数据结构学期 Project

院　　系：计算机科学技术学院

专　　业：保密管理

姓　　名：陈辛楷

学　　号：15300240005

指导教师：陈彤兵

2016 年 12 月

目录

一、题目描述

我选择了第一题——交通推荐系统。题目如下：

阿鑫经常出国旅游，经常采用不同的交通出行方式，如何优化行程以达到开心旅游的效果是他所在意的！

现在，已知阿鑫行走的速度是 10km/h，地铁的行驶速度为40km/h。假设阿鑫是幸运的，即当阿鑫到达一个地铁站，一列火车就会在那里，阿鑫可以马上上车，车也会马上启动，即不考虑等待车的到达和启动时间。 阿鑫可以任意的上下地铁，可以在任意的线路间切换，所有的地铁均是双向的。任意两点均是可以直线到达，不考虑障碍物。

输入:

输入第一行为测试组数目，之后行按照以下格式:

之后一行两个 x,y 坐标，以及出行方式，前一对是阿鑫出发的坐标，后一对是目的地的坐标，出行方式以 0 或 1 编号（0 对应时间最短，1 对应步行最短）。

后一行，是一个整数 n，表示接下来会输入 n 个地铁线。最后 n 行，每行若干对整数，每对整数代表一个站点的坐标，坐标均不小于 0， 最后会以一对-1,-1 表示这一行结束。相邻站点之间会有双向地铁连通。每行至少有两个站点。

总共最多有 1000 个站点。所有的坐标均是以 m 为单位的。

输出:

阿鑫从出发坐标到目的点最少的时间或最少步行距离。

最少时间以分钟为单位， 精确到整数。

最少步行距离以 m 为单位，精确到整数。

二、程序运行环境

编程环境：C++

IDE：Visual Studio 2015

三、题目理解和分析

交通推荐系统的核心在于不同出行方式之间不同搭配的比较。本题中，有两种出行方式：步行和乘坐地铁。在单次出行中，阿鑫可以两种方式并行，以达到行程优化的目的。行程优化的结果，一是步行距离最短，二是总时间最短，这都是日常生活中人们希望达到的结果。

阿鑫可以任意的上下地铁，可以在任意的线路间切换，所有的地铁均是双向的。任意两点均是可以直线到达，不考虑障碍物。这些条件都提示我们，各个站点和起点、终点其实组成了一个带权的无向连通图，而我们要在其中选择最佳的路线。

首先，考虑步行最短的情况。起点、终点到图上各个点的边的权值即是两点的直线距离，而同一条地铁线的点之间的距离可以看做0，即乘坐地铁的过程不计入步行距离当中。不同地铁线的点之间，也是计直线距离。如果有共用地铁站点的情况，即可看做是不同线路的两

个点重合，因此这样的情况也是符合计算公式的。

其次，考虑时间最短的情况。即图上各条边的权值应当是距离除以速度，如果是步行，则除以步行速度；如果乘坐地铁，如果是同一条地铁线之间则除以地铁速度，若不是则还是计算步行时间。

上述两种情况对应的是两种不同的图，在构建时根据数据选择其中一种进行构建，然后清空图，进行下一次图的构建。至于解决时间最短和步行最短的问题，即是求图上从起点到终点的最短路问题，这时我们考虑到使用 Dijkstra 算法等求最短路径的算法，以帮助我们解决问题。

四、所用数据结构及其分析

很显然，我们需要利用的是图的数据结构。

```
class Vertex
{
public:
    //顶点的信息，1和2表示起点和终点，i(i>2)表示该点属于第i-2条地铁线
    int info;
    //顶点编号
    int num;
    //顶点横坐标
    double x;
    //顶点纵坐标
    double y;
};
```

首先，我们先定义图中的顶点，用一个 Vertex 的类，表示顶点包含顶点信息、顶点编号，以及横纵坐标四项信息。其中需要说明的是

顶点信息 info，info 为 1 时表示该点为起点，info 为 2 时表示该点为终点，info≥3 时表示它是地铁线，这样便于判断两个点是否在同一条地铁线上。

其次，就是图的数据结构了。

```
class Graph
{
private:
   int vexsnum;  //存放顶点数
   int linesnum;  //地铁线的数量
   Vertex vexs[VerticleMaxNumber];  //存放顶点
   double Cost[VerticleMaxNumber][VerticleMaxNumber];  //邻接矩阵
   double Dist[VerticleMaxNumber];  //存放从顶点0到其它各顶点的最短路径长度
   int Pre[VerticleMaxNumber];  //存放在最短路径上该顶点的前一顶点的顶点号
   int S[VerticleMaxNumber];  //已求得的在最短路径上的顶点的顶点号
public:
   Graph();
   void clear();
   double ShortestPath(Vertex s, Vertex e);  //求最短路径
   void InsertVertex(Vertex v);  //往图中添加顶点
   double MT_Distance(Vertex s, Vertex e);  //minimum time distance 最小时间的距离
   double SW_Distance(Vertex s, Vertex e);  //shortest walking distance 最短步行距离
   //double SL_Distance(Vertex s, Vertex e); //straight line distance 两点间直线距离
   void CreateMGraph(int ind);  //创建邻接矩阵
};
```

