

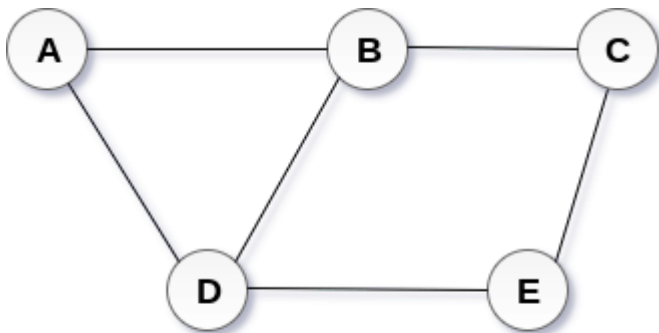
### 1. Definition, Terminologies and Types of Graphs

A graph can be defined as group of vertices and edges that are used to connect these vertices. A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.

#### Definition

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

A Graph  $G(V, E)$  with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.



### Undirected Graph

#### Graph Terminology

##### Path

A path can be defined as the sequence of nodes that are followed in order to reach some terminal node  $V$  from the initial node  $U$ .

##### Closed Path

A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if  $V_0 = V_N$ .

##### Simple Path

If all the nodes of the graph are distinct with an exception  $V_0 = V_N$ , then such path  $P$  is called as closed simple path.

##### Cycle

A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.

### Connected Graph

A connected graph is the one in which some path exists between every two vertices  $(u, v)$  in  $V$ . There are no isolated nodes in connected graph.

### Complete Graph

A complete graph is the one in which every node is connected with all other nodes. A complete graph contains  $n(n-1)/2$  edges where  $n$  is the number of nodes in the graph.

### Weighted Graph

In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge  $e$  can be given as  $w(e)$  which must be a positive (+) value indicating the cost of traversing the edge.

### Digraph

A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.

### Loop

An edge that is associated with the similar end points can be called as Loop.

### Adjacent Nodes

If two nodes  $u$  and  $v$  are connected via an edge  $e$ , then the nodes  $u$  and  $v$  are called as neighbours or adjacent nodes.

### Degree of the Node

A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.

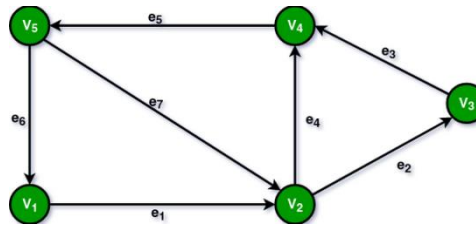
Out-degree,  $\text{outdeg}(v)$ , is the number of outgoing edges.

In-degree,  $\text{indeg}(v)$ , is the number of incoming edges.

### Types:

In data structures, there are several types of graphs that are commonly used to represent relationships between objects or entities. The main types of graphs are:

1. **Undirected Graph:** An undirected graph is a graph in which edges have no orientation. In other words, the edges do not have a specific direction associated with them. The relationships between nodes are symmetric, meaning that if there is an edge between node A and node B, then there is also an edge between node B and node A.
2. **Directed Graph (Digraph):** A directed graph is a graph in which edges have a specific direction. Each edge has a starting vertex (tail) and an ending vertex (head). The relationships between nodes can be asymmetric, meaning that there may be an edge from node A to node B, but not vice versa.



3. **Weighted Graph:** A weighted graph is a graph in which each edge has a weight or cost associated with it. These weights can represent various quantities such as distance, cost, or capacity. Weighted graphs are used when the relationships between nodes have associated values or costs.
4. **Unweighted Graph:** An unweighted graph is a graph in which edges do not have any associated weights or costs. It represents a simple relationship between nodes without considering any additional values.
5. **Cyclic Graph:** A cyclic graph is a graph that contains at least one cycle. A cycle is a path that starts and ends at the same vertex, visiting one or more vertices in between. In other words, it is possible to follow a series of edges and return to the starting vertex.
6. **Acyclic Graph:** An acyclic graph is a graph that does not contain any cycles. It is a directed graph that does not have any directed cycles. Acyclic graphs are often used in applications where a strict ordering or hierarchy needs to be represented.
7. **Connected Graph:** A connected graph is an undirected graph in which there is a path between every pair of vertices. In other words, there are no isolated vertices in a connected graph. Each vertex can be reached from any other vertex by traversing the edges of the graph.
8. **Disconnected Graph:** A disconnected graph is an undirected graph in which there are two or more vertices that are not reachable from each other. It consists of two or more connected components, where each component is a connected subgraph.

These are some of the common types of graphs used in data structures. Each type of graph has its own characteristics and applications, and the choice of graph type depends on the problem or scenario being addressed.

## 2. Representation of Graphs

There are two ways to store Graphs into the computer's memory:

- **Sequential representation** (or, Adjacency matrix representation)
- **Linked list representation** (or, Adjacency list representation)

In sequential representation, an adjacency matrix is used to store the graph. Whereas in linked list representation, there is a use of an adjacency list to store the graph.

### 2.1 Adjacency Matrix

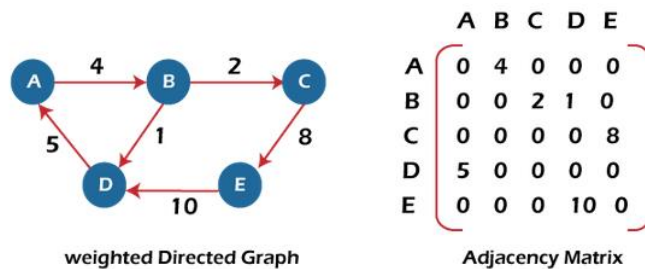
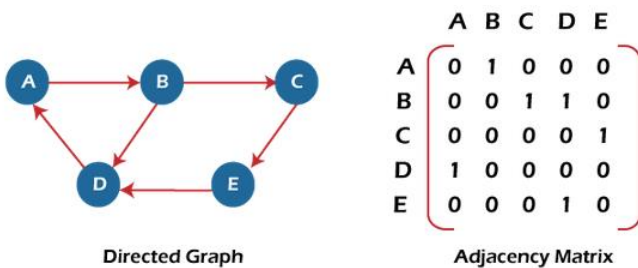
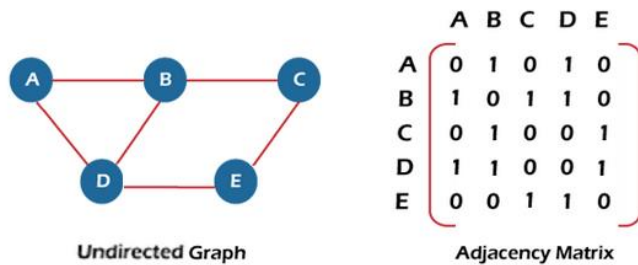
If an Undirected Graph  $G$  consists of  $n$  vertices then the adjacency matrix of a graph is an  $n \times n$  matrix  $A = [a_{ij}]$  and defined by

$$a_{ij} = \begin{cases} 1, & \text{if } \{V_i, V_j\} \text{ is an edge i. e., } v_i \text{ is adjacent to } v_j \\ 0, & \text{if there is no edge between } v_i \text{ and } v_j \end{cases}$$

## Unit 7: Graphs

If there exists an edge between vertex  $v_i$  and  $v_j$ , where  $i$  is a row and  $j$  is a column then the value of  $a_{ij}=1$ .

If there is no edge between vertex  $v_i$  and  $v_j$ , then value of  $a_{ij}=0$ .



### 2.2 Incidence Matrix

If an Undirected Graph  $G$  consists of  $n$  vertices and  $m$  edges, then the incidence matrix is an  $n \times m$  matrix  $C = [c_{ij}]$  and defined by

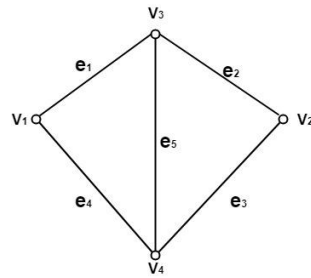
$$c_{ij} = \begin{cases} 1, & \text{if the vertex } V_i \text{ incident by edge } e_j \\ 0, & \text{otherwise} \end{cases}$$

There is a row for every vertex and a column for every edge in the incident matrix.

The number of ones in an incidence matrix of the undirected graph (without loops) is equal to the sum of the degrees of all the vertices in a graph.

