课程编号：C0801207040

数据结构与算法
课程设计报告

姓　　名	李思宽	学　　号	20175153
班　　级	软英 1701	指　导　教　师	张　莉
实　验　名　称	数据结构与算法课程设计		
开　设　学　期	2018-2019 学年第 2 学期		
开　设　时　间	第 20 周——第 21 周		
报　告　日　期	2019.1.23		
评　定　成　绩		评　定　人	
		评　定　日　期	2019-1-25

东北大学软件学院

第一章 系统分析

（一）系统背景

时下大多数人生活优越，信息业，交通业等行业的高速发展，带动了各地对旅游资源的开发。但景区管理仅靠人力管控较为困难，游客也难以得到较为舒适的服务。

为了使游客能够更加有效地掌握景区的相关信息，并方便管理者管控景区，开发、完成一个景区管理系统的实现，能够帮助景区管理者高效地对景区进行管理，同时该系统的实现及运用也能较好的满足游客们的具体游玩需求，让游客得到更加舒适智能的服务。

该系统将为游客及管理员提供全面准确的景区景点查询管理服务。

同时，该系统能还够提高学生对软件设计思路的理解以及进一步熟悉数据结构方面尤其是图论的算法。

（二）功能需求

管理员：

- 对景点进行增删改查
- 对景点之间的道路进行增删改查
- 发布公告通知

游客：

- 查询景点的分布图
- 查询推荐的导游路线
- 查询两景点间的最短路径
- 根据景点名称或其描述进行关键字查询
- 景点根据受欢迎程度或景点岔路口数进行排序查询
- 景区停车场车辆进出信息

（三）分析系统可能的解决方案

此次数据结构课程设计的主要关键点及难点在于各种数据结构的实现及将其运用于景区管理系统当中。系统的主要难点在于数据结构的设计实现以及算法的选择、设计以及优化过程。经过深思熟虑地思考，查询资料及与指导教师和同学讨论后，针对各功能的实现给出以下的解决方案：

➢ 主界面

系统运行后将进入主菜单界面，并在之后进行景点信息初始化工作，游客和管理员均可看到该界面并于此界面进行操作。该界面包括一个选项卡界面：

- ◆ 景区景点分布大地图选项卡
- ◆ 为游客推荐导游路线选项卡

- 查询两景点间的最短路径选项卡
- 景点查询及排序选项卡：
    - 根据景点名称或其描述进行关键字查询
    - 景点根据受欢迎程度或景点岔路口数进行排序查询
- 管理景区停车场车辆进出信息选项卡

还包括一个公告通知发布栏。

➢ 景点信息初始化

景点信息需从景点信息文件中读入，

景点信息包括：

- 景点的名称
- 景点的描述
- 景点的欢迎度
- 有无休息区
- 有无公厕

将所有景点信息存入邻接链表之后，系统将链表信息加载到主界面。并可用于主界面中各选项卡界面。

➢ 管理员登录

管理员登录可以通过主界面进行登录，登录时需要输入用户名和密码（为方便测试人员进行测试，在界面左上角有提示信息）。如果输入的账号或者密码错误，则会有错误信息的提示。如果验证通过，会进入相应的 *Administer* 界面。

➢ 管理员界面

完成登录过程后可以进入管理员界面，并完成界面及景点信息初始化工作，只有管理员可看到该界面并于此界面进行操作。该界面同样包括一个选项卡界面：

- 景区景点插入选项卡
- 景区景点删除选项卡
- 两景点间的道路插入选项卡
- 两景点间的道路删除选项卡：
- 发布公告通知选项卡

➢ 发布通知

设定一个专门的栏目，管理员有权限在该栏目上发布通告，同时， 在游客端可以看到管理员发布的消息，但是游客无法发布通告。

➢ 对景点进行增删改查

当有新的景点需要加入景区的时候，管理员可以插入新的景点。管理员需要输入景点的名称，描述，欢迎度，有无休息区，有无公厕的信息。系统会对景点的名称进行查重，一旦有重名景点，系统将提示景点添加失败。

当一个景点不再对外界开放的时候，管理员可以删除该景点，同时也将删除与其相连的边。如果需要删除的景点不存在，系统会给出删除失败的提示。

当一个景点的信息需要被更新的时候，管理员可以更改景点的名称，描述等。如果未找到需要修改的景点，系统会提示修改失败。

当一个景点的信息需要被查询的时候，根据输入的关键字对景点信息进行匹配，找出相应的景点信息，如果景点不存在，返回景点不存在的提示。（位于主界面的景点查询及排序选项卡）

- **对景点之间的道路进行增删改查**

用一个邻接链表存储景点信息和道路信息，当对道路进行增删改查的时候，直接在邻接链表上进行操作即可更新道路信息。

- **管理景区停车场**

当有车辆到达的时候，管理员输入车牌号，将车的信息压入栈中。 当车位已满时，达到的车辆将进入等待区。

- **查询景点的分布图**

在主界面初始化后，读取邻接链表的信息，并进行解析，在用户界面处显示出相应的景点分布图。

- **查询推荐的导游路线**

使用汉密尔顿算法，欧拉算法，深度优先算法以及 BFS、DFS 相结合的 BDFS 创新算法等，用户可以对各个算法进行选择，然后可得到推荐的导游路线图，同时输出路径的长度，以便游客进行挑选及判断。