在这其中我们可以看到，用 private 封装的有顶点数 vexsnum、地铁线数 linesnum、存放顶点的 vexs 数组、用 Cost 数组表示的邻接矩阵，以及 Dist、Pre、S 数组，这三个数组是使用于最短路径算法 ShortestPath，它以 Dijkstra 算法为核心。

之所以使用邻接矩阵，是因为邻接矩阵使算法更加简洁明了，操作起来比较方便。并且，由于本问题中结点数和边数都较多，使用邻接矩阵和使用邻接表比起来，不会逊色得很多。

接下来我们来看函数。第一个是构造函数就不再赘述，第二个的 clear 函数即是程序中很重要的一个清空图的函数。因为每一次构建图进行计算之后，都要将图中所有的信息清空，即 private 中的各项信息，否则下一次进行构建时会出现问题。ShortestPath 函数在上文已经提到，InsertVertex 函数也是图的结构中经常使用到的函数，即往图中添加新的顶点。CreateMGraph 函数则是构建一个图的核心算法，它根据指示器 indicator 的值（0 或 1）来选择是构建求时间的图还是构建求步行距离的图。

此外，还有两个求距离的函数，MT_Distance 函数和 SW_Distance 函数。MT_Distance 即是 Minimum Time Distance，求的是在总时间最短的要求下两个结点之间的距离；SW_Distance 即是 Shortest Walking Distance，求得是在步行距离最短的要求下两个结点之间的距离。在 CreateMGraph 函数当中将会调用它们，以求得不同情形下的两点距离。在截图中也可以看到，原本还有一个 SL_Distance 函数，代表 Straight Line Distance，求的是两点之间的直线距离。但在之后优化的过程当中，发现调用这个函数比较费时，于是直接用 sqrt((s.x - e.x)*(s.x - e.x) + (s.y - e.y)*(s.y - e.y))求 s 和 e 两点间的直线距离，代替了该函数的调用，节省了时间，故将该函数略去。

对于数据结构的介绍就到这里，接下来将对各个算法进行具体的分析。

五、所用具体算法及其分析

（1）构造函数 Graph()

```
Graph::Graph()
{
   for (int i = 0;i < VerticleMaxNumber;i++) //邻接矩阵初始化
      for (int j = 0;j < VerticleMaxNumber;j++)
         Cost[i][j] = 0;
   for (int i = 0;i < VerticleMaxNumber;i++) //Dist、Pre、S数组初始化
   {
      Dist[i] = 0;
      Pre[i] = 0;
      S[i] = 0;
   }
   memset(vexs, 0, sizeof(vexs)); //vexs数组初始化
   //将边数和顶点数设为0
   vexsnum = 0;
   linesnum = 0;
}
```

构造函数并无太多值得说明，即把 Cost、Dist、Pre 和 S 四个数组进行初始化，将边数和顶点数赋为 0。它和之后用来清空图的 clear 函数其实是类似的。

（2）添加顶点的函数 InsertVertex

```
void Graph::InsertVertex(Vertex v)
{
   vexs[++vexsnum] = v;
   v.num = vexsnum;
}
```

该函数十分简单，即在 vexs 数组中加入新的结点，同时结点数 vexsnum 加一。由于 vexs 数组由 1 开始编号，新加入结点的编号即等

于结点总数。

（3）求最短路径算法 ShortestPath

```
double Graph::ShortestPath(Vertex s, Vertex e)
{
   int i, j, k;
   int v = s.num;
   double min;
   for (i = 1;i <= vexsnum;i++)  //进行Dist数组和S数组的初始化
   {
      Dist[i] = Cost[v][i];
      S[i] = 0;
      if (Dist[i] < MAXINT)
         Pre[i] = v;
      else
         Pre[i] = 0;
   }
   S[v] = 1;
   Pre[v] = 0;
   for (i = 1; i <= vexsnum; i++)
   {
      min = MAXINT;
      k = 0;
      for (j = 1; j <= vexsnum; j++)  //选择当前不在集合S中具有最短路径的顶点k
         if (S[j] == 0)
            if (Dist[j] != 0 && Dist[j] < min)
            {
               min = Dist[j];
               k = j;
            }
      if (k == 0)  break;  //找不到新的最短路径
      S[k] = 1;   //将顶点k加入S集合，表示它已在最短路径上
      for (j = 1; j <= vexsnum; j++)
         if (S[j] == 0 && Cost[k][j] < MAXINT)  //如果发现更短的路径，进行修改
            if (Dist[k] + Cost[k][j] < Dist[j])
            {
               Dist[j] = Dist[k] + Cost[k][j];
               Pre[j] = k;
            }
   }
   return Dist[e.num];  //返回Dist数组中e点到源点s点的距离
}
```

该算法以 Dijkstra 算法为核心，是常见的一种最短路算法，它是本题中最重要的算法之一。s 点表示 start，e 点表示 end，该算法即求起点到终点的最短距离。

