禁手判断、评估价值

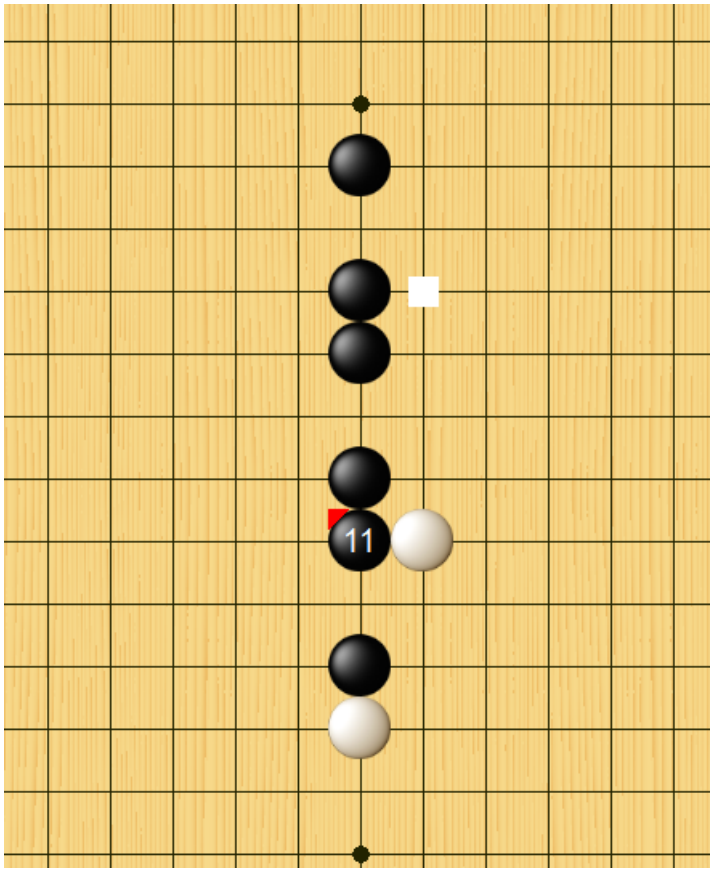
> 为了方便看代码，ban.c里只放了与禁手判断和评估价值有关的函数和内容。
> 去年写的代码太抽象了，屎山我自己看也感觉很痛苦。

(一) 收集判断禁手和价值所需要的信息

【1】收集信息所用数据结构：

```
int Same[8] = {0};
int Same_Empty[8] = {0};
int Same_Empty_Same[8] = {0};
int Same_Empty_Same_Empty[8] = {0};
int Same_Empty_Same_Empty_Same[8] = {0};
```

这里命名确实比较抽象，不过可以结合例子来理解：



如上图(忽略方形白色，呢个是围棋软件的落子提示)

黑11即为即将落下的黑子，其坐标对应参数中的(hang,lie)。

对他而言：(数组下标0-7依次对应8个方位，按顺序为上方，右上，右，右下，下，左下，左，左上)

- 向上方：(下标为0)
  - 它上面只有1颗连着的黑子，因此Same[0] = 1。 ([0]对应“上面”，= 1 对应“有一颗”)
  - 紧挨着连着的黑子的是一个空位，因此Same_Empty[0] = 1。
  - 空位再往上是两颗黑子，因此Same_Empty_Same[0] = 2
  - 两颗黑子再往上是一个空位，因此Same_Empty_Same_Empty[0] = 1。
  - 空位再往上又是一颗黑子，因此Same_Empty_Same_Empty_Same[0] = 1。
- 向右上：(下标为1)
  - 右上没有连着的黑子，因此Same[1] = 0。
  - 右上全是空位，因此Same_Empty[1] >= 5(具体多少得根据边界而定，这里没法给出)
  - 空位直至边界，因此Same_Empty_Same[1] = Same_Empty_Same_Empty[1] = Same_Empty_Same_Empty_Same[1] = 0。
- 向右方：(下标为2)
  - 它的右方既没有连着的黑子也没有空位，因此Same[2] = Same_Empty[2] = Same_Empty_Same[2] = Same_Empty_Same_Empty[2] = Same_Empty_Same_Empty_Same[2] = 0
- 向右下：(下表为3)
  - 与向右上类似，不再赘述。
- 向下方：(下标为4)
  - 下方没有紧挨的黑子，因此Same[4] = 0
  - 下方紧挨着一个空位，因此Same_Empty[4] = 1
  - 空位再往下紧挨着一个黑子，因此Same_Empty_Same[4] = 1
  - 再往下被白棋堵死，因此Same_Empty_Same_Empty[4] = Same_Empty_Same_Empty_Same[4] = 0

