Unit-V Graphs And Their Applications

5.1 Graphs It’s Representation (Matrix/Adjacency)

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

* A finite set of vertices also called nodes.
* A finite set of ordered pair of the form (u, v) called edge. The pair is ordered because (u, v) is not the same as (v, u) in the 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.

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

Example of undirected graph with 5 vertices

*Example of undirected graph with 5 vertices*

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. See [this](http://en.wikipedia.org/wiki/Graph_theory#Applications)for more applications of graph. In computer science, a graph is a data structure that is used to represent connections or relationships between objects. A graph consists of a set of vertices (also known as nodes) and a set of edges (also known as arcs) that connect the vertices. The vertices can represent anything from cities in a map to web pages in a network, and the edges can represent the relationships between them, such as roads or links.

A graph can be visualized as a collection of points (vertices) connected by lines (edges), where each vertex represents a point of interest and each edge represents a connection between two points. The edges can be directed or undirected, meaning they can either have a specific direction or be bidirectional. For example, a map of a city may have directed edges that represent the direction of one-way streets, while a social network may have undirected edges that represent friendships between individuals.

**5.1 Representations of Graphs:-**

The following two are the most commonly used representations of a graph.

1. **Adjacency List**
2. **Adjacency Matrix**

There are other representations also like, Incidence Matrix and Incidence List. The choice of graph representation is situation-specific. It totally depends on the type of operations to be performed and the ease of use

1.Adjacency List-

*An array of linked lists is used. The size of the array is equal to the number of vertices. Let the array be an****array[]****. An entry****array[i]****represents the linked list of vertices adjacent to the****ith****vertex.*

*This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs.*

Consider the following graph:

Example of undirected graph with 5 vertices

*Example of undirected graph with 5 vertices*

Following is the adjacency list representation of the above graph.

Adjacency List representation of the above graph

*Adjacency List representation of the above graph*

* **Advantages of Adjacency List:**

1. Saves space. Space taken is O(|V|+|E|). In the worst case, there can be C(V, 2) number of edges in a graph thus consuming O(V2) space.
2. Adding a vertex is easier.
3. Computing all neighbors of a vertex takes optimal time.

* **Disadvantages of Adjacency List:**

1. Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done O(V).

2.Adjacency Matrix-

Adjacency Matrix is a 2D array of size **V x V** where **V** is the number of vertices in a graph. Let the 2D array be **adj[][]**, a slot **adj[i][j] = 1** indicates that there is an edge from vertex **i** to vertex **j**. The adjacency matrix for an undirected graph is always symmetric.

Adjacency Matrix is also used to represent weighted graphs. If **adj[i][j] = w**, then there is an edge from vertex **i** to vertex **j** with weight **w**.

We follow the below pattern to use the adjacency matrix in code:

* In the case of an undirected graph, we need to show that there is an edge from vertex **i** to vertex **j** and vice versa. In code, we assign adj[i][j] = 1  and adj[j][i] = 1.
* In the case of a directed graph, if there is an edge from vertex **i** to vertex **j** then we just assign **adj[i][j]=1**.\

See the undirected graph shown below:

Example of undirected graph with 5 vertices

*Example of undirected graph with 5 vertices*

The adjacency matrix for the above example graph is:

Adjacency matrix representation

*Adjacency matrix representation*

* **Advantages of Adjacency Matrix:**

1. Representation is easier to implement and follow.
2. Removing an edge takes O(1) time.
3. Queries like whether there is an edge from vertex ‘u’ to vertex ‘v’ are efficient and can be done O(1).

* **Disadvantages of Adjacency Matrix:**

1. Consumes more space O(V2). Even if the graph is sparse(contains less number of edges), it consumes the same space.
2. Adding a vertex takes O(V2) time. Computing all neighbors of a vertex takes O(V) time (Not efficient).

**5.2 Applications of Graphs:-**

In **computer science** graph theory is used for the **study of algorithms** like:

1. Dijkstra's Algorithm
2. Prims's Algorithm
3. Kruskal's Algorithm
4. Graphs are used to define the **flow of computation**.
5. Graphs are used to represent **networks of communication**.
6. Graphs are used to represent **data organization**.
7. Graph transformation systems work on rule-based in-memory manipulation of graphs. Graph databases ensure **transaction-safe, persistent storing and querying of graph structured data**.
8. Graph theory is used to find **shortest path in road** or a network.
9. In **Google Maps**, various locations are represented as vertices or nodes and the roads are represented as edges and graph theory is used to find the shortest path between two nodes.

**5.2 Traversal of Graphs (Depth First Search/ Breadth First Search):-**

Graph traversal is a technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.

