Two algorithm implementations of Ambulance **D**ispatch

Author's Name

Date: 2021**-11-2**8

Chapter 1: Introduction

This question gives a map of a city with all ambulance dispatch centers and all pickup locations. We need to write a program to handle emergency calls, and select the most appropriate ambulance center based on the judgment conditions given by the subject to dispatch an ambulance from the most appropriate street to the boarding location for assistance.

The main data structure we used in this question is a linked list, and two algorithm solutions are used, namely the `floyd` algorithm and the `dijkstra` algorithm. Both algorithms have successfully completed the task and are in the process of implementation and testing. Each has its own merits. Below I will elaborate on the implementation process and test performance of the two algorithms, as well as the comparative analysis and possible improvements of the two algorithms.

Chapter 2: Algorithm Specification

2.1 Statement

2.1.1 Data structure

```
typedef struct _Node{
    int value;
    struct _Node *next;
}Node;
```

For convenience, we define a linked list ourselves. The `value` of the dumb node stores the length of the linked list, and the remaining nodes store the variable values related to the topic.

2.1.2 Some shared functions

`int size(Node *Head)`本函数的目的是为了得到链表的长度。

```
Input:&Head
return Head->value;
```

The function of this function is similar to `.size()`. We have already said when defining the linked list that the `value` of a dumb head node stores the length of the linked list, so we directly return the value of the head node. The length of the linked list is obtained, so the definition is very convenient when updating the linked list.

`int at(Node *Head,int index)` This function is to return the value of the `index` node in the linked list.

```
if index>=size(Head) then
    return -1
end if
Node *p ← Head->next
while i<index do
    p ← p->next;
    i ← i+1;
end while
```

If the input node position is greater than the length of the linked list, the value `-1` is returned directly, if not, the traversal is performed to obtain the node value at the `index` position. In the output part of this question, you need to use this function to output the target node, that is, the centers and spots stored in the linked list. We have previously stored all the waypoints in the path in the linked list.

`void append(Node *Head,int x)` The function of this function is to add elements to the end of the linked list.

```
Node *T,malloc();
T->value ← x;
Node *p ← Head;
while p->next!=NULL do
    p ← p->next;
end while
T->next ← p->next;
p->next ← T;
Head->value ← Head->value+1;
```

The function of this function is similar to `.push_back()`, we need to use this function in the part of adding nodes to the path, which is completed by simple traversal.

`void remove_back(Node *Head)` The function of this function is to remove an element in the linked list.

```
Node *p ← Head;
Node *t;
while p->next!=NULL do
    t ← p;
    p ← p->next;
end while
t->next ← t->next->next;
free(p);
Head->value ← Head ->value-1;
```

The function of this function is similar to `.pop_back()`. We need to use this function when deleting the last element. The idea is to complete it by simple traversal.

`void clear(Node *Head)` This function is to clear the linked list.

```
Node *p ← Head->next;
Node *t;
while t!=NULL do
    t ← p;
    p ← p->next;
    free(t);
end while
Head->next ← NULL;
Head->value ← 0;
```

In many parts of the program, we need to clear a linked list, so we only need to traverse the linked list to delete each node and release the space of each node.

`Node* Copy(Node *Head)` The purpose of this function is to copy a linked list.

```
Node *pNode ← Head;
Node *Thead,*Tnode;
if pNode!=NULL then
    Thead ← (Node*)malloc(sizeof(Node));
    Thead->value ← pNode->value;
    Thead->next ← NULL;
    Tnode ← Thead;
    pNode ← pNode->next;
end if
while pNode!=NULL do
    Node *tempNode ← (Node*)malloc(sizeof(Node));
    tempNode->value ← pNode->value;
    tempNode->next ← NULL;
    Tnode->next ← tempNode;
    Tnode ← Tnode->next;
    pNode ← pNode->next;
end while
return Thead;
```

In many parts of the program, we need to use this function to copy a linked list, but at the same time we don't want to modify one linked list while affecting another linked list, so we can't simply assign the head node of a linked list to For another linked list, we need to traverse and assign values to each node, so that we get two identical but independent linked lists.

