Public Bike Manage

Author Names

邓墨琳、崔瑜翔、耿飙

Date: 2016-12-04

1. Introduction

The city which provides a bike service for people must be faced with the problem that how to keep bikes in each station at a proper amount. To get over that problem, we must send bikes to the station with few bikes and take bikes from the station with many bikes. This project aims at providing a best path for sending or taking bikes. There are three principles for choosing the best path:

(1) the shortest path will be selected.

(2) If there is more than one shortest path, the path which needs fewest bikes for sending will be selected.

(3) If there is more than one shortest path which needs the same number of bikes for sending, the one which needs fewest bikes for taking will be selected.

To achieve this goal, we use the DFS(Depth-First-Search) algorithm and some judgements to find the path we want.

2. Algorithm Specification

2.1 The Outline

Our strategy to solve the project is:

(1) Read the data of the graph;

(2) Traversal the graph by using the DFS algorithm and calculate the number of bikes that should be sent or taken, then choose the best path;

(3) Print the path.

2.2 Data Structure

(1) The graph:

```
struct GNode {
	int Nv;
	int Ne;
	int G[MaxVertexNum][MaxVertexNum];
};
typedef PtrToGNode MGraph;

MGraph PBMC;
```

(2)The number of bikes in each station:

```
int bike[MaxVertexNum] = {0};
```

(3)The path that will be printed

```
int final_path[MaxVertexNum] = {0};
```

(4)The array to store whether the vertex is visited or not

```
int visited[MaxVertexNum] = {false};
```

2.3 The DFS Algorithm

The character of DFS is to start from an vertex and walk along only one path to visit vertexes until the end of this path. If there is no vertex we want in this path, it will return to the previous vertex to traversal the

other path. Trying to get as deep as possible is the essence of such an algorithm. The pseudo-code is like this:

```
    void dfs (Vertex V){
/*1*/    for each W adjacent to V
/*2*/        if (!visited[ W ])
/*3*/            visit[W] = True;
/*4*/            dfs (W);
/*5*/    visit[ W ] = False;
}
```

From the pseudo-code, we can know that the DFS usually has an array to store whether the vertex is visited or not.

The state of the vertex will decide whether the algorithm continue or not.

And of course we can add some extra operations at the beginning or between the line 4 and line 5 to do somethings to make the function return the information we want, just as we have done in our code.

2.4 The Algorithm To Calculate The Number Of Bikes For Sending Or Taking

We must know the bikes we need for the path, so we can decide whether the current path is better than the previous one. To do this, our algorithm will use the rules that the stations' amount of bikes is larger or smaller than half of the capacity should be adjusted to the half. One thing that we must pay attention to is that the project also asks that we cannot take the bikes of the station to one which is in front of it, but can take its bikes to the station behind it. Our algorithm simplifies the progress by concentrating on the change of the "take". We will change the value of take for checking each station and if the value>=0, we know we should take how many bikes. If the value<0,we know we must send bikes and the value of "send" is equal to the minus value of the"take".

Here is the pseudo-code:

```
void g(int* path){//
	send = take = 0;
	for each station on the path {
		if(the number of bikes > capacity/2)
			take increases the difference;
		else{
			if(take + the difference of the station and capacity/2) >= 0){
				take increases the difference;
			}
			else{
				send += -(take +the difference of station and capacity/2);
				take = 0;
			}
		}
	}
}
```

2.5 The Algorithm To Update The Path

This algorithm will work in the DFS. Just use the 3 principles we have mentioned in the chapter1. As we have get the information of each path, it is not difficult to do that :

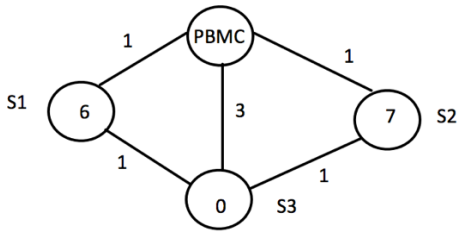
```
	//Update the path
	g(path);
	if(cur == Sp){
			if( principle1 or principle 2 or principle 3 is destroyed){
			  change the previous path to the better one
		}
```

3. Testing Results

The result of our code tested by PTA:

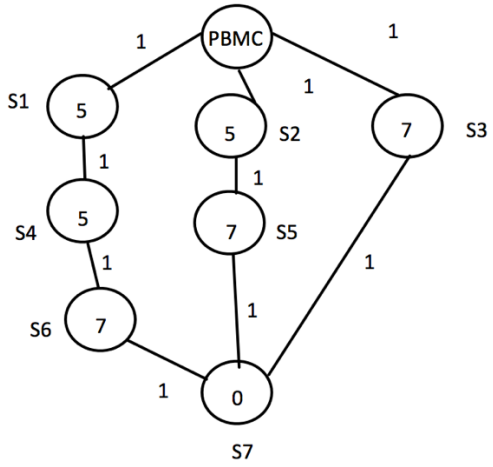
结果	得分	题目
答案正确	35	5-3

	结果	得分/满分
	答案正确	17/17
	答案正确	2/2
	答案正确	2/2
	答案正确	2/2
	答案正确	2/2
	答案正确	2/2
	答案正确	2/2
	答案正确	3/3
	答案正确	2/2
	答案正确	1/1



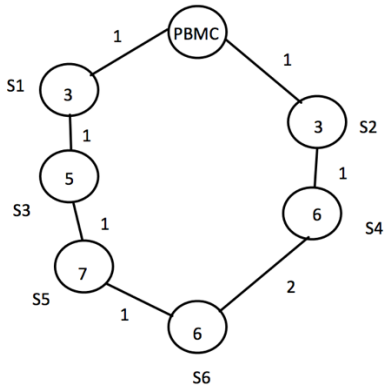

```
10 3 3 5
6 7 0
0 1 1
0 2 1
0 3 3
1 3 1
2 3 1
3 0->2->3 0
Program ended with exit code: 0
```

As shown in the figures above, the algorithm can come out with the right answer to the example. This example also shows that this algorithm can find the best path when there are more than one path with the same shortest length.



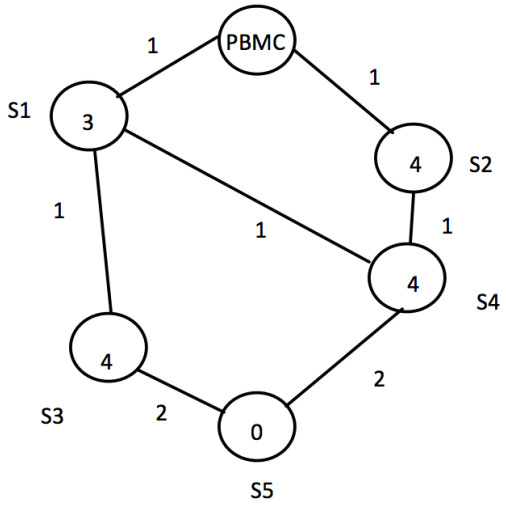

```
10 7 7 9
5 5 7 5 7 7 0
0 1 1
0 2 1
0 3 1
1 4 1
2 5 1
3 7 1
4 6 1
5 7 1
6 7 1
3 0->3->7 0
Program ended with exit code: 0
```

And in this two results of examples, we can get another conclusion. There are three paths with different lengths. But they all need three bikes from the PBMC. And the path 0->3->7 has the shortest length of three. So the best path is 0->3->7.

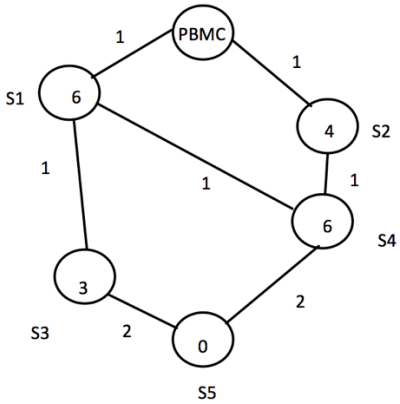



```
10 6 6 7
3 3 5 6 7 6
0 1 1
1 3 1
3 5 1
5 6 1
0 2 1
2 4 1
4 6 1
2 0->2->4->6 2
Program ended with exit code: 0
```

And if the lengths of the paths are the same, so are the numbers of bikes to send. The number of bikes to take should be considered. And as shown above. This algorithm can find the path with the least bikes to take.
Examples above only considered about the cases with divided paths. What if there are duplicate parts in different paths.




```
10 5 5 7
3 4 4 4 0
0 1 1
1 3 1
1 4 1
0 2 1
2 4 1
3 5 2
4 5 2
7 0->2->4->5 0
Program ended with exit code: 0
```




```
10 5 5 7
6 4 3 6 0
0 1 1
1 3 1
1 4 1
0 2 1
2 4 1
3 5 2
4 5 2
3 0->1->4->5 0
Program ended with exit code: 0
```

As shown above, even with the duplicate part, this algorithm can also find the best path correctly.

