UNIT- TEN
GRAPHS

Topics to be Covered

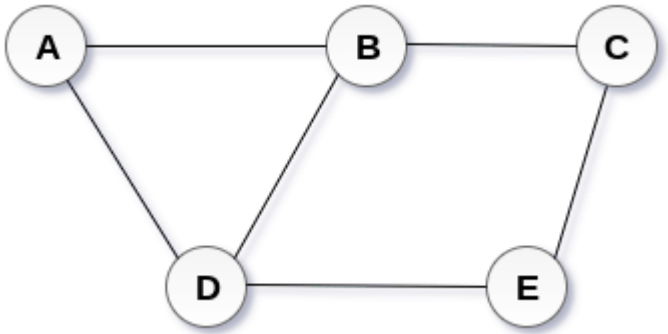
- Introduction
- Graphs as an ADT
- Transitive closure
- Warshall's Algorithm
- Types of Graph
- Graph Traversal and spanning forest
- Kruskal's and round robin algorithm
- Shortest path algorithm
- Greedy algorithm
- Dijkstra's Algorithm

Introduction: 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.

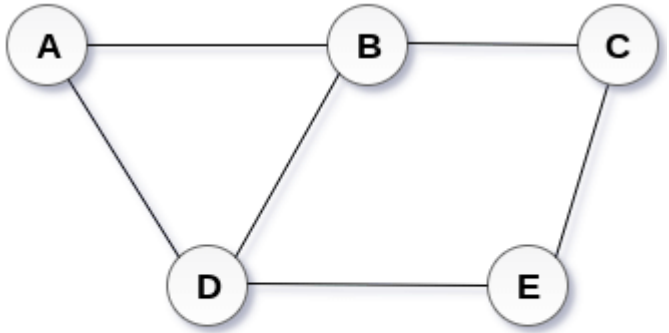
Definition

- 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.

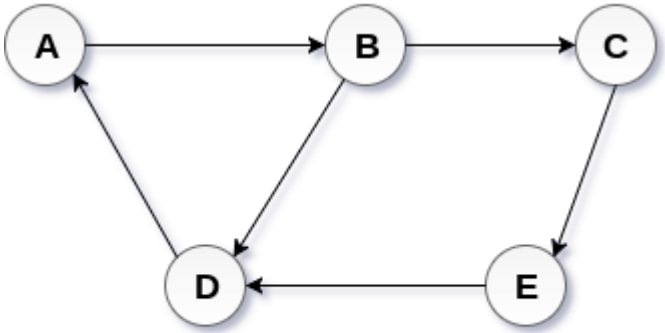


Directed and Undirected Graph:

- an undirected graph, edges are not associated with the directions with them. In undirected graph, 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.



Undirected Graph

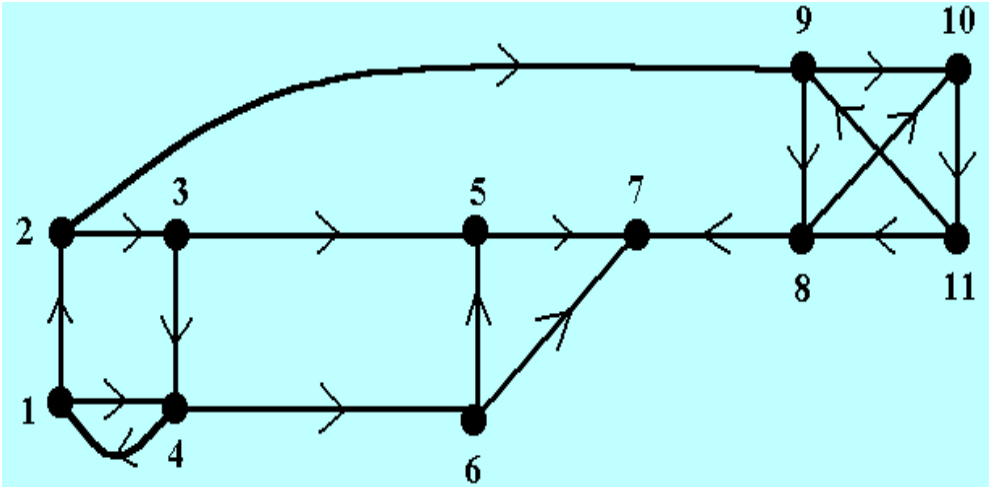


Directed Graph

Graph Terminologies

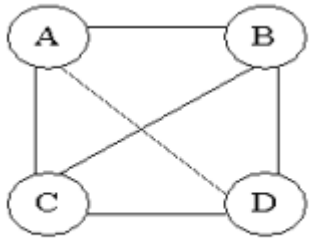
- Path: A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.
- Closed Path: A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if $V_0=V_N$.
- Simple Path: If all the 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.

In the digraph shown here,
2 - 9 - 8 - 10 - 11 - 9 - 8 - 7 is a path (neither simple nor closed)
1 - 4 - 6 - 5 - 7 is a path which is simple.
9 - 8 - 10 - 11 - 9 - 10 - 11 - 9 is a path which is closed.
1 - 2 - 3 - 4 - 1 is a cycle.



Graph Terminologies

- Connected Graph: A connected graph (figure1)is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph.
- Disconnected Graph: A graph(figure2) is disconnected if at least two vertices of the graph are not connected by a path. If a graph G is disconnected, then every maximal connected subgraph of G is called a connected component of the graph G.
- Regular Graph: Regular graph is the graph in which nodes are adjacent to each other, i.e., each node is accessible from any other node.



A regular graph

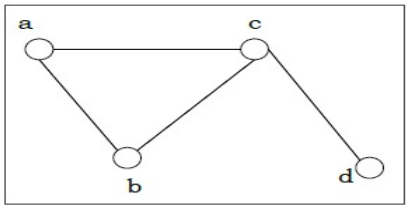


Fig: 1

Vertex 1	Vertex 2	PATH
a	b	a b
a	c	a b c, a c
a	d	a b c d, a c d
b	c	b a c , b c
c	d	c d

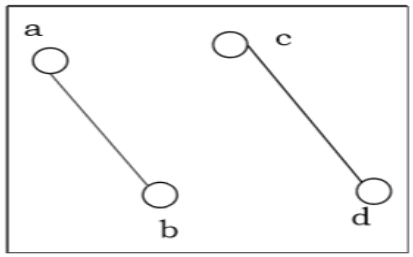


Fig: 2

Vertex 1	Vertex 2	PATH
a	b	a b
a	c	Not Available
a	d	Not Available
b	c	Not Available
c	d	c d

- Complete Graph: A complete graph (figure1) 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(figure2), 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.

