数据结构课程设计



班级： 1618403

学号： 161840227

姓名： 韦 鑫

指导教师：孙 涵

目录

1.采用的数据结构 ………………………………………… 3

2.算法设计思想 …………………………………………… 3

3.关键代码 ………………………………………………… 4

4.测试数据和结果 ………………………………………… 16

5.算法的时间复杂度及其改进方法 ……………………… 19

6.结束语 …………………………………………………… 19

一、采用的数据结构

采用邻接表存图

struct station

{

string name; //车站名

int line; //车站所在路

int index; //车站编号

int pass; //主要用于BFS记录经过的站点数

int transfer; //记录转乘次数

};

//用于Dijkstra算法堆优化中优先级队列元素的结构

struct node

{

string name;

int index;

int dis; //边权代表换乘次数

int pass;

//重载运算符<,用于优先级队列，换乘次数小优先级大

friend bool operator < (node a, node b)

{

if (a.dis == b.dis) return a.pass > b.pass;

return a.dis > b.dis;

}

};

二、算法设计思想

①存图：对文件一行一行的扫描，一行代表一条线，先让一条线上的相邻车站连通；最后通过使用multimap对同名车站但不同线路的车站进行连通处理。

②求两车站经过最少的车站路线：使用BFS广度优先搜索算法对其进行求解，且需保证在经过站数相同的情况下，换乘次数尽量少。

③求两车站最少换乘次数的路线：以需不需要换乘（0/1）作为边权，使用Dijkstra算法进行求解，且在换乘次数相同的情况下保证经过的站数尽量少。

三、关键代码

//运行环境：Visual Studio 2017

#include<map>

#include<queue>

#include<vector>

#include<string>

#include<cctype>

#include<cstdio>

#include<cstring>

#include<iostream>

#include<algorithm>

#pragma warning(disable:4996)

using namespace std;

const int maxn = 50000;

const int inf = 1 << 30 - 1;

struct station

{

string name; //车站名

int line; //车站所在路

int index; //车站编号

int pass; //主要用于BFS记录经过的站点数

int transfer; //记录转乘次数

};

//用于Dijkstra算法堆优化中优先级队列元素的结构

struct node

{

string name;

int index;

int dis; //边权代表换乘次数

int pass;

//重载运算符<,用于优先级队列，换乘次数小优先级大

friend bool operator < (node a, node b)

{

if (a.dis == b.dis) return a.pass > b.pass;

return a.dis > b.dis;

}

};

bool vis[maxn];

int d[maxn]; //d[i]表示起始车站到编号为i的车站的转乘次数

vector<station>g[maxn]; //g[i]存储着编号为i的车站能到达的车站

multimap<string, int>match; //车站名与编号的对应关系

typedef multimap<string, int>::iterator iter;

int cnt = 0; //总车站数

string sname[maxn]; //sname[i]表示编号为i的车站名

int num[maxn]; //num[i]表示编号为i的车站所在路线

int spass[maxn]; //spass[i]表示起点到编号为i的车站需要经过车站数

/\*------------------------------------------------------------------------------------------------\*/

//读取文件信息

void read\_file()

{

FILE \*fp;

if ((fp = fopen("Nanjing\_bus\_routes.txt", "r")) == NULL)

{

printf("打开文件失败！\n");

system("pause");

exit(0);

}

char str[1024], line[20], blank[4];

fscanf(fp, "%s", line);

fgets(blank, 4, fp);

fgets(str, 1024, fp);

while (!feof(fp))

{

//cout << line << str << endl;

int linenum = 0, t = 0;//计算路线数

while (line[t] >= '0' && line[t] <= '9')

{

linenum = linenum \* 10 + line[t] - '0';

t++;

}

station temp1, temp2;

int index1, index2;

int step = 0;

char \*p = strtok(str, ",");//提取车站名

while (p != NULL)

{

string name;

//为了防止每一路最后一个车站名带'\n'造成不必要的麻烦

for (; \*p != '\0' && \*p != '\n'; p++)

