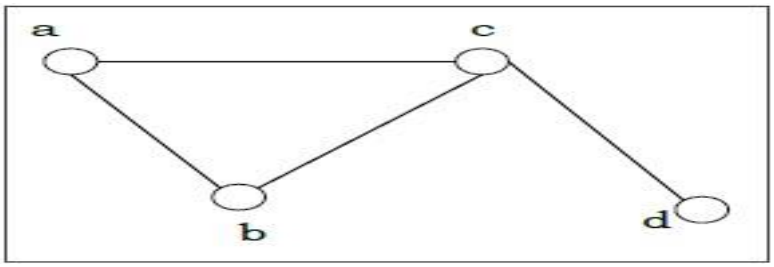
UNIT-V
Graphs

Basic Concepts,

Graph Theory

Definition − A graph (denoted as G = (V, E)) consists of a non-empty set of vertices or nodes V and a set of edges E. A vertex a represents an endpoint of an edge. An edge joins two vertices **a,** and **b** is represented by set of vertices it connects.

Example − Let us consider, a Graph is G = (V, E) where V = {a, b, c, d} and E = {{a, b}, {a, c}, {b, c}, {c, d}}



Degree of a Vertex − The degree of a vertex V of a graph G (denoted by deg (V)) is the number of edges incident with the vertex V.

Vertex	Degree	Even / Odd
a	2	even
b	2	even
c	3	odd
d	1	odd

Even and Odd Vertex − If the degree of a vertex is even, the vertex is called an even vertex and if the degree of a vertex is odd, the vertex is called an odd vertex.

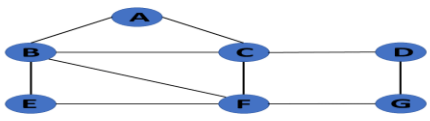
Degree of a Graph − The degree of a graph is the largest vertex degree of that graph. For the above graph the degree of the graph is 3.

Types of Graphs in Data Structures

There are different types of graphs in data structures, each of which is detailed below.

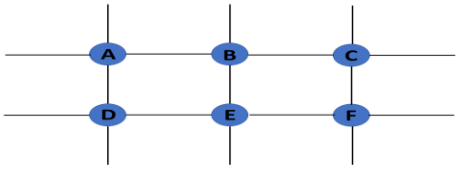
1. Finite Graph

The graph G=(V, E) is called a finite graph if the number of vertices and edges in the graph is limited in number



2. Infinite Graph

The graph G=(V, E) is called a finite graph if the number of vertices and edges in the graph is interminable.

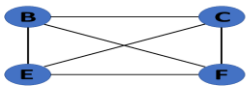


3. Trivial Graph

A graph G= (V, E) is trivial if it contains only a single vertex and no edges.

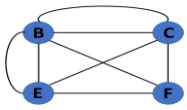
4. Simple Graph

If each pair of nodes or vertices in a graph G=(V, E) has only one edge, it is a simple graph. As a result, there is just one edge linking two vertices, depicting one-to-one interactions between two elements.



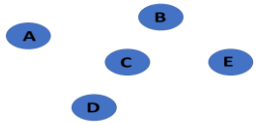
5. Multi Graph

If there are numerous edges between a pair of vertices in a graph G= (V, E), the graph is referred to as a multigraph. There are no self-loops in a Multigraph.



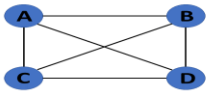
6. Null Graph

It's a reworked version of a trivial graph. If several vertices but no edges connect them, a graph G= (V, E) is a null graph.



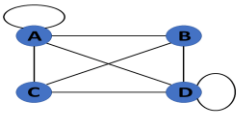
7. Complete Graph

If a graph G= (V, E) is also a simple graph, it is complete. Using the edges, with n number of vertices must be connected. It's also known as a full graph because each vertex's degree must be n-1.



8. Pseudo Graph

If a graph G= (V, E) contains a self-loop besides other edges, it is a pseudograph.

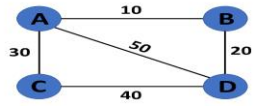


9. Regular Graph

If a graph G= (V, E) is a simple graph with the same degree at each vertex, it is a regular graph. As a result, every whole graph is a regular graph.

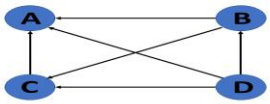
10. Weighted Graph

A graph G= (V, E) is called a labeled or weighted graph because each edge has a value or weight representing the cost of traversing that edge.



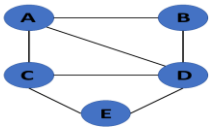
11. Directed Graph

A directed graph also referred to as a digraph, is a set of nodes connected by edges, each with a direction.



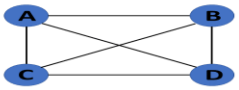
12. Undirected Graph

An undirected graph comprises a set of nodes and links connecting them. The order of the two connected vertices is irrelevant and has no direction. You can form an undirected graph with a finite number of vertices and edges.



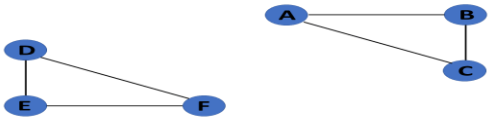
13. Connected Graph

If there is a path between one vertex of a graph data structure and any other vertex, the graph is connected.



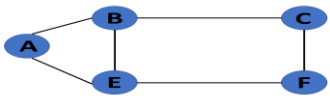
14. Disconnected Graph

When there is no edge linking the vertices, you refer to the null graph as a disconnected graph.



15. Cyclic Graph

If a graph contains at least one graph cycle, it is considered to be cyclic.

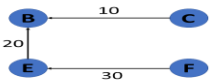


16. Acyclic Graph

When there are no cycles in a graph, it is called an acyclic graph.

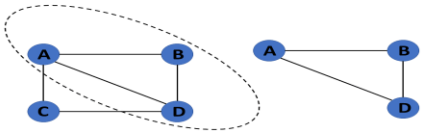
17. Directed Acyclic Graph

It's also known as a directed acyclic graph (DAG), and it's a graph with directed edges but no cycle. It represents the edges using an ordered pair of vertices since it directs the vertices and stores some data.



18. Subgraph

The vertices and edges of a graph that are subsets of another graph are known as a subgraph.



After you learn about the many types of graphs in graphs in data structures, you will move on to graph terminologies.

Terminologies of Graphs in Data Structures

Following are the basic terminologies of graphs in data structures:

1. An edge is one of the two primary units used to form graphs. Each edge has two ends, which are vertices to which it is attached.
2. If two vertices are endpoints of the same edge, they are adjacent.
3. A vertex's outgoing edges are directed edges that point to the origin.
4. A vertex's incoming edges are directed edges that point to the vertex's destination.

