

课程编号: C0801207040

数据结构与算法 课程设计报告



姓 名	熊悦	学 号	20175362
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 学年秋季学期		
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

系统背景

本次课程设计的题目是设计一个大型景区的管理系统。系统用户包括管理员和游客两类，管理员负责管理景区的景点维护；游客可以根据自己的需求对景区进行各种信息查询以及路线规划等。

功能需求

根据系统的使用者分析功能需求。

对于游客模块，需要实现以下几个功能

1.获取景区景点和路的分布情况，需要在终端输出由景点信息和路所构成的邻接矩阵。对于此功能，首先需要从文件中获取景区景点的基本信息并初始化景区图结构。根据以此生成的邻接链表转化成邻接矩阵，并输出结果。

2.通过关键字对景区景点的查询，查询的范围包括景点的名字和简介。实现这一功能，需要使用相关算法将关键字与相应范围内的数据进行匹配，如果匹配成功则返回景点以及相关简介，如果查找不成功就给予相关提示。

3.根据不同需求对景区景点进行排序。这里主要需要根据景点的欢迎度或者景点的岔路数进行排序。

4.获取两个景点之间的最短路径和最短距离。

5.获取导游路线图，即周游所有景点的路线图。游客可以选择的周游方式有两种，第一种是从进入景区和离开景区的景点相同；另一种则是分别从不同的景点分别进入和离开景区。

6.游客车辆进出停车场时，输出车辆的进出信息。系统需要在车辆请求进入停车场时判断是否让其进入便道等待；汽车离开停车场时，系统需要进行车费结算，并让该车后面的车辆暂时进入缓冲区避让，若此时有车在便道等待，则需让其在最后进入停车场。

对于管理员模块，需要实现以下几个功能

1.对景点的插入和删除。景区可能会开发或者维护景点，因此需要将景点进行删除，而在图中，对于景点的删除会将相关的边删除。为了减轻系统负荷，这里需要在删除的过程中避免重现构建数组或者图结构。

2.对于路的插入和删除。不同景点之间可能会添加新路或者删除已经存在的。

3.发布通知公告，将信息显示给用户。

解决方案

游客模块

景区分布图显示：

该模块并不复杂，将邻接链表转化为邻接矩阵即可。。

景点查找：

鉴于功能主要是字符串匹配，因此采用当下效率最高的字符串匹配算法之一 Boyer

Moore 算法来进行匹配。

景点排序：

针对根据景点欢迎度和景点岔路数两种不同需求的排序方式，采用快速排序分别进行处理和排序。

获取两点间最短路：

目前普遍通用的获取图中节点间最短路的算法由 Dijkstra 算法和 Floyd 算法，笔者在项目中分别实现了两种算法。由于该功能要解决问题是求取单源最短路，故使用 Dijkstra 算法效率更高，在此采用这种算法。

获取导游路线图：

导游路线图的获取问题在一定意义上可以转化为求取图中最短汉密尔顿回路的问题。但在实际中又会出现一些特殊情况，比如不存在汉密尔顿回路，一些边或者节点必须经过两次。针对这样的需求，笔者采取深度遍历+Floyd 求取最短路的方式，在能够找到最短汉密尔顿回路的情况下直接输出回路。如果不行，就需要先将各种遍历图情况记录下来，之后再使用 Floyd 算法求取各次遍历的终点到出口的最短距离进行比较，获取最短路径。

停车场模块：

针对停车场管理流程，设计的算法思路如下。当车辆进入停车场时，先检验停车场是否已满，如果未满则车辆进入，如果已满，则车辆进入便道等待。

车辆离开停车场时，先让在它之后进入停车场的车辆退出停车场进入缓冲区等待，之后再依次进入停车场，此时若便道上有车辆等待，则最先等待的车辆进入停车场。

在这个过程中，需要用栈来分别模拟停车场和缓冲区，用一个队列来模拟便道。

管理员模块

景点的插入和删除：

景点插入在实现中并不复杂，直接在集合中加入相应景点即可。有关景点的删除，在删除该景点的同时，还应注意将与之相关的边一同删除。

边的插入与删除：

和景点的插入与删除类似，在插入和删除边时，需要对与边相关的两个点进行操作，删除相关边。

主要工作












根据用户对系统的功能，笔者对系统各项功能和需求进行了分析和实现。构建相关数据结构和算法。除此之外，为了优化用户使用系统的体验，还实现了系统的可视化

第二章 系统设计

在本项目，主要设计并使用了以下抽象数据结构








链表(MyLinkedList)

实现了标准链表中的一些基本接口，包括添加节点，删除节点。根据项目需求，还实现了诸如不重复的添加(addNotSame)等方法。

C  MyLinkedList		
m 	size()	int
m 	get(int)	T
m 	add(T)	void
m 	contain(T)	boolean
m 	addNotSame(T)	boolean
m 	add(int, T)	void
m 	remove(int)	void
<hr/>		
p 	size	int
p 	empty	boolean
p 	iterator	Iterator<T>








队列(MyQueue)

为了提高效率，使用链表构建队列，实现了普通队列的相关接口。

C  MyQueue		
m 	pop()	T
m 	peek()	T
m 	push(T)	void
m 	size()	int
<hr/>		
p 	empty	boolean
p 	iterator	Iterator<T>

栈(MyStack)

基于链表实现的栈，实现了普通栈的相关接口。

C  MyStack		
m 	pop()	T
m 	peek()	T
m 	push(T)	void
m 	size()	int
<hr/>		
p 	empty	boolean
p 	iterator	Iterator<T>

下面分别介绍项目中的数据类型：

图(Graph)

类图

Graph	
m getPosition(String)	int
m getDistance(int, int)	int
m addEdge(String[])	boolean
m addVertex(ScenicSpot)	void
m removeEdge(String[])	void
m removeVex(String)	int
m outputAdj()	int[]
m outputGraph()	String[]
m sortByWelcomed()	String[]
m sortByPaths()	String[]
m findWay(String, String)	String[]
m createTourSortGraph(String, String)	String[]
vertices	MyLinkedList<Vertex>
vertexNum	int
INFINITY	int
edgeNum	int

属性说明

本项目几乎所有的功能都涉及到对图的应用。在图中创建了以下属性：用链表存储的所有节点集合(vertices)，图中节点和边的数量(vertexNum, indexNum)以及代表无限大的数值(INFINITY)。

方法说明

getPosition(String): 输入节点名字，返回节点在节点数组中的索引

getDistance(int, int): 输入两个节点在节点数组中的索引位置，返回它们之间边的权值。

若无边则返回无穷大

addEdge(String[]): 输入一个包含边的信息的数组，添加相关边

addVertex(ScenicSpot): 输入一个景点对象，产生一个节点

removeEdge(String[]): 输入一个包含边的信息的数组，删除相关边

outputAdj(): 产生邻接矩阵

sortedByPaths(): 返回一个按照岔路数排序的景点名字数组

sortedByWelcome(): 返回一个按照欢迎度排序的景点名字数组

findWay(String, String): 输入起点和终点，返回它们之间的最短路

createTourSortGraph(String, String): 输入起点和终点，返回导游路线图

点(Vertex)

类图:

Vertex	
m compareTo(Vertex)	int
m addEdge(EdgeNode)	boolean
m removeEdge(String)	boolean
m equals(Object)	boolean
name	String
edge	MyLinkedList<EdgeNode>
data	ScenicSpot

属性说明

组成图的基本元素，属性包括点的名字(name)，数据(data)和与该点相关的边集(edge)。

方法说明

compareTo(Vertex): 用于按照节点岔路数比较节点

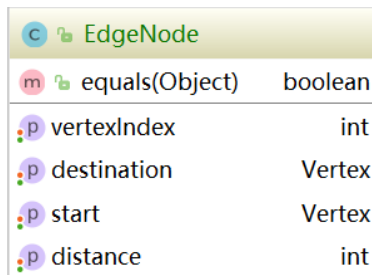
addEdge(EdgeNode): 输入一个边, 构成包含该节点的一条边

removeEdge(String): 输入边的另一个节点, 删除与该节点相关的边

equals(String): 用于判断节点是否相等

边(EdgeNode)

