数据结构与算法　实验报告

教务三班　15336999　严紫熙
教务四班　15336251　郑树诚

2016 年 12 月 16 日

1　实验目的

熟悉图论的最短路算法，搜集相关数据，完成一个简单的广州地铁的线路查询系统。

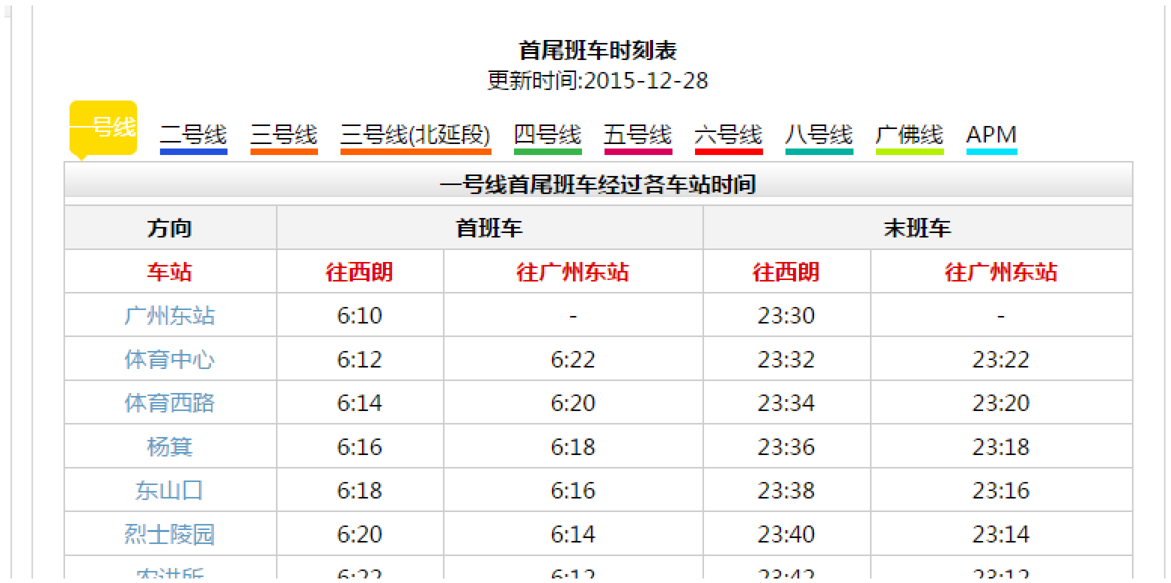
2　实验环境

1. Windows
2. Visual Studio

3　实验内容

3.1　数据搜集

要建立广州地铁的线路模型，我们首先需要线路、站点有关的信息。我们首先参考了广州地铁官网。在服务时间网页，我们找到了广州地铁每条线路的首末班车时刻表（来自http://cs.gzmtr.com/ckfw/fwsj）：



首尾班车时刻表
更新时间:2015-12-28

一号线　二号线　三号线　三号线(北延段)　四号线　五号线　六号线　八号线　广佛线　APM

一号线首尾班车经过各车站时间

方向	首班车		末班车	
车站	往西朗	往广州东站	往西朗	往广州东站
广州东站	6:10	-	23:30	-
体育中心	6:12	6:22	23:32	23:22
体育西路	6:14	6:20	23:34	23:20
杨箕	6:16	6:18	23:36	23:18
东山口	6:18	6:16	23:38	23:16
烈士陵园	6:20	6:14	23:40	23:14

考虑到末班车从始发站依次经过每个站点，我们用每个站点的末班车时间减去始发站的末班车时间，并将此作为每个站距离始发站的时间。以一号线为例，我们整理后的数据形如：

```
广州东站 0
体育中心 2
体育西路 4
杨箕 6
东山口 8
⋮
```

在代码中，我们读取每个站点距离始发站的时间，减去上一个站的时间，就能获得站点两两之间的时间。

3.2 数据建模

地铁线路之间的换乘是通过换乘站来完成的，而换乘站的特点为，相同的站点名称在多条线路中同时出现。因此，在读入每条线路的站点信息之后，我们首先通过站点名称标记站点：对于不同线路中出现的相同名称的站点，我们给予它们同样的序号，以表达它们是同一个站点。这一步我们使用了`STL`中的`std::map`来完成。

地铁线网上，相邻的两站相互可达，而所需的时间各不相同。因此，我们选择无向带权图来表达地铁线网。其中，每一个顶点代表一个地铁站；每一条边代表这条边两个顶点所代表的的地铁站之间可以互达，权值表示这一站路所需的时间或者这两站之间的距离。

建立出代表地铁线网的图之后，我们就可以在图上执行最短路算法，以获得两站之间的最短距离或者最短时间了。

3.3 算法细节

在最短路算法的选择上，我们并没有选择课上介绍的Dijkstra算法，而是选择了改良自Bellman-Ford算法的SPFA。关于SPFA算法的详细内容，请参考报告末的原始论文。

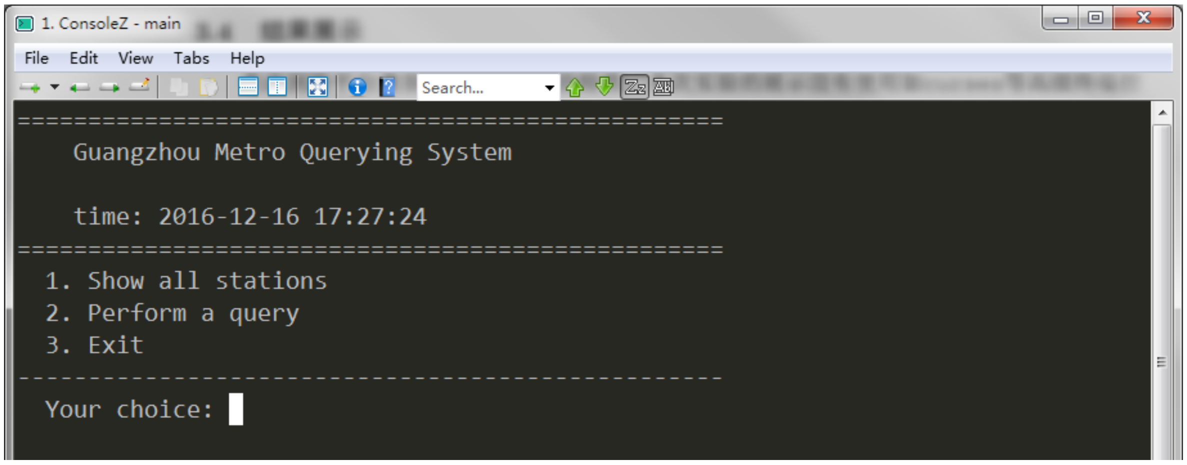
3.4 结果展示

考虑到跨平台兼容性和`C++`语言的特性，本次实验的展示没有使用如`curses`等高级终端控制库。为了达到更好的展示效果，我们在代码中按平台定义了一些内容，如相关头文件的调用：

```c
#ifdef WIN32
#include <Windows.h>
#else
#include <unistd.h>
#endif
```

