Transportation Hub

LU Jialiang

Thesis submitted to
Zhejiang University

in partial fulfilment for the course

FUNDAMENTAL DATA STRUCTURE
COMPUTER SCIENCE

College of Information Science & Electronic Engineering
https://github.com/Riverlu233

April 29, 2025

Contents

Chapter 1 Introduction

1.1 Problem Description

Given the map of a country, there could be more than one shortest path between any pair of cities. A transportation hub is a city that is on no less than k shortest paths (the source and the destination NOT included). Your job is to find, for a given pair of cities, the transportation hubs on the way.

1.2 Background

The question can be transformed into an abstract graph problem, that is, given a pair of vertices, find all the shortest paths between them, and then find out the vertices whose number of occurrences on those paths meet or exceed a certain threshold. The question itself already shows a real application , that is the question of where a transport hub should be located between two places with a high traffic flow. In addition, we can imagine that some mathematical problems that can be transformed into similar graph problems like this and they can also be solved with this algorithm.

Chapter 2 Algorithm Specification

2.1 General Description

Main Data Structure: Graph

The design idea of the algorithm is very simple and consists of three main parts. The first part is used to process the input and convert the input graph structure into an adjacency matrix, which is easier to process.Given the simplicity of the input part, the detailed process description is skipped here, and the specific implementation can be referred to the source code in Appendix A or refer to the source code in the 'code' file.

The second part is to simplify the graph, where some properties of Dijkstra-algorithm are exploited, so that we can ensure that certain nodes must not appear in the shortest path. Then we can ignore these nodes in the subsequent processing, that is, the size of the graph is reduced, so that the processing is more efficient.

The last part is to find the vertices that satisfy the conditions in the simplified graph. This part is done by Dijkstra algorithm and two recursive algorithms. Firstly the shortest distance between any two nodes on the graph is derived by the Dijkstra algorithm and is stored in the form of a shortest distance matrix. Then the first recursive algorithm is used to find the number of shortest paths for each node to reach the destination. The second recursive algorithm uses the number of shortest paths from each node to the destination to derive the number of occurrences of each node on the shortest path. Finally, it is determined whether any node satisfying the condition exists and output.

2.2 Symplification Algorithm

To illustrate the method of simplification, it is first necessary to state the following lemma:

Lemma 1 *If the shortest path from A to B is longer than or equals to the shortest path from A to C, then B must not be on the shortest path from A to C.*

This lemma is almost obvious. We can use the method of contradiction. Assuming that B is on the shortest path from A to C, then this shortest path is divided into two parts: A to B and B to C. That is, this path is the path from A to B plus the path from B to C. However, in this problem, there is no negative weight, so the path from A to C is greater than the path from A to B, which is contradictory! So the null hypothesis is not valid, Lemma 1 has been proven.

Based on the above lemma, we execute Dijkstra Algorithm once (set source to be the start vertex) and check all vertices in the return result. If the vertex have a longger shortest-path to the destination than from the source or equal to it, then we should remove them from the graph to obtain a new adjacency matrix.The following is the pseudocode for this section:
Pseudo-Code:

```
function simplify_map(original_graph, distance_table, destination_node) -> graph*:
    # Phase 1: Node Filtering and Mapping
    new_name_map = ARRAY[original_graph.num] # Original -> Compressed index
    new_index_map = ARRAY[original_graph.num] # Compressed -> Original index
    preserved_count = 0

    for each node i in original_graph:
        if distance_table.dist[i] >= distance_table.dist[destination_node] AND i !=
            destination_node:
            MARK node i as invalid (distance_table.state[i] = 0)
        if node i is valid (distance_table.state[i] == 1):
            # Create bidirectional mapping
            new_name_map[i] = preserved_count
            new_index_map[preserved_count] = i
            preserved_count += 1

    # Phase 2: Create Compressed Graph (if needed)
    if preserved_count < original_graph.num:
        # Initialize new graph structure
        compressed_graph = init_graph(preserved_count)
        compressed_graph.name = new_name_map
        compressed_graph.index = new_index_map

        # Build compressed adjacency matrix
        for each row i in 0 to preserved_count-1:
            original_row = new_index_map[i]
            for each column j in 0 to preserved_count-1:
                original_col = new_index_map[j]
                compressed_graph.matrix[i][j] =
                    original_graph.matrix[original_row][original_col]
        return compressed_graph
    else:
        # No compression needed, return original
```

