Public Bike Management

2019-11-30

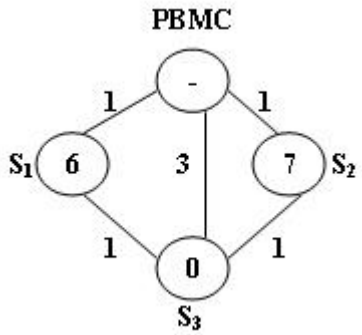
CONTENES

1 Introduction

1.1 Description

This problem is a shortest path problem. There's a PMBC that monitoring all the bike stations in Hangzhou City. Each station has a same bike capacity which is an even number. If the number of bikes of a station is half of the capacity, then the station is in perfect condition. In this problem, we need to adjust the problem station which is reported in the input to perfect condition, and all the stations along the path will be adjusted as well (by taking bikes from PMBC and the stations along the way or taking back to PMBC). For the sake of efficiency, we have to choose the shortest path from PMBC to the problem station. The "shortest path" is defined as follows:

- The path which takes the minimum time to travel from PMBC to the problem station.
- If there're more than one paths that takes the same minimum time, then the path which needs to take the minimum number of bikes from PMBC will be selected.
- If there're still more than one paths that takes the same number of minimum bikes from PMBC, then the one which takes the minimum number of bikes back to PMBC will be chosen. (the judge's data guarantee that such a path is unique)



For example, in this graph, there're two paths that takes the same minimum time from PMBC to the problem station S_3: PMBC->S_1->S_3 and PMBC->S_1->S_3. But the latter one needs less bike (3) from PMBC, so that one will be chosen.

2 Algorithm Specification

2.1 Dijkstra Algorithm

The algorithm of graph reading is well known, so we won't go into details here. We will show the implementation of two algorithms. One is Dijkstra algorithm to find the shortest path. **Unlike the general situation, we add an array named *size* to record the number of the same shortest path of each vertex.**

Here is the Dijkstra algorithm:

```
1 function Dijkstra(Graph, **path, *size, *isKnown, *dist):
2
3     create vertex set Q
4
5     for each vertex v in Graph:
6         dist[v] ← INFINITY
```

```
7         isKnown[v] ← UNDEFINED
8         add v to Q
9      //0 is the index of PMBC
10      dist[0] ← 0
11
12      while Q is not empty:
13          u ← vertex in Q with min dist[u]
14         remove u from Q
15
16          for each neighbor v of u:
17              alt ← dist[u] + length(u, v)
18     //the shortest path is found
19             if alt < dist[v]:
20                 dist[v] ← alt
21                 prev[v] ← u
22                  size[u] ← 1
23                  path[u][size[u]-1] ← v
24     //find another path with same distance
25     //increment size[u] and append the last node to path[u]
26             if alt == dist[v]:
27                  size[u]++
28                  path[u][size[u] - 1] ← v
29
30     for each vertex v in Graph:
31           if dist[v] == INFINITY
32                  dist[v] ← -1
```

2.2 Depth First Search Algorithm

Another algorithm we want to show is to use **depth first search** to find the shortest path with the least cost, that is, to find our output. We use **stack** in this algorithm.

Here is the depth first search algorithm:

```
1  function DFS_Search(Graph, *path, *size, V):
2     //tempPath is an array that store the temporary shortest
3   path and the resultPath is the result of the shortest path.
4     Push(tempPath, V);
5     //if reach the search end, that is V==0
6     if V == 0 :
7        for each s in the tempPath:
8           bikeDiff ← Graph->bike[s]- Capacity / 2
9     //more than perfect, send back to margin
10           if bikeDiff > 0:
```

```
11              tempback ← tempback + bikeDiff
12    //if bikes are needed to be sent back
13    //more than current station's needs
14          else if tempback + bikeDiff > 0:
15                tempback ← tempback + bikeDiff
16    //all the temporary back serve as the need for statio.
17              else tempNeed ← tempNeed – bikeDiff -tempBack
18    //set the tempBack to zero
19                    tempBack = 0
20    //update the resultPath
21       if tempNeed < minNeed or tempBack < minBack:
22          copy tempPath to resultPath
23          minNeed = tempNeed
24          minBack = tempBack
25        Pop(tempPath)
26    for each u in size[V]:
27       DFS_Search(Graph, *path, *size, path[V][u])
28 Pop(tempPath)
```

