武漢大學
WUHAN UNIVERSITY

《数据结构与算法课程实习》
实习报告

学院：遥感信息工程学院

专业：遥感科学与技术

指导教师：杨宗亮

姓名：殷雨薇

班级：2010班

学号：2020302131119

引言：随着人们物质生活水平的不断提高，综合交通运输网络日趋完善，人们可以去到更多更远的地方。在此背景下，获取到达目的地的最优路径成为人们的关键需求。在这次实习中，我们依据所给的数据，构造出带权有向图，根据交通网络图中顶点间的路线时间、路线费用权值属性，通过 Dijkstra 算法，解决有向图中源点到其他顶点的最短路径问题。

一、实习目的：

1、了解 Dijkstra 算法概念，掌握求最短路径的基本思路。
2、深化对数据结构的理解，提升代码能力。
3、能够利用计算机计算人们从出发地到目的地所花费的时间或金钱，得到最优路径，解决实际问题，为人们出行提供便利。

二、实习内容：

1、CSV格式数据文件的读写
2、图的创建（邻接矩阵或邻接表）
3、图的遍历（广度优先或深度优先）
4、图的最短路径，并具体给出（A到B）的最短路径及其数值
5、最短路径的地图可视化展示
6、 算法的时间复杂度分析

三、实习要求：

1、每个人必须完成1）、2）、4）三 种算法；
2、3）、5）选一个
3、按照“数据结构与算法”课程要求，进行规范的数据结构、算法、以及ADT设计，并进行算法的时间复杂度分析和实际统计，算法、代码注释清晰易读

四、实习方法或技术路线：

1、算法原理

我们先将实际问题抽象为我们意识中的数学模型——有向图。把每个城市视为图的顶点，如果两个城市之间有已经提供的路线信息，便在图的这两个顶点添加弧。当所有的城市信息、路线信息全部读取完毕后，有向图便创建完成了。然后，我们将有向图再进一步进行抽象，用计算机可以接受的邻接表或邻接矩阵的方式存储。

关于算法，这里采用的是迪杰斯特拉算法和深度优先遍历法。

迪杰斯特拉算法是用来寻找从某一顶点出发、到达其余各个顶点的最短路径，是典型最短路径算法。它的主要特点是以起始点为中心向外层层扩展(广度优先搜索思想)，直到扩展到终点为止。算法本身并不是按照我们的思维习惯——求解从原点到第一个点的最短路径，再到第二个点的最短路径，直至最后求解完成到第n个点的最短路径，而是求解从原点出发的各有向路径的从小到大的排列。

深度优先遍历采用的搜索方法的特点是尽可能先对纵深方向进行搜索。它需要从某个顶点出发，先访问此顶点，再依次从其未被访问的邻接点出发深度优先遍历图，直至该路径没有可以继续访问的邻接点（不撞南墙不回头），则回退到上一个顶点。若该顶点仍有邻接点未被访问，则继续对其进行深度优先遍历。通过递归调用，直至图中每个顶点都被访问到为止。

最后，利用html语言代码，对得到的最短路径在网页地图上进行地图可视化展示。

2、算法的模块化设计与实现

（1）设计结构体，读取CSV格式数据文件，存储原始数据

首先设计了City和Route两个结构体，分别用于存储城市信息及路线信息。为接下来对CSV格式数据文件的读取做准备。

```
//建立一个结构City
struct City
{
    bool pass;  //判断路径是否经过该城市
    string country, city;  //国家名，城市名
    float latitude, longitude;  //纬度，经度
};

//建立一个结构Route
struct Route
{
    string origion_city, destination_city,  //出发地，目的地
        transport, other_information;  //交通工具，信息地址
    float time, cost;  //该段路线所花费的时间，金钱
};
```

通过查询资料了解到，.csv文件以纯文本形式存储表格数据（数字和文本）。纯文本意味着该文件是一个字符序列，不含必须像二进制数字那样被解读的数据。每条记录由字段组成，字段间的分隔符是其它字符或字符串，最常见的是逗号或制表符。例如下图是cities.csv中三条记录：

```
Chile,Santiago,-33.4,-70.6667
China,Beijing,39.9167,116.333
Colombia,Bogota,4.56667,-74
```