5. The total number of edges occurring to a vertex in a graph is its degree.
6. The out-degree of a vertex in a directed graph is the total number of outgoing edges, whereas the in-degree is the total number of incoming edges.
7. A vertex with an in-degree of zero is referred to as a source vertex, while one with an out-degree of zero is known as sink vertex.
8. An isolated vertex is a zero-degree vertex that is not an edge's endpoint.
9. A path is a set of alternating vertices and edges, with each vertex connected by an edge.
10. The path that starts and finishes at the same vertex is known as a cycle.
11. A path with unique vertices is called a simple path.
12. For each pair of vertices x, y, a graph is strongly connected if it contains a directed path from x to y and a directed path from y to x.
13. A directed graph is weakly connected if all of its directed edges are replaced with undirected edges, resulting in a connected graph. A weakly linked graph's vertices have at least one out-degree or in-degree.
14. A tree is a connected forest. The primary form of the tree is called a rooted tree, which is a free tree.
15. A spanning subgraph that is also a tree is known as a spanning tree.
16. A connected component is the unconnected graph's most connected subgraph.
17. A bridge, which is an edge of removal, would sever the graph.
18. Forest is a graph without a cycle.

Representations of Graphs

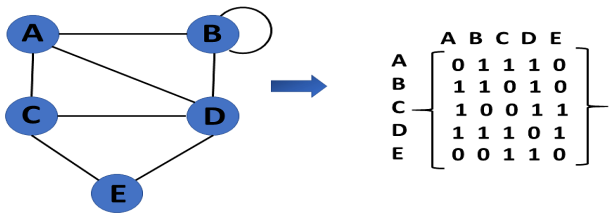
The most frequent graph representations are the two that follow:

1. Adjacency matrix
2. Adjacency list

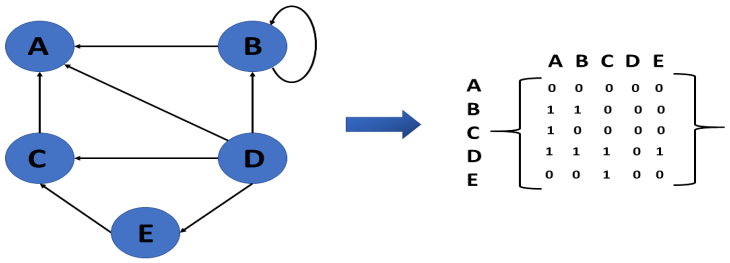
Adjacency Matrix

- A sequential representation is an adjacency matrix.
- It's used to show which nodes are next to one another. I.e., is there any connection between nodes in a graph?
- You create an MXM matrix G for this representation. If an edge exists between vertex a and vertex b, the corresponding element of G, gi,j = 1, otherwise gi,j = 0.
- If there is a weighted graph, you can record the edge's weight instead of 1s and 0s.

Undirected Graph Representation

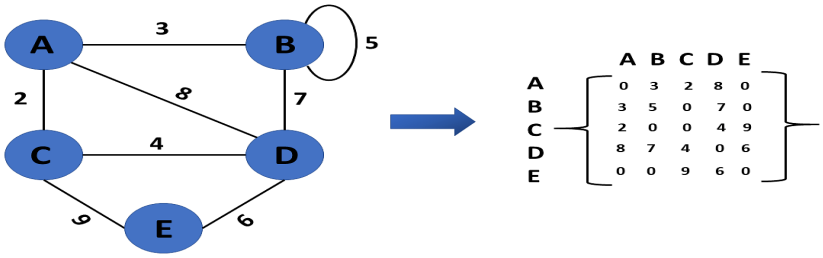


Directed Graph Representation



Weighted Undirected Graph Representation

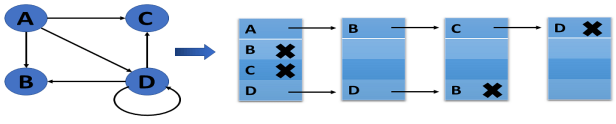
Weight or cost is indicated at the graph's edge, a weighted graph representing these values in the matrix.



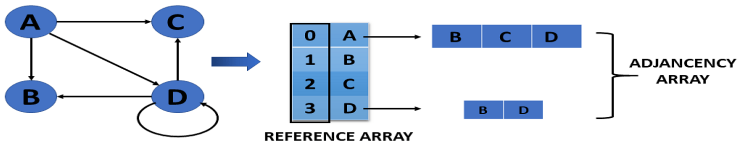
Adjacency List

- A linked representation is an adjacency list.
- You keep a list of neighbors for each vertex in the graph in this representation. It means that each vertex in the graph has a list of its neighboring vertices.
- You have an array of vertices indexed by the vertex number, and the corresponding array member for each vertex x points to a singly linked list of x's neighbors.

directed Graph Representation Using Linked-List



Weighted directed Graph Representation Using an Array



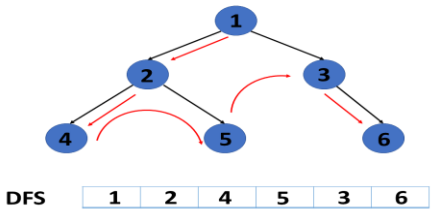
Graph Traversal Algorithm

The process of visiting or updating each vertex in a graph is known as graph traversal. The sequence in which they visit the vertices is used to classify such traversals. Graph traversal is a subset of tree traversal.

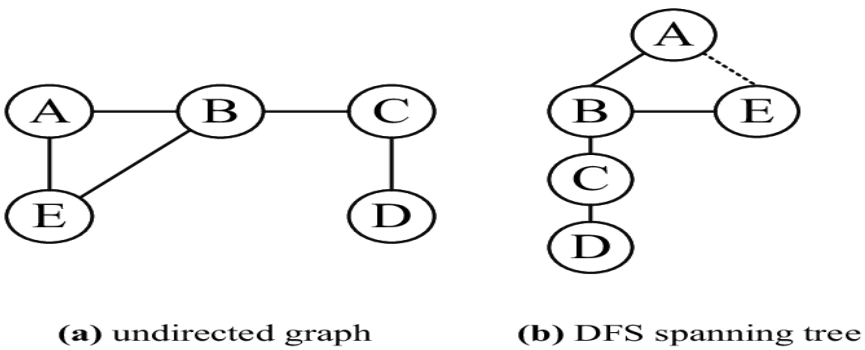
There are two techniques to implement a graph traversal algorithm:

- Breadth-first search
- Depth-first search

Depth first search:



Example:

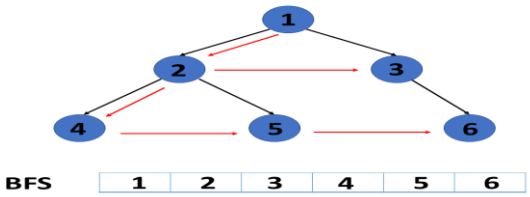


(a) undirected graph **(b)** DFS spanning tree

```
Algorithm DFS(G)
{
        Mark each vertex in V with '0' as a mark of being "unvisited";
        count =0;
        for (each vertex 'v' in V) do
        {
                if (v is marked with '0') then
                        dfs(v);
        }
}

Algorithm dfs(v)
{
        count = count+1;
        mark v with count;
        for (each vertex 'w' in V adjacent to 'v' ) do
        {
                if ('w' is marked with 0) then
                        dfs(w);
        }
}
```

Breadth first search



Example:

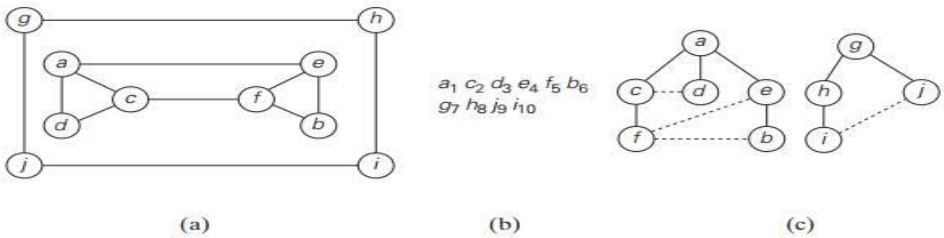


FIGURE 3.11 Example of a BFS traversal. (a) Graph. (b) Traversal queue, with the numbers indicating the order in which the vertices are visited, i.e., added to (and removed from) the queue. (c) BFS forest with the tree and cross edges shown with solid and dotted lines, respectively.

```
Algorithm BFS(G)
{
        Mark each vertex in V with '0' as a mark of being "unvisited";
        count =0;
        for (each vertex 'v' in V) do
        {
                if ('v' is marked with 0) then
                        bfs(v);
        }
}
Algorithm bfs(v)
{
        count = count+1;
        mark v with count and initialize a queue with v;
        while(the queue is not empty) do
        {
                for (each vertex 'w' in V adjacent to the front vertex) do
                {
                        if (w is marked with 0) then
                        {
                                count = count+1;
                                mark 'w' with count;
                                add 'w' to the queue;
                        }
                }
                Remove the front vertex from the queue;
        }
}
```

Applications

MINIMUM COST SPANNING TREE

TREE:
A Tree of a undirected connected graph G = (V, E) is its connected acyclic sub graph.

WEIGHTED GRAPH:
A collection of vertices , edges and also weights on the edges then the graph is said to be weighted graph.

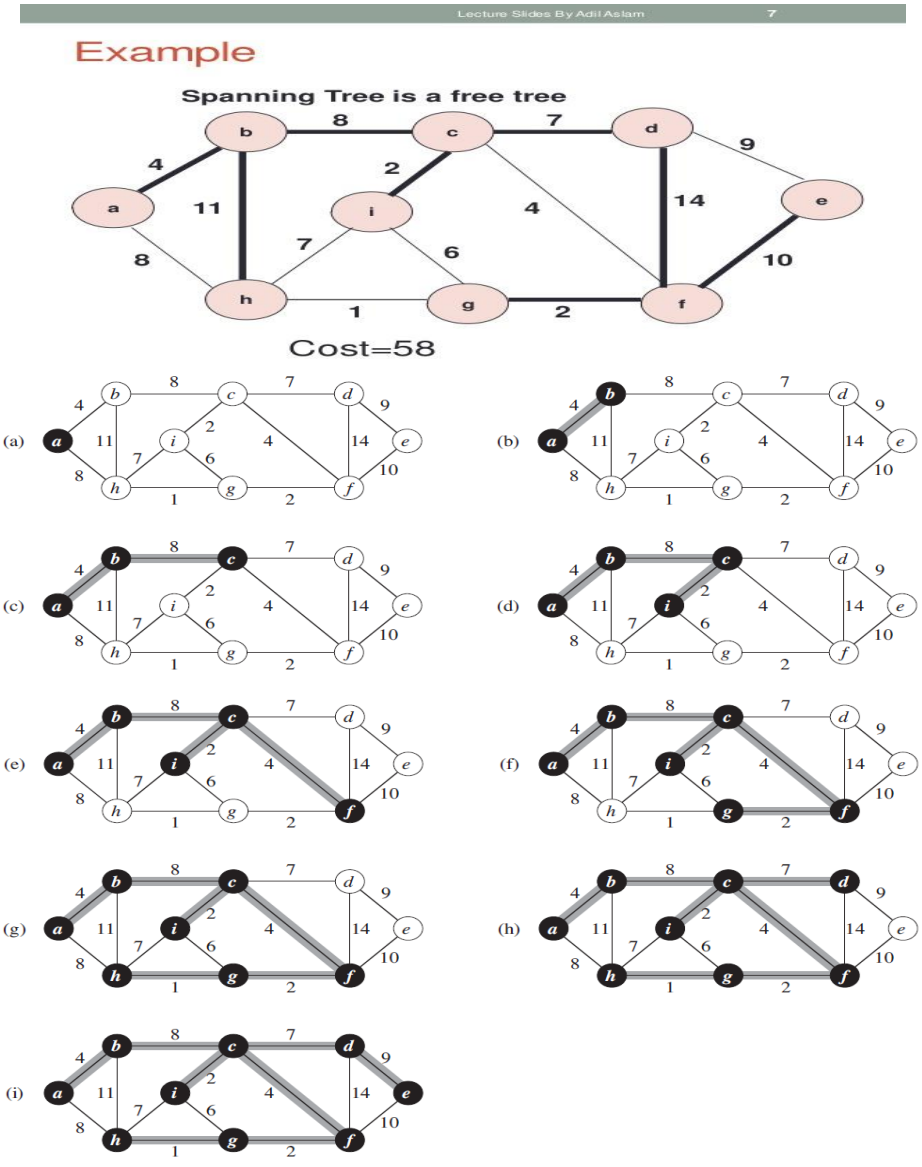
SPANNING TREE:
Any tree consisting of all vertices of a graph , then it is called a spanning tree.

MINIMUM COST SPANNING TREE:
A spanning tree with minimum cost is called minimum cost spanning tree.

To find the minimum cost spanning tree we will use two standard algorithms.

- Prim's Algorithm
- krushkal's Algorithm

PRIMS ALGORITHM:
Step 1: Take minimum cost edges from all the edges
Step 2: Take another edge having minimum weight among those vertices which are adjacent to previous edge.
Step 3: Repeat the above process until spanning tree contains (n – 1) edges.



