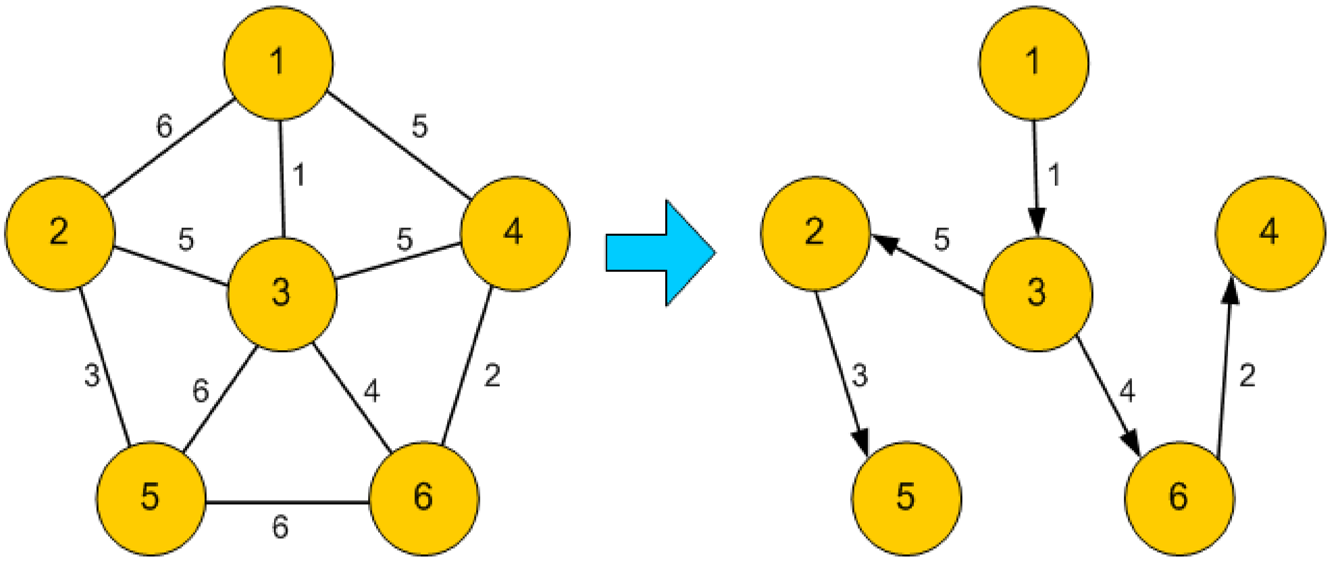
Graph	**Tree**
Non-linear data structure.	Non-linear data structure.
Collection of vertices and edges.	Collection of vertices and edges.
Each node can have any number of edges.	In binary trees every node can have at the most 2 child nodes.
There is no unique node like trees.	There is a unique node called root node.
A cycle can be formed.	There will not be any cycle.
Applications: used for finding shortest path in networking graph.	Applications: used for expression trees, and decision tree.

Spanning Tree:

- A spanning tree has a property that for any pair of vertices there exists only one path between them, and the insertion of any edge to a spanning tree does not form a cycle.
- The spanning tree resulting from a call to depth first tree is known as depth first spanning tree. The spanning tree resulting from a call to breadth first tree is known as a breadth first spanning tree.
- The minimum spanning tree for a graph is the spanning tree of minimum cost for that graph.

Algorithm:

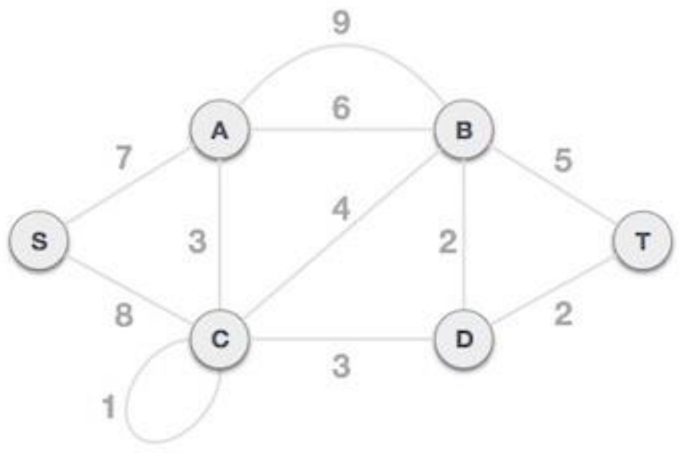
1. Insert the first vertex into the tree.
2. From every vertex already in the tree, examine the total path length to all adjacent vertices not in the tree. Select the edge with the minimum total path weight and insert it into the tree.
3. Repeat step 2 until all vertices are in the tree.



