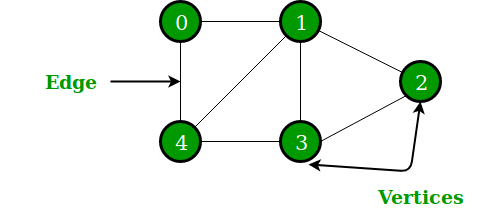
***Graph***

1. A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as,
2. *A Graph consists of a finite set of vertices(or nodes) and a set of Edges which connect a pair of nodes.*
3. Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, locale etc.

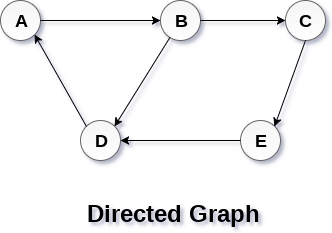
**A finite set of ordered pairs of the form (u, v) called an edge. The pair is ordered because (u, v) is not the same as (v, u) in case of a directe**

**d 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.**



***Undirected Graph***

In the above Graph, the set of vertices V = {0,1,2,3,4} and the set of edges E = {01, 12, 23, 34, 04, 14, 13}.



**A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.**

**A Graph G(V, E) with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.**

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

**1.** Adjacency Matrix

**2.** Adjacency List

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

**Adjacency Matrix:**

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

The adjacency matrix for the above example graph is:

Adjacency Matrix Representation

*Pros:* 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).

*Cons:* Consumes more **space O(V^2).** Even if the graph is sparse(contains less number of edges), it consumes the same space. **Adding** a **vertex** is **O(V^2)** time. Mainly used in **dense graphs.**

**Python implementation of adjacency matrix.**

#A simple representation of graph using Adjacency Matrix

class Graph:

def \_\_init\_\_(self,numvertex):

self.adjMatrix = [[-1]\*numvertex for x in range(numvertex)]

self.numvertex = numvertex

self.vertices = {}

self.verticeslist =[0]\*numvertex

def set\_vertex(self,vtx,id):

if 0<=vtx<=self.numvertex:

self.vertices[id] = vtx

self.verticeslist[vtx] = id

def set\_edge(self,frm,to,cost=0):

frm = self.vertices[frm]

to = self.vertices[to]

self.adjMatrix[frm][to] = cost

#for directed graph do not add this

self.adjMatrix[to][frm] = cost

def get\_vertex(self):

return self.verticeslist

def get\_edges(self):

edges=[]

for i in range (self.numvertex):

for j in range (self.numvertex):

if (self.adjMatrix[i][j]!=-1):

edges.append((self.verticeslist[i],self.verticeslist[j],self.adjMatrix[i][j]))

return edges

def get\_matrix(self):

return self.adjMatrix

G =Graph(6)

G.set\_vertex(0,'a')

G.set\_vertex(1,'b')

G.set\_vertex(2,'c')

G.set\_vertex(3,'d')

G.set\_vertex(4,'e')

G.set\_vertex(5,'f')

G.set\_edge('a','e',10)

G.set\_edge('a','c',20)

G.set\_edge('c','b',30)

G.set\_edge('b','e',40)

G.set\_edge('e','d',50)

G.set\_edge('f','e',60)

print("Vertices of Graph")

print(G.get\_vertex())

print("Edges of Graph")

print(G.get\_edges())

print("Adjacency Matrix of Graph")

print(G.get\_matrix())

#This code is contributed by Rajat Singhal

**Output**

**Vertices of Graph**

**['a', 'b', 'c', 'd', 'e', 'f']**

**Edges of Graph**

**[('a', 'c', 20), ('a', 'e', 10), ('b', 'c', 30), ('b', 'e', 40), ('c', 'a', 20), ('c', 'b', 30), ('d', 'e', 50), ('e', 'a', 10), ('e', 'b', 40), ('e', 'd', 50), ('e', 'f', 60), ('f', 'e', 60)]**

**Adjacency Matrix of Graph**

**[[0 , 0 , 20, 0 , 10 , 0],**

**[0 , 0 , 30, 0 , 40 , 0],**

**[20, 30, 0 , 0 , 0 , 0],**

**[0 , 0 , 0 , 0, 50 , 0],**

**[10, 40, 0 , 50, 0, 60],**

**[0 , 0 , 0 , 0, 60, 0]]**

**Adjacency List:**

An array of 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 list of vertices adjacent to the ***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above graph.