{

name += \*p;

}

//让同一条线路上的车站相连

if (step == 0)

{

temp1.name = name;

temp1.line = linenum;

temp1.index = ++cnt;

temp1.pass = temp1.transfer = 0;

pair<string, int> p1(name, cnt);

match.insert(p1);

sname[cnt] = name;

num[cnt] = linenum;

index1 = cnt;

p = strtok(NULL, ",");//获取下一个车站名

name = "";

for (; \*p != '\0' && \*p != '\n'; p++)

{

name += \*p;

}

temp2.name = name;

temp2.line = linenum;

temp2.pass = temp2.transfer = 0;

temp2.index = ++cnt;

pair<string, int> p2(name, cnt);

match.insert(p2);

sname[cnt] = name;

num[cnt] = linenum;

index2 = cnt;

g[index1].push\_back(temp2);

g[index2].push\_back(temp1);

step++;

p = strtok(NULL, ",");

}

else

{

temp1 = temp2;

index1 = index2;

temp2.name = name;

temp2.line = linenum;

temp2.index = ++cnt;

temp2.pass = temp2.transfer = 0;

pair<string, int> p2(name, cnt);

match.insert(p2);

sname[cnt] = name;

num[cnt] = linenum;

index2 = cnt;

g[index1].push\_back(temp2);

g[index2].push\_back(temp1);

p = strtok(NULL, ",");

}

}

fscanf(fp, "%s", line);

fgets(blank, 4, fp);

fgets(str, 1024, fp);

}

//让名字相同(但他们所处线路不同)的车站两两相连

for (iter it = match.begin(); it != match.end(); it++)

{

//传入车站名调用multimap的equal\_range来获取车站名与编号键值对

pair<iter, iter> pr = match.equal\_range(it->first);

//index用来存储同个车站名对应的所有编号

vector<int>index;

for (iter ii = pr.first; ii != pr.second; ii++) index.push\_back(ii->second);

//说明有同个车站分布在不同线路上

//按照这样的方式让分布在不同线路的同一车站连通会造成重复，所以应在建图时去重，降低后边寻路遍历的复杂度

if (index.size() > 1)

{

int flag = 0;

for (int i = 0; i < index.size() - 1; i++)

{

/\*------------以下是为了避免重复而进行的操作------------\*/

if (flag == 0)

{

int j;

for (j = 0; j < g[index[i]].size(); j++)

{

//表示编号为index[i]的可变长数组中存在自身的名字，说明不需要重复执行该操作

if (g[index[i]][j].name == it->first)

{

flag = 1;

break;

}

}

if (j == g[index[i]].size())flag = -1;//能完整执行一边循环说明无重复,则不需要重复执行以上操作

}

if (flag == 1)break;

/\*------------以上是为了避免重复而进行的操作------------\*/

for (int j = i + 1; j < index.size(); j++)

{

station temp1, temp2;

temp1.name = temp2.name = it->first;

temp1.line = num[index[i]]; temp2.line = num[index[j]];

temp1.index = index[i]; temp2.index = index[j];

temp1.pass = temp2.pass = temp1.transfer = temp2.transfer = 0;

g[index[i]].push\_back(temp2);

g[index[j]].push\_back(temp1);

}

}

}

}

//输出测试

/\*for (int i = 1; i <= cnt; i++)

{

cout << i << sname[i] << " ";

for (int j = 0; j < g[i].size(); j++)

{

cout << g[i][j].line << ":" << g[i][j].name << " index:" << g[i][j].index << " ";

}

cout << endl;

}\*/

fclose(fp);

}

/\*------------------------------------------------------------------------------------------------\*/

//输出乘车路线

void print\_path(int t, int pre[])

{

if (pre[t] == -1)

{

cout << sname[t] << endl;

return;

}

else

{

print\_path(pre[t], pre);

if (num[pre[t]] != num[t])

{

printf

(

"↓\n"

"↓( 换乘%d路 )\n"

"↓\n"

, num[t]);

cout << sname[t] << endl;