- 其他4个方向与以上类似，不再赘述。

【2】如何计算这五个数组(收集信息)

对每个方向，依次使用5个for循环即可完成信息的收集，具体如下：(以向上和向右上为例)

```
    //计算各个方向的信息
    int x,y;
    //向上搜索即y--;
    for(x=lie,y=hang-1;y>=0 && aRecordBoard[y][x]==color;y--,Same[0]++);

    for(;y>=0 && aRecordBoard[y][x]==NONE;y--,Same_Empty[0]++);

    for(;y>=0 && aRecordBoard[y][x]==color;y--,Same_Empty_Same[0]++);

    for(;y>=0 && aRecordBoard[y][x]==NONE;y--,Same_Empty_Same_Empty[0]++);

    for(;y>=0 && aRecordBoard[y][x]==color;y--,Same_Empty_Same_Empty_Same[0]++);

    //向右上搜索即y--,x++;
    for(x=lie+1,y=hang-1;x<SIZE && y>=0 && aRecordBoard[y][x]==color;x++,y-
-,Same[1]++);

    for(;x<SIZE && y>=0 && aRecordBoard[y][x]==NONE;x++,y--,Same_Empty[1]++);

    for(;x<SIZE && y>=0 && aRecordBoard[y][x]==color;x++,y-
-,Same_Empty_Same[1]++);

    for(;x<SIZE && y>=0 && aRecordBoard[y][x]==NONE;x++,y-
-,Same_Empty_Same_Empty[1]++);

    for(;x<SIZE && y>=0 && aRecordBoard[y][x]==color;x++,y-
-,Same_Empty_Same_Empty_Same[1]++);
```

这里的for循环还是比较容易看懂的，不细嗦了。

> 至此我们完成了判断禁手和计算价值所必须的信息的收集，下一步我们将根据这些信息来判断禁手和计算价值。

二 判断禁手和计算价值

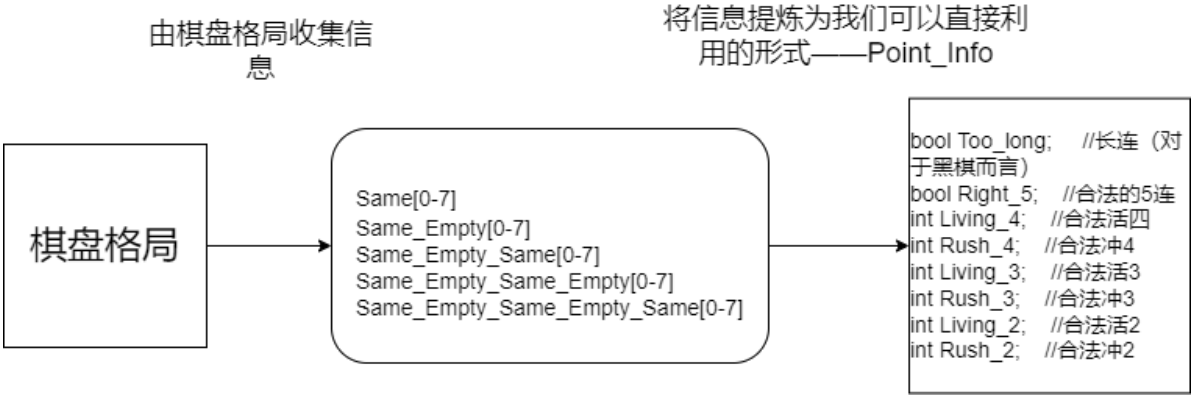
【1】信息的进一步提炼

回顾我们定义的结构体

```
//2:棋子落点处的连子信息(info --> information)
typedef struct Point_Info
{
    /*
```

```
    *这里将长连与合法5连设置为布尔类型的原因是
    *我们并不关心其数量，而是只关心它的真假
    *从上至下一般来说价值是递减的
    *我们根据info可以计算出某点的价值
    */
    bool Too_long;      //长连（对于黑棋而言）
    bool Right_5;       //合法的5连
    int Living_4;       //合法活四
    int Rush_4;         //合法冲4
    int Living_3;       //合法活3
    int Rush_3;         //合法冲3
    int Living_2;       //合法活2
    int Rush_2;         //合法冲2
}Point_Info, *Ptr_to_Point_Info;
```

我们显然没法直接通过5个数组来得出是否有禁手以及每个点的价值，因为这五个数组属于meta data元数据，我们需要首先对这些元数据进行提炼从而得到容易被我们理解和使用的形式——Point_Info.



