Project 3 Dijkstra Sequence

May 2, 2024

Contents

1 Chapter 1

We all know that Dijkstra's algorithm is a renowned method used for finding the shortest path from a source vertex to all other vertices in a weighted graph. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.

The algorithm maintains a set of vertices included in the shortest path tree. At each step, it selects the vertex not yet included in the set that has the minimum distance from the source and adds it to the set. Consequently, the algorithm generates an ordered sequence of vertices, termed the Dijkstra sequence, which represents the shortest paths from the source vertex to all other vertices in the graph.

OK. Today our task is to figure out whether an input sequence is a Dijkstra sequence. The input includes all the datas of a graph and the cases waiting to be tested. The expected output is simple, just print "Yes" for the Dijkstra Sequence, and "No" otherwise.

2 Chapter 2

Algorithm 1 Macro definitions and global variables

```
#define inf 101;
int distance[1001];
int test[1001];
int visited[1001];
```

Algorithm 2 Structure: Graph

```
struct Graph {
    int nv, ne;
    int e[1001][1001];
}
```

Algorithm 3 Function: create

```
function CREATE(nv, ne)
    g ← allocate memory for Graph
    g.nv ← nv
    g.ne ← ne
    for i ← 0 to nv do
        for j ← 0 to nv do
            g.e[i][j] ← inf
        end for
    end for
    return g
end function
```

Algorithm 4 Function: findmin

```
function FINDMIN(g)
    min ← inf
    minindex ← 0
    for i ← 0 to g.nv do
        if not visited[i] and distance[i] < min then
            min ← distance[i]
            minindex ← i
        end if
    end for
    return minindex
end function
```

