武汉大学实习报告

院系：遥感信息工程学院　专业：遥感科学与技术

实习名称	数据结构与算法课程实习					指导教师	邬建伟
姓名	马文卓	年级	2020级	学号	2020302131249	成绩	

实习报告包括：

一、实习目的

本次实习课程旨在对理论课程回顾练习的同时加强同学们的分析较复杂问题能力、动手写代码能力和独立调试代码能力，以达到积累编程经验、清晰编程思路、将课本和实践有机联系起来的效果。

二、实习的主要内容（按以下几点具体简单解释展开）

1）CSV格式数据文件的读写

本次课题文件资源有两个：199个城市文件、1975个路线文件（文件格式为csv文件）。实习过程中首先是对于csv文件中数据的读取和存储。我通过c语言中的文件操作函数来实现这一功能。把读取的数据放在数组中以方便后续使用。

2）图的创建（邻接矩阵或邻接表）

为了达到存储数据和数据之间关系的目的，我们需要一个图载体来存储。这里我选择临接矩阵的方式来存储这个有向图。这里由于有时间和成本两个判据，所有有必要构建两个临接矩阵，分别存储时间最优和成本最少。

3）图的遍历（深度优先）

存储了数据之后，我们通过图的深度优先遍历将城市数据信息输出。这我们通过两个函数DFS、DSFTraverse实现深度优先遍历。其中DFS为深度优先遍历子函数，而DFSTraverse为整张图的深度优先遍历函数。

4）图的最短路径，并具体给出（A到B）的最短路径及其数值

有向图构建好之后就可以通过迪杰斯特拉算法来求时间最短路径Shortest_Path_DIJ_Time和成本最短路径Shortest_Path_DIJ-Cost.其具体思路下文会详细讲解。

5）最短路径的地图可视化展示

求完最短路径之后可以把最短路径写到html文件里面，最后放到百度地图中达到可视化的目的，使最短路径清晰明了。

6）算法的时间复杂度分析

最后对算法的时间复杂度进行分析计算，优化程序。

三、实习方法与技术路线（详细介绍展开）

1）算法原理

A、临接矩阵的创建原理：动态分配一个顶点数 x 顶点数的二维数组作为矩阵来存储有向图。先把矩阵中出对角线外的元素全部设置为 MAXN 表示暂无路径，对角线的元素设置为 0，因为自身到自身的成本为 0。然后再矩阵中填充路线，以行下标为起点，列下标为终点，把 1975 条路线的成本和时间填入其中。如果遇见两个城市之间有多条路径，则按照最优判据填入时间/成本少的那条路径。

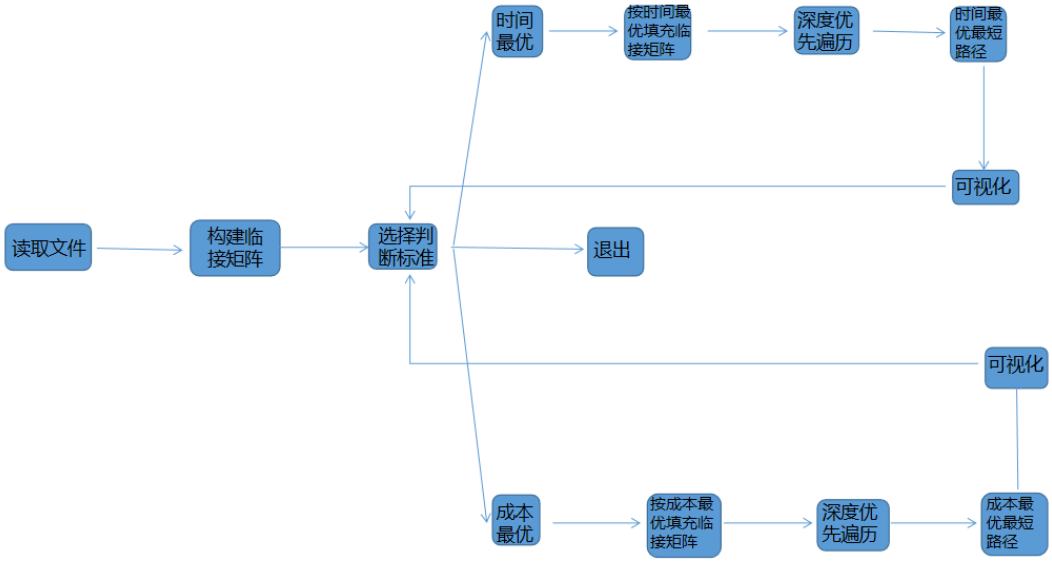
B、深度优先遍历原理：设置一个访问标记数组，来标志顶点是否已经被访问，最开始全部设为 false，一个顶点被遍历之后，其标志设为 true。算法大致思路是从起点开始进行深度优先遍历，如果他的临接点没有被访问过，就递归调用深度优先遍历子函数。当一个连通子图中所有的顶点全部被遍历后，通过整张图的深度遍历函数从下一个连通子图的一点开始进行深度优先遍历，直到所有的顶点都被遍历完成。

