Public Bike Management Project

Author:蒋鹏飞，张雯琪

Date: 2019-12-11

Chapter 1: Introduction

1.1 Problem description:

This project aims to design one public bike management system in Hangzhou city. Each bike station is at the mercy of the Public Bike Management Center(PBMC) and can reach the perfect state only when the number of bikes stored is equal to its half capacity.

When circumstances occur like one station is full or empty, PBMC will send or take back bikes from this station as well as adjusting the stations along the way to the perfect state.

The path will be chosen only when it costs the shortest time and requires the least number of bikes that should be sent or taken back.

1.2 Algorithm Background

1.2.1 Dijkstra's Shortest Path Algorithm

To find out the least time spent from PBMC to the problem station.

```
procedure Dijkstra_Insert(x, y : vertex; w_xy : weight);
begin
Step 1
    insert edge (x, y) with weight w_xy in graph G;
Step 2
    if d(x) + w_x,y ≥ d(y) then EXIT;  {no distance from the source has been improved}
Step 3
    Q ⟵ ∅;  {initialization of the global heap}
    insert in Q vertex y with priority d(x) + w_x,y and candidate parent x;
    for all vertices q ∈ V do mark(q) ⟵ false;
Step 4
    while non-Empty(Q) do
        begin
            delete from Q vertex q with minimum priority b_q and candidate parent z;
            d(q) ⟵ b_q;
            set the new parent of vertex q to be z;
            mark(q) ⟵ true;
            for all edges (q, r) do
                if d(q) + w_q,r < d(r) and mark(r) = false
                  then if r is not in Q
                           then insert r in Q with priority d(q) + w_q,r and candidate parent q
                           else if d(q) + w_q,r ≤ current-priority of r in Q
                                then begin
                                         update priority of r in Q to d(q) + w_q,r;
                                         set q to be the candidate parent of r;
                                     end
        end
end
```

1.2.2 time.h

To use C's standard library time.h to measure the performance of the function. By repeating the function calls for K times, we can obtain a long enough total run time and divide it to get a more accurate duration for a single run of the function.

Chapter 2: Algorithm Specification

2.1 Main data structures

2.1.1 Graph

```c
typedef struct GraphRecord *PtrToGraph;
struct GraphRecord{
    int Nv;
    int Ne;
    int Edge[MaxNum+1][MaxNum+1];
};
typedef PtrToGraph Graph;
```

For all the bike stations and each path, establish a graph to those variables, with the number of stations stored in Nv, the number of paths between two stations stored in Ne, time spent between two stations stored in Edge[station1][station2]. In additional, as it is an undirected graph, Edge[station1][station2] is equal to Edge[station2][station1].

2.1.2 Path

```c
int Path[MaxNum][MaxNum]
```

For all the paths from the PBMC to the problem station, store the index of the path in Path[index], the number of stations along the way in Path[][0], and each station from Path[][1] to the end.

2.2 Algorithms

2.2.1 Function ShortestTime()

Use Dijkstra algorithm to count the shortest time from PBMC to the problem station

```
function ShortestTime()
    Station V
    while 1 do
        Find one unknown station V with the least time
        for i = 1 to G->Nv do
            if known[i] = 0 and time[i] < time[V] then
                V <- i
            end if
        end for

        if V = the problem station then
            break
        end if

        Update the time spent from the PBMC to the station V
        for i = 0 to G->Nv do
            if G->Edge[V][i] > 0 then
                if time[V] + G->Edge[V][i] < time[i] then
                    time[i] <- time[V] + G->Edge[V][i]
                end if
            end if
        end for
    end while
    return time[V]
end function
```

