### 第4题：公交路线提示

#### 数据结构：

采用了邻接表存储公交路线图的数据，邻接表的相关类或结构体如下：

class ArcNode //邻接表结点类

{

public:

int id; //边的下一个id

ArcNode \*next;

ArcNode(int id) : id(id), next(nullptr) {}

};

class VertexHead //邻接表表头类

{

public:

int id; //表头id

ArcNode \*first;

VertexHead() : first(nullptr) {}

};

//存储公交路线图中的路线类

class Line

{

public:

int no; //路线编号

};

//存储公交路线图中的站点类

class Station

{

public:

int id; //站点id

string name; //站点名称

set<int> buss; //经过站点的公交车的路线，set便于去重以及集合运算

Station() {}

Station(string name, int busNo) : name(name) { buss.insert(busNo); }

};

//将路线+站点封装为一个类

class LineBus

{

public:

Line line; //路线

vector<int> stations; //每一条路线的所有站点

};

//邻接表类记录相关数据

class Graph

{

public:

VertexHead heads[MAX]; //所有的表头

vector<LineBus> lines; //文件中的所有路线

vector<Station> stations; //存储所有的站点

vector<int> lines\_no; //所有路线编号

int line\_num; //路线数

int station\_num; //站点数

Graph()

{

for (int i = 0; i < MAX; ++i)

{

heads[i].id = i;

}

}

void insertArc(int &s1, int &s2)

{

ArcNode \*p = heads[s1].first;

while (p != nullptr)

{

if (p->id == s2)

{

return;

}

p = p->next;

}

ArcNode \*arc = new ArcNode(s2);

arc->next = heads[s1].first;

heads[s1].first = arc;

ArcNode \*arc2 = new ArcNode(s1);

arc2->next = heads[s2].first;

heads[s2].first = arc2;

}

};

#### 算法设计思想：

##### 1.核心算法

获取经过站点最少路线时，使用迪杰斯特拉算法；获取最少换乘路径时，使用BFS算法。

##### 2.程序其他部分说明以及为何要大量使用STL库中的容器

将核心的显示功能（menu）封装为函数和根据选择封装为一个函数，进行数据处理（enquiry）封装为一个函数。将这两个函数需要的初始化和处理细节分别封装为相关函数。

本程序的核心在于如何承载数据，以及如何方便地使用相关数据。

由于需要进行类似于站点与路线的一对多或多对一映射，以及需要进行类似于集合的运算，还需要快速查找并获取到相关数据。同时，这些数据之间的对应个数并不确定。因而本程序需要大量使用STL库中的相关容器。

##### 3.相关容器

将站点转换为数字可以便捷地进行相关操作，而要在之后的操作中快速查找到相关数字所对应的站点或者相关站点所对应的数字。因此使用两个map容器进行相互映射。

BFS算法需要使用队列，故使用queue容器，输出时则需要使用stack容器

为了快速找到两个站点间有哪些路线可以相通，最好是将将每个站点所“拥有”的路线作为一个集合，两个集合进行交集运算即可

要转车的点必要有多个，同时每个站点又大概率又多个路线，且个数不是固定的。故转车的点的路线用vector存储，然后将点和路线有用pair存储（pair<int, vector<int>>），多个点又用vector存储。

将转车的点（pair<int, vector<int>>）以及我们要前往的点所映射的数字又可以合并为一个pair，再用vector存储，可以获得存储换乘路线的一个个结点，连起来便于输出路线。

##### 4.函数说明

Menu函数，输出菜单

Enquiry函数，根据指令调用相关函数

Check函数，输入起点和终点时在map容器里查找是否有对应的站点，方便提示用户

readFileRoutine函数，从文件中读取公交路线，获取每条路线号并存放进相关vector容器里，读取该路线的所有站点存放进邻接表里，如果读取到的站点之前就出现过，那么它会是某一条路线的始发站，在对应站点的表头插入下一站点。

Dijkstra函数，找到最少站点路径，使用Dijkstra算法。

findLeastTransfer函数，找到最少换乘路线，使用BFS算法，进行拓展时用到intersection函数

intersection函数，获取两个站点之间路线的交集

show函数，输出路线，根据参数传进来的栈，进行栈pop操作然后将找到的相映射的所要输出信息存进buss，再将buss处理输出最终路线即可

#### 源程序：

##### Buses.h

#ifndef BUSES\_H

#define BUSES\_H

#include <bits/stdc++.h>

using namespace std;

const int MAX = 12000;

const int line\_max = 2000; //最多2000条公交路线

class ArcNode

{

public:

int id; //边的下一个id

ArcNode \*next;