2.3 The Finding Algorithm

The first step of this algorithm is to establish a shortest path length matrix, where the elements in the i-th row and j-th column store the length of the shortest path required from vertex i to node j. Here, this matrix is completed by using Dijkstra Algorithm for each vertex in the simplified graph. After building the matrix, we must introduce the following lemma to demonstrate the effectiveness of the following two recursive algorithms:

Lemma 2 *If the shortest path length from A to C is equal to the sum of the shortest path from A to B and the shortest path from B to C, then there must be a shortest path from A to C that places B on this path.*

Lemma 3 *The number of the shortest paths from A to C is equal to the sum of the shortest paths from adjacent vertices of A to C that can be located on the shortest path from A to C.*

The correctness of Lemma 2 can be considered as follows: combining the shortest path from A to B and the shortest path from B to C, we obtain a path from A to C. And we know that the length of this path is equal to the shortest path from A to C. Therefore, the path we construct is the one of the shortest paths from A to C, and B is on this path.

Lemma 3 goes further on the basis of Lemma 2. The set of shortest paths from A to C that satisfy Lemma 2 must be on the shortest path from A to C. Therefore, the set of shortest paths from A to C can definitely be divided into shortest paths from A to different adjacent vertices and then to C, and these sets are disjoint. Therefore, the potential of a set is equal to the sum of the potentials of each subset under a partition, and Lemma 3 is proven.

The first recursive algorithm operates based on Lemma 2 and 3, recursively finding the number of shortest paths from each vertex in the graph to the endpoint. The recursion starts at the source, while the exit of the recursion is to reach the destination, when it returns 1 indicating the existence of a path. And in this algorithm, each node will only be updated once. After the shortest path count of the vertex is updated once, the state flag will be changed, so that it will not be mistakenly increased due to being called by different vertices.The following is the pseudocode for this algorithm:

Pseudo-Code:

```
function count_ways(graph, dist_matrix, count_array, current_node, target_node,
    remaining_dist) -> int:
    # Base case: reached target node
    if current_node == target_node:
        return 1

    # Initialize path counter for current node
    path_count = 0
    # Explore all possible neighbors
    for neighbor in 0 to graph.num-1:
        # Skip invalid paths and self-loops
        if graph.matrix[current_node][neighbor] == INFTY or neighbor ==
            current_node:
            continue
        # Check if this neighbor lies on a shortest path
        required_condition = (dist_matrix[neighbor][target_node]
                          + graph.matrix[current_node][neighbor]
                          == remaining_dist)
        # Process unvisited nodes that satisfy condition
        if required_condition and count_array[current_node].state == NOT_VISITED:
            # Recursive depth-first search
            sub_paths = count_ways(graph, dist_matrix, count_array,
                                   neighbor, target_node,
                                   remaining_dist - graph.matrix[current_node][neighbor])
            # Accumulate path counts
            path_count += sub_paths
    # Update memoization state and store result
    count_array[current_node].state = VISITED
    count_array[current_node].ways = path_count
    return path_count
```

On the other hand, the second recursive algorithm is used to obtain the number of occurrences in all shortest paths. Its entrance is also the source, and its recursive exit is also the destination. But the difference is that this function does not return a result, as it determines its occurrence in the shortest path based on the number of times it is called. When it is called once, it means that a vertex will use it as a successor vertex to generate a shortest path. If the number of shortest paths from that vertex to the destination is n, then n different shortest paths will definitely be generated in this call. And different vertices' calls to it mean different paths, so each call must accumulate the shortest path from itself to the destination, and finally we can obtain the answer.The following is the pseudocode for this algorithm:

Pseudo-Code:

```
FUNCTION count_times(graph, dist_matrix, count_array, current_node, target_node,
    remaining_dist):
    # Base case: reached destination
    IF current_node == target_node:
        return
    # Traverse all potential paths
    for neighbor in 0 to graph.num-1:
        # Validate edge existence and non-loopback
        if graph.matrix[current_node][neighbor] == INFTY or neighbor ==
            current_node:
            continue
        # Verify shortest path condition
        path_condition = (dist_matrix[neighbor][target_node]
                          + graph.matrix[current_node][neighbor]
                          == remaining_dist
        if path_condition:
            # Recursive depth-first exploration
            count_times(graph, dist_matrix, count_array,
                        neighbor, target_node,
                        remaining_dist - graph.matrix[current_node][neighbor])
    # Backtrack accumulation - critical step
    count_array[current_node].times += count_array[current_node].ways
```

Chapter 3 Testing Results

Instructions: Due to the difficulty in describing the figures, each example is visualized here, and the specific shapes are labeled in Appendix A. Readers can refer to and view them.

3.1 The choice of the test cases

Based on the difference between the following two directions, we set up different test cases and the scenarios they test are shown in the header of the table.Since increasing the character storage space in the tree nodes simply allows for longer variable names, the space sizes here are all set to 4.

Case	Test Purpose	Treshold&Input	Output	Test Results
1	Examples	$3\&\{1,6\},\{7,0\},\{5,5\}$	{2,3,4,5},{None},{None}	Pass
2	Sparse Graph	$1\&\{1,3\},\{1,4\}$	{2},{0}	Pass
3	Acyclic Graph	$2\&\{1,3\},\{1,4\}$	{None},{None}	Pass
4	Dense Graph	$2\&\{0,5\},\{1,4\}$	{1,2,3,4},{0,2,3,5}	Pass
5	Complete Graph	$3\&\{1,3\},\{1,4\}$	{None},{None}	Pass
6	Diff Lengths for 4	$2\&\{0,5\},\{1,4\}$	{1,2,3,4},{2}	Pass
7	Big Graph	$1\&\{4,57\}\{34,22\},$	{0,1,5,9,60},{26,30}	Pass

Chapter 4 Analysis & Comments

Conclusions: The time complexity of the whole algorithm is $O(V^3)$, while the space complexity is $\Theta(V^2)$.

4.1 The Read-in Algorithm

4.1.1 Time Complexity

The operations in the read-in section mainly include the initialization of the structure and the structuring of the input data. Assuming that the input graph has V vertices and E edges, the initialization of the structure involves assigning initial values to the adjacency matrix, with a complexity of $\Theta(V)$. The structuring operation mainly updates the adjacency matrix, with a complexity of $\Theta(E)$. Therefore, the total complexity is $\Theta(V+E)$. The worst-case occurs when the graph is dense, i.e. $E=\Theta(V^2)$, which means the complexity at this time is $O(V^2)$.

4.1.2 Space Complexity

In the input part, the vast majority of storage space is consumed in the representation of the adjacency matrix. In addition, there is a portion of constant space used to support the operation of the algorithm and store feature quantities of graphs that are independent of the size of the graph. Therefore, it is easy to obtain the space complexity at this time is $\Theta(V^2)$.

4.1.3 Optimization Direction

One improved idea is to use hash tables to store graphs, which has the advantage of saving space, considering that in most cases graphs are sparse. In addition, we can see that in the second part of the differentiation diagram, we can save a lot of effort because it is troublesome to remove rows and columns in matrix, but it is easy to implement in hashing.
But this also has obvious drawbacks, that is, the readability of the code will be greatly reduced and the complexity of the code will be significantly increased. This is because at this point, we need to add a lot of judgment conditions to the traversal steps, but in the third part, whether in Dijkstra Algorithm or recursive algorithm, the number of traversal times is very high.

4.2 The Symplificattion Algorithm

4.2.1 Time Complexity

The specific steps of simplification include traversing the table T returned by Dijkstra Algorithm, establishing a new adjacency matrix, and establishing the correspondence between the new graph and the old graph. The complexity of traversing table T is obviously $\Theta(V)$. And the establishment of corresponding relationships is included in the same loop with constant steps, so the complexity is also $\Theta(V)$.
However, the establishment steps of the new adjacency matrix depend on how many vertices will be optimized, so we can only give an upper bound (the lower bound is obviously constant time, in the case where the new graph only contains 2 vertices). In the worst case, no node will be optimized, which means that the new graph and the old graph will be exactly the same (of course, in this algorithm, we will not build a new adjacency matrix in this case, but as long as one vertex is optimized, a new adjacency matrix will be established, so it can be assumed that the worst case is when the two are exactly the same). The complexity of establishing a new adjacency matrix at this point is $O(V^2)$, which means assigning a one-to-one value to each element in the matrix. In summary, the time complexity of this part of the algorithm is $O(V^2)$.