2.3 Correctness of the Algorithm

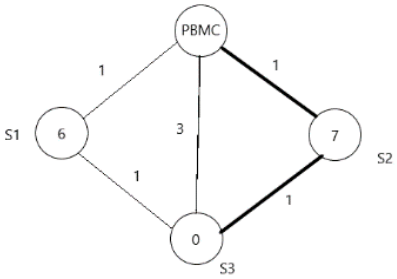
In this algorithm, we first using Dijkstra's Algorithm to find all the shortest paths in the graph. In the normal implementation of Dijkstra, once we find the minimum unknown vertex(MUV), we have to update all the vertices adjacent to it by setting the MUV as their previous vertex along the shortest path, so that we can trace the path after the Dijkstra's Algorithm. In this case, however, there may be more than one shortest paths, so the previous vertex of a specific vertex may not be unique. So we use a pointer array "path", which is malloced to store all the previous vertices of a vertex, along with the array "size" to store the number of the previous vertices. For example, let's say vertex *6* is a vertex in several shortest paths, and its previous vertices are *2, 7, 9,* and *8* (that is to say, edge (2,6), (7,6), (9,6) and (8,6) are all edges in some shortest paths). In this case, the "path[6]" will store a pointer to an array [2,7,9,8], and "size[6]" will be set to 4 since the size of the path[6] is 4.

After all the shortest paths are recorded, we then use Depth First Search to find the path that most qualifies the definition of "shortest" we have brought up in the **Description** section.

3 Testing Results

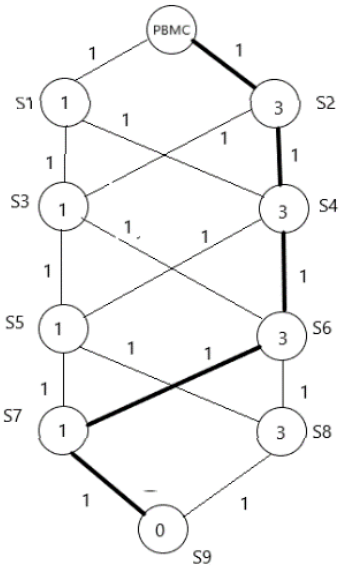
3.1 sample equivalent

```
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
```



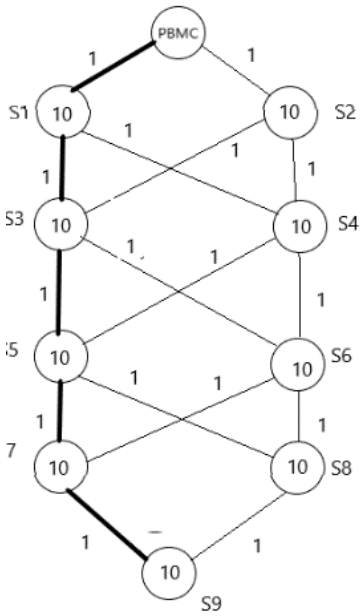
3.2 less bike, but longer path

```
4 9 9 16
1 3 1 3 1 3 1 3 0
0 1 1
0 2 1
1 3 1
1 4 1
2 3 1
2 4 1
3 5 1
3 6 1
4 5 1
4 6 1
5 7 1
5 8 1
6 7 1
6 8 1
7 9 1
8 9 1
0 0->2->4->6->7->9 0
```



