UNIT 5

Syllabus: Graph ADT Graph: Introduction – Representations – Traversals - Topological Sorting – Connected and Bi-Connected Components – Articulation Point – Shortest path algorithms (Dijkstra's and Floyd's algorithms) - Minimum spanning tree (Prim's and Kruskal's algorithms).

Graph Introduction

Graphs are non-linear data structures representing the "connections" between the elements. These elements are known as the **Vertices**, and the lines or arcs that connect any two vertices in the graph are known as the **Edges**. More formally, a Graph comprises **a set of Vertices (V)** and **a set of Edges (E)**. The Graph is denoted by **G(V, E)**.

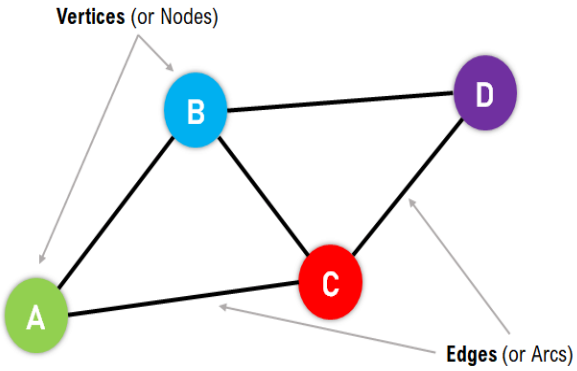


Figure 1: Graphical Representation of a Graph

Components of a Graph

1. **Vertices:** Vertices are the basic units of the graph used to represent real-life objects, persons, or entities. Sometimes, vertices are also **known as Nodes.**
2. **Edges:** Edges are drawn or used to connect two vertices of the graph. Sometimes, edges are also **known as Arcs.**

In the above figure, the Vertices/Nodes are denoted with Colored Circles, and the Edges are denoted with the lines connecting the nodes.

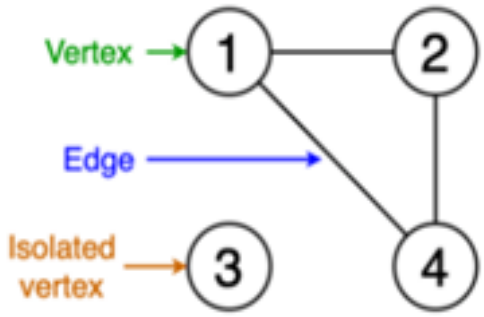


Figure 3: A Simple Undirected Graph

Types of Graphs

The Graphs can be categorized into two types:

1. Undirected Graph
2. Directed Graph

1. **Undirected Graph:** A Graph with edges that do not have a direction is termed an Undirected Graph. The edges of this graph imply a two-way relationship in which each edge can be traversed in both directions. The following figure displays a simple undirected graph with four nodes and five edges.

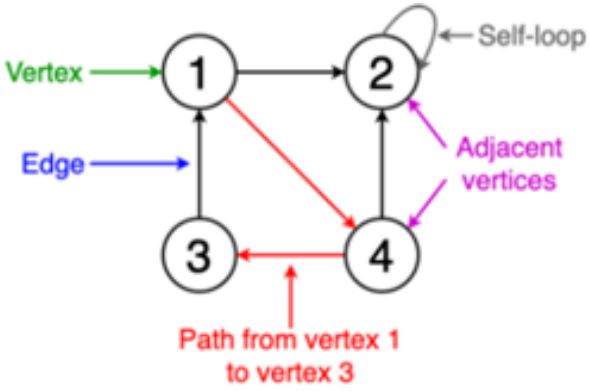
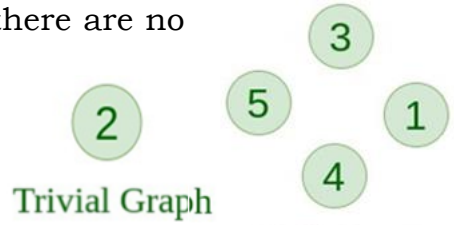


Figure 4: A Simple Directed Graph

2. **Directed Graph:** A Graph with edges with direction is termed a Directed Graph. The edges of this graph imply a one-way relationship in which each edge can only be traversed in a single direction. The following figure displays a simple directed graph with four nodes and five edges.

3. **Null Graph -** A graph is known as a null graph if there are no edges in the graph.

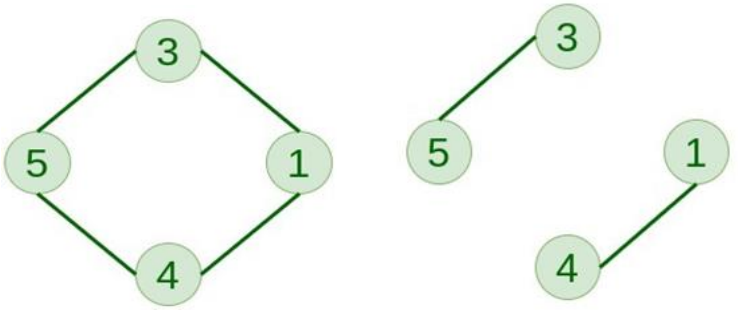
4. **Trivial Graph -** Graph having only a single vertex, it is also the smallest graph possible.



Trivial Graph Null Graph

5. **Connected Graph -** The graph in which from one node we can visit any other node in the graph is known as a connected graph.

6. **Disconnected Graph -** The graph in which at least one node is not reachable from a node is known as a disconnected graph.

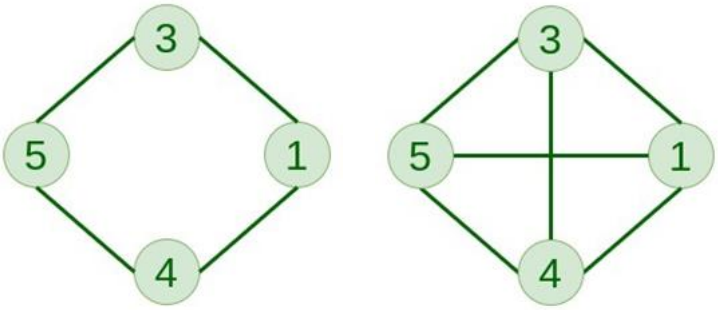


Connected Graph Disconnected Graph

7. **Regular Graph -** The graph in which the **degree of every vertex is equal** to K is called K regular graph.

8. **Complete Graph -** The graph in which from **each node** there is an **edge to each other node.**

9. **Cycle Graph -** The graph in which the **graph is a cycle in itself,** the degree of each **vertex is 2.**

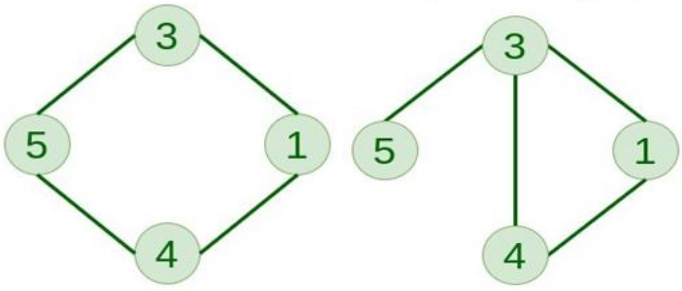


2-Regular Complete Graph

10. **Cyclic Graph -** A graph containing at **least one cycle** is known as a Cyclic graph.

11. **Directed Acyclic Graph -** A Directed Graph that **does not contain any cycle.**

12. **Bipartite Graph -** A graph in which vertex can be divided into two sets such that vertex in each set does not contain any edge between them.



Cycle Graph Cyclic Graph

13. **Weighted Graph -** A graph in which the edges are already specified with suitable weight is known as a weighted graph. Weighted graphs can be further classified as directed weighted graphs and undirected weighted graphs.

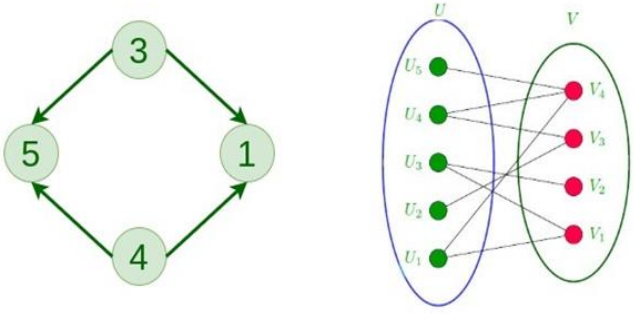
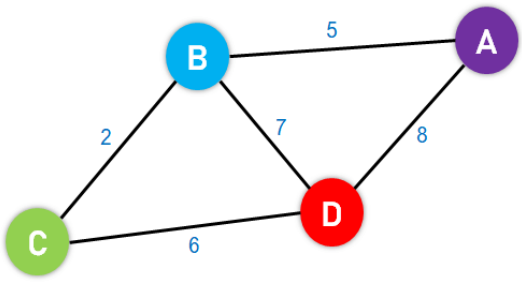


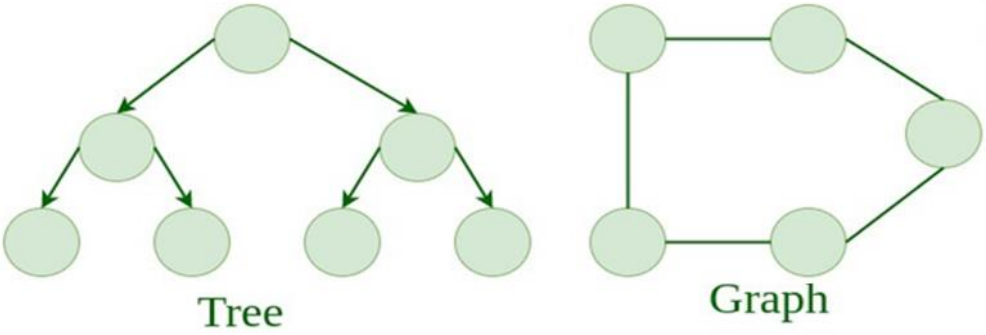
Figure 5: An Example of a Weighted Graph

Directed Acyclic Graph Bipartite Graph

A Graph is said to be Weighted if each edge is assigned a 'weight'. The weight of an edge can denote distance, time, or anything that models the 'connection' between the pair of vertices it connects. For instance, we can observe a blue number next to each edge in the following figure of the Weighted Graph. This number is utilized to signify the weight of the corresponding edge.



Tree v/s Graph



Comparison	Graph	Tree
Definition	Graph is a non-linear data structure.	Tree is a non-linear data structure.
Structure	It is a collection of vertices/nodes and edges.	It is a collection of nodes and edges.
Structure cycle	A graph can be connected or disconnected, can have cycles or loops, and does not necessarily have a root node.	A tree is a type of graph that is connected, acyclic (meaning it has no cycles or loops), and has a single root node.
Edges	Each node can have any number of edges.	If there is n nodes then there would be n-1 number of edges
Types of Edges	They can be directed or undirected	They are always directed
Root node	There is no unique node called root in graph.	There is a unique node called root(parent) node in trees.
Loop Formation	A cycle can be formed.	There will not be any cycle.
Traversal	For graph traversal, we use Breadth-First Search (BFS), and Depth-First Search (DFS).	We traverse a tree using in-order, pre-order, or post-order traversal methods.
Applications	For finding shortest path in networking graph is used.	For game trees, decision trees, the tree is used.
Node relationships	In a graph, nodes can have any number of connections to other nodes, and there is no	In a tree, each node (except the root node) has a parent

Comparison	Graph	Tree
	strict parent-child relationship.	node and zero or more child nodes.
Commonly used for	Graphs are commonly used to model complex systems or relationships, such as social networks, transportation networks, and computer networks.	Trees are commonly used to represent data that has a hierarchical structure, such as file systems, organization charts, and family trees.
Connectivity	In a graph, nodes can have any number of connections to other nodes.	In a tree, each node can have at most one parent, except for the root node, which has no parent.

Basic Operations on Graphs - Below are the basic operations on the graph:

- **Insertion of Nodes/Edges in the graph** – Insert a node into the graph.
- **Deletion of Nodes/Edges in the graph –** Delete a node from the graph.
- **Searching on Graphs –** Search an entity in the graph.
- **Traversal of Graphs –** Traversing all the nodes in the graph.

Usage of graphs

- Maps can be represented using graphs and then can be used by computers to provide various services like the **shortest path between two cities.**
- When various tasks depend on each other then this situation can be represented using a **Directed Acyclic graph and we can find the order** in which tasks can be performed using **topological sort.**
- **State Transition Diagram** represents what can be the legal moves from current states. In-game of **tic tac toe** this can be used.

Real-Life Applications of Graph

1. Graph data structures can be used to represent the **interactions between players on a team**, such as passes, shots, and tackles.
2. **Analyzing** these **interactions** can provide insights into team dynamics and areas for improvement.
3. Commonly used to **represent social networks**, such as networks of friends on social media.
4. Graphs can be used to represent the **topology of computer networks,** such as the connections between routers and switches.
5. Graphs are used to represent the **connections between different places** in a transportation network, such as roads and airports.
6. **Neural Networks:** Vertices represent neurons and edges represent the synapses between them. Neural networks are used to **understand how our brain works and how connections change** when we learn.
7. **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.

8. **Robot planning:** Vertices represent states the robot can be in and the edges the possible transitions between the states. Such graph plans are used, for example, in planning paths for autonomous vehicles.

Advantages:

1. Graphs are a **versatile data structure** that can be used to represent a wide range of relationships and data structures.
2. They can be **used to model** and **solve a wide range of problems,** including pathfinding, data clustering, network analysis, and machine learning.
3. Graph algorithms are often **very efficient** and can be used to solve **complex problems quickly and effectively**.
4. Graphs can be used to represent complex data structures in a simple and intuitive way, making them **easier to understand and analyze.**

Disadvantages:

1. Graphs can be **complex and difficult to understand**, especially for people who are not familiar with graph theory or related algorithms.
2. Creating and manipulating graphs can be **computationally expensive,** especially for very large or complex graphs.
3. Graph algorithms can be **difficult to design and implement correctly**, and can be prone to bugs and errors.
4. Graphs can be **difficult to visualize and analyze,** especially for very large or complex graphs, which can make it challenging to extract meaningful insights from the data.

Graph Representations

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.

