

浙江大学



Fundamental Data Structure Project 2 Report

Public Bike Management

Date:2019-12-3

Contents

Part1: Introduction to the project

Part2: Algorithm and Program Instructions

Part3: Testing Results

Part4: Analysis of Algorithm and Codes

Part5: Source code

Part6: Summary

Part1: Introduction to the project

The topic of project 2 is Public Bike Management. In this project, we are focus on the public bike service in Hangzhou, where each bike station has a max capacity to keep the bikes. Besides, we define a bike station is in perfect condition if it is exactly half-full. When the number of bikes in a station is 0 or Max Capacity, the Public Bike Management Center (PBMC) will prepare to balance the number of bikes of the target station and those stations on the road to target station.

Therefore, in this project, we will get the following information:

- 1.Max Capacity of a station
- 2.The number of bike stations
- 3.The target bike station
- 4.The number of bikes in each station
- 5.The paths between bike stations

Our task is to find a shortest way from PMBC to the target station also with least numbers of bikes that will be taken from and bring back to PMBC.

The priority is shortest path>least taken bikes>least returned bikes

Part2: Algorithm and Program Instructions

Now we will show the pseudocode of the functions:

ALGORITHM: Public Bike Management

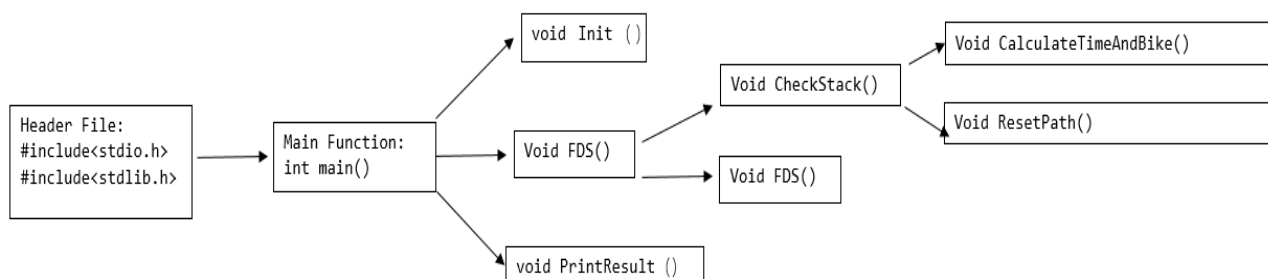
Procedure public bike management

INPUT: The data stated in the questions

OUTPUT: The min number of taken bikes and returned bikes, the shortest path from PMBC to the target station (mintake, shortest path, min returned)

```
ReadGraph();
InitializeArrays (); //These steps are written in function Init ()
Depth_First_Search (0);
// Using DFS to find all the paths from PMBC (in position 0) to the
target station
//This step is in function DFS ()
CalculateTimeAndBike ()
//compute the time cost and number of taken and returned bikes
    If time < mintime:
        mintime=time
    else if time==mintime:
        if take<mintake:
            mintake=take
        else if take==mintake
            if back<minback:
                minback=back
// compare the result in this path with the min_result with priority
if mintime || mintake || minback is changed:
    ReSetPath ( ) //mark down the new shortest path
    DFS( ) //Try to search next path
```

And the basic structure of this program is



Part3: Testing Results

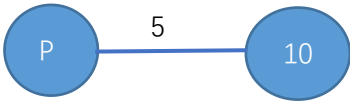
According to the question, the test data can be divided into the following four cases:

- ① There is only one path so no need for time comparison.
- ② There are multiple paths, but there is a way cost shortest time
- ③ There are multiple paths, and the shortest way is not unique. We need to compare the number of bikes sent from PBMC.
- ④ There are multiple paths, and there are shortest ways cost the same time and same number of bikes sent from PBMC. We need to compare the number of bikes sent back to PBMC.

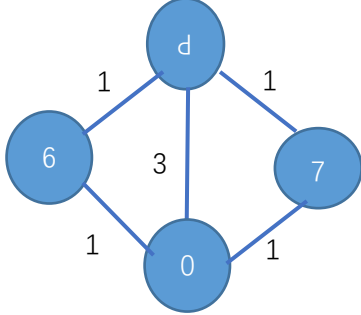
Therefore, through the above analysis, We made up some test data to test the correction of our code.

