一、系统需求分析

本次课程设计要实现模拟武汉地铁乘车的路线推荐系统，具体实现的功能大致分为五条：

一、构图。

将线路信息、站点信息的数据文件读入系统，并进行构图，把用户指定的线路的站点依次显示出来。要求用户输入线路的序号数字，系统会顺次输出本线路的所有站点。对于用户错误输入的线路，如不存在的五号线，系统能够提示用户输入错误，并进行重新输入。

二、浏览。

指定线路、起始站点，并从该站点逐步找出下一站点，或再换乘站点进行换线，实现对地铁路线站点的依次浏览。要求用户输入指定的线路和站点名字，系统会输出下一站点（两个方向可选），若到换乘站点可以指定换乘到另一条线路，并继续浏览后续站点。对于用户错误输入的线路或站点，系统能够提示用户输入错误，并进行重新输入。

三、列车时刻表。

设置当前时间，为后续的乘车匹配运行时刻表。要求用户输入当前时间，要查询的线路和站点，以及时间范围，系统会输出当前时间前后指定时间范围的离开和即将到来的列车时间。对于用户错误输入的时间、线路或站点，系统能够提示用户错误输入，并进行重新输入。

四、查询。

起始与终点的两站路线推荐，本系统的核心功能。要求用户输入查询的起始线路，起始站点序号，终点线路，终点站点序号，并设置当前时间，选择是否考虑拥挤度，若考虑，是否自己配置拥挤度，是否自己设置容忍的拥挤度下限。系统根据用户的上述要求，求解 1~3 条合理路线，有要求到达终点站点时间在该线路的正常运行时间范围内，否则为不可到达，合理线路的顺序也要按票价由低到高的方式排列，同时输出每一条合理线路的乘车期间的总加权拥挤度。若用户考虑了拥挤度，在上面计算每个站点间在相应的时间段内的拥挤百分数的基础上，按照拥挤因子折合乘车时间的加权结果，按照加权总时间尽量短的方式推荐排序。若用户自己配置了拥挤度，则要根据配置更改拥挤度，若用户设置了容忍的拥挤度下限，则输出路线要避开相应拥挤度的线路。

五、信息安全。

信息安全方面的要求。本系统由于并不复杂，且武汉地铁线路为公开信息，本系统选择加密用户输入的所有信息进行信息保护，本系统通过编写 python 脚本的方式对用户输入的所有信息进行 md5 加密，保存在本地。

二、总体设计

从全局角度出发，本系统大致分为五个模块：主程序模块、数据结构定义模块、主要功能模块、计算函数模块、信息安全模块。

主程序模块是统筹其他模块的关键，程序从主程序模块开始，先显示菜单，然后提示用户输入信息，并根据用户的输入，调用其他模块。

数据结构定义模块定义了本系统核心数据结构：无向图的邻接表和邻接矩阵，以及各个函数所用的数据结构如向量、栈、队列等。本模块主要先后定义邻接边表，节点，以及图结构，定义初始化图函数，邻接边表插入函数，以及图的插入函数。

主要功能模块定义了本系统的核心功能以及核心算法，包括文件的读入，实现沿线路查询站的函数，设置当前时间函数，根据当前时间匹配列车时刻表函数，dijkstra 算法求最短距离路径，dijkstra 算法求最短时间路径以及 dijkstra 求最少换乘次数路径，以及导航询问函数等。

计算函数模块是将根据路程换算票价的函数，根据时间计算拥挤度的函数，根据拥挤度计算时间系数的函数，根据路径和当前时间计算路径的加权拥挤度的函数整合到一个模块，完成对相关数据的计算。

信息安全模块是将用户输入的所有的内容导出到一个 txt 中，并额外用 python 脚本写 md5 将其加密，保护用户的输入信息。

系统框架图如下：

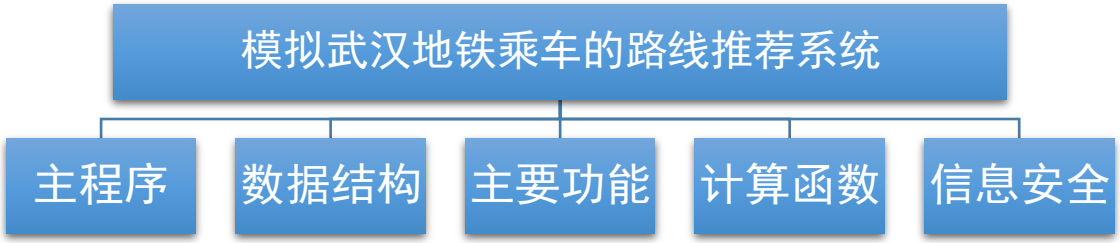

图 2-1 系统主要程序模块结构图

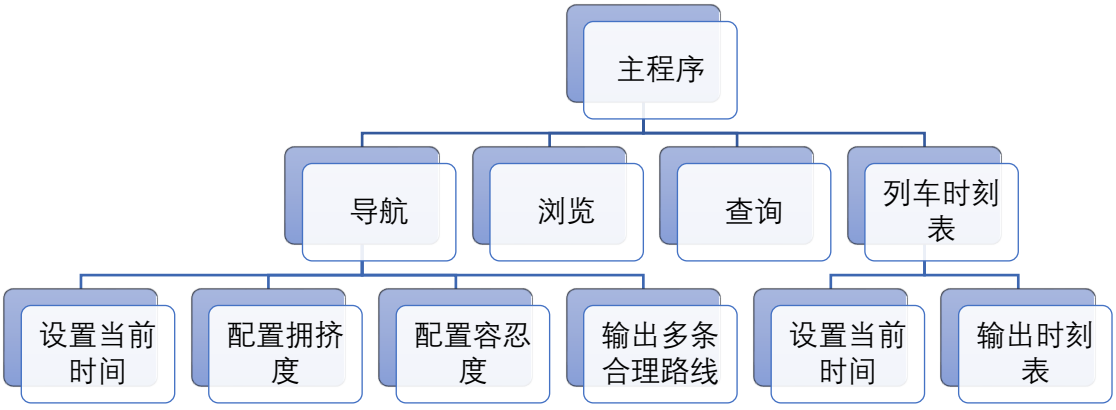

图 2-2 程序的功能框图

三、数据结构设计

本程序实现功能主要用到的数据结构是图（Graph），向量（Vector），数组（Array），哈希表（HashMap），栈（Stack）。

图的存储方式由邻接表与邻接矩阵实现。

图的邻接表包括表头节点和邻接边表，表头节点包含两个域，分别是信息域和指针域，信息域储存站名、所在线路、是否可以换乘，指针域储存 ArcNode 类型指针，指向第一个连接该节点的边。邻接边表的节点由两部分组成，分别是信息域和指针域。信息域储存邻接点在数组 NodeList 的位置下标、该边所在线路和两站间平均长度和该边属于哪条线，指针域储存指向下一个邻接点的指针，具体如下图所示：

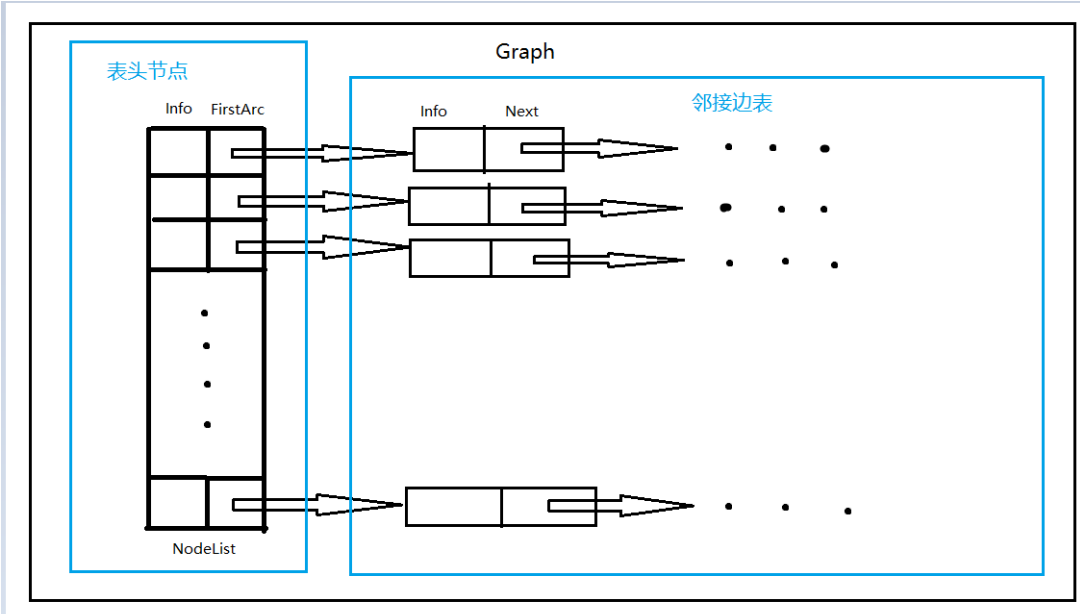

图 3-1 图的邻接表储存结构示意图

具体实现代码如下：

```
/*-----------数据结构-----------*/
// 边表
typedef struct ArcNode
{
    // 邻接点在数组中的位置下标
    int adjVex;
    // 指向下一个邻接点的指针
    struct ArcNode* next;
    //该边属于哪条线
    int line;
    //两站间平均长度
    double distance;

}ArcNode;


//顶点表
typedef struct VerNode
{
    //站名
    string name;
    //所在线路
    int atLine;
    //是否可以换乘
    bool canChange;
    // 顶点的第一条弧
    struct ArcNode* firstArc;
}VerNode;

//图
typedef struct Graph
{
    // 顶点数组
    struct VerNode NodeList[MAX_NUM];
    // 总顶点数
    int totalVertex;
    // 总边数
    int totalArc;
}Graph;
```

图 3-2 图的邻接表储存结构代码

此外，除了邻接表，本程序还用了邻接矩阵储存图，以二维数组的

定义 NodeList 中的数组下标为每个站点的 id，则本程序大部分所用的数组（除了储存邻接矩阵的）都与每个站点的 id 有关，比如 dijkstra 算法中的 d 数组，d[i]表示 id 为 i 的站距离起始点的最小距离。

本程序所用到的向量大部分当作动态数组使用，除了一个特殊的二维向量。在本系统中，不同线上的同一个站当作多个站处理，它们的 id 不同，本程序用一个二维向量储存它们，第二维储存具有相同站名的站的 id，所以如果这一维向量的大小大于 1，则表示该站是换乘站，第一维是所有二维向量的集合，第一维的大小即为所有站的数量。结构如[[a],[b,c],[d],…]表示 a 和 d 不是换乘站，b 和 c 是一个站且为换乘站，但 b 和 c 处在不同的线上，当作两个站。