	A	B	C	D
1	Afghanista	Kabul	34.4667	69.1833
2	Albania	Tirane	41.3	19.8167
3	Algeria	Algiers	36.7	3.13333
4	American S	Pago Pago	-14.2667	-170.717
5	Andorra	Andorra la	42.5167	1.53333
6	Angola	Luanda	-8.83333	13.25
7	Antigua an	W. Indies	17.3333	-61.8
8	Argentina	Buenos Air	-36.5	-60
9	Armenia	Yerevan	40.1667	44.5167
10	Aruba	Oranjestac	12.5333	-70.0333
11	Australia	Canberra	-35.25	149.133
12	Austria	Vienna	48.2	16.3667

cities.csv 中的部分城市信息

	A	B	C	D	E	F	G
1	Abu Dhabi	Canberra	plane	24	1339	Qatar Airways, Thai	
2	Abu Dhabi	Lima	plane	30	1967	KLM Airways, 1 stop	
3	Abu Dhabi	London	plane	10	666	Turkish Airlines, 1	
4	Abu Dhabi	Manama	plane	1.25	130	Direct Flight, KLM	
5	Abu Dhabi	Masqat	bus	7	30	Oman National Trans	
6	Abu Dhabi	Masqat	plane	1	122	Direct flight, Etih	
7	Abu Dhabi	Pretoria	plane	16	714	Egyptair, 1 stop, M	
8	Abu Dhabi	Riyadh	plane	3	172	Gulf Air, 1 stop, w	
9	Abu Dhabi	Singapore	plane	11	545	SriLankan Airlines,	
10	Abu Dhabi	Washingtor	plane	14	585	Direct flight, Momo	
11	Abuja	Cairo	plane	16	971	http://www.kayak.co	
12	Abuja	N'Djamena	bus	0.25	0.4	COPY: http://www.lo	

routes.csv 中的部分路径信息

在.csv文件中，城市信息从左到右依次为：国家，城市，纬度，经度；路径信息从左到右依次为：起始地，目的地，出行方式，出行时间（h），出行花费，信息地址。

因此，以读取cities.csv文件为例，先用fopen打开文件，在读入文件中的城市信息时，通过fgetc()函数，将字符一个一个读入当前城市的country数组，直至遇到分隔符‘，’再进行下一个信息（比如：city)的读取。数字则直接用fscanf()函数读取。需要注意的是，每一行数据以‘\n’结尾，而不是‘，’。

```
//读取文件数据
FILE* fp=NULL;
fp = fopen("D:\\学习\\大一下yyw\\数据结构与算法\\实习\\cities.csv", "r");
if (fp == NULL)  //判断异常
{
    printf("error1!");
    exit(0);
}
char ch;  //控制循环
float f;
int i = 0, j;
City ct[199];

while (!feof(fp))  //读至文件末尾的条件
{
    ch = fgetc(fp);
    for (; ch != ','; ch = fgetc(fp))  //以‘,’为界，逐个字符读国家名并存入
    {
        ct[i].country += ch;
    }
    ch = fgetc(fp);
    for (; ch != ','; ch = fgetc(fp))  //以‘,’为界，逐个字符读城市名并存入
    {
        ct[i].city += ch;
    }
    fscanf(fp, "%f,", &f);  //读维度
    ct[i].latitude = f;
    fscanf(fp, "%f\n", &f);  //读经度
    ct[i].longitude = f;
    i++;  //为读取下一个city信息做准备
}
fclose(fp);  //关闭文件
```

（2）利用读取的数据，用邻接矩阵进行图的创建

这里采用了邻接矩阵的方法创建图，虽然有199个顶点，只有1975条边，其占用的空间远大于其真正所需要的空间，是稀疏图。但邻接矩阵的存储方法比邻接表在时间上更节约，所以我选择牺牲部分空间，来换取时间上的节约。

```
//建立边
struct Arc
{
    float time;  //所花费的时间
    float cost;  //所花费的金钱
    string transport;  //交通工具
    string otherInfo;  //信息地址
};

//图的邻接矩阵
struct Graph
{
    City* city;  //指向城市的指针
    Arc arcs[199][199];  //邻接矩阵
    int vexnum, arcnum;  //定义顶点数，边数
};
```