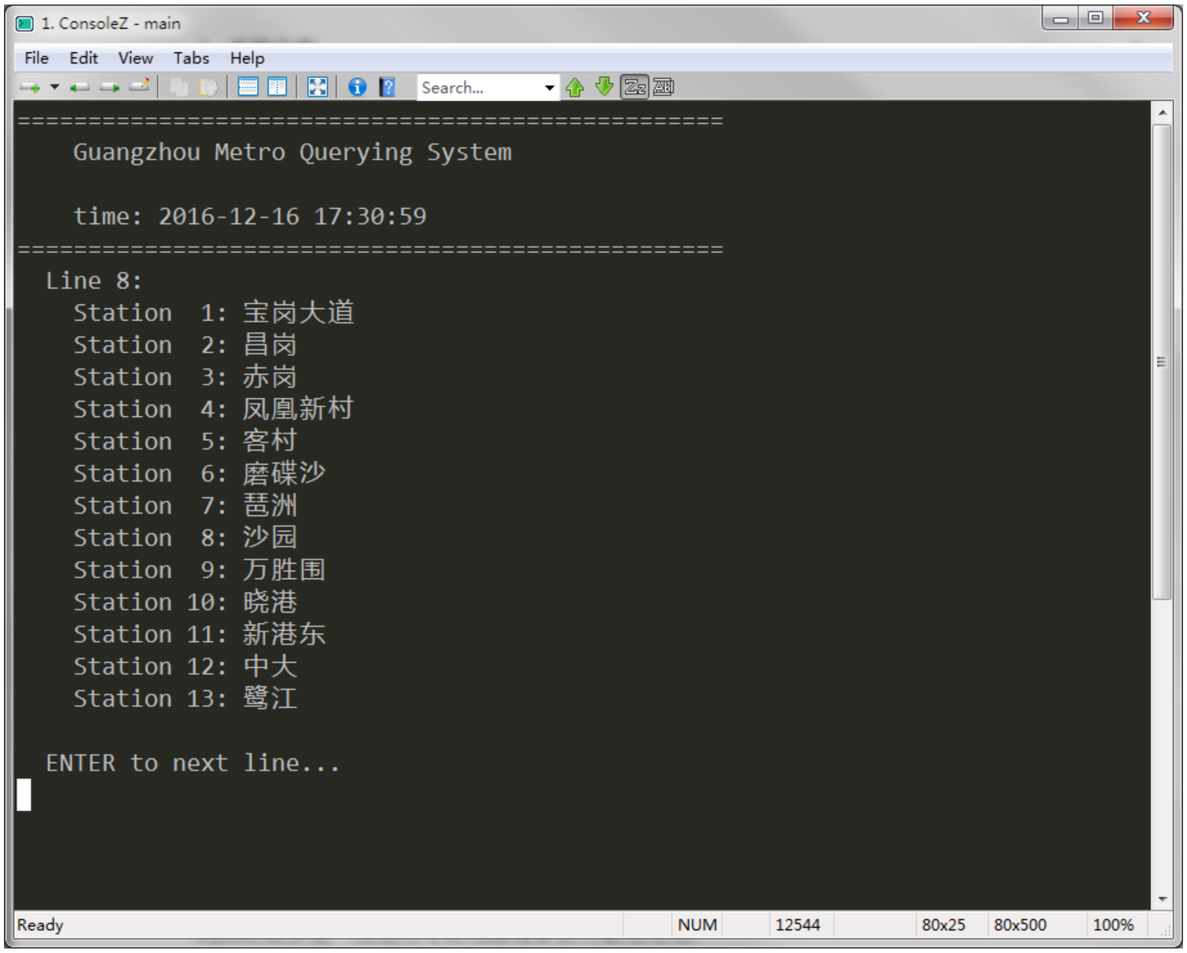
延时控制函数和清理终端内容的函数也做了类似定义，详见随文所附代码，这里不再赘述。

程序主要提供两个功能：查询两个站点之间最短时间、最短距离或最便宜的票价，这是核心功能；以及查看所有线路的所有站点，做为辅助功能；程序运行界面如下：



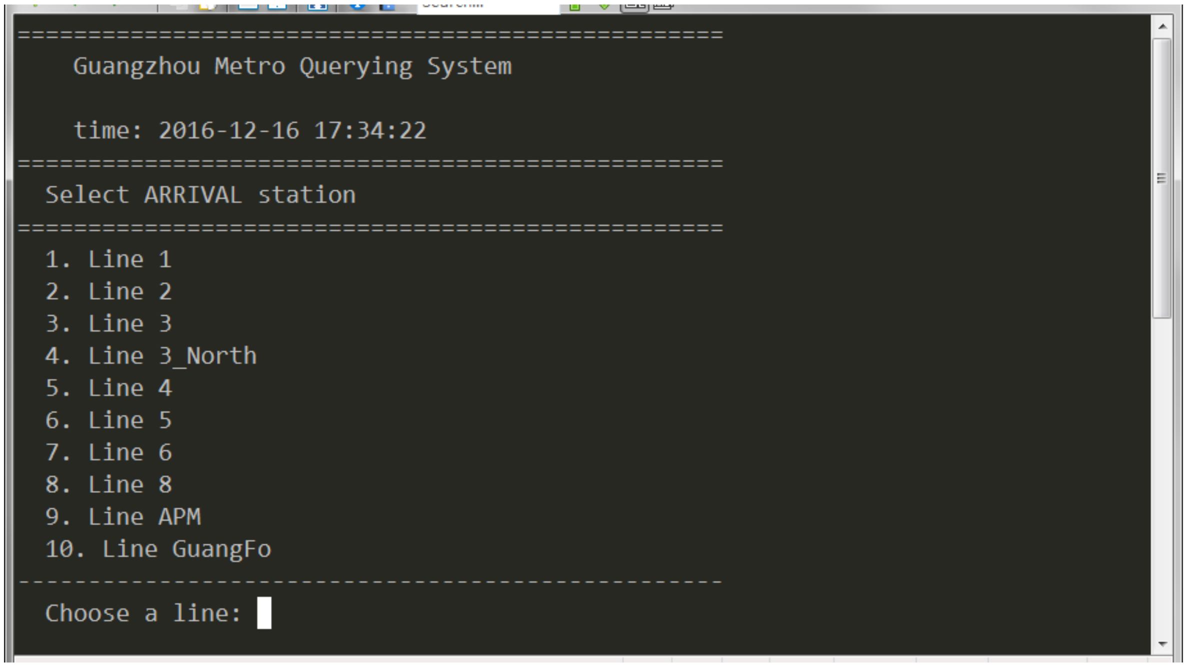
```
==================================================
    Guangzhou Metro Querying System

    time: 2016-12-16 17:27:24
==================================================
  1. Show all stations
  2. Perform a query
  3. Exit
--------------------------------------------------
  Your choice:
```

