Fundamentals of Data Structures

Projects 3: Normal Dijkstra Sequence

Table of Contents

Chapter 1: Introduction

1.1) Background

The Dijkstra algorithm, developed by Edsger W. Dijkstra in 1956, is a fundamental algorithm in graph theory that solves the single-source shortest path problem. Its significance in computer science and practical applications has made it one of the most studied greedy algorithms.

1.2) Algorithm Overview

The algorithm works by maintaining a set of vertices whose shortest path distances from the source have been determined. In each iteration:

- It selects the vertex with the minimum distance value from the set of unvisited vertices
- Adds this vertex to the visited set
- Updates the distance values of its adjacent vertices

This process generates an ordered sequence of vertices, known as a Dijkstra sequence.

1.3) Problem Description

Given a connected weighted graph with N vertices and E edges:

- Each edge has a positive integer weight (≤100)
- Multiple sequences of vertex permutations are provided
- Each sequence starts with a source vertex
- We need to determine whether each sequence is a valid Dijkstra sequence

The challenge lies in verifying if each given sequence could be generated by Dijkstra's algorithm based on the graph structure and edge weights.

1.4) Key Concepts

- A sequence is considered a valid Dijkstra sequence if:
  - It represents a possible order of vertex selection by Dijkstra's algorithm
  - Each vertex selected must have the minimum distance from the source among all unvisited vertices
  - The first vertex must be the source vertex

1.5) Input Constraints

- Number of vertices (N): $\leq 10^3$
- Number of edges (E): $\leq 10^5$
- Number of test sequences (K): ≤ 100
- Edge weights: Positive integers ≤ 100

Chapter 2: Algorithm Specification

2.1) Data Structures

2.1.1) Graph Representation

- Adjacency Matrix:
  - Size: `MAXV × MAXV` (where `MAXV = 1003`)
  - `G[i][j]`: Weight of edge between vertices i and j
  - `G[i][j] = INF` if no direct edge exists
  - `G[i][i] = 0` for all vertices

2.1.2) Distance Array

- dist[]: Stores shortest known distances from source
  - Size: `MAXV`
  - `dist[i]`: Current shortest distance to vertex i
  - Initially set to `INF` except `dist[source] = 0`

2.1.3) Visited Set

- visited[]: Tracks processed vertices
  - Size: `MAXV`
  - Boolean array (`0` for unvisited, `1` for visited)

2.2) Core Algorithm Components

2.2.1) Graph Initialization

1. Set all edge weights to `INF`
2. Set diagonal elements to `0`
3. Read edges and populate adjacency matrix

2.2.2) Sequence Validation

1. Initialize distances and visited array
2. For each vertex in sequence:
   - Mark as visited
   - Update distances to adjacent vertices
   - Verify next vertex has minimum distance

2.2.3) Distance Update Process

1. For each unvisited adjacent vertex:
   - Calculate potential new distance
   - Update if new distance is shorter
   - Track if any distance was improved

2.3) Pseudo-Code

```
Input: Graph G, Sequence S, Number of vertices N
Output: true if S is valid Dijkstra sequence, false otherwise

// Data structures initialization
function Initialize():
    dist[1..N] = [INF, INF, ..., INF]    // Distance array
    visited[1..N] = [false, false, ..., false]    // Visited status array
    dist[S[0]] = 0    // Set source distance to 0

// Main validation function
function CheckSequence(G, S, N):
    Initialize()

    // Process each vertex in sequence
    for i from 0 to N-1:
        v = S[i]    // Current vertex

        // Check if current vertex is reachable
        if dist[v] == INF:
            return false    // Unreachable vertex

        visited[v] = true

        // Update distances
        for each unvisited vertex u:
            if G[v][u] != INF:    // If edge exists
                newDist = dist[v] + G[v][u]
                if newDist < dist[u]:
                    dist[u] = newDist

        // Verify next vertex selection
        if i < N-1:
            next = S[i+1]
            // Check if any unvisited vertex has shorter distance
            for each unvisited vertex u:
                if dist[u] < dist[next]:
                    return false

    return true    // Valid sequence

// Main program
Main():
    Read N (vertices), E (edges)
    Initialize graph G

    // Read edges
    for i from 1 to E:
        Read v1, v2, weight
        G[v1][v2] = G[v2][v1] = weight

    Read K (number of queries)
```

```
    for i from 1 to K:
        Read sequence S of length N
        if CheckSequence(G, S, N):
            Print "Yes"
        else:
            Print "No"
```

