23CCE201 Data Structures

Name : E Eeshwar

Roll No : CB.EN.U4CCE23012

AIM :

- To implement the various shortest path problems and solutions in weighted directed graphs.

LOGIC :

- To implement shortest path problems and solutions in weighted directed graphs, there are three main methods: Dijkstra's Algorithm, Dynamic Programming Algorithm and Floyd's Algorithm.
- **In Dijkstra's Algorithm**, each vertex is added to a set. While being added, the weight of the edge between the source vertex and the weight calculated through shortest path and the shortest weight is updated. Through this method, the shortest distances between the source vertex and the other vertices are calculated, allowing us to find the shortest path from a source vertex to all other vertices.
- **In Dynamic Programming Algorithm**, the distance from each vertex towards a destination vertex is found. This is done by reversing the graph edges and treat the destination vertex as the source vertex. Then the process of Dijkstra's Algorithm is followed, where the distance between the vertices from the selected vertex is updated by comparing the edges and the traversed weight between the vertices.
- **In Floyd's Algorithm**, the shortest distances between all the vertices are found. In this process, a matrix is created in which the distances are stored according to the vertices. Then, the distances are compared by checking between the direct distance and the distance between the two selected vertices via an intermediate vertex, and according to the calculation, the minimum distance will be updated.

ALGORITHM :

- **Dijkstra's Algorithm :**

    1. Get input from the user for the number of vertices, information about the vertices, weights and its neighbours, and store it in a dictionary.
    2. Initialize all the distances as infinity and set the distance from source vertex towards itself as 0.
    3. Add vertex to the visited set whenever a vertex has been processed.
    4. For each vertex's neighbours, calculate the minimum value between tentative distance and the current recorded distance, and update the value accordingly.
    5. Repeat this process until all the available vertices have been visited.
    6. Check whether any of the vertices are out of reach. If true, then keep the initialized value as it is.
    7. Return the distances from all vertices from the source vertex.

- **Dynamic Programming Algorithm :**

    1. Get input from the user for the number of vertices, information about the vertices, weights and its neighbours, and store it in a dictionary.
    2. Reverse the whole graph, changing the directions of all the directed edges towards its opposite direction.
    3. Initialize bellman_ford function and initialize all distances as infinity. Set the distance to the destination vertex as 0.
    4. Check for a shorter path via a neighbouring vertex and compare it with the available distance. If there exists a shorter path, then update the distance between those two vertices.
    5. Check whether any of the vertices are out of reach. If true, then keep the initialized value as it is.
    6. Return the distances from all vertices towards the destination vertex.

- **Floyd's Algorithm :**

    1. Get input from the user for the number of vertices, information about the vertices, weights and its neighbours, and store it in a 2D dictionary.
    2. Initialize all distances to infinity and set the distance of each vertex to itself as 0.
    3. Initialize floyd function, where the code iterates over all the vertices, checking for a shorter path via an intermediate vertex.
    4. If there exists a shorter path, replace the old distance with the new distance.
    5. Check whether any of the vertices are out of reach. If true, then keep the initialized value as it is.
    6. Return the distances from all vertices towards all other vertices.

CODE :

- **Dijkstra's Method :**

```
def dijkstra(graph,source):
    shortest = {vertex:float('inf') for vertex in graph}
    shortest[source] = 0
    visited = set()
    while len(visited)<len(graph):
        curr = None
        curr_distance = float('inf')
        for vertex in graph:
            if vertex not in visited and shortest[vertex]<curr_distance:
                curr = vertex
                curr_distance = shortest[vertex]
        if curr==None:
            break
        visited.add(curr)
        for neighbour,weight in graph[curr]:
            distance = curr_distance+weight
            if neighbour not in visited and shortest[neighbour]>distance:
                shortest[neighbour] = distance
```

```
    return shortest

def user_input():
    graph={}
    n = int(input("Enter the number of vertices: "))
    for i in range(n):
        vertex = input("\nEnter the vertex: ")
        neighbours = []
        m = int(input("Enter the number of edges: "))
        for i in range(m):
            neighbour = input(f"Enter the neighbour for vertex {vertex}:")
            weight = int(input(f"Enter the weight for the edge {vertex} ->
{neighbour}: "))
            neighbours.append((neighbour,weight))
        graph[vertex]=neighbours
    return graph

def display_graph(graph):
    print("\nGraph:")
    for vertex,neighbours in graph.items():
        for neighbour,weight in neighbours:
            print(f"{vertex} -> {neighbour} (Weight: {weight})")



def unreachable_vertex(shortest,source):
    count = 0
    for vertex,distance in shortest.items():
        if distance == float('inf'):
            print(f"\nThe vertex {vertex} is unreachable from the source
vertex {source}.")
            count = 1
    if count==0:
        print(f"\nThere is no such vertex that is unreachable from the
source {source}")

graph = user_input()
display_graph(graph)
```

```
source = input("\nEnter the source vertex : ")
shortest = dijkstra(graph, source)
print("\nShortest distance from the source: ",source," ->",shortest)
unreachable_vertex(shortest, source)
```

