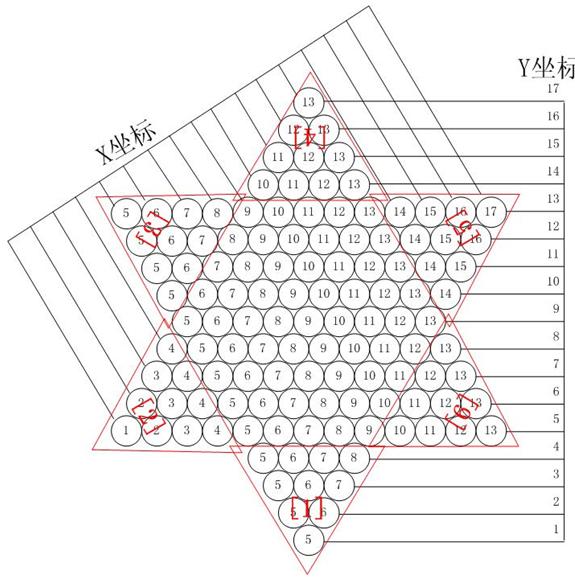
**http://blog.csdn.net/xiejinfeng850414/article/details/8533882**

**四、 跳棋算法:**

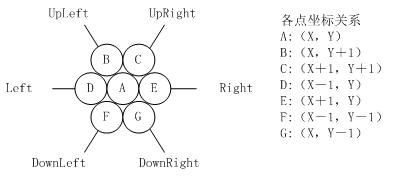
**1. 棋盘坐标定义**

  
坐标定义如图所示，红色数字标识棋盘的区域.

**2. 边界限制**

如图所示，当X坐标为1时，Y的坐标只能为5，当X坐标为2时，Y的坐标可以5或6。于是我们建立一个数组：   
final static private int[][] pos = {  
         {5,5}, //X坐标为1，Y的上限是5，下限是5  
         {5,6}, //X坐标为2，Y的上限是5，下限是6  
         {5,7}, //X坐标为3，Y的上限是5，下限是7  
         {5,8}, //X坐标为4，Y的上限是5，下限是8  
         {1,13}, //X坐标为5，Y的上限是1，下限是13  
         {2,13}, //6  
         {3,13}, //7  
         {4,13}, //8  
         {5,13}, //9  
         {5,14}, //10  
         {5,15}, //11  
         {5,16}, //12  
         {5,17}, //13  
         {10,13}, //14  
         {11,13}, //15  
         {12,13}, //16  
         {13,13}, //17  
};  
在Position类中IsLegalPosition函数可以确定一个坐标是否合法  
public static boolean IsLegalPosition(int x, int y) {  
  if ((x < 1) || (x > 17)) {  
   return false;  
  }  
  if ((y < pos[x - 1][0]) || (y > pos[x - 1][1])) {  
   return false;  
  }  
  return true;    
}

### 4. 下棋规则的算法

λ **坐标的六个方向及相邻坐标的关系**  
在任一个坐标位置上（非边界位置）有相邻六个坐标位置，我们称六个方向分别为：UpLeft、UpRight、Left、Right、DownLeft、DownRight。相邻坐标的关系如下图：  
  
下面的代码可以得到相邻坐标：（在Position类实现的）  
public Position getJoint(Director director) {  
  int x = 0;  
  int y = 0;  
  if (director == Director.UpLeft) {  
   x = this.x;  
   y = this.y + 1;  
  } else if (director == Director.UpRight) {  
   x = this.x + 1;  
   y = this.y + 1;  
  } else if (director == Director.Left) {  
   x = this.x - 1;  
   y = this.y;  
  } else if (director == Director.Right) {  
   x = this.x + 1;  
   y = this.y;  
  } else if (director == Director.DownLeft) {  
   x = this.x - 1;  
   y = this.y - 1;  
  } else if (director == Director.DownRight) {  
   x = this.x;  
   y = this.y - 1;  
  }  
    
  if (IsLegalPosition(x, y)) {//判断坐标是出边界了  
   return new Position(x, y);  
  } else {  
   return null;   
  }    
      
}  
λ **棋子跳动算法**  
该算法在ChessBoard.CanJumpTo函数中实现  
  
 找出一个棋子所有可走位置，并建成数据结构“图”的形式λ  
这个算法有点象图的广度优先遍历算法。文字说明如下：  
1．把当前棋子坐标加入“图”中。  
2．从当前棋子位置出发，从六个方向查找棋子可跳的位置，  
3．如果第2步找到坐标没有加入“图”中，就把这些坐标加入“图”中去。  
4．从第3步刚加入“图”中的坐标出发，继续执行第2步。一直到没有坐标加入“图”中为止。  
λ **最短路径搜索**  
当用户指定下棋位置以后，还要计算出棋子从当前位置到目标位置的最短路径，为游戏的播放棋子走动动画作准备。这个算法还是有点象图的广度优先遍历算法，文字说明如下：  
1．把所有坐标的“权值”设为－1  
2．把当前棋子坐标的“权值”设为0。  
3．从当前棋子位置出发，查找棋子可跳的位置。  
3．如果第3步找到坐标没有“权值”为－1，就把这些坐标的“权值”在上一坐标的基础上加1。  
4．从第4步刚设定“权值”的坐标出发，继续执行第2步。一直到所有坐标都设定“权值”为止。  
5．从目标位置开始，倒退着开始查找，每次查找的位置应该是“权值”比当前位置减小1的坐标。一直找到起始位置。这样就找到最短路径了。  
这些的都是数据结构基础知识，有点抽象，只要理解加想象就没有问题了。哦，上次“平安”考试中有一个题目：什么是前序遍厉，什么是后序遍厉，我真的记不住。数据结构的知识能理解能运用，考试却不能得分，冤呀。

local function checkPosValid(x, y)

if x < 1 or x > 3 or y < 1 or y > 3 then

return 0

end

if grid[x][y] ~= 0 then

return 0

end

if gvalue[x][y] == '#' then

return 0

end

return 1

end

local function find\_first(x, y)

gvalue[x][y] = '#'

if 1 == checkPosValid(x-1,y) then

if touch.x == x-1 and touch.y == y then

gvalue[x-1][y] = '#'

return 1

end

end

if 1 == checkPosValid(x+1,y) then

if touch.x == x+1 and touch.y == y then

gvalue[x+1][y] = '#'

return 1

end

end

if 1 == checkPosValid(x,y-1) then

if touch.x == x and touch.y == y-1 then

gvalue[x][y-1] = '#'

return 1

end

end

if 1 == checkPosValid(x,y+1) then

if touch.x == x and touch.y == y+1 then

gvalue[x][y+1] = '#'

return 1

end

end

gvalue[x][y] = 0

return 0

end

local function find\_path(x, y)

print(x, y)

if 1 == checkPosValid(x,y) then

if touch.x == x and touch.y == y then

gvalue[x][y] = '#'

return 1

end

gvalue[x][y] = '#'

-- right

if y+1 < 3 and grid[x][y+1] > 1 then

if 1 == find\_path(x, y+2) then

return 1

end

end

-- down

if x+1 < 3 and grid[x+1][y] > 1 then

if 1 == find\_path(x+2, y) then

return 1

end

end

-- left

if y-1 > 0 and grid[x][y-1] > 1 then

if 1 == find\_path(x, y-2) then

return 1

end

end

-- up

if x-1 > 0 and grid[x-1][y] > 1 then

if 1 == find\_path(x-2, y) then

return 1

end

end

gvalue[x][y] = 0

return 0

end

return 0

end