Dijktras :

import sys

# Class representing a Graph

class Graph:

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

self.V = vertices

# Initialize adjacency matrix with 0s

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

# Utility function to print shortest distance from source

def printSolution(self, dist):

print("\nVertex \tDistance from Source")

for node in range(self.V):

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

# Find the vertex with minimum distance not yet included in the shortest path tree

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

# Dijkstra’s algorithm implementation

def dijkstra(self, src):

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

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

sptSet = [False] \* self.V # Set of vertices included in shortest path tree

for \_ in range(self.V):

x = self.minDistance(dist, sptSet)

sptSet[x] = True

# Update distances of adjacent vertices

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]

self.printSolution(dist)

# ---- Main Execution ----

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

print("\nName: Prachi Karande")

print("Roll No: TACO22134")

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

graph = []

print("Enter the adjacency matrix row by row (use 0 for no edge):")

for i in range(vertices):

row = list(map(int, input(f"Row {i}: ").split()))

graph.append(row)

g = Graph(vertices)

g.graph = graph

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

g.dijkstra(src)

# Heuristic explanation

print("\nHeuristic Function Used:")

print("This implementation uses Dijkstra's Algorithm, which is a greedy algorithm.")

print("The algorithm selects the unvisited node with the minimum known distance from the source node.")

print("Function minimized: f(node) = g(node)")

print("Where g(node) is the actual shortest distance from the source node to the current node.")

krushkal:  
# Disjoint Set data structure with path compression and union by rank

class DisjointSet:

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

self.parent = list(range(n))

self.rank = [0] \* n

def find(self, x):

# Heuristic: Path Compression

if self.parent[x] != x:

self.parent[x] = self.find(self.parent[x])

return self.parent[x]

def union(self, x, y):

rootX = self.find(x)

rootY = self.find(y)

if rootX != rootY:

# Heuristic: Union by Rank

if self.rank[rootX] > self.rank[rootY]:

self.parent[rootY] = rootX

elif self.rank[rootX] < self.rank[rootY]:

self.parent[rootX] = rootY

else:

self.parent[rootY] = rootX

self.rank[rootX] += 1

# Kruskal's algorithm to find MST

def kruskal(n, edges):

ds = DisjointSet(n)

mst = []

total\_weight = 0 # Initialize total weight of the MST

# Sort edges based on weight

edges.sort(key=lambda x: x[2])

for u, v, weight in edges:

if ds.find(u) != ds.find(v):

ds.union(u, v)

mst.append((u, v, weight))

total\_weight += weight

return mst, total\_weight

# ---- Main Execution ----

print("\nName: Prachi Karande")

print("Roll no: TACO22134")

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

e = int(input("Enter the number of edges: "))

edges = []

for \_ in range(e):

u, v, weight = map(int, input("Enter edge (u v weight): ").split())

edges.append((u, v, weight))

# Running Kruskal's algorithm

mst, total\_weight = kruskal(n, edges)

# Output the MST

print("\nMinimum Spanning Tree using Kruskal's Algorithm:", mst)

# Output the total cost of the MST

print(f"Total cost of the Minimum Spanning Tree: {total\_weight}")

# Expression explanation

print("\nExpression for Edge Selection in Kruskal's Algorithm:")

print("f(u, v) = w(u, v) if find(u) != find(v)")