接下来我们将完成从五个数组到Point_Info的转化：

> 首先我们需要注意到，在判断5连、活4、冲4、活3等时，原本的8个方向变成了4个方向，因为此时的上下、左右、左上右下、右上左下对我们而言分别是一个方向，而对这4个方向上的信息判断过程是完全相同的，因此只要完成其中一个方向的转化，外面加一个for(int i=0; i<4; i++)的for循环即可。

以下按价值从大到小，分别判断各类形状：

注意到各种情况以if else连接，因为在单个方向上，有五连当然没有四连，有四连当然没有三连。

1：五连

首先注意这里说的5连指算上欲落子(hang，lie)刚好在某个方向连成5个子的情况。

这是最好判断的情况：

```
        //判断是否构成不禁手的五连,如果正确的五连实现，那其他的都不用判断了，直接return
        if(Same[i]+Same[i+4] == 4 || (Same[i]==4 && Same[i+4]==4))
        {
            //后者条件是禁手的特殊情况，即上下都是4个，此时下两个4中间不构成长连禁手
            Evaluated_Board[hang][lie].Direction[color-1].Right_5=true;
            return;
        }
```

值得注意的是 i 和 i+4恰好是对称的方向，如上下，左右，这一原则后面将一直用到

还有一种特殊情况，即对称方向上各有4个连子，此时落子后构成9连，但此时黑棋并不构成长连禁手，这个规则非常抽象。

2：长连禁手

首先必须要区分颜色，如果是白棋则无禁手，将长连记为5连即可(等价为0），但如果是黑棋则视为长连禁手：

```
        //如果是白棋，则长连也算赢
        if(Same[i]+Same[i+4]>=5 && color==WHITE){    //只有黑棋有禁手
            Evaluated_Board[hang][lie].Direction[color-1].Right_5=true;
            return;
        }
        //判断长连禁手，如果禁手出现也是其他的都不用判断了
        if(Same[i]+Same[i+4]>=5 && color==BLACK){    //只有黑棋有禁手
            Evaluated_Board[hang][lie].Direction[color-1].Too_long=true;
            return;
        }
```

3：四连

```
        else if(Same[i]+Same[i+4]==3)
        {
            //flag用来记录两边有没有能下的空位
            int flag=0;
            if(Same_Empty[i]>0 && judge_next(hang,lie,Same[i],i,color)){
                flag++;
            }
            if(Same_Empty[i+4]>0 && judge_next(hang,lie,Same[i+4],i+4,color)){
                flag++;
            }

            if(flag==2){    //活四
                Evaluated_Board[hang][lie].Direction[color-1].Living_4 += 1;
            }
            else if(flag==1){       //冲四
                Evaluated_Board[hang][lie].Direction[color-1].Rush_4 += 1;
            }
            else{
                ;       //两边都被堵了或者由于禁手下不了，没用
            }
        }
```

首先大的条件是Same[i]+Same[i+4]==3，这意味着在某个方向上连同欲落子刚好能凑成4个子。

在此基础上：

- 如果4子的两边都有能下的空位，则为活4；
- 若只有一个能下的空位，则为冲4，
- 若边都下不了，则啥也不是(姑且称为死4)
- 值得注意的是，以上三种并不是活四和冲四的全部情况，因为构成活四和冲四并不一定非得直接的连四个