4.2.2 Space Complexity

Similar to time complexity, the size of the newly generated adjacency matrix for problem-solving depends on how many vertices can be optimized. In the worst-case scenario, the complexity is also $O(V^2)$. In addition to storing the adjacency matrix, the new graph also stores a new set of corresponding relationships. Since this mapping is not one-to-one, its size is determined by the larger one,apparently the old graph, so the complexity is always $\Theta(V)$. Overall, the complexity is $O(V^2)$.

4.2.3 Optimization Direction

In addition to the method mentioned earlier of using hashing to store graphs and reduce the number of operations during simplification, we can also consider more powerful simplification rules. For example, in Dijkstra Algorithm, since we have actually completed the traversal of the sizes of all edges, we can find a minimum edge and a maximum edge. As long as the shortest distance from a node to the starting point and the shortest distance from the starting point to the endpoint do not fall within this range, then that node should also be discarded (in fact, it is a reinforcement of Lemma 1).

4.3 The Finding Algorithm

4.3.1 Time Complexity

Obviously, this result is related to the size of the simplified new image, and we are still discussing the worst-case where the old and new graphs are completely correlated. In the process of establishing the shortest distance table, we apply the Dijkstra algorithm to each vertex, and combined with the discussion in the simplification algorithm section, the complexity reaches $O(V^3)$.
The complexity of the two recursive algorithms depends not only on the number of vertices in the graph, but also on the shape of the graph. The denser the graph, the more times recursion needs to be performed, because a vertex is called by adjacent vertices more times. The worst-case scenario is that the diagram is completely connected, meaning that any two nodes are connected. At this point, each node will be called by every node except for itself, so the complexity is $O(V * V) = O(V^2)$. Therefore, the time complexity is $O(V^3)$.

4.3.2 Space Complexity

The spatial complexity of the final step depends on the array **count** that stores the final result and the recursive process. Due to the one-to-one correspondence between **count** and each node, its complexity is $\Theta(V)$. For recursive algorithms, any node will definitely reach the endpoint after passing through at most $V - 1$ times, so in the worst case, there are V recursive functions called at the same time, and the variables in each function only occupy constant space, so its complexity is $O(V)$. Overall, the spatial complexity of this algorithm is $\Theta(V)$.

4.3.3 Optimization Direction

The first part is to establish the shortest distance table. Actually, given that this is an undirected graph, the table must be symmetrical, so this feature can be considered to save some steps and storage space. Specifically, before each execution of Dijkstra Algorithm, it is possible to consider marking some vertices with known minimum distances, but this cannot avoid the problem of the shortest path for possible vertices in the remaining fixed vertices starting from these vertices. In addition, if a BDS algorithm combined with backtracking can be directly adopted, it may have better performance.