}

else cout << "↓\n↓\n" << sname[t] << endl;

}

}

/\*------------------------------------------------------------------------------------------------\*/

//广度优先搜索——用于找到两车站经过最少车站的路线

//返回值为真\假，表示是否存在这两车站，不存在用户输入的车站需重新输入

bool BFS(string s, string t)

{

if (match.count(s) == 0 && match.count(t) == 0)

{

printf("起始车站与目的车站均不存在,请重新输入！\n");

return false;

}

else if (match.count(s) == 0)

{

printf("起始车站不存在,请重新输入！\n");

return false;

}

else if (match.count(t) == 0)

{

printf("目的车站不存在,请重新输入！\n");

return false;

}

else

{

int pre[maxn]; //记录该车站的前驱，通过递归可得路线

int pass = inf; //经过的站数

int transfer = inf; //转乘的次数

int fstart, fend; //最终选择的起点和终点

pair<iter, iter> sp = match.equal\_range(s);

for (iter sii = sp.first; sii != sp.second; sii++)

{

int start = sii->second; //起始点

pair<iter, iter> tp = match.equal\_range(t);

for (iter tii = tp.first; tii != tp.second; tii++)

{

int end = tii->second; //终点

queue<station>Q; //广搜必不可少的队列

int temppre[maxn]; //记录该车站的前驱（临时），通过比较得到最优解

for (int i = 1; i <= cnt; i++)//初始化

{

vis[i] = 0;

temppre[i] = -1;

}

station temp;

temp.name = s;

temp.index = start;

temp.line = num[start];

temp.pass = 0; //经过站点数

temp.transfer = 0; //换乘次数

Q.push(temp);

vis[start] = 1;

while (!Q.empty())

{

station fnt = Q.front();

Q.pop();

if (fnt.index == end)//若为终点则结束搜索

{

//当经过的站数更少，或者经过站数不变但换乘次数更少时更新pre数组、经过的站数、转乘次数和最终的起点和终点

if (fnt.pass < pass || (fnt.pass == pass && fnt.transfer < transfer))

{

for (int i = 1; i <= cnt; i++)

{

pre[i] = temppre[i];

}

pass = fnt.pass;

transfer = fnt.transfer;

fstart = start;

fend = end;

}

break;

}

else

{

int u = fnt.index;

for (int j = 0; j < g[u].size(); j++)

{

int v = g[u][j].index;

if (!vis[v])

{

temp.name = sname[v];

temp.line = num[v];

temp.index = v;

//若前后两站的车站名相同，则不算经过另一个车站

//若前后两站的车站名相同，线路不同，说明换乘

if (sname[u] == sname[v])

{

temp.pass = fnt.pass;

if (num[u] != num[v])temp.transfer = fnt.transfer + 1;

else temp.transfer = fnt.transfer;

//pre[v] = pre[u];

}

else

{

temp.pass = fnt.pass + 1;

if (num[u] != num[v])temp.transfer = fnt.transfer + 1;

else temp.transfer = fnt.transfer;

//pre[v] = u;

}

temppre[v] = u; //记录编号为v的车站的前驱为u

Q.push(temp);

vis[v] = 1;

}

}

}

}

}

}

cout << "起点：" << num[fstart] << "路" << sname[fstart] << "\n↓\n终点：" << num[fend] << "路" << sname[fend] << endl << endl;

printf

(

"最少需要经过%d个站点,转乘%d次\n"

"路线为：\n"

"( %d路上车 )\n"

, pass, transfer, num[fstart]);

print\_path(fend, pre);

printf("( %d路下车 )\n", num[fend]);

}

}

/\*------------------------------------------------------------------------------------------------\*/

//迪杰斯特拉算法——用于找到两车站转乘次数最少的车站路线

//调用STL优先级队列对其进行堆优化.能将O(n\*n)的复杂度下降到O(nlogn)

bool Dijkstra(string s, string t)

{

if (match.count(s) == 0 && match.count(t) == 0)