TEST1:

There is only one path so no need for time comparison.

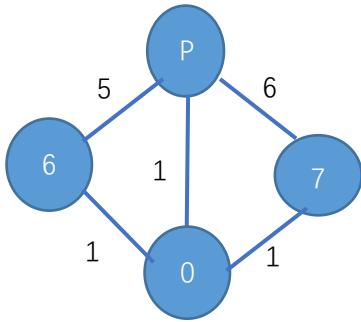
Data input	10 1 1 1 10 0 1 5
Corresponding picture	
Actual output	0 0->1 5
Testing result	Pass

TEST2: sample data, corresponding to the situation ②

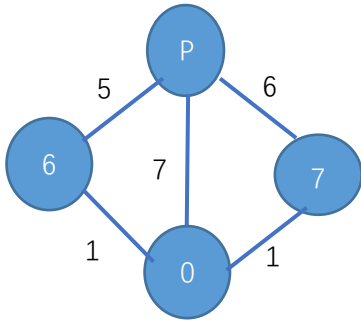
Data input	10 3 3 5 6 7 0 0 1 1 0 2 1 0 3 3 1 3 1 2 3 1
Corresponding picture	
Actual output	3 0->2->3 0
Testing result	Pass

TEST3: sample data with changed path length to test whether the program could choose the shortest path

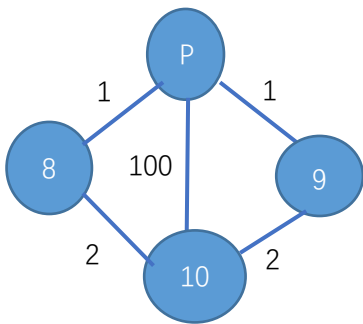
Data input	10 3 3 5 6 7 0 0 1 5 0 2 6 0 3 1 1 3 1 2 3 1
------------	--

Corresponding picture	
Actual output	5 0->3 0
Testing result	Pass

TEST4: 2 shortest paths in the graph and different taken bikes to test whether the program could choose the least taken bikes when facing more than 1 shortest path

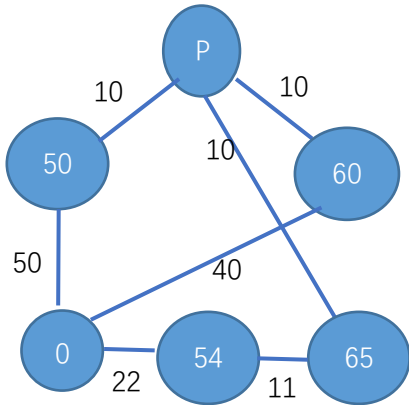
Data input	10 3 3 5 6 7 0 0 1 5 0 2 6 0 3 7 1 3 1 2 3 1
Corresponding picture	
Actual output	3 0->2->3 0
Testing result	Pass

TEST5: 2 shortest path in the graph with same number of taken bikes but different returned bikes to test whether the program could choose the least returned bikes when path length and taken bikes are all the same.

Data input	10 3 3 5 8 9 10 0 1 1 0 2 1 0 3 100 1 3 2 2 3 2
Corresponding picture	
Actual output	0 0->1->3 8
Testing result	Pass

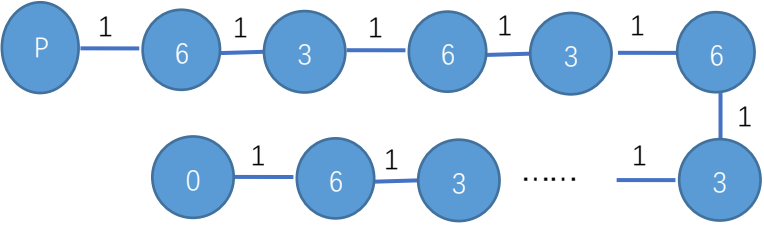
TEST6: more stations and paths to test the program when facing large datas

Data input	100 5 3 7 50 60 0 54 65 0 1 10 0 2 10 0 5 10 1 3 50 2 3 40 3 4 22 4 5 11
------------	--

Corresponding picture	
Actual output	31 0->5->4->3 0
Testing result	Pass

TEST6: a special condition that the graph only has one long path.