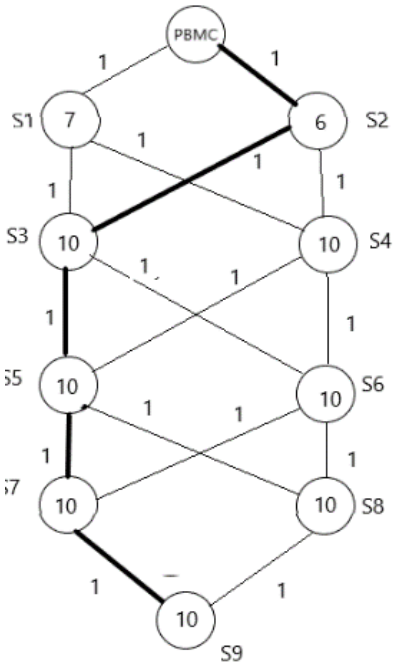
3.3 all too full

```
10 9 9 16
10 10 10 10 10 10 10 10 10
0 1 1
0 2 1
1 3 1
1 4 1
2 3 1
2 4 1
3 5 1
3 6 1
4 5 1
4 6 1
5 7 1
5 8 1
6 7 1
6 8 1
7 9 1
8 9 1
0 0->1->3->5->7->9 25
```



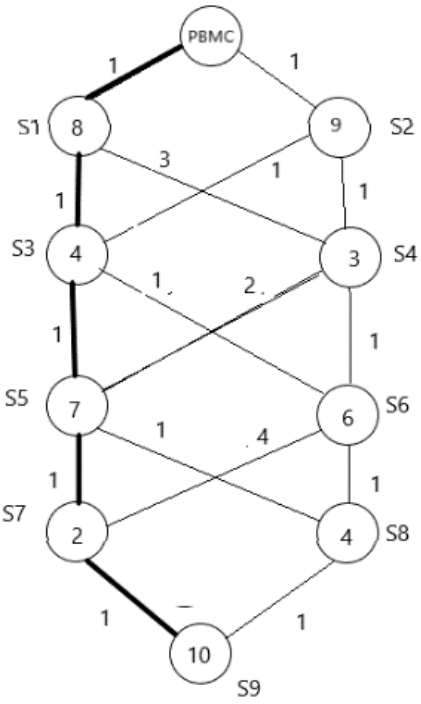
3.4 begin with less full vertex, then too full

```
10 9 9 16
7 6 10 10 10 10 10 10 10
0 1 1
0 2 1
1 3 1
1 4 1
2 3 1
2 4 1
3 5 1
3 6 1
4 5 1
4 6 1
5 7 1
5 8 1
6 7 1
6 8 1
7 9 1
8 9 1
0 0->2->3->5->7->9 21
```



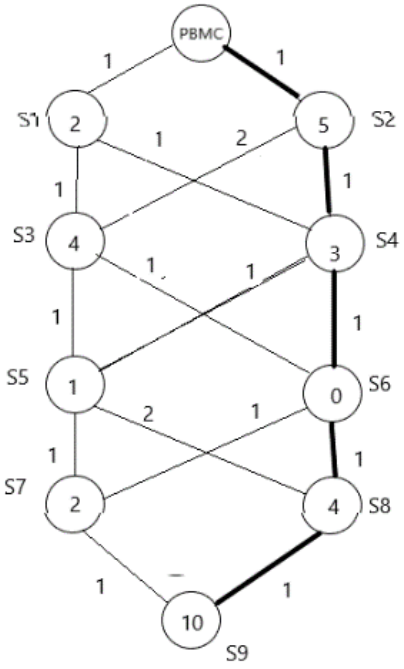
3.5 more and less, alternating

```
10 9 9 16
8 9 4 3 7 6 2 4 10
0 1 1
0 2 1
1 3 1
1 4 3
2 3 1
2 4 1
3 5 1
3 6 1
4 5 2
4 6 1
5 7 1
5 8 1
6 7 4
6 8 1
7 9 1
8 9 1
0 0->1->3->5->7->9 6
```



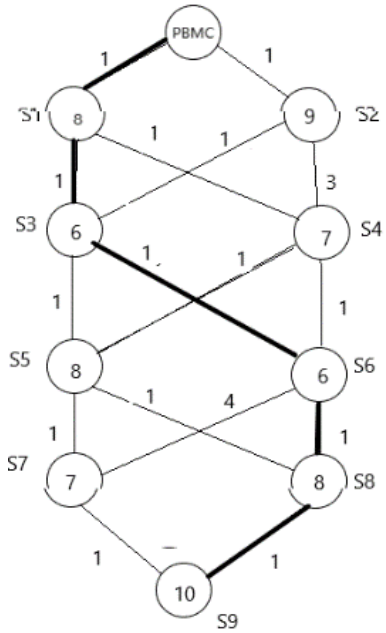
3.6 Sp is all full, but others are less full

```
10 9 9 16
2 5 4 3 1 0 2 4 10
0 1 1
0 2 1
1 3 1
1 4 1
2 3 2
2 4 1
3 5 1
3 6 1
4 5 1
4 6 1
5 7 1
5 8 2
6 7 1
6 8 1
7 9 1
8 9 1
8 0->2->4->6->8->9 5
```