**Example:** Consider the undirected graph  $G$  as shown in fig. Find its incidence matrix  $M_I$ .

## Unit 7: Graphs



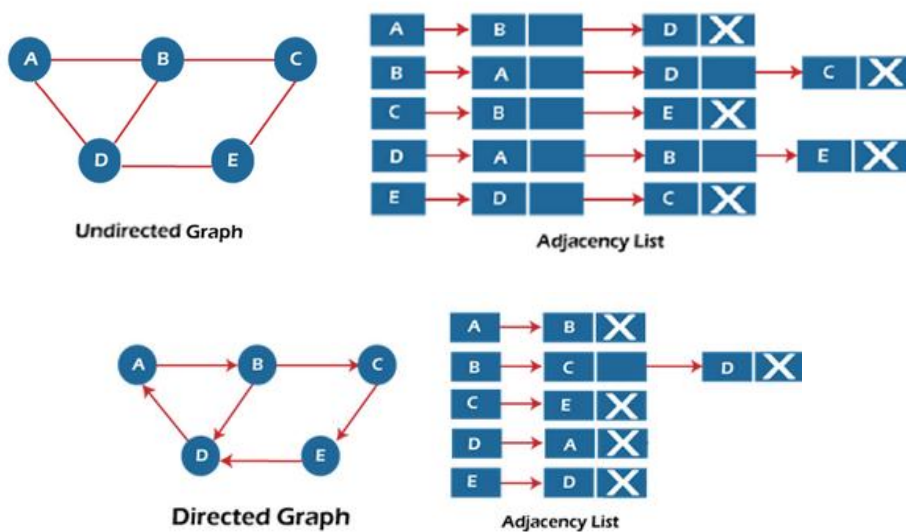
**Solution:** The undirected graph consists of four vertices and five edges. Therefore, the incidence matrix is an 4 x 5 matrix, which is shown in Fig:

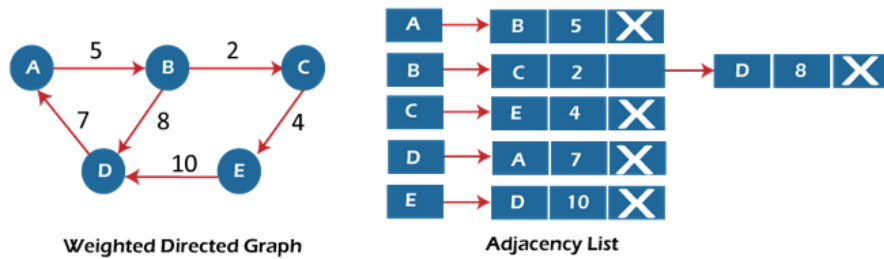
$$M_I = \begin{matrix} & e_1 & e_2 & e_3 & e_4 & e_5 \\ \begin{matrix} V_1 \\ V_2 \\ V_3 \\ V_4 \end{matrix} & \begin{bmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix} \end{matrix}$$

### 2.3 Adjacency List

An adjacency list is used in the linked representation to store the Graph in the computer's memory. It is efficient in terms of storage as we only have to store the values for edges.

Let's see the adjacency list representation of an undirected graph.





### 3. Transitive Closure and Warshall's Algorithm

Transitive closure of a graph  $G$  is a new graph that captures all possible paths between nodes in  $G$ . If there exists a path from node  $A$  to node  $B$  in the original graph, then there will be a corresponding edge from node  $A$  to node  $B$  in the transitive closure graph.

To compute the transitive closure of a graph, various algorithms can be used, such as Warshall's algorithm. These algorithms iteratively determine the existence of paths between pairs of nodes and update the transitive closure matrix accordingly.

The transitive closure can be useful in many graph-related applications, such as analyzing dependencies, determining reachability or accessibility, and identifying connected components. It helps in understanding the full extent of connectivity in a graph by capturing all indirect relationships between nodes.

#### Input and Output

Input:

```
1 1 0 1
0 1 1 0
0 0 1 1
0 0 0 1
```

Output:

The matrix of transitive closure

```
1 1 1 1
0 1 1 1
0 0 1 1
0 0 0 1
```

#### Warshall's Algorithm

Warshall's algorithm computes the transitive closure of a directed graph. It determines if there is a path from one node to another, considering all possible intermediate nodes. The algorithm maintains an adjacency matrix, initially filled with the graph's adjacency information. It iteratively updates the matrix by checking paths through each node, marking reachability. The process continues until all intermediate nodes are considered. The resulting matrix represents the transitive closure, indicating if there is a path between each pair of nodes.

## Unit 7: Graphs

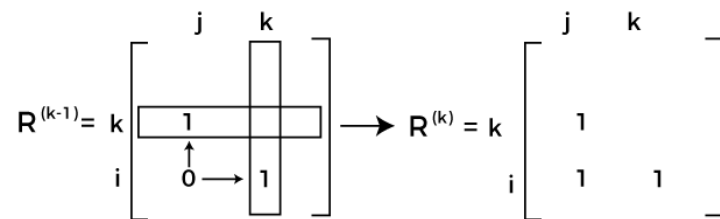
Recurrence relating elements  $R(k)$  to elements of  $R(k-1)$  can be described as follows:

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

In order to generate  $R(k)$  from  $R(k-1)$ , the following rules will be implemented:

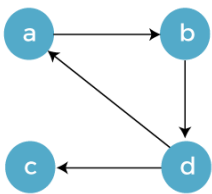
**Rule 1:** In row  $i$  and column  $j$ , if the element is 1 in  $R(k-1)$ , then in  $R(k)$ , it will also remain 1.

**Rule 2:** In row  $i$  and column  $j$ , if the element is 0 in  $R(k-1)$ , then the element in  $R(k)$  will be changed to 1 iff the element in its column  $j$  and row  $k$ , and the element in row  $i$  and column  $k$  are both 1's in  $R(k-1)$ .



### Application of Warshall's algorithm to the directed graph

In order to understand this, we will use a graph, which is described as follow:



For this graph  $R(0)$  will be looked like this:

$$R^{(0)} = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{matrix}$$

Here  $R(0)$  shows the adjacency matrix. In  $R(0)$ , we can see the existence of a path, which has no intermediate vertices. We will get  $R(1)$  with the help of a boxed row and column in  $R(0)$ .

In  $R(1)$ , we can see the existence of a path, which has intermediate vertices. The number of the vertices is not greater than 1, which means just a vertex. It contains a new path from d to b. We will get  $R(2)$  with the help of a boxed row and column in  $R(1)$ .

$$R^{(1)} = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{matrix}$$

## Unit 7: Graphs

In  $R(2)$ , we can see the existence of a path, which has intermediate vertices. The number of the vertices is not greater than 2, which means a and b. It contains two new paths. We will get  $R(3)$  with the help of a boxed row and column in  $R(2)$ .

$$R^{(2)} = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{matrix}$$

In  $R(3)$ , we can see the existence of a path, which has intermediate vertices. The number of the vertices is not greater than 3, which means a, b and c. It does not contain any new paths. We will get  $R(4)$  with the help of a boxed row and column in  $R(3)$ .

$$R^{(3)} = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{matrix}$$

In  $R(4)$ , we can see the existence of a path, which has intermediate vertices. The number of the vertices is not greater than 4, which means a, b, c and d. It contains five new paths.

$$R^{(4)} = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{matrix}$$

### Implementaion of Warshall's Algorithm

```
#include <iostream>
```

```
using namespace std;
```

```
void warshalls(int a[20][20], int n) {
    int i, j, k;
    for (k = 0; k < n; k++) {
        for (i = 0; i < n; i++) {
            for (j = 0; j < n; j++) {
                a[i][j] = a[i][j] || (a[i][k] && a[k][j]);
            }
        }
    }
}
```

```
cout << "\nThe transitive closure is:\n";
```

```
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
```



```
        cout << a[i][j] << "\t";
    }
    cout << endl;
}
}
```

```
int main() {
    int a[20][20], n, i, j;

    cout << "\nEnter the number of vertices of the graph:\n";
    cin >> n;

    cout << "\nEnter the adjacency matrix:\n";
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            cin >> a[i][j];
        }
    }

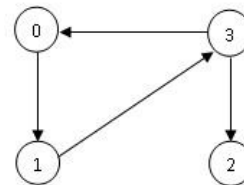
    warshalls(a, n);

    return 0;
}
```

### OUTPUT:

```
Enter the number of vertices of the graph:
4
Enter the adjacency matrix:
0 1 0 0
0 0 0 1
0 0 0 0
1 0 1 0
The transitive closure is:
1 1 1 1
1 1 1 1
0 0 0 0
1 1 1 1
```

**Input Graph:**



### 4. Graph Traversals:

Graph traversal is a fundamental operation in graph data structures. It refers to the process of visiting and exploring all the vertices or nodes of a graph in a specific order. 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 commonly used methods for graph traversal:

- Breadth-first search (BFS)
- Depth-first search (DFS)

### 4.1 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.

**Algorithm:**

```
void BFTraversal(Graph G)
{
    for each vertex v in G
        visited[v] = false;
    for each vertex v in G
        if(visited[v] == false)
        {
            enqueue(V, Q);
            do
            {
                v = dequeue(q);
                visited[v] = true;
                visited(v);
                for each vertex w adjacent to v
                    if(visited[w] == false)
                        enqueue(w, q);
            } while(isempty(Q) == false)
        }
}
```

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

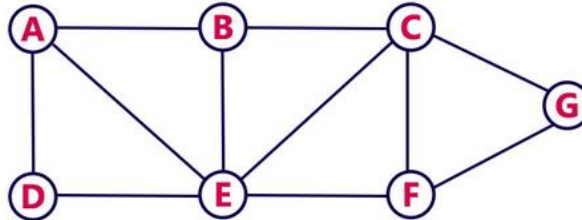
- Step 1 - Define a Queue of size total number of vertices in the graph.
- Step 2 - Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
- Step 3 - Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert 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.