类图



参数说明

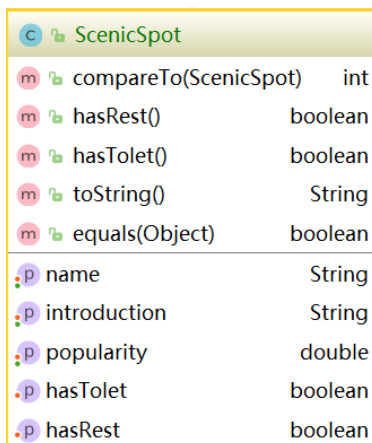
本项目是通过存储构成边的起点终点以及边的权重信息来表示边的, 此外, 为了方便之后的操作, 在边中创建一个索引变量(vertexIndex)来获取边的两点在点集数组中的位置。需要注意的是, 边在图(Graph)类中并不独立存在, 而是存储在各个与之相关的点中保存在图里。

方法说明

equals(Object): 用于判断边是否相同

景点(ScenicSpot)

类图



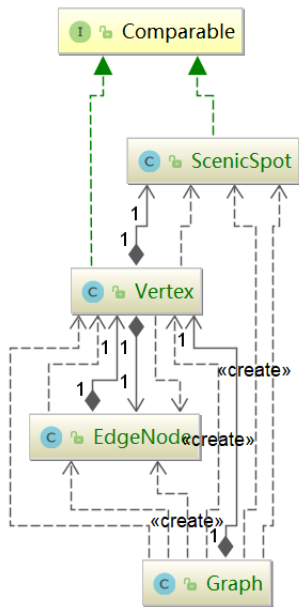
参数说明

它是图中节点所包含的数据, 依据项目任务书要求, 包含景点名字(name), 景点介绍(introduction), 受欢迎程度(popularity)以及是否有厕所(hasToilet)和休息室(hasRest)。

方法说明

compareTo(): 判断景点是否相等。

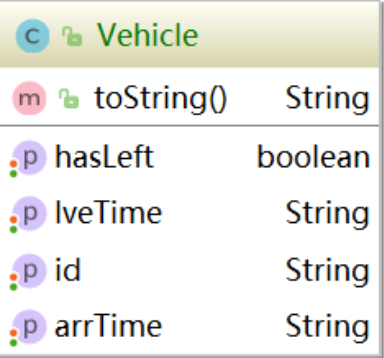
以上几个模块构成了景区景点有关的所有数据结构。总结一下，图中包含有点集和边集，边以节点的形式存储在节点中，景点以数据的形式存储在节点中，具体关系如下图所示。



下面介绍停车场管理相关的数据结构。

汽车(Vehicle)

类图



属性说明

停车管理系统中最基本的数据类型，包含以下几个属性，是否离开(hasLeft)，离开时间(lveTime)，车牌号(id)，到达时间(arrTime)。

算法设计方案

下面依次给出各个模块的算法设计方案

Boyer Moore 算法设计方案

目标

输入参数包括关键字和目标字符，需要判断目标字符中是否包含关键字。

设计方案

根据 Boyer Moore 算法特性预先根据目标字符串生成《坏字符规则表》和《好后缀规则表》。使用时，根据“坏字符规则”和“好后缀规则”从尾部开始逐位比较，分别计算出关键字后移位数，取得最大值得到移位数。若发现全部匹配，则搜索结束。

Dijkstra 算法设计方案

目标

输入参数包括起点终点在点集中的索引位置和它们所在的图，获取两点间的最短距离并输出最短路径。

设计方案

首先通过图(Graph)中封装获取所有点集，引入两个集合（S、U），S 集合包含已求出的最短路径的点（以及相应的最短长度），U 集合包含未求出最短路径的点。通过迭代从起点开始，每次循环找到为探索节点中到该点最短距离的点，将其加入集合 S，再更新 U 集合，直至遍历结束，所有的点都在集合 S 中，即可获得最短距离。再根据中途保存的到达矩阵获得路径。

Floyd 算法设计思路

目标

输入参数包括起点终点在点集中的索引位置和它们所在的图，需要获取两点间的最短距离并输出最短路径。

设计方案

引入两个矩阵，dist[][]和 path[][]，分别存储点 i 到点 j 的距离和 i 到 j 所经过的顶点。设图中由 n 个点，则对 dist[][]和 path[][]进行 N 次更新，通过扫描每一个点并以此为基点再遍历所有的 dist[][]值，看是否可以通过该点让这对顶点的距离变小。迭代完成时，再分别根据两个矩阵获取路径和路径最小值。

Hamilton 算法设计思路

目标

输入参数包括起点和终点的名字和它们所在的图，需要获得以它们为起点和终点的汉密尔顿路，如果不存在，则通过允许走重复节点的方式遍历图中所有节点并到达终点

设计方案

从起点开始先进行深度优先搜索，当遍历完所有节点时便检查是否到达终点，如果是，就输出路径；否则，记录当前路径。若无法找到从起点到终点的汉密尔顿路，则用 Floyd 算法求出所有保存路径的终点到输入终点的最短距离，取到最小值，进行输出。

快速排序算法设计思路

目标

输入参数包括待排序的数组，排序起始位置，排序终止位置和一个用于给抽象数据类排序的 Comparator，需要获取一个按照要求从大到小排列的数组。

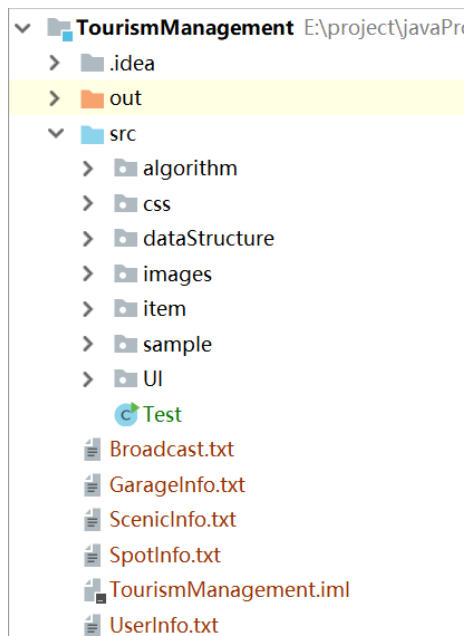
设计方案

通过分治法思想去一个基准将数组分为两部分，使左子区变量小于基准，右子区变量均大于等于基准，再用递归重复此过程，当递归调用结束时，再进行组合，获得有序数组。

第三章 系统实现

3.1 系统结构

本系统主要由以下几个包组成



通过分离操作方法，数据存储，让程序符合 MVC 设计模式。

3.2 系统特点

本系统在实现过程中运用良好的面向对象程序设计方法，对各个数据相关方法进行了封装，使用 UI 类将程序需求功能和实体类区分开，降低代码耦合度，提升程序的鲁棒性。针对功能需求，项目将系统分为了景区模块和停车场模块并分别创造两个不同的 UI 类对功能进行实现。