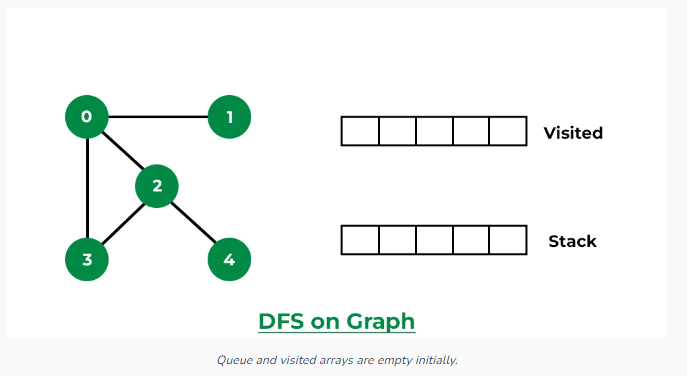
There are two graph traversal techniques and they are as follows...

1. **DFS (Depth First Search)**
2. **BFS (Breadth First Search)**
3. DFS (Depth First Search)

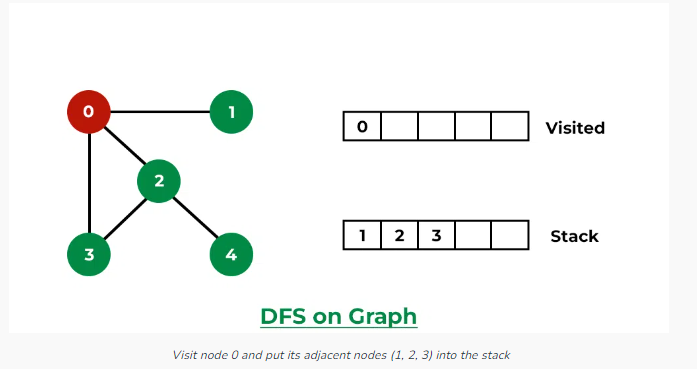
DFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal.  
  
Follow the below method to implement DFS traversal.

* **Step 1:** Create a set or array to keep track of visited nodes.
* **Step 2:** Choose a starting node.
* **Step 3:**Create an empty stack and push the starting node onto the stack.
* **Step 4:**Mark the starting node as visited.
* **Step 5:**While the stack is not empty, do the following:
  + Pop a node from the stack.
  + Process or perform any necessary operations on the popped node.
  + Get all the adjacent neighbors of the popped node.
  + For each adjacent neighbor, if it has not been visited, do the following:
    - Mark the neighbor as visited.
    - Push the neighbor onto the stack.
* **Step 6:** Repeat step 5 until the stack is empty.

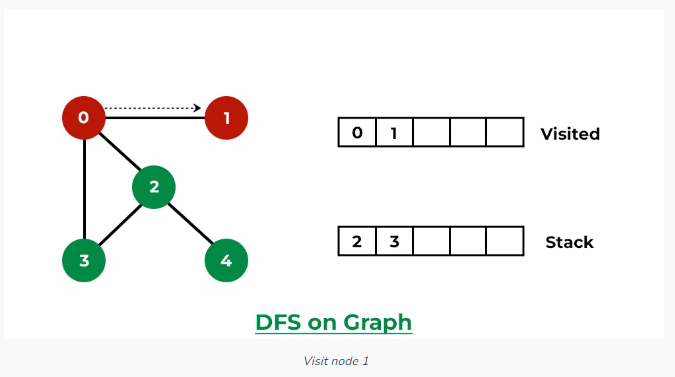
***Step1:****Initially queue and visited arrays are empty.*

`

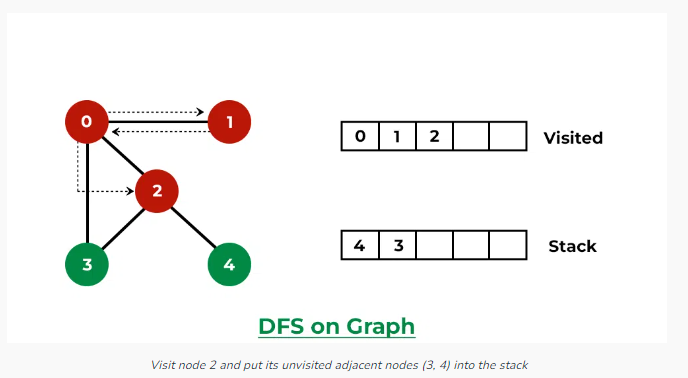
***Step 2:****Visit 0 and put its adjacent nodes which are not visited yet into the stack.*

**

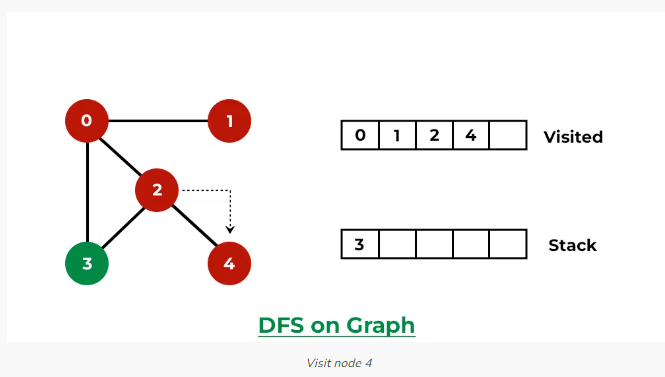
***Step 3:****Now, Node 1 at the top of the stack, so visit node 1 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.*

