**GRAPHS**

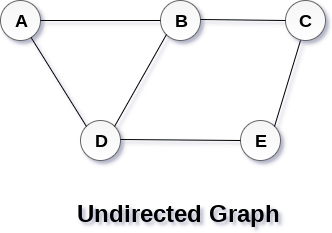
# **Graph**

**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.**

****

**What is graph , how it is differ from tree ?**

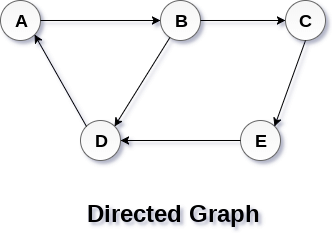
**A graph is a set of vertices/nodes and edges. A tree is a set of nodes and edges. In the graph, there is no unique node which is known as root.**

## Directed and Undirected Graph

**A graph can be directed or undirected. However, in an undirected graph, edges are not associated with the directions with them. An undirected graph is shown in the above figure since its edges are not attached with any of the directions. If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.**

**In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called initial node while node B is called terminal node.**

**A directed graph is shown in the following figure.**

****

## 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 V0=VN.**

### **Simple Path**

**If all the nodes of the graph are distinct with an exception V0=VN, 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 contain 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.**

# **Graph Representation**

**By Graph representation, we simply mean the technique which is to be used in order to store some graph into the computer's memory.**

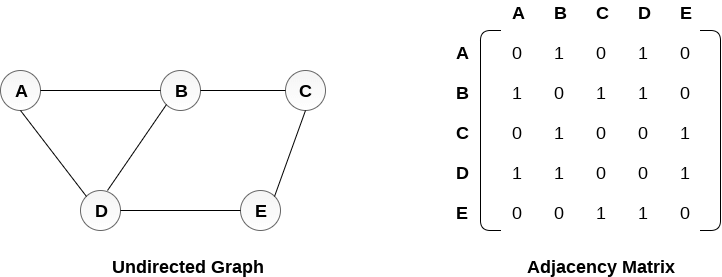
**There are two ways to store Graph into the computer's memory. In this part of this tutorial, we discuss each one of them in detail.**

## 1. Sequential Representation

**In sequential representation, we use adjacency matrix to store the mapping represented by vertices and edges. In adjacency matrix, the rows and columns are represented by the graph vertices. A graph having n vertices, will have a dimension n x n.**

**An entry Mij in the adjacency matrix representation of an undirected graph G will be 1 if there exists an edge between Vi and Vj.**

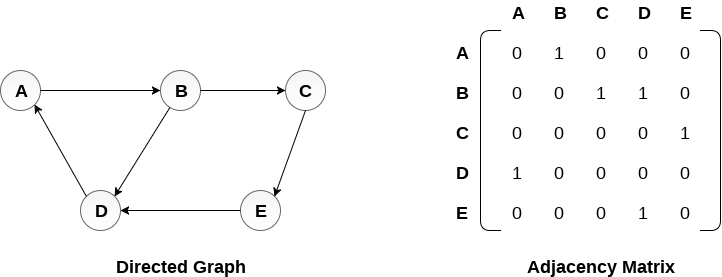
**An undirected graph and its adjacency matrix representation is shown in the following figure.**

****

**in the above figure, we can see the mapping among the vertices (A, B, C, D, E) is represented by using the adjacency matrix which is also shown in the figure.**

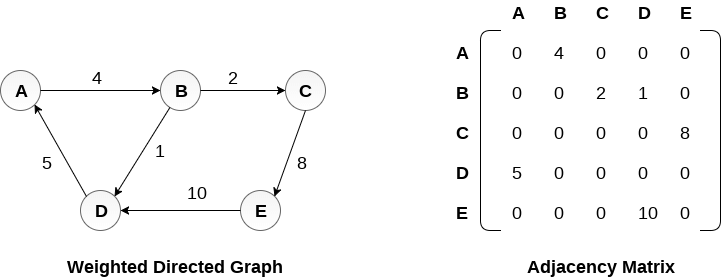
**There exists different adjacency matrices for the directed and undirected graph. In directed graph, an entry Aij will be 1 only when there is an edge directed from Vi to Vj.**