Kruskal's Spanning Tree Algorithm

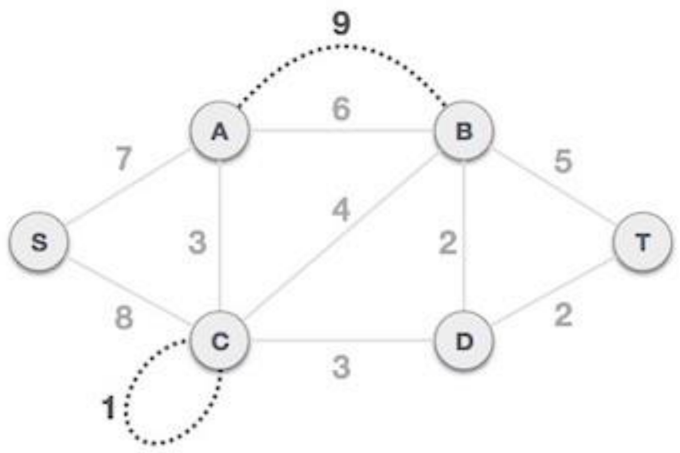
- Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree.
- A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

To understand Kruskal's algorithm let us consider the following example −

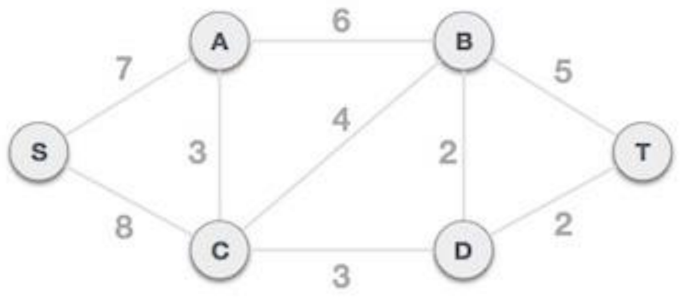


Step 1 - Remove all loops and Parallel Edges

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 others.



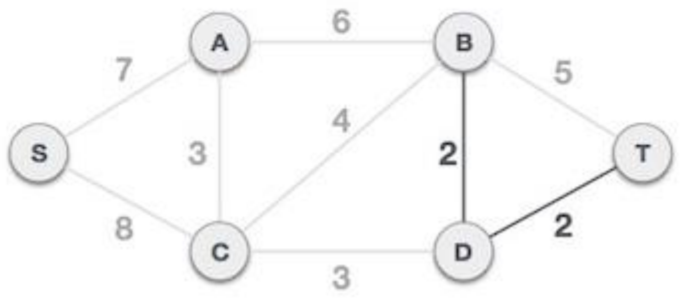
Step 2 - Arrange all edges in their increasing order of weight

The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

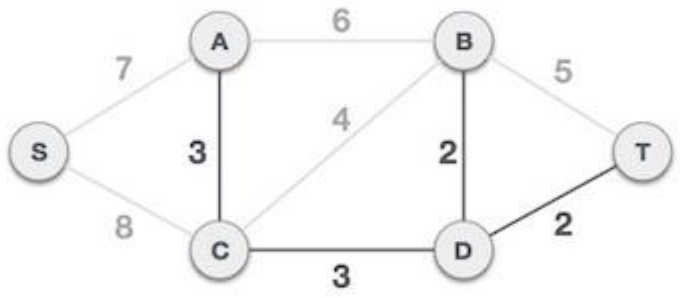
B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

