Dijkstra Sequence

Yunfan Li, 3200102555

December 13, 2023

1 Introduction

Dijkstra's algorithm, a cornerstone in the field of computer science and graph theory, is designed for finding the shortest paths from a single source vertex to all other vertices in a weighted graph. This report delves into the implementation and validation of Dijkstra's algorithm in the C programming language. Specifically, it focuses on validating sequences generated by the algorithm to determine their compliance with the shortest path criteria set forth by Dijkstra's algorithm in a given graph.

2 Algorithm Specification

The solution's centerpiece is the implementation of Dijkstra's algorithm, followed by a validation process. This process involves checking if a given sequence of vertices in a graph matches the shortest path sequence as determined by Dijkstra's algorithm from a specified source vertex.

2.1 Pseudocode

The following is the pseudocode for Dijkstra's algorithm and the sequence validation process:

```
Algorithm Dijkstra(Graph, source):
    dist[source] ← 0
    create vertex set Q

    for each vertex v in Graph:
        if v  source
            dist[v] ← INFINITY
            prev[v] ← UNDEFINED
        Q.add_with_priority(v, dist[v])

    while Q is not empty:
        u ← Q.extract_min()
        for each neighbor v of u:
            alt ← dist[u] + length(u, v)
            if alt < dist[v]
                dist[v] ← alt
```

```
            prev[v] ← u

Algorithm IsValidDijkstraSequence(Graph, sequence, source):
    Dijkstra(Graph, source)
    for i = 1 to length(sequence) - 1
        if dist[sequence[i]] < dist[sequence[i-1]]
            return false
    return true
```

3 Implementation Details

The C implementation of Dijkstra's algorithm utilizes arrays to store the vertices, their distances, and a boolean set to track the shortest path. A two-dimensional array represents the graph's adjacency matrix, with each edge having a corresponding weight. The implementation also considers edge cases and ensures accuracy in the shortest path calculation.

4 Testing Results

Here is the submission details according to PTA online tests.

Submission Detail

Test Case	hint	Memory(KB)	Time(ms)	Status	Score
0		352	3	Accepted	15 / 15
1		356	3	Accepted	8 / 8
2		360	3	Accepted	2 / 2
3		5880	315	Accepted	5 / 5

Figure 1: Testing Results

5 Analysis and Comments

The C implementation of Dijkstra's algorithm exhibits $O(V^2)$ time complexity, where V represents the number of vertices. This complexity is due to the use of arrays for storing vertices and edges. For large-scale graphs, an optimized approach using priority queues can significantly reduce the computational complexity. The sequence validation logic has proven effective in distinguishing valid Dijkstra sequences, affirming the implementation's reliability.

In addition to time complexity, the space complexity of the algorithm is also a critical factor. The implementation of Dijkstra's algorithm in C primarily utilizes arrays to represent the graph and to store distances and shortest path information. The space complexity is $O(V^2)$ for the adjacency matrix representation, where V is the number of vertices. This is because a two-dimensional array is used to store the weights of edges between each pair of vertices. For sparse graphs, this can be space-inefficient. An adjacency list representation would offer a more space-efficient alternative, particularly for sparse graphs, with space complexity being $O(V + E)$, where E is the number of edges.

6 Conclusion

The C program effectively demonstrates the principles of Dijkstra's algorithm and its application in real-world scenarios. The ability to validate Dijkstra sequences further extends its utility, providing a comprehensive solution for graph analysis in various computational fields.

7 Appendix: Source Code (in C)

```
#include <limits.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

#define MAX_VERTICES 1003
#define MAX_EDGES 100005

// Structure to represent an edge in the graph
typedef struct {
  int vertex; // Destination vertex of the edge
  int weight; // Weight of the edge
} Edge;

Edge graph[MAX_VERTICES]
          [MAX_VERTICES];    // Graph represented as an adjacency matrix
int distances[MAX_VERTICES]; // Array to store shortest distance from source
bool shortestPathSet[MAX_VERTICES]; // Boolean array to track vertices included in 
int edgeCount[MAX_VERTICES];        // Count of edges for each vertex

// Function to perform Dijkstra's algorithm for a given source vertex
void dijkstra(int source, int vertices) {
  for (int i = 1; i <= vertices; i++) {
    distances[i] = INT_MAX; // Initialize distances to maximum value
    shortestPathSet[i] =
        false; // Mark all vertices as not included in shortest path yet
  }

  distances[source] = 0; // Distance of source vertex from itself is always 0

  // Find shortest path for all vertices
  for (int count = 0; count < vertices - 1; count++) {
    int minDistance = INT_MAX, minIndex;

    // Pick the minimum distance vertex from the set of vertices not yet processed
    for (int v = 1; v <= vertices; v++) {
      if (!shortestPathSet[v] && distances[v] <= minDistance) {
        minDistance = distances[v], minIndex = v;
      }
    }
```

```
    // Mark the picked vertex as processed
    shortestPathSet[minIndex] = true;

    // Update distance value of the adjacent vertices of the picked vertex
    for (int i = 0; i < edgeCount[minIndex]; i++) {
      Edge e = graph[minIndex][i];
      if (!shortestPathSet[e.vertex] && distances[minIndex] != INT_MAX &&
          distances[minIndex] + e.weight < distances[e.vertex]) {
        distances[e.vertex] = distances[minIndex] + e.weight;
      }
    }
  }
}

// Function to check if a given sequence is a Dijkstra sequence
bool isDijkstraSequence(int vertices, int sequence[]) {
  for (int i = 1; i <= vertices; i++) {
    if (distances[sequence[i - 1]] != INT_MAX) {
      for (int j = i; j < vertices; j++) {
        if (distances[sequence[j]] < distances[sequence[i - 1]]) {
          return false; // If any vertex has a shorter distance, sequence is not va
        }
      }
    } else {
      return false; // If any vertex is unreachable, sequence is not valid
    }
  }
  return true; // If all checks pass, sequence is valid
}

int main() {
  int vertices, edges;
  scanf("%d␣%d", &vertices, &edges);

  // Reading edges and constructing the graph
  for (int i = 0; i < edges; i++) {
    int src, dest, weight;
    scanf("%d␣%d␣%d", &src, &dest, &weight);
    graph[src][edgeCount[src]].vertex = dest;
    graph[src][edgeCount[src]++].weight = weight;
    // Assuming undirected graph
    graph[dest][edgeCount[dest]].vertex = src;
    graph[dest][edgeCount[dest]++].weight = weight;
  }

  int queries;
  scanf("%d", &queries);

  // Processing each query
  while (queries--) {
    int sequence[MAX_VERTICES];
    for (int i = 0; i < vertices; i++) {
      scanf("%d", &sequence[i]);
    }
    dijkstra(sequence[0], vertices);
    printf("%s\n", isDijkstraSequence(vertices, sequence) ? "Yes" : "No");
  }
```

```
    return 0;
}
```

8 Declaration

I hereby declare that all the work done in this project is my independent effort.