**Performance Measurement (POW)**

作者姓名

日期：2019-11-23

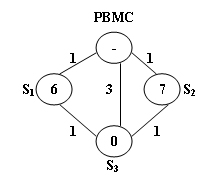
**Ch1** 问题描述及背景

**1.1** 问题描述

杭州市有一个公共自行车服务中心，为来自世界各地的游客提供极大的便利。你可以在任何一个车站租一辆自行车，然后把它送回城里的任何一个车站。

公共自行车管理中心（PBMC）持续监控所有车站的实时通行能力。据说，如果一个车站正好是半满的，那么它就处于完美的状态。如果一个车站是满的或空的，公共自行车管理中心（PBMC）将收集或发送自行车，以调整该车站的条件以完善，而且途中的所有车站也将进行调整。

当一个问题站点被报告时，PBMC总是选择到达该站点的最短路径。如果有多条最短路径，则选择需要从PBMC发送的最少自行车数的路径。



上图是一个例子。桩号由顶点表示，道路与边缘相对应。边上的数字是从一个终点站到另一个终点站所用的时间。在一个顶点S内写的数字是在S中存储的当前自行车的数目，因为每个站的最大容量是10。

有两条最短路径：

1. PBMC->S1->S3.
2. PBMC->S2->S3.

但是由于2选择所需发送的自行车数量小于1，则应该选择2.

**1.2** 算法介绍

所用到的一些函数：

MGraph ReadGraph(**int** N, **int** M);//build a graph

Vertex Findmin(MGraph Graph, **int** dist[], **int** Known[]);//Return the vertex in the graph which have not been visited and have the minimum dist

MGraph ShortestPath(MGraph Graph, Vertex S);//find out several shortest paths

**void** DepthFirstSearch(MGraph Graph, Vertex S);//find the path which needs the least bikes

**void** PrintPath();//output the relevant information

1. 基于Dijkstra算法得到最短路径
2. 基于深度搜索算法解决最终选择方案
3. 读取数据构建图的算法

**Ch2** 算法描述

1. 基于Dijkstra算法得到最短路径

描述：

我们使用Dijkstra算法找出几个最短路径，并将它们存储在结构数组中。我们定义了两个数组：dist和Known。第一个用于节省最小距离从顶点到源顶点，另一个用于保存顶点的访问。每次我们访问已知数组的最小dist值时，都会更新它的相邻点，并在路径结构中保存最短路径的前导节点数组，直到访问所有顶点。

伪代码：

1）初始化：设定除源节点以外的其它所有节点到源节点的距离为INFINITE(一个很大的数)，且这些节点都没被处理过。

2）从源节点出发，更新相邻节点(图中为2，3，6)到源节点的距离。然后在所有节点中选择一个最段距离的点作为当前节点。

3）标记当前节点为done(表示已经被处理过)，与步骤2类似，更新其相邻节点的距离。(这些相邻节点的距离更新也叫松弛，目的是让它们与源节点的距离最小。因为你是在当前最小距离的基础上进行更新的，由于当前节点到源节点的距离已经是最小的了，那么如果这些节点之前得到的距离比这个距离大的话，我们就更新它)。

4）步骤3做完以后，设置这个当前节点已被done，然后寻找下一个具有最小代价(cost)的点，作为新的当前节点，重复步骤3.

5)如果最后检测到目标节点时，其周围所有的节点都已被处理，那么目标节点与源节点的距离就是最小距离了。如果想看这个最小距离所经过的路径，可以回溯，前提是你在步骤3里面加入了当前节点的最优路径前驱节点信息。

二、基于深度搜索算法解决最终选择方案

描述：

我们从最短路径中找到最佳路径。我们递归地从路径结构数组中找出每个最短路径，

将路径存储在TempPath中，然后计算所需的自行车数量每一条最短的路径和剩余的自行车数量，以便在LastPath中存储。

伪代码：

int check(参数)

{

if(满足条件)

return 1;

return 0;

}

void dfs(int step)

{

判断边界

{

相应操作

}

尝试每一种可能

{

满足check条件

标记

继续下一步dfs(step+1)

恢复初始状态（回溯的时候要用到）

}

}

1. 读取数据构建图的算法

描述：

我们将站点信息存储在图形结构中。首先，输入N个非负数，表示一个站点当前的自行车数量。为了方便起见，我们从当前数字中减去半满状态的自行车数量，这样我们就可以通过判断正值和负值来判断车站是否需要自行车或仍然需要自行车。然后M条线，每一条线包含两个道路站和在它们之间移动所需的时间我们把它们都存储在一个图表中。