首先我们看 Dist、Pre 和 S 三个数组。Dist 数组是存放从源点到其他各顶点的最短路径长度；Pre 数组则存放在最短路径上该顶点的前一顶点的顶点号；而 S 数组，则存放已求得的在最短路径上的顶点的顶点号。使用数组 Dist，一旦从源点 v 到顶点 k 的最短路径已求出，则 Dist[k]就是从源点到顶点 k 的最短路径长度。使用数组 Pre，数组元素 Pre[j]存放从源点 v 到顶点 j 的最短路径中 j 前面的顶点。有了 Pre 数组，就能很容易地求得从源点 v 到其它各个顶点的路径。同时，用 MAXINT=99999 表示邻接矩阵中的∞。

这三个数组是 Dijkstra 算法中主要使用的数据结构，不过在本题中，仅仅求出最短路径的值，其实可以不需要 Pre 数组，但它对于时间消耗几乎没有影响，我们不妨将它保留，如果日后这个问题需要实现更多的功能，保持算法的完整性还是很有必要的。

算法刚开始即是对 Dist、Pre 和 S 三个数组进行初始化，之后将起点加入 S 集合中，再从起点确定 vexsnum-1 条路径，选择当前不在集合 S 中的具有最短路径的顶点，将其加入 S 集合中。最后，如果发现了更短的路径，就进行修改。最终只要从 Dist 数组中找到我们所要求的最短路径长度 Dist[e.num]。

很显然，这个 ShortestPath 算法的时间复杂度是 $O(n^2)$。

（4）计算总时间最短情况下的各边权值的函数 MT_Distance

```
double Graph::MT_Distance(Vertex s, Vertex e)
{
   if (s.info == e.info)
   {
      if (s.info >= 3)
      {
         if (s.num - e.num == -1 )
            return sqrt((s.x - e.x)*(s.x - e.x) + (s.y - e.y)*(s.y - e.y)) / (sub_v / 3.6 * 60);
         else return sqrt((s.x - e.x)*(s.x - e.x) + (s.y - e.y)*(s.y - e.y)) / (man_v / 3.6 * 60);
      }
      else
         return -1;
   }
   else
      return sqrt((s.x - e.x)*(s.x - e.x) + (s.y - e.y)*(s.y - e.y))/ (man_v / 3.6 * 60);
}
```

正如上文中提到的，图上各边权值表示的是时间的情况下，若是同一条地铁线上的点，则以地铁的速度计算，否则，以步行的速度计算。因此函数首先比较了 s.info 和 e.info，在相同而且顶点信息合法的情况下进行计算。其中，sub_v 表示的是地铁速度 40km/h，man_v 表示的是步行速度 10km/h，但因为结果的单位是 min，我将速度先化为 m/s 再化为 m/min，故速度为(sub_v/3.6*60)。另外，上文中也以提到，在该函数中不再调用两点间距离公式的函数，这样能节约时间。

应该说明的是，在函数中，只有 s、e 相邻且 e 点后加入图的情况下（s.num-e.num==-1），才返回的是地铁行走的时间。因为在 CreateMGraph 函数当中，已经通过对称性更快地把权值赋给邻接矩阵，即在邻接矩阵 Cost[i][j]中 j 永远小于 i，故不需考虑 s 后加入图的情况。

这样计算是考虑到地铁线可能存在折线等情况，如果盲目把两个地铁站点间的直线距离除以速度，可能会出现问题。因此我们只需把相邻的地铁站点的距离求出，之后通过最短路径算法将它们相加，就可以顺利地得到结果。

（5）计算步行距离最短情况下的各边权值函数 SW_Distance

```
double Graph::SW_Distance(Vertex s, Vertex e)
{
   if (s.info >= 3 && e.info >= 3)
   {
      if (s.info == e.info)
         return 0;
      else return sqrt((s.x - e.x)*(s.x - e.x) + (s.y - e.y)*(s.y - e.y));
   }
   else
      return sqrt((s.x - e.x)*(s.x - e.x) + (s.y - e.y)*(s.y - e.y));
}
```

计算步行距离最短情况下的各边权值，该函数相对来说简单，首先因为它所求的即是距离而非时间，其次它涉及的情况也比较简单。如果是在同一条地铁线内的两个点，那么显然，步行距离为 0，其它情况下的步行距离即是两点之间的直线距离。

由上面两个函数可以看出，顶点信息 info 在判断不同情况的过程中还是很关键的。

（6）建立邻接矩阵的函数 CreateMGraph

```
void Graph::CreateMGraph(int ind) //根据indicator的取值，建立不同的邻接矩阵
{
   if (ind == 0)  //建立对应时间最短的邻接矩阵
   {
      for (int i = 1;i <= vexsnum;i++)
         for (int j = 1;j < i;j++)
         {
            Cost[i][j] = Cost[j][i] = MT_Distance(vexs[j], vexs[i]);
         }
   }
   if (ind == 1)  //建立对应最短步行距离的邻接矩阵
   {
      for (int i = 1;i <= vexsnum;i++)
         for (int j = 1;j <i;j++)
         {
            Cost[i][j] = Cost[j][i] = SW_Distance(vexs[i], vexs[j]);
         }
   }
}
```

