The 2nd-shortest Path Report

2023.12.1

浙江大學

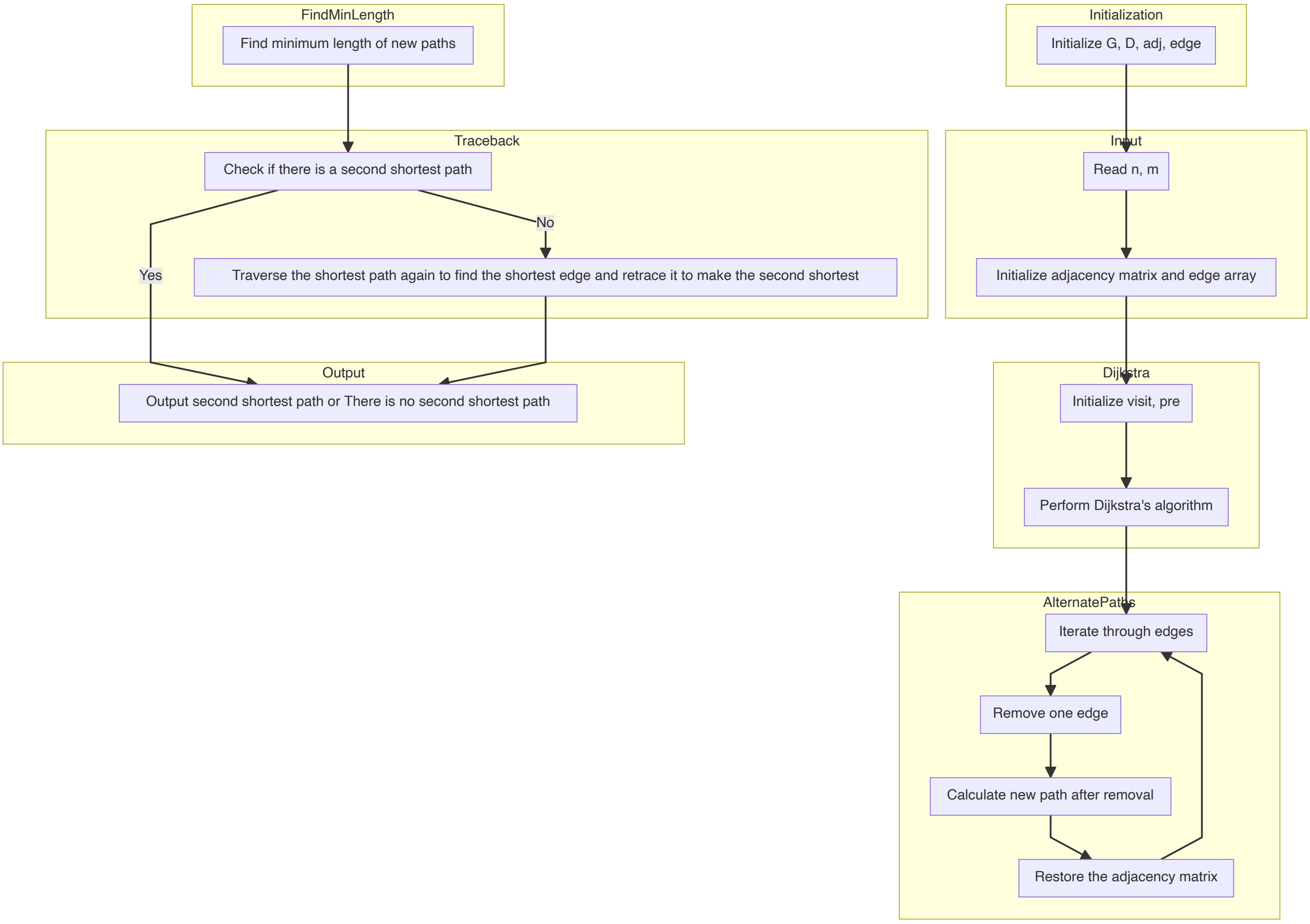
Chapter 1: Introduction and Problem Description

The task at hand involves helping Lisa find the second-shortest path from station 1 to station M in a railway network. The input consists of the number of stations (M) and the number of unidirectional railways (N). Each railway is defined by three integers A, B, and D, representing a road from station A to station B with a length of D. The goal is to output the length of the second-shortest path and the nodes' indices in order.