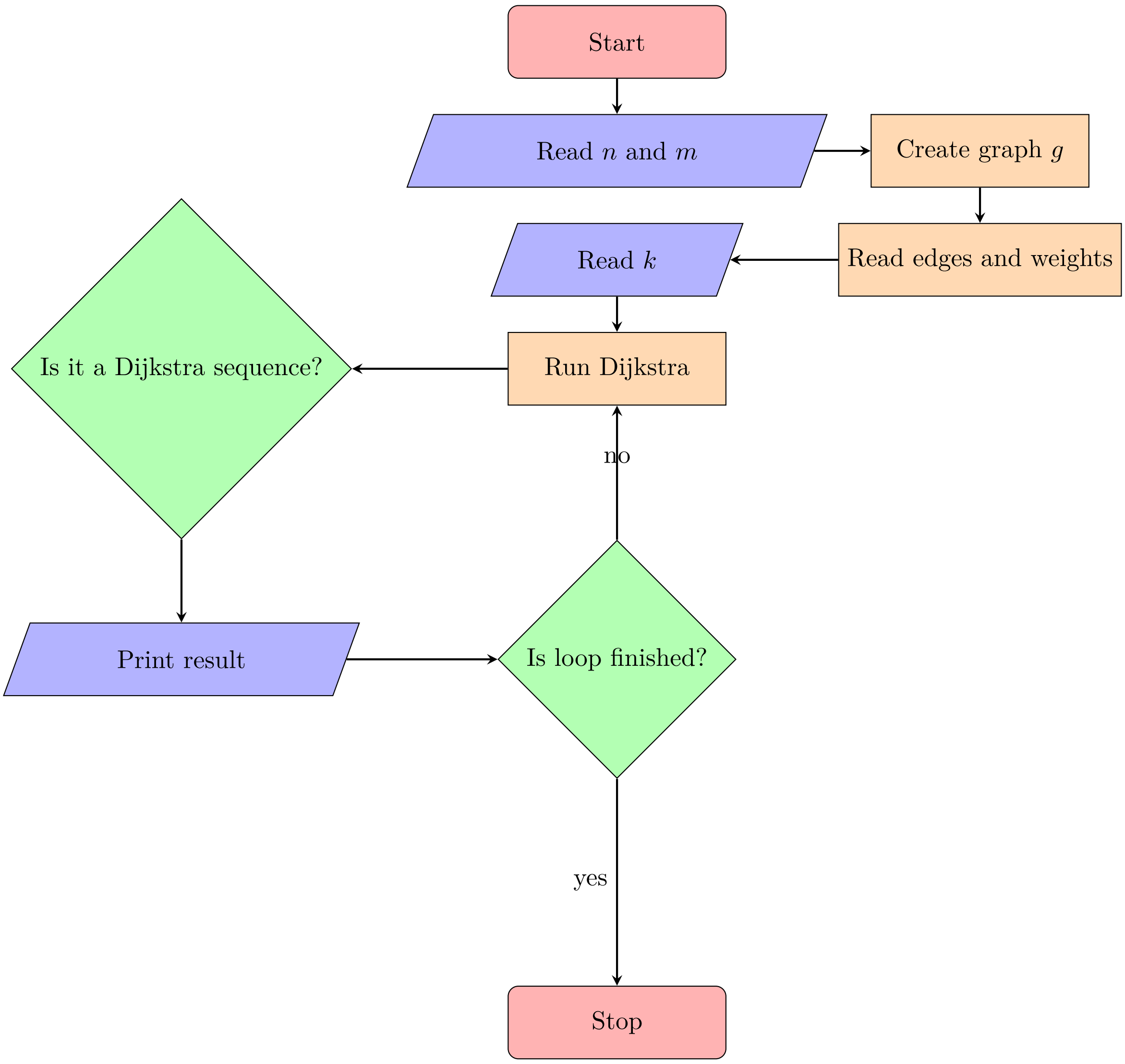
Algorithm 5 Function: dijkstra

```
function DIJKSTRA(g)
    Initialize distance array with infinity
    Initialize visited array to track visited vertices
    Initialize test array to store test case values
    for i ← 0 to g.nv do
        Read test case value kk and set test[i] ← kk − 1
        distance[i] ← inf
        visited[i] ← 0
    end for
    source ← test[0]
    distance[source] ← 0
    visited[source] ← 1
    for i ← 0 to g.nv do
        if i ≠ source and g.e[source][i] ≠ inf then
            distance[i] ← g.e[source][i]
        end if
    end for
    for i ← 1 to g.nv do
        currentVertex ← FINDMIN(g)
        if distance[currentVertex] ≠ distance[test[i]] then
            return 0
        end if
        visited[currentVertex] ← 1
        for j ← 0 to g.nv do
            if not visited[j] and g.e[currentVertex][j] ≠ inf then
                newDistance ← distance[currentVertex] + g.e[currentVertex][j]
                if newDistance < distance[j] then
                    distance[j] ← newDistance
                end if
            end if
        end for
    end for
    return 1
end function
```

Algorithm 6 Main Program

Read n and m
$g \leftarrow$ CREATE(n, m)
for $i \leftarrow 1$ **to** m **do**
  Read edge (a, b) and weight c
  $g.e[a-1][b-1] \leftarrow c$
  $g.e[b-1][a-1] \leftarrow c$
end for
Read k
for $i \leftarrow 1$ **to** k **do**
  $u \leftarrow$ DIJKSTRA(g)
  if $u = 1$ **then**
    Print "Yes"
  else
    Print "No"
  end if
end for

Below is the sketch of the main program:



3 Chapter 3

Table 1: Test Cases

Testing Number	Input	Expected Output	Testing Purpose	Actual Output
1	5 7 1 2 2 1 5 1 2 3 1 2 4 1 2 5 2 3 5 1 3 4 1 4 5 1 3 4 2 5 3 1 2 4 2 3 4 5 1 3 2 1 5 4	Yes Yes Yes No	Test whether the program can handle the sample correctly	Yes Yes Yes No
2	2 1 1 2 100 1 2 1	Yes	Test whether the program can handle small data sets correctly	Yes
3	1 0 1 1	Yes	The smallest scale of data	Yes