There are two ways to store a graph:

- Adjacency Matrix
- Adjacency List

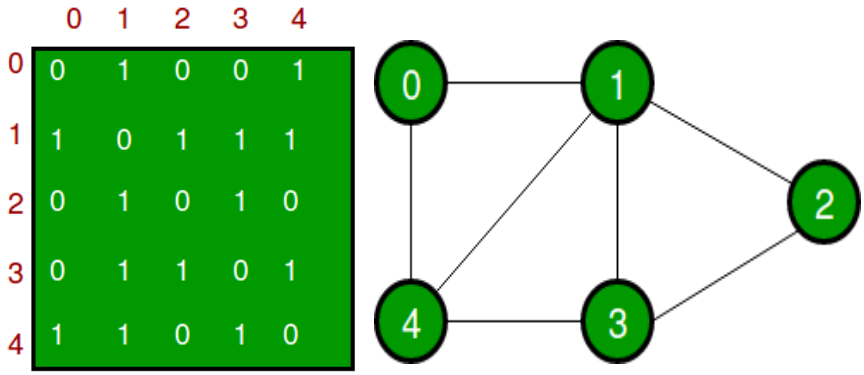
Adjacency Matrix - An adjacency matrix is a **two-dimensional array** that **represents the graph** by storing a **1** at position (i, j) if there is an **edge from vertex i to vertex j**, and **0 otherwise**.

In this method, the **graph is stored** in the form of the **2D matrix** where **rows and columns denote vertices**. Each entry in the matrix represents the weight of the edge between those vertices. 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.

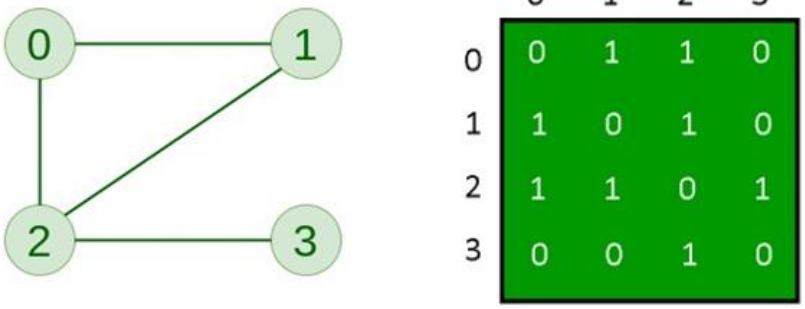
Adjacency Matrix is also used to represent weighted graphs. If **adj[i][j] = w**, then there is an edge from vertex **i** to vertex **j** with weight **w**.

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**.



Example of undirected graph with 5 & 4 vertices. The adjacency matrix for the above example graph is:



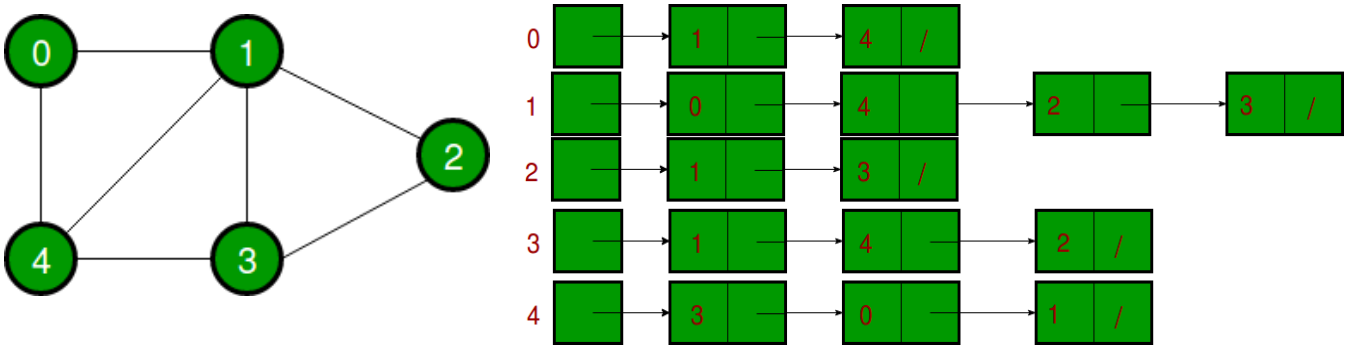
Advantages of Adjacency Matrix:

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

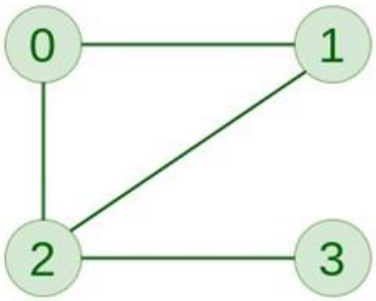
Disadvantages of Adjacency Matrix:

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

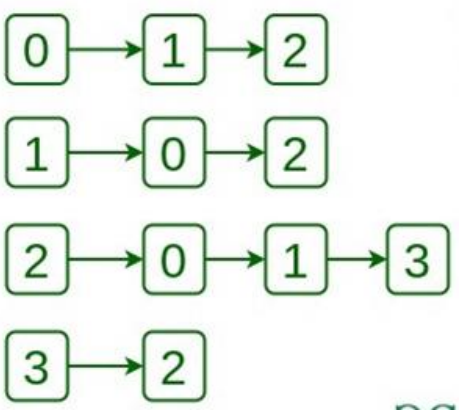
Adjacency List - An adjacency list is a simple **way to represent** a graph as a list of vertices, where each **vertex has a list** of its adjacent vertices. This graph is represented as a **collection of linked lists**. There is an **array of pointer** which points to the edges connected to that vertex.



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
Following is the 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 neighbours 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).

Comparison between Adjacency Matrix and Adjacency List

When the graph contains a large number of edges then it is good to store it as a matrix because only some entries in the matrix will be empty. An algorithm such as Prim's and Dijkstra adjacency matrix is used to have less complexity.

Action	Adjacency Matrix	Adjacency List
Adding Edge	O(1)	O(1)
Removing an edge	O(1)	O(N)
Initializing	O(N*N)	O(N)

Graph Traversals

The graph has two types of traversal algorithms. These are called the Breadth First Search and Depth First Search.

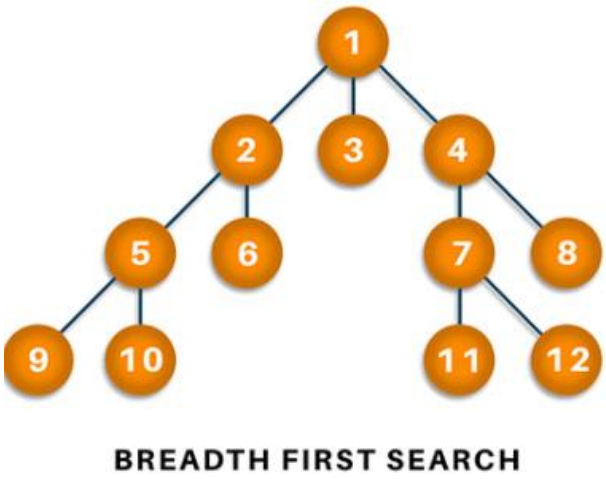
Breadth-First Search - Traversing or searching is one of the most used operations that are undertaken while working on graphs. Therefore, in **breadth-first-search** (BFS), you start at a particular vertex, and the algorithm tries to visit all the neighbors at the given depth before moving on to the next level of traversal of vertices. Unlike trees, graphs may contain cyclic paths where the first and last vertices are remarkably the same always. Thus, in BFS, you need to keep note of all the track of

the vertices you are visiting. To implement such an order, you use a queue data structure which First-in, First-out approach.

To avoid processing a node more than once, we divide the vertices into two categories:

- Visited and
- Not visited.

A boolean visited array is used to mark the visited vertices. For simplicity, it is assumed that all vertices are reachable from the starting vertex. BFS uses a **queue data structure** for traversal.



BREADTH FIRST SEARCH

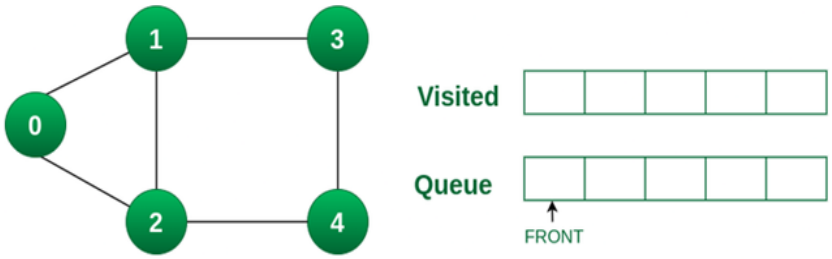
Algorithm

1. Start putting anyone vertices from the graph at the back of the queue.
2. First, move the front queue item and add it to the list of the visited node.
3. Next, create nodes of the adjacent vertex of that list and add them which have not been visited yet.
4. Keep repeating steps two and three until the queue is found to be empty.

How does BFS work?

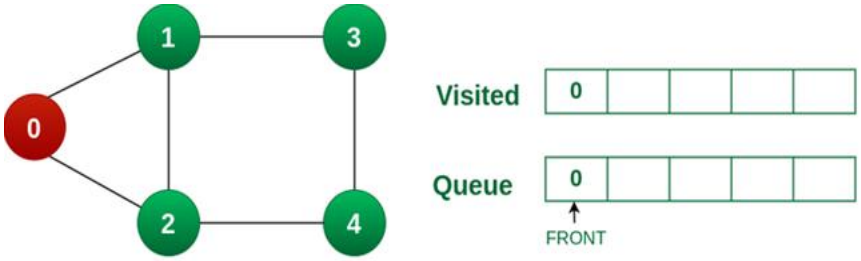
Starting from the root, all the nodes at a particular level are visited first and then the nodes of the next level are traversed till all the nodes are visited. To do this a queue is used. All the adjacent unvisited nodes of the current level are pushed into the queue and the nodes of the current level are marked visited and popped from the queue.

Example: Let us understand the working of the algorithm with the help of the following example.



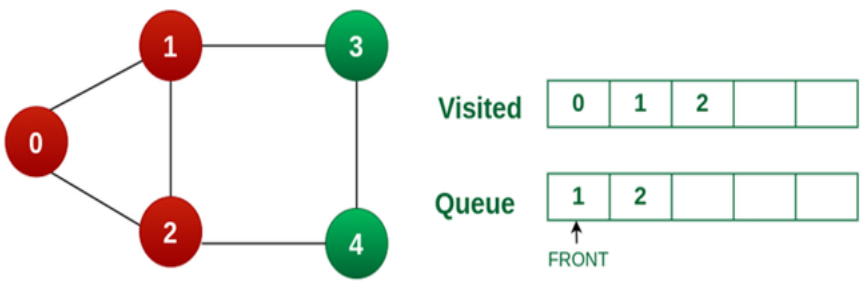
Step1: Initially queue and visited arrays are empty.

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.

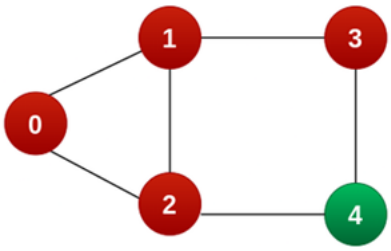


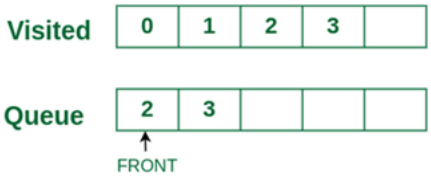
Step 3: Remove node 0 from the front of queue and visit the unvisited neighbours and push them into queue.

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.

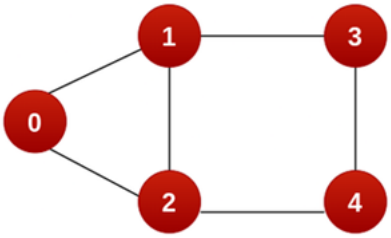
Remove node 1 from the front of queue and visited the unvisited neighbours and push

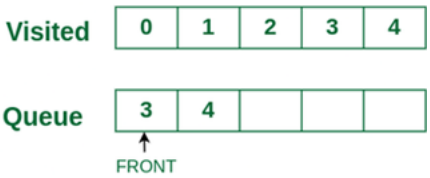




Step 5: Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.

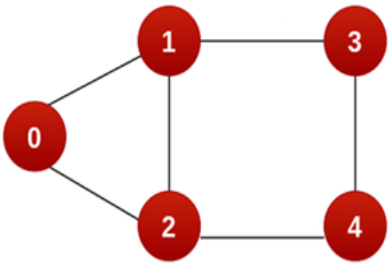
Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.

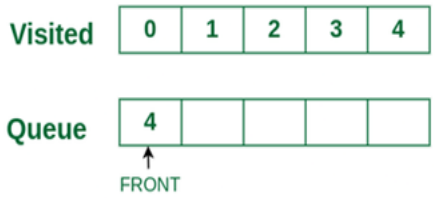




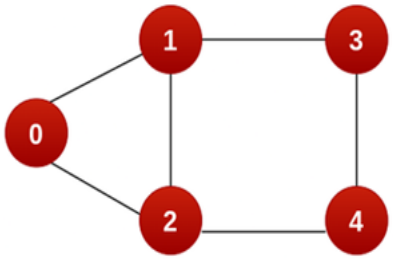
Step 6: Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.

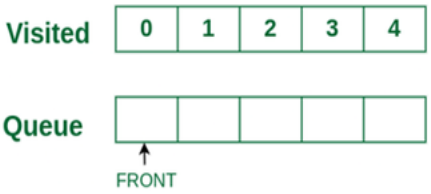
As we can see that every neighbours of node 3 is visited, so move to the next node that are in the front of the queue. Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.





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.





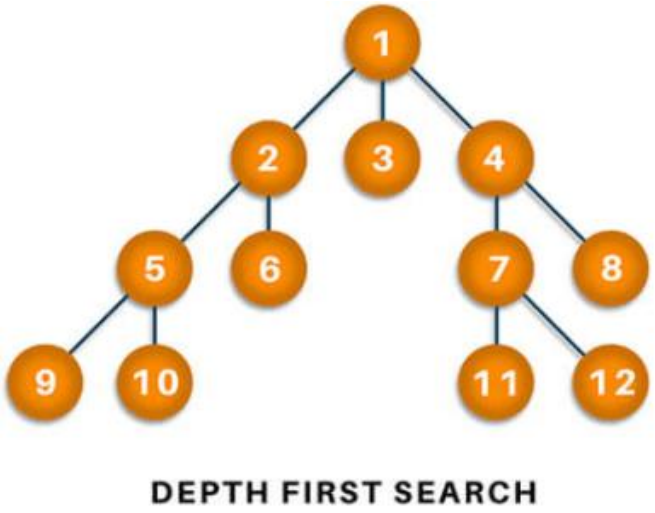
Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue. Now, Queue becomes empty, So, terminate these process of iteration

Time Complexity: O(V+E) **Auxiliary Space:** O(V)
where V is the number of nodes and E is the number of edges.

