## 项目研究背景与意义

**1. 问题背景**

在之前的课程学习之中，学习到了如何利用栈、DFS、BFS算法来遍历一个迷宫，从而找到迷宫的出口。故而我们联想到了迷宫的生成问题，且生成的迷宫从入口到出口只有单一的路径。根据数据结构的课程知识，我们想到可以用以下方法来生成迷宫：1. 递归回溯（即DFS应用）2. 集合划分法（类似于prim & Kruskal 算法）3. 递归分割算法(块状分割)

经过查阅相关资料，我们了解到还有两种不错的迷宫生成算法，即：Aldous-Broder算法和 Eller算法。Aldous-Broder算法实现原理即为深度优先搜索，而Eller算法是目前了解到的最快且没有明显瑕疵的迷宫生成算法。

**2. 应用意义**

迷宫的产生来源于希腊神话，迷宫现广泛运用于电子游戏领域以及各种益智游戏中。此外，迷宫还可以用来训练小白鼠和简单的人工智能。

结构设计合理的迷宫在一些以解密为主的游戏中尤其重要，迷宫质量直接决定了游戏的整体质量。然而设计一个层次结构合理的迷宫并非是什么易事，特别是当地图的规模十分巨大时，光靠人类来设计出如此复杂的迷宫显然是不可能的，此时便应用到了计算机来随机生成出结构合理且复杂的迷宫。

## 需求分析

2.1 问题描述

构建大小N\*N（N为奇数）的从左上角到右下角的随机迷宫，在这一迷宫上寻找迷宫路径。该设计一共包括如下三个部分：

1. 生成随机迷宫
2. 寻找迷宫路径
3. 迷宫图片导出

2.2 系统目标

随机生成迷宫的算法有很多种，我们挑选了三种最具有代表性的算法进行迷宫的实现以及比较这三种算法的优劣，三种算法分别是：DFS算法、PRIM算法，ELLER算法。

利用java swing构建迷宫程序界面，将三种算法生成的迷宫进行比较，实现迷宫寻路动画、迷宫图片保存、迷宫刷新等功能。

2.3 系统功能要求

1. DFS迷宫生成算法实现
2. PRIM迷宫生成算法实现
3. ELLER迷宫生成算法实现
4. 迷宫寻路算法实现
5. 迷宫寻路动画、程序界面设计（包括界面布局、事件监听等）
6. 迷宫图片保存、迷宫大小可输入调整等附加功能实现

2.4 系统关键技术分析

（1）DFS迷宫生成算法实现

采用栈来存储遍历访问过的cell（格子），类似于DFS算法中的Visit数组。思想就是：每次把新找到的未访问迷宫单元作为优先，寻找其相邻的未访问过的迷宫单元，直到所有的单元都被访问到。Aldous-Broder算法生成的迷宫有一条十分明显的主线路，且迷宫整体十分的扭曲。

（2）PRIM迷宫生成算法实现

利用链表来存储邻墙。算法的本质是prim算法的一种变形，初始迷宫全是墙，随机选一个cell，然后将这个cell的邻墙放入链表中，在表中随机选一堵墙：

1. 如果其对面的cell不是迷宫的通路：把墙打通，即把此墙从链表中删去，并把对面的cell的邻墙纳入链表中。
2. 如果对面的cell已经是迷宫的通路，则直接将此墙从链表中删去因此集合划分法相当于一个随机为路径赋予权值的prim算法。

（3）ELLER迷宫生成算法实现

由于ELLER算法不是本人主要负责，在此便不详细介绍。ELLER算法主要利用集合来进行迷宫生成。

（4）迷宫寻路算法实现

利用深度优先搜索（DFS）算法递归实现，采用栈来存储遍历过的cell(格子)，和DFS迷宫生成算法实现类似。

（5）迷宫寻路动画、程序界面设计（包括界面布局、事件监听等）

迷宫寻路动画调用迷宫寻路函数，利用java swing中paint()方法打印路径并实时刷新。

程序界面设计主要使用JFrame中的JPanel, JButton等进行界面布局，JPanel以网格式布局为主。迷宫刷新、迷宫生成、迷宫导出、迷宫寻路等功能通过点击对应的按钮运行，为这些功能按钮增添了相应的事件监听，事件监听中调用对应的对象或函数。

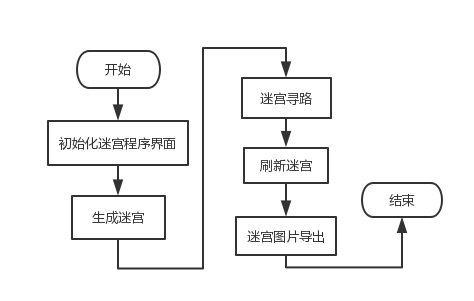
（6）迷宫图片保存、迷宫大小可输入调整等附加功能实现

图片保存功能利用java一个类的Robot进行迷宫所在的Panel进行抓取保存，并以png格式保存至工程目录下。

迷宫大小输入利用正则表达式对输入数值进行范围限制，调用类中的函数进行迷宫大小重新设置，并同时刷新三个迷宫界面。

## 概要设计

3.1 迷宫程序基本工作流程



3.2 迷宫数组的数据结构定义

单个迷宫方块类定义：

public class MazeGrid extends Canvas {

private boolean mark;// 标记是否是通路，TRUE为通路，FALSE为不通   
private boolean isVisited;// 标记是否是访问过的,这是在生成迷宫的时候判断的。  
private boolean isPersonCome;// 标记是否已经走过   
private boolean isStart;// 判断是否为入口   
private boolean isEnd;// 判断是否为出口

MazeGrid(boolean mark, int width, int height);//构造函数

public void paint(Graphics g);//覆写paint方法，包括格子的颜色等信息

boolean noVisited();

void setVisited();

boolean isPersonCome();

void setPersonCome(boolean isPersonCome);

void setStart();

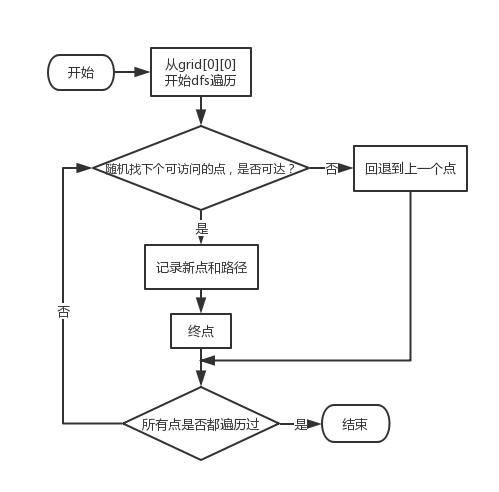
void setEnd();

}

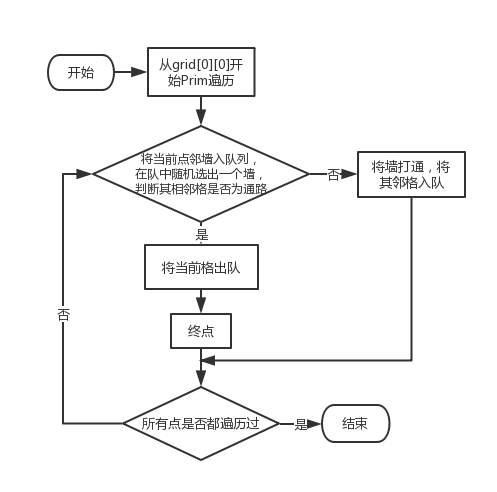
迷宫数组定义：

private MazeGrid dfsGrid[][];//dfs迷宫  
private MazeGrid primGrid[][];//prim迷宫  
private MazeGrid ellerGrid[][];//eller迷宫

