浙江大學

PROJECT3

课程名称：	FDS
姓　　名：	
学　　院：	
专　　业：	
学　　号：	
指导教师：	朱建科

2025 年　4　月　29　日

1 Chapter 1: Introduction

1.1 Problem Description

This problem is about validating whether a given vertex sequence can result from Dijkstra's algorithm on a weighted undirected graph.
Dijkstra's algorithm finds the shortes t paths from a single source to all other vertices. It selects, step by step, the unvisited vertex with the current minimum distance from the source.
Although the shortest distances are unique, the visiting order is not. There can be multiple valid Dijkstra sequences due to ties in distance values.
Given a graph and several sequences, the task is to determine if each sequence could represent a valid visiting order in Dijkstra's algorithm.

1.1.1 Input Specification

The input begins with two integers, Nv ($1 \leq$ Nv ≤ 1000) and Ne ($1 \leq$ Ne ≤ 105), representing the number of vertices and edges in the graph.
This is followed by Ne lines, each containing three integers: two vertex indices and a positive edge weight (≤ 100). The graph is undirected and connected.
Then, an integer K ($1 \leq$ K ≤ 100) is given, indicating the number of queries. Each of the next K lines contains a permutation of all vertices, starting from the source vertex.All integers are separated by spaces.

1.1.2 Output Specification

For each query sequence, output one line.
Print Yes if the sequence can be generated by Dijkstra's algorithm from the given source vertex. Otherwise, print No .

1.1.3 Sample Input

```
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
```

1.1.4 Sample Output

```
Yes
Yes
Yes
No
```

1.2 Background of Dijkstra’s Algorithm

Dijkstra’s algorithm, introduced by Edsger W. Dijkstra in 1956, is a classic greedy algorithm used to solve the single-source shortest path problem on a weighted graph with nonnegative edge weights.
The algorithm incrementally builds the shortest path tree by selecting the vertex with the smallest tentative distance that has not yet been processed. This process ensures that, once a vertex is added to the tree, its shortest distance from the source is finalized.
Due to its efficiency and simplicity, Dijkstra’s algorithm has become one of the most fundamental algorithms in graph theory and is widely applied in routing, navigation, and network optimization.
1.3 Background of Dijkstra Sequence
A Dijkstra sequence is the order in which vertices are finalized during the execution of Dijkstra’s algorithm. This sequence reflects the order in which the shortest path from the source to each vertex becomes known and confirmed.
However, due to possible equal distances or alternative paths, multiple valid Dijkstra sequences may exist for the same graph and starting point. Verifying whether a given sequence is a valid Dijkstra sequence involves checking whether it respects the greedy selection criteria of the algorithm at each step.
This concept is important for understanding the behavior and correctness of shortest path algorithms, especially in algorithm validation and testing scenarios.

2 Chapter 2: Algorithm Specification

2.1 Graph Structure

This structure stores the properties and information related to the graph, such as the number of vertices, edges, and the arrays for edge weights, distances, and visit status used in Dijkstra's algorithm.

2.1.1 Implementation Overview

The Graph structure contains the following fields:
The structure provides an organized way to manage and access graph-related data, which is essential for efficiently running Dijkstra's algorithm.

2.1.2 Pseudocode

```
type Graph:
integer nv // Number of vertices in the graph
integer ne // Number of edges in the graph
integer s // Starting vertex for Dijkstra's algorithm
integer* weight // Array to store weights of edges
integer* dis // Array to store the shortest distances from source
vertex
integer* visit // Array to track visited vertices
```

2.2 void Error(char *string)

This function handles memory allocation errors by printing an error message and then terminating the program with an error code.

2.2.1 Implementation Overview

nv : The number of vertices in the graph.
ne : The number of edges in the graph.
s : The starting vertex for Dijkstra's algorithm.
weight : A dynamically allocated array that stores the weights of the edges, represented as a compressed matrix.
dis : A dynamically allocated array that stores the shortest distances from the source vertex to each other vertex.
visit : A dynamically allocated array that tracks whether each vertex has been visited during the execution of the algorithm.