ArcNode(int id) : id(id), next(nullptr) {}

};

class VertexHead

{

public:

int id; //表头id

ArcNode \*first;

VertexHead() : first(nullptr) {}

};

//路线类

class Line

{

public:

int no; //路线编号

};

//站点类

class Station

{

public:

int id; //站点id

string name; //站点名称

set<int> buss; //经过站点的公交车的路线

Station() {}

Station(string name, int busNo) : name(name) { buss.insert(busNo); }

};

//路线+站点

class LineBus

{

public:

Line line; //路线

vector<int> stations; //每一条路线的所有站点

};

class Graph

{

public:

VertexHead heads[MAX]; //所有的表头

vector<LineBus> lines; //文件中的所有路线

vector<Station> stations; //存储所有的站点

vector<int> lines\_no; //所有路线编号

int line\_num; //路线数

int station\_num; //站点数

Graph()

{

for (int i = 0; i < MAX; ++i)

{

heads[i].id = i;

}

}

void insertArc(int &s1, int &s2)

{

ArcNode \*p = heads[s1].first;

while (p != nullptr)

{

if (p->id == s2)

{

return;

}

p = p->next;

}

ArcNode \*arc = new ArcNode(s2);

arc->next = heads[s1].first;

heads[s1].first = arc;

ArcNode \*arc2 = new ArcNode(s1);

arc2->next = heads[s2].first;

heads[s2].first = arc2;

}

};

#endif

##### Main.cpp

#include "buses.h"

#include <bits/stdc++.h>

using namespace std;

void menu();

map<string, int> maps;

map<int, string> remaps;

vector<pair<int, vector<int>>> path; //存放转车点

vector<pair<pair<int, vector<int>>, int>> path2; //存放转车点

Graph \*graph = new Graph();

/\* 从文件中读取公交线路 \*/

void readFileRoutine()

{

ifstream in("routine.txt");

if (in.fail())

{

cout << "公交线路打开失败！" << endl;

exit(0);

}

string bus;

string stations;

string station;

stringstream ss;

int count = 0;

while (in.peek() != EOF)

{

LineBus cur\_line;

//获取每一条线路号

getline(in, bus, ' ');

int busNo = atoi(bus.substr(0, bus.size() - 1).c\_str());

cur\_line.line.no = busNo;

graph->lines\_no.push\_back(busNo);

//获取该路线所有的站点

getline(in, stations, '\n');

while (stations[0] == ' ')

{

stations = stations.substr(1, stations.size() - 1);

}

ss.str(stations);

while (getline(ss, station, ','))

{

// 1. 判断station是否已经出现在了图中

bool flag = false;

int index = -1;

for (int i = 0; i < graph->stations.size(); ++i)

{

if (graph->stations[i].name == station)

{

flag = true;

index = i;

break;

}

}

if (flag)

{

//如果当前站点之前就出现过

graph->stations[index].buss.insert(busNo);

cur\_line.stations.push\_back(graph->stations[index].id);

if (cur\_line.stations.size() != 1)

{

//说明是始发站

graph->insertArc(cur\_line.stations[cur\_line.stations.size() - 1],

cur\_line.stations[cur\_line.stations.size() - 2]);

}

}

else

{

Station s;

count++;

s.id = count;

s.name = station;

s.buss.insert(busNo);

graph->stations.push\_back(s);

maps.insert(make\_pair(station, count));

remaps.insert(make\_pair(count, station));

cur\_line.stations.push\_back(count);

if (cur\_line.stations.size() != 1)

{

//说明是始发站

graph->insertArc(cur\_line.stations[cur\_line.stations.size() - 1],

cur\_line.stations[cur\_line.stations.size() - 2]);

}

}

}

graph->lines.push\_back(cur\_line);

bus = "";

ss.clear();

ss.str("");

stations = "";

station = "";

}

in.close();

graph->line\_num = graph->lines\_no.size();

graph->station\_num = graph->stations.size();

}

/\* 获取两个站点之间路线的交集 \*/

bool intersection(int start\_id, int end\_id)

{

vector<int> inter(line\_max);

int s\_index = -1;

int e\_index = -1;

for (size\_t i = 0; i < graph->stations.size(); i++)

{

if (s\_index != -1 && e\_index != -1)

{

break;

}

else

{

if (graph->stations[i].id == start\_id)

{

s\_index = i;

}

else if (graph->stations[i].id == end\_id)

{

e\_index = i;