**A directed graph and its adjacency matrix representation is shown in the following figure.**

****

**Representation of weighted directed graph is different. Instead of filling the entry by 1, the Non- zero entries of the adjacency matrix are represented by the weight of respective edges.**

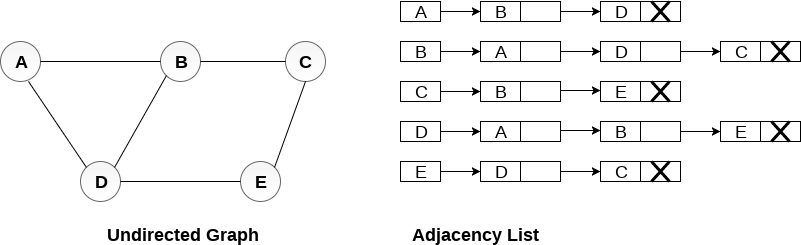
**The weighted directed graph along with the adjacency matrix representation is shown in the following figure.**

****

## Linked Representation

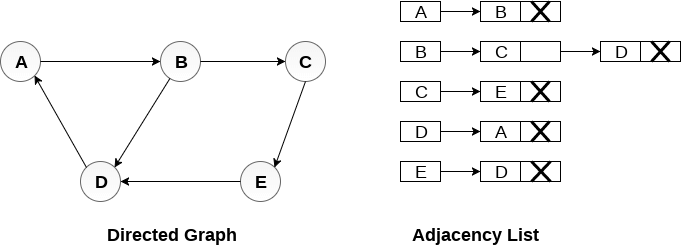
**In the linked representation, an adjacency list is used to store the Graph into the computer's memory.**

**Consider the undirected graph shown in the following figure and check the adjacency list representation.**

****

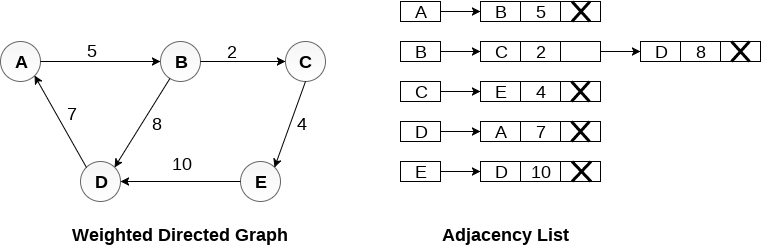
**An adjacency list is maintained for each node present in the graph which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed then store the NULL in the pointer field of last node of the list. The sum of the lengths of adjacency lists is equal to the twice of the number of edges present in an undirected graph.**

**Consider the directed graph shown in the following figure and check the adjacency list representation of the graph.**

****

**In a directed graph, the sum of lengths of all the adjacency lists is equal to the number of edges present in the graph.**

**In the case of weighted directed graph, each node contains an extra field that is called the weight of the node. The adjacency list representation of a directed graph is shown in the following figure.**

****

**What is adjacency list in graph?**

**n graph theory and computer science, an adjacency list is a collection of unordered lists used to represent a finite graph. Each unordered list within an adjacency list describes the set of neighbors of a particular vertex in the graph.**

**What is adjacency in graph?**

**In a graph, two vertices are said to be adjacent, if there is an edge between the two vertices. Here, the adjacency of vertices is maintained by the single edge that is connecting those two vertices. In a graph, two edges are said to be adjacent, if there is a common vertex between the two edges.**

**Graph Traversal**

**Graph traversal (also known as graph search) refers to the process of visiting (checking and/or updating) each vertex in a graph. Such traversals are classified by the order in which the vertices are visited.**

# **Graph Traversal Algorithm**

**In this part of the tutorial we will discuss the techniques by using which, we can traverse all the vertices of the graph.**

**Traversing the graph means examining all the nodes and vertices of the graph. There are two standard methods by using which, we can traverse the graphs. Lets discuss each one of them in detail.**

* **Breadth First Search**
* **Depth First Search**

# **Depth First Search (DFS) Algorithm**

**Depth first search (DFS) algorithm starts with the initial node of the graph G, and then goes to deeper and deeper until we find the goal node or the node which has no children. The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.**