{

printf("起始车站与目的车站均不存在,请重新输入！\n");

return false;

}

else if (match.count(s) == 0)

{

printf("起始车站不存在,请重新输入！\n");

return false;

}

else if (match.count(t) == 0)

{

printf("目的车站不存在,请重新输入！\n");

return false;

}

else

{

int pre[maxn]; //记录该车站的前驱，通过递归可得路线

int temppre[maxn]; //临时存放前驱

int fpass = inf; //经过的站数

int ftransfer = inf; //转乘的次数

int fstart = -1, fend = -1; //最终选择的起点和终点

pair<iter, iter> sp = match.equal\_range(s);

for (iter sii = sp.first; sii != sp.second; sii++)

{

int start = sii->second; //起始点

pair<iter, iter> tp = match.equal\_range(t);

for (iter tii = tp.first; tii != tp.second; tii++)

{

int end = tii->second; //终点

//cout << start << sname[start] << " " << end << sname[end] << endl;

for (int i = 1; i <= cnt; i++)//初始化

{

d[i] = inf;

spass[i] = inf;

vis[i] = 0;

temppre[i] = -1;

}

d[start] = 0;

spass[start] = 0;

priority\_queue<node>Q;

node temp;

temp.name = s;

temp.index = start;

temp.dis = 0;

temp.pass = 0;

Q.push(temp);

while (!Q.empty())

{

node fnt = Q.top();

Q.pop();

int u = fnt.index, dis = fnt.dis, pass = fnt.pass;

if (u == end)break;

if (vis[u])continue;

vis[u] = 1;

for (int i = 0; i < g[u].size(); i++)

{

int v = g[u][i].index;

if (!vis[v])

{

int tempdis, temppass;

if (num[u] == num[v]) //若线路相同

{

tempdis = dis; //说明不需要换乘

temppass = pass + 1;//经过车站数加一

}

else//若线路不同

{

tempdis = dis + 1; //换乘

temppass = pass; //在同个车站进行换乘，所经过的算一个车站

}

if ((tempdis < d[v]) || (tempdis == d[v] && temppass < spass[v]))//到达编号为v的车站转乘数可优化

{

d[v] = tempdis;

spass[v] = temppass;

temp.name = g[u][i].name;

temp.index = g[u][i].index;

temp.dis = d[v];

temp.pass = spass[v];

temppre[v] = u; //temppre[v]表示编号为v的车站的前驱车站编号为u

Q.push(temp);

}

}

}

}

//若换乘次数更少、或者在换乘次数相同的情况下经过车站数更少时更新路径

if ((d[end] < ftransfer) || (d[end] == ftransfer && spass[end] < fpass))

{

for (int i = 1; i <= cnt; i++)

{

pre[i] = temppre[i];

}

fstart = start;

fend = end;

ftransfer = d[end];

fpass = spass[end];

}

}

}

cout << "起点：" << num[fstart] << "路" << sname[fstart] << "\n↓\n终点：" << num[fend] << "路" << sname[fend] << endl << endl;

printf

(

"最少需要换乘%d次,经过了%d个车站\n"

"路线为:\n"

"( %d路上车 )\n"

, ftransfer, fpass, num[fstart]);

print\_path(fend, temppre);

printf("( %d路下车 )\n", num[fend]);

}

}

int main()

{

read\_file();

string s, t;

while (1)

{

system("cls");

printf("请输入起始车站：");

cin >> s;

printf("请输入目的车站：");

cin >> t;

printf("---------------------------\n");

if (!BFS(s, t))

{

system("pause");

continue;

}

printf("\n");

printf("---------------------------\n");

if (!Dijkstra(s, t))

{

system("pause");

continue;

}

char ch;

printf("是否继续查询？（y/n）:");

getchar();

scanf("%c", &ch);

if (toupper(ch) == 'N')break;