3.3 DFS算法流程图



3.4 PRIM算法流程图



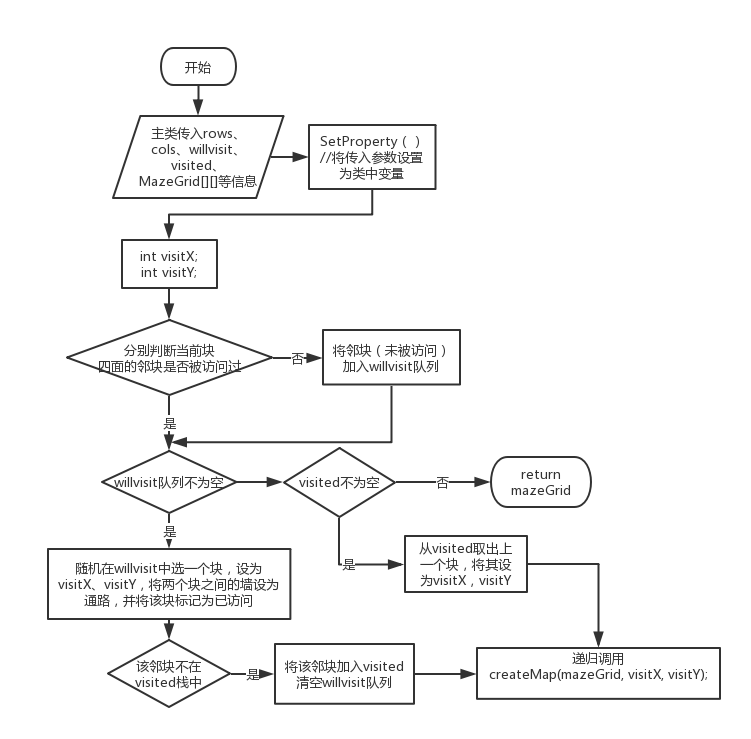
## 详细设计

4.1 DFS迷宫生成算法详细设计

4.1.1 DFS算法所用数据结构、变量

private int rows;// rows 和cols只能是奇数  
private int cols;  
private List<String> willVisit; //保存待访问的当前方格的相邻方格  
private List<String> visited; //保存图中所有已访问过方格  
private MazeGrid grid[][];//迷宫数组

4.1.2 DFS算法详细流程图



4.1.3 DFS算法主要代码

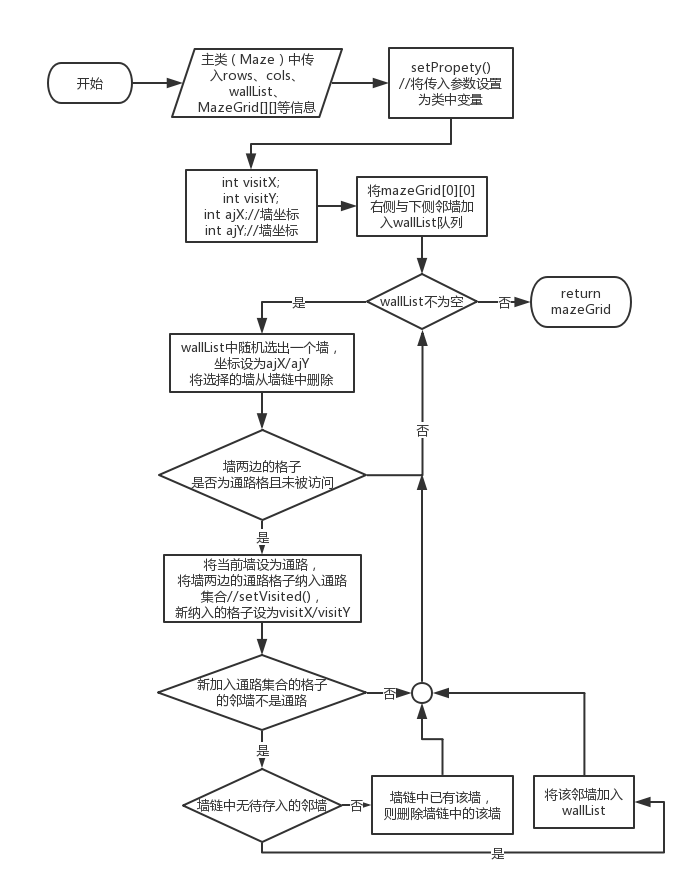
private MazeGrid[][] createMap(MazeGrid mazeGrid[][], int x, int y) {  
 int visitX; int visitY;  
 if (x - 2 >= 0) {  
 if (mazeGrid[x - 2][y].noVisited())//左侧方块未visit  
 willVisit.add((x - 2) + "#" + y); }  
 if (x + 2 < cols) {  
 if (mazeGrid[x + 2][y].noVisited())//右侧方块未visit  
 willVisit.add((x + 2) + "#" + y); }  
 if (y - 2 >= 0) {  
 if (mazeGrid[x][y - 2].noVisited())//下侧方块未visit  
 willVisit.add(x + "#" + (y - 2)); }  
 if (y + 2 < rows) {  
 if (mazeGrid[x][y + 2].noVisited()) //上侧方块未visit  
 willVisit.add(x + "#" + (y + 2)); }  
 if (!willVisit.isEmpty()) { //如果willVisit列表不为空(有邻接点访问)  
 int visit = (int) (Math.random() \* willVisit.size()); //产生0~size的随机数  
 String id = willVisit.get(visit); //根据随机数获取列表中随机的一个相邻块  
 visitX = Integer.parseInt(id.split("#")[0]); //根据"#"分割字符串，将x,y坐标获取  
 visitY = Integer.parseInt(id.split("#")[1]);//将获取的x,y(string类型)转成int类型，并赋给visitX/Y  
 mazeGrid[(visitX + x) / 2][(visitY + y) / 2].setMark(); //将当前块与邻接块中间的块(墙)设为通路  
 mazeGrid[visitX][visitY].setVisited(); //标记该邻接块已visit过  
 if (!visited.contains(id)) // 将这个方块加到已访问中去  
 visited.add(id);  
 willVisit.clear(); //将邻接点列表清空  
 createMap(mazeGrid, visitX, visitY); //递归调用，下次起点从mazeGrid[visitX][visitY]开始  
 } else {  
 if (!visited.isEmpty()) { //无邻接点访问，返回上一个访问过的方块  
 String id = visited.remove(visited.size() - 1);// 取出最后一个元素(退回上一个访问过的方块)  
 visitX = Integer.parseInt(id.split("#")[0]);  
 visitY = Integer.parseInt(id.split("#")[1]);  
 mazeGrid[visitX][visitY].setVisited();  
 createMap(mazeGrid, visitX, visitY); //递归调用，下次起点从退回的(上一个)方块开始  
 }  
 }  
 return mazeGrid;  
}

4.2 PRIM迷宫生成算法详细设计

4.2.1 PRIM算法所用数据结构、变量

private int rows; // rows 和cols只能是奇数  
private int cols;  
private List<String> wallList;//墙链，保存邻接墙块  
private MazeGrid grid[][];//迷宫数组

4.2.2 PRIM算法详细流程图



4.2.2 PRIM算法主要代码

private MazeGrid[][] createMap(MazeGrid[][] mazeGrid) {  
 int visitX; int visitY; int ajX; int ajY;  
 //初始化墙表 将mazeGrid[0][0]的邻墙(右和下)加入墙列表中  
 wallList.add((1) + "#" + 0); wallList.add(0 + "#" + (1));  
 while (!wallList.isEmpty()) { //循环条件：墙链不为空  
 //wallList中随机选出一个墙  
 int visit = (int) (Math.random() \* wallList.size()); //产生0~size的随机数  
 String id = wallList.remove(visit); //将选择的墙从墙链中删除,并返回值给id  
 ajX = Integer.parseInt(id.split("#")[0]); //根据"#"分割字符串，将x,y坐标获取  
 ajY = Integer.parseInt(id.split("#")[1]); //ajX ajY 为墙坐标