选择Show all stations查看所有站点的界面：

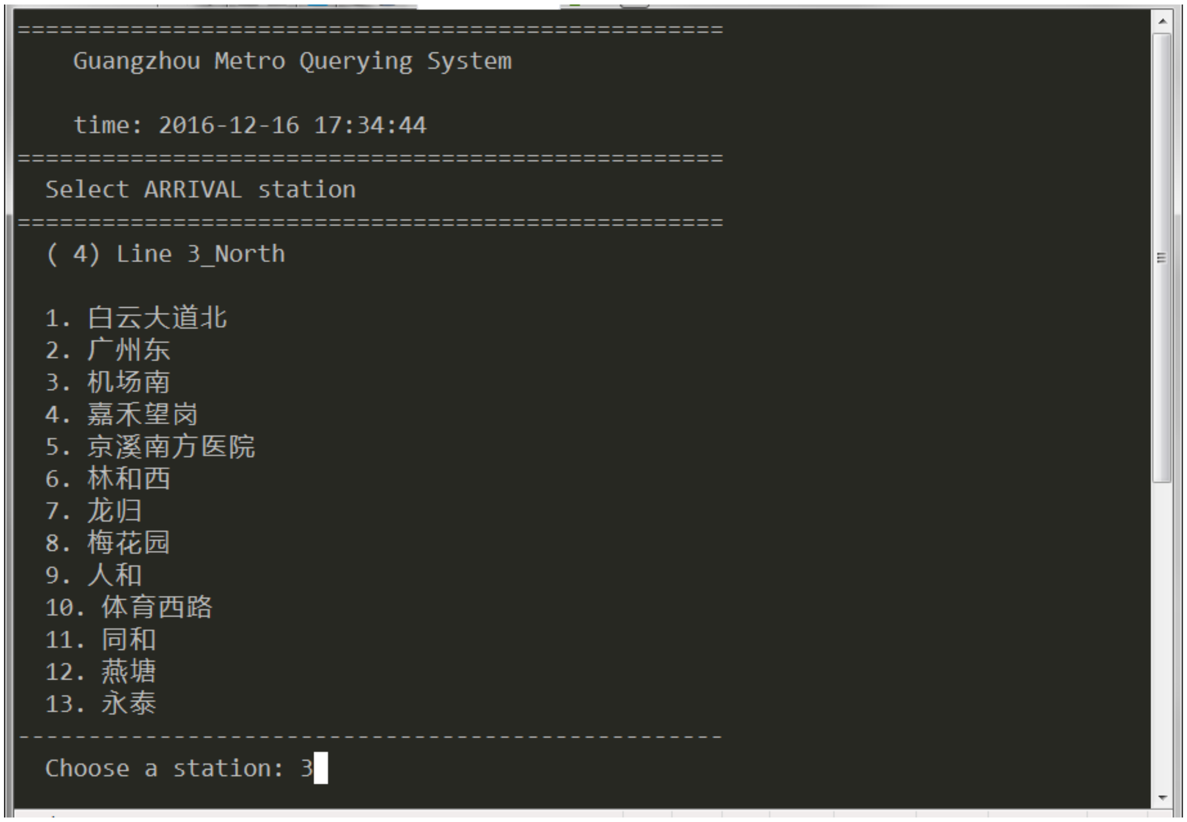


```
==================================================
    Guangzhou Metro Querying System

    time: 2016-12-16 17:30:59
==================================================
  Line 8:
    Station  1: 宝岗大道
    Station  2: 昌岗
    Station  3: 赤岗
    Station  4: 凤凰新村
    Station  5: 客村
    Station  6: 磨碟沙
    Station  7: 琶洲
    Station  8: 沙园
    Station  9: 万胜围
    Station 10: 晓港
    Station 11: 新港东
    Station 12: 中大
    Station 13: 鹭江

  ENTER to next line...
```