- **查询景点间的最短路径**

使用迪杰斯特拉算法,弗洛伊德算法，Bellmand-Ford 算法，SPFA 算法等，用户可以对各个算法进行选择，然后对两点间进行最短路径的计算并显示。

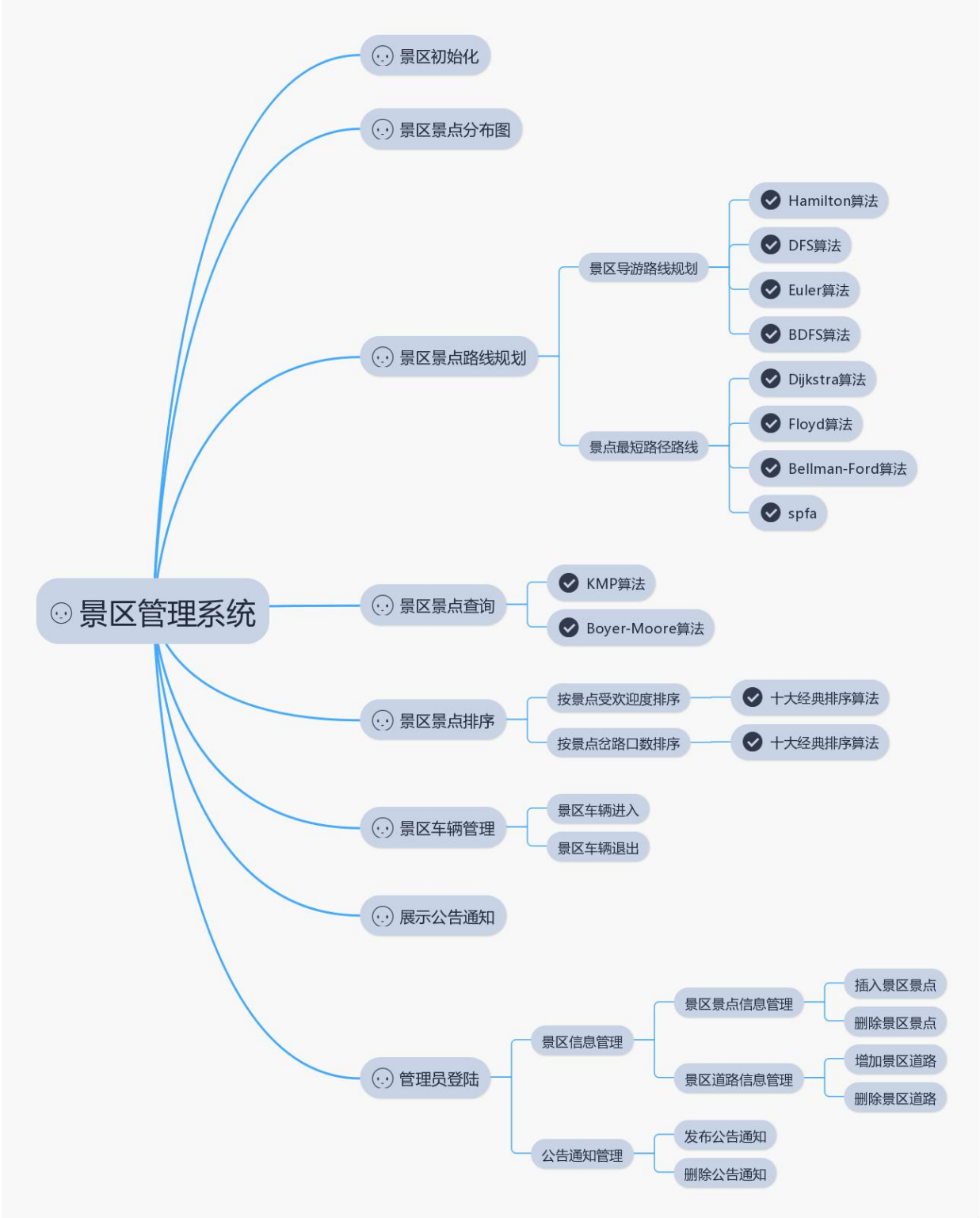
- **根据景点名称或其描述进行关键字查询**

使用 KMP, Boyer-Moore 等算法对用户输入的一段字符串进行子字符串的匹配，查找出与之相匹配对应的景点，在用户界面上显示各个匹配景点的按钮，用户可根据自己的意愿惊醒点击按钮，查看相应的景点信息。

- **景点排序查询**

使用希尔排序，桶排序等十种常用的排序方法对景点的相应属性（受欢迎程度，景点岔路口数）进行排序，用户可以对各个算法进行选择，然后将排序结果显示在用户界面上。

- **景区系统实现思维导图：**


景区管理系统
景区初始化
景区景点分布图
景区景点路线规划
景区导游路线规划
Hamilton算法
DFS算法
Euler算法
BDFS算法
景点最短路径路线
Dijkstra算法
Floyd算法
Bellman-Ford算法
spfa
景区景点查询
KMP算法
Boyer-Moore算法
景区景点排序
按景点受欢迎度排序
十大经典排序算法
按景点岔路口数排序
十大经典排序算法
景区车辆管理
景区车辆进入
景区车辆退出
展示公告通知
管理员登陆
景区信息管理
景区景点信息管理
插入景区景点
删除景区景点
景区道路信息管理
增加景区道路
删除景区道路
公告通知管理
发布公告通知
删除公告通知

（三）个人工作和主要内容

个人工作主要包含：

✧ **路线规划问题：**

实现使用常用的算法来进行路线规划，解决游客导游路线规划的困扰，使用了 Hamilton 算法，Euler 算法， DFS 算法，此外，结合深度优先遍历和广度优先遍历的优点，进行了融合，来实现路线规划，能够达到较好的效果，简称为 BDFS+, 关于 BDFS+ 算法与以上算法的分析和比较将在后续部分详细阐述。

✧ **最短路径问题：**

实现使用 Dijkstra 算法, Floyd 算法， Bellmand-Ford 算法，SPFA 算法四个方法进行求解，此外基于 Dijkstra 算法实现了启发式 A*算法，能够更好地实现查找最短路径，关于五种算法的分析和比较将在后面部分详细阐述。

✧ **排序算法的问题：**

使用冒泡排序，插入排序，选择排序，希尔排序，归并排序，快速排序，堆排序，基数排序，计数排序和桶排序十种常用的排序算法。关于十种算法的分析和比较会在后面的部分阐述。

✧ **修建道路问题：**

用产生最小生成树的 Kruskal 算法和 Prim 算 法。

✧ **关键字查询的问题：**

使用字符串匹配的算法，KMP 算法和 Boyer-Moore 算法，关于两种算法的分析和比较会在后面部分详细阐述。

✧ **数据结构的设计实现及相关功能的实现：**

正确高效完成其余功能，包括数据结构设计，增删改查，景点分布图等。

✧ **用户可视化交互界面：**

使用 JavaFX 框架进行设计。

第二章 系统设计

（一）数据结构设计

根据景区管理系统的需求分析，设计数据结构。

系统主要有两大模块，分别为景点分布及其规划路线模块和停车场模块。可以知道这两个模块互相独立，所以可以分别设计数据结构。

实现了三种基本数据结构的底层: MyList, MyQueue, MyStack

➢ **景点分布及其规划：**

该模块要实现景点信息的读取、展示以及景点和路线的增删改查，运用抽象思维，可以将景点抽象为节点，将路抽象为边，将整个景区（不包含停车场）抽象为一个图，这样，就可以利用图论的知识进行进一步的实现操作。为节点和边和图设计类。图类中存储了一个包含节点类的 list 以及节点和边的总数，节点类存储一个包含边类的 list，这种数据结构即为邻接链表。

✧ **增加节点：**

直接在图中的存储节点的 list 里面添加节点，同时递增图类中记录的节点的数量。

✧ **删除节点：**

想法是：先得到该节点，然后先将该节点中储存的边删去，最后将此节点删去。

首先获得要删除的节点在图类的 list 中的索引位置,然后遍历每一个节点中存储边的 list，删除所有包含了要删除的节点的边，并递减边的数量，然后在图类的 list 中删除该节点，最后递减图类中的节点数。

✧ **增加边：**

直接在节点类中的存储边的 list 里面添加节点，同时递增图类中记录的边的数量。

✧ **删除边：**

想法是：遍历每一个节点中存储边的 list，删除所有包含了要删除的边，递减边的数量，注意图为无向图。

✧ **计算两节点之间的距离：**

直接取出某一节点的 list 进行遍历，如果找到相应的边就可以直接得到距离，否则置为 32767。

✧ **输出邻接矩阵：**

利用两节点之间距离的计算方法，进行格式化输出，得到邻接矩阵。

➢ 停车场管理：

游客及管理员均可以使用停车场模块对车辆进行调度（调入，调出）。

根据停车场管理需求，一共为停车场模块设计了三种数据结构，分别为 Vehicle 类，MyQueue 和 MyStack。

停车场主要由一个队列和两个栈构成：

- ◆ 队列：用于实现停车场停满车辆时，继续停车时位于等待区域的便道。
- ◆ 两个栈：分别用于模拟实际停车场车位和移动车时的道路。

✧ 增加车辆：

首先要对停车场进行遍历，判断车牌号是否已存在与停车场内，不存在，则急需判断停车场是否已满，来判断将车辆添加到车位栈还是候车道栈。

✧ 删除车辆：

每次有车辆离开时，判断是否存在该车，就需要判断是否需要把后方的车移动到临时的栈中，移动结束以后再把临时栈中的元素加入到原本的栈中。

在移动车辆过程中，完成计算停车时间和应缴费用。

（二）算法设计

设计重点在于算法的设计及具体实现，实现了较多的算法（测试过程中可根据意愿自由选择），以及自己创新了一个算法，初步实验根据具体景点实例证明，在一部分景点的回路计算中（比如起点北门到终点北门），比汉密尔顿回路的效果更好。下面详细介绍已经实现的算法的思路。

➢ 导游路线规划算法设计：

BDFS + 算法：（创新结合）该算法的基本思想是**结合深度优先和广度优先的优点**，引入深度优先控制因子，回溯控制因子，终点控制因子，已访问控制因子四个变量来控制深度和广度的权重，从而在其中达到一个平衡。另外，如果起点到终点最短路径所需要的边越多，从直观上来看，应该增加广度优先的权重，反之，则增加广度优先的权重。更加具体的说明会在**系统实现**处讲述。

✧ 汉密尔顿算法：

此方法要求每个节点有且只能被经过一次，采用回溯法即试探法。基本思想是：从一条路往前走，能进则进，不能进则退回来，换一条路再试。利用深度优先算法搜索解空间，在递归的搜索过程中，不断修正路线，直到能够达到要求为止。

◆ 算法的时间复杂度和空间复杂度：

回溯算法在最坏情况下可能需要更新当前最优解 O((n-1)!)次，每次更新最优解需计算时间 O(n)，从而整个算法的计算时间复杂性为 O(n!)。

```
int[] path //记录哈密顿路径
for(int i = 0;i < 节点数;i++) {
    初始化，所有顶点均未被遍历
    初始化，未选中起点及到达任何顶点
}
dfs(); //从第 0 个顶点开始进行深度优先遍历,如果存在哈密顿路径，输出一条路径，否则无输出
```