## Unit 7: Graphs

- Step 6 - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

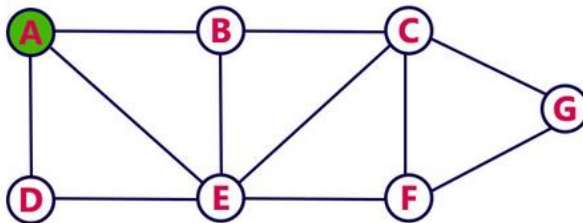
Example

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.

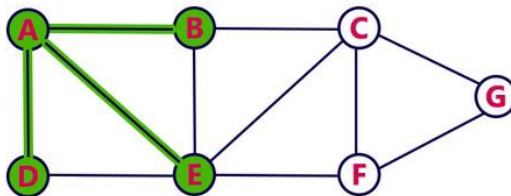


### Queue



### 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.

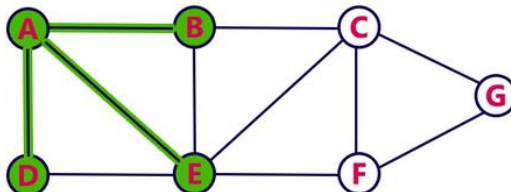


### Queue



### Step 3:

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



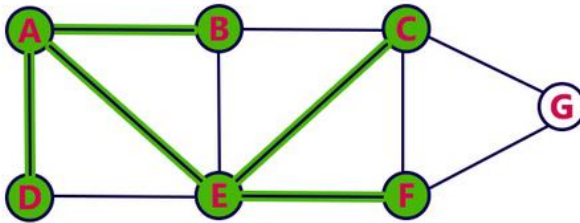
### Queue



## Unit 7: Graphs

### 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.

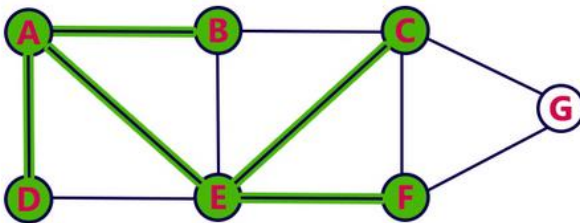


### Queue



### Step 5:

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

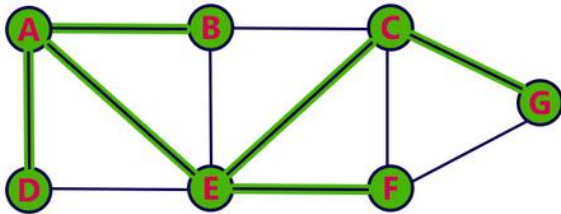


### Queue



### 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.

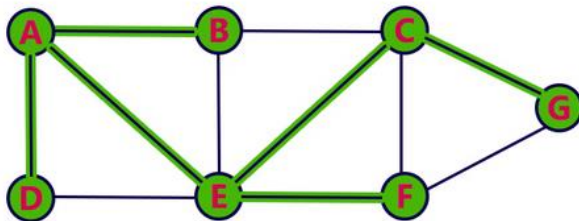


### Queue



### Step 7:

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

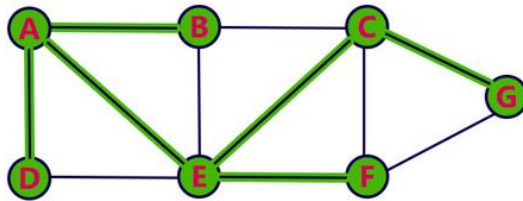


### Queue



### Step 8:

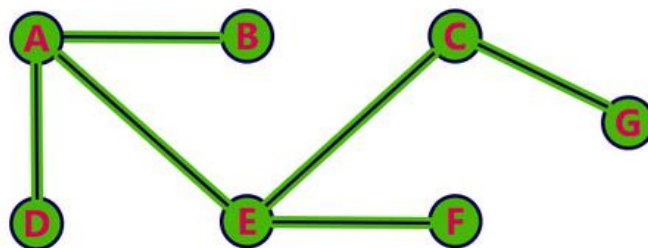
- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue



- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



Result : A D E B C F G

### 4.2 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.

#### Algorithm:

```
void DFTraversal(Graph G)
{
    for each vertex v in G
        Visited[v]=false;
    for each vertex v in G
        If(visited[v] == false)
            Traverse(v);
}
void Traverse(Vertex v)
{
    Visited[v]=true;
    Visited(v);
}
```

## Unit 7: Graphs

For each vertex  $w$  adjacent to  $v$

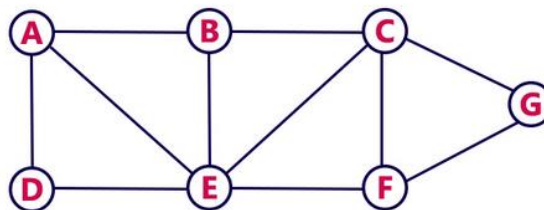
```
{  
    If(visited[w] == false)  
        Traverse(w);  
}
```

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

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

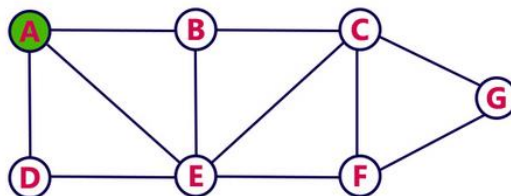
Example

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.

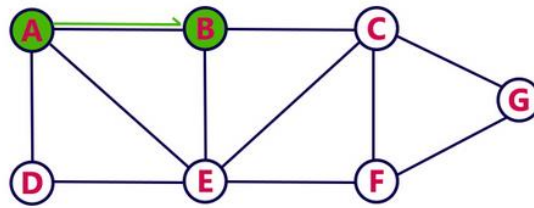


**Stack**

## Unit 7: Graphs

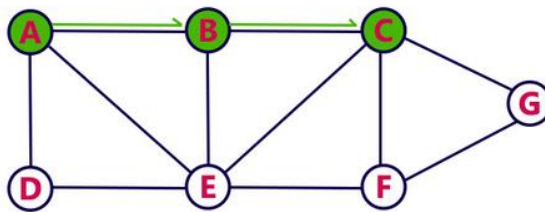
### Step 2:

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



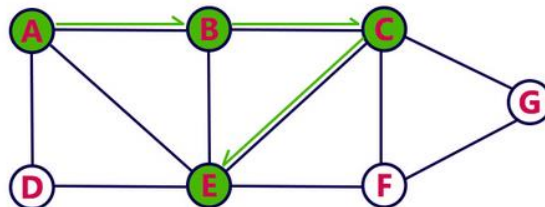
### Step 3:

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



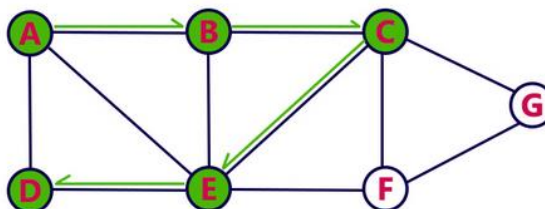
### Step 4:

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



### Step 5:

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

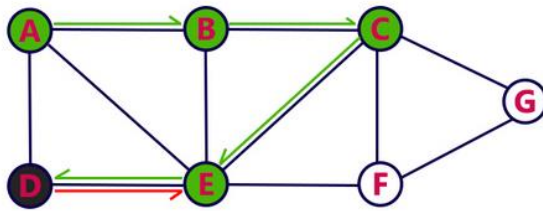




## Unit 7: Graphs

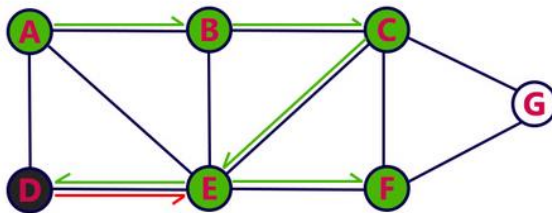
### Step 6:

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



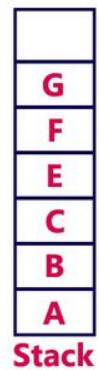
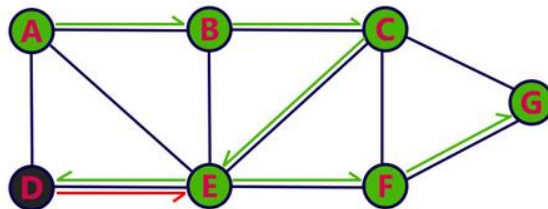
### Step 7:

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



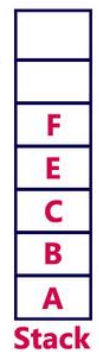
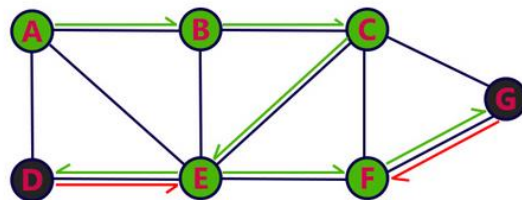
### Step 8:

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



### Step 9:

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

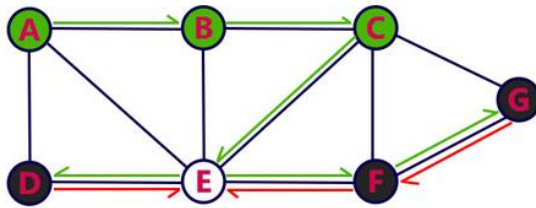




## Unit 7: Graphs

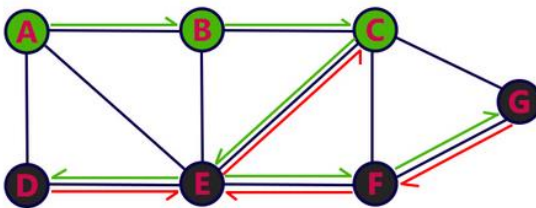
### Step 10:

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



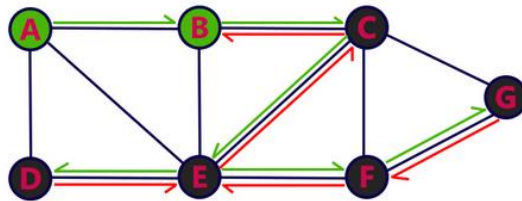
### Step 11:

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



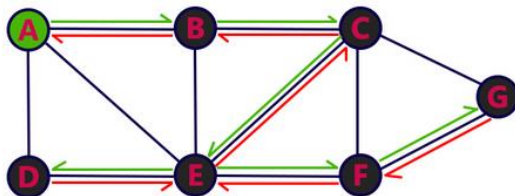
### Step 12:

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



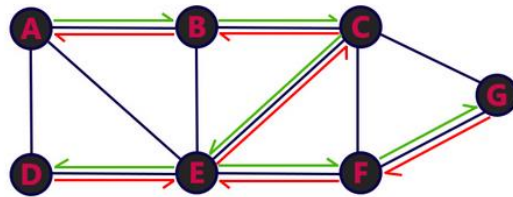
### Step 13:

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

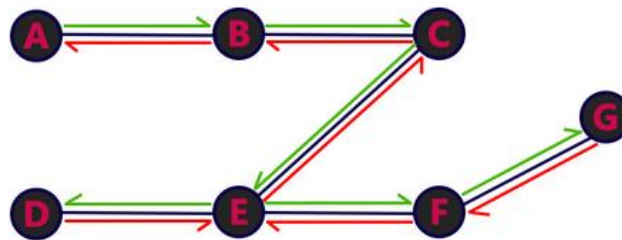


## Step 14:

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



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



Result: A B C E D F G

## 4.3 Topological Sort

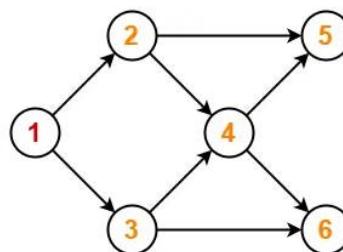
Topological Sort is a linear ordering of the vertices in such a way that if there is an edge in the DAG going from vertex 'u' to vertex 'v', then 'u' comes before 'v' in the ordering.

It is important to note that-

- Topological Sorting is possible if and only if the graph is a **Directed Acyclic Graph**.
- There may exist multiple different topological orderings for a given directed acyclic graph.

### Topological Sort Example-

Consider the following directed acyclic graph-



Topological Sort Example

## Unit 7: Graphs

For this graph, following 4 different topological orderings are possible-

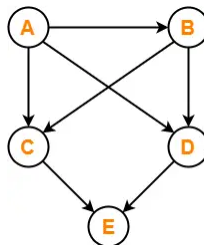
- 1 2 3 4 5 6
- 1 2 3 4 6 5
- 1 3 2 4 5 6
- 1 3 2 4 6 5

Few important applications of topological sort are-

- Scheduling jobs from the given dependencies among jobs
- Instruction Scheduling
- Determining the order of compilation tasks to perform in makefiles
- Data Serialization

### Example:

Find the number of different topological orderings possible for the given graph-

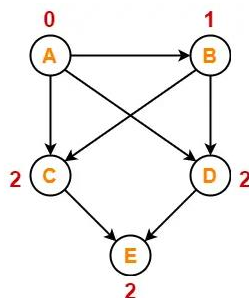


### Solution-

The topological orderings of the above graph are found in the following steps-

#### Step-01:

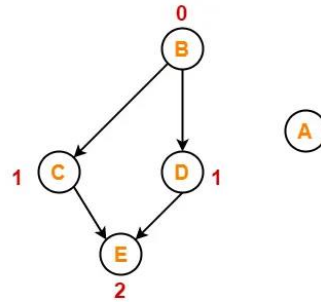
Write in-degree of each vertex-



#### Step-02:

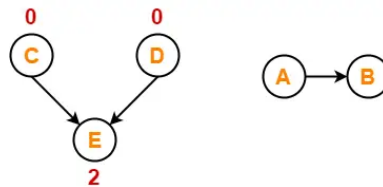
- Vertex-A has the least in-degree.
- So, remove vertex-A and its associated edges.
- Now, update the in-degree of other vertices.

## Unit 7: Graphs



### Step-03:

- Vertex-B has the least in-degree.
- So, remove vertex-B and its associated edges.
- Now, update the in-degree of other vertices.



### Step-04:

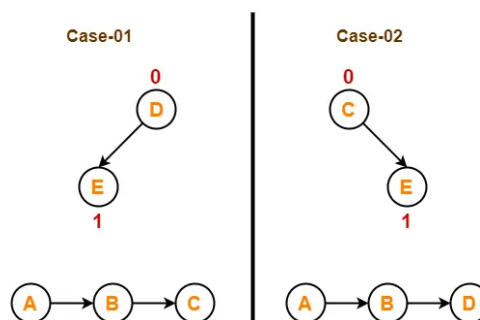
There are two vertices with the least in-degree. So, following 2 cases are possible-

In case-01,

- Remove vertex-C and its associated edges.
- Then, update the in-degree of other vertices.

In case-02,

- Remove vertex-D and its associated edges.
- Then, update the in-degree of other vertices.



### Step-05:

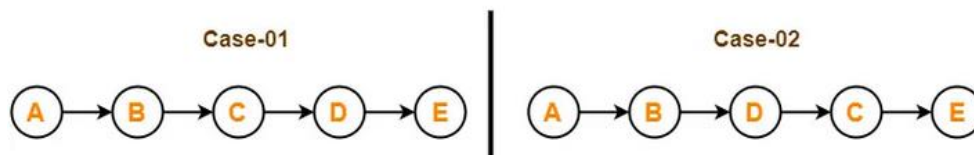
Now, the above two cases are continued separately in the similar manner.

In case-01,

- Remove vertex-D since it has the least in-degree.
- Then, remove the remaining vertex-E.

In case-02,

- Remove vertex-C since it has the least in-degree.
- Then, remove the remaining vertex-E.



### Conclusion-

For the given graph, following **2** different topological orderings are possible-

- **A B C D E**
- **A B D C E**

## 5. Minimum Spanning Tree

### Spanning tree

A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges. If a vertex is missed, then it is not a spanning tree.

If there are **n** number of vertices, the spanning tree should have **n - 1** number of edges.

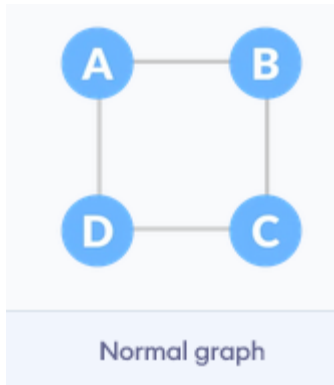
The edges may or may not have weights assigned to them.

The total number of spanning trees with n vertices that can be created from **a complete graph** is equal to  $n^{(n-2)}$ .

