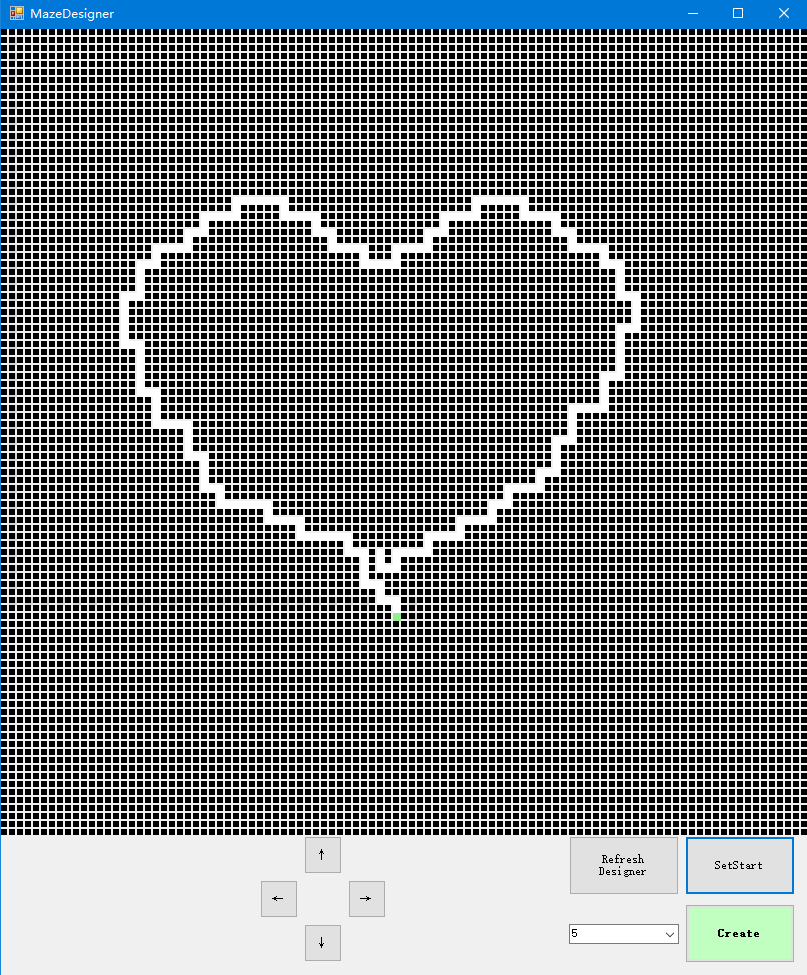
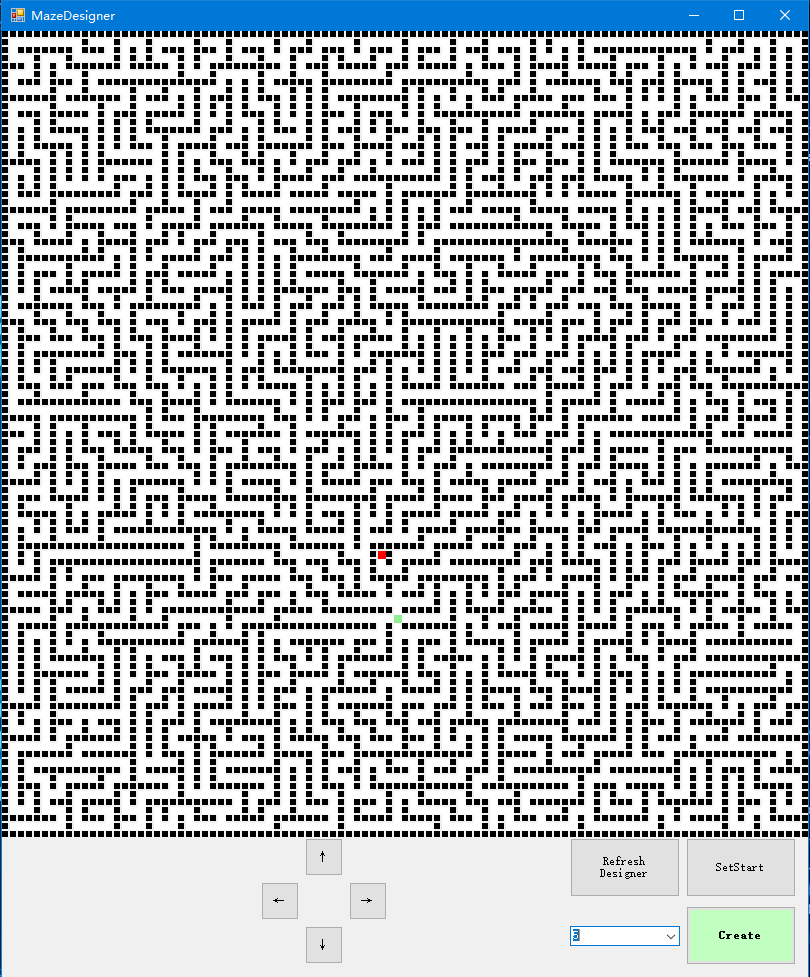
# BFS Maze Designer