✧ DFS 算法：

此方法是深度优先搜索算法，思路是只要遇到一条边，直接选择这条边，不考虑其他情况。由于深度遍历算法较为常见而且在上面的汉密尔顿算法中也有使用。

◆ 算法的时间复杂度和空间复杂度：

查找所有顶点的邻接点所需时间为 O(E)，访问顶点的邻接点所花时间为 O（V），此时，总的时间复杂度为 O(V+E)。

```
dfs{
    if (起点，终点不同){
        需遍历的节点数 = 节点总数-2;
    }
    else{
        需遍历的节点数 = 节点总数-1;
    }
    if(有节点未被访问){
        if(最后一个访问的定点能够到达终点){
            Return true;
        }
        Return false;
    }
    else{
        访问所有节点的边
        dfs();
        if(最后一个节点与终点之间无边){
            回溯;
        }
    }
```

✧ **欧拉回路算法：**

此方法的目标是找到一个路径包括每个边恰好一次。该算法的思路是不断削减节点的边，直到找到一条回路为止。（如果是对于景区系统而言，道路风景优美的情况下，采用 *Euler* 是比较好的导游方案，不然因所有路都经过的原因，路程会大幅增加）。

其伪代码如下：

```
EulerTour(){
        if(无欧拉回路){
                return;
        Copy 所有节点中的 list;
        While(flag<已有的路径的节点数) {
                if(索引为 flag 位置的节点的边不为 0) {
                        //定位加入路线图的 list 的位置
                        Count = flag+1;
                        //flag 用于在队列中推进，寻找仍有边的路径点
                        findEulerTour();
                }
                flag++;
        }
}
findEulerTour(){
 //随机找一条边，终结点一定绕回到出发点，并且出发点的边会遍历完毕
        if( 指定节点的边数不为 0 ){
                删除第一条边;
        }
        findEulerTour();
        }
}
```

➢ **最短路径算法：**

✧ **Dijkstra 算法：**

计算最短路径的常用算法，Dijkstra 算法采用的是一种贪心的策略，声明一个数组 dis 来保存源点到各个顶点的最短距离和一个保存已经找到了最短路径的顶点的集合：T，初始时，原点 s 的路径权重被赋为 0 （dis[s] = 0）。若对于顶点 s 存在能直接到达的边（s,m），则把 dis[m]设为 w（s, m）,同时把所有其他（s 不能直接到达的）顶点的路径长度设为无穷大。初始时，集合 T 只有顶点 s。

然后，从 dis 数组选择最小值，则该值就是源点 s 到该值对应的顶点的最短路径，并且把该点加入到 T 中，此时完成一个顶点， 然后，我们需要看看新加入的顶点是否可以到达其他顶点并且看看通过该顶点到达其他点的路径长度是否比源点直接到达短，如果是，那么就替换这些顶点在 dis 中的值。 然后，又从 dis 中找出最小值，重复上述动作，直到 T 中包含了图的所有顶点。

◆ **算法的时间复杂度：**

总的时间复杂度为 O(n^2)。

此方法的伪代码如下：

```
void dijkstra(){
    初始化距离数组;
    while(i < 节点数){
        if（节点未访问并且有边相连）{
            寻找最小的边;
        }
        if(当前距离 + 当前边的长度 < 距离数组中的距离){
            距离数组中的距离 =当前距离 + 当前边的长度;
            记录当前节点的父节点;
        }
    }
}
```

✧ **Floyd 算法:**

通过 Floyd 计算图 G=(V,E)中各个顶点的最短路径时，需要引入两个矩阵，矩阵 S 中的元素 distance[i][j]表示顶点 i(第 i 个顶点)到顶点 j(第 j 个顶点)的距 离。矩阵 P 中的元素 path [i][j]，表示顶点 i 到顶点 j 经过了 b[i][j]记录的值所表示的顶点。设图 G 中顶点个数为 N，则需要对矩阵 D 和矩阵 P 进行 N 次更新。初始时，矩阵 D 中顶点 distance[i][j]的距离为顶点 i 到顶点 j 的权值；如果 i 和 j 不相邻，则 distance[i][j]=∞，矩阵 P 的值为顶点 path[i][j]的 j 的值。 接下来开始，对矩阵 D 进行 N 次更新。第 1 次 更 新 时 ， 如 果 ”distance [i][j] 的 距 离 ” > “distance [i][0]+ distance[0][j]” ， 则 更 新 distance[i][j] 为 ”distance[i][0]+ distance[0][j]”,更新 path[i][j]= path[i][0]。 同理，第 k 次更新时，如果” distance[i][j]的距离” > “distance[i][k-1]+distance[k-1][j]”，则 更 新 distance[i][j] 为 ”distance[i][k-1]+ distance[k-1][j]”，path[i][j]= path [i][k-1]。更新 N 次之后，操作完成。

◆ **算法的时间复杂度和空间复杂度：**

查找所有顶点的邻接点所需时间为 O(E)，访问顶点的邻接点所花时间为 O（V）,此时，总的时间复杂度为 O(V+E)。

Floyd 算法的时间复杂度为 $O(N^3)$，空间复杂度为 $O(N^2)$。

其伪代码如下：

```
初始化距离数组 distance
初始化路径数组 path
for (int k = 0; k < 节点数; k++) {
    for (int i = 0; i < 节点数; i++) {
        for (int j = 0; j <节点数; j++) {
            if(i 与 k 无边|| k 与 j 无边) {
                临时距离变量 = INF;
            }
            else{
                临时距离变量 = ik 距离 + jk 距离
                if (ij 距离 > 临时距离变量) {
                    ij 距离 =临时距离变量;
                    加入路径
                }
             }
         }
     }
}
```

✧ Bellman-Ford 算法：

Bellman-Ford 算法采用动态规划（*Dynamic programming*）进行设计，实现的时间复杂度为 O(V*E)，其中 V 为顶点数量，E 为边的数量。

算法描述：

- ◆ 初始化：
  创建源顶点 v 到图中所有顶点的距离的集合 distSet，为图中的所有顶点指定一个距离值，开始均为 Infinite，源顶点距离为 0；
- ◆ 迭代求解：
  计算最短路径，执行 V - 1 次遍历；
- ◆ 检验负权回路：
  对于图中的每条边：如果起点 u 的距离 d 加上边的权值 w 小于终点 v 的距离 d，则更新终点 v 的距离值 d；
- ◆ 检测图中是否有负权边形成了环:
  遍历图中的所有边，计算 u 至 v 的距离，如果对于 v 存在更小的距离，则说明存在环；

下面是伪代码的实现过程：

```
BELLMAN-FORD(G, w, s)
  INITIALIZE-SINGLE-SOURCE(G, s)
  for i  1 to |V[G]| - 1 :
       do for each edge (u, v)  E[G]:
            do RELAX(u, v, w)
  for each edge (u, v)  E[G]:
       do if d[v] > d[u] + w(u, v)
            then return FALSE
  return TRUE
```

✧ SPFA 算法：

该算法的基本思想是使用一个队列来进行维护。 相当于在 Bellman-ford 算法的基础上加上一个队列优化，减少了冗余的松弛操作，是一种高效的最短路径算法。初始时将源点加入队列。 每次从队列中取出一个元素，并对所有与他相邻的点进行松弛，若某个相邻的点松弛成功，如果该点没有在队列中，则将其入队。直到队列为空时算法结束。

```
SPFA（String source, String des）
    初始化最短距离数组为 INF
    initialize-queue(Q);
    将起始点加入队列
    While(true){
        if(边距离+起始点距离 < 终点距离){
        终点距离 = 边距离 + 起始点距离
            存入父节点
            如果该节点不在队列中，加入队列
            if(队列空){
                break;
       }
    }
}
```

