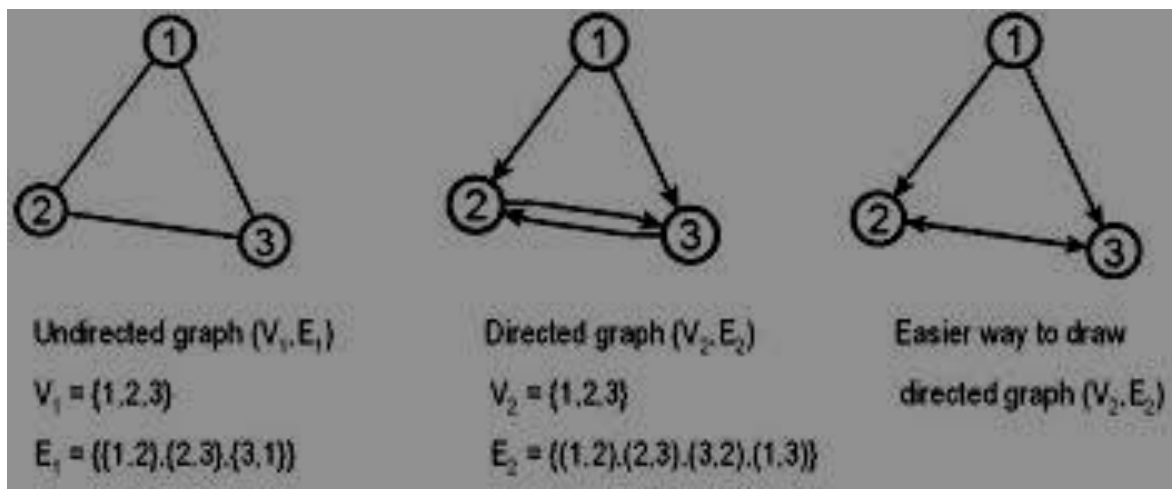
UNIT – III

GRAPHS

Definition

A graph G consists of two sets V and E. V is a finite non-empty set of vertices. E is a set of pairs of vertices, these pairs are called edges. V(G) and E(G) will represent the set of vertices and edges of graph G. G(V,E) to represent a graph.



Undirected Graph

- An undirected graph the pairs of vertices representing any edge is unordered. Thus, the pairs (v1,v2) and (v2,v1) represent the same edge.
- The vertices are represented by circles and the edges by lines.
- An edge with no orientation in our undirected edge.

Directed Graph

- In a directed graph each edge is represented by a directed pair <v1, v2>. V1 is the tail and v2 is the head of the edge. Therefore <v2,v1> and <v1,v2> represent two different edges.
- An edge with an orientation is a directed edge.
- if all the edges are directed; then the graph is a directed graph. A directed graph is also called as digraph.

Complete Graph

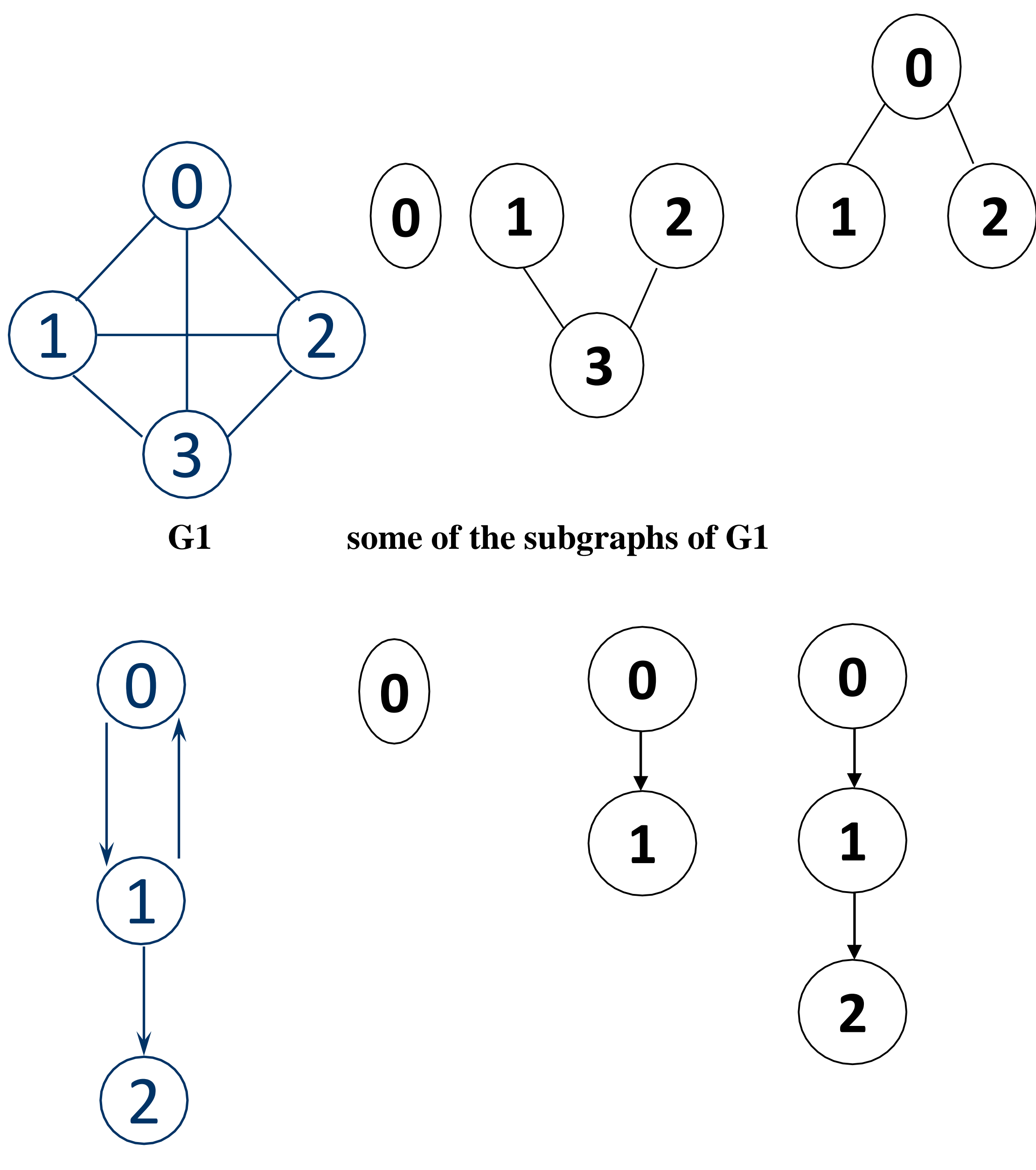
- A complete **graph** is a simple undirected graph in which every pair of distinct vertices is connected by a unique edge.
- therefore, the **complete digraph** is a directed graph in which every pair of distinct vertices is connected by a pair of unique edges (one in each direction).
- The complete graph on n vertices is denoted by K_n. K_n has $n(n-1)/2$ edges and is a regular graph of degree $n-1$.

Adjacent and Incident

- If (v0, v1) is an edge in an undirected graph,
  - v0 and v1 are adjacent
  - The edge (v0, v1) is incident on vertices v0 and v1
- if <v0, v1> is an edge in a directed graph
  - v0 is adjacent **to** v1, and v1 is adjacent **from** v0
  - The edge <v0, v1> is incident on v0 and v1

Subgraph

A graph G = (V1, E1) is called subgraph of a graph G(V, E) if V1(G) is a subset of V(G) and E1(G) is a subset of E(G) such that each edge of G1 has same end vertices as in G.