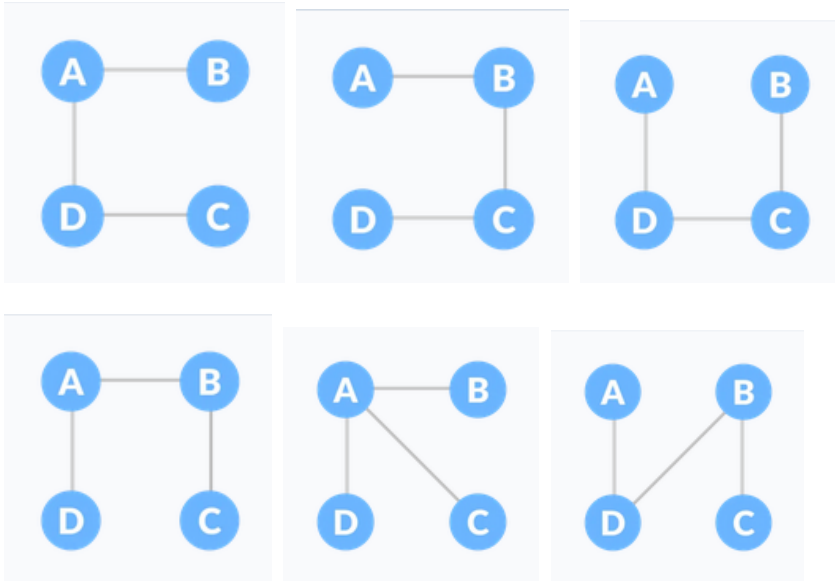
If we have  $n = 4$ , the maximum number of possible spanning trees is equal to  $4^{4-2} = 16$ . Thus, 16 spanning trees can be formed from a complete graph with 4 vertices.

### Example:

Let the original graph be:



Some of the possible spanning trees that can be created from the above graph are:

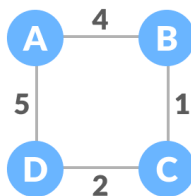


### Minimum Spanning Tree

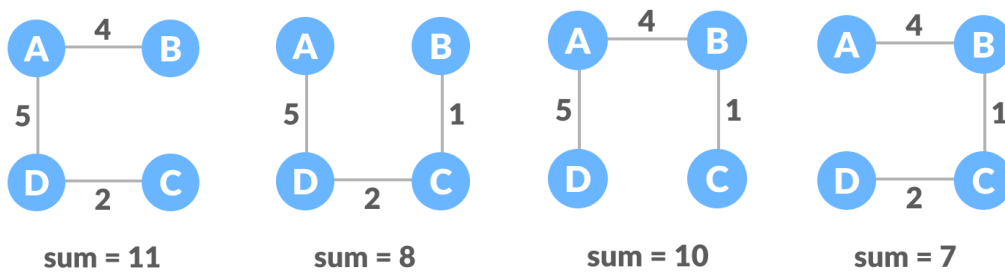
A minimum spanning tree is a spanning tree in which the sum of the weight of the edges is as minimum as possible. To derive an MST, Prim's algorithm or Kruskal's algorithm can be used.

Let's understand the above definition with the help of the example below.

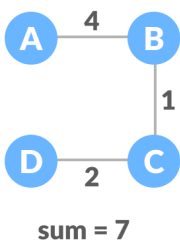
The initial graph is:



The possible spanning trees from the above graph are:



The minimum spanning tree from the above spanning trees is:



Minimum spanning tree

The minimum spanning tree from a graph is found using the following algorithms:

1. Prim's Algorithm
2. Kruskal's Algorithm

### 5.1 Kruskal's Algorithm

Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

- form a tree that includes every vertex
- has the minimum sum of weights among all the trees that can be formed from the graph

#### How Kruskal's algorithm works

It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.

We start from the edges with the lowest weight and keep adding edges until we reach our goal.

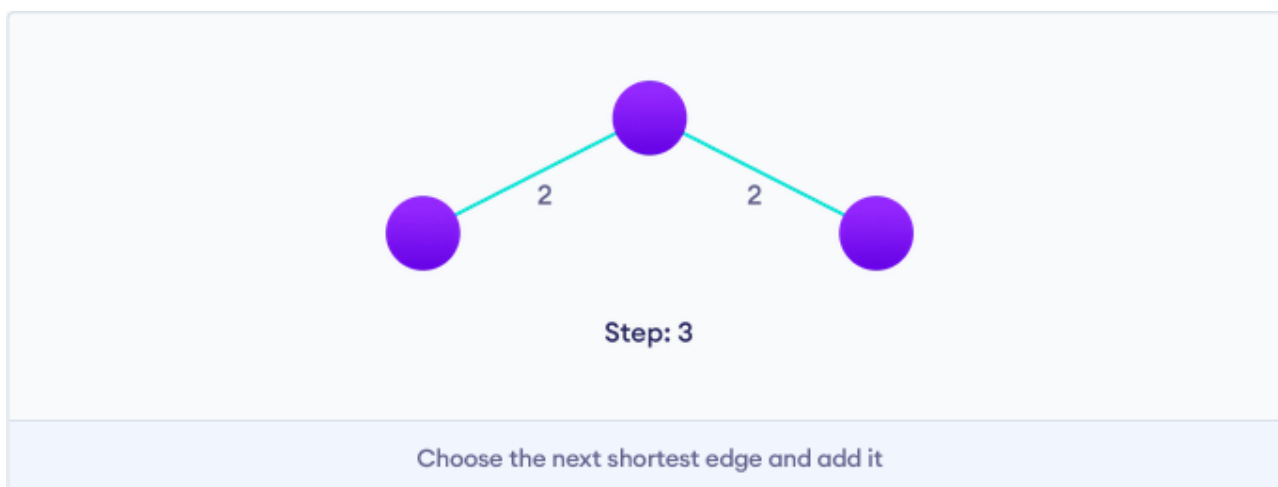
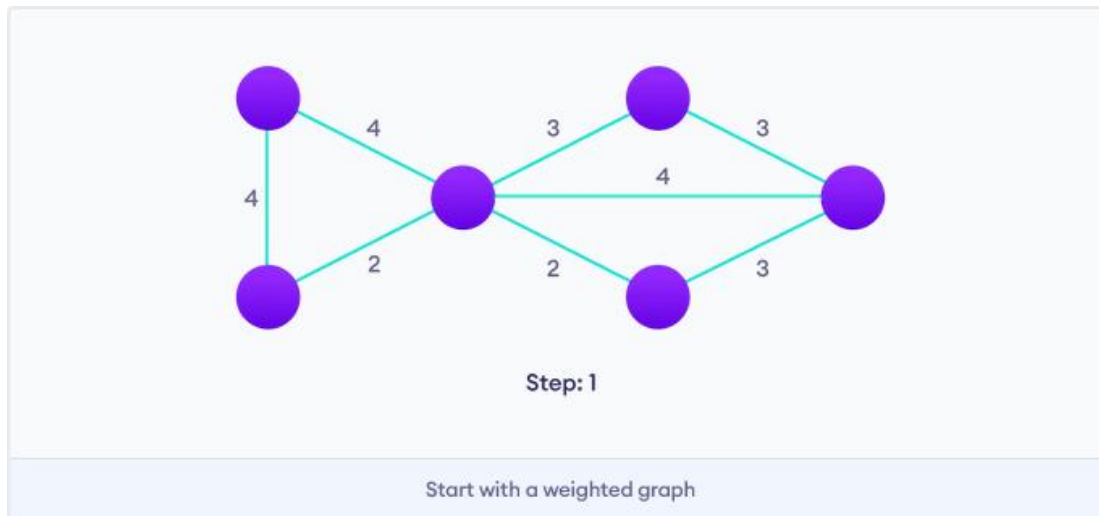
The steps for implementing Kruskal's algorithm are as follows:

1. Sort all the edges from low weight to high
2. Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.

## Unit 7: Graphs

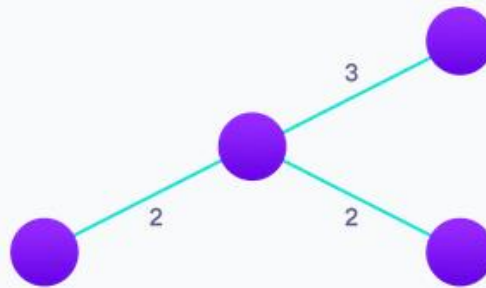
3. Keep adding edges until we reach all vertices.