➢ 道路修建规划建议（最小生成树）：

✧ Kruskal 算法：

克鲁斯卡尔算法的基本思想是以边为主导地位，始终选择当前可用（所选的边不能构成回路）的最小权植边。

具体实现过程如下：

◆ 〈1〉 设一个有 n 个顶点的连通网络为 G（V,E），最初先构造一个只有 n 个顶点，没有边的非连通图 T={V,空}，图中每个顶点自成一格连通分量。

◆ 〈2〉 在 E 中选择一条具有最小权植的边时，若该边的两个顶点落在不同的连通分量上，则将此边加入到 T 中；否则，即这条边的两个顶点落到同一连通分量 上，则将此边舍去（此后永不选用这条边），重新选择一条权植最小的边。

◆ 〈3〉 如此重复下去，直到所有顶点在同一连通分量上为止。

◆ 算法的时间复杂度和空间复杂度：

kruskal 算法的时间复杂度为 O(eloge)跟边的数目有关

其伪代码如下：

```
// 把所有边按大小排序，记第 i 小的边为 e[i] (1<=i<=m)m 为边的个数
// 初始化 MST 为空
// 初始化连通分量，使每个点各自成为一个独立的连通分量

for (int i = 0; i < m; i++){
    if (e[i].u 和 e[i].v 不在同一连通分量){
        // 把边 e[i]加入ＭＳＴ
        // 合并 e[i].u 和 e[i].v 所在的连通分量
    }
}
```

✧ **Prim 算法：**

该算法需要选定起始点，选定不同的起点，得到的最小生成树不唯一，该算法的基本流程如下：

1）输入：一个加权连通图，其中顶点集合为 V，边集合为 E；

2）初始化：Vnew = {x}，其中 x 为集合 V 中的任一节点（起始点），Enew = {},为空；

3）重复下列操作，直到 Vnew = V：a.在集合 E 中选取权值最小的边<u, v>，其中 u 为集合 Vnew 中的元素，而 v 不在 Vnew 集合当中，并且 v∈V（如果存在有多条满足前述条件即具有相同权值 的边，则可任意选取其中之一）； b.将 v 加入集合 Vnew 中，将<u, v>边加入集合 Enew 中；

4）输出：使用集合 Vnew 和 Enew 来描述所得到的最小生成树。

◆ **算法的时间复杂度和空间复杂度：**

其时间复杂度为 O(n^2)，其算法时间复杂度与顶点数目有关系，与边得数目无关

其伪代码如下

```
void Prim(MGraph g, int v0, int &sum){
    int lowcost[maxSize],vset[maxSize];
    int i, j, k;
    v = v0;
    for(i = 1;i <= g.n; i++){
        lowcost[i] = g.edgs[v0][i];
        vset[i] = 0;
    }
    vset[v0] = 1;
    sum = 0;
    for(i = 1; i <= g.n; i++){
        min = INF;
        for(j = 1;j <= g.n; j++){
            if(min > lowcost[j]){
                min = lowcost[j];
                k = j;
            }
        }
        vset[k] = 1;
        v = k;
        sum += min;
        for(j = 1; j <= g.n ;++j){
            if(vset[j]==0 && lowcost[j]>g.edges[v][j] ){
                lowcost[j] = g.edges[v][j];
            }
        }
    }
}
```

➢ 字符串匹配算法:

✧ KMP 算法

该算法的关键是利用已匹配成功的信息,尽量减少模式串与主串的匹配次数以达到快速匹配的目的。具体实现就是实现一个 next()函数,函数本身包含了模式串的局部匹配信息。伪代码如下:

```
private boolean KMP(){
    int[] next = calculateK(keyword);
    int i = 0, j = 0;
    while(i <字符串长度 && j <子字符串长度)){
        // 如果 j = -1，或者当前字符匹配成功，都令 i++，j++
        if(j==-1 || 字符匹配){
            i++;
            j++;
        }else{
             //如果 j != -1，且当前字符匹配失败，则令 i 不变，j = next[j]
            //next[j]即为 j 所对应的 next 值
            j = next[j];
        }
    }
    if(匹配成功){
        return true;
    }
    return false;
}

private int[] calculateK(String keyword){
    int[] next = new int[子字符串长度];
    next[0] = -1;
    int k = -1;
    int j = 0;
    while (j < 子字符串长度 - 1) {
        //k 表示前缀，j 表示后缀
        if (k == -1 || 字符匹配) {
            ++k;
            ++j;
            next[j] = k;
        } else {
            k = next[k];
        }
     }
    return next;
}
```

✧ **Boyer-Moore 算法：**

该算法的核心是对要搜索的字符串进行倒序的字符比较，并且当字符比较不 匹配时无需对整个模式串再进行搜索。 伪代码如下：

```
private boolean boyerMoore(String doc, String keyword){
    Map<Character, Integer> right = new HashMap<>();
    //初始化 right
    for(int i = 0; i < 子字符串长度; i++){
        right.put(子字符串第 i 位字符, i);
    }
    //匹配
    for(int i = 0; i <= 字符串长度 - 子字符串长度; i++){
        skip = 0;
        for(int j = keyLength-1; j >= 0 ; j++){
            if(不匹配){
                if(right 有相应的字符)){
                    skip = j - right.get(相应的字符);
                }
                if(skip < 1){
                    skip = 1;
                }
                 break;
            }
            if(skip == 0){
                return true;
            }
        }
    }
    return false;
}
```

- **排序算法：**
  - 冒泡排序：

```
BubbleSort(lst):
    n=len(lst)
    if n<=1:
        return lst
    for i in range (0,n):
        for j in range(0,n-i-1):
            if lst[j]>lst[j+1]:
                (lst[j],lst[j+1])=(lst[j+1],lst[j])
```

  - 快速排序：

```
QuickSort(lst):
    # 此函数完成分区操作
    partition(arr, left, right):
        key = left  // 划分参考数索引,默认为第一个数为基准数，可优化
        while left < right:
            // 如果列表后边的数,比基准数大或相等,则前移一位直到有比基准数小的数出现
            while left < right and arr[right] >= arr[key]:
                right -= 1
            // 如果列表前边的数,比基准数小或相等,则后移一位直到有比基准数大的数出现
            while left < right and arr[left] <= arr[key]:
                left += 1
            // 此时已找到一个比基准大的书，和一个比基准小的数，将他们互换位置
            (arr[left], arr[right]) = (arr[right], arr[left])

        // 当从两边分别逼近，直到两个位置相等时结束，将左边小的同基准进行交换
        (arr[left], arr[key]) = (arr[key], arr[left])
        // 返回目前基准所在位置的索引
        return left

    quicksort(arr, left, right):
        if left >= right:
            return
        // 从基准开始分区
        mid = partition(arr, left, right)
        // 递归调用
        # print(arr)
        quicksort(arr, left, mid - 1)
        quicksort(arr, mid + 1, right)
```

✧ 选择排序：

```
SelectSort(lst):
   n=len(lst)
   if n<=1:
      return lst
   for i in range(0,n-1):
      minIndex=i
      for j in range(i+1,n):       //比较一遍，记录索引不交换
         if lst[j]<lst[minIndex]:
            minIndex=j
      if minIndex!=i:          //按索引交换
         (lst[minIndex],lst[i])=(lst[i],lst[minIndex])
   return lst
```

✧ 希尔排序：

```
ShellSort(lst):
    shellinsert(arr,d):
        n=len(arr)
        for i in range(d,n):
            j=i-d
            temp=arr[i]              //记录要出入的数
            while(j>=0 and arr[j]>temp):   //从后向前，找打比其小的数的位置
                arr[j+d]=arr[j]               //向后挪动
                j-=d
            if j!=i-d:
                arr[j+d]=temp
    n=len(lst)
    if n<=1:
        return lst
    d=n//2
    while d>=1:
        shellinsert(lst,d)
        d=d//2
    return lst
```

✧ 计数排序：

```
CountSort(lst):
    n=len(lst)
    num=max(lst)
    count=[0]*(num+1)
    for i in range(0,n):
        count[lst[i]]+=1
    arr=[]
    for i in range(0,num+1):
        for j in range(0,count[i]):
            arr.append(i)
    return arr
```

✧ 基数排序：

```
radix_ sort (A,d, k){
    for i=1 to d :
        counting_ sort(A,i,k)
    }
```

✧ 桶排序：

```
bucket_Sort(var A:List){
    n:=length(A);
    for i:=1 to n do
        将 A[i]插到表 B[floor(n*A[i])]中;
    for i:=0 to n-1 {
        用插入排序对表 B[i]进行排序;
        将表 B[0],B[1],...,B[n-1]按顺序合并;
    }
}
```