Tree vertices	Remaining Vertices
a(-, -)	b(a, 4), c(-, ∞), d(-,∞), e(-,∞), f(-,∞), g(-,∞), h(a, 8), i(-,∞)
b(a, 4)	c(b, 8), d(-,∞), e(-,∞), f(-,∞), g(-,∞), h(a, 8), i(-,∞)
c(b, 8)	d(c,7), e(-,∞), f(c,4), g(-,∞), h(a, 8), i(c,2)
i(c, 2)	d(c,7), e(-,∞), f(c,4), g(i,6), h(i, 7)
f(c, 4)	d(c,7), e(f,10), g(f, 2), h(i, 7)
g(f, 2)	d(c,7), e(f,10), h(g, 1)
h(g, 1)	d(c,7), e(f,10)
d(c, 7)	e(d, 9)
e(d, 9)	

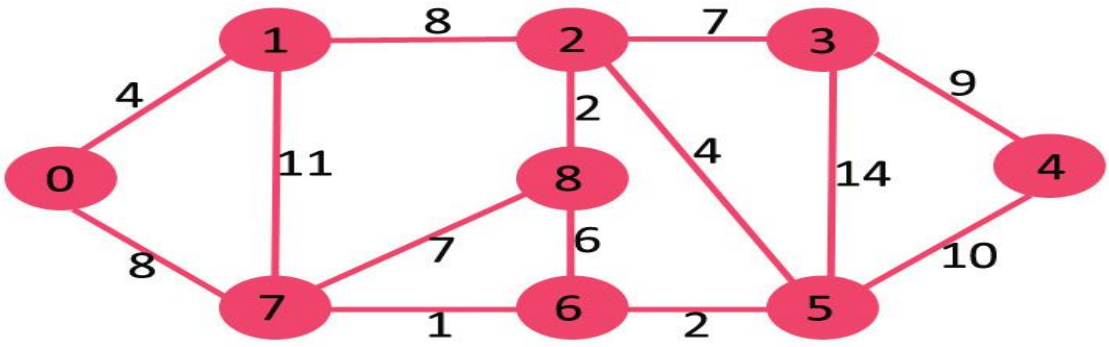
Total cost = 4 + 8 + 2 + 4 + 2 + 1 + 7 + 9 = 37.

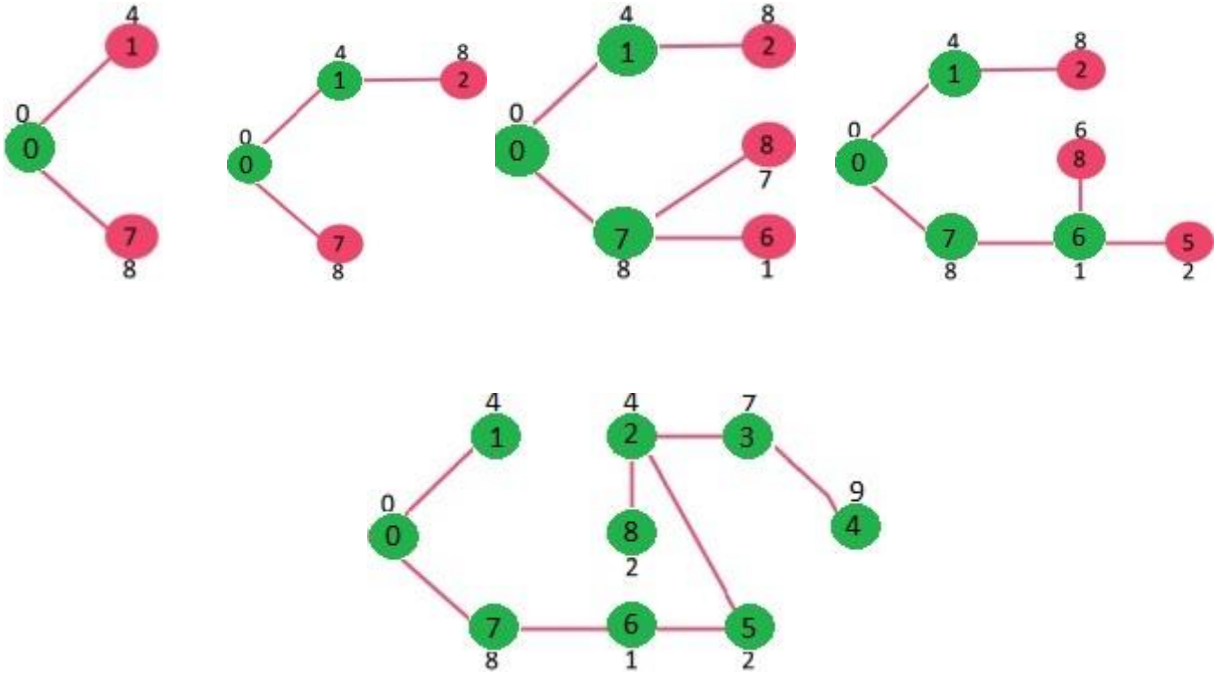
Algorithm:

```
ALGORITHM PRIMS ( G ):
//INPUT: A weighted connected graph G = <V , E>
//OUTPUT: T minimum spanning tree of G
{
        V = {V0};
        T = ∅;                  // empty graph
        for i = 1 to n – 2
        {

                choose nearest neighbor Vj   of  V that is adjacent to Vi
                Vi belongs to V and for which edge e i j = (Vi ,Vj  ) does not form a
                cycle with members of T;
                V = V union { Vj }
                T =  T union { e i j }
        }
        return T;
}
```

Let us understand with the following example:

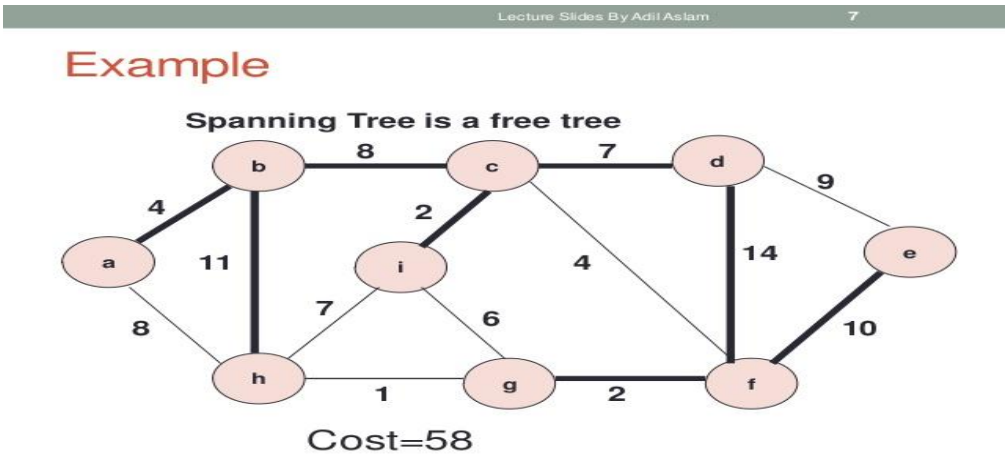


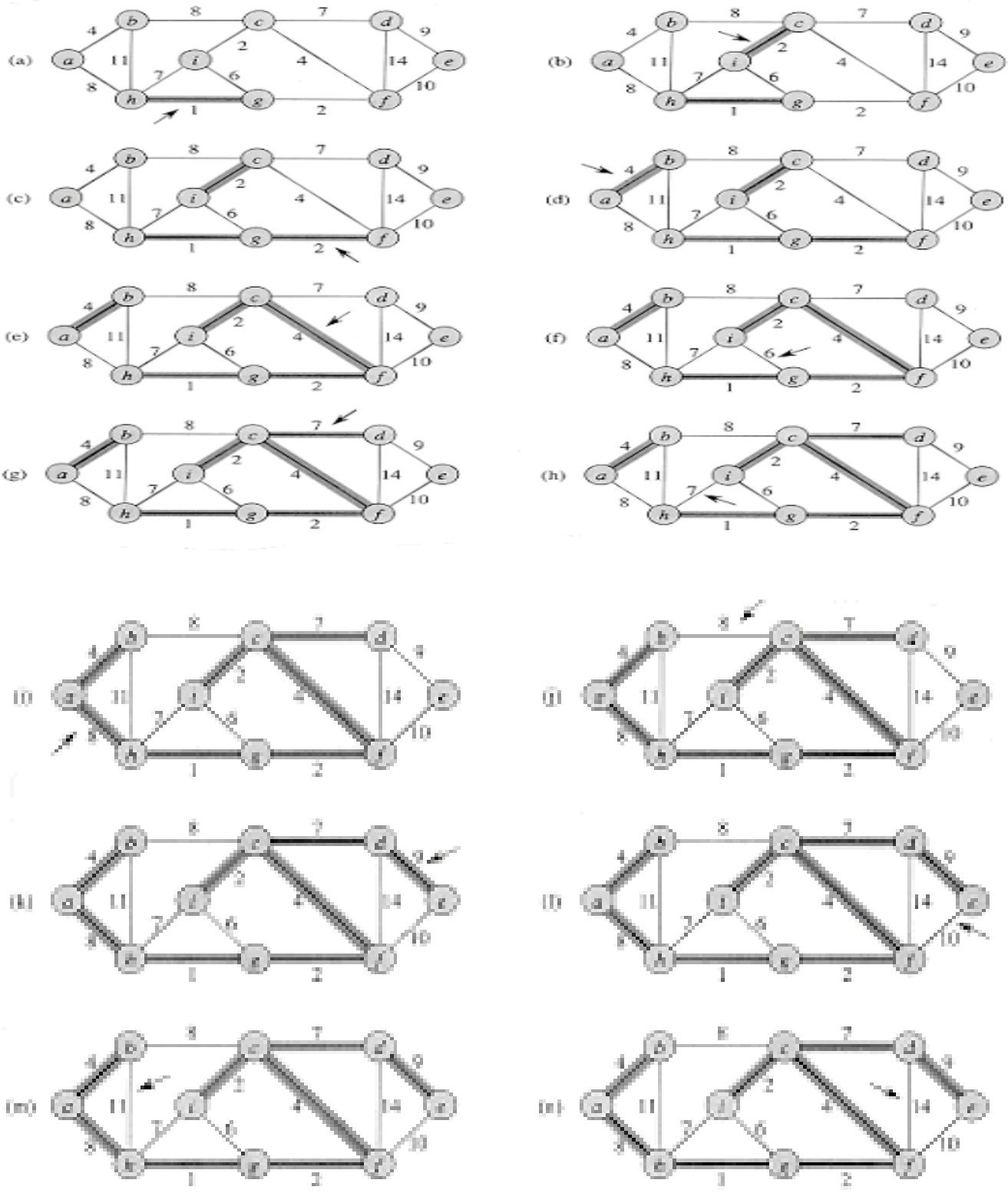


KRUSHKAL'S ALGORITHM:

1. Sort all the edges in non-decreasing order of their weight.
2. 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.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.

EXAMPLE:



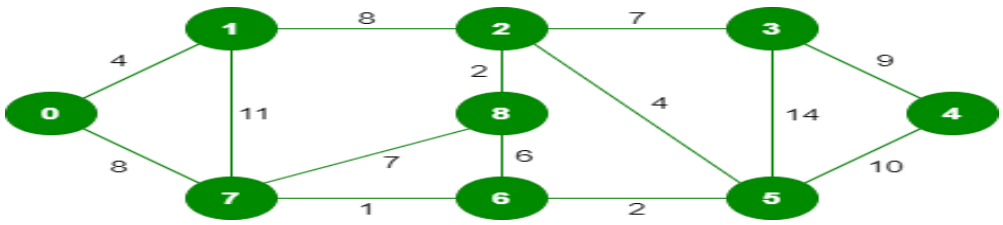

(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
(k)
(l)
(m)
(n)

Tree edges	Selected edge	action	Edges in sorting order
--	gh(1)	Add	gh(1), ci(2), fg(2), ab(4), cf(4), ig(6), cd(7), hi(7), ah(8), bc(8), de(9), ef(10), bh(11), df(14)
gh(1)	ci(2)	Add	ci(2), fg(2), ab(4), cf(4), ig(6), cd(7), hi(7), ah(8), bc(8), de(9), ef(10), bh(11), df(14)
ci(2)	fg(2)	Add	fg(2), ab(4), cf(4), ig(6), cd(7), hi(7), ah(8), bc(8), de(9), ef(10), bh(11), df(14)
fg(2)	ab(4)	Add	ab(4), cf(4), ig(6), cd(7), hi(7), ah(8), bc(8), de(9), ef(10), bh(11), df(14)
ab(4)	cf(4)	Add	cf(4), ig(6), cd(7), hi(7), ah(8), bc(8), de(9), ef(10), bh(11), df(14)
cf(4)	ig(6)	Reject	ig(6), cd(7), hi(7), ah(8), bc(8), de(9), ef(10), bh(11), df(14)
	cd(7)	Add	cd(7), hi(7), ah(8), bc(8), de(9), ef(10), bh(11), df(14)
cd(7)	hi(7)	Reject	hi(7), ah(8), bc(8), de(9), ef(10), bh(11), df(14)
	ah(8)	Add	ah(8), bc(8), de(9), ef(10), bh(11), df(14)
ah(8)	bc(8)	Reject	bc(8), de(9), ef(10), bh(11), df(14)
	de(9)	Add	de(9), ef(10), bh(11), df(14)
de(9)	ef(10)	Reject	ef(10), bh(11), df(14)
	bh(11)	Reject	bh(11), df(14)
	df(14)	Reject	df(14)