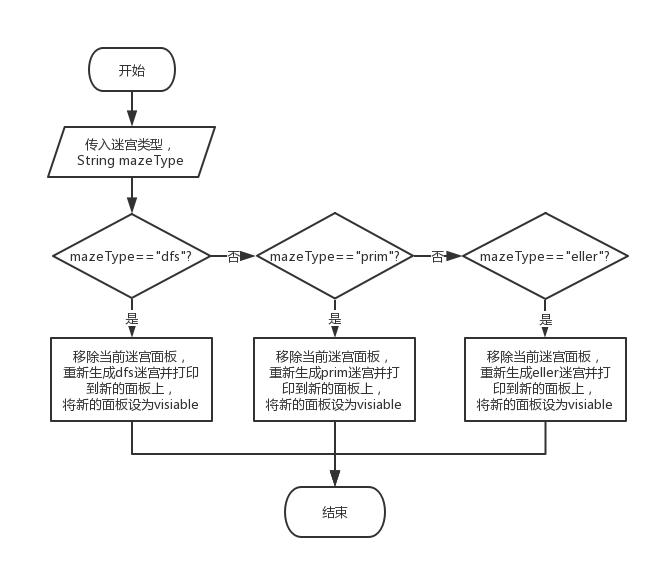
visitX = ajX; visitY = ajY;  
 //将墙两边的格子放入通路集合  
 if (((ajX - 1) >= 0) && mazeGrid[ajX - 1][ajY].isMark() && mazeGrid[ajX - 1][ajY].noVisited()) {  
 mazeGrid[ajX][ajY].setMark();//将当前墙设为通路  
 mazeGrid[ajX - 1][ajY].setVisited();//将墙两边的通路格子纳入通路集合  
 visitX = ajX - 1; //visitX为通路块坐标  
 } else if (((ajX + 1) < cols) && mazeGrid[ajX + 1][ajY].isMark() && mazeGrid[ajX + 1][ajY].noVisited()) {  
 mazeGrid[ajX][ajY].setMark();//将当前墙设为通路  
 mazeGrid[ajX + 1][ajY].setVisited();//将墙两边的通路格子纳入通路集合  
 visitX = ajX + 1;  
 } else if (((ajY - 1) >= 0) && mazeGrid[ajX][ajY - 1].isMark() && mazeGrid[ajX][ajY - 1].noVisited()) {  
 mazeGrid[ajX][ajY].setMark();//将当前墙设为通路  
 mazeGrid[ajX][ajY - 1].setVisited();//将墙两边的通路格子纳入通路集合  
 visitY = ajY - 1;  
 } else if (((ajY + 1) < rows) && mazeGrid[ajX][ajY + 1].isMark() && mazeGrid[ajX][ajY + 1].noVisited()) {  
 mazeGrid[ajX][ajY].setMark();//将当前墙设为通路  
 mazeGrid[ajX][ajY + 1].setVisited();//将墙两边的通路格子纳入通路集合  
 visitY = ajY + 1;  
 }  
 //将新加入通路集合的格子的邻墙加入墙链中  
 String isnWallList; //保存邻墙的坐标 int findId;   
 //条件：邻墙不是通路  
 if (((visitX - 1) >= 0) && !mazeGrid[visitX - 1][visitY].isMark()) {  
 isnWallList = (visitX - 1) + "#" + visitY;  
 if (!wallList.contains(isnWallList)) { //如果墙链中无待存入的邻墙  
 wallList.add(isnWallList);  
 } else {  
 findId = wallList.lastIndexOf(isnWallList);  
 wallList.remove(findId); //如果墙链中已有该墙，则删除墙链中的墙  
 }  
 }  
 if (((visitX + 1) < cols) && !mazeGrid[visitX + 1][visitY].isMark()) {  
 isnWallList = (visitX + 1) + "#" + visitY;  
 if (!wallList.contains(isnWallList)) { //如果墙链中无待存入的邻墙  
 wallList.add(isnWallList);  
 } else {  
 findId = wallList.lastIndexOf(isnWallList);  
 wallList.remove(findId); //如果墙链中已有该墙，则删除墙链中的墙  
 }  
 }  
  
 if (((visitY - 1) >= 0) && !mazeGrid[visitX][visitY - 1].isMark()) {  
 isnWallList = visitX + "#" + (visitY - 1);  
 if (!wallList.contains(isnWallList)) { //如果墙链中无待存入的邻墙  
 wallList.add(isnWallList);  
 } else {  
 findId = wallList.lastIndexOf(isnWallList);  
 wallList.remove(findId); //如果墙链中已有该墙，则删除墙链中的墙  
 }  
 }  
  
 if (((visitY + 1) < rows) && !mazeGrid[visitX][visitY + 1].isMark()) {  
 isnWallList = visitX + "#" + (visitY + 1);  
 if (!wallList.contains(isnWallList)) { //如果墙链中无待存入的邻墙  
 wallList.add(isnWallList);  
 } else {  
 findId = wallList.lastIndexOf(isnWallList);  
 wallList.remove(findId); //如果墙链中已有该墙，则删除墙链中的墙  
 }  
 }  
 }//while  
 return mazeGrid;  
}

4.3 迷宫刷新函数详细设计

函数头：

void refreshMap(String mazeType)

刷新函数流程图：



## 算法设计与分析

* 1. 迷宫数组的定义

首先是单个迷宫格子的定义

public class MazeGrid extends Canvas {

private boolean mark;// 标记是否是通路，TRUE为通路，FALSE为不通   
private boolean isVisited;// 标记是否是访问过的,这是在生成迷宫的时候判断的。  
private boolean isPersonCome;// 标记是否已经走过   
private boolean isStart;// 判断是否为入口   
private boolean isEnd;// 判断是否为出口

MazeGrid(boolean mark, int width, int height);//构造函数

public void paint(Graphics g);//覆写paint方法，包括格子的颜色等信息

boolean noVisited();

void setVisited();

boolean isPersonCome();

void setPersonCome(boolean isPersonCome);

void setStart();

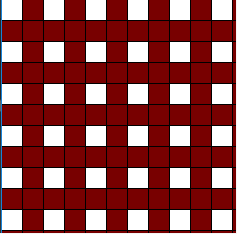
void setEnd();

}

其中mark值的真假代表着格子是墙还是通路，而迷宫由n\*n(n为奇数)个格子构成的数组表示。对应的三个迷宫分别是：

private MazeGrid dfsGrid[][];//dfs迷宫  
private MazeGrid primGrid[][];//prim迷宫  
private MazeGrid ellerGrid[][];//eller迷宫

如下图所示，红色代表墙，白色代表通路



* 1. DFS迷宫生成算法分析

变量定义：

private int rows;// rows 和cols只能是奇数  
private int cols;  
private List<String> willVisit; //保存待访问的当前方格的相邻方格  
private List<String> visited; //保存图中所有已访问过方格  
private MazeGrid grid[][];//迷宫数组

函数定义：

CreateDfsMap()//构造函数

void SetProperty(int rows, int cols, List<String> willVisit, List<String> visited, MazeGrid grid[][])//将Maze主类中的willVisit(队列)、visited(栈)等参数传入并设置变量