G1 **some of the subgraphs of G1**

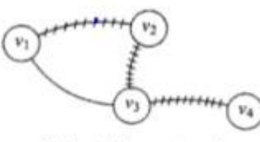
G3 **some of the sub graphs of G3**

Path

A path from vertex vp to vertex vq in a graph G is a sequence of vertices, vp, vi1, vi2, ..., vin, vq, such that (vp, vi1), (vi1, vi2), ..., (vin, vq) are edges in an undirected graph. The length of a path is the number of edges on it.

Simple path

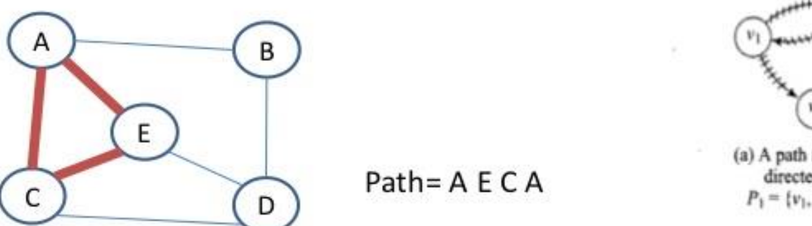
- A graph with no loops or multiple edges is called a simple graph.
- A path with no repeated vertices is called a **simple path**.
- The path from v_1 to v_4 is said to be simple path as vertices is touched more than once.
- The path from v1 to v4 is not simple as v1 is touched twice or looped.



(b) A path from v_1 to v_4 in undirected graph G_2 $P_2 = \{v_1, v_2, v_3, v_4\}$

cycle:

- Simple path, except that the last vertex is the same as the first vertex. Its also known as a **circuit or circular path.**

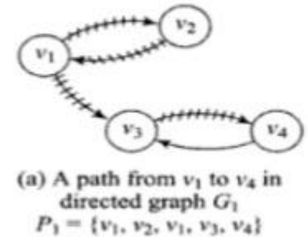


(a) A path from v_1 to v_4 in directed graph G_1 $P_1 = \{v_1, v_2, v_1, v_3, v_4\}$

Connected graph

- Two vertices v_i, v_j in a graph G is said to be connected only if there is a path in G between v_i and v_j.

- A undirected graph is said to be **connected graph** if every pair of distinct vertices vi, vj are connected.

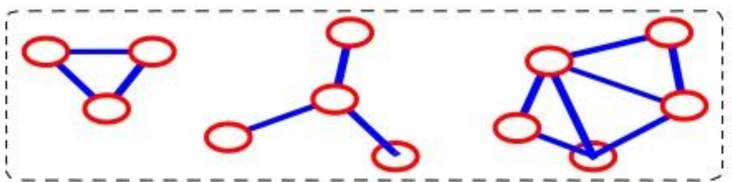


(a) A path from v_1 to v_4 in directed graph G_1 $P_1 = \{v_1, v_2, v_1, v_3, v_4\}$

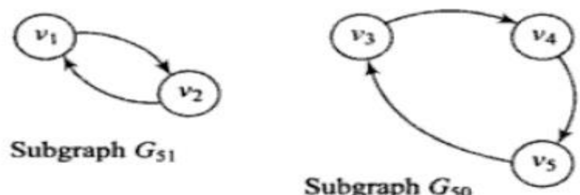
Connected undirected graph

- In the case of an undirected graph, which is not connected,the maximal connected subgraph is called as a **connected component** or simply a **component.**

The graph below has 3 connected components.

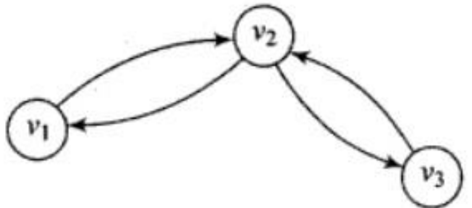


- Strongly connected components of directed graph
- The below graph is not strongly connected but is said to possess two strongly connected components.



Connected directed graph

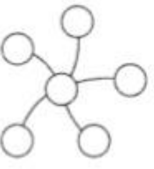
- A directed graph is said to be strongly connected only if every pair of distinct vertices v_i, v_j are connected.
- If there is a directed path from vi to vj then there must be a directed path from v_j to v_i.



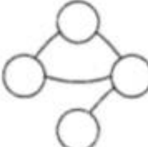
A strongly connected directed graph

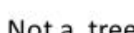
Trees

- A tree is defined to be a connected acyclic graph. The following properties are satisfied by a tree:
    - ➢There exist a path between any two vetices of the tree
    - ➢No cycles must be present in the tree ie, trees are acyclic.



A Tree





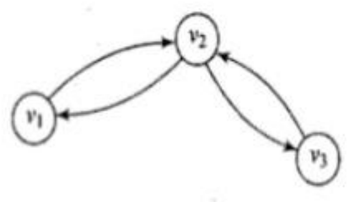
Not a tree

Degree in directed graph

- Degree of directed graph has two types
    1. Indegree

       No of edges with their head towards the vertex.
    2. Outdegree

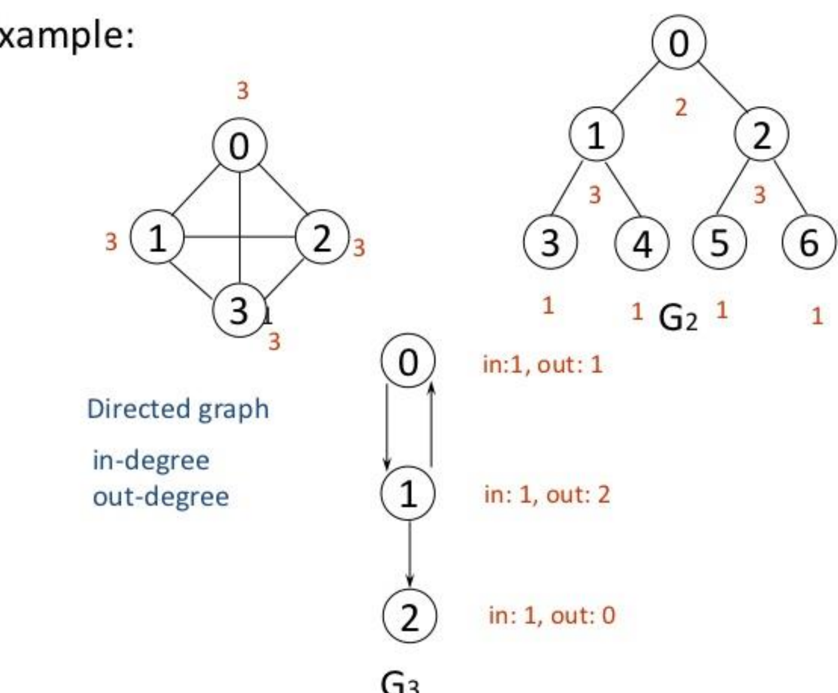
       No of edges with their tail towards the vertex.



Indegree of vertex v_2 is 2

and

Outdegree of vertex v_1 is 1.



ADT for Graph

Structure Graph is

objects: a nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices

functions: for all *graph* ∈ *Graph*, *v*, v_1 and v_2 ∈ *Vertices*

Graph Create()::=return an empty graph

Graph InsertVertex(*graph*, *v*)::= return a graph with *v* inserted. *v* hasno incident edge.