Data input	10 20 20 20 6 3 6 3 6 3 6 3 6 3 6 3 6 3 6 0 0 1 1 1 2 1 2 3 1 3 4 1 4 5 1 5 6 1 6 7 1 7 8 1 8 9 1 9 10 1 10 11 1 11 12 1 12 13 1 13 14 1 14 15 1 15 16 1 16 17 1 17 18 1 18 19 1 19 20 1
------------	---

Corresponding picture	
Actual output	13 0->1->2->3->4->5->6->7->8->9->10->11->12->13->14->15->16->17->18->19->20 0
Testing result	Pass

We also test the max amount of data to test the boundary conditions, but to make the project report not a messy one, we will not put the test data here.

Part4: Analysis of Algorithm and Codes

When we tried to analyze the time and complexity of the algorithm, we found it really difficult. We spent a lot of time to analyze the big- $O(n)$ complexity of our algorithm as I've written it -- all simple paths in a graph using depth-limited search as described here, and implemented here. In another word, given two vertices, we are finding all simple paths between them using a depth-first search.

We know that the time complexity of DFS is $O(V + E)$ and its space

complexity is $O(V)$, and my intuition is that the complexity of an all-paths search will be more than that.

I. Time complexity of the algorithms

Because a lot of data in this program, there are only two steps to estimate the time complexity: building a graph and finding the shortest path.

It's easy to know that the time complexity of building a graph is $O(n^2)$. We used an adjacency matrix to save all edges, and entering all edges will cost $O(n^2)$.

When the DFS () function is called to evaluate the result, the Depth-first search algorithm is used. Considering the worst case, each point relates to other points. According to the depth search, there are $n-1$ choices in the first layer, $n-2$ choices in the second layer, ... Until there is only one choice, so the time complexity is $O((n-1)!)$.

Therefore, the time complexity is $O((n-1)!)$.

II. Space complexity of the algorithms

We used an adjacency matrix to save all edges. Therefore, the space complexity is $O(n^2)$. Since the maximum path is n vertices plus the source point, the maximum space complexity of the push stack caused by recursion is $O(n^2)$, and the space complexity of the graph is $O(n^2)$.

III. Comments

After seeing the time complexity of our algorithm, you may think using the depth search algorithm directly will lead to high time complexity.

However, our algorithm only took at most 6ms in this project. If you check our code carefully, you will find we use some steps to optimize the algorithm and it's rarely to reach the worst situation.

Part5: Source Code

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

#define INFINITY 1000000 /*INFINITY means Station A and Station B aren't connected.*/
#define MaxStationNum 501 /*The maximum number of the stations.*/

typedef int WeightType;
typedef struct GNode *PtrToGNode;

/*The data structure for the graph.*/
struct GNode {
    int Cmax;    // the maximum capacity of each station
    int N;       // the total number of stations
    int Sp;      // the index of the problem station
    int M;       // the number of roads
    int Time[MaxStationNum][MaxStationNum];
    WeightType Capacity[MaxStationNum];
};

typedef PtrToGNode Graph;

void Init();
void DFS(int current);
void CheckStack();
void ResetPath();
void PrintResult();
void CaculateTimeAndBike();
```