接下来，便按照创建好的顶点及边，创建图。这里先定义了一个LocatVex()函数，传入城市的名称和ct[]数组便可获取城市在顶点数组中的序号。通过这一序号来代替具体的城市名称，在一定程度上可以减少算法所耗用的时间、空间。

```
//在图中查找顶点
int LocatVex(const string& origin, City ct[])
{
    int i = 0;
    for (; i < 199; i++)  //依次查找输入的城市的序号
    {
        if (origin == ct[i].city)  //输入城市的名称与第i+1个城市名称相同
            return i;  //返回i值
    }
    return N;  //若没找到，返回一个无穷大的数
}
```

在创建图的邻接矩阵过程中，利用LocatVex()函数得到出发地和目的地城市的序号，分别作为矩阵的行值和列值，把刚才从.csv文件中读取到的数据一一对应存入邻接矩阵中。

```
//建立图的邻接矩阵
for (i = 0; i < M; i++)
{
    int ori , des;
    ori = LocatVex(route[i].origion_city, ct);  //找到出发地城市的序号
    des = LocatVex(route[i].destination_city, ct);  //找到目的地城市的序号
    g.arcs[ori][des].cost = route[i].cost;  //将路线上的金钱权值存入矩阵中
    g.arcs[ori][des].time = route[i].time;  //将路线上的时间权值存入矩阵中
    g.arcs[ori][des].transport = route[i].transport;  //将路线的交通工具存入矩阵中
    g.arcs[ori][des].otherInfo = route[i].other_information;  //将路线的信息地址存入矩阵中
}
```

当遇到堆栈使用量超过预设阈值的问题时，通过在菜单栏中更改堆栈大小，即设置 stack reserve 为更大的值来解决这一问题。

（3）图的遍历（深度优先）

首先在函数外定义了一个visited[]数组，用于记录某顶点是否被访问过。当DFS()函数传入城市在顶点数组中的序号时，先给起点城市visited[]数组赋值TRUE，并输出该城市名称。再由此顶点出发，访问它的任一邻接顶点w1，访问过的城市的visited[]数组会被更改为TRUE；再从w1出发，访问与w1邻接但还未被访问过的顶点w2；然后再从w2出发，进行类似的访问，如此进行下去，直至到达所有的邻接顶点都被访问过的顶点u为止。接着，退回一步，退回到前一次刚访问过的顶点，看是否还有其它没有被访问(visited[]为FALSE)的邻接顶点。如果有，则访问此顶点，之后再从此顶点出发，进行与前述类似的访问；如果没有，就再退回一步进行搜索。通过递归调用DFS()函数重复上述过程，直到连通图中所有顶点都被访问过为止。

```
//邻接矩阵的深度优先递归算法遍历
void DFS(Graph &g, int r)
{
    int s;
    visited[r] = TRUE;  //遍历起点城市被访问，改变其visited的值为1
    cout<<g.city[r].city<<"->";  //输出遍历起点的城市名称

    for (s = 0; s < g.vexnum; s++)  //循环至遍历完所有顶点
    {
        if (g.arcs[r][s].time != 0 && g.arcs[r][s].time != N && visited[s]== FALSE)  //如果该点有路线没有访问，则对其进行访问
        {
            DFS(g, s);  //递归调用
        }
    }
}
```