哈希表是键值对一一对应结构，本程序仅仅用哈希表对程序的 id 和名字做简单的对应处理，方便一些操作，不再赘述。

本程序用到的栈，用来储存由 dijkstra 算法找到的路径，栈顶为路径起点，栈底为路径终点，结构大致如图：

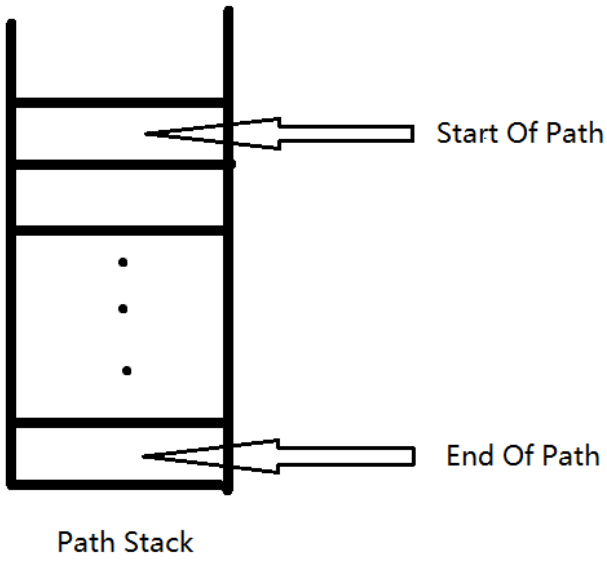

图 3-3 栈的储存结构

四、详细设计

本程序所有求解路径的算法均为 dijkstra 算法，这里重点阐述 dijkstra 算法。

算法思想：设 G=(V,E)是一个带权有向图，把图中顶点集合 V 分为两组，第一组为已求出最短路径的顶点集合（用 S 表示，初始时 S 中只有一个源点，以后每求得一条最短路径 ，就将加入到集合 S 中，直到全部顶点都加入到 S 中，算法就结束了）,第二组为其余未确定最短路径的顶点集合（用 U 表示）,按最短路径的的递增次序依次把第二组中的顶点加入 S 中。在加入的过程中，总保持从源点 v 到 S 中各个顶点的最短路径长度不大于从源点 v 到 U 中任何路径的长度。此外，每个顶点对应一个距离，S 中的顶点的距离就是从 v 到此顶点的最短路径长度，U 中的顶点的距离，是从 v 到此顶点只包括 S 中的顶点为中间顶点的当前路径的最短长度。

算法步骤：

（1）初始时，只包括源点，即 S = {v}，v 的距离为 0。U 包含除 v 以外的其他顶点，即：U ={其余顶点}，若 v 与 U 中顶点 u 有边，则(u,v)为正常权值，若 u 不是 v 的出边邻接点,则(u,v)权值 ∞（程序中用 INT16_MAX 即 32767 代替）;

（2）从 U 中选取一个距离 v 最小的顶点 k，把 k，加入 S 中（该选定的距离就是 v 到 k 的最短路径长度）。

（3）以 k 为新考虑的中间点，修改 U 中各顶点的距离；若从源点 v 到顶点 u 的距离（经过顶点 k）比原来距离（不经过顶点 k）短，则修改顶点 u 的距离

值，修改后的距离值的顶点 k 的距离加上边上的权。即 d[u] = min{d[u],d[k] + matrix[k][u]}，其中 matrix 是图的距离邻接矩阵。

（4）重复步骤（2）和（3）直到所有顶点都包含在 S 中。

在本程序中，由于不是遍历图找到非起始点的所有点到起始点的最短距离，而是确定了终点，则只需找到终点到起始点的最短距离后结束算法即可。

流程图如下：

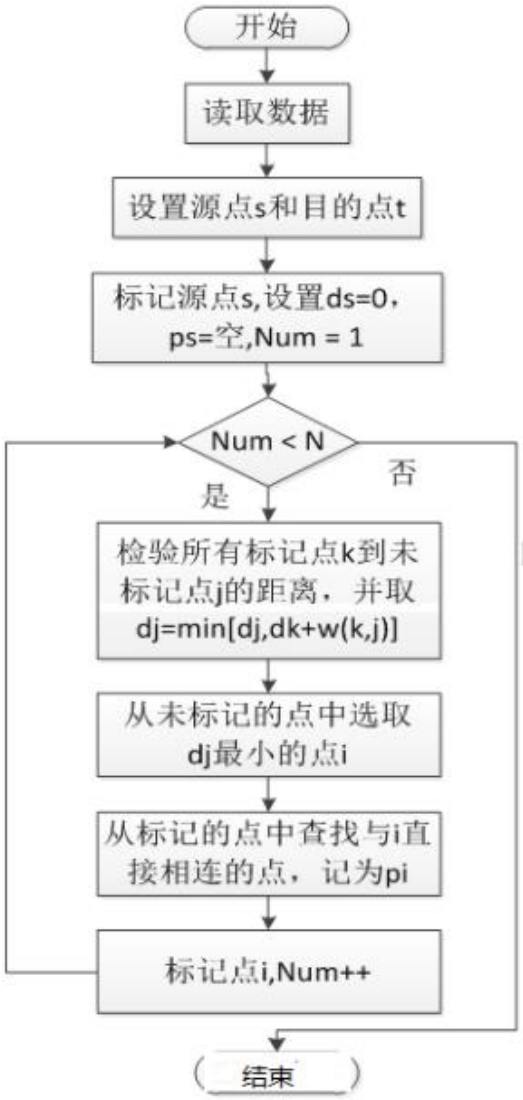

图 4-1 dijkstra 算法的流程图

代码如图所示：

```
void dijkstra(Graph g,int start,int end){
    //顶点最大个数
    int n = MAX_NUM;
    // 判断是否已存入该点到S集合中 为真则表示该点已经确立了距离起始点的最小值
    bool s[n];
    for(int i = 0; i < n;i++){
        //初始化d数组为两点之间的距离
        d[i] = matrix[start][i];
        // 初始都未用过该点
        s[i] = false;
        //将与初始点连接的站点的pre设为初始点 若不连接 设为-1
        if(d[i] == INT16_MAX){
            pre[i] = -1;
        }else{
            pre[i] = start;
        }
    }
    d[start] = 0;
    s[start] = true;
    //除了初始点剩余n-1个点 循环n-1次
    for (int i = 0; i < n - 1 ; ++i) {
        //通过minD寻找与每一次循环与初始点距离最近的点 即min{d[i]}
        int minD = INT16_MAX;
        // 找出当前未使用的点j的dist[j]最小值
        int u = start;
        for(int j = 0; j < n; j++){
            if(!s[j] && d[j] < minD){
                //第i+1次循环 寻找第i+1个与初始点距离最近的点
                // u保存当前邻接点中距离最小的点的号下标
                u = j;
                minD = d[j];
            }
        }
        // 设为真
        s[u] = true;
        //对于从u出发的所有边(u,y)且没有确立距离起始点最小值, 更新d[y] = min{d[y], d[u]+w(u,y)}
        for(int j = 0; j < n; j++){
            if(!s[j] && matrix[u][j] < INT16_MAX){
                //在通过新加入的u点路径找到离v0点更短的路径
                if(d[u] + matrix[u][j] < d[j]){
                    //更新dist
                    d[j] = d[u] + matrix[u][j];
                    //记录前驱顶点
                    pre[j] = u;
                }
            }
        }
    }
```

图 4-2 dijkstra 算法的源代码

本程序在不考虑拥挤度时，由于选择了 dijkstra 算法，选择推荐在合理时间范围内的三条路径为：一条时间最短的路径，一条路径最短的路径，一条换乘次数最少的路径，按票价由低到高排序输出。在推荐时间最短路径时，将距离邻接矩阵 matrix 改为时间邻接矩阵 timeMatrix 即可，若 i 和 j 不是一个站，即判断站名不相等，则 timeMatrix[i][j]＝3（停车 1 分钟+行车 2 分钟），若 i 和 j 是不同线的同一个站，即是换乘站，则 timeMatrix[i][j]＝3.5，而推荐换乘次数最少路径时，只需考虑将换乘站的时间邻接矩阵的值调至相对 3.5 很大的数，增加换乘成本，则可以找到合理时间范围内的最少换乘次数路径。

加权计算部分：

本程序在考虑拥挤度计算最短加权时间时，选择将 timeMatrix 进行改进，改进为 timeMatrixCrowd，除了换乘站间的时间不随拥挤度改变，其余都会随拥挤度乘一个时间系数，传入当前时间 hAndMin 参数，每走一站（即程序中 s[u] = true），选择更新当前时间，并根据更新后的当前时间更新 timeMatrixCrowd，更新方式为根据当前时间算出每一线的拥挤度，根据拥挤度算出时间系数，将 timeMatrixCrowd 的每一个非换乘站间的时间乘时间系数即可。

```
//更新timeMatrixCrowd矩阵
hAndMin += timeMatrix[preTCrowd[u]][u];
update(g);
```

图 4-3 更新时间矩阵

```
//根据当前时间hAndMin更新考虑拥挤度的时间邻接矩阵
void update(Graph g){
    for (int i = 0; i < MAX_NUM; ++i) {
        for (int j = 0; j < MAX_NUM; ++j) {
            timeMatrixCrowd[i][j] = timeMatrix[i][j];
        }
    }
    for (int i = 0; i < MAX_NUM; ++i) {
        for (int j = 0; j < MAX_NUM; ++j) {
            if(timeMatrixCrowd[i][j] != INT16_MAX){
                //换乘的时候不需要考虑拥挤度
                if(g.NodeList[i].name != g.NodeList[j].name){
                    timeMatrixCrowd[i][j] *= timeRatio(crowd(g,hAndMin,i),avoid,setedCrowd);
                }
            }
        }
    }
}
```

图 4-4 update 函数

其中 timeRatio 函数是根据拥挤度和当前时间以及是否设置容忍度算出的时间系数。

此外，图的插入函数在本程序中也十分重要。在读取文件时要进行构图，设计出图的邻接表结构后要进行插入，图的插入函数要注意因为是无向图所以在 a 插入 b 表头后面时也要 b 插入 a 表头后面。具体表现为两个函数 insertArcNode 和 insert，insertArcNode 函数是下标 index 的节点插入到表头为 pre 的节点的

邻接边表的结尾，设一个 ArcNode 类型的指针 p，动态分配内存，储存传入的数据。再设一个 ArcNode 类型的指针 q，赋值为 pre 的 firstArc，若不为空即 pre 邻接边表的首节点存在，则通过循环找到结尾，插入 p 即可，若为空即 pre 的邻接边表的首节点不存在则在首节点插入即可，即令 firstArc = p。而 insert 函数则是两个 insetArcNode 函数，完成无向图的两次插入。