2.2.2 Function FindPath

Store each path in a stack. For each top elements, if its adjacent vertex meet the requirements of the vertex along the shortest path, push it into the stack until the problem station is reached. Else if there exists no such vertex, pop out the top element.

```
function FindPath()
    station a
    if there is one path from station i to j then
        visit[i][j] <- 1
    end if
    Push(PMBC, Stack)
    while top > -1 do
        cnt <- 0
        a <- Top(Stack)
        for i = 1 to G->Nv do
            if G->Edge[a][i] > 0 and visit[a][i] = 0 and i ∉ Stack then
                visit[a][i] <- 1
                CntTime <- CntTime + G->Edge[a][i]
                if CntTime > MinTime then
                    CntTime <- CntTime - G->Edge[a][i]
                    cnt <- cnt + 1
                    continue
                else
                    Push(i, Stack)
                    if i = P and CntTime = MinTime then
                        for each station j in the stack do
                            Path[PathNum][j+1] <- Stack[j]
                        end for
                        Path[PathNum][0] <- the number of stations along this path
                        CntTime <- CntTime - G->Edge[a][i]
                        Pop(Stack)
                        cnt <- cnt + 1
                        continue
                    else
                        break
                    end if
                end if
            else
                cnt <- cnt + 1
            end if
        end for
        if cnt = G->Nv then
            a <- Top(Stack)
            CntTime <- CntTime - G->Edge[Stack[top-1]][a]
            for i = 1 to G->Nv do
                visit[a][i] <- 0
            Pop(Stack)
        end if
    end while
end function
```

2.2.3 Function CntBike()

BikeNow means the number of bikes that can be carried along the path while BikeRecord means the number of bikes should be sent from PBMC. For each vertex along each path, update the number of bikes carried and sent.

```
function CntBike()
    int BikeRecord, BikeNow
    for i = 0 to PathNum-1 do
        BikeNow <- 0
        BikeRecord <- 0
        for j = 2 to Path[i][0] do
            if BikeNum[Path[i][j]] >= cap/2 then
                BikeNow <- BikeNow + BikeNum[Path[i][j]] - cap/2
            else if BikeNow >= cap/2 - BikeNum[Path[i][j]] then
                BikeNow <- BikeNow - cap/2 - BikeNum[Path[i][j]]
            else
                BikeRecord <- BikeRecord + cap/2 - BikeNum[Path[i][j]] - BikeNow
                BikeNow <- 0
            end if
        end for
        SentBike[i] <- BikeRecord
        TakeBack[i] <- BikeNow
    end for
end function
```

2.2.4 Function SortBike()

Figure out the smallest SentBike for each path, and choose the one with smaller TakeBike if their SentBikes are the same.

```
function SortBike()
    int minsent, mintake <- INFINITY
    for i = 0 to PathNum-1 do
        if SentBike[i] = minsent then
            if TakeBack[i] < mintake then
                mintake <- TakeBack[i]
                index <- i
            end if
        end if
        else if minsent > SentBike[i] then
            minsent <- SentBike[i]
            mintake <- TakeBack[i]
            index <- i
        end if
    end for
    return index
end function
```

Chapter 3: Testing Results

Case 1(from PTA, more than 1 shortest path):

Input:

```
10 3 3 5
6 7 0
0 1 1
0 2 1
0 3 3
1 3 1
2 3 1
```

Output:
```
3 0->2->3 0
```

Case 2(cycle):

Input:

```
12 7 6 12
6 7 12 8 10 0 7
0 1 1
0 2 1
0 3 3
1 3 1
1 4 1
2 3 2
2 6 8
2 7 2
4 5 4
3 5 5
3 6 6
7 6 7
```

Output:
```
0 0->1->3->6 0
```

Case 3(complex):

Input:

```
100 48 48 64
51 52 53 54 55 56 ... 93 94 95 96 97 98
0 1 1
0 2 1
3 4 1
3 5 1
6 7 1
6 8 1
9 10 1
9 11 1
12 13 1
12 14 1
15 16 1
15 17 1
18 19 1
18 20 1
21 22 1
21 23 1
24 25 1
24 26 1
27 28 1
27 29 1
30 31 1
30 32 1
33 34 1
33 35 1
36 37 1
36 38 1
39 40 1
39 41 1
42 43 1
42 44 1
45 46 1
45 47 1
3 2 1
3 1 1
6 5 1
6 4 1
9 8 1
9 7 1
12 11 1
12 10 1
15 14 1
15 13 1
18 17 1
18 16 1
21 20 1
21 19 1
24 23 1
24 22 1
27 26 1
27 25 1
30 29 1
30 28 1
33 32 1
33 31 1
36 35 1
36 34 1
39 38 1
39 37 1
42 41 1
42 40 1
45 44 1
45 43 1
48 47 1
48 46 1
```

Output:

```
0 0->1->3->4->6->7->9->10->12->13->15->16->18->19->21->22->24->25->27->28->31->33->34->36->37->39->40->42->43->45->46->48 784
```

Other cases:

In chapter 4 we write a function to randomly create input data to test the time complexity, thus omitted here.

