《数据结构与算法集中实习》

实习报告

学　　院:　　遥感信息工程学院

班　　级:　　1804

实习地点:　　教学实验大楼 101 机房

指导教师:　　李林宜

组　　员:　　2018302130120　　杨雨辰

2019 年　　06 月　　21 日

一、实习目的

应用方面，在交通日益方便的今日，跨国旅游逐渐成为了不少人生活中的一部分。为了各个国家首都之间出行的便捷性，我们希望通过在连通图的基础下，实现各个首都的遍历，并且通过单源最短路算法为希望价格优先或者时间优先的人们提供出行方案。

在专业知识方面，处理生活中的关系，往往是处理多对多的关系，而计算机中各种数据结构反映多对多关系的是图。由此，在实现应用的方面，我们需要对图的创建、遍历、生成树有较为深入的了解。而其中，单源最短路的迪杰斯特拉算法，则是整个问题的核心，承载着编程语言向实际应用转换的重要作用，需要我们着重了解并且掌握。

同时，涉及多语言，多平台的程序设计和多学科交叉的要求也为我们对迪杰斯特拉单元最短路算法和相关文件存取知识有一定的理解提升。同时，也提供了全新的视角，来看待单元最短路的实际应用，其中，地图可视化更是成果的直接体现。并且，需要我们在分析算法空间、时间复杂度的基础上，重新认识自己的程序，也提倡有新的思路与改进方案。

二、实习内容与要求

1.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） 算法的时间和空间复杂度分析

2.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，并进行算法的时间和空间复杂度分析和实际统计，算法、代码注释清晰易读

三、算法设计与实现

1.算法原理

迪杰斯特拉算法核心是按照路径长度递增的次序产生最短路径，而生成路径的方法是根据下一条最短路径是在经过最短路路径上顶点所到达的。

深度优先遍历是从某个顶点出发，访问此顶点，然后依次从其未被访问的邻

接点出发深度优先遍历图，直至图中所有和该点有路径的所有顶点都被访问。若此时图中尚有顶点未被访问，则选取图中一个未被访问的顶点作为起始点，递归调用，直至图中每个顶点都被访问到为止。

2.算法设计

结构阶段：

设计了自定义结构体 City 和 Route 分别用来读取城市和路径两个 csv 文件中的信息。其中，路径需要有相关 char 型出发城市与目的地城市，便于比较。

在结构体 City 和 Route 基础上设计了 Graph 结构用来以邻接矩阵的方式创建图。其中图需要边、节点个数，分别与路径数量和国家数量相对应，而边需要新的结构体来存储，其中需要与特定路径的花费、时间、交通工具、路径信息相对应。

对于程序实现过程中不断需要城市转化为编号，用于表示邻接矩阵的数组下标。于是，设置了**关键字结构体**，包含有城市名、城市编号用来匹配，节省算法空间。

对于迪杰斯特拉算法实际应用中，此问题需要实现对更新路径过程中转点的存储。在综合考虑迪杰斯特拉更新的过程，是一个一个点引入，故采用了含有各种信息的不同类型的 vector **路径结构体**，来存储各类信息。迪杰斯特拉每一次更新距离，就需要拷贝此前路径的信息，并且 push_back 该点的信息来实现存储。

```
struct City  //（类似于顶点） 城市
{
    double longitude, latitude;    //城市的经度 纬度
    char city[30], country[30];    //城市名 城市所在国家
};

struct Route  //（类似于边）  路径
{
    char origin_city[30], destin_city[30];  //出发城市和目的地城市
    string transport;       //交通工具
    double time, cost;
    string info;            //城市相关信息网站
};

struct Arc          //边节点的结构（用于时间和消费两种优先）
{
    double cost;    //所需
    double time;    //所需时间
    string transport;
    string info;
};

struct Graph
{
    int arcnum;       //边的个数（路径数量）
    int vexnum;       //节点个数（国家的个数）
    struct Arc **arcs;        //邻接矩阵
    bool pass = false;    //Dijkistra 遍历判断条件
};

struct Key              //关键字搜索结构
{
    int key_num;
    char city_name[30];
};

struct Way              //用于Dijkistra中记录经过的点
{
    vector <int> city;
    vector <double> cost;
    vector <string> info;
    vector <string> transport;
};
```