Table 2: Test Cases: Continued

Testing Number	Input	Expected Output	Testing Purpose	Actual Output
4	5 4 1 2 1 1 3 1 1 4 1 1 5 1 4 1 2 3 4 5 1 3 4 5 2 1 4 5 3 2 1 3 5 2 5	Yes Yes Yes Yes	All nodes are connected to the source node while the weights are the same	Yes Yes Yes Yes
5	5 6 1 2 2 1 3 4 2 3 1 2 4 7 3 5 3 4 5 2 7 2 1 3 4 5 2 3 1 5 4 3 2 1 5 4 3 5 4 2 1 4 5 3 2 1 4 1 2 3 5 5 4 3 2 1	No Yes Yes No Yes No Yes	Test whether the program can handle normal situation and scale data correctly	No Yes Yes No Yes No Yes

Table 3: Test Cases: Continued

Testing Number	Input	Expected Output	Testing Purpose	Actual Output
6	5 10 1 2 1 1 3 100 1 4 100 1 5 100 2 3 1 2 4 1 2 5 100 3 4 100 3 5 1 4 5 1 4 1 2 3 4 5 2 3 1 4 5 2 5 3 4 1 3 4 5 1 2	Yes Yes No No	In a scenario where weights are extremely unbalanced, the running status of the program	Yes Yes No No

Table 4: Test Cases: Continued

Testing Number	Input	Expected Output	Testing Purpose	Actual Output
7	Extremely huge scale of input, Close to the upper limit	Too many to be typed. In fact, randomly generated sequences in dense graphs have a high probability of not being ideal Dijkstra sequences	To test whether the program can run correctly undergo huge data flows	The same as expected. Due to the too much data, it cannot be typed out either.

4 Chapter 4

Here ,we're going to analyze the complexities of time and space of our program.

4.1 Time Complexity

4.1.1 Create Function

The time complexity of the create function is $O(n^2)$, where n is the number of vertices in the graph.Because we have two nested loops that iterate over all the vertices in the graph.

4.1.2 FindMin Function

The time complexity of the findMin function is $O(n)$, where n is the number of vertices in the graph.Because we have a loop that iterates over all the vertices in the graph.

4.1.3 Dijkstra Function

The time complexity of the Dijkstra function is $O(n^2)$, where n is the number of vertices in the graph.Because we have two nested loops that iterate over all the vertices in the graph.For each loop, we have a findMin function that iterates over all the vertices in the graph.And we have a loop that iterates over all the vertices in the graph.

4.1.4 Main Program

The time complexity of the main program is $O(n^2)$, where n is the number of vertices in the graph.Because we have a loop that iterates over all the vertices in the graph, and for each loop, we have a Dijkstra function that iterates over all the vertices in the graph.

4.2 Space Complexity

4.2.1 Create Function

The space complexity of the create function is $O(n^2)$, where n is the number of vertices in the graph.Because we have a two-dimensional array that stores the edges between the vertices in the graph.

4.2.2 FindMin Function

The space complexity of the findMin function is $O(1)$, because we only have a few variables that store the minimum distance and the index of the minimum distance while no extra space is used.

4.2.3 Dijkstra Function

The space complexity of the Dijkstra function is $O(1)$, because we only have a few variables that store the distance, visited vertices, and test case values while no extra space is used.

4.2.4 Main Program

The space complexity of the main program is $O(n^2)$, where n is the number of vertices in the graph.Because we have a two-dimensional array that stores the edges between the vertices in the graph.