MazeGrid[][] generateMap()//调用createMap方法，返回生成的dfs迷宫给Maze类

private MazeGrid[][] createMap(MazeGrid mazeGrid[][], int x, int y)//生成dfs迷宫

* 1. PRIM迷宫生成算法分析

变量定义：

private int rows; // rows 和cols只能是奇数  
private int cols;  
private List<String> wallList;//墙链，保存邻接墙块  
private MazeGrid grid[][];//迷宫数组

函数定义：

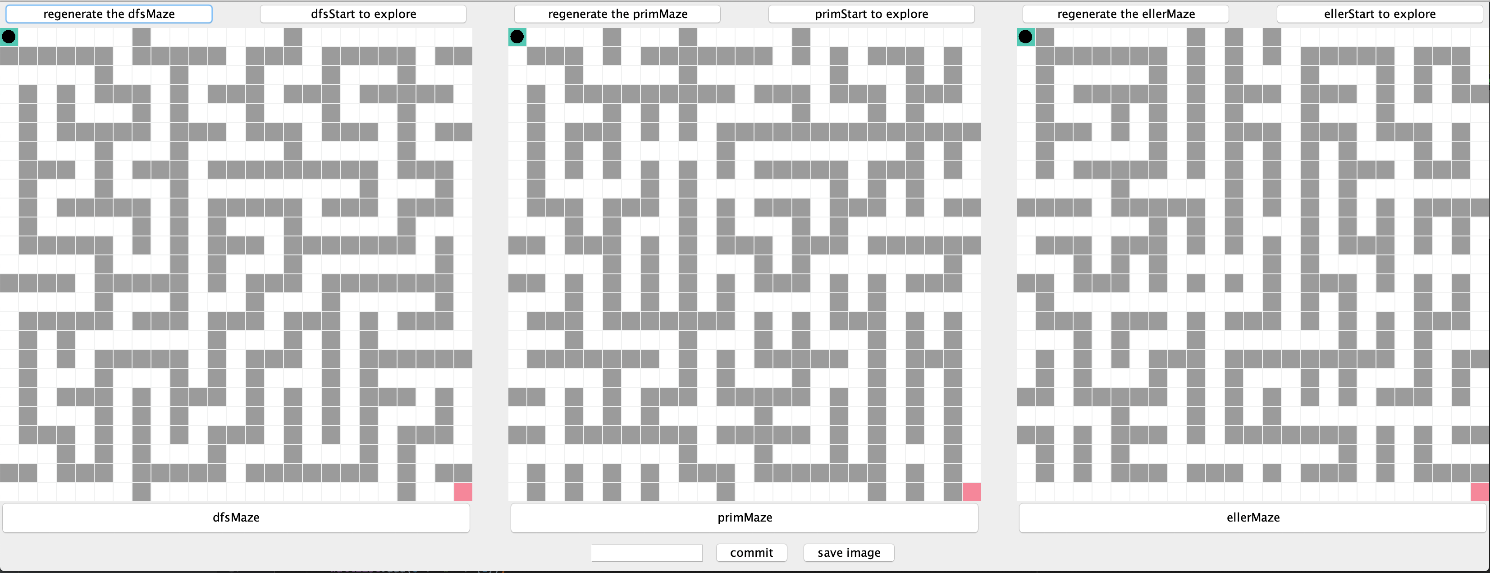
CreatePrimMap()//构造函数

void SetProperty(int rows, int cols, List<String> wallList, MazeGrid grid[][])//将Maze主类中的wallList（墙链）等参数传入并设置变量

MazeGrid[][] generateMap()//调用createMap方法，返回生成的prim迷宫给Maze类

private MazeGrid[][] createMap(MazeGrid[][] mazeGrid) //生成prim迷宫

## 调试分析

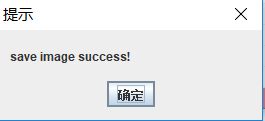


（图为25\*25大小迷宫）

调试：

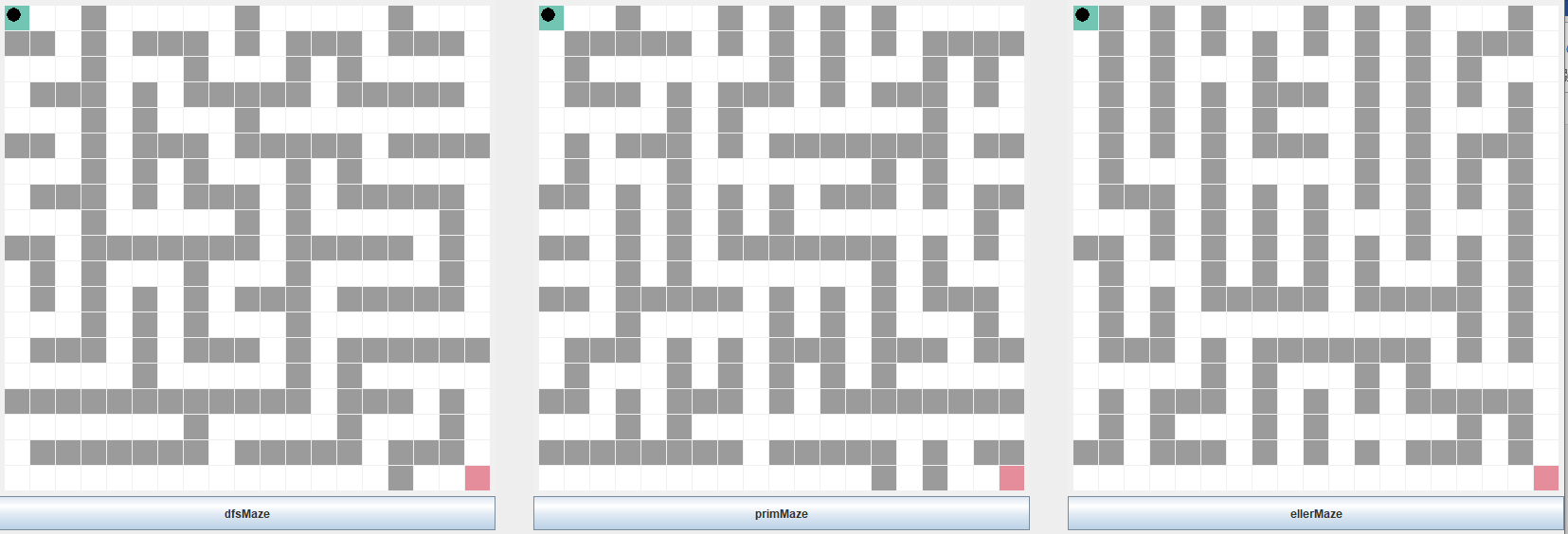
运行程序：迷宫生成时间：3980ms

1. 点击dfsStart to explore开始寻路 寻路时间：10s (10344ms)
2. 点击primStart to explore开始寻路 寻路时间：9s (9746ms)
3. 点击ellerStart to explore开始寻路 寻路时间：19s (19762ms)
4. 点击regenerate the dfsMaze 刷新时间：1059ms
5. 点击regenerate the primMaze 刷新时间：936ms
6. 点击regenerate the ellerMaze 刷新时间：961ms
7. 点击save image 弹出提示框 图片成功保存至工程目录下