函数阶段：

设计了三个自编功能函数，LocateCity、Match、Min，分别实现输入城市匹配城市编号、输入城市编号匹配城市和返回二者比较的最小值。

设计了读取 csv 文件信息的函数，ReadCityFile、ReadRouteFile 分别读取城市和路径信息。此后设计 CreateGraph 用来生成相应的图，并利用 Dijkstra 函数生成传入出发地的单源最短路径，利用函数 DFSTraverse 进行深度优先遍历。

设计了 Display 和 VisualDisplay 函数来分别在屏幕上展示迪杰斯特拉函数生成路径过程中的从出发地到目的地的最短路，和向 html 文件中按 JavaScript 格式存储相应信息，用来可视化。

3.算法实现

初始化阶段：

在 Read 函数中，根据 csv 文件存储结构的特点，采用 char 型变量逐个读取，通过判断数据之间分隔符来分别存入 City 和 Route 的相关信息。以 City 为

例，其存储形式为“国家名称+，+城市名称+，+ 纬度 + 经度 + \n”，故在第一次读取到逗号前，将 char 型变量逐个存储到 City 结构体数组的国家成员中，在第二次读取到逗号前，将 char 型变量逐个存储到 City 结构体数组的城市名称中，再采用 fscanf 和%lf 组合分别读取 double 型的经度和纬度，继续读取，到读取至换行符，开始下一个城市信息的读取。

在成功读取城市和路线信息后，根据价格优先和距离优先的方式分别创建用来搜索最优路径的图。首先调用 LocateCity 函数返回出发地、目的地的城市编号作为邻接表下标表示的根据。再通过循环判断路线信息中的途径城市符合的数组下标，将路线信息传入对于数组成员。值得注意的是路线信息中**包含相同两座城市之间的多种出行方式**，需要根据用户的出行路线的优先顺序进行判断，加以选择。如图中所表示的，如果用户采用价格优先，则在输入路径的时候，判断此条路径是否已经存在，若存在则进行比较，若不存在则直接输入。

```
if (strcmp(priority, "TIME") == 0)   //如果价格优先
{
    if (g.arcs[origin_num][dest_num].cost != DBL_MAX          //只用判断cost在前一次被传入即可
        && route[i].time< g.arcs[origin_num][dest_num].time)   //对于两城市之间有多种选择
        {
        g.arcs[origin_num][dest_num].cost = route[i].cost;
        g.arcs[origin_num][dest_num].time = route[i].time;
        g.arcs[origin_num][dest_num].transport = route[i].transport;
        g.arcs[origin_num][dest_num].info = route[i].info;
        }

    else if(g.arcs[origin_num][dest_num].cost = DBL_MAX){
        g.arcs[origin_num][dest_num].cost = route[i].cost;
        g.arcs[origin_num][dest_num].time = route[i].time;
        g.arcs[origin_num][dest_num].transport = route[i].transport;
        g.arcs[origin_num][dest_num].info = route[i].info;
    }
}
```

两座城市间路径选择的判断 （以时间优先判断为例）

实际问题实现阶段：

首先，根据深度优先遍历原理，找到路线中起始城市编号与需要遍历的序号相同的路径，遍历该点，然后通过路线结构体的目的地城市，找到下一个需要遍历的点。重复步骤，递归调用。