`void initialize()` The function of this function is to initialize the linked list.

```
temp=(Node*)malloc(sizeof(Node));
temp->value ← 0;
temp->next ← NULL;

for i ← 0 to 15 do
    for j ← 0 to 15 do
        pre[i][j]=(Node*)malloc(sizeof(Node));
        pre[i][j]->value ← 0;
        pre[i][j]->next ← NULL;
    end for
end for
```

We used two linked lists in this topic, and the initialization process is shown above. At the beginning of the program, we need to call this function to initialize the linked list, that is, to initialize the head node, we need to use this function.

`int StrToInt(char *s)`

```
int result ← 0;
for i ← 0 to strlen(s) do
    result ← result*10+s[i]-'0';
end for
return result;
```

When describing the distance between spots or centers, we need three variables, set the first two variables describing the location as a string, and then judge whether it is spot or center in the main function, and then call this function to convert the string As an integer variable, the third variable is stored in a two-dimensional array describing the distance between any two points. The

number of rows and columns of the two-dimensional array is determined by the coordinates of the two points.

2.2 Algorithm 1: Floyd

2.2.1 Functions

`void Floyd(int Ns,int Na)` This function is the core of this algorithm and explains the main content of the `Floyd` algorithm.

```
Input:Ns,Na
n ← Ns+Na
for k ← 0 to n do
    for i ← 0 to n do
        for j ← 0 to n do
            if d[i][k]+d[k][j]<d[i][j] then
                d[i][j] ← d[i][k]+d[k][j];
                path[i][j] ← k;
            end if
        end for
    end for
end for
```

This function is mainly the implementation of the `Floyd` algorithm. The input is the sum of the ambulance dispatch centers and all the pick-up spots. After three traversals, the shortest path (shortest time) between any two points is obtained. The shortest path between any two points is obtained as the transit point.

This algorithm mainly uses the idea of dynamic programming to find the shortest path between any two points under the condition that which points are allowed as the transit point, that is, from the vertex of `i` to the vertex of `j`, it only passes through the first `k` Although the algorithm process is more complicated for the shortest distance to the number point, the implementation of the code is very simple and clear, and the core code is only a short five lines.

`int Number(int i,int j)` This function is to record the number of streets passed between two points.

```
Input:i,j
if path[i][j]==-1 then
    return 0;
else
    return 1+Number(i,path[i][j])+Number(path[i][j],j);
end if
```

This function uses recursion to continuously call the `Number` function, and finally gets the number of streets that the shortest path passes between any two points.

`void Path(int i,int j)` This function is to record the intermediate point of the shortest path between any two points.

```
Input:i,j
if path[i][j]==-1 then
    return;
end if
Path(i,path[i][j]);
append(ans,path[i][j]);
Path(path[i][j],j);
```

This function uses recursion to continuously call the `Path` function, and finally gets all the intermediate points of the shortest path between any two points, and stores these points in our pre-defined linked list.

2.2.2 Main program

The main program is mainly divided into three parts. The first part is the reading of various variables and the initialization of custom variables, the second part is to judge the target path based on conditions, and the third part is to output the selected path.

The first part is implemented by several `for` loops, which is very simple and will not be repeated here.

The second part is realized by multiple `if` judgments, the process is as follows. When a vertex sends out a help signal, for a rescue center with a number of remaining vehicles greater than `0`, find the shortest path between the vertex and the rescue center, if the elapsed time is the smallest, select the smallest path, if the same, compare more vehicles If the time and number are the same, compare the number of streets passed and choose the rescue center with the smallest number of streets.

```
if ambulance[j]>0 then
    if d[j+Ns][spot]<minTime then
        minTime ← d[j+Ns][spot];
        maxAmbulance ← ambulance[j];
        minNumber ← Number(j+Ns,spot);
        index ←  j;
    else if d[j+Ns][spot]==minTime&&ambulance[j]>maxAmbulance then
        maxAmbulance ← ambulance[j];
        minNumber ← Number(j+Ns,spot);
        index ←  j;
    else if d[j+Ns]
[spot]==minTime&&ambulance[j]==maxAmbulance&&Number(j+Ns,spot)<minNumber then
        minNumber ← Number(j+Ns,spot);
        index ←  j;
    end if
end if
```

The third part judges the situation, judges whether the situation is `All busy` at this time, if it is, output the sentence, if it is not, output the selected path, and output the value of each node of the linked list and the corresponding two endpoints by traversing.

```
for i ← 0 to size(ans) do
    if at(ans,i)>Ns then
        printf(" A-%d",at(ans,i)-Ns);
    else
        printf(" %d",at(ans,i)-Ns);
    end if
end for
```

In this process, you need to pay attention to the output of each line of spaces, as well as the `clear()` operation on the variable `ans`.