Ch3

测试结果

Dijkstra算法

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | 3 | 4 | 5 | 6 | 10 | 20 | 30 | 50 |
| 时间 | 0.012 | 0.021 | 0.023 | 0.033 | 0.067 | 0.247 | 0.537 | 1.533 |

根据测试结果我们发现dijkstra算法的运行时间跟跟n^2成正比，这与这个算法本身的时间复杂度为O(n^2)是相符合的，而且只要n确定那么程序运行的时间就不会有太大变化，这是因为即使图不一样，但是在遍历时还是要遍历n阶矩阵中的所有元素

深度优先搜索算法，在本题的这个算法中，由于在dijkstra算法中已经找到了从起始节点到最终节点的最短路径，需要的只是将这些路径一一的走一遍，判断哪个是最优解，所以时间复杂度是只和最短路径的条数，以及每条最短路径的长度有关的，和总的点数无关，为了验证我们的理论，下面是分别是算法运行时间与最短路径条数，最短路径长度的关系。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 路径长度 | 3 | 4 | 5 | 6 | 10 | 20 | 30 |
| 时间 | 0.834 | 1.108 | 1.179 | 1.404 | 1.954 | 3.308 | 4.588 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 最短路径条数 | 1 | 2 | 3 | 4 | 5 | 10 |
| 时间 | 0.617 | 1.291 | 1.914 | 2.503 | 3.132 | 6.141 |

根据上述结果我们发现运行时间确实与最短路径条数以及路径的长度有关，并且时间恰好都是其自变量的一次多项式的函数，也就是说这个算法的时间复杂度为O(k,p)，其中k为最短路径的条数，p为这些最短路径中最长的路径的长度。

而实际上由于在dijkstra已经将最短路径储存起来了，所以第二个算法（dfs）所需时间和n的关系并不大，而只和图的结构有关。

**ch4** 算法分析

1. 基于Dijkstra算法得到最短路径

在我们程序算法设计中，该算法由两个函数组成：

Vertex Findmin(MGraph Graph, int dist[], int Known[]);

MGraph ShortestPath(MGraph Graph, Vertex S);

具体内容见附录，下面分析该算法复杂度

1. **时间复杂度分析**

Vertex Findmin(MGraph Graph, int dist[], int Known[]);

函数中for循环次数与图结构的点有关，且为一次关系，其余操作都为常数时间消耗

**则**

**T(Findmin())=O(V)+O(1)=O(V)**

MGraph ShortestPath(MGraph Graph, Vertex S);

函数中开始由一个T=O(V)的循环

之后进入while循环：

每次循环调用一个Findmin(),T=O(V)

之后由一个与n成线性关系的循环：T=O(V)

其余操作均为常数时间消耗：T=O(1)

而整个循环需要访问所有图的点，即循环次数与V线性相关

**则该函数时间复杂度：**

**T(ShortestPath())=V(O(V)+O(V)+O(1))=O(V2)**

综上所述：该算法的时间复杂度为O(V2)

1. **空间复杂度分析**

Vertex Findmin(MGraph Graph, int dist[], int Known[]);

仅定义有限个可数的变量，即有

S=O(1)

MGraph ShortestPath(MGraph Graph, Vertex S);

1程序中由于测试数据量小，定义了一个固定数目的内存空间，即空间复杂度为O(1)

2而事实上，该算法需要遍历有所的边，在边的需求上需要创建S=O(E)的内存空间

综上，该算法的空间复杂度为

S=O(E)

1. **进一步改进的意见**

使用邻接矩阵实现的dijkstra算法的复杂度是O(V²)。使用邻接表的话，更新最短距离只需要访问每条边一次即可，因此这部分的复杂度是O(E).但是每次要枚举所有的顶点来查找下一个使用的顶点，因此最终复杂度还是O(V²)。在|E|比较小时，大部分的时间都花在了查找下一个使用的顶点上，因此需要使用合适的数据结构进行优化。

需要优化的是数值的插入(更新)和取出最小值两个操作，因此使用堆就可以了。把每个顶点当前的最短距离用堆来维护，在更新最短距离时，把对应的元素往根的方向移动以满足堆的性质。而每次从堆中取出的最小值就是下一次要用的顶点。这样堆中的元素共有O(V)个，更新和取出的操作有O(E)次，因此整个算法的复杂度是O(ElogV)。

二、基于深度搜索算法解决最终选择方案

**1.时间复杂度分析**

本程序中，dfs算法的时间复杂度取决于两个因素：最短路径的长度L和最短路径数N。

与图的点和边没有关系，即

T=O(LN)