(8)在输入框中分别输入-13 与 77& 弹出提示框

(9) 在输入框中输入19，点击commit, 程序刷新迷宫为19\*19的大小迷宫



反馈信息：三个迷宫生成时间：2343ms

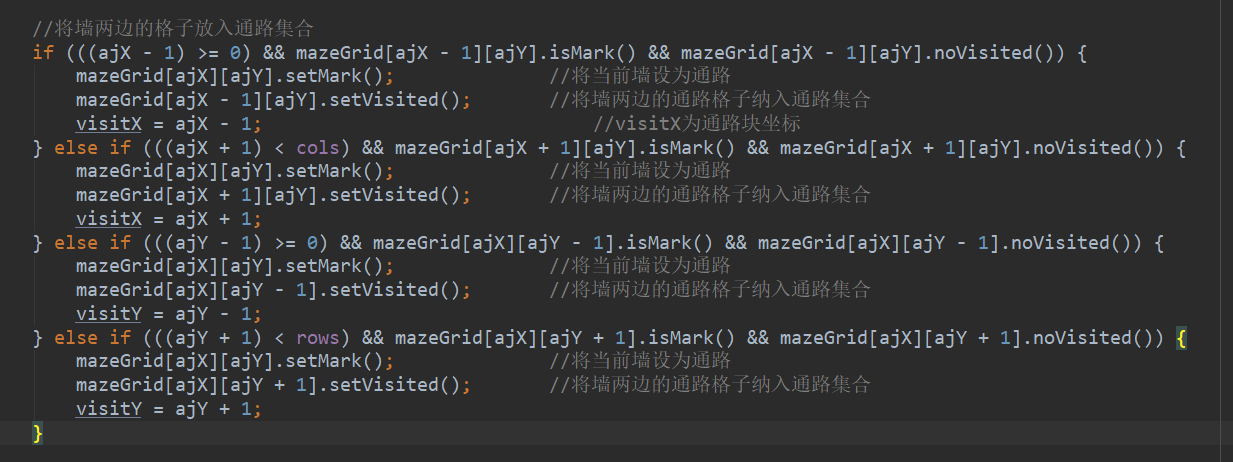
迷宫生成分析：

根据调试得出的数据结果，可以看出，在迷宫大小不大的情况下，PRIM和ELLER迷宫生成算法所需时间接近，而DFS迷宫生成算法时间明显偏长一些，根据源码分析，DFS算法虽然判断较少，但由于调用了递归，因此时间复杂度稍大于

O（n2）,而PRIM算法由于采用的是数组的形式进行存储，时间复杂度约为O(n2)。

算法改进：

PRIM算法改进：



在以上判断结束后，可以直接结束当前轮while循环，直接进入下一轮，省去了下面的邻墙是否在墙链中的判断，可以小幅度的提高算法效率。

此外，由于邻墙存于墙链中，在加入新墙时需要用contains()和lastIndexOf()方法进行查找与提取，如果改为哈希存储的方式，可以直接根据给定的坐标查找并取出重复的墙块，由此可以进一步提高查找和删除效率。

