Analysis of Algorithm

Practical no 6

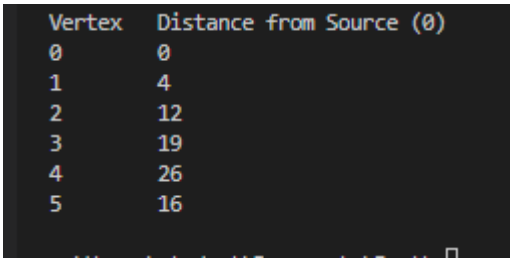
Dijsktra's Algorithm

Code :

```
import java.util.*;
import java.io.*;
public class Dijkstra {
  static int getMinVertex(int[] distance, boolean[] visited, int vertices) {
    int minVertex = -1;
    for (int i = 0; i < vertices; i++) {
      if (!visited[i] && (minVertex == -1 || distance[i] < distance[minVertex])) {
        minVertex = i;
      }
    }
    return minVertex;
  }
  static void dijkstra(int[][] graph, int source, int vertices) {
    int[] distance = new int[vertices];
    boolean[] visited = new boolean[vertices];
    Arrays.fill(distance, Integer.MAX_VALUE);
    distance[source] = 0;
    for (int i = 0; i < vertices - 1; i++) {
      int minVertex = getMinVertex(distance, visited, vertices);
      visited[minVertex] = true;
      for (int j = 0; j < vertices; j++) {
        if (graph[minVertex][j] != 0 && !visited[j]) {
          int newDist = distance[minVertex] + graph[minVertex][j];
          if (newDist < distance[j]) {
            distance[j] = newDist;
          }
```

```
            }
          }
        }
        System.out.println("Vertex \t Distance from Source (" + source + ")");
        for (int i = 0; i < vertices; i++) {
          System.out.println(i + " \t " + distance[i]);
        }
    }
    public static void main(String[] args) {
        int[][] graph = {
          {0, 4, 0, 0, 0, 0},
          {4, 0, 8, 0, 0, 0},
          {0, 8, 0, 7, 0, 4},
          {0, 0, 7, 0, 9, 14},
          {0, 0, 0, 9, 0, 10},
          {0, 0, 4, 14, 10, 0}
        };
        int source = 0;
        int vertices = 6;
        dijkstra(graph, source, vertices);
    }
}
```

Output :



Analysis :

1. **getMinVertex Method**:
    - This method is used to find the vertex with the minimum distance that has not been visited yet. The method iterates through all vertices and checks the distance array to find the vertex with the smallest value.
2. **dijkstra Method**:
    - This is the core method where Dijkstra's algorithm is implemented. It works as follows:
        1. Initialize two arrays:
            - distance[]: Holds the shortest known distance from the source to each vertex. Initially, all distances are set to infinity, except the source vertex which is set to 0.
            - visited[]: Marks whether a vertex has been visited.
        2. For each vertex (except the source), it finds the unvisited vertex with the smallest distance (using getMinVertex()), marks it as visited, and updates the distances of its neighbors.
        3. Once all vertices are visited, the method prints the shortest distances from the source to all other vertices.
3. **main Method**:
    - A graph is represented as an adjacency matrix (graph[][]), where graph[i][j] represents the weight of the edge between vertex i and vertex j. If there's no edge, it's represented by 0.
    - The source vertex is 0, and the number of vertices in the graph is 6.
    - The dijkstra() method is called with the graph, source, and number of vertices.

Time Complexity Analysis

The time complexity of Dijkstra's algorithm depends on how the minimum vertex is found and how the distances are updated. Let's analyze the time complexity step by step.

1. **getMinVertex Method**:
    - This method iterates through all the vertices to find the one with the minimum distance. It performs a linear scan of the distance[] array, which takes **O(V)** time, where **V** is the number of vertices in the graph.
2. **dijkstra Method**:
    - In the dijkstra method, we repeat the getMinVertex operation **V-1** times (since one vertex is processed in each iteration of the loop).
    - Inside the main loop, after choosing the minimum vertex, we update the distances for all **V** neighboring vertices (using the adjacency matrix). This loop runs for each of the **V-1** iterations.

The time complexity of each iteration of the outer loop is dominated by the inner loop that updates the distances, which runs in **O(V)** time.

So, the total time complexity of the algorithm is:

O(V)×O(V)=O(V2)O(V) \times O(V) = O(V^2)O(V)×O(V)=O(V2)

Space Complexity Analysis

1. **Graph Representation**:
    - The graph is represented as an adjacency matrix, which takes **O(V^2)** space. This is because we need to store the weight of edges between each pair of vertices, which requires a 2D matrix of size **V × V**.
2. **Arrays for Distance and Visited**:
    - The distance[] array stores the shortest distance for each vertex, and it requires **O(V)** space.
    - The visited[] array is used to track whether a vertex has been visited, and it also requires **O(V)** space.

Therefore, the total space complexity is:

O(V2) (for the adjacency matrix)+O(V) (for distance array)+O(V) (for visited array)=O(V2)O(V^2) \text{ (for the adjacency matrix)} + O(V) \text{ (for distance array)} + O(V) \text{ (for visited array)} = O(V^2)O(V2) (for the adjacency matrix)+O(V) (for distance array)+O(V) (for visited array)=O(V2)

Time and Space Complexity for Best, Average, and Worst Cases

- **Best Case**: The best-case time complexity occurs when the graph is already in an optimal form. However, Dijkstra's algorithm will still run in **O(V^2)** time, because it always needs to check every vertex and edge in the graph, regardless of the graph's structure.
    - **Time Complexity (Best Case)**: **O(V^2)**
    - **Space Complexity (Best Case)**: **O(V^2)**
- **Average Case**: The average case occurs in typical random graphs where the algorithm will still need to iterate over all vertices and edges. The time complexity does not change because we still need to check all vertices and their edges to find the shortest path.
    - **Time Complexity (Average Case)**: **O(V^2)**
    - **Space Complexity (Average Case)**: **O(V^2)**
- **Worst Case**: The worst-case time complexity occurs when the graph has the maximum possible number of edges, and the algorithm must examine each vertex and edge multiple times. In this case, the time complexity remains **O(V^2)** because of the way the algorithm processes each vertex and its neighbors.
    - **Time Complexity (Worst Case)**: **O(V^2)**
    - **Space Complexity (Worst Case)**: **O(V^2)**