2.3 Algorithm 2: Dijkstra

2.3.1 Functions

`void Dijsktra(int index)` This function is the core of this algorithm and explains the main content of `Dijsktra` algorithm.

```
Input:index
for i ← 0 to maxn do
    flag[i] ← 0;
    d[index][i] ← inf;
end for
d[index][index+Ns] ← 0;
for i ← 0 to Ns+Na do
    u ← -1;
    min ← inf;
    for j ← 0 to Ns+Na do
        if flag[j]==0&&d[index][j]<min then
            min ← d[index][j];
            u ← j;
        end if
    end for
    if u==-1 then
        return
    end if
    flag[u] ← -1;
    for v ← 0 to Ns+Na do
        if flag[v]==0&&G[u][v]!=inf then
            if d[index][u]+G[u][v]<d[index][v] then
                clear(pre[index][v]);
                append(pre[index][v],v);
                d[index][v]=d[index][u]+G[u][v];
            else if d[index][u]+G[u][v]==d[index][v] then
                append(pre[index][v],v);
            end if
        end if
    end for
end for
```

In the above algorithm, the variables are initialized first, and then the shortest path between two points is obtained by traversing. This process is to determine whether the path will be closer by selecting the intermediate point. If it is closer, the distance is updated. After all the points are traversed The shortest distance from the starting point to all points is obtained. In this process, we use functions such as `clear()` and `append()` to perform corresponding operations on the linked list to update the path.

When the number of summary points is relatively large, we first use the `dijsktra` algorithm to get the shortest path from each rescue center to each rescue point, and save it with the `pre[][]` array.

`void DFS(int spot, int center)` This function is to get the corresponding path.

```
Input:spot,center
if spot==center+Ns then
    if size(path==0) then
        path ← copy(temp);
    else
        if size(temp)<=size(path)
            path ← copy(temp);
        end if
    end if
    remove_back(temp);
    return ;
end if
for i ← 0 to size(pre[center][spot]) do
    DFS(at(pre[center][spot],i),center);
end for
remove_back(temp);
```

After getting all the possible shortest paths, we use the depth-first algorithm to traverse to get the corresponding path that our goal needs. In this process, we need to use functions such as `copy()` and `remove_back()` to perform corresponding operations on the linked list to update the path.

2.3.2 Main program

The main program part of this algorithm is generally similar to the main program part of the previous algorithm. The idea is the same. It is still divided into three parts, and each part has only changed in some details.

```
if ambulance[j]>0 then
    if d[j][spot]<minTime then
        DFS(spot,j);
        minTime ← d[j][spot];
        maxAmbulance ← ambulance[j];
        res ← copy(path);
        clear(path);
        index ← j;
    else if d[j][spot]==minTime&&ambulance[j]>maxAmbulance then
        DFS(spot,j);
        maxAmbulance ← ambulance[j];
        res ← copy(path);
        clear(path);
        index ← j;
    else if d[j][spot]==minTime&&ambulance[j]==maxAmbulance then
        DFS(spot,j);
        if size(path)<size(res) then
            res ← copy(path);
            index ← j;
        end if
        clear(path);
    end if
```

```
end if
```

It should be noted that the indexing method of the two-dimensional array here is different from the previous algorithm, which needs to be paid attention to when inputting and outputting and updating the path.

Chapter 3: Testing Results

3.1 Operation result

Both algorithms have successfully passed the sample test, which shows that the correctness has been greatly guaranteed. The following only lists the output results of the `dijkstra` algorithm program. In fact, the output results of the two programs are completely consistent.