## 运行结果与分析

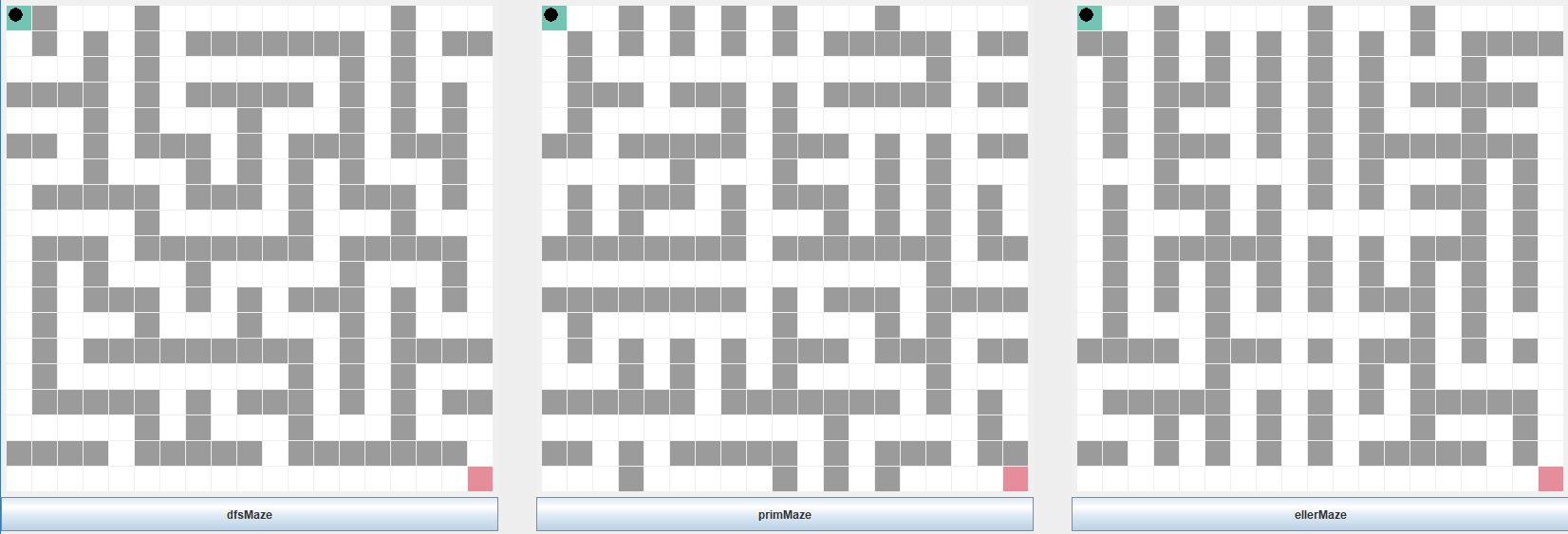
迷宫大小：19\*19

三个迷宫生成时间：2139ms

DFS：7s(7709ms) 刷新: 398ms

PRIM: 5s(5907ms) 刷新: 394ms

ELLER: 13s(13303ms) 刷新: 398ms



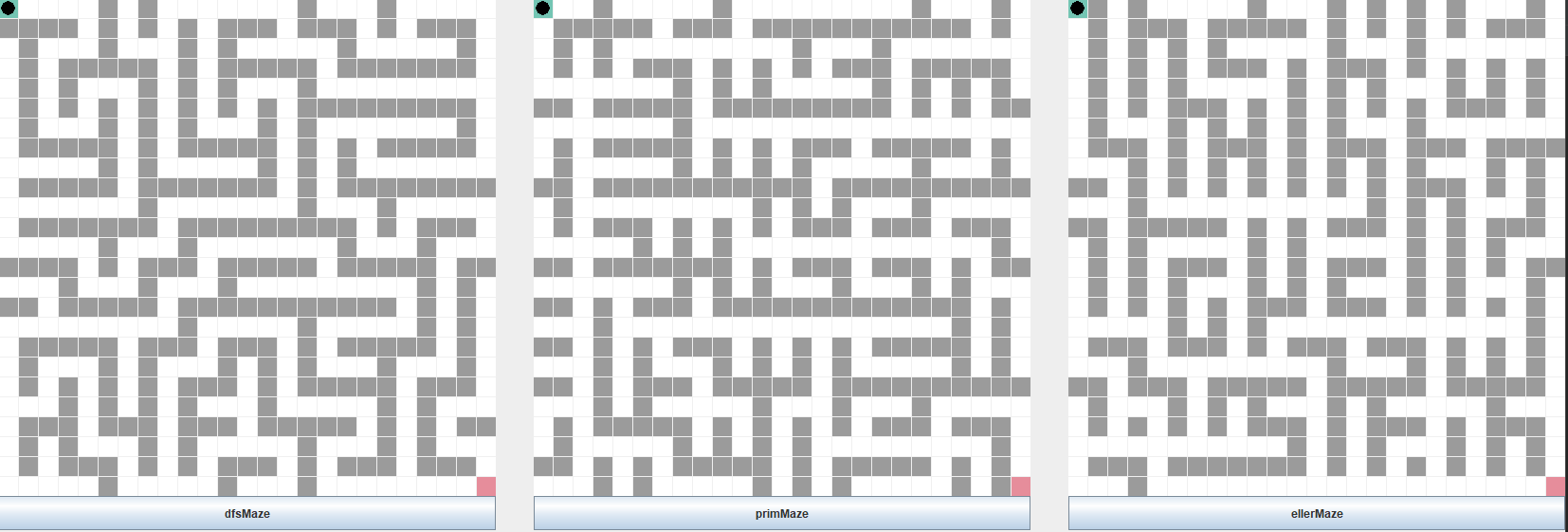
迷宫大小：25\*25

三个迷宫生成时间：4190ms

DFS: 10s(10104ms) 刷新: 1059ms

PRIM: 5s(5710ms) 刷新: 936ms

ELLER: 22s(22606ms) 刷新: 961ms



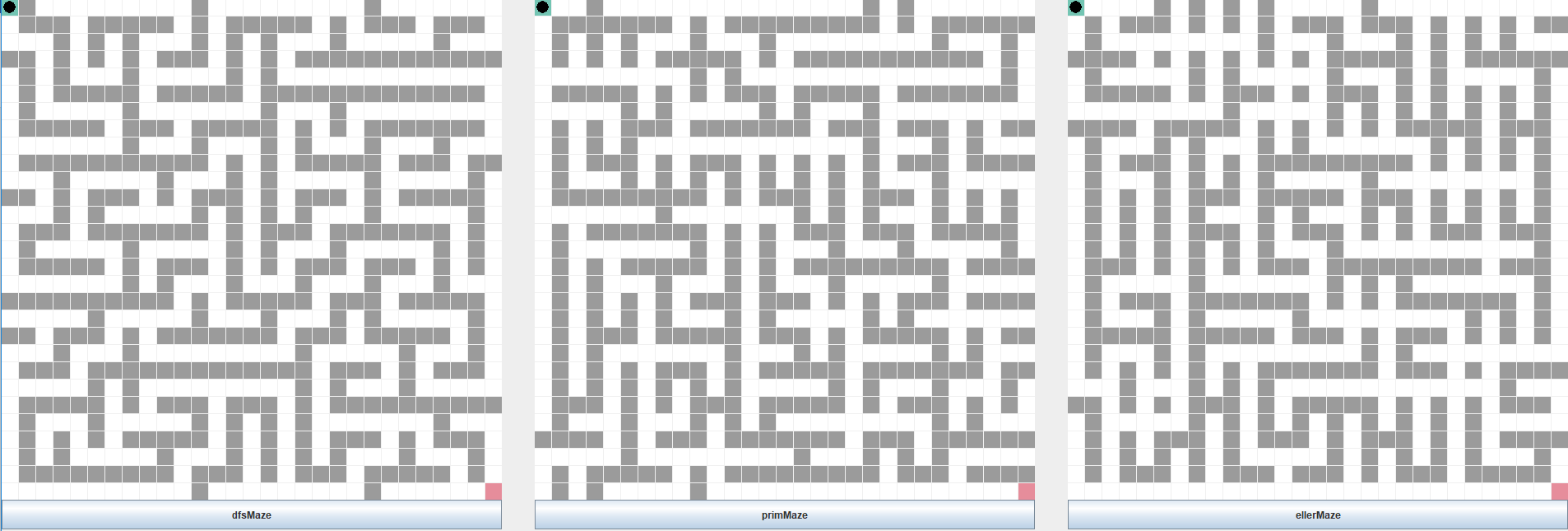
迷宫大小：29\*29

三个迷宫生成时间：8951ms

DFS：21s(21705ms) 刷新: 1770ms

PRIM: 29s(29305ms) 刷新: 1787ms

ELLER:33s(33909ms) 刷新: 1696ms



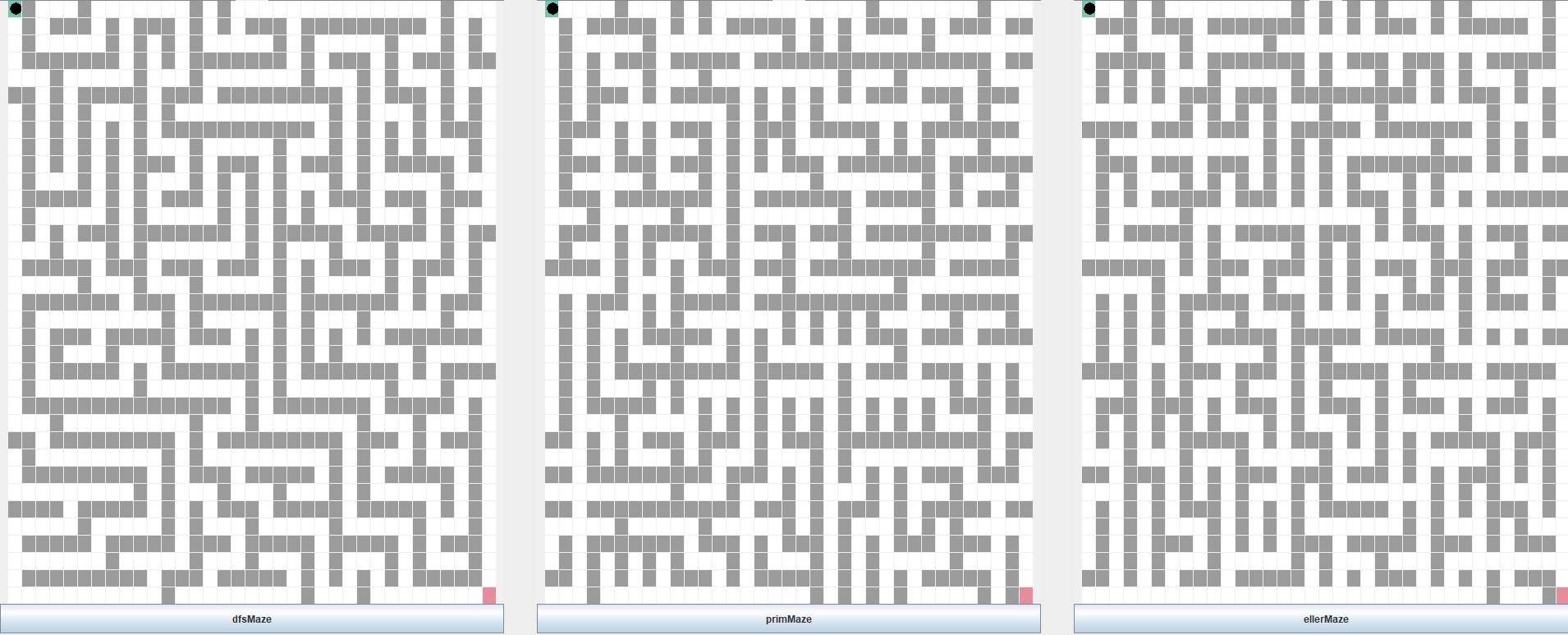
迷宫大小：35\*35

三个迷宫生成时间：20987ms

DFS：31s(31908ms) 刷新: 3225ms

PRIM: 33s(33706ms) 刷新: 3776ms

ELLER: 31s(31702ms) 刷新: 3541ms



## 问题难点

问题和难点主要在于PRIM生成迷宫算法实现以及如何打印迷宫并实现迷宫寻路的刷新动画。

PRIM算法难点分析：

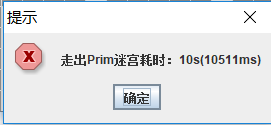
相比于DFS生成迷宫算法，PRIM算法的判断条件更复杂，首先是邻接墙块的存储，采取链表的形式进行存储。其次是如何求得取出的墙块两侧的格子并对其进行是否在同一个集合中的判断，这里利用格子数据结构中的isvisited()和ismark()进行区别判断。如果两边的格子是联通(在isvisited集合中)时（直接在wallList中删除（利用visitX+x/2找到在wallList中的坐标），不用改变mark值）。