由此在完成深度优先遍历后，可以确认图的连通性。

此后，根据迪杰斯特拉单源最短路生成原理，先将距离数组初始化为最大，并将出发点的信息导入路径结构体数组中，根据用户需要，根据优先顺序利用建立好的图第一次更新。此后，利用循环，找出一个到出发点最短路径的下一个节点，遍历该点，再根据最短路径更新其他点经过此点时更优的路径。同时，更新最优路径时，需要将该点和前往该点的路径信息导入。如右图所示，以时间最优为例，way[i]表示记录从出发点到城市编号为 i 的路径,若需要更新路径，则导入此前到加入点的位置信息，并在城市中加入转折点。而实现过程中，C++STL 模板库的 vector 利用 push_back 函数很好地完成了路径中逐个更新时逐个传入点的信息这一过程。

```
for (int i = 0; i < graph.vexnum; i++)
{
    if (visit[i] == false && distance_D[i] > distance_D[start] + graph.arcs[start][i].time)
    {

        way[i].city = way[start].city;
        way[i].city.push_back(start);

        way[i].cost = way[start].cost;
        way[i].cost.push_back(graph.arcs[start][i].time);

        way[i].info = way[start].info;
        way[i].info.push_back(graph.arcs[start][i].info);

        way[i].transport = way[start].transport;
        way[i].transport.push_back(graph.arcs[start][i].transport);

    }
    //记录各点通过路径

    if (visit[i] == false) {
        distance_D[i] = Min(distance_D[i],
            distance_D[start] + graph.arcs[start][i].time);
    }
}
```

迪杰斯特拉更新路径过程中的路径记录

在检查迪杰斯特拉生成结果后，可将用户需要的优先情况下最短路径存储到路径结构体数组中，完成对路径的记录。

展示阶段：

在算法完成的基础上，首先实现了结果在用户屏幕上的展示。通过调用匹配城市编号和城市名的函数以及匹配城市名和城市编号的函数，找到此前路径所记录的城市及相关信息。按照一定的格式，较为美观和实用地输出到屏幕上。

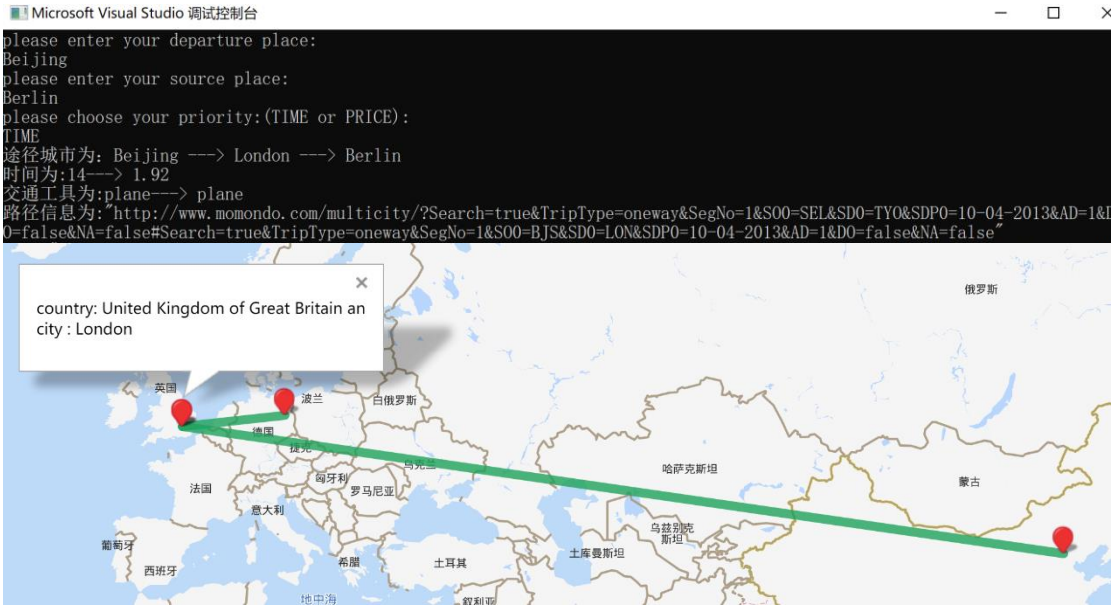
其次，实现了结果在地图上的可视化。通过将 JavaScript 语言存入 html 文件中，基于百度地图实现路径点的标记，路径间折线的画图，路径信息的存储，从而形成可视化。根据存储格式需要，出发点和终止点的存储格式与中间转折点的格式有部分差异，故将其单独存入实现。其余转折点，则依据 vector 数组长度来设置循环，将每一段信息从记录路径的结构体数组中输出至文件中。实现过程中，除了需要按照语言规范格式化存储外，还需要判断城市、国家和相关信息中**是否含有引起歧义的部分**，从而导致语言不合规范，进而不能完成可视化。如图例，部分国家名称中含有引号等字符，从而在读入文件的过程中使语句提前结束，引起歧义而无法达到目的效果。解决办法是在格式化输入之前，搜寻路径城市中含有这些歧义的字符的城市并且对歧义字符定位，移动该位置之后至结尾的字符，再添加转义符\，将更新后的城市名存储到路径中，等待后续格式化存储时使用。

```
for (int i=0;i+1< way[destination_num].city.size();i++)    //判断对于N'Djamena 引起歧义的城市
{
    char change_city[30];
    strcpy(change_city, Match(way[destination_num].city[i], key));
    for (int j = 0; j < strlen(change_city); j++)
    {
        if (change_city[j] == '\'')     //含有' 添加转义符后移动数组
        {
            char ch,trans;
            ch = change_city[j];
            change_city[j] = '\\';
            for (int x = j+1; x <= strlen(change_city) - j+1; x++)
            {
                trans= change_city[x];
                change_city[x] = ch;
                ch = trans;
            }
            strcpy(city[way[destination_num].city[i]].city, change_city);
        }

    }
}
```