}

}

}

auto it =

set\_intersection(graph->stations[s\_index].buss.begin(),

graph->stations[s\_index].buss.end(),

graph->stations[e\_index].buss.begin(),

graph->stations[e\_index].buss.end(), inter.begin());

inter.resize(it - inter.begin());

if (inter.empty())

{

return false;

}

else

{

path.push\_back(make\_pair(start\_id, inter));

path2.push\_back(make\_pair(make\_pair(start\_id, inter), end\_id));

return true;

}

}

int min(int dist[], bool flag[], int size)

{

int min = MAX;

int min\_index = -1;

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

{

if (!flag[i] && dist[i] < min)

{

min = dist[i];

min\_index = i;

}

}

return min\_index;

}

void show(stack<int> line, string start, string end)

{

vector<vector<int>> buss;

int \*stations = new int[line.size()];

int count = 0;

while (!line.empty())

{

int top = line.top();

line.pop();

stations[count] = top;

if (line.empty())

{

continue;

}

int next = line.top();

vector<int> tmp;

for (set<int>::iterator it1 = graph->stations[top].buss.begin();

it1 != graph->stations[top].buss.end(); ++it1)

{

for (set<int>::iterator it2 = graph->stations[next].buss.begin();

it2 != graph->stations[next].buss.end(); ++it2)

{

if (\*it1 == \*it2)

{

tmp.push\_back(\*it1);

}

}

}

buss.push\_back(tmp);

++count;

}

/\* 输出最终路线 \*/

int cur\_line = buss[0][0];

cout << "从 " << start << " 到 " << end << " 的路线如下： \n";

int pre\_line = cur\_line;

for (int i = 0; i < count; ++i)

{

bool flag = false;

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

{

if (buss[i][j] == cur\_line)

{

flag = true;

break;

}

}

if (!flag)

{

if (buss[i].size() > 0)

{

cur\_line = buss[i][0];

cout << "\t乘坐： " << pre\_line << " 号公交到 "

<< graph->stations[i].name << " \n";

pre\_line = cur\_line;

}

}

}

cout << "\t乘坐： " << pre\_line << " 号公交到 " << end << " \n";

}

void Dijkstra(string start, string end)

{

int start\_id = maps[start];

int end\_id = maps[end];

int size = graph->stations.size();

int \*dist = new int[size + 1];

int \*p = new int[size + 1];

bool \*flag = new bool[size + 1];

for (int i = 0; i <= size; ++i)

{ /\*1. 初始化\*/

flag[i] = false;

dist[i] = MAX;

p[i] = 0;

}

ArcNode \*ptr = graph->heads[start\_id].first;

while (ptr != NULL)

{

dist[ptr->id] = 1;

p[ptr->id] = start\_id;

ptr = ptr->next;

}

flag[start\_id] = true;

for (size\_t i = 1; i < size; i++)

{

if (flag[end\_id])

{

break;

}

int min\_index = min(dist, flag, size);

flag[min\_index] = true;

int min\_dist = dist[min\_index];

/\* 更新 \*/

if (min\_dist < MAX)

{

for (size\_t j = 1; j <= size; j++)

{

if (!flag[j])

{

int distance = MAX;

ArcNode \*ptr = graph->heads[min\_index].first;

while (ptr != nullptr)

{

if (ptr->id == j)

{

distance = 1;

break;

}

ptr = ptr->next;

}

if (!flag[j] && distance != MAX && min\_dist + distance < dist[j])

{

dist[j] = min\_dist + distance;

p[j] = min\_index;

}

}

}

}

}

int cur\_index = end\_id;

int pre\_index = p[cur\_index];

stack<int> line;

while (pre\_index != cur\_index)

{

line.push(cur\_index);

cur\_index = pre\_index;

pre\_index = p[pre\_index];

}

line.push(start\_id);

show(line, remaps[start\_id], remaps[end\_id]);

delete[] dist;

delete[] p;

delete[] flag;

}

void findLeastTransfer(string start, string end)

{

//找到起始点和终点

int start\_id = maps[start];

int end\_id = maps[end];

//设定转车最大次数

int N = 4;

int n = 0;

queue<pair<pair<int, int>, int>> que;

que.push(make\_pair(make\_pair(start\_id, 0), 0));

int pre\_level = 0;

int preParent = 0;

while (n < N)

{

int parent = que.front().second;

start\_id = que.front().first.first;

int level = que.front().first.second;

if (level != pre\_level)

{

n++;

pre\_level = level;

}

que.pop();

if (parent != 0)

{

intersection(parent, start\_id);

}

bool flag = intersection(start\_id, end\_id);

if (flag)

{ //找到通路

break;

}

else

{ //没有找到

if (!path.empty())