Chapter 2: Algorithm Overview

sketch of the algorithm

The algorithm is based on Dijkstra's algorithm. The algorithm is as follows:



Data Structure

The code uses the following data structures:

```
int G[1001][1001];  // The adjacency matrix of the graph

int D[1001];
```

The presedo code of the algorithm

DIJKSTRA ALGORITHM

```
function dijkstra(s, t):
    initialize visit array with zeros
    initialize D array with distances from source vertex s to all vertices
    set distance from source vertex s to itself as 0
    mark source vertex s as visited
    initialize pre array with -1 representing previous vertices in the shortest path
    for i from 1 to n:
        min = MAX
```

```
        index = undefined
        // Find the unvisited vertex with the smallest distance from the source vertex
        for j from 1 to n:
            if visit[j] is not marked and D[j] < min:
                min = D[j]
                index = j
        // If it's the first iteration, set the source vertex as the predecessor of the selected vertex
        if i is 1:
            pre[index] = i
        // Mark the selected vertex as visited
        mark visit[index] as 1

        // Update distances to neighboring vertices
        for j from 1 to n:
            if visit[j] is not marked and D[index] + G[index][j] < D[j]:
                update D[j] with D[index] + G[index][j]
                set pre[j] as index
    // Save the shortest path
    create a path_of_road object ab
    set ab.len as D[t]
    set ab.index as 0
    set temp as t
    // Save the vertices of the shortest path
    while temp is not equal to s:
        set ab.path[ab.index] as temp
        increment ab.index
        set temp as pre[temp]
    // Save the source vertex
    set ab.path[ab.index] as s

    return ab
```

Algorithm for finding the second shortest path

```
for all the edges:
    remove the edge from the graph
    calculate the new path after removing the edge
    save the new path
    restore the edge to the graph
for all the new paths:
    if the length of the new path is smaller than the length of the shortest path and larger than the length of the second shortest path:
        update the length of the second shortest path
        update the index of the second shortest path
// If the length of the second shortest path is still MAX, there is no second shortest path
```

The algorithm for finding the second shortest path when there seems to be no second shortest path

```
if the path to the destination is larger than 2:
   calculate the shortest path again
   output the second shortest path and its vertices
   find the edges of the shortest path
   find the edge with the smallest weight
   output the edge with the smallest weight
   output the vertices of the second shortest path
   the second shortest path is to traverse the path again
else:
   output "There is no second shortest path"
```

The analysis of the algorithm

The program first uses Dijkstra's algorithm to find the shortest path from the source vertex to the target vertex.

Then, the program iterates through all the edges and calculates the new path after removing one edge.

Finally, the program finds the second shortest path from all the new paths.

The reason why this works is that the second shortest path must be one of the new paths,there must be a edge in the shortest path that is not in the second shortest path, and the second shortest path must be one of the new paths.

Chapter 3: Testing Results

Test Cases

The table below outlines a framework for test cases:

```
Input:
5 6
1 2 50
2 3 100
2 4 150
3 4 130
3 5 70
4 5 40
Output:
The second shortest path is 240
The path is 1 2 4 5
```

This test case is a test case with a small number of vertices and edges.

```
Input:
3 2
1 2 10
2 3 20
Output:
The second shortest path is 50
The path is 1 2 1 2 3
```

This test aims to test the case where the second shortest path is the reverse of the shortest path.

```
Input:
8 14
1 2 2
1 3 5
1 4 3
2 3 2
2 5 4
3 4 3
3 5 7
4 6 1
5 6 5
5 7 8
6 8 4
7 8 3
2 8 9
3 8 2
Output:
The second shortest path is 7
The path is 1 3 8
```

this is a complex test case with a large number of vertices and edges.

```
Ibput:
2 1
1 2 10
Output:
There is no second shortest path
```

this test case is a test case with a the smallest number of vertices and edges and there is no second shortest path.

```
Input:
4 5
1 2 1
1 3 2
1 4 3
2 3 4
3 4 5
Output:
The second shortest path is 7
The path is 1 3 4
```

This is a normal test case with a small number of vertices and edges.