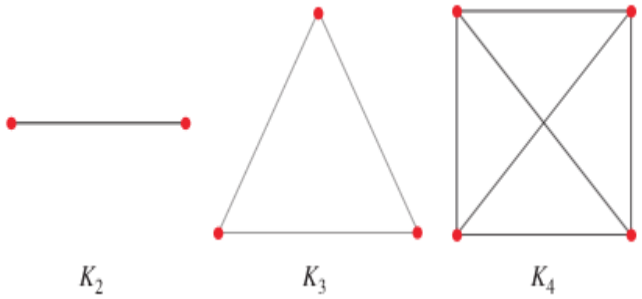


Figure:1

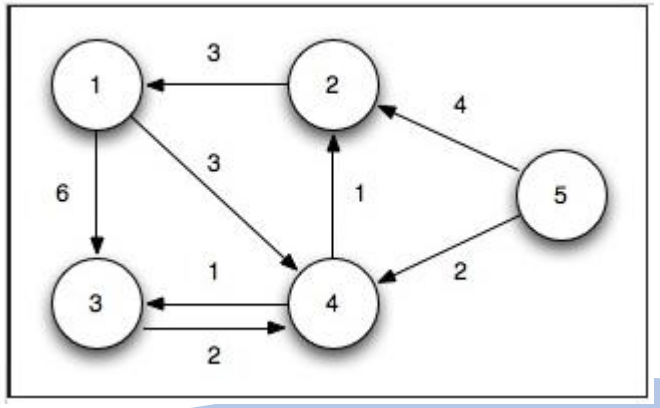


Figure 2

- 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.

- Indegree :Total number of incoming edges connected to a vertex is said to be indegree of that vertex.
- Outdegree: Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

deg(a) = 2, as there are 2 edges meeting at vertex ‘a’.
deg(b) = 3, as there are 3 edges meeting at vertex ‘b’.
deg(c) = 1, as there is 1 edge formed at vertex ‘c’
So ‘c’ is a **pendent vertex**.
deg(d) = 2, as there are 2 edges meeting at vertex ‘d’.
deg(e) = 0, as there are 0 edges formed at vertex ‘e’.
So ‘e’ is an **isolated vertex**.

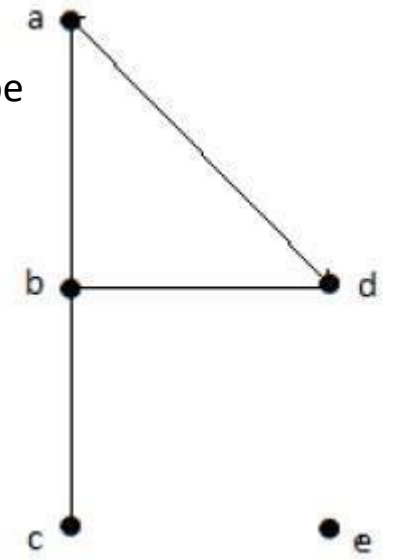


Figure: undirected graph

Graph as an ADT:

Structure Graph is

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

functions: for all graph ε Graph, v, v1 and v2 ε Vertices

Graph Create()::=return an empty graph

Graph InsertVertex(graph, v)::= return a graph with v inserted. v has no edge.

Graph InsertEdge(graph, v1,v2)::= return a graph with new edge between v1 and v2

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

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

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

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

Graph Representations:

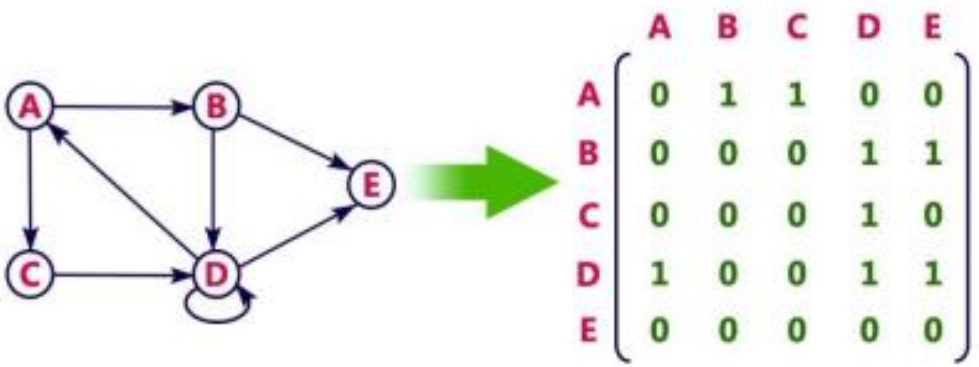
Graph data structure is represented using following representations

1. Adjacency Matrix
2. Adjacency List

Adjacency Matrix:

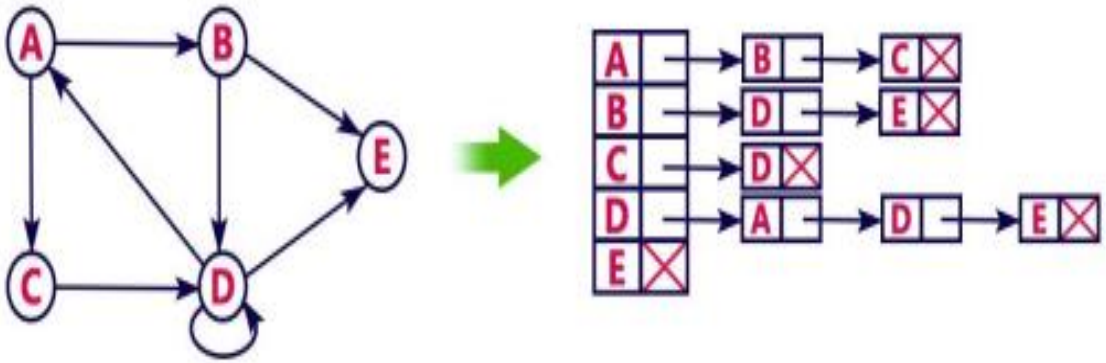
- In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices; means if a graph with 4 vertices can be represented using a matrix of 4X4 size.
- In this matrix, rows and columns both represent vertices. This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.
- Adjacency Matrix : let G = (V, E) with n vertices, n > 1. The adjacency matrix of G is a 2-dimensional n × n matrix, A, A(i, j) = 1 iff (vi , vj) εE(G) (<vi , vj> for a diagraph), A(i, j) = 0 otherwise.