✧ 归并排序：

```
MergeSort(lst):
    //合并左右子序列函数
    merge(arr,left,mid,right):
        temp=[]     //中间数组
        i=left          //左段子序列起始
        j=mid+1   //右段子序列起始
        while i<=mid and j<=right:
            if arr[i]<=arr[j]:
                temp.append(arr[i])
                i+=1
            else:
                temp.append(arr[j])
                j+=1
        while i<=mid:
            temp.append(arr[i])
            i+=1
        while j<=right:
            temp.append(arr[j])
            j+=1
        for i in range(left,right+1):    //  !注意这里，不能直接 arr=temp,他俩大小都不一定一样
            arr[i]=temp[i-left]
    //递归调用归并排序
    mSort(arr,left,right):
        if left>=right:
            return
        mid=(left+right)//2
        mSort(arr,left,mid)
        mSort(arr,mid+1,right)
        merge(arr,left,mid,right)

    n=len(lst)
    if n<=1:
        return lst
    mSort(lst,0,n-1)
    return lst
```

✧ 堆排序：

```
HeapSort(lst):
    heapadjust(arr,start,end):  //将以 start 为根节点的堆调整为大顶堆
        temp=arr[start]
        son=2*start+1
        while son<=end:
            if son<end and arr[son]<arr[son+1]:  //找出左右孩子节点较大的
                son+=1
            if temp>=arr[son]:        //判断是否为大顶堆
                break
            arr[start]=arr[son]      //子节点上移
            start=son                //继续向下比较
            son=2*son+1
        arr[start]=temp              //将原堆顶插入正确位置

    n=len(lst)
    if n<=1:
        return lst
    //建立大顶堆
    root=n//2 - 1    //最后一个非叶节点（完全二叉树中）
    while(root>=0):
        heapadjust(ls,root,n-1)
        root-=1
    //掐掉堆顶后调整堆
    i=n-1
    while(i>=0):
        (lst[0],lst[i])=(lst[i],lst[0])  //将大顶堆堆顶数放到最后
        heapadjust(lst,0,i-1)    //调整剩余数组成的堆
        i-=1
    return lst
```

附上各个排序算法在各自最好最差情况时间空间复杂度：

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)

第三章 系统实现

系统实现内容主要包含：

- 底层数据结构的设计及实现
- 各大算法的设计及实现
- 可视化图形界面 GUI 的设计及实现

底层数据结构的实现：

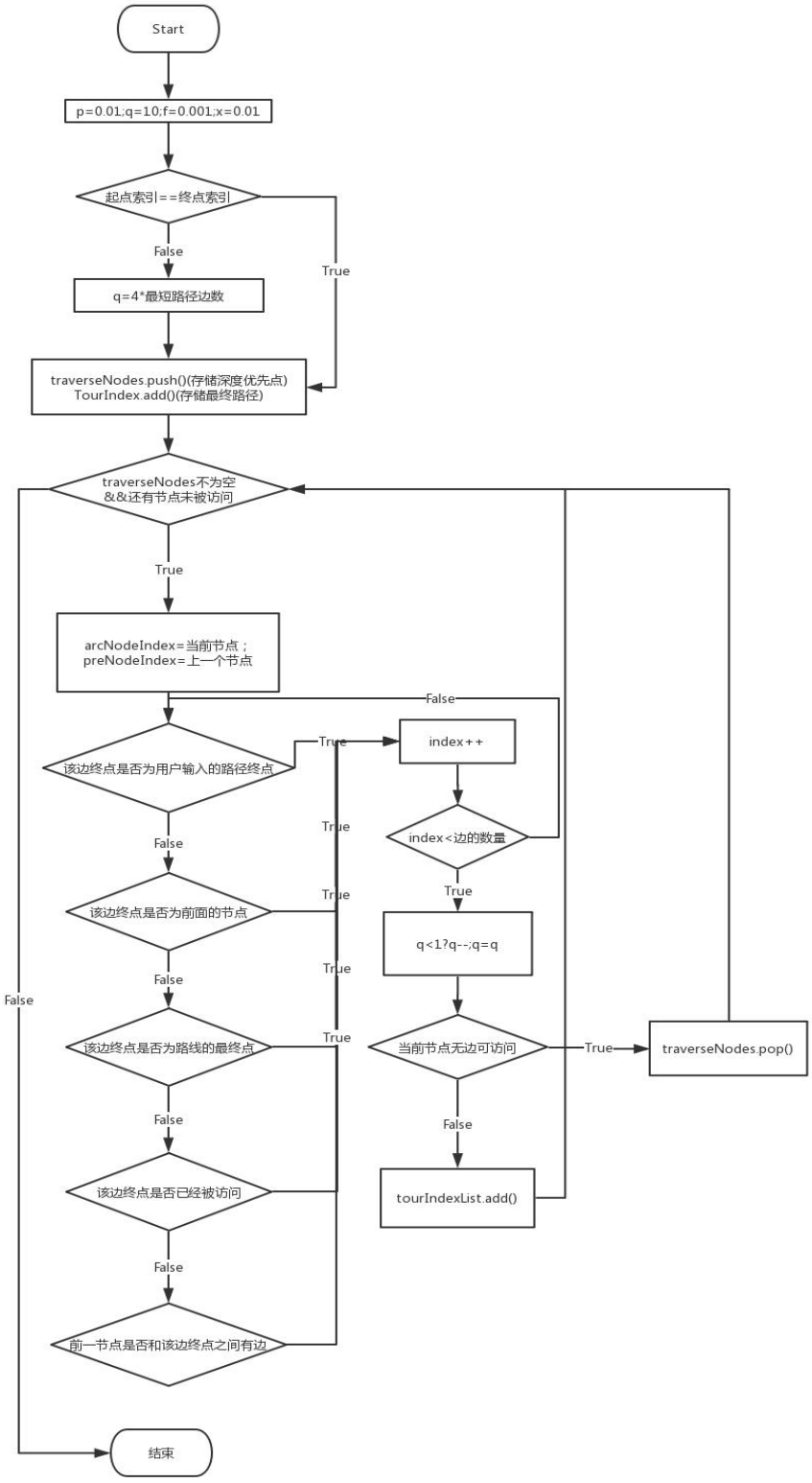
主要实现了三种基本数据结构的底层: MyList， MyQueue， MyStack，并构建了图类，还有根据景点，道路，停车场，车辆等信息设计了相应的 Node 类。

在第二章系统设计中的数据结构设计中有较为详细的根据上述实现的基本数据结构来进行景区管理系统相应功能的构建。在此再不进行赘述。

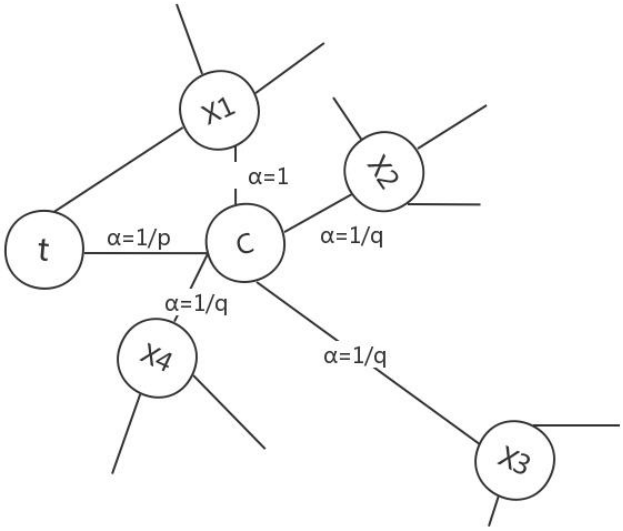
各大算法的设计及实现：

各大算法的基本描述，时间空间复杂度以及伪代码的实现均在上一章有描述下面讲述 BDFS+算法的实现。

下面是算法的流程图：



该算法的基本思想是结合深度优先和广度优先的优点，是俩者的一个结合，并可根据实际需要通过改变因子的值来是的该算法偏向 DFS 还是 BFS 算法。我通过引入深度优先控制因子 q，回溯控制因子 p，终点控制因子 f，已访问控制因子 x 四个变量来控制深度和广度的权重，从而在其中达到一个能够自由控制算法偏向性的一个程度。具体的做法如下：