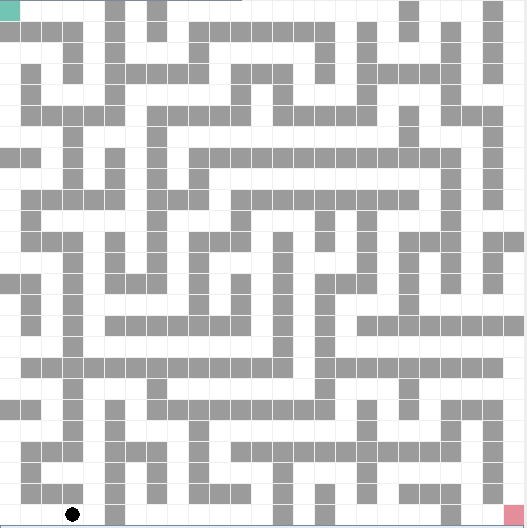
最重要的步骤在于，每次将新的邻墙加入墙链中时要对其进行判断，如果墙链中已经有此墙，则将其从墙链中删除（ps:一面墙最多只有可能被两个格子共有，故无需考虑下次该墙块还会被加进来）。

打印迷宫与寻路刷新动画难点分析：

打印迷宫依靠MazeGrid格子类中定义的isMark，如果isMark为false则为墙，对应灰色块，若为true则为通路，对应白色块。同时覆写MazeGrid中的paint方法（MazeGrid继承Canvas类）。之后在InitDfsPanel和InitPrimPanel类中调用repaint方法，将MazeGrid数组对应的DFS迷宫和PRIM迷宫循环打印出来。

寻路刷新动画调用goMaze()寻路方法，并用repaint方法将其每走一步刷新一次界面，形成动态寻路效果。在事件监听中调用，并以窗口弹出形式反馈回寻路所花的时间信息。





## 主要代码

CreateDfsMap类

import java.util.List;  
class CreateDfsMap {  
 private int rows;// rows 和cols只能是奇数  
 private int cols;  
 private List<String> willVisit; //保存待访问的当前方格的相邻方格  
 private List<String> visited; //保存图中所有已访问过方格  
 private MazeGrid grid[][];  
 CreateDfsMap() {  
 }  
 void SetProperty(int rows, int cols, List<String> willVisit, List<String> visited, MazeGrid grid[][]) {  
 this.rows = rows;  
 this.cols = cols;  
 this.willVisit = willVisit;  
 this.visited = visited;  
 this.grid = grid;  
 }  
 MazeGrid[][] generateMap() {  
 return createMap(grid, 0, 0);  
 }  
 private MazeGrid[][] createMap(MazeGrid mazeGrid[][], int x, int y) {  
  
 int visitX;  
 int visitY;  
 if (x - 2 >= 0) {  
 if (mazeGrid[x - 2][y].noVisited()) { //左侧方块未visit  
 willVisit.add((x - 2) + "#" + y);  
 }  
 }  
 if (x + 2 < cols) {  
 if (mazeGrid[x + 2][y].noVisited()) { //右侧方块未visit  
 willVisit.add((x + 2) + "#" + y);  
 }  
 }  
 if (y - 2 >= 0) {  
 if (mazeGrid[x][y - 2].noVisited()) { //下侧方块未visit  
 willVisit.add(x + "#" + (y - 2));  
 }  
 }  
 if (y + 2 < rows) {  
 if (mazeGrid[x][y + 2].noVisited()) { //上侧方块未visit  
 willVisit.add(x + "#" + (y + 2));  
 }  
 }  
 if (!willVisit.isEmpty()) { //如果willVisit列表不为空(有邻接点访问)  
 int visit = (int) (Math.random() \* willVisit.size()); //产生0~size的随机数  
 String id = willVisit.get(visit); //根据随机数获取列表中随机的一个相邻块  
 visitX = Integer.parseInt(id.split("#")[0]); //根据"#"分割字符串，将x,y坐标获取  
 visitY = Integer.parseInt(id.split("#")[1]); //将获取的x,y(string类型)转成int类型，并赋给visitX/Y  
 mazeGrid[(visitX + x) / 2][(visitY + y) / 2].setMark(); //将当前块与邻接块中间的块(墙)设为通路  
  
 mazeGrid[visitX][visitY].setVisited(); //标记该邻接块已visit过  
 if (!visited.contains(id)) {// 将这个方块加到已访问中去  
 visited.add(id);  
 }  
 willVisit.clear(); //将邻接点列表清空  
 createMap(mazeGrid, visitX, visitY); //递归调用，下次起点从mazeGrid[visitX][visitY]开始  
 } else {  
 if (!visited.isEmpty()) { //无邻接点访问，返回上一个访问过的方块  
 String id = visited.remove(visited.size() - 1);// 取出最后一个元素(退回上一个访问过的方块)  
 visitX = Integer.parseInt(id.split("#")[0]);  
 visitY = Integer.parseInt(id.split("#")[1]);  
 mazeGrid[visitX][visitY].setVisited();  
 createMap(mazeGrid, visitX, visitY); //递归调用，下次起点从退回的(上一个)方块开始  
 }  
 }  
 return mazeGrid;  
 }  
}