- For a directed graph : Adjacency matrix



Adjacency list:

- In this representation, every vertex of graph contains list of its adjacent vertices. The n rows of the adjacency matrix are represented as n chains. The nodes in chain I represent the vertices that are adjacent to vertex i.
- It can be represented in two forms. In one form, array is used to store n vertices and chain is used to store its adjacencies. Example:



C implementation of a graph

Consider a **non-weighted graph** with number of vertices MAX=20

Syntax:

```
#define MAX 20
struct Node
{
   int vertex;
   struct Node *next;
}node1;
```

Declaring adjacency list on the basis of number of vertices:

```
Node1 *adj[MAX];
```

Consider a **weighted** graph with number of vertices MAX=20

Syntax:

```
struct Node
{
   int vertex;
   int weight;
   struct Node *next;
}node2;
```

Declaring adjacency list on the basis of number of vertices:

```
Node2 *adj[MAX];
```

Transitive Closure of a graph:

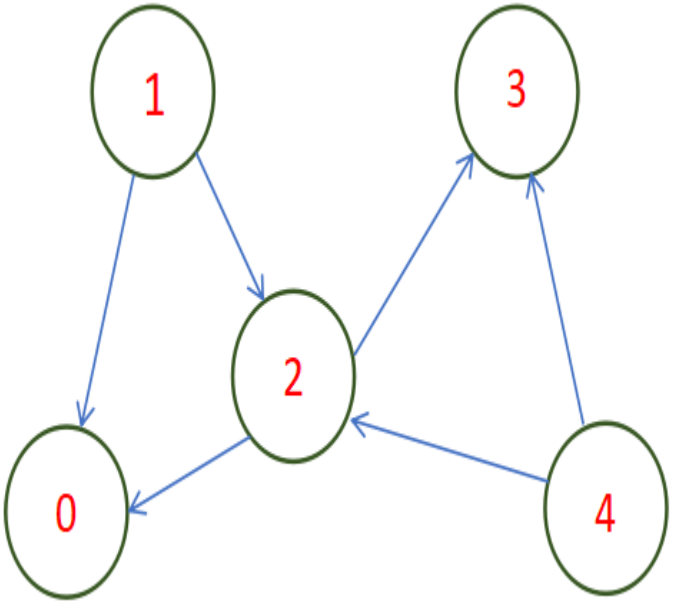
In any Directed Graph, let's consider a node i as a starting point and another node j as ending point. For all (i,j) pairs in a graph, transitive closure matrix is formed by the reachability factor, i.e if j is reachable from i (means there is a path from i to j) then we can put the matrix element as 1 or else if there is no path, then we can put it as 0.

Reachability matrix

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

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

Fig: a directed graph

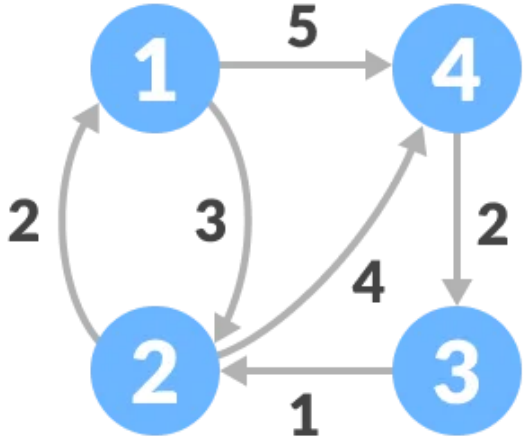


Floyd Warshall's Algorithm:

- Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).

How Floyd-Warshall Algorithm Works?

- Let the given graph be:

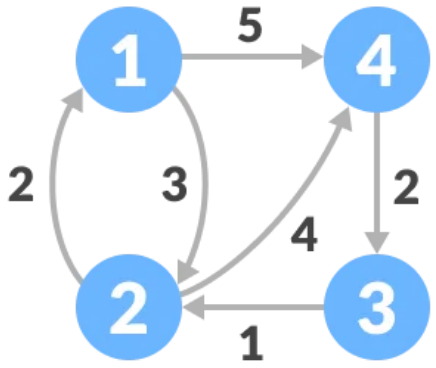


Follow the steps below to find the shortest path between all the pairs of vertices:

1. Create a matrix A^0 of dimension n*n where n is the number of vertices. The row and the column are indexed as i and j respectively. i and j are the vertices of the graph.

Each cell A[i][j] is filled with the distance from the ith vertex to the jth vertex. If there is no path from ith vertex to jth vertex, the cell is left as infinity.

$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$



Fiig: Weighted graph

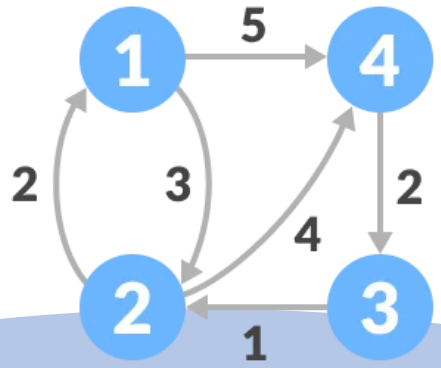
2.Now, create a matrix A^1 using matrix A^0. The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.

Let k be the intermediate vertex in the shortest path from source to destination. In this step, k is the first vertex. A[i][j] is filled with (A[i][k] + A[k][j]) if (A[i][j] > A[i][k] + A[k][j]).

That is, if the direct distance from the source to the destination is greater than the path through the vertex k, then the cell is filled with A[i][k] + A[k][j].

In this step, k is vertex 1. We calculate the distance from source vertex to destination vertex through this vertex k.

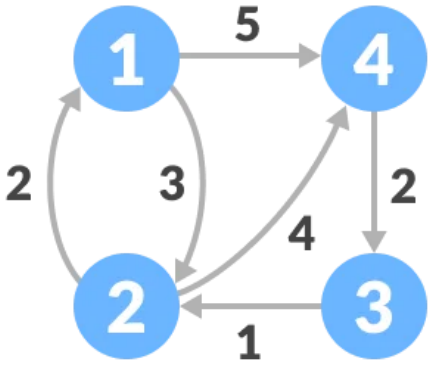
$$A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & & \\ \infty & & 0 & \\ \infty & & & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & 9 & 4 \\ \infty & 1 & 0 & 8 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$



3: Similarly, A^2 is created using A^1. The elements in the second column and the second row are left as they are.

In this step, k is the second vertex (i.e. vertex 2). The remaining steps are the same as in step 2.

$$A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & & \\ 2 & 0 & 9 & 4 \\ & 1 & 0 & \\ & \infty & & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$



Step 4: Similarly, A^3 and A^4 is also created

$$A^3 = \begin{array}{c} \begin{matrix} & 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \begin{bmatrix} 0 & & \infty & \\ & 0 & 9 & \\ \infty & 1 & 0 & 8 \\ & & 2 & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \begin{matrix} & 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

$$A^4 = \begin{array}{c} \begin{matrix} & 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \begin{bmatrix} 0 & & & 5 \\ & 0 & & 4 \\ & & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \begin{matrix} & 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \begin{bmatrix} 0 & 3 & 7 & 5 \\ 2 & 0 & 6 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

Step 5: A^4 gives the shortest path between each pair of vertices.