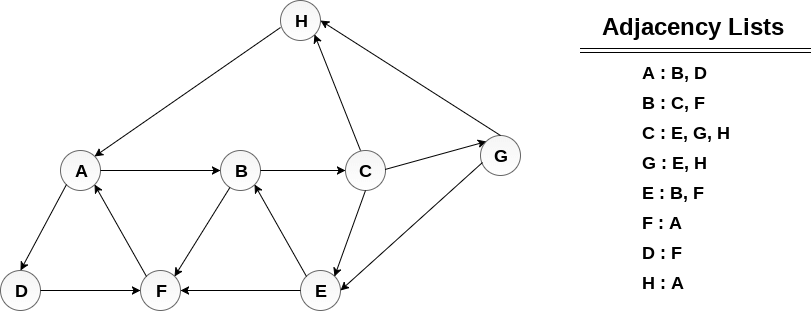
**The data structure which is being used in DFS is stack. The process is similar to BFS algorithm. In DFS, the edges that leads to an unvisited node are called discovery edges while the edges that leads to an already visited node are called block edges.**

## Algorithm

* **Step 1: SET STATUS = 1 (ready state) for each node in G**
* **Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)**
* **Step 3: Repeat Steps 4 and 5 until STACK is empty**
* **Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)**
* **Step 5: Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and set their  
  STATUS = 2 (waiting state)  
  [END OF LOOP]**
* **Step 6: EXIT**

### **Example :**

**Consider the graph G along with its adjacency list, given in the figure below. Calculate the order to print all the nodes of the graph starting from node H, by using depth first search (DFS) algorithm.**

****

### **Solution :**

**Push H onto the stack**

1. **STACK : H**

**POP the top element of the stack i.e. H, print it and push all the neighbours of H onto the stack that are is ready state.**

1. **Print H**
2. **STACK : A**

**Pop the top element of the stack i.e. A, print it and push all the neighbours of A onto the stack that are in ready state.**

1. **Print A**
2. **Stack : B, D**

**Pop the top element of the stack i.e. D, print it and push all the neighbours of D onto the stack that are in ready state.**

1. **Print D**
2. **Stack : B, F**

**Pop the top element of the stack i.e. F, print it and push all the neighbours of F onto the stack that are in ready state.**

1. **Print F**
2. **Stack : B**

**Pop the top of the stack i.e. B and push all the neighbours**

1. **Print B**
2. **Stack : C**

**Pop the top of the stack i.e. C and push all the neighbours.**

1. **Print C**
2. **Stack : E, G**

**Pop the top of the stack i.e. G and push all its neighbours.**

1. **Print G**
2. **Stack : E**

**Pop the top of the stack i.e. E and push all its neighbours.**

1. **Print E**
2. **Stack :**

**Hence, the stack now becomes empty and all the nodes of the graph have been traversed.**

**The printing sequence of the graph will be :**

1. **H → A → D → F → B → C → G → E**

## Breadth First Search (BFS) Algorithm

**Breadth first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighbouring nodes. Then, it selects the nearest node and explore all the unexplored nodes. The algorithm follows the same process for each of the nearest node until it finds the goal.**

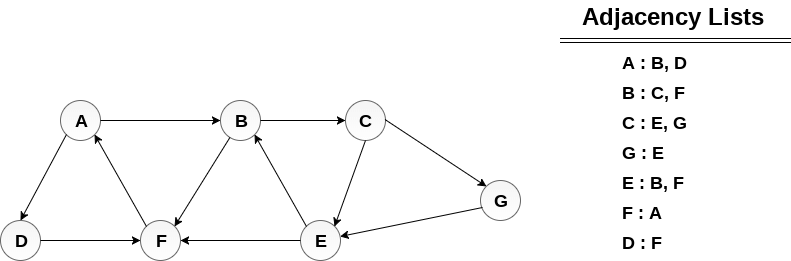
**The algorithm of breadth first search is given below. The algorithm starts with examining the node A and all of its neighbours. In the next step, the neighbours of the nearest node of A are explored and process continues in the further steps. The algorithm explores all neighbours of all the nodes and ensures that each node is visited exactly once and no node is visited twice.**