{

path.pop\_back();

}

int s\_index = -1;

for (size\_t i = 0; i < graph->stations.size(); i++)

{

if (graph->stations[i].id == start\_id)

{

s\_index = i;

break;

}

}

//把所有跟跟站点相关的站点都放入队列 d

for (auto it = graph->stations[s\_index].buss.begin(); it != graph->stations[s\_index].buss.end(); ++it)

{

for (size\_t i = 0; i < graph->lines.size(); i++)

{

if (graph->lines[i].line.no == \*it)

{

for (size\_t j = 0; j < graph->lines[i].stations.size(); j++)

{

if (graph->lines[i].stations[j] != start\_id)

{

que.push(make\_pair(make\_pair(graph->lines[i].stations[j], level + 1), start\_id));

}

}

}

}

}

}

}

if (n >= N)

{

cout << "从 " << start << " 到 " << end

<< " 乘坐公交车需要大于10次转车，建议选择其它交通工具！";

return;

}

if (path.size() == 0)

{

cout << "查询失败！" << endl;

exit(0);

}

stack<pair<int, vector<int>>> final\_res; //最终公交车的结果

auto tmp = path2[path2.size() - 1];

final\_res.push(tmp.first);

int index = path2.size() - 2;

if (index >= 0)

{

int bef\_end = tmp.first.first;

for (int i = index; i >= 0; --i)

{

int i\_index = -1;

for (int j = i; j >= 0; --j)

{

if (path2[j].second == bef\_end)

{

i\_index = j;

}

}

if (i\_index != -1)

{

final\_res.push(path2[i\_index].first);

bef\_end = path2[i\_index].first.first;

}

i = i\_index;

--i;

}

}

cout << "\n从 " << start << " 到 " << end << " 的乘车路线如下： \n\t";

auto t\_start = final\_res.top();

final\_res.pop();

cout << "从 " << remaps[t\_start.first] << " 乘坐： ";

for (size\_t j = 0; j < t\_start.second.size(); j++)

{

cout << t\_start.second[j] << "线";

if (j != t\_start.second.size() - 1)

{

cout << " 或 ";

}

}

while (!final\_res.empty())

{

auto t = final\_res.top();

cout << " 到 " << remaps[t.first] << ",然后乘坐: ";

for (size\_t j = 0; j < t.second.size(); j++)

{

cout << t.second[j] << "线";

if (j != t.second.size() - 1)

{

cout << " 或 ";

}

}

final\_res.pop();

}

cout << " 到终点站： " << end;

}

bool check(string start, string end)

{

return maps.find(start) != maps.end() && maps.find(end) != maps.end();

}

void enquiry(int num)

{

string start;

string end;

cout << "\n请输入起点： ";

cin >> start;

cout << "请输入终点： ";

cin >> end;

if (check(start, end))

{

if (start == end)

{

cout << "输入的起始站点与终点一致！无需换乘！" << endl;

return;

}

if (num == 1)

{

findLeastTransfer(start, end);

path.clear();

}

else if (num == 2)

{

Dijkstra(start, end);

path.clear();

}

}

else

{

cout << "输入有误，请重新输入！" << endl;

}

cout << "\n\n";

}

int main()

{

readFileRoutine();

int num = 0;

while (1)

{

system("cls");

menu();

cout << "\n\t请输入你要查询的方式： ";

cin >> num;

switch (num)

{

case 1:

{

enquiry(1);

system("pause");

break;

}

case 2:

{

enquiry(2);

system("pause");

break;

}

case 3:

{

cout << "\n\t谢谢使用！" << endl;

exit(0);

}

default:

break;

}

}

return 0;

}

void menu()

{

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

cout << "\*\* 欢迎使用公交查询系统 \*\*" << endl;

cout << "\*\* [1] 查询最少换乘路线 \*\*" << endl;

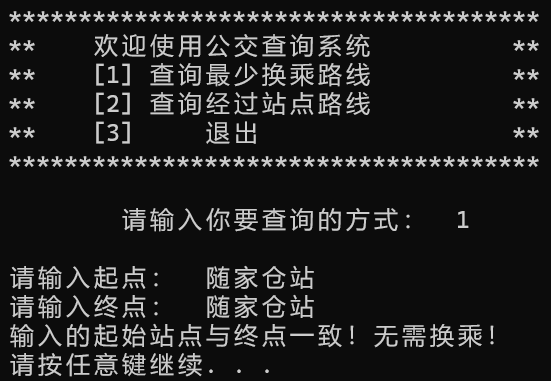
cout << "\*\* [2] 查询经过站点路线 \*\*" << endl;