Graph InsertEdge(*graph*, *v1*,*v2*)::= return a graph with newedge between *v1* and*v2*

Graph DeleteVertex(*graph*, *v*)::= return a graph in which *v* and alledges incident to it are removed

Graph DeleteEdge(*graph*, *v1*, *v2*)::=return a graph in which the edge (*v1*, *v2*) is removed

Boolean IsEmpty(*graph*)::= if (*graph*==*empty graph*) returnTRUE else return FALSE

List Adjacent(*graph*,*v*)::= return a list of all vertices that are adjacent to*v*

Graph Representations

In graph theory, a graph representation is a technique to store graph into the memory of computer.

To represent a graph, we just need the set of vertices, and for each vertex the neighbors of the vertex (vertices which is directly connected to it by an edge). If it is a weighted graph, then the weight will be associated with each edge.

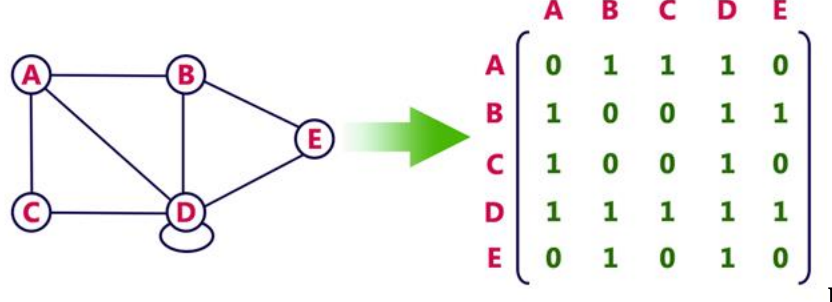
There are different ways to optimally represent a graph, depending on the density of its edges, type of operations to be performed and ease of use.

Adjacency Matrix

- Adjacency matrix is a sequential representation.
- It is used to represent which nodes are adjacent to each other. i.e. is there any edge connecting nodes to a graph.
- In this representation, we have to construct a nXn matrix A. If there is any edge from a vertex i to vertex j, then the corresponding element of A, $a^{i,j} = 1$, otherwise $a^{i,j} = 0$.
- If there is any weighted graph then instead of 1s and 0s, we can store the weight of the edge.

Example

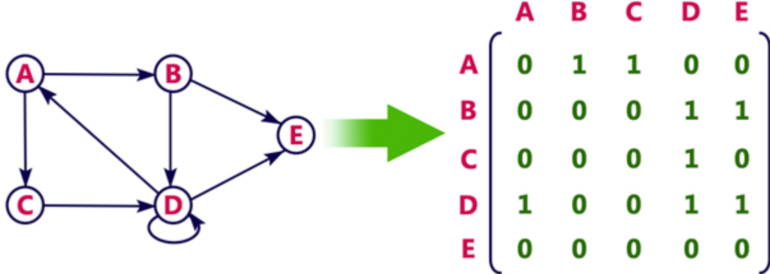
Consider the following **undirected graph representation**:



n

Directed graph represenation

See the directed graph representation:



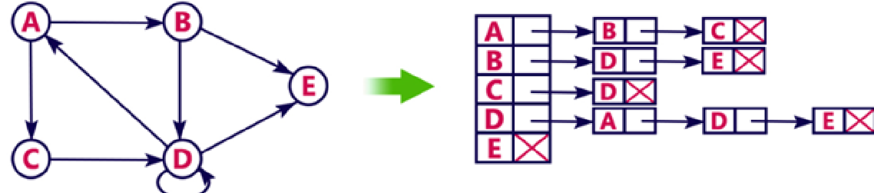
In the above examples, 1 represents an edge from row vertex to column vertex, and 0 represents no edge from row vertex to column vertex.

Adjacency List

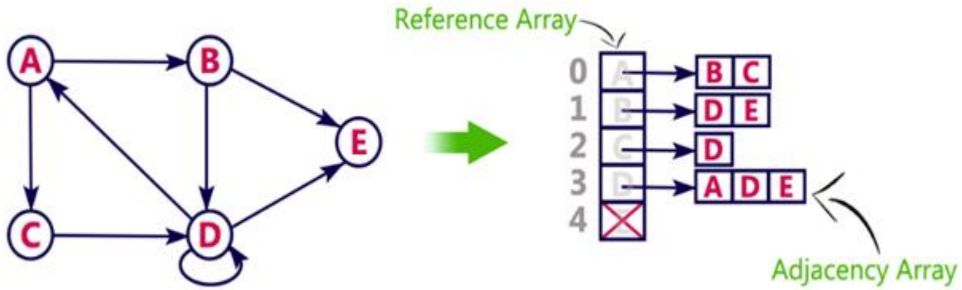
- Adjacency list is a linked representation.
- In this representation, for each vertex in the graph, we maintain the list of its neighbors. It means, every vertex of the graph contains list of its adjacent vertices.
- We have an array of vertices which is indexed by the vertex number and for each vertex v, the corresponding array element points to a **singly linked list** of neighbors of v.

Example

Let's see the following directed graph representation implemented using linked list:



We can also implement this representation using array as follows:



Advantages

- Adjacency list saves lot of space.
- We can easily insert or delete as we use linked list.
- Such kind of representation is easy to follow and clearly shows the adjacent nodes of node.

Disadvantages

- The adjacency list allows testing whether two vertices are adjacent to each other but it is slower to support this operation.

Adjacency MultiLists

- Adjacency Multi-lists are an edge, rather than vertex based, graph representation. In the Multilist representation of graph structures; these are two parts, a directory of Node information and a set of linked list of edge information.
- There is one entry in the node directory for each node of the graph. The directory entry for node i points to a linked adjacency list for node i. each record of the linked list area appears on two adjacency lists: one for the node at each end of the represented edge.

Graph Traversal

Graph traversal is the problem of visiting all the nodes in a graph in a particular manner, updating and/or checking their values along the way. The order in which the vertices are visited may be important, and may depend upon the particular algorithm.

The two common traversals:

- Breadth-first search
- Depth-first search

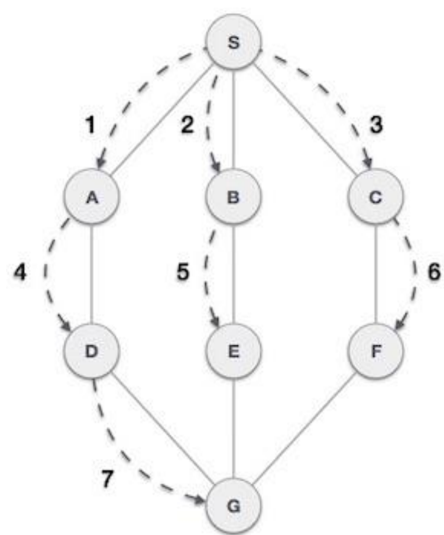
Breadth-first search

- Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, the algorithm explores their unexplored neighbour nodes, and so on, until it finds the goal.
- A breadth-first search (BFS) explores nodes nearest the root before exploring nodes further away.

```
Procedure BFS(v)
    VISITED(v) ← 1
    Initialize Q with vertex v in it
    while Q not empty do
        call DELETEQ(v, Q)
        for all vertices w adjacent to v do
            if VISITED(w) = 0
                then [ call ADDQ(w, Q); VISITED(w) ← 1]
        end
    end
```