该函数是根据指示器 indicator 的取值来选择建立什么样的图。上文已经提到，我们可以根据无向图邻接矩阵的对称性来构建图，只遍历 j<i 的部分，从而减少程序运行的时间。其中由于 MT_Distance 函数的两个结点是有顺序的，第一个点先加入图，因此是 vexs[j]在先，而在 SW_Distance 函数中则没有影响。

由于之前在两个函数中已经算好了各边的权值，因此构建邻接矩阵的过程也相对简单，直接进行赋值即可。它的时间复杂度也是 $O(n^2)$。

（7）清空图的函数 clear

```
void Graph::clear()
{
    memset(Dist, 0, sizeof(Dist));
    memset(Pre, 0, sizeof(Pre));
    memset(S, 0, sizeof(S));
    memset(Cost, 0, sizeof(Cost));
    memset(vexs, 0, sizeof(vexs));
    vexsnum = 0;
    linesnum = 0;
}
```

清空图的函数也比较简单，它和构造函数 Graph()其实十分类似。在 clear 函数中使用了 memset 函数对五个数组进行初始化，比起遍历的方式，十分简便。之后再把边数和顶点数赋为 0，整个图就被清空，可以进行下一次的构建了。

六、输入和输出

```
void main()
{
    Graph G;
    ifstream rfile("input.txt");
    if (!rfile)
    {
        cout << "不可以打开文件" << endl;
        exit(1);
    }
    ofstream outfile;
    outfile.open("output.txt");
    if (!outfile)
    {
        cout << "不可以打开文件" << endl;
        exit(1);
    }
    int count = 0, count1 = 0;  //count为总的计算次数，count1为每次计算时的地铁线条数
    int ind,k = 3, p = 3;  //p代表顶点编号，k代表存入第k-2条地铁线，它们都从3开始计数
    Vertex s, e;
    Vertex t;
    rfile >> count;
    for (int i = 0;i < count;i++)  //读入起点和终点
    {
        rfile >> s.x >> s.y >> e.x >> e.y >> ind;
        s.num = 1; s.info = 1;
        e.num = 2; e.info = 2;
        G.InsertVertex(s);
        G.InsertVertex(e);
        rfile >> count1;
        for (int i = 0;i < count1;i++)  //读入地铁站点
        {
            while (rfile >> t.x >> t.y && t.x != -1 && t.y != -1)
            {
                t.info = k;
                t.num = p++;
                G.InsertVertex(t);  //往图中加入顶点
            }
            k++;
        }
        G.CreateMGraph(ind);  //构建邻接矩阵
        double result = G.ShortestPath(s, e);  //计算最短路径
        outfile << round(result) << endl;  //用round函数进行四舍五入的取整
        p = 3, k = 3;
        G.clear();
    }
    rfile.close();
    outfile.close();
}
```

介绍完具体的算法，就到了最后的环节——读入和输出。本次 pj

仍然是通过文件进行输入和输出，因此 rfile 和 outfile 肯定是必要的。我设置了 count 和 count1，前者表示总共的计算次数，比如 input.txt 中是 1000 组数据，那么 count 即是 1000。而 count1 则表示在每组数据中的地铁线条数。另外，我定义了变量 ind，代表 indicator，它的取值（0 或 1）决定了要建立怎样的图。而之后的 p 和 k，p 代表顶点编号（即 num），k 代表存入第 k-2 条地铁线（即 info），由于起点和终点的编号是 1 和 2，因此 p 和 k 都从 3 开始计数。

接下来，就要构建一个图，创建始点 s 和终点 e，还有代表地铁站点的结点 t，它的值在不断读入的过程中不断更新，直到遇到终止的标志(-1,-1)。这时 count1 的一次循环结束，它要加 1，同时 k 也要加 1，代表现在读的是下一条地铁线。

当一组数据中所有的地铁线都读完，就开始输出，我在 result 中保存了所求的结果，并用 round 函数进行取整并输出。之后就是图的清空，调用了 clear 函数。并且 p 和 k 也要相应地初始化为 3，否则将会出现问题。

七、源代码

（1）Graph.h

```
#define VerticleMaxNumber 1000
#include<iostream>
#include<math.h>
using namespace std;
```

```
class Vertex
{
public:
    //顶点的信息，1和2表示起点和终点，i(i>2)表示该点属于第i-2条地铁线
    int info;
    //顶点编号
    int num;
    //顶点横坐标
    double x;
    //顶点纵坐标
    double y;
};

class Graph
{
private:
    int vexsnum;  //存放顶点数
    int linesnum;  //地铁线的数量
    Vertex vexs[VerticleMaxNumber];  //存放顶点
    double Cost[VerticleMaxNumber][VerticleMaxNumber];  //邻接矩阵
    double Dist[VerticleMaxNumber];  //存放从顶点0到其它各顶点的最短路径长
度
    int Pre[VerticleMaxNumber];  //存放在最短路径上该顶点的前一顶点的顶点号
    int S[VerticleMaxNumber];  //已求得的在最短路径上的顶点的顶点号
public:
    Graph();
    void clear();
    double ShortestPath(Vertex s, Vertex e);  //求最短路径
    void InsertVertex(Vertex v);  //往图中添加顶点
    double MT_Distance(Vertex s, Vertex e);  //minimum time distance 最小时
间的距离
    double SW_Distance(Vertex s, Vertex e);  //shortest walking distance 最
短步行距离
    void CreateMGraph(int ind);  //创建邻接矩阵
};
```