5 Chapter 5 Source Code

```c
#include<stdio.h>
#include<stdlib.h>
#define inf 101

// Array to store the shortest distance from the source vertex
   to each vertex
int distance[1001];

// Array to store the test cases
int test[1001];

// Array to keep track of visited vertices
int visited[1001];

// Structure to represent a graph
typedef struct Graph {
    int nv; // Number of vertices
    int ne; // Number of edges
    int e[1001][1001]; // Adjacency matrix to store edge
       weights
} *graph;

// Function to create a graph
graph create(int nv, int ne) {
    graph g = (graph)malloc(sizeof(struct Graph));
    g->nv = nv;
```

```c
    g->ne = ne;

    // Initialize all edge weights to infinity
    for (int i = 0; i < nv; i++) {
        for (int j = 0; j < nv; j++) {
            g->e[i][j] = inf;
        }
    }
    return g;
}

// Function to find the vertex with minimum distance from the source vertex
int findmin(graph g) {
    int min = inf;
    int minindex = 0;
    for (int i = 0; i < g->nv; i++) {
        if (!visited[i] && distance[i] < min) {
            min = distance[i];
            minindex = i;
        }
    }
    return minindex;
}

// Function to perform Dijkstra's algorithm
int dijkstra(graph g) {
    int kk, z;

    // Read the test case values and initialize distance and visited arrays
    for (int i = 0; i < g->nv; i++) {
        scanf("%d", &kk);
        distance[i] = inf;
        visited[i] = 0;
        test[i] = kk - 1;
    }

    // Set the distance of the source vertex to 0 and mark it as visited
    distance[test[0]] = 0;
    visited[test[0]] = 1;

    // Update the distance array with the weights of the edges connected to the source vertex
    for (int i = 0; i < g->nv; i++) {
        if (i != test[0] && g->e[test[0]][i] != inf) {
            distance[i] = g->e[test[0]][i];
        }
    }

    // Perform Dijkstra's algorithm for the remaining vertices
```

```
    for (int i = 1; i < g->nv; i++) {
        z = findmin(g);

        // If the distance of the current vertex is not equal to the distance of the previous vertex, return 0
        if (distance[z] != distance[test[i]]) {
            return 0;
        }
        // Otherwise, the vertice could form a dijkstra sequence ,continue with the algorithm
        // Mark the current vertex as visited
        visited[test[i]] = 1;

        // Update the distance array with the weights of the edges connected to the current vertex
        for (int j = 0; j < g->nv; j++) {
            if (!visited[j] && g->e[test[i]][j] != inf && distance[test[i]] + g->e[test[i]][j] < distance[j]) {
                distance[j] = distance[test[i]] + g->e[test[i]][j];
            }
        }
    }
    return 1;
}

// Function to perform Dijkstra's algorithm with file input
int dijkstra_file(graph g, FILE* file){
    int kk, z;

    // Read the test case values from the file and initialize distance and visited arrays
    for (int i = 0; i < g->nv; i++) {
        fscanf(file, "%d", &kk);
        distance[i] = inf;
        visited[i] = 0;
        test[i] = kk - 1;
    }

    // Set the distance of the source vertex to 0 and mark it as visited
    distance[test[0]] = 0;
    visited[test[0]] = 1;

    // Update the distance array with the weights of the edges connected to the source vertex
    for (int i = 0; i < g->nv; i++) {
        if (i != test[0] && g->e[test[0]][i] != inf) {
            distance[i] = g->e[test[0]][i];
        }
    }
```

```

    // Perform Dijkstra's algorithm for the remaining vertices
    for (int i = 1; i < g->nv; i++) {
        z = findmin(g);

        // If the distance of the current vertex is not equal to the distance of the previous vertex, return 0
        if (distance[z] != distance[test[i]]) {
            return 0;
        }

        // Mark the current vertex as visited
        visited[test[i]] = 1;

        // Update the distance array with the weights of the edges connected to the current vertex
        for (int j = 0; j < g->nv; j++) {
            if (!visited[j] && g->e[test[i]][j] != inf && distance[test[i]] + g->e[test[i]][j] < distance[j]) {
                distance[j] = distance[test[i]] + g->e[test[i]][j];
            }
        }
    }
    return 1;
}

int main() {
    int m, n;
    FILE* file;

    // Ask the user for the input method
    printf("Enter 1 for manual input, 2 for file input: ");
    int method;
    scanf("%d", &method);
    getchar(); // This is to capture the newline character after entering the method

    if (method == 2) {
        // If the user selects file input, open the file
        file = fopen("test_data.txt", "r");
        if (file == NULL) {
            fprintf(stderr, "Failed to open the file\n");
            return EXIT_FAILURE;
        }
    }

    // Read the number of vertices and edges
    if (method == 1) {
        scanf("%d %d", &n, &m);
    } else {
```

```
            fscanf(file, "%d %d", &n, &m);
        }

        // Create a graph
        graph g = create(n, m);

        // Read the edges and their weights
        for (int i = 0; i < m; i++) {
            int a, b, c;
            if (method == 1) {
                scanf("%d %d %d", &a, &b, &c);
            } else {
                fscanf(file, "%d %d %d", &a, &b, &c);
            }
            g->e[a - 1][b - 1] = c;
            g->e[b - 1][a - 1] = c;
        }

        // Read the number of test cases
        int k,u;
        if (method == 1) {
            scanf("%d", &k);
        } else {
            fscanf(file, "%d", &k);
        }

        // Perform Dijkstra's algorithm for each test case
        for (int i = 0; i < k; i++) {
            if (method == 1) {
                u = dijkstra(g);
            } else {
                u = dijkstra_file(g, file);
            }
            if (u == 1) {
                printf("Yes\n");
            }// If the function returns 1, which means it is a probable Dijkstra Sequence, so print "Yes"
            else {
                printf("No\n");
            }// If the function returns 0, which means it is not a probable Dijkstra Sequence, so print "No"
        }
        if (method == 2) {
            fclose(file);
        }
        return 0;
    }
```

