23CCE201 - Data Structures

Name: Anirudh Anand

Roll No.: CB.EN.U4CCE23002

Directed Graphs – Shortest Path Problems

I. Aim:

To provide various solutions to the shortest path problems with respect to the Weighted Directed Graph Data Structure.

II. Logic:

In this report we will be using three different approaches to solve the shortest path problem on weighted directed graphs; namely the Dijkstra's algorithm, Floyd-Warshall algorithm and dynamic programming method(Belman-Ford algorithm.)

1. **Dijkstra's Algorithm**:
   It follows a greedy approach with a priority queue, wherein we start with the source node marking distance to itself as 0 and other nodes as infinity. It proceeds to then repeatedly select nodes with the smallest known distance from the priority queue and marks them as 'visited', updates the shortest distances to its neighbours. This process continues till all the nodes are processed or all reachable nodes have shortest distance calculated. The drawback of this method is that it doesn't work effectively with negative weights as it may lead to revisiting of nodes.
2. **Floyd-Warshall Algorithm**:
   It uses dynamic programming method and obtains the shortest distances for all the pairs in the graph. In this method, we consider a matrix (i,j) where each cell represents the direct edge weight from node i to j, with the weight as the value if edge exists or infinity if no direct edge exists. This matrix is iteratively updated by considering each node as an intermediate node. Meaning, we check if we go from node A to node B, we check if it is easier to go through the node C. By the end of this process, we will obtain the shortest paths from all nodes to all other nodes(for all pairs).
3. **Bellman-Ford Algorithm**:
   It uses dynamic programming method and also follows 'edge relaxation' method. In this method we set the distance from the source to all vertices and to itself set as 0. For each vertex, it "relaxes" every edge by checking if a shorter path to a destination vertex can be achieved by traveling through a different source vertex. Here, relaxation means for a given edge connecting two nodes u and v with weight w, we check if the current known shortest distance to v can be minimized by going through u first. If it can, we update (or "relax") the shortest path distance to v to reflect this improved, shorter path

via u. This process is repeated for n-1 times where n is the number of vertices. One additional iteration is also carried out to ensure there are no negative cycles.

III. Algorithm:

- **Dijkstra's Algorithm**:
  1. Initialize distances, setting starting node as 0 and to all other nodes as infinity
  2. Initialize a visited set in which all the visited notes are marked and updated into
  3. From the unvisited nodes select the one with the shortest distance from starting node
  4. Now for each neighbouring node of the selected node:-
     - Check if the distance to the neighbour can be shortened by travelling through the selected node
     - If yes, update the shortest distance to the neighbour node
  5. After checking all the neighbour nodes of the selected nodes, mark it as visited
  6. Repeat from step 3 until all nodes are visited
  7. End

- **Floyd-Warshall Algorithm**:
  1. Initialise a distance matrix where each cell (i,j) represents the shortest known distances from node i to node j:
     a. Set (i,i) elements to 0 as the distance from any node to itself is 0
     b. For each edge from u to v with weight w, set u,v as w
     c. Set all other distances to infinity as there is no direct edge between the nodes
  2. Iterate through each node k as a possible intermediate node
  3. For each of the pair of node (i,j):
     a. Check if the distance from i to j can be shortened by travelling through k
     b. If yes, update the distance from i to j, now travelling through k
  4. Repeat from step 3 for all pairs (i,j) with each intermediate node k until all possibilities are covered
  5. End

- **Bellman-Ford Algorithm**:
  1. Initialise distances setting the source nodes distance to 0 and for all other nodes to infinity
  2. Repeat this step n-1 where n is the number of vertices:
     a. For each edge from node u to node v with weight w:
        i. Check if the distance to v can be shortened by going through u
        ii. If yes, update the distance to v
  3. Check for a negative cycle by repeating the relaxation step once:
     a. For each edge from u to v with weight w:

i. If the distance to v can still be reduced by going through u, there is a negative cycle in the graph.

