**Assignment No : 3**

**Title: Implement Greedy search algorithm for any of the following application:**

**• Selection Sort**

**• Prim's Minimal Spanning Tree Algorithm**

**• Kruskal's Minimal Spanning Tree Algorithm**

**• Dijkstra's Minimal Spanning Tree Algorithm**

**1. Selection sort**

import sys

A = [64, 25, 12, 22, 11]

for i in range(len(A)):

min\_idx = i

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

if A[min\_idx] > A[j]:

min\_idx = j

A[i], A[min\_idx] = A[min\_idx], A[i]

print ("Sorted array")

for i in range(len(A)):

print("%d" %A[i],end=" , ")

OUTPUT:

Sorted array

11 , 12 , 22 , 25 , 64

**2. Prim's Minimal Spanning Tree Algorithm**

**import** sys

**class** Graph():

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

self.V **=** vertices

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

**def** printMST(self, parent):

**print**("Edge \tWeight")

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

**print**(parent[i], "-", i, "\t", self.graph[i][parent[i]]) **def** minKey(self, key, mstSet):

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

**def** primMST(self):

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

parent **=** [None] **\*** self.V

key[0] **=** 0

mstSet **=** [False] **\*** self.V

parent[0] **= -**1

**for** cout **in** range(self.V):

u **=** self.minKey(key, mstSet)

mstSet[u] **=** True

**for** v **in** range(self.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)

**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()

OUTPUT:

Edge Weight

0 - 1 2

1 - 2 3

0 - 3 6

1 - 4 5

Time complexity:

O(V2)

**3. Kruskal's Minimal Spanning Tree Algorithm class Graph:**

def \_init\_(self, vertices):

self.V = vertices

self.graph = []

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

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

def find(self, parent, i):

if parent[i] != i:

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

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

if rank[x] < rank[y]:

parent[x] = y

elif rank[x] > rank[y]:

parent[y] = x

else:

parent[y] = x

rank[x] += 1

def KruskalMST(self):

result = []

i = 0

e = 0

self.graph = sorted(self.graph,

key=lambda item: item[2]) parent = []

rank = []

for node in range(self.V):

parent.append(node)

rank.append(0)

while e < self.V - 1:

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

i = i + 1

x = self.find(parent, u)

y = self.find(parent, v)

if x != y:

e = e + 1

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

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

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) 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)

g.KruskalMST()

OUTPUT:

Edges in the constructed MST

2 -- 3 == 4

0 -- 3 == 5

0 -- 1 == 10

Minimum Spanning Tree 19

TIME COMPLEXITY:

O(E \* logE)

**4. Dijkstra's Minimal Spanning Tree Algorithm**

import sys

vertices = [[0, 0, 1, 1, 0, 0, 0],

[0, 0, 1, 0, 0, 1, 0],

[1, 1, 0, 1, 1, 0, 0],

[1, 0, 1, 0, 0, 0, 1],

[0, 0, 1, 0, 0, 1, 0],

[0, 1, 0, 0, 1, 0, 1],

[0, 0, 0, 1, 0, 1, 0]]

edges = [[0, 0, 1, 2, 0, 0, 0],

[0, 0, 2, 0, 0, 3, 0],

[1, 2, 0, 1, 3, 0, 0],

[2, 0, 1, 0, 0, 0, 1],

[0, 0, 3, 0, 0, 2, 0],

[0, 3, 0, 0, 2, 0, 1],

[0, 0, 0, 1, 0, 1, 0]]

def to\_be\_visited():

global visited\_and\_distance

v = -10

for index in range(num\_of\_vertices):

if visited\_and\_distance[index][0] == 0 \

and (v < 0 or visited\_and\_distance[index][1] <= visited\_and\_distance[v][1]):

v = index

return v

num\_of\_vertices = len(vertices[0])

visited\_and\_distance = [[0, 0]]

for i in range(num\_of\_vertices-1):

visited\_and\_distance.append([0, sys.maxsize])

for vertex in range(num\_of\_vertices):

# Find next vertex to be visited

to\_visit = to\_be\_visited()

for neighbor\_index in range(num\_of\_vertices):

# Updating new distances

if vertices[to\_visit][neighbor\_index] == 1 and \ visited\_and\_distance[neighbor\_index][0] == 0: new\_distance = visited\_and\_distance[to\_visit][1] \ + edges[to\_visit][neighbor\_index]

if visited\_and\_distance[neighbor\_index][1] > new\_distance: visited\_and\_distance[neighbor\_index][1] = new\_distance

visited\_and\_distance[to\_visit][0] = 1

i = 0

# Printing the distance

for distance in visited\_and\_distance:

print("Distance of ", chr(ord('a') + i),

" from source vertex: ", distance[1])

i = i + 1

OUTPUT:

Distance of a from source vertex: 0 Distance of b from source vertex: 3 Distance of c from source vertex: 1 Distance of d from source vertex: 2 Distance of e from source vertex: 4 Distance of f from source vertex: 4 Distance of g from source vertex: 3

TIME COMPLEXITY:

O(E Log V)