Listing 1: Main Program

```
    // generator.c
    #include <stdio.h>
    #include <stdlib.h>
```

```
#include <time.h>

// Function to generate and print out a random permutation of vertices
void printRandomPermutation(int n, FILE *file) {
    int a[n];
    for (int i = 0; i < n; i++) { // Fill the array with 'n' vertices
        a[i] = i + 1;
    }
    for (int i = n - 1; i > 0; i--) { // Shuffle array elements
        int j = rand() % (i + 1);
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }
    for (int i = 0; i < n; i++) { // Print the randomized array
        fprintf(file, "%d ", a[i]);
    }
    fprintf(file, "\n");
}

int main() {
    // Seed the random number generator to get different results each time
    srand(time(NULL));

    FILE *file = fopen("test_data.txt", "w");
    if (file == NULL) {
        fprintf(stderr, "Error opening file for writing test data.\n");
        return EXIT_FAILURE;
    }

    // Define maximum vertices and edges according to the problem statement
    int maxVertices = 1000;
    int maxEdges = 100000;

    int Nv = maxVertices; // Vertex count
    int Ne = rand() % (maxEdges - Nv + 1) + Nv; // Ensure at least Nv-1 edges

    fprintf(file, "%d %d\n", Nv, Ne);

    // Generating edges with random weights
    for (int i = 0; i < Ne; ++i) {
        int u = rand() % Nv + 1;
        int v = rand() % Nv + 1;
        while (u == v) { // Ensure u is not equal to v
            v = rand() % Nv + 1;
        }
```

```
            int weight = rand() % 100 + 1; // Weights between 1 and
                100
            fprintf(file, "%d %d %d\n", u, v, weight);
        }

        // Generating queries (sequences)
        int queries = rand()%100+1;
        fprintf(file, "%d\n", queries);
        // Generate and print random permutations
        for (int i = 0; i < queries; ++i) {
            printRandomPermutation(Nv, file);
        }

        fclose(file);

        printf("Test data generated successfully!\n");
        return EXIT_SUCCESS;
    }
```

Listing 2: Generator

6 Declaration

I hereby declare that all the work done in this project titled "Project 3:Dijkstra Sequence" is of my independent effort.