Fundamentals of Data Structures

Laboratory Projects

Ambulance Dispatch

周承扬 **3190102371**

Date: 2021-11-25

Chapter 1: Introduction

This project is designed to help the city's ambulance system respond faster.

In such situation, we first input 2 integers "a" and "b", which respectively represent the total pick-up spots and ambulance dispatch centers. And the next line the i-th integer means the i-th dispatch center has i ambulances. Next input an integer "c" on behalf of the total streets between them, with following lines consisting of the 2 ends of the street with the time taken to pass the street. In the end, input the integer "d", followed by d indices of pick up spots.

In the output, we need to find the fastest way to send the ambulance, output the rescue route with the total time. If there are 2 same situation, we choose the dispatch which has more ambulances. But if all the ambulances are working, we can only output "All Busy".

Chapter 2: Algorithm Specification

The algorithm of the program is based on the classic Dijkstra algorithm.

When the map is given, the paths from the pick-up spots to the ambulance dispatch center are determined. Then we can find the best path from each ambulance dispatch center to pick-up spots by order.

If several centers have equally good path, we choose the one with more ambulances. And when all the ambulance are sent to spots, we just need to output "All Busy".

Chapter 3: Testing Results

The test sample can be completed:

```
7 3
3 2 2
16
A-1 2 4
A-1 3 2
3 A-2 1
4 A-3 1
A-1 4 3
6 7 1
1 7 3
1 3 3
3 4 1
6 A-3 5
6 5 2
5 7 1
A-2 7 5
A-2 1 1
3 5 1
5 A-3 2
8
6 7 5 4 6 4 3 2
A-3 5 6
4
A-2 3 5 7
3
A-3 5
2
A-2 3 4
2
A-1 3 5 6
5
A-1 4
3
A-1 3
2
All Busy
```

Simple sample for same best path:

```
1 2
2 1
2
A-1 1 1
A-2 1 1
1
1
A-1 1
1
```

Chapter 4: Analysis and Comments

According to the Dijkstra algorithm, the time complexity of the program is $T = O(Na * ((Na + Ns)^2 + M))$.

The space complexity of the program is $T = O(Ns * Na * ((Na + Ns)^2 + M))$.

Appendix: Source Code (in C)