Applications

1. It is used to determine the **shortest path**
2. It is used in **minimum spanning tree.**
3. It is also used in web crawlers to creates web page indexes.
4. It is also used as powering search engines on social media networks and helps to find out peer-to-peer networks in BitTorrent.

Depth-first search - Depth First Traversal (**DFS)** for a graph is **similar to Depth First Traversal of a tree.** The only catch here is, that, unlike trees, **graphs may contain cycles** (a node may be visited twice). To avoid processing a node more than once, use a **boolean visited array.**



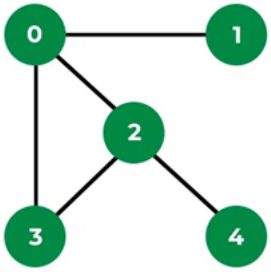
DEPTH FIRST SEARCH

A graph can have more than one DFS traversal. Depth-first search is an algorithm for **traversing or searching tree** or graph data structures. The algorithm **starts at the root node** (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. In depth-first-search (DFS), you **start** by particularly from the **vertex** and **explore as much as** you along all the branches before backtracking. In DFS, it is essential to keep note of the tracks of visited nodes, and for this, you **use stack data structure.**

Algorithm

1. **Start** by putting **one of the vertexes** of the graph on the stack's top.
2. **Put the top item** of the stack and add it to the **visited vertex list.**
3. **Create a list of all the adjacent nodes** of the vertex and then add those nodes to the **unvisited at the top** of the stack.
4. Keep **repeating steps 2 and 3,** and the stack becomes empty.

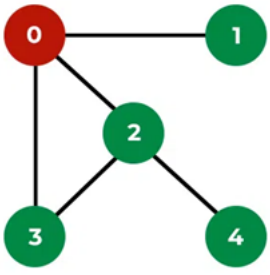
Example: Let us understand the working of **Depth First Search** with the help of the following illustration:




Visited


Stack

Step1: Initially stack and visited arrays are empty.



Visited

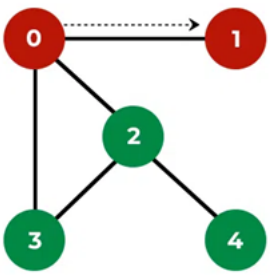
Step 2: Visit 0 and put its adjacent nodes which are not visited yet into the stack.

Visit node 0 and put its adjacent nodes (1, 2, 3) into the stack

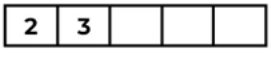
Stack

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.

Visit node 1

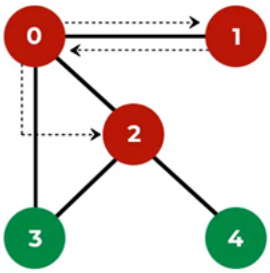


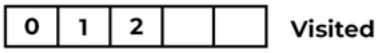
Visited

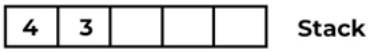


Stack

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.

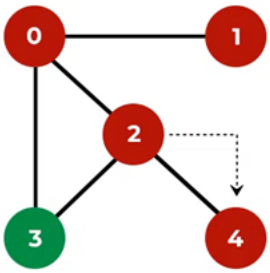


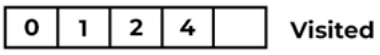




Visit node 2 and put its unvisited adjacent nodes (3, 4) into the stack

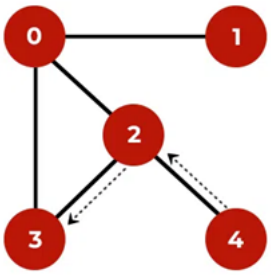
Step 5: Now, Node 4 at the top of the stack, so visit node 4 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.

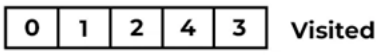


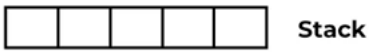


Visit node 4

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.







Visit node 3, Now, Stack becomes empty, which means we have visited all the nodes and our DFS traversal ends.

Time complexity: O(V + E), **Auxiliary Space:** O(V + E)
where V is the number of vertices and E is the number of edges in the graph. Since an extra visited array of size V is required, And stack size for iterative call to DFS function.

Applications

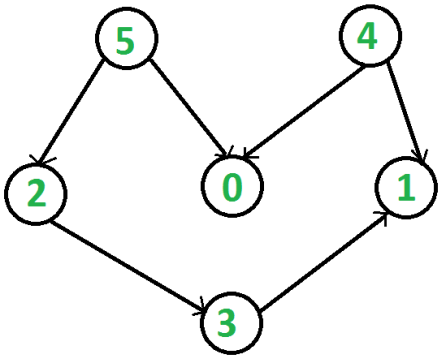
1. DFS finds its application when it comes to finding paths between two vertices and detecting cycles.
2. Topological sorting can be done using the DFS algorithm easily.
3. DFS is also used for one-solution puzzles.

Topological Sorting

A topological sort or topological ordering of a directed graph is a **linear ordering of its vertices** in which **u occurs before v** in the ordering for every directed edge uv from vertex u to vertex v. **For example,** the graph's vertices could represent jobs to be completed, and the edges could reflect requirements that one work must be completed before another.

In this case, a topological ordering is just a legitimate task sequence. A topological sort is a **graph traversal** in which each node **v is only visited after all of its dependencies have been visited**. If the graph contains **no directed cycles**, then it is a **directed acyclic graph**. Any DAG has at least one topological ordering, and there exist techniques for building topological orderings in linear time for any DAG.

Topological sorting has many applications, particularly in **ranking issues** like the feedback arc set. Even if the DAG **includes disconnected components**, topological sorting is possible. Topological sorting for Directed Acyclic Graph (DAG) is a **linear ordering of vertices** such that for every directed edge u v, vertex u comes before v in the ordering. Topological Sorting for a graph is **not possible** if the graph is **not a DAG.**



For example, a topological sorting of the following graph is "5 4 2 3 1 0". There can be more than one topological sorting for a graph. Another topological sorting of the following graph is "4 5 2 3 1 0". The first vertex in **topological sorting is always** a vertex with an **in-degree of 0** (a vertex with no incoming edges).

Topological Sorting vs Depth First Traversal (DFS):

In DFS, we **print a vertex** and then **recursively call** DFS for its adjacent vertices. In topological sorting, we need to **print a vertex before** its adjacent vertices.

For example, in the given graph, the vertex '5' should be printed before vertex '0', but unlike DFS, the vertex '4' should also be printed before vertex '0'. So Topological sorting is different from DFS. For example, a DFS of the shown graph is "5 2 3 1 0 4", but it is not a topological sorting.

Algorithm for Topological Sorting:

Prerequisite: DFS, We can modify DFS to find the Topological Sorting of a graph.

1. We start from a vertex, we first print it, and then
2. Recursively call DFS for its adjacent vertices.
   a. In topological sorting,
3. We use a temporary stack.
4. We don't print the vertex immediately,
5. we first recursively call topological sorting for all its adjacent vertices, then push it to a stack.
6. Finally, print the contents of the stack.

A vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in the stack

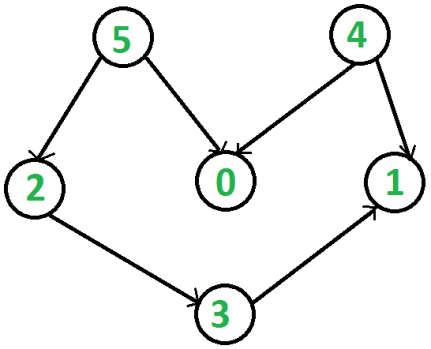
Approach:

1. Create a **stack** to store the nodes.
2. Initialize **visited** array of size **N** to keep the record of visited nodes.
3. Run a loop from **0** till **N**
   3.1. if the node is not marked **True** in **visited** array
   3.2. Call the recursive function for topological sort and perform the following steps.
4. Mark the current node as **True** in the **visited** array.
5. Run a loop on all the nodes which has a directed edge to the current node
   5.1. if the node is not marked **True** in the **visited** array:

5.1.1. Recursively call the topological sort function on the node
5.1.2. Push the current node in the stack.

6. Print all the elements in the stack.

Below image is an illustration of the above approach:



Adjacent List (G)
0 →
1 →
2 → 3
3 → 1
4 → 0, 1
5 → 2, 0

Visited Nodes	0	1	2	3	4	5
Visit Status	False	False	False	False	False	False

Step 1: Topological Sort (0) Visited Node [0] = True
List is empty no more recursion call

Stack	0					

Step 2: Topological Sort (1) Visited Node [1] = True
List is empty no more recursion call

Stack	0	1				

Step 3: Topological Sort (2) Visited Node [2] = True
Topological Sort (3) Visited Node [3] = True
1 is already visited no more recursive call

Stack	0	1	3	2		

Step 4: Topological Sort (4) Visited Node [4] = True
0, 1 is already visited no more recursive call

Stack	0	1	3	2	4	

Step 5: Topological Sort (5) Visited Node [5] = True
2, 0 is already visited no more recursive call

Stack	0	1	3	2	4	5

Step 6: Print all element from top to bottom

Visited Nodes	0	1	2	3	4	5
Visit Status	True	True	True	True	True	True

Advantages of Topological Sorting

1. Topological Sorting is mostly used to **schedule jobs** based on their dependencies. Instruction scheduling, ordering formula cell evaluation when recomputing formula values in spreadsheets.

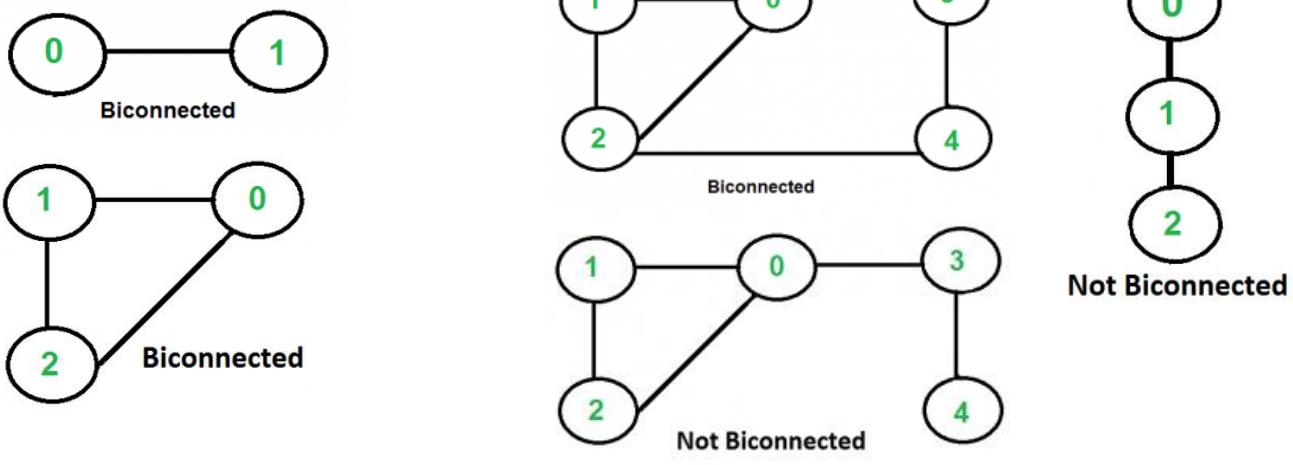
2. **Determining the order of compilation** tasks to perform in make files, data serialization, and resolving symbol dependencies in linker are all examples of applications of this type in computer science.
3. **Finding cycle in a graph:** Only directed acyclic graphs may be ordered topologically (DAG). It is impossible to arrange a circular graph topologically.
4. **Operation System deadlock detection:** A deadlock occurs when one process is waiting while another holds the requested resource.
5. **Dependency resolution:** Topological Sorting has been proved to be very helpful in Dependency resolution.
6. **Sentence Ordering:** The order of the sentences can be represented using a DAG. Here the sentences (Si) represent the vertices, and the edges represent the ordering between sentences. For example, if we have a directed edge between S1 to S2, then S1 must come before S2.
7. **Critical Path Analysis:** A project management approach known as critical route analysis. It's used to figure out how long a project should take and how dependent each action is on the others.
8. **Course Schedule problem:** Topological Sorting has been proved to be very helpful in solving the Course Schedule problem.

Connected and Bi-Connected Components

An undirected graph is called Biconnected if there are two vertex-disjoint paths between any two vertices. In a Biconnected Graph, there is a simple cycle through any two vertices. By convention, two nodes connected by an edge form a biconnected graph, but this does not verify the above properties. For a graph with more than two vertices, the above properties must be there for it to be Biconnected. Or in other words A graph is said to be Biconnected if:

1. It is connected, i.e. it is possible to reach every vertex from every other vertex, by a simple path.
2. Even after removing any vertex the graph remains connected.

Following are some examples:



How to find if a given graph is Biconnected or not?

A connected graph is Biconnected if it is connected and doesn't have any Articulation Point. We mainly need to check two things in a graph.

1. The graph is connected.

2. There is not articulation point in graph.

We start from any vertex and do DFS traversal. In DFS traversal, we check if there is any articulation point. If we don't find any articulation point, then the graph is Biconnected. Finally, we need to check whether all vertices were reachable in DFS or not. If all vertices were not reachable, then the graph is not even connected

Biconnected components

A maximal biconnected subgraph is a Biconnected component. Its really simple if you understand the sentence carefully. Take a biconnected sub-graph of a graph, then keep on adding the neighbouring vertices and edges and take a step back if you find an articulation point. So now you will have a biconnected sub-graph of the given graph with maximum nodes possible such that there are no articulation point.

In simple words a biconnected component is a part of a graph that is connected in a special way. Imagine you have a graph with nodes (called vertices) and lines (called edges) connecting them. A biconnected component is a group of vertices and edges that are all connected to each other in a way that you can always get from one vertex to another using two different paths.