（4）图的最短路径，并具体给出（A到B）的最短路径及其数值

迪杰斯特拉算法求出的是从图的某一个顶点出发，到达其余各个顶点的最短路径。通过引进两个集合S和U。①初始时，S只包含起点s；V包含除s外的其他顶点，且V中顶点的距离为“起点s到该顶点的距离”。②然后，从V中选出”距离最短的顶点k”，并将顶点k加入到S中；③同时，从V中移除顶点k。最后，利用k来更新V中各个顶点到起点s的距离。重复步骤②和③，直到遍历完所有顶点。

```
//Dijkstra算法求最短路径
void DIJ(const Graph& g, int depart, int dest, Way way[][199], float dist[][199],float min)
{
    for (int v = 0; v < g.vexnum; ++v) { ... }
    dist[depart][depart] = 0;
    g.city[depart].pass = true;    //出发城市pass标记为为true,其最短路径为0.此顶点以后不会再用到
    for (int i = 0; i < g.vexnum; ++i)  //开始主循环，每次求得到出发城市到某个城市的最短路径，并将该顶点添加到S中
    {
        int v = 0;
        double mincost = N;
        for (int w = 0; w < g.vexnum; ++w)  //找出当前源点到其余点的最短路径
        {
            if (g.city[w].pass == false)
                if (dist[depart][w] < mincost)  //如果w城市离出发城市更近，且当前城市在集合V-S中
                {
                    v = w;  //用v记录离出发城市更近的w城市的序号
                    mincost = dist[depart][w];  //用mincost记录权值（这里是时间）
                }
        }
        g.city[v].pass = true;  //更新v城市的pass状态
        for (int w = 0; w < g.vexnum; ++w) { ... }
        if (g.city[dest].pass == true) { ... }
    }
}
```

在这里，因为要找的是两个城市间的最短路径，不需要求出某城市到达全部城市的最短路径，所以依据实际问题，在判断出到达目的地城市的最短路径已经找到后，便可以直接结束函数并返回最短时间或最少花费（默认是最短时间，若想得到最少花费，则修改time为cost即可）。

此算法中用到的变量：

bool pass:用于记录最短路径是否经过该城市。

float dist[199][199]：这个数组用于记录权值（时间/金钱），函数执行结束后dist[i][j]表示从i城市到达编号为j的城市总共所需的时间/价格。

int *path：用于记录途径的城市信息，path[i]表示到达编号为i的城市所需经过的城市的编号。(初始化为-1)

（5）最短路径的地图可视化展示

因为之前没有接触过html语言，所以这一板块对我来说是最难的。通过 html 语言，将最短路径上的点和路线写成 html 文件，通过浏览器来识别，最终将这个 html 文件“翻译”成可视的网页。

