**DSA ASSIGNMENT:GRAPHS**

**Roll Number: CB.EN.U4CCE22046**

**GITHUB LINK:**

**Dijkstra's Algorithm:**

**AIM:**

**Determine the briefest routes from an initial vertex to every other vertex within a graph with weighted edges.**

**ALGORITHM:**

**Initializing:**

**• Set the distance array (dist) for all vertices, except the source, to a large value (intmax), with the source set to 0.**

**• Create an array count to track vertices included in the shortest path tree, initially marked as False for all vertices.**

**Main Loop:**

**Iterate through the following steps for each vertex:**

**Select Vertex (x):**

**• Identify the vertex x with the minimum distance value from the set of unprocessed vertices (count[x] == False).**

**Include Vertex (count[x] = True):**

**• Mark the selected vertex x as processed.**

**Updating the Distances (dist):**

**• For each vertex y not in the shortest path tree and adjacent to x:**

**• If the distance to y through x is shorter than the current distance to y, update the distance.**

**Code:**

**class ShortestPathFinder:**

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

**self.V = vertices**

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

**def print\_shortest\_paths(self, distances):**

**print("Vertex \tDistance from Source")**

**for node, distance in enumerate(distances):**

**print(node, "\t\t ", distance)**

**print()**

**def find\_min\_distance\_vertex(self, distances, visited):**

**min\_distance = float('inf')**

**min\_vertex = -1**

**for v in range(self.V):**

**if distances[v] < min\_distance and not visited[v]:**

**min\_distance = distances[v]**

**min\_vertex = v**

**return min\_vertex**

**def dijkstra\_algorithm(self, source):**

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

**distances[source] = 0**

**visited = [False] \* self.V**

**for \_ in range(self.V):**

**current\_vertex = self.find\_min\_distance\_vertex(distances, visited)**

**visited[current\_vertex] = True**

**for neighbor\_vertex in range(self.V):**

**if (**

**not visited[neighbor\_vertex]**

**and self.graph[current\_vertex][neighbor\_vertex] > 0**

**and distances[neighbor\_vertex] > distances[current\_vertex] + self.graph[current\_vertex][**

**neighbor\_vertex]**

**):**

**distances[neighbor\_vertex] = distances[current\_vertex] + self.graph[current\_vertex][neighbor\_vertex]**

**self.print\_shortest\_paths(distances)**

**# Test Case 1**

**graph1 = ShortestPathFinder(6)**

**graph1.graph = [**

**[0, 5, 2, 0, 4, 0],**

**[5, 0, 5, 0, 0, 0],**

**[2, 5, 0, 0, 0, 0],**

**[0, 3, 0, 0, 0, 2],**

**[4, 0, 0, 0, 0, 0],**

**[0, 0, 0, 3, 0, 0]**

**]**

**print("Test Case 1:")**

**graph1.dijkstra\_algorithm(5)**

**# Test Case 2**

**graph2 = ShortestPathFinder(5)**

**graph2.graph = [**

**[0, 4, 0, 0, 0],**

**[4, 0, 7, 0, 0],**

**[0, 7, 0, 3, 0],**

**[0, 0, 3, 0, 5],**

**[0, 0, 0, 5, 0]**

**]**

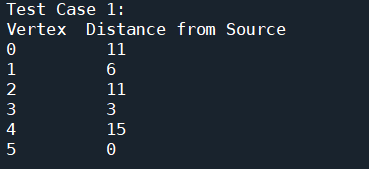
**print("Test Case 2:")**

**graph2.dijkstra\_algorithm(0)**

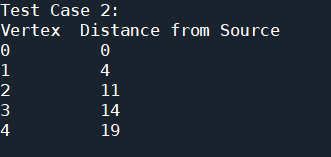
**Output :**

* **Printing the final distances from the source vertex to all other vertices.**

Test Case 1:



Test Case 2:



**Floyd's Algorithm:**

**AIM:**

**The aim of this simulation is to employ Floyd's algorithm for determining the shortest paths connecting all pairs of vertices within a weighted graph.**

