( AI )

A-3

Implement Greedy search algorithm for any of the following application:

I. Selection Sort

def selectionSort(array, size):

for step in range(size - 1):

min\_idx = step

for i in range(step + 1, size):

if array[i] < array[min\_idx]:

min\_idx = i

(array[step], array[min\_idx]) = (array[min\_idx], array[step])

data = [20, 12, 10, 15, 2]

size = len(data)

selectionSort(data, size)

print('Sorted Array in Ascending Order:')

print(data)

Output :-

Sorted Array in Ascending Order:

[2, 10, 12, 15, 20]

ii. Single source shortest path

def dijkstra(graph, src):

dist = {vertex: float('inf') for vertex in graph} # Dictionary to store the shortest distance from source to vertex

visited = {vertex: False for vertex in graph} # Dictionary to keep track of visited vertices

prev = {vertex: None for vertex in graph} # Dictionary to store the predecessor of each vertex in the shortest path

dist[src] = 0 # Distance from source to itself is 0

while True:

u = min(dist, key=dist.get) # Find the vertex with the minimum distance

if visited[u]:

break

visited[u] = True

for v in graph[u]: # Update the distance of the adjacent vertices

if not visited[v] and dist[u] + graph[u][v] < dist[v]:

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

prev[v] = u

return dist, prev

graph = {

'A': {'B': 1, 'C': 4},

'B': {'A': 1, 'C': 2, 'D': 5},

'C': {'A': 4, 'B': 2, 'D': 1},

'D': {'B': 5, 'C': 1}

}

dist, prev = dijkstra(graph, 'A')

print('Shortest distances from vertex A:')

for vertex, distance in dist.items():

print(f'Vertex {vertex}: Distance = {distance}, Predecessor = {prev[vertex]}')

Output :-

Shortest distances from vertex A:

Vertex A: Distance = 0, Predecessor = None

Vertex B: Distance = 1, Predecessor = A

Vertex C: Distance = 4, Predecessor = A

Vertex D: Distance = inf, Predecessor = None

iii. Job Scheduling Problem

def jobScheduling(jobs):

jobs.sort(key=lambda x: (-x[1], x[0])) # Sort jobs by deadline and profit in descending order

sequence, total\_profit = [], 0

for job in jobs:

time\_slot = job[0]

while sequence and sequence[-1][1] > time\_slot:

time\_slot = sequence[-1][1] + 1

sequence.append((job[0], time\_slot, job[2]))

total\_profit += job[2]

return sequence, total\_profit

jobs = [(1, 4, 20), (2, 2, 100), (3, 1, 40), (4, 3, 35)]

sequence, total\_profit = jobScheduling(jobs)

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

for job in sequence:

print(job[0], end=" ")

print("\nTotal Profit:", total\_profit)

Output :-

Following is maximum profit sequence of jobs:

1 4 2 3

Total Profit: 195

IV) Prim's Minimal Spanning Tree Algorithm

def prim(graph):

n = len(graph)

visited = [False] \* n

mst = []

visited[0] = True

for \_ in range(n - 1):

min\_edge = float('inf')

u, v = -1, -1

for i in range(n):

if visited[i]:

for j in range(n):

if not visited[j] and graph[i][j] < min\_edge:

min\_edge = graph[i][j]

u, v = i, j

mst.append((u, v, min\_edge))

visited[v] = True

return mst

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]

]

mst = prim(graph)

print("Minimum Spanning Tree:")

for edge in mst:

print(f"{edge[0]} - {edge[1]} : {edge[2]}")

Output :-

Minimum Spanning Tree:

0 - 2 : 0

0 - 4 : 0

2 - 3 : 0

0 - 1 : 2

V) Kruskal's Minimal Spanning Tree Algorithm

class Graph:

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

self.V = vertices

self.graph = []

def add\_edge(self, u, v, w):

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

# Search function

def find(self, parent, i):

if parent[i] == i:

return i

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

def apply\_union(self, parent, rank, x, y):

xroot = self.find(parent, x)

yroot = self.find(parent, y)

if rank[xroot] < rank[yroot]:

parent[xroot] = yroot

elif rank[xroot] > rank[yroot]:

parent[yroot] = xroot

else:

parent[yroot] = xroot

rank[xroot] += 1

# Applying Kruskal algorithm

def kruskal\_algo(self):

result = []

i, e = 0, 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.apply\_union(parent, rank, x, y)

for u, v, weight in result:

print("%d - %d: %d" % (u, v, weight))

g = Graph(6)

g.add\_edge(0, 1, 4)

g.add\_edge(0, 2, 4)

g.add\_edge(1, 2, 2)

g.add\_edge(1, 0, 4)

g.add\_edge(2, 0, 4)

g.add\_edge(2, 1, 2)

g.add\_edge(2, 3, 3)

g.add\_edge(2, 5, 2)

g.add\_edge(2, 4, 4)

g.add\_edge(3, 2, 3)

g.add\_edge(3, 4, 3)

g.add\_edge(4, 2, 4)

g.add\_edge(4, 3, 3)

g.add\_edge(5, 2, 2)

g.add\_edge(5, 4, 3)

g.kruskal\_algo()

Output :-

0 - 1: 4

2 - 3: 3

2 - 5: 2

VI) Dijkstra's Minimal Spanning Tree Algorithm

import heapq

class Graph:

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

self.V = vertices

self.graph = []

def add\_edge(self, u, v, w):

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

def dijkstra(self, src):

dist = [float('inf')] \* self.V

dist[src] = 0

visited = [False] \* self.V

heap = [(0, src)]

while heap:

d, node = heapq.heappop(heap)

if visited[node]:

continue

visited[node] = True

for edge in self.graph:

if edge[0] == node and not visited[edge[1]]:

new\_dist = dist[node] + edge[2]

if new\_dist < dist[edge[1]]:

dist[edge[1]] = new\_dist

heapq.heappush(heap, (new\_dist, edge[1]))

return dist

g = Graph(9)

g.add\_edge(0, 1, 4)

g.add\_edge(0, 7, 8)

g.add\_edge(1, 2, 8)

g.add\_edge(1, 7, 11)

g.add\_edge(2, 3, 7)

g.add\_edge(2, 5, 4)

g.add\_edge(2, 8, 2)

g.add\_edge(3, 4, 9)

g.add\_edge(3, 5, 14)

g.add\_edge(4, 5, 10)

g.add\_edge(5, 6, 2)

g.add\_edge(6, 7, 1)

g.add\_edge(6, 8, 6)

g.add\_edge(7, 8, 7)

distances = g.dijkstra(0)

print(distances)

Output :-

[0, 4, 12, 19, 28, 16, 18, 8, 14]