此外，计算路径 s 的加权拥挤度的算法也很重要，传入参数为路径栈 s，栈顶为路径起点，以及起始时间 time，由于行驶时的拥挤度就是上车后的拥挤度，所以线的拥挤度可以转化为站的拥挤度，则加权拥挤度即为去掉终点站的所有站点的拥挤度除以站点数目（不考虑重复站），使用哈希表 map<string,int>记录已经用过的站名，主要是找到换乘站站名的上一个的 id，减去该站的拥挤度，并因为不考虑重复，让站点数目减一，具体实现代码如下：

```
double pathCrowd(Graph g,stack<int> s,double time){
    double totalCrowd = 0;
    int n = s.size();
    //使用map记录换乘站的上一个id
    map<string,int> name;
    //不考虑目的站的拥挤度
    //因为乘车时的拥挤度认为是上车时的拥挤度
    while (s.size() > 1){
        int a = s.top();
        //如果a站不是换乘站 或者a站是换乘站但是在此站下车进行换乘 拥挤度要换到本站的另一条线算 但是先在此处算上 下面else再减
        if(name.count(g.NodeList[a].name) == 0){
            totalCrowd += crowd(g,time,a);
            name.insert(pair<string,int>(g.NodeList[a].name,a));
        }
        //a站是换乘站的另一条线 在本线上车 把上一个a站的拥挤度减去 加上此时线路a站的拥挤度 再n-- 注意此时的time已经变了 要减掉换线时间
        else{
            totalCrowd -= crowd(g,time - WALK,name.find(g.NodeList[a].name)->second);
            n--;
            totalCrowd += crowd(g,time,a);
            name.insert(pair<string,int>(g.NodeList[a].name,a));
        }
        s.pop();
        int b = s.top();
        time += timeMatrix[a][b];
    }
    return totalCrowd / (n - 1);
}
```

图 4-5 求路径加权拥挤度的算法

边界处理部分：

本程序关于时间的设置做了很多简化处理，在若用户输入的时间在早班车发车时间或末班车时间边界外，则提示用户重新输入，本程序针对“运营时间”为6~23 点的条件，设定末班车在 23 点前发出，且在 23 点前到达终点站。

本程序对于最后一班车的设定是，由于由 dijkstra 算法求得最短时间路径，根据用户输入的当前时间 hAndMin，判断 hAndMin + 最短时间路径所用时间与23：00 的关系，若大于 23：00，则不可到达，否则可以由到达路径。

换乘步行时间为 3.5 分钟，定义列车每 5 分钟发一趟，计算出头班车的到该站的时间，之后每隔五分钟都会有一辆，所以判断用户步行过去后的时间减去头班车到该站的时间，然后模 5，如果结果在 0~1 之间，则可以上车，否则还需要等待 5-该模 5 结果分钟的时间。

五、系统实现

开发环境：JetBrains CLion 2019.2.3 x64

系统: Windows 10 x64 Home

Cmake 最低版本要求：3.15

C++标准：C++14

支持包：C++ STL，hashlib（python 用）

数据结构模块：

```
//图初始化 对图g赋基本值
void initGraph(Graph *g);

// id为pre的表头的邻接边表后插入节点id为index节点
// 并为邻接边表的信息域赋值两站间平均距离distance和所在线line
void insertArcNode(Graph *g, int pre, int index, double distance,int line);

// 图的插入函数，如果两个节点邻接，将彼此插入对方的邻接边表
void insert(Graph *g, int index1, int index2, double distance,int line);
```

调用关系：insert 函数调用 insertArcNode 函数

主要功能模块：

```
//将线路信息、站点信息的数据文件读入系统，并进行构图g
void read(Graph *g);
//所用全局变量
//每一号线的不同时间的拥挤度
double crowds[7][3];
//去除重复的总站数
int total;
//储存最短长度路径的栈
stack<int> sDis;
```

```
//储存最短时间路径的栈
stack<int> sTime;
stack<int> sTimeCrowd;
//储存最少换线次数路径的栈
stack<int> sChange;
//根据站的序号找到名字 读取文件用
map<int,string> nameMap;
//储存站的序号的二维数组 同名的站序号放在一个一维数组里 读取文件用
vector<vector<int>> numberV;
//图的距离邻接矩阵
double matrix[MAX_NUM][MAX_NUM];
//图的时间邻接矩阵
double timeMatrix[MAX_NUM][MAX_NUM];
//图的考虑拥挤度的时间邻接矩阵
double timeMatrixCrowd[MAX_NUM][MAX_NUM];
//图的换乘邻接矩阵（时间邻接矩阵换乘站3.5变为20）
double changeMatrix[MAX_NUM][MAX_NUM];
```

调用关系：read 函数调用 insert 函数

```
//把指定线路的站点依次显示出来，用户输入线路，打印出g中相应储存的信息
void print(Graph g);

/***
 * 指定线路、起始站点 并从该站点开始逐步找出下一站点
 * 或在换乘站点进行换线 实现对地铁路线站点的依次浏览
 * */
void nextOrChange(Graph g);

//设置当前时间
void setTime();
//所用到的全局变量
//时
double hour;
//分
```

```
double minute;

//时加分全部转化为分
double hAndMin;


//匹配当前时刻的前后多少分的几号线的进站和出站时刻
void matchTime(Graph g);
```

调用关系：matchTime 函数调用 setTime 函数。

```
//dijkstra算法求最短长度的路径  详细设计中已经进行了详细阐述
void dijkstra(Graph g,int start,int end);

//用到的全局变量

//每一点到起始点的最短距离(dijkstra用)
double d[MAX_NUM];

//dijkstra用的下标数组
int pre[MAX_NUM];

//最短路径票价
int priceDis;

//储存最短长度路径的栈
stack<int> sDis;


//求最短时间的路径
void dijkstraTime(Graph g,int start,int end);

//用到的全局变量

//每一点到起始点的最短时间(dijkstraTime用)
double t[MAX_NUM];

//dijkstraTime用的下标数组
int preT[MAX_NUM];

//最短时间票价
int priceTime;

//储存最短时间路径的栈
stack<int> sTime;


//求换乘次数最少的路径
void dijkstraChange(Graph g,int start,int end);

//用到的全局变量

//dijkstraChange用的下标数组
```

```
int preC[MAX_NUM];
//最少换乘次数票价
int priceChange;
//储存最少换线次数路径的栈
stack<int> sChange;
//每一点到起始点的最短距离(dijkstraChange用)
double dc[MAX_NUM];

//根据当前时间hAndMin更新考虑拥挤度的时间邻接矩阵
void update(Graph g);
//求最短加权时间的路径
void dijkstraTimeCrowd(Graph g,int start,int end);
//用到的全局变量
//每一点到起始点的最短时间(dijkstraTimeCrowd用)
double tCrowd[MAX_NUM];
//下标数组
int preTCrowd[MAX_NUM];
//最短加权时间票价
int priceTimeCrowd;
//储存最短加权时间路径的栈
stack<int> sTimeCrowd;

//导航询问系统
void navigation(Graph g);
```

调用关系：navigation 函数调用 dijkstra 函数、dijkstraTime 函数、dijkstraTimeCrowd 函数和 dijkstraChange 函数。

主程序模块：

```
//程序入口，统筹其他模块

int main();
```

调用关系：main 函数调用 changeOrNext 函数、matchTime 函数、navigation 函数和 print 函数。

计算函数模块：

```
/*票价：按里程分段计价
    4 公里以内（含 4 公里）2 元；4 - 12 公里（含 12 公里），1 元 / 4 公里；
    12 - 24 公里（含 24 公里），1 元 / 6 公里；
    24 - 40 公里（含 40 公里），1 元 / 8 公里；
    40 - 50 公里（含 50 公里），1 元 / 10 公里;*/
int price(double distance);


//计算id为index的站在time时间的拥挤度
double crowd(Graph g,double time,int index);


//综合各站点间地铁车厢人员拥挤情况，得到整个乘车期间的拥挤程度；
//路径储存于栈s中 栈顶为起点
double pathCrowd(Graph g,stack<int> s,double time);


/*
 * 一条地铁线路的拥挤程度在某段时间内认为是相同的，20%以下为宽松，
 * 可以找到座位，50%为一般拥挤程度，75%为拥挤，不能超过 100%；
 * 约定宽松情况下，乘车时间系数为 0.6，拥挤情况下时间系数为 1.5，
 * 一般拥挤程度为1.2，宽松以上但不到一般拥挤为0.8
 */
//根据拥挤度返回时间系数
double timeRatio(double crowd,bool avoid,double setCrowd);
```