C、迪杰斯特拉算法原理：用 final 数组标志亥顶点是否已经求出最短路径；用 minPath 数

组储存最短路径（前驱储存法）；用 minAdj 数组储存各点到起点的最短距离。先初始化，把除起点外的顶点的 final 数组设为 false，起点可以直接到达的顶点的 minPath 设置为起点下标，其他顶点的 minPath（前驱）设为-1 表示还没有找到最短路径，把有与起点相联通的路径的顶点的 minAdj 设为亥路径的权值，没有路径的设置为 MAXN。再求最短路径，对剩下的 n-1 个点循环，寻找其中离起点最近的顶点，把它加入到最短路径中（final=true），更新剩余各点到起点的最短距离（如果如果亥点到刚加入的点的直线距离加上刚加入点到起点的最短距离小于亥点的 minAdj，则更新 minAdj），到达终点时结束算法。

2）算法的模块化设计与实现（提供设计的思路与相应的数据结构定义等，可以适当提供算法设计流程图等）

下面是算法设计流程图：



下面是算法设计思路和数据结构定义概述：

1. 基本数据类型定义：

a、城市City：

```
typedef struct City {//定义顶点City
    string country;//国家
    string city;//城市
    double latitude = 0;//维度
    double longitude = 0;//经度};
```

b、路线Route：

```
typedef struct Route {//定义弧Route
    string origin_city;//起点城市
    string destination_city;//终点城市
    string transport;//交通方式
    double time = 0;//通行时间
    double cost = 0;//成本
    string other_information;//其他信息
};
```

C、有向图Gragh：

```
typedef struct Graph {//定义有向图
    City* vertex;//顶点数组
    Route** arc;//邻接矩阵
    int vexnum = 0, arcnum = 0;//顶点数，弧数
};
```

2. 设计思路

A、读取文件

从用户使用程序的需求来分析，由于每次读取的文件中记录数目是不同的，所以我们要实现程序自己判断文件中记录个数的多少。这种功能我们可以通过两次读取文件实现，第一次读取文件用来获取文件记录数目，第二次读取文件用来获取数据。

这里使用fopen函数打开文件，判断文件是否打开成功，然后遍历文件每读一行就将num1++，就可以获取记录数目。需要注意的是，第一次读取完成之后，文件指针已经到达文件尾部，为了下一次读取成功，需要使用rewind（）函数把文件指针指向文件首部。第二次读取，采用fgetc和fscanf函数来读取，读取时一定要注意‘，’和‘\n’的读取。分别用ct数组和route数组存储数据。示例代码如下：

```
FILE* fp1;
    fp1 = fopen("cities.csv", "r");

    if (!fp1) {//保证文件正常打开
        cout << "无法打开文件" << endl;
        exit(0);
    }
    City ct[MAXN];
    int num1;
    int i = 0;
    char ch;
    num1 = 0;
    while (!feof(fp1)) {//判断文件记录个数
        char em[1000];
        fgets(em,1000,fp1);
        ch = fgetc(fp1);
        num1++;
    }
    rewind(fp1);
    while (!feof(fp1)) {

        if (i == num1) break;//到达指定个数停止读取
        ch = fgetc(fp1);
        for (;ch != ',';ch = fgetc(fp1)) {//读取国家
            ct[i].country += ch;
        }
        ch = fgetc(fp1);
        for (;ch != ',';ch = fgetc(fp1)) {//读取城市
            ct[i].city += ch;
        }
        fscanf(fp1, "%lf,%lf\n", &ct[i].latitude, &ct[i].longitude);//读取经纬度 记得读取换行符
        i++;
    }
    fclose(fp1);//关闭文件
```

B、构建有向图和临接矩阵

由于数据数量的不确定性，我们采用动态分配的方式来构建矩阵。矩阵分配之后我们将对角线上的权值设为0，其他的元素权值设为MAXN。我们构建的有向图g，g.vertex来存储ct数组也就是顶点信息，g.arc来存储矩阵也就是

路线信息,g.vexnum和g.arcnum存储顶点数和弧数。示例代码如下：

```
Graph g;//构建一个有向图g
    //初始化
    g.vexnum = num1;//存入顶点数
    g.arc = new Route * [num1];
    for (int i = 0;i < num1;i++) g.arc[i] = new Route[num1];//动态分配一个矩阵
    g.arcnum = num2;//存入边数
    g.vertex = ct;//导入顶点数据
    for (int i = 0;i < num1;i++) {
        for (int j = 0;j < num1;j++) {
            if (i == j) {//对于对角线上的表示从本地出发到本地 时间和成本都为0
                g.arc[i][j].time = 0;
                g.arc[i][j].cost = 0;
            }
            else {//否则 全部设为最大MAXN
                g.arc[i][j].time = MAXN;
                g.arc[i][j].cost = MAXN;
            }

        }
        //cout << endl;
    }
```