3、算法复杂度分析与实测结果

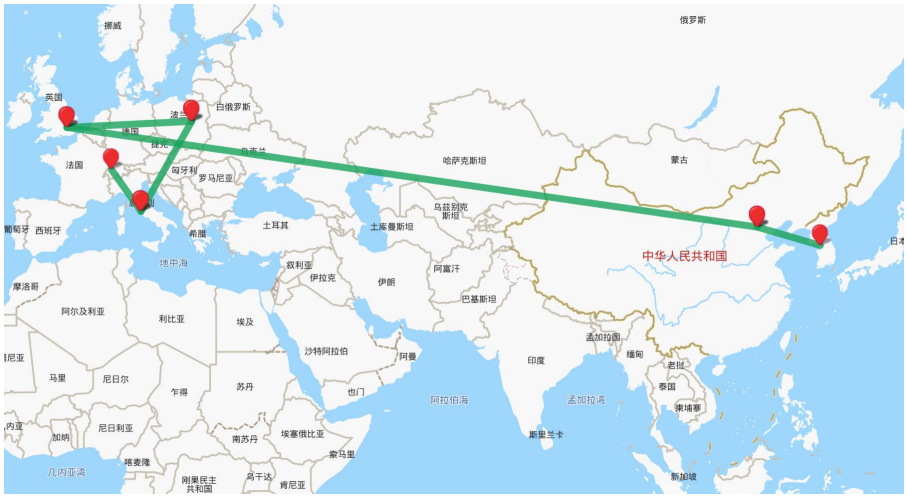
（1）深度优先遍历结果：

```
深度优先遍历:
请输入遍历起点:Beijing
Beijing->Canberra (Use Sydney)->W. Indies->Lima->Oranjestad->Luanda->Yerevan->Baku->Tehran->Kabul->Algiers->
Nassau->Havana->Singapore->Bridgetown->Kingstown->Pretoria (Use Johannesburg)->Manama->Riyadh->Baghdad->Amma
n->Cairo->Sofia->Paris->Andorra la Vella->Madrid->Rome->Vienna->Prague->Berlin->Brussels->Luxembourg->London
->Belmopan->Guatemala->San Salvador->Tegucigalpa->Managua->San Jose->Panama->Bogota->Brasilia->La Paz->Bueno
s Aires->Santiago->Washington DC->Sarajevo->Zagreb->Budapest->Georgetown->Lisbon->Paramaribo->Cayenne->Carac
as->Warsaw->Minsk->Vilnius->Riga->Tallinn->Helsinki->Reykjavik->Copenhagen->Torshavn->Stockholm->Oslo->Nuuk-
>Moskva->Astana->Bishkek->Tashkent->Ashgabat->Kiev->Chisinau ->Bucuresti->Belgrade->Tirane->Athens->Skopje->
Bratislava->Ljubljana->Road Town->San Juan->Santo Domingo->Port-au-Prince->Seoul->Tokyo->P'yongyang->Saipan-
>Manila->Bandar Seri Begawan->Hanoi->Phnom Penh->Vientiane->Yangon->Dhaka->Thimphu->New Delhi->Male->Kathman
du->Masqat->Abu Dhabi->Islamabad->Bangkok->Kuala Lumpur->Jakarta->Dili->Port Moresby->Charlotte Amalie->Ouag
adougou->Porto-Novo->Lome->Accra->Yamoussoukro->Conakry->Libreville->Monrovia->Freetown->Banjul->Dakar->Bama
ko->Addis Ababa->Djibouti->Asmara->Mogadishu->Niamey->N'Djamena->Yaounde->Bangui->Brazzaville->Khartoum->Kig
ali->Bujumbura->Kinshasa->Nairobi->Kampala->Dodoma->Lilongwe->Maputo->Mbabane->Lusaka->Gaborone->Windhoek->H
arare->Ankara->Nicosia->Abuja->Ottawa->Saint-Pierre->Praia->George Town->Roseau->Fort-de-France->Quito->Mala
bo->Suva->Nuku'alofa->Funafuti->Papeete->St. Peter Port->Jerusalem->Kingston->Maseru->Valletta->Tunis->Kuwai
t->Tripoli->Dublin->Mexico->Amsterdam->Willemstad->Noumea->Koror->Asuncion->San Marino->Bern->Vaduz->Dushanb
e->Montevideo->Port-Vila->Wellington->Beirut->Damascus->Doha->Bissau->T'bilisi->END
```

（2）最短路径执行结果：

```
请输入您的出发地:
Bern

请输入您的目的地:
Seoul

总时长:21.7

时间最短路径:Bern->Rome->Warsaw->London->Beijing->Seoul

算法运行时间为：7 ms
```

（3）地图可视化执行结果：



（4）算法复杂度分析

时间复杂度：

理论上，迪杰斯特拉算法中最多嵌套了两重循环，由时间复杂度的计算公式可得 $T(n) = O(n^2)$。这里调用了clock( )函数，计算出整个程序执行具体的所需的时间大约在7ms左右。

空间复杂度：

相对而言比较占用内存空间的是邻接矩阵， 其顶点个数为n=199，所以其空间复杂度为 $O(n^2)$，虽然这远远大于邻接表占用的存储空间，但邻接矩阵有更低的时间复杂度。

五、实习结论：

这是我第一次面对一个真正的实际问题，将实际问题逐层抽象，最终用代码来解决。虽然过程有些艰难，但现在回想起来，感觉收获很大。

通过这次实习，我又复习回顾了一遍图的相关知识，这加深了我对数据结构与算法的理解，提高了我利用代码解决问题的能力。第一次拿到这个题目有点儿不知所措，但是慢慢地将其分解成几个结构，理清楚思路后发现并不是很难。有时候程序会出一些小差错，得花好久上网搜索，设置断点才能解决。通过不断地

调试代码，一部分一部分地完成代码，并保证其正确性，最后再将其组装起来，这个过程有些漫长但是很值得。

最后，谢谢杨老师这一学期的教导！您辛苦啦！