Zeal College of Engineering and Research, Pune

Department of Computer Engineering

DATA STRUCTURE LAB

Assignment No. 7

# Name :PUKALE PRASHANT GUNDA

# Branch : SE

Div. : B

# Roll No. : S212032

**Problem Statement**

You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimumtotal cost. Solve the problem by suggesting appropriate data structures.

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 doesn'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()

# For Prim’s

import sys # Library for INT\_MAX

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)

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();