IV. Analysis and Comments

4.1Analysis

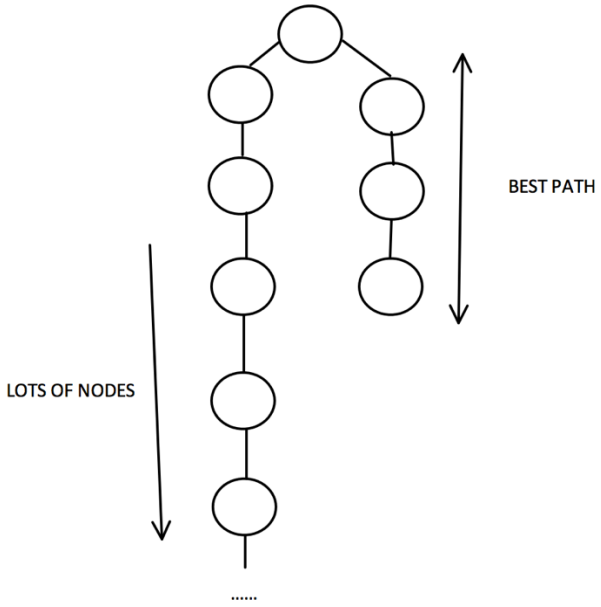
We use the adjacency matrix to represent the connections between different nodes. So the worst situation is that the algorithm should cost the time of traversal of the whole matrix. That means the time complexity is $O(N^2)$. And the space complexity is O(N) because there may be a path that contains all of the nodes.

Our algorithm is based on the theory of Depth-First-Search. Compared to the Breadth-First-Search, this way of searching doesn't have to store all the paths leaded by nodes adjacent to the PBMC, it focuses on only one path that it is going through, and the most space that it demands is the

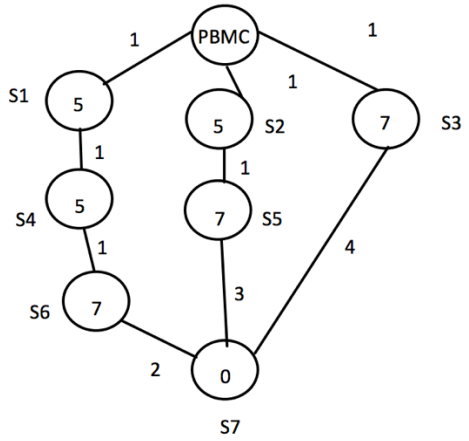
space that the nodes in this path need, not all of the nodes in the graph. So it's space complexity is much simpler.
But as the problems we will talk about in the following, this algorithm may have some extreme situations that demands much more time than Breadth-First-Search.

4.2Problems

In fact, there are some kinds of special cases that this algorithm can not handle very well.
Firstly, there is an extreme situation that one of the nodes adjacent to PBMC leads a path that is so long that the traversal of this path may take a lot of time. But in fact the best path is very short if you start with another node adjacent to the PBMC. As shown below.



And here is the second problem. It is demanded that there is only one unique path with the least bikes to send and the shortest path at the same time. But in the real life, the situation is much more complicated. And if there the example above is changed to this.




```
10 7 7 9
5 5 7 5 7 7 0
0 1 1
1 4 1
4 6 1
6 7 2
0 2 1
2 5 1
5 7 3
0 3 1
3 7 4
3 0->3->7 0
Program ended with exit code: 0
```

As shown above, all the three paths are the best paths as required. But apparently, the path 0->3->7 is the path with the least number of paths. So it is much more easy to put into action in the real life. But if we change the sequence of these three paths, this algorithm will always come out with the paths starting with the last node adjacent to PBMC. So if it is going to be used in real life, this algorithm need adjustment.

4.3Comments

As demanded, the number of the nodes in this graph is no more than 500. So there won't be a lot of differences in time for this two algorithm, even in the extreme situation. So we can say that this algorithm is suitable for this problem. As the testing results shows it is also reliable for all kinds of problems. So we made it.