景区 UI

车库 UI

下面详细介绍各个算法的实现(有关算法的相关介绍包含在注释中)。

3.3.1 Boyer Moore 算法实现

```
/**
 * 方法类, 包含一个 Boyer Moore 算法方法, 做字符串匹配
 */
public class BoyerMoore {

    public static int match(String pattern, String target) {
        int tLen = target.length(); // 目标字符串长度
        int pLen = pattern.length(); // 输入字符串长度
        int[] bad_table = build_bad_table(pattern); // 坏字符规则表
        int[] good_table = build_good_table(pattern); // 好后缀规则表

        // 先判断字符串长度能否比较
        if (pLen > tLen) {
            return -1;
        }

        // 比较
        for (int i = pLen - 1, j; i < tLen;) {
            for (j = pLen - 1; target.charAt(i) == pattern.charAt(j); i--, j--) {

                if (j == 0) {
                    return i;
                }
            }

            // 取好后缀和坏字符规则移位最大值
            i += Math.max(good_table[pLen - j - 1], bad_table[target.charAt(i)]);
        }
        return -1;
    }

    /**
     * 字符信息表, 构建坏字符规则表
     */
    public static int[] build_bad_table(String pattern) {
        final int table_size = 40000;
        int[] bad_table = new int[table_size];
        int pLen = pattern.length();

        for (int i = 0; i < bad_table.length; i++) {
            bad_table[i] = pLen; // 默认初始化全部为匹配字符串长度
        }
        for (int i = 0; i < pLen - 1; i++) {
            int k = pattern.charAt(i);
            bad_table[k] = pLen - 1 - i;
        }

        return bad_table;
    }
}
```

```

    * 字符信息表, 构建好后缀规则表
    */
    public static int[] build_good_table(String pattern) {
        int pLen = pattern.length();
        int[] good_table = new int[pLen];
        int lastPrefixPosition = pLen;

        for (int i = pLen - 1; i >= 0; --i) {
            if (isPrefix(pattern, i + 1)) {
                lastPrefixPosition = i + 1;
            }
            good_table[pLen - 1 - i] = lastPrefixPosition - i + pLen - 1;
        }

        for (int i = 0; i < pLen - 1; ++i) {
            int slen = suffixLength(pattern, i);
            good_table[slen] = pLen - 1 - i + slen;
        }
        return good_table;
    }

    /**
     * 前缀匹配
     */
    private static boolean isPrefix(String pattern, int p) {
        int patternLength = pattern.length();
        // 前缀移位位数
        for (int i = p, j = 0; i < patternLength; ++i, ++j) {
            if (pattern.charAt(i) != pattern.charAt(j)) {
                return false;
            }
        }
        return true;
    }

    /**
     * 后缀匹配
     */
    private static int suffixLength(String pattern, int p) {
        int pLen = pattern.length();
        int len = 0;
        // 移位位数
        for (int i = p, j = pLen - 1; i >= 0 && pattern.charAt(i) ==
pattern.charAt(j); i--, j--) {
            len += 1;
        }
        return len;
    }
}

```

3.3.2 Dijkstra 算法实现

```

/**
 * Dijkstra 算法
 */
public class Dijkstra {

```

```

public static String[] dijkstra(Graph graph, int start, int dest) {
    int num=0;
    int vertexNum=graph.getVertexes().size();
    int[] prev = new int[vertexNum]; //prev—到达i 节点的前一个节点
    int[] dist = new int[vertexNum]; //dist—长度数组。
    Vertex[] matrix=new Vertex[vertexNum];
    MyStack<String[]> path=new MyStack<>();
    for(int i=0; i<vertexNum; i++){
        matrix[i]=graph.getVertexes().get(i);
    }

    // flag[i]=true 表示"顶点vs"到"顶点i"的最短路径已成功获取。
    boolean[] flag = new boolean[vertexNum];

    // 初始化
    for (int i = 0; i < vertexNum; i++) {
        flag[i] = false;           // 顶点i 的最短路径还没获取到。
        prev[i] = 0;               // 顶点i 的前驱顶点为0。
        dist[i] = graph.getDistance(start, i); // 顶点i 的最短路径为起点到"
        顶点i"的权。
    }

    // 对起点进行初始化
    flag[start] = true;
    dist[start] = 0;

    // 遍历; 每次找出一个顶点的最短路径。
    int k = 0;
    for (int i = 1; i < vertexNum; i++) {

        // 寻找当前最小的路径;
        // 即, 在未获取最短路径的顶点中, 找到离vs 最近的顶点(k)。
        int min = graph.getINFINITY();
        for (int j = 0; j < vertexNum; j++) {
            if (!flag[j] && dist[j]<min) {
                min = dist[j];
                k = j;
            }
        }

        // 标记"顶点k"为已经获取到最短路径
        flag[k] = true;

        // 修正当前最短路径和前驱顶点
        // 即, 当已经"顶点k 的最短路径"之后, 更新"未获取最短路径的顶点的最短路径和前驱顶点"。
        for (int j = 0; j < vertexNum; j++) {
            int tmp = graph.getDistance(k, j);
            tmp = (tmp==graph.getINFINITY() ? graph.getINFINITY() : (min +
            tmp)); // 防止溢出
            if (!flag[j] && (tmp<dist[j])) {
                {
                    dist[j] = tmp;
                    prev[j] = k;
                }
            }
        }
    }
}

```

```

//打印结果
int p=dest;
while(p!=start){
    String p1=matrix[p].getName();
    p=prev[p];
    String p2=matrix[p].getName();
    String[] tmp={p2, p1};
    path.push(tmp);
}

String[] res=new String[path.size()];
while(!path.isEmpty()){
    String[] tmp=path.pop();
    res[num++]=tmp[0]+"—" +tmp[1];
}
return res;
}
}

```

3.3.3Floyd 算法实现

```

/**
 * Floyd 算法
 */
public class Floyd {
    public static String[] floyd(Graph graph, int start, int dest) {
        //初始化
        int vertexNum=graph.getVertexes().size();
        int[][] path = new int[vertexNum][vertexNum]; // 路径。path[i][j]=k 表示, "顶点i"到"顶点j"的最短路径会经过顶点k。
        int[][] dist = new int[vertexNum][vertexNum]; // 长度数组。即, dist[i][j]=sum 表示, "顶点i"到"顶点j"的最短路径的长度是sum。
        Vertex[] matrix=new Vertex[vertexNum];
        for(int i=0; i<vertexNum; i++){
            matrix[i]=graph.getVertexes().get(i);
        }
        // 初始化
        for (int i = 0; i < vertexNum; i++) {
            for (int j = 0; j < vertexNum; j++) {
                dist[i][j] = graph.getDistance(i, j); // "顶点i"到"顶点j"的路径长度为"i 到j 的权值"。
                path[i][j] = j; // "顶点i"到"顶点j"的最短路径是经过顶点j。
            }
        }

        // 计算最短路径
        for (int k = 0; k < vertexNum; k++) {
            for (int i = 0; i < vertexNum; i++) {
                for (int j = 0; j < vertexNum; j++) {

                    // 如果经过下标为k 顶点路径比原两点间路径更短,则更新dist[i][j]和path[i][j]
                    int tmp = (dist[i][k]==graph.getINFINITY() || dist[k][j]==graph.getINFINITY()) ? graph.getINFINITY() : (dist[i][k] + dist[k][j]);
                    if (dist[i][j] > tmp) {

```

```

        // "i 到j 最短路径"对应的值设, 为更小的一个(即经过k)
        dist[i][j] = tmp;
        // "i 到j 最短路径"对应的路径, 经过k
        path[i][j] = path[i][k];
    }
}
}

// 打印floyd 最短路径的结果
String[] res=new String[2];
int tmp = path[start][dest];
String road="";
road+=matrix[start].getName();
while(tmp != dest){
    road+="-> " + matrix[tmp].getName();
    tmp = path[tmp][dest];
}
road+="-> " +matrix[dest].getName();
res[0]=road;
res[1]=dist[start][dest]+"";
return res;
}
}

```

3.3.4 Hamilton 算法实现

```

/**
 * 通过 Floyd 和 DFS 获取最短导游图和汉密尔顿回路
 */
public class MyHamilton {
    private int beginNum; //起始节点在数组中的位置
    private int endNum; //终点在数组中的位置
    private int vertexNum; //节点数量
    private String begin; //起始节点名字
    private String end; //结束节点名字
    private Graph graph; //根据节点和边生成的图
    private Vertex[] matrix; //节点数组
    private boolean key; //判断深度遍历后能否到达终点
    private MyLinkedList<MyLinkedList<String>> roads; //存储路径
    private String road; //存储路径

    public MyHamilton(String begin, String end, Graph graph){
        vertexNum=graph.getVertexes().getSize();
        this.graph=graph;
        this.begin=begin;
        this.end=end;
        matrix=new Vertex[vertexNum];
        for(int i=0; i<vertexNum; i++){
            matrix[i]=graph.getVertexes().get(i);
        }
        beginNum=graph.getPosition(begin);
        endNum=graph.getPosition(end);
        roads =new MyLinkedList<>();
        road="";
        haven="";
    }
}