**

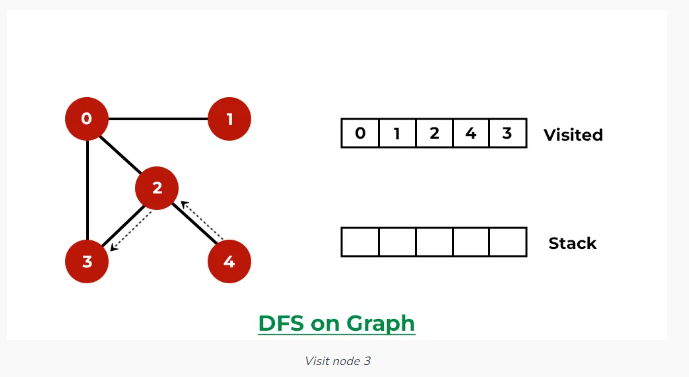
***Step 4:****Now,**Node 2 at the top of the stack, so visit node 2 and pop it from the stack and put all of its adjacent nodes which are not visited (i.e, 3, 4) in the stack.*

**

***Step 5:****Now, Node 4 at the top of the stack, so visit node 4 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.*



***Step 6:****Now, Node 3 at the top of the stack, so visit node 3 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.*



*Now, Stack becomes empty, which means we have visited all the nodes and our DFS traversal ends.*

1. BFS (Breadth First Search)

*The breadth-first search (BFS) algorithm is used to search a tree or graph data structure for a node that meets a set of criteria. It starts at the tree’s root or graph and searches/visits all nodes at the current depth level before moving on to the nodes at the next depth level. Breadth-first search can be used to solve many problems in graph theory*

**Algorithm of Breadth-First Search:**

* **Step 1:** Consider the graph you want to navigate.
* **Step 2:** Select any vertex in your graph (say **v1**), from which you want to traverse the graph.
* **Step 3:** Utilize the following two data structures for traversing the graph.
* Visited array(size of the graph)
* Queue data structure
* **Step 4:** Add the starting vertex to the visited array, and afterward, you add v1’s adjacent vertices to the queue data structure.
* **Step 5:** Now using the FIFO concept, remove the first element from the queue, put it into the visited array, and then add the adjacent vertices of the removed element to the queue.
* **Step 6:** Repeat step 5 until the queue is not empty and no vertex is left to be visited.

**Examples:**

In the following graph, we start traversal from vertex 2.



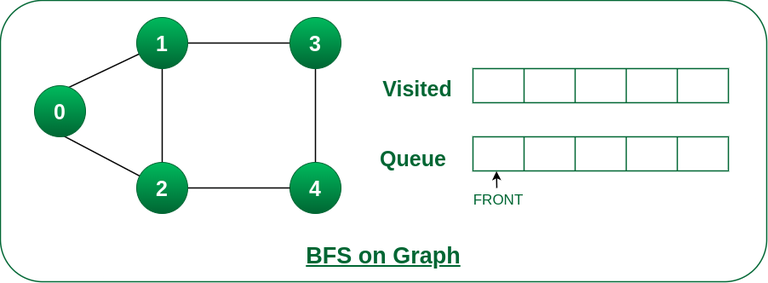
When we come to **vertex 0**, we look for all adjacent vertices of it.

* 2 is also an adjacent vertex of 0.
* If we don’t mark visited vertices, then 2 will be processed again and it will become a non-terminating process.

There can be multiple BFS traversals for a graph. Different BFS traversals for the above graph :  
2, 3, 0, 1  
2, 0, 3, 1

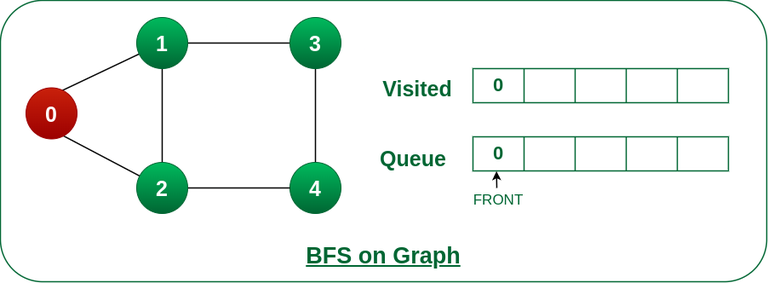
* Follow the below method to implement BFS traversal.
* Declare a queue and insert the starting vertex.
* Initialize a **visited** array and mark the starting vertex as visited.
* Follow the below process till the queue becomes empty:
  + Remove the first vertex of the queue.
  + Mark that vertex as visited.
  + Insert all the unvisited neighbors of the vertex into the queue.