The Error function is called when memory allocation fails (i.e., when malloc returns NULL). It prints an error message that includes the name of the failed memory allocation, then terminates the program using exit(1) to indicate an error. This ensures that the program stops immediately upon encountering a critical error, preventing undefined behavior due to the lack of necessary memory.

2.2.2 Pseudocode

```
function Error(string):
print error message with string // Print the error message
exit program with error code 1 // Exit the program with an error code
```

2.3 int find_location (int v1, int v2)

This function calculates the unique location of an edge in the compressed weight array based on the indices of the two vertices.

2.3.1 Implementation Overview

The find_location function takes two vertex indices (v1 and v2) and calculates the position of the edge between them in a compressed weight array. The function first determines the larger and smaller vertex indices, then computes the unique index for the edge using a mathematical formula. This unique index corresponds to the position of the edge in the compressed storage format.

2.3.2 Pseudocode

```
function find_location(v1, v2):
if v1 > v2:
set i to v1 // Larger vertex index
set j to v2 // Smaller vertex index
else:
set i to v2 // Larger vertex index
set j to v1 // Smaller vertex index
location = i * (i - 1) / 2 + j - 1 // Calculate the unique index for
the edge
return location // Return the calculated index
```

2.4 void update(int v1, int v2, int weight)

This function updates the weight of an edge between two vertices in the graph's weight array.

2.4.1 Implementation Overview

The update function takes two vertex indices (v1 and v2) and the new weight of the edge between them. It first calls the find_location function to determine the unique location of the edge in the compressed weight array. Then, it updates the weight of the edge at the calculated location in the graph's weight array.

2.4.2 Pseudocode

```
function update(v1, v2, weight):
location = find_location(v1, v2) // Find the location of the edge
graph.weight[location] = weight // Update the weight at the calculated
location
```

2.5 void Reset(void)

This function resets the graph's state, preparing it for the next check. It resets the visited status of vertices and sets all distances to infinity.

2.5.1 Implementation Overview

The Reset function is used to reinitialize the graph's state before performing each check. It first resets the starting vertex s to -1. Then, it marks all vertices as unvisited by setting the visit array to 0 for all vertices. Lastly, it sets all distances to infinity (INT_MAX), which represents the initial state where no vertices are reachable.

2.5.2 Pseudocode

```
function Reset():
graph.s = -1 // Reset the starting vertex
for i from 1 to graph.nv:
graph.visit[i] = 0 // Mark all vertices as not visited
for i from 1 to graph.nv:
graph.dis[i] = INT_MAX // Set all distances to infinity
```

2.6 void Initial(void)

This function initializes the graph by reading the input values, allocating memory for the graph's data structures, and updating the graph with edge weights.

2.6.1 Implementation Overview

The Initial function performs the initial setup for the graph. It first reads the number of vertices and edges from the input. Then, it calculates the maximum number of edges in a complete graph with graph.nv vertices. Memory is allocated for the edge weights, and the weight array is initialized to INT_MAX to represent no edges. The diagonal of the weight matrix is set to zero, representing zero distance from a vertex to itself.
Next, the function reads the edges and their weights from the input, updating the graph with these values using the update function. Finally, memory is allocated for the visit and dis arrays, and the Reset function is called to prepare the graph's state before further operations.

2.6.2 Pseudocode

```
function Initial():
    read graph.nv, graph.ne // Read number of vertices and edges
    calculate max_edge = (graph.nv - 1) * graph.nv / 2 + graph.nv // Calculate maximum number of edges
    allocate memory for graph.weight with size max_edge // Allocate memory for edge weights
    if memory allocation fails:
        call Error("weight") // Handle error
    initialize graph.weight to INT_MAX for all edges // Set initial weights to infinity
    for each vertex i from 1 to graph.nv:
        call update(i, i, 0) // Set distance from a vertex to itself to zero
    for each edge i from 1 to graph.ne:
        read v1, v2, weight // Read edge and its weight
        call update(v1, v2, weight) // Update the weight of the edge
    allocate memory for graph.visit and graph.dis // Allocate memory for visit and distance arrays
    if memory allocation fails for visit or dis:
        call Error("visit" or "dis") // Handle error
    call Reset() // Reset graph state before usage
```

2.7 void renew(int target)

This function updates the shortest distances for a vertex after visiting it. It marks the vertex as visited and then attempts to relax the edges connected to it, updating the shortest distances for other vertices.