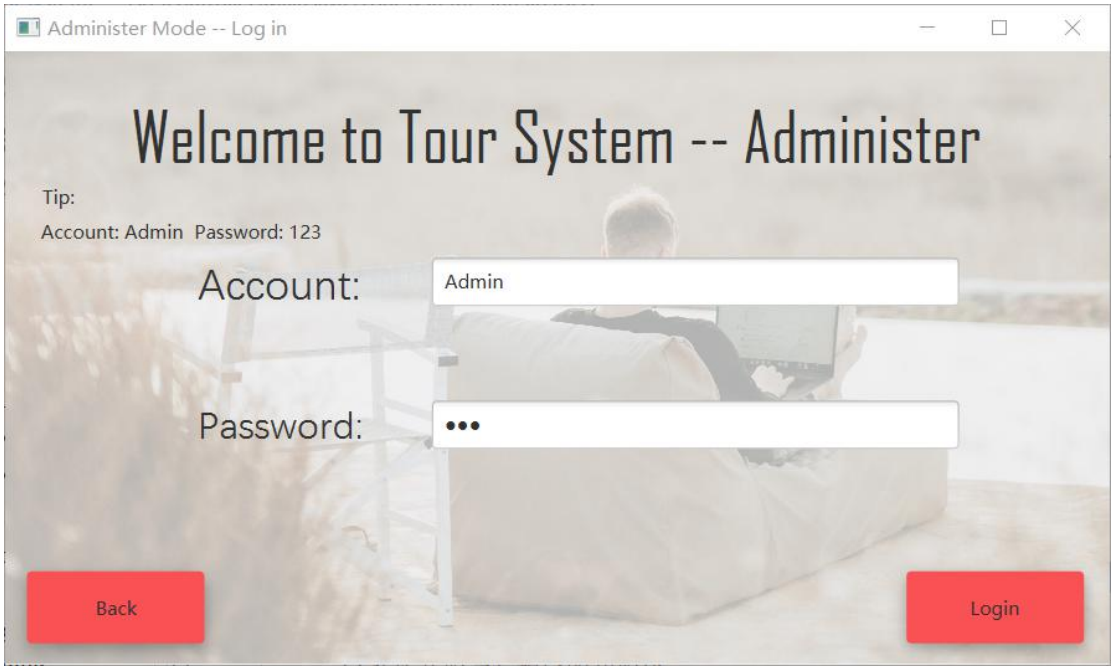
如图，C 是当前的节点，t 是路径的上一个节点。此时，算法应该做的是在图中的四条边中选取一条边继续向下遍历行走。对于节点 x1 来说，它与 t 相连，说明 t,C,x1 在一个局部区域内，意味着上一次的行走选择了 C 而未选择 x1，则令 α=1，代表着广度优先算法的权重。对于 x2,x3，x4 它们与 t 无边相连，它们由 1/q 控制， 代表着深度优先算法的权重。 对于 t，它是上一次访问的节点，由 1/p 控制。 同理，已访问的节点和终点分别由 1/x 和 1/f 控制（图中未展现）。这样，对每一条边，我们都可以求出对应的 α，再让 α 与边的权重相乘，得出一个新的权重，然后访问新的权重最小的边。

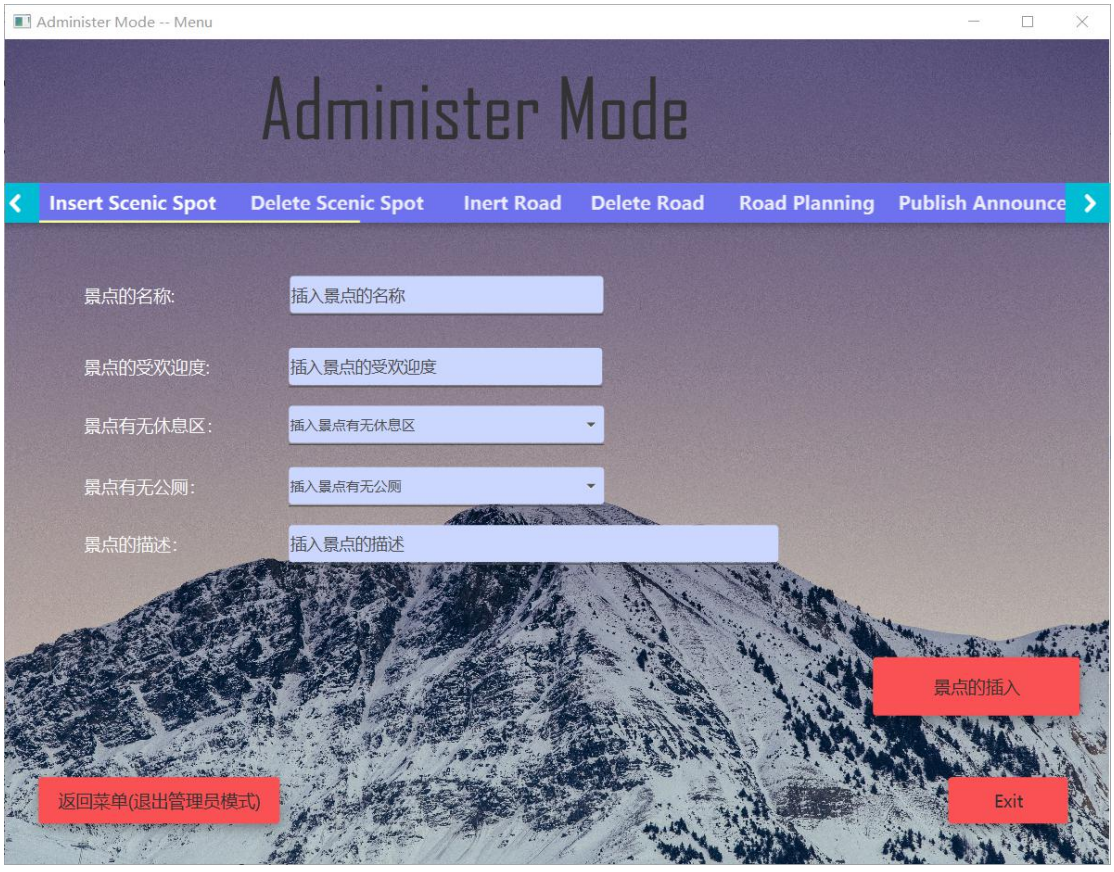
我们可以调整四个因子的值来达到我们在实际运用此算法的过程中想要的效果。比如说，当我们把 q 设置的非常大的时候，对应的边的新权值就会很小，这样整个路线更倾向于深度优先遍历。当我们把 p 设置的非常很大的时候，整个路线更倾向于广度优先遍历。其余两个因子也是同样的道理。