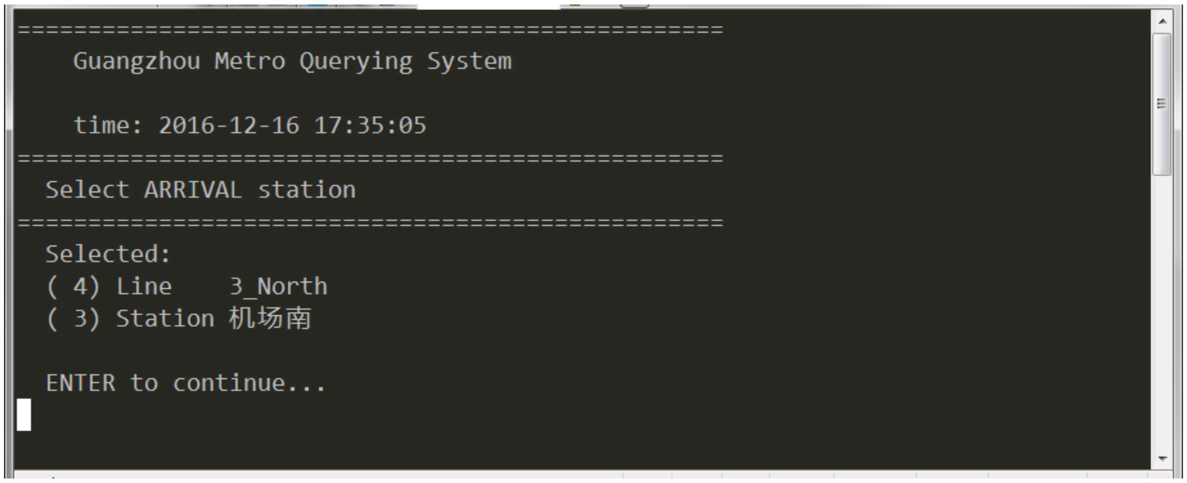
选择Perform a query之后，首先要分别选择出发站（Departure）和终到站（Arrival）。选择站点的界面如下：



```
==================================================
    Guangzhou Metro Querying System

    time: 2016-12-16 17:34:22
==================================================
  Select ARRIVAL station
==================================================
  1. Line 1
  2. Line 2
  3. Line 3
  4. Line 3_North
  5. Line 4
  6. Line 5
  7. Line 6
  8. Line 8
  9. Line APM
  10. Line GuangFo
--------------------------------------------------
  Choose a line: 
```

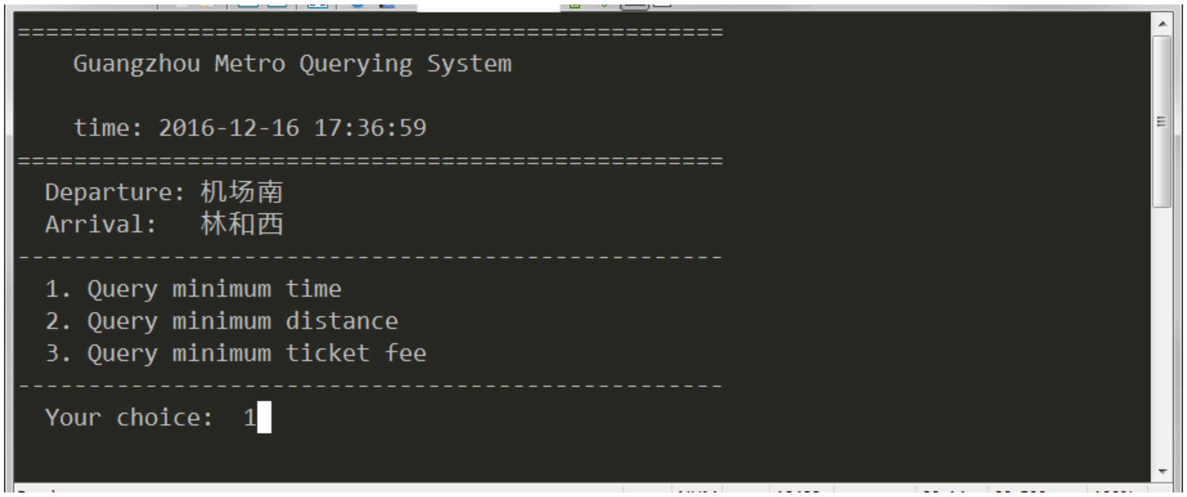


```
==================================================
    Guangzhou Metro Querying System

    time: 2016-12-16 17:34:44
==================================================
  Select ARRIVAL station
==================================================
  ( 4) Line 3_North

  1. 白云大道北
  2. 广州东
  3. 机场南
  4. 嘉禾望岗
  5. 京溪南方医院
  6. 林和西
  7. 龙归
  8. 梅花园
  9. 人和
  10. 体育西路
  11. 同和
  12. 燕塘
  13. 永泰
--------------------------------------------------
  Choose a station: 3
```



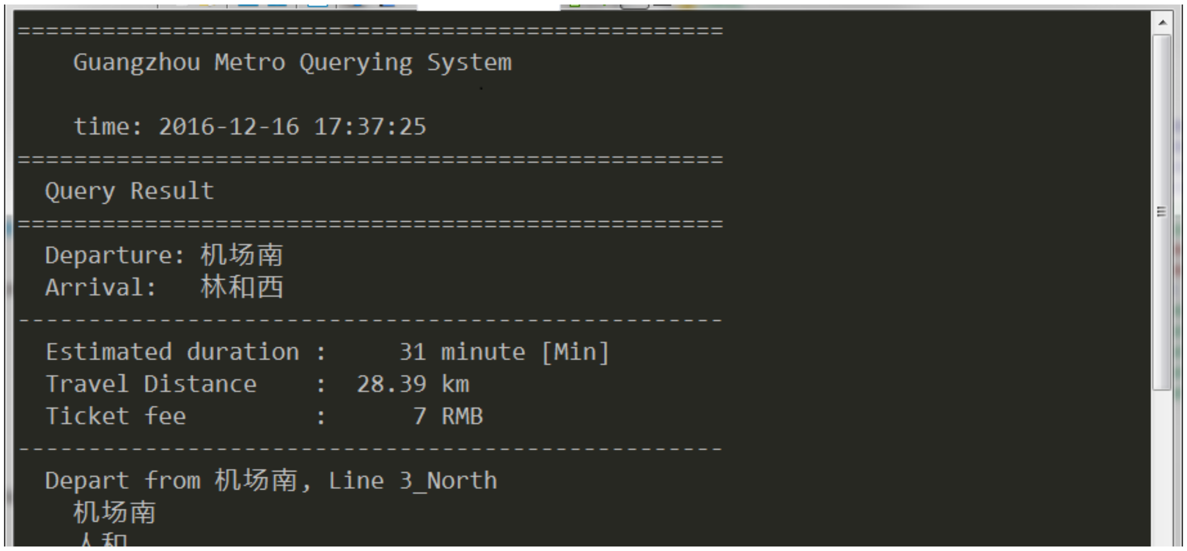
```
==================================================
    Guangzhou Metro Querying System

    time: 2016-12-16 17:35:05
==================================================
  Select ARRIVAL station
==================================================
  Selected:
  ( 4) Line    3_North
  ( 3) Station 机场南

  ENTER to continue...
```