end

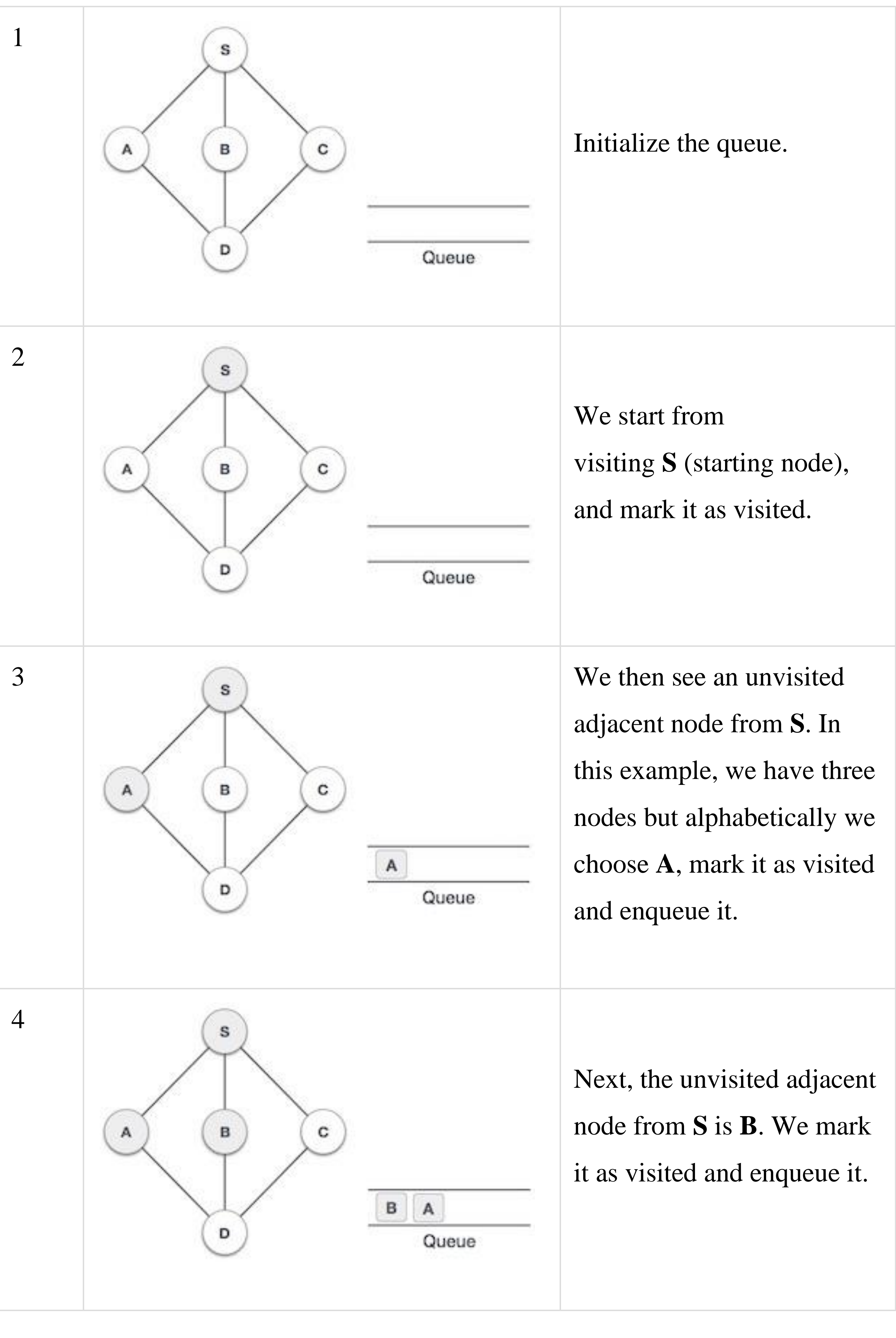
Example



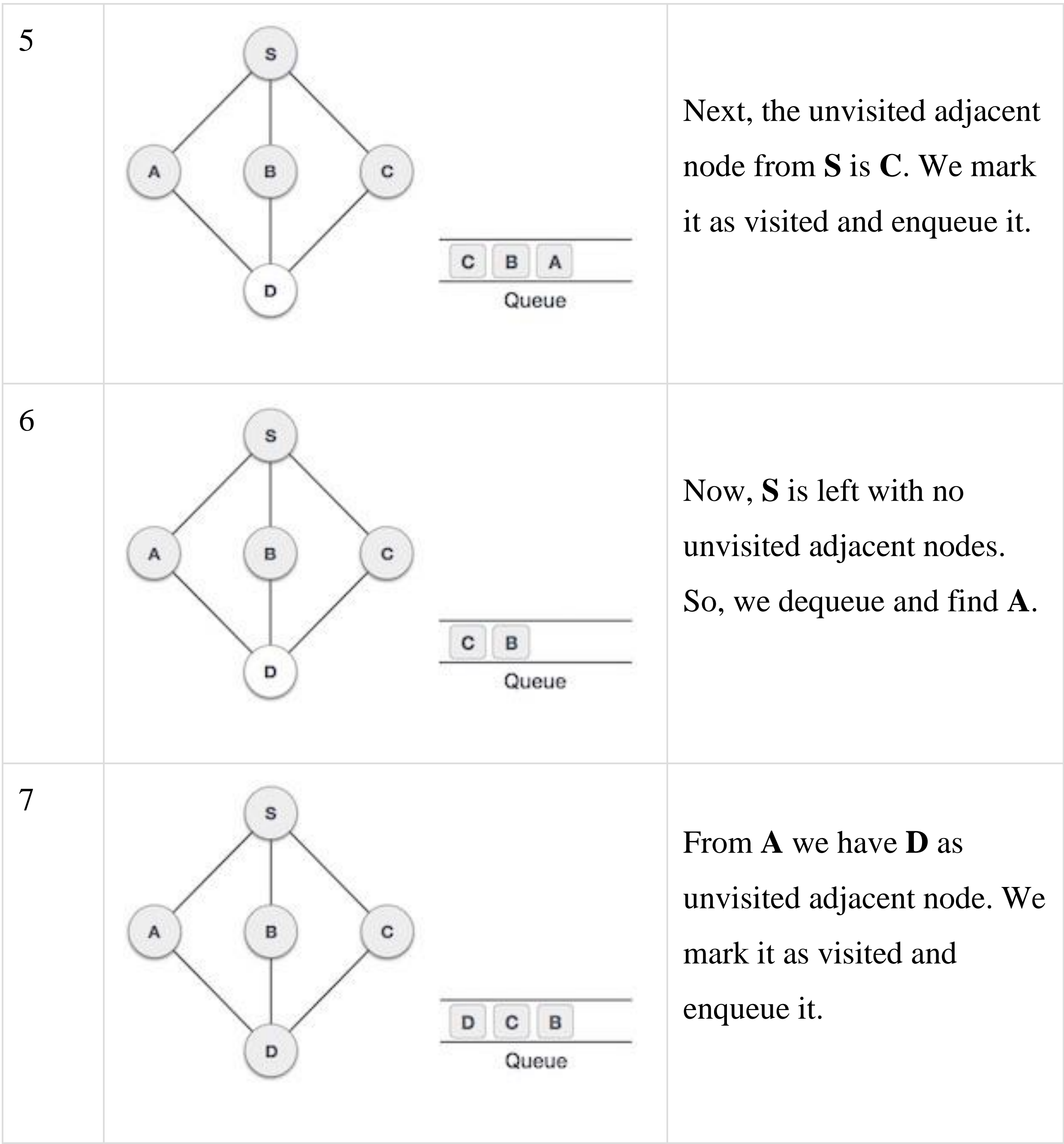
As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** − If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** − Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description



1		Initialize the queue.
2		We start from visiting **S** (starting node), and mark it as visited.
3		We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.
4		Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.