**ALGORITHM:**

**Initialization:**

**v: Represents the number of vertices in the graph.**

**INF: A sizable value serving as a representation of infinity.**

**Function: floyd\_warshall(G)**

**Input:**

**G: The graph's adjacency matrix (where G[i][j] denotes the weight of the edge from vertex i to vertex j).**

**Output:**

**Outputs the shortest distances between every pair of vertices.**

**Steps:**

**Generate a duplicate of the graph to serve as the distance matrix.**

**Iterate through all vertices k.**

**For each pair of vertices i and j, update the distance if the path through k is shorter.**

**Display the solution.**

**Function: print\_solution(distance)**

**Input:**

**distance: The matrix containing the shortest distances between all pairs of vertices.**

**CODE:**

**INF = float('inf')**

**def floyd\_warshall(graph):**

**vertices = len(graph)**

**distance\_matrix = [row[:] for row in graph]**

**for k in range(vertices):**

**for i in range(vertices):**

**for j in range(vertices):**

**if distance\_matrix[i][k] + distance\_matrix[k][j] < distance\_matrix[i][j]:**

**distance\_matrix[i][j] = distance\_matrix[i][k] + distance\_matrix[k][j]**

**print\_solution(distance\_matrix)**

**def print\_solution(distance\_matrix):**

**print("Shortest Distances between all pairs of vertices:")**

**for row in distance\_matrix:**

**print(row)**

**# Test Case 1**

**graph1 = [**

**[0, 6, INF, 0, 8, 0],**

**[6, 0, 7, 0, 0, 0],**

**[3, 7, 0, 0, 0, 0],**

**[0, 4, 0, 0, 0, 5],**

**[8, 0, 0, 0, 0, 0],**

**[0, 0, 0, 9, 0, 0]**

**]**

**print("Test Case 1:")**

**floyd\_warshall(graph1)**

**# Test Case 2**

**graph2 = [**

**[0, 3, INF, INF, INF],**

**[3, 0, 8, INF, INF],**

**[INF, 8, 0, 4, INF],**

**[INF, INF, 4, 0, 6],**

**[INF, INF, INF, 6, 0]**

**]**

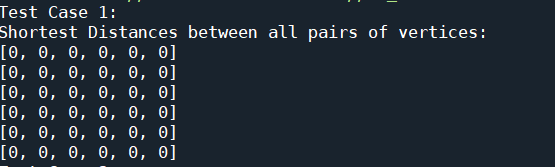
**print("Test Case 2:")**

**floyd\_warshall(graph2)**

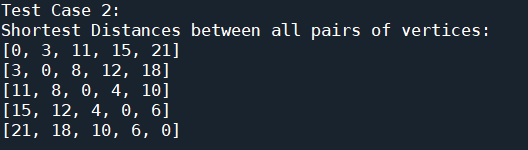
**Output :**

**Prints the matrix showing the shortest/quickest distances between every pair of vertices.**

Test Case 1:



Test Case 2:



**Breadth-First Search (BFS):**

**AIM:**

**The goal of the provided code is to execute a Breadth-First Search (BFS) traversal on a graph represented in the form of an adjacency list.**

**ALGORITHM:**

**Graph Representation:**

**- The graph is presented as an adjacency list, where each vertex serves as a key in a dictionary, and the corresponding value is a list containing its adjacent vertices.**