选择好站点之后，选择查询项目：



```
==================================================
    Guangzhou Metro Querying System

    time: 2016-12-16 17:36:59
==================================================
  Departure: 机场南
  Arrival:   林和西
--------------------------------------------------
  1. Query minimum time
  2. Query minimum distance
  3. Query minimum ticket fee
--------------------------------------------------
  Your choice:  1
```

得到查询结果的概况部分。其中，作为查询项目的耗时后面标注了`[Min]`:

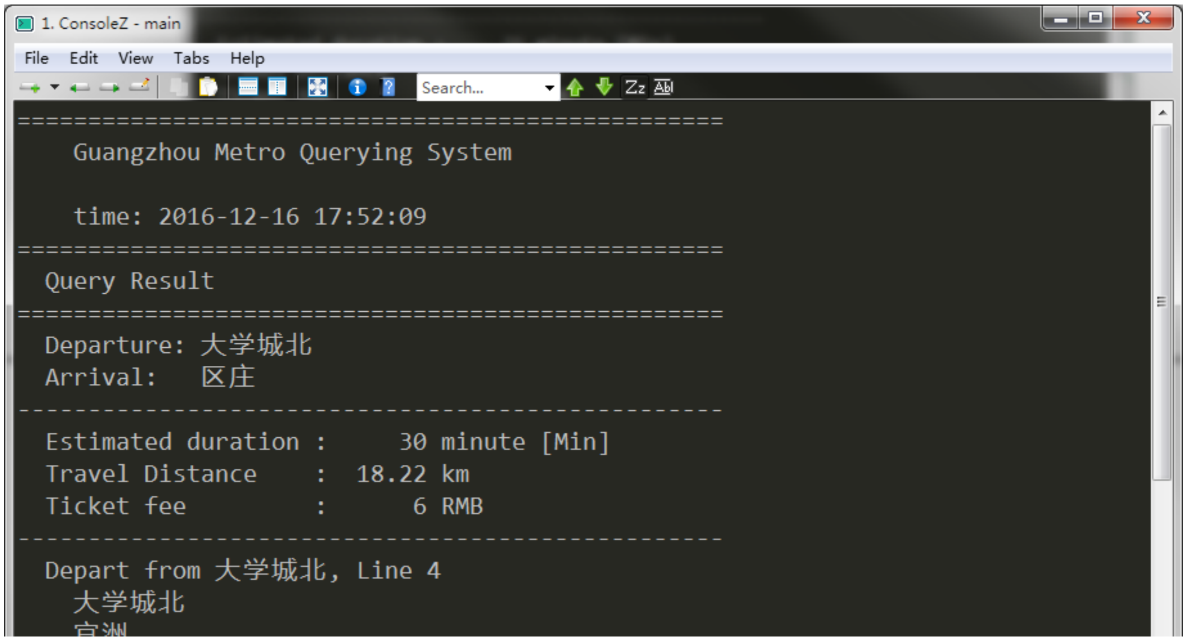


```
==================================================
    Guangzhou Metro Querying System

    time: 2016-12-16 17:37:25
==================================================
  Query Result
==================================================
  Departure: 机场南
  Arrival:   林和西
--------------------------------------------------
  Estimated duration :     31 minute [Min]
  Travel Distance    :  28.39 km
  Ticket fee         :      7 RMB
--------------------------------------------------
  Depart from 机场南, Line 3_North
    机场南
    人和
```

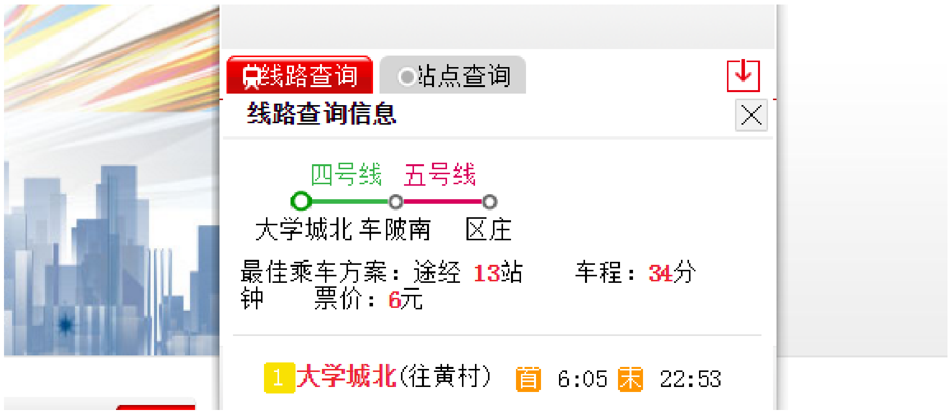
当线路需要换乘时的展示：

4 实验分析

我们查询一段较长的路程，并将查到的结果和官网的查询系统进行比较：



```
==================================================
    Guangzhou Metro Querying System

    time: 2016-12-16 17:52:09
==================================================
  Query Result
==================================================
  Departure: 大学城北
  Arrival:   区庄
--------------------------------------------------
  Estimated duration :     30 minute [Min]
  Travel Distance    :  18.22 km
  Ticket fee         :       6 RMB
--------------------------------------------------
  Depart from 大学城北, Line 4
    大学城北
    官洲
```



最佳乘车方案：途经 13站 车程：34分钟 票价：6元

1 大学城北(往黄村) 首 6:05 末 22:53

可以看到，我们得到的费用正确，所需时间比官网略少。我们做出如下猜测：

1. 官网的查询系统计算了线路换乘的时间
2. 官网的查询系统考虑了高峰期所需时间较长的问题

考虑站点换乘所需的时间和结合官网数据考虑高峰拥堵情况，也是我们的程序下一步可以努力的方向。

5 参考文献

1. 段凡丁. 关于最短路径的SPFA 快速算法[J]. 西南交通大学学报, 1994, 29(2): 207-212.