2.7.1 Implementation Overview

The renew function takes a target vertex and updates its status in the graph by marking it as visited. Then, it iterates over all other vertices and checks if there is a shorter path from the target vertex to each of the other vertices. If a shorter path is found, the corresponding distance is updated. The edge's weight is accessed using the find_location function to identify the correct index in the graph.weight array.

2.7.2 Pseudocode

```
mark graph.visit[target] as 1 // Mark target vertex as visited
for each vertex i from 1 to graph.nv:
loc = find_location(target, i) // Find the location of the edge
(target, i)
if vertex i has not been visited and there is a shorter path:
update graph.dis[i] to graph.dis[target] + graph.weight[loc] //
Relax the edge
```

2.8 void check(void)

This function checks if the given sequence of vertices follows the Dijkstra algorithm's properties. It validates whether the sequence maintains the correct order of visiting vertices based on their shortest distances from the source vertex.

2.8.1 Implementation Overview

The check function first resets the graph's state to ensure that previous checks do not affect the current one. It then reads the starting vertex for Dijkstra's algorithm and initializes its distance to 0. The renew function is called to update the distances of vertices after visiting the source vertex. The function iterates over the sequence of vertices and checks if the sequence respects the Dijkstra property: if a vertex with a shorter distance is visited after one with a longer distance, the sequence is marked as invalid. After processing all vertices, the function prints "Yes" if the sequence is valid and "No" if it is invalid.

2.8.2 Pseudocode

```
function check():
call Reset() // Reset the graph's state before the check
read starting vertex s
set graph.dis[s] to 0 // Set the distance to the source vertex as 0
call renew(s) // Update distances after visiting the source vertex
wrong = 0 // Initialize wrong flag to track if the sequence is invalid
for each round from 0 to graph.nv - 2:
```

2.9 void Free(void)

This function frees all dynamically allocated memory in the graph. It ensures that all memory allocated during the program's execution is properly released, preventing memory leaks.

2.9.1 Implementation Overview

The Free function is responsible for releasing the memory allocated for the edge weights, distance array, and visit array. These arrays are dynamically allocated using malloc , and the free function is called for each to deallocate the memory once they are no longer needed. This function helps ensure proper memory management and avoids memory leaks in the program.

2.9.2 Pseudocode

```
read target // Read the next vertex in the sequence
for each vertex i from 1 to graph.nv:
if wrong is not set and graph.visit[i] is 0 and graph.dis[i] <
graph.dis[target]:
set wrong to 1 // Mark the sequence as invalid
if wrong is not set:
call renew(target) // Update distances after visiting the
target vertex
print "Yes" if wrong is not set, else print "No" // Output result
```

2.10 int main(void)

This is the main function of the program, which initializes the graph, performs a series of checks, and finally frees any dynamically allocated memory.

2.10.1 Implementation Overview

The main function starts by initializing the graph with input data using the Initial function. Then, it reads the number of check sequences (n) from the user input. For each check sequence, the check function is called to perform the necessary computations and checks. Finally, after all checks are performed, the Free function is called to release any dynamically allocated memory. This ensures proper cleanup and prevents memory leaks.