Sample Input 1:

```
//Common case
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
```

Sample 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
```

Sample Input 2:

```
//Minimal case
1 1
2
1
A-1 1
2
1 1
```

Sample Output 2:

```
A-1 1
2
```

Sample Input 3:

```
//The input is the largest case, because there are too many lines, the test
sample text file is in the folder.
```

Sample Output 3:

```
//There are too many lines, and the output text file is in the folder.
```

3.2 Algorithm comparison

In the process of solving this project, I used two algorithms to obtain the results. Both algorithms have their own merits, and both successfully solved the problem.

In terms of the complexity of the algorithm, the time complexity of the `floyd` algorithm is $O(N^3)$, and the time complexity of the `dijkstra` algorithm is $O(N^2)$, the time complexity of the former Slightly inferior to the latter. But in terms of the form of presentation, as far as this topic is concerned, the implementation of the `floyd` algorithm is more concise and clear than the `dijkstra` algorithm, which has been reflected to a considerable extent in the complexity of the function.

Chapter 4: Analysis and Comments

4.1 The complexities

For functions, the time complexity and space complexity are shown in the following table.

	Time complexity	Space complexity
`Floyd`	$O(N^3)$	$O(N^2)$
`Dijkstra`	$O(N^2)$	$O(N^2)$

For the entire program, because the `dijkstra` algorithm still nests a `for` loop when calling the function in the main program, the overall time complexity is still $O(N^2 \times N) = O(N^3)$.

4.2 Algorithm improvement

In general, in the process of solving the problem this time, I used two algorithms to solve the problem, and I have proposed and completed a certain degree of improvement to the program. Compared with the `floyd` algorithm, the `dijkstra` algorithm improves its time complexity, and the test results are better for larger data samples. For `dijkstra`, the `floyd` algorithm is greatly reduced The workload of the program will appear to be much more concise overall.

As far as the program is concerned, I think my program still has a lot of room for improvement in the definition of variables. The index of the two-dimensional array elements in the current program is rather confusing, so the two algorithms cannot reach the form in the final output part. Unity. In addition, this program has done a good job in the release of space, but there is still room for improvement. In the process of solving the problem this time, I basically use global variables, and I basically don't need to pass them as parameters to the function. In fact, local variables can also be defined. When calling the function, the variables are passed as parameters to the function, which is more convenient. To solve the problem of dynamic changes in priority.

For the question itself, if these judgment conditions are still not enough to determine the only path, we can add `=` to the judgment condition, so that the last path that satisfies the condition can be obtained.

Appendix: Source Code

Algorithm 1: Floyd

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define inf 1000000000
#define maxn 1050

typedef struct _Node{//First we write a linked list ourselves
    int value;
    struct _Node *next;
}Node;//The value of the head node stores the length of the linked list

int ambulance[15]={0};//Number of ambulances in rescue centers
int G[maxn][maxn];//Distance between nodes
int d[maxn][maxn];//The distance between the nodes in the question
int path[maxn][maxn];//The midpoint of the shortest path between any two points
int Ns,Na,M,K;//Variables given in the title
Node *ans;//Head node

int size(Node *Head){//Returns the length of the linked list
    return Head->value;
}
Node* copy(Node *Head){//Copy a linked list
    Node *pNode=Head;//Define a new node
    Node *Thead,*Tnode;//Define new nodes
    if(pNode!=NULL){//
        Thead=(Node*)malloc(sizeof(Node));
        Thead->value=pNode->value;
        Thead->next=NULL;
        Tnode=Thead;
        pNode=pNode->next;
    }//The case where the head node is not empty
```

```
    while(pNode!=NULL){//Enter the loop to traverse one by one and assign values
        Node *tempNode=(Node*)malloc(sizeof(Node));
        tempNode->value=pNode->value;
        tempNode->next=NULL;
        Tnode->next=tempNode;
        Tnode=Tnode->next;
        pNode=pNode->next;
    }
    return Thead;//Return the head node of the new linked list
}
void append(Node *Head,int x){//Add elements to the end of the linked list
    Node *T = (Node*)malloc(sizeof(Node));
    T->value = x;
    Node *p = Head;
    while(p->next!=NULL) p = p->next;//Traverse to reach the target node
    T->next = p->next;
    p->next = T;
    Head->value++;//the length of the linked list increases by 1
}
void remove_back(Node *Head){//remove an element in the linked list
    Node *p = Head, *t;
    while(p->next!=NULL){//Traverse to reach the target node
        t=p;
        p=p->next;
    }
    t->next=t->next->next;
    free(p);//Free up space
    Head->value--;//the length of the linked list decreases by 1