调用关系：主功能模块的相应函数调用 price 函数、crowd 函数、pathCrowd 函数和 timeRatio 函数。

六、运行测试与结果分析

一、菜单显示

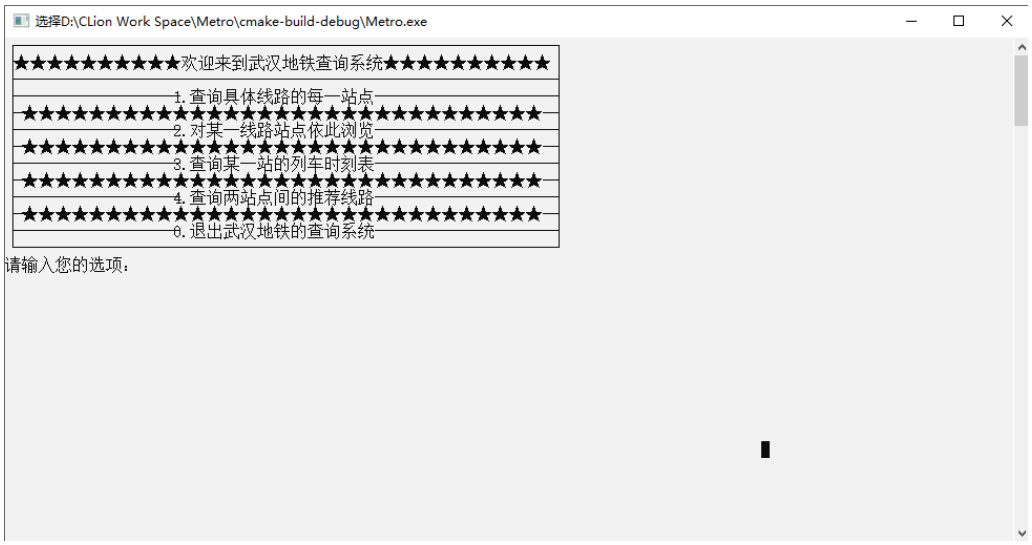

图 6-1 菜单显示

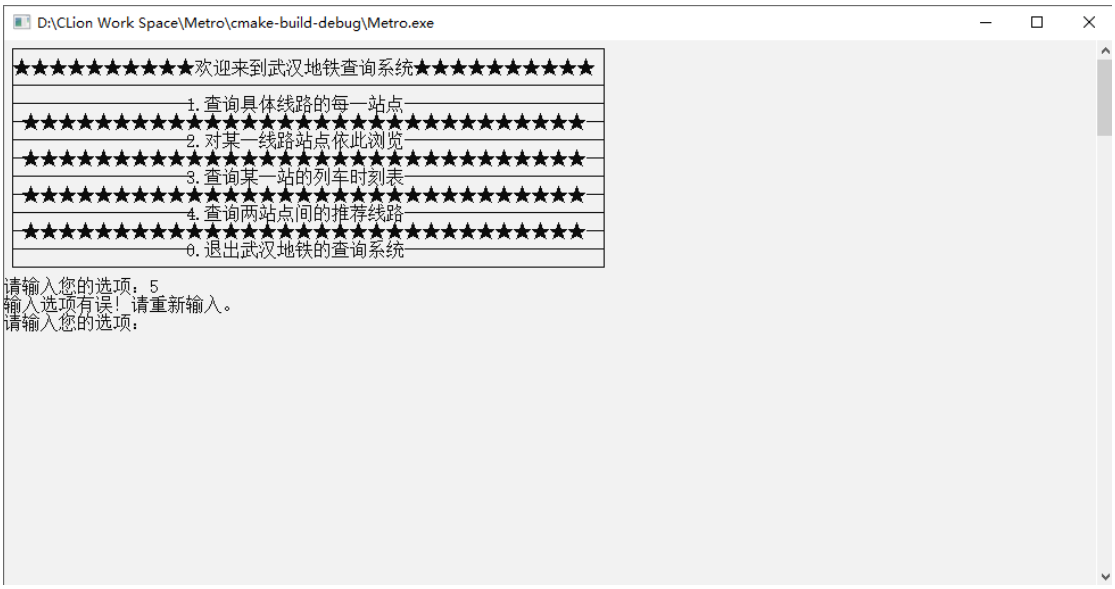

图 6-2 菜单显示（纠错）

本菜单大致实现美观效果，提示用户输入的信息详细，并且有输入纠错能力，可以达到标准的菜单效果。

二、查询功能

图 6-3 查询 2 号线（带纠错能力）

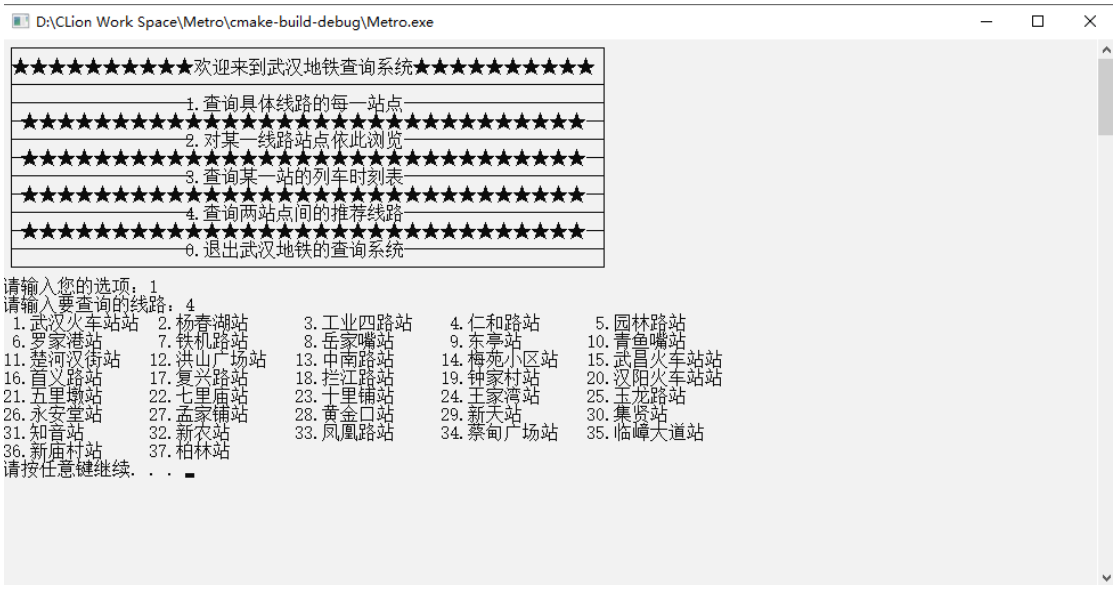

图 6-4 查询 4 号线

本功能能格式化输出每一条线路的站点，且做到顺次输出，准确输出，并且有一定的纠错能力，可以达到标准的查询路线站点的效果。

三、浏览功能

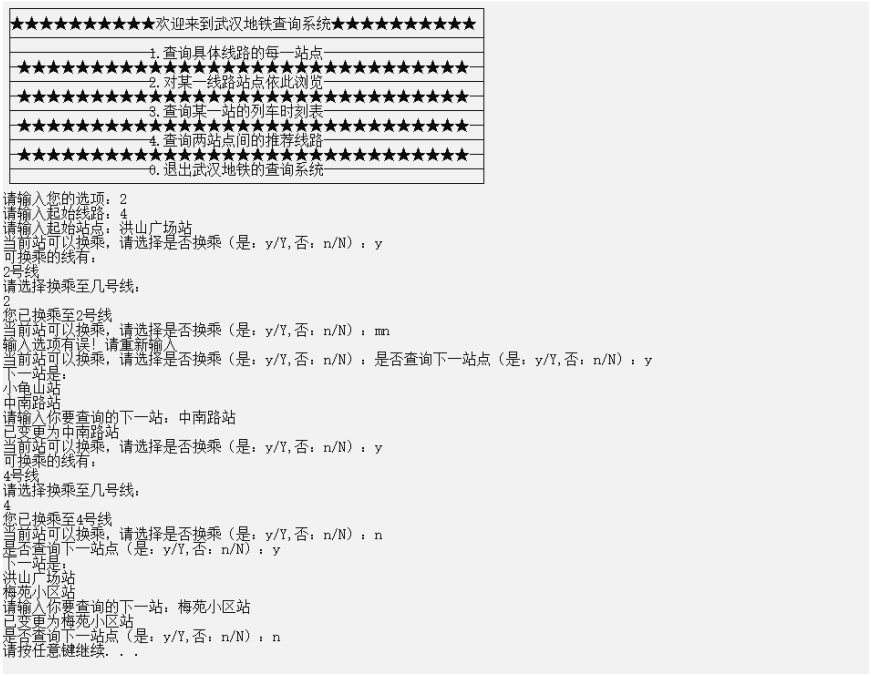

图 6-5 浏览功能实现换乘与非换乘站的浏览

本功能可以实现沿路线站点的依次浏览，且可以实现到达换乘站时进行换成到下一条线，并继续浏览沿新路线的站点，达到了设计的基本要求。

四、列车时刻表

图 6-6 列车时刻表功能

列车时刻表功能可以查询具体到某一线路的某一站当前时间的某一时间范围内的两个方向的离开和即将到达的列车的时刻，基本达到了设计要求。

五、导航

图 6-7 不考虑拥挤度时的导航功能（1）

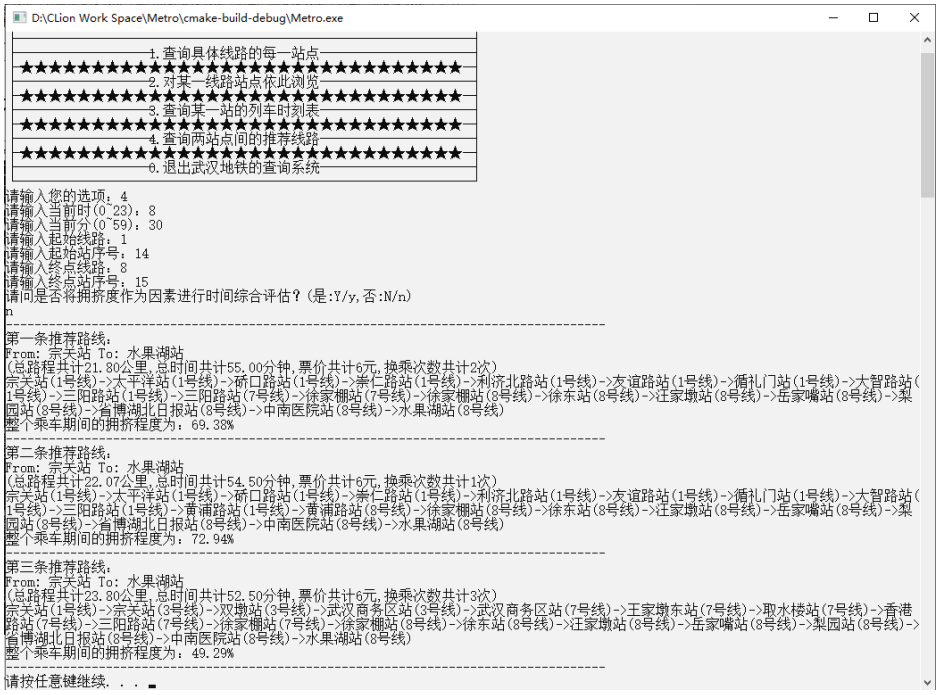

图 6-8 不考虑拥挤度时的导航功能（2）

图 6-9 考虑边界问题

在不考虑拥挤度时，本程序选择推荐三条路线：最短时间路线，最短路程路线，最少换乘次数路线，并计算出每一条路线的长度，所用时间，换乘次数，以及整个乘车期间的加权拥挤度，并按照票价由低到高的方式输出，基本达到了设计要求。