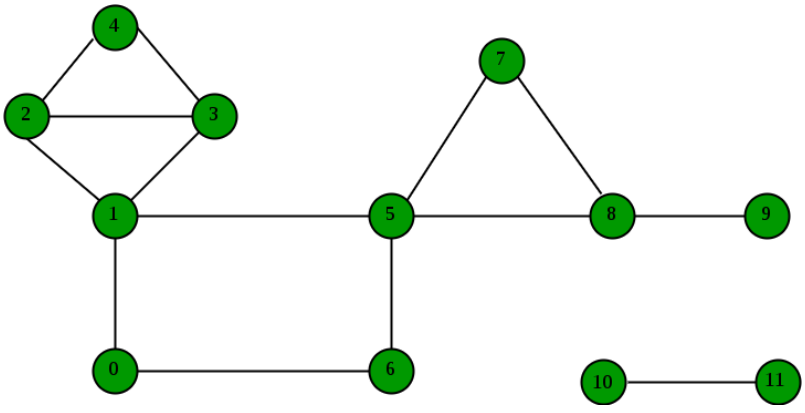
The concept of biconnected components is based on the concept of the disc and low values algorithms. This article aims to focus on the algorithm developed by Robert Tarjan and John Hopcroft's view of biconnected components in a graph data structure. The maximal sub-graph is known to be the concept behind Biconnected Components.

Biconnected components are a part of the graph data structure created. When the articulation points are found, the edges stored in the stack data structure will form an entity of biconnected components. A biconnected component is a maximal biconnected subgraph.

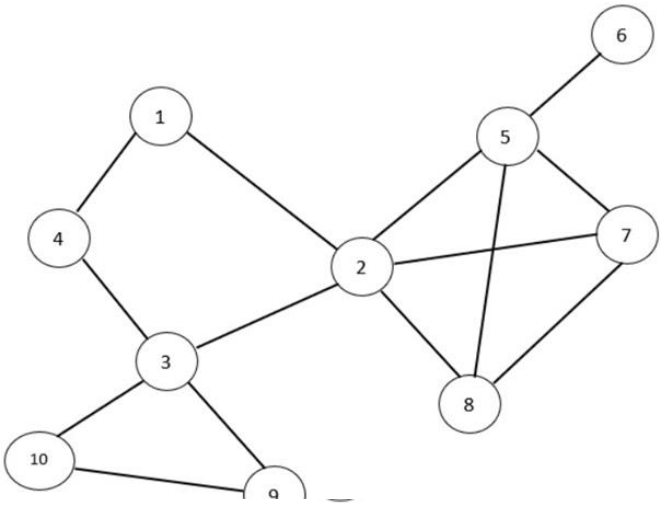
In graph shown below, following are the biconnected components:

4–2 3–4 3–1 2–3 1–2
8–9
8–5 7–8 5–7
6–0 5–6 1–5 0–1
10–11

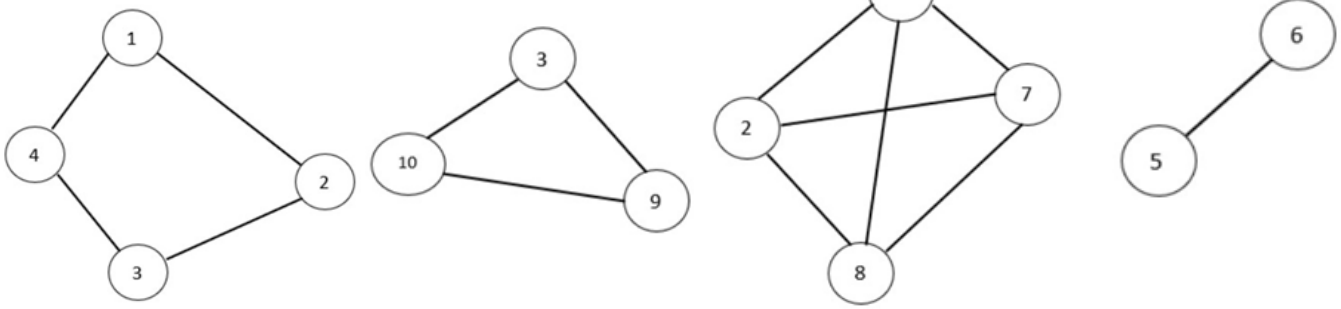
Algorithm is based on Disc and Low Values. Idea is to store visited edges in a stack while DFS on a graph and keep looking for Articulation Points (highlighted in above figure). As soon as an Articulation Point u is found, all edges visited while DFS from node u onwards will form one biconnected component. When DFS completes for one connected component, all edges present in



stack will form a biconnected component. If there is no Articulation Point in graph, then graph is biconnected and so there will be one biconnected component which is the graph itself.



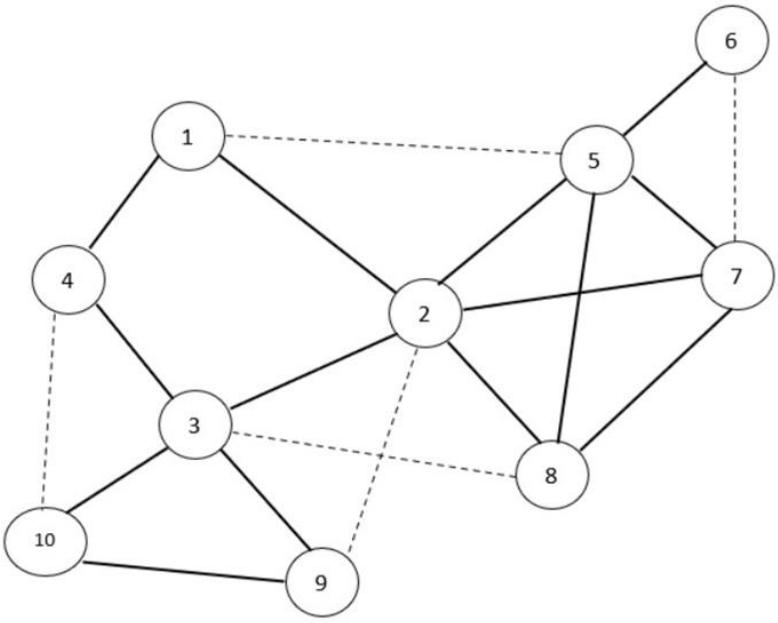
Example: Here the given graph has 3 articulation points ie, vertex 2, 3, 5, as the graph splits into various parts if we remove one vertex and the edges incident to it. So the following are the biconnected components in the graph. As you might observe if we take two of them together, they will have at most 1 vertex in common and that vertex is none other than the articulation point. Take any pair and check.



How To Build A Biconnected Graph - Steps to follow to build a biconnected graph

1. Check whether the given graph is biconnected or not.
2. If its not biconnected identify all the articulation point.
3. After obtaining the articulation points, determine the set of edges whose inclusion makes the graph connected in the absence of articulation point.

Now lets look the steps for our graph Fig.



1. The given graph G is not a biconnected graph, the articulation points are 2,3 and 5.
2. To transform G to biconnected graph new edges are included to te articulation point.
   - Edges corresponding to the articulation point 2 are (1,5) and (3,8).
   - Edges corresponding to the articulation point 3 are (4,10) and (2,9).
   - Edges corresponding to the articulation point 5 are (6,7).

So lets add these edges. The edges added are represented as dotted lines. After the edition of these lines we can observe that it is a biconnected graph.

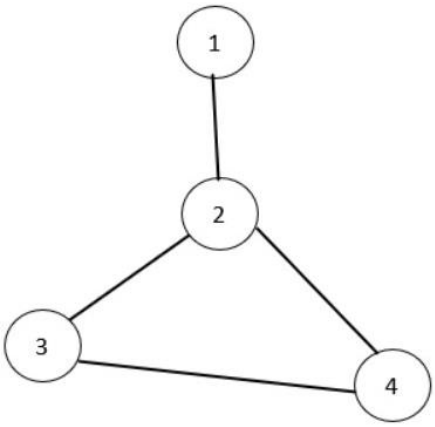
Identify Biconnected Components

How to find a biconnected component in a graph. For this we will use Tarjan's algorithm for finding biconnected components in a graph. It was developed by Robert Tarjan in 1972.

Algorithmic steps:

1. Initialize a counter dfn to 1, which will be used to assign a depth-first search (DFS) number to each vertex. Initialize an empty stack S and a list of biconnected components components.
2. Run a DFS on the graph, keeping track of the following information for each vertex v:
   - dfn[v]: the DFS number of v
   - low[v]: the lowest DFS number reachable from v
   - parent[v]: the parent of v in the DFS tree
3. For each vertex v visited in the DFS, do the following:
   - If v is the root of the DFS tree, and it has more than one child, add all of its children to the current biconnected component C.
   - If v is not the root of the DFS tree, and low[v] >= dfn[v], then v is an articulation point. In this case, add v and all vertices on the stack S that have a DFS number greater than low[v] to the current biconnected component C.
4. For each edge (u, v) in the graph, do the following:
   - If dfn[u] < low[v], then (u, v) is a back edge. In this case, set low[v] = dfn[u].
   - If dfn[u] > dfn[v], then (u, v) is a forward edge. In this case, push v onto the stack S.
5. When the DFS is complete, the list of biconnected components components will contain all of the biconnected components of the graph.

Articulation Point



A vertex V in a connected graph G is an articulation point (cut point) if the deletion of vertex V along with all edges incident to V disconnects the graph into two or more non-empty components.

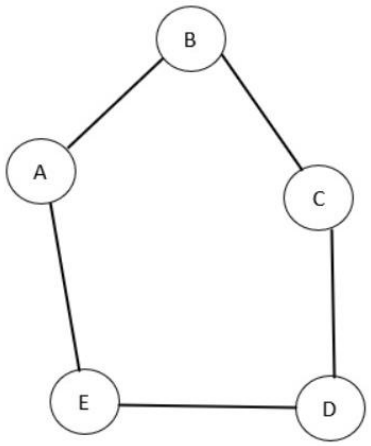
Example: Fig shows a connected graph

Here we can say vertex 2 is an articulation point, because if we remove the vertex 2 along with the incident edges we can observe that the graph splits into two non-empty component.

After deleting vertex 2 and the incident edges of vertex 2, the given graph is divided into two non-empty components as in the figure above => vertex 2 is an articulation point

Biconnected Graph - We can call a graph G a biconnected graph if it contains zero articulation point.

Example: The above graph contains no articulation point => The graph is Biconnected Graph.



Identify Articulation Point In A Graph

An articulation point (or cut vertex) in a graph is a vertex that, when removed, increases the number of connected components in the graph. In other words, an articulation point is a vertex that "separates" the graph into two or more separate parts, and if it is removed, those parts will become disconnected. One of the simplest algorithms for identifying articulation points in a given graph is to perform a depth-first search (DFS) traversal of the graph and keep track of the following information for each vertex:

1. The DFS number of the vertex: This is the order in which the vertex was visited during the DFS traversal.
2. The low-link value of the vertex: This is the minimum DFS number of any vertex that can be reached from the current vertex by following zero or more DFS tree edges and then at least one back edge.

To identify the articulation points in the graph, we can use the following steps:

1. Start by setting the DFS number and low-link value of each vertex to be the same.
2. For each vertex, do the following:
    - 2.1. Consider all of the vertices that are reachable from the current vertex by a single edge.
    - 2.2. For each of these vertices, compute the low-link value using the following rule:
        - 2.2.1. If the low-link value of the current vertex is greater than the low-link value of the reachable vertex, set the low-link value of the current vertex to be the same as the low-link value of the reachable vertex.
        - 2.2.2. If the low-link value of the current vertex is equal to its DFS number, then it is an articulation point.
3. Repeat this process for all vertices in the graph.

Time Complexity - This algorithm has a time complexity of O(V+E), where V is the number of vertices and E is the number of edges in the graph. This is because the algorithm performs a depth-first search (DFS) traversal of the graph, which has a time complexity of O(V+E). The space complexity of the algorithm is also O(V+E), as it stores the DFS number and low-link value for each vertex in the graph.

Why These Are Important?

1. An articulation point (or cut vertex) in a graph is a vertex that, when removed along with all the edges incident to it, increases the number of connected components in the graph. In other words, an articulation point is a vertex that plays a key role in keeping the graph connected.
2. Biconnected components, on the other hand, are subgraphs of a graph that are themselves connected and do not have any articulation points.

3. These concepts are important because they can be used to analyze the connectivity and structural properties of a graph. For example, an articulation point can be used to identify the most critical vertices in a network, such as bridges in a road network or routers in a computer network. Biconnected components can be used to identify the most robust parts of a graph, as they remain connected even if one of their vertices is removed.
4. In addition, algorithms for finding articulation points and biconnected components are important building blocks for other graph algorithms, such as finding bridges and cycles in a graph, and for analyzing the structure of graphs in general.
5. By the end of this article at OpenGenus, you must have complete knowledge on biconnected components, lets test your knowledge with a question!

Shortest path algorithms - Dijkstra's Algorithms

Dijkstra's Algorithm. **Dijkstra's Algorithm** is a Graph algorithm **that finds the shortest path** from a source vertex **to all other vertices** in the Graph (single source shortest path). It is a type of **Greedy Algorithm** that only works on Weighted Graphs having positive weights. The time complexity of Dijkstra's Algorithm is **$O(V^2)$** with the help of the adjacency matrix representation of the graph. This time complexity can be reduced to **O((V + E) log V)** with the help of an adjacency list representation of the graph, where **V** is the number of vertices and **E** is the number of edges in the graph.

History of Dijkstra's Algorithm - Dijkstra's Algorithm was designed and published by **Dr. Edsger W. Dijkstra**, a Dutch Computer Scientist, Software Engineer, Programmer, Science Essayist, and Systems Scientist. He developed the shortest path algorithm and later executed it for ARMAC, He re-discovered the algorithm called Prim's minimal spanning tree algorithm and published it in the year 1959.

Fundamentals of Dijkstra's Algorithm - The following are the basic concepts of Dijkstra's Algorithm:

1. Dijkstra's Algorithm **begins at the** node we select (the **source node**), and it examines the graph to find the **shortest path between that node** and all the other nodes in the graph.
2. The Algorithm **keeps records** of the presently **acknowledged shortest distance** from each node to the source node, and it **updates** these values **if it finds any shorter path**.
3. Once the Algorithm has **retrieved the shortest path** between the source and another node, that node is **marked as 'visited'** and included in the path.
4. The procedure **continues until all the nodes** in the graph have been included in the path. In this manner, we have a path connecting the source node to all other nodes, following the shortest possible path to reach each node.

Understanding the Working of Dijkstra's Algorithm

A **graph** and **source vertex** are requirements for Dijkstra's Algorithm. This Algorithm is established on Greedy Approach and thus finds the locally optimal choice (local minima in this case) at each step of the Algorithm.

Each Vertex in this Algorithm will have two properties defined for it:
1. Visited Property

2. Path Property

Visited Property:

1. The 'visited' property **signifies whether or not the node** has been visited.
2. We are using this property so that we **do not revisit any node.**
3. A node is **marked visited** only when the **shortest path has been found.**

Path Property:

1. The 'path' property **stores the value** of the current **minimum path** to the node.
2. The current minimum path implies the **shortest way** we have reached this node till now.
3. This property is **revised** when any neighbour of the **node is visited.**
4. This property is significant because it **will store the final answer** for each node.