（2）Graph.cpp

```
#include "Graph.h"

const double man_v = 10;
const double sub_v = 40;
```

```
Graph::Graph()
{
    for (int i = 0;i < VerticleMaxNumber;i++)  //邻接矩阵初始化
        for (int j = 0;j < VerticleMaxNumber;j++)
            Cost[i][j] = 0;
    for (int i = 0;i < VerticleMaxNumber;i++)  //Dist、Pre、S数组初始化
    {
        Dist[i] = 0;
        Pre[i] = 0;
        S[i] = 0;
    }
    memset(vexs, 0, sizeof(vexs));  //vexs数组初始化
    //将边数和顶点数设为0
    vexsnum = 0;
    linesnum = 0;
}

void Graph::clear()
{
    memset(Dist, 0, sizeof(Dist));
    memset(Pre, 0, sizeof(Pre));
    memset(S, 0, sizeof(S));
    memset(Cost, 0, sizeof(Cost));
    memset(vexs, 0, sizeof(vexs));
    vexsnum = 0;
    linesnum = 0;
}

double Graph::ShortestPath(Vertex s, Vertex e)
{
    int i, j, k;
    int v = s.num;
    double min;
    for (i = 1;i <= vexsnum;i++)  //进行Dist数组和S数组的初始化
    {
        Dist[i] = Cost[v][i];
        S[i] = 0;
        if (Dist[i] < MAXINT)
            Pre[i] = v;
        else
            Pre[i] = 0;
    }
    S[v] = 1;
```

```
    Pre[v] = 0;
    for (i = 1; i <= vexsnum; i++)
    {
        min = MAXINT;
        k = 0;
        for (j = 1; j <= vexsnum; j++)   //选择当前不在集合S中具有最短路径的顶点
k
            if (S[j] == 0)
                if (Dist[j] != 0 && Dist[j] < min)
                {
                    min = Dist[j];
                    k = j;
                }
        if (k == 0)   break;   //找不到新的最短路径
        S[k] = 1;    //将顶点k加入S集合，表示它已在最短路径上
        for (j = 1; j <= vexsnum; j++)
            if (S[j] == 0 && Cost[k][j] < MAXINT)   //如果发现更短的路径，进行
修改
                if (Dist[k] + Cost[k][j] < Dist[j])
                {
                    Dist[j] = Dist[k] + Cost[k][j];
                    Pre[j] = k;
                }
    }
    return Dist[e.num];   //返回Dist数组中e点到源点s点的距离
}

void Graph::InsertVertex(Vertex v)
{
    vexs[++vexsnum] = v;
    v.num = vexsnum;
}

double Graph::MT_Distance(Vertex s, Vertex e)
{
    double tempDist = 0;
    if (s.info == e.info)
    {
        if (s.info >= 3)
        {

            if (s.num - e.num == -1 )
                return sqrt((s.x - e.x)*(s.x - e.x) + (s.y - e.y)*(s.y - e.y)) / (sub_v
/ 3.6 * 60);
```

```
            else return sqrt((s.x - e.x)*(s.x - e.x) + (s.y - e.y)*(s.y - e.y)) /
(man_v / 3.6 * 60);
        }
        else
            return -1;
    }
    else
    {
        return sqrt((s.x - e.x)*(s.x - e.x) + (s.y - e.y)*(s.y - e.y))/ (man_v / 3.6 *
60);

    }
}

double Graph::SW_Distance(Vertex s, Vertex e)
{
    if (s.info >= 3 && e.info >= 3)
    {
        if (s.info == e.info)
            return 0;
        else return sqrt((s.x - e.x)*(s.x - e.x) + (s.y - e.y)*(s.y - e.y));
    }
    else
        return sqrt((s.x - e.x)*(s.x - e.x) + (s.y - e.y)*(s.y - e.y));
}

void Graph::CreateMGraph(int ind) //根据indicator的取值，建立不同的邻接矩阵
{
    if (ind == 0)   //建立对应时间最短的邻接矩阵
    {
        for (int i = 1;i <= vexsnum;i++)
            for (int j = 1;j < i;j++)
            {
                Cost[i][j] = Cost[j][i] = MT_Distance(vexs[j], vexs[i]);
            }
    }
    if (ind == 1)   //建立对应最短步行距离的邻接矩阵
    {
        for (int i = 1;i <= vexsnum;i++)
            for (int j = 1;j <i;j++)
            {
                Cost[i][j] = Cost[j][i] = SW_Distance(vexs[i], vexs[j]);
            }
    }
```

