23CCE201 DATA STRUCTURES

ASSIGNMENT

NAME: VETHAVIYASH.K **ROLL.NO:** CB.EN.U4CCE23063

Report on Shortest Path Algorithms on Weighted Directed Graphs

---

1. Aim

To implement and analyze shortest path algorithms for directed graphs with non-negative weights, calculating minimum path costs for:

1. A designated **source** to each vertex (Dijkstra's Algorithm)
2. Each vertex to a **single destination** (Dynamic Programming)
3. **All pairs** of vertices (Floyd's Algorithm)

Given a directed graph G=(V,E) with non-negative edge costs, the goal is to find the shortest path costs, where the path cost is the total of the edge costs along that path.

---

2. Logic Followed

- **Dijkstra's Algorithm**: This algorithm is used for finding the shortest paths from a starting vertex to all other vertices in a graph with non-negative weights. It uses a priority queue to continuously select the vertex with the smallest tentative distance, updating distances to its neighbors.
- **Dynamic Programming Approach**: For specific types of shortest path problems, such as finding the shortest path in acyclic graphs, dynamic programming provides an efficient solution by breaking the problem down into simpler sub problems.
- **Floyd-War shall Algorithm**: This algorithm calculates shortest paths between all pairs of vertices, making it suitable for dense graphs or when multiple shortest paths between pairs are needed. It iteratively updates the shortest paths by considering each vertex as an intermediate point.

---

3. Algorithm

Notations Used in the Algorithms

- **Q** → A priority queue used to manage vertices based on distance.
- **D[]** → An array that stores the shortest distance from the source vertex to each vertex.
- **J[]** → An array that holds the shortest path costs from each vertex to the destination vertex.
- **A[n][n]** → A matrix representing the shortest path costs between all pairs of vertices.
- **u** → The current vertex being processed, extracted from the priority queue.
- **v** → A neighboring vertex of u or the current vertex being processed in the topological order.
- **w** → A neighboring vertex that can be reached directly from vertex v.
- **i** → An index representing the source vertex in the pair (i, j).
- **j** → An index representing the destination vertex in the pair (i, j).
- **k** → An index representing an intermediate vertex being considered in the path from i to j.
- **C(u, v)** → The cost (or weight) of the edge connecting vertex u to vertex v.
- **C(v, w)** → The cost (or weight) of the directed edge from vertex v to vertex w.
- **C(i, j)** → The cost (or weight) of the direct edge from vertex i to vertex j, if it exists.
- **alt** → The alternative distance calculated from the source to vertex v through vertex u.
- **z** → The destination vertex to which the shortest path is being calculated.

Dijkstra's Algorithm (Single Source Shortest Path)

1. Initialize a priority queue Q to manage vertices based on distance.
2. Create a distance array D[] to store the shortest distance from the source to each vertex.
3. Set all distances in D[] to infinity, except for the source vertex, which is set to 0.
4. Add all vertices to the priority queue Q.
5. While Q is not empty:
   - Extract the vertex u with the smallest distance from D[].
   - For each neighbor v of u:
     - Calculate the alternative distance alt = D[u] + C(u, v).
     - If alt is less than D[v]:
       - Update D[v] to alt.
       - Adjust the priority of v in Q to reflect the new distance.
6. Return the distance array D[] containing the shortest distances from the source to all vertices.

Dynamic Programming Algorithm (Shortest Path to a Destination in a DAG)

1. Initialize an array J[] to hold the shortest path costs from each vertex to the destination.
2. Set all values in J[] to infinity, except for the destination vertex, which is set to 0.
3. Perform a topological sort of the vertices in the directed acyclic graph (DAG).
4. Process each vertex v in the sorted order:
    - For each directed edge (v, w):
        - Update J[v] as the minimum of J[v] and C(v, w) + J[w].
5. Return the array J[], which contains the minimum costs to reach the destination from each vertex.

Floyd's Algorithm (All-Pairs Shortest Path)

1. Initialize a matrix A[n][n] to represent shortest path costs between all pairs of vertices.
2. Set A[i][j] to 0 if i equals j; if there is a direct edge from i to j, set A[i][j] to C(i, j); otherwise, set A[i][j] to infinity.
3. Perform n iterations over the matrix to consider each vertex k as an intermediate vertex.
4. For each pair of vertices (i, j):
    - Update A[i][j] as the minimum of A[i][j] and A[i][k] + A[k][j] to check if the path through k is shorter.
5. Return the matrix A, which contains the shortest path lengths for all vertex pairs.

4. Codes

Dijkstra’s Algorithm (Python Code)

```
import heapq

def dijkstra(graph, start):
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    priority_queue = [(0, start)]

    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)

        if current_distance > distances[current_node]:
            continue

        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances
```

Dynamic Programming for Shortest Paths in Acyclic Graphs

```
def shortest_path_dag(graph, start):
    topological_order = topological_sort(graph)
    distances = {node: float('inf') for node in graph}
    distances[start] = 0

    for node in topological_order:
        if distances[node] != float('inf'):
            for neighbor, weight in graph[node].items():
                if distances[node] + weight < distances[neighbor]:
                    distances[neighbor] = distances[node] + weight

    return distances
```

Floyd-Warshall Algorithm

```
def floyd_warshall(graph):
    num_vertices = len(graph)
    distance = [[float('inf')] * num_vertices for _ in range(num_vertices)]

    for u in range(num_vertices):
        distance[u][u] = 0
        for v, w in graph[u].items():
```

```
        distance[u][v] = w

for k in range(num_vertices):
    for i in range(num_vertices):
        for j in range(num_vertices):
            if distance[i][j] > distance[i][k] + distance[k][j]:
                distance[i][j] = distance[i][k] + distance[k][j]

return distance
```

---

5. Results:

Dijkstra’s Algorithm:

- **Output**:
  - An array D[] containing the shortest distances from the starting vertex to each vertex in the graph.
  - A predecessor or parent array to reconstruct the shortest paths.

Dynamic Programming (for Acyclic Graphs):

- **Output**:
  - An array J[] that holds the shortest path costs from each vertex to a specified destination vertex.
  - The shortest path can be reconstructed using backtracking with additional predecessor information if stored.

Floyd-War shall Algorithm:

- **Output**:
  - A matrix A[n][n] where each entry A[i][j] represents the shortest path cost from vertex iii to vertex j.
  - The ability to reconstruct the shortest paths can be enhanced by maintaining a predecessor matrix.

---

6. **Inference**

- **Dijkstra's Algorithm** is very efficient for graphs with fewer connections, solving single-source shortest path problems with non-negative weights. It provides a straightforward list of distances and paths from the starting point to all other points.
- **Dynamic Programming** works best for finding the shortest paths in graphs without cycles (acyclic graphs). It uses a specific order (topological) to process vertices, making it fast and accurate based on the number of connections and points.
- **Floyd-War shall Algorithm** is ideal for finding the shortest paths between all pairs of points in highly connected graphs. Although it requires more computation, it has an advantage over Dijkstra's Algorithm for solving all-pairs shortest path problems, as it provides a complete table of shortest path costs between every pair of points.