Initially, we **mark all** the vertices, or **nodes, unvisited** as they have yet to be visited. The path to all the nodes is also **set to infinity** apart from the source node. Moreover, the path to the **source node** is **set to zero (0)**. We then select the source node and mark it as visited. After that, we access all the neighbouring nodes of the source node and **perform relaxation** on every node. Relaxation is the **process of lowering the cost** of reaching a node with the help of another node. In the process of relaxation, the **path of each node is revised** to the minimum value amongst the **node's current path,** the sum of the path to the previous node, and the path from the previous node to the current node.

Let us suppose that **p[n]** is the **value of the current path** for node n, **p[m]** is the **value of the path** up to the **previously visited node m**, and w is the weight of the edge between the current node and previously visited one (edge weight between n and m).In the mathematical sense, relaxation can be exemplified as:

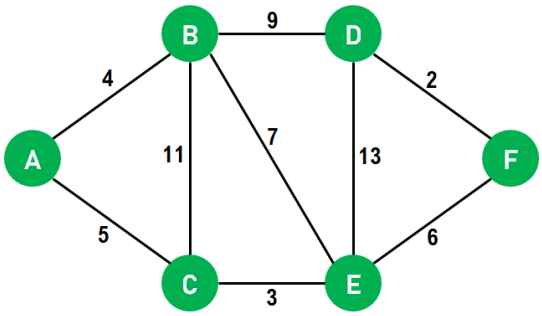
p[n] = minimum(p[n], p[m] + w)

We then mark an unvisited node with the least path as visited in every subsequent step and update its neighbour's paths. We r**epeat this procedure** until all the nodes in the graph are marked visited. Whenever **we add a node** to the **visited set**, the **path** to all its neighbouring nodes **also changes accordingly.** If any node is left **unreachable** (disconnected component), its path **remains** '**infinity**'. In case the source itself is a separate component, then the path to all other nodes remains 'infinity'.

Dijkstra's Algorithm

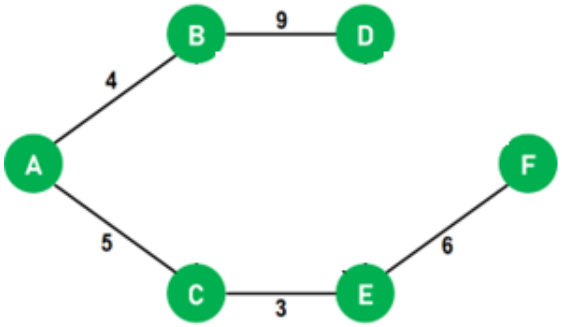
1. Set all the vertices to infinity, excluding the source vertex.
2. Push the source in the form (distance, vertex) and put it in the min-priority queue.
3. From the priority, queue pop out the minimum distant vertex from the source vertex.
4. Update the distance after popping out the minimum distant vertex and calculate the vertex distance using (vertex distance + weight < following vertex distance).
5. If you find that the visited vertex is popped, move ahead without using it.
6. Apply the steps until the priority queue is found to be empty.

Example - Implementation of the algorithm with the help of an example:



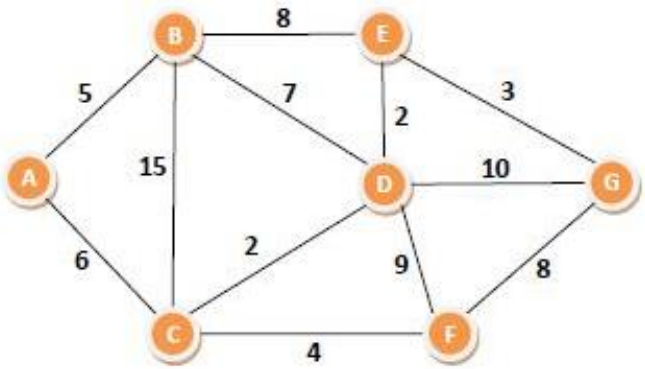
1. We will use the above graph as the input, with node **A** as the source.
2. First, we will mark all the nodes as unvisited.
3. We will set the path to **0** at node **A** and **INFINITY** for all the other nodes.
4. We will now mark source node **A** as visited and access its neighbouring nodes. We have only accessed the neighbouring nodes, not visited them.
5. We will now update the path to node **B** by **4** with the help of relaxation because the path to node **A** is **0** and the path from node **A** to **B** is **4**, and the **minimum((0 + 4), INFINITY)** is **4**.
6. We will also update the path to node **C** by **5** with the help of relaxation because the path to node **A** is **0** and the path from node **A** to **C** is **5**, and the **minimum((0 + 5), INFINITY)** is **5**. Both the neighbours of node **A** are now relaxed; therefore, we can move ahead.
7. We will now select the next unvisited node with the **least path and visit** it. Hence, we will visit node **B** and perform relaxation on its unvisited neighbours. After performing relaxation, the path to node **C** will remain **5**, whereas the path to node **E** will become **11**, and the path to node **D** will become **13**.
8. We will now visit node **E** and perform relaxation on its neighbouring nodes **B, D**, and **F**. Since only node **F** is unvisited, it will be relaxed. Thus, the path to node **B** will remain as it is, i.e., **4**, the path to node **D** will also remain **13**, and the path to node **F** will become **14 (8 + 6)**.
9. Now we will visit node **D**, and only node **F** will be relaxed. However, the path to node **F** will remain unchanged, i.e., **14**.
10. Since only node **F** is remaining, we will visit it but not perform any relaxation as all its neighboring nodes are already visited.
11. Once all the nodes of the graphs are visited, the program will end.

Hence, the final paths we concluded are:



1. A = 0
2. B = 4 (A -> B)
3. C = 5 (A -> C)
4. D = 4 + 9 = 13 (A -> B -> D)
5. E = 5 + 3 = 8 (A -> C -> E)
6. F = 5 + 3 + 6 = 14 (A -> C -> E -> F)

The working of the algorithm can be best understood using an example. Consider the following graph having nodes marked from A to G, connected by weighted edges as follows −

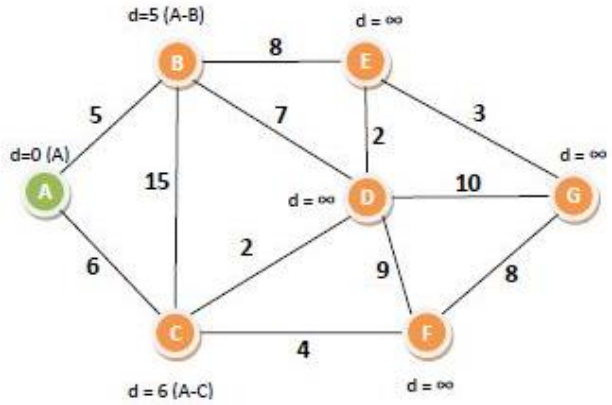


The initializations will be as follows −

dist[7]={0,∞,∞,∞,∞,∞,∞}
Q={A,B,C,D,E,F,G}
S*S*= ∅

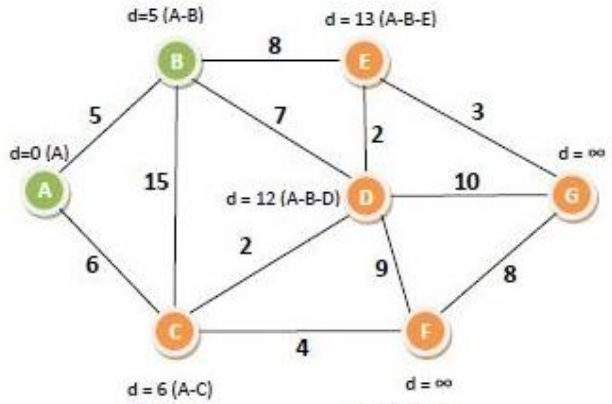
Pass 1 – We choose node **A** from **Q** since it has the lowest **dist[]** value of **0** and put it in S. The neighbouring nodes of A are B and C. We update dist[] values corresponding to B and C according to the algorithm. So the values of the data structures become -

dist[7]={0,5,6,∞,∞,∞,∞}
Q={B,C,D,E,F,G}
S={A}



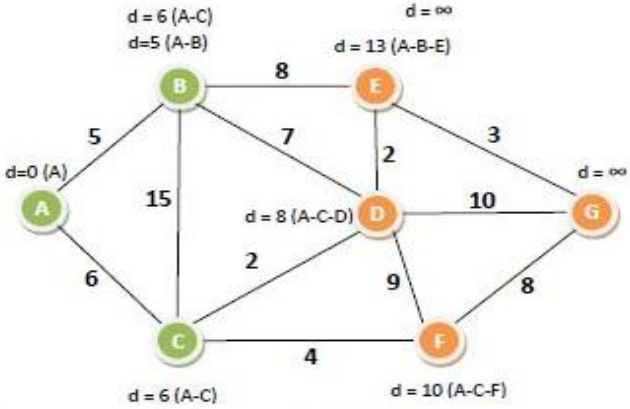
Pass 2 – We choose node **B** from **Q** since it has the lowest **dist[]** value of **5** and put it in **S.** The neighbouring nodes of B are **C, D** and **E**. We update dist[] values corresponding to **C, D** and E according to the algorithm. So the values of the data structures become –

dist[7]={0,5,6,12,13,∞,∞}
Q={C,D,E,F,G}
S={A,B}



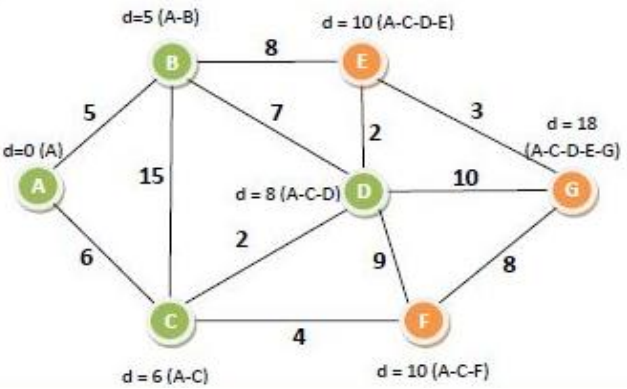
Pass 3 – We choose node **C** from **Q** since it has the lowest dist[] value of 6 and put it in S. The neighbouring nodes of C are D and F. We update dist[] values corresponding to D and F. So the values of the data structures become –

dist[7]={0,5,6,8,13,10,∞}
Q={D,E,F,G}
S={A,B,C}

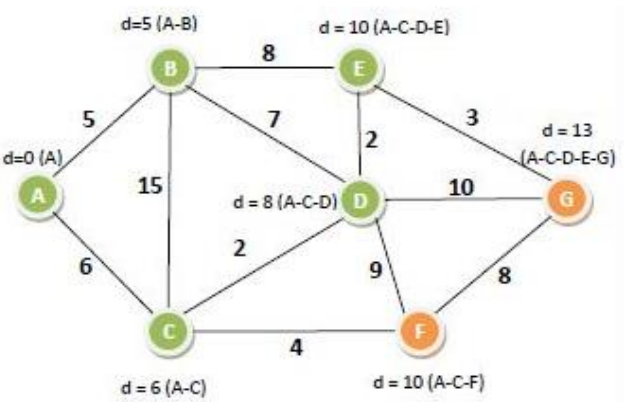


Pass 4 – We choose node **D** from **Q** since it has the lowest **dist[]** value of 8 and put it in S. The neighbouring nodes of D are E, F and G. We update **dist[]** values corresponding to **E, F** and **G**. So the values of the data structures become –

dist[7]={0,5,6,8,10,10,18}
Q={E,F,G}
S={A,B,C,D}

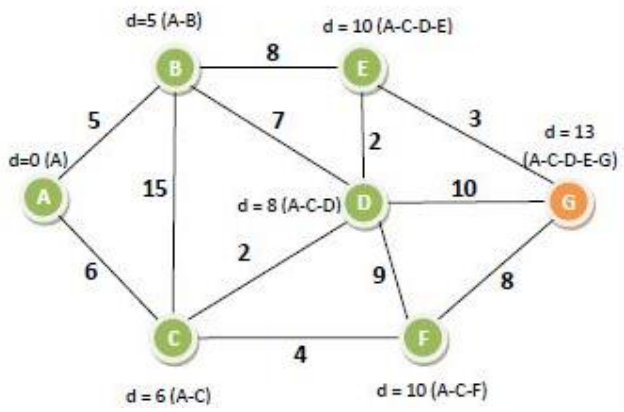


Pass 5 – We can choose either node E or node **F** from **Q** since both of them have the lowest **dist[]** value of **10**. We select any one of them, say **E**, and put it in **S**. The neighbouring nodes of **D** is **G**. We update **dist[]** values corresponding to G. So the values of the data structures become –



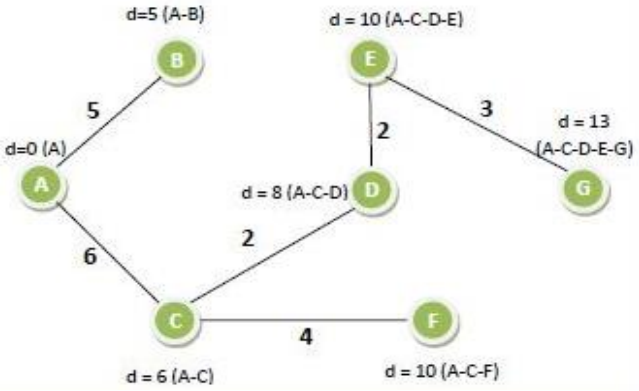
dist[7]={0,5,6,8,10,10,13}
Q={F,G}
S={A,B,C,D,E}

Pass 6 – We choose node **F** from **Q** since it has the lowest **dist[]** value of **10** and put it in **S**. The neighbouring nodes of **F** is **G**. The **dist[]** value corresponding to G is less than that through **F**. So it remains same. The values of the data structures become –



dist[7]={0,5,6,8,10,10,13}
Q={G}
S={A,B,C,D,E,F}

Pass 7 – There is just one node in **Q**. We remove it from **Q** put it in S. The dist[] array needs no change. Now, **Q** becomes empty, **S** contains all the nodes and so we come to the end of the algorithm. We eliminate all the edges or routes that are not in the path of any route. So the shortest path tree from source node A to all other nodes are as follows –



dist[7]={0,5,6,8,10,10,13}
Q=Ø
S={A,B,C,D,E,F,G}

Time and Space Complexity

1. The Time Complexity of Dijkstra's Algorithm is **O(E log V)**, where E is the number of edges and V is the number of vertices.
2. The Space Complexity of Dijkstra's Algorithm is O(V), where V is the number of vertices.