Graph Traversal: Two methods

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

Difference between traversal in graph and tree

BASIS FOR COMPARISON	TREE	GRAPH
Path	Only one between two vertices.	More than one path is allowed.
Root node	It has exactly one root node.	Graph doesn't have a root node.
Loops	No loops are permitted.	Graph can have loops.
Complexity	Less complex	More complex comparatively
Traversal techniques	Pre-order, In-order and Post-order.	Breadth-first search and depth-first search.
Number of edges	n-1 (where n is the number of nodes)	Not defined
Model type	Hierarchical	Network

BFS

Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes.

Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node.

It is a recursive algorithm to search all the vertices of a tree or graph data structure.

BFS puts every vertex of the graph into two categories - **visited and non-visited.**

It selects a single node in a graph and, after that, visits all the nodes adjacent to the selected node.

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

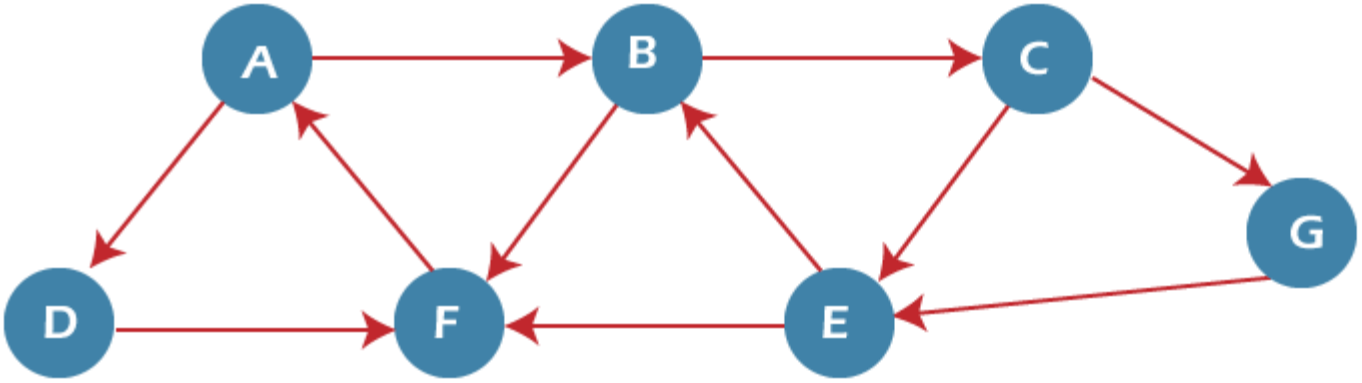
Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).

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: Stop

BFS: Example



Adjacency Lists

A : B, D
B : C, F
C : E, G
G : E
E : B, F
F : A
D : F

In the above graph, minimum path 'P' can be found by using the BFS that will start from Node A and end at Node 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.

Step 1 - First, add A to queue1 and NULL to queue2.

Status:
QUEUE1 = {A}
QUEUE2 = {NULL}

Step 2 - Now, delete node A from queue1 and add it into queue2. Insert all neighbors of node A to queue1.

Status:
QUEUE1 = {B, D}
QUEUE2 = {A}

Step 3 - Now, delete node B from queue1 and add it into queue2. Insert all neighbors of node B to queue1.

Status:
QUEUE1 = {D, C, F}
QUEUE2 = {A, B}

Step 4 - Now, delete node D from queue1 and add it into queue2. Insert all neighbors of node D to queue1. The only neighbor of Node D is F since it is already inserted, so it will not be inserted again.

Status:
QUEUE1 = {C, F}
QUEUE2 = {A, B, D}

Step 5 - Delete node C from queue1 and add it into queue2. Insert all neighbors of node C to queue1.

Status:
QUEUE1 = {F, E, G}
QUEUE2 = {A, B, D, C}

Step 6 - Delete node F from queue1 and add it into queue2. Insert all neighbors of node F to queue1. Since all the neighbors of node F are already present, we will not insert them again.

Status:
QUEUE1 = {E, G}
QUEUE2 = {A, B, D, C, F}

Step 7 - Delete node E from queue1. Since all of its neighbors have already been added, so we will not insert them again. Now, all the nodes are visited, and the target node E is encountered into queue2.

Status:
QUEUE1 = {G}
QUEUE2 = {A, B, D, C, F, E}

Final BFS traversal of graph is A,B,D,C,F,E,G

DFS

- The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.
- The step by step process to implement the DFS traversal is given as follows -
  - First, create a stack with the total number of vertices in the graph.
  - Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
  - After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
  - Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
  - If no vertex is left, go back and pop a vertex from the stack.
  - Repeat steps 2, 3, and 4 until the stack is empty.

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

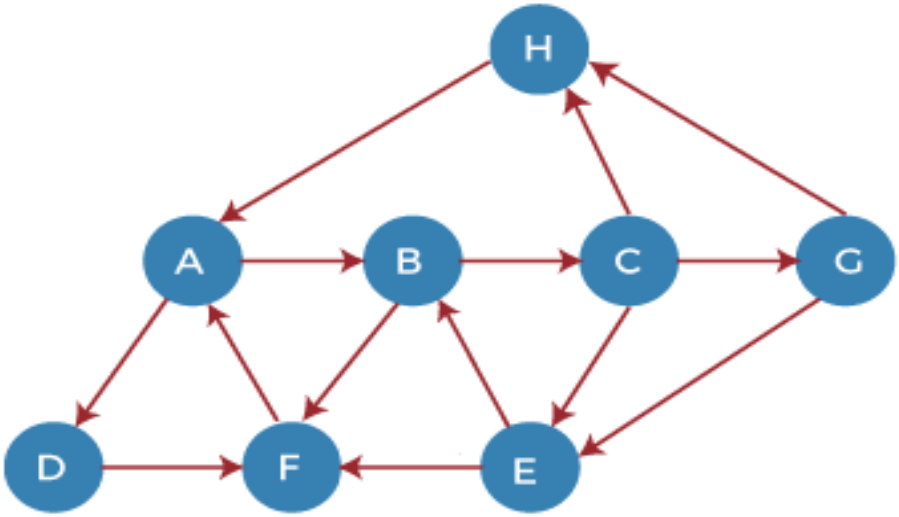
Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

Step 5: Push on the stack all the neighbors 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



Adjacency Lists

A : B, D
B : C, F
C : E, G, H
G : E, H
E : B, F
F : A
D : F
H : A

Step 1 - First, push H onto the stack.