}
void clear(Node *Head){//Empty the linked list
    Node *p = Head->next, *t;
    while(p!=NULL){//Free up space
        t = p;
        p = p->next;
        free(t);
    }
    Head->next=NULL;
    Head->value=0;//Then the length of the linked list is reset to zero
}
int at(Node *Head,int index){//Returns the value of the index node
    if(index >= size(Head))return -1;//If it is greater than the length of the
linked list
    int i=0;
    Node *p = Head->next;
    while(i<index){
        p = p->next;
        i++;
    }//Traverse to reach the target node
    return p->value;
}
void initialize(){//Initialize the linked list when the program starts, the
first line of the main program
    ans = (Node*)malloc(sizeof(Node));
    ans->value=0;//The length of the linked list is 0
    ans->next=NULL;//The head node points to NULL
}
int StrToInt(char *s){//Convert a string into an integer variable
    int result = 0,i;
```

```
    for(i=0; i<strlen(s); i++){
        result = result*10+s[i]-'0';
    }//To transform one by one
    return result;
}

void Floyd(int Ns,int Na){
    int i,j,k;
    int n=Ns+Na;
    for(k=1; k<=n; k++){//Enter the loop to find the shortest path
        for(i=1; i<=n; i++){
            for(j=1; j<=n; j++){
                if(d[i][k]+d[k][j]<d[i][j]){//Update if shorter than current
                    d[i][j] = d[i][k]+d[k][j];
                    path[i][j] = k;//Update midpoint
                }
            }
        }
    }
}
int Number(int i, int j){
    if(path[i][j]==-1) return 0;
    else return 1+Number(i,path[i][j])+Number(path[i][j],j);//Return the value
of the number of streets
}
void Path(int i, int j){
    if(path[i][j]==-1) return;
    Path(i,path[i][j]);//The path from the first point to the middle point
    append(ans,path[i][j]);//Update the shortest path by operating on the linked
list
    Path(path[i][j],j);//The path from the middle point to the second point
}

int main(){
    initialize();//Initialize the linked list
    int i,j;
    for(i=0;i<maxn;i++){//Initialize two-dimensional arrays
        for(j=0;j<maxn;j++){
            path[i][j]=-1;
            G[i][j]=inf;
        }
    }
    for(i=0; i<maxn; i++){
        G[i][i] = 0;//The distance from the point to itself is 0
    }
    scanf("%d%d",&Ns,&Na);
    for(i=1; i<=Na; i++){
        scanf("%d",&ambulance[i]);//Input variable
    }
    scanf("%d",&M);
    int u,v;
    char s1[10],s2[10];
    int time;
    for(i=0; i<M; i++){
        scanf("%s%s%d",s1,s2,&time);//Input variable
        //Convert string to integer
        if(s1[0]=='A') u = Ns+StrToInt(s1+2);
        else u = StrToInt(s1);
```

```
        if(s2[0]=='A') v = Ns+StrToInt(s2+2);
        else v = StrToInt(s2);
        G[u][v] = G[v][u] = time;//Get the distance between the two coordinates
entered
    }
    for(i=0; i<maxn; i++){
        for(j=0; j<maxn; j++){
            d[i][j] = G[i][j];//Two-dimensional array is symmetric
        }
    }
    Floyd(Ns,Na);//Use the floyd algorithm to traverse all nodes to get the
shortest distance between any two points
    int spot;
    scanf("%d",&K);
    for(i=0; i<K; i++){
        scanf("%d",&spot);
    /*  printf("spot=%d\n",spot);*/
        int minTime=inf,maxAmbulance=-1,minNumber=inf,index=-1;
        for(j=1; j<=Na; j++){
            if(ambulance[j]>0){//Is the number of ambulances greater than 0
                if(d[j+Ns][spot]<minTime){//If the current time is less than the
shortest time available
                    minTime = d[j+Ns][spot];//Update the shortest time
                    maxAmbulance = ambulance[j];//Update the maximum quantity
                    minNumber = Number(j+Ns,spot);//Minimum number of streets
                    index =  j;
                }
                else if(d[j+Ns][spot]==minTime&&ambulance[j]>maxAmbulance){//If
the quantity is greater than the current maximum quantity
                    maxAmbulance = ambulance[j];//Update the maximum quantity
                    minNumber = Number(j+Ns,spot);//Minimum number of streets
                    index =  j;
                }
                else if(d[j+Ns]
[spot]==minTime&&ambulance[j]==maxAmbulance&&Number(j+Ns,spot)<minNumber){//If
the number of streets is less than the current minimum number of streets
                    minNumber = Number(j+Ns,spot);//Update the minimum number of
streets
                    index =  j;
                }
            }
        }
        if(index==-1) printf("All Busy\n");//If no ambulance is available
        else{
            clear(ans);
            Path(index+Ns,spot);//Get the most suitable path for a certain point
            printf("A-%d",index);
            for(j=0; j<size(ans); j++){//Output points passed by the path
                if(at(ans,j)>Ns) printf(" A-%d",at(ans,j)-Ns);
                else printf(" %d",at(ans,j));
            }
            printf(" %d\n",spot);
            printf("%d\n",minTime);//Output shortest time
            ambulance[index]--;//Number of ambulances minus one
        }
    }
    return 0;
}
```

```
// 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
```