Adjacency List Representation of Graph

Note that in the below implementation, we use dynamic arrays (vector in C++/ArrayList in Java) to represent adjacency lists instead of the linked list. The vector implementation has advantages of cache friendliness.

| **A Python program to demonstrate the adjacency**  **list representation of the graph**    # A class to represent the adjacency list of the node  class AdjNode:  def \_\_init\_\_(self, data):  self.vertex = data  self.next = None      # A class to represent a graph. A graph  # is the list of the adjacency lists.  # Size of the array will be the no. of the  # vertices "V"  class Graph:  def \_\_init\_\_(self, vertices):  self.V = vertices  self.graph = [None] \* self.V    # Function to add an edge in an undirected graph  def add\_edge(self, src, dest):  # Adding the node to the source node  node = AdjNode(dest)  node.next = self.graph[src]  self.graph[src] = node    # Adding the source node to the destination as  # it is the undirected graph  node = AdjNode(src)  node.next = self.graph[dest]  self.graph[dest] = node    # Function to print the graph  def print\_graph(self):  for i in range(self.V):  print("Adjacency list of vertex {}\n head".format(i), end="")  temp = self.graph[i]  while temp:  print(" -> {}".format(temp.vertex), end="")  temp = temp.next  print(" \n")      if \_\_name\_\_ == "\_\_main\_\_":  V = 5  graph = Graph(V)  graph.add\_edge(0, 1)  graph.add\_edge(0, 4)  graph.add\_edge(1, 2)  graph.add\_edge(1, 3)  graph.add\_edge(1, 4)  graph.add\_edge(2, 3)  graph.add\_edge(3, 4)    graph.print\_graph()    **Output** |
| --- |

**Adjacency list of vertex 0**

**head -> 1-> 4**

**Adjacency list of vertex 1**

**head -> 0-> 2-> 3-> 4**

**Adjacency list of vertex 2**

**head -> 1-> 3**

**Adjacency list of vertex 3**

**head -> 1-> 2-> 4**

**Adjacency list of vertex 4**

**head -> 0-> 1-> 3**

***Pros****:* **Saves space O(|V|+|E|)** . In the worst case, there can be **C(V, 2) number of edges** in a graph thus consuming **O(V^2) space.** Adding a vertex is easier.

***Cons:*** Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done **O(V).Searching needs O(V) time.**

BFS and DFS

### **Breadth First Search**

[BFS stands for **Breadth First Search**](https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/) is a vertex based technique for finding a shortest path in graph. It uses a [Queue data structure](https://www.geeksforgeeks.org/queue-data-structure/) which follows first in first out. In BFS,. It is **slower** **than** **DFS**.

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

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

**Algorithm**

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

**Python3 Program to print BFS traversal**

# from a given source vertex. BFS(int s)

# traverses vertices reachable from s.

from collections import defaultdict

# This class represents a directed graph

# using adjacency list representation

class Graph:

# Constructor

def \_\_init\_\_(self):

# default dictionary to store graph

self.graph = defaultdict(list)

# function to add an edge to graph

def addEdge(self,u,v):

self.graph[u].append(v)

# Function to print a BFS of graph

def BFS(self, s):

# Mark all the vertices as not visited

visited = [False] \* (max(self.graph) + 1)

# Create a queue for BFS

queue = []

# Mark the source node as

# visited and enqueue it

queue.append(s)

visited[s] = True

while queue:

# Dequeue a vertex from

# queue and print it

s = queue.pop(0)

print (s, end = " ")

