



RGPVNOTES.IN

Program : **B.Tech**

Subject Name: **Data Structure**

Subject Code: **CS-303**

Semester: **3rd**



LIKE & FOLLOW US ON FACEBOOK

facebook.com/rgpvnotes.in

Unit 4

Syllabus:

Graphs: Introduction, Classification of graph: Directed and Undirected graphs, etc, Representation, Graph Traversal: Depth First Search (DFS), Breadth First Search (BFS), Graph algorithm: Minimum Spanning Tree (MST)- Kruskal, Prim's algorithms. Dijkstra's shortest path algorithm; Comparison between different graph algorithms, Application of graphs.

Graphs:

A graph is a mathematical abstraction used to represent "connectivity information". A graph consists of vertices and edges that connect them, e.g.

A graph $G = (V, E)$ is:

a set of vertices V and a set of edges $E = \{ (u, v) : u \text{ and } v \text{ are vertices} \}$.

Two types of graphs:

- Undirected graphs: the edges have no direction.
- Directed graphs: the edges have direction.

Classification of graph:

Undirected graphs:

Graph is a data structure that consists of following two components:

1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form (u, v) called as edge.

The pair is ordered because (u, v) is not same as (v, u) in case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v . The edges may contain weight/value/cost. Graphs are used to represent many real-life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex (or node). Each node is a structure and contains information like person id, name, gender and locale.

Following is an example of an undirected graph with 5 vertices.

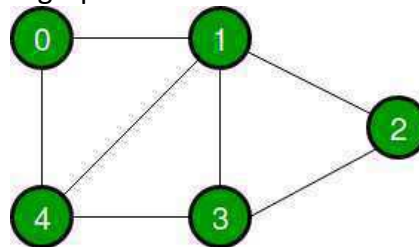


Figure 4.1 Undirected Graph

Directed graphs:

A **directed graph** is a Data Structure containing a vertex set V and an arc set A , where each arc (or edge, or link) is an ordered pair of vertices (or nodes, or sommets). The arcs may be thought of as arrows, each one starting at one vertex and pointing at precisely one other.

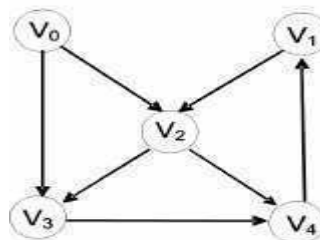


Figure 4.2 Directed Graph

Graph Traversal:
Depth First Search (DFS):

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

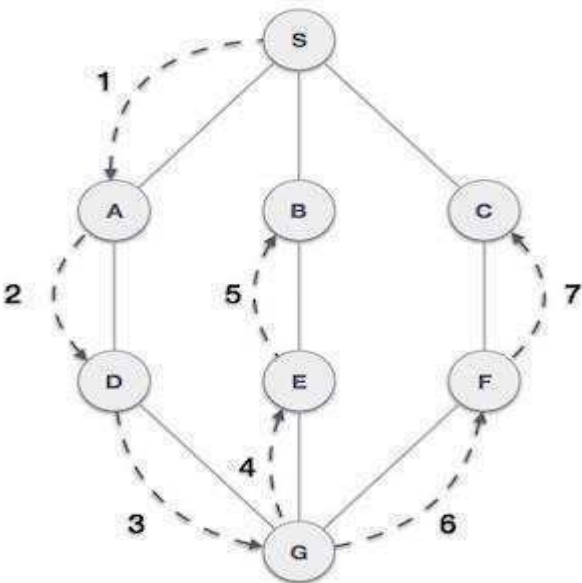
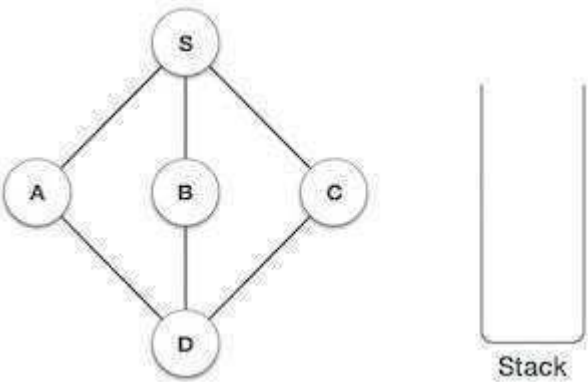
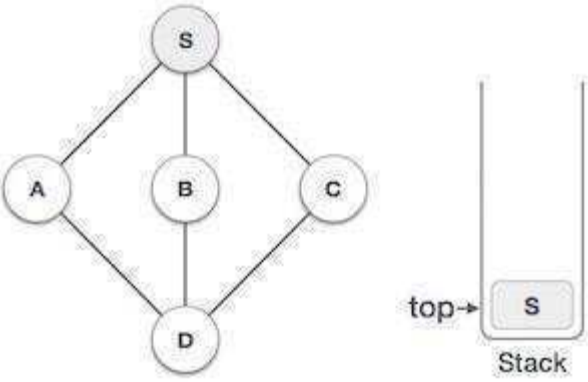
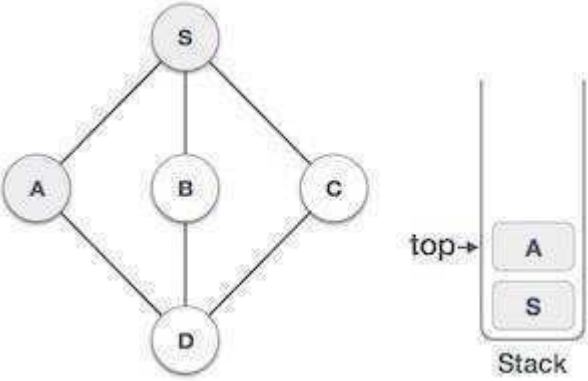
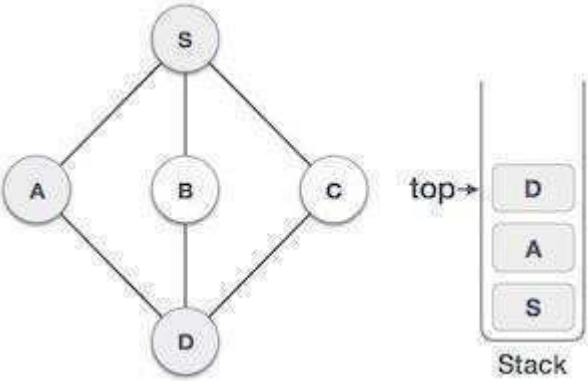