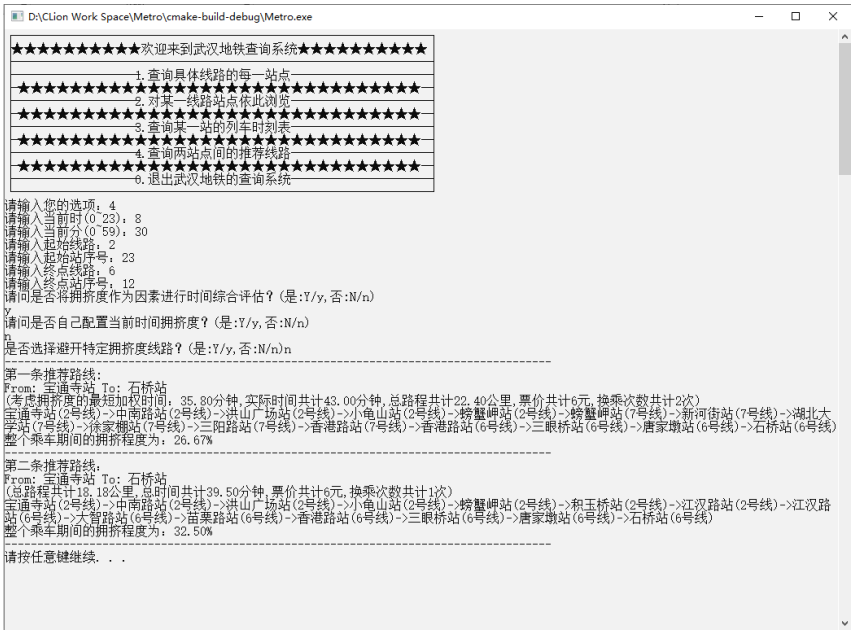

图 6-10 考虑拥挤度时的导航功能（1）

图 6-11 考虑拥挤度时的导航功能（2）

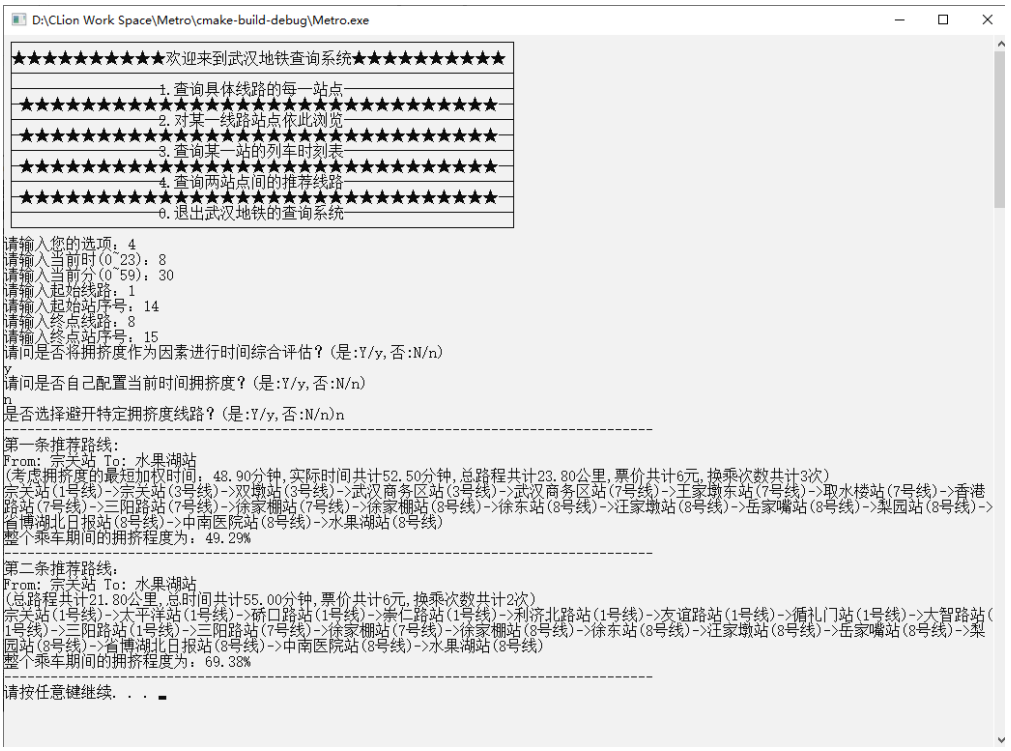

图 6-12 考虑拥挤度时的导航功能（3）

本程序在考虑拥挤度时设置了开关，由用户自己选择是否考虑，推荐第一条为最短加权时间路径，第二条为合理加权时间范围内最短长度路径，并且每条路线都输出实际花费时间，票价以及路径长度和加权拥挤度，基本达到设计要求。

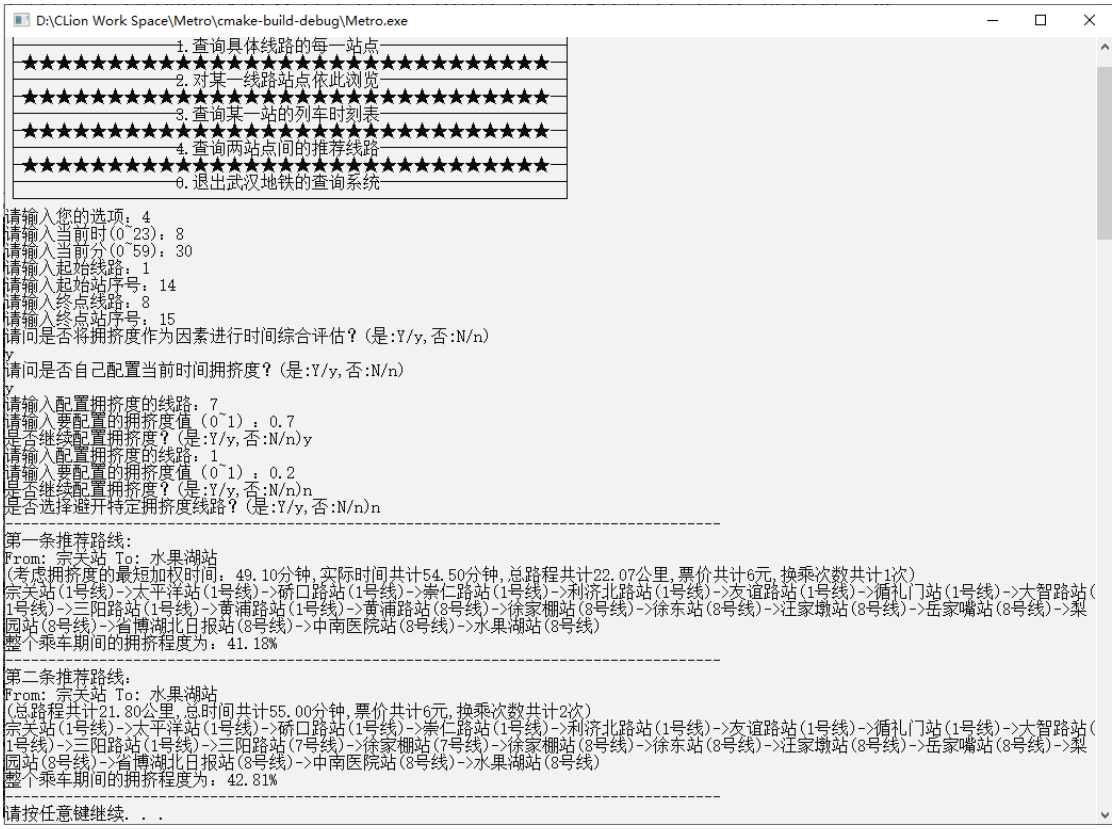

图 6-13 配置拥挤度时的导航功能（1）

配置 7 号线拥挤度为 0.7，1 号线拥挤度 0.2，1 号线的宗关到 8 号线的水果湖的路线推荐发生变化，最短加权时间变为 49.1 分钟，路线推荐变为一号线宗关坐到黄浦路换乘 8 号线再坐到水果湖。本程序可以做到配置多条路线拥挤度，并且实现基本功能，基本达到设计要求。

下面考虑 1 号线径河站到八号线野芷湖站，最短加权时间路线经历七号线较多，观察到走七号线站点较多，后续更改七号线拥挤度，更改七号线拥挤度为 0.8，最短加权时间路径发生变化，变为走六号线转二号线再转八号线。

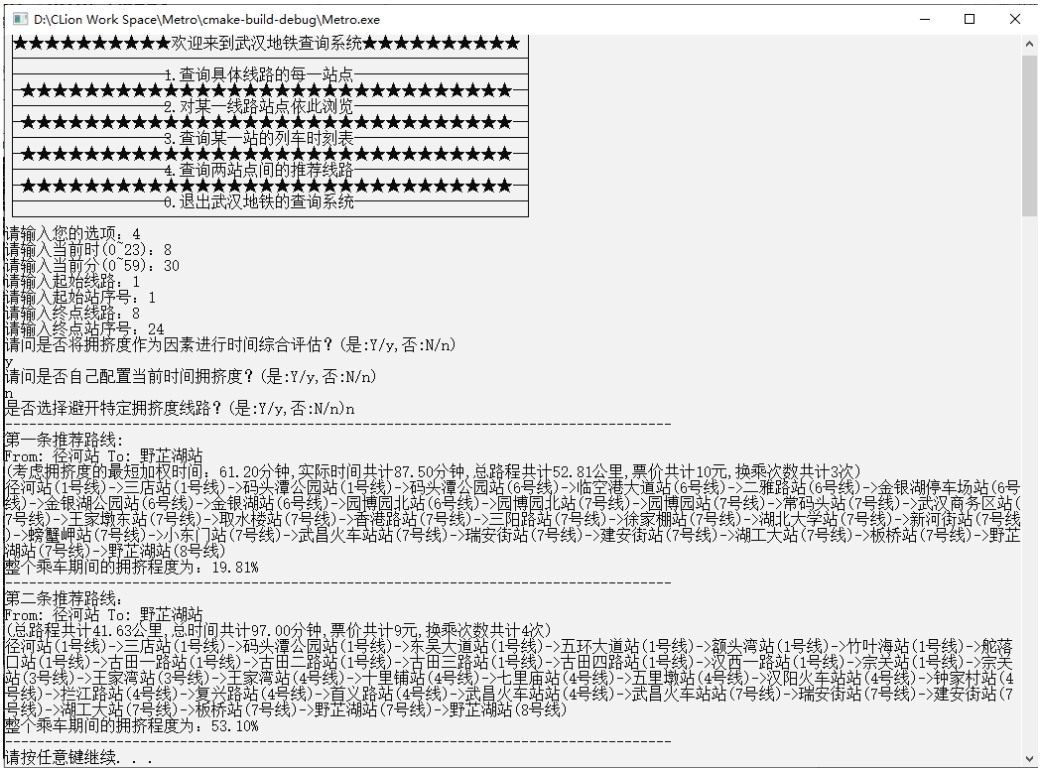

图 6-14 配置拥挤度时的导航功能（2）

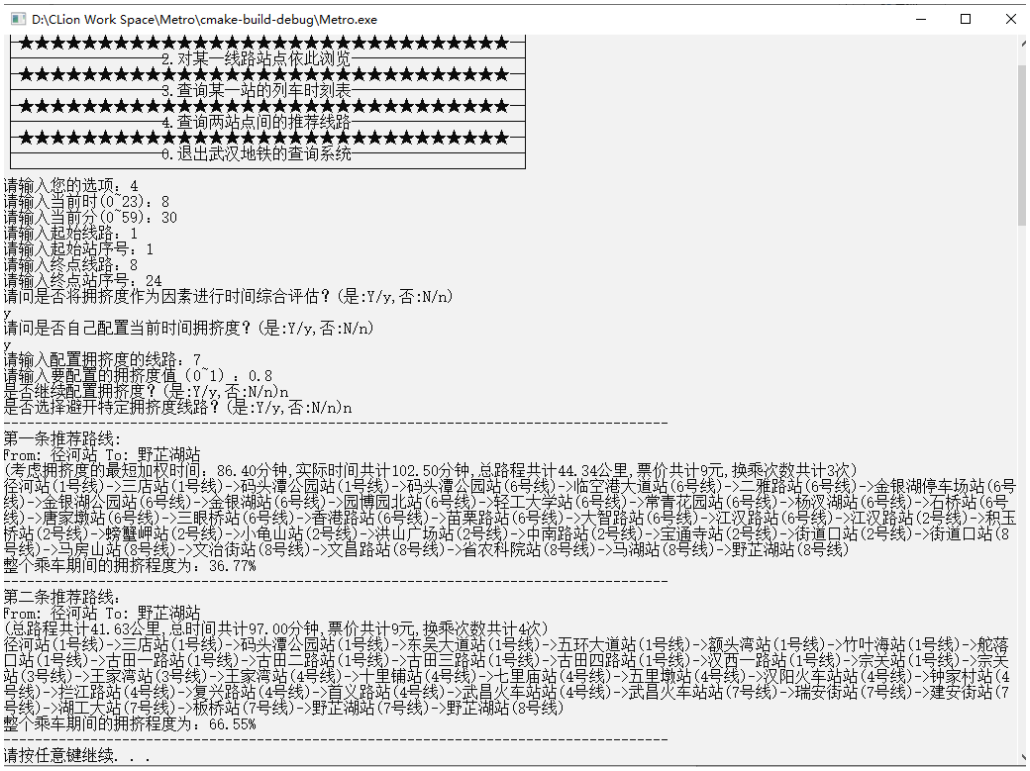

图 6-15 配置拥挤度时的导航功能（3）

同时，提供容忍度配置，2 号线天河机场到 8 号线洪山路，没有配置拥挤度容忍度时最短加权时间时 59.2 分钟，经历了 7 号线较多，后续更改 7 号线拥挤度，配置 7 号线拥挤度 0.8，容忍度 0.75，发现第一条推荐路线不再经过七号线。