Status:
Stack: H

Step 2 - POP the top element from the stack, i.e., H, and print it. Now, PUSH all the neighbors of H onto the stack that are in ready state.

Status:
Print: H| STACK: A

Step 3 - POP the top element from the stack, i.e., A, and print it. Now, PUSH all the neighbors of A onto the stack that are in ready state.

Status:
Print: A | STACK: B, D

Step 4 - POP the top element from the stack, i.e., D, and print it. Now, PUSH all the neighbors of D onto the stack that are in ready state.

Status:
Print: D |STACK: B, F

Step 5 - POP the top element from the stack, i.e., F, and print it. Now, PUSH all the neighbors of F onto the stack that are in ready state.

Status:
Print: F | STACK: B

Step 6 - POP the top element from the stack, i.e., B, and print it. Now, PUSH all the neighbors of B onto the stack that are in ready state.

Status:
Print: B |STACK: C

Step 7 - POP the top element from the stack, i.e., C, and print it. Now, PUSH all the neighbors of C onto the stack that are in ready state.

Status:
Print: C |STACK: E, G

Step 8 - POP the top element from the stack, i.e., G and PUSH all the neighbors of G onto the stack that are in ready state.

Status:
Print: G |STACK: E

Step 9 - POP the top element from the stack, i.e., E and PUSH all the neighbors of E onto the stack that are in ready state.

Status:
Print: E |STACK:

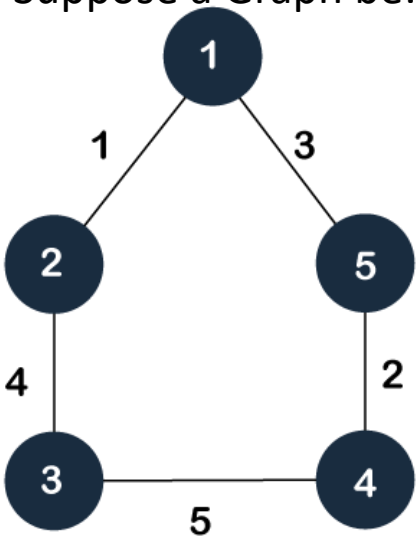
N*ow, all the graph nodes have been traversed, and the stack is empty. Final DFS traversal is H,A,D,F,B,C,G,E*

Spanning forest/ Spanning Tree

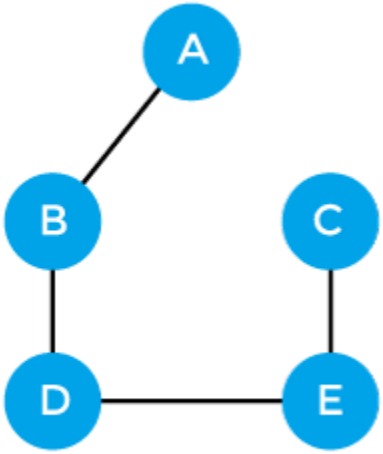
- A spanning tree can be defined as the subgraph of an undirected connected graph. It includes all the vertices along with the least possible number of edges.
- If any vertex is missed, it is not a spanning tree. A spanning tree is a subset of the graph that does not have cycles, and it also cannot be 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 $\mathbf{n^{n-2}}$ 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 $\mathbf{5^{5-2} = 125.}$

Example of Spanning tree

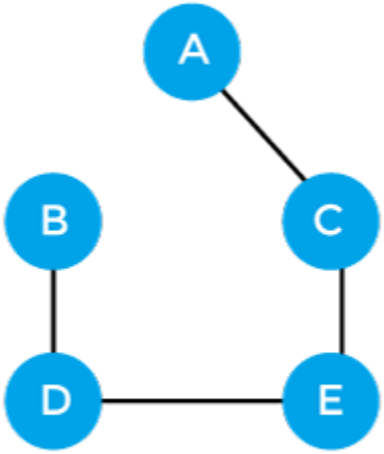
Suppose a Graph be:



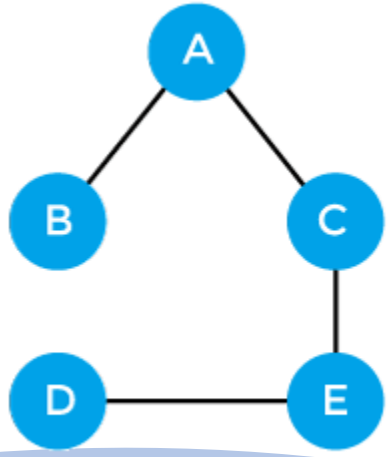
Some of the possible spanning trees that will be created from the above graph are given as follows -



Spanning tree 1



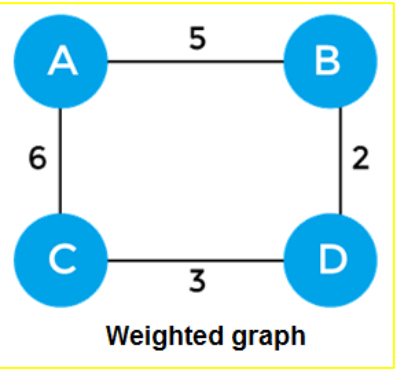
Spanning tree 2



Spanning tree 3

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.



Weighted graph

There are number of techniques for creating minimum spanning tree for weighted graph: Kruskal Algorithm, Prim's Algorithm