Advantages

1. One primary advantage of using Dijkstra's Algorithm is that it has an **almost linear time and space complexity.**
2. We can use this algorithm to **calculate the shortest path from a single vertex** to all other vertices and a single source vertex to a single destination vertex by stopping the algorithm once we get the shortest distance for the destination vertex.
3. This algorithm **only works** for **directed weighted graphs**, and all the edges of this graph should be non-negative.

Disadvantages

1. Dijkstra's Algorithm performs a concealed exploration that utilizes a **lot of time** during the process.
2. This algorithm is impotent to **handle negative edges.**

3. Since this algorithm heads to the **acyclic graph**, it **cannot calculate the exact** shortest path.
4. It also requires **maintenance to keep a record** of vertices that have been visited.

Some Applications of Dijkstra's Algorithm

1. Digital Mapping Services in Google Maps
2. Social Networking Applications
3. Telephone Network
4. Flight Program
5. IP routing to find Open Shortest Path First
6. Robotic Path
7. Designate the File Server.

Shortest path algorithms - Floyd's Algorithms

The Floyd-Warshall algorithm is a graph algorithm that is deployed to **find the shortest path** between all the vertices present in **a weighted graph.** This algorithm is different from other shortest path algorithms; to describe it simply, this algorithm uses **each vertex** in the graph **as a pivot** to **check if** it provides the **shortest way** to travel from **one point to another.**

Floyd-Warshall algorithm works on **both directed and undirected weighted graphs** unless these graphs **do not contain** any **negative cycles** in them. By negative cycles, it is meant that the **sum of all the edges** in the graph must not lead to a negative number. Since, the algorithm deals with **overlapping sub-problems** – the path found by the vertices acting as pivot are stored for solving the next steps – it uses the **dynamic programming approach.**

Floyd-Warshall algorithm is one of the methods in All-pairs shortest path algorithms and it is solved **using the Adjacency Matrix** representation of graphs. Consider a graph, **G = {V, E}** where **V** is the set of all vertices present in the graph and E is the set of all the edges in the graph. The graph, **G**, is represented in the form of an adjacency matrix, **A**, that contains all the weights of every edge connecting two vertices.

Algorithm - This algorithm follows the dynamic programming approach to find the shortest paths.

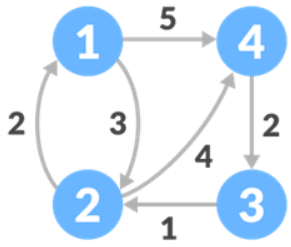
Step 1 – Construct an adjacency matrix **A** with all the costs of edges present in the graph. If there is no path between two vertices, mark the value as ∞.

Step 2 – Derive another adjacency matrix $\mathbf{A_1}$ from **A** keeping the first row and first column of the original adjacency matrix intact in $\mathbf{A_1}$. And for the remaining values, say $\mathbf{A_1[i,j]}$, if **A[i,j]>A[i,k]+A[k,j]** then replace $\mathbf{A_1[i,j]}$ with **A[i,k]+A[k,j]**. Otherwise, do not change the values. Here, in this step, **k = 1** (first vertex acting as pivot).

Step 3 – Repeat **Step 2** for all the vertices in the graph by changing the **k** value for every pivot vertex until the final matrix is achieved.

Step 4 – The final adjacency matrix obtained is the final solution with all the shortest paths.

How Floyd-Warshall Algorithm Works?



Example - Let the given graph be: Initial graph

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.

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

   Each cell A[i][j] is filled with the distance from the i^{th} vertex to the j^{th} vertex. If there is no path from i^{th} vertex to j^{th} vertex, the cell is left as infinity. Fill each cell with the distance between ith and jth vertex

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].

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

   In this step, k is vertex 1. Calculate the distance from the source vertex to destination vertex through this vertex k, For example: For $A^1[2, 4]$, 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, $A^0[2, 4]$ is filled with 4.

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

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

   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

4. Similarly, A^3 and A^4 is also created. Calculate the distance from the source vertex to destination vertex through this vertex 3

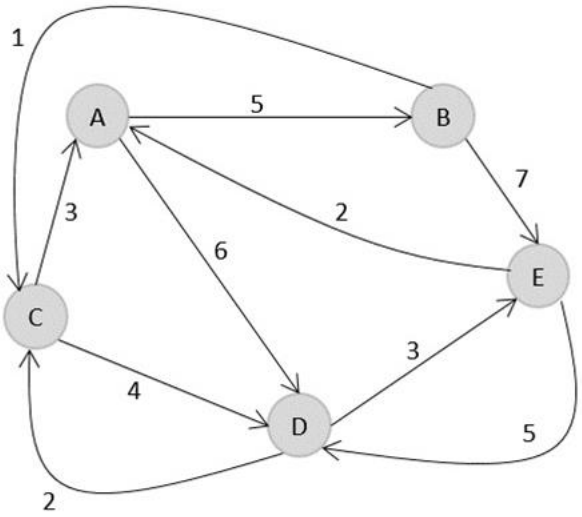
   $$3 = A^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & & \infty & \\ & 0 & \infty & \\ 3 & 1 & 0 & 5 \\ & & 2 & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array}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

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

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

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

Example - Consider the following directed weighted graph **G = {V, E}**. Find the shortest paths between all the vertices of the graphs using the Floyd-Warshall algorithm.



Solution

Step 1 - Construct an adjacency matrix **A** with all the distances as values.

$$A = \begin{bmatrix} 0 & 5 & \infty & 6 & \infty \\ \infty & 0 & 1 & \infty & 7 \\ 3 & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 2 & \infty & \infty & 5 & 0 \end{bmatrix}$$

Step 2 - Considering the above adjacency matrix as the input, derive another matrix A_0 by keeping only first rows and columns intact. Take **k = 1**, and replace all the other values by **A[i,k]+A[k,j]**.

$$A = \begin{bmatrix} 0 & 5 & \infty & 6 & \infty \\ \infty & & & & \\ 3 & & & & \\ \infty & & & & \\ 2 & & & & \end{bmatrix} \qquad A_1 = \begin{bmatrix} 0 & 5 & \infty & 6 & \infty \\ \infty & 0 & 1 & \infty & 7 \\ 3 & 8 & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 2 & 7 & \infty & 5 & 0 \end{bmatrix}$$

Step 3 - Considering the above adjacency matrix as the input, derive another matrix A_0 by keeping only first rows and columns intact. Take **k = 1**, and replace all the other values by **A[i,k]+A[k,j]**.

$$A_2 = \begin{bmatrix} & 5 & & & \\ \infty & 0 & 1 & \infty & 7 \\ & 8 & & & \\ & \infty & & & \\ & 7 & & & \end{bmatrix} \qquad A_2 = \begin{bmatrix} 0 & 5 & 6 & 6 & 12 \\ \infty & 0 & 1 & \infty & 7 \\ 3 & 8 & 0 & 4 & 15 \\ \infty & \infty & 2 & 0 & 3 \\ 2 & 7 & 8 & 5 & 0 \end{bmatrix}$$

Step 4 - Considering the above adjacency matrix as the input, derive another matrix A_0 by keeping only first rows and columns intact. Take **k = 1**, and replace all the other values by **A[i,k]+A[k,j]**.

$$A_3 = \begin{bmatrix} & & 6 & & \\ & & 1 & & \\ 3 & 8 & 0 & 4 & 15 \\ & & 2 & & \\ & & 8 & & \end{bmatrix} \qquad A_3 = \begin{bmatrix} 0 & 5 & 6 & 6 & 12 \\ 4 & 0 & 1 & 5 & 7 \\ 3 & 8 & 0 & 4 & 15 \\ 5 & 10 & 2 & 0 & 3 \\ 2 & 7 & 8 & 5 & 0 \end{bmatrix}$$

Step 5 - Considering the above adjacency matrix as the input, derive another matrix A_0 by keeping only first rows and columns intact. Take **k = 1**, and replace all the other values by **A[i,k]+A[k,j]**.

$$A_4 = \begin{matrix} & & & & 6 \\ & & & & 5 \\ & & & & 4 \\ 5 & 10 & 2 & 0 & 3 \\ & & & & 5 \end{matrix} \qquad A_4 = \begin{matrix} 0 & 5 & 6 & 6 & 9 \\ 4 & 0 & 1 & 5 & 7 \\ 3 & 8 & 0 & 4 & 7 \\ 5 & 10 & 2 & 0 & 3 \\ 2 & 7 & 7 & 5 & 0 \end{matrix}$$

Step 6 - Considering the above adjacency matrix as the input, derive another matrix **A_0** by keeping only first rows and columns intact. Take **k = 1**, and replace all the other values by **A[i,k]+A[k,j]**.

$$A_5 = \begin{matrix} & & & & 9 \\ & & & & 7 \\ & & & & 7 \\ & & & & 3 \\ 2 & 7 & 7 & 5 & 0 \end{matrix} \qquad A_5 = \begin{matrix} 0 & 5 & 6 & 6 & 9 \\ 4 & 0 & 1 & 5 & 7 \\ 3 & 8 & 0 & 4 & 7 \\ 5 & 10 & 2 & 0 & 3 \\ 2 & 7 & 7 & 5 & 0 \end{matrix}$$

Minimum spanning tree - Prim's Algorithms

Before starting the main topic, we should discuss the basic and important terms such as spanning tree and minimum spanning tree.

Spanning tree - A spanning tree is the **subgraph** of an **undirected** connected graph.

Minimum Spanning tree - 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.

Prim's algorithm is also a **Greedy algorithm.** This algorithm always **starts with** a **single node** and **moves** through **several adjacent nodes**, in order to explore all of the connected edges along the way. The **algorithm starts** with an **empty spanning** tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, and the other set contains the vertices not yet included. At **every step,** it **considers all the edges** that connect the two sets and **picks the minimum** weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

A **group of edges** that connects two sets of vertices in a graph is **called cut in graph theory**. So, at every step of **Prim's algorithm, find a cut**, pick the minimum weight edge from the cut, and include this vertex in MST Set (the set that contains already included vertices).

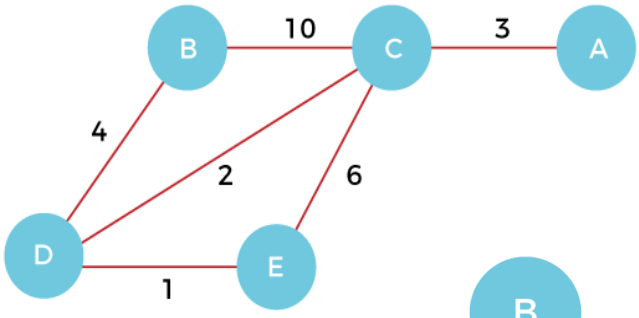
How does Prim's Algorithm Work? - The working of Prim's algorithm can be described by using the following steps:

Step 1: Determine an arbitrary vertex as the **starting vertex** of the MST.
Step 2: Follow **steps 3 to 5** till there are vertices that are not included in the MST (known as fringe vertex).
Step 3: Find edges connecting any tree vertex with the fringe vertices.
Step 4: Find the minimum among these edges.
Step 5: Add the chosen edge to the MST if it does not form any cycle.
Step 6: Return the MST and exit

The applications of prim's algorithm are –

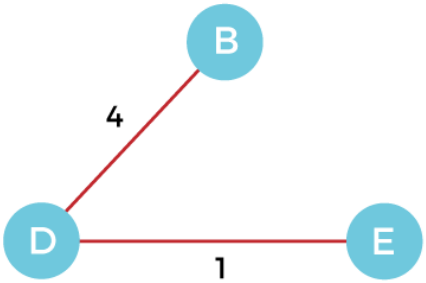
- Prim's algorithm can be used in network designing.
- It can be used to make network cycles.
- It can also be used to lay down electrical wiring cables.

Example - Illustration of Prim's Algorithm: Suppose, a weighted graph is –

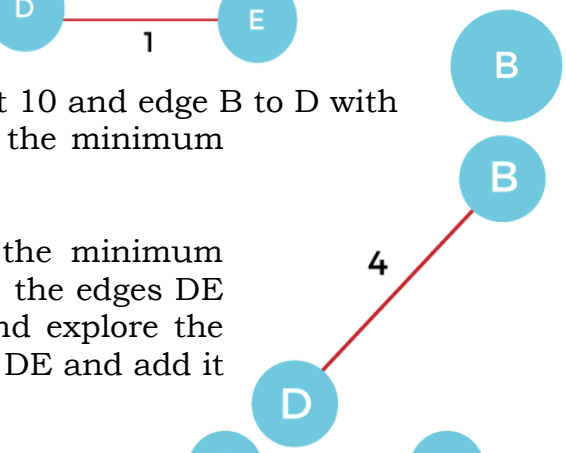


Step 1 - First, we have to choose a vertex from the above graph. Let's choose B.

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

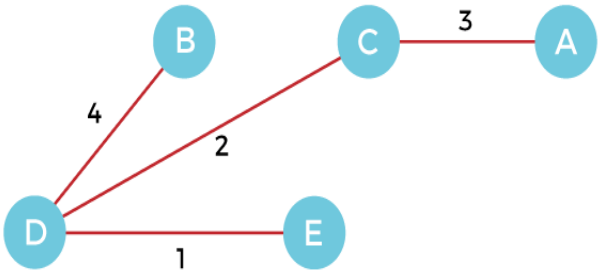


Step 3 - Now, again, choose the edge with the minimum weight among all the other edges. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C, i.e., E and A. So, select the edge DE and add it to the MST.



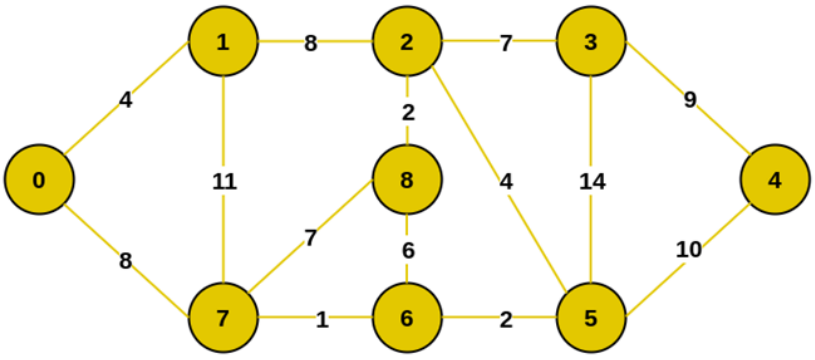
Step 4 - Now, select the edge CD, and add it to the MST.

