三、算法与分析

1.算法选择

这个问题实际上是一个最短路径问题。图中已知 n 个点（n 个地铁站），m 条边（连接相邻两个地铁站的路线），每条边都有权值，权值对应的是相邻站点之间的距离，最后得到一个点到另一个点，经过权值和最小的路线。

最短路径问题，首先想到的就是使用暴力法，不遗漏遍地历出所有路线，找到其中权值和最小的路线。最简单的方法就是将 n 个结点，全排列，找其中符合要求且相通的，但排列共有 n!种，时间复杂度太高。可以使用深度优先搜索改进，但时间复杂度仍较高。

前人已经为我们总结出了一些好用的算法。常用的计算最短路径的算法有 Floyd 算法、Bellman-Ford 算法、SPFA 算法，Dijkstra 算法等。这里选用了任务要求的 Dijkstra 算法。

2.Dijkstra 算法分析

Dijkstra 算法是典型的单源最短路径算法，用于计算一个结点到其他所有结点的最短路径。

(1)主要特点

以起始点为中心向外层层扩展，直到扩展到终点为止。

(2)算法思想

每一步都是当前最短的路径，那么下一步在这一步的基础上，选择最短的，这样也就得到了到下一个结点的当前最短路径。这实际上应用了贪心法的思想，每次都抄近路走，那么肯定可以找到最短路径。

首先，需要指定起点 s(即从顶点 s 开始计算)。此外，引进两个集合 S 和 U。

S 的作用是记录已求出最短路径的顶点(以及相应的最短路径长度)，而 U 则是记录还未求出最短路径的顶点(以及该顶点到起点 s 的距离)。初始时，S 中只有起点 s；U 中是除 s 之外的顶点，并且 U 中顶点的路径是"起点 s 到该顶点的路径"。然后，从 U 中找出路径最短的顶点，并将其加入到 S 中；接着，更新 U 中的顶点和顶点对应的路径。然后，再从 U 中找出路径最短的顶点，并将其加入到 S 中；接着，更新 U 中的顶点和顶点对应的路径。 … 重复该操作，直到遍历完所有顶点。

Dijkstra 的每次迭代，只需要检查上次已经确定最短路径的那些结点的邻居即可，算法是高效的，时间复杂度为 O(n^2)。

(3)具体步骤：

1）选中起点 s，集合 S 中只包含 s。初始化 U 中顶点到 s 的距离，无法到达点的距离的记为无穷。

2）从 U 中选出"距离顶点 s 最短的顶点 k"，并将顶点 k 加入到 S 中。同时，从 U 中移除顶点 k。

3）更新 U 中各个顶点到起点 s 的距离，dis(s,v)=min(dis(s,v),dis(s,k)+dis(k,v))。

4）重复步骤(2)和(3)，直到遍历完所有顶点。

3.数据存储

(1)站名存储

使用一个一维的 string 数组用于存储站名站名，每一个站名都对应着其在数组中的下标，之后在使用站名时，使用其对应下标即可。为了后面使用方便，数组下标为 0 的位置不使用。

```
const string stationName[num+1]={"","pingguoyuan","gucheng","bajiao amusement park","babaoshan","yuquanlu","wukesong",
                                 "wanshoulu","gongzhufen","military museum","muxidi","nanlishilu","fuxingmen","fuchengmen",
                                 "chegongzhuang","xizhimen","xinjiekou","ping'anli","xisi","lingjinghutong","xidan","zhangchunjie",
                                 "xuanwumen","tian'anmenxi(W)","jishuitan","guloudajie","andingmen","tian'anmendong(E)","wangfujing",
                                 "hepingmen","qianmen","chongwenmen","dongdan","dengshikou","dongsi","zhangzizhonglu","beixinqiao"
                                 ,"yonghegong lama temple","dongzhimen","dongsishitiao","chaoyangmen","jianguomen","beijing railway statio
                                 "guomao","dawanglu","sihui","sihuidong(E)"};
//1 苹果园2 古城3 八角游乐园4 八宝山5 玉泉路6 五棵松7 万寿路8 公主坟9 军事博物馆10 木樨地11 南礼士路12 复兴门13 阜成门
//14 车公庄15 西直门16 新街口17 平安里18 西四19 灵境胡同20 西单21 长椿街22 宣武门23 天安门西24 积水潭25 鼓楼大街26 安定门
//27 天安门东28 王府井29 和平门30 前门31 崇文门32 东单33 灯市口34 东四35 张自忠路36 北新桥
//37 雍和宫38 东直门39 东四十条40 朝阳门41 建国门42 北京站 43 永安里 44 国贸 45 大望路46 四惠47 四惠东
```

