FUNDAMENTALS OF DATA STRUCTURE

PROJECT 2

PUBLIC BIKE MANAGEMENT

LABORATORY REPORT

Group 7

Program: Li Haipeng

Test: Hao xiangpeng

Document: Yang Kefan

November 26, 2016

CONTENT

Declaration 3

Chapter 1: Introduction

1.1 Problem Description 3

Chapter 2: Algorithm Specification

2.1 Overall Idea. 4

2.2 Input. 4

2.3 Find Shortest Paths.. 5

2.4 Count Number of Bikes. 6

2.5 Output. 7

Chapter 3: Test

3.1 Test Result. 8

3.2 Analyses & Comments. 11

Appendix

4.1 Source Code(in C Language).. 13

DECLARATION

We hereby declare that all the work done in this project titled "Public Bike Management" is of our independent effort as a group.

1 INTRODUCTION

1.1 Problem Specification

Public bikes are commonly used in many cities nowadays. In a city, bikes are usually distributed to different stations. We say a station is in **perfect** condition if it is exactly half-full because such stations can provide enough bikes for tourists and also have plenty of room for parking. The Public Bike Management Center(PBMC) are trying to keep stations in perfect condition. When a station is full or empty, PBMC will find out the shortest path to that station and send staff to adjust all the stations on the path to perfect condition. If there are multiple shortest paths, PBMC will choose the one which requires least bikes sent from the center. The task is to find out the shortest path and the number of bikes PBMC needs to send when any problem station occurs.

Obviously, we can simplify this problem into finding out the shortest path in a weighted undirected graph. The stations are

vertexes and the distance between two stations is regarded as the weight of an edge. When multiple paths occur, another algorithm is needed to find out the number of bikes sent from the source vertex(PBMC) in each path.

2 ALGORITHM SPECIFICATION

2.1 Overall Idea

To solve the problem, the program first input all needed information. Then an algorithm is used to find shortest paths from the source vertex to the problem vertex. Another algorithm is used to find out path with least bikes to send. Finally, the program output the number of bikes and the shortest path.

```
#pseudocode for overall idea
graph = Input()
paths = FindShortest(graph)
result = FindLeast(paths)
Output(result)
```

2.2 Input

The information needed to solve the problem is the number of stations, the number of roads, the weighted graph, the number of bikes in each station(so we can also know the index of the problem station) and the capacity of stations(assume

that all stations have the same capacity). The program will store this information in the data structure **graph.**

```
#pseudocode for the data structure of graph
graph = {
    Capacity
    NumberOfStation
    NumberOfRoad
    IndexOfProblemStation
    GraphMatrix[][]
}
```

Note that in this data structure we use a connectivity matrix(a 2-degree array) to represent the weighted graph. If there is a road between station 1 and 3 with a distance of 4, for instance, we set

```
GraphMatrix[1][3] = GraphMatrix[3][1] = 4
```

2.3 Find Shortest Path

Since no negative cost is involved in the problem, the Dijkstra's Algorithm is used in this step to find out the shortest path between source vertex and the problem vertex. The idea is to maintain a set containing vertexes whose shortest paths have been found and continuously update this set.

Since if multiple shortest paths exist, the program is supposed record all the options in this step, we adjust the original Dijkstra's Algorithm to record multiple source paths if they have the same distance:

```
#pseudocode for find shortest paths
FindShortest( graph )
{
    Vertex  V, W
    Array T = vertex in graph
    while true
        V = smallest unknown distance vertex
        if V == NotAVertex:
            break
        T[V].Known = true
        for each W adjacent to V:
            if ( T[W].Known!=0 ):
                #Cvw == cost of edge from v to w
                if T[V].Dist + Cvw < T[W].Dist:
                    T[W].Dist = T[V].Dist + Cvw
                    T[W].Path = V
                # record it if the distances are equal
                if T[V].Dist + Cvw == T[W].Dist:
                    add V to T[W].Path
    return T
}
```