```

int Known[MaxStationNum]; /*To mark if we have passed the
i-
th station: Known[i] equals 0 when we haven't passed the i
-
th station, while Known[i] equals 1 when we have passed th
e i-th station.*/
int Stack[MaxStationNum + 3]; /*The path we're dealing wit
h.*/
/*'Stack[MaxStationNum]' means the time we need to reach P
BMC.*/
/*'Stack[MaxStationNum + 1]' means the bikes we need to ta
ke from PBMC.*/
/*'Stack[MaxStationNum + 2]' means the bikes we need to ta
ke back to PBMC.*/
int Path[MaxStationNum + 3]; /*The most-
required path among the previous paths from PBMC to the pr
oblem station.*/
/*'Path[MaxStationNum]' means the time we need to reach PB
MC.*/
/*'Path[MaxStationNum + 1]' means the bikes we need to tak
e from PBMC.*/
/*'Path[MaxStationNum + 2]' means the bikes we need to tak
e back to PBMC.*/
int flag; /*'flag' is to verify if the array is defined.*/
int top; /*The index of the toppest element in the array n
amed 'Stack'.*/
Graph G;

int main()
{
    /*Initing the graph and the global variables*/
    Init();

    /*Using the method of depth-first-search to find the
required path and store it to the array named 'Path'.*/
    /
    DFS(0);

    /*To print the final result.*/
    PrintResult();
}

```

```

    system("pause");
    return 0;
}

/*This function is to init the graph and the global variables.*/
void Init()
{
    G = (Graph)malloc(sizeof(struct GNode));
    scanf("%d %d %d %d", &(G->Cmax), &(G->N), &(G->Sp), &(G->M));
    int i, j, k;
    G->Capacity[0] = 0;
    for(i = 1; i <= G->N; i++) scanf("%d", &(G->Capacity[i]));
    /* initialization: set the time spent per edge INFINITY*/
    for(i = 0; i <= G->N; i++) {
        for(j = 0; j <= G->N; j++) {
            G->Time[i][j] = INFINITY;
            if(i == j) G->Time[i][j] = 0;
        }
    }
    /* input the time spent per edge*/
    for(i = 1; i <= G->M; i++) {
        int StationA, StationB, time;
        scanf("%d %d %d", &StationA, &StationB, &time);
        G->Time[StationA][StationB] = time;
        G->Time[StationB][StationA] = time;
    }
    /* initialization: Known[k],Stack[k],Path[k]*/
    for(k = 0; k < MaxStationNum; k++) {
        Known[k] = 0;
        Stack[k] = -1;
        Path[k] = -1;
    }
    Path[MaxStationNum] = 0;
    Path[MaxStationNum + 1] = 0;
    Path[MaxStationNum + 2] = 0;
    top = -1;
}

```

```

    flag = 0;
}

/*This function is to use the method of depth-first-search
to find the most required path to solve the problem
*/
void DFS(int current)
{
    Known[current] = 1;
    Stack[++top] = current;
    /*Stack[MaxStationNum] is to save the time cost.*/
    if(current == 0) Stack[MaxStationNum] = 0;
    else Stack[MaxStationNum] += G->Time[Stack[top - 1]][Stack[top]];

    /*To judge if the current station is the problem station. If it's the problem station, check the current path with the array 'Path' and pop the top element out.*/
    if(current == G->Sp) {
        CheckStack();
        Stack[MaxStationNum] -= G->Time[Stack[top - 1]][Stack[top]];
        top--;
        Known[current] = 0;
        return;
    }

    /*To improve our algorithm, we add this piece of codes: If we have stored a qualified path in array 'Path', compare the current time cost with the path in array 'Path'. If its time cost is already larger than the path in array 'Path', quit the current path and search for the next qualified path.*/
    if(flag == 1 && Stack[MaxStationNum] > Path[MaxStationNum]) {
        if(current != 0) Stack[MaxStationNum] -= G->Time[Stack[top - 1]][Stack[top]];
        top--;
        Known[current] = 0;
        return;
    }
}

```

```

    }
    int i;
    /* Depth first search:search every road leading to the
station */
    for(i = 0; i <= G->N; i++) {
        if(Known[i] == 0 && G->Time[current][i] < INFINITY
) {
            DFS(i);
        }
    }
    /* Flashing back */
    Stack[MaxStationNum] -
= G->Time[Stack[top - 1]][Stack[top]];
    top--;
    Known[current] = 0;
}