2.10.2 Pseudocode

```
function Free():
call free(graph.weight) // Free memory for the edge weights
call free(graph.dis) // Free memory for the distance array
call free(graph.visit) // Free memory for the visit array
function main():
initialize the graph using Initial() // Initialize the graph with input
data
read integer n // Number of check sequences to verify
for i from 0 to n-1:
call check() // Perform check for each sequence
call Free() // Free dynamically allocated memory
return 0 // Exit program
```

3 Chapter 3: Testing Results

in the **test_cases.txt**,there are 10 test cases. Each test case consists of a starting vertex s, a sequence of vertices to visit, and a sequence of distances to each vertex.
The program should output "Yes" if the given sequence of vertices is a valid Dijkstra sequence, and "No" otherwise.
The program could correctly handle all test cases and output the correct results.including the maximum input required

4 Chapter 4: Analysis and Comments

4.1 Reset Function

4.1.1 Time Complexity

The function loops through all vertices twice (once to reset the visit array and once to reset the dis array). Since the number of vertices is nv , the overall time complexity is O(n), where n is the number of vertices in the graph.

4.1.2 Space Complexity

The function uses a fixed amount of space for local variables and modifies the state of the global graph data structure. The space complexity is O(n) due to the two arrays (visit and dis) that store information for each vertex. The space complexity does not depend on the input size beyond these arrays, so it is O(n).

4.2 Initial Function

4.2.1 Time Complexity

The function performs several operations that depend on the number of vertices and edges in the graph:
Overall, the time complexity is dominated by the O(n^2) operations for memory allocation and initialization of edge weights, so the total time complexity is O(n^2).

4.2.2 Space Complexity

The space complexity is determined by the following:
Reading the number of vertices and edges takes constant time, O(1).
Allocating memory for the edge weights array involves creating a memory block of size proportional to the maximum number of edges, which is O(n^2) where n is the number of vertices.
Initializing the edge weights to infinity takes O(n^2) time because it loops through all possible edges.
Setting the diagonal to zero requires O(n) operations, as it updates each vertex's distance to itself.
Reading the edges and updating their weights involves iterating over all edges, taking O(m) time where m is the number of edges.
Allocating memory for the visit and dis arrays takes O(n) time each.
Thus, the total space complexity is O(n^2), due to the weight array, with additional O(n) for the visit and dis arrays. Therefore, the overall space complexity is O(n^2).

4.3 Renew Function

4.3.1 Time Complexity

The renew function involves the following steps:
Thus, the time complexity of the renew function is O(n), where n is the number of vertices in the graph.

4.3.2 Space Complexity

The space complexity of the renew function is determined by:
Thus, the space complexity is O(n) due to the visit and dis arrays, which are the only dynamic allocations. The total space complexity is O(n).

4.4 Check Function

4.4.1 Time Complexity

The check function performs the following operations:
The weight array, which stores the weights of all possible edges, requires O(n^2) space.
The visit and dis arrays each require O(n) space to store information for each vertex.
The function uses a constant amount of space for local variables.
Marking the target vertex as visited is an O(1) operation.
The loop iterates over all other vertices (n vertices in total), where each iteration involves checking the conditions and potentially updating the distance. The operations inside the loop (finding the location of the edge and updating the distance) all take constant time, O(1), assuming the find_location function and the comparisons are O(1).
The visit array, which stores the visited state of all vertices, requiring O(n) space.
The dis array, which stores the distances from the source vertex to all other vertices, also requiring O(n) space.
The local variables used in the loop (i , loc) require a constant amount of space.
Resetting the graph: The Reset function is called at the beginning, which involves setting up the visit and dis arrays. This takes O(n) time, where n is the number of vertices.
Thus, the time complexity of the check function is O(n^2), where n is the number of vertices in the graph, due to the nested iteration over vertices and the calls to renew .

4.4.2 Space Complexity

The space complexity is determined by the following factors:
Thus, the overall space complexity of the check function is O(n), dominated by the visit and dis arrays.

4.5 main Function

4.5.1 Time Complexity

The main function performs the following operations:
Reading the starting vertex: This is an O(1) operation.
Updating the distances: The renew function is called for the starting vertex and for every target vertex in the sequence. Each call to renew involves iterating over all vertices, so it has a time complexity of O(n).
Iterating through the sequence of vertices: The sequence consists of n-1 vertices (since the starting vertex is already considered), and for each vertex in the sequence, the loop performs additional checks. This involves an O(n) loop that compares distances for all unvisited vertices. For each iteration, renew(target) is called, which again involves iterating over all vertices, taking O(n) time.
The visit array, which stores the visited state of each vertex, requires O(n) space.