Step 3 - Add the edge which has the least weightage

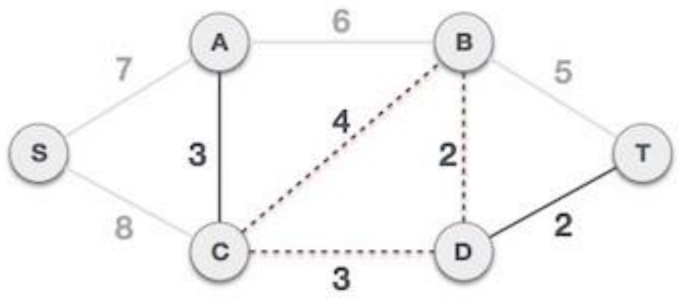
- Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact.
- In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.



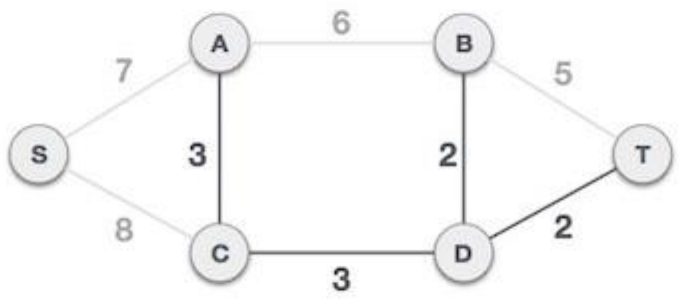
- The least cost is 2 and edges involved are B,D and D,T. We add them.
- Adding them does not violate spanning tree properties, so we continue to our next edge selection.
- Next cost is 3, and associated edges are A,C and C,D. We add them again −



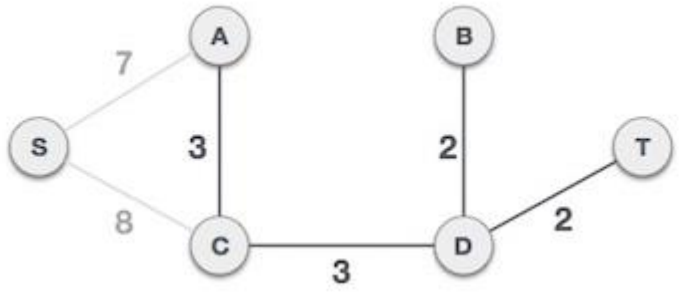
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. −



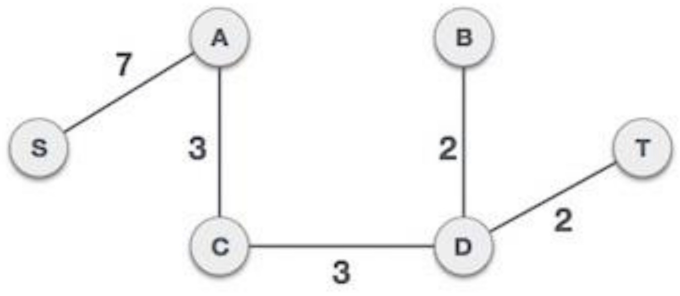
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



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



Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.