**Function: bfs(node)**

**Input:**

**- node: The initial node for the Breadth-First Search (BFS) traversal.**

**Output:**

**- Outputs the BFS traversal starting from the provided node.**

**Steps:**

**- Initialize a boolean array 'visited' to keep track of visited vertices.**

**- Initialize an empty queue.**

**- Mark the starting node as visited and enqueue it.**

**- While the queue is not empty:**

**- Dequeue a vertex v.**

**- Print v.**

**- Enqueue all adjacent vertices of v that have not been visited and mark them as visited.**

**Execution:**

**- The `bfs('A')` function is invoked with the starting node 'A'**.

**Code:**

**from collections import defaultdict, deque**

**class Graph:**

**def \_\_init\_\_(self):**

**self.graph = defaultdict(list)**

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

**self.graph[u].append(v)**

**def bfs(self, start\_node):**

**visited = set()**

**queue = deque([start\_node])**

**while queue:**

**current\_node = queue.popleft()**

**if current\_node not in visited:**

**print(current\_node, end=" ")**

**visited.add(current\_node)**

**for neighbor in self.graph[current\_node]:**

**if neighbor not in visited:**

**queue.append(neighbor)**

**# Test Case 1**

**g1 = Graph()**

**g1.add\_edge('A', 'B')**

**g1.add\_edge('A', 'C')**

**g1.add\_edge('B', 'D')**

**g1.add\_edge('B', 'E')**

**g1.add\_edge('C', 'F')**

**print("BFS traversal starting from node 'A' (Test Case 1):")**

**g1.bfs('A')**

**# Test Case 2**

**g2 = Graph()**

**g2.add\_edge('X', 'Y')**

**g2.add\_edge('X', 'Z')**

**g2.add\_edge('Y', 'W')**

**g2.add\_edge('Z', 'V')**

**g2.add\_edge('W', 'U')**

**print("\nBFS traversal starting from node 'X' (Test Case 2):")**

**g2.bfs('X')**

**Output :**

Test Case 1:



Case 2:



**Depth First Search:**

**AIM:**

**The purpose of the provided code is to execute Depth First Search (DFS) traversal on a graph, which is represented in the form of an adjacency list.**

**ALGORITHM:**

**Traversal Implementation:**

**- Create an initially empty set called "visited" to keep track of visited nodes.**

**- Define a graph using a dictionary, where keys represent nodes, and values consist of lists of neighbors.**

**Depth-First Search (DFS) Procedure:**

**- Check for Unvisited Node:**

**- If the current node "node" is not in the "visited" set:**

**- Visit the Node:**

**- Print the current node.**

**- Add the node to the "visited" set.**

**- Explore Neighbors:**

**- For each neighbor "neighbour" in the list of neighbors of the node in the graph:**

**- Recursively call the DFS function with the neighbor as the new node.**

**Driver Code:**

**- Display a message indicating the initiation of DFS.**

**- Invoke the DFS function with the "visited" set, the graph, and the starting node.**

**Code:**

**class Graph:**

**def \_\_init\_\_(self):**

**self.graph = {}**

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

**if u not in self.graph:**

**self.graph[u] = []**

**if v not in self.graph:**

**self.graph[v] = []**

**self.graph[u].append(v)**

**self.graph[v].append(u)**

**def dfs(self, start\_node, visited=None):**

**if visited is None:**

**visited = set()**

**if start\_node not in visited:**

**print(start\_node, end=" ")**

**visited.add(start\_node)**

**for neighbor in self.graph[start\_node]:**

**self.dfs(neighbor, visited)**

**# Test Case 1**

**g1 = Graph()**

**g1.add\_edge('A', 'B')**

**g1.add\_edge('A', 'C')**

**g1.add\_edge('B', 'D')**

**g1.add\_edge('B', 'E')**

**g1.add\_edge('C', 'F')**

**print("DFS traversal starting from node 'A' (Test Case 1):")**

**g1.dfs('A')**

**# Test Case 2**

**g2 = Graph()**

**g2.add\_edge('X', 'Y')**

**g2.add\_edge('X', 'Z')**

**g2.add\_edge('Y', 'W')**

**g2.add\_edge('Z', 'V')**

**g2.add\_edge('W', 'U')**

**print("\nDFS traversal starting from node 'X' (Test Case 2):")**

**g2.dfs('X')**

**Output :**

**TEST CASE1:**

****

**TEST CASE 2:**



**Prim's Algorithm :**

**AIM:**

**Prim's Algorithm is designed to discover the minimum spanning tree (MST) of a connected, undirected graph. This MST is a subset of edges that constitutes a tree incorporating every vertex while minimizing the total edge weight as much as possible.**