(2)地图存储

a.邻接矩阵

邻接矩阵可以表示相邻顶点之间的连接关系。在程序中，使用一个二维数组来记录。空间复杂度为 O(n^2)。

该问题的图为无向图，所以连接的车站之间要记录两次。标号为 i 的车站和标号为 j 的车站对应的数组位置即为 map[i][j]和 map[j][i]。该问题要存储的图为带权图，所以数组中标记的数为两个车站之间的距离。在初始时，不同车站都先标记为一个无穷大的数（这里用 0x3f3f3f 代替），同一个车站标记为 0。

```
# define inf 0x3f3f3f3f                //定义一个充当无穷大的数
# define num 48;                       //车站数
int map[num+1][num+1];                 //存图的二维数组
void init()                            //初始化数据函数
{
    for(int i=1;i<=num;i++)
    {
        for(int j=1;j<=num;j++)
            if(i==j)
                a[i][j]=0;             //同一个车站标记为 0
            else
                a[i][j]=inf;           //初始距离为无穷大
    }
    //站点距离
    a[1][2]=a[2][1]=2606;
    a[2][3]=a[3][2]=1921;
    a[3][4]=a[4][3]=1953;
    a[4][5]=a[5][4]=1479;
    a[5][6]=a[6][5]=1810;
    a[6][7]=a[7][6]=1778;
    //此处省略了边连接的代码
}
```

b.链式前向星

本问题的车站中，相邻的只占少数，所以图为一个稀疏图。邻接矩阵的空间复杂度为 O(n^2)，用邻接矩阵存储的话有一点浪费空间。还可以采用邻接表或者

链式前向星来存图，空间复杂度都为 O(m)。对于数据量更大的问题，邻接矩阵和其他两种存图方式的所占空间差别就更加明显了。比如，1000 个结点的图，其中有一个结点只与一个结点相连，与其他结点均不相连，如果用邻接矩阵的话，需要将这个结点与其他 999 个结点的连接情况都记录下来。如果使用邻接表或者链式前向星的话，只需存储一条边即可。

这里选择了使用链式前向星。链式前向星中存储的是每一条边。定义一个结构体代表边，其成员包括终点，与这条边同起点的下一条边的编号，这条边的长度。

```
struct edge                                  //边
{
    int to;                                  //终点
    int next;                                //与这条边同起点的下一条边的终点
    int length;                              //这条边的长度；
}e[edg eNum+2];
```

还需要一个 head 数组用来记录相同起点的边中的最后一条的编号。比如要找到以 1 号结点为起点的所有的边，那么 head[1]就为以 1 号结点为起点的边的最后一条的编号。e[head[1]]就可以得到这条边的信息。e[head[1]].next 就可以得到这条边的前一条边的编号，一直向前找下去，就完成了对以 1 号结点为起点的边的遍历。

```
int head[edgeNum+2];                         //记录起点相同的边中最后一条边
for(int i=head[1];i;i=e[i].next)
{
    //以 1 号结点为起点的边的遍历
    //对边进行相关操作
}
```

添加边，定义一个 cnt 变量记录当前边的数量，每存一条边对 cnt＋1，每一条边的存储记录其终点，长度，它的前一条边编号。同时将该起点的 head 数组指向新边。该问题为无向图，同一条边存两次。

```
void add(int u,int v,int w)                  //加边  u 起点 v 终点  w 边长
{
    static int cnt=0;                        //记录边的标号
    e[++cnt].to=v;                           //第 cnt 条边起点
    e[cnt].length=w;                         //第 cnt 条边长度
    e[cnt].next=head[u];                     //第 cnt 条边的下一条边
    head[u]=cnt;                             //head 数组指向新边
    e[++cnt].to=u;                           //无向边 存两次
```

```
    e[cnt].length=w;
    e[cnt].next=head[v];
    head[v]=cnt;
}
```