```

```

}

public String[] getHamiltonCircuit(){
    boolean[] used=new boolean[vertexNum]; //用于标记图中顶点是否被访问
    int[] path=new int[vertexNum]; //记录哈密顿回路路径
    for(int i=0; i<vertexNum; i++){
        used[i]=false; //初始化, 所有顶点均未被遍历
        path[i]=-1; //初始化, 未选中起点及到达任何顶点
    }

    used[beginNum]=true; //表示从第1个顶点开始遍历
    path[0]=beginNum; //表示哈密顿回路起点为第0个顶点
    int[][] adj=graph.outputAdj();
    String[] result=new String[2];
    key=dfs(adj, path, used, 1); //从第0个顶点开始进行深度优先遍历, 如果存在哈密顿回路, 输出一条回路, 否则无输出

    result=addPath();
    return result;
}

/*
 * 参数 step: 当前行走的步数, 即已经遍历顶点的个数
 */
public boolean dfs(int[][] adj, int[] path, boolean[] used, int step){
    if(step==adj.length){ //当已经遍历完图中所有顶点

        //如果找到终点并且起点和终点相同
        if((beginNum==endNum)&&(adj[path[step - 1]][endNum] != 32767)) {
            for (int i = 0; i < path.length; i++)
                road += matrix[path[i]].getName() + "—>";
            road += matrix[endNum].getName();
            return true;
        }

        //如果找到终点并且起点和终点不同
        if((beginNum!=endNum)&&(path[step - 1] == endNum)){
            for (int i = 0; i < path.length-1; i++)
                road += matrix[path[i]].getName() + "—>";
            road+=matrix[path[path.length-1]].getName();
            return true;
        }

        //遍历后未找到终点, 记录当前位置和路径
        else {
            MyLinkedList<String> temp = new MyLinkedList<>();
            for (int i = 0; i < path.length-1; i++)
                road+=matrix[path[i]].getName() + "—>";
            temp.add(road);
            temp.add(path[path.length - 1] + "");
            roads.add(temp);
            road="";
            haven="";
            return false;
        }

        //进行遍历
    }
}

```



```

    }else {
        for(int i=0; i<adj.length; i++){
            if(!used[i]&&adj[path[step-1]][i]!=32767){
                used[i]=true;
                path[step]=i;
                if(dfs(adj, path, used, step+1)){
                    return true;
                }
            }else{
                used[i]=false;
                path[step]=-1;
            }
        }
    }
}
return false;
}

/**
 * 生成并返回最短路
 */
public String[] addPath(){
    // 如果找到终点, 直接返回路径
    String[] result=new String[2];
    if(key==true){
        result[0]=road;
        return result;
    }

    // 如果未能找到终点, 就用 Floyd 寻找当前到达点和终点的最短路并进行比较获得最
    短路
    else {
        try {
            int[] pend = new int[roads.size()];
            int shortestIndex = 0;
            String[] shortest = Floyd.floyd(graph,
Integer.parseInt(roads.get(0).get(1)), endNum);
            for (int i = 1; i < pend.length; i++) {
                String[] tempShortest = Floyd.floyd(graph,
Integer.parseInt(roads.get(i).get(1)), endNum);
                int a = Integer.parseInt(tempShortest[1]);
                int b = Integer.parseInt(shortest[1]);
                if (a < b) {
                    shortest = tempShortest;
                    shortestIndex = i;
                }
            }
            result[0] = roads.get(shortestIndex).get(0) + shortest[0];
            return result;
        }catch (Exception ex){
            result[0]="";
            return result;
        }
    }
}
}
}

```

3.3.5 快速排序算法实现

```
/**
 * 快速排序类，提供快速排序方法
 */
public static void quickSort(Object[] a,int low,int high, Comparator c){
    //可以对所有数据类型排序，需要Comparator
    int start = low;
    int end = high;
    Object key = a[low];

    while(end>start){
        //从后往前比较
        while(end>start&& c.compare(a[end], key)>=0) //如果没有比关键值小的，比较下一个，直到有比关键值小的交换位置，然后又从前往后比较
            end--;
        if(c.compare(a[end], key)<=0){
            Object temp = a[end];
            a[end] = a[start];
            a[start] = temp;
        }
        //从前往后比较
        while(end>start&& c.compare(a[start], key)<=0) //如果没有比关键值大的，比较下一个，直到有比关键值大的交换位置
            start++;
        if(c.compare(a[start], key)>=0){
            Object temp = a[start];
            a[start] = a[end];
            a[end] = temp;
        }
        //此时第一次循环比较结束，关键值的位置已经确定了。左边的值都比关键值小，右边的值都比关键值大，但是两边的顺序还有可能是不一样的，进行下面的递归调用
    }
    //递归
    if(start>low) quickSort(a,low,start-1, c); //左边序列。第一个索引位置到关键值索引-1
    if(end<high) quickSort(a,end+1,high, c); //右边序列。从关键值索引+1 到最后一个
}
```

3.4 功能实现部分

游客部分

系统初始化&输出景区分布图(邻接矩阵)

这一部分主要涉及的是文件的读取和数据的输出

输出景区分布图

返回一个由景点生成的邻接矩阵

```
/**
 * 打印景区分布图
 */
```

```

public String[][] outputGraph(){
    vertexNum=vertexes.size(); // 获取图中节点数量
    String[][] res=new String[vertexNum+1][vertexNum+1]; // 带返回的结果

    // 用迭代获取每一个节点所包含的边对应的另一个节点的位置
    for(int i=0; i<vertexNum; i++){
        Vertex tmp=vertexes.get(i);
        EdgeNode tmpEdge;
        for(int j=0; j<tmp.getEdge().size(); j++){
            tmpEdge=tmp.getEdge().get(j);

            tmpEdge.setVertexIndex(getPosition(tmpEdge.getDestination().getName()));
        }
        res[0][0]="景点分布图"; // 方便打印

        // 打印的表头
        for(int i=0; i<vertexNum; i++){
            res[0][i+1]=vertexes.get(i).getName();
        }
        int[] length; // 用于更新距离
        MyLinkedList<EdgeNode> tmp;
        EdgeNode edgeNode;

        // 先预设所有节点不相连，访问每一个节点的所有边，若存在边，将距离由无穷大更新为新距离
        for(int i=0; i<vertexNum; i++){
            length=new int[vertexNum];

            // 初始化距离为无穷大
            for(int j=0; j<vertexNum; j++){
                length[j]=INFINITY;
            }
            length[i]=0;
            tmp=vertexes.get(i).getEdge();
            for(int j=0; j<tmp.size(); j++){
                edgeNode=tmp.get(j);
                length[edgeNode.getVertexIndex()]=edgeNode.getDistance();
            }
            res[i+1][0]=vertexes.get(i).getName();
            for(int j=0; j<vertexNum; j++) {
                res[i+1][j+1]=length[j]+"";
            }
        }

        return res;
    }
}

```

通过关键字对景区信息进行查找

输入关键字，若匹配成功，返回相关景点，若无，返回空值

关键代码

```

/**
 * 根据关键字查找对应景点
 */
public String[][] search(String key){

```

```

ScenicSpot tmp;
String[][] res=new String[scenicspots.size()][2];
int j=0;

for(int i=0; i<scenicspots.size(); i++){
    tmp=scenicspots.get(i);
    if(BoyerMoore.match(key,tmp.getName())!=-1){
        res[j][0]=tmp.getName();
        res[j++][1]=tmp.getIntroduction();
    }else if(BoyerMoore.match(key, tmp.getIntroduction())!=-1){
        res[j][0]=tmp.getName();
        res[j++][1]=tmp.getIntroduction();
    }
}
if(j==0){
    return null;
}
else {
    return res;
}
}

```

根据需求对景点进行排序(以按欢迎度排序为例)