形如 OXOOOXO (X为空位)也可以为活四，但它的基本形状为三连，故我们留到4：三连中进行讨论。

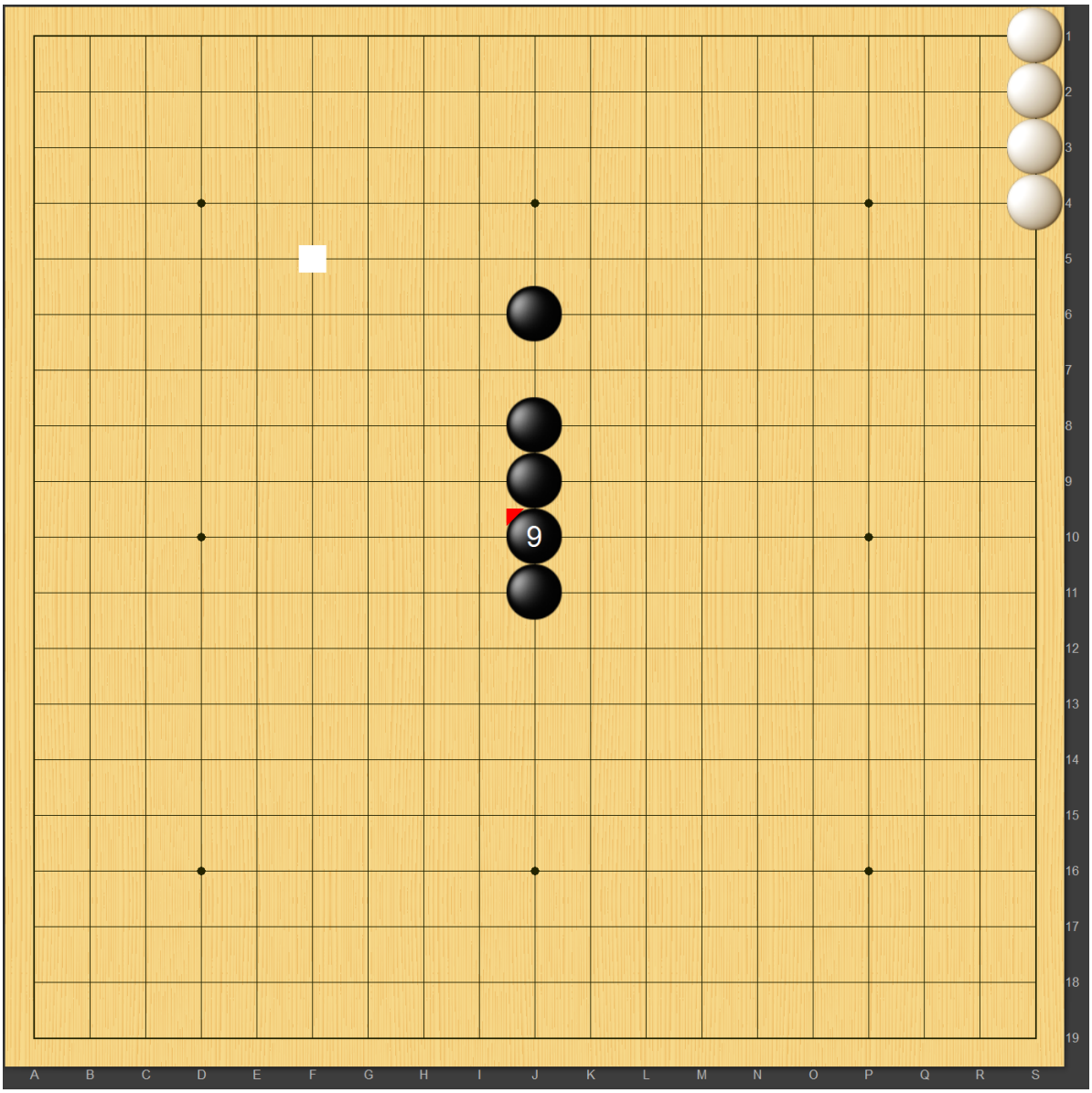
但注意！！！！！！！！！！！

这里“能下的空位”中空位是一回事，而能下又是一回事，因为有些空位可能在未落当前子之前就已经是禁手，而有些空位在未落当前子之前不是禁手，但一旦落了当前子之后就变成了禁手，这两种情况都不能称为能下的空位。而显然某个空位在落子后是否会变成禁手是需要进一步判断的(即需要将当前子落下后才能判断)

而即使落下当前子之后去判断空位是否有禁手时，又会遇到新的空位是否有禁手的情况，这里显然是一个递归的过程。

我们使用了judge_next函数来递归判断某个空位是否会被禁手限制，至于judge_next如何实现，我们暂且搁置，放到后面解决。

但这里先对需要judge_next的情况举个例子以证明它是必要的：



如图假设黑9为欲落子，显然它首先在上下方向构成了四连，其次我们需要判断 (J,7)位是否“是可下的空位”，它当然是空位，且在黑9落子之前并不是禁手，但当你黑9落子之后，它就会变成长连禁手，因此黑9只是冲4而非活四。这样的例子非常多，如果不处理这种情况，想必不会获得很好地棋力。