Chapter 4: Analysis and Comments

4.1 Time complexity analysis

We consider that the time complexity is related to two variables. One is the number of vertexes, represented by V, the other is the number of edges, represented by E.

The main function consists of 4 functions and one output process：

Function ReadGraph() O(V, E)=V+V^2+E

As there are four loops and one is inside another.

Function ShortestTime() O(V, E)=V+ElogV

As there is one loop for initializing and the use of Dijkstra algorithm.

Function FindPath() O(V, E)=EV^2

As in the worst case, each vertex will be visited for V times and repeated for E times.

Function CntBike() O(V, E)=EV

As in the worst case, the number of path is equal to E, and the number of vertex along each path is equal to V.

Output Process O(V, E)=V

As in the worst case, the number of vertex that should be printed out is equal to V.

Thus the time complexity O(V, E)=V+E+EV+V^2+EV^2+ElogV

4.2 Time complexity test

The variable ‘vertexnum’ is the number of vertexes, and the variable ‘saturation’ is defined as the number of average edges at one vertex, equal to E divided by vertexnum-1.

In order to test time complexity, we write a function to create input data randomly. Here is the definition of the function:

```
void CreatData(int vertexnum,int maxnum,int maxdistance,double saturation,int problemstation)
```

Variable ‘maxnum’ is the maximum number of bikes of each station; the value of each edge is from 1 to variable ‘maxdistance’; the variable ‘problemstation’ is the number of the problem station, it’s value would not affect the result because of randomness.

In the function, each station has random bikes from 0 to maxnum. And the probability of ‘saturation’ for each edge[i][j](i<j) have a value of rand(1,maxdistance),other’s value is INFINITY. So we build such a random input data. The detail of this function is in the appendix.

The table of time complexity test result:

saturatuon / time(ms) / vertexnum	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
10	0.124	0.129	0.134	0.15	0.158	0.165	0.164	0.177	0.257	0.208
100	1.57	2.66	3.86	5.03	6.1	7.3	8.4	9.6	10.7	11.92
200	5.97	10.58	15.42	20.4	25	29.67	34.45	39.46	44.17	48.93
500	35.75	66.04	96.07	126.29	156.52	187.66	217.47	247.4	271	304.4
1000	143	268.3	386.8	508.2	631.3	752.4	877.7	998.2	1120	1236
2000	596	1147	1598	2099	2604	3149	3622	4128	4656	5113
5000	3826	6958	10162	13284	16406	19641	22781	25837	29141	32226
10000	14982	27904	40242	53402	65729	79314	92422	107542	119521	132827

4.3 Space complexity analysis

O(N)=N^2+N, as there exist 2-dimension array and 1-dimension array variables.

Appendix: Source Code (in C)

BikeManagement.c

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

#define INFINITY 1000000
#define MaxNum 500//Number of stations
typedef int Vertex;//Index of stations

typedef struct GraphRecord *PtrToGraph;
struct GraphRecord{
    int Nv;
    int Ne;
    int Edge[MaxNum+1][MaxNum+1];           //Time cost between two stations
};
typedef PtrToGraph Graph;                   //Define the structure of the graph
Graph ReadGraph();                          //Handle the input files
int ShortestTime();                  //Get the shorteset time
void FindPath();                    //Store every path within shortest time in Path[PathNum][]
void CntBike();                             //Count the number of bikes sent and taken back along the path
int SortBike();                             //Find out the path with the least number of bikes sent and taken back