6 附录

这里附上本次实验的程序的全部代码。

1. main.cpp

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

#include "metro.h"

#ifdef WIN32
#include <Windows.h>
#else
#include <unistd.h>
#endif

void clearScreen()
{
    #ifdef WIN32
    system("cls");
    #else
    system("clear");
    #endif
}

void milliSleep(size_t ms) {
    #ifdef WIN32
    Sleep(ms);
    #else
    timespec ts;
    ts.tv_sec = ms / 1000;
    ts.tv_nsec = (ms % 1000) * 1000000L;
    nanosleep(&ts, NULL);
    #endif
}

```

```
int getInt()
{
    int res = 0;
    char c = getchar();
    for ( ; c < '0' || '9' < c; c = getchar());
    for ( ; '0' <= c && c <= '9'; c = getchar())
        res = res * 10 + c - '0';
    return res;
}

char * time()
{
    char * res = new char[100];
    const time_t t = time(NULL);
    struct tm * current = localtime(&t);
    sprintf(res, "%04d-%02d-%02d %02d:%02d:%02d", current->tm_year +
       1900,
            current->tm_mon + 1, current->tm_mday, current->tm_hour,
            current->tm_min, current->tm_sec);
    return res;
}

void enterConfirm()
{
    for (char c = getchar(); c != '\n'; c = getchar());
}

void bigLine()
{
    for (int i = 0; i < 50; ++i)
        putchar('=');
    puts("");
}

void smallLine()
{
    for (int i = 0; i < 50; ++i)
        putchar('-');
    puts("");
}
```

```

void header()
{
    clearScreen();
    bigLine();
    printf("    Guangzhou Metro Querying System\n\n");
    printf("    time: %s\n", time());
    bigLine();
}

void action1(Metro * metro)
{
    vector< pair< string, vector<string> > > allLines =
       metro->list_all_subway();
    for (size_t i = 0; i < allLines.size(); ++i) {
        header();
        printf("  Line %s:\n", allLines[i].first.c_str());
        vector<string> &allStations = allLines[i].second;
        for (size_t j = 0; j < allStations.size(); ++j)
            printf("    Station %2lu: %s\n", j + 1,
               allStations[j].c_str());
        if (i != allLines.size() - 1) {
            printf("\n  ENTER to next line...\n");
            enterConfirm();
        }
    }
    printf("\n\n  ENTER to continue..\n");
    enterConfirm();
}

int pickStation(const char msg[], Metro * metro)
{
    header();
    puts(msg);
    bigLine();
    vector< pair< string, vector<string> > > allLines =
       metro->list_all_subway();
    for (size_t i = 0; i < allLines.size(); ++i)
        printf("  %lu. Line %s\n", i + 1, allLines[i].first.c_str());
    smallLine();
    int line;
    while (true) {
        printf("  Choose a line: ");
        line = getInt();
        if (0 < line && line <= (int)allLines.size())
            break;
        printf("\n  Invalid input!\n\n");
    }
    header();
    puts(msg);
    bigLine();
    printf("  (%2d) Line %s\n\n", line, allLines[line -
       1].first.c_str());
    vector<string> &allStations = allLines[line - 1].second;
    for (size_t i = 0; i < allStations.size(); ++i)
        printf("  %lu. %s\n", i + 1, allStations[i].c_str());
    smallLine();
    int station;
    while (true) {
        printf("  Choose a station: ");
        station = getInt();
        if (0 < station && station <= (int)allStations.size())
            break;
        printf("\n  Invalid input!\n\n");
    }
    header();
    puts(msg);
    bigLine();
    printf("  Selected:\n");
    printf("  (%2d) Line    %s\n", line, allLines[line -
       1].first.c_str());
    printf("  (%2d) Station %s\n", station, allStations[station -
       1].c_str());
    printf("\n  ENTER to continue...\n");
    enterConfirm();
    return metro->query_station_index(allStations[station - 1]);
}

void printQueryResult(const Response &response, int query_type)
{
    vector< pair<string, vector<string> > > path = response.path;
    int money = response.money;
    int cost_time = response.cost_time;
    double distance = response.distance;
    vector<int> time_between_station = response.time_between_station;

    header();
    printf("  Query Result\n");
    bigLine();
    printf("  Departure: %s\n", path.front().second.front().c_str());
    printf("  Arrival:   %s\n", path.back().second.back().c_str());
    smallLine();
    char msg[100];
    sprintf(msg, "  Estimated duration : %6d minute", cost_time);
    printf("%.40s %s\n", msg, (query_type == 1 ? "[Min]" : ""));
    sprintf(msg, "  Travel Distance    : %6.2lf km    ", distance);
    printf("%.40s %s\n", msg, (query_type == 2 ? "[Min]" : ""));
    sprintf(msg, "  Ticket fee         : %6d RMB   ",    money);
    printf("%.40s %s\n", msg, (query_type == 3 ? "[Min]" : ""));
    smallLine();
    for (size_t i = 0; i < path.size(); ++i) {
        string& line = path[i].first;
        vector<string>& stations = path[i].second;
        if (i == 0)
            printf("  Depart from %s, Line %s\n",
                stations[0].c_str(), line.c_str());
        else
            printf("  Interchange to Line %s\n", line.c_str());
        for (size_t j = 0; j < stations.size(); ++j)
            printf("    %s\n", stations[j].c_str());
    }
    printf("  Arrived at %s, Line %s\n",
       path.back().second.back().c_str(), path.back().first.c_str());
    printf("\n\n  ENTER to continue...\n");
    enterConfirm();
}

void subMenu2(Metro * metro)
{
    int src_ind, dest_ind;
    while (true) {
        if ((src_ind = pickStation("  Select DEPARTURE station",
            metro)) != -1)
            break;
        printf("\n  Unable to find such station!\n");
        milliSleep(750);
    }
    puts("");
    while (true) {
        if ((dest_ind = pickStation("  Select ARRIVAL station",
            metro)) != -1)
            break;
        printf("\n  Unable to find such station!\n");
        milliSleep(750);
    }
    string src = metro->query_station_name(src_ind);
    string dest = metro->query_station_name(dest_ind);
    int query_type;
    Response response;
    while (true) {
        header();
        printf("  Departure: %s\n", src.c_str());
        printf("  Arrival:   %s\n", dest.c_str());
        smallLine();
        printf("  1. Query minimum time\n");
        printf("  2. Query minimum distance\n");
        printf("  3. Query minimum ticket fee\n");
        smallLine();
        printf("  Your choice:  ");
        switch (query_type = getInt()) {
            case 1:
                response = metro->query_time(src_ind, dest_ind);
                break;
            case 2:
                response = metro->query_distance(src_ind, dest_ind);
                break;
            case 3:
                response = metro->query_money(src_ind, dest_ind);
                break;
            default:
                printf("\n  Invalid input!\n");
                milliSleep(750);
                continue;
        }
        break;
    }
    printQueryResult(response, query_type);
}

bool mainMenu(Metro * metro)
{
    clearScreen();
    header();
    printf("  1. Show all stations\n");
    printf("  2. Perform a query\n");
    printf("  3. Exit\n");
    smallLine();
    printf("  Your choice: ");
    switch (getInt()) {
        case 1:
            action1(metro);
            return true;
        case 2:
            subMenu2(metro);
            return true;
        case 3:
            return false;
        default:
            printf("\n  Invalid input!\n");
            milliSleep(750);
    }
    return true;
}

void exitMessage()
{
    clearScreen();
    header();
    printf("    Thanks for your using!\n");
    milliSleep(1000);
```