图 6-16 配置拥挤度时的导航功能（1）

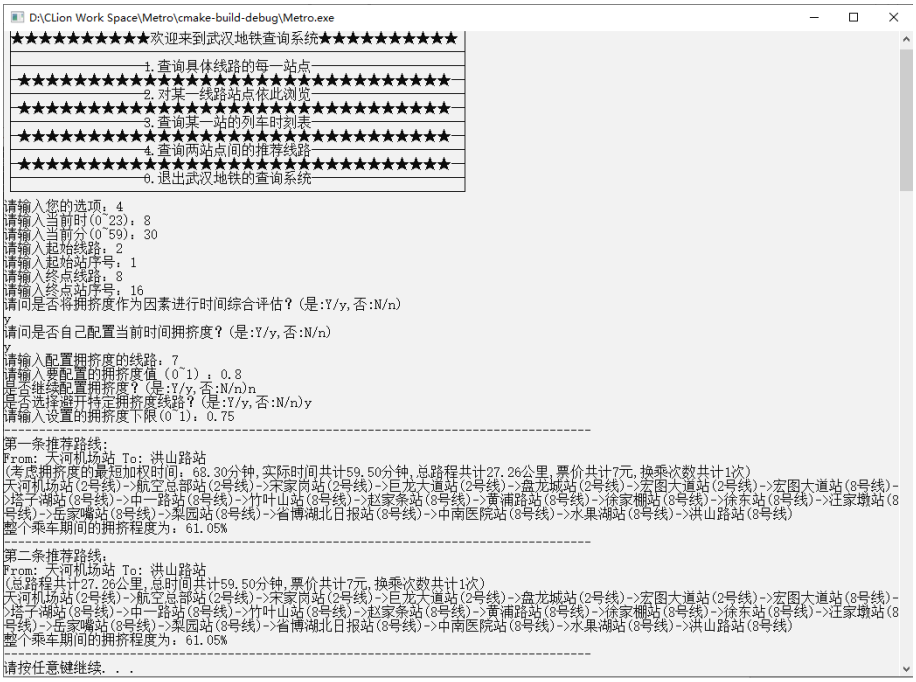

图 6-17 配置拥挤度时的导航功能（2）

图 6-18 信息安全方面的应用

由于武汉地铁数据为公开数据，加密它没有意义，选择加密用户输入的各种信息，采用 python 编写脚本，进行 md5 加密，大致达到了预期效果。

复杂度分析：

时间复杂度：

本程序所采用的主要算法为 dijkstra 算法，所以时间复杂度为 $O(n^2)$，其余函数中的一些初始化操作等语句的时间复杂度都为 $O(n^2)$或 $O(n)$。

空间复杂度：

本程序所用空间复杂度较高，每个 dijkstra 算法都开辟了二维数组，且部分函数额外开辟了 Map 和 Vector 进行辅助，在找路径时，开辟 Stack 储存每一条路径，储存图的数据结构采用邻接表储存，读取和输出的文件等等，都造成了较大的空间复杂度，比如单一个 update 函数每次循环都要对 280*280 的二维数组进行赋值操作，不得不承认空间开销过大是本程序的一大缺点，本次将充分吸取教训，在未来的程序设计中注意空间开销这一点，做到尽量少的使用空间。

七、总结

首先，不得不说自己在工程思维上有了突破，之前不管是写 C/C++还是 Java 语言，无论是小程序还是相对较大的程序，都是一个 cpp，一个文件从头到尾结束，最多将很多抽象出来的函数分开写，从来没有写过抽象出多个模块，写多个文件的工程，在这里的工程思维是确实得到升华的。

但是，自己的工程思维只是很初步的，因为在写本次设计时，很多数据已经做了简化，但是自己的程序还是会有几个的不符合要求的地方，那么其实在真正的工程中肯定是不能做这么多简化的，所以借此机会要严格要求自己，切实进一步提高工程架构能力，其次要提升自己的调试能力，在本次程序设计过程中，由于变量，函数数目过多，在产生 bug 时，自己设置断点的调试水平很差，只能一步步看代码去解决 bug，效率很低，还有一点是自己的测试能力不好，测试岗位也是很重要的岗位，如何选择“怪异”的测试点去测试你的程序也是技术活，自己在调试程序时采用的测试样例没有出问题，但是跑别人的测试样例时发现了问题，就说明自己在测试程序时没有精准把控程序的边界点，测试水平有待提高。

低水平的调试能力也间接导致了写程序时的算法选取，其实我本来选择了深度优先搜索（DFS）算法来找最短路径，这样应该也能够实现 K 短路，但是自己写出的深度优先搜索算法无论是直接递归还是调用栈来实现，都会产生程序运行出错或者输出结果与预期结果相差很大等问题，找了很久也没有找到解决问题的办法，我不得不放弃深度优先搜索算法尝试别的思路，而我也没有正确理解 K 短

路的设计要求，选择了我熟悉的 dijkstra 算法，求多个标准下的最优路，后续我会认真学习 K 短路算法。

通过这次实验，我掌握了 C++的文件读取，STL 的基本使用，达到了熟练运用 dijkstra 算法的水平，算是很大的收获。但是不得不说本次课程设计碰壁和失败或者失误更多，除了上述教训外还有空间的使用不当导致开销过大的问题，以及我重复代码过多导致代码复用的问题，很明显，我有多个类 dijkstra 函数，好多代码是一样的，那么代码复用的问题很严重，本次一定吸取教训，做到代码最简化。通过这次课程设计我也基本了解了 python 语言的使用，导入包，以及 python 的各种容器，流程控制应用，以及便捷的文件操作，此处收获较多。

非常感谢本次课程设计愿意与我一起讨论的多位同学，是他们提供给我很多思路，很多算法和数据结构设计吸取的他们的经验，很大程度上缩短了我写程序的时间，以后我也会积极与同学们讨论，学习，提升自己。即使是在吸取他人经验的前提下，我也是用了第一周除了上课外的所有时间，基本是除了上课就是在寝室写课程设计，过程很苦，确实体会到了科研人员的不易，虽然我们做的不是一种东西，但都是全身心投入到一个项目中去，用心钻研。

再次衷心地感谢本次课程设计中帮助我的老师，同学！通过本次课设着实有很大提升，希望未来能够有更多机会锻炼自己，不断成长。

附录一：参考文献

[1] 浅谈最短路中的 Dijskra 算法：http://www.voidcn.com/article/p-oeedleog-bms.html

[2] 严蔚敏等.数据结构（C 语言版）. 清华大学出版社

[3] Jacek Galowicz 著.C++17 STL Cookbook

附录二：主要程序片段

一、特色说明

1. 虽然程序采用控制台输出，但每一条功能的输出以及菜单栏的输出都尽可能地做到美观整洁，排版合理。
2. 功能覆盖到全部地设计要求，每个功能分界合理，且导航功能内部的配置拥挤度，配置容忍度，是否考虑拥挤度等均设有开关选项，且可以选择是否配置多条线路的拥挤度，以及每个让用户输入的内容均做到了纠错功能，如果用户输入非法选项，均可以重新输入。
3. 本程序在不考虑拥挤度时提供了三条路线，分别是最短时间路线，最短距离路线以及最少换乘次数路线，考虑到对不同的人考虑不同标准进行最优的推荐，在考虑拥挤度时，除了推荐最短加权时间路径，也给出了合理加权时间范围内的最短距离路径。
4. 本程序的信息安全方面的应用考虑到武汉地铁数据为公开数据，没有加密的必要，选择对用户输入的所有信息进行加密，保护用户隐私。

