# Python3 code for Job Scheduling

# function to schedule the jobs take 2 arguments array and no of jobs to schedule

def printJobScheduling(arr, t):

# length of array

n = len(arr)

# Sort all jobs according to decreasing order of profit

for i in range(n):

for j in range(n - 1 - i):

if arr[j][2] < arr[j + 1][2]:

arr[j], arr[j + 1] = arr[j + 1], arr[j]

# To keep track of free time slots

result = [False] \* t

# To store result (Sequence of jobs)

job = ['-1'] \* t

# Iterate through all given jobs

for i in range(len(arr)):

# Find a free slot for this job (Note that we start from the last possible slot)

for j in range(min(t - 1, arr[i][1] - 1), -1, -1):

# Free slot found

if result[j] is False:

result[j] = True

job[j] = arr[i][0]

break

# print the sequence

print(job)

# Driver's Code

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

arr = [['a', 2, 100], # Job Array

['b', 1, 19],

['c', 2, 27],

['d', 1, 25],

['e', 3, 15]]

print("Following is maximum profit sequence of jobs")

# Function Call

printJobScheduling(arr, 3)

Output:

Following is maximum profit sequence of jobs

['c', 'a', 'e']

# 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()

Output:

Edges in the constructed MST

2 -- 3 == 4

0 -- 3 == 5

0 -- 1 == 10

Minimum Spanning Tree 19

# Python program for Dijkstra's single source shortest path algorithm. The program is for adjacency matrix representation of the graph

class Graph():

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

self.V = vertices

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

for row in range(vertices)]

def printSolution(self, dist):

print("Vertex \t Distance from Source")

for node in range(self.V):

print(node, "\t\t", dist[node])

# A utility function to find the vertex with minimum distance value, from the set of vertices not yet included in shortest path tree

def minDistance(self, dist, sptSet):

# Initialize minimum distance for next node

min = 1e7

# Search not nearest vertex not in the shortest path tree

for v in range(self.V):

if dist[v] < min and sptSet[v] == False:

min = dist[v]

min\_index = v

return min\_index

# Function that implements Dijkstra's single source shortest path algorithm for a graph represented using adjacency matrix representation

def dijkstra(self, src):

dist = [1e7] \* self.V

dist[src] = 0

sptSet = [False] \* self.V

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.minDistance(dist, sptSet)

# Put the minimum distance vertex in the shortest path tree

sptSet[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):

if (self.graph[u][v] > 0 and

sptSet[v] == False and

dist[v] > dist[u] + self.graph[u][v]):

dist[v] = dist[u] + self.graph[u][v]

self.printSolution(dist)

# Driver program

g = Graph(9)

g.graph = [[0, 4, 0, 0, 0, 0, 0, 8, 0],

[4, 0, 8, 0, 0, 0, 0, 11, 0],

[0, 8, 0, 7, 0, 4, 0, 0, 2],

[0, 0, 7, 0, 9, 14, 0, 0, 0],

[0, 0, 0, 9, 0, 10, 0, 0, 0],

[0, 0, 4, 14, 10, 0, 2, 0, 0],

[0, 0, 0, 0, 0, 2, 0, 1, 6],

[8, 11, 0, 0, 0, 0, 1, 0, 7],

[0, 0, 2, 0, 0, 0, 6, 7, 0]

]

g.dijkstra(0)

Output:

Vertex Distance from Source

0 0

1 4

2 12

3 19

4 21

5 11

6 9

7 8

8 14

# Python program for implementation of Selection Sort

import sys

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

# Traverse through all array elements

for i in range(len(A)):

# Find the minimum element in remaining unsorted array

min\_idx = i

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

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

min\_idx = j

# Swap the found minimum element with the first element

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

# Driver code to test above

print ("Sorted array")

for i in range(len(A)):

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

Output:

Sorted array

11 12 22 25 64

# Python 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()

Output:

Edge Weight

0 - 1 2

1 - 2 3

0 - 3 6

1 - 4 5