Chapter 4: Analysis and Comments

Time and Space Complexity Analysis

Time Complexity

DIJKSTRA ALGORITHM

In this code, the Dijkstra's algorithm is called once outside the loop and then within the loop for each edge removal, making the overall time complexity for Dijkstra's algorithm in the loop O(|V|^2).

That's because the STRUCT is made of two for loops, and the time complexity of the two for loops is O(|V|^2).

Edge deletion and restoration:

Inside of the loop of E, the time complexity is O($|V|^2$) for Dijkstra's algorithm a, making the overall time complexity O($|E| * |V|^2) sothetimecomplexityofedgedeletionandrestorationisO$(|E| * |V|^2 $).

TIME COMPLEXITY OF THE OVERALL PROGRAM

The overall time complexity is dominated by the Dijkstra's algorithm within the loop, so the final time complexity is O($|E| * |V|^2$).

It is consist of the time complexity of edge deletion and restoration and the time complexity of Dijkstra's algorithm,which added up to O($|E| * |V|^2$).

Space Complexity

Adjacency Matrix and List:

The space complexity for the adjacency matrix is O(|V|^2).(N*N)

DIJKSTRA ALGORITHM

The space complexity for Dijkstra's algorithm is O(|V|) for the distance array, O(|V|) for the visit array, O(|V|) for the pre array, and O(|V|) for temporary variables.

Thus , the space complexity for Dijkstra's algorithm is O(|V|).

Path and Edge Arrays:

The space complexity for the path array is O(|E|) since it stores alternate paths for each edge.

edge deletion and restoration:

The space complexity for edge deletion and restoration is O(|V|^2) for the adjacency matrix and list.

It used a struct Path to store the path and other value of the second shortest path, and the space complexity is O(|V|).And there are at most |E| paths, so the space complexity is O(|E|*|V|).

TIME COMPLEXITY OF THE OVERALL PROGRAM

The overall space complexity is dominated by the adjacency matrix and list, making it O(|V|*(|V|+|E|)).(N^2)*

Chapter 5: Source Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>// INT_MAX

#define MAX 1000000

int G[1001][1001];  // The adjacency matrix of the graph

int D[1001];

typedef struct node *nodeptr;
struct node { // adjacency list node
    int v;
```

```
    int w;
    nodeptr next;
};

struct edge { // edge structure
    int u;
    int v;
    int w;
} edge[1001];

typedef struct Path path_of_road;
struct Path {
    int path[1001];// The path of the second shortest path
    int len;
    int index;
} path[1001];

nodeptr adj[1001];  // adjacency list
int n, m;           // n represents the number of vertices, and m represents the number of edges

// Function to perform Dijkstra's algorithm
path_of_road dijkstra(int s, int t) {
    int i, j;
    int visit[1001];
    memset(visit, 0, sizeof(visit));// Initialize the visit array to 0

    for (i = 1; i <= n; i++) {
        D[i] = G[s][i];// Initialize the distance array
    }

    D[s] = 0;// The distance from the source vertex to itself is 0
    visit[s] = 1;// The source vertex has been visited

    int min;
    int index;
    int pre[1001];// Initialize the pre array to -1
    memset(pre, -1, sizeof(pre));
    // pre[i] represents the previous vertex of vertex i in the shortest path
    for (i = 1; i <= n; i++) {
        min = MAX;
    /// Find the vertex with the smallest distance from the source vertex
        for (j = 1; j <= n; j++) {
            if (visit[j] == 0 && D[j] < min) {
                min = D[j];
                index = j;
            }
        }
```

```
        // If the vertex with the smallest distance is the target vertex, the algorithm ends
        if (i == 1)
            pre[index] = i;
        // Mark the vertex with the smallest distance as visited
        visit[index] = 1;
        // Update the distance array
        for (j = 1; j <= n; j++) {
            if (visit[j] == 0 && D[index] + G[index][j] < D[j]) {
                D[j] = D[index] + G[index][j];
                pre[j] = index;
            }
        }
    }
    // Save the shortest path
    path_of_road ab;
    ab.len = D[t];
    ab.index = 0;
    int temp = t;
    // Save the vertices of the shortest path
    while (temp != s) {
        ab.path[ab.index] = temp;
        ab.index++;
        temp = pre[temp];
    }
    // Save the source vertex
    ab.path[ab.index] = s;
    return ab;
}

int main() {
    int pre[1001];// Initialize the pre array to -1
    memset(pre, -1, sizeof(pre));

    scanf("%d %d", &n, &m);// Input the number of vertices and edges
    int i, j;

