Project 3 Dijkstra Sequence

May 10, 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 Struct definition

```
typedef struct Node {
    int vertex;
    int distance;
};
```

Algorithm 5 Function to check non-decreasing order

```
function isUndecreases(a[], n)
        for i ← 0 to n − 2 do
            if a[i].distance > a[i + 1].distance then
                return 0;
        return 1;
end function
```

Algorithm 6 Struct definition for Heap

```
typedef struct Heap {
    Node a[100000];
    int heapSize;
} *heap;
```

Algorithm 7 Insert function

```
procedure Insert(h, vertex, distance)
    h.heapSize ← h.heapSize + 1
    i ← h.heapSize
    h.a[i].vertex ← vertex
    h.a[i].distance ← distance
    while i > 1 and h.a[i/2].distance > h.a[i].distance do
        temp ← h.a[i]
        h.a[i] ← h.a[i/2]
        h.a[i/2] ← temp
        i ← i/2
    end while
end procedure
```

Algorithm 8 ExtractMin function

```
procedure ExtractMin(h)
    minNode ← h.a[1]
    h.a[1] ← h.a[h.heapSize]
    h.heapSize ← h.heapSize − 1
    i ← 1
    while 2 × i ≤ h.heapSize do
        left ← 2 × i
        right ← 2 × i + 1
        min ← left
        if right ≤ h.heapSize and h.a[right].distance < h.a[left].distance then
            min ← right
        end if
        if h.a[i].distance ≤ h.a[min].distance then
            break
        end if
        temp ← h.a[i]
        h.a[i] ← h.a[min]
        h.a[min] ← temp
        i ← min
    end while
    return minNode
end procedure
```

Algorithm 9 CreateHeap function

function CreateHeap
  $h \leftarrow$ allocate memory for a Heap
  $h.heapSize \leftarrow 0$
  return h
end function

Algorithm 10 IsEmpty function

function IsEmpty(h)
  return $h.heapSize == 0$
end function

Algorithm 11 DijkstraHeap function

procedure DijkstraHeap($g, start$)
  $h \leftarrow$ CreateHeap()
  for $i \leftarrow 0$ **to** $g.nv - 1$ **do**
    $distance[i] \leftarrow inf$
    $visited[i] \leftarrow 0$
  end for
  $distance[start] \leftarrow 0$
  Insert($h, start, 0$)
  while not IsEmpty(h) **do**
    $minNode \leftarrow$ ExtractMin(h)
    $minVertex \leftarrow minNode.vertex$
    $visited[minVertex] \leftarrow 1$
    for $v \leftarrow 0$ **to** $g.nv - 1$ **do**
      if not $visited[v]$ **and** $g.e[minVertex][v] \neq inf$ **and** $distance[minVertex] + g.e[minVertex][v] < distance[v]$ **then**
        $distance[v] \leftarrow distance[minVertex] + g.e[minVertex][v]$
        Insert($h, v, distance[v]$)
      end if
    end for
  end while
  free(h)
end procedure