C、选择判断标准

根据使用的需求分析，我们肯定希望获取两个地方的一种判据下的最短路径之后还能继续获取另外一种判据下的的最短路径，或者是获取另外两个城市的最短路径，所以我们必须建立循环和选择机制，我们用goto语句实现这个功能。

```
loop:
    read("您希望的判断标准是什么  是时间还是花费");
    cout << "您所希望的路径最短评判标准是什么？ 1.时间  2.花费 3.退出" << endl;
    cout << "请做出选择：";
    int pan = 0;
    cin >> pan;
    if (pan != 1 && pan != 2&&pan!=3) { cout << "请输入正确的选择" << endl;goto loop;}
    if (pan == 1){...}
    if (pan == 2){...}
```

D、按照判据填充临接矩阵（后面步骤均以时间最短为例）

现在明确了判据，我们只需要把route数组中的路线信息填充到临接矩阵中就可以获得一个完整的路线临接矩阵了这其中涉及到一个定位函数Locate，其作用为在顶点数组中定位给定的城市名称的城市下标。

示例代码如下：

```
 for (int i = 0;i < g.arcnum;i++) {
            int origin = -1, destination = -1;//保存起点和终点
            origin = Locate(route[i].origin_city, ct, num1);//定位坐标
            destination = Locate(route[i].destination_city, ct, num1);//假定 矩阵的行坐标为origin
列坐标为destinaton
            if (g.arc[origin][destination].cost > route[i].cost)//花费最优
                g.arc[origin][destination].cost = route[i].cost;//将花费赋值到里面
            g.arc[origin][destination].time = route[i].time;
        }
//定位函数如下
int Locate(string s, City* ct, int num1) {//定位函数，在city[]数组中寻找s城市 返回其在city中的下标
如果没有这个城市 返回-1
```

```
    for (int i = 0;i < num1;i++) {
      // cout << i << endl;
       if (s == ct[i].city) return i;

    }

    return -1;
}
```

E、深度优先遍历

在创建好有向图之后我们可以根据上面所叙述的深度优先遍历原理来深度遍历这个有向图。这个算法的核心思想是递归。我们可以通过深度优先遍历子函数DFS和整张图的深度优先遍历函数DFSTraverse来实现。我们先设置一个访问标志数组visited[MAXN]来标志一个顶点是否被访问过。对于DFS，我们要实现的就是访问传入的顶点（输出顶点信息），然后把这个顶点的visited设置为true，最后遍历其他没有访问过的临接点，对他们递归调用DFS。而对DFSTraverse这个函数，我们首先要做的是把visited数组全部初始化为false，然后对剩下没访问过的顶点调用DFS。示例代码如下：

```
void DFS(Graph g, int v) {//深度遍历子函数
    visited[v] = true;//访问这个顶点并且标志已经遍历
    cout << g.vertex[v].city << "->";
    //cout << "国家：" << g.vertex[v].country << "  城市：" << g.vertex[v].city << "  维度：" <<
g.vertex[v].latitude << "  经度：" << g.vertex[v].longitude << endl;
    for (int j = 0;j < g.vexnum;j++) {//遍历每一个顶点的邻接点
        if (g.arc[v][j].cost < MAXN && g.arc[v][j].time < MAXN && !visited[j]) {//如果有i顶点通向j
顶点的路径并且没有被访问过 就再次进行深度遍历
            DFS(g, j);
        }
    }
}
void DFSTraverse(Graph g) {//深度优先遍历函数
    for (int i = 0;i < g.vexnum;i++) {//初始化访问标志数组
        visited[i] = false;
    }
    for (int i = 0;i < g.vexnum;i++) {//对于没有访问的顶点 进行深度优先遍历

        if (!visited[i]) {
            cout << endl;
            DFS(g, i);
        }
    }
}
```