计导学习报告——何沁霖

**写在前面：**

直截了当地说，我不擅长写这些报告，我的笔记也原本就是电子版的，非常简洁，只记录了关键字，这样更符合我对于复习的需求，也能更方便地放进我编写的用于复习的小程序中进行复习。之前已经尝试了将这个报告写成对数据结构的报告，但是发现入门困难，而且因为我的学习逻辑是回溯的学习逻辑，即先找到兴趣点，再回溯到我应该学的地方去，所以很难形成完整且有体系的报告，我还想过把之前写过的K-Means，DB-Scan，迷宫，复习用的可交互随机加权抽样器都拿出来写一下，但是发现自己先前写的代码过于幼稚，不论是语法还是思路可读性都较差，再加上我因为没去上机导致要提前交报告，要想统统说一遍实在有些困难。

上机当天我在寝室开始敲我的BFS(breadth first search)迷宫，因为有DFS迷宫的原形，没多久便改完了，不妨本报告就谈谈这个迷宫，也标志着我对算法的学习的最近的进展。

P.S.所有BFS的代码均为手敲，其实现的具体思路也是我在知道BFS可以产生的效果后所构思的。

P.S.S.建议将翻页空白双击隐藏以获得更好的浏览体验

**好的迷宫与BFS：**

通常来说，我们常接触的迷宫的本质其实就是一个树结构，一个好的迷宫，应当具有相当数目的分支，且这些分支都能让人有“认为这一分支就是通向终点”的感觉，故BFS是生成迷宫的一个极佳的算法。对比DFS(深度优先搜索)与BFS(广度优先搜索)，DFS可以通俗地说成一条路走到底，遇见死路即回头，寻找下一个可以走的地方：这更像是我们走迷宫时的思路；而BFS则是走到岔路就设置一个记录点，从起点逐步辐射出去，每遍历一个节点就不再遍历该节点，逐步辐射到新的节点：我相信没有人像这样走迷宫——因为这更符合生成迷宫的方式——BFS生成的迷宫相对于DFS生成的迷宫更具有迷惑性，如果起点没有被堵住，起点处就会有4条岔路可供选择，相对于DFS一路走到底且岔路短，在实际走BFS迷宫时常会走入岔路，而且难以分辨哪个才是出路。以上的结论是我使用BFS和DFS生成迷宫后观察得出的经验性结论。

**BFS的关键实现要求：**

1. 一个存储迷宫的二维数组
2. 一个存储已遍历节点的队列
3. 回溯到队列储存的节点开始新的生成
4. 生成时随机选择路的行进方向
5. 判定是否生成到了死路

**实现BFS迷宫的具体过程：**

首先，这是我们在实现BFS的过程中需要的全局变量，放入声明的G ( Global )类中：

public static int[,] Maze = new int[101, 101];

// 0 = 墙 ；1 = 路 ，构建 101x101 的迷宫

public static int[,] a\_Wall = new int[2, 5];

//上左下右数墙壁，1是有墙，0是为空；第一行Around（四周），第二行Behind（隔一个格子的四周）；一至四列上左下右，第五列为墙壁的数量的总计

public static int[,] a\_Cal = { { 0, -1, 0, 1 }, { -1, 0, 1, 0 } };

//用于对当前坐标实施上左下右计算，第一行X第二行Y，\*2变为Behind

public static int[,] a\_PtPriority = new int[800, 50];

//记录节点优先级，每个节点占两行；节点X坐标 = 2 \* 优先级 ，X坐标 = 2 \* 优先级 + 1；路的最大长度为400格

public static Point pt\_CurrentPt = new Point(1,1);

//当前被操作的点

public static bool b\_MainRoad = true;

//继续生成迷宫的开关

public static bool b\_Singular, b\_DeadEnd;

//分别是，奇数点开关，死路开关

public static Random ran = new Random();

//用于取随机数

public static int i\_Chosen;

//被选中的方向(0 1 2 3 对应 上左下右)

public static int EndCount = 1, i\_PriorityCount = 0;

//分别是，已经挖出的路的格子数量计数，优先级计数

明确了常被使用的变量意义，也构建了一个迷宫雏形（因为以new新建的数组默认全部为0，故很反常地设计成0是墙，1是路的样子，否则需要先把迷宫全部设为1，而那样做是不必要的且徒增耗时的，因为迷宫数组并不输出，对象不会知道幕后的数据是如何安排的），接下来将考虑如何高效生成好的迷宫