Chapter 3: Testing Results

3.1) Test Case 1: Basic Graph

Purpose: Verify sequence validation for simple graph structure

Input:

```
5 6
1 2 2
1 3 4
2 3 1
2 4 3
3 4 1
4 5 2
2
1 2 3 4 5
1 2 4 3 5
```

Expected Output:

```
Yes
No
```

Analysis: First sequence follows shortest paths, second violates minimum distance property

Status: pass

3.2) Test Case 2: Multiple Valid Paths

Purpose: Test graphs with multiple valid Dijkstra sequences

Input:

```
4 4
1 2 1
1 3 1
2 4 1
3 4 1
3
1 2 3 4
1 3 2 4
2 1 3 4
```

Expected Output:

```
Yes
Yes
No
```

Analysis: First two sequences are valid due to equal path lengths, third invalid due to wrong source

Status: pass

3.3) Test Case 3: Maximum Size Graph

Purpose: Test performance with large input

Input:

```
1000 100000
// ... (1000 vertices, 100000 edges)
```

Expected Output: Correct validation within time limit

Analysis: Program handles maximum constraints efficiently

Status: pass

3.4) Test Case 4: Single Vertex Graph

Purpose: Test the simplest possible graph case

Input:

```
1 0
1
1
```

Expected Output:

```
Yes
```

Analysis: Trivial case with only one vertex, any single-vertex sequence starting with itself is valid

Status: pass

3.5) Test Case 5: Complete Graph

Purpose: Test graph with all possible edges

Input:

```
4 6
1 2 1
1 3 1
1 4 1
2 3 1
2 4 1
3 4 1
2
1 2 3 4
1 3 2 4
```

Expected Output:

```
Yes
Yes
```

Analysis: All vertices are directly connected with equal weights, multiple valid sequences exist

Status: pass

3.6) Test Case 6: Linear Graph (Path)

Purpose: Test sequence validation in a path graph

Input:

```
5 4
1 2 1
2 3 1
3 4 1
4 5 1
1
1 2 3 4 5
```

Expected Output:

```
Yes
```

Analysis: Only one valid Dijkstra sequence possible due to linear structure

Status: pass

3.7) Test Case 7: Cyclic Graph

Purpose: Test graph with cycle and equal weights

Input:

```
4 4
1 2 1
2 3 1
3 4 1
4 1 1
2
1 2 3 4
1 4 3 2
```

Expected Output:

```
No
No
```

Analysis: Equal weights in cycle, but sequence must follow shortest path order

Status: pass

Chapter 4: Analysis and Comments

4.1) Time Complexity Analysis

4.1.1) Graph Initialization

- Adjacency Matrix Creation: $O(V^2)$
  - Initializing all elements to INF requires two nested loops
  - V is the number of vertices ($\leq 10^3$)
- Edge Input: $O(E)$
  - Processing E edges, each requiring constant time
  - E is the number of edges ($\leq 10^5$)

4.1.2) Sequence Validation

- Distance Array Initialization: $O(V)$
- Main Loop: $O(V^2)$
  - Outer loop: Processes V vertices
  - Inner loop: Updates distances for up to V vertices
- Validation Check: $O(V^2)$
  - For each vertex, checks all remaining unvisited vertices
- Total: $O(V^2)$

4.2) Space Complexity Analysis

4.2.1) Static Memory

- Adjacency Matrix: $O(V^2)$
  - Size: $MAXV \times MAXV$ integers
  - Dominates the space complexity
- Distance Array: $O(V)$
- Visited Array: $O(V)$

4.2.2) Dynamic Memory

- No dynamic memory allocation required
- All arrays can be statically allocated
- Total: $O(V^2)$

4.3) Potential Improvements

4.3.1) Algorithm Optimizations

1. Alternative Graph Representation:
   - Use adjacency lists instead of matrix
   - Reduces space to $O(V + E)$
   - Update time becomes $O(E \log V)$ with priority queue
2. Early Termination:
   - Stop validation when finding first violation