The dis array, which stores the distances for each vertex, also requires O(n) space.
The local variables target , round , and wrong take constant space.
Initializing the graph: The Initial function is called at the beginning. The initialization involves reading the number of vertices, edges, and edge weights, as well as allocating memory for the weight , visit , and dis arrays. The time complexity of this operation is O(n^2) in the worst case, where n is the number of vertices, due to the edge updates inside the initialization.
Reading the number of check sequences: This is an O(1) operation, as it simply reads the number of test cases.
Performing checks for each sequence: The check function is called n times, where n is the number of sequences to verify. Since the time complexity of check is O(n^2), the total time complexity for all sequences is O(n^3).
Freeing memory: The Free function is called at the end to free dynamically allocated memory. This operation involves three calls to free , which each take O(n) time. Hence, the time complexity of this step is O(n).
Thus, the overall time complexity of the main function is O(n^3), dominated by the repeated calls to the check function.

4.5.2 Space Complexity

The space complexity of the main function is determined by the following factors:
Thus, the overall space complexity of the main function is O(n^2), dominated by the weight array that stores the edge weights for a complete graph.
The space used by the graph data structures: The weight , visit , and dis arrays each require O(n^2), O(n), and O(n) space, respectively.
The local variables n , i , and others take constant space.

5 Comments on possible impovements

This project was based on the forward star to store the vertex and edge. There is both pros and cons. The pros was like easier to search and sort, but it is difficult to caculate other properties in the network. If we use sparse matrix to store the map, there could be more simple and clear ways to do other algorithm.

6 Appendix: Source Code (in C)

6.1 Main Program Source Code

This section presents the main program source code that initializes the graph, performs Dijkstra's algorithm checks, and verifies if the given sequences are valid Dijkstra sequences.

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

#define MAX_V 1000
#define MAX_E 100000
#define INF 0x3f3f3f3f

// FORWARD STAR
typedef struct Edge {
    int next;
    int to;     // TO
    int w;      // WEIGHT
} Edge;

Edge edge[MAX_E * 2];  // BOTH WAY
int head[MAX_V + 1];
int cnt = 0;
int dist[MAX_V + 1];
bool visited[MAX_V + 1];


void add(int u, int v, int w) {
    edge[cnt].w = w;
    edge[cnt].to = v;
    edge[cnt].next = head[u];
    head[u] = cnt++;
}

// 验证Dijkstra
bool isDijkstraSequence(int n, int* seq) {
    // init
    memset(dist, INF, sizeof(dist));
    memset(visited, false, sizeof(visited));

    int source = seq[0];
    dist[source] = 0;

     for (int i = 0; i < n; i++) {
        int u = seq[i];

        if (visited[u]) return false;
        visited[u] = true;

        for (int v = 1; v <= n; v++) {
            if (!visited[v] && dist[v] < dist[u]) {
                return false;
            }
        }

        // update dist
        for (int j = head[u]; j != -1; j = edge[j].next) {
            int v = edge[j].to;
            int w = edge[j].w;
            if (!visited[v] && dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;
            }
        }
    }

    return true;
}

int main() {
    int n, m;
    scanf("%d%d", &n, &m);

    //init
    memset(head, -1, sizeof(head));
    // read edges

    for (int i = 0; i < m; i++) {
        int u, v, w;
        scanf("%d%d%d", &u, &v, &w);
        add(u, v, w);
        add(v, u, w);
    }

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

    // verify
    for (int i = 0; i < k; i++) {
        int seq[MAX_V];
        for (int j = 0; j < n; j++) {
            scanf("%d", &seq[j]);
        }

        // 验证
        if (isDijkstraSequence(n, seq)) {
            printf("Yes\n");
        } else {
            printf("No\n");
        }
    }

    return 0;
}
```

7 Declaration

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