Total cost = 1 + 2 + 2 + 4 + 4 + 7 + 8 + 9 = 37

```
ALGORITHM KRUSKAL ( G ):
//INPUT: A weighted connected graph G = <V, E>
//OUTPUT: T minimum spanning tree of G
{
        T = ∅;                  // empty graph
        for i = 1 to n – 2 do
        {
                e := any edge in G with smallest weight that does not
                        form a cycle when added to T
                T :=  T union { e }
        }
        return T;
}
```

Consider the below input graph.

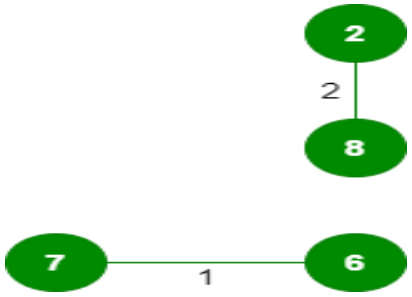


The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.
After sorting:

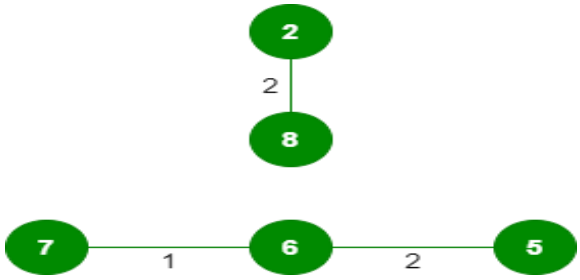
Weight	Source Vertex	Destination Vertex
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

Now pick all edges one by one from the sorted list of edges
1. *Pick edge 7-6:* No cycle is formed, include it.

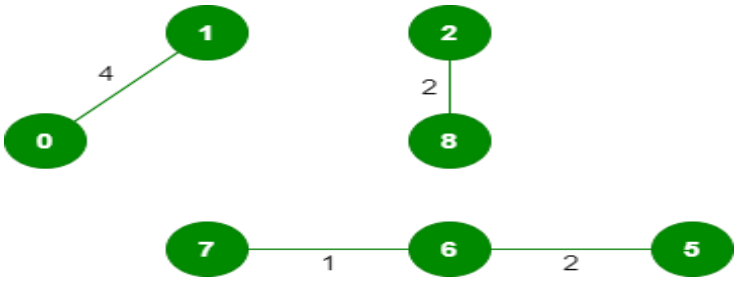
2. *Pick edge 8-2:* No cycle is formed, include it.



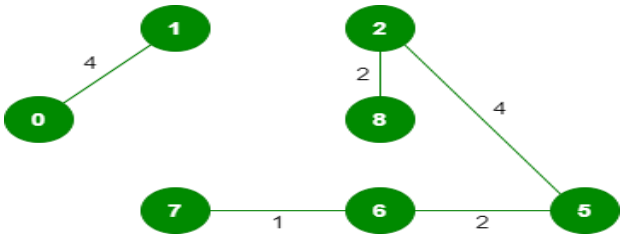
3. *Pick edge 6-5:* No cycle is formed, include it.



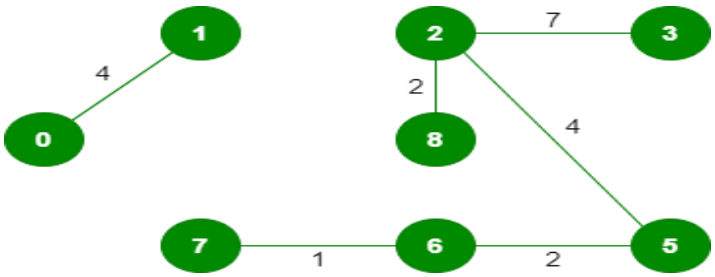
4. *Pick edge 0-1:* No cycle is formed, include it.



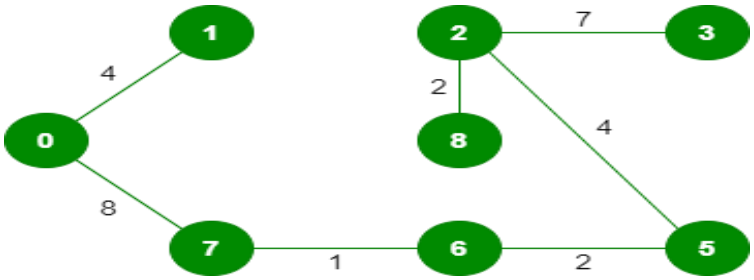
5. *Pick edge 2-5:* No cycle is formed, include it.



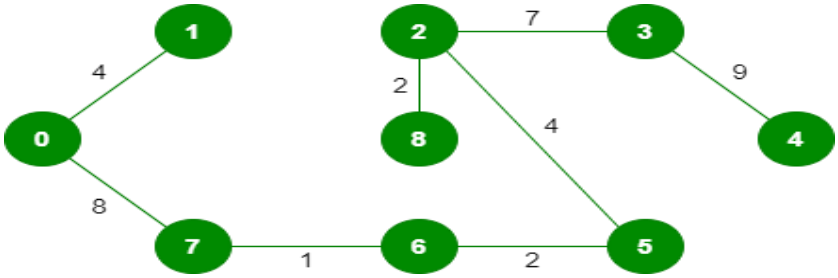
6. *Pick edge 8-6:* Since including this edge results in the cycle, discard it.
7. *Pick edge 2-3:* No cycle is formed, include it.



8. *Pick edge 7-8:* Since including this edge results in the cycle, discard it.
9. *Pick edge 0-7:* No cycle is formed, include it.



10. *Pick edge 1-2:* Since including this edge results in the cycle, discard it.
11. *Pick edge 3-4:* No cycle is formed, include it.