# Get all adjacent vertices of the

# dequeued vertex s. If a adjacent

# has not been visited, then mark it

# visited and enqueue it

for i in self.graph[s]:

if visited[i] == False:

queue.append(i)

visited[i] = True

# Driver code

# Create a graph given in

# the above diagram

g = Graph()

g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(1, 2)

g.addEdge(2, 0)

g.addEdge(2, 3)

g.addEdge(3, 3)

print ("Following is Breadth First Traversal"

" (starting from vertex 2)")

g.BFS(2)

# This code is contributed by Neelam Yadav

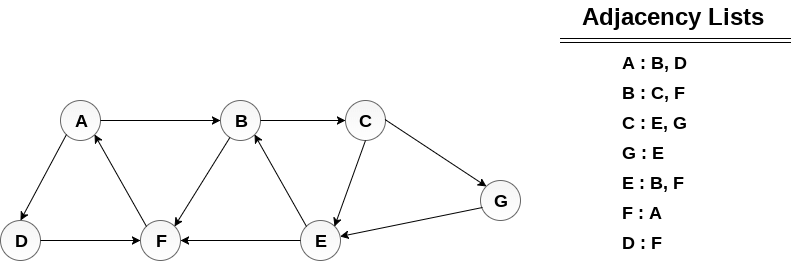
**Output:**

Following is Breadth First Traversal (starting from vertex 2)

2 0 3 1

About The defaultdict in collection Library refer

https://www.geeksforgeeks.org/python-collections-module/

**Ex-**

**Output**

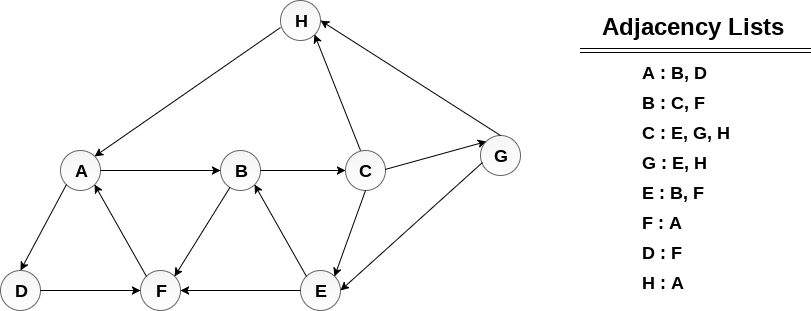
**BFS = {A, B, D, C, F, E,G}**

**Depth First Search**

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

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

**Ex-**

**Output DFS :**

1. H → A → D → F → B → C → G → E

## Algorithm

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

**Complexity Analysis:**

* **Time complexity: O(V + E), where V is the number of vertices and E is the number of edges in the graph.**
* **Space Complexity: O(V).   
  Since, an extra visited array is needed of size V.**

**Python3 program to print DFS traversal**

from collections import defaultdict

# This class represents a directed graph using

# adjacency list representation

class Graph:

# Constructor

def \_\_init\_\_(self):

# default dictionary to store graph

self.graph = defaultdict(list)

# function to add an edge to graph

def addEdge(self, u, v):

self.graph[u].append(v)

# A function used by DFS

def DFSUtil(self, v, visited):

# Mark the current node as visited

# and print it

visited.add(v)

print(v, end=' ')

# Recur for all the vertices

# adjacent to this vertex

for neighbour in self.graph[v]:

if neighbour not in visited:

self.DFSUtil(neighbour, visited)

# The function to do DFS traversal. It uses

# recursive DFSUtil()

def DFS(self, v):

# Create a set to store visited vertices

visited = set()

# Call the recursive helper function

# to print DFS traversal

self.DFSUtil(v, visited)

# Driver code

# Create a graph given

# in the above diagram

g = Graph()

g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(1, 2)

g.addEdge(2, 0)

g.addEdge(2, 3)

g.addEdge(3, 3)

print("Following is DFS from (starting from vertex 2)")