2.4 Count Number of Bikes

Once the shortest paths are found, the program needs to figure out the number of bikes to be sent by the source vertex. Now we can denote the number of bikes we collect as **N** and the number of bikes needed to be sent as **S**. The idea is that for each station we compare the number of bikes it has now with the number it should have in perfect condition. If there are more bikes now, we add the difference to **N**. If there are less, we subtract the the absolute value of the difference from **N**. Whenever **N**<0, we add its absolute value to **S** and reset **N** to

zero.

```
#pseudocode for find number of bikes
FindBikeNum( path )
{
    S = N = 0
    for each V in path:
        diff = V.bikeNum - capacity/2
        N += diff
        if N<0:
            S -= N
            N = 0
    return S
}
```

Then we simply find out the smallest one in the return values of all the shortest paths:

```
#pseudocode to find path with least bikes to send
FindLeast( paths )
{
    min = Infinity
    for path in paths:
        bikeNum = FindBikeNum(path)
        if BikeNum < min:
            min = bikeNum
            leastPath = path
    return leastPath
}
```

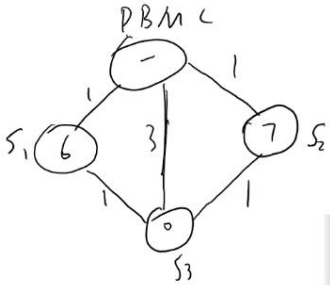
2.4 Ouput

This part is pretty simple: the program print out the shortest path and the number of bikes needed to send in the required format.

3 TEST

3.1 Test Result

Test Case 1



Capacity:10

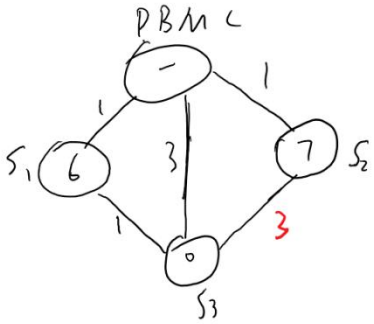
Expected Output:

3 0→2→3 0

Actual Output:

```
root@DESKTOP-V5VVDID
10 3 3 5
6 7 0
0 1 1
0 2 1
0 3 3
1 3 1
2 3 1
3 0->2->3 0
```

Test Case 2



Capacity:10

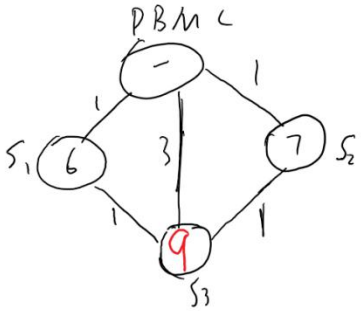
Expected Output:

4 0→1→3 0

Actual Output:

```
10 3 3 5
6 7 0
0 1 1
0 2 1
0 3 3
1 3 1
2 3 3
4 0->1->3 0
```

Test Case 3



Capacity:10

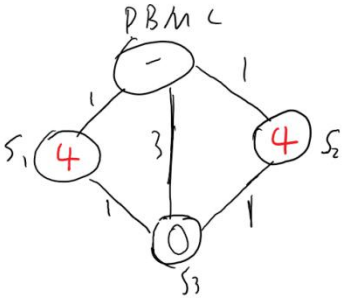
Expected Output:

0 0→1→3 5

Actual Output:

```
10 3 3 5
6 7 9
0 1 1
0 2 1
0 3 3
1 3 1
2 3 1
0 0->1->3 5
```

Test Case 4



Capacity:4

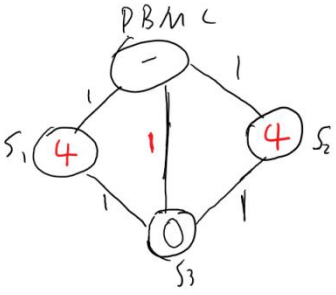
Expected Output:

0 0→1→3 0

Actual Output:

```
4 3 3 5
4 4 0
0 1 1
0 2 1
0 3 3
1 3 1
2 3 1
0 0->1->3 0
```

Test Case 5



Capacity:4

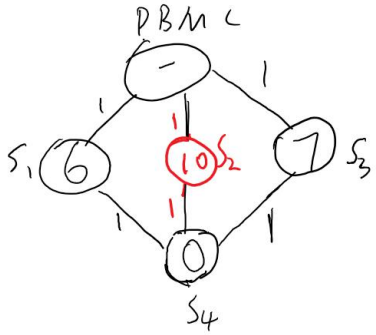
Expected Output:

2 0→3 0

Actual Output:

```
4 3 3 5
4 4 0
0 1 1
0 2 1
0 3 1
1 3 1
2 3 1
2 0->3 0
```

Test Case 6



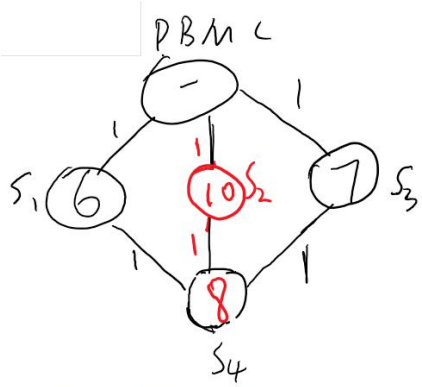
Expected Output:

0 0→2→4 0

Actual Output:

```
10 4 4 6
6 10 7 0
0 1 1
0 2 1
0 3 1
1 4 1
2 4 1
3 4 1
0 0->2->4 0
```

Test Case 7



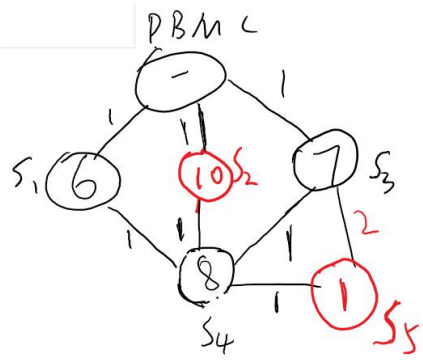
Expected Output:

0 0→1→4 4

Actual Output:

```
10 4 4 6
6 10 7 8
0 1 1
0 2 1
0 3 1
1 4 1
2 4 1
3 4 1
0 0->1->4 4
```

Test Case 8



Capacity:10

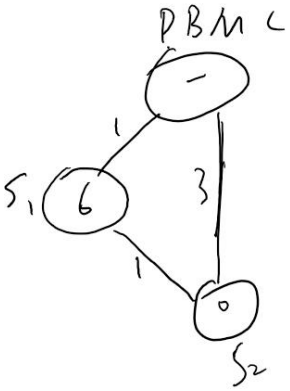
Expected Output:

0 0→1→4→5 0

Actual Output:

```
10 5 5 8
6 10 7 8 1
0 1 1
0 2 1
0 3 1
1 4 1
2 4 1
3 4 1
4 5 1
3 5 2
0 0->1->4->5 0
```

Test Case 9



Capacity:10

Expected Output:

4 0→1→2 0

Actual Output:

```
10 2 2 3
6 0
0 1 1
0 2 3
1 2 1
4 0->1->2 0
```

3.2 Analyses & Comments

The test case was given in the following order:

The program passed all the tests provided and works fine without any performance issues.

We also noticed that, the first version of the program, just ignored the capacity by simply assuming it to be 10, which,

caused a wrong answer. After a small bug fix iteration, the program passed all the test involves capacity change.

In the test case 8, we especially add a node which does not have a direct path to the root, just to test if the programmer ignored such test case. To my surprise, he successfully handled such situation and passed the test case smoothly.

4 APPENDIX

4.1 Source Code(in C Language)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAXNUMOFSTATIONS 1010
#define INFINITE 100000
typedef    struct System* PtrToSystem;
struct System{
    int Capacity,NumOfStations,NumOfProblem,NumOfRoads;
    int CurrentNum[MAXNUMOFSTATIONS];
    int G[MAXNUMOFSTATIONS][MAXNUMOFSTATIONS];
};

int AllRecord[MAXNUMOFSTATIONS][MAXNUMOFSTATIONS];/*AllRecord[i][0~j] stores the previous stations' number, [0~j] means there could be more than one paths with the same time comsuming*/
int PtrToAllRecord[MAXNUMOFSTATIONS];
int AllPaths[MAXNUMOFSTATIONS][MAXNUMOFSTATIONS];
int NumOfShortestPath;