}

(3)travel.cpp(即 main.cpp)

```
#include"Graph.h"
#include<fstream>
#include<sstream>
#include <string>

void main()
{
    Graph G;
    ifstream rfile("input.txt");
    if (!rfile)
    {
        cout << "不可以打开文件" << endl;
        exit(1);
    }
    ofstream outfile;
    outfile.open("output.txt");
    if (!outfile)
    {
        cout << "不可以打开文件" << endl;
        exit(1);
    }
    int count = 0, count1 = 0;   //count为总的计算次数，count1为每次计算时的地铁线条数
    int ind,k = 3, p = 3;   //p代表顶点编号，k代表存入第k-2条地铁线，它们都从3开始计数
    Vertex s, e;
    Vertex t;
    rfile >> count;
    for (int i = 0;i < count;i++)   //读入起点和终点
    {
        rfile >> s.x >> s.y >> e.x >> e.y >> ind;
        s.num = 1; s.info = 1;
        e.num = 2; e.info = 2;
        G.InsertVertex(s);
        G.InsertVertex(e);
        rfile >> count1;
        for (int i = 0;i < count1;i++)   //读入地铁站点
        {
```

```
            while (rfile >> t.x >> t.y && t.x != -1 && t.y != -1)
            {
                t.info = k;
                t.num = p++;
                G.InsertVertex(t);   //往图中加入顶点
            }
            k++;
        }
        G.CreateMGraph(ind);   //构建邻接矩阵
        double result = G.ShortestPath(s, e);   //计算最短路径
        outfile << round(result) << endl;   //用round函数进行四舍五入的取整
        p = 3, k = 3;
        G.clear();
    }
    rfile.close();
    outfile.close();
}
```

八、测试和运行

我借用了同学的测试数据进行测试，该同学的数据满足题目要求：30%的数据站点数小于 10 个;70%的数据站点数小于 100 个;100%的数据站点数小于 1000 个。所有的坐标都不小于 0，且不超过(100000,100000)。

input.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

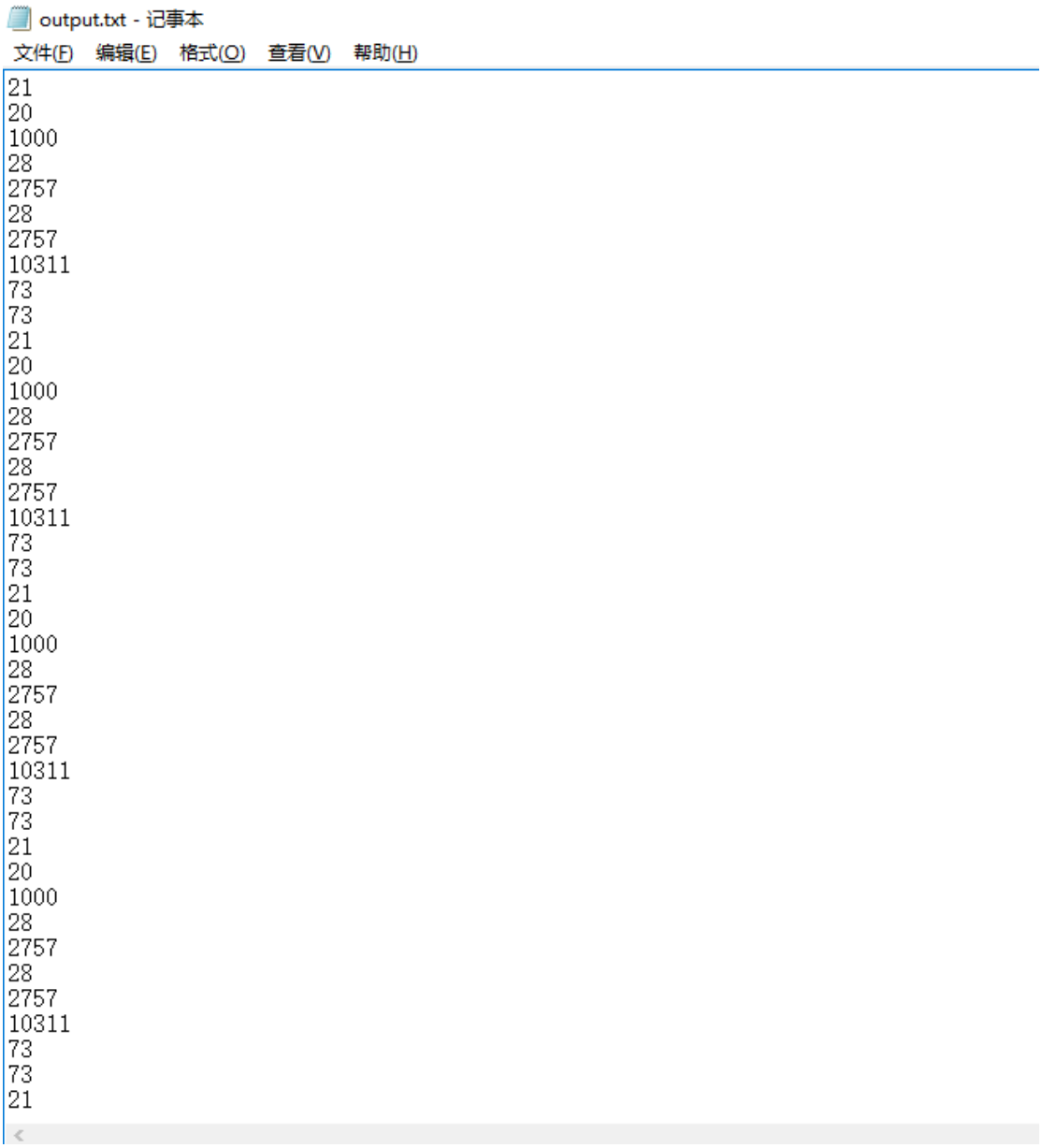
```
1000
0 0 10000 1000 0
2
0 200 5000 200 7000 200 -1 -1
2000 600 5000 600 10000 600 -1 -1
0 0 10000 1000 0
2
0 200 5000 200 5000 700 -1 -1
2000 600 5000 600 10000 600 -1 -1
0 0 10000 1000 1
2
0 200 5000 200 7000 200 -1 -1
2000 600 5000 600 10000 600 -1 -1
0 0 10000 1000 0
9
0 100 2500 100 5000 100 7500 100 -1 -1
0 200 2500 200 5000 200 7500 200 -1 -1
0 300 2500 300 5000 300 7500 300 -1 -1
0 400 2500 400 5000 400 7500 400 -1 -1
0 500 2500 500 5000 500 7500 500 -1 -1
0 600 2500 600 5000 600 7500 600 -1 -1
0 700 2500 700 5000 700 7500 700 -1 -1
0 800 2500 800 5000 800 7500 800 -1 -1
0 900 2500 900 5000 900 7500 900 -1 -1
0 0 10000 1000 1
9
0 100 2500 100 5000 100 7500 100 -1 -1
0 200 2500 200 5000 200 7500 200 -1 -1
0 300 2500 300 5000 300 7500 300 -1 -1
0 400 2500 400 5000 400 7500 400 -1 -1
0 500 2500 500 5000 500 7500 500 -1 -1
0 600 2500 600 5000 600 7500 600 -1 -1
0 700 2500 700 5000 700 7500 700 -1 -1
0 800 2500 800 5000 800 7500 800 -1 -1
0 900 2500 900 5000 900 7500 900 -1 -1
0 0 10000 1000 0
9
0 100 2500 100 5000 100 7500 100 -1 -1
0 200 2500 200 5000 200 7500 200 -1 -1
0 300 2500 300 5000 300 7500 300 -1 -1
0 400 2500 400 5000 400 7500 400 -1 -1
```