5		Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.
6		Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.
7		From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

Running time of BFS is $O(n^2)$.

Applications of Breadth-First Search Algorithm

Breadth-first search can be used to solve many problems such as:

- Finding all connected components in a graph G.
- Finding all nodes within an individual connected component.
- Finding the shortest path between two nodes, u and v, of an unweighted graph.
- Finding the shortest path between two nodes, u and v, of a weighted graph.

Depth First Search

Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

Procedure DFS (v)

VISITED (v) ⟵ 1

for each vertex w adjacent to v do

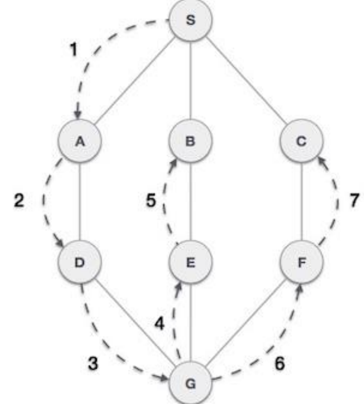
if VISITED(w) = 0 **then call** DFS(w)

end

end

Running time is $O(n^2)$

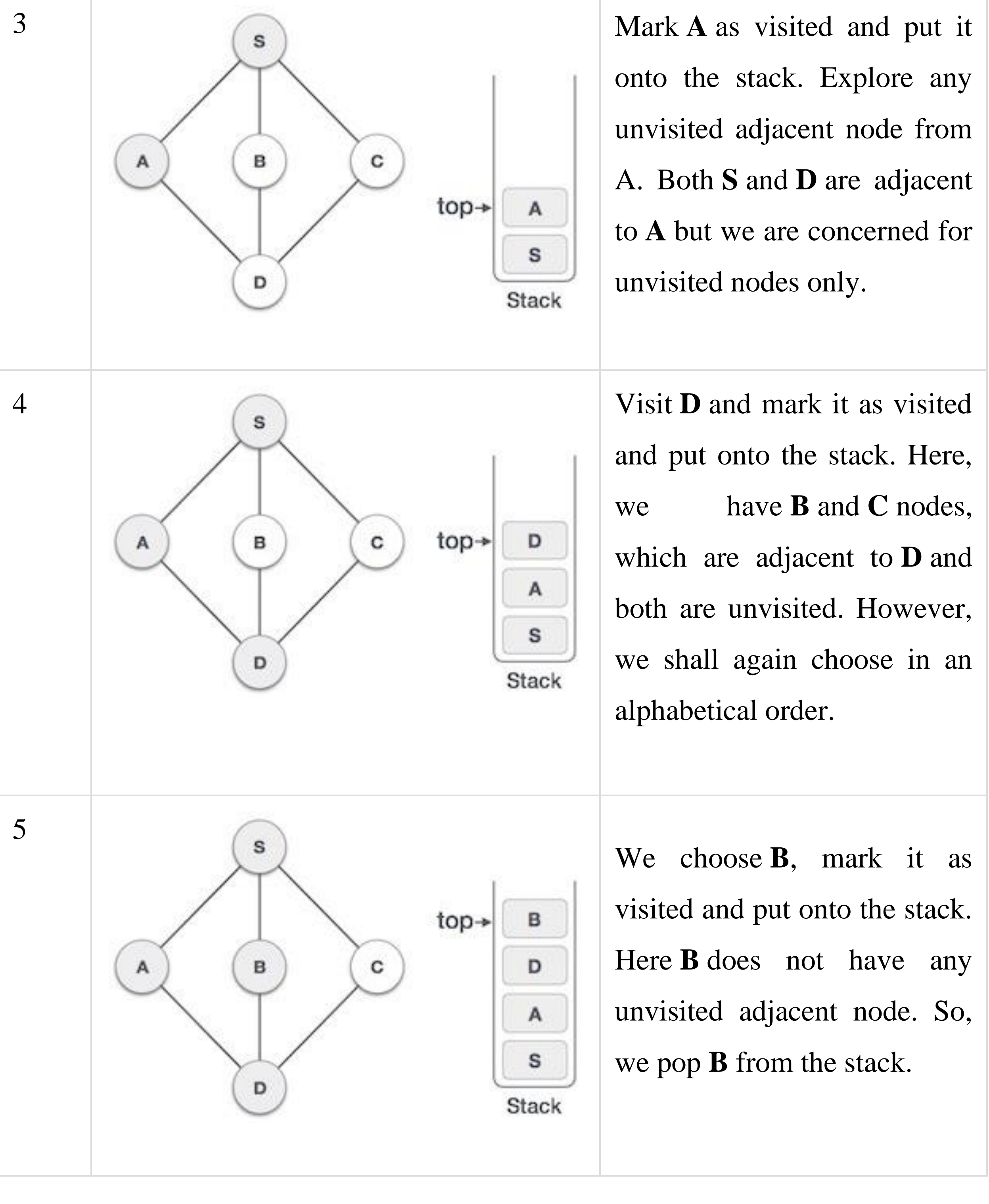
Example



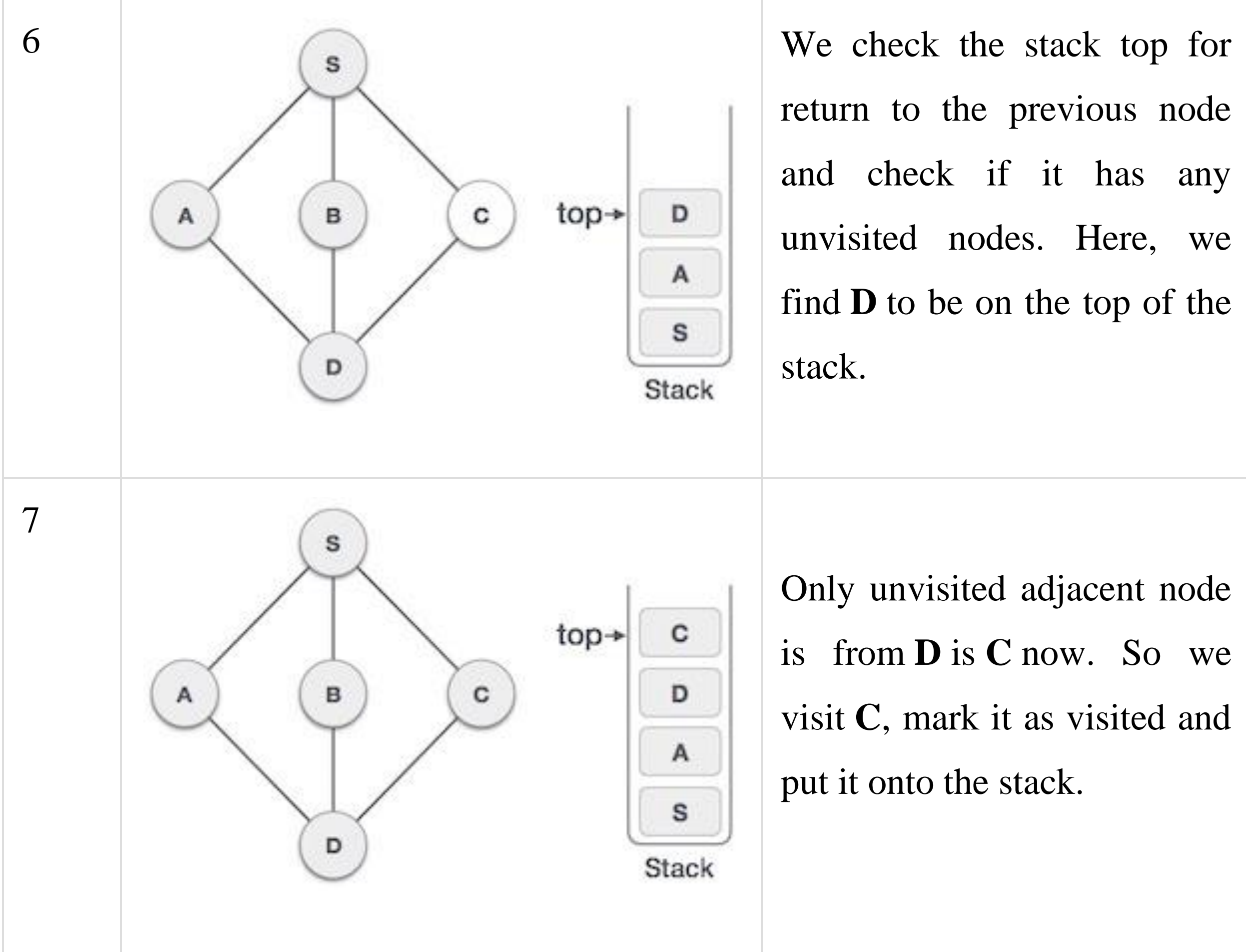
As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