Vertex P;//The index of the problem station
Graph G;
int cap;//Record the capibility of each station
int BikeNum[MaxNum];//Record the number of bikes in each station

int MinTime, time[MaxNum], CntTime = 0;//Record the least time, the time to reach PBMC, the time spent along one selected way
int Path[MaxNum][MaxNum], PathNum = 0;//Store each path, and the number of vertex along the path is stored in Path[PathNum][0]
int Stack[MaxNum], top = -1;//Store index along the path in process
static int SentBike[MaxNum], TakeBack[MaxNum];//Record the number of bikes sent or taken back along the way
int main()
{
    G = ReadGraph();//Read the graph
    MinTime = ShortestTime();//Get the shortest time spent
    FindPath();//Find each path from PBMC to the problem station
    CntBike();//Get the number of bikes sent or taken back along each path
    int num = SortBike();//Get the index of the path with the least number of sent bikes
    printf("%d ", SentBike[num]);
    for(int i = 1; i <= Path[num][0]; i++)
    {
        if(i == 1) printf("0");
        else printf("->%d", Path[num][i]);
    }
    printf(" %d", TakeBack[num]);
    return 0;
}

Graph ReadGraph()
{
    int i, j;
    Graph G = (Graph)malloc(sizeof(struct GraphRecord));
    scanf("%d %d %d %d", &cap, &G->Nv, &P, &G->Ne);
    for(i = 1; i <= G->Nv; i++)
        scanf("%d", &BikeNum[i]);
    for(i = 0; i <= MaxNum; i++)
        for(j = 0; j <= MaxNum; j++)
            G->Edge[i][j] = -1;//Initialize

    for(i = 0; i < G->Ne; i++)
    {
        int m, n, tmp;
        scanf("%d %d", &m, &n);
        scanf("%d", &G->Edge[m][n]);
        G->Edge[n][m] = G->Edge[m][n];
    }
    return G;
}
```

```c
int ShortestTime()
{
    int i;
    static int known[MaxNum];
    for(i = 1; i <= G->Nv; i++)
    {
        if(G->Edge[0][i] > 0)
            time[i] = G->Edge[0][i];
        else    time[i] = INFINITY;
    }
    time[0] = 0;
    known[0] = 1;//Initialize
    Vertex V;

    for(;;)
    {
        for(i = 1; i <= G->Nv; i++)
            if(!known[i]) break;
        V = i;
        for(i = 1; i <= G->Nv; i++)
            if(!known[i] && time[i] < time[V])
                V = i;//Find the unknown vertex with the least time
        if(V == P) break;//Reach the problem station within the least time, end the loop

        known[V] = 1;
        for(i = 0; i <= G->Nv; i++)
        {
            if(G->Edge[V][i] > 0)
            {
                if(time[V] + G->Edge[V][i] < time[i])
                    time[i] = time[V] + G->Edge[V][i];//Update the time spent along the path
            }
        }
    }
    return time[V];
}
```

```c
void FindPath()
{
    int i, j, cnt;
    static int visit[MaxNum][MaxNum];//Whether Edge[i][j] is visited
    Stack[++top] = 0;//Push PBMC into the stack
    while(top != -1)
    {
        cnt = 0;
        Vertex a = Stack[top];
        for(i = 1; i <= G->Nv; i++)
        {
            for(j = 0; j <= top; j++)
                if(i == Stack[j])
                    break;
            if(G->Edge[a][i] > 0 && !visit[a][i] && j > top)//For each adjacent and unvisited edge
            {
                visit[a][i] = 1;
                CntTime += G->Edge[a][i];
                if(CntTime > MinTime)//Invalid vertex
                {
                    CntTime -= G->Edge[a][i];
                    cnt++;
                    continue;
                }
                else
                {
                    Stack[++top] = i;//Push into the stack
                    if(i == P && CntTime == MinTime)//Reach the problem station within the least time
                    {
                        int j;
                        for(j = 0; j <= top; j++)
                            Path[PathNum][j+1]= Stack[j];
                        Path[PathNum++][0] = j;//The number of vertex along the path
                        CntTime -= G->Edge[a][i];
                        top--;//Pop the top vertex in the stack and continue the loop
```

```
                        cnt++;
                        continue;
                    }
                    else break;
                }

            }
            else cnt++;
        }
        if(cnt == G->Nv)//Invalid top vertex
        {
            a = Stack[top];
            CntTime -= G->Edge[Stack[top-1]][a];
            for(i = 1; i <= G->Nv; i++)
                visit[a][i] = 0;
            top--;
        }
    }
}
```