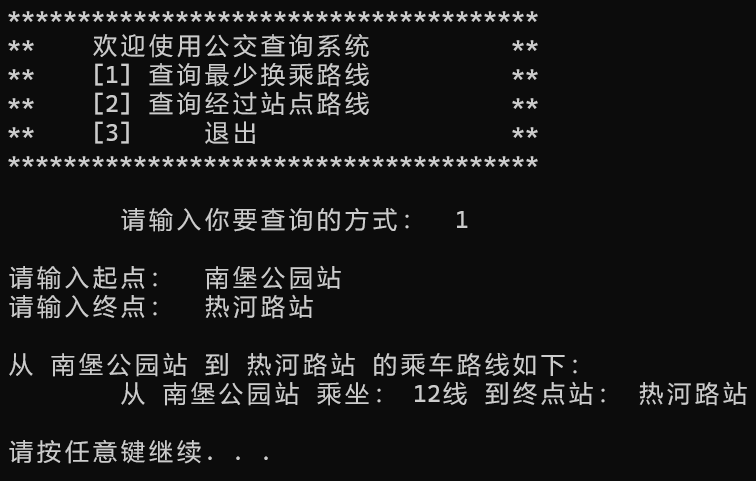
cout << "\*\* [3] 退出 \*\*" << endl;

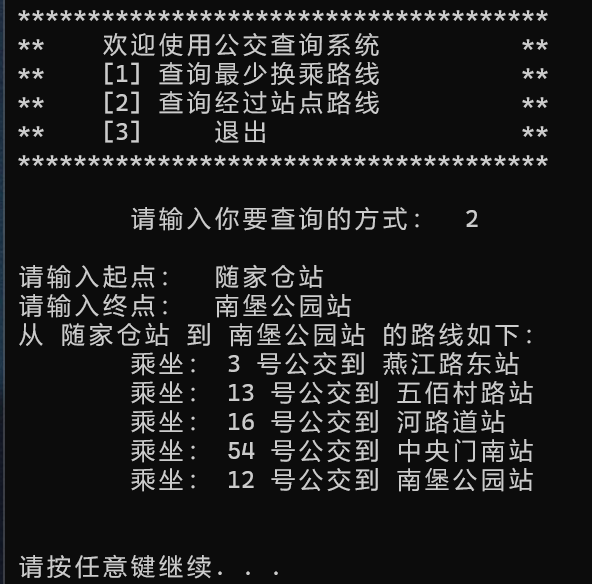
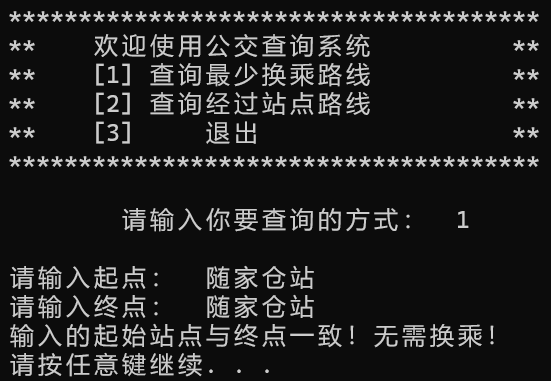
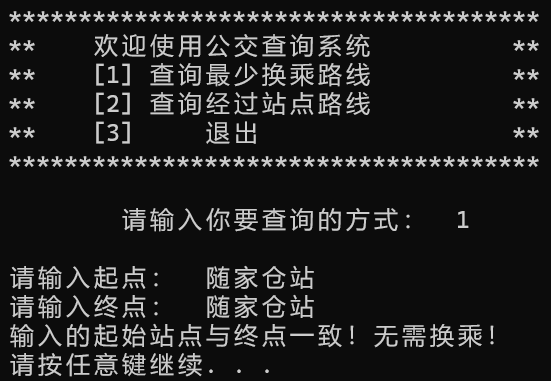
cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

}

#### 测试数据和结果





#### 时间复杂度：

迪杰斯特拉算法O(n^2)

BFS算法O（N）

#### 反思：

BFS算法理论上来说，由于在求路径时我们已经知道起点和终点，实际上我可以考虑使用双向BFS进行优化，即从起点和终点都进行BFS，减少不必要的搜索。（可以在之后选做题中的词梯和迷宫问题中使用）

#### 改进：

1. 增加控制最大转车次数，如果要换乘的次数超过10次会进行提示，建议改乘坐其他交通工具。
2. 如果起点和终点一致直接输出不必调用求路径的函数

#### 该题代码行：

604行