返回一个按照相应需求排序的景点数组

关键代码

```

/**
 * 排序功能：按照欢迎度排序
 */
public String[] sortByWelcomed(){
    ScenicSpot[] matrix=new ScenicSpot[vertexes.size()];
    for(int i=0; i<vertexes.size(); i++){
        matrix[i]=vertexes.get(i).getData();
    }
    String[] res=new String[vertexes.size()];
    int num=0;
    QuickSort.quickSort(matrix, 0, vertexes.size()-1, new
PopularityComparator()); //快速排序
    for(int i=0; i<vertexes.size(); i++){
        res[num++]=matrix[i].getName()+" 人气值: "+matrix[i].getPopularity();
    }
    return res;
}
}

```

输出两个景点之间最短路径 (Dijkstra)

关键代码

返回一个包含最短路径和最短路的数组

```

/**
 * 查找两个节点之间的最短路
 */
public String[] findWay(String start, String dest){
    int p1=getPosition(start);
    int p2=getPosition(dest);
    if(p1==-1||p2==-1){
        return null;
    }
}

```

```

        //return Floyd.floyd(this, p1, p2);
        return Dijkstra.dijkstra(this, p1, p2);
    }

```

输出导游路线图

返回一个包含导游图的数组

关键代码

```

public String[] createTourSortGraph(String start, String dest){
    MyHamilton myHamilton=new MyHamilton(start, dest, this);
    String[] res=myHamilton.getHamiltonCircuit();
    //MyHamilton.getHamiltonCircuit(start, dest, this);
    return res;
}

```

停车场部分

汽车进入停车场

输入车牌号，返回车辆进入信息

```

/**
 * 车辆请求进入停车场
 * 分进入停车场和进入便道两种情况处理
 */
public String[] inGarage(String id){
    String[] res=new String[2];
    String time = format.format(date.getTime());

    for(int i=0; i<index; i++){
        if(id.equals(vehicles[i].getId())){
            return null;
        }
    }
    //如果空间充足，进入停车场
    if(checkCapacity()){
        Vehicle vehicle=new Vehicle(time, id);
        garage.push(vehicle);
        res[0]="到达时间: "+time;
        res[1]="停放在停车场: "+garage.size()+"车道";
        vehicles[index++]=vehicle;
    }

    //否则，进入便道
    else{
        Vehicle vehicle=new Vehicle(time, id);
        waitedQueue.push(vehicle);
        res[0]="停车场空间不足";
        res[1]="停放在便道: "+waitedQueue.size()+"车道";
        vehicles[index++]=vehicle;
    }
    return res;
}

```

汽车离开停车场

输入车牌号，返回车辆离场信息

```

/**
 * 汽车离开停车场
 * 离开车辆之后的汽车先进入缓冲区，车辆离开后，其他车辆依次进入
 * 若有等待车辆，则等待车辆依次进入停车场
 */
public String[] outGarage(String id)throws ParseException{
    String time = format.format(date.getTime()); //获取当前时间
    boolean flag=true; //判断车牌信息是否有误
    String[] res=new String[2];
    while(!garage.isEmpty()&&flag){
        flag=false;

        //后面车辆进入缓冲区
        if(!garage.peek().getId().equals(id)){
            buffer.push(garage.pop());
            flag=true;
        }
    }

    //若车牌信息无误，对离开车辆进行停车费结算
    if(!flag){
        Vehicle vehicle=garage.pop();
        String arrTime=vehicle.getArrTime();
        Date d1 = format.parse(arrTime);
        Date d2 = format.parse(time);
        long charge = (d2.getTime() - d1.getTime()) / (60 * 60 * 1000);
        System.out.println(charge);
        int fee=(int)charge*feePerHour;
        vehicle.setLveTime(time);
        res[0]="车牌号: "+vehicle.getId()+"\n"+"停车时间: "+charge+"\n"+"费用: "+fee;
        //vehicles[++index]=vehicle;

        //缓冲区车辆回到车库
        while(!buffer.isEmpty()){
            garage.push(buffer.pop());
        }

        //便道上的车辆一次进入车库
        if(!waitedQueue.isEmpty()){
            vehicle=waitedQueue.pop();
            vehicle.setArrTime(time);
            garage.push(vehicle);
            res[1]="车牌号: "+vehicle.getId()+"\n"+"到达时间: "+time+"\n"+"停放
在停车场: "+garage.size()+"车道";
        }
    }

    //车牌信息有误
    else {
        while(!buffer.isEmpty()){
            garage.push(buffer.pop());
        }
    }
    return res;
}

```

管理员部分

对景点的增加和删除

景点增加

输入景点对象，生成节点并加入节点数组

```
/**
 * 给当前图增添节点
 */
public void addVertex(ScenicSpot data){
    Vertex vertex = new Vertex(data.getName(), data);
    vertexes.add(vertex);
}
```

景点删除

输入景点名称，从节点中删除相关节点

```
/**
 * 删除节点，不仅要删掉节点，还要删掉包含该节点的边
 */
public int removeVex(String name){
    int p1=getPosition(name);
    if(p1!=-1){
        for(int i=0; i<vertexes.size(); i++){
            vertexes.get(i).removeEdge(name);
        }
        vertexes.remove(p1);
        return p1;
    }else{
        System.out.println("顶点"+name+"不存在");
        return -1;
    }
}
```

对边的增加和删除

输入一个包含边的信息的数组，从相关点集中删去该边的信息

```
/**
 * 给当前图删除边
 * 需要在与该边相关的两个节点分别删除对应信息
 */
public void removeEdge(String[] edge){
    int p1=getPosition(edge[0]);
    int p2=getPosition(edge[1]);

    // 需要对两个节点进行操作才能删掉一条边
    if(p1!=-1&& p2!=-1){
        Vertex vertex=vertexes.get(p1);
        vertex.removeEdge(edge[1] );
        vertex=vertexes.get(p2);
        vertex.removeEdge(edge[0]);
    }
}
```

发布公告

输入公告内容，再公告栈中加入该公告

```

/**
 * 发布通告
 * @param broadCast
 */
public void broadcast(String broadCast){
    String time = format.format(date.getTime());
    String[] res={broadCast, time};
    broadCasts.push(res);
}

```

3.5 算法复杂度分析

最后，就本次项目的各种底层算法复杂度进行分析。

Boyer Moore 算法

算法分为两个阶段：预处理阶段和搜索阶段

假设文本串 `text` 长度为 `n`，模式串 `pattern` 长度为 `m`，则预处理阶段(产生坏字符和好后缀表)的时间和空间复杂度都是 $O(m+)$ ，搜索阶段的时间复杂度为 $O(mn)$ ，最坏情况下(输入字符串为非周期的，需要 $3n$ 次字符比较操作，最好情况(在文本串 `b` 中搜索模式串 `ab`)时间复杂度为 $O(n/m)$

Dijkstra 算法

时间复杂度: $O(n^2)$

空间复杂度 $O(n)$

Floyd 算法

时间复杂度: $O(n^3)$

空间复杂度 $O(n^2)$

快速排序

时间复杂度: 平均时间复杂度 $O(n\log(n))$; 最差时间复杂度: $O(n^2)$

空间复杂度: $O(n)$

Hamilton 算法

时间复杂度: $O(n^2)$

空间复杂度: $O(n)$

第四章 系统测试

系统的测试采取自底向上的测试方法，先测试底层开发算法，确定无误后再测试功能是否能正确运行。在程序基本功能检验完成后，笔者还针对特殊情况，如添加重复的景点数据，进行测试，提升系统的鲁棒性。

下面分模块给出系统功能和不同算法的测试结果

4.1 底层算法部分

4.1.1 Boyer Moore 算法测试

测试代码

```
public class Test {  
    public static void main(String[] args) throws Exception {  
        String a="北门 Example";  
        String b="南门";  
        String c="amp";  
        System.out.println(BoyerMoore.match(b, a));  
        System.out.println(BoyerMoore.match(c, a));  
    }  
}
```

测试结果

-1

4

Process finished with exit code 0

算法正确

4.1.2 Dijkstra 算法测试

测试代码

```
public class Test {  
    public static void main(String[] args) throws Exception {  
        ScenicMangement sm=new ScenicMangement();  
        System.out.println(Arrays.toString(Dijkstra.dijkstra(sm.getMap(), 0,  
11)));  
    }  
}
```

测试结果

[北门——仙云石, 仙云石——九曲桥, 九曲桥——朝日峰, 33]