```
}

int main()
{
    Metro *metro = new Metro(Metro::SUBWAY_NAME, 10);
    while (mainMenu(metro))
        ;
    exitMessage();
    return 0;
}
```

2. metro.h

```
#include <cstdio>
#include <iomanip>
#include <cstring>
#include <map>
#include <string>
#include <vector>
#include <cmath>
#include <set>
#include <queue>
#include <algorithm>
#include <fstream>
#include <sstream>
#include <iostream>
using namespace std;

#define foreach(x, y) \
    for(__typeof((y).begin()) x = (y).begin(); x != (y).end(); ++x)
#define INF (10001)

struct Response {
    vector<pair<string, vector<string> > > path;
    int money, cost_time;
    double distance;
    vector<int> time_between_station;

    Response() {
        money = cost_time = 0, distance = 0.;
    }
};
```

```

struct Edge {
    string start, end;
    int cost_time;
    double distance;

    Edge(
            const string &start,
            const string &end,
            const int &cost_time,
            const double &distance
        ) :
        start(start),
        end(end),
        cost_time(cost_time),
        distance(distance) {
    }

    bool operator <(const Edge &t) const {
        return end < t.end;
    }
};

struct State {
    string pre_station;
    int cost_time, cost_money, interchange;
    double distance, real_distance;

    State(
            const string &pre_station = "",
            const int &cost_time = INF,
            const int &cost_money = INF,
            const int &interchange = INF,
            const double &distance = INF
        ):
        pre_station(pre_station),
        cost_time(cost_time),
        cost_money(cost_money),
        interchange(interchange),
        distance(distance) {
```

```
        real_distance = 0.;
    }

    int get_cost() const {
        int ret = cost_money + 2;
        const double EPS = 1e-5;
        if(distance > 4. + EPS)
            ret += ceil((min(distance, 12.) - 4. - EPS) / 4.);
        if(distance > 12. + EPS)
            ret += ceil((min(distance, 24.) - 12. - EPS) / 6.);
        if(distance > 24. + EPS)
            ret += ceil((distance - 24. - EPS) / 8.);
        return ret;
    }

    double get_distance() const {
        double ret = distance + real_distance;
        return ret;
    }
};

struct Comp {
    string dominate;

    Comp(const char *dominate):dominate(dominate) {}

    bool operator ()(const State &a, const State &b) {
        if(dominate == "Distance" || dominate == "distance")
            return a.get_distance() < b.get_distance();
        if(dominate == "Money" || dominate == "money")
            return a.get_cost() < b.get_cost();
        return a.cost_time < b.cost_time;
    }
};


class Metro {
private :
    map<string, int> station_name_index;
    map<int, string> station_index_name;
    map<string, set<string> > subway_stations, station_belong;
    map<string, set<Edge> > graph;
    int tot_station;

    int get_station_index(string &name) {
        if(station_name_index.find(name) == station_name_index.end())
            {
            station_name_index[name] = ++tot_station;
            station_index_name[tot_station] = name;
        }
        return station_name_index[name];
    }

    string get_subway_on(const Edge &e) {
        string a = e.start, b = e.end;
        foreach(it, subway_stations) {
            if(it->second.find(a) == it->second.end()) continue;
            if(it->second.find(b) == it->second.end()) continue;
            return it->first;
        }
        return "";
    }

    string get_subway_on(string &a, string &b) {
        foreach(it, subway_stations) {
            if(it->second.find(a) == it->second.end()) continue;
            if(it->second.find(b) == it->second.end()) continue;
            return it->first;
        }
        return "";
    }

    Response parse_response(
            map<string, State> &dist,
            string &start,
            string &end
            ) {
        Response ret;
        int &money = ret.money, &cost_time = ret.cost_time;
        double &distance = ret.distance;
        vector<pair<string, vector<string> > > &path = ret.path;
        vector<int> &time_between_station = ret.time_between_station;
        if(dist.find(end) == dist.end()) {
            money = cost_time = INF, distance = INF;
            path.clear(), time_between_station.clear();
        } else {
            money = dist[end].get_cost();
            cost_time = dist[end].cost_time;
            distance = dist[end].get_distance();

            string pre_subway = "", pre_station = end;
            vector<string> pass;
            pass.push_back(end);
            while(pre_station != start) {
                State pre = dist[pre_station];
                string now_station = pre.pre_station;
                State now = dist[now_station];
                string now_subway = get_subway_on(now_station,
                   pre_station);
                if(now_subway != pre_subway && pre_subway != "") {
                    if(pass.size()) {
                        reverse(pass.begin(), pass.end());
                        path.push_back(make_pair(pre_subway, pass));
                        pass.clear();
                    }
                }

                pass.push_back(pre_station);
                pass.push_back(now_station);
                time_between_station.push_back(
                        pre.cost_time - now.cost_time
                    );

                pre_station = now_station, pre_subway = now_subway;

                if(pre_station == start && pass.size()) {
                    reverse(pass.begin(), pass.end());
                    path.push_back(make_pair(now_subway, pass));
                    pass.clear();
                }
            }
            reverse(time_between_station.begin(),
               time_between_station.end());
            reverse(path.begin(), path.end());
        }
        return ret;
    }

public :
    static const char* SUBWAY_NAME[];

    void read_data(
            const string &filename,
            vector<pair<string, int> > &station_time,
            vector<pair<string, double> > &station_distance
        ) {
        station_time.clear(), station_distance.clear();

        ifstream in(filename.c_str(), ios::in);
        // ofstream out((filename + ".out").c_str(), ios::out);
        string line_data;
        while(getline(in, line_data)) {
            if((int) line_data.size() <= 0) break;
            int pos = line_data.find(' ');
            string name = line_data.substr(0, pos);
            string time_number = line_data.substr(pos + 1,
               line_data.size() - pos);
            int time;
            sscanf(time_number.c_str(), "%d", &time);
            station_time.push_back(make_pair(name, time));
            // foreach(c, name) out << hex << ((int) *c) << ' ';
            // out << endl;
            // out << name << ' ' << time << endl;
        }
        // out << flush;
        while(getline(in, line_data)) {
            if((int) line_data.size() <= 0) break;
            double distance = 0.;
            string name;
            if(line_data.find(' ') == string::npos)
```

