# Ambulance Dispatch

刘思锐

**Date: 2021-11-28**

# Chap.1-Introduction

There’re many streets and some Ambulance Dispatch Centers (ADC) in our city. Once you receive an emergency call, how can you tell which dispatch center should send an ambulance and which way you should take to get the spot in minimum time? To make the algorithm more realistic, the ambulances of each dispatch center is limited. Maybe sometimes you have to send ambulance from a further dispatch center because the nearest one has sent all ambulances. It is the case this project trying to solve.

For each case, all the inputs in a line are separated by a space. The first line contains two positive integers ***Ns***(≤1000) and ***Na***(≤9), which are the total number of pick-up spots and the number of ambulance dispatch centers, respectively. The next line gives Na non-negative integers, where the i-th integer is the number of available ambulances at the i-th center. All the integers are no larger than 100.

In the next line a positive number ***M***(≤10000) is given as the number of streets connecting the spots and the centers. Then ***M*** lines follow, each contains three parameters, the name of two ends of the path and the time it takes, respectively. The pick-up spots are named from ***1*** to ***Ns***, and the ambulance dispatch centers are named from ***A−1*** to ***A−Na***. The time must be an integer no larger than 100. You should make sure that your graph is well connected and there’s only one path directly from one place to another.

Finally the number of emergency calls, ***K***, is given as a positive integer no larger than 1000, followed by ***K*** indices of pick-up spots.

If there’s still valid ambulance, output the nearest way from an ADC to the calling spot and the time it will take to get there. If not, output “All Busy” instead.

# Chap.2-Algorithm Specification

It’s basically the Dijkstra algorithm. Every time you pick up the nearest unknown vertex in the graph Update its information and update the info of vertexes adjacent to it. Repeat it until connected vertexes have all been updated. Pseudocode follows.

```
void
Dijkstra ( table T )
{
    vertex v, w;
    for( ; ; )
    {
        v = smallest unknown distance vertex;
        if( no available vertex)
            break;

        T[v].known = true;
        for each w adjacent to v
            if( !T[w].known )
                if( T[v].Distance + Cvw < T[w].Distance )
                {
                    update w;
                }
    }
}
```

The change in my program is, I need arrays to store the minimum time and the shortest path from a calling spot to many different dispatch centers and run Dijkstra algorithm ***Na*** times and store the result respectively.

The data structures and the purpose of each components are followed. *verList[numADC + numVer]* is unique, which stores all vertexes info. Each vertex in *verList[]* has its own *adjVerList* node to represent the paths and adjacent vertexes. To simplify the index, elements in arrays are numbered starting from 1. *XXX[0]* is always unavailable in this program.

```
struct AdjVer {
    int ver;//adjacent vertex (aka calling spots)
    int dist;//the distance to ver
    struct AdjVer *next;
};
struct Vertex {
    int *path;//path[i] stores the last step of the nearest way to A-i
    int *dist;//dist[i] stores the distance to A-i
    int *known;//used in dijkstra algorithm
    struct AdjVer *adjVerList;//
};
struct Graph {
    int numADC;//number of ambulance dispatch centers
    int numVer;//number of vertex
    int *avlAmb;//number of available ambulances in each center
    struct Vertex *verList;
    /*
     * ambulance dispatch centers are verList[1]~verList[numADC]
     * calling spots are verList[numADC+1]~verList[numADC+numVer]
     */
};
```

# Chap.3-Testing Result

Example 1 is an ordinary test case.

```
//input 1
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

//output 1
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
```

Example 2 is also an ordinary testcase based on testcase 1.

```
//input 2
7 3
3 4 2
16
A-1 2 4
A-1 5 7
3 A-1 1
4 A-3 7
A-1 4 3
6 3 4
2 7 3
5 3 3
3 4 2
6 A-3 5
6 5 4
5 7 8
A-2 6 5
A-2 1 3
3 5 7
5 A-3 2
10
6 7 5 4 6 4 3 2 7 1

//output 2
A-1 3 6
5
A-1 2 7
7
A-3 5
```

```
2
A-1 4
3
A-2 6
5
A-3 4
7
A-2 6 3
9
A-2 6 3 A-1 2
14
A-2 6 5 7
17
All Busy
```

Example 3 is a test case that has a very small size.

```
//input 3
1 1
15
1
A-1 1 3
1
1

//output 3
A-1 1
1
```

Example 4 is a test case that has very large size.

```
//input 4
100 10
10 10 10 10 10 10 10 10 10 10
110
//(110 lines of edges are omitted here)
5
32 76 65 53 24

//output 4
A-3 31 32
7
A-2 74 75 76
11
A-2 74 73 72 71 70 69 68 67 66 65
26
A-4 45 46 47 48 49 50 51 52 53
```

```
24
A-5 24
5
```

# Chap.4-Analysis and Comment

There’re ***Na + Ns*** elements in array *verList[]*. Each vertex in it needs ***3Na*** space in all to store the path, the distance and the known flag for each ADC. The space I need to represent the paths is ***2M***, for it is an undirected graph so paths should be represented in both two directions. The total space complexity is ***O( (Na + Ns) * Na + M )***.