**Step1:**Initially queue and visited arrays are empty.



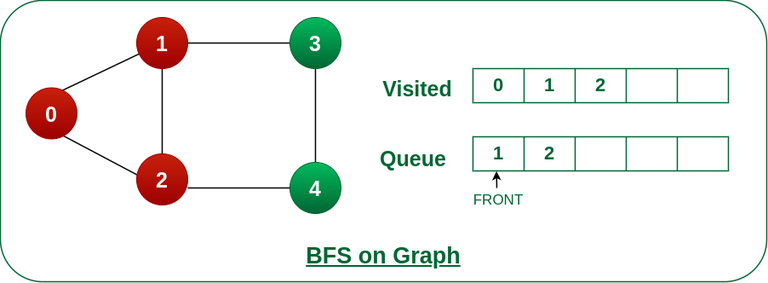
Queue and visited arrays are empty initially.

**Step2:**Push node 0 into queue and mark it visited.



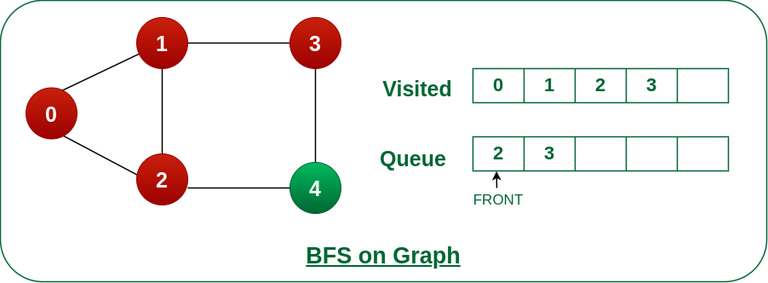
Push node 0 into queue and mark it visited.

**Step 3:** Remove node 0 from the front of queue and visit the unvisited neighbours and push them into queue.



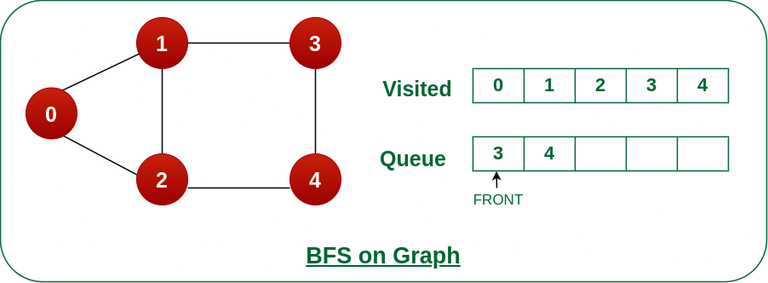
Remove node 0 from the front of queue and visited the unvisited neighbours and push into queue.

**Step 4:** Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.



Remove node 1 from the front of queue and visited the unvisited neighbours and push

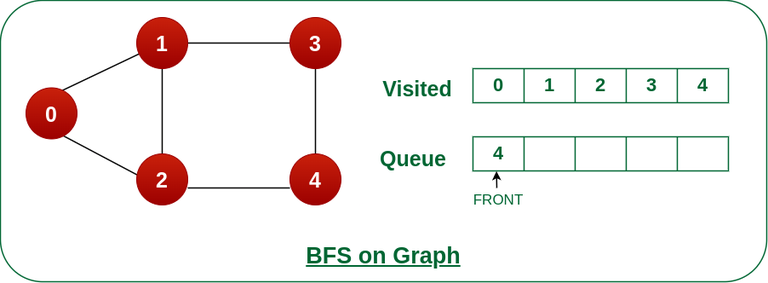
**Step 5:** Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.



Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.

**Step 6:**Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.

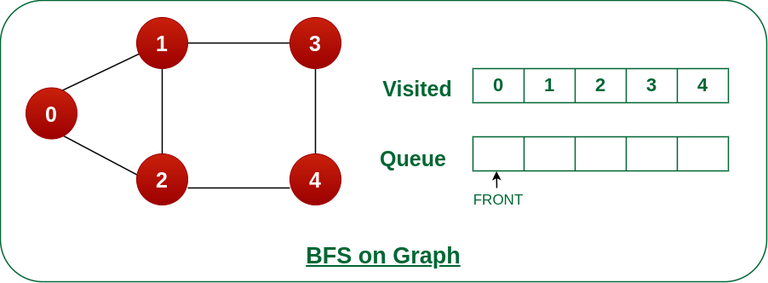
As we can see that every neighbours of node 3 is visited, so move to the next node that are in the front of the queue.



Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.

**Steps 7:**Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.

As we can see that every neighbours of node 4 are visited, so move to the next node that is in the front of the queue.



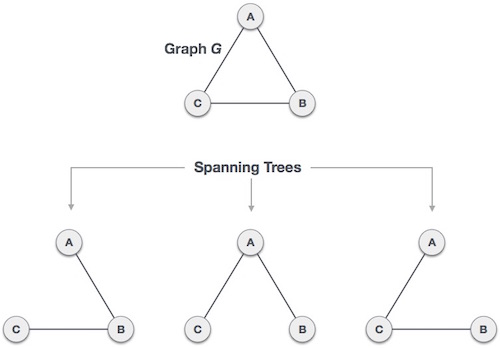
Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.