- **Dynamic Programming Method :**

```
from collections import defaultdict
def bellman_ford(graph, vertices, destination):
  distances = {vertex: float('inf') for vertex in vertices}
  distances[destination] = 0
  for _ in range(len(vertices) - 1):
    for vertex in graph:
      for neighbour, weight in graph[vertex]:
        if distances[vertex] != float('inf') and distances[vertex] +
weight < distances[neighbour]:
          distances[neighbour] = distances[vertex] + weight
  for vertex in vertices:
    if distances[vertex] == float('inf'):
      print(f"The vertex {vertex} is unreachable from the destination
vertex {destination}.")
  return distances

def reverse_graph(graph):
  reversed_graph = defaultdict(list)
  for vertex, neighbours in graph.items():
    for neighbour, weight in neighbours:
      reversed_graph[neighbour].append((vertex, weight))
  return reversed_graph

def user_input():
  graph = defaultdict(list)
  vertices = set()
  n = int(input("Enter the number of vertices: "))
  for _ in range(n):
    vertex = input("Enter the vertex: ")
    vertices.add(vertex)
    m = int(input(f"Enter the number of edges for vertex {vertex}: "))
```

```
        for _ in range(m):
            neighbour = input(f"Enter the neighbour for vertex {vertex}: ")
            weight = int(input(f"Enter the weight for the edge {vertex} ->
{neighbour}: "))
            graph[vertex].append((neighbour, weight))
            vertices.add(neighbour)
    return graph, vertices

def display_distances(distances, destination):
    print(f"\nShortest distances to the destination ({destination}):")
    for vertex, distance in distances.items():
        if distance == float('inf'):
            print(f"Vertex {vertex} is unreachable.")
        else:
            print(f"Distance to vertex {vertex}: {distance}")

graph, vertices = user_input()
reversed_graph = reverse_graph(graph)
destination = input("\nEnter the destination vertex: ")
distances = bellman_ford(reversed_graph, vertices, destination)
display_distances(distances, destination)
```

- **Floyd's Method :**

```
def floyd_warshall(graph, vertices):
    dist = {v: {u: float('inf') for u in vertices} for v in vertices}
    for v in vertices:
        dist[v][v] = 0
    for u in graph:
        for v, weight in graph[u]:
            dist[u][v] = weight
    for k in vertices:
        for i in vertices:
            for j in vertices:
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
    for v in vertices:
        if dist[v][v] < 0:
```

```
            print("Graph contains a negative weight cycle.")
            return None
    return dist

def user_input():
    graph = {}
    vertices = set()
    n = int(input("Enter the number of vertices: "))
    for _ in range(n):
        vertex = input("Enter the vertex: ")
        vertices.add(vertex)
        m = int(input(f"Enter the number of edges for vertex {vertex}: "))
        neighbours = []
        for _ in range(m):
            neighbour = input(f"Enter the neighbour for vertex {vertex}: ")
            weight = int(input(f"Enter the weight for the edge {vertex} ->
{neighbour}: "))
            neighbours.append((neighbour, weight))
            vertices.add(neighbour)
        graph[vertex] = neighbours
    return graph, vertices

def display_distances(dist, vertices):
    print("\nShortest distances between all pairs of vertices:")
    for i in vertices:
        row = [f"{dist[i][j]:>4}" if dist[i][j] != float('inf') else " inf" for j in
vertices]
        print(row)

graph, vertices = user_input()
dist = floyd_warshall(graph, vertices)
if dist:
    display_distances(dist, vertices)
```

RESULTS :