int main() {
    PtrToSystem NewSystem();
    PtrToSystem G;/*New system */
    int Time[MAXNUMOFSTATIONS];/*Sum of all time spent*/
    int Bikes[MAXNUMOFSTATIONS];/*the number of bikes needed, "<0" means taking back while ">0" means sending out*/
    int Path[MAXNUMOFSTATIONS];
    int IndexOfMin,MinSend,MinReceive;
    void Find(PtrToSystem G,int Time[]);/*Find Time[] and Bikes[]*/
    void FindBest(PtrToSystem G,int Bikes[],int Path[]);/*Find the path that need least number of bikes*/
    void OutPut(PtrToSystem G,int MinSend,int MinReceive,int Path[]);/*Output information*/
    void FindLeast(PtrToSystem G,int* IndexOfMin,int* MinSend,int* MinReceive);
    void ConvertToPath(PtrToSystem G);
    G=NewSystem();/*New system created*/
    Find(G,Time);/*Find AllRecord*/
    ConvertToPath(G);
    FindLeast(G,&IndexOfMin,&MinSend,&MinReceive);
    OutPut(G,MinSend,MinReceive,AllPaths[IndexOfMin]);/*Output information*/
    return 0;
```

```
}
PtrToSystem NewSystem(){/*to create a new system*/
	PtrToSystem G=(PtrToSystem)malloc(sizeof(struct System));/*New System*/
	if(G==NULL)return NULL;

scanf("%d %d %d %d",&G->Capacity,&G->NumOfStations,&G->NumOfProblem,&G->NumOf
Roads);/*Essential Information Input*/
	for (int i = 1; i <= G->NumOfStations; ++i) {/*for each station*/
		scanf("%d",G->CurrentNum+i);/*Essential Information Input*/
	}
	G->CurrentNum[0]=0;
	for (int i = 0; i <= G->NumOfRoads; ++i) {/*all to be 0*/
		for (int j = 0; j <= G->NumOfRoads; ++j) {
			G->G[i][j]=0;/*all to be 0*/
		}
	}
	for (int i = 1; i <= G->NumOfRoads; ++i) {/*for each station*/
		int A,B,Time;/*temp variables*/
		scanf("%d %d %d",&A,&B,&Time);/*Essential Information Input*/
		G->G[B][A]=G->G[A][B]=Time;/*Essential Information Input*/
	}
	return G;
}
void Find(PtrToSystem G,int Time[]){

	int Known[G->NumOfStations+1];
	int findmin(PtrToSystem,int[],int[]);
	int CurrentStation=0;
	for (int i = 0; i <=G->NumOfStations ; ++i) {
		Time[i]=INFINITE;/*pretreatment*/
		Known[i]=-1;/*pretreatment*/
		PtrToAllRecord[i]=-1;
	}
	Known[0]=1;/*pretreatment*/
	Time[0]=0;/*pretreatment*/
	while(1){/*do it until all done*/
		for (int i = 0; i <= G->NumOfStations; ++i) {/*traverse all stations that are
Unknown*/
			if(G->G[CurrentStation][i]>0){/*if Unknown*/
				if (Time[i] >= (G->G[CurrentStation][i] + Time[CurrentStation])){/*find a
path with less or equal time*/

					if (Time[i] == (G->G[CurrentStation][i] + Time[CurrentStation])){/*if
```

```
equal time, compare the bikes requirement*/
                                /*create a new line in AllRecord*/
                                AllRecord[++NumOfShortestPath][i]=CurrentStation;/*save  the
path of previous stations*/
                                for (int j = 0; j <= G->NumOfStations; ++j) {
                                    if(!Known[j])continue;
                                    if(j==i)continue;

AllRecord[NumOfShortestPath][j]=AllRecord[NumOfShortestPath-1][j];
                                }
                            }
                            else {
                                Time[i]          =          (G->G[CurrentStation][i]          +
Time[CurrentStation]);/*find a path with less time*/
                                for (int j = 0; j <= NumOfShortestPath; ++j) {
                                    AllRecord[j][i]=CurrentStation;
                                }
                            }
                        }