Algorithm 2: Dijkstra

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define inf 100000000
#define maxn 1050

typedef struct _Node{//First we write a linked list ourselves
    int value;
    struct _Node *next;
}Node;//The value of the head node stores the length of the linked list

int ambulance[15];//Number of ambulances in rescue centers
int G[maxn][maxn];//Distance between nodes
int d[15][maxn];//The distance between the nodes in the question
int flag[maxn];//The value is only 0 or 1
int Ns,Na,M,K;//Variables given in the title
Node *temp;
Node *path;
Node *pre[15][maxn];

int size(Node *Head){//Returns the length of the linked list
    return Head->value;
}
Node* copy(Node *Head){//Copy a linked list
    Node *pNode=Head;//Define a new node
    Node *Thead,*Tnode;//Define new nodes
    if(pNode!=NULL){//
        Thead=(Node*)malloc(sizeof(Node));
        Thead->value=pNode->value;
        Thead->next=NULL;
        Tnode=Thead;
        pNode=pNode->next;
```

```
    }//The case where the head node is not empty
    while(pNode!=NULL){//Enter the loop to traverse one by one and assign values
        Node *tempNode=(Node*)malloc(sizeof(Node));
        tempNode->value=pNode->value;
        tempNode->next=NULL;
        Tnode->next=tempNode;
        Tnode=Tnode->next;
        pNode=pNode->next;
    }
    return Thead;//Return the head node of the new linked list
}
void append(Node *Head,int x){//Add elements to the end of the linked list
    Node *T = (Node*)malloc(sizeof(Node));
    T->value = x;
    Node *p = Head;
    while(p->next!=NULL) p = p->next;//Traverse to reach the target node
    T->next = p->next;
    p->next = T;
    Head->value++;//the length of the linked list increases by 1
}
void remove_back(Node *Head){//remove an element in the linked list
    Node *p = Head, *t;
    while(p->next!=NULL){//Traverse to reach the target node
        t=p;
        p=p->next;
    }
    t->next=t->next->next;
    free(p);//Free up space
    Head->value--;//the length of the linked list decreases by 1
}
void clear(Node *Head){//Empty the linked list
    Node *p = Head->next, *t;
    while(p!=NULL){//Free up space
        t = p;
        p = p->next;
        free(t);
    }
    Head->next=NULL;
    Head->value=0;//Then the length of the linked list is reset to zero
}
int at(Node *Head,int index){//Returns the value of the index node
    if(index >= size(Head))return -1;//If it is greater than the length of the
linked list
    int i=0;
    Node *p = Head->next;
    while(i<index){
        p = p->next;
        i++;
    }//Traverse to reach the target node
    return p->value;
}
void initialize(){//Initialize the linked list when the program starts, the
first line of the main program
    temp = (Node*)malloc(sizeof(Node));
    temp->value=0;//The length of the linked list is 0
    temp->next=NULL;//The head node points to NULL

    path = (Node*)malloc(sizeof(Node));
```

```
    path->value=0;
    path->next=NULL;

    int i,j;
    for(i=0;i<15;i++){
        for(j=0;j<maxn;j++){
            pre[i][j] = (Node*)malloc(sizeof(Node));
            pre[i][j]->value=0;
            pre[i][j]->next=NULL;
        }
    }//The same is true for the linked list of a two-dimensional array, which is
defined one by one
}
int StrToInt(char *s){//Convert a string into an integer variable
    int result = 0,i;
    for(i=0; i<strlen(s); i++){
        result = result*10+s[i]-'0';
    }//To transform one by one
    return result;
}