Now, Queue becomes empty, So, terminate these process of iteration.

* **5.3 Spanning Trees-**

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.



We found three spanning trees off one complete graph. A complete undirected graph can have maximum **nn-2** number of spanning trees, where **n** is the number of nodes. In the above addressed example, **n is 3,** hence **33−2 = 3** spanning trees are possible.

## General Properties of Spanning Tree

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G −

* A connected graph G can have more than one spanning tree.
* All possible spanning trees of graph G, have the same number of edges and vertices.
* The spanning tree does not have any cycle (loops).
* Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
* Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

## Mathematical Properties of Spanning Tree

* Spanning tree has **n-1** edges, where **n** is the number of nodes (vertices).
* From a complete graph, by removing maximum **e - n + 1** edges, we can construct a spanning tree.
* A complete graph can have maximum **nn-2** number of spanning trees.

Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

## Application of Spanning Tree

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are −

* **Civil Network Planning**
* **Computer Network Routing Protocol**
* **Cluster Analysis**

Let us understand this through a small example. Consider, city network as a huge graph and now plans to deploy telephone lines in such a way that in minimum lines we can connect to all city nodes. This is where the spanning tree comes into picture.

**5.4** Minimum Spanning-Tree Algorithm (Kruskal's & Prim’s Algorithm)

* **5.4.1 Kruskal's Spanning Tree Algorithm**

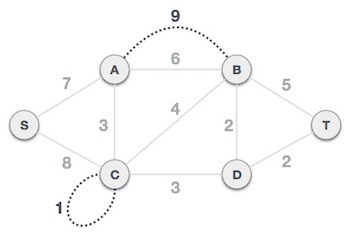
Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

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



## Step 1 − Remove all loops and Parallel Edges

Remove all loops and parallel edges from the given graph.

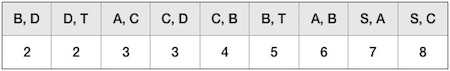


In case of parallel edges, keep the one which has the least cost associated and remove all others.



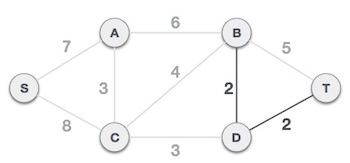
## Step 2 − Arrange all edges in their increasing order of weight

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



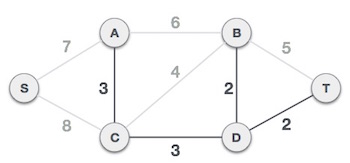
## Step 3 − Add the edge which has the least weightage

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.

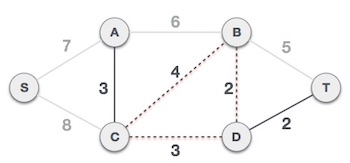


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

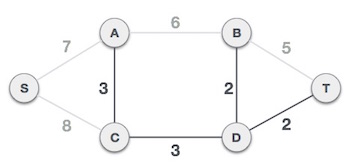
Next cost is 3, and associated edges are A,C and C,D. We add them again −



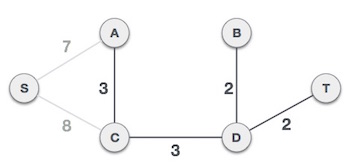
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. −



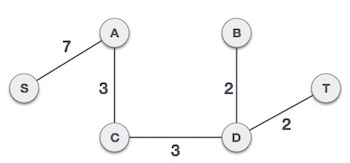
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



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.

* **5.4.2 Prim's Spanning Tree Algorithm**

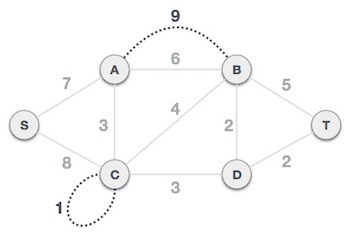
Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.

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 −



**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.

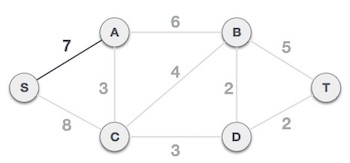


**Step 2 - Choose any arbitrary node as root node**

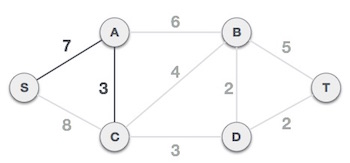
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then 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**

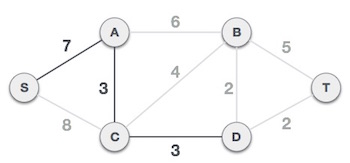
After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



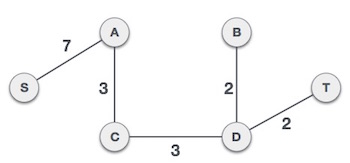
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



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.



