Krushakal+djikshtra

#krushkals

class Graph:

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

self.V = vertices

self.graph = []

self.heuristics = {}

def addEdge(self, u, v, w, h=0):

# Add edge to the graph and heuristic values to the dictionary

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

self.heuristics[(u, v)] = h

self.heuristics[(v, u)] = h

def find(self, parent, i):

# Find function with path compression

if parent[i] != i:

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

return parent[i]

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

# Union function to connect two components

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 = [] # To store the resulting MST

i = 0 # Initial edge index

e = 0 # Initial count of edges in the MST

# Sort edges based on the weight and heuristic

self.graph.sort(key=lambda item: item[2] + self.heuristics.get((item[0], item[1]), 0))

# Initialize disjoint sets

parent = list(range(self.V))

rank = [0] \* self.V

while e < self.V - 1 and i < len(self.graph):

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

i += 1

x = self.find(parent, u)

y = self.find(parent, v)

if x != y: # If u and v are not in the same set, add edge to MST

e += 1

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

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

# Print the result

minimumCost = sum([weight for \_, \_, weight in result])

print("\nEdges in the constructed MST:")

for u, v, weight in result:

print(f"{u} -- {v} == {weight}")

print("Minimum Spanning Tree Cost:", minimumCost)

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

print("Name: minimum spanning tree-krushkals ")

print("Kruskal's Algorithm is a greedy algorithm used to find the Minimum Spanning Tree (MST) .we sort all edges here pick smallest edge and check if it doesn'St form cycle ")

print("Heuristic is used for helping to choose the next edge. Edge with low weight+ low heuristic value is chosen")

# Input the number of vertices

vertices = int(input("Enter the number of vertices: "))

g = Graph(vertices)

# Input the number of edges

edges = int(input("Enter the number of edges(V-1): "))

print("Enter the edges in the format: u v weight heuristic")

# Input edges and heuristics

for \_ in range(edges):

u, v, w, h = map(int, input().split()) # Read edge: u, v, weight, heuristic

g.addEdge(u, v, w, h)

# Run Kruskal's Algorithm to find the Minimum Spanning Tree (MST)

g.KruskalMST()

#sample input

#Enter the number of vertices: 5

#Enter the number of edges: 4

#Enter the edges in the format: u v weight heuristic

#0 1 10 2

# 2 4 6 1

# 0 3 5 0

#1 3 15 1

#djikshtra

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 printSolution(self, dist, heuristics):

print("\nFinal Solution:")

print("Vertex \tDistance from Source \tHeuristic Value")

for node in range(self.V):

print(f"{node} \t\t {dist[node]} \t\t {heuristics[node]}")

print("Heuristic function is the estimated cost from the current node")

def minDistance(self, dist, sptSet):

min\_val = sys.maxsize

min\_index = -1

for u in range(self.V):

if dist[u] < min\_val and not sptSet[u]:

min\_val = dist[u]

min\_index = u

return min\_index

def dijkstra(self, src, heuristics):

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

dist[src] = 0

sptSet = [False] \* self.V

for cout in range(self.V):

x = self.minDistance(dist, sptSet)

sptSet[x] = True

for y in range(self.V):

if self.graph[x][y] > 0 and not sptSet[y] and dist[y] > dist[x] + self.graph[x][y]:

dist[y] = dist[x] + self.graph[x][y]

# Call printSolution to display final result

self.printSolution(dist, heuristics)

if \_\_name\_\_ == "\_\_main\_\_":

vertices = int(input("Enter the number of vertices: "))

graph = []

print("Enter the graph matrix:")

for i in range(vertices):

row = list(map(int, input().split()))

graph.append(row)

g = Graph(vertices)

g.graph = graph

# Get heuristic values from the user

heuristics = []

print("Enter the heuristic values for each vertex:")

for i in range(vertices):

h\_value = int(input(f"Heuristic value for vertex {i}: "))

heuristics.append(h\_value)

src = int(input("Enter the source vertex: "))

g.dijkstra(src, heuristics)

# Enter the number of vertices: 4

# Enter the graph matrix:

# 10 25 14 12

# 21 30 15 13

# 21 16 27 31

# 2 1 45 20

# Enter the heuristic values for each vertex:

# Heuristic value for vertex 0: 2

# Heuristic value for vertex 1: 3

# Heuristic value for vertex 2: 1

# Heuristic value for vertex 3: 0

# Enter the source vertex: 0

# Final Solution:

# Vertex Distance from Source Heuristic Value

# 0 0 2

# 1 13 3

# 2 14 1

# 3 12 0

# Heuristic function is the estimated cost from the current node