    // Initialize the adjacency matrix
    for (i = 1; i <= n; i++) {
        for (j = 1; j <= n; j++) {
            G[i][j] = MAX;
        }
    }

    int a, b, c;

    // Populate the adjacency matrix and edge array
    for (i = 0; i < m; i++) {
```

```
        scanf("%d %d %d", &a, &b, &c);
        G[a][b] = c;
        G[b][a] = c;
        edge[i].u = a;
        edge[i].v = b;
        edge[i].w = c;
    }
    // Initialize the adjacency list
    int s, t;
    s = 1, t = n;

    // Calculate the shortest path using Dijkstra's algorithm
    path_of_road ab = dijkstra(s, t);

    int min_act = ab.len;

    // Iterate through all edges and calculate alternate paths
    for (j = 0; j < m; j++) {
        G[edge[j].u][edge[j].v] = MAX;
        G[edge[j].v][edge[j].u] = MAX;

        // Calculate the new path after removing one edge
        struct Path tmp = dijkstra(s, t);

        path[j].len = tmp.len;
        path[j].index = tmp.index;
        path[j].path[0] = 1;
        int con = 0;

        // Save the vertices of the new path
        for (i = tmp.index - 1; i >= 0; i--) {
            if (tmp.path[i] != -1 && tmp.path[i] != 0)
                path[j].path[con++] = tmp.path[i];
        }

        path[j].index = con;
        // Restore the adjacency matrix
        G[edge[j].u][edge[j].v] = edge[j].w;
        G[edge[j].v][edge[j].u] = edge[j].w;
    }

    int min = MAX;
    int minindex = -1;

    // Find the minimum length of the new paths
    for (i = 0; i < m; i++) {
        if (path[i].len < min && path[i].len > min_act) {
```

```
            min = path[i].len;
            minindex = i;
        }
    }

    // Check if there is a second shortest path
    if (minindex == -1) {
        if (n > 2) {
            // If n>2, consider backtracking once to find the second shortest path
            s = 1, t = n;
            ab = dijkstra(s, t);
            // Output the second shortest path and its vertices
            printf("1 ");
            for (i = ab.index - 1; i >= 0; i--) {
                if (ab.path[i] != -1)
                    printf("%d ", ab.path[i]);
            }

            // Find the edges of the shortest path
            int min_act = ab.len;
            int lens = ab.index;
            int indexs = 0;
            int M[1001];
            M[0] = G[1][ab.path[1]];
            int mins = M[0];
            // Find the edge with the smallest weight
            for (j = 1; j < lens - 1; j++) {
                M[j] = G[ab.path[j]][ab.path[j + 1]];
                if (M[j] < mins && ab.path[j + 1] != n) {
                    mins = M[j];
                    indexs = j;
                }
            }
            // Output the edge with the smallest weight
            printf("%d ", indexs);

            // The second shortest path is to traverse the path again
            printf("The second shortest path is ");
            printf("%d\n", ab.len + 2 * mins);
            printf("The path is ");

            ab.path[ab.index] = 1;
            // Output the vertices of the second shortest path
            for (i = ab.index; i >= 0; i--) {
                if (ab.path[i] != -1)
                    printf("%d ", ab.path[i]);

```

```
                if (ab.path[i] == indexs + 2) {
                    printf("%d ", ab.path[i + 1]);
                    printf("%d ", ab.path[i]);
                }
            }
        } else {
            printf("There is no second shortest path");
        }
    } else {
        // Output the second shortest path and its vertices
        printf("The second shortest path is ");
        printf("%d\n", min);
        printf("The path is ");
        printf("1 ");
        // Output the vertices of the second shortest path
        // reason : for i=1 is that the first vertex is 1
        for (i = 0; i < path[minindex].index; i++) {
            printf("%d ", path[minindex].path[i]);
        }
    }

    return 0;
}
```

Declaration

I hereby declare that all the work done in this project titled "Autograd for Algebraic Expressions Report" is of my independent effort.