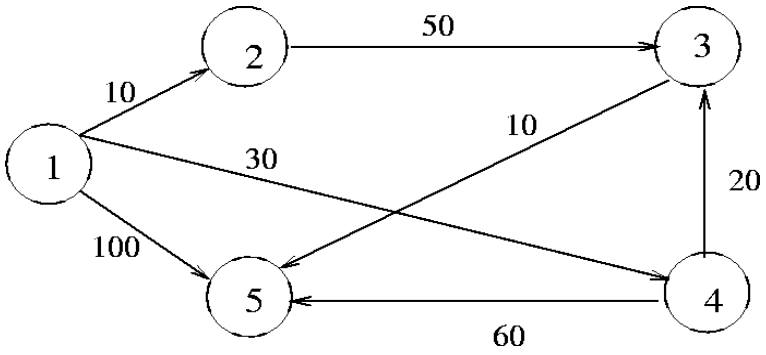


Dijkstra's Algorithm
SINGLE SOURCE SHORTEST PATH PROBLEM

- Graphs can be used to represent distance between states or country with vertices representing cities and edge representing sections of highway. The edges can be assign weights which may be either distance between two cities connected by edge or average time to drive along that section of highway.
- Given a directed weighted graph G = (V, E) and a source vertex V0. The problem is to determine the shortest paths from V0 to all remaining vertices of G.

Example:

Find the shortest paths from vertex '1' to remaining all vertices in the given digraph.



S	Vertex selected	Distances				
		1	2	3	4	5
---	1	0	10	∞	30	100
{1}	2	0	10	60	30	100
{1,2}	4	0	10	50	30	90
{1,2,4}	3	0	10	50	30	60
{1,2,4,3}	5	0	10	50	30	60

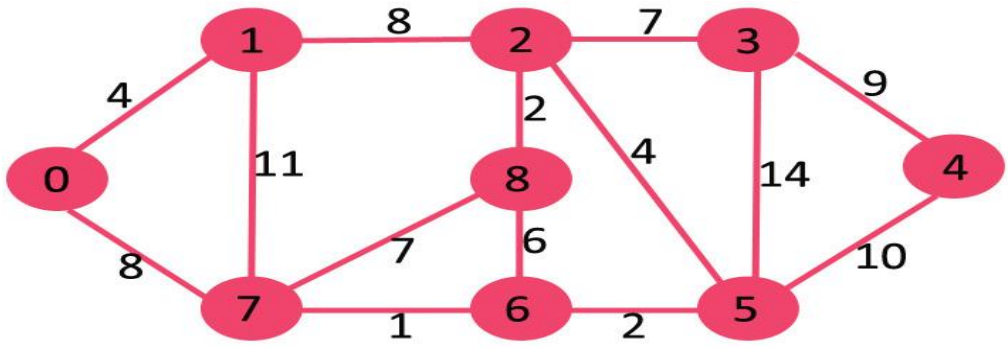
Shortest paths	path	distance
1-2	1-2	10
1-3	1-4-3	50
1-4	1-4	30
1-5	1-4-3-5	60

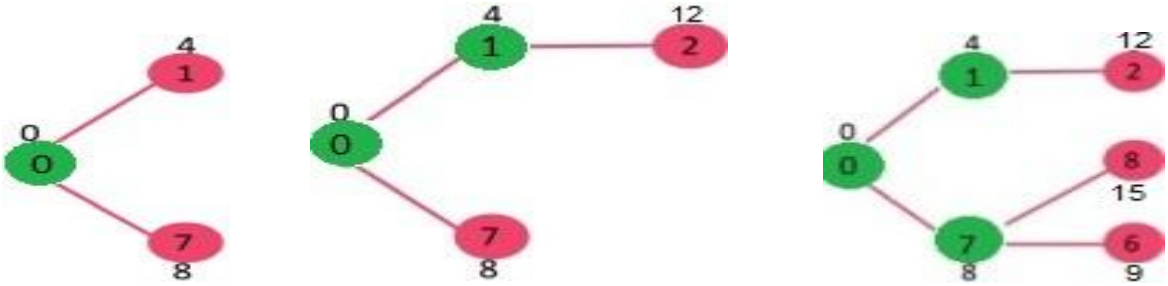
ALGORITHM:-

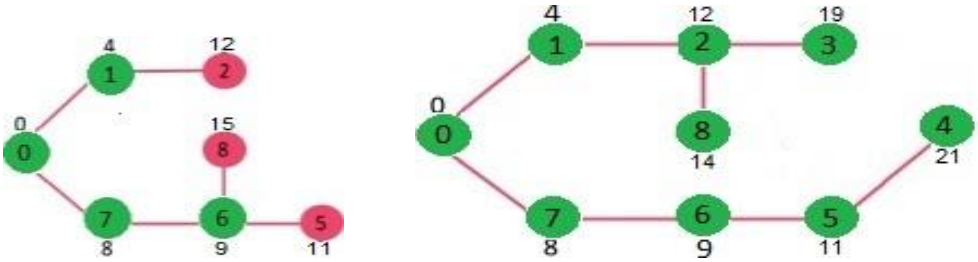
```
Algorithm shortest path (v, cost, dist, n)
{
    for i:= 1 to n do
    {
            s[i] := false;
            dist[i] := cost[v, i];
    }
    s[v] = true;
    dist[v] =0.0;
    for num := 2 to n do
    {
    // determine n-1 paths from v
    Choose u from among those vertices not in s
            Such that dist[u] is minimum;
            s[u]:= true;
            for (each w adjacent to u with s[w] = false) do
            if(dist[w] > dist[u]+cost[u, w]) then
```

```
                    dist[w] = dist[u] + cost[u, w];
        }
    }
```

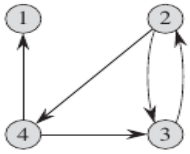
Example:
Let us understand with the following example:







Transitive closure

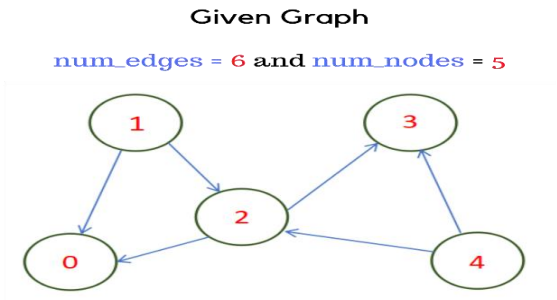


$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

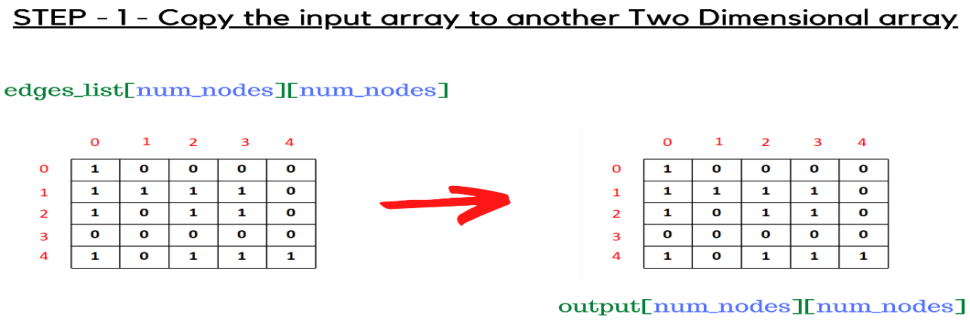
$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Figure 25.5 A directed graph and the matrices $T^{(k)}$ computed by the transitive-closure algorithm.