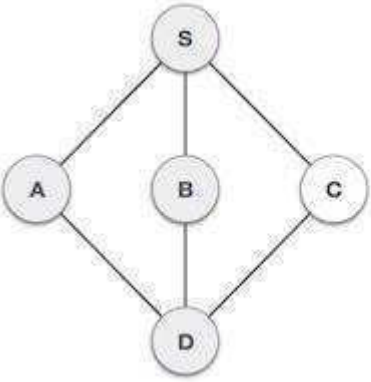
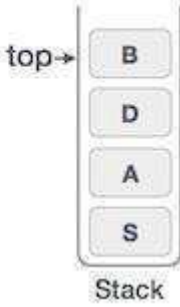
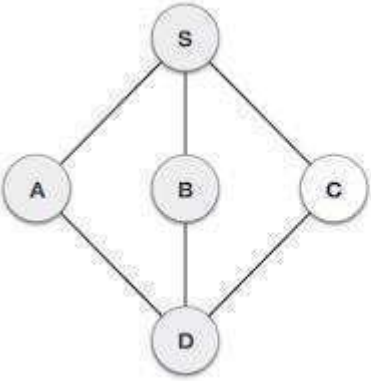
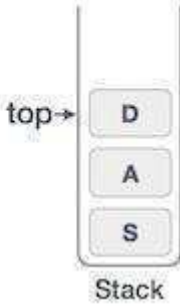
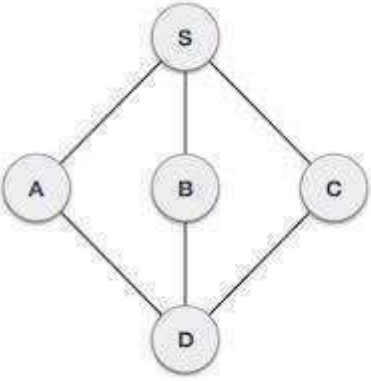
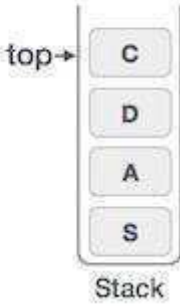
Figure 4.3: DFS Example

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

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

Step	Traversal	Description
1		Initialize the stack.

2	 <p>Graph structure: S is connected to A, B, and C. A, B, and C are connected to D. A stack is shown with S at the top.</p>	Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S . We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3	 <p>Graph structure: S is connected to A, B, and C. A, B, and C are connected to D. A stack is shown with A and S, with A at the top.</p>	Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A . Both S and D are adjacent to A but we are concerned for unvisited nodes only.
4	 <p>Graph structure: S is connected to A, B, and C. A, B, and C are connected to D. A stack is shown with D, A, and S, with D at the top.</p>	Visit D and mark it as visited and put onto the stack. Here, we have B and C nodes, which are adjacent to D and both are unvisited. However, we shall again choose in an alphabetical order.

5	 	<p>We choose B, mark it as visited and put onto the stack.</p> <p>Here B does not have any unvisited adjacent node. So, we pop B from the stack.</p>
6	 	<p>We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find D to be on the top of the stack.</p>
7	 	<p>Only unvisited adjacent node is from D is C now. So we visit C, mark it as visited and put it onto the stack.</p>

Breadth First Search (BFS):

Breadth First Search (BFS) algorithm traverses a graph in a breadth ward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