将图中的所有边添加进去。

```
void init()
{
    add(1,2,2606);                          //添加边
    add(2,3,1921);
    add(3,4,1953);
    add(4,5,1479);
    add(5,6,1810);
    //此处省略了其他边的添加
}
```

4.Dijkstra 算法实现

(1)邻接矩阵

定义一个 vis 数组用于标记已经使用的顶点，dis 数组记录各个结点到起点的最短距离。每一个结点仅使用一次，遍历所有未使用的结点，找出其中到起点距离最短的结点，更新与该结点相连的结点到起点的最短距离，同时将该结点标记为已使用。重复该过程，直至所有结点均已使用，那么此时，所有结点到起点的距离均为最短。

```
int dis[num+1];                             //到起点 s 的距离
int vis[num+1];                             //用于标记一个结点是否已经使用过
void dijkstra(int start)
{
    vis[start]=1;                           //标记起点已经使用
    for(int i=1;i<=num;i++)                 //初始化所有结点到起点的距离
        dis[i]=a[start][i];
    bool tmp=1;                             //标记变量
    while(1)
    {
        tmp=1;                              //恢复为 1
        int min=inf;                        //最小的刚开始为无穷
        int work=0;
        for(int i=1;i<=num;i++)
        {
```

```
            if(dis[i]<min&&!vis[i])       //距离小且没有使用过该结点
            {
                min=dis[i];               //找到剩余结点中距离最短的
                work=i;                   //记录该结点的下标
                tmp=0;                    //将标记变量置为0
            }
        }
        if(tmp)        //如果上一步中没有找到，说明所有结点都使用了，退出循环
            break;
        vis[work]=1;                      //标记该次选中的距离最短的结点
        for(int i=1;i<=num;i++)           //更新距离
            if(!vis[i]&&dis[i]>dis[work]+a[work][i])
                dis[i]=dis[work]+a[work][i]; //更新为最小值
    }
}
```

(2)链式前向星

首先定义一个结构 node，用于储存该结点的编号，到起点的距离，。定义一个 vis 数组用于标记已经使用的顶点，dis 数组记录各个结点到起点的最短距离。

```
int vis[nodeNum+2];                       //标记结点是否使用
int dis[nodeNum+2];                       //记录各结点到要求起点最短距离
struct node                               //结点
{
    int now;                              //该点编号
    int w;                                //起点到该结点的距离
    node(int a,int b)                     //构造函数
    {
        now=a;
        w=b;
    }
    //重写优先队列的排序函数  小的先出
    bool operator<(const node& x) const
    {
        return w>x.w;                     //到起点距离小的排在前面
    }
};
```

每次需要选择到起点距离最短的结点，选择使用优先队列来实现。直接使用 C++模板库的优先队列，默认是从大到小排序，在上面结点结构体的定义中重写了该函数。

```
priority_queue<node>q;                    //定义一个优先队列，用于存储选出的结点
```

首先现将初始距离设为无穷。将起点加入到队列之中，使用一个循环遍历队列中的首元素。因为是优先队列，所以首元素就是到起点距离最短的。标记该点，更新与该点相连的所有边的点到起点的距离，同时将被更新的点加入到队列之中。当队列中的元素数量为 0 时，说明所有点都遍历过，到起点的距离都达到了最小值，结束循环。

```
void dijkstra(int s)
{
    for(int i=1;i<=nodeNum;i++)
        dis[i]=inf;                                  //初始距离都设为无穷
    dis[s]=0;
    q.push((node){s,0});                             //起始点加入队列
    while(q.size())                                 //队列为空了，就说明所有点都是最短距离了
    {
        node x=q.top();                              //读出队列中距离最小的
        q.pop();                                     //使刚读入的结点出列
        int r=x.now;
        if(vis[r])                                   //如果该结点使用过，则继续下一个循环
            continue;
        vis[r]=1;                                    //标记这次选用的结点
        for(int i=head[r];i;i=e[i].next)             //遍历所有与 r 相连的点
        {
            int t=e[i].to;
            if(dis[t]>dis[r]+e[i].length)            //更新到结点 t 的距离
            {
                dis[t]=dis[r]+e[i].length;
                q.push((node){t,dis[t]});            //将结点 t 加入队列
            }
        }
    }
}
```