**奇数点可转：**

生成迷宫时若只在奇数点（横纵坐标都是奇数）转弯，而在偶数点直行，生成的迷宫就会非常整洁，也不会需要去可以避免生成两格宽的路或墙壁

以下为实现的具体代码，非常容易：

private void JudgeSingular()

{

if (G.pt\_CurrentPt.X % 2 != 0 && G.pt\_CurrentPt.Y % 2 != 0)

{

G.b\_Singular = true;//若为奇数点将奇数开关设为true

}

}

**环境扫描：**

在生成迷宫时，免不了对正在操作的点的周围环境的扫描，以判定是否走到死路，是否应该避障（不能与其他道路形成交叉，因为我们生成的是以树为基础的迷宫），对此，可以以该点为中心，十字形向四周伸展两格，获取这8个格子的通路情况，这在上文中的全局变量中的 a\_Wall 以及 a\_Cal 均有体现

特例——边角扫描：

当在执行环境扫描时遇到处于迷宫边角的点时，因为数组的长度是固定的，如果仍然执行上述环境扫描，则会在扫描BehindWall的时候报错，大致是：超出数组界限，对此，先要判定坐标执行了 a\_Cal 的运算后是否超出数组界限，再放入 Maze 数组中进行操作

以下为实现的具体代码：

private void SingularScan()//奇数点的环境检测

{

for (int i = 0; i < 4; i++)

{

if (G.pt\_CurrentPt.X + 2 \* G.a\_Cal[0, i] == 101 ||

G.pt\_CurrentPt.X + 2 \* G.a\_Cal[0, i] == -1 ||

G.pt\_CurrentPt.Y + 2 \* G.a\_Cal[1, i] == 101 ||

G.pt\_CurrentPt.Y + 2 \* G.a\_Cal[1, i] == -1)

//防止边角点报错

{

G.a\_Wall[0, i] = 1;

G.a\_Wall[0, 4]++;

//判定为边角时，认定Behind为路

}

else//非边角点则正常处理

{

if (G.Maze[G.pt\_CurrentPt.X + G.a\_Cal[0, i],

G.pt\_CurrentPt.Y + G.a\_Cal[1, i]] == 0)

{

G.a\_Wall[0, i] = 1;

G.a\_Wall[0, 4]++;

}

//标注周围墙体位置，有墙是 1 ，无墙是 0

if (G.Maze[G.pt\_CurrentPt.X + 2 \* G.a\_Cal[0, i],

G.pt\_CurrentPt.Y + 2 \* G.a\_Cal[1, i]] == 0)

{

G.a\_Wall[1, i] = 1;

G.a\_Wall[1, 4]++;

}

//标注墙后情况，有墙是 1 ，无墙是 0

}

}

}

**偶数点直行**

在偶数点的直行只需要使用一半的环境扫描，即只扫描相邻的点，而不需要顾及墙后面的情况，在扫描相邻的4个点后，有一个点必定是空的，因为那是来的方向，故只需在使用 a\_Cal 做坐标计算时把原本的 + 换成 – 就成了来向的反向，即应当直行的方向

以下为实现的具体代码：

private void PluralSolution()//偶数点直行 输出到 G.i\_Chosen

{

for(int i = 0; i < 4; i ++)

{

if(G.a\_Wall[0,i] == 0)

{

G.i\_Chosen = i; //利用i\_Chosen找到来的方向

i = 4;

}

}

G.pt\_CurrentPt.X = G.pt\_CurrentPt.X **-** G.a\_Cal[0, G.i\_Chosen];

G.pt\_CurrentPt.Y = G.pt\_CurrentPt.Y - G.a\_Cal[1, G.i\_Chosen];

//得到来向的反向

}

**死路判定：**

有了环境扫描和奇数点可转的基础，死路判定也有一个简单的方法，只需看 a\_Wall 中第二行（表示Behind）的总计，若 Behind 总计 == 0，则来到了死路。对于这一部分的代码实现，只要有了这个思路，一个判断就可以搞定，重点是在于想到 死路 = 四周都是路

以下为实现的具体代码：

private void JudgeDeadEnd()

{

if(G.a\_Wall[1,4] == 0)

{

G.b\_DeadEnd = true;

}

}

**回溯：**

当来到了死路，必然要开始回溯，回溯的过程即为BFS，是生成迷宫的核心步骤，回溯的方式直接决定了生成的迷宫的性质。在此用一个数组

int[,] a\_PtPriority = new int[800, 50];

来记录点的优先级，给这一数组套上队列的性质，虽然移动数据非常麻烦，但是目前因为只是用来生成不成规模的迷宫，也已经足够了。但可见这是一个非常稀疏的数组，理应有更好的结构来容纳这一数据，

