K. J. Somaiya College of Engineering, Mumbai-77
(A Constituent College of Somaiya Vidyavihar University)

Roll No.: 16010123011 and **16010123012**
Name of the student: Aaryan Dubey and **Aaryan Sharma**
Div: A
Branch: Computer Engineering
IA No: IA2
Date: 13 / 10 / 24
Subject: Discrete Mathematics

TITLE: Finding shortest path in Weighted Graphs using Dijkstra's Algorithm
AIM: To implement a program that implements Dijkstra's algorithm to efficiently find the shortest path from a given source(start) node to all other nodes in a weighted, non-negative graph

Literature survey/Theory: Dijkstra's algorithm works by progressively selecting the shortest path from the starting node to each other node in the graph. It uses a priority queue (commonly implemented as a min-heap) to explore nodes in the order of their current known distances from the source node. The algorithm ensures that once a node's shortest distance is known, it is not revisited, thus making the solution efficient for graphs without negative edge weights.

Key Concepts

1. **Graph Representation**:
   The graph is typically represented as a set of vertices (nodes) connected by edges (paths). Each edge has a non-negative weight (cost) representing the distance between the two connected nodes.
2. **Single Source Shortest Path**:
   The goal is to determine the shortest distance from a given source vertex to all other vertices in the graph.
3. **Priority Queue (Min-Heap)**:
   Dijkstra's algorithm relies on a priority queue to select the next node with the smallest distance. This queue ensures that the algorithm explores nodes with the shortest known distance first, avoiding unnecessary calculations.
4. **Relaxation**:
   The process of updating the shortest distance to a neighboring node is known as relaxation. If the newly calculated distance is smaller than the previously known distance, the algorithm updates the distance.

Limitations

1. **No Negative Weights**:
   Dijkstra's algorithm fails if the graph contains negative-weight edges, as it assumes that once a node's shortest distance is determined, it will not change. For graphs with negative weights, the Bellman-Ford algorithm is preferred.
2. **Dense Graphs**:
   For graphs with a large number of edges, the performance of Dijkstra's algorithm can degrade since the algorithm may need to explore many edges even if they do not contribute to the shortest path.

Mathematical Concept:

Given a graph G = (V,E)G = (V,E)G=(V,E) with a set of vertices V and edges E, the task is to find the shortest path from a source vertex s to all other vertices. Let:

- d(u) represents the shortest distance from s to u,
- w(u,v) represent the weight of the edge between vertex u and vertex v.

The relaxation step of the algorithm ensures that:

d(v)=min(d(v), d(u) + w(u,v))

Where u is the current node, and v is a neighbor of u.

Algorithm:

Step 1: Input the Graph

1. Input number of vertices (num_vertices)
2. For each vertex v_i:
   - Input the name of the vertex v_i., the number of edges originating from this vertex.
     For each edge:
     - Input the neighbor vertex neighbor_j that this edge connects to. Input the weight of the edge. Ensure that the weight is non-negative.
     - If a negative weight is entered, show an error message and prompt the user to re-enter a valid weight.

- Store the vertex and its corresponding neighbors with their weights in an adjacency list format (dictionary of lists).

Step 2: Dijkstra's Algorithm to Find Shortest Paths

1. Initialize shortest path distances:
   - Create a dictionary shortest_paths where each vertex has a default distance of infinity (∞).
   - Set the distance of the start vertex to 0.
2. Initialize the priority queue:
   - Create a priority queue (shortest_path) initialized with the start vertex and its distance (0).
   - The priority queue is used to explore vertices based on the smallest known distance.
3. Process vertices from the priority queue:
   - While the priority queue is not empty, do the following:
     - Extract the vertex current_vertex with the smallest known distance (current_distance) from the priority queue.
     - If the current distance is greater than the shortest known distance for this vertex, skip further processing.
     - For each neighbor of current_vertex:
       - Calculate the new distance to the neighbor as current_distance + weight of edge to neighbor.
       - If the new distance is shorter than the current known distance for the neighbor:
         Update shortest_paths[neighbor] with the new distance and push the neighbor with the updated distance to the priority queue.
4. Repeat the above step until all reachable vertices have been processed.
5. Output the shortest path distances from the start vertex to all other vertices.

Step 3: Output the Shortest Path Distances

1. For each vertex in the graph, Print the vertex name along with the shortest distance from the start vertex.

Pseudocode/Flowchart:

1. Function dijkstra(graph, start):
   - Initialize shortest_paths with all vertices set to infinity, except start set to 0
   - Create priority queue shortest_path with (0, start)
2. While shortest_path is not empty:
   - Pop current_vertex with smallest current_distance
   - If current_distance > shortest_paths[current_vertex], continue
   - For each neighbor, calculate new distance:
     - If smaller, update shortest_paths and push into queue
3. Return shortest_paths
4. Function input_graph():
   - Input number of vertices
   - For each vertex, input name, edges, and weights (ensure non-negative)
5. Main:
   - Call input_graph(), input start, call dijkstra(), and print results