F、最短路径

我们采用的是迪杰斯特拉算法来求最短路径。首先我们要做的是输入起点origin_和终点destination_，然后把它们转化为下标数字origin和destination（可以用上面的Locate函数实现），然后再通过Shortest_Patn_DIJ_Time函数来获取最短路径（反序的）和最短距离，最后再把最短路径正向输出。这其中Shortest_Path_DIJ_Time函数的实现在上面的算法原理中提到过。我们来详细论述一下：首先传入的参数中minPath数组用来存储最短路径（通过记录一个顶点的前驱实现，所以是反序的） minAdj数组用来存储每个顶点到起点的最短距离，再者设置一个final[MAXN]数组来标志一个顶点是否已经求出最短路径。然后开始初始化，把除起点外的顶点的final数组设为false，起点可以直接到达的顶点的minPath设置为起点下标，其他顶点的minPath（前驱）设为-1表示还没有找到最短路径，把有与起点相联通的路径的顶点的minAdj设为亥路径的权值，没有路径的设置为MAXN。再求最短路径，对剩下的n-1个点循环，寻找其中离起点最近的顶点，把它加入到最短路径中（final=true），更新剩余各点到起点的最短距离（如果如果亥点到刚加入的点的直线距离加上刚加入点到起点的最短距离小于亥点的minAdj，则更新minAdj），到达终点时结束算法。值得注意的我们如此得到的最短路径是反序的，我们还需要一个反转输出，这一点指针可以轻松完成。示例代码如下：

```
void Shortest_Path_DIJ_Time(Graph g,int origin,int destination,int *minPath,double *minAdj) {//迪杰斯特拉算法求时间最短路径
    //minPath数组储存最短路径上v的双亲 minAdj数组储存最短路径的权值
    bool final[MAXN];//标志是否已经求出最短路径
    for (int i = 0;i < g.vexnum;i++) {//初始化
        final[i] = false;//所有点都没有求出最短路径
        minAdj[i]=g.arc[origin][i].time;//最短路径为直线
        minPath[i] = -1;//设空路径
        if (minAdj[i] < MAXN) {
            minPath[i] = origin;//如果起点到其他顶点有路径 则亥为最短路径
        }
    }
    minAdj[origin] = 0;
    minPath[origin] = -1;
    final[origin] = true;//起点先进入最短路径
    int v;
    for (int i = 1;i < g.vexnum;i++) {//主循环 对剩下n-1个点操作 求最短路径
        double min = MAXN;
        for (int w = 0;w < g.vexnum;w++) {//求当前到起点最近的点
            if (!final[w] && minAdj[w] < min) {
                v = w;
                min = minAdj[w];
            }
        }
        final[v] = true;//将最近的点加入最短路径

        if (v == destination) break;//到达终点就停止
        for (int w = 0;w < g.vexnum;w++) {//更新各点到起点的距离
            if (!final[w] && (min + g.arc[v][w].time) < minAdj[w]) {//如果前面求得的最短路径加上直线距离小于原来的距离，则更新
                minAdj[w] = min + g.arc[v][w].time;
                minPath[w] = v;//储存路径
            }
        }
    }
}
int origin = 0, destination = 0;
        string origin_, destination_;//输入起点终点
        read("请输入起点");
        cout << "请输入起点：";
        cin.get();
        getline(cin, origin_);
        read("请输入终点");
        cout << "请输入终点：";
        getline(cin, destination_);
        QueryPerformanceFrequency(&Frequency);
        QueryPerformanceCounter(&BeginTime);//开始计时
        for (int i = 0;i < g.vexnum;i++) {//确定起点和终点的位置
            if (g.vertex[i].city == origin_) origin = i;
            if (g.vertex[i].city == destination_) destination = i;
        }
        double minAdj[MAXN];
```

```
        int minPath[MAXN];
        Shortest_Path_DIJ_Time(g, origin, destination, minPath, minAdj);//求时间最短路径
        QueryPerformanceCounter(&EndTime);
        read("您的最短时间花费和具体路线如下");
        cout << "您最短的时间花费为：" << minAdj[destination] << "h" << endl;
        cout << "具体路线为:";
        int path[MAXN];
        int* p = path;//存储最短路径
        int k = destination;
        *p = k;
        while (1) {//沿着前驱往上走 知道找到起点为止
            if (k == origin) break;
            p++;
            *p = minPath[k];
            k = minPath[k];
        }
        int zheng[MAXN];//正向存储的路径
        int ji = 0;
        for (;p >= path;p--) { //输出路径
            zheng[ji++] = *p;
            cout << g.vertex[*p].city;
            if (p != path) cout << "->";
        }
        cout << endl;
        cout << "算法运行时间：" << double(EndTime.QuadPart-BeginTime.QuadPart)/Frequency.QuadPart
<< "s" << endl;
        system("pause");
```