以下为实现的具体代码：

private void Back()//该过程将使CurrentPt变成非死点的最高级母点并更新优先级数组

{

G.i\_PriorityCount = 0; //重置优先级计数

while( G.b\_DeadEnd )

{

G.b\_DeadEnd = false;

G.pt\_CurrentPt.X = G.a\_PtPriority[0, 0];

G.pt\_CurrentPt.Y = G.a\_PtPriority[1, 0];

//将当前被操作的点移到优先级最高的点上

SingularScan();//进行环境扫描

JudgeDeadEnd();//判定该点是否已经不能发展新的分支，即是否为死点

if(G.b\_DeadEnd)//是死点则修改表格并重新Back()

{

if (G.a\_PtPriority[0, 1] != 0)//判定是否还有最高级节点

{

for(int i = 0; i < 50; i ++ )

{

G.a\_PtPriority[0, i] = G.a\_PtPriority[0, i + 1];

G.a\_PtPriority[1, i] = G.a\_PtPriority[1, i + 1];

G.a\_PtPriority[0, i + 1] = 0;

G.a\_PtPriority[1, i + 1] = 0; //有将最高级的一行整体前移

if (G.a\_PtPriority[0, i + 2] == 0)

{

i = 50;

}

//防止做无意义的前移，因为通常不会到达数组界限，故不需要单独判定一遍

}

}

else//最高级节点消耗完毕，整体优先级升高

{

for (int i = 1; i < 400; i++)

{

for (int j = 0; j < 50; j++)

{

G.a\_PtPriority[(i - 1)\*2, j] =G.a\_PtPriority[i\*2, j];

G.a\_PtPriority[(i - 1)\*2+1,j]=G.a\_PtPriority[i\*2+1, j];

G.a\_PtPriority[i \* 2, j] = 0;

G.a\_PtPriority[i \* 2 + 1, j] = 0;

if (G.a\_PtPriority[i \* 2, j + 1] == 0)

{

j = 50;

}

}

if (G.a\_PtPriority[(i+1)\*2,0]==0)

{

i = 400;

}

}

}

}

}//执行完此 while 循环后，当前正在操作的点就变成了：目前可用的最高级节点

}

**判定迷宫是否绘制完成：**

这一步中，首先考虑的是：如何判定迷宫已经绘制完成，还有便是，如何优化这一步骤开始执行的时间——因为并不希望刚开始生成迷宫时就开始判定迷宫是否绘制完成。

首先是如何判定迷宫已经绘制完成：只需遍历迷宫本体（Maze数组），一旦发现存在 2x2的四个相连的墙壁，则可以断定迷宫没有绘制完成，反之则已完成

其次是优化这一步骤开始执行的时间：就算每次遍历中加入一段代码，使其一旦发现迷宫未绘制完成就终止遍历，也产生了不必要的过程，因为完全可以计算出：101x101的迷宫最少要挖出多少格子才能没有格子可挖。显然，这个值是4990，将迷宫化为单链就可以得到了，至此便省去了4990次不必要的不完全遍历

以下为实现的具体代码：

private void JudgeEnd()

{

if(G.EndCount > 4990)

{

G.b\_MainRoad = false;

for (int i = 0; i < 100; i ++ )

{

for(int j = 0; j < 100; j ++ )

{

if(G.Maze[i, j] == 0 && G.Maze[i + 1, j + 1] == 0 &&

G.Maze[i, j + 1] == 0 && G.Maze[i + 1, j] == 0)

{

G.b\_MainRoad = true;

i = 100;

j = 100; //当发现需要继续绘制，及时终止遍历

}

}

}

}

}

**大循环的构建：**

当写完了所有的过程，现在剩下的便是将这些过程串联起来，组成绘制迷宫的大循环，部分过程未在报告中体现，比如所有与Graphics有关的过程，还有那些不太重要或者非常简单的过程，但根据过程的命名就可以大致知道其作用

以下为实现的具体代码：

while (G.b\_MainRoad)

{

JudgeEnd();

if(G.b\_MainRoad)

{

Reset();

JudgeSingular();

if (G.b\_Singular)//奇数点走这里

{

SingularScan();

JudgeDeadEnd();

if (G.b\_DeadEnd || G.i\_PriorityCount == 399)//死路走这里

{

Back();

}

else//非死路走这里

{

CountPriority();

RandomChoose();//随机选择行进方向

MoveToChosen();

DigCurrentPt();//Graphics代码，在界面中挖掉一个黑色方块

}

}

else//偶数点走这里

{

PluralScan();//砍掉一般的SingularScan，省略了扫描墙后的情况

PluralSolution();

DigCurrentPt();//Graphics代码，在界面中挖掉一个黑色方块

}

G.EndCount++;

}

}

============================ **完** ============================