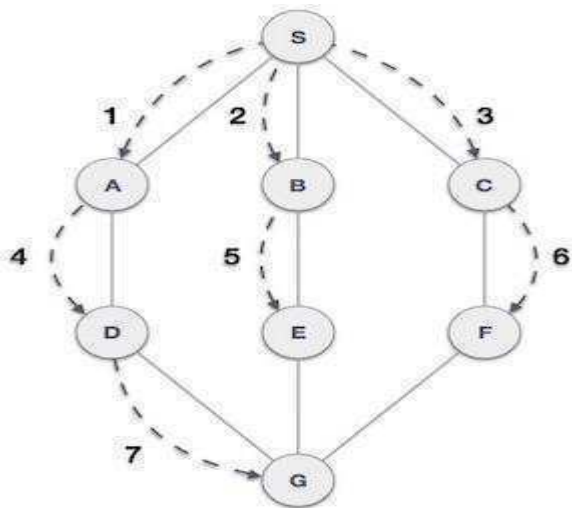
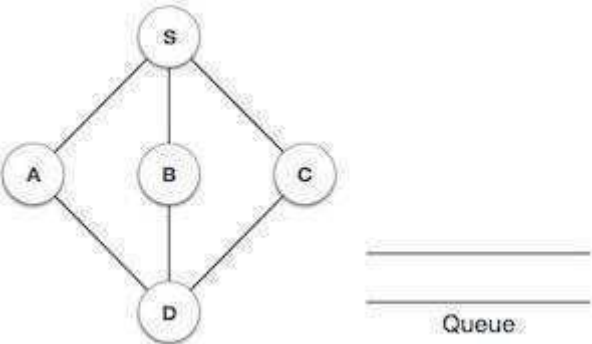
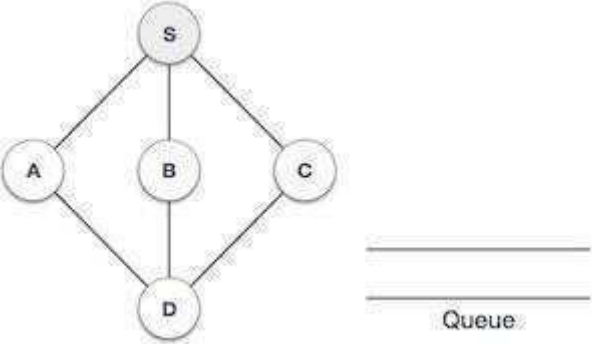
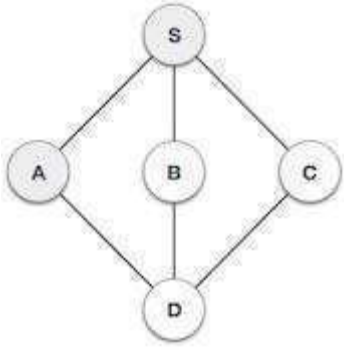
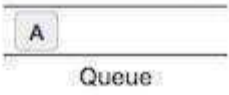
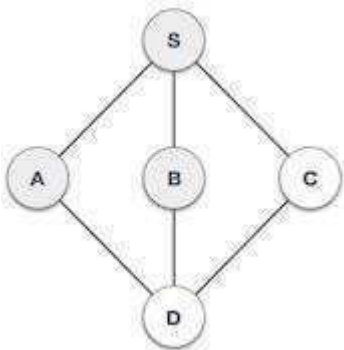
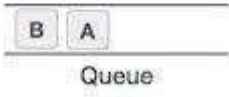
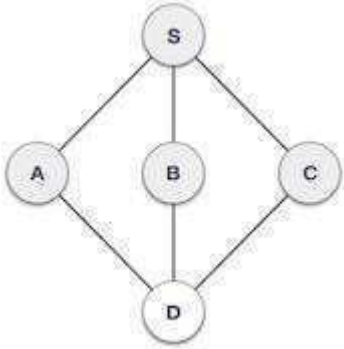
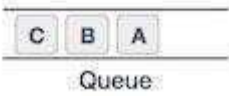
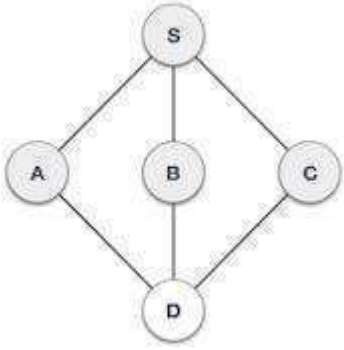



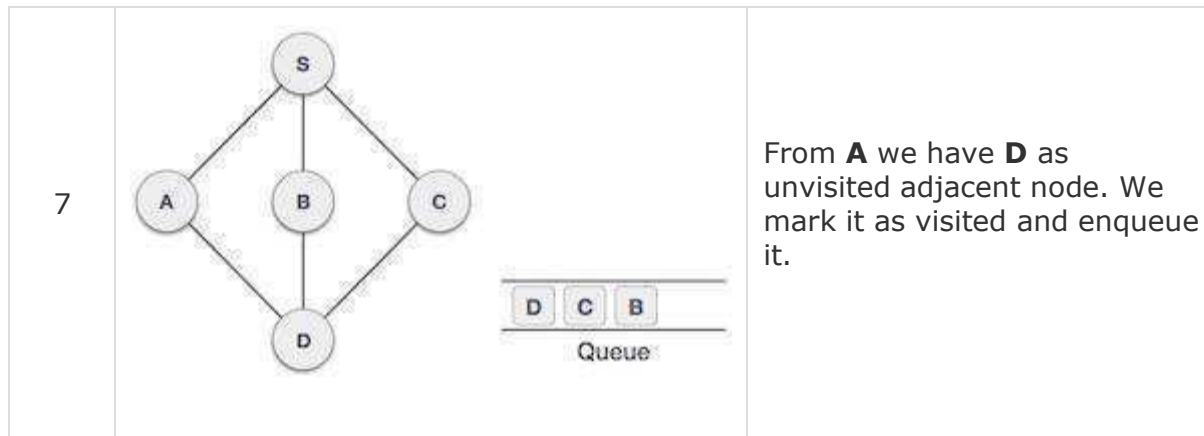
Figure 4.4: BFS Example

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

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

Step	Traversal	Description
1		Initialize the queue.
2		We start from visiting S (starting node), and mark it as visited.