input.txt - 记事本

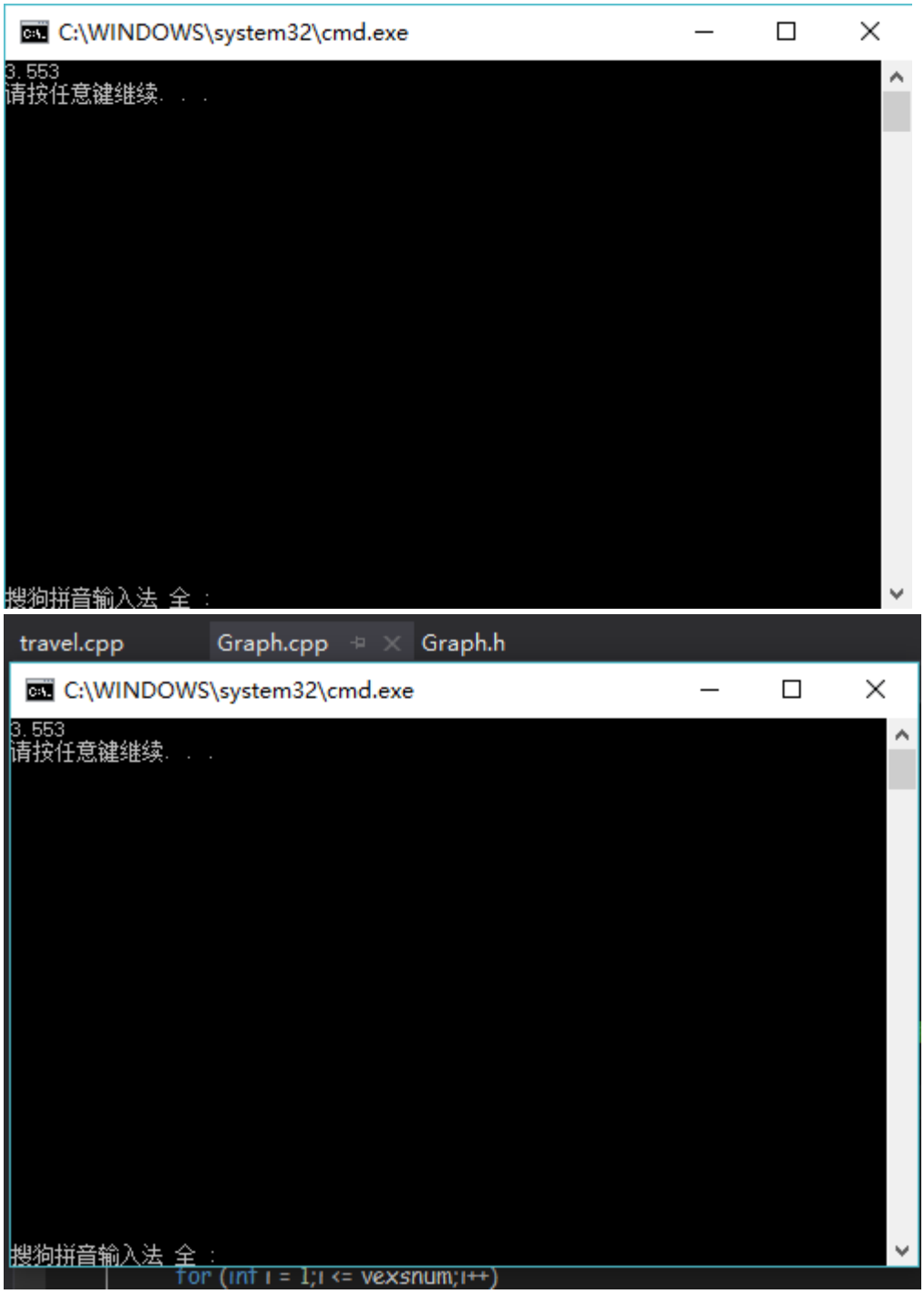
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
0 700 2500 700 5000 700 7500 700 -1 -1
0 800 2500 800 5000 800 7500 800 -1 -1
0 900 2500 900 5000 900 7500 900 -1 -1
0 0 10000 10000 1
99
0 100 2500 100 5000 100 7500 100 -1 -1
0 200 2500 200 5000 200 7500 200 -1 -1
0 300 2500 300 5000 300 7500 300 -1 -1
0 400 2500 400 5000 400 7500 400 -1 -1
0 500 2500 500 5000 500 7500 500 -1 -1
0 600 2500 600 5000 600 7500 600 -1 -1
0 700 2500 700 5000 700 7500 700 -1 -1
0 800 2500 800 5000 800 7500 800 -1 -1
0 900 2500 900 5000 900 7500 900 -1 -1
0 1000 2500 1000 5000 1000 7500 1000 -1 -1
0 1100 2500 1100 5000 1100 7500 1100 -1 -1
0 1200 2500 1200 5000 1200 7500 1200 -1 -1
0 1300 2500 1300 5000 1300 7500 1300 -1 -1
0 1400 2500 1400 5000 1400 7500 1400 -1 -1
0 1500 2500 1500 5000 1500 7500 1500 -1 -1
0 1600 2500 1600 5000 1600 7500 1600 -1 -1
0 1700 2500 1700 5000 1700 7500 1700 -1 -1
0 1800 2500 1800 5000 1800 7500 1800 -1 -1
0 1900 2500 1900 5000 1900 7500 1900 -1 -1
0 2000 2500 2000 5000 2000 7500 2000 -1 -1
0 2100 2500 2100 5000 2100 7500 2100 -1 -1
0 2200 2500 2200 5000 2200 7500 2200 -1 -1
0 2300 2500 2300 5000 2300 7500 2300 -1 -1