4. End

IV. Code:

```
class Graph:
    def __init__(self, vertices):
        self.V = vertices  # Number of vertices
        self.graph = []    # Adjacency list to hold the graph

    def add_edge(self, u, v, w):
        """Add an edge to the graph."""
        self.graph.append((u, v, w))

    def dijkstra(self, src):
        """Implement Dijkstra's algorithm."""
        dist = [float("inf")] * self.V
        dist[src] = 0
        visited = [False] * self.V

        for _ in range(self.V):
            min_dist = float("inf")
            min_index = 0

            for v in range(self.V):
                if not visited[v] and dist[v] < min_dist:
                    min_dist = dist[v]
                    min_index = v

            visited[min_index] = True
```

```
            for u, v, w in self.graph:
                if u == min_index and not visited[v] and dist[min_index] + w < dist[v]:
                    dist[v] = dist[min_index] + w

        return dist

    def floyd_warshall(self):
        """Implement Floyd-Warshall algorithm."""
        dist = [[float("inf")] * self.V for _ in range(self.V)]

        for u, v, w in self.graph:
            dist[u][v] = w

        for i in range(self.V):
            dist[i][i] = 0

        for k in range(self.V):
            for i in range(self.V):
                for j in range(self.V):
                    if dist[i][j] > dist[i][k] + dist[k][j]:
                        dist[i][j] = dist[i][k] + dist[k][j]

        return dist

    def bellman_ford(self, src):
        """Implement Bellman-Ford algorithm."""
        dist = [float("inf")] * self.V
        dist[src] = 0
```

```
        for _ in range(self.V - 1):
            for u, v, w in self.graph:
                if dist[u] != float("inf") and dist[u] + w < dist[v]:
                    dist[v] = dist[u] + w
        # Check for negative-weight cycles
        for u, v, w in self.graph:
            if dist[u] != float("inf") and dist[u] + w < dist[v]:
                raise ValueError("Graph contains a negative-weight cycle")
        return dist
    def display_distances(self, distances):
        """Display the result distances in a readable format."""
        print("Vertex distances:")
        for i, distance in enumerate(distances):
            print(f"Distance from source to vertex {i}: {distance}")
    def display_floyd_matrix(self, matrix):
        """Display the Floyd-Warshall distance matrix."""
        print("Distance matrix (Floyd-Warshall):")
        for row in matrix:
            print(" ".join(f"{d:7}" if d != float("inf") else "" for d in row))
print("The number of vertices is 5\n")
g = Graph(5)
g.add_edge(0, 1, 10)
g.add_edge(0, 2, 3)
g.add_edge(1, 2, 1)
g.add_edge(1, 3, 2)
g.add_edge(2, 1, 4)
g.add_edge(2, 3, 8)
g.add_edge(2, 4, 2)
g.add_edge(3, 4, 7)
g.add_edge(4, 3, 9)
```

```
source_vertex=int(input("Enter the source vertex: "))
distances_dijkstra = g.dijkstra(source_vertex)
g.display_distances(distances_dijkstra)
floyd_matrix = g.floyd_warshall()
g.display_floyd_matrix(floyd_matrix)
distances_bellman_ford = g.bellman_ford(source_vertex)
g.display_distances(distances_bellman_ford)
```

V. Results:

```
Enter the number of vertices: 5
Enter the source vertex: 2
Vertex distances:
Distance from source to vertex 0: inf
Distance from source to vertex 1: 4
Distance from source to vertex 2: 0
Distance from source to vertex 3: 6
Distance from source to vertex 4: 2
Distance matrix (Floyd-Warshall):
      0       7       3       9       5
              0       1       2       3
              4       0       6       2
                              0       7
                              9       0
Vertex distances:
Distance from source to vertex 0: inf
Distance from source to vertex 1: 4
Distance from source to vertex 2: 0
Distance from source to vertex 3: 6
Distance from source to vertex 4: 2
```

VI. Inference:

Thus, for given weighted directed graph of n vertices, and a source vertex, we have obtained the shortest distances from the source vertex to every other vertex in the graph using both Dijkstra's algorithm and Bellman-Ford algorithm and have obtained the same output. Using the Floyd-Warshall Algorithm we have also obtained the shortest distance for all pairs of nodes which are possible to reach. Upon checking negative weights, if the graph has only positive weights, both Dijkstra's and Bellman-Ford outputs for the shortest path from a single source will be the same. If the graph has negative weights but no negative cycles, Bellman-Ford will correctly handle it, whereas Dijkstra's might give incorrect results. Bellman-Ford is the only algorithm here that detects negative-weight cycles. If a ValueError is raised, it indicates that the graph contains a negative cycle reachable from the source vertex.