```
                name = line_data;
            else {
                int pos = line_data.find(' ');
                name = line_data.substr(0, pos);
                string distance_number = line_data.substr(pos + 1,
                   line_data.size() - pos);
                sscanf(distance_number.c_str(), "%lf", &distance);
            }
            station_distance.push_back(make_pair(name, distance));
            // foreach(c, name) out << hex << ((int) *c) << ' ';
            // out << endl;
            // out << name << ' ' << distance << endl;
        }
        in.close();
    }

    void check_reverse(
            vector<pair<string, int> > &station_time,
            vector<pair<string, double> > &station_distance) {

        if((station_distance.begin())->first !=
                (station_time.begin())->first) {
            vector<pair<string, double> > temp;
            for(int i = station_time.size() - 1; i >= 0; --i)
                if(i) temp.push_back(make_pair(
                            station_distance[i].first,
                            station_distance[i - 1].second
                        ));
                else temp.push_back(make_pair(
                            station_distance[i].first,
                            0.
                        ));

            station_distance = temp;
        }
    }

    Metro(const char **a, int station_number) {
        tot_station = 0;

        // debug
        cout << station_number << endl;
        for(int i = 0; i < station_number; ++i) {
            const string subway_name(SUBWAY_NAME[i]);
            vector<pair<string, int> > station_time;
            vector<pair<string, double> > station_distance;
            read_data("data\\" + subway_name + ".txt", station_time,
               station_distance);

            if(station_time.size() != station_distance.size()) {
                cout << subway_name << " data format is not valid."
                   << endl;
                continue;
            }

        //  for(size_t i = 0; i < station_time.size(); ++i)
        //      cout << (station_time[i].first ==
              station_distance[i].first) << endl;

            check_reverse(station_time, station_distance);
            int num_station = station_time.size();

            set<string> &subway_contain =
               subway_stations[subway_name];
            subway_contain.clear();
            foreach(it, station_time)
                subway_contain.insert(it->first),
                station_belong[it->first].insert(subway_name);
            for(int i = 0; i < num_station - 1; ++i) {
                string start = station_time[i].first,
                       end = station_time[i + 1].first;
                int time_delta =
                    station_time[i + 1].second -
                       station_time[i].second;
                double distance = station_distance[i].second;
                graph[start].insert(Edge(start, end, time_delta,
                   distance));
                graph[end].insert(Edge(end, start, time_delta,
                   distance));

                get_station_index(start);
                get_station_index(end);
            }
        }
    }

    vector<pair<int, string> > list_all_stations() {
        vector<pair<int, string> > ret;
        foreach(it, station_index_name)
            ret.push_back(make_pair(it->first, it->second));
        return ret;
    }

    vector<pair<string, vector<string> > > list_all_subway() {
        vector<pair<string, vector<string> > > ret;
        foreach(it, subway_stations) {
            string subway_name = it->first;
            int num_stations = it->second.size();
            vector<string> stations(num_stations);
            copy(it->second.begin(), it->second.end(),
                stations.begin());
            ret.push_back(make_pair(subway_name, stations));
        }
        return ret;
    }

    Response spfa(string &start, string &end, Comp &cmp) {
        if(start == end) {
            Response res;
            return res;
        }

        map<string, State> dist;
        queue<string> que;
        map<string, bool> inque;

        dist[start] = State("", 0, 0, 0, 0.);
        que.push(start), inque[start] = true;
        while(!que.empty()) {
            string u = que.front();
            que.pop(), inque[u] = false;
            State pre_state = dist[u];
            string pre_subway = get_subway_on(pre_state.pre_station,
               u);
            foreach(sub, graph[u]) {
                Edge e = *sub;
                string now_subway = get_subway_on(e);
                State now(u);
                int &cost_time = now.cost_time,
                    &cost_money = now.cost_money,
                    &interchange = now.interchange;
                double &distance = now.distance, &real_distance =
                   now.real_distance;
                cost_time = pre_state.cost_time + e.cost_time;
                distance = pre_state.distance;
                cost_money = pre_state.cost_money;
                real_distance = pre_state.real_distance;
                interchange = pre_state.interchange;
                if(now_subway != pre_subway && pre_subway != "")
                    ++interchange, cost_time += 2;
                if(now_subway != "APM") distance += e.distance;
                else real_distance += e.distance;
                if(now_subway != pre_subway && now_subway == "APM") {
                    cost_money = now.get_cost() - 2;
                    now.real_distance += now.distance;
                    now.distance = 0.;
                }

                State nex = dist[e.end];
                if(cmp(now, nex)) {
                    dist[e.end] = now;
                    if(!inque[e.end])
                        inque[e.end] = true, que.push(e.end);
                }
            }
        }

        Response path = parse_response(dist, start, end);
        return path;
    }

    Response query(const int &start, const int &end, const string
        &dominate) {
        Comp comp(dominate.c_str());
        string start_name = query_station_name(start),
               end_name = query_station_name(end);
        // debug
        cout << start_name << ' ' << end_name << endl;
        Response ret = spfa(start_name, end_name, comp);
        return ret;
    }

    Response query_money(const int &start, const int &end) {
        return query(start, end, "Money");
    }

    Response query_distance(const int &start, const int &end) {
        return query(start, end, "Distance");
    }

    Response query_time(const int &start, const int &end) {
        return query(start, end, "Time");
    }

    int query_station_index(const string &name) {
        if(station_name_index.find(name) != station_name_index.end())
            return station_name_index[name];
        return -1;
    }

    string query_station_name(const int &index) {
        if(station_index_name.find(index) != station_index_name.end())
            return station_index_name[index];
        return "";
    }

};
const char* Metro::SUBWAY_NAME[] = {
    "1",
    "2",
    "3",
    "3_North",
    "4",
    "5",
    "6",
    "8",
    "GuangFo",
    "APM"
};
```