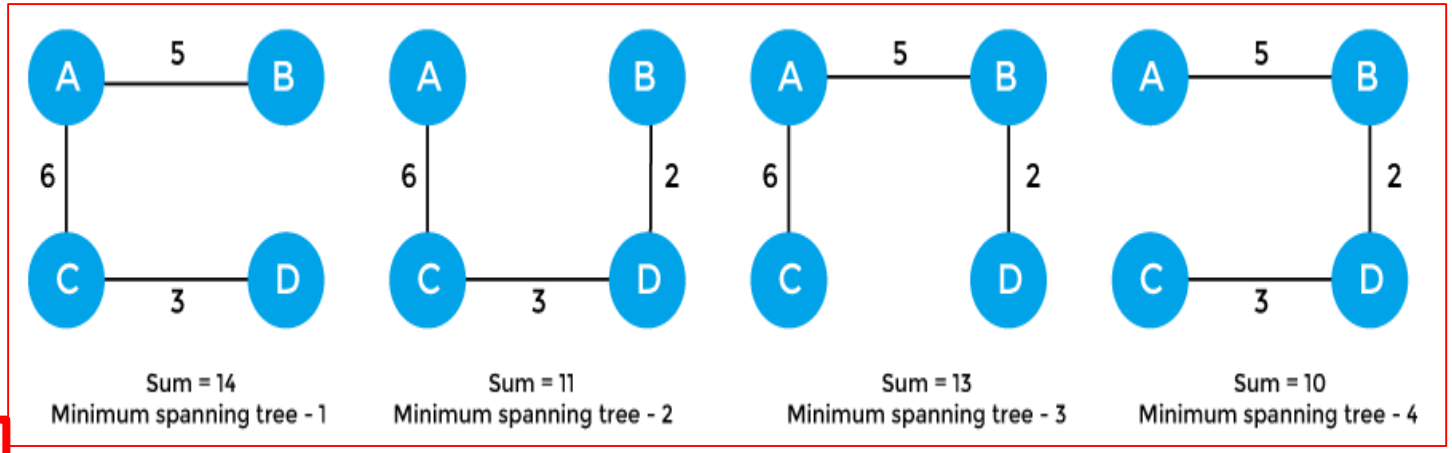


Fig: 1 Fig: 2 Fig: 3 Fig: 4

the minimum spanning tree that is selected from the above spanning trees for the given weighted graph is – fig 4.

Minimum Spanning tree: Kruskal's Algorithm

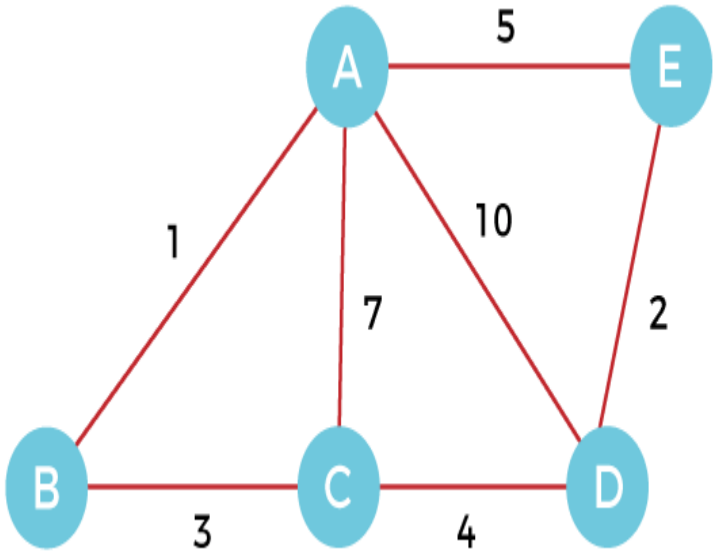
- **Kruskal's Algorithm** is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which we can traverse every vertex of the graph. It follows the greedy approach that finds an optimum solution at every stage instead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

Working procedure:

- 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.

Example

- Suppose a weighted graph is -



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

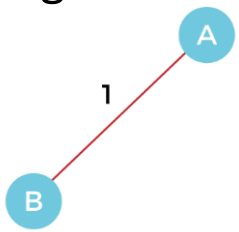
Edge	AB	AC	AD	AE	BC	CD	DE
Weight	1	7	10	5	3	4	2

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

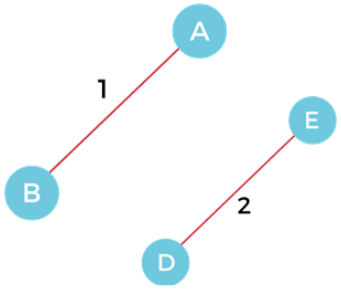
Edge	AB	DE	BC	CD	AE	AC	AD
Weight	1	2	3	4	5	7	10

Now, let's start constructing the minimum spanning tree(MST).

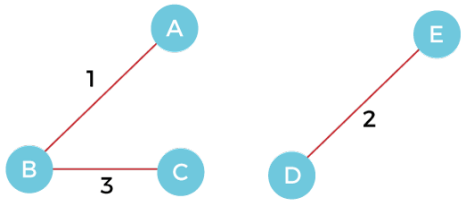
Step 1 - First, add the edge **AB** with weight **1** to the MST.



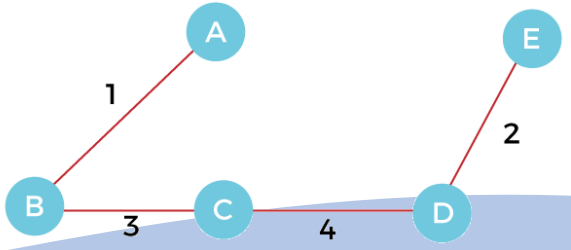
Step 2 - Add the edge **DE** with weight **2** to the MST as it is not creating the cycle.



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.

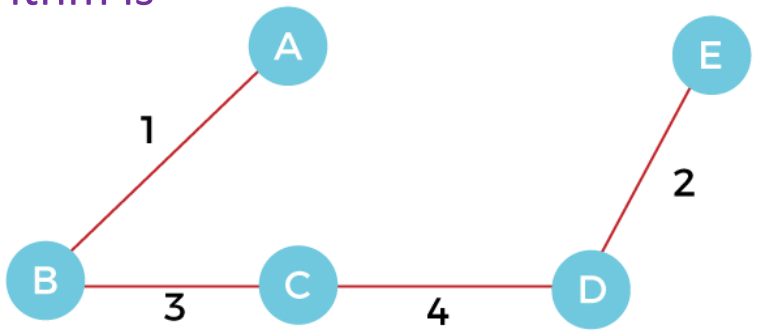


Step 5 - After that, pick the edge **AE** with weight **5.** Including this edge will create the cycle, so discard it.

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

Step 1: Create a forest F in such a way that every vertex of the graph is a separate tree

Step 2: Create a set E that contains all the edges of the graph.

Step 3: Repeat Steps 4 and 5 **while** E is NOT EMPTY and F is not spanning

Step 4: Remove an edge from E with minimum weight

Step 5: IF the edge obtained in Step 4 connects two different trees, then add it to the forest F

(**for** combining two trees into one tree).

ELSE

Discard the edge

Step 6: END

Greedy Algorithm

- A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result.
- The algorithm never reverses the earlier decision even if the choice is wrong. It works in a top-down approach.
- This algorithm may not produce the best result for all the problems. It's because it always goes for the local best choice to produce the global best result.
- However, we can determine if the algorithm can be used with any problem if the problem has the following properties:

i) Greedy Choice Property

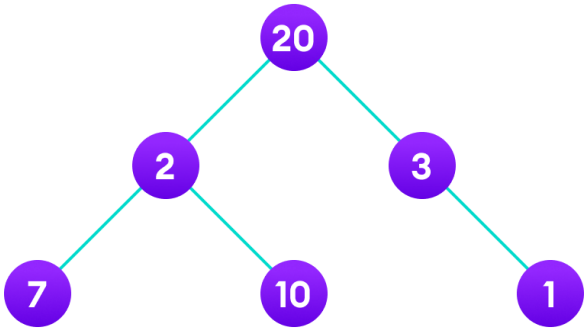
ii) Optimal Substructure

iii) Advantages of Greedy Approach

- The algorithm is **easier to describe**.
- This algorithm can **perform better** than other algorithms (but, not in all cases).