                    }
                }
                CurrentStation=findmin(G,Time,Known);/*find next station*/
                if(CurrentStation==-1)break;/*all done, exit loop*/
                Known[CurrentStation]=1;/*mark the flag of Known*/
            }


}
void FindLeast(PtrToSystem G,int* IndexOfMin,int* MinSend,int* MinReceive) {
    *MinSend=INFINITE;
    *MinReceive=INFINITE;
    int CurrentRequirementOfSend;
    int CurrentRequirementOfReceive;
    void                    Calculate(PtrToSystem                    G,int
i,int*CurrentRequirementOfSend,int*CurrentRequirementOfReceive);
    for(int i=0;i<=NumOfShortestPath;i++){/*Calculate the num needed to sent*/
        Calculate(G,i,&CurrentRequirementOfSend,&CurrentRequirementOfReceive);
        if (*MinSend>=CurrentRequirementOfSend){
            if(*MinSend==CurrentRequirementOfSend){
                if(*MinReceive>CurrentRequirementOfReceive){
                    *IndexOfMin=i;
                    *MinReceive=CurrentRequirementOfReceive;
                }
```

```
                }
                else{
                        *MinSend=CurrentRequirementOfSend;
                        *MinReceive=CurrentRequirementOfReceive;
                        *IndexOfMin=i;
                }

        }
    }

}

void Calculate(PtrToSystem G, int i, int *CurrentRequirementOfSend, int *CurrentRequirem
entOfReceive){
    int CurrentNum=AllPaths[i][1];
    int PtrToIndex=1;
    int Requirement;/*Requirement For current single station*/
    *CurrentRequirementOfReceive=*CurrentRequirementOfSend=0;
    while(AllPaths[i][PtrToIndex-1]!=G->NumOfProblem){
        Requirement=G->Capacity/2-G->CurrentNum[CurrentNum];
        if(Requirement>0)/*Need send*/{
            if(*CurrentRequirementOfReceive-Requirement<0){/*No enough to provide, so we
have to send them from 0*/

*CurrentRequirementOfSend+=Requirement-*CurrentRequirementOfReceive;
                    *CurrentRequirementOfReceive=0;
                }
                else{/*what we received can afford this station*/
                    *CurrentRequirementOfReceive-=Requirement;
                }
            }
            else{/*need receive*/
                *CurrentRequirementOfReceive+=-(Requirement);
            }
            CurrentNum=AllPaths[i][++PtrToIndex];
    }
}
int findmin(PtrToSystem graph, int t[], int know[])
{
    int i;
    int count;
    int min;
    count=0;
    for (min = 0; min<=graph->NumOfStations; min++){
```

```
            if (know[min] == -1)/*find first Unknown station*/
                break;/*exit the loop*/
        }
        for (i = 0; i<=graph->NumOfStations; i++){
            if (t[min]>t[i] && know[i] == -1)/*compare and save smaller one*/
                min = i;
            if (know[i] == 1)/*next station to be checked*/
                count++;
        }
        if (count != graph->NumOfStations+1)
            return min;/*return min*/
        else return -1;/*fail to find*/
}
void ConvertToPath(PtrToSystem G) {/*use Stack to Convert Records into Paths*/
        int Stack[G->NumOfStations+1];
        int PtrToOnePath;
        int PtrToStack = -1;
        int i;
        for(int j=0;j<=NumOfShortestPath;j++){
            PtrToOnePath = -1;
            i = G->NumOfProblem;
            while (    i != 0) {
                Stack[++PtrToStack] = i;/*push one by one*/
                i = AllRecord[j][i];
            }
            Stack[++PtrToStack]=0;
            while (PtrToStack >= 0) {
                AllPaths[j][++PtrToOnePath] = Stack[PtrToStack--];/*pop to AllPaths*/
            }
            AllPaths[j][++PtrToOnePath]=-1;/*the mark of end*/
        }

}
void OutPut(PtrToSystem G,int MinSend,int MinReceive,int Path[]){
        int Out=0, In=0;
        int i=1;
        printf("%d 0",MinSend);
        while(Path[i]!=-1){
            printf("->%d",Path[i]);/*Out information*/
            i++;
        }
        printf(" %d\n",MinReceive);/*Out information*/
}
```