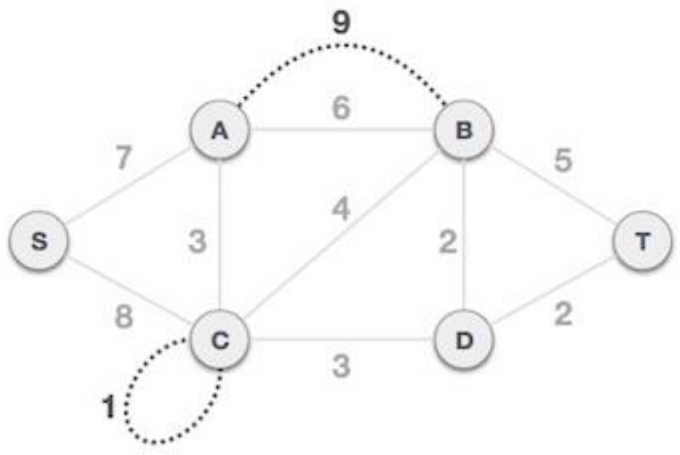


By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

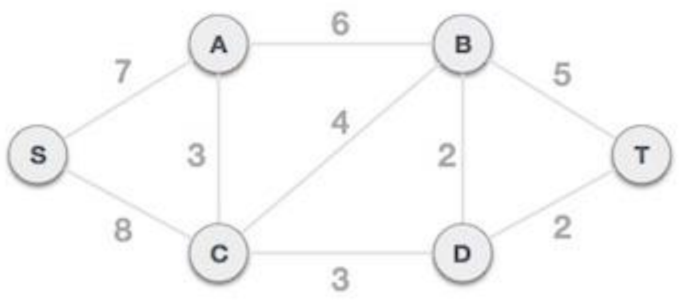
Prim's Spanning Tree Algorithm

- Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach.
- Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

Step 1 - Remove all loops and parallel edges



- 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 others.

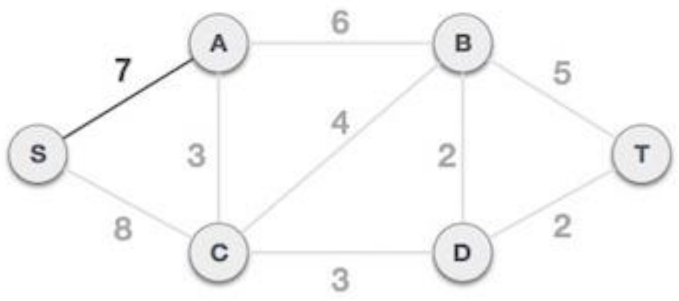


Step 2 - Choose any arbitrary node as root node

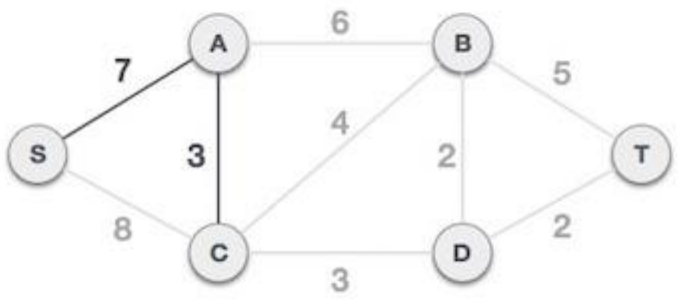
- In this case, we choose **S** node as the root node of Prim's spanning tree.
- This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node.
- So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected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

Step 3 - Check outgoing edges and select the one with less cost

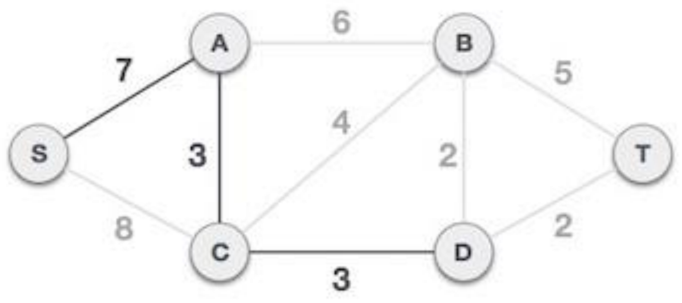
- After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



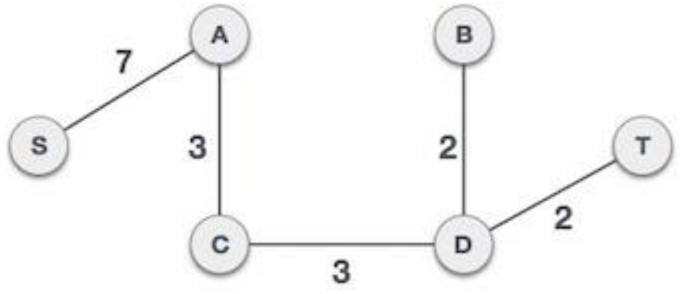
- Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



- After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



- After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one.
- But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



Applications of Graph: -

1. In computer networking such as Local Area Network (LAN), Wide Area Network (WAN), internetworking.
2. In telephone cabling graph theory is efficiently used.
3. In job scheduling algorithms.