## Algorithm

* **Step 1: SET STATUS = 1 (ready state)  
  for each node in G**
* **Step 2: Enqueue the starting node A  
  and set its STATUS = 2  
  (waiting state)**
* **Step 3: Repeat Steps 4 and 5 until  
  QUEUE is empty**
* **Step 4: Dequeue a node N. Process it  
  and set its STATUS = 3  
  (processed state).**
* **Step 5: Enqueue all the neighbours of  
  N that are in the ready state  
  (whose STATUS = 1) and set  
  their STATUS = 2  
  (waiting state)  
  [END OF LOOP]**
* **Step 6: EXIT**

### **Example**

**Consider the graph G shown in the following image, calculate the minimum path p from node A to node E. Given that each edge has a length of 1.**

****

## Solution:

**Minimum Path P can be found by applying breadth first search algorithm that will begin at node A and will end at E. the algorithm uses two queues, namely QUEUE1 and QUEUE2. QUEUE1 holds all the nodes that are to be processed while QUEUE2 holds all the nodes that are processed and deleted from QUEUE1.**

**Lets start examining the graph from Node A.**

**1. Add A to QUEUE1 and NULL to QUEUE2.**

1. **QUEUE1 = {A}**
2. **QUEUE2 = {NULL}**

**2. Delete the Node A from QUEUE1 and insert all its neighbours. Insert Node A into QUEUE2**

1. **QUEUE1 = {B, D}**
2. **QUEUE2 = {A}**

**3. Delete the node B from QUEUE1 and insert all its neighbours. Insert node B into QUEUE2.**

1. **QUEUE1 = {D, C, F}**
2. **QUEUE2 = {A, B}**

**4. Delete the node D from QUEUE1 and insert all its neighbours. Since F is the only neighbour of it which has been inserted, we will not insert it again. Insert node D into QUEUE2.**

1. **QUEUE1 = {C, F}**
2. **QUEUE2 = { A, B, D}**

**5. Delete the node C from QUEUE1 and insert all its neighbours. Add node C to QUEUE2.**

1. **QUEUE1 = {F, E, G}**
2. **QUEUE2 = {A, B, D, C}**

**6. Remove F from QUEUE1 and add all its neighbours. Since all of its neighbours has already been added, we will not add them again. Add node F to QUEUE2.**

1. **QUEUE1 = {E, G}**
2. **QUEUE2 = {A, B, D, C, F}**

**7. Remove E from QUEUE1, all of E's neighbours has already been added to QUEUE1 therefore we will not add them again. All the nodes are visited and the target node i.e. E is encountered into QUEUE2.**

1. **QUEUE1 = {G}**
2. **QUEUE2 = {A, B, D, C, F,  E}**

**Now, backtrack from E to A, using the nodes available in QUEUE2.**

**The minimum path will be A → B → C → E.**

### **What is a spanning tree?**

**A spanning tree can be defined as the subgraph of an undirected connected graph. It includes all the vertices along with the least possible number of edges. If any vertex is missed, it is not a spanning tree. A spanning tree is a subset of the graph that does not have cycles, and it also cannot be disconnected.**

**A spanning tree consists of (n-1) edges, where 'n' is the number of vertices (or nodes). Edges of the spanning tree may or may not have weights assigned to them. All the possible spanning trees created from the given graph G would have the same number of vertices, but the number of edges in the spanning tree would be equal to the number of vertices in the given graph minus 1.**

**A complete undirected graph can have nn-2 number of spanning trees where n is the number of vertices in the graph. Suppose, if n = 5, the number of maximum possible spanning trees would be 55-2 = 125.**

### **Applications of the spanning tree**

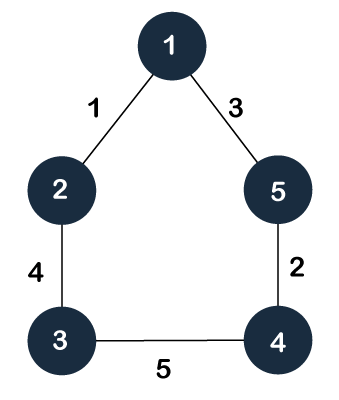
**Basically, a spanning tree is used to find a minimum path to connect all nodes of the graph. Some of the common applications of the spanning tree are listed as follows -**