二、主要程序片段

```
/*-----------graph.h-----------*/
/*-----------数据结构-----------*/
// 边表
typedef struct ArcNode
{
    // 邻接点在数组中的位置下标
    int adjVex;
    // 指向下一个邻接点的指针
```

```
    struct ArcNode* next;
    //该边属于哪条线
    int line;
    //两站间平均长度
    double distance;

}ArcNode;



//顶点表
typedef struct VerNode
{
    //站名
    string name;
    //所在线路
    int atLine;
    //是否可以换乘
    bool canChange;
    // 顶点的第一条弧
    struct ArcNode* firstArc;
}VerNode;



//图
typedef struct Graph
{
    // 顶点数组
    struct VerNode NodeList[MAX_NUM];
    // 总顶点数
    int totalVertex;
    // 总边数
    int totalArc;
```

```
}Graph;
// Number为pre的邻接边表后插入节点index
void insertArcNode(Graph *g, int pre, int index, double distance,int line) {
    if (g && pre < MAX_NUM && index < MAX_NUM) {
        ArcNode* p,* q;
        //动态分配内存
        p = (ArcNode*)malloc(sizeof(ArcNode));
        //动态分配内存失败
        if (!p) {
            return;
        }
        p->adjVex = index;
        p->line = line;
        p->distance = distance;
        p->next = nullptr;
        //开始插入节点
        q = g->NodeList[pre].firstArc;
        //如果首节点存在,找到结尾，然后插入
        if (q) {
            while (q->next) {
                q = q->next;
            }
            q->next = p;
        }
            //如果首节点不存在，在首节点位置插入
        else {
            g->NodeList[pre].firstArc = p;
        }
        g->NodeList[pre].count++;

    }
}
```

```
// 图的插入函数，如果两个节点邻接，将彼此插入对方的邻接边表
void insert(Graph *g, int index1, int index2, double distance,int line) {
    if (g && index1 < MAX_NUM && index2 < MAX_NUM) {
        matrix[index1][index2] = distance;
        matrix[index2][index1] = distance;
        timeMatrix[index1][index2] = DRIVE;
        timeMatrix[index2][index1] = DRIVE;
        insertArcNode(g, index1, index2, distance, line);
        insertArcNode(g, index2, index1, distance, line);
        g->totalArc++;
    }
}
/*-----------mainfunction.h-----------*/
//把指定线路的站点依次显示出来
void print(Graph g){
    int line;
    while(true) {
        cout << "请输入要查询的线路：";
        cin >> line;
        ofs<<line<<endl;
        if(existLine(line)){
            break;
        }else{
            cout<<"输入线路有误！请重新输入。"<<endl;
        }
    }
    for (int i = 1; i <= stations[LineToIndex(line)]; ++i) {
        cout << setw(2) <<i<< "." << g.NodeList[Number(LineToIndex(line), i -
1)].name<<"\t";
```

```
        if (i % 5 == 0) {
            cout << endl;
        }
    }
    cout<<endl;
}
//求最短长度的路径
void dijkstra(Graph g,int start,int end){
    //顶点最大个数
    int n = MAX_NUM;
    // 判断是否已存入该点到S集合中 为真则表示该点已经确立了距离起始点的最小值
    bool s[n];
    for(int i = 0; i < n;i++){
        //初始化d数组为两点之间的距离
        d[i] = matrix[start][i];
        // 初始都未用过该点
        s[i] = false;
        //将与初始点连接的站点的pre设为初始点 若不连接 设为-1
        if(d[i] == INT16_MAX){
            pre[i] = -1;
        }else{
            pre[i] = start;
        }
    }
    d[start] = 0;
    s[start] = true;
    //除了初始点剩余n-1个点 循环n-1次
    for (int i = 0; i < n - 1 ; ++i) {
        //通过minD寻找与每一次循环与初始点距离最近的点 即min{d[i]}
        int minD = INT16_MAX;
        // 找出当前未使用的点j的dist[j]最小值
```

```
        int u = start;
        for(int j = 0; j < n; j++){
            if(!s[j] && d[j] < minD){
                //第i+1次循环 寻找第i+1个与初始点距离最近的点
                // u保存当前邻接点中距离最小的点的号下标
                u = j;
                minD = d[j];
            }
        }
        // 设为真
        s[u] = true;
        //对于从u出发的所有边(u,y)且没有确立距离起始点最小值，更新d[y] = min{d[y],
d[u]+w(u,y)}
        for(int j = 0; j < n; j++){
            if(!s[j] && matrix[u][j] < INT16_MAX){
                //在通过新加入的u点路径找到离v0点更短的路径
                if(d[u] + matrix[u][j] < d[j]){
                    //更新dist
                    d[j] = d[u] + matrix[u][j];
                    //记录前驱顶点
                    pre[j] = u;
                }
            }
        }
    }
    int a = end;
    while(a != start){
        sDis.push(a);
        a = pre[a];
    }
    sDis.push(start);
    priceDis = price(d[end]);
```

```
}
//打印最短长度的路径
void printDis(Graph g,int start,int end){
    int a = end;
    int changeTime = 0;
    double pathTime = 0;
    while(a != start){
        pathTime += timeMatrix[pre[a]][a];
        if(g.NodeList[a].name == g.NodeList[pre[a]].name){
            changeTime++;
        }
        a = pre[a];
    }
    if(hAndMin + pathTime < 23 * 60){
        cout << "From: " << g.NodeList[start].name << " To: " <<
g.NodeList[end].name << endl << "(总路程共计"
             << setiosflags(ios::fixed) << setprecision(2) << d[end] << "公里,总
时间共计" << pathTime << "分钟,票价共计" << priceDis
             << "元,换乘次数共计" << changeTime << "次)" << endl;
        printStack(g, sDis);
        cout << "整个乘车期间的拥挤程度为: " << pathCrowd(g, sDis, hAndMin) * 100
<< "%" << endl;
    }else{
        cout << "由于时间问题，地铁停运无法到达目的地！" << endl;
    }
}
//根据当前时间hAndMin更新考虑拥挤度的时间邻接矩阵
void update(Graph g){
    for (int i = 0; i < MAX_NUM; ++i) {
        for (int j = 0; j < MAX_NUM; ++j) {
            timeMatrixCrowd[i][j] = timeMatrix[i][j];
        }
```

```
    }
    for (int i = 0; i < MAX_NUM; ++i) {
        for (int j = 0; j < MAX_NUM; ++j) {
            if(timeMatrixCrowd[i][j] != INT16_MAX){
                //换乘的时候不需要考虑拥挤度
                if(g.NodeList[i].name != g.NodeList[j].name){
                    timeMatrixCrowd[i][j] *=
timeRatio(crowd(g,hAndMin,i),avoid,setedCrowd);
                }
            }
        }
    }
}


//计算路径s的加权时间
double pathWeightedTime(Graph g,stack<int> s){
    double weightedTime = 0;
    update(g);
    while (s.size() > 1){
        int a = s.top();
        s.pop();
        int b = s.top();
        weightedTime += timeMatrixCrowd[a][b];
        hAndMin += timeMatrix[a][b];
        update(g);
    }
    return weightedTime;
}
/*-----------compute.h-----------*/
//综合各站点间地铁车厢人员拥挤情况，得到整个乘车期间的拥挤程度；
//路径储存于栈s中 栈顶为起点
```

```
double pathCrowd(Graph g,stack<int> s,double time){
    double totalCrowd = 0;
    int n = s.size();
    //使用map记录换乘站的上一个id
    map<string,int> name;
    //不考虑目的站的拥挤度
    //因为乘车时的拥挤度认为是上车时的拥挤度
    while (s.size() > 1){
        int a = s.top();
        //如果a站不是换乘站 或者a站是换乘站但是在此站下车进行换乘 拥挤度要换到本站的另一条线算 但是先在此处算上 下面else再减
        if(name.count(g.NodeList[a].name) == 0){
            totalCrowd += crowd(g,time,a);
            name.insert(pair<string,int>(g.NodeList[a].name,a));
        }
        //a站是换乘站的另一条线 在本线上车 把上一个a站的拥挤度减去 加上此时线路a站的拥挤度 再n-- 注意此时的time已经变了 要减掉换线时间
        else{
            totalCrowd -= crowd(g,time -
WALK,name.find(g.NodeList[a].name)->second);
            n--;
            totalCrowd += crowd(g,time,a);
            name.insert(pair<string,int>(g.NodeList[a].name,a));
        }
        s.pop();
        int b = s.top();
        time += timeMatrix[a][b];
    }
    return totalCrowd / (n - 1);
}
/*-----------encode.py-----------*/
```

```
import hashlib
list =[]
list2 = []
with open("InputData.txt", "r+") as f:
    for line in f.readlines():
        line = line.strip("\n")
        list.append(line)
for l in list:
    list2.append(hashlib.md5(l.encode(encoding='utf-8')).hexdigest())
with open("InputDataEncoded.txt","w") as f:
    for l in list2:
        f.write(l+"\n")
```