- **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** − Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
1	S A B C D Stack	Initialize the stack.
2	S A B C D top→ S Stack	Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.



3		Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.
4		Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.
5		We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.



6		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.
7		Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

Applications of Depth-First Search Algorithm

Depth-first search is useful for:

- Finding a path between two specified nodes, u and v, of an unweighted graph.
- Finding a path between two specified nodes, u and v, of a weighted graph.
- Finding whether a graph is connected or not.
- Computing the spanning tree of a connected graph.

Connected Components

- If G is an undirected graph, then one can determine whether or not it is connected by simply making a call to either DFS or BFS and then determining if there is any unvisited vertex.

- $O(n^2)$ If adjacency matrices are used and O(e) if adjacency lists are used.

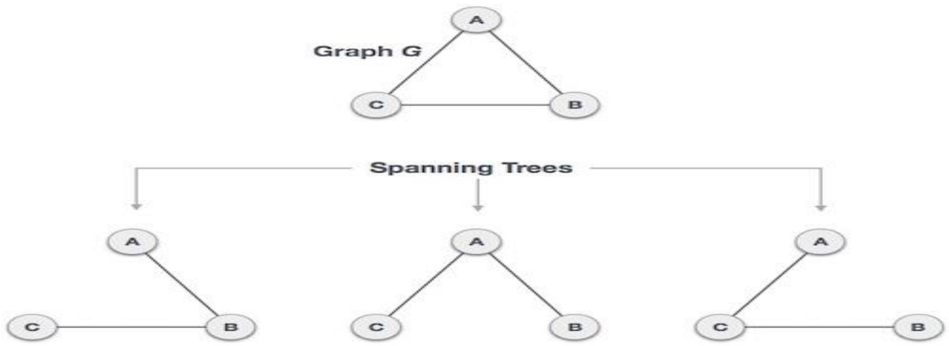
- Algorithm COMP which determines all the connected components of G.
- The algorithm uses DFS.

Procedure COMP (G, n)
  for i ⟵ 1 **to** n **do**
    VISITED (i) ⟵ 0
  end
  for i ⟵ 1 **to** n **do**
    if VISITED (i) ⟵ 0 **then [**call DFS(i); output all newly visited vertices together with all edges incident to them]
  end
end COMP

- if G is represented by its adjacency lists, then the total time taken by DFS is O(e). The output can be completed in time O(e) if DFS keeps a list of all newly visited vertices.
- The total time to generate all the connected components is O (n+e).

Spanning Tree

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



The spanning tree resulting from a call to DFS is known as a depth first spanning tree. When BFS is used, the resulting spanning tree is called as breadth first spanning tree.

Minimum Spanning Tree (MST)

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

Kruskal's 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.

T ⟵ 0

while T contains less than n-1 edges **do**

    choose an edge (v,w) from E of lowest cost;

    delete (v,w) from E;

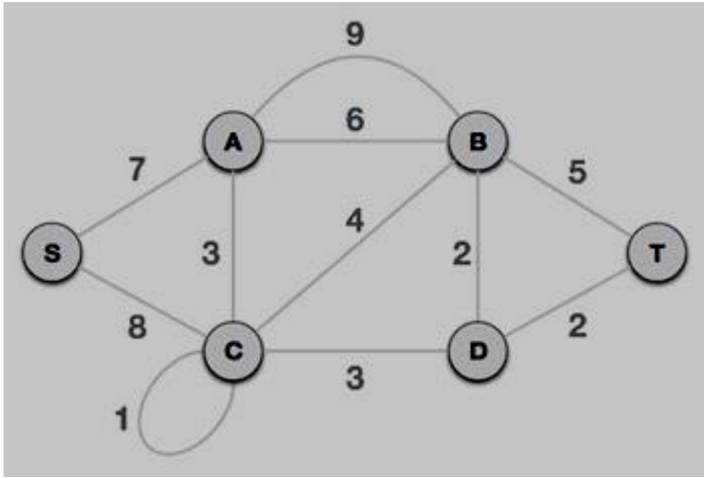
    if(v,w) does not create a cycle in T

        then add(v,w) to T

    else discard(v,w)

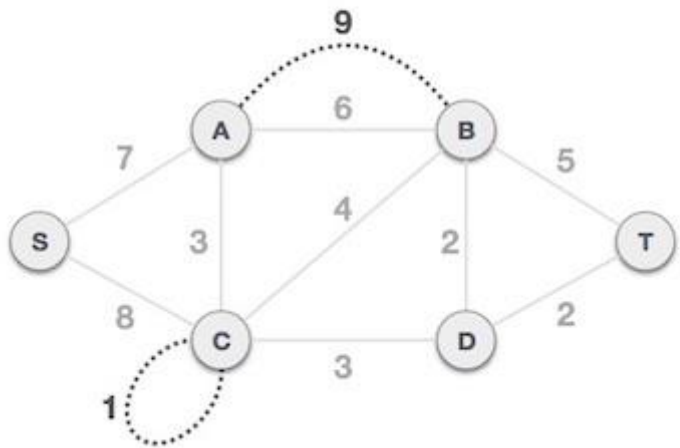
end

Example

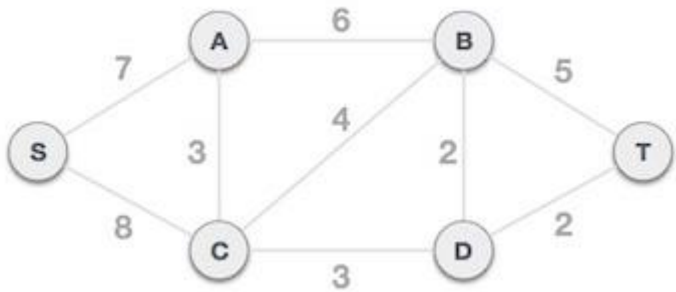


Step 1 - Remove all loops and Parallel Edges

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.