G、可视化

根据用户的需求分析，我们发现用户是希望他输入起点终点之后就能名确的知道最短路径以及路过城市的信息的，他肯定不希望得到的只是一个黑色框框中的几行代码。所以为了满足这种便捷化的要求，我们有必要实现最短路径的可视化。我们可以通过把最短路径写入html文件当中，再显示到百度地图上面，这样就一目了然了。这里要注意，由于上面得到的path是反序的，所以我们还需要反转最短路径数组path，得到正向的最短路径zheng。示例代码如下：

```
 //创建一个html文件，将百度API可视化所需要的代码和信息写到里面
        FILE* fp3;
        fp3 = fopen("graph.htm", "w");
        if (!fp3) {
            cout << "无法打开文件" << endl;
            exit(1);
        }
        //HTML文件头部说明
        char a[1024], b[1024];//储存起点和终点
        strcpy(a, origin_.data());
        strcpy(b, destination_.data());

        std::string s = "<!DOCTYPE html><html><head>\
<style type='text/css'>body, html{width: 100%;height: 100%;margin:0;font-family:'微软雅黑
';}#allmap{height:100%;width:100%;}#r-result{width:100%;}</style>\
<script type='text/javascript'
src='http://api.map.baidu.com/api?v=2.0&ak=nSxiPohfziUaCuONe4ViUP2N'></script>";

        fprintf(fp3, "%s<title>Shortest path from %s to %s</title></head><body>\
```

```
<div id='allmap'></div></div></body></html><script type='text/javascript'>\
var map = new BMap.Map('allmap');\
var point = new BMap.Point(0, 0);\
map.centerAndZoom(point, 2);\
map.enableScrollWheelZoom(true);", s.c_str(), a, b);

        i = 0;
        int j = 0;
        int bycity = origin, bycity2;

        while (bycity != destination) {
            //路径中一条弧弧头及其信息
            fprintf(fp3, "var marker%d = new BMap.Marker(new BMap.Point(%.4f, %.4f));\
map.addOverlay(marker%d);\n\
var infoWindow%d = new BMap.InfoWindow(\"<p style = 'font-
size:14px;'>country: %s<br/>city: %s</p>\");\
marker%d.addEventListener(\"click\", function(){\
this.openInfoWindow(infoWindow%d);}); ", j, g.vertex[bycity].longitude, g.vertex[bycity].latitude,
j, j, g.vertex[bycity].country.c_str(), g.vertex[bycity].city.c_str(), j, j);
            j++;
            bycity2 = bycity;
            bycity = zheng[i++];
            //路径中一条弧弧尾及其信息
            fprintf(fp3, "var marker%d = new BMap.Marker(new BMap.Point(%.4f, %.4f));\
map.addOverlay(marker%d);\n\
var infoWindow%d = new BMap.InfoWindow(\"<p style = 'font-
size:14px;'>country: %s<br/>city: %s</p>\");\
marker%d.addEventListener(\"click\", function(){\
this.openInfoWindow(infoWindow%d);}); ", j, g.vertex[bycity].longitude, g.vertex[bycity].latitude,
j, j, g.vertex[bycity].country.c_str(), g.vertex[bycity].city.c_str(), j, j);

            //路径中一条弧的信息
            fprintf(fp3, "var contentString0%d = '%s, %s --> %s, %s (%s - %.0f hours - $%.0f -
 %s)';\
var path%d = new BMap.Polyline([new BMap.Point(%.4f, %.4f),new BMap.Point(%.4f, %.4f)],
{strokeColor:'#18a45b', strokeWeight:8, strokeOpacity:0.8});\
map.addOverlay(path%d);\
path%d.addEventListener(\"click\", function(){\
alert(contentString0%d);});", j, g.vertex[bycity2].city.c_str(), g.vertex[bycity2].country.c_str(),
g.vertex[bycity].city.c_str(), g.vertex[bycity].country.c_str(),
g.arc[bycity2][bycity].transport.c_str(), g.arc[bycity2][bycity].time, g.arc[bycity2][bycity].cost,
g.arc[bycity2][bycity].other_information.c_str(), j, g.vertex[bycity2].longitude,
g.vertex[bycity2].latitude, g.vertex[bycity].longitude, g.vertex[bycity].latitude, j, j, j);
        }
        fprintf(fp3, "</script>\n");//html文件结束
        ShellExecute(0, L"open", L"graph.htm", 0, 0, 1);//自动打开hml文件
        fclose(fp3);
        goto loop;
    }
```

H、细节&两点

a、为了避免每次获取完最短路径手动打开文件，我实现了程序自动打开html文件。示例代码如下：

```
ShellExecute(0,L"open",L"graph.htm", 0,0,1);//自动打开hml文件
```

b、为了方便分析算法的速度，我加入了计时功能。示例代码如下：

```
LARGE_INTEGER BeginTime;
LARGE_INTEGER EndTime;
LARGE_INTEGER Frequency;
QueryPerformanceFrequency(&Frequency);
        QueryPerformanceCounter(&BeginTime);//开始计时
        for (int i = 0;i < g.vexnum;i++) {//确定起点和终点的位置
            if (g.vertex[i].city == origin_) origin = i;
            if (g.vertex[i].city == destination_) destination = i;
        }
        double minAdj[MAXN];
        int minPath[MAXN];
        Shortest_Path_DIJ_Time(g, origin, destination, minPath, minAdj);//求时间最短路径
          QueryPerformanceCounter(&EndTime);
  cout << "算法运行时间：" << double(EndTime.QuadPart-BeginTime.QuadPart)/Frequency.QuadPart << "s"
  << endl;
```