* **Cluster Analysis**
* **Civil network planning**
* **Computer network routing protocol**

**Now, let's understand the spanning tree with the help of an example.**

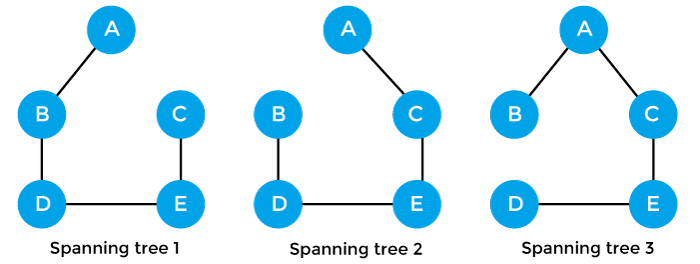
### **Example of Spanning tree**

**Suppose the graph be -**

****

**As discussed above, a spanning tree contains the same number of vertices as the graph, the number of vertices in the above graph is 5; therefore, the spanning tree will contain 5 vertices. The edges in the spanning tree will be equal to the number of vertices in the graph minus 1. So, there will be 4 edges in the spanning tree.**

**Some of the possible spanning trees that will be created from the above graph are given as follows -**

****

## Properties of spanning-tree

**Some of the properties of the spanning tree are given as follows -**

* **There can be more than one spanning tree of a connected graph G.**
* **A spanning tree does not have any cycles or loop.**
* **A spanning tree is minimally connected, so removing one edge from the tree will make the graph disconnected.**
* **A spanning tree is maximally acyclic, so adding one edge to the tree will create a loop.**
* **There can be a maximum nn-2 number of spanning trees that can be created from a complete graph.**
* **A spanning tree has n-1 edges, where 'n' is the number of nodes.**
* **If the graph is a complete graph, then the spanning tree can be constructed by removing maximum (e-n+1) edges, where 'e' is the number of edges and 'n' is the number of vertices.**

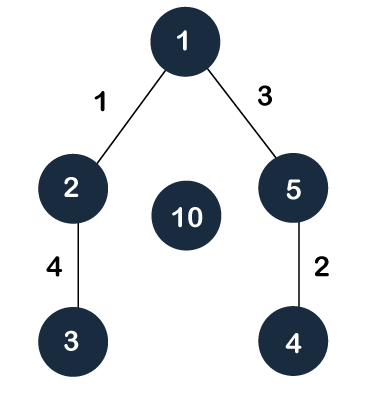
**So, a spanning tree is a subset of connected graph G, and there is no spanning tree of a disconnected graph.**

## Minimum Spanning tree

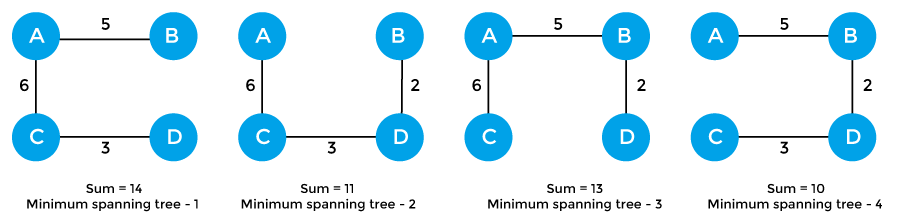
**A minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree. In the real world, this weight can be considered as the distance, traffic load, congestion, or any random value.**

### **Example of minimum spanning tree**

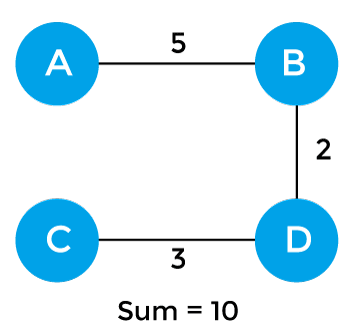
**Let's understand the minimum spanning tree with the help of an example.**