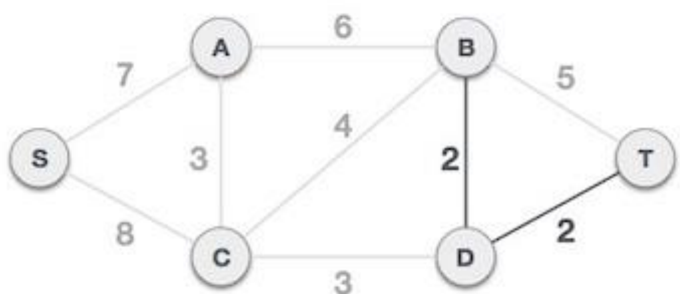
Step 2 - Arrange all edges in their increasing order of weight

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

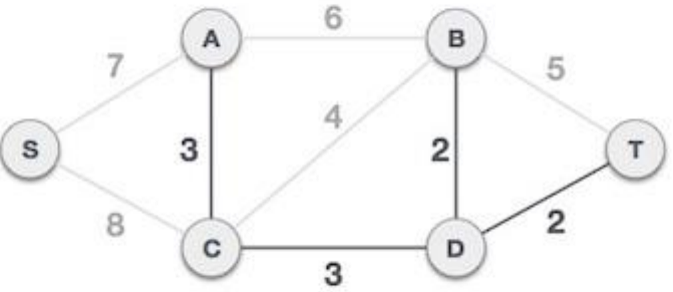
Step 3 - Add the edge which has the least weightage

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.

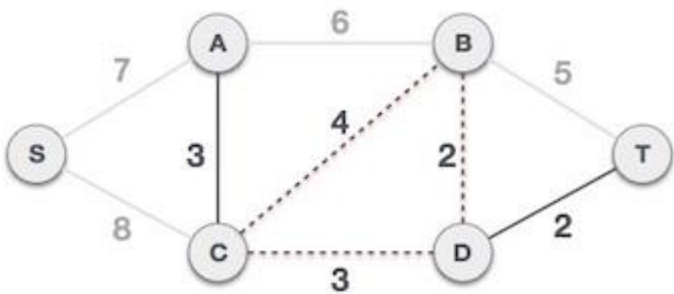


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

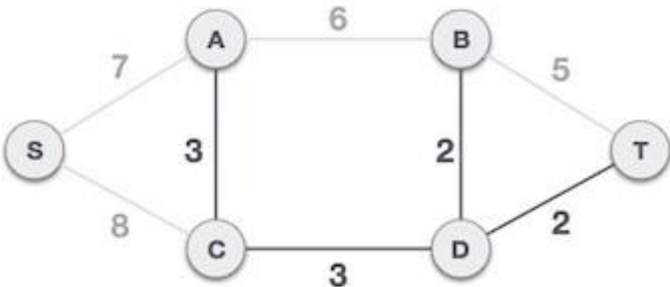
Next cost is 3, and associated edges are A,C and C,D. We add them again −



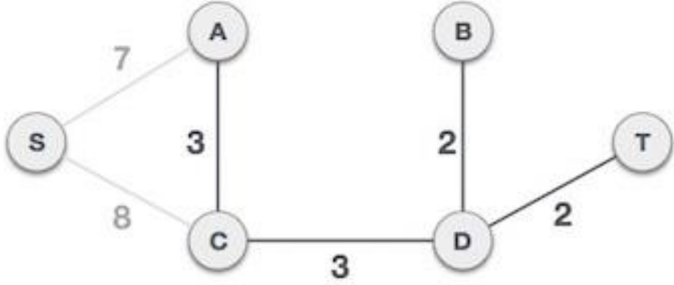
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. −



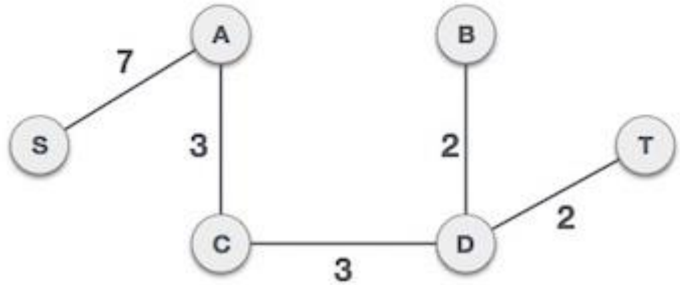
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



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



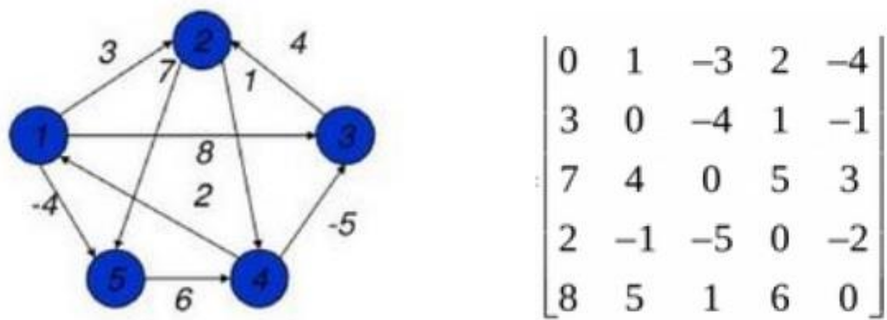
Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



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

All Pairs Shortest Paths

- The all pair shortest path algorithm is also known as Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph.
- As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.



$$\begin{bmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

- At first the output matrix is same as given cost matrix of the graph. After that the output matrix will be updated with all vertices k as the intermediate vertex.
- The time complexity of this algorithm is O(V3), here V is the number of vertices in the graph.
- The graph G is represented by its cost adjacency matrix with COSt(i,i)=0 and COST(i,j)=+∞ in case edge <i, j>, i≠j is not in G

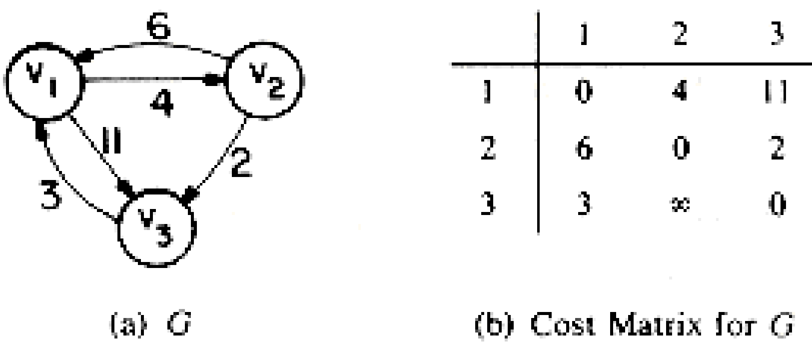
- $A^k(i,j)$ to be the cost of the shortest path from i to j going through no intermediate vertex of index greater than k.
- All pairs shortest algorithm is to successively generate the matrices A^0, A^1, A^2,..., A^n.
- To generate A^k by realizing that for any pair of vertices i, j either
    1. The shortest paths from i to j going through no vertex with index greater than k does not go through the vertex with index k and so its cost is $A^{k-1}(i,j)$

       $A^k(i,j) = \min\{A^{k-1}(i,j), A^{k-1}(i,k)+A^{k-1}(k,j)\}$, $k \geq 1$ and

       $A^0(i,j) = COST(i,j)$

```
Procedure ALL_COSTS(COST, A, n)
for i := 1 to n, do
      for j := 1 to n, do
            A(i,j) ←——  COST(i,j)
      end
end
for k := 1 to n, do
      for i := 1 to n, do
         for j := 1 to n, do
                 A(i,j) = min{A(i,j), A(i,k)+A(k,j)}
                 end
         end
end
```

Example



	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

(a) G

(b) Cost Matrix for G

A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

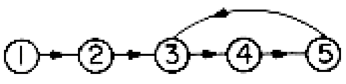
A^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

A^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

A^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

Transitive Closure

- A problem related to the all pairs shortest path problem is that of determining for every pair of vertices *i,j* in *G* the existence of a path from *i* to *j*.
- Two cases are of interest, one when all path lengths (i.e., the number of edges on the path) are required to be positive and the other when path lengths are to be nonnegative.
- If *A* is the adjacency matrix of *G*, then the matrix A^+ having the property $A^+(i,j) = 1$ if there is a path of length > 0 from *i* to *j* and 0 otherwise is called the *transitive closure* matrix of *G*.
- The matrix A^* with the property $A^*(i,j) = 1$ if there is a path of length ≥ 0 from *i* to *j* and 0 otherwise is the *reflexive transitive closure* matrix of *G*.