- Add checks for impossible sequences early

3. Memory Optimization:
   - Use bit array for visited set
   - Reduces memory usage by factor of 32

4.3.2) Implementation Enhancements

1. Input Processing:
   - Buffer multiple test cases
   - Use faster I/O methods

2. Data Structures:
   - Custom priority queue for large graphs
   - Sparse matrix for low-density graphs

3. Parallelization:
   - Process multiple queries concurrently
   - Parallelize distance updates for large graphs

4.4) Performance Bottlenecks

4.4.1) Current Limitations

1. Memory Usage:
   - Adjacency matrix requires $O(V^2)$ space
   - Inefficient for sparse graphs

2. Computational Overhead:
   - Checking all unvisited vertices in each step
   - Redundant distance updates

4.4.2) Theoretical Bounds

- Best Case: O(V) when sequence is invalid
- Worst Case: $O(V^2)$ for complete graphs
- Average Case: O(V log V) for sparse graphs

4.5) Final Complexity Summary

Operation	Time Complexity	Space Complexity
Graph Construction	$O(V^2 + E)$	$O(V^2)$
Sequence Validation	$O(V^2)$	$O(V)$
Total (per query)	$O(V^2)$	$O(V^2)$

Appendix: Source Code (in C)

File sol.c:

```
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXV 1003
#define MAXE 100005
#define INF 0x3f3f3f3f

// Adjacency matrix representation of the graph
int G[MAXV][MAXV];
int Nv, Ne;

// Initialize the graph with infinite weights
void initGraph() {
    for(int i = 1; i <= Nv; i++) {
        for(int j = 1; j <= Nv; j++) {
            G[i][j] = INF;
        }
        G[i][i] = 0; // Distance to itself is 0
    }
}

// Verify if a sequence is a valid Dijkstra sequence
// Returns 1 if valid, 0 if invalid
int checkSequence(int sequence[]) {
    // Arrays for storing shortest distances and visited status
    int dist[MAXV];
    int visited[MAXV] = {0};    // 0: unvisited, 1: visited

    // Initialize all distances to infinity
    for (int i = 1; i <= Nv; i++) {
        dist[i] = INF;
    }

    // Set source vertex distance to 0
    int source = sequence[0];
    dist[source] = 0;

    // Process each vertex in the sequence
    for (int i = 0; i < Nv; i++) {
        int v = sequence[i];

        // Check if current vertex is reachable from source
        if (dist[v] == INF) {
            return 0;   // Unreachable vertex, sequence invalid
        }

        visited[v] = 1;    // Mark current vertex as visited

        // Update distances to all unvisited adjacent vertices
        for (int j = 1; j <= Nv; j++) {
```

```
            if (!visited[j] && G[v][j] != INF) {
                int newDist = dist[v] + G[v][j];
                // Update if new path is shorter
                if (newDist < dist[j]) {
                    dist[j] = newDist;
                }
            }
        }

        // Verify next vertex selection (if not the last vertex)
        if (i < Nv - 1) {
            int next = sequence[i + 1];
            // Check if any unvisited vertex has shorter distance
            for (int j = 1; j <= Nv; j++) {
                if (!visited[j] && dist[j] < dist[next]) {
                    return 0;   // Found better choice, sequence invalid
                }
            }
        }
    }
    // All vertices processed successfully
    return 1;    // Valid Dijkstra sequence
}

// Main function - handles input/output and program flow
int main() {
    // Read number of vertices and edges
    scanf("%d %d", &Nv, &Ne);
    initGraph();    // Initialize adjacency matrix

    // Read edge information and construct graph
    for(int i = 0; i < Ne; i++) {
        int v1, v2, weight;
        scanf("%d %d %d", &v1, &v2, &weight);
        // Store edge weight (undirected graph)
        G[v1][v2] = G[v2][v1] = weight;
    }

    // Process queries
    int K;
    scanf("%d", &K);    // Read number of sequences to check

    // Check each sequence
    for(int i = 0; i < K; i++) {
        int sequence[MAXV];
        // Read vertex sequence
        for(int j = 0; j < Nv; j++) {
            scanf("%d", &sequence[j]);
        }
        // Output result
        printf("%s\n", checkSequence(sequence) ? "Yes" : "No");
```

```
    }

    return 0;
}
```

Declaration

I hereby declare that all the work done in this project titled "Normal Dijkstra Sequence" is of my independent effort.