}

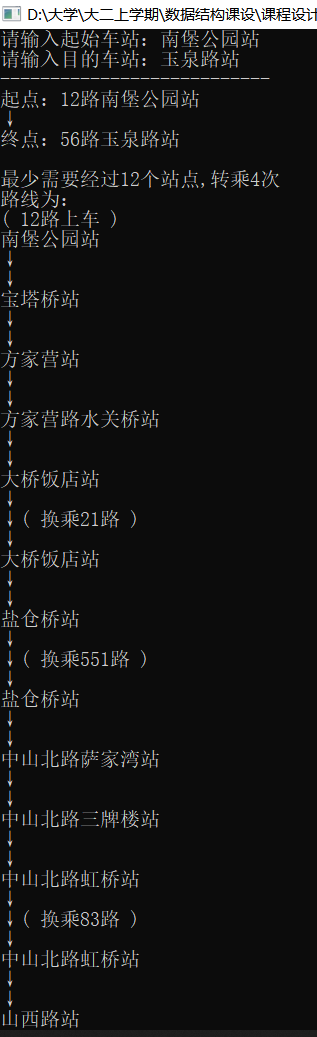
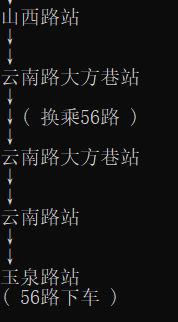
system("pause");

return 0;

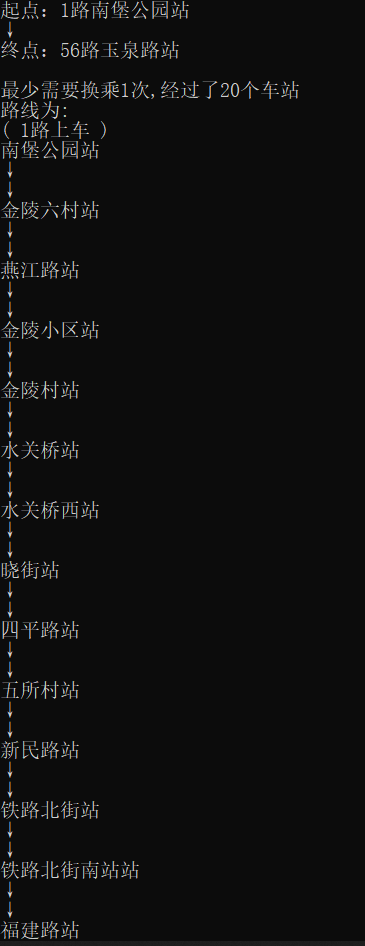
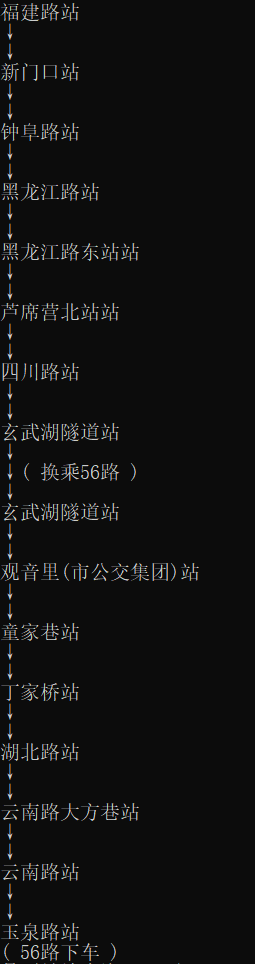
}

四、测试数据和结果

经过最少站数：

换乘次数最少：

五、算法的时间复杂度即改进方法

经过站数最少，BFS算法：O(n)

换乘次数最少，Dijkstra算法：O(n\*logn)

Dijkstra算法以用优先级队列进行堆优化

六、结束语

代码共535行

实际上，我所写的BFS、Dijkstra算法的时间复杂度要比上面给的高出两个数量级，这是因为题目并未给出两个车站的线路数，而是只给了车站名，这使得我需要用两个for对起点、终点进行组合，以此得到题目所需。我认为题目应该做出修改，加入输入起点、终点线路数的要求，以便更加快速的得出结果。