Chapter A Pictures of testing cases

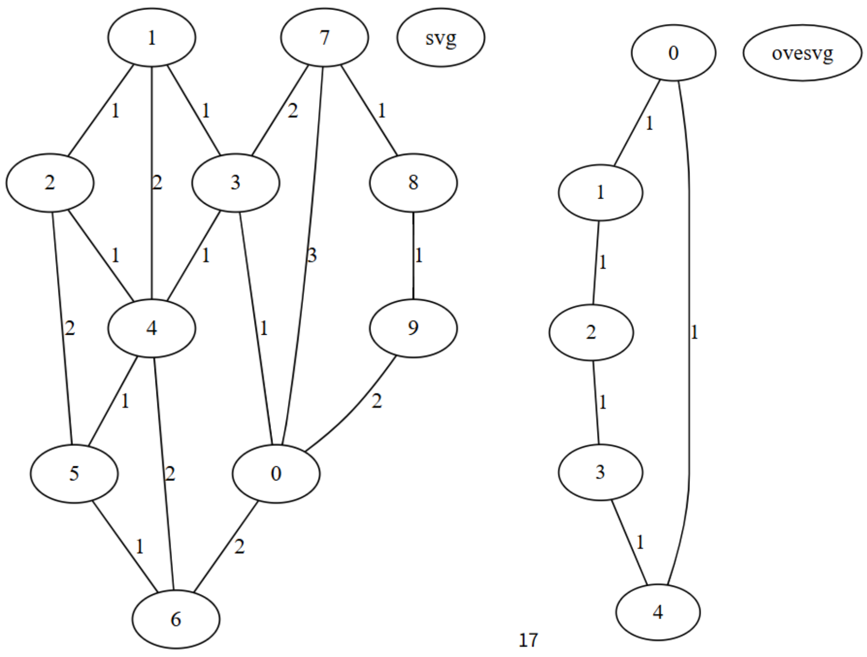

Figure 1: Case 1-2

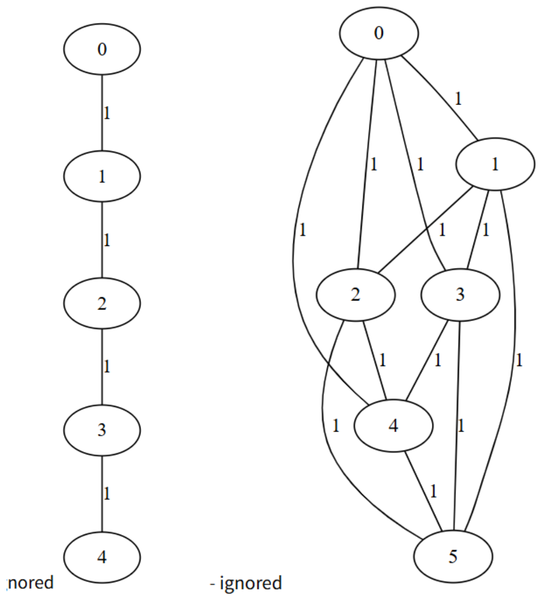

Figure 2: Case 3-4

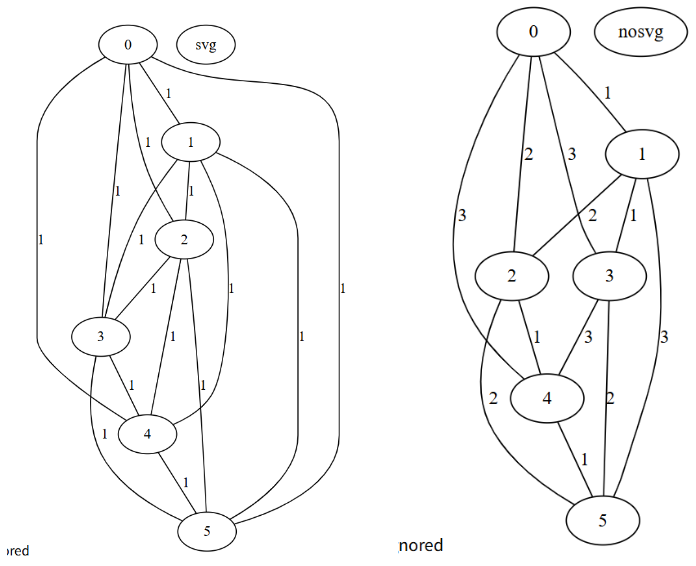

Figure 3: Case 5-6

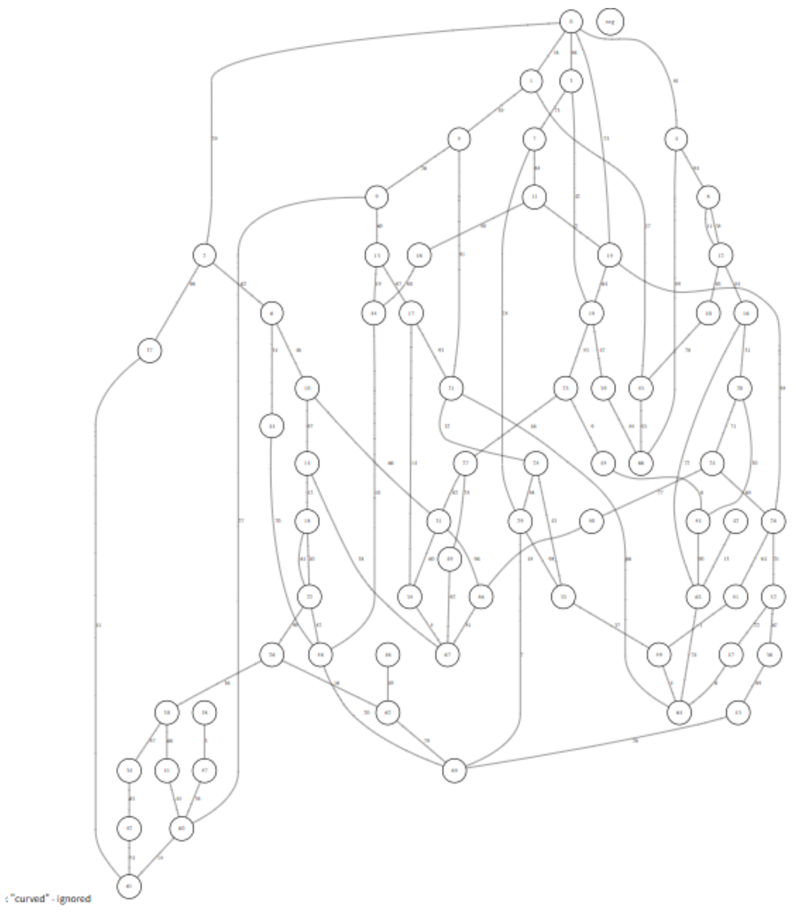

Figure 4: Case 7

Chapter B Source Code in C

```c
#include <stdio.h>
#include <stdlib.h>
#define INFTY 50000 //much bigger than the length of the sum of the paths

typedef struct Graph{
   int num; //to save the numbers of vertices
   int* name; //old map to new map
   int* index; //new map to old map
   int** matrix; //adjacent matrix
}graph;

typedef struct Table{
   int* dist; //to save the distance from source
   int* state; //to save the state that whether a vertex has found its shortest
      path
}table;

typedef struct Path{
   int state; //whether the number of the shortest ways to the destination has
      been cnfirmed
   int ways; //the number of the shortest ways to the destination
   int times; //times appearing on the shortest paths from source to destination
}path;

graph* init_graph(int vert_num); //init a graph with vert_num vertices
void free_graph(graph* map); //free the space of the graph

table* init_table(int ver_num); //init a table with ver_num vertices
void free_table(table* T); //free the space of the table

int** init_matrix(int length); //init an int matrix length times length
void free_matrix(int** matrix,int length); //free the matrix

void update_path(graph* map,int path_num); //to update the length of the path in
   adjacent matrix

graph* simplify_map(graph* map,table* T,int des); //simplify the graph
int find_Tmin(table* T,int num); //find the vertex which has smallest diistance
   and hasn't been confirmed

table* Dijkstra(graph* map,int src); //Dijkstra Algorithm

void find_mindist(graph* map,int** T_matrix);
//to update the matrix that save the shortest distance between any two vertices

int count_ways(graph* map,int** T_matrix,path** count,int src,int des,int dist);
//count the number of shortest paths to the destination
```

```c
void count_times(graph* map,int** T_matrix,path** count,int src,int des,int dist);
//count the times appearing on the shortest paths from source to destination
void find_tph(graph* map,int src,int des,int threshold);
//find eligible transportation hubs

int main(){
    int city_num,path_num,threshold; //def the variables
    int source,destination,pair_num;
    int i,j;
    scanf("%d %d %d",&city_num,&path_num,&threshold); //read-in
    graph* map=init_graph(city_num); //init graph
    update_path(map,path_num); //update graph
    scanf("%d",&pair_num);
    for (i=0;i<pair_num;i++){
        scanf("%d %d",&source,&destination); //for each pair, find transportation
            hubs