5.数据处理

(1)读入数据

题目中一行输入起点和终点，之间用分号隔开，同时之间可能含有空格。有空格用 getline 函数输入。将整行存入一个 string 变量之中，调用 string 类的 find 函数，查找“;”的下标。使用 string 类的 substr 函数将提取分号前后的内容。

```
    string str;
```

```
    getline(cin,str);                          //不能用 cin，有的站名包含空格
    int p=str.find(";");                       //找到；对应的下标
    string start,end;
    start=str.substr(0,p);                     //划分；前的
    end=str.substr(p+1);                       //划分；后的
```

后来发现 getline 函数可以传入第三个参数，读入的数据到第三个参数截止。

```
    string start,end;
    getline(cin,start,';');                    //读到分号
    getline(cin,end);
```

(2)价钱计算

按照官网的计费规则计算价钱。

```
int fee(int distance)                          //计算费用
{
    if(distance==0)
        return 0;
    int d=distance/1000;
    if(d<=6)
        return 3;
    else if(d<=12)
        return 4;
    else if(d<=22)
        return 5;
    else if(d<=32)
        return 6;
    else
        return 6+(d-32)/20;
}
```

四、实验与测试

1.测试代码调整

为了方便测试，我们在代码中加入了路径输出，让其打印出从起点到终点的最短路径，途径的车站和其他辅助检验是否正确的信息（提交的代码中没有）。

定义一个 pre 数组，用于储存从起点到达某个结点 i 途径的上一个顶点的编号。Dijkstra 算法的特点就是下一个结点的最短路径是在其上一个结点最短路径

的基础之上建立的。所以从后向前，找到每个结点的前一个结点，直至起点，这样就得到了从起点到终点途径的所有结点。

```
int pre[nodeNum+2];                                    //存储前驱路径
```

在使用 Dijkstra 算法过程中，每一次更新最短距离，同时更新其前一个结点的信息。

```
for(int i=head[r];i;i=e[i].next)                       //遍历所有与 r 相连的点
       {
           int t=e[i].to;
           if(dis[t]>dis[r]+e[i].length)               //更新距离
           {
               dis[t]=dis[r]+e[i].length;
               q.push((node){t,dis[t]});               //将 t 压入队列
               //其他内容同算法与分析中所写，下面这行是添加的
               pre[t]=r;                               //更新前一个结点
           }
       }
```

由于是从后向前找途径的结点，那么它们是倒序的，不能直接输出。那么就有先找到的结点最后输出，后找到的结点先输出。这里选择使用了栈存储结点信息。直接使用 C++标准模板库的 stack。

```
void print(int e)
{
    for(int i=e;i;i=pre[i])                            //找前驱结点
        route.push(i);                                 //加入栈中
    while(route.size())                                //栈空了，说明所有结点都已输出
    {
        cout<<stationName[route.top()]<<endl;
        route.pop();                                   //输出之后，将其弹出栈
    }
}
```