Process finished with exit code 0

算法正确

4.1.3 Floyd 算法测试

测试代码

```
public class Test {  
    public static void main(String[] args) throws Exception {  
        ScenicMangement sm=new ScenicMangement();  
        System.out.println(Arrays.toString(Floyd.floyd(sm.getMap(), 0, 11)));  
    }  
}
```

测试结果

[北门-> 仙云石-> 九曲桥-> 朝日峰, 33]

Process finished with exit code 0

算法正确

4.1.4 Hamilton 算法测试

测试代码

```
public class Test {  
    public static void main(String[] args) throws Exception {  
        ScenicMangement sm=new ScenicMangement();  
        MyHamilton myHamilton=new MyHamilton("北门", "朝日峰", sm.getMap());  
        System.out.println(myHamilton.getHamiltonCircuit()[0]);  
    }  
}
```

测试结果

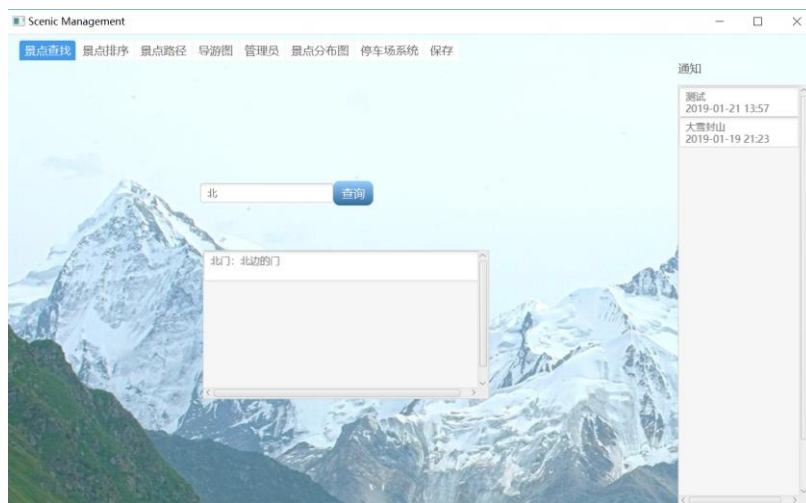
北门——>仙云石——>九曲桥——>仙武湖——>碧水潭——>观云台——>飞流瀑——>狮子山——>一线天——>花卉园——>红叶亭——>朝日峰

Process finished with exit code 0

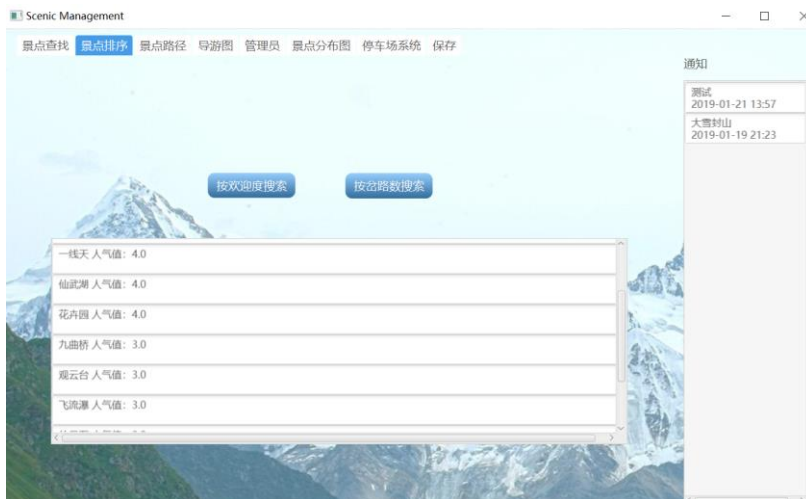
结果正确

4.2 顶层功能测试

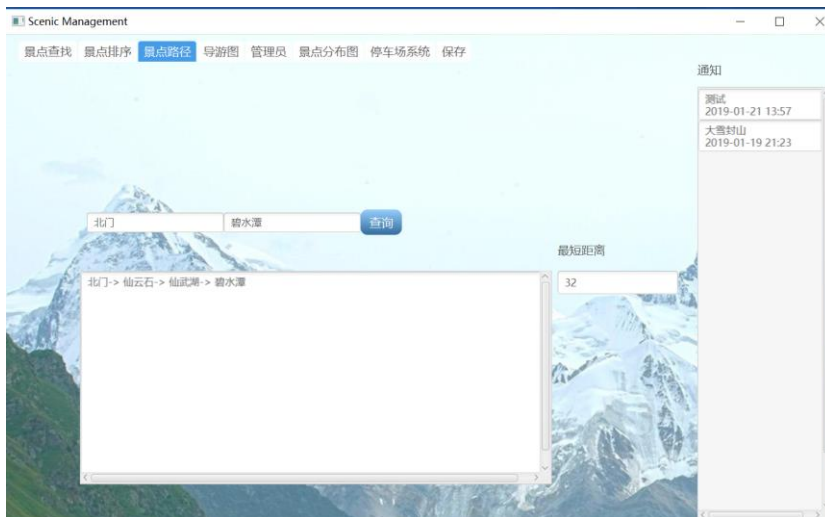
4.2.1 景点查找



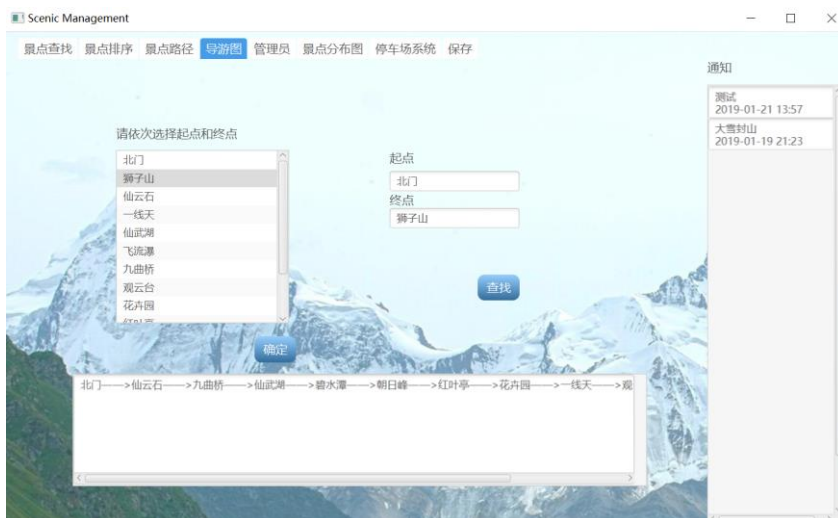
4.2.2 景点排序(以按欢迎度排序为例)