可视化存储中引起歧义的解决 （以 N'Djamena 的引号为例）

总体来看，主函数中，用户输入出发地和目的地的城市名和需要选择的价格或者时间优先被程序记录。通过传入相关参数和用户需求，读取 csv 文件中的信息。利用信息进行遍历和生成树，找到用户需要的单源最短路。再分别输出到屏幕和 html 文件上，利用 html 文件实现地图可视化，反映路径及相关信息。



最终程序输出结果

（以北京到柏林时间最优为例）

4.算法复杂度分析

时间复杂度：

①事后分析

利用 Timer.h 中包含的函数时间记录功能，计算了程序运行的时间。以北京到柏林为例，计算了程序运行的时间。但是此种分析方法，受运行程序的编译器、电脑等多种因素影响，并且也与用户输入的出发地与目的地有关。如此时间复杂度的分析，只是作为参考的一个标准。

```
Elapsed time is: <356.84> ms
```

程序运行时间

②事先估计

事先估计法，首先需要找到复杂度最高的程序块，确定时间复杂度的阶数，再综合分析低阶、同阶的程序块的个数，来综合考虑程序的时间复杂度。

阶数最高的函数为可视化展示中城市读取时消除歧义的算法，为三次方。在算法设计过程中，没有采用在读取文件信息时就进行判断，而是在确定了最短道路之后在道路中进行判断，从而减少了循环的次数。其最多调用为次数路径长度的循环次数*在所有城市中匹配的次数*消除歧义时移动数组的位次数。

```
for (int i=0;i+1< way[destination_num].city.size();i++)    //判断对于N'Djamena 引起歧义的城市
{
    char change_city[30];
    strcpy(change_city, Match(way[destination_num].city[i], key));
    for (int j = 0; j < strlen(change_city); j++)
    {
        if (change_city[j] == '\'')     //含有'  添加转义符后移动数组
        {
            char ch,trans;
            ch = change_city[j];
            change_city[j] = '\\';
            for (int x = j+1; x <= strlen(change_city) - j+1; x++)
            {
                trans= change_city[x];
                change_city[x] = ch;
                ch = trans;
            }
            strcpy(city[way[destination_num].city[i]].city, change_city);
        }

    }
}
```