****

**The sum of the edges of the above graph is 16. Now, some of the possible spanning trees created from the above graph are -**

****

**So, the minimum spanning tree that is selected from the above spanning trees for the given weighted graph is -**

****

### **Applications of minimum spanning tree**

**The applications of the minimum spanning tree are given as follows -**

* **Minimum spanning tree can be used to design water-supply networks, telecommunication networks, and electrical grids.**
* **It can be used to find paths in the map.**

**Prim's Algorithm**

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

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

## How does the prim's algorithm work?

**Prim's algorithm is a greedy algorithm that starts from one vertex and continue to add the edges with the smallest weight until the goal is reached. The steps to implement the prim's algorithm are given as follows -**

* **First, we have to initialize an MST with the randomly chosen vertex.**
* **Now, we have to find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree.**
* **Repeat step 2 until the minimum spanning tree is formed.**

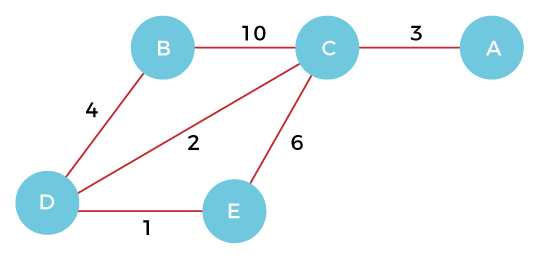
**The applications of prim's algorithm are -**

* **Prim's algorithm can be used in network designing.**
* **It can be used to make network cycles.**
* **It can also be used to lay down electrical wiring cables.**

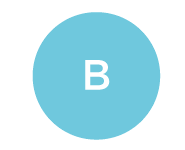
## Example of prim's algorithm

**Now, let's see the working of prim's algorithm using an example. It will be easier to understand the prim's algorithm using an example.**

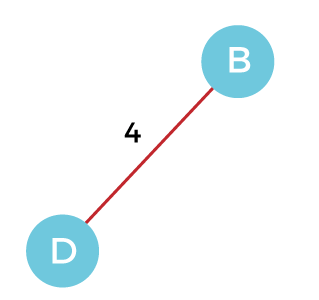
**Suppose, a weighted graph is -**

****

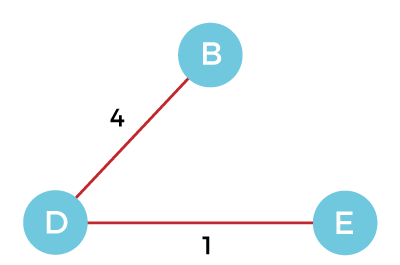
**Step 1 - First, we have to choose a vertex from the above graph. Let's choose B.**

****

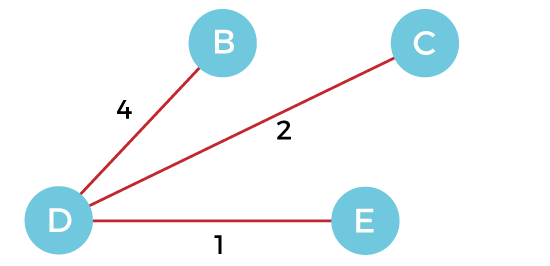
**Step 2 - Now, we have to choose and add the shortest edge from vertex B. There are two edges from vertex B that are B to C with weight 10 and edge B to D with weight 4. Among the edges, the edge BD has the minimum weight. So, add it to the MST.**

****

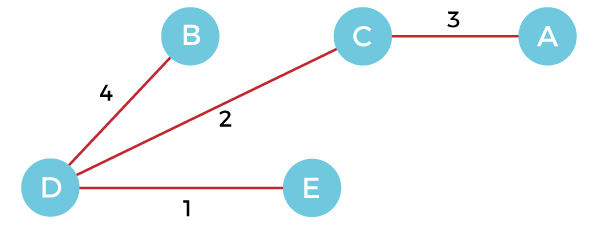
**Step 3 - Now, again, choose the edge with the minimum weight among all the other edges. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C, i.e., E and A. So, select the edge DE and add it to the MST.**