1. **空间复杂读分析**

DFS算法是一一个递归算法，需要借助一个递归工作栈，其递归深度与最短路径数目N成线性关系，故它的空间复杂度为

T=O(N)

(注意N和V的区别！)

1. **进一步改进的意见**

我们组认为该算法应该是目前能想到的效率最高，思路最清晰的算法，因为在Dijkstra算法处理中提取出了最短路径的长度L和最短路径数N。则在算法效率上从基于边E和点V的数量级降为基于最短路径的长度L和最短路径数N的数量级，设计层次大大提高了空间的利用和运行效率。

三、读取数据构建图的算法

此算法并非解决问题的关键，仅仅做数据的输入处理和图结构的建立，因此在这不做具体算法分析

附录：

#include<stdio.h>

#include<stdlib.h>

#include<math.h>

#define INFINITY 1000000

#define MaxStationNum 501

#define Vertex int

#define RoadWeight int

#define ERROR -1

//define the storage structure of the graph

**typedef** **struct** GNode \*PtrtoGNode;

**struct** GNode {

**int** Nv;//the number of vertices of the graph

**int** Ne;//the number of edges of the graph

RoadWeight G[MaxStationNum][MaxStationNum];//the length of edges of the graph

Vertex NumberWeight[MaxStationNum];//the weight of vertices of the graph

};

**typedef** PtrtoGNode MGraph;

//define the storage structure of the edge

**typedef** **struct** ENode \*PtrtoENode;

**struct** ENode {

Vertex V1, V2;//two vertices of the edge

RoadWeight Weight;//the length of the edge

};

**typedef** PtrtoENode Road;

//define the storage structure of the shortest paths between the PBMC and the problem station

**struct** PNode {

Vertex pre[MaxStationNum];//the precursor vertex of the vertex

Vertex top;//the number of the shortest paths

};

//As these variables are invoked in multiple functions, we define them as global variables.

Vertex Cmax;

**int** MinNeed = INFINITY, MinRemain = INFINITY;

**struct** PNode Path[MaxStationNum], TempPath, LastPath;

//function definition

MGraph ReadGraph(**int** N, **int** M);//build a graph

Vertex Findmin(MGraph Graph, **int** dist[], **int** Known[]);//Return the vertex in the graph which have not been visited and have the minimum dist

MGraph ShortestPath(MGraph Graph, Vertex S);//find out several shortest paths

**void** DepthFirstSearch(MGraph Graph, Vertex S);//find the path which needs the least bikes

**void** PrintPath();//output the relevant information

/\*

First of all, we store the station information in the graph structure,

then use the Dijkstra's Algorithm to find out several shortest paths and store them in the structure array,

after that, we use the depth first search algorithm to find the path which needs the least bikes,

and finally output the relevant information

\*/

**int** main()

{

**int** N, M;

Vertex Sp;

//Input the maximum capacity of each station, the total number of stations,

//the index of the problem station, the number of roads.

scanf("%d%d%d%d", &Cmax, &N, &Sp, &M);

MGraph Graph;

Graph = ReadGraph(N, M);

Graph = ShortestPath(Graph, 0);

DepthFirstSearch(Graph, Sp);

PrintPath();

}

/\*

 In this function, we store the station information in the graph structure.

 Firstly,input N non-negative numbers,which stand for the current number of bikes at a station.

 For convenience, we subtract the number of bikes in half full state from the current number,

 so that we can judge whether the station needs or remains bikes by judging the positive and negative values.

 Then M lines follow, each contains two stations of a road and the time taken to move between them,​

 and we store all of them in a graph.

 \*/

MGraph ReadGraph(**int** N, **int** M)

{

MGraph Graph;

Road E;

Vertex V, W;

**int** i;

//Initialization

Graph = (MGraph)malloc(**sizeof**(**struct** GNode));

Graph->Nv = N + 1;

Graph->Ne = 0;

TempPath.top = 0;

**for** (V = 0; V <= N; V++) {

**for** (W = 0; W <= N; W++) {

Graph->G[V][W] = INFINITY;

}

}

//Input the current number of bikes at a station

**for** (V = 1; V <= N; V++) {

scanf("%d", &Graph->NumberWeight[V]);

Graph->NumberWeight[V] -= Cmax / 2;

Path[V].top = 0;

}

Graph->NumberWeight[0] = 0;//As the PBMC neither needs nor remains bikes, we assume it's zero.