0 2400 2500 2400 5000 2400 7500 2400 -1 -1
0 2500 2500 2500 5000 2500 7500 2500 -1 -1
0 2600 2500 2600 5000 2600 7500 2600 -1 -1
0 2700 2500 2700 5000 2700 7500 2700 -1 -1
0 2800 2500 2800 5000 2800 7500 2800 -1 -1
0 2900 2500 2900 5000 2900 7500 2900 -1 -1
0 3000 2500 3000 5000 3000 7500 3000 -1 -1
0 3100 2500 3100 5000 3100 7500 3100 -1 -1
0 3200 2500 3200 5000 3200 7500 3200 -1 -1
0 3300 2500 3300 5000 3300 7500 3300 -1 -1
0 3400 2500 3400 5000 3400 7500 3400 -1 -1
0 3500 2500 3500 5000 3500 7500 3500 -1 -1
0 3600 2500 3600 5000 3600 7500 3600 -1 -1
```

对这样的数据进行测试之后，得到了正确的结果。



在程序中我使用了<time.h>计算程序运行的时间，在程序开头定义 time1，结尾定义 time2，最后输出 1.0*(time2 - time1) / CLOCKS_PER_SEC，得到结果是大约 3.6s 左右。



启用 Visual Studio 的诊断工具，诊断得使用的内存大约是 10.2MB。

九、写 PJ 过程中的各项工作及代码优化

刚开始写程序的时候，其实处于有思路但是比较模糊的状态，之后我有上网查阅了一些论文和资料，比如《城际公共交通系统最短路算法》，以及网络上一些关于交通指南但是题目不同的报告，也和同学有一些交流，对我的思路进行了启发，最终我还是吸收了一些好的想法，独立完成了这个项目。

在正确完成程序之后，我开始对一些步骤进行优化。比如邻接矩阵的读入可以通过对称性来更快实现；文件读入时用一个变量进行保存而不是使用字符串的读取，也减少了程序运行的时间；直接使用两点间距离公式而不调用函数，等等。原来我的程序跑完需要十多秒，现在可以稳定在四秒以内。

十、结语

通过这次写 PJ 的过程，我不仅对图的相关问题，包括其数据结构以及最短路等问题有了更深入的理解，也对代码的分析、优化有了更多的认识。同时，我了解到计算机程序在日常生活当中的应用是十分广泛的，它能为我们的生活带来极大的便利。因此，我也有了更大的热情和更多的兴趣去写程序，去学习计算机的相关知识。在这个过程中我受益匪浅。