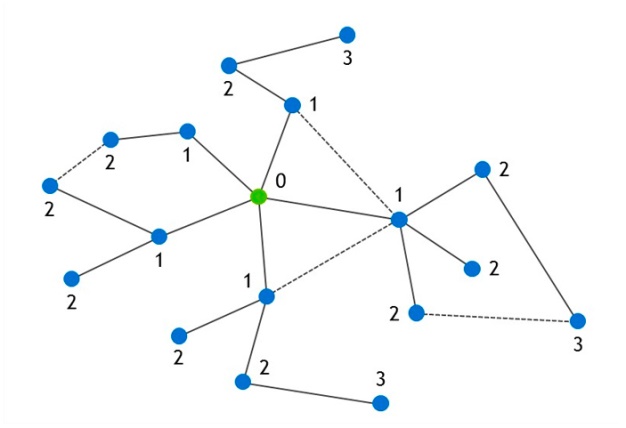
After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



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

* **5.5** Shortest Paths and All Pair Shortest Path (Dijkstra’s, Floyd- Warshall Algorithm)
  + 5.5.1 - SHORTEST PATH

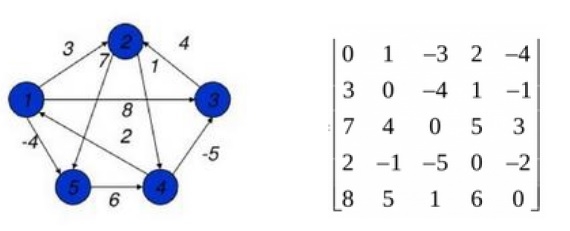
problem involves finding the shortest path between two vertices (or nodes) in a graph. Algorithms such as the Floyd-Warshall algorithm and different variations of Dijkstra's algorithm are used to find solutions to the shortest path problem. Applications of the shortest path problem include those in road networks, logistics, communications, electronic design, power grid contingency analysis, and community detection.



Shortest Path Problem

5.5.2 - ALL SHORTEST PATH PAIR

The all pair shortest path algorithm is also known as Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph. As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.



At first the output matrix is same as given cost matrix of the graph. After that the output matrix will be updated with all vertices k as the intermediate vertex.

The time complexity of this algorithm is O(V3), here V is the number of vertices in the graph.

**Input − The cost matrix of the graph.**

0 3 6 ∞ ∞ ∞ ∞

3 0 2 1 ∞ ∞ ∞

6 2 0 1 4 2 ∞

∞ 1 1 0 2 ∞ 4

∞ ∞ 4 2 0 2 1

∞ ∞ 2 ∞ 2 0 1

∞ ∞ ∞ 4 1 1 0

**Output − Matrix of all pair shortest path.**

0 3 4 5 6 7 7

3 0 2 1 3 4 4

4 2 0 1 3 2 3

5 1 1 0 2 3 3

6 3 3 2 0 2 1

7 4 2 3 2 0 1

7 4 3 3 1 1 0

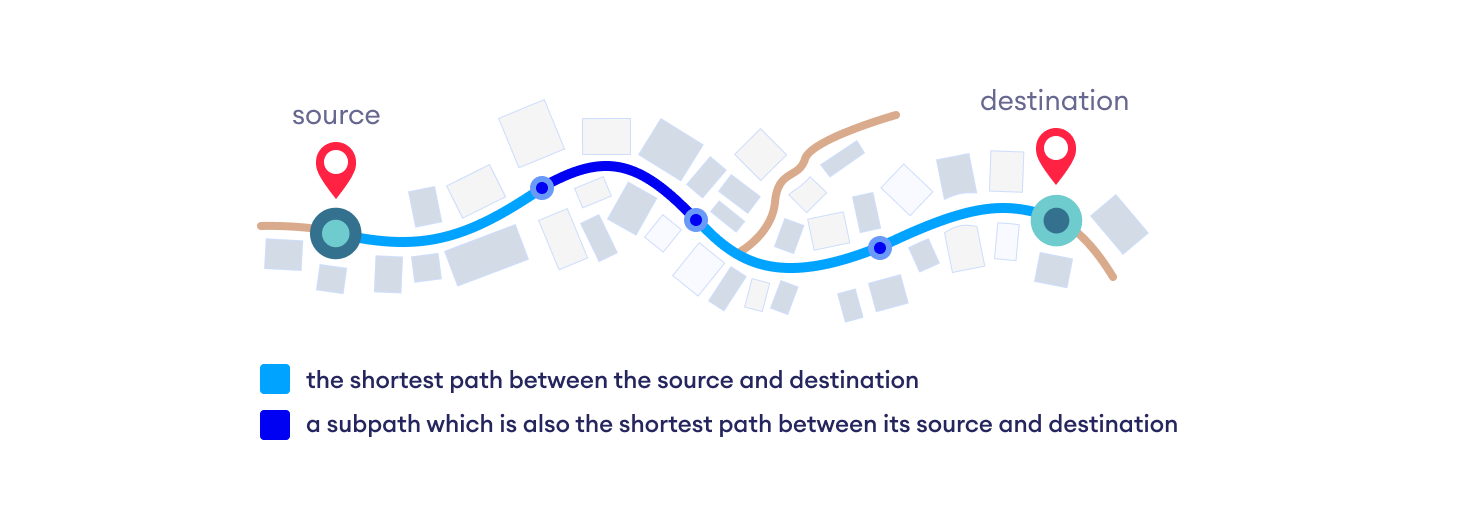
* **Dijkstra's Algorithm**

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.