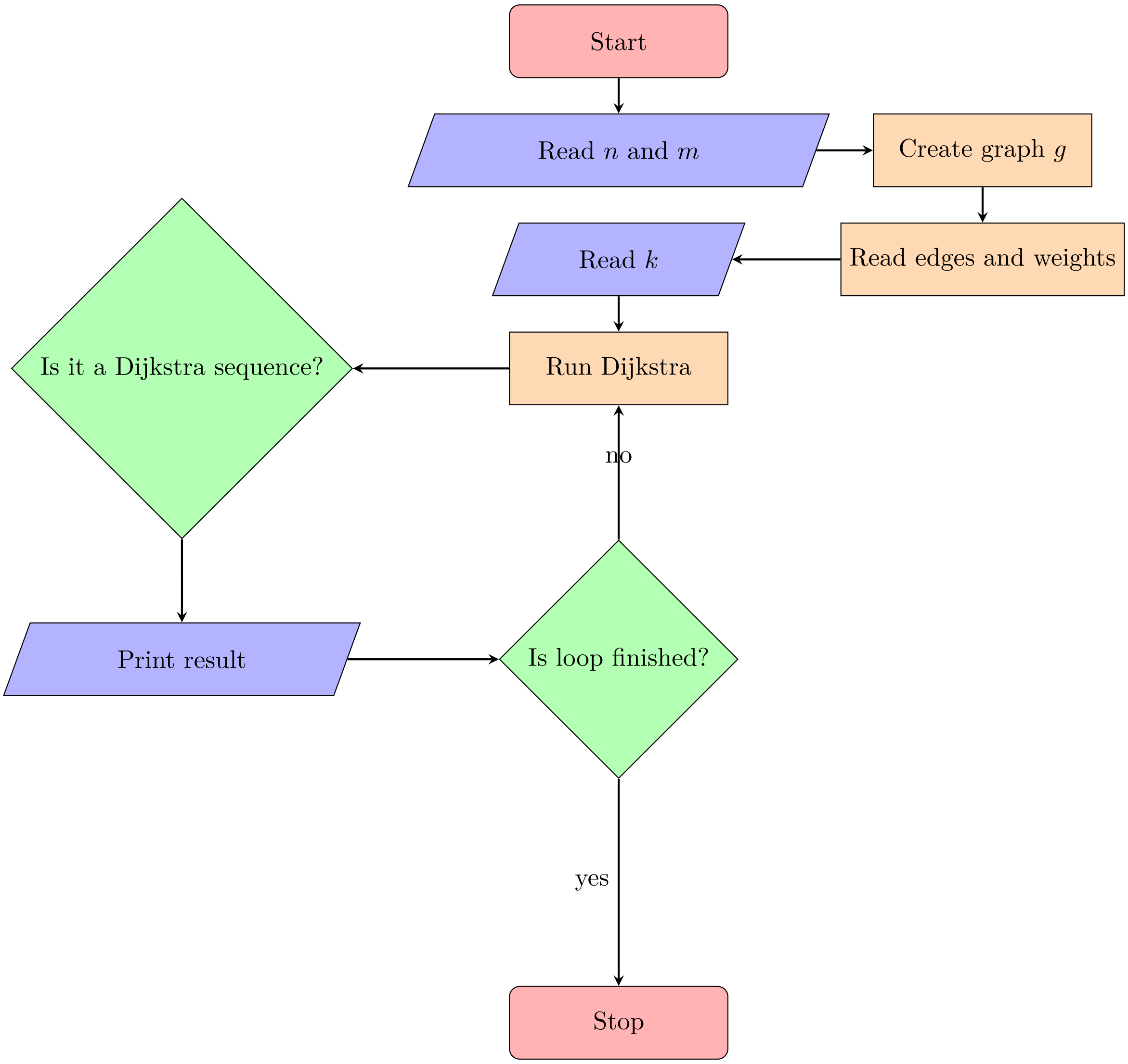
Algorithm 12 DijkstraFile function

```
procedure DijkstraFile(g, file, start)
    for i ← 0 to g.nv − 1 do
        distance[i] ← inf
        visited[i] ← 0
    end for
    distance[start] ← 0
    visited[start] ← 1
    h ← CreateHeap()
    Insert(h, start, 0)
    while not IsEmpty(h) do
        minNode ← ExtractMin(h)
        minVertex ← minNode.vertex
        visited[minVertex] ← 1
        for v ← 0 to g.nv − 1 do
            if not visited[v] and g.e[minVertex][v] ≠ inf and distance[minVertex] + g.e[minVertex][v] < distance[v] then
                distance[v] ← distance[minVertex] + g.e[minVertex][v]
                Insert(h, v, distance[v])
            end if
        end for
    end while
    free(h)
end procedure
```