The time complexity of Dijkstra Algorithm is ***(Ns + Na)$^2$*** which is declared in our textbook. As we need to run Dijkstra Algorithm ***Na*** times. The time complexity of this part is ***Na * (Na + Ns)$^2$***. Initialization takes a constant time. Handling input takes a linear time. The time of output depends on how many vertexes are there in paths. But it must be a fraction of the time of Dijkstra Algorithm because we need to get that path trough it. The total time complexity is ***O( Na * (Na + Ns)$^2$ )***.

# Appendix-Source Code

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

#define MaxStrSize (30)
#define Infinity (1000000)
#define Invalid (-1)

struct AdjVer {
    int ver;//adjacent vertex (aka calling spots)
    int dist;//the distance to ver
    struct AdjVer *next;
};
struct Vertex {
    int *path;//path[i] stores the last step of the nearest way to A-
i
    int *dist;//dist[i] stores the distance to A-i
    int *known;//used in dijkstra algorithm
```

```c
    struct AdjVer *adjVerList;//
};
struct Graph {
    int numADC;//number of ambulance dispatch centers
    int numVer;//number of vertex
    int *avlAmb;//number of available ambulances in each center
    struct Vertex *verList;
    /*
     * ambulance dispatch centers are verList[1]~verList[numADC]
     * calling spots are verList[numADC+1]~verList[numADC+numVer]
     */
};

//tool function that convert a string which only contains '0'~'9' to
a decimal integer
int StrToDec(char *str) {
    int rst = 0, len = strlen(str);
    for (int i = 0; i < len; i++) {
        rst *= 10;
        rst += str[i] - '0';
    }
    return rst;
}

//add a path v1->v2 with distance "dist" into a certain graph
void PathToMap(struct Graph *Map, int v1, int v2, int dist) {
    struct AdjVer *newNode, *ptr;
    //malloc new space
    newNode = calloc(1, sizeof(struct AdjVer));
    newNode->ver = v2;
    newNode->dist = dist;
    //insert new path
    if (Map->verList[v1].adjVerList == NULL) {
        Map->verList[v1].adjVerList = newNode;
    } else {
        ptr = Map->verList[v1].adjVerList;
        while (ptr->next)
            ptr = ptr->next;
        ptr->next = newNode;
    }
}

//change a string in certain format into an integer
int MyGetVer(struct Graph *Map) {
    char str[MaxStrSize];
    int temp = Invalid;
    scanf("%s", str);
    if (str[0] == 'A')
        temp = StrToDec(str + 2);
    else
        temp = StrToDec(str) + Map->numADC;

    return temp;
}

//load a path in a certain format, the details of input format are in
my report
void LoadPath(struct Graph *Map) {
    int v1, v2, dist;

    v1 = MyGetVer(Map);
    v2 = MyGetVer(Map);
    dist = MyGetVer(Map) - Map->numADC;
    //undirected graph, path a mutual for vertexes
    PathToMap(Map, v1, v2, dist);
    PathToMap(Map, v2, v1, dist);
};

//return the nearest unknown vertex for ambulance dispatch center i
int NearestVertex(struct Graph *Map, int index) {
    int rst = Invalid, temp = Infinity;
    for (int i = 1; i <= Map->numADC + Map->numVer; i++) {
        if (Map->verList[i].dist[index] < temp &&
Map->verList[i].known[index] != 1) {
            rst = i;
            temp = Map->verList[i].dist[index];
        }
    }
    return rst;
}

//main algorithm
void Dijkstra(struct Graph *Map) {
    struct AdjVer *ptr;
    int tempVer, tempDist, heapSize;
    //there are i dispatch centers and each of them owns private
parameters
    //so we need to run Dijkstra algorithm i times in all
    //and store the result in i different places
    for (int i = 1; i <= Map->numADC; i++) {
        //initialize A-i
        Map->verList[i].dist[i] = 0;
        Map->verList[i].path[i] = Invalid;
        //find the nearest vertex
        while ((tempVer = NearestVertex(Map, i)) != Invalid) {
            //update info of the nearest
            Map->verList[tempVer].known[i] = 1;
            tempDist = Map->verList[tempVer].dist[i];
            //update info of adjacent vertexes of the nearest one
            ptr = Map->verList[tempVer].adjVerList;
            while (ptr) {
                if (Map->verList[ptr->ver].known[i] != 1) {
                    if (Map->verList[ptr->ver].dist[i] > ptr->dist +
tempDist) {
                        Map->verList[ptr->ver].dist[i] = ptr->dist +
tempDist;
                        Map->verList[ptr->ver].path[i] = tempVer;
                    }
                }
                ptr = ptr->next;
            }
        }
    }
}