****

**Step 4 - Now, select the edge CD, and add it to the MST.**

****

**Step 5 - Now, choose the edge CA. Here, we cannot select the edge CE as it would create a cycle to the graph. So, choose the edge CA and add it to the MST.**

****

**So, the graph produced in step 5 is the minimum spanning tree of the given graph. The cost of the MST is given below -**

**Cost of MST = 4 + 2 + 1 + 3 = 10 units.**

## Algorithm

1. **Step 1: Select a starting vertex**
2. **Step 2: Repeat Steps 3 and 4 until there are fringe vertices**
3. **Step 3: Select an edge 'e' connecting the tree vertex and fringe vertex that has minimum weight**
4. **Step 4: Add the selected edge and the vertex to the minimum spanning tree T**
5. **[END OF LOOP]**
6. **Step 5: EXIT**

## Complexity of Prim's algorithm

**Now, let's see the time complexity of Prim's algorithm. The running time of the prim's algorithm depends upon using the data structure for the graph and the ordering of edges. Below table shows some choices -**

* **Time Complexity**

|  |  |
| --- | --- |
| **Data structure used for the minimum edge weight** | **Time Complexity** |
| **Adjacency matrix, linear searching** | **O(|V|2)** |
| **Adjacency list and binary heap** | **O(|E| log |V|)** |
| **Adjacency list and Fibonacci heap** | **O(|E|+ |V| log |V|)** |

**Prim's algorithm can be simply implemented by using the adjacency matrix or adjacency list graph representation, and to add the edge with the minimum weight requires the linearly searching of an array of weights. It requires O(|V|2) running time. It can be improved further by using the implementation of heap to find the minimum weight edges in the inner loop of the algorithm.**

**The time complexity of the prim's algorithm is O(E logV) or O(V logV), where E is the no. of edges, and V is the no. of vertices.**

**Kruskal's Algorithm**

**Kruslal’s Algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which we can traverse every vertex of the graph. It follows the greedy approach that finds an optimum solution at every stage instead of focusing on a global optimum.**

## How does Kruskal's algorithm work?

**In Kruskal's algorithm, we start from edges with the lowest weight and keep adding the edges until the goal is reached. The steps to implement Kruskal's algorithm are listed as follows -**

* **First, sort all the edges from low weight to high.**
* **Now, take the edge with the lowest weight and add it to the spanning tree. If the edge to be added creates a cycle, then reject the edge.**
* **Continue to add the edges until we reach all vertices, and a minimum spanning tree is created.**

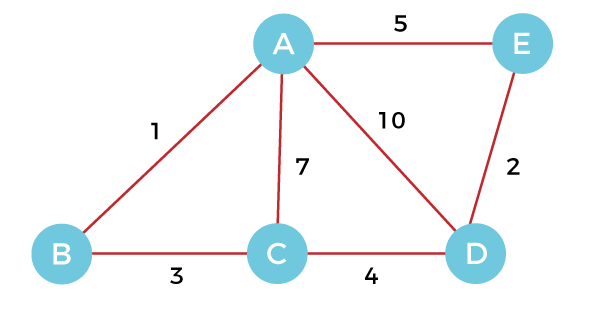
**The applications of Kruskal's algorithm are -**

* **Kruskal's algorithm can be used to layout electrical wiring among cities.**
* **It can be used to lay down LAN connections.**

## Example of Kruskal's algorithm

**Now, let's see the working of Kruskal's algorithm using an example. It will be easier to understand Kruskal's algorithm using an example.**

**Suppose a weighted graph is -**

****

**The weight of the edges of the above graph is given in the below table -**

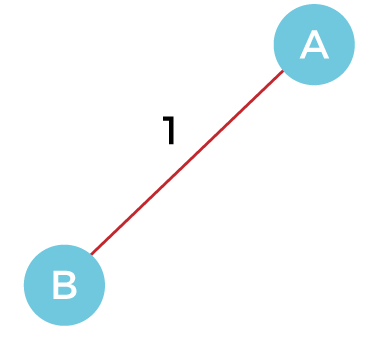
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Edge** | **AB** | **AC** | **AD** | **AE** | **BC** | **CD** | **DE** |
| **Weight** | **1** | **7** | **10** | **5** | **3** | **4** | **2** |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Edge** | **AB** | **DE** | **BC** | **CD** | **AE** | **AC** | **AD** |
| **Weight** | **1** | **2** | **3** | **4** | **5** | **7** | **10** |