It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.

* How Dijkstra's Algorithm works

Dijkstra's Algorithm works on the basis that any subpath  B -> D of the shortest path A -> D between vertices A and D is also the shortest path between vertices B and D.



Each subpath is the shortest path

Djikstra used this property in the opposite direction i.e we overestimate the distance of each vertex from the starting vertex. Then we visit each node and its neighbors to find the shortest subpath to those neighbors.

The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.

* **Dijkstra's Algorithm Complexity**

Time Complexity: O(E Log V)

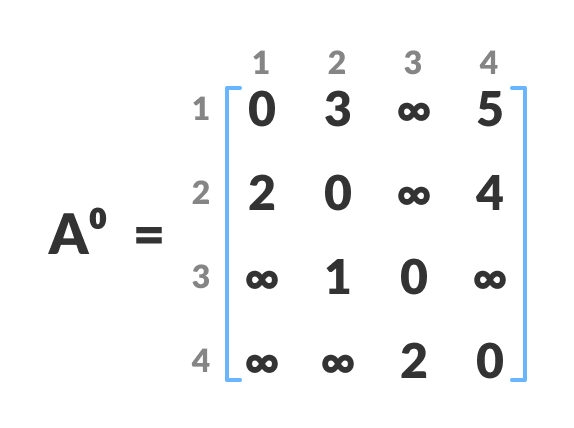
where, E is the number of edges and V is the number of vertices.

Space Complexity: O(V)

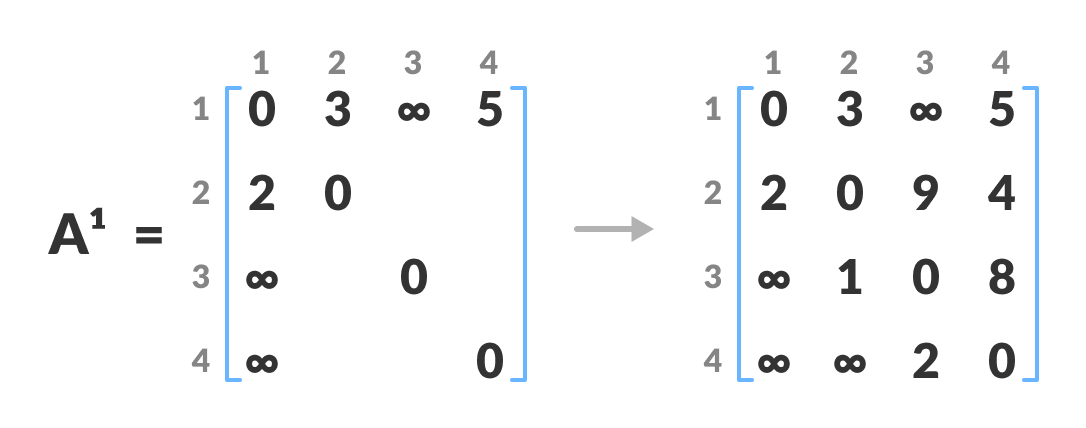
* **Dijkstra's Algorithm Applications**
* To find the shortest path
* In social networking applications
* In a telephone network
* To find the locations in the map
* **Floyd-Warshall Algorithm**
* Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices 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).
* A weighted graph is a graph in which each edge has a numerical value associated with it.
* Floyd-Warhshall algorithm is also called as Floyd's algorithm, Roy-Floyd algorithm, Roy-Warshall algorithm, or WFI algorithm.
* This algorithm follows the [dynamic programming](https://www.programiz.com/dsa/dynamic-programming) approach to find the shortest paths.

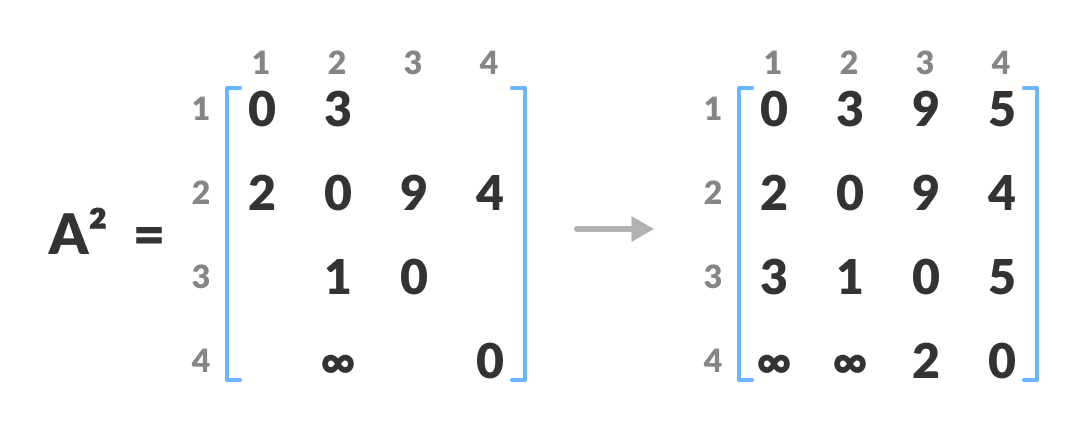
**Follow the steps below to find the shortest path between all the pairs of vertices.**