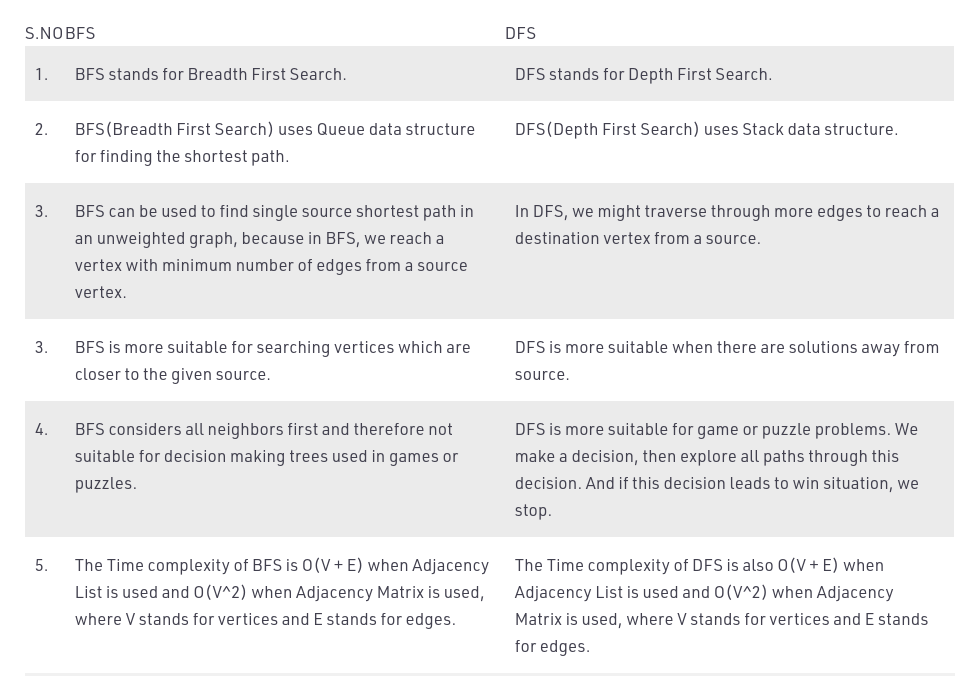
g.DFS(2)

**Output**

Following is Depth First Traversal (starting from vertex 2)

2 0 1 9 3

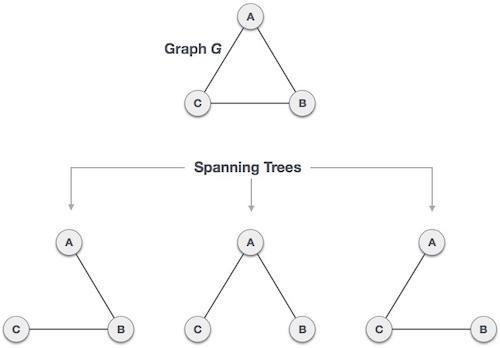
**BFS vs DFS**



# Spanning Tree

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

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



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

## General Properties of Spanning Tree

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

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

## Mathematical Properties of Spanning Tree

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

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

## Application of Spanning Tree

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

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

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

## Minimum Spanning Tree (MST)

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

## Minimum Spanning-Tree Algorithm

We shall learn about two most important spanning tree algorithms here −

* [Kruskal's Algorithm](https://www.tutorialspoint.com/data_structures_algorithms/kruskals_spanning_tree_algorithm.htm)
* [Prim's Algorithm](https://www.tutorialspoint.com/data_structures_algorithms/prims_spanning_tree_algorithm.htm)

Both are greedy algorithms.

Kruskal’s Minimum Spanning Tree Algorithm | Greedy Algo-2

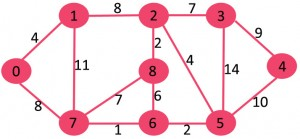
steps for finding MST using Kruskal’s algorithm

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 a cycle is not formed, include this edge. Else, discard it.

3. Repeat step#2 until there are (V-1) edges in the spanning tree.

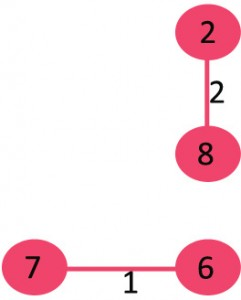
A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

.