其他信息的输出。

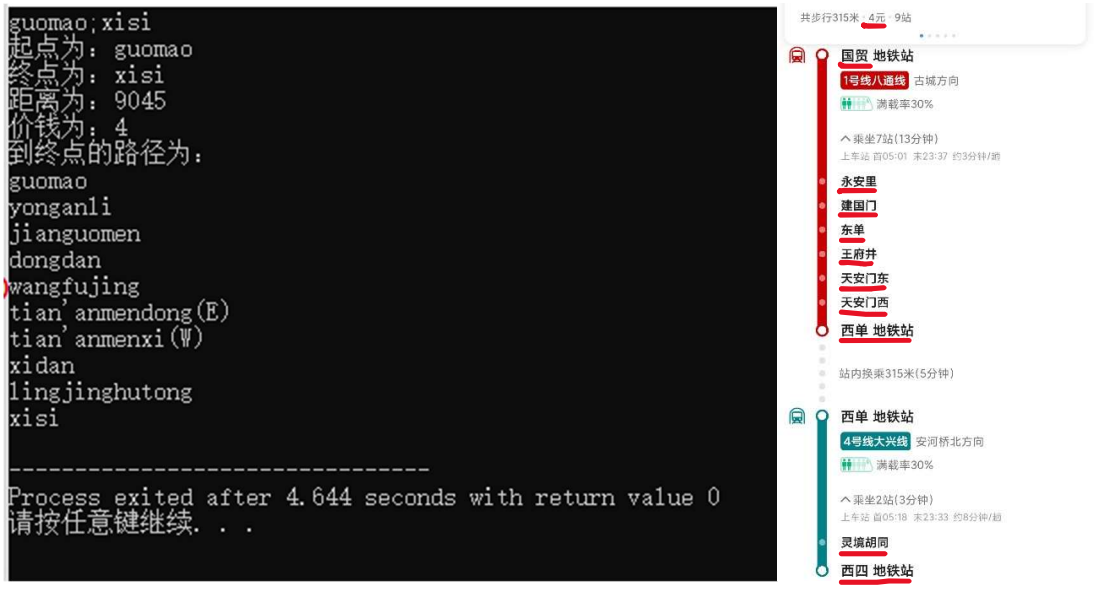
```
    cout<<"起点为："<<start<<endl;
    cout<<"终点为："<<end<<endl;
    cout<<"距离为："<<dis[en]<<endl;
    cout<<"价钱为："<<fee(dis[en])<<endl;
    cout<<"到终点的路径为："<<endl;
    print(en);
```

2.代码测试

我们选取了三组具有代表性的数据进行测试。

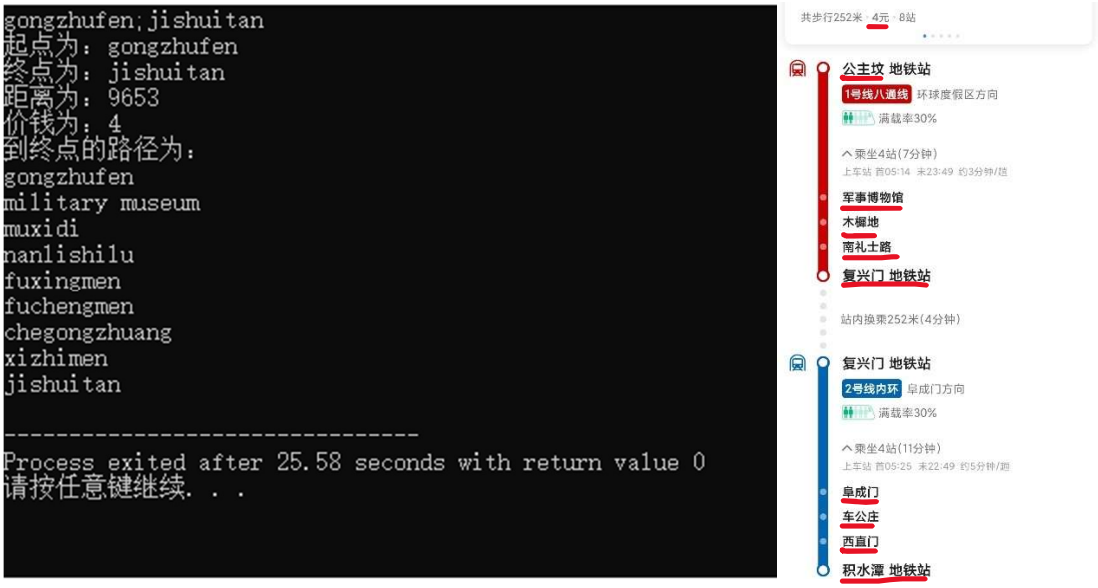
1）国贸->西四

程序输出票价 4 元。途径国贸、永安里、建国门、东单、王府井、西安门东、西安门西、西单、灵境胡同、西四。与地图软件搜索结果相同。



2）公主坟->积水潭

程序输出票价 4 元。途径公主坟、军事博物馆、木樨地、南礼士路、复兴门、阜成门、车公庄、西直门、积水潭。与地图软件搜索结果相同。



3）灯市口->灵境胡同

程序输出票价 3 元。途径灯市口、东单、崇文门、天安门东、天安门西、西单灵境胡同。与地图软件搜索结果相同。