//initialize my Map
void initMap(struct Graph *Map) {
    //to simplify the index
    //each array are numbered from 1 to n
    //so we need n+1 spaces and XXX[0] is always unavailable in this
project
    scanf("%d %d", &Map->numVer, &Map->numADC);
    //allocate new space
    Map->avlAmb = calloc(Map->numADC + 1, sizeof(int));
    Map->verList = calloc((Map->numVer + Map->numADC + 1),
sizeof(struct Vertex));
    for (int i = 1; i < Map->numVer + Map->numADC + 1; i++) {
        Map->verList[i].path = calloc(Map->numADC + 1, sizeof(int));
        Map->verList[i].dist = calloc(Map->numADC + 1, sizeof(int));
        Map->verList[i].known = calloc(Map->numADC + 1, sizeof(int));
        //initialize
        for (int j = 1; j <= Map->numADC; j++) {
            Map->verList[i].path[j] = Invalid;
            Map->verList[i].dist[j] = Infinity;
        }
        Map->verList[i].adjVerList = NULL;
    }
    //load the number of available ambulance of each ADC
    for (int i = 1; i <= Map->numADC; i++)
        scanf("%d", &Map->avlAmb[i]);
}

//print the nearest way from a certain ADC to the streets
void PrintPath(struct Graph *Map, int ADC, int street) {
    //buff[] works as a simple stack
    int buff[Map->numADC + Map->numVer], size = Map->numADC +
Map->numVer;
    int nowVer = street;
    //push
    buff[--size] = nowVer;
    //push
    while (Map->verList[nowVer].path[ADC] != Invalid) {
        buff[--size] = Map->verList[nowVer].path[ADC];
        nowVer = buff[size];
    }
    //pop
    while (size < Map->numADC + Map->numVer) {
        if (buff[size] <= Map->numADC)
            printf("A-%d ", buff[size]);
        else
            printf("%d ", buff[size] - Map->numADC);
        size++;
    }
    printf("\n");
}

//if two ADC have the same distance, judge which one owns a less
number of edge
int LessEdge(struct Graph *Map, int street, int ADC1, int ADC2) {
    int cnt1 = 0, cnt2 = 0, temp1 = Map->verList[street].path[ADC1],
temp2 = Map->verList[street].path[ADC2];
    while (temp1 != ADC1) {
        temp1 = Map->verList[temp1].path[ADC1];
        ++cnt1;
    }
    while (temp2 != ADC2) {
        temp2 = Map->verList[temp2].path[ADC2];
        ++cnt2;
    }
    return cnt1 < cnt2 ? ADC1 : ADC2;
}

//read a number as the destination and output the dispatch info
void Dispatch(struct Graph *Map) {
    int street, ADC = Invalid, tempDist = Infinity;
    scanf("%d", &street);
    street += Map->numADC;
    //find the nearest ADC
    for (int i = 1; i <= Map->numADC; i++) {
        if (Map->verList[street].dist[i] < tempDist &&
Map->avlAmb[i] > 0) {
            ADC = i;
            tempDist = Map->verList[street].dist[i];
        } else if (Map->verList[street].dist[i] == tempDist &&
Map->avlAmb[i] > 0)//if the dist is the same
            if (Map->avlAmb[ADC] == Map->avlAmb[i])//if the avlAmb is
the same
                ADC = LessEdge(Map, street, i, ADC);
            else {
                ADC = Map->avlAmb[ADC] > Map->avlAmb[i] ? ADC : i;
            }
    }
    //output info
    if (ADC == Invalid) {
        printf("All Busy\n");
    } else {
        Map->avlAmb[ADC]--;
        PrintPath(Map, ADC, street);
        printf("%d\n", tempDist);
    }
}

int main() {
    //1st step: initialize Map
    int numEdge, numCall;
    struct Graph *Map = malloc(sizeof(struct Graph));
    initMap(Map);
    //2nd step: fulfill the Map with paths (aka edges)
    scanf("%d", &numEdge);
    for (int i = 0; i < numEdge; i++)
```

```
        LoadPath(Map);
    //3rd step: use Dijkstra algorithm to find the nearest path
    Dijkstra(Map);
    //4th step: handle dispatches and output info
    scanf("%d", &numCall);
    for (int i = 0; i < numCall; i++)
        Dispatch(Map);

    return 0;
}
```

## Declaration

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