复杂度较高的一段程序

同时，迪杰斯特拉在外层循环寻找最短路径点和内层更新最短路径过程中，均需要在全部节点中搜索。故算法复杂度为 $O(n^2)$。

深度优先遍历的算法用到了 DFS 的递归调用，除了外层循环寻找需要遍历的点，内层循环的递归调用也使算法复杂度为 $O(n^2)$。

此外初始化图的过程中，调用了城市与编号匹配的函数，且匹配过程为直接查找，故除了初始化的外层循环，内层匹配的循环也为需要同次数的执行，故算法复杂度为 $O(n^2)$。在可视化的信息存入文件的过程中，由于路径为节省空间，未采用两个 vector 数组分别存放时间和花费，故又调用城市与编号匹配函数，从而除了循环输入路径，内部还需要将编号和城市名匹配，使得复杂度为 $O(n)$。

综合来看，多个 $O(n^2)$ 复杂度所产生的程序依此运行的过程中，其共同产生的函数渐进增长也值得考虑。随着 n 次数的增加，多个低阶复杂度的算法由于其运行次数不同，也会因此产生较大的差距。

空间复杂度：

空间复杂度是分析程序存储变量所占用空间的一项复杂度，有一部分可以通过调整算法的结构来减少存储空间，另一部分则需要和时间复杂度权衡。在此次实习过程中，程序在编译器初始堆栈空间存储内存的基础上即可运行。

方案：主要占用空间的变量，将其调整为全局变量，减少了函数体内部不断创建成员变量，并且采用引用的方式减少了重复变量副本的产生。本程序采用

了存储路径的全局结构体数组、表示图的全局结构体数组、表示最短路径的 double 型数组，长度分别为路径数 1975、图的边数 199*199 和两城市间最短路径数 199。在函数调用过程中，读文件时，构建城市和编号的关键字匹配数组，长度为城市数 199。

其中，也有通过改变算法来减少空间复杂度的例子。在格式化存储过程中，调用 string 类型的 c_str()函数，直接返回 char*类型，完成格式化存储，从而减少了新建一个 char 型数组来暂时存储目标 string 类，再通过暂存数组输入的过程，进而减少了空间复杂度。

```
//string转换为char
fprintf(fp, "(%s-", way[destination_num].transport[i - 1].c_str());
```

算法调整减少空间复杂度

此外也有空间复杂度和时间复杂度不可兼得的情况。同样是在格式化输出的过程中，如空间复杂度提及的，在可视化的信息存入文件的过程中，由于路径为节省空间减少时间复杂度，未采用两个 vector 数组分别存放时间和花费，故又调用城市与编号匹配函数，增加了空间复杂度。综合分析，由于数据量较大，并且已经声明了许多存储大数据的成员变量，故采用了这种方式，多调用函数，在不增加总时间复杂度阶数的情况下，减少了一部分空间复杂度。

```
int route_num;  //记录是哪一条路径（由于此前优先选择的时候未在way中存储除了时间/价格的信息）
for (route_num = 0; route_num < CityNum; route_num++)
{
    if (way[destination_num].city[i - 1] == LocateCity(route[route_num].origin_city, key, CityNum)
        && way[destination_num].city[i] == LocateCity(route[route_num].destin_city, key, CityNum))
        break;
}
fprintf(fp, "%lfhours-$%lf-", route[route_num].time, route[route_num].cost);

fprintf(fp, "%s)';", route[route_num].info.c_str());
```

牺牲时间复杂度 减少空间复杂度

四、实验结果

根据实习内容和要求，完成了必须的 CSV 格式数据文件的读写，利用邻接矩阵创建了图，利用深度优先搜索完成了图的遍历，运用迪杰斯特拉求出图中用户所需的最短路径，并给出路径及相关信息，并且在 html 文件中完成了可视化展示，最后在实习报告中分析了总体算法的时间空间复杂度。除了按照要求完成了三个算法，完成了可供选择算法的两个算法。