Algorithm 13 Main function

```
function Main
    Declare variables: m, n, method, k
    Declare file pointer: file

    Ask the user for the input method
    Print "Enter 1 for manual input, 2 for file input: "
    Read method from input
    Consume the newline character

    if method = 2 then
        Open the file "test_data.txt" for reading: file ← fopen("test_data.txt", "r")
        if file = NULL then
            Print "Failed to open the file"
            return EXIT_FAILURE
        end if
    end if

    if method = 1 then
        Read n and m from input
    else
        Read n and m from file
    end if

    Create a graph: g ← create(n, m)

    for i ← 0 to m − 1 do
        Read integers a, b, c
        if method = 1 then
            Read a, b, c from input
        else
            Read a, b, c from file
        end if
        g.e[a − 1][b − 1] ← c
        g.e[b − 1][a − 1] ← c
    end for
    if method = 1 then
        Read k from input
    else
        Read k from file
    end if
    for i ← 0 to k − 1 do
        for j ← 0 to g.nv − 1 do
            if method = 1 then
                Read test[j] from input
            else
                Read test[j] from file
            end if
        end for
        if method = 1 then
            Perform Dijkstra's algorithm using heap: dijkstra_heap(g, test[0] − 1)
        else
            Perform Dijkstra's algorithm using file input: dijkstra_file(g, file, test[0] − 1)
        end if
        if is_undecreases(test, n) then
            Print "Yes"
        else
            Print "No"
        end if
    end for
    if method = 2 then
        Close the file: fclose(file)
    end if
end function
```

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 isUndecreases Function

The time complexity of the isUndecreases 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.

4.1.2 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.

4.1.3 Insert Function

The time complexity of the insert function is $O(\log n)$, where n is the number of vertices in the graph.Because we have a Minheap that maintains the minimum distance from the source vertex to each vertex.The insert function inserts a node into the heap and moves the element up the heap until the heap property is satisfied.

4.1.4 ExtractMin Function

The time complexity of the extractMin function is $O(\log n)$, where n is the number of vertices in the graph.Because we have a Minheap that maintains the minimum distance from the source vertex to each vertex.The extractMin function extracts the minimum element from the heap and moves the last element down the heap until the heap property is satisfied.

4.1.5 Dijkstra Function

Since we use heap to implement the Dijkstra algorithm, the time complexity of the Dijkstra function is $O(n \log n)$, where n is the number of vertices in the graph.Because we have a loop that iterates over all the vertices in the graph, and for each loop, we have an insert function that iterates over all the vertices in the graph.

4.1.6 Main Program

With the use of heap to implement the Dijkstra algorithm, the time complexity of the main program is $O(n^2)$, where n is the number of vertices in the graph.Because we need to initialize the graph and read the input data, and then perform the Dijkstra algorithm for each test case.However, the time complexity of the main program is mainly determined by the Dijkstra algorithm because the initialization and input reading are really fast.

4.2 Space Complexity

4.2.1 isUndecreases Function

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

4.2.2 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.

4.2.3 Insert Function

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

4.2.4 ExtractMin Function

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

4.2.5 Dijkstra Function

The space complexity of the Dijkstra function is $O(n)$, where n is the number of vertices in the graph.Because we have an array that stores the shortest distance from the source vertex to each vertex, an array that stores the test cases, and an array that keeps track of visited vertices and a heap that stores the distances of vertices.

4.2.6 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, an array that stores the shortest distance from the source vertex to each vertex, an array that stores the test cases, and an array that keeps track of visited vertices.

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
```

```c
    int ne; // Number of edges
    int e[1001][1001]; // Adjacency matrix to store edge weights
} *graph;

typedef struct Node{
    int vertex;
    int distance;
}node;

// Function to check if an array is in non-decreasing order
int isundecreaes(int *a,int n){
    for(int i=0;i<n-1;i++){
        if(distance[a[i]-1]>distance[a[i+1]-1]){
            return 0;
        }
    }
    return 1;
}

typedef struct Heap {
    node a[100000] ;
    int heapSize;
} *heap;

// Function to create a graph
graph create(int nv, int ne) {
    graph g = (graph)malloc(sizeof(struct Graph));
    g->nv = nv;
    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 insert a node into the heap
void insert(heap h, int vertex, int distance) {
    // Insert the vertex and distance into the heap
    h->heapSize++;
    int i = h->heapSize;
    h->a[i].vertex = vertex;
    h->a[i].distance = distance;

    // Move the element up the heap until the heap property is satisfied
    while (i > 1 && h->a[i / 2].distance > h->a[i].distance) {
        node temp = h->a[i];
        h->a[i] = h->a[i / 2];
        h->a[i / 2] = temp;
        i = i / 2;
    }
}

