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.

In [11]:

*# A Python3 program for*

*# Prim's Minimum Spanning Tree (MST) algorithm.*

*# The program is for adjacency matrix*

*# representation of the graph*

*# Library for INT\_MAX*

import sys

class Graph():

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

self.V = vertices

self.graph = [[0 for column in range(vertices)]

for row in range(vertices)]

*# A utility function to print*

*# the constructed MST stored in parent[]*

def printMST(self, parent):

print("Edge \tWeight")

for i in range(1, self.V):

print(parent[i], "-", i, "\t", self.graph[i][parent[i]])

*# A utility function to find the vertex with*

*# minimum distance value, from the set of vertices*

*# not yet included in shortest path tree*

def minKey(self, key, mstSet):

*# Initialize min value*

min = sys.maxsize

for v in range(self.V):

if key[v] < min and mstSet[v] == False:

min = key[v]

min\_index = v

return min\_index

*# Function to construct and print MST for a graph*

*# represented using adjacency matrix representation*

def primMST(self):

*# Key values used to pick minimum weight edge in cut*

key = [sys.maxsize] \* self.V

parent = [None] \* self.V *# Array to store constructed MST*

*# Make key 0 so that this vertex is picked as first vertex*

key[0] = 0

mstSet = [False] \* self.V

parent[0] = -1 *# First node is always the root of*

for cout in range(self.V):

*# Pick the minimum distance vertex from*

*# the set of vertices not yet processed.*

*# u is always equal to src in first iteration*

u = self.minKey(key, mstSet)

*# Put the minimum distance vertex in*

*# the shortest path tree*

mstSet[u] = True

*# Update dist value of the adjacent vertices*

*# of the picked vertex only if the current*

*# distance is greater than new distance and*

*# the vertex in not in the shortest path tree*

for v in range(self.V):

*# graph[u][v] is non zero only for adjacent vertices of m*

*# mstSet[v] is false for vertices not yet included in MST*

*# Update the key only if graph[u][v] is smaller than key[v]*

if self.graph[u][v] > 0 and mstSet[v] == False \

and key[v] > self.graph[u][v]:

key[v] = self.graph[u][v]

parent[v] = u

self.printMST(parent)

*# Driver's code*

if \_\_name\_\_ == '\_\_main\_\_':

g = Graph(5)

g.graph = [[0, 2, 0, 6, 0],

[2, 0, 3, 8, 5],

[0, 3, 0, 0, 7],

[6, 8, 0, 0, 9],

[0, 5, 7, 9, 0]]

g.primMST()

Edge Weight

0 - 1 2

1 - 2 3

0 - 3 6

1 - 4 5

The Graph class has the following attributes:

V: The number of vertices in the graph. graph: The adjacency matrix of the graph.

The Graph class has the following methods:

**init**: This method initializes the Graph object.

printMST: This method prints the MST of the graph.

minKey: This method finds the vertex with the minimum distance value, from the set of vertices not yet included in the shortest path tree.

primMST: This method constructs and prints the MST of the graph.

**Kruskal's Minimal Spanning Tree Algorithm**

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.

In [9]:

*# Python program for Kruskal's algorithm to find*

*# Minimum Spanning Tree of a given connected,*

*# undirected and weighted graph*

*# Class to represent a graph*

class Graph:

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

self.V = vertices

self.graph = []

*# Function to add an edge to graph*

def addEdge(self, u, v, w):

self.graph.append([u, v, w])

*# A utility function to find set of an element i*

*# (truly uses path compression technique)*

def find(self, parent, i):

if parent[i] != i:

*# Reassignment of node's parent*

*# to root node as*

*# path compression requires*

parent[i] = self.find(parent, parent[i])

return parent[i]

*# A function that does union of two sets of x and y*

*# (uses union by rank)*

def union(self, parent, rank, x, y):

*# Attach smaller rank tree under root of*

*# high rank tree (Union by Rank)*

if rank[x] < rank[y]:

parent[x] = y

elif rank[x] > rank[y]:

parent[y] = x

*# If ranks are same, then make one as root*

*# and increment its rank by one*

else:

parent[y] = x

rank[x] += 1

*# The main function to construct MST*

*# using Kruskal's algorithm*

def KruskalMST(self):

*# This will store the resultant MST*

result = []

*# An index variable, used for sorted edges*

i = 0

*# An index variable, used for result[]*

e = 0

*# Sort all the edges in*

*# non-decreasing order of their*

*# weight*

self.graph = sorted(self.graph,

key=lambda item: item[2])

parent = []

rank = []

*# Create V subsets with single elements*

for node in range(self.V):

parent.append(node)

rank.append(0)

*# Number of edges to be taken is less than to V-1*

while e < self.V - 1:

*# Pick the smallest edge and increment*

*# the index for next iteration*

u, v, w = self.graph[i]

i = i + 1

x = self.find(parent, u)

y = self.find(parent, v)

*# If including this edge doesn't*

*# cause cycle, then include it in result*

*# and increment the index of result*

*# for next edge*

if x != y:

e = e + 1

result.append([u, v, w])

self.union(parent, rank, x, y)

*# Else discard the edge*

minimumCost = 0

print("Edges in the constructed MST")

for u, v, weight in result:

minimumCost += weight

print("%d -- %d == %d" % (u, v, weight))

print("Minimum Spanning Tree", minimumCost)

*# Driver code*

if \_\_name\_\_ == '\_\_main\_\_':

g = Graph(4)

g.addEdge(0, 1, 10)

g.addEdge(0, 2, 6)

g.addEdge(0, 3, 5)

g.addEdge(1, 3, 15)

g.addEdge(2, 3, 4)

*# Function call*

g.KruskalMST()

Edges in the constructed MST

2 -- 3 == 4

0 -- 3 == 5

0 -- 1 == 10

Minimum Spanning Tree 19

The algorithm first creates a list of all of the edges in the graph. It then sorts the edges by weight. Next, it creates two lists: parent and rank. The parent list stores the parent of each vertex in the graph. The rank list stores the rank of each vertex in the graph. The rank of a vertex is a measure of how deep the vertex is in the tree.