**Example:**



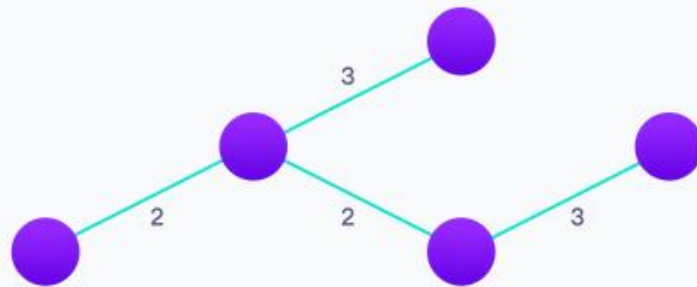


## Unit 7: Graphs



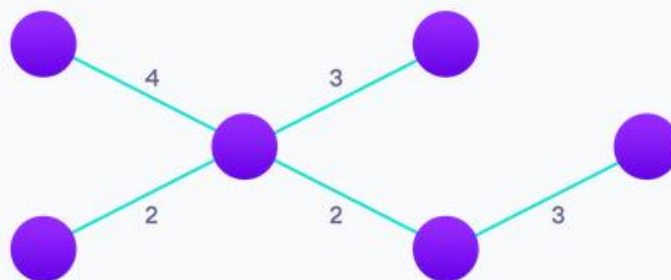
Step: 4

Choose the next shortest edge that doesn't create a cycle and add it



Step: 5

Choose the next shortest edge that doesn't create a cycle and add it



Step: 6

Repeat until you have a spanning tree

### 5.2 Prim's Algorithm

Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

- form a tree that includes every vertex
- has the minimum sum of weights among all the trees that can be formed from the graph

#### How Prim's algorithm works

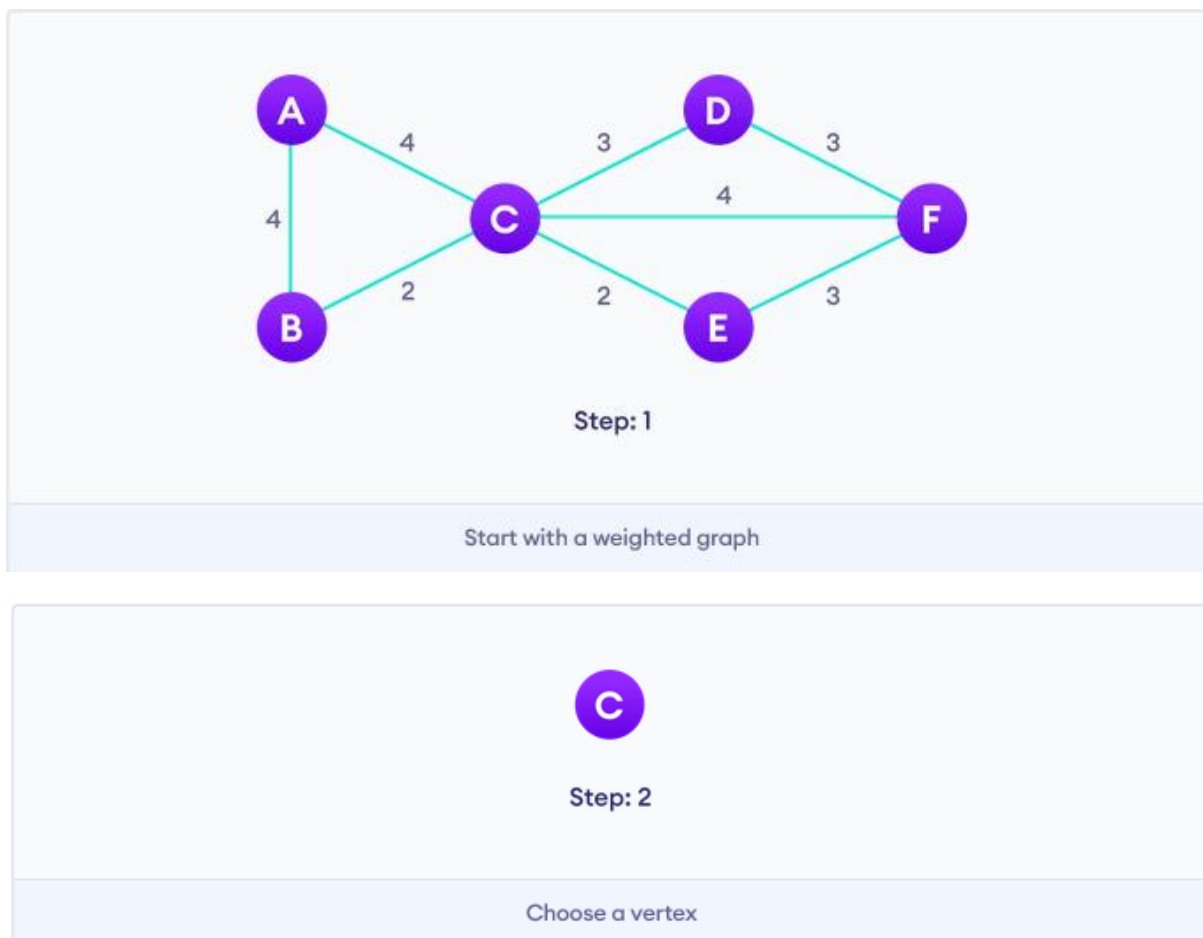
It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.

We start from one vertex and keep adding edges with the lowest weight until we reach our goal.

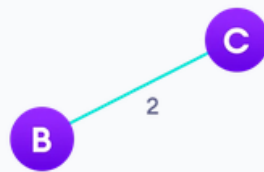
The steps for implementing Prim's algorithm are as follows:

1. Initialize the minimum spanning tree with a vertex chosen at random.
2. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
3. Keep repeating step 2 until we get a minimum spanning tree

#### Example:

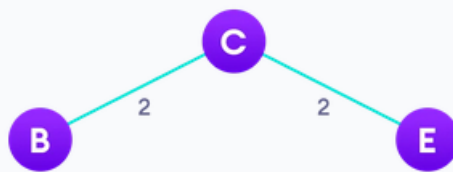


## Unit 7: Graphs



Step: 3

Choose the shortest edge from this vertex and add it



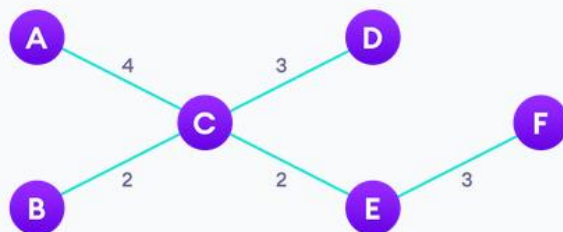
Step: 4

Choose the nearest vertex not yet in the solution



Step: 5

Choose the nearest edge not yet in the solution, if there are multiple choices, choose one at random



Step: 6

Repeat until you have a spanning tree

### 6. Shortest-Paths Problems

Shortest path problems are a class of optimization problems in graph theory and network analysis that involve finding the shortest path between two vertices in a graph. The "shortest path" is defined as the path with the minimum sum of edge weights or costs. These problems have numerous applications in various fields, such as computer networks, transportation systems, logistics, and routing algorithms.

#### 6.1 Types

Types of Shortest Path Problems:

1. **Single Source Shortest Path (SSSP):** In this type of problem, the objective is to find the shortest path from a single source vertex to all other vertices in the graph. One of the most famous algorithms for solving this problem is Dijkstra's algorithm, which works efficiently for graphs with non-negative edge weights.
2. **Single Destination Shortest Path:** Similar to SSSP, but the goal is to find the shortest path from all vertices to a single destination vertex. The most common approach to solve this is to reverse the graph and use algorithms like Dijkstra's or Bellman-Ford from the destination vertex.
3. **Single Pair Shortest Path:** This problem involves finding the shortest path between a specific source and destination vertex. Algorithms like Dijkstra's or Bellman-Ford can be used for this purpose.
4. **All Pairs Shortest Path (APSP):** In this problem, the goal is to find the shortest path between all pairs of vertices in the graph. The Floyd-Warshall algorithm is a popular approach for solving APSP problems and is efficient for dense graphs.

Shortest path algorithms, such as Dijkstra's, Bellman-Ford, and Floyd-Warshall, are fundamental in solving these problems and are widely used in real-world applications to optimize routing and resource allocation. The choice of the algorithm depends on the characteristics of the graph and the specific requirements of the problem at hand.

#### 6.2 Single-Source Shortest Path Problem- Dijkstra's Algorithm

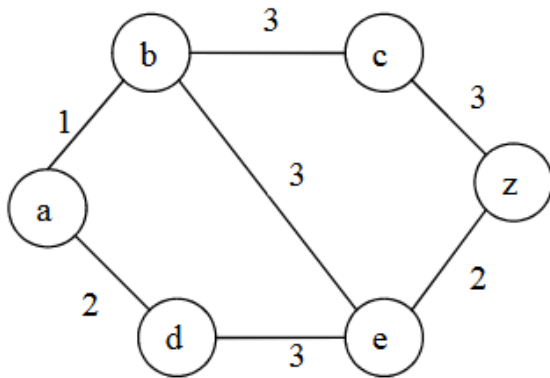
**Dijkstra's Algorithm** is a Graph algorithm **that finds the shortest path** from a source vertex to all other vertices in the Graph (single source shortest path).

This algorithm was designed and published by **Dr. Edsger W. Dijkstra**, a Dutch Computer Scientist, Software Engineer, Programmer, Science Essayist, and Systems Scientist. It is a type of Greedy Algorithm that only works on Weighted Graphs having positive weights. The time complexity of Dijkstra's Algorithm is  $O(V^2)$  with the help of the adjacency matrix representation of the graph. This time complexity can be reduced to  $O((V + E) \log V)$  with the help of an adjacency list representation of the graph, where  $V$  is the number of vertices and  $E$  is the number of edges in the graph.

Let source vertex be  $a$ .

We will find the shortest path to all the other vertices.