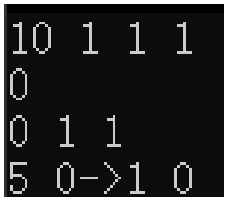


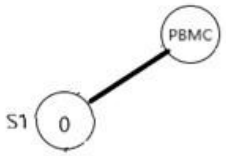
3.7 Sp is all full, and others are too full

```
10 9 9 16
8 9 6 7 8 6 7 8 10
0 1 1
0 2 1
1 3 1
1 4 1
2 3 1
2 4 3
3 5 1
3 6 1
4 5 1
4 6 1
5 7 1
5 8 1
6 7 4
6 8 1
7 9 1
8 9 1
0 0->1->3->6->8->9 13
```



3.8 Direct Move





3.9 Maxsize

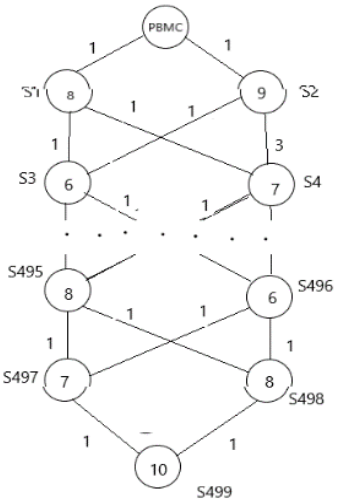
- Input: see the “max size test.txt” file
- Output: 0 0->187->500 9

4 Analysis and Comments

4.1 Time complexity:

Let V to be the vertex number, E to be the edge number. First, we need to find the shortest path. In our algorithm, we use Dijkstra Algorithm to accomplish this task. For each loop, we find the minimum unknown vertex. Keep doing this until we find all the shortest path. We define an array to record distance, the time complexity of initial the array is O(*V*). In the process, we search distance to find the minimum unknown vertex, the time complexity is O(*V*). In average, we do this loop for V times, so the time complexity is O(V^2).

Then, we search the qualified shortest path. We calculate the bike needed of all the path from Sp to the PMBC to determine the path we need, so we think the average time complexity of depth first search is O(*V+E*).



However, the interesting thing is coming. If there are V/2 layers, we can see that there are $2^{\frac{V}{2}}$ shortest paths in this case. We need to store them, and then there will be shortest paths. This situation is very extreme, which is not considered in general.

4.2 Space complexity:

In the Dijkstra Algorithm, we define *isKnown to remark the situation of the vertex, and *dist to record the distance of each vertex, size to count the number of the last vertex, path to remember the different shortest path. So the space complexity is O(V+E).

In the search process, we record path, so the space complexity is O(E).

In conclusion, the space complexity of our algorithm is O(V+E).

As for further improvement, in the finding the shortest part, we can keep distances in a priority queue, get the shortest path in a quicker way.

5 Appendix

Source Code (in C):

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

typedef enum { false, true } bool;
#define INFINITY 1000000
#define MaxVertexNum 501  /* maximum number of vertices */
typedef int Vertex;     /* stations are numbered from 1 to MaxVertexNum */
typedef int WeightType;
typedef struct GNode* PtrToGNode;
typedef PtrToGNode MGraph;

/*****************************************************
* The data structure of the Graph
* Nv:      the number of vertices of the graph(that is the number of stations puls the PBMC)
* Ne:      the number of edges of the graph
* bike[]:  an array to store the initial number of bikes of each station
* G[][]:   the adjacency matrix that stores the time needed to travel from one station to another
*****************************************************/
struct GNode {
    int Nv;
    int Ne;
    int bike[MaxVertexNum];
    WeightType G[MaxVertexNum][MaxVertexNum];
};