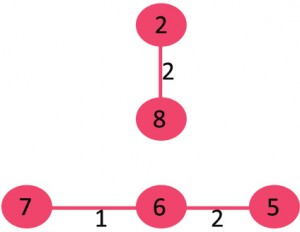
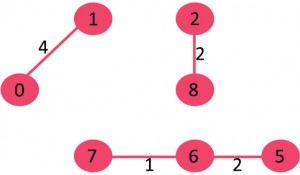
The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

Now pick all edges one by one from sorted list of edges

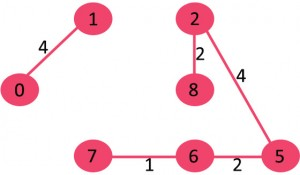
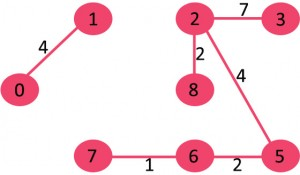
1. Pick edge 7-6: No cycle is formed, include it.

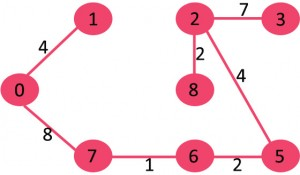
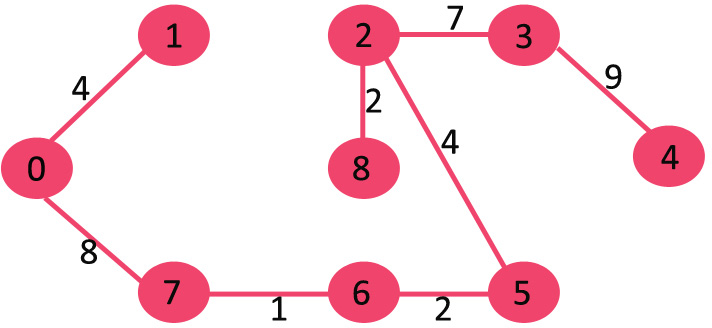
 

1. (2)

(3) (4)

(7) (8)

Since the number of edges included equals (V – 1), the algorithm stops here.

For Program of Kruskals Algorithm

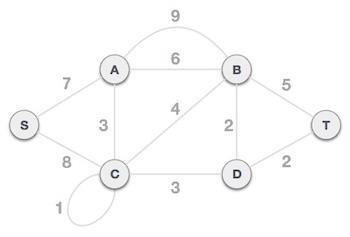
<https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/>

# Prim's Spanning Tree Algorithm

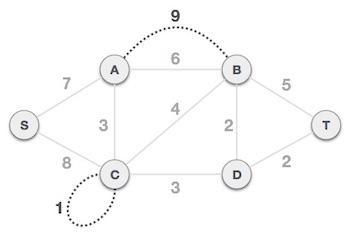
Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the shortest path first algorithms.

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

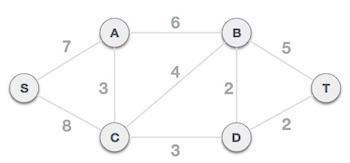
To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example −



## Step 1 - Remove all loops and parallel edges



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

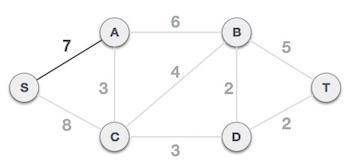


## Step 2 - Choose any arbitrary node as root node

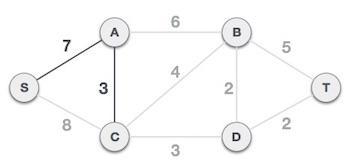
In this case, we choose S node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there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

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

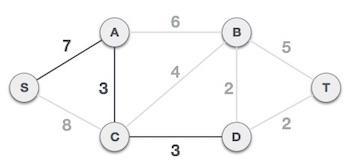
After choosing the root node S, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



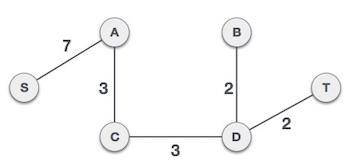
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



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



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



For Programof Prim’s algorithm

<https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/>