CreatePrimMap类

import java.util.List;  
class CreatePrimMap {  
 private int rows;  
 private int cols;  
 private List<String> wallList;  
 private MazeGrid grid[][];  
 CreatePrimMap() {}  
 void SetProperty(int rows, int cols, List<String> wallList, MazeGrid grid[][]) {  
 this.rows = rows;  
 this.cols = cols;  
 this.wallList = wallList;  
 this.grid = grid;  
 }  
 MazeGrid[][] generateMap() {  
 return createMap(grid);  
 }  
 private MazeGrid[][] createMap(MazeGrid[][] mazeGrid) {  
 int visitX;  
 int visitY;  
 int ajX;  
 int ajY;  
 //初始的mazeGrid[0][0].isvisited()=true  
 //初始化墙表 将mazeGrid[0][0]的邻墙(右和下)加入墙列表中  
 wallList.add((1) + "#" + 0);  
 wallList.add(0 + "#" + (1));  
 //循环条件：墙链不为空  
 while (!wallList.isEmpty()) {  
 //wallList中随机选出一个墙  
 int visit = (int) (Math.random() \* wallList.size()); //产生0~size的随机数  
 String id = wallList.remove(visit); //将选择的墙从墙链中删除,并返回值给id  
 //ajX ajY 为墙坐标  
 ajX = Integer.parseInt(id.split("#")[0]); //根据"#"分割字符串 x,y坐标获取  
 ajY = Integer.parseInt(id.split("#")[1]); //将获取的x,y(string类型)转成int类型，并赋给ajX/ajY  
 visitX = ajX;  
 visitY = ajY;  
 //将墙两边的格子放入通路集合  
 if (((ajX - 1) >= 0) && mazeGrid[ajX - 1][ajY].isMark() && mazeGrid[ajX - 1][ajY].noVisited()) {  
 mazeGrid[ajX][ajY].setMark();//将当前墙设为通路  
 mazeGrid[ajX - 1][ajY].setVisited();//将墙两边的通路格子纳入通路集合  
 visitX = ajX - 1; //visitX为通路块坐标  
 } else if (((ajX + 1) < cols) && mazeGrid[ajX + 1][ajY].isMark() && mazeGrid[ajX + 1][ajY].noVisited()) {  
 mazeGrid[ajX][ajY].setMark(); //将当前墙设为通路  
 mazeGrid[ajX + 1][ajY].setVisited(); //将墙两边的通路格子纳入通路集合  
 visitX = ajX + 1;  
 } else if (((ajY - 1) >= 0) && mazeGrid[ajX][ajY - 1].isMark() && mazeGrid[ajX][ajY - 1].noVisited()) {  
 mazeGrid[ajX][ajY].setMark(); //将当前墙设为通路  
 mazeGrid[ajX][ajY - 1].setVisited(); //将墙两边的通路格子纳入通路集合  
 visitY = ajY - 1;  
 } else if (((ajY + 1) < rows) && mazeGrid[ajX][ajY + 1].isMark() && mazeGrid[ajX][ajY + 1].noVisited()) {  
 mazeGrid[ajX][ajY].setMark(); //将当前墙设为通路  
 mazeGrid[ajX][ajY + 1].setVisited(); //将墙两边的通路格子纳入通路集合  
 visitY = ajY + 1;  
 }  
 //将新加入通路集合的格子的邻墙加入墙链中  
 String isnWallList; //保存邻墙的坐标  
 int findId;   
 //条件：邻墙不是通路  
 if (((visitX - 1) >= 0) && !mazeGrid[visitX - 1][visitY].isMark()) {  
 isnWallList = (visitX - 1) + "#" + visitY;  
 if (!wallList.contains(isnWallList)) { //如果墙链中无待存入的邻墙  
 wallList.add(isnWallList);  
 } else {  
 findId = wallList.lastIndexOf(isnWallList);  
 wallList.remove(findId); //如果墙链中已有该墙，则删除墙链中的墙  
 }  
 }  
 if (((visitX + 1) < cols) && !mazeGrid[visitX + 1][visitY].isMark()) {  
 isnWallList = (visitX + 1) + "#" + visitY;  
 if (!wallList.contains(isnWallList)) { //如果墙链中无待存入的邻墙  
 wallList.add(isnWallList);  
 } else {  
 findId = wallList.lastIndexOf(isnWallList);  
 wallList.remove(findId); //如果墙链中已有该墙，则删除墙链中的墙  
 }  
 }  
 if (((visitY - 1) >= 0) && !mazeGrid[visitX][visitY - 1].isMark()) {  
 isnWallList = visitX + "#" + (visitY - 1);  
 if (!wallList.contains(isnWallList)) { //如果墙链中无待存入的邻墙  
 wallList.add(isnWallList);  
 } else {  
 findId = wallList.lastIndexOf(isnWallList);  
 wallList.remove(findId);//如果墙链中已有该墙则删除墙链中的墙  
 }  
 }  
 if (((visitY + 1) < rows) && !mazeGrid[visitX][visitY + 1].isMark()) {  
 isnWallList = visitX + "#" + (visitY + 1);  
 if (!wallList.contains(isnWallList)) { //如果墙链中无待存入的邻墙  
 wallList.add(isnWallList);  
 } else {  
 findId = wallList.lastIndexOf(isnWallList);  
 wallList.remove(findId); //如果墙链中已有该墙，则删除墙链中的墙  
 }  
 }  
 }//while  
 return mazeGrid;  
 }  
}

MazeGrid类

import java.awt.Graphics;  
import java.awt.Color;  
import java.awt.Canvas;  
public class MazeGrid extends Canvas {  
 private boolean mark;// 标记是否是通路，TRUE为通路，FALSE为不通   
 private boolean isVisited;// 标记是否是访问过的,这是在生成迷宫的时候判断的。  
 private boolean isPersonCome;// 标记是否已经走过   
 private boolean isStart;// 判断是否为入口   
 private boolean isEnd;// 判断是否为出口   
 MazeGrid(boolean mark, int width, int height) { //单个方块图像生成  
 this.mark = mark;  
 this.setSize(width, height);  
 if (mark) {  
 this.setBackground(new Color(240, 240, 240)); //通路(mark=true) ：白色  
 } else {  
 this.setBackground(new Color(240, 240, 240)); //墙壁(mark=false)  
 }  
 }  
 boolean isMark() {  
 return mark;  
 }  
 void setMark() {  
 this.mark = true;  
 }  
 public void paint(Graphics g) {  
 if (this.mark) {  
 if (this.isStart) {  
 this.setBackground(new Color(115, 197, 179));  
 } else if (this.isEnd) {  
 this.setBackground(new Color(230, 141, 155));  
 } else {  
 this.setBackground(new Color(255, 255, 255));  
 }  
 } else {  
 this.setBackground(new Color(155, 155, 155));  
 }  
 if (this.isPersonCome) {  
 g.setColor(Color.BLACK);  
 g.fillOval(2, 2, 15, 15);  
 }  
 }  
  
 boolean noVisited() {  
 return !isVisited;  
 }  
 void setVisited() {  
 this.isVisited = true;  
 }  
 boolean isPersonCome() {  
 return isPersonCome;  
 }  
 void setPersonCome(boolean isPersonCome) {  
 this.isPersonCome = isPersonCome;  
 }  
 void setStart() {  
 this.isStart = true;  
 }  
 void setEnd() {  
 this.isEnd = true;  
 }  
}

Maze类中refresh方法

private void refreshMap(String mazeType) {  
 comeX = 0;  
 comeY = 0;  
 willVisit.clear();  
 comed.clear();  
 switch (mazeType) {  
 case "dfs":  
 centerPanel.setVisible(false);  
 centerPanel.removeAll();  
 centerPanel.setLayout(new GridLayout(1, 3, 40, 1));  
 InitDfsPanel dfsPanel = new InitDfsPanel(rows, cols);  
 dfsCenterPanel = dfsPanel.getMazePanel();  
 this.dfsPanel = dfsPanel;  
 primCenterPanel = this.primPanel.getMazePanel();  
 ellerCenterPanel = this.ellerPanel.getMazePanel();  
 dfsGrid = dfsPanel.getMaze();  
 centerPanel.add(dfsCenterPanel);  
 centerPanel.add(primCenterPanel);  
 centerPanel.add(ellerCenterPanel);  
 centerPanel.setVisible(true);  
 break;  
 case "prim":  
 centerPanel.setVisible(false);  
 centerPanel.removeAll();  
 centerPanel.setLayout(new GridLayout(1, 3, 40, 1));  
 InitPrimPanel primPanel = new InitPrimPanel(rows, cols);  
 dfsCenterPanel = this.dfsPanel.getMazePanel();  
 primCenterPanel = primPanel.getMazePanel();  
 this.primPanel = primPanel;  
 ellerCenterPanel = this.ellerPanel.getMazePanel();  
 primGrid = primPanel.getMaze();  
 centerPanel.add(dfsCenterPanel);  
 centerPanel.add(primCenterPanel);  
 centerPanel.add(ellerCenterPanel);  
 centerPanel.setVisible(true);  
 break;  
 case "eller":  
 centerPanel.setVisible(false);  
 centerPanel.removeAll();  
 centerPanel.setLayout(new GridLayout(1, 3, 40, 1));  
 InitEllerPanel ellerPanel = new InitEllerPanel(rows, cols);  
 ellerCenterPanel = ellerPanel.getMazePanel();  
 dfsCenterPanel = this.dfsPanel.getMazePanel();  
 primCenterPanel = this.primPanel.getMazePanel();  
 this.ellerPanel = ellerPanel;  
 ellerGrid = ellerPanel.getMaze();  
 centerPanel.add(dfsCenterPanel);  
 centerPanel.add(primCenterPanel);  
 centerPanel.add(ellerCenterPanel);  
 centerPanel.setVisible(true);  
 break;  
 }

}