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

```c
typedef struct nodeT* nodePtr;
typedef struct pathT* pathPtr;

//define the structure of nodes and paths
struct nodeT {
    int cars;
    int* distances;//the length betweenthe nodes
    pathPtr* paths;
};

struct pathT {
    int* node;
    int size;
};

void InputMap(nodePtr *nodes, int centerCnt, int spotCnt);
void handleCall(int spot, nodePtr *nodes, int centerCnt, int spotCnt);
int compareDistance(nodePtr* nodes, int startNode, int middleNode, int
endNode);
void createPath(nodePtr* nodes, int startNode, int middleNode, int endNode,
int distance);
void printPath(int startNode, pathPtr path, int centerCnt);

int main() {

    int centerCnt, spotCnt, calls, spot, i, j;
    nodePtr *nodes;

    scanf("%d %d", &spotCnt, &centerCnt); //the number of pick-up spots and
ambulance dispatch centers
    nodes = (nodePtr *)malloc(sizeof(nodePtr) * (spotCnt + centerCnt));

    for(i=0; i<spotCnt+centerCnt; i++) {
        nodes[i] = (nodePtr)malloc(sizeof(struct nodeT));
        nodes[i]->distances = (int *)malloc(sizeof(int) * (spotCnt +
centerCnt));
        memset(nodes[i]->distances, 0, sizeof(int) * (spotCnt +
centerCnt));
        nodes[i]->paths = (pathPtr *)malloc(sizeof(pathPtr) * (spotCnt +
centerCnt));
        memset(nodes[i]->paths, 0, sizeof(pathPtr) * (spotCnt +
centerCnt));
        for(j=0; j<spotCnt+centerCnt; j++) {
            nodes[i]->paths[j] = (pathPtr)malloc(sizeof(struct pathT));
```

```
            nodes[i]->paths[j]->node = (int *)malloc(sizeof(int) * (spotCnt
+ centerCnt));
            nodes[i]->paths[j]->size = 0;
        }
    }

    InputMap(nodes, centerCnt, spotCnt);

    scanf("%d", &calls); //the total calls for ambulances

    for(i = 0; i < calls; i++) {
        scanf("%d", &spot);
        handleCall(spot+centerCnt-1, nodes, centerCnt, spotCnt);
    }
    return 0;
}

//process the input data
void InputMap(nodePtr *nodes, int centerCnt, int spotCnt) {
    int i, streets, startNode, endNode, distance;
    char node[4];

    for(i=0; i<centerCnt; i++) {
        scanf("%d", &nodes[i]->cars);
    } //the number of ambulances in every dispatch center

    for(i=0; i<spotCnt; i++) {
        nodes[centerCnt+i]->cars = 0;
    }

    scanf("%d", &streets); //the number of total streeets
    for(i=0; i<streets; i++) {
        getchar();
        scanf("%[A0-9-]", node); //the input is restricted in A and 0-9 and
-
        if(node[0] == 'A') {
            startNode = node[2] - '1';
        } else {
            startNode = node[0] - '1' + centerCnt;
        }
        getchar();
        scanf("%[A0-9-]", node);
        if(node[0] == 'A') {
            endNode = node[2] - '1';
```

```
        } else {
            endNode = node[0] - '1' + centerCnt;
        } //process every one input street

        scanf("%d", &distance);//input the distance between 2 nodes
        createPath(nodes, startNode, -1, endNode, distance); //`1 menas
without middle node
        createPath(nodes, endNode, -1, startNode, distance);
    }
}

//caculate the most time saving ambulance sending
void handleCall(int spot, nodePtr *nodes, int centerCnt, int spotCnt) {
    int i, j, k, m, min = 101, mini = -1;
    int minDistance, minIndex[centerCnt+spotCnt], minSize;

    for(i=0; i<centerCnt; i++) {
        if(nodes[i]->cars == 0) continue; //if a center has no ambulance left,
change another one

        m = 0;
        while(1) { //Dijkstra algorithm
            minDistance = 101; minSize = 0;
            for(j=0; j<centerCnt+spotCnt; j++) {
                if(minDistance > nodes[i]->distances[j] &&
nodes[i]->distances[j] > m) {
                    minDistance = nodes[i]->distances[j];
                    minIndex[0] = j;
                    minSize = 1;
                } else if(minDistance == nodes[i]->distances[j]) {
                    minIndex[minSize++] = j;
                }
            }

            if(nodes[i]->distances[spot] > minDistance) {
                for(j=0; j<minSize; j++) {
                    if(compareDistance(nodes, i, minIndex[j], spot)) {
                        createPath(nodes, i, minIndex[j], spot, 0);
                    }
                }
            } else if(nodes[i]->distances[spot] == minDistance) {
                break; //already find the shortest path
            } else {
                for(j=0; j<minSize; j++) {
```

```
                    for(k=0; k<centerCnt+spotCnt; k++) {
                        if(i != k && minIndex[j] != k &&
compareDistance(nodes, i, minIndex[j], k)) {
                            createPath(nodes, i, minIndex[j], k, 0);
                        }
                    }
                }
            }
            m = minDistance;
            if( m==101 ) break; //if there is no path, exit
        }

        if((nodes[i]->distances[spot] > 0 && nodes[i]->distances[spot] <
min)
        || (nodes[i]->distances[spot] == min && (mini == -1 ||
nodes[i]->cars > nodes[mini]->cars
            || (nodes[i]->cars == nodes[mini]->cars
                && nodes[i]->paths[spot]->size <
nodes[mini]->paths[spot]->size)))) {
            min = nodes[i]->distances[spot]; //update the shorter path
            mini = i;
        }
    }

    if(mini == -1) {
        printf("All Busy\n");
    } else {
        nodes[mini]->cars--; //successfully send an ambulance
        printPath(mini, nodes[mini]->paths[spot], centerCnt);
        printf("%d\n", nodes[mini]->distances[spot]); //output the
shortest time
    }

}
//judege if the path need to be updated
int compareDistance(nodePtr* nodes, int startNode, int middleNode, int
endNode) {
    return nodes[middleNode]->distances[endNode] > 0 &&
(nodes[startNode]->distances[endNode] == 0
        || nodes[startNode]->distances[endNode] >
nodes[startNode]->distances[middleNode]
        + nodes[middleNode]->distances[endNode]
        || (nodes[startNode]->distances[endNode] ==
nodes[startNode]->distances[middleNode]
```

```
        + nodes[middleNode]->distances[endNode]
        && nodes[startNode]->paths[endNode]->size >
nodes[startNode]->paths[middleNode]->size
        + nodes[middleNode]->paths[endNode]->size)); //e.g. A->B->C is
better than A->C, update the path
}

// create a path between 2 nodes
void createPath(nodePtr* nodes, int startNode, int middleNode, int endNode,
int distance) {
    int i;
    if(middleNode < 0) { //without middle node
        nodes[startNode]->distances[endNode] = distance;
        nodes[startNode]->paths[endNode] = (pathPtr)malloc(sizeof(struct
pathT));
        nodes[startNode]->paths[endNode]->size = 1;
        nodes[startNode]->paths[endNode]->node = (int
*)malloc(sizeof(int));
        nodes[startNode]->paths[endNode]->node[0] = endNode;
    } else { //with middle node
        nodes[startNode]->distances[endNode] =
nodes[startNode]->distances[middleNode]
        + nodes[middleNode]->distances[endNode];

        nodes[startNode]->paths[endNode]->size =
nodes[startNode]->paths[middleNode]->size
        + nodes[middleNode]->paths[endNode]->size;
        for(i=0; i<nodes[startNode]->paths[middleNode]->size; i++) {
            nodes[startNode]->paths[endNode]->node[i] =
nodes[startNode]->paths[middleNode]->node[i];
        }
        for(i=0; i<nodes[middleNode]->paths[endNode]->size; i++) {
            nodes[startNode]->paths[endNode]->node[i+nodes[startNode]->p
aths[middleNode]->size] =
            nodes[middleNode]->paths[endNode]->node[i];
        }
    }

}

//output the chosen path
void printPath(int startNode, pathPtr path, int centerCnt) {
    int i;
    if(startNode < centerCnt) {
```

```
        printf("A-%d", startNode + 1);
    } else {
        printf("%d", startNode - centerCnt + 1);
    }
    for(i=0; i<path->size; i++) {
        if(path->node[i] < centerCnt) {
            printf(" A-%d", path->node[i] + 1);
        } else {
            printf(" %d", path->node[i] - centerCnt + 1);
        }
    }
    printf("\n");
}
```

Declaration

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