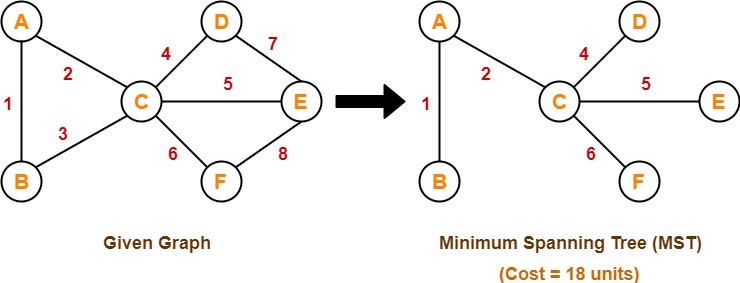
## Practical 7

### **Aim:** Write a program to implement Kruskal’s algorithm

**Theory:**

1. Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which.
2. It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.
3. The most frequent character gets the smallest code and the least frequent character gets the largest code.
4. The main target of the algorithm is to find the subset of edges by using which we can traverse every vertex of the graph
5. In Kruskal's algorithm, we start from edges with the lowest weight and keep adding the edges until the goal is reached.

**Example:**



**Algorithm:**

MST-KRUSKAL(G,w)

1. A = Not Null
2. for each vertex v ∈ G.v
3. MAKE-SET(v)
4. sort the edges of G.E into nondecreasing order by weight w
5. for each edge (u,v) ∈ G.E, taken in nondecreasing order by weight
6. if FIND-SET(u) ≠ FIND-SET(v)
7. A = ∪ {(u,v)}
8. UNION(u,v)
9. return A

**Code:**

class Graph:

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

self.V = vertices # No. of vertices

self.graph = []

# to store 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):

result = [] # This will store the resultant MST

# An index variable, used for sorted edges

i = 0

# An index variable, used for result[]

e = 0

# Step 1: Sort all the edges in

# non-decreasing order of their

# weight. If we are not allowed to change the

# given graph, we can create a copy of graph

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 equal to V-1

while e < self.V - 1:

# Step 2: 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's 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()

print("Neeraj Appari 021")

**Output**:

Edges in the constructed MST

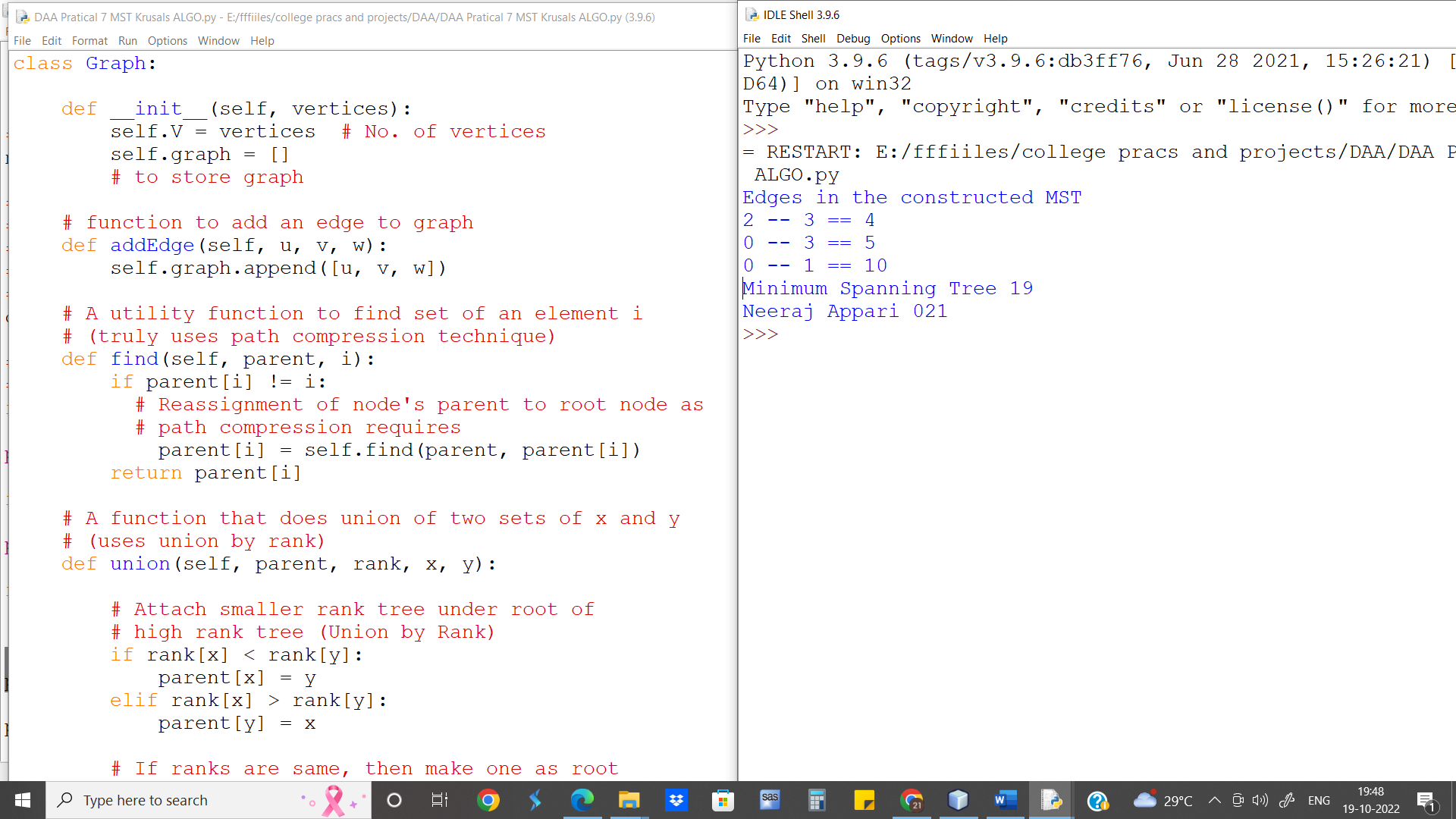
2 -- 3 == 4

0 -- 3 == 5

0 -- 1 == 10

Minimum Spanning Tree 19

Neeraj Appari 021



**Runtime for Kruskal’s Algorithm is O(E log E)**

**Conclusion:** Kruskal’s Algorithm was used to create a MST and calculate the cost of Tree