Declaration

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

Duty Assignments:

Programmer: 邓墨琳3150103457

Tester:崔瑜翔3150105215

Report Writer: 耿飙3150105222

Appendix: Source Code (in C)

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

typedef enum { false, true } bool;
#define MaxVertexNum 500//最多节点数量
#define INFINITY 1000000//定义顶点不可达

typedef struct GNode *PtrToGNode;
struct GNode {
	int Nv;//顶点数
	int Ne;//边的数目
	int G[MaxVertexNum][MaxVertexNum];//邻接矩阵
};
typedef PtrToGNode MGraph;

MGraph PBMC;
int bike[MaxVertexNum] = {0};//每个站点现有的自行车数量
int Send, Take, Minsend = INFINITY, Mintake = INFINITY;
int capacity;//每个站点的最大容量
int Sp;//问题站点
int final_path[MaxVertexNum] = {0};//最终的要打印的路径
int currwei = 0;//当前的路径长
int visited[MaxVertexNum] = {false};//是否遍历过
int kk = 0;//记录下标
int minwei = INFINITY;//最短的路径长度

void g(int* path);//计算一条路径的 send 和 take
void ReadG(void);//读入数据
void dfs(int cur, int* path);//深度优先搜索算法

int main() {
	/*
	  整个主函数的思路：读入数据->通过 dfs 算法遍历图，
	  找到一条最优路径，同时算出 send 和 take->打印
	  */
	int i;
	ReadG();
	int* path = (int*)malloc(sizeof(int)*PBMC->Nv);

	for(i = 0; i < PBMC->Nv; i++) {//初始化路径
		path[i] = -1;
```

```
    }

    visited[0] = true;//置总站为已经访问
    dfs(0, path);

    printf("%d 0",Minsend);//打印路径
    for(i = 0; final_path[i] != 0; i++){
        printf("->%d",final_path[i]);
    }
    printf(" %d",Mintake);

    return 0;
}

void ReadG(void){
    PBMC = (MGraph)malloc(sizeof(struct GNode));
    //读入容量，站点数，问题站点，边的数量
    scanf("%d %d %d %d",&capacity,&(PBMC->Nv),&Sp,&(PBMC->Ne));
    PBMC->Nv++;
    int i, j, v, w, temp;
    for(i = 0; i < PBMC->Nv; i++)//初始化邻接矩阵
        for(j = 0; j < PBMC->Nv; j++)
            PBMC->G[i][j] = INFINITY;

    for(i = 1; i < PBMC->Nv; i++){//读入每个站点的单车数
        scanf("%d", &bike[i]);
    }
    for(i = 0; i < PBMC->Ne; i++){//读入邻接矩阵
        scanf("%d %d %d",&v, &w, &temp);
        PBMC->G[w][v] = PBMC->G[v][w] = temp;
    }
}

void g(int* path){
    /*注意到我们只能在去的路上对站点进行遍历，当前多出的车不能补在前面的站点中
      但是可以补在后面的站点中*/
    int i;
    Send = Take = 0;//初始化
    for(i = 0; path[i] != -1; i++){
        if(bike[path[i]] > capacity/2)//当前站点单车数量多于一半，take 增加
            Take += bike[path[i]] - capacity/2;
        else{//当前站点单车数量少于一半
            if(Take + (bike[path[i]] - capacity/2) >= 0){//从前面累计的 take 可
```

```
以补足
                    Take = Take + (bike[path[i]] - capacity/2);
                }
                else{//前面累计的 take 不能补足
                    Send += -(Take + (bike[path[i]] - capacity/2));
                    Take = 0;
                }
            }
        }
}
void dfs(int cur, int* path){
    int i;
    if(cur == Sp){//说明找到了一条符合要求的路径
        g(path);
        if( currwei < minwei ||
            (minwei == currwei && Send < Minsend) ||
            (minwei == currwei && Send == Minsend && Take < Mintake)){//
如果路径更优，那么赋值给 final path
            Minsend = Send;
            Mintake = Take;
            minwei = currwei;
            for(i = 0; i<500; i++)//每次保存 path 时，先要将 path 清空
                final_path[i] = 0;
            for(i = 0; path[i] != -1; i++)
                final_path[i] = path[i];
        }
        return;
    }
    for(i = 0; i < PBMC->Nv; i++){
        if(PBMC->G[cur][i] != INFINITY && !visited[i]){//dfs 算法的主体
            visited[i] = true;
            path[kk++] = i;
            currwei += PBMC->G[cur][i];
            dfs(i, path);//继续往深层搜索
            visited[i] = false;//重置为未访问
            path[--kk] = -1;//在当前路径中删除该站点
            currwei -= PBMC->G[cur][i];//返回当前路径长
        }
    }
}
```