## Unit 7: Graphs

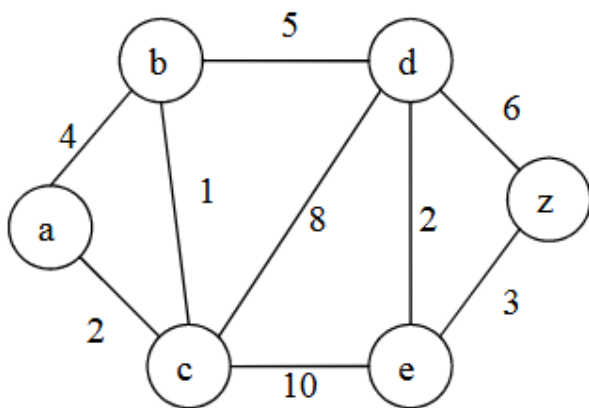


Vertex	Cost	Path
b	1	a, b
c	4	a, b, c
d	2	a, d
e	4	a, b, e
z	6	a, b, e, z

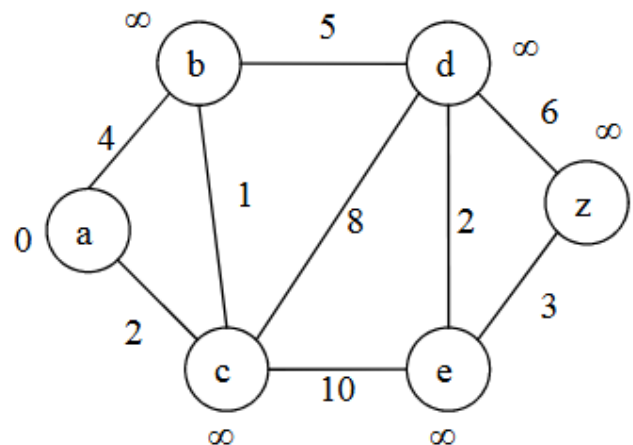
### Dijkstra's Algorithm to find the shortest path

1. Mark all the vertices as unknown
2. for each vertex  $u$  keep a distance  $d_u$  from source vertex  $s$  to  $u$  initially set to infinity except for  $s$  which is set to  $d_s = 0$
3. repeat these steps until all vertices are known
  - i. select a vertex  $u$ , which has the smallest  $d_u$  among all the unknown vertices
  - ii. mark  $u$  as visited
  - iii. for each vertex  $v$  adjacent to  $u$ 
    - if  $v$  is unvisited and  $d_u + \text{cost}(u, v) < d_v$
    - update  $d_v$  to  $d_u + \text{cost}(u, v)$

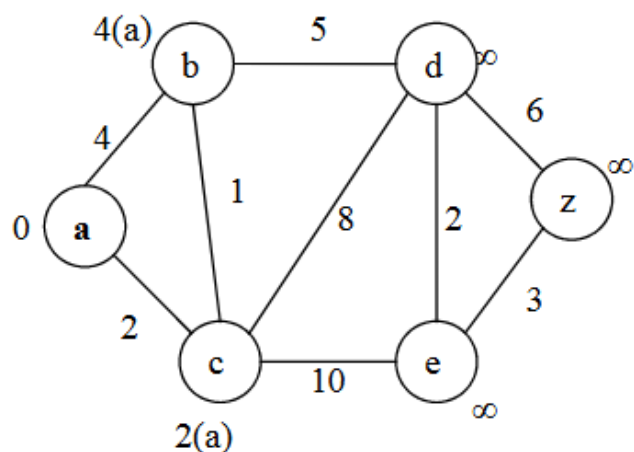
Example:



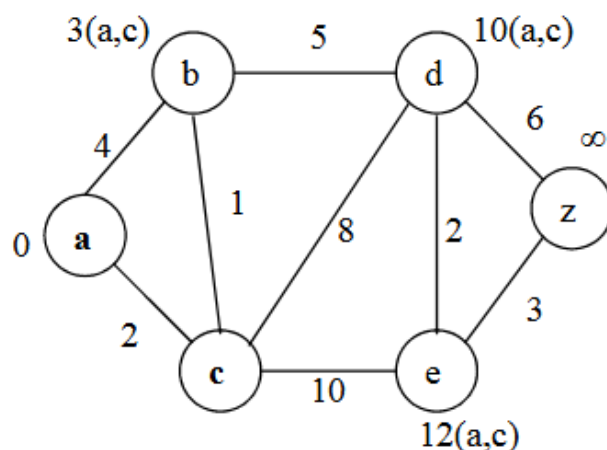
Given graph with weights



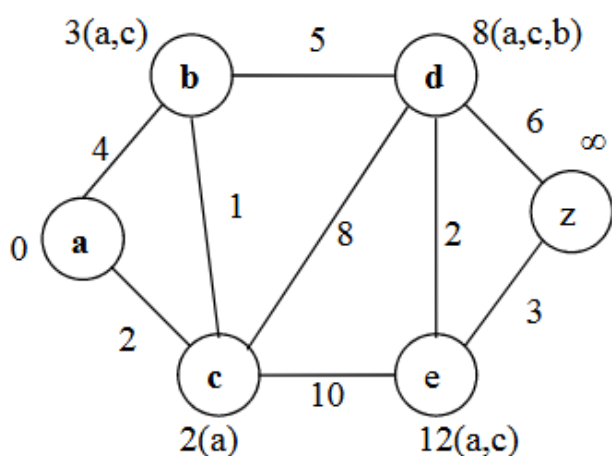
(a)



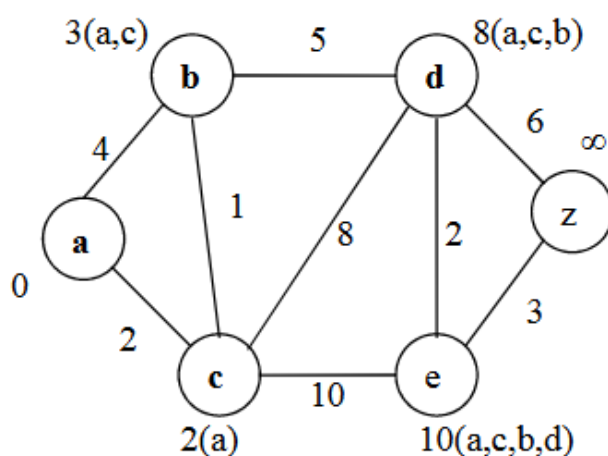
(b)



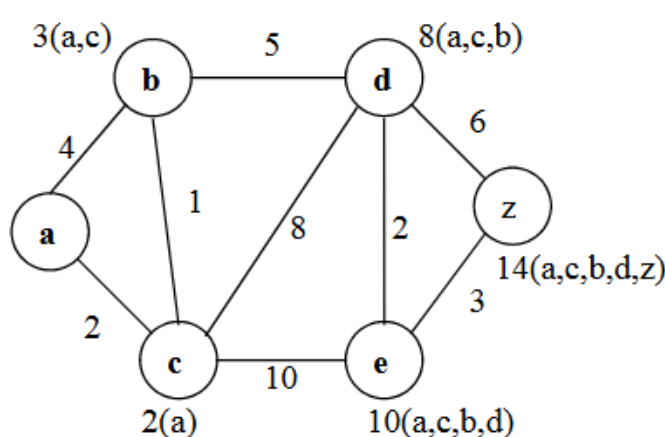
(c)



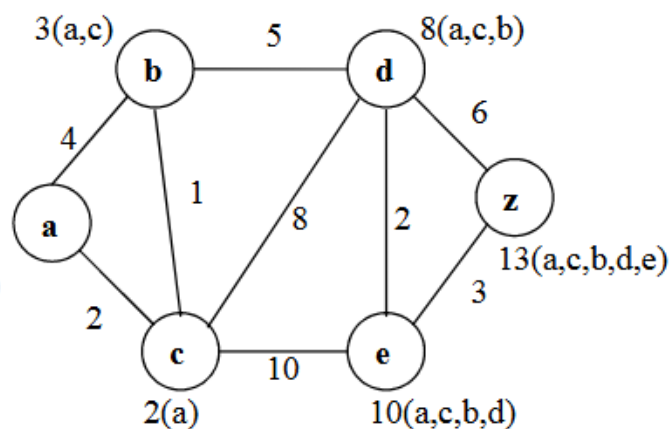
(d)



(e)



(f)



(g)

**Fig: Using Dijkstra's Algorithm to find shortest path from a to z**

See this: <https://www.analyticssteps.com/blogs/dijkstras-algorithm-shortest-path-algorithm>