Example: Use of greedy algorithm

For example, suppose we want to find the longest path in the graph below from root to leaf.



Greedy Approach:

1. Let's start with the root node 20. The weight of the right child is 3 and the weight of the left child is 2.
2. Our problem is to find the largest path. And, the optimal solution at the moment is 3. So, the greedy algorithm will choose 3.
3. Finally the weight of an only child of 3 is 1. This gives us our final result 20 + 3 + 1 = 24. However, it is not the optimal solution. There is another path that carries more weight (20 + 2 + 10 = 32) as shown in fig: 2.

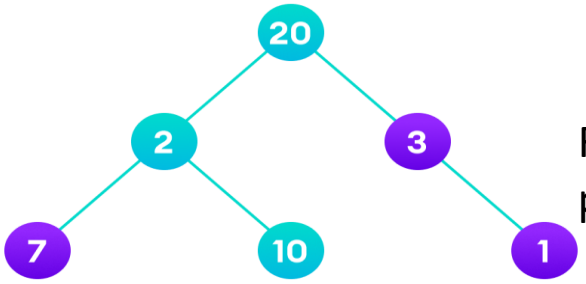


Fig: 2. Actual longest path

Shortest – Path Algorithm

- The shortest path problem is about finding a path between two vertices in a graph such that the total sum of the edges weights is minimum.
- This problem could be solved easily using **(BFS)** if all edge weights were (1), but here weights can take any value. Three different algorithms

a) Dijkstra's Algorithm

b) Warshall's Algorithm

c) Bellman Ford's Algorithm

Dijkstra's Algorithm

- The Dijkstra’s algorithm finds the shortest path from a particular node, called the source node to every other node in a connected graph. It produces a shortest path tree with the source node as the root. It is profoundly used in computer networks to generate optimal routes with the aim of minimizing routing costs.
- Input − A graph representing the network; and a source node, **S**
- Output − A shortest path tree, spt[], with **S** as the root node.

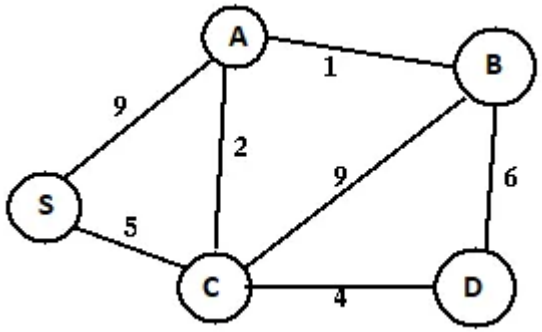
Algorithm

1. An array of distances **dist[]** of size **|V|** (number of nodes), where **dist[s]** = **0** and **dist[u]** = ∞ (infinite), where u represents a node in the graph except s.
2. An array, **Q**, containing all nodes in the graph. When the algorithm runs into completion, **Q** will become empty.
3. An empty set, **S**, to which the visited nodes will be added. When the algorithm runs into completion, **S** will contain all the nodes in the graph.
4. Repeat while **Q** is not empty −
   i. Remove from **Q**, the node, **u** having the smallest **dist[u]** and which is not in **S**. In the first run, **dist[s]** is removed.
   ii. Add **u** to **S**, marking **u** as visited.
   iii. For each node **v** which is adjacent to **u**, update **dist[v]** as −

   If (**dist[u] + weight of edge u-v**) **< dist[v],** Then

   Update **dist[v] = dist[u] + weight of edge u-v**
5. The array **dist[]** contains the shortest path from s to every other node.

- Consider the following graph:



Step 1: Initialize the distance array (dist) using the following steps.

1.1- Set dist[s]=0, S=ϕ // u is the source vertex and S is a 1-D array having all the visited vertices

1.2- For all nodes v except s, set dist[v]= ∞

Set of visited vertices (S)	S	A	B	C	D
	0	∞	∞	∞	∞

Step 2: Choose the source vertex s as dist[s] is minimum and s is not in S.

2.1: find q not in S such that dist[q] is minimum // vertex should not be visited

Visit s by adding it to S

2.2- add q to S // add vertex q to S since it has now been visited

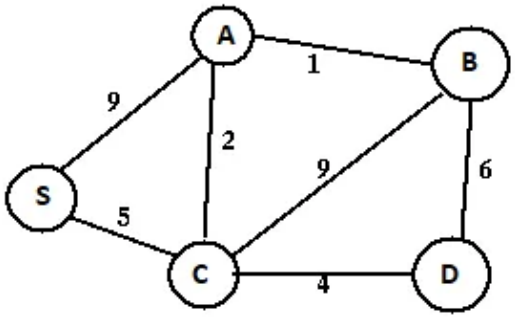
Step 3: For all adjacent vertices of s which have not been visited yet (are not in S) i.e A and C, update the distance array using the following steps of algorithm –

3.1: update dist[r] for all r adjacent to q such that r is not in S

dist[r]=min(dist[r], dist[q]+cost[q][r])

Set of visited vertices (S)	S	A	B	C	D
[s]	0	9	∞	5	∞

- Given graph:



Set of visited vertices (S)	S	A	B	C	D
[s]	0	9	∞	5	∞
[s,C]	0	7	14	5	9

Step 4: Repeat Step 2 by:

- Choosing and visiting vertex C since it has not been visited (not in S) and dist[C] is minimum
- Updating the distance array for adjacent vertices of C i.e. A, B and D

dist[A]=min(dist[A], dist[C]+cost(C,A)) = min(9, 5+2)= 7

dist[B]= min(dist[B], dist[C]+cost(C,B)) = min(∞, 5+9)= 14

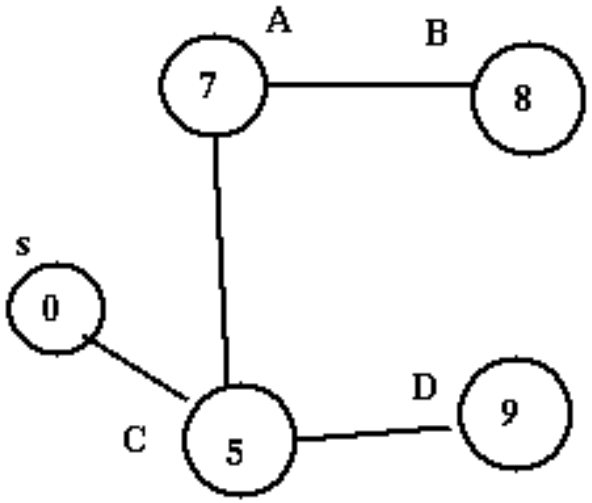
dist[D]= min(dist[D], dist[C]+cost(C,D))= min((∞,5+4)=9

