**Practical No:** 3 Roll No:

**Subject:** Artificial Intelligence

**Title:** Implement Greedy search algorithm for Kruskal's Minimal Spanning Tree Algorithm

**----------------------------------------------------------------------------------------------------------------**

**Program Code: extra.py**

from collections import defaultdict

# Class to represent a graph

class Graph:

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

self.V = vertices # No. of vertices

self.graph = [] # default dictionary

# 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

# (uses path compression technique)

def find(self, parent, i):

if parent[i] == i:

return i

return self.find(parent, 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):

xroot = self.find(parent, x)

yroot = self.find(parent, y)

# Attach smaller rank tree under root of

# high rank tree (Union by Rank)

if rank[xroot] < rank[yroot]:

parent[xroot] = yroot

elif rank[xroot] > rank[yroot]:

parent[yroot] = xroot

# If ranks are same, then make one as root

# and increment its rank by one

else:

parent[yroot] = xroot

rank[xroot] += 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 does't

# cause cycle, include it in result

# and increment the indexof 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

g = Graph(4)

g.addEdge(0, 1, 10)

g.addEdge(0, 2, 6)

g.addEdge(0, 3, 5)

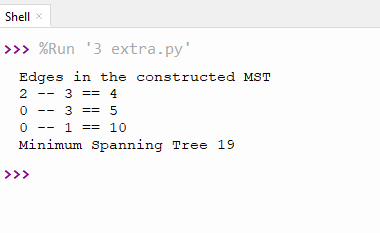
g.addEdge(1, 3, 15)

g.addEdge(2, 3, 4)

# Function call

g.KruskalMST()

**Output :**

****

**Program Code: selection\_sort.py**

def selectionSort(arr):

for i in range(len(arr)):

min = float('-inf')

for j in range(i + 1, len(arr)):

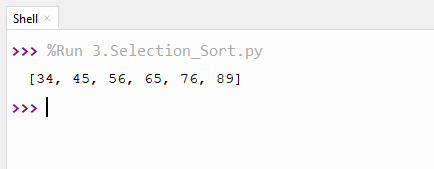
if arr[i] > arr[j]:

arr[i],arr[j] = arr[j], arr[i]

return arr

print(selectionSort([89,56,45,34,65,76]))

**Output :**

****

**Program Code : Job\_scheduling.py**

# Jobs, Profit, Slot

profit = [15,27,10,100, 150]

jobs = ["j1", "j2", "j3", "j4", "j5"]

deadline = [2,3,3,3,4]

profitNJobs = list(zip(profit,jobs,deadline))

profitNJobs = sorted(profitNJobs, key = lambda x: x[0], reverse = True)

slot = []

for \_ in range(len(jobs)):

slot.append(0)

profit = 0

ans = []

for i in range(len(jobs)):

ans.append('null')

for i in range(len(jobs)):

job = profitNJobs[i]

#check if slot is occupied

for j in range(job[2], 0, -1):

if slot[j] == 0:

ans[j] = job[1]

profit += job[0]

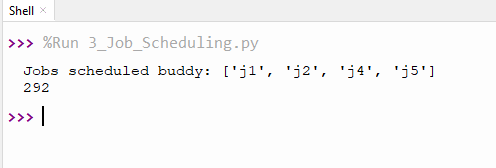
slot[j] = 1

break

print("Jobs scheduled buddy:",ans[1:])

print(profit)

**Output :**

****