### **Program to implement Dijkstra's Algorithm**

```
#include <iostream>

using namespace std;

void dijkstra(int n, int cost[10][10], int source, int dest, int d[], int p[])
{
    int i, j, u, v, min, s[10];
    for (i = 0; i < n; i++)
    {
        d[i] = cost[source][i];
        s[i] = 0;
        p[i] = source;
    }
    s[source] = 1;
    for (i = 0; i < n; i++)
    {
        min = 999;
        u = -1;
        for (j = 0; j < n; j++)
        {
            if (s[j] == 0)
            {
                if (d[j] < min)
                {
                    min = d[j];
                    u = j;
                }
            }
        }
        if (u == -1)
            return;
        s[u] = 1;
        if (u == dest)
            return;
        for (v = 0; v < n; v++)
        {
            if (s[v] == 0)
            {
                if (d[u] + cost[u][v] < d[v])
                {
                    d[v] = d[u] + cost[u][v];
                    p[v] = u;
                }
            }
        }
    }
}
```

```
}

void print_path(int source, int destination, int d[], int p[])
{
    int i;
    i = destination;
    while (i != source)
    {
        cout << i << "<- ";
        i = p[i];
    }
    cout << source << "=" << d[destination] << endl;
}

int main()
{
    int cost[10][10], n, d[10], p[10], i, j, src;

    cout << "Enter the number of nodes in the graph: ";
    cin >> n;
    cout << "Enter the cost adjacency matrix:\n";
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            cin >> cost[i][j];
        }
    }
    cout << "Enter the source vertex: ";
    cin >> src;

    for (j = 0; j < n; j++)
    {
        dijkstra(n, cost, src, j, d, p);
        if (d[j] == 999)
            cout << j << " is not reachable from " << src << endl;
        else if (src != j)
            print_path(src, j, d, p);
    }
    return 0;
}
```

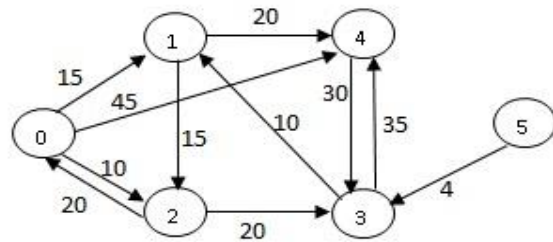
### **OUTPUT:**



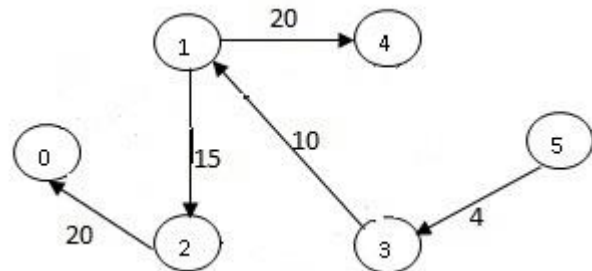
```

Enter the no of nodes in the graph
6
Enter the cost adjacency matrix
0 15 10 999 45 999
999 0 15 999 20 999
20 999 0 20 999 999
999 10 999 0 35 999
999 999 999 30 0 999
999 999 999 4 999 0
enter the source vertex
5
0<-2<-1<-3<-5=49
1<-3<-5=14
2<-1<-3<-5=29
3<-5=4
4<-1<-3<-5=34
    
```

**Input Graph:**



**Output Graph:**



## Implementation of Kruskal's Algorithm

```

#include <iostream>
using namespace std;
int find(int b, int s[]) {
    while (s[b] != b)
        b = s[b];
    return b;
}

void kruskal(int n, int cost[10][10]) {
    int count, i, s[10], min, j, u, v, k, t[10][2], sum;
    for (i = 0; i < n; i++)
        s[i] = i;
    count = 0;
    sum = 0;
    k = 0;
    while (count < n - 1) {
        min = 999;
        for (i = 0; i < n; i++) {
            for (j = 0; j < n; j++) {
                if (cost[i][j] != 0 && cost[i][j] < min) {
                    min = cost[i][j];
                    u = i;
                    v = j;
                }
            }
        }
        if (min == 999)
            break;
        i = find(u, s);
        j = find(v, s);
        if (i != j) {
    
```

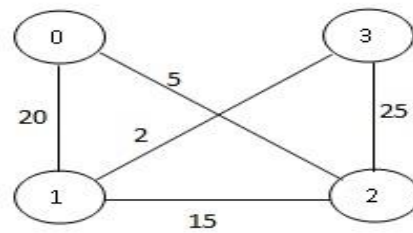
```
t[k][0] = u;
t[k][1] = v;
k++;
count++;
sum = sum + min;
s[j] = i;
}
cost[u][v] = cost[v][u] = 999;
}
if (count == n - 1) {
    cout << "Cost of spanning tree = " << sum << endl;
    cout << "Spanning Tree is shown below" << endl;
    for (k = 0; k < n - 1; k++) {
        cout << t[k][0] << " -> " << t[k][1] << endl;
    }
} else {
    cout << "Spanning Tree does not exist" << endl;
}
}

int main() {
    int n, cost[10][10], i, j;
    cout << "Enter the number of nodes: ";
    cin >> n;
    cout << "Enter the cost adjacency matrix:\n";
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            cin >> cost[i][j];
        }
    }
    kruskal(n, cost);
    return 0;
}
```

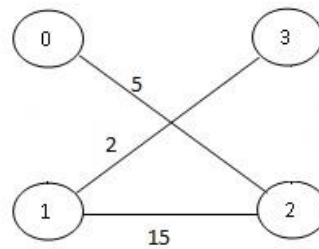
### **OUTPUT:**

```
Enter the number of nodes
4
Enter the adjacency matrix
0 20 2 999
20 0 15 5
2 15 0 25
999 5 25 999
Cost of spanning tree = 22
Spanning Tree is shown below
0 -> 2
1 -> 3
1 -> 2
```

**Input Graph:**



**Output Graph:**



### 1. Implementation of breadth-first search to traverse a graph

```
#include <iostream>
#include <queue>
```

```
const int n = 6; // Number of vertices in the graph
```

```
using namespace std;
```

```
void BFTraversal(int adjMatrix[][n], int startVertex) {
    bool visited[n] = {0};
    queue<int> q;

    q.push(startVertex);
    while (!q.empty()) {
        int vertex = q.front();
        q.pop();

        if (!visited[vertex]) {
            visited[vertex] = 1;
            cout << "Visited: " << vertex << endl;

            for (int w = 0; w < n; ++w) {
                if (adjMatrix[vertex][w] == 1 && !visited[w]) {
                    q.push(w);
                }
            }
        }
    }
}
```

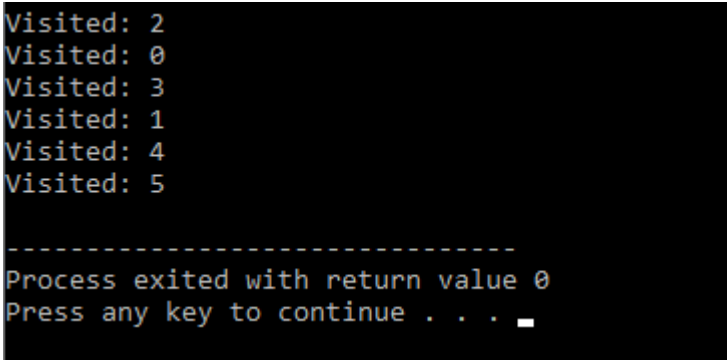
```
int main() {
    // Sample usage with a predefined graph of 6 vertices (adjacency matrix)
```

```
int adjMatrix[n][n] = {
    {0, 1, 1, 0, 0, 0},
    {1, 0, 0, 1, 0, 0},
    {1, 0, 0, 1, 0, 0},
    {0, 1, 1, 0, 1, 1},
    {0, 0, 0, 1, 0, 0},
    {0, 0, 0, 1, 0, 0}
};

// Start the Breadth-First Traversal from vertex 0
BFTraversal(adjMatrix, 2);

return 0;
}
```

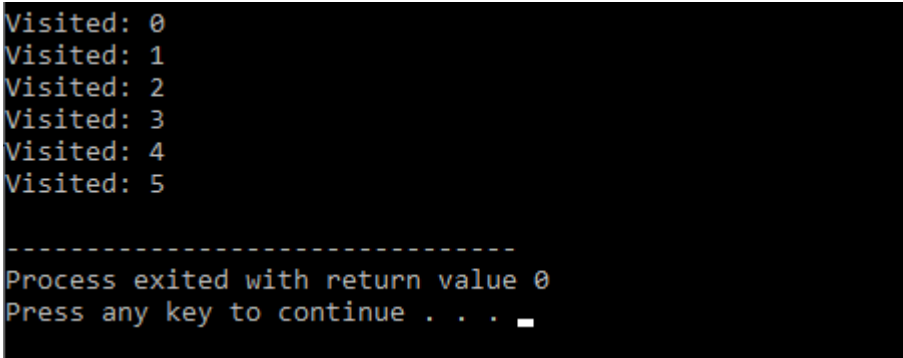
### Output 1 (Source = 2):



```
Visited: 2
Visited: 0
Visited: 3
Visited: 1
Visited: 4
Visited: 5

-----
Process exited with return value 0
Press any key to continue . . .
```

### Output 2 (Source = 0):



```
Visited: 0
Visited: 1
Visited: 2
Visited: 3
Visited: 4
Visited: 5

-----
Process exited with return value 0
Press any key to continue . . .
```