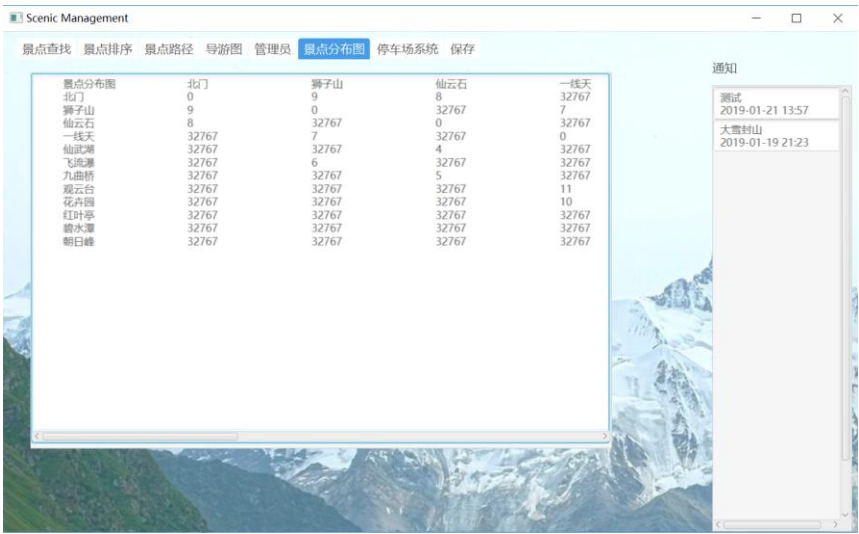
4.2.3 景点间最短路径获取



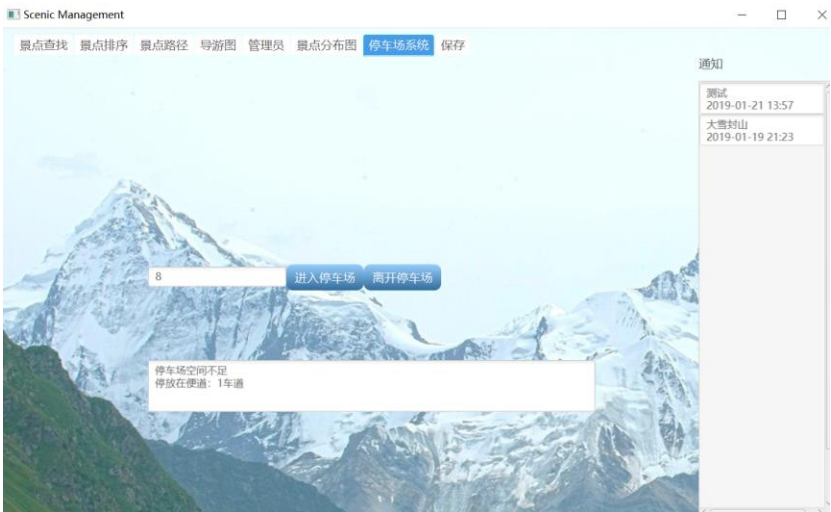
4.2.4 获取导游图



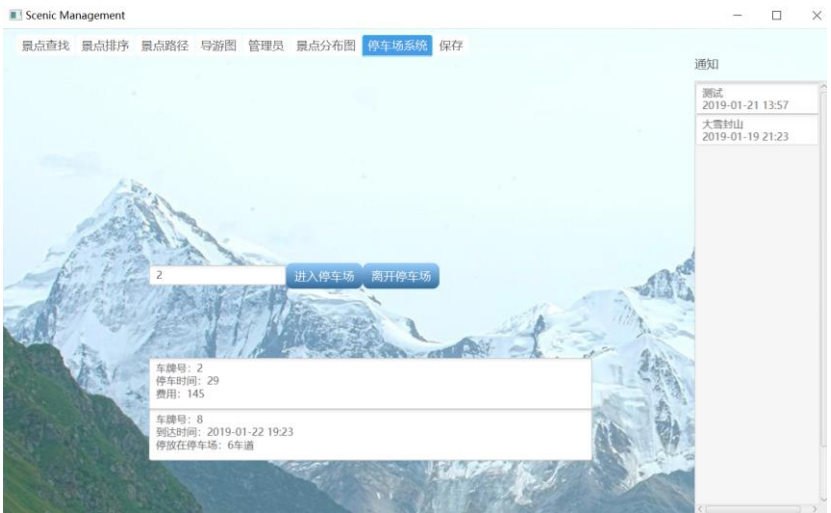
4.2.5 获取景点分布图(打印邻接矩阵)



4.2.6 汽车进入停车场(空间不足停放在便道)

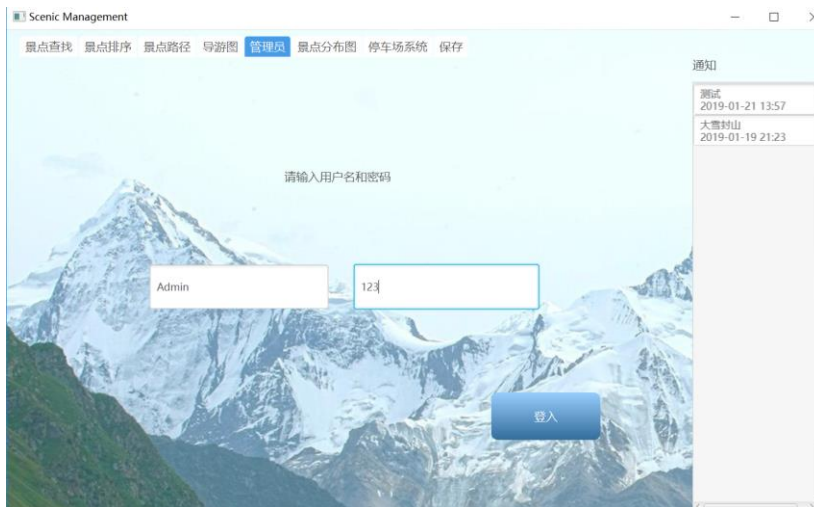


4.2.7 弃车离开停车场(便道中等待车辆自动进入)