        find_tph(map,source,destination,threshold);
    }
    free_graph(map); //free the map
}

graph* init_graph(int vert_num){
    graph* new_graph=(graph*)malloc(sizeof(graph));
    if (!new_graph){
        printf("The initialization failed.");
        exit(-1);
    }
    int i;
    new_graph->num=vert_num;
    new_graph->name=(int *)malloc(sizeof(int)*vert_num);
    new_graph->index=(int *)malloc(sizeof(int)*vert_num);
    for (i=0;i<vert_num;i++){
        new_graph->name[i]=i; //default: old map and new map are as the same
        new_graph->index[i]=i;
    }
    new_graph->matrix=init_matrix(vert_num); //init the adjacent matrix
    return new_graph;
}

void free_graph(graph* map){
    free_matrix(map->matrix,map->num);
    free(map->name);
    free(map->index);
    free(map);
    return;
}

table* init_table(int ver_num){
    int i;
    table* T=(table*)malloc(sizeof(table));
    T->dist=(int*)malloc(sizeof(int)*ver_num);
    T->state=(int*)malloc(sizeof(int)*ver_num);
    for (i=0;i<ver_num;i++){
        T->dist[i]=INFTY; //default: All is infty
        T->state[i]=0; //default:hasn't been checked
    }
    return T;
}

void free_table(table* T){
    free(T->dist);
    free(T->state);
    free(T);
}

int** init_matrix(int length){
    int i,j;
    int** matrix=(int**)malloc(sizeof(int*)*length);
    for (i=0;i<length;i++){
        matrix[i]=(int*)malloc(sizeof(int)*length);
        for (j=0;j<length;j++){
            if (i==j){ //default: to itself is 0, to others is infty
                matrix[i][j]=0;
            }
            else{
                matrix[i][j]=INFTY;
            }
        }
    }
    return matrix;
}

void free_matrix(int** matrix,int length){
    int i;
    for (i=0;i<length;i++){
        free(matrix[i]);
    }
    free(matrix);
}

path** init_path(int num){
    int i;
    path** new_path=(path**)malloc(sizeof(path*)*num);
    for (i=0;i<num;i++){
        new_path[i]=(path*)malloc(sizeof(path));
        new_path[i]->state=0; //default:hasn't been confirmed, both ways and times
            are 0
        new_path[i]->ways=0;
        new_path[i]->times=0;
    }
    return new_path;
}

void free_path(path** tar_path,int num){
    int i;
    for (i=0;i<num;i++){
        free(tar_path[i]);
    }
    free(tar_path);
}

void update_path(graph* map,int path_num){
    int i;
    int src,des,length;
    for (i=0;i<path_num;i++){
        scanf("%d %d %d",&src,&des,&length); //read in the length from src to des
        map->matrix[src][des]=length; //update the distance between src & des
        map->matrix[des][src]=length;
    }
    return;
}

graph* simplify_map(graph* map,table* T,int des){
    int i,j,new_num=0;
    int* new_name=(int*)malloc(map->num*sizeof(int));
    int* new_index=(int*)malloc(map->num*sizeof(int));