c、为了让程序变得生动人性化，我加入了语音提示功能。示例代码如下：

```
void MSSSpeak(LPCTSTR speakContent) {
    ISpVoice* pVoice = NULL;//初始化COM接口
    if (FAILED(::CoInitialize(NULL))) MessageBox(NULL, (LPCWSTR)L"COM接口初始化失败", (LPCWSTR)L"提
示", MB_ICONWARNING | MB_CANCELTRYCONTINUE | MB_DEFBUTTON2);//获取SpVoice接口
    HRESULT hr = CoCreateInstance(CLSID_SpVoice, NULL, CLSCTX_ALL, IID_ISpVoice, (void**)&pVoice);
    if (SUCCEEDED(hr)) {
        pVoice->SetVolume((USHORT)100);//设置音量在0-100
        pVoice->Speak(speakContent, 0, NULL);
        pVoice->Release();
        pVoice = NULL;
    }
    ::CoUninitialize();

}
std::wstring StringToWString(const std::string& s) {
    std::wstring wszStr;
    int nLength = MultiByteToWideChar(CP_ACP, 0, s.c_str(), -1, NULL, NULL);
    wszStr.resize(nLength);
    LPWSTR lpwszStr = new wchar_t[nLength];
    MultiByteToWideChar(CP_ACP, 0, s.c_str(), -1, lpwszStr, nLength);
    wszStr = lpwszStr;
    delete[] lpwszStr;
    return wszStr;
}
void read(string temp) {
    wstring a = StringToWString(temp);
    LPCWSTR str = a.c_str();
    MSSSpeak(str);
  }
  read（“HelloWorld”）;
```

3）算法复杂度分析：从时间和空间两个角度分析评价（主要分析迪杰斯特拉算法）

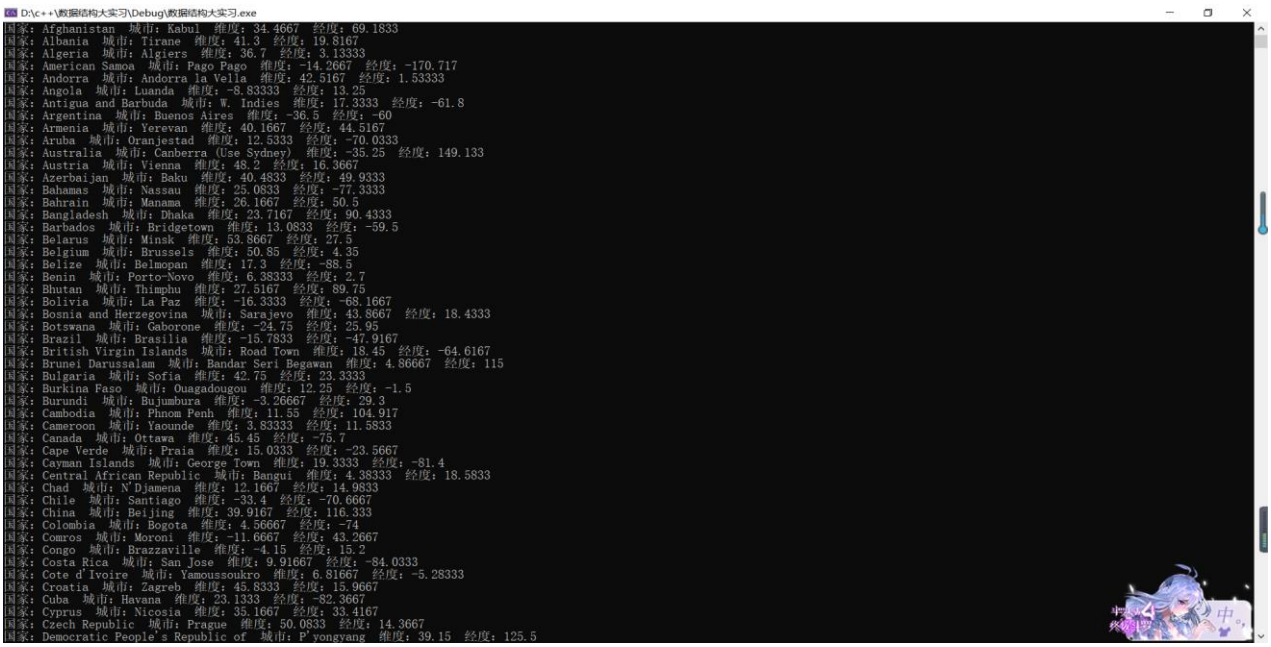
时间复杂度 O（n^2）：因为在算法中涉及到双层循环（求最近点和更新 minAdj 的时候）