3	 	<p>We then see an unvisited adjacent node from S. In this example, we have three nodes but alphabetically we choose A, mark it as visited and enqueue it.</p>
4	 	<p>Next, the unvisited adjacent node from S is B. We mark it as visited and enqueue it.</p>
5	 	<p>Next, the unvisited adjacent node from S is C. We mark it as visited and enqueue it.</p>
6	 	<p>Now, S is left with no unvisited adjacent nodes. So, we dequeue and find A.</p>



Graph algorithm:

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

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.

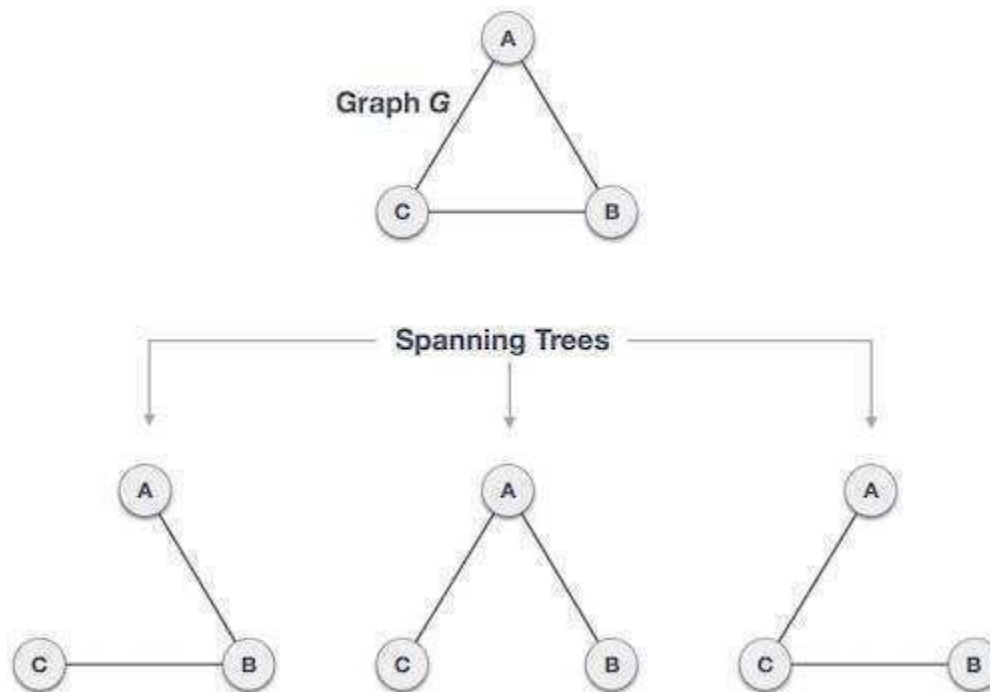


Figure 4.5: Graph and Spanning Trees

We found three spanning trees off one complete graph. A complete undirected graph can have maximum n^{n-2} number of spanning trees, where n is the number of nodes. In the above addressed example, n is 3, hence $3^{3-2} = 3$ spanning trees are possible.

Minimum Spanning Tree (MST)

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

Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

To understand Kruskal's algorithm let us consider the following example –

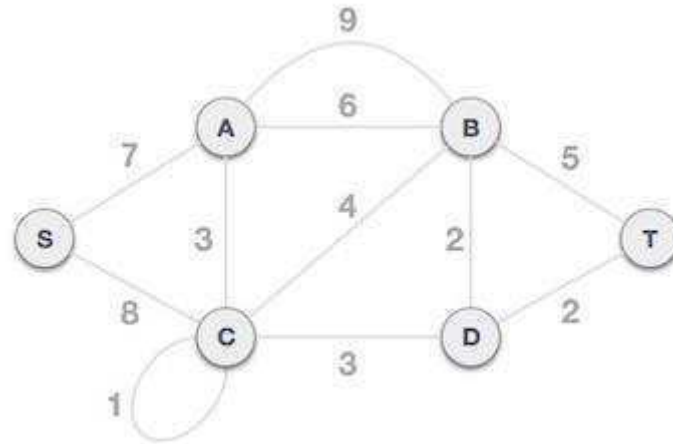
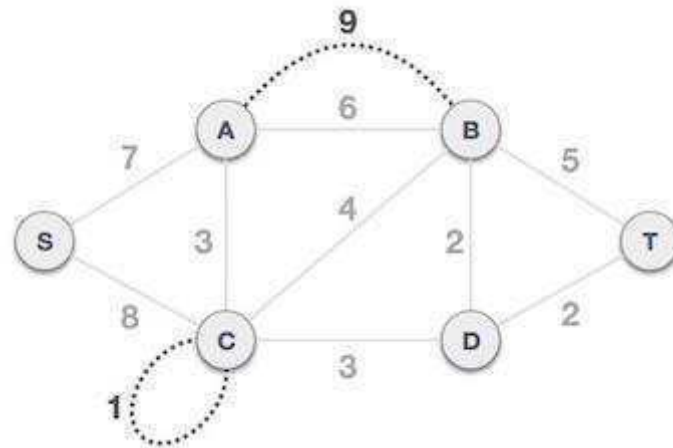


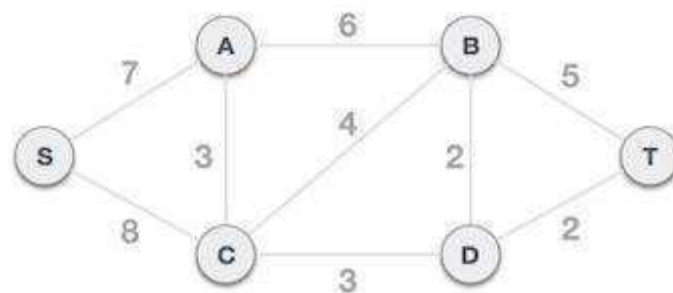
Figure 4.6: Kruskal's Algorithm Example

Step 1 - Remove all loops and Parallel Edges

Remove all loops and parallel edges from the given graph.