**AlGORITHM:**

* + **Initialization:**
  + **- INF: A significant constant representing infinity.**
  + **- V: Number of vertices in the graph.**
  + **- G: Adjacency matrix representing the weighted graph.**
  + **- selected: An array for tracking selected vertices in the MST.**
  + **- no\_edge: Counter to keep track of the number of edges selected in the MST.**
  + **Select Starting Vertex:**
  + **- Mark the initial vertex as selected (selected[0] = True).**
  + **Repeat Until MST is Formed:**
  + **- Identify the minimum-weight edge connecting a selected vertex to an unselected vertex.**
  + **- Print the selected edge and its weight.**
  + **- Mark the unselected vertex as selected.**
  + **- Increment the no\_edge counter.**
  + **Output MST:**
  + **- The selected edges and their weights collectively form the Minimum Spanning Tree..**

**Code:**

**class PrimMST:**

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

**self.V = vertices**

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

**def add\_edge(self, u, v, weight):**

**self.graph[u][v] = weight**

**self.graph[v][u] = weight**

**def prim\_mst(self):**

**INF = float('inf')**

**selected = [False] \* self.V**

**# Initialize the key values with infinity**

**key = [INF] \* self.V**

**key[0] = 0 # Start with the first vertex**

**for \_ in range(self.V):**

**# Find the minimum key value vertex from the set of vertices not yet included in MST**

**u = self.minimum\_key(key, selected)**

**selected[u] = True**

**# Update key values of the adjacent vertices**

**for v in range(self.V):**

**if self.graph[u][v] > 0 and not selected[v] and key[v] > self.graph[u][v]:**

**key[v] = self.graph[u][v]**

**self.print\_mst(key)**

**def minimum\_key(self, key, selected):**

**INF = float('inf')**

**min\_key = INF**

**min\_index = -1**

**for v in range(self.V):**

**if key[v] < min\_key and not selected[v]:**

**min\_key = key[v]**

**min\_index = v**

**return min\_index**

**def print\_mst(self, key):**

**print("Edge \tWeight")**

**for i in range(1, self.V):**

**print(f"{i} - {key[i]}")**

**# Test Case 1**

**g1 = PrimMST(5)**

**g1.add\_edge(0, 1, 2)**

**g1.add\_edge(0, 2, 3)**

**g1.add\_edge(1, 2, 1)**

**g1.add\_edge(1, 3, 1)**

**g1.add\_edge(2, 4, 4)**

**print("Minimum Spanning Tree (Test Case 1):")**

**g1.prim\_mst()**

**# Test Case 2**

**g2 = PrimMST(4)**

**g2.add\_edge(0, 1, 1)**

**g2.add\_edge(1, 2, 3)**

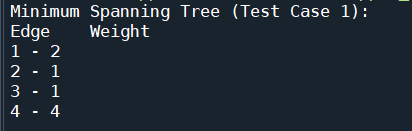
**g2.add\_edge(2, 3, 4)**

**print("\nMinimum Spanning Tree (Test Case 2):")**

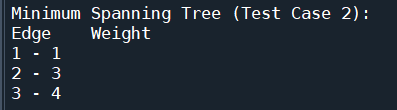
**g2.prim\_mst()**

**Output :**

**Test Case 1:**



**Test Case 2:**

****

**Kruskal's algorithm:**

**AIM:**

**Kruskal's Algorithm, employed to identify the minimum spanning tree in a connected, undirected graph, adopts a distinct methodology. The approach involves sorting all edges in ascending order based on their weights and subsequently incorporating them into the Minimum Spanning Tree (MST) only if they do not contribute to the formation of a cycle**

**ALGORITHM:**

**1. Arrange the edges in ascending order according to their weights.**

**2. Begin with an empty graph.**

**3. Include the smallest edge in the graph, ensuring it does not create a cycle.**

**4. Iterate through step 3 until the graph encompasses all vertices.**

**Code:**

class KruskalMST:

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

self.V = vertices

self.graph = []

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

self.graph.append((u, v, weight))

def kruskal\_mst(self):

self.graph.sort(key=lambda edge: edge[2]) # Sort edges in ascending order of weight

parent = [i for i in range(self.V)]

result = []

def find\_set(node):

if parent[node] == node:

return node

parent[node] = find\_set(parent[node]) # Path compression

return parent[node]

def union\_sets(u, v):

root\_u = find\_set(u)

root\_v = find\_set(v)

parent[root\_u] = root\_v

for edge in self.graph:

u, v, weight = edge

if find\_set(u) != find\_set(v):