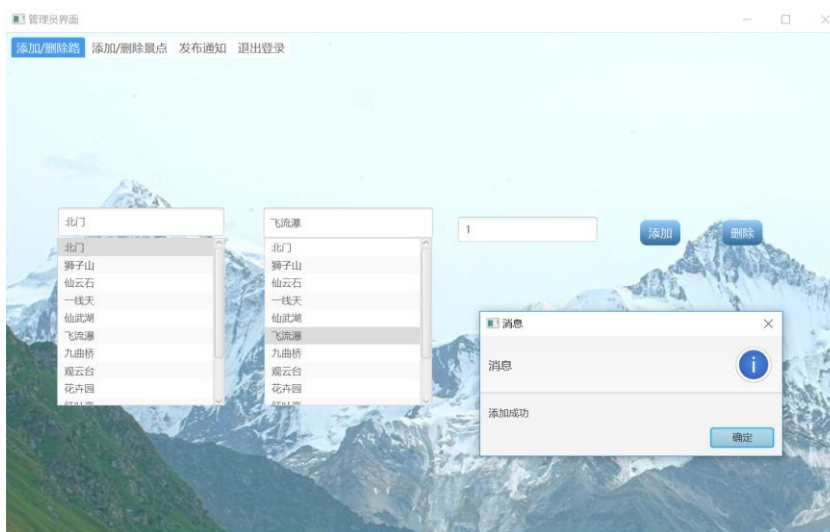


管理员部分

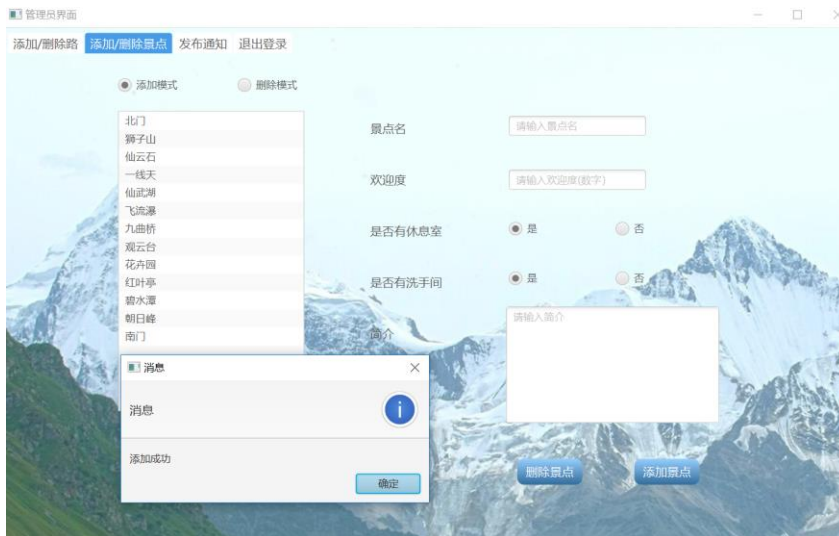
4.3.1 登陆界面



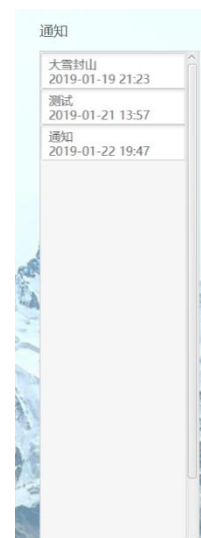
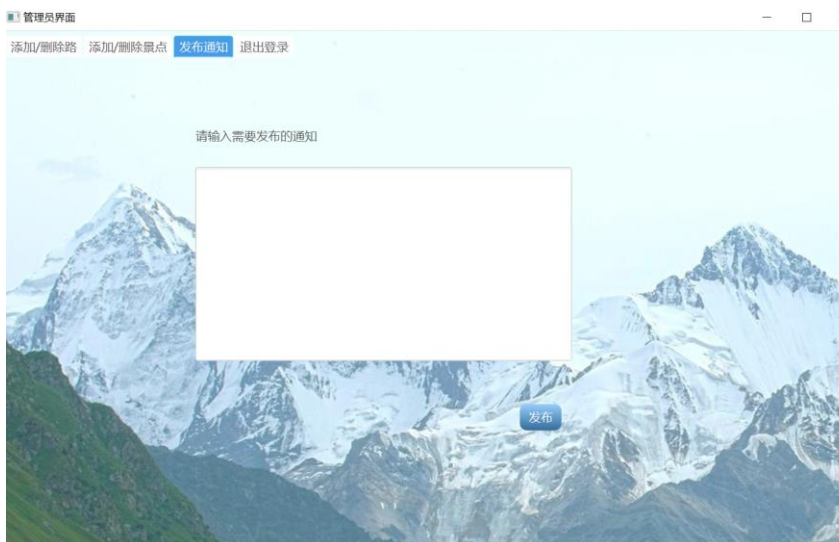
4.3.2 添加/删除路径



4.3.3 添加/删除景点



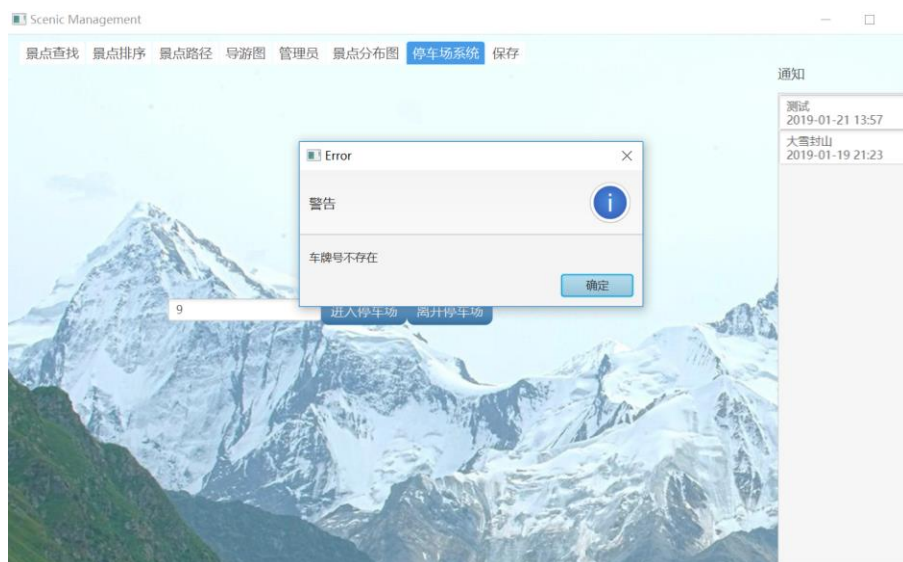
4.3.4 发布公告



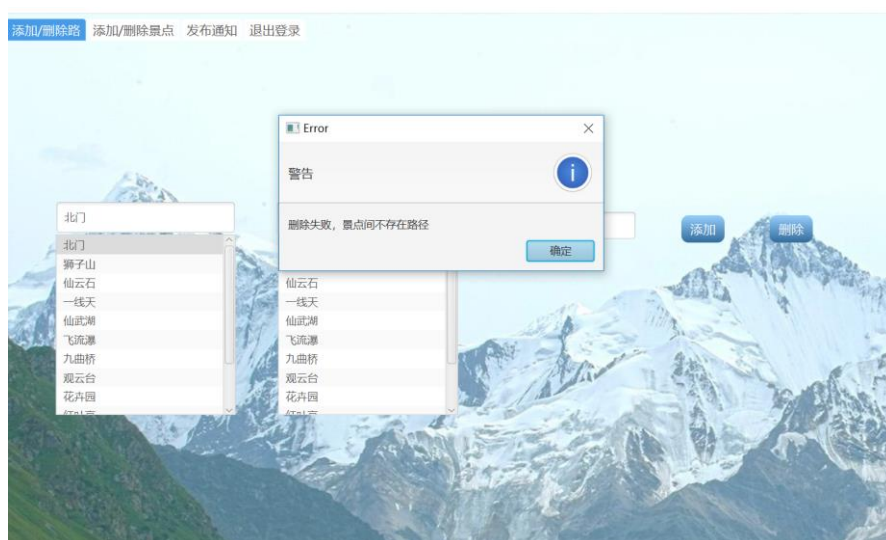
4.4 异常处理

可能的异常情况包括管理员添加重复景点/边；停车场模块中有重复车牌号的车进入车库或者试图将一个不存在车库中的车开出车库，以下是针对这些异常的处理。

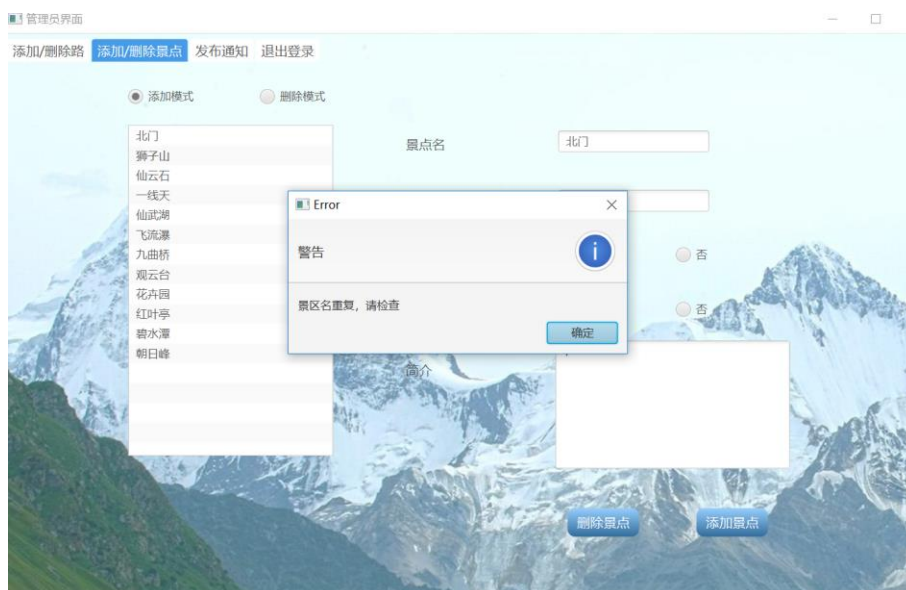
4.4.1 停车系统异常



4.4.2 管理员系统异常(删除不存在的路径)



4.4.3 管理员系统异常(添加重复景点)



第五章 结论

本次实验实现了多个数据结构以及算法。总的来看，本系统的算法复杂度适中，设计和使用的数据结构合理。在底层算法方面，笔者使用 Boyer Moore 算法出色的提升了字符串的匹配效率；在创新性方面，运用 JavaFX 实现了可视化的界面设计，界面样式使用 css 来设置，对系统用户更为友好。除此之外，系统实现了操作方法、数据存储与界面的分离，满足了 MVC 设计模式。最后，系统在实现了任务书所提出的各种需求的同时，通过不断完善细节和异常处理使得系统有着较高的鲁棒性，进一步的完善了它的功能。

参考文献

- [1] 阮一峰.字符串匹配的 Boyer-Moore 算法[OL]
http://www.ruanyifeng.com/blog/2013/05/boyer-moore_string_search_algorithm.html
- [2] 数据结构 Java 实现[M]，机械工业出版社

附录:

《数据结构与算法课程设计》实验成绩评定表



评价内容	具 体 要 求	分值	得分
学习能力	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	
分析和设计的能力	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	
解决问题的能力	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			