    //ready to build the new grapgh
    for (i=0;i<map->num;i++){
        if (T->dist[i]>=T->dist[des]&&i!=des){
            T->state[i]=0;
        }
        if (T->state[i]){
            new_name[i]=new_num; //i-th vertex in name:old[i]->new[new_num]
            new_index[new_num]=i; //Reversely,new[new_num]->old[i]
            new_num++; //count the number of vertices in the new graph
        }
    }
    graph* new_map=init_graph(new_num); //make a new graph,avoid destructing the
        old graph
    free(new_map->index); //to clear the default case
    free(new_map->name);
    new_map->name=new_name; //to save the correct case
    new_map->index=new_index;
    for (i=0;i<new_num;i++){
        for (j=0;j<new_num;j++){
            //copy the effective path from the old to the new
            new_map->matrix[i][j]=map->matrix[new_index[i]][new_index[j]];
            new_map->matrix[j][i]=map->matrix[new_index[j]][new_index[i]];
        }
    }
    return new_map;
}


int find_Tmin(table* T,int num){
    int i;
    int dist_min=INFTY;
    int index_min=-1;
    for (i=0;i<num;i++){
        //scan the table to find the one having the samllest distance
        if (T->dist[i]<dist_min&&T->state[i]==0){ //ensure that the vertex hasn't
            been checked
            index_min=i;
            dist_min=T->dist[i];
        }
    }
    return index_min;
}

table* Dijkstra(graph* map,int src){
    int i,u,v;
    table* T=init_table(map->num); //init a table to save the result
    T->dist[src]=0;
    for (i=0;i<map->num;i++){
        u=find_Tmin(T,map->num);
        if (u==-1){ //means there are only infty remains, none of them can be
            connected
            printf("The map is not connected.Results are as followed.\n");
            return T; //just return because it can't continue anymore, it is
                exactly the result
        }
        T->state[u]=1;
        for (v=0;v<map->num;v++){ //traverse all the nodes
            if ((map->matrix[u][v]<INFTY)&&(T->state[v]==0)){ //u,v are connected &
                v hasn't been confirmed
                if (T->dist[u]+map->matrix[u][v]<T->dist[v]){ //find a shorter path
                    T->dist[v]=T->dist[u]+map->matrix[u][v]; //update a shorter path
                }
            }
        }
    }
    return T;
}

void find_mindist(graph* map,int** T_matrix){
    int i,j;
    table* T=NULL;
    for (i=0;i<map->num;i++){ //traverse all vertices
        T=Dijkstra(map,i); //set i to be the source, find the shortest path to the
            left each
        for (j=0;j<map->num;j++){
            T_matrix[i][j]=T->dist[j]; //update the result in to the shortest path
                matrix
        }
        free_table(T);
    }
}