在完成所有内容的基础上，个人对于算法实现方面仍有许多不足，在有限的时间和能力范围内仍需要**改进和探索**。

从结构创建上，可以充分利用 STL 模板库中的现有工具，比如在匹配城市和城市编号的过程上，可以利用 map 类，将编号作为关键字赋给城市，从而减少了寻找和匹配的难度，并且利用 map 类已经封装的函数，也可以精简程序的体积，更加直观。

从复杂度方面，对于本题中路径只有 1975 条，而城市数有 199 个，创建出的 199*199 的图不能算上是稠密图。故可以学习采用堆优化后的迪杰斯特拉算法，使寻找最短路的复杂度变为(m+n)logn，减少程序运行时间。同样，在匹配的过程中，可以采用 KMP 算法更快地直接比较子串来匹配查找。在减少歧义的算法中，可以借助排序的思想减少移动数组元素的个数，减少现有程序的时间复杂

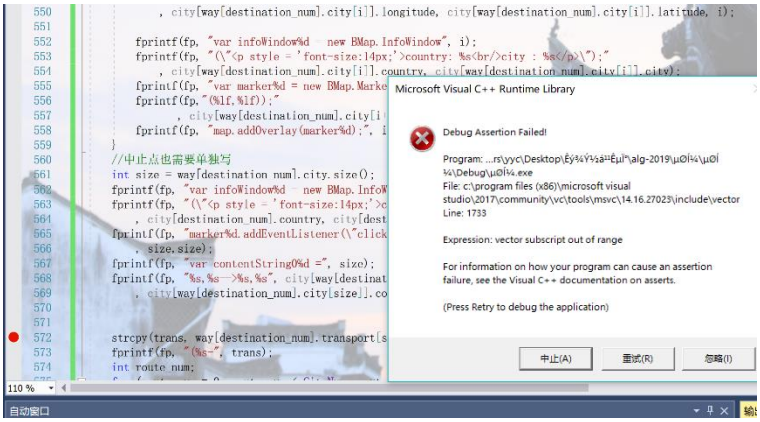
度。

五、实习心得体会

为期六天的上机实习，基本上是充分利用了在机房的时间完成代码，并且也由于进度不够，利用课余时间完成。虽然过程辛苦，完成了六项算法相关的要求，但是在地图可视化成功实现的一刻，充满了欣喜与收获。

整个过程中，加强了我对算法和 STL 等模板类的理解和运用，通过函数调用，不断练习各种结构体的使用和之间相互关系的权衡与选择，并且在大数据量的原始文件中找寻错误，一次次深化着对算法的理解和认识。

然后是写代码新习惯的养成。首先是部分检核，在数据量较大的文件中读取，找寻逻辑错误变得更加困难，所以在编写的过程中，我逐步形成了到关键步骤就检核的习惯，而不是此前的一口气编完再从头找错误。在格式化存储的过程中尤为显著，面对满屏的 JavaScript 语言，加上读入时的双引号干扰，让找错误尤为困难。而细节上的错误使得整个 html 文件打开之后不显示地图。虽然仍有错误出现，但是逐行输出的检核，让错误减少并且较快暴露。其次是工作量分配，根据自己空闲时间的情况，可以更好地分配任务量，并且在实习期限内完成基本任务，并且还可以完成可选择的任务。



调试中栈溢出的情况

并且在代码空间时间复杂度的把控上，也有了更全面的认识。

这次实习，是第一次接触地图方面的工作。除了对于算法有更深的理解，对于 csv、html 文件格式有了更深理解外，也对其有了更广的应用展望。通过 Java 和类 C 语言结合，通过地图学和编程结合，让单源最短路的迪杰斯特拉算法有了实际的应用。这是我此前从未碰到的，也开拓了我的视野。