In case of parallel edges, keep the one which has the least cost associated and remove all others.



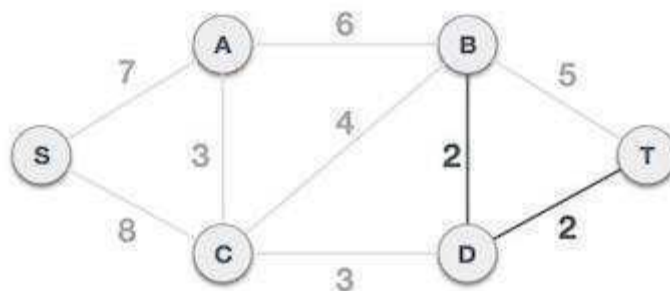
Step 2 - Arrange all edges in their increasing order of weight

The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

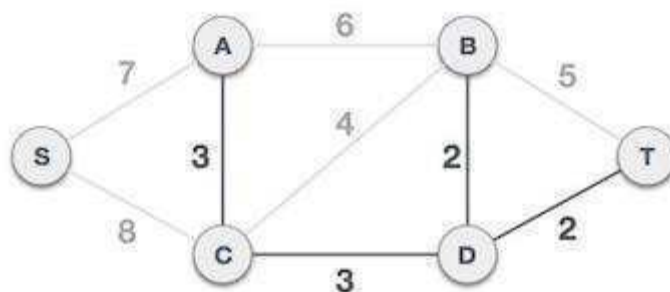
Step 3 - Add the edge which has the least weightage

Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

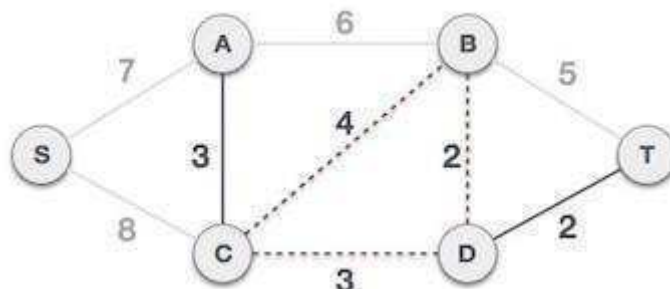


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

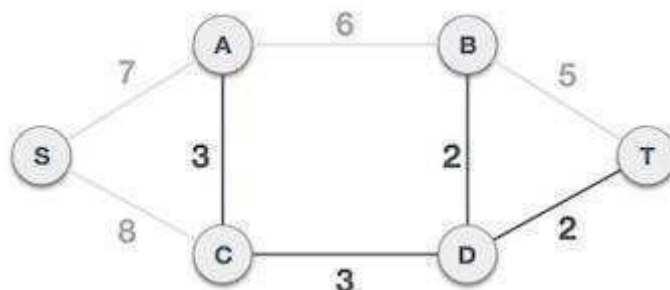
Next cost is 3, and associated edges are A,C and C,D. We add them again –



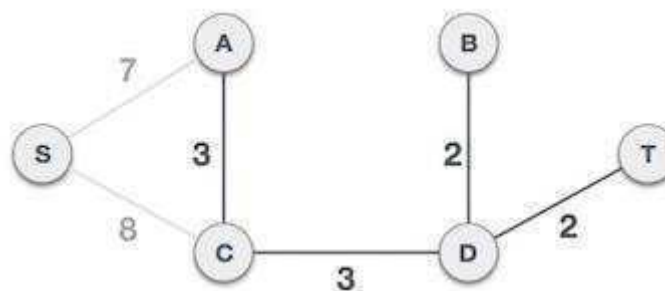
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. –



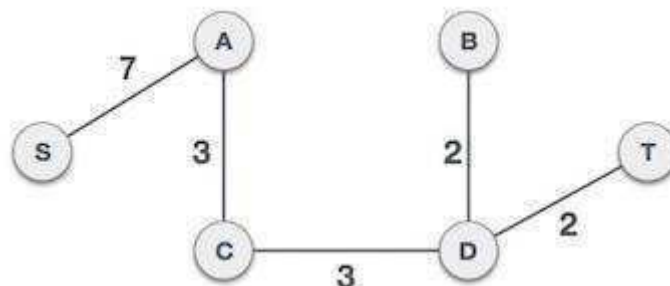
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



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



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



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

Prim's Algorithm:

Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.

Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example –

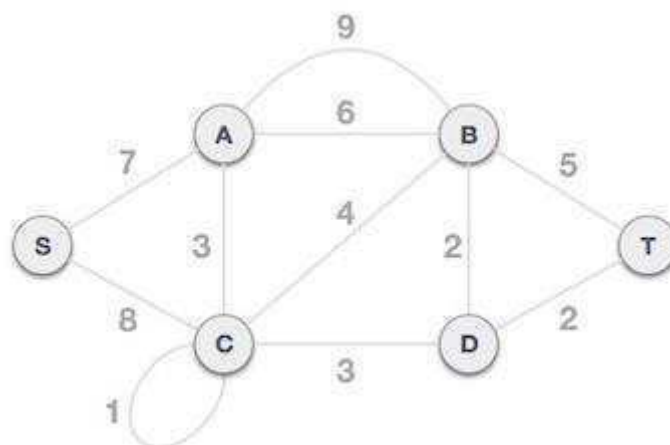
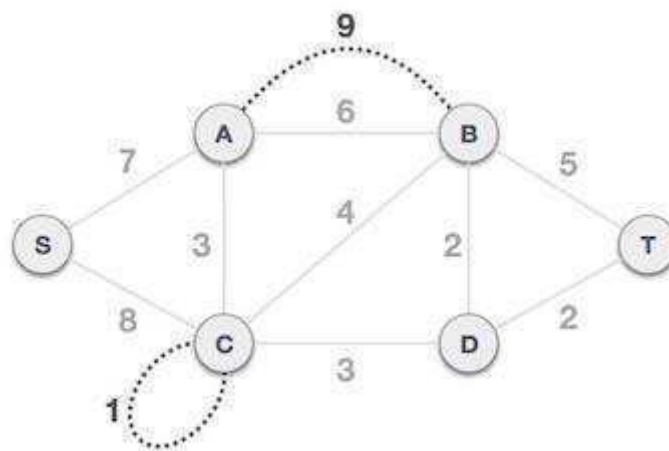
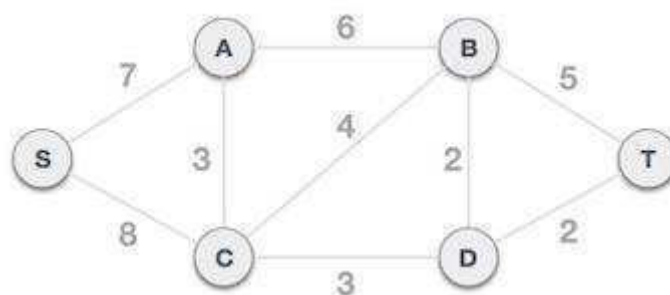


Figure 4.6: Prim's Algorithm Example

Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

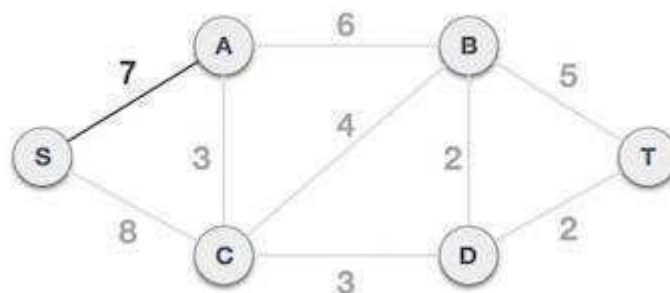


Step 2 - Choose any arbitrary node as root node

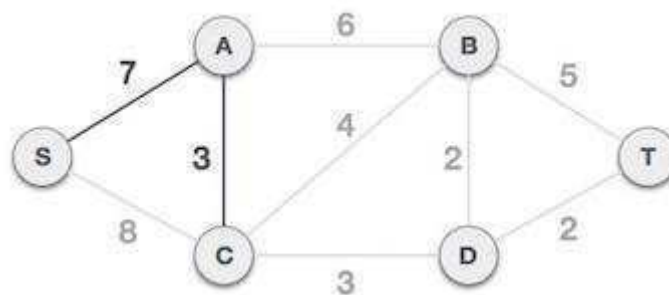
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any node can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3 - Check outgoing edges and select the one with less cost

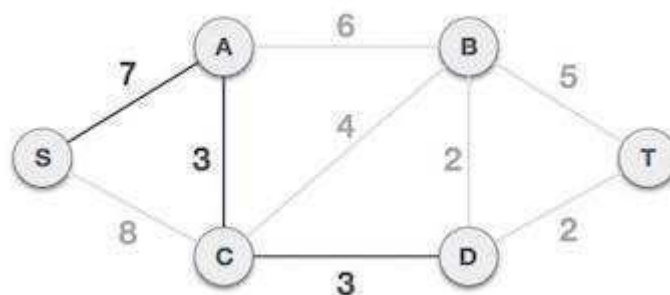
After choosing the root node **S**, we see that S, A and S, C are two edges with weight 7 and 8, respectively. We choose the edge S, A as it is lesser than the other.



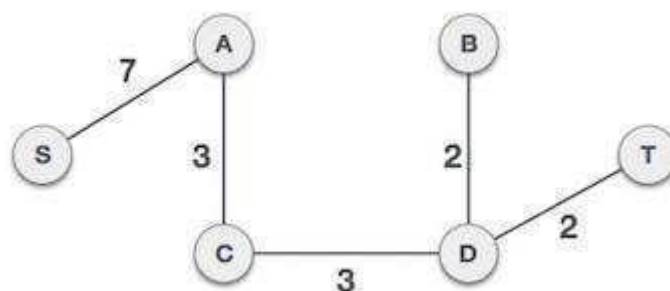
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



We may find that the output spanning tree of the same graph using two different algorithms is same.