空间复杂度 S（n）：算法中设置了一个外部内存数组 final

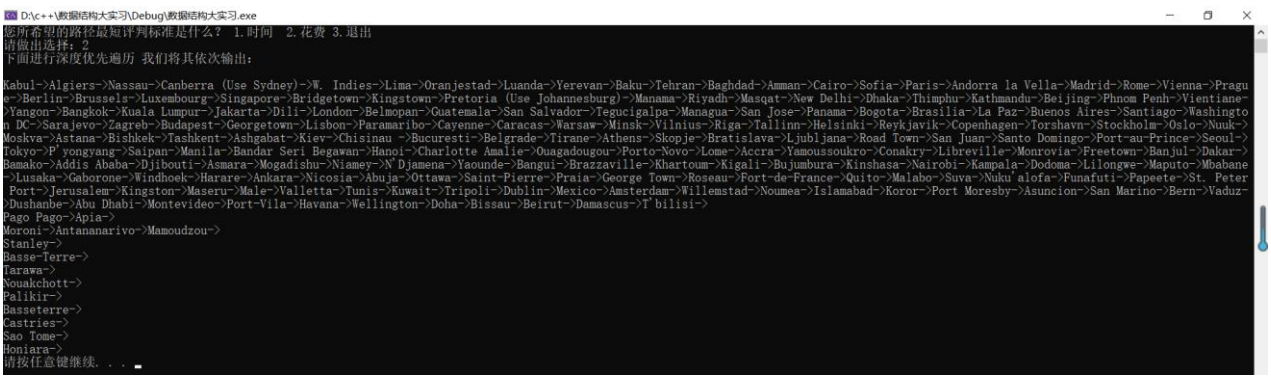
四、实习成果

程序不同功能的实测结果（注意提供相应的测试界面与结果截图）

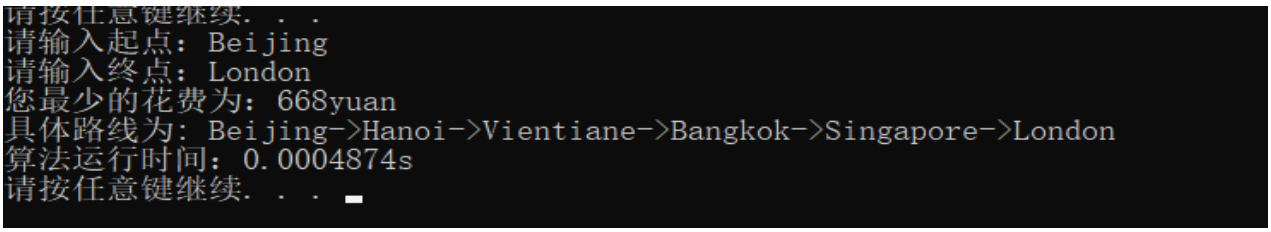
1. 读取文件（只显示了部分结果）



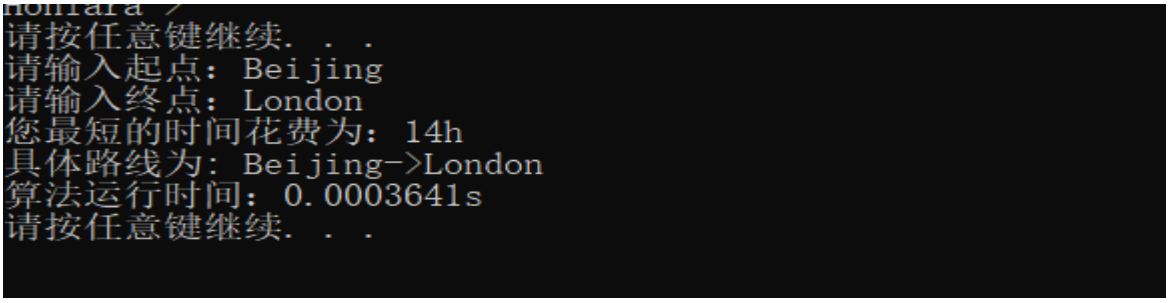
2. 图的创建和深度优先遍历



3. 最短路径

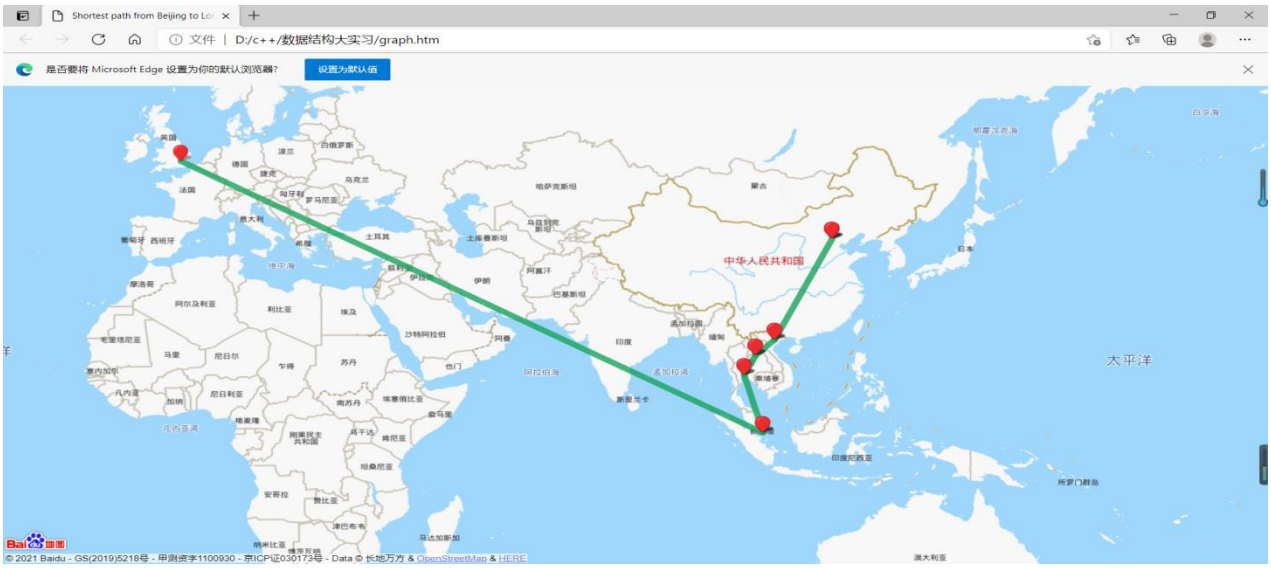


（上图为成本最短时的最短路径）

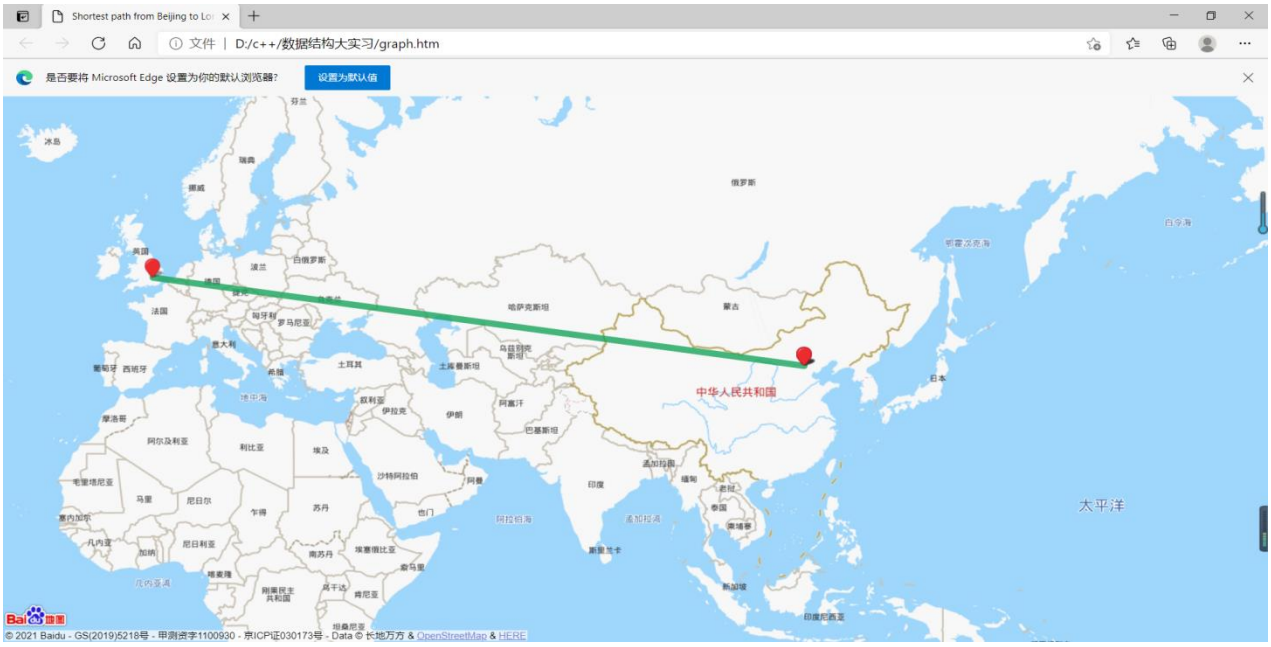


（上图为时间最短下的最短路径）

4. 可视化



（上图为成本最少时的最短路径可视化）



（上图为时间最短的最短路径的可视化）

五、实习结论与总结（包括心得体会）

本次实习过后我收获的不紧紧是一个程序的完工，更多的是经验的积累和能力的长进。下面我来谈谈通过本次实习我的收获和结论：