// Function to extract the minimum node from the heap
node extractMin(heap h) {
    // Extract the minimum element from the heap
    node minNode = h->a[1];
    h->a[1] = h->a[h->heapSize];
```

```c
        h->heapSize--;

        // Move the last element down the heap until the heap property is satisfied
        int i = 1;
        while (2 * i <= h->heapSize) {
            int left = 2 * i;
            int right = 2 * i + 1;
            int min = left;
            if (right <= h->heapSize && h->a[right].distance < h->a[left].distance) {
                min = right;
            }
            if (h->a[i].distance <= h->a[min].distance) {
                break;
            }
            node temp = h->a[i];
            h->a[i] = h->a[min];
            h->a[min] = temp;
            i = min;
        }
        return minNode;
    }

    // Function to create a heap
    heap create_heap() {
        heap h = (heap)malloc(sizeof(struct Heap));
        h->heapSize = 0;
        return h;
    }

    // Function to check if the heap is empty
    int is_empty(heap h) {
        return h->heapSize == 0;
    }

    // Function to perform Dijkstra's algorithm using heap
    void dijkstra_heap(graph g,int start) {
        // Create a min heap to store distances of vertices
        heap h = create_heap();

        // Initialize all distances to infinity
        for (int i = 0; i < g->nv; i++) {
            distance[i] = inf;
            visited[i] = 0;
        }

        // Set the distance of the source vertex to 0
        distance[start] = 0;

        // Insert the source vertex into the heap
        insert(h, start, 0);

        // Perform Dijkstra's algorithm
        while (!is_empty(h)) {
            // Extract the vertex with the minimum distance from the heap
            node minNode = extractMin(h);
            int minVertex = minNode.vertex;

            // Mark the extracted vertex as visited
            visited[minVertex] = 1;
```

```c

            // Update the distances of the adjacent vertices
            for (int v = 0; v < g->nv; v++) {
                if (!visited[v] && g->e[minVertex][v] != inf && distance[
                    minVertex] + g->e[minVertex][v] < distance[v]) {
                    // Update the distance of vertex v
                    distance[v] = distance[minVertex] + g->e[minVertex][v];

                    // Insert distance of vertex v into the heap
                    insert(h, v, distance[v]);
                }
            }
        }
        free(h);
    }

    // Function to perform Dijkstra's algorithm using file input
    void dijkstra_file(graph g, FILE* file,int start) {
        // Read the source vertex for the test case
        for (int i = 0; i < g->nv; i++) {
            distance[i] = inf;
            visited[i] = 0;
        }

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

        // Create a min heap to store distances of vertices
        heap h = create_heap();

        // Insert the source vertex into the heap
        insert(h, start, 0);

        // Perform Dijkstra's algorithm
        while (!is_empty(h)) {
            // Extract the vertex with the minimum distance from the heap
            node minNode = extractMin(h);
            int minVertex = minNode.vertex;

            // Mark the extracted vertex as visited
            visited[minVertex] = 1;

            // Update the distances of the adjacent vertices
            for (int v = 0; v < g->nv; v++) {
                if (!visited[v] && g->e[minVertex][v] != inf && distance[
                    minVertex] + g->e[minVertex][v] < distance[v]) {
                    // Update the distance of vertex v
                    distance[v] = distance[minVertex] + g->e[minVertex][v];

                    // Insert distance of vertex v into the heap
                    insert(h, v, distance[v]);
                }
            }
        }
        free(h);
    }

    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;
        }
        int k;
        // Read the number of test cases
        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++) {
            for(int j=0;j<g->nv;j++){
                if(method==1){
                    scanf("%d",&test[j]);
                }else{
                    fscanf(file,"%d",&test[j]);
                }
            }
            if (method == 1) {
                dijkstra_heap(g,test[0]-1);
            } else {
                dijkstra_file(g, file,test[0]-1);
            }
            if(isundecreaes(test,n)){
```

```c
            printf("Yes\n");
        } else {
            printf("No\n");
        }
    }

    if (method == 2) {
        fclose(file);
    }

    return 0;
}
```

Listing 1: Main Program

```c
// 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.