```
void CntBike()
{
    int BikeRecord, BikeNow;
    int i, j;
    for(i = 0; i < PathNum; i++)//i = the index of the path select
    {
        BikeNow = 0;//The number of bikes taken from stations along the path
        BikeRecord = 0;//The number of bikes taken from the PBMC
        for(j = 2; j <= Path[i][0]; j++)// j = the index of the station along the path, start from the first station
        {
            if(BikeNum[Path[i][j]] >= cap/2)
                BikeNow += BikeNum[Path[i][j]] - cap/2;
            else if(BikeNow >= cap/2 - BikeNum[Path[i][j]])
                BikeNow -= cap/2 - BikeNum[Path[i][j]];
            else
            {
                BikeRecord += cap/2 - BikeNum[Path[i][j]] - BikeNow;
                BikeNow = 0;
            }
        }
        SentBike[i] = BikeRecord;
        TakeBack[i] = BikeNow;
    }
}
```

```
int SortBike()
{
    int i;
    int minsent = INFINITY, mintake = INFINITY;
    int index;
    for(i = 0; i < PathNum; i++)
    {
        if(SentBike[i] == minsent)//Compare the paths with the same number of sent bikes
        {
            if(TakeBack[i] < mintake)
            {
                mintake = TakeBack[i];
                index = i;
            }
        }
        else if(minsent > SentBike[i])
        {
            minsent = SentBike[i];
            mintake = TakeBack[i];
            index = i;
        }
    }
    return index;
}
```

Createdata.c

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
int INF = 1000000;

void CreatData();
int main()
{
    CreatData();
    return 0;
}

void CreatData()
{
    int vertexnum;    //The number of stations
    int maxnum;           //Maximum parking capacity
    int maxdistance;        //Maximum distance
    double saturation;   //The number of average edges at one vertex
    int problemstation;
    scanf("%d%d%d%d%lf", &vertexnum, &maxnum, &maxdistance,&problemstation, &saturation);
    int** edge;
    edge = (int**)malloc(sizeof(int*) * vertexnum);
    for (int i = 0; i < vertexnum; i++)
    {
        edge[i] = (int*)malloc(sizeof(int) * vertexnum);
        for (int j = 0; j < vertexnum; j++)
            edge[i][j] = INF;
    }
    int* currentnum;
    currentnum = (int*)malloc(sizeof(int) * vertexnum);

    srand(time(NULL));
    for (int i = 0; i < vertexnum; i++)   //Randomly generate the number of existing shared bikes
        currentnum[i] = rand() % (maxnum + 1);

    int n = 0;
    for (int i = 0; i < vertexnum-1; i++)   //Randomly generate the edge
    {
        int a = 0;
        for (int j = i + 1; j < vertexnum; j++)
        {
            if (((double)rand() / RAND_MAX) <= saturation) //The probability of saturation between two adjacent vertex
            {
                edge[i][j] = rand() % maxdistance+1;
                edge[j][i] = edge[i][j];
                n++;
                a = 1;
            }
        }
        if (a == 0)                     //Each vertex has at least one edge.
        {
            int j = rand() % (vertexnum - i -1) + i+1;
            if (edge[i][j] == INF)
            {
                edge[i][j] = rand() % maxdistance+1;
                edge[j][i] = edge[i][j];
                n++;
            }
        }
    }

    FILE* fp;                           //The generated data is exported to E://data.txt.
    fp = fopen("E://data.txt", "w");
    fprintf(fp, "%d %d %d %d\n", maxnum, vertexnum-1, problemstation, n);
    for (int i = 1; i < vertexnum; i++)  //0 is PBMC
        fprintf(fp, "%d ", currentnum[i]);
    for (int i = 0; i < vertexnum; i++)
    {
        for (int j = i+1; j < vertexnum; j++)
        {
            if (edge[i][j] != INF)
                fprintf(fp, "\n%d %d %d", i, j, edge[i][j]);
        }
    }
    fclose(fp);
}
```

Declaration

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

Programmer: *张雯琪，also complete ch1&2 and analysis in ch4*

Tester: *蒋鹏飞*