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

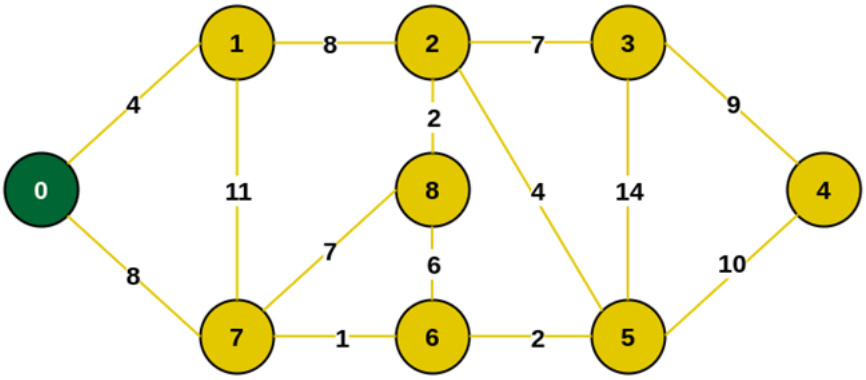


So, the graph produced in step 5 is the minimum spanning tree of the given graph. The cost of the MST is given below - **Cost of MST = 4 + 2 + 1 + 3 = 10 units.**

Example 2. Consider the following graph as an example for which we need to find the Minimum Spanning Tree (MST). Example of a graph

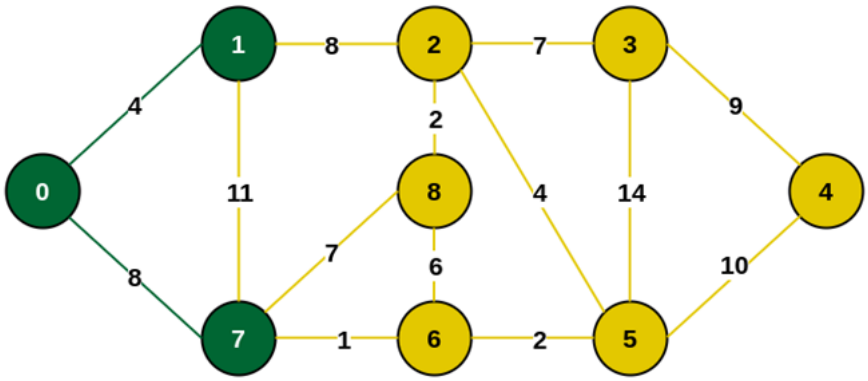


Step 1: Firstly, we select an arbitrary vertex that acts as the starting vertex of the Minimum Spanning Tree. Here we have selected vertex **0** as the starting vertex.



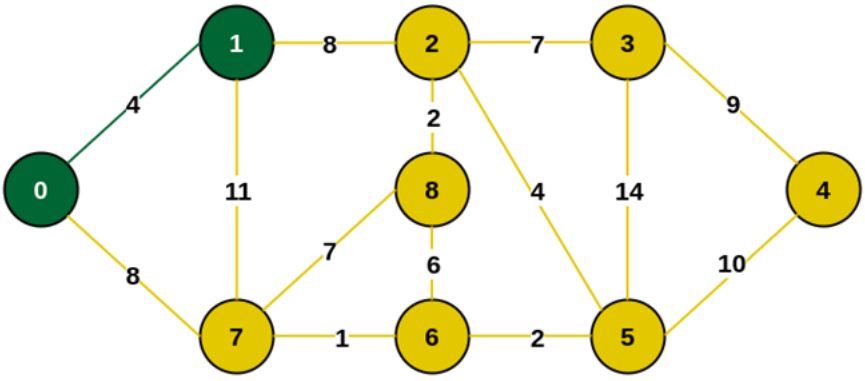
Selected node - 0 is selected as starting vertex

Step 2: All the edges connecting the incomplete MST and other vertices are the edges {0, 1} and {0, 7}. Between these two the edge with minimum weight is {0, 1}. So include the edge and vertex 1 in the MST.



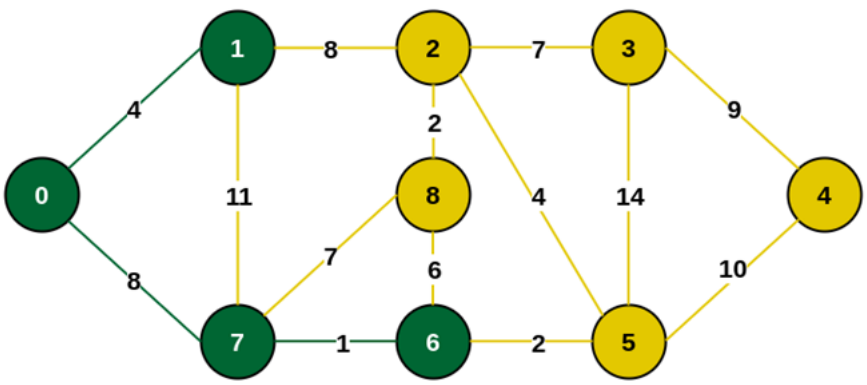
Selected node – 0, 1 is added to the MST

Step 3: The edges connecting the incomplete MST to other vertices are {0, 7}, {1, 7} and {1, 2}. Among these edges the minimum weight is 8 which is of the edges {0, 7} and {1, 2}. Let us here include the edge {0, 7} and the vertex 7 in the MST. [We could have also included edge {1, 2} and vertex 2 in the MST].

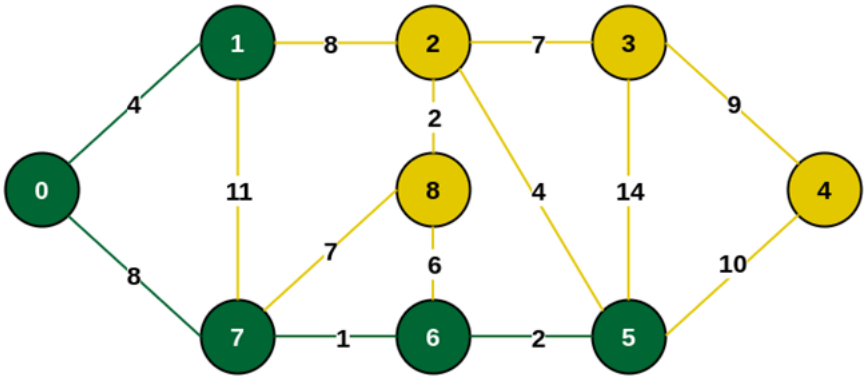


Selected node – 0, 1, 7 is added in the MST

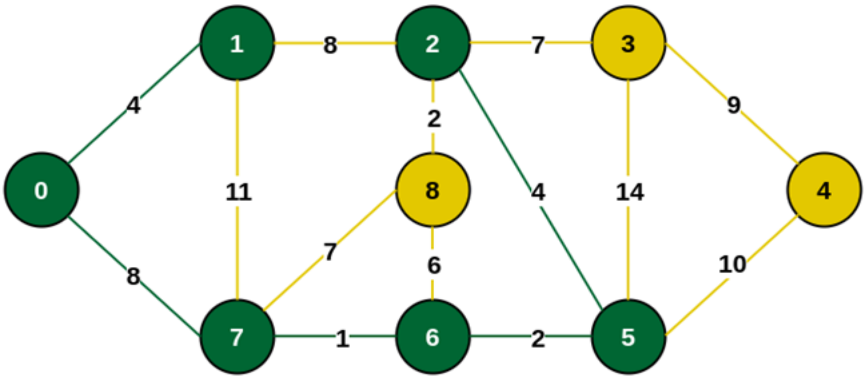
Step 4: The edges that connect the incomplete MST with the fringe vertices are {1, 2}, {7, 6} and {7, 8}. Add the edge {7, 6} and the vertex 6 in the MST as it has the least weight (i.e., 1).



Selected node – 0, 1, 7, 6 is added in the MST

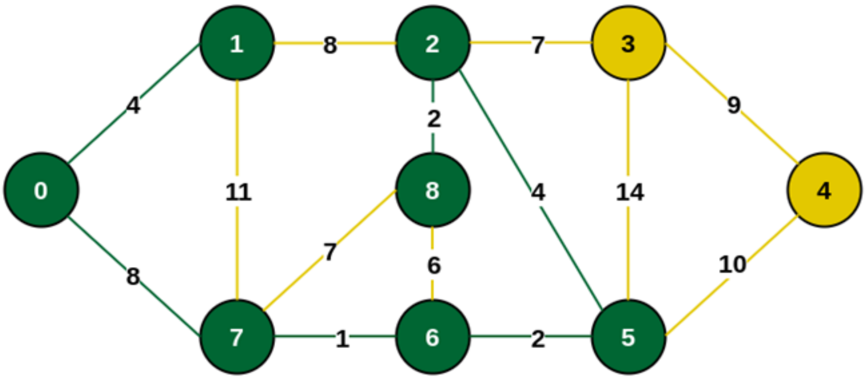


Step 5: The connecting edges now are {7, 8}, {1, 2}, {6, 8} and {6, 5}. Include edge {6, 5} and vertex 5 in the MST as the edge has the minimum weight (i.e., 2) among them.



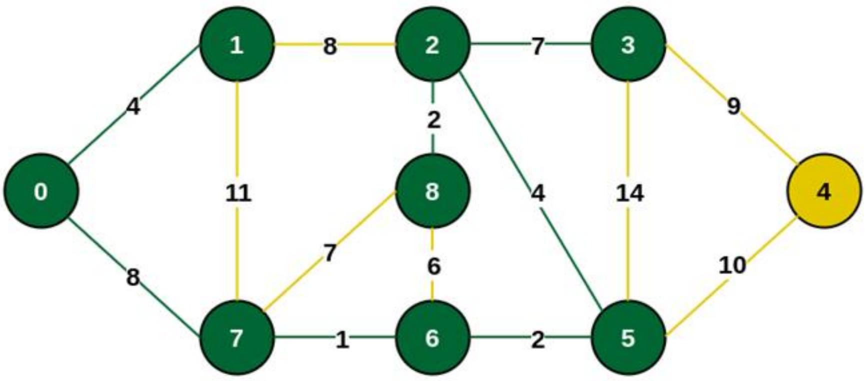
Selected node – 0, 1, 7, 6, 5 Include vertex 5 in the MST

Step 6: Among the current connecting edges, the edge {5, 2} has the minimum weight. So include that edge and the vertex 2 in the MST.



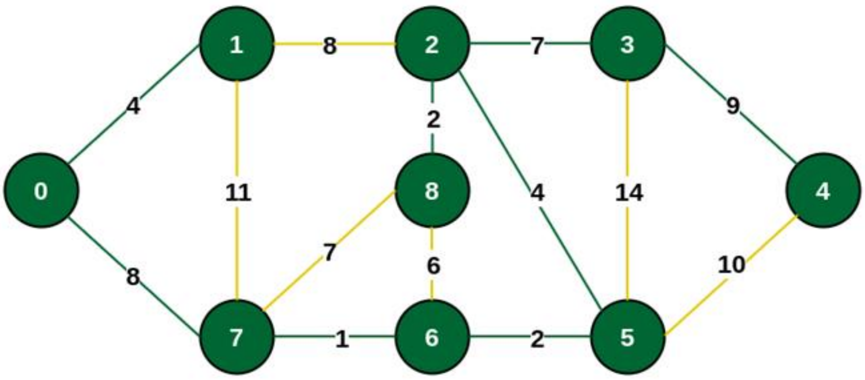
Selected node – 0, 1, 7, 6, 5 , 2 Include vertex 2 in the MST

Step 7: The connecting edges between the incomplete MST and the other edges are {2, 8}, {2, 3}, {5, 3} and {5, 4}. The edge with minimum weight is edge {2, 8} which has weight 2. So include this edge and the vertex 8 in the MST.



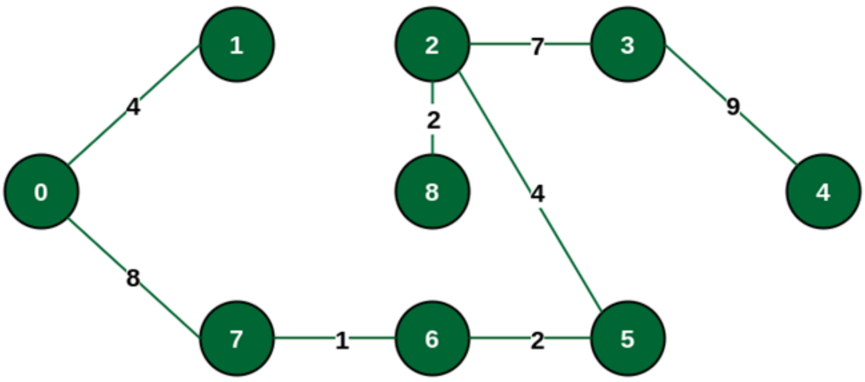
Selected node – 0, 1, 7, 6, 5 , 2 , 8 Add vertex 8 in the MST

Step 8: See here that the edges {7, 8} and {2, 3} both have same weight which are minimum. But 7 is already part of MST. So we will consider the edge {2, 3} and include that edge and vertex 3 in the MST.



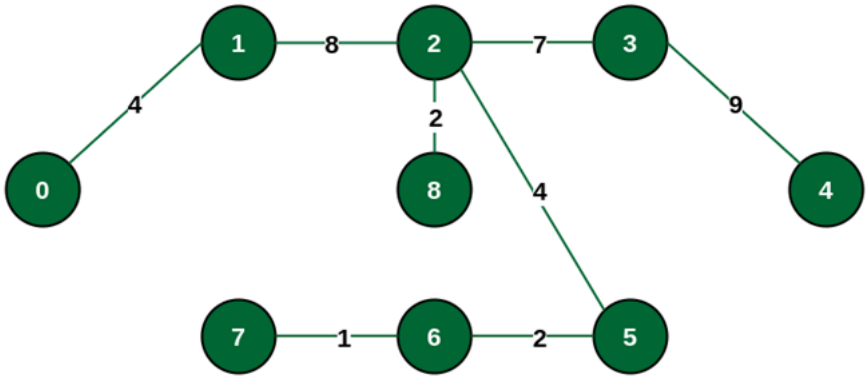
Selected node – 0, 1, 7, 6, 5 , 2 , 8 , 3 Include vertex 3 in MST

Step 9: Only the vertex 4 remains to be included. The



minimum weighted edge from the incomplete MST to 4 is {3, 4}.