/*****************************************************
* Global variable
* Capacity:   the mamximum capacity of each station
* N:          the total number of stations
* Sp:         the index of the problem station
* M:          the number of roads(edges)
* minNeed:    the the minimum number of bikes needed from the PBMC
* minBack:    the minimum number of bikes which will be sent back to the PBMC after adjustion
* resultPath: a stack which stores the result path (in reverse order), and the resultPointer is
its stack pointer
* tempPath:   a stack which stores the temporary path (in reverse order), and the tempPointer is
its stack pointer
*****************************************************/
int Capacity, N, Sp, M;
int minNeed = INFINITY, minBack = INFINITY;
Vertex resultPath[MaxVertexNum], tempPath[MaxVertexNum];
```

```
int resultPointer = -1, tempPointer = -1;

/*****************************************************
* Functions Specification
* ReadG():       read the graph in
* Dijkstra():    using Dijkstra's Algorithm to find all the shorteset paths, and store all the
shortest paths in the adjacency list path[], with size in the array size[]
* search():      using DFS to search the qualified shortest path
* printResult(): output the result
* adjust():      store the temprary result path in the resultPath(update resultPath)
* push():        the basic stack operations needed for the search function
* pop():         the basic stack operations needed for the search function
*****************************************************/
MGraph ReadG();
void Dijkstra(MGraph Graph, Vertex* path[], int* size);
void search(MGraph Graph, Vertex* path[], int* size, Vertex S);
void printResult(void);
void adjust(Vertex* resultPath, Vertex* tempPath);
void push(Vertex* stack, Vertex V);
void pop(Vertex* stack);

int main()
{
    int size[MaxVertexNum] = { 0 };         //store the number of last nodes
    Vertex* path[MaxVertexNum] = { NULL };  //store all the last nodes of the vertex, since the
path is not unique

    MGraph Graph = ReadG();                 //read in the graph

    Dijkstra(Graph, path, size);            //find all the shotest paths and store them in tha
array "path"

    search(Graph, path, size, Sp);          //find the path that qualify the definition of
"shortest"

    printResult();                          //output the result in the requested form

    return 0;
}

MGraph ReadG()
{
    int i, j, k;
    MGraph Graph = (MGraph)malloc(sizeof(struct GNode));
```

```
    WeightType Cij;
    scanf("%d %d %d %d", &Capacity, &N, &Sp, &M);
    Graph->Nv = N + 1, Graph->Ne = M;           //the number of vertices if the number of station
plus the PBMC

    for (i = 1; i < Graph->Nv; i++)             // read the number of bikes for each station and
initial the number of bikes of PBMC to INFINITY
        scanf("%d", &Graph->bike[i]);
    Graph->bike[0] = INFINITY;

    for (i = 0; i < Graph->Nv; i++)             //initialize the time taken from one station to
another
        for (j = 0; j < Graph->Nv; j++) {
            if (i == j)                         //the time taken from one station to itself is
zero
                Graph->G[i][j] = 0;
            else
                Graph->G[i][j] = INFINITY;      //otherwise initialize it to INFINITY

        }

    for (k = 0; k < Graph->Ne; k++) {           //read the time taken to move between station i
and j
        scanf("%d %d %d", &i, &j, &Cij);
        Graph->G[i][j] = Graph->G[j][i] = Cij;
    }

    return Graph;
}

void Dijkstra(MGraph Graph, Vertex* path[], int* size)
{
    bool* isKnown = (bool*)calloc(Graph->Nv, sizeof(bool));     //an auxiliary array that helps to
record if a station is known
    WeightType* dist = (WeightType*)calloc(Graph->Nv, sizeof(WeightType));  //an auxiliary array
that helps to find the minimum unknown vertex
    int minUnknownVertex, minUnknownLen, i, flag;

    for (i = 0; i < Graph->Nv; i++)
        dist[i] = INFINITY;
    dist[0] = 0;

    while (1) {                                     // find the minimum unknown vertex
        minUnknownLen = INFINITY;
```