void Dijsktra(int index){
    int i,j;
    for(i=0;i<maxn;i++){
        flag[i]=0;
        d[index][i]=inf;
    }
    d[index][index+Ns]=0;
    for(i=1;i<=Ns+Na;i++){
        int u=-1;
        int min=inf;
        for(j=1;j<=Ns+Na;j++){
            if(flag[j]==0&&d[index][j]<min){
                min=d[index][j];
                u=j;
            }
        }
        if(u==-1) return;
        flag[u]=1;
        int v;
        for(v=1;v<=Ns+Na;v++){
            if(flag[v]==0&&G[u][v]!=inf){
                if(d[index][u]+G[u][v]<d[index][v]){
                    clear(pre[index][v]);
                    append(pre[index][v],u);
                    d[index][v]=d[index][u]+G[u][v];
                }else if(d[index][u]+G[u][v]==d[index][v]) append(pre[index]
[v],u);
            }
        }
    }
}
void DFS(int spot, int center){
    append(temp,spot);
    if(spot==center+Ns){
        if(size(path)==0) path=copy(temp);
        else{
            if(size(temp)<=size(path)) path=copy(temp);
```

```
        }
        remove_back(temp);
        return;
    }
    int i;
    for(i=0;i<size(pre[center][spot]);i++){
        DFS(at(pre[center][spot],i),center);
    }
    remove_back(temp);
}

int main(){
    initialize();//Initialize the linked list
    int i,j;
    for(i=0;i<maxn;i++){//Initialize two-dimensional arrays
        for(j=0;j<maxn;j++){
            G[i][j]=inf;
        }
    }
    scanf("%d%d",&Ns,&Na);
    for(i=1;i<=Na;i++){
        scanf("%d",&ambulance[i]);//Input variable
    }
    scanf("%d",&M);
    int u,v;
    char s1[10],s2[10];
    int time;
    for(i=0; i<M; i++){
        scanf("%s%s%d",s1,s2,&time);//Input variable
        //Convert string to integer
        if(s1[0]=='A') u = Ns+StrToInt(s1+2);
        else u = StrToInt(s1);
        if(s2[0]=='A') v = Ns+StrToInt(s2+2);
        else v = StrToInt(s2);
        G[u][v] = G[v][u] = time;//Get the distance between the two coordinates
entered
    }
    for(i=1;i<=Na;i++){
        Dijsktra(i);
    }
    scanf("%d",&K);
    for(i=0;i<K;i++){
        int spot;
        int minTime=inf,maxAmbulance=-1,index=-1;
        Node *res;
        res = (Node*)malloc(sizeof(Node));
        res->value=0;
        res->next=NULL;
        scanf("%d",&spot);
        for(j=1;j<=Na;j++){
            if(ambulance[j]>0){//Is the number of ambulances greater than 0
                if(d[j][spot]<minTime){//If the current time is less than the
shortest time available
                    DFS(spot,j);
                    minTime=d[j][spot];//Update the shortest time
                    maxAmbulance=ambulance[j];//Update the maximum quantity
                    res=copy(path);
                    clear(path);
```

```
                    index=j;
                }
                else if(d[j][spot]==minTime&&ambulance[j]>maxAmbulance){//If the
quantity is greater than the current maximum quantity
                    DFS(spot,j);
                    maxAmbulance=ambulance[j];//Update the maximum quantity
                    res=copy(path);
                    clear(path);
                    index=j;
                }
                else if(d[j][spot]==minTime&&ambulance[j]==maxAmbulance){
                    DFS(spot,j);
                    if(size(path)<size(res)){//If the number of streets is less
than the current minimum number of streets
                        res=copy(path);
                        index=j;
                    }
                    clear(path);
                }
            }
        }
        if(maxAmbulance==-1) printf("All Busy\n");//If no ambulance is available
        else{
            for(j=size(res)-1;j>=0;j--){//Output points passed by the path
                if(at(res,j)>Ns) printf("A-%d",at(res,j)-Ns);
                else printf("%d",at(res,j));
                if(j==0) printf("\n");//Wrap
                else printf(" ");//Output space
            }
            printf("%d\n",d[index][spot]);
            ambulance[index]--;//Number of ambulances minus one
        }
    }
}

// 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
```