Example-2:

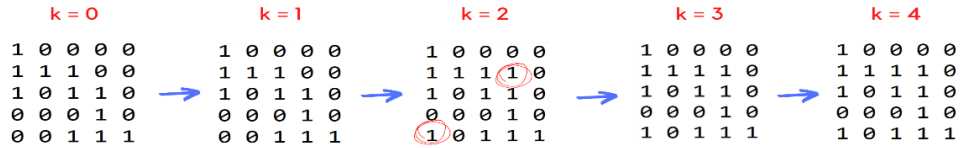


As per the algorithm, the first step is to allocate O(V^2) space as another two dimensional array named **output** and copy the entries in edges_list to the output matrix. The edges_list matrix and the output matrix are shown below.



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

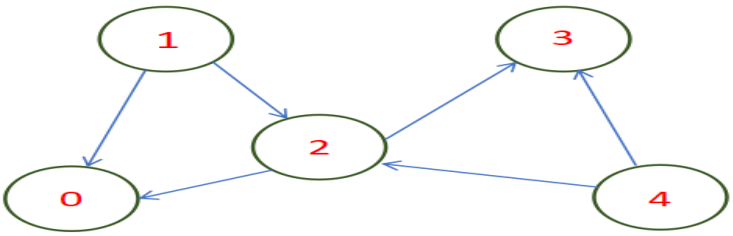
For the Outer Loop iteration from k = 0 to k = num_nodes-1



Changes being highlighted by

Similarly you can come up with a pen and paper and check manually on how the code works for other iterations of i and j. After the entire loop gets over, we will get the desired transitive closure matrix.

Suppose we are given the following Directed Graph,

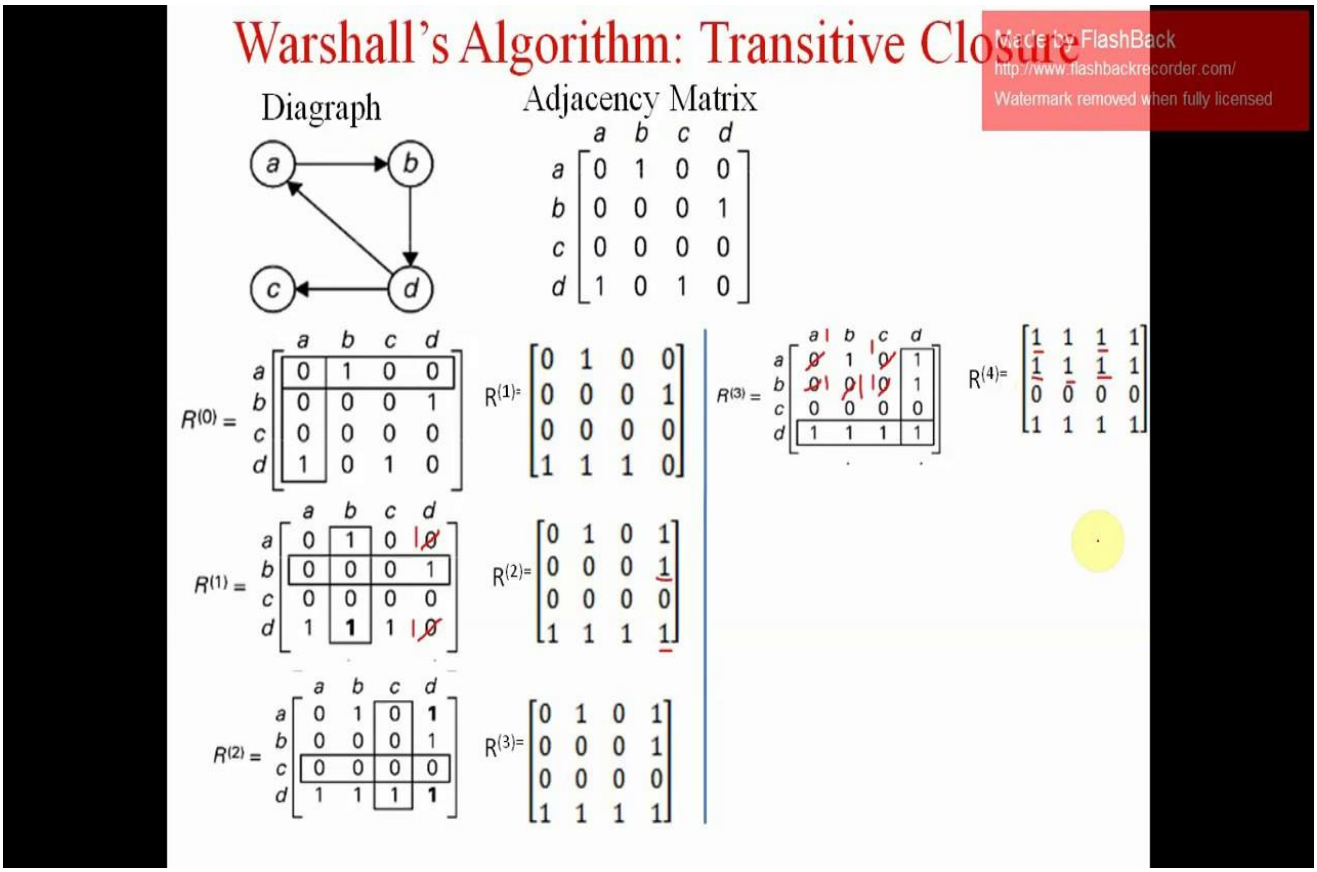


Then, the reachability matrix of the graph can be given by,

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

This matrix is known as the transitive closure matrix, where '1' depicts the availability of a path from i to j, for each (i,j) in the matrix.

Warshall's Algorithm.



Algorithm

```
TRANSITIVE-CLOSURE(G)
 1  n ← |V[G]|
 2  for i ← 1 to n
 3       do for j ← 1 to n
 4            do if i = j or (i, j) ∈ E[G]
 5                 then t_ij^(0) ← 1
 6                 else t_ij^(0) ← 0
 7  for k ← 1 to n
 8       do for i ← 1 to n
 9            do for j ← 1 to n
10                 do t_ij^(k) ← t_ij^(k-1) ∨ (t_ik^(k-1) ∧ t_kj^(k-1))
11  return T^(n)
```