- **Dijkstra's Method :**

```
Enter the number of vertices: 5

Enter the vertex: A
Enter the number of edges: 3
Enter the neighbour for vertex A:B
Enter the weight for the edge A -> B: 10
Enter the neighbour for vertex A:D
Enter the weight for the edge A -> D: 30
Enter the neighbour for vertex A:E
Enter the weight for the edge A -> E: 100

Enter the vertex: B
Enter the number of edges: 1

Enter the vertex: C
Enter the number of edges: 1
Enter the neighbour for vertex C:E
Enter the weight for the edge C -> E: 10

Enter the vertex: D
Enter the number of edges: 2
Enter the neighbour for vertex D:C
Enter the weight for the edge D -> C: 20
Enter the neighbour for vertex D:E
Enter the weight for the edge D -> E: 60

Enter the vertex: E
Enter the number of edges: 0
```

```
Graph:
A -> B (Weight: 10)
A -> D (Weight: 30)
A -> E (Weight: 100)
B -> C (Weight: 50)
C -> E (Weight: 10)
D -> C (Weight: 20)
D -> E (Weight: 60)

Enter the source node: A

Shortest distance from the source:  A  -> {'A': 0, 'B': 10, 'C': 50, 'D': 30, 'E': 60}

There is no such vertex that is unreachable from the source A
```

- **Dynamic Programming Method :**

```
Enter the number of vertices: 6
Enter the vertex: 2
Enter the number of edges for vertex 2: 2
Enter the neighbour for vertex 2: 1
Enter the weight for the edge 2 -> 1: 6
Enter the neighbour for vertex 2: 3
Enter the weight for the edge 2 -> 3: -2
Enter the vertex: 3
Enter the number of edges for vertex 3: 2
Enter the neighbour for vertex 3: 1
Enter the weight for the edge 3 -> 1: 4
Enter the neighbour for vertex 3: 4
Enter the weight for the edge 3 -> 4: -2
Enter the vertex: 4
Enter the number of edges for vertex 4: 1
Enter the neighbour for vertex 4: 1
Enter the weight for the edge 4 -> 1: 5
Enter the vertex: 5
Enter the number of edges for vertex 5: 2
Enter the neighbour for vertex 5: 2
Enter the weight for the edge 5 -> 2: -1
Enter the neighbour for vertex 5: 3
Enter the weight for the edge 5 -> 3: 3
Enter the vertex: 6
Enter the number of edges for vertex 6: 2
Enter the neighbour for vertex 6: 4
Enter the weight for the edge 6 -> 4: -1
Enter the neighbour for vertex 6: 5
Enter the weight for the edge 6 -> 5: 3
Enter the vertex: 1
Enter the number of edges for vertex 1: 0
```

```
Enter the destination vertex: 1

Shortest distances to the destination (1):
Distance to vertex 1: 0
Distance to vertex 3: 3
Distance to vertex 2: 1
Distance to vertex 4: 5
Distance to vertex 5: 0
Distance to vertex 6: 3
```

- **Floyd's Method :**

```
Enter the number of vertices: 4
Enter the vertex: A
Enter the number of edges for vertex A: 2
Enter the neighbour for vertex A: B
Enter the weight for the edge A -> B: 3
Enter the neighbour for vertex A: C
Enter the weight for the edge A -> C: 10
Enter the vertex: B
Enter the number of edges for vertex B: 2
Enter the neighbour for vertex B: C
Enter the weight for the edge B -> C: 2
Enter the neighbour for vertex B: D
Enter the weight for the edge B -> D: 7
Enter the vertex: C
Enter the number of edges for vertex C: 1
Enter the neighbour for vertex C: D
Enter the weight for the edge C -> D: 1
Enter the vertex: D
Enter the number of edges for vertex D: 0

Shortest distances between all pairs of vertices:
['   0', '   3', '   5', '   6']
[' inf', '   0', '   2', '   3']
[' inf', ' inf', '   0', '   1']
[' inf', ' inf', ' inf', '   0']
```

INFERENCES :

- The three methods of finding the shortest path between vertices of a graph offers a practical solution to shortest path problems. Dijkstra's method allows us to calculate the shortest path from a single source vertex to all the other available vertices of the graph. Dynamic Programming method allows us to find the shortest path from all available vertices of a graph to a single destination vertex. In Floyd's method, we are able to find the shortest distances between all available vertices in the graph.