/*This function is to find the better path by comparing fr
om 3 angles,the
    time we need to reach PBMC, the number of the bikes we n
eed to take from
    PBMC and the number of the bikes we need to take back to
PBMC, respectively.*/
void CheckStack()
{
    /*For the path stored in the 'Stack', to caculate the
time cost,
    the number of bikes it needs to tack from and bring
back to PBMC*/
    CaculateTimeAndBike();
    if(flag == 0) {
        ResetPath();
        flag = 1;
    }
    /*Firstly, we need to compare the time cost.*/
    else if(Path[MaxStationNum] > Stack[MaxStationNum]) Re
setPath();
    else if(Path[MaxStationNum] == Stack[MaxStationNum]) {
        /*If such a path is not unique, output the one that re
quires

```



```

        minimum number of bikes that we must take back to PB
MC.*/
        if(Path[MaxStationNum + 1] > Stack[MaxStationNum +
1]) ResetPath();
        else if(Path[MaxStationNum + 1] == Stack[MaxStatio
nNum + 1]) {
            if(Path[MaxStationNum + 2] > Stack[MaxStationN
um + 2]) ResetPath();
        }
    }
}

/*This function is to reset the array 'Path' when we find
a better path.*/
void ResetPath()
{
    int i;
    Path[MaxStationNum] = Stack[MaxStationNum];
    Path[MaxStationNum + 1] = Stack[MaxStationNum + 1];
    Path[MaxStationNum + 2] = Stack[MaxStationNum + 2];
    for(i = 0; i <= top; i++) Path[i] = Stack[i];
}

/*This function is to caculate the time we need to reach
PBMC,the number
of the bikes we need to take from PBMC and the number of
the bikes we
need to take back to PBMC if we choose the the path stor
ed in array 'Stack'.*/
void CaculateTimeAndBike()
{
    /*The variable named 'temp' is used to save the bikes
we take with us after passing the current station.*/
    int i, temp = 0;
    /*Init the the time cost, the number of bikes it needs
to tack from and bring back to PBMC.*/
    Stack[MaxStationNum] = 0;
    Stack[MaxStationNum + 1] = 0;
    Stack[MaxStationNum + 2] = 0;
    for(i = 0; i <= top; i++) {

```

```

        if(i < top) Stack[MaxStationNum] += G->Time[Stack[
i]][Stack[i + 1]];
        if(i > 0) {
            /*If the bikes in the Stack[i] are more than o
ne half of the max capacity.*/
            if(G->Capacity[Stack[i]] >= G->Cmax / 2)
                temp += (G->Capacity[Stack[i]] - G->Cmax /
2);
            /*If the bikes in the Stack[i] are less than o
ne half of the max capacity.*/
            else if(G->Capacity[Stack[i]] < G->Cmax / 2) {
                if(temp >= G->Cmax / 2 - G->Capacity[Stack
[i]]) temp -= (G->Cmax / 2 - G->Capacity[Stack[i]]);
                else {
                    Stack[MaxStationNum + 1] += (G->Cmax /
2 - G->Capacity[Stack[i]] - temp);
                    temp = 0;
                }
            }
        }
    }
    /*After passing the last station, variable 'temp' equa
ls to the number of bikes we need to take bake to PBMC.*/
    Stack[MaxStationNum + 2] = temp;
}

/*This function is to print the final result.*/
void PrintResult()
{
    /*To print the bikes we need to take from PBMC.*/
    printf("%d ", Path[MaxStationNum + 1]);
    /*To print the path.*/
    int i = 0;
    for(i = 0; ; i++) {
        printf("%d", Path[i]);
        if(Path[i] != G->Sp) printf("->");
        else {
            printf(" ");
            break;
        }
    }
}

```

```
}  
/*To print the bikes we need to take back to PBMC.*/  
printf("%d", Path[MaxStationNum + 2]);  
}
```

Part6: Summary

In this project, we divided the tasks in 3 parts, code writing, report writing and designing test samples. Project 2 is much harder than Project 1. It needs a precise comprehension on algorithm design, the method of depth-first-search as well as algorithm improving.

When the coder checked our code in the PTA platform, we found that our code cannot pass a test sample for the reason of Running-time-out. After our analysis and discussion, we found that we can improve our algorithm further.

Declaration

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