Selected node – 0, 1, 7, 6, 5 , 2 , 8 , 3 , 4 Include vertex 4 in the MST



The final structure of the MST is as follows and the weight of the edges of the MST is (4 + 8 + 1 + 2 + 4 + 2 + 7 + 9) = **37**.

The structure of the MST formed using the above method

Note: If we had selected the edge {1, 2} in the third step then the MST would look like the following.

Prim's Algorithm

- Create a set **mstSet** that keeps track of vertices already included in MST.
- Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign the key value as 0 for the first vertex so that it is picked first.
- While **mstSet** doesn't include all vertices
- Pick a vertex **u** that is not there in **mstSet** and has a minimum key value.
- Include **u** in the **mstSet**.
- Update the key value of all adjacent vertices of **u**. To update the key values, iterate through all adjacent vertices.
- For every adjacent vertex **v**, if the weight of edge **u-v** is less than the previous key value of **v**, update the key value as the weight of **u-v**.

The idea of using key values is to **pick the minimum weight edge** from the cut. The key values are used only for vertices that are not yet included in MST, the key value for these vertices indicates the minimum weight edges connecting them to the set of vertices included in MST.

Time Complexity: $O(V^2)$, If the input graph is represented using an adjacency list, then the time complexity of Prim's algorithm can be **reduced to O(E * logV)** with the **help of a binary heap**. In this implementation, we are always considering the spanning tree to start from the root of the graph.

Auxiliary Space: O(V)

Minimum spanning tree - Kruskal's Algorithms

Minimum Spanning tree - 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.

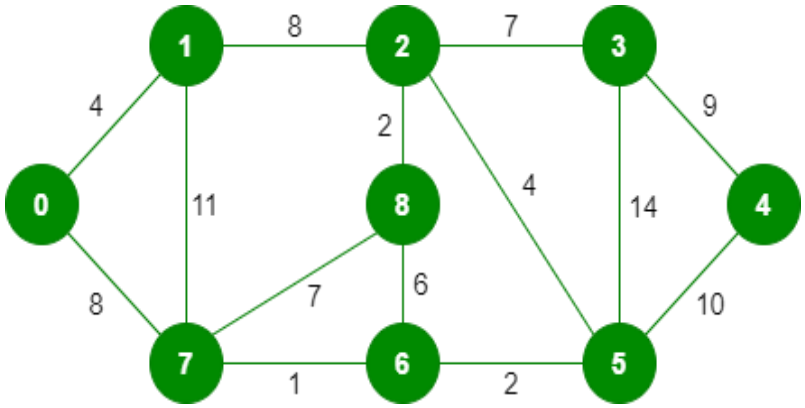
A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected, undirected graph is a spanning tree with a **weight less than or equal to** the weight of every **other spanning tree. Kruskal's algorithm** is used to find the MST of a given **weighted graph**. In Kruskal's algorithm, **sort all edges** of the given

graph in **increasing order**. Then it keeps on adding new edges and nodes in the MST if the **newly added edge does not form a cycle**. It picks the minimum weighted edge at first at the maximum weighted edge at last. Thus we can say that it makes a locally optimal choice in each step in order to find the optimal solution. Hence this is a **Greedy Algorithm**.

How to find MST using Kruskal's algorithm? - Below are the steps for finding MST using Kruskal's algorithm:

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If the cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.

Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far. Let us understand it with an example:



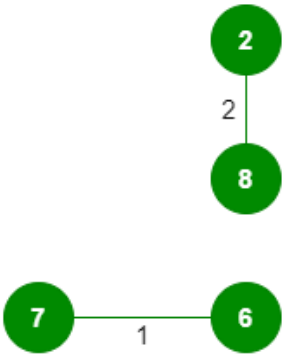
Weight	Source	Destination
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

Example: Below is the illustration of the above approach - Input Graph: The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

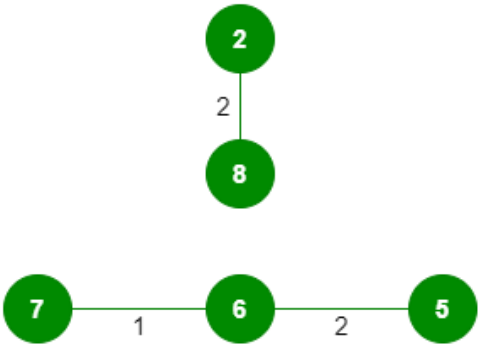
After sorting: we create table Now pick all edges one by one from the sorted list of edges

Step 1: Pick edge 7-6. No cycle is formed, include it. Add edge 7-6 in the MST

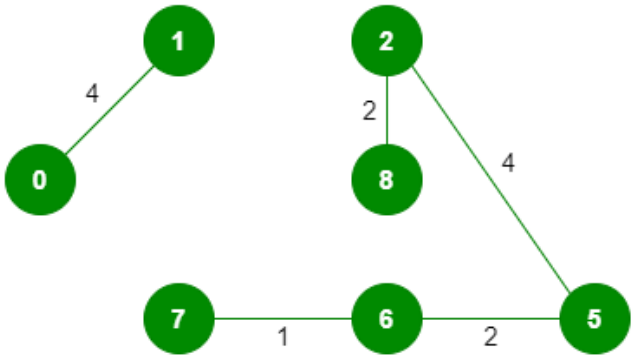
Step 2: Pick edge 8-2. No cycle is formed, include it. Add edge 8-2 in the MST

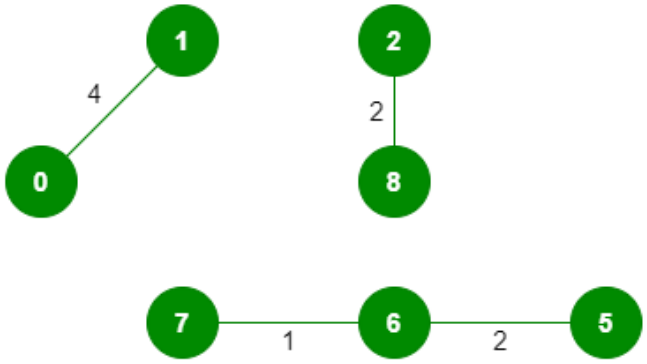


Step 3: Pick edge 6-5. No cycle is formed, include it. Add edge 6-5 in the MST



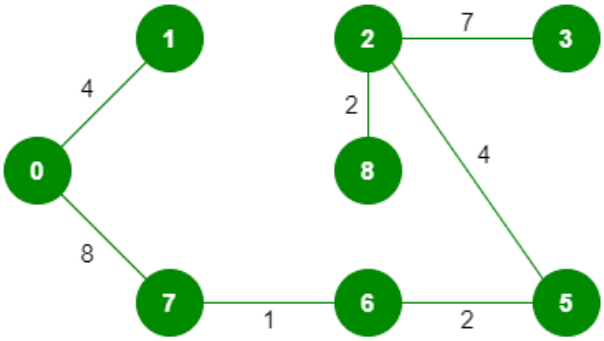
Step 4: Pick edge 0-1. No cycle is formed, include it. Add edge 0-1 in the MST

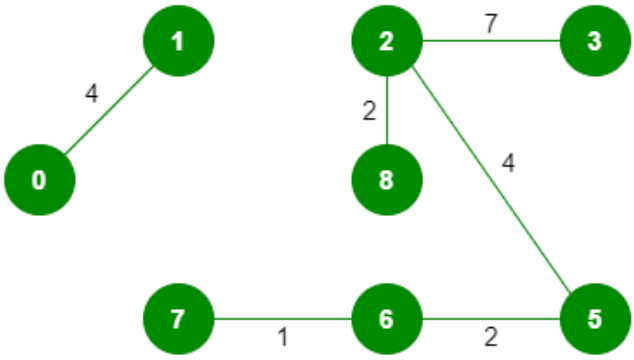




Step 5: Pick edge 2-5. No cycle is formed, include it. Add edge 2-5 in the MST

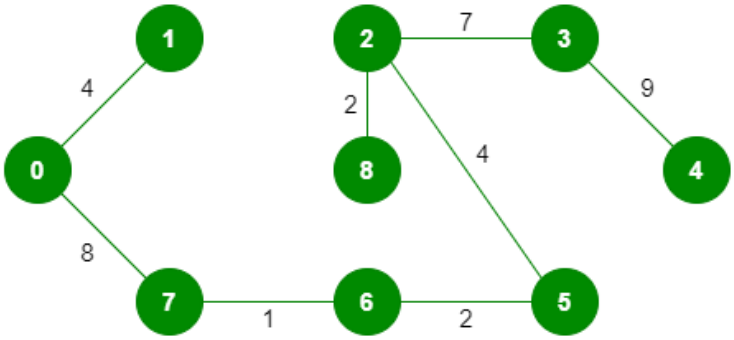
Step 6: Pick edge 8-6. Since including this edge results in the cycle, discard it. Pick edge 2-3: No cycle is formed, include it. Add edge 2-3 in the MST



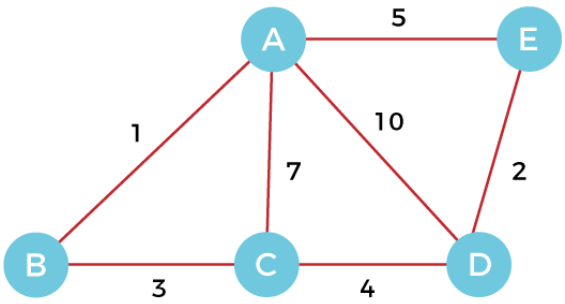


Step 7: Pick edge 7-8. Since including this edge results in the cycle, discard it. Pick edge 0-7. No cycle is formed, include it. Add edge 0-7 in MST

Step 8: Pick edge 1-2. Since including this edge results in the cycle, discard it. Pick edge 3-4. No cycle is formed, include it. Add edge 3-4 in the MST



Example 2 - Suppose a weighted graph is -

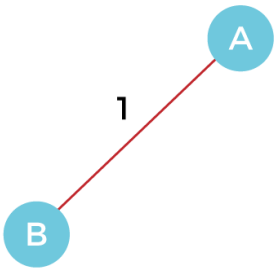


The weight of the edges of the above graph is given in the below table -

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

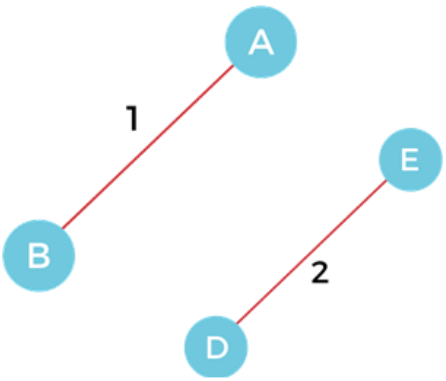
Now, sort the edges given above in the ascending order of their weights.

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



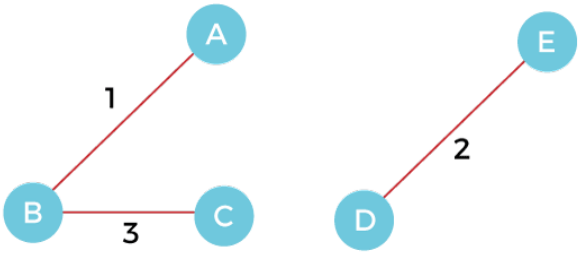
Now, let's start constructing the minimum spanning tree.

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



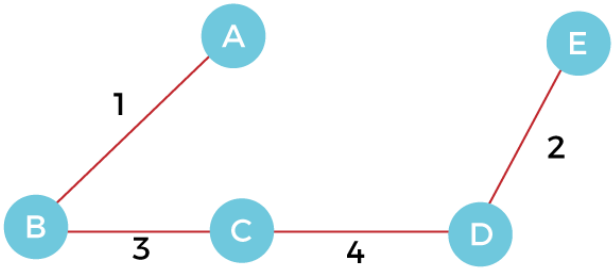
Step 2 - Add the edge **DE** with weight **2** to the MST as it is not creating the cycle.

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



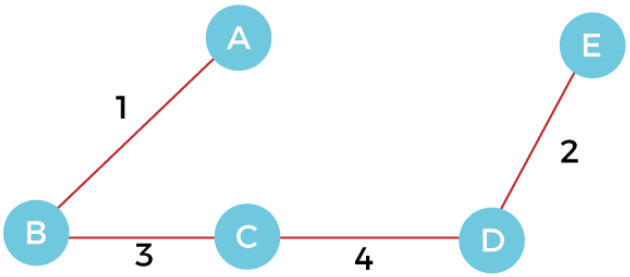
Step 4 - Now, pick the edge **CD** with weight **4** to the MST, as it is not forming the cycle.

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



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

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



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

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

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

Algorithm

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

ELSE
Discard the edge
6. Step 6: END

Time Complexity: O(E * logE) or O(E * logV)

- Sorting of edges takes O(E * logE) time.
- After sorting, we iterate through all edges and apply the find-union algorithm. The find and union operations can take at most O(logV) time.
- So overall complexity is O(E * logE + E * logV) time.
- The value of E can be at most $O(V^2)$, so O(logV) and O(logE) are the same. Therefore, the overall time complexity is O(E * logE) or O(E*logV)

Auxiliary Space: O(V + E), where V is the number of vertices and E is the number of edges in the graph.

Difference Between Prims and Kruskal Algo.

Sr. No.	Prim's Algorithm	Kruskal's Algorithm
1	It starts to build the Minimum Spanning Tree from any vertex in the graph.	It starts to build the Minimum Spanning Tree from the vertex carrying minimum weight in the graph.
2	It traverses one node more than one time to get the minimum distance.	It traverses one node only once.
3	Prim's algorithm has a time complexity of $O(V^2)$, V being the number of vertices and can be improved up to O(E log V) using Fibonacci heaps.	Kruskal's algorithm's time complexity is O(E log V), V being the number of vertices.
4	Prim's algorithm gives connected component as well as it works only on connected graph.	Kruskal's algorithm can generate forest(disconnected components) at any instant as well as it can work on disconnected components
5	Prim's algorithm runs faster in dense graphs.	Kruskal's algorithm runs faster in sparse graphs.
6	It generates the minimum spanning tree starting from the root vertex.	It generates the minimum spanning tree starting from the least weighted edge.
7	Applications of prim's algorithm are Travelling Salesman Problem, Network for roads and Rail tracks connecting all the cities etc.	Applications of Kruskal algorithm are LAN connection, TV Network etc.
8	Prim's algorithm prefer list data structures.	Kruskal's algorithm prefer heap data structures.