//Input the road information

Graph->Ne = M;

**if** (M != 0)

{

E = (Road)malloc(**sizeof**(**struct** ENode));

**for** (i = 0; i < M; i++) {

scanf("%d%d%d", &E->V1, &E->V2, &E->Weight);

Graph->G[E->V1][E->V2] = E->Weight;

Graph->G[E->V2][E->V1] = E->Weight;

}

}

**return** Graph;

}

/\*

 In this function, we use the Dijkstra's Algorithm to find out several shortest paths,

 and store them in the structure array.

 We define two arrays: dist and Known. The first one is used to save the minimum distance

 from the vertex to the source vertex, and the other is used to save the visit of the vertex.

 Each time we visit the minimum dist value from the array Known, we update the dist value of

 its adjacent points, and save the precursor node of the shortest path in the Path structure

 array until all the vertices are visited.

 \*/

MGraph ShortestPath(MGraph Graph, Vertex S)

{

//Initializing the arrays

**int** N;

N = Graph->Nv;

**int** dist[MaxStationNum], Known[MaxStationNum];

Vertex V, W;

**for** (V = 0; V < N; V++) {

dist[V] = Graph->G[S][V];

Known[V] = 0;

}

dist[S] = 0;

//Visit vertices in order, update the shortest distance and save it in the array structure

**while** (1) {

V = Findmin(Graph, dist, Known);

**if** (V == ERROR) **break**;

Known[V] = 1;

**for** (W = 0; W < N; W++) {

**if** (Known[W] == 0 && Graph->G[V][W] < INFINITY)

{

**if** (dist[V] + Graph->G[V][W] < dist[W])

{

dist[W] = dist[V] + Graph->G[V][W];

Path[W].top = 0;

Path[W].pre[Path[W].top] = V;

Path[W].top++;

}

**else** **if** (dist[V] + Graph->G[V][W] == dist[W])

{

Path[W].pre[Path[W].top] = V;

Path[W].top++;

}

}

}

}

**return** Graph;

}

/\*

 In this function, we find the lowest dist among the vertices we've never visited.

 If all of the vertices are visited, return ERROR.

\*/

Vertex Findmin(MGraph Graph, **int** dist[], **int** Known[])

{

Vertex V, MinV;

**int** Mindist = INFINITY;

**for** (V = 0; V < Graph->Nv; V++) {

**if** (Known[V] == 0 && dist[V] < Mindist)

{

Mindist = dist[V];

MinV = V;

}

}

**if** (Mindist == INFINITY) **return** ERROR;

**return** MinV;

}

/\*

 In this function, we find the best path from the shortest paths.

 We recursively find out each shortest path from the Path structure array,

 store the path in TempPath, and then calculate the number of bikes required for

 each shortest path and the number of bikes remained,

 so as to save the better path in LastpPath.

 \*/

**void** DepthFirstSearch(MGraph Graph, Vertex S)

{

**int** i;

**if** (S == 0) //recursive boundary

{

TempPath.pre[TempPath.top] = S;//store the vertices in the path to TempPath

TempPath.top++;

**int** need = 0, remain = 0;

//calculate the number of bikes required or remained in the path from the source station to the problem station

**for** (i = TempPath.top - 1; i >= 0; i--)

{

Vertex id = TempPath.pre[i];

**int** w = Graph->NumberWeight[id];

**if** (w > 0) //we need to take some bikes

{

remain += w;

}

**else**

{

**if** (remain > abs(w)) //current bikes remain is sufficient

{

remain -= abs(w);

}

**else** //current bikes remain is not sufficient

{

need += abs(w) - remain; //we need to carry somes bikes from PBMC

remain = 0;

}

}

}

//Optimize the path

**if** (need < MinNeed)

{

MinNeed = need;

MinRemain = remain;

LastPath = TempPath;

}

**else** **if** (need == MinNeed && remain < MinRemain)

{

MinRemain = remain;

LastPath = TempPath;

}

TempPath.top--;

**return**;

}

TempPath.pre[TempPath.top] = S; //recursively store the vertices in the path to TempPath

TempPath.top++;

**for** (i = 0; i < Path[S].top; i++) //loop through each path

DepthFirstSearch(Graph, Path[S].pre[i]);

TempPath.top--;

}

/\*

 In this function, we output the relevant information from the best path.

 \*/

**void** PrintPath()

{

**int** i;

printf("%d ", MinNeed);

**for** (i = LastPath.top - 1; i >= 0; i--) {

printf("%d", LastPath.pre[i]);

**if** (i > 0) printf("->");

}

printf(" %d", MinRemain);

**return**;

}

**Declaration**

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