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

union\_sets(u, v)

self.print\_mst(result)

def print\_mst(self, result):

print("Edge \tWeight")

for edge in result:

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

# Test Case 1

g1 = KruskalMST(5)

g1.add\_edge(0, 1, 2)

g1.add\_edge(0, 2, 3)

g1.add\_edge(1, 2, 1)

g1.add\_edge(1, 3, 1)

g1.add\_edge(2, 4, 4)

print("Minimum Spanning Tree (Test Case 1):")

g1.kruskal\_mst()

# Test Case 2

g2 = KruskalMST(4)

g2.add\_edge(0, 1, 1)

g2.add\_edge(1, 2, 3)

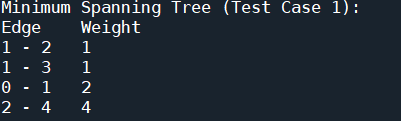
g2.add\_edge(2, 3, 4)

print("\nMinimum Spanning Tree (Test Case 2):")

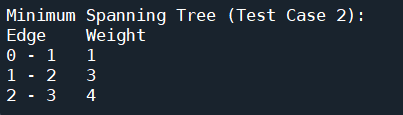
g2.kruskal\_mst()

**Output :**

**TEST CASE 1:**



**TEST CASE 2:**



**Dynamic Programming Algorithm:**

**AIM:**

**The objective is to determine the shortest paths from a designated source vertex to all other vertices in a weighted graph. This method accommodates graphs with edges bearing negative weights, but it is capable of identifying the presence of negative cycles.**

**AlGORITHM:**

**Initialization:**

**- Initialize arrays for distances and predecessors.**

**- Set the distance to the source vertex as 0, and all other distances as infinity.**

**- Assign None to predecessors for all vertices.**

**Relax Edges Repeatedly:**

**- Iterate the relaxation step V-1 times, where V is the number of vertices.**

**- For each edge (u, v) in the graph, update the distance and predecessor if the distance to u plus the weight of the edge to v is less than the current distance to v.**

**Check for Negative Cycles:**

**- After V-1 iterations, reevaluate all edges.**

**- If further relaxation occurs, it indicates the presence of a negative cycle in the graph.**

**Result:**

**- The final arrays for distances and predecessors store the shortest paths from the source vertex.**

**Code:**

class BellmanFord:

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

self.V = vertices

self.graph = []

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

self.graph.append((u, v, weight))

def bellman\_ford(self, source):

# Step 1: Initialize distances and predecessors

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

predecessors = [None] \* self.V

distances[source] = 0

# Step 2: Relax edges repeatedly

for \_ in range(self.V - 1):

for u, v, weight in self.graph:

if distances[u] + weight < distances[v]:

distances[v] = distances[u] + weight

predecessors[v] = u

# Step 3: Check for negative cycles

for u, v, weight in self.graph:

if distances[u] + weight < distances[v]:

print("Graph contains a negative cycle")

return

# Step 4: Print the result

self.print\_result(distances, predecessors)

def print\_result(self, distances, predecessors):

print("Vertex\tDistance\tPredecessor")

for i in range(self.V):

print(f"{i}\t{distances[i]}\t\t{predecessors[i]}")

# Test Case 1

g1 = BellmanFord(5)

g1.add\_edge(0, 1, 2)

g1.add\_edge(0, 2, 3)

g1.add\_edge(1, 2, 1)

g1.add\_edge(1, 3, 1)

g1.add\_edge(2, 4, 4)

print("Shortest Paths from Source 0 (Test Case 1):")

g1.bellman\_ford(0)

# Test Case 2

g2 = BellmanFord(4)

g2.add\_edge(0, 1, 1)

g2.add\_edge(1, 2, 3)

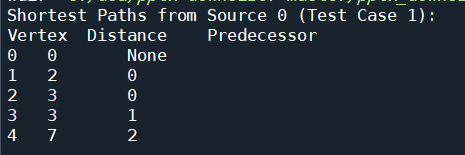
g2.add\_edge(2, 3, 4)

print("\nShortest Paths from Source 0 (Test Case 2):")

g2.bellman\_ford(0)

**Output and Test Cases:**

**Case 1:**



**Case 2:**