Implementation:

```
#Imports priority queue to
import heapq

# Dijkstra's algorithm to find the shortest path from a start node to all
other nodes in a weighted graph
def dijkstra(graph, start):
    # Dictionary to store the shortest path distances from the start node
    shortest_paths = {vertex: float('infinity') for vertex in graph}
    shortest_paths[start] = 0

    # Priority queue to explore the minimum distance vertex first
    shortest_path = [(0, start)]  # (distance, vertex)

    while shortest_path:
        current_distance, current_vertex = heapq.heappop(shortest_path)

        # If the distance is larger than the shortest known path, skip it
        if current_distance > shortest_paths[current_vertex]:
            continue

        # Explore neighbors of the current vertex
        for neighbor, weight in graph[current_vertex]:
            distance = current_distance + weight

            # If found a shorter path, update and push to the queue
            if distance < shortest_paths[neighbor]:
                shortest_paths[neighbor] = distance
                heapq.heappush(shortest_path, (distance, neighbor))

    return shortest_paths

# Function to input the graph from the user
def input_graph():
    graph = {}
    num_vertices = int(input("Enter the number of vertices: "))

    for i in range(num_vertices):
        vertex = input(f"Enter vertex {i+1} name: ")
        graph[vertex] = []
```

```
        num_edges = int(input(f"Enter the number of edges from {vertex}: 
"))

        for j in range(num_edges):
            neighbor = input(f"  Enter neighbor {j+1} name: ")
            while True:  # Loop until valid (non-negative) weight is 
provided
                weight = int(input(f"  Enter weight of edge from {vertex} 
to {neighbor}: "))
                if weight < 0:
                    print("Error: Edge weight cannot be negative. Please 
enter a valid non-negative weight.")
                else:
                    break

            graph[vertex].append((neighbor, weight))

    return graph

graph = input_graph()

# Main
start = input("Enter the starting vertex: ")
shortest_distances = dijkstra(graph, start)

print(f"Shortest distances from {start}:")
for vertex, distance in shortest_distances.items():
    print(f"{vertex}: {distance}")
```

Output:

```
PS D:\KJSCE\SY\DSM\IA> python -u "d:\KJSCE\SY\DSM\IA\Dijastra.py"
Enter the number of vertices: 6
Enter vertex 1 name: A
Enter the number of edges from A: 2
  Enter neighbor 1 name: B
  Enter weight of edge from A to B: 5
  Enter neighbor 2 name: C
  Enter weight of edge from A to C: 7
Enter vertex 2 name: B
Enter the number of edges from B: 2
  Enter neighbor 1 name: C
  Enter weight of edge from B to C: 6
  Enter neighbor 2 name: D
  Enter weight of edge from B to D: 8
Enter vertex 3 name: C
Enter the number of edges from C: 2
  Enter neighbor 1 name: D
  Enter weight of edge from C to D: 3
  Enter neighbor 2 name: E
  Enter weight of edge from C to E: 9
Enter vertex 4 name: D
Enter the number of edges from D: 3
  Enter neighbor 1 name: E
  Enter weight of edge from D to E: 5
  Enter neighbor 2 name: F
  Enter weight of edge from D to F: 2
  Enter neighbor 3 name: C
  Enter weight of edge from D to C: 6
Enter vertex 5 name: E
Enter the number of edges from E: 1
  Enter neighbor 1 name: F
  Enter weight of edge from E to F: 4
Enter vertex 6 name: F
Enter the number of edges from F: 0
Enter the starting vertex: A
Shortest distances from A:
A: 0
B: 5
C: 7
D: 10
E: 15
F: 12
```

Result/Discussion:

Time Complexity - The time complexity of the Dijkstra's algorithm with a priority queue (using a binary heap) is O((V+E)logV), where V is the number of vertices and E is the number of edges. Space Complexity - The space complexity is O(V+E), where V is the number of vertices and E is the number of edges.

The implementation of Dijkstra's Algorithm successfully computes the shortest path from the starting vertex to all others in a weighted graph. It efficiently handles user inputs for vertices, edges, and weights, ensuring non-negative weights. The algorithm provides correct shortest path distances, demonstrating its effectiveness. Dijkstra's algorithm is widely used for solving the shortest path problem in weighted graphs, employing a greedy approach and a priority queue for optimal performance. Though it cannot handle negative edge weights, it remains a fundamental technique in graph theory and algorithm design for various applications.

Applications:

1. **Network Routing**:
   Dijkstra's algorithm is widely used in routing protocols like OSPF (Open Shortest Path First) to find the shortest path between routers in a network. It ensures efficient packet delivery by determining optimal paths.
2. **Geographical Information Systems (GIS)**:
   In mapping services such as Google Maps, Dijkstra's algorithm is used to calculate the shortest driving or walking route between two locations.
3. **Telecommunications**:
   It helps in determining the most efficient path for data transmission in telecommunication networks, minimizing delays and improving the quality of service.
4. **AI and Robotics**:
   Dijkstra's algorithm is used in pathfinding algorithms in robotics and AI to navigate agents through complex environments.

References/Research Papers: (In IEEE format)

https://ieeexplore.ieee.org/document/9190342
https://www.sciencedirect.com/topics/computer-science/dijkstra-algorithms
https://brilliant.org/wiki/dijkstras-short-path-finder/
https://www.sciencedirect.com/topics/computer-science/dijkstra-algorithms