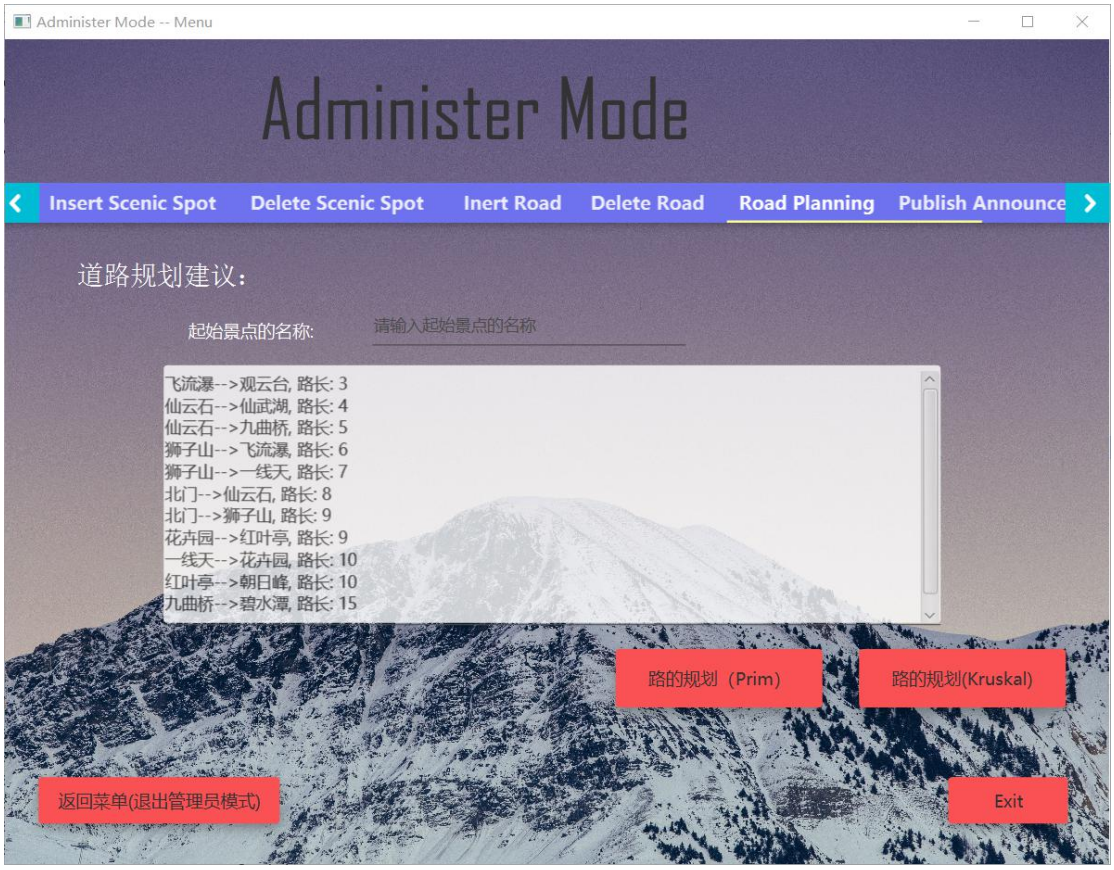
在实验过程我发现，当起点和终点之间最短路径所经过的节点越少（极限情况是起点与终点相同），**深度遍历因子越大，实验的效果越好**，反之亦然。这可以在图上有较为直观的感觉，所以进一步的尝试可以放在是否根据起点和终点的情况来调整深度遍历因子（广度遍历因子恒为 1），甚至说可以根据起点和终点的距离实时情况来调整深度遍历因子。实验结果说明，这样的调整在一些情况下是卓有成效的，具体内容将在第四章展现。

可视化图形界面 GUI 的设计及实现：

GUI 的实现，使用了 JavaFX。能够直观较好进行操作，较好地展示出景点导游路线，景点间最短路线，关键字查询与排序以及选择一些算法。下面是少量的图片展示：


Administer Mode -- Log in
Welcome to Tour System -- Administer
Tip:
Account: Admin Password: 123
Account:
Admin
Password:
Back
Login


Administer Mode -- Menu
Administer Mode
Insert Scenic Spot
Delete Scenic Spot
Inert Road
Delete Road
Road Planning
Publish Announce
景点的名称:
插入景点的名称
景点的受欢迎度:
插入景点的受欢迎度
景点有无休息区:
插入景点有无休息区
景点有无公厕:
插入景点有无公厕
景点的描述:
插入景点的描述
景点的插入
返回菜单(退出管理员模式)
Exit



第四章 系统测试

文件 I/O：

文件读取与运用上，读取存储过程不再赘述，在运用过程中，主要使用 JSON 的数据形式，将数据从文件中读取后，传入构造的函数中转化为相应的 JSON 数据形式，方便运用。在具体运用过程中也使用了 fastjson-1.2.2.jar 包对 JSON 数据进行转化和处理。

各大算法测试分析比较：

✧ 首先进行两种字符串匹配算法的测试和比较：

	KMP	Boyer - Moore
预处理时间	O(M)	O(N+M^2)
时间复杂度	O(N)	O(N)

✧ 十种排序算法的比较

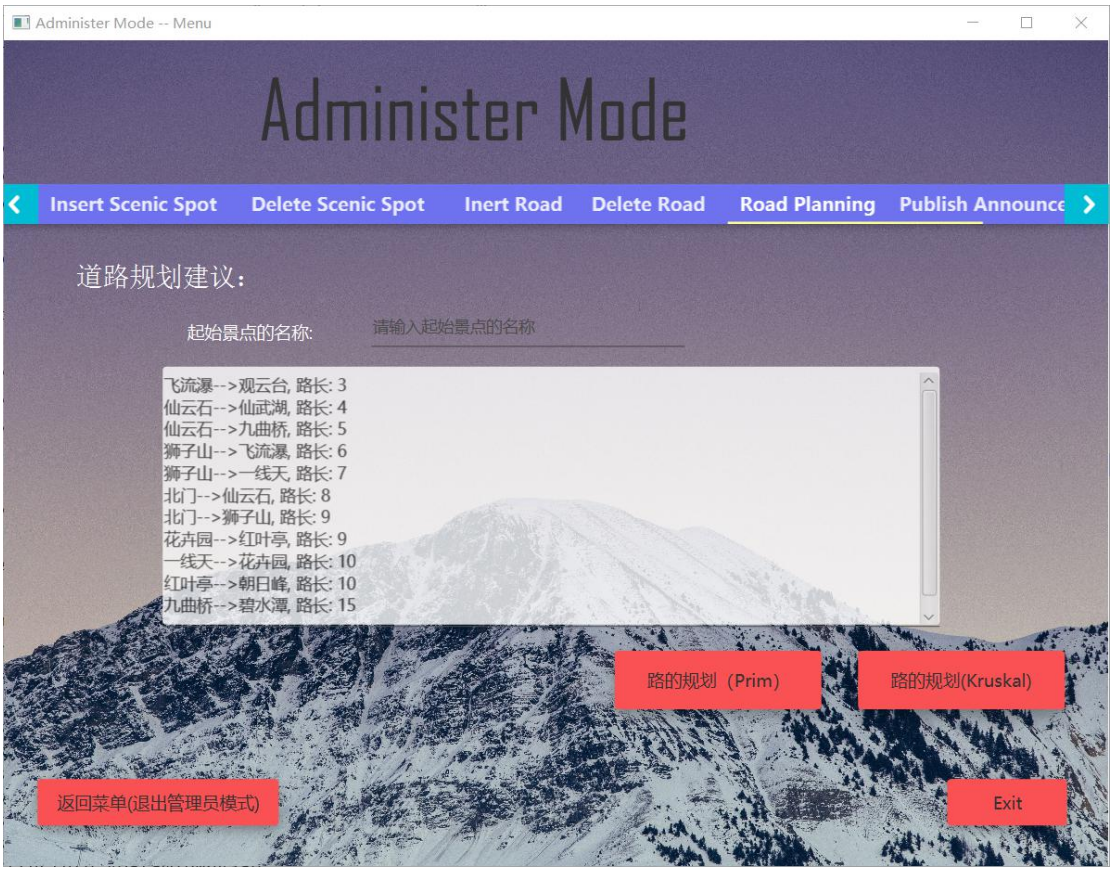
	冒泡排序	快速排序	选择排序	堆排序	希尔排序	基数排序	插入排序	归并排序	计数排序	桶排序
时间复杂度	O(n^2)	O(nlogn)	O(n^2)	O(nlogn)	O(nlogn)	O(n*k)	O(n^2)	O(nlogn)	O(n+k)	O(n)
空间复杂度	无	无	无	无	无	O(n)	无	O(n)	O(n)	O(n)
稳定	是	否	否	否	否	否	是	是	是	是