(a) Digraph *G*

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

(b) Adjacency Matrix A for G

	1	2	3	4	5
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

(c) A^+

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

(d) A^*

Graph G and Its Adjacency Matrix A, A^+ and A^*

- The only difference between A^* and A^+ is in the terms on the diagonal. $A^+(i,i) = 1$ iff there a cycle of length > 1 containing vertex i while $A^*(i,i)$ is always one as there is a path of length 0 from i to i.
- If we use algorithm ALL_COSTS with COST$(i,j) = 1$ if $<i,j>$ is an edge in G and COST$(i,j) = +\infty$ if $<i,j>$ is not in G, then we can easily obtain A^+ from the final matrix A by letting $A^+(i,j) = 1$ iff $A(i,j) < +\infty$. A^* can be obtained from A^+ by setting all diagonal elements equal 1.

Single Source All Destinations

- In a **Single Source Shortest Paths Problem**, we are given a Graph G = (V, E), we want to find the shortest path from a given source vertex s ∈ V to every vertex v ∈ V.
- Let *S* denote the set of vertices (including v_o) to which the shortest paths have already been found. For *w* not in *S*, let DIST(*w*) be the length of the shortest path starting from v_o going through only those vertices which are in *S* and ending at w.
- If the next shortest path is to vertex *u*, then the path begins at v_o, ends at *u* and goes through only those vertices which are in *S*.
- The destination of the next path generated must be that vertex *u* which has the minimum distance, DIST(*u*), among all vertices not in *S*.
- Having selected a vertex *u* as in (ii) and generated the shortest v_o to *u* path, vertex *u* becomes a member of *S*. At this point the length of the shortest paths starting at v_o, going through vertices only in *S* and ending at a vertex *w* not in *S*
- a path from v_o to *u* to *w* where the path from v_o to *u* is the shortest such path and the path from *u* to *w* is the edge <*u,w*>. *The length of this path is DIST(*u*) + length (*<u,w>)

```
procedure SHORTEST-PATH (v,COST,DIST,n)
declare S (1: n)
1    for i ← 1 to n do    //initialize set S to empty//
2      S(i) ← 0; DIST(i) ← COST(v,i)
3    end
4    S(v) ← 1; DIST(v) ← 0; num ← 2      //put vertex v in set
S//
5    while num < n do    //determine n - 1 paths from vertex v//
6      choose u: DIST(u) = min {DIST(w)}
S(w)=0
7      S(u) ← 1; num ← num + 1    //put vertex u in set S//
8      for all w with S(w) = 0 do  //update distances//
```

9 $\quad$ *DIST(w) ← min {DIST(w),DIST(u) + COST(u,w)}*
10 $\quad$ **end**
11 $\quad$ **end**
12 **end** *SHORTEST-PATH*

Example

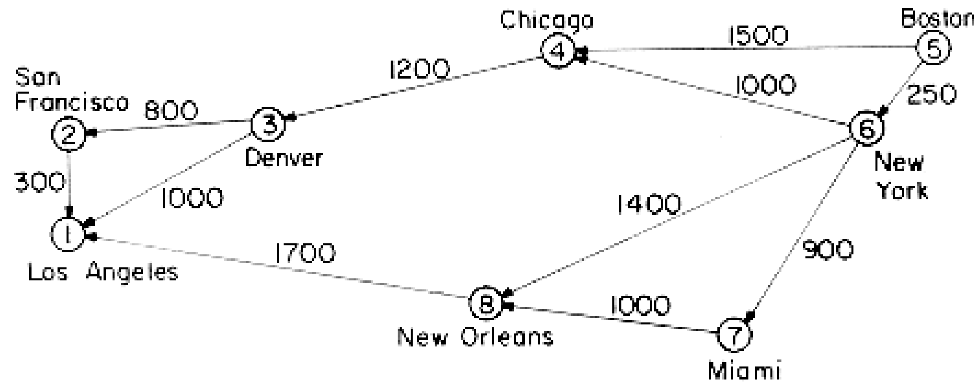
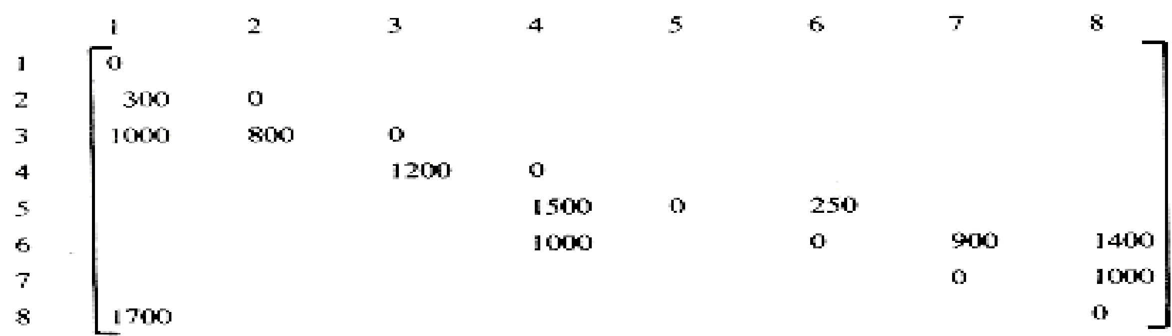


Figure (a)



	1	2	3	4	5	6	7	8
1	0							
2	300	0						
3	1000	800	0					
4			1200	0				
5				1500	0	250		
6				1000		0	900	1400
7							0	1000
8	1700							0

Figure (b) Cost Adjacency Matrix for Figure (a).

		Vertex	LA	SF	D	C	B	NY	M	NO
Iteration	S	Selected	**(1)**	**(2)**	**(3)**	**(4)**	**(5)**	**(6)**	**(7)**	**(8)**
Initial		--	$+\infty$	$+\infty$	$+\infty$	1500	0	250	$+\infty$	$+\infty$
1	5	6	$+\infty$	$+\infty$	$+\infty$	1250	0	250	1150	1650
2	5,6	7	$+\infty$	$+\infty$	$+\infty$	1250	0	250	1150	1650
3	5,6,7	4	$+\infty$	$+\infty$	2450	1250	0	250	1150	1650
4	5,6,7,4	8	3350	$+\infty$	2450	1250	0	250	1150	1650
5	5,6,7,4,8	3	3350	3250	2450	1250	0	250	1150	1650
6	5,6,7,4,8,3	2	3350	3250	2450	1250	0	250	1150	1650
	5,6,7,4,8,3,2									

Action of SHORTEST_PATH