1. **Create a matrix A0 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. Fill each cell with the distance between ith and jth vertex



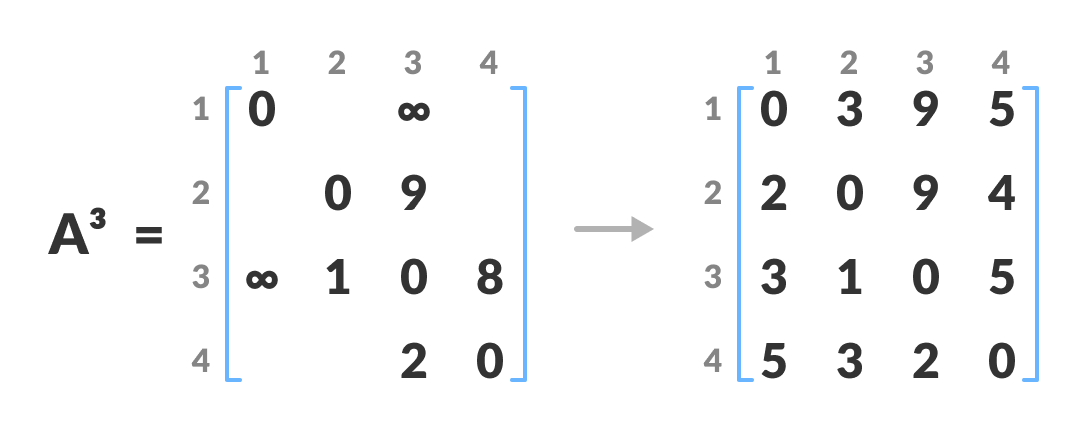
1. **Now, create a matrix A1 using matrix A0. The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.**  
   Let k be the intermediate vertex in the shortest path from source to destination. In this step, k is the first vertex. A[i][j] 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.Calculate the distance from the source vertex to destination vertex through this vertex k

  
For example: For A1[2, 4], the direct distance from vertex 2 to 4 is 4 and the sum of the distance from vertex 2 to 4 through vertex (ie. from vertex 2 to 1 and from vertex 1 to 4) is 7. Since 4 < 7, A0[2, 4] is filled with 4.

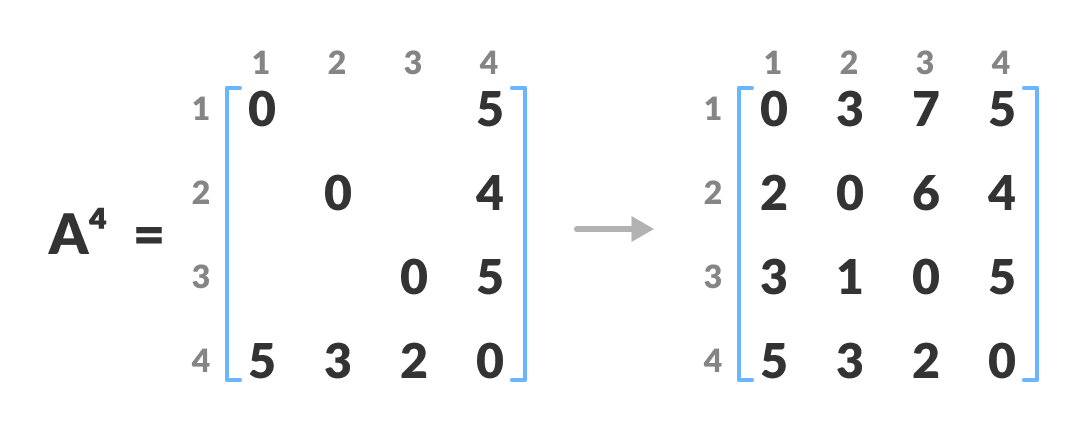
1. **Similarly, A2 is created using A1. The elements in the second column and the second row are left as they are.  
   In this step, k is the second vertex (i.e. vertex 2). The remaining steps are the same as in step 2.**

Calculate the distance from the source vertex to destination vertex through this vertex 2

1. **Similarly, A3 and A4 is also created.**



Calculate the distance from the source vertex to destination vertex through this vertex



Calculate the distance from the source vertex to destination vertex through this vertex 4

1. **A4 gives the shortest path between each pair of vertices.**

## Floyd Warshall Algorithm Complexity

### Time Complexity

### There are three loops. Each loop has constant complexities. So, the time complexity of the Floyd-Warshall algorithm is O(n3).

### Space Complexity

The space complexity of the Floyd-Warshall algorithm is O(n2).