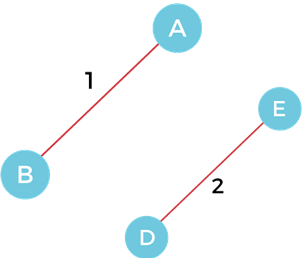
**Now, sort the edges given above in the ascending order of their weights.**

**Now, let's start constructing the minimum spanning tree.**

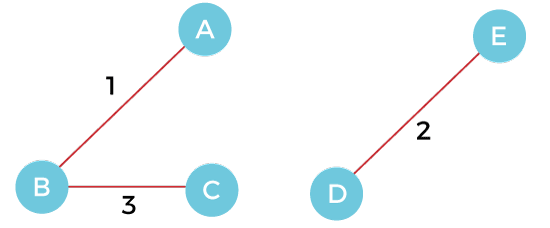
**Step 1 - First, add the edge AB with weight 1 to the MST.**

****

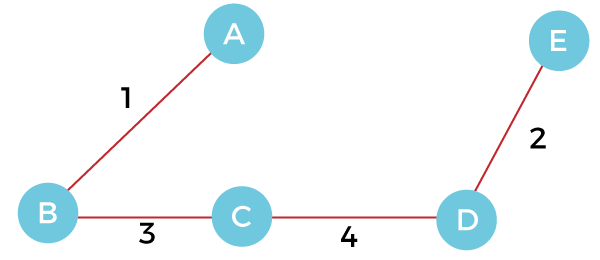
**Step 2 - Add the edge DE with weight 2 to the MST as it is not creating the cycle.**

****

**Step 3 - Add the edge BC with weight 3 to the MST, as it is not creating any cycle or loop.**

****

**Step 4 - Now, pick the edge CD with weight 4 to the MST, as it is not forming the cycle.**

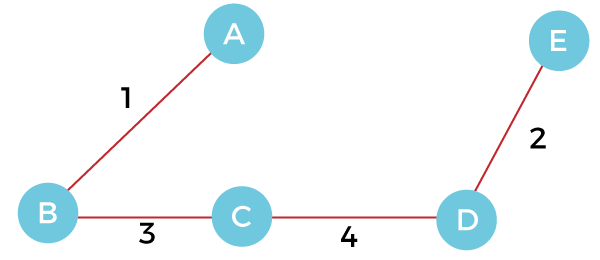
****

**Step 5 - After that, pick the edge AE with weight 5. Including this edge will create the cycle, so discard it.**

**Step 6 - Pick the edge AC with weight 7. Including this edge will create the cycle, so discard it.**

**Step 7 - Pick the edge AD with weight 10. Including this edge will also create the cycle, so discard it.**

**So, the final minimum spanning tree obtained from the given weighted graph by using Kruskal's algorithm is –**

****

**The cost of the MST is = AB + DE + BC + CD = 1 + 2 + 3 + 4 = 10.**

**Now, the number of edges in the above tree equals the number of vertices minus 1. So, the algorithm stops here.**

## Algorithm

1. **Step 1: Create a forest F in such a way that every vertex of the graph is a separate tree.**
2. **Step 2: Create a set E that contains all the edges of the graph.**
3. **Step 3: Repeat Steps 4 and 5 while E is NOT EMPTY and F is not spanning**
4. **Step 4: Remove an edge from E with minimum weight**
5. **Step 5: IF the edge obtained in Step 4 connects two different trees, then add it to the forest F**
6. **(for combining two trees into one tree).**
7. **ELSE**
8. **Discard the edge**
9. **Step 6: END**

## Complexity of Kruskal's algorithm

**Now, let's see the time complexity of Kruskal's algorithm.**

**The time complexity of Kruskal's algorithm is O(E logE) or O(V logV), where E is the no. of edges, and V is the no. of vertices.**