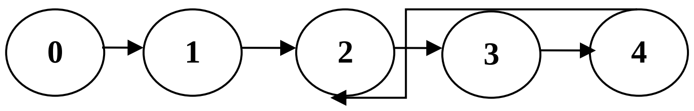
Since graph is nothing but the collection of nodes (vertices) and edges. One can find the best suited distance between the two vertices is reduced then the cost of calling and the time traveling through the nodes will get reduced.

Transitive Closure:

- The Transitive closure matrix, denoted as A^+, of a directed graph G, is a matrix such that

A^+ [i][j] = 1; if there is a path of length > 0 from i to j;

A^+ [i][j] = 0; Otherwise.



Graph

	0	1	2	3	4
0.	0	1	0	0	0
1.	0	0	1	0	0
2.	0	0	0	1	0
3.	0	0	0	0	1
4.	0	0	1	0	0

Adjacency Matrix A

	0	1	2	3	4
1.	0	1	1	1	1
2.	0	0	1	1	1
3.	0	0	1	1	1
4.	0	0	1	1	1
5.	0	0	1	1	1

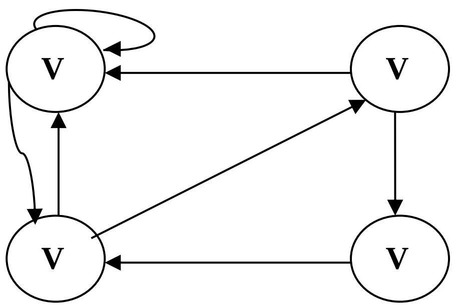
Adjacency Matrix A^+

Dijkstra’s shortest path Algorithm

- Dijkstra’s shortest path is used to find the shortest path between two vertices in a network. For example, if the network represents the routes flown by an airline, when we travel we want to find the least expensive route between home and our destination.
- When the weights in the route graph are the flight fare, our minimum cost is the shortest path between 2 nodes. Edsger dijkstra developed a classic algorithm for just this problem. This algorithm is generally known simply as Dijkstra’s shortest path algorithm.

This algorithm can be understood with the help of an example.

Consider the digraph shown in figure:



The adjacency matrix shows the cost of edges for the digraph. Also, the matrix on the right side indicates the path between two vertices.

$$\begin{array}{r|cccc} & 1 & 2 & 3 & 4 \\ \hline 1. & 7 & 5 & - & - \\ 2. & 7 & - & - & 2 \\ 3. & - & 3 & - & - \\ 4. & 4 & - & 1 & - \end{array} \quad \begin{pmatrix} 11 & 12 & - & - \\ 21 & - & - & 24 \\ - & 32 & - & - \\ 41 & - & 43 & - \end{pmatrix}$$

- Some of the values of adjacency matrix are infinity (-), which indicates that there is no direct path between the vertices.
- The blank sign in right hand side matrix indicates that till now the path is not determined.
- The minimum cost between two vertices is determined by considering one vertex at one time.
- Then the path from particular vertex is calculated. If the cost is found to be smaller then previous cost then it is replaced with the new cost.

Consider the vertex 1 in the given example, the adjacency matrix now holds the values as follows:

$$\begin{array}{r|cccc} & 1 & 2 & 3 & 4 \\ \hline 1. & 7 & 5 & - & - \\ 2. & 7 & \boxed{12} & - & 2 \\ 3. & - & 3 & - & - \\ 4. & 4 & \boxed{9} & 1 & - \end{array} \quad \begin{pmatrix} 11 & 12 & - & - \\ 21 & \boxed{212} & - & 24 \\ - & 32 & - & - \\ 41 & \boxed{412} & 43 & - \end{pmatrix}$$

- The values shown in square are the values that are added.
- Here the path 412 is possible as vertex 1 is considered. But the path 124 is not possible because till now the vertex 2 is not considered. So, via 2 we cannot travel to any other vertex.