Dijkstra's Shortest Path Algorithm:

Dijkstra's algorithm can be used to determine the shortest path from one node in a graph to every other node within the same graph data structure, provided that the nodes are reachable from the starting node. This algorithm will continue to run until all of the reachable vertices in a graph have been visited, which means that we could run Dijkstra's algorithm, find the shortest path between any two reachable nodes, and then save the results somewhere. Once we run Dijkstra's algorithm just *once*, we can look up our results from our algorithm again and again — without having to actually run the algorithm itself.

The abstracted rules to solve the algorithm are as follows:

1. Every time that we set out to visit a new node, we will choose the node with the smallest known distance/cost to visit first.
2. Once we've moved to the node we're going to visit, we will check each of its neighboring nodes.
3. For each neighboring node, we'll calculate the distance/cost for the neighboring nodes by summing the cost of the edges that lead to the node we're checking from the starting vertex.

4. Finally, if the distance/cost to a node is *less than* a known distance, we'll update the shortest distance that we have on file for that vertex.

Example: Find the shortest path in the following multistage graph-

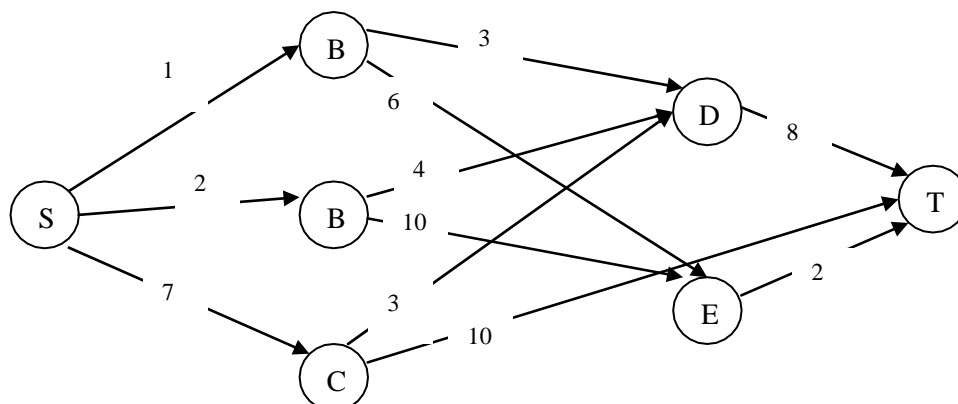


Figure 4.7: Prim's Algorithm Example

There is a single vertex in stage 1, then 3 vertices in stage 2, then 2 vertices in stage 3 and only one vertex in stage 4 (this is a target stage).

Backward approach

$$d(S, T) = \min \{1+d(A, T), 2+d(B, T), 7+d(C, T)\} \dots (1)$$

We will compute $d(A, T)$, $d(B, T)$ and $d(C, T)$.

$$d(A, T) = \min \{3+d(D, T), 6+d(E, T)\} \dots (2)$$

$$d(B, T) = \min \{4+d(D, T), 10+d(E, T)\} \dots (3)$$

$$d(C, T) = \min \{3+d(E, T), d(C, T)\} \dots (4)$$

Now let us compute $d(D, T)$ and $d(E, T)$.

$$d(D, T) = 8$$

$$d(E, T) = 2 \text{ backward vertex} = E$$

Let us put these values in equations (2), (3) and (4)

$$d(A, T) = \min \{3+8, 6+2\}$$

$$d(A, T) = 8 \text{ A-E-T}$$

$$d(B, T) = \min \{4+8, 10+2\}$$

$$d(B, T) = 12 \text{ A-D-T}$$

$$d(C, T) = \min \{3+2, 10\}$$

$$d(C, T) = 5 \text{ C-E-T}$$

$$d(S, T) = \min \{1+d(A, T), 2+d(B, T), 7+d(C, T)\}$$

$$= \min \{1+8, 2+12, 7+5\}$$

$$= \min \{9, 14, 12\}$$

$$d(S, T) = 9 \text{ S-A-E-T}$$

The path with minimum cost is S-A-E-T with the cost 9.

Forward approach

$$d(S, A) = 1$$

$$d(S, B) = 2$$

$$d(S, C) = 7$$



$$d(S,D)=\min\{1+d(A,D),2+d(B,D)\}$$

$$=\min\{1+3,2+4\}$$

$$d(S,D)=4$$

$$d(S,E)=\min\{1+d(A,E),2+d(B,E),7+d(C,E)\}$$

$$=\min\{1+6,2+10,7+3\}$$

$$=\min\{7,12,10\}$$

$$d(S,E)=7 \text{ i.e. Path S-A-E is chosen.}$$

$$d(S,T)=\min\{d(S,D)+d(D,T),d(S,E),d(E,T),d(S,C)+d(C,T)\}$$

$$=\min\{4+8,7+2,7+10\}$$

$$d(S,T)=9 \text{ i.e. Path S-E, E-T is chosen.}$$

The minimum cost=9 with the path S-A-E-T.

Using dynamic approach programming strategy, the multistage graph problem is solved. This is because in multistage graph problem we obtain the minimum path at each current stage by considering the path length of each vertex obtained in earlier stage.

Comparison between different graph algorithms:

The first distinction is that Dijkstra's algorithm solves a different problem than Kruskal and Prim. Dijkstra solves the shortest path problem (from a specified node), while Kruskal and Prim finds a minimum-cost spanning tree.

For any graph, a spanning tree is a collection of edges sufficient to provide exactly one path between every pair of vertices. This restriction means that there can be no circuits formed by the chosen edges.

A minimum-cost spanning tree is one which has the smallest possible total weight (where weight represents cost or distance). There might be more than one such tree, but Prim and Kruskal are both guaranteed to find one of them.