```
        minUnknownVertex = -1;
        flag = 0;
        for (i = 0; i < Graph->Nv; i++) {
            if (!isKnown[i] && dist[i] < minUnknownLen) {       //if a vertex is still unknown and the distance to PBMC is less than the current minUnknownLen, upudate all the variables and set the flag that we have found one vertex(but may not be the minUnknownVertex)
                flag = 1;
                minUnknownLen = dist[i];
                minUnknownVertex = i;
            }
        }
        if (!flag)          //all the vertices are known
            break;

        isKnown[minUnknownVertex] = true;
        for (i = 0; i < Graph->Nv; i++) {
            int len = Graph->G[minUnknownVertex][i];
            if (len == INFINITY)                                        //i is not adjacent to the minUnknownVertex
                continue;
            else if (len < INFINITY) {
                if (!isKnown[i]) {
                    if (dist[minUnknownVertex] + len < dist[i]) {           //the shortest path is newly found, set the size[i] to 1 and put the last node(minUnknownVertex) in the path[i]
                        dist[i] = dist[minUnknownVertex] + len;
                        size[i] = 1;
                        path[i] = realloc(path[i], size[i] * sizeof(Vertex));
                        path[i][size[i] - 1] = minUnknownVertex;
                    }
                    else if (dist[minUnknownVertex] + len == dist[i]) {     //find another path with same distance, increment size[i] and append the last node to the path[i]
                        path[i] = realloc(path[i], (++size[i]) * sizeof(Vertex));
                        path[i][size[i] - 1] = minUnknownVertex;
                    }
                }
            }
        }
    }
    for (i = 0; i < Graph->Nv; i++)
        dist[i] = (dist[i] == INFINITY ? -1 : dist[i]);

    free(isKnown);
    free(dist);
}
```

```
void search(MGraph Graph, Vertex* path[], int* size, Vertex V)
{
    int i;
    push(tempPath, V);
    if (V == 0) {                           //get to the search end
        int tempNeed = 0, tempBack = 0;
        Vertex S;
        WeightType bikeDiff;
        for (i = tempPointer - 1; i >= 0; i--) {
            S = tempPath[i];
            bikeDiff = Graph->bike[S] - Capacity / 2;
            if (bikeDiff > 0)           //more than perfect, send back the margin
                tempBack += bikeDiff;
            else {
                if (tempBack > (-1 * bikeDiff)) //if the bikes needed to send back is more than
the current station's needs, then reduce the send back
                    tempBack += bikeDiff;
                else {
                    tempNeed += (-1 * bikeDiff) - tempBack; //all the temprary back serve as the
need for this station, and set tempBack to zero
                    tempBack = 0;
                }
            }
        }
        if (tempNeed < minNeed) {                       //update the resultPath
            minNeed = tempNeed;
            minBack = tempBack;
            adjust(resultPath, tempPath);
        }
        else if (tempNeed == minNeed && tempBack < minBack) {
            minBack = tempBack;
            adjust(resultPath, tempPath);
        }
        pop(tempPath);
        return;
    }
    for (i = 0; i < size[V]; i++)                       //search along the path
        search(Graph, path, size, path[V][i]);
    pop(tempPath);
}

void printResult(void)
{
```

```
    int i = 0;
    printf("%d 0", minNeed);
    for (i = resultPointer - 1; i >= 0; i--)
        printf("->%d", resultPath[i]);
    printf(" %d\n", minBack);
}


void push(Vertex* stack, Vertex V)              //push the stack of the caller
{
    if (stack == resultPath)
        resultPath[++resultPointer] = V;
    else tempPath[++tempPointer] = V;
}

void pop(Vertex* stack)                     //pop the stack of the caller
{
    if (stack == resultPath)
        resultPointer--;
    else tempPointer--;
}

void adjust(Vertex* resultPath, Vertex* tempPath)           //copy the tempPath to the resultPath
{
    int i;
    for (i = 0; i <= tempPointer; i++)                      //just store all the elements in the
tempPath to resultPath
        resultPath[i] = tempPath[i];
    resultPointer = tempPointer;
}
```

6 Declaration

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

7 Duty Assignment:

Programmer: 王睿

Tester: 杨云皓

Report Writer:张佳文