附录三：程序使用说明

使用前的准备：将 Metro.exe、encode.py 以及 MetroData.txt 三个文件放在一个文件夹里。

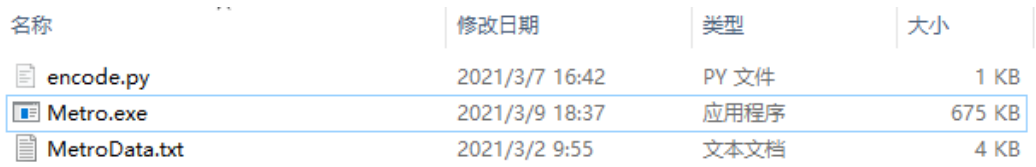

图 10-1 三个文件放在一个文件夹中

双击 Metro.exe 开始使用，根据提示输入序号以及相关信息即可得到对应输出。

功能一输入/输出样例：

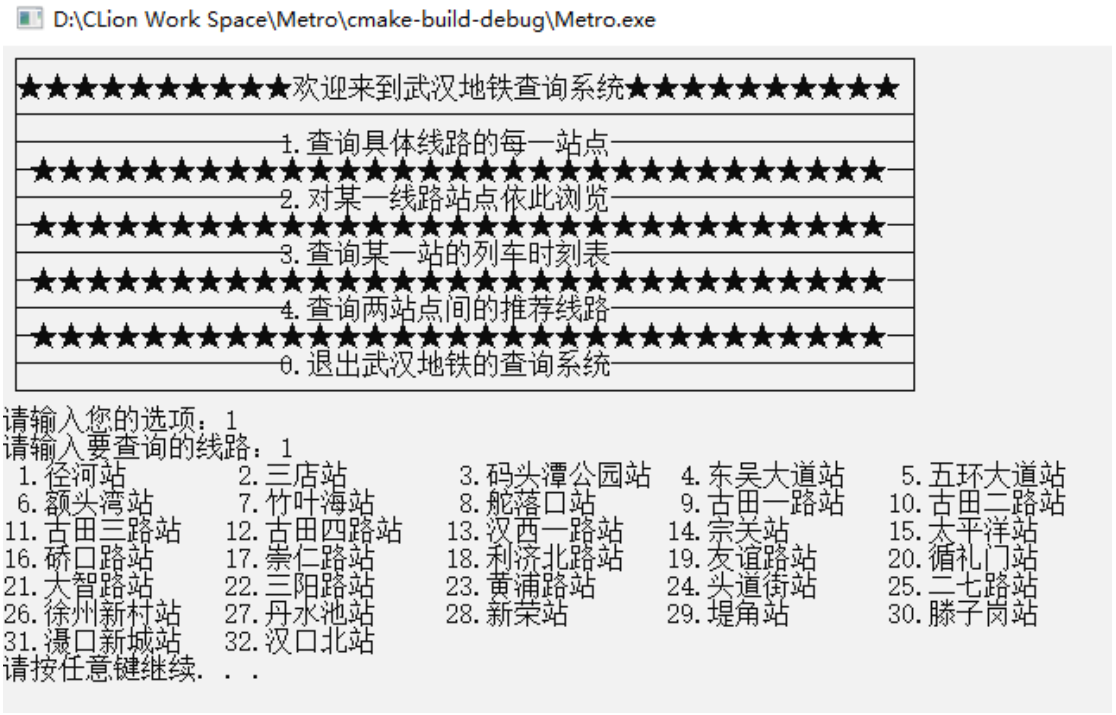

图 10-2 功能一输入/输出样例

功能二输入/输出样例：

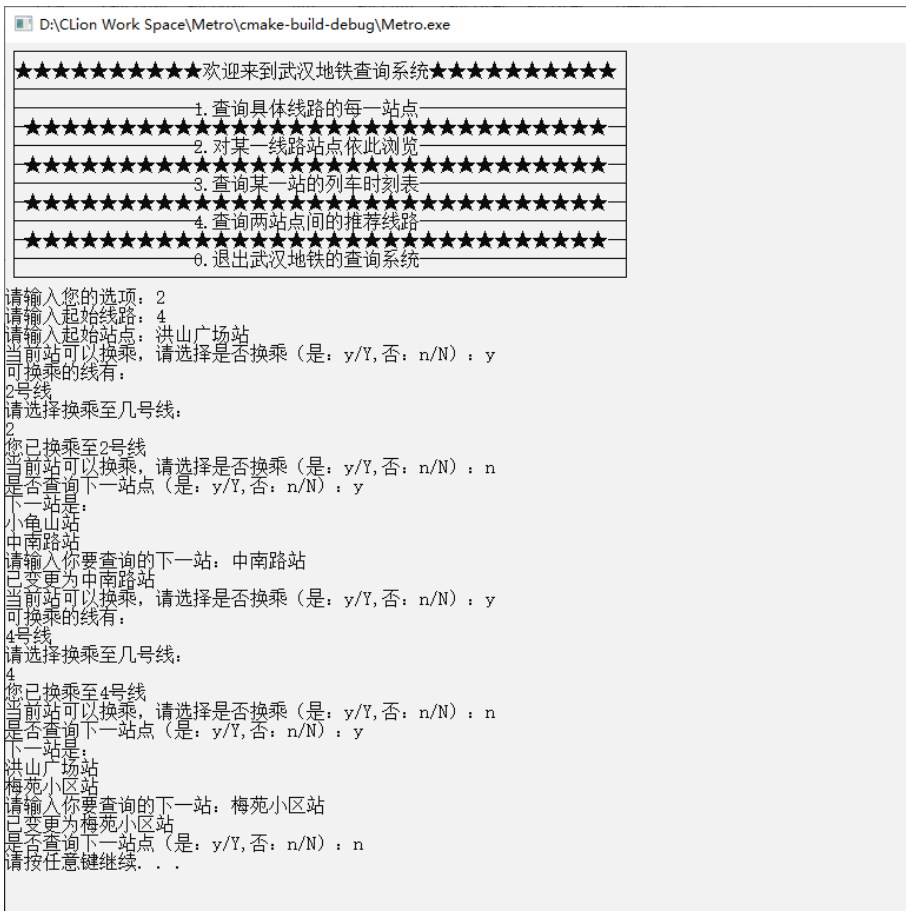

图 10-3 功能二输入/输出样例

功能三输入/输出样例：

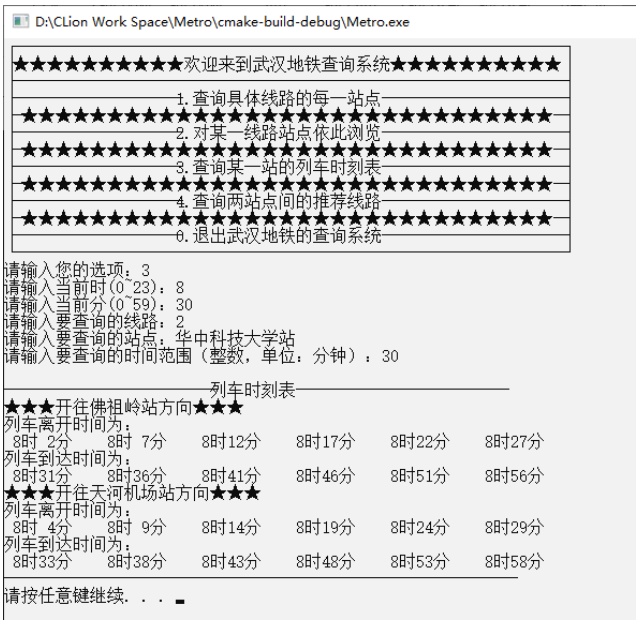

图 10-4 功能三输入/输出样例

功能四输入/输出样例：

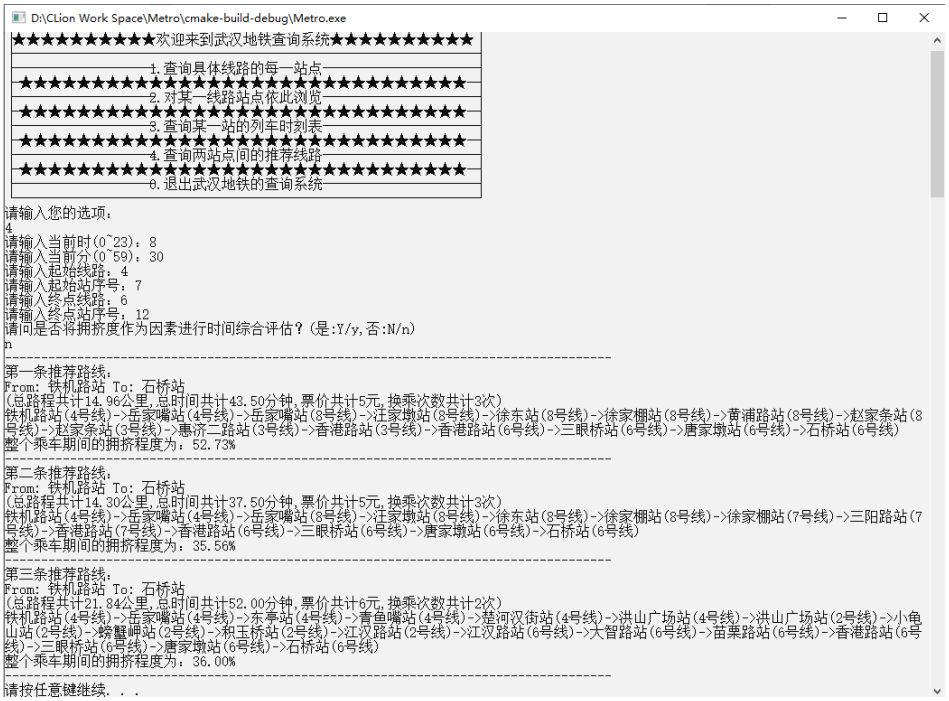

图 10-5 功能四输入/输出样例（1）

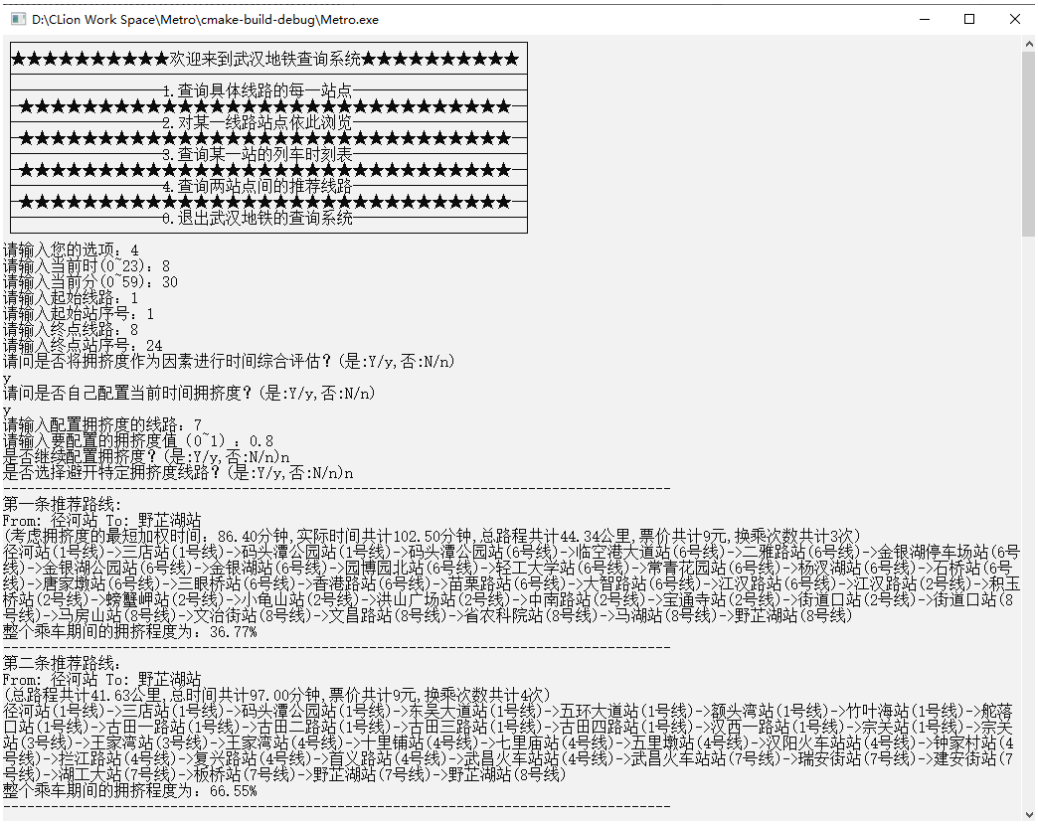

图 10-6 功能四输入/输出样例（2）

用户输入的所有信息保存在 InputData.txt 中：

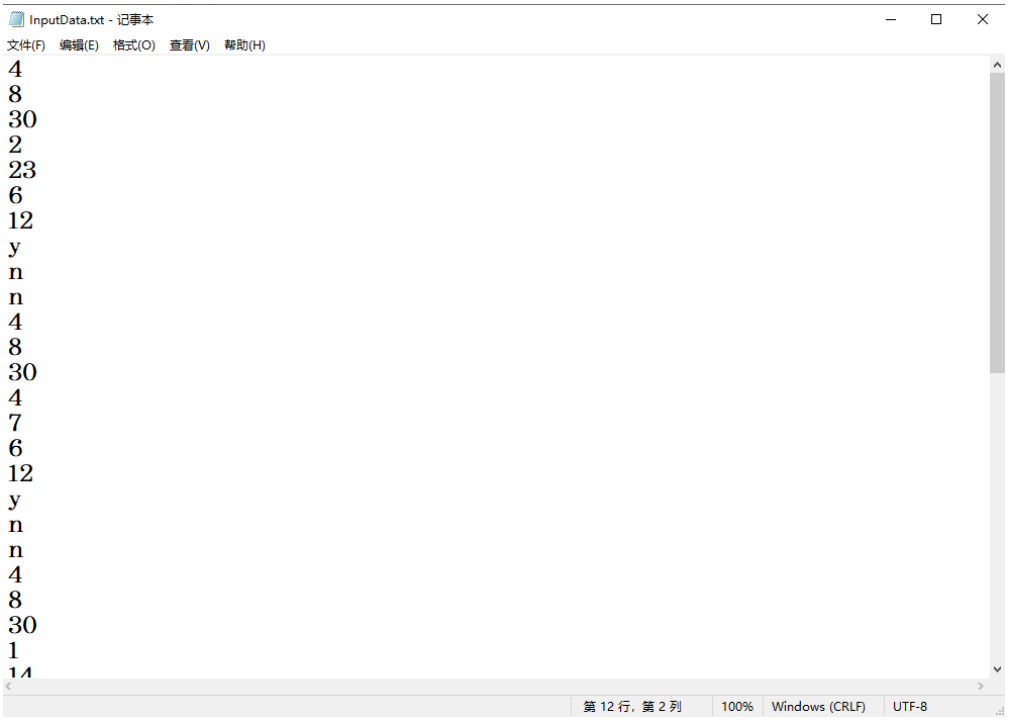

图 10-7 InputData.txt

用户输入的被加密的信息保存在 InputDataEncoded.txt 中：

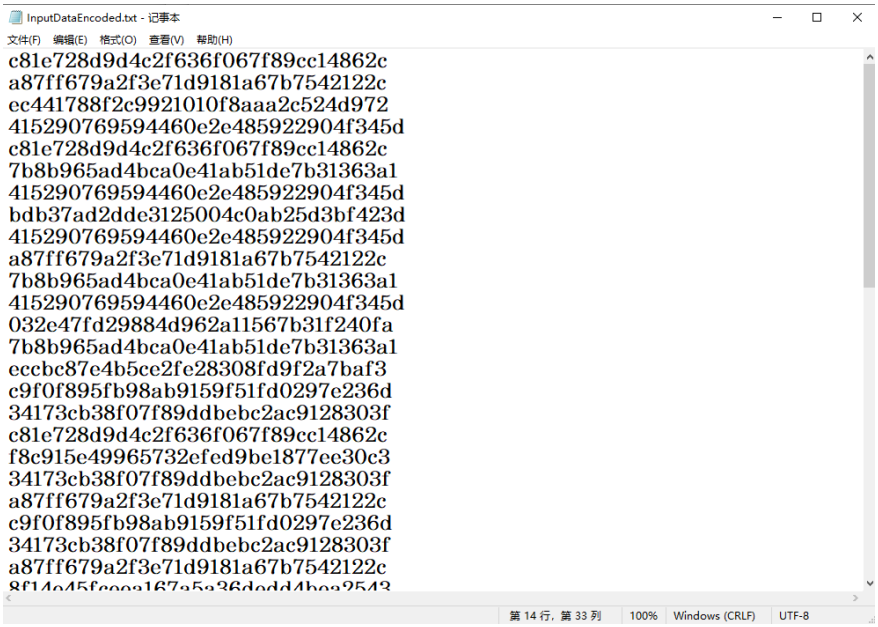

图 10-8 InputDataEncoded.txt