Now the vertex 2 is considered, then the adjacency matrix is

$$\begin{array}{r|cccc} & 1 & 2 & 3 & 4 \\ \hline 1. & 7 & 5 & - & \boxed{7} \\ 2. & 7 & 12 & - & 2 \\ 3. & \boxed{10} & 3 & - & \boxed{5} \\ 4. & 4 & 9 & 1 & \boxed{11} \end{array} \quad \begin{pmatrix} 11 & 12 & - & \boxed{124} \\ 21 & 212 & - & 24 \\ \boxed{321} & 32 & - & \boxed{324} \\ 41 & 412 & 43 & \boxed{4124} \end{pmatrix}$$

- Now the vertex 3 is considered, the paths between the vertices 4-2 and 4-4 are changed because these are the shortest as compared to earlier ones.

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1. & 7 & 5 & - & 7 \\ 2. & 7 & 12 & - & 2 \\ 3. & 10 & 3 & - & 5 \\ 4. & 4 & \boxed{4} & 1 & \boxed{6} \end{array} \quad \begin{pmatrix} 11 & 12 & - & 124 \\ 21 & 212 & - & 24 \\ 321 & 32 & - & 324 \\ 41 & \boxed{432} & 43 & \boxed{4324} \end{pmatrix}$$

- Finally, the vertex 4 is considered, which results into the adjacency matrix that holds the shortest path between any two vertices.

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1. & 7 & 5 & \boxed{8} & 7 \\ 2. & \boxed{6} & \boxed{6} & \boxed{3} & 2 \\ 3. & \boxed{9} & 3 & \boxed{6} & 5 \\ 4. & 4 & 4 & 1 & 6 \end{array} \quad \begin{pmatrix} 11 & 12 & \boxed{1243} & 124 \\ 21 & \boxed{2432} & \boxed{243} & 24 \\ \boxed{3241} & 32 & \boxed{3243} & 324 \\ 41 & 432 & 43 & 4324 \end{pmatrix}$$

Warshall's Algorithm: -

- This algorithm determines if there is a path from any vertex V_i to another vertex V_j either directly (or) through one (or) more intermediate vertices.
- With this algorithm, we can test the reachability of all the pairs of vertices in a graph.
- To compute the path matrix of a given graph, this algorithm is helpful.
- This algorithm treats the entries in the adjacency matrix as bit entries and perform AND (^) and OR (v) boolean operations on them.

Algorithm:

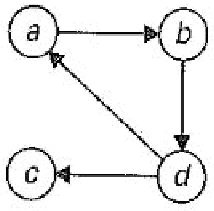
Input: A graph G whose pointer to its adjacency matrix is GPtr and vertices are labeled as 1, 2, 3, …., N; N being the number of vertices in the graph.
Output: The path matrix P
Data structure: Matrix representation of graph with pointer as GPtr.

```
Step 1: for i=1 to N do
        1: for j=1 to N do
                1: P [i][j] = GPtr [i][j]
        2: end for
        3: for k=1 to N do
                1: for i=1 to N do
                        1: for j=1 to N do
                                1: P [i][j] = P [i][j] v (P [i][k] ^ P {k][j])
                        2: end for
                2: end for
        4: end for
        5: return
        6: stop
```

The functionality of this algorithm can be determined as follows:

- In step1, the path matrix P is initialized with the adjacency matrix GPtr, which signifies the initial path matrix. This matrix, P [i][j], maintains path which are direct i.e. if there is a direct path from V_i to V_j.
- In step3, the outer most loop will execute N times. For each k, it decides whether there is a path from V_i to V_j (for all i, j = 1, 2, …., N) either directly (or) via k, i.e., from V_i to V_k

and then from V_k to V_j. It therefore sets the P [i][j] entries to 0 (or) 1 accordingly.



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

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 $R^{(1)}$.

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

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 $R^{(2)}$.

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

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 $R^{(3)}$.

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

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 $R^{(4)}$.

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

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).

FIGURE . Application of Warshall's algorithm to the digraph shown. New ones are in bold.