For a specified vertex (say X), a shortest path tree is a spanning tree such that the path from X to any other vertex is as short as possible (i.e., has the minimum possible weight).

Prim and Dijkstra "grow" the tree out from a starting vertex. In other words, they have a "local" focus; at each step, we only consider those edges adjacent to previously chosen vertices, choosing the cheapest option which satisfies our needs. Meanwhile, Kruskal is a "global" algorithm, meaning that each edge is (greedily) chosen from the entire graph.

To find a minimum-cost spanning tree:

- **Kruskal (global approach):** At each step, choose the cheapest available edge *anywhere* which does not violate the goal of creating a spanning tree.
- **Prim (local approach):** Choose a starting vertex. At each successive step, choose the cheapest available edge attached to any previously chosen vertex which does not violate the goal of creating a spanning tree.

To find a shortest-path spanning tree:

- **Dijkstra:** At each step, choose the edge attached to any previously chosen vertex (the local aspect) which makes the total distance from the starting vertex (the global aspect) as small as possible, and

does not violate the goal of creating a spanning tree

Minimum-cost trees and shortest-path trees are easily confused, as are the Prim and Dijkstra algorithms that solve them. Both algorithms "grow out" from the starting vertex, at each step choosing an edge which connects a vertex Y which is in the tree to a vertex Z which is not. However, while Prim chooses the cheapest such edge, Dijkstra chooses the edge which results in the shortest path from X to Z.

Application of graphs:

Since they are powerful abstractions, graphs can be very important in modeling data. In fact, many problems can be reduced to known graph problems. Here we outline just some of the many applications of graphs.

1. Social network graphs: To tweet or not to tweet. Graphs that represent who knows whom, who communicates with whom, who influences whom or other relationships in social structures. An example is the twitter graph of who follows whom. These can be used to determine how information flows, how topics become hot, how communities develop, or even who might be a good match for who, or is that whom.

2. Transportation networks: In road networks vertices are intersections and edges are the road segments between them, and for public transportation networks vertices are stops and edges are the links between them. Such networks are used by many map programs such as Google maps, Bing maps and now Apple IOS 6 maps (well perhaps without the public transport) to find the best routes between locations. They are also used for studying traffic patterns, traffic light timings, and many aspects of transportation.

3. Utility graphs: The power grid, the Internet, and the water network are all examples of graphs where vertices represent connection points, and edges the wires or pipes between them. Analyzing properties of these graphs is very important in understanding the reliability of such utilities under failure or attack, or in minimizing the costs to build infrastructure that matches required demands.

4. Document link graphs: The best known example is the link graph of the web, where each web page is a vertex, and each hyperlink a directed edge. Link graphs are used, for example, to analyze relevance of web pages, the best sources of information, and good link sites.

5. Protein-protein interactions graphs: Vertices represent proteins and edges represent interactions between them that carry out some biological function in the cell. These graphs can be used, for example, to study molecular pathways—chains of molecular interactions in a cellular process. Humans have over 120K proteins with millions of interactions among them.

6. Network packet traffic graphs: Vertices are IP (Internet protocol) addresses and edges are the packets that flow between them. Such graphs are used for analyzing network security, studying the spread of worms, and tracking criminal or non-criminal activity.

7. Scene graphs: In graphics and computer games scene graphs represent the logical or spacial relationships between objects in a scene. Such graphs are very important in the computer games industry.

8. Finite element meshes: In engineering many simulations of physical systems, such as the flow of air over a car or airplane wing, the spread of earthquakes through the ground, or the structural vibrations of a

building, involve partitioning space into discrete elements. The elements along with the connections between adjacent elements form a graph that is called a finite element mesh.

9. Robot planning: Vertices represent states the robot can be in and the edges the possible transitions between the states. This requires approximating continuous motion as a sequence of discrete steps. Such graph plans are used, for example, in planning paths for autonomous vehicles.

10. Neural networks: Vertices represent neurons and edges the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 1011 neurons and close to 1015 synapses.

11. Graphs in quantum field theory: Vertices represent states of a quantum system and the edges the transitions between them. The graphs can be used to analyze path integrals and summing these up generates a quantum amplitude (yes, I have no idea what that means).

12. Semantic networks: Vertices represent words or concepts and edges represent the relationships among the words or concepts. These have been used in various models of how humans organize their knowledge, and how machines might simulate such an organization.

13. Graphs in epidemiology: Vertices represent individuals and directed edges the transfer of an infectious disease from one individual to another. Analyzing such graphs has become an important component in understanding and controlling the spread of diseases.

14. Graphs in compilers: Graphs are used extensively in compilers. They can be used for type inference, for so called data flow analysis, register allocation and many other purposes. They are also used in specialized compilers, such as query optimization in database languages.

15. Constraint graphs: Graphs are often used to represent constraints among items. For example the GSM network for cell phones consists of a collection of overlapping cells. Any pair of cells that overlap must operate at different frequencies. These constraints can be modeled as a graph where the cells are vertices and edges are placed between cells that overlap.

16. Dependence graphs: Graphs can be used to represent dependences or precedence among items. Such graphs are often used in large projects in laying out what components rely on other components and used to minimize the total time or cost to completion while abiding by the dependences.



RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at
<https://qp.rgpvnotes.in> .

If you have any queries or you want to submit your
study notes please write us at
rgpvnotes.in@gmail.com



LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in