Continuing on similar lines, Step 2 gets repeated till all the vertices are visited (added to S). dist[] also gets updated in every iteration, resulting in the following –

Set of visited vertices (S)	S	A	B	C	D
[s]	0	9	∞	5	∞
[s,C]	0	7	14	5	9
[s, C, A]	0	7	8	5	9
[s, C, A, B]	0	7	8	5	9
[s, C, A, B, D]	0	7	8	5	9

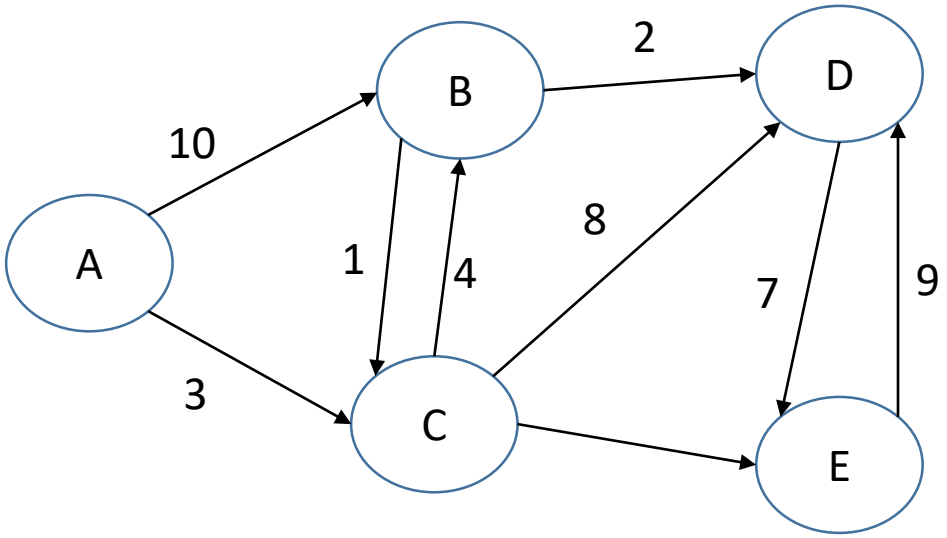
- The last updation of dist[] gives the shortest path values from s to all other vertices

The resultant shortest path spanning tree for the given graph is as follows-

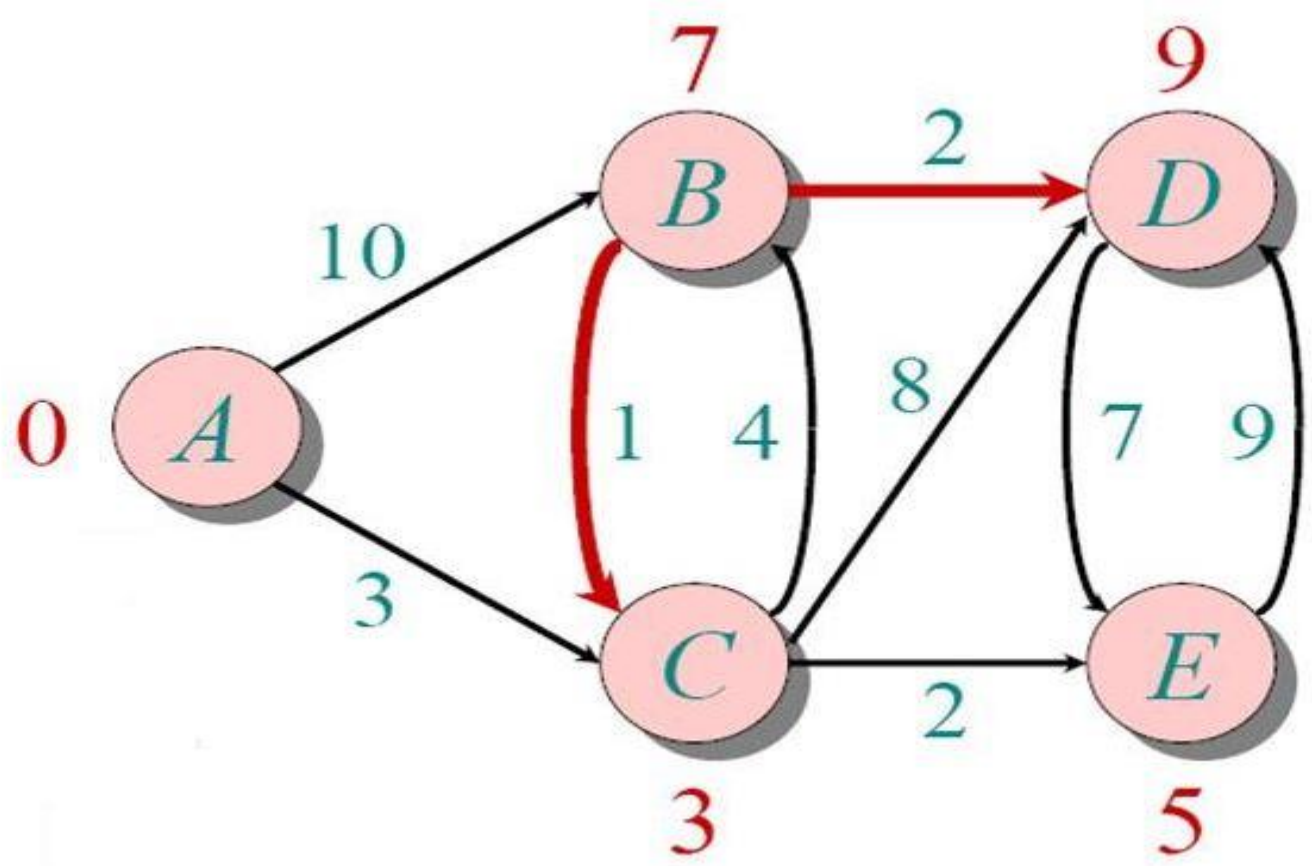


- ***Note-****There can be multiple shortest path spanning trees for the same graph depending on the source vertex*

Find the shortest path using Dijkstra's Algorithm



Solution:



Any Query?