1. 对于一个问题，我们首先要做的的不应该是上手写代码，而是分析问题，做好需求分析，做好程序框架。只有在大方向确定和整体框架完成的情况下才可以开始写程序。
2. 对于需求去分析，我们要以使用者的身份来揣测需求，只有这样才能够做出面向用户的好程序。例如，上面提到的自动打开文件、自动判断记录个数、循环的选择机制、时间的统计，这些都是对算法没有任何优化的东西，但是它们在程序与用户的交互中起到了巨大的作用。
3. 对于文件读取一定基于文件格式进行读取。其中的分隔符和换行符都要进行充分的考虑。与此同时，文件多次读取的时候一定要关注文件指针的位置，否则可能造成意想不到的错误。
4. 对于算法而言，我们应该事先思考算法的时间复杂度和空间复杂度，看是否达到了我们的要求。对于同一种算法，我们也可以进行不同的优化，使其更上一层楼。
5. 对于写程序而言，不一定要弄懂每一行代码（在如今开源模式下，可以有很多的借鉴），但是你一定要有清晰的头脑（一定要明白你是写了干什么的），也一定要有一定的创新和亮点（无论是算法还是形式上，只有这样才能脱颖而出）。
6. 调试程序时写代码的必修课，合理运用vs工具和逻辑思路判断错误来源是一个程序员能力的体现。

教师评语

指导教师_____ 2021年_____月_____日