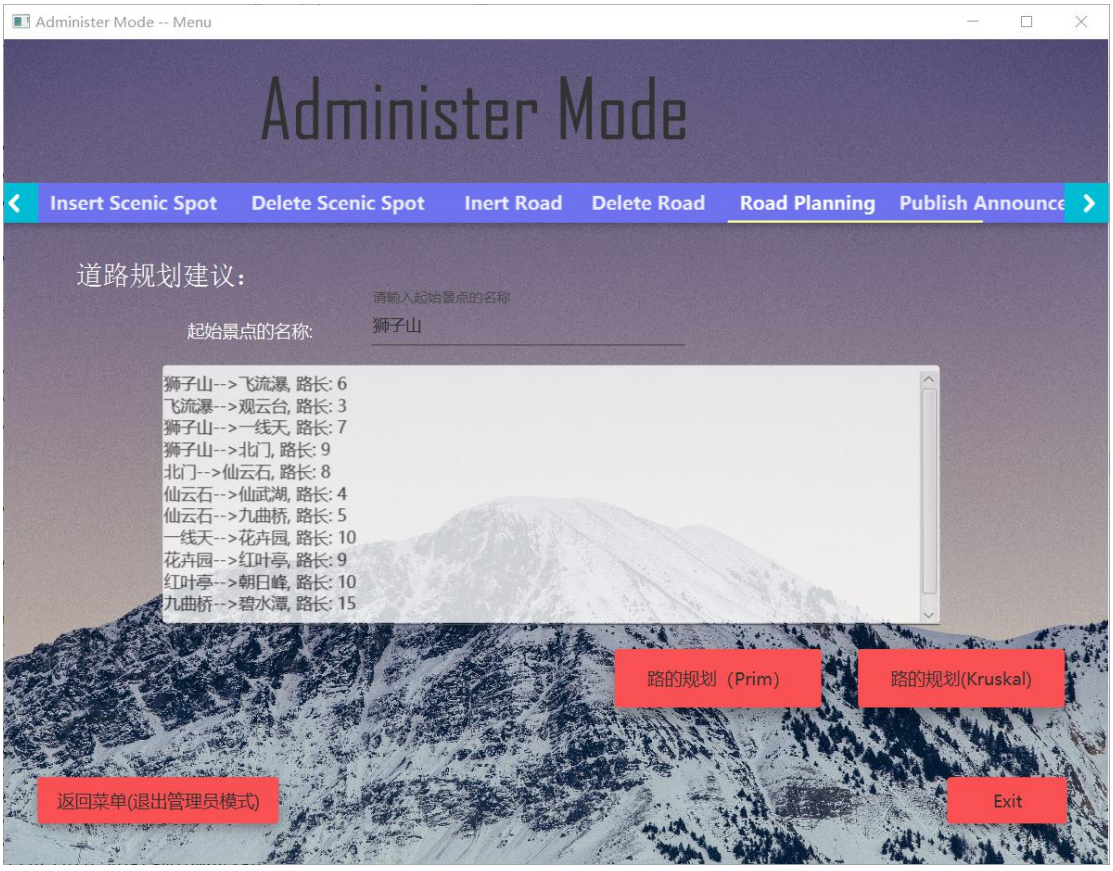
✧ 两种最小生成树的算法的比较

Prim 算法和 Kruskal 算法都是从连通图中找出最小生成树的经典算法。

从策略上来讲，Prim 算法是直接查找，多次寻找邻边的权重最小值，而 Kruskal 是需要先对权重排序后查找的。

所以可以看到使用 Kruskal 进行规划路线是有序的结果，而 Prim 算法是无序的。


Administer Mode -- Menu
Administer Mode
Insert Scenic Spot
Delete Scenic Spot
Inert Road
Delete Road
Road Planning
Publish Announce
道路规划建议：
起始景点的名称:
请输入起始景点的名称
飞流瀑-->观云台, 路长: 3
仙云石-->仙武湖, 路长: 4
仙云石-->九曲桥, 路长: 5
狮子山-->飞流瀑, 路长: 6
狮子山-->一线天, 路长: 7
北门-->仙云石, 路长: 8
北门-->狮子山, 路长: 9
花卉园-->红叶亭, 路长: 9
一线天-->花卉园, 路长: 10
红叶亭-->朝日峰, 路长: 10
九曲桥-->碧水潭, 路长: 15
路的规划（Prim）
路的规划(Kruskal)
返回菜单(退出管理员模式)
Exit


Administer Mode -- Menu
Administer Mode
Insert Scenic Spot
Delete Scenic Spot
Inert Road
Delete Road
Road Planning
Publish Announce
道路规划建议：
请输入起始景点的名称
起始景点的名称:
狮子山
狮子山-->飞流瀑, 路长: 6
飞流瀑-->观云台, 路长: 3
狮子山-->一线天, 路长: 7
狮子山-->北门, 路长: 9
北门-->仙云石, 路长: 8
仙云石-->仙武湖, 路长: 4
仙云石-->九曲桥, 路长: 5
一线天-->花卉园, 路长: 10
花卉园-->红叶亭, 路长: 9
红叶亭-->朝日峰, 路长: 10
九曲桥-->碧水潭, 路长: 15
路的规划（Prim）
路的规划(Kruskal)
返回菜单(退出管理员模式)
Exit

所以说，Kruskal 在算法效率上是比 Prim 快的，因为 Kruskal 只需一次对权重的排序就能找到最小生成树，而 Prim 算法需要多次对邻边排序才能找到。

✧ 四种最短路径算法的比较：

测试和分析了其时间复杂度，空间复杂度以及是否能解决负权问题。

	Dijkstra	Floyd	Bellman - Ford	SPFA
空间复杂度	O(M)	O(N^2)	O(M)	O(M)
时间复杂度	O((M+N)logN)	O(N^3)	O(MN)	O(MN)
负权问题	不可以	可以	可以	可以

✧ 四种路线规划的算法的分析，比较。

	汉密尔顿算法	DFS 算法	欧拉算法	BDFS+
北门—北门	136	171	263	126
狮子山—狮子山	111	148	263	111
红叶亭—红叶亭	109	191	263	131
北门—狮子山	100	157	263	141
一线天—碧水潭	108	132	263	124

由表格可知，欧拉算法本就是需走过每一条路，景区可能在道路两旁也设置了或者本就是非常优美的风景，所以欧拉算法在实际运用中是有其意义的。

从表格中的测试数据来看深度优先算法不太理想，过于死板，效果难以达标。

汉密尔顿算法确实是很稳定的一个算法，鲁棒性较高，但在测试环节中，BDFS+很多时候也会比汉密尔顿算法效果理想。同时我们可以注意到，汉密尔顿算法的时间复杂度(O(n!n))略高。

创新的 BDFS+算法的四个因子的设置，经过手动调试设置为设定回溯因子 p=0.01，深度优先因子 q = 4 * 节点之间最短路径的距离（回路则 q = 10），终点因子 f = 0.001 ，已访问因子 x = 0.01，此时能达到较好的效果。

因子的设置如果在大数据量的条件下，可以利用深度学习来进行调试。

在现实环境下，更多的反馈属性（如受欢迎度）的增加也能够帮助我们来优化 BDFS+。

第五章 结论

本此课设实验完成开发完成了一个简易的景区管理系统。

景区管理系统功能完善，包括：

管理员：

- 对景点进行增删改查
- 对景点之间的道路进行增删改查
- 发布公告通知

游客：

- 查询景点的分布图
- 查询推荐的导游路线
- 查询两景点间的最短路径
- 根据景点名称或其描述进行关键字查询
- 景点根据受欢迎程度或景点岔路口数进行排序查询
- 景区停车场车辆进出信息

主要实现了一些底层数据结构（如 list，stack，queue），帮助我巩固了数据结构方面的知识并加深了印象。同时还实现了 20 余种常用算法以及自己思考得出的一个算法，帮助自己在运用数据结构和思考问题方面成长学习到了很多。

参考文献

1.百度百科：SPFA 算法 https://baike.baidu.com/item/SPFA%E7%AE%97%E6%B3%95/8297411?fr=aladdin
2.百度百科：Bellman-Ford 算法 https://baike.baidu.com/item/Bellman-Ford%E7%AE%97%E6%B3%95

附录：

《数据结构课程设计》实验成绩评定表

评价内容	具　体　要　求	分值	得分
平时表现	课程设计过程中，无缺勤、迟到、早退现象，学习态度积极。能够主动查阅文献，积极分析系统中数据结构与算法的多种可能的设计方案，并认真地对所选择方案进行实现、测试、分析与总结。	20	
分析与解决问题的能力	能够理解复杂数据结构及算法的设计思路和基本原理；能够应用所学数据结构与算法等相关知识和技能去解决实验系统中要求的各个题目；设计或实现思路有独特见解。	20	
实验结果与工作量	能够按实验要求完成系统的开发与测试，并达到实验要求的预期结果；能够认真记录实验数据，并对实验结果分析准确，归纳总结充分；工作量饱满。	20	
报告质量	实验报告文字通顺、格式规范，体例符合要求；报告内容充实、正确，实验目的归纳合理到位。	40	
总　　分			