int count_ways(graph* map,int** T_matrix,path** count,int src,int des,int dist){
    if (src==des){ //find the destination->possible paths +1
        return 1;
    }
    int i;
    for (i=0;i<map->num;i++){
        if (map->matrix[src][i]!=INFTY&&i!=src&&count[src]->state==0){
            //only when src and i are adjacent and i!=src and src hasn't been
                checked,we update it
            if (T_matrix[i][des]+map->matrix[src][i]==dist){
                //lemma 2, it shows i must be on one of the shortest paths form src
                    to des
                count[src]->ways+=count_ways(map,T_matrix,count,i,des,dist-map->matrix[src][i]);
                //lemma 3,recursively calculate the number of shortest paths to des
            }
        }
    }
    count[src]->state=1; //after checked ,update the state
    return count[src]->ways; //when it is called, return the ways
}

void count_times(graph* map,int** T_matrix,path** count,int src,int des,int dist){
    if (src==des){
        return; //meet the des means exit the recursion
    }
    int i;
    for (i=0;i<map->num;i++){
        if (map->matrix[src][i]!=INFTY&&i!=src){ //the same condition
            if (T_matrix[i][des]+map->matrix[src][i]==dist){
                count_times(map,T_matrix,count,i,des,dist-map->matrix[src][i]);
                //recursively call the adjacent vertex
            }
        }
    }
    count[src]->times+=count[src]->ways;
    //when one vertex is called, the times should plus its ways.Explanation can be
        seen in pdf
    return;
}

void find_tph(graph* map,int src,int des,int threshold){
    int i,flag=1;
    table* T=Dijkstra(map,src);
    graph* sf_map=simplify_map(map,T,des);
    free_table(T);
    path** count=init_path(sf_map->num);
    int** T_matrix=init_matrix(sf_map->num);
    find_mindist(sf_map,T_matrix);
    count[sf_map->name[src]]->ways=count_ways(sf_map,T_matrix,count,sf_map->name[src],sf_map->
    count_times(sf_map,T_matrix,count,sf_map->name[src],sf_map->name[des],T_matrix[sf_map->nam
    for (i=0;i<sf_map->num;i++){
        //find the eligible transportation hubs
        if
            (count[i]->times>=threshold&&sf_map->index[i]!=src&&sf_map->index[i]!=des){
            if (flag){
                printf("%d",sf_map->index[i]);
                flag=0;
            }
            else{
                printf(" %d",sf_map->index[i]);
            }
        }
    }
    if (flag){
        printf("None\n");
    }
    else{
        printf("\n");
    }
    free_matrix(T_matrix,sf_map->num); //free the space
    free_path(count,sf_map->num);
    free_graph(sf_map);
}
```

Chapter C Declaration

I hereby declare that all the work done in this project titled ”Transportation Hub” is of my independent effort.