4：三连

```
else if(Same[i]+Same[i+4]==2){
    //检验上下方有没有可能冲四,两边都是冲四就是活四     形如  “xxxox”
    int flag=0;
    if(Same_Empty[i]==1 && Same_Empty_Same[i]==1)
    {
        if(judge_next(hang,lie,Same[i],i,color))
        {       //检验“xxxox”的o处
           flag++;
        }
    }
    if(Same_Empty[i+4]==1 && Same_Empty_Same[i+4]==1)
    {
        if(judge_next(hang,lie,Same[i+4],i+4,color))
        {      //检验“xoxxx”处的o处
            flag++;
        }
    }

    if(flag==2)
    {
        //形如  "xoxxxox"且两个0处均能落子形成活四
        Evaluated_Board[hang][lie].Direction[color-1].Living_4+=1;
    }
    else if(flag==1)
    {
        //形如 "xxxox"且0处能落子形成冲四
        Evaluated_Board[hang][lie].Direction[color-1].Rush_4+=1;
    }
    else
    {
        //检查上方是否有活三，即下一步落子在上方第一个空位上能否形成活四；如果没有活三，判断是否有冲3
        //例如（以右为上）"ooxxxooo" / "ooxxxoo|" / "|oxxxooo" 下一步落子在右边第一个o处(也是judge_next应传入的参数都            能形成活四，故为活三
        //而形如"xoxxxooo"  / "ooxxxoox" 下一步落子在右边第一个o处时是无法形成活四的（禁手限制）

        //flag1 is for up ; flag2 is for down
        bool flag1_1=(Same_Empty[i]>2 || (Same_Empty[i]==2 && Same_Empty_Same[i]==0))? true : false;
        bool flag1_2=(Same_Empty[i]==1 && Same_Empty_Same[i]==0)? true : false;
        bool flag1_3=(Same_Empty[i+4]>1 || (Same_Empty[i+4]==1 && Same_Empty_Same[i+4]==0))? true : false;

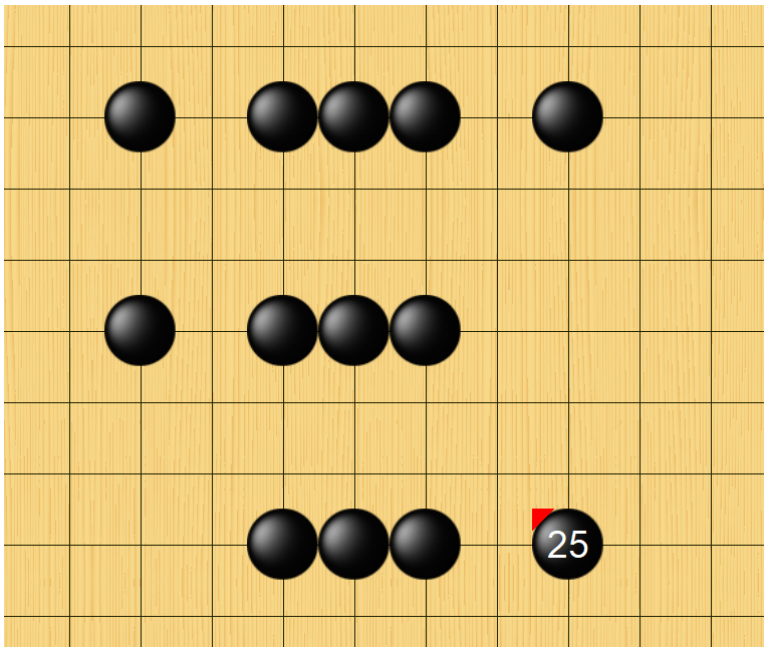
        bool flag2_1=(Same_Empty[i+4]>2 || (Same_Empty[i+4]==2 && Same_Empty_Same[i+4]==0))? true : false;
        bool flag2_2=(Same_Empty[i+4]==1 && Same_Empty_Same[i+4]==0)? true : false;
        bool flag2_3=(Same_Empty[i]>1 || (Same_Empty[i]==1 && Same_Empty_Same[i]==0))? true : false;

        if((flag1_1 && flag1_3 && judge_next(hang,lie,Same[i],i,color)) ||
           (flag2_1 && flag2_3 && judge_next(hang,lie,Same[i+4],i+4,color)))
```

```
        {
            Evaluated_Board[hang][lie].Direction[color-1].Living_3+=1;
        }
        else if( (((flag1_2 && flag1_3) || (flag1_1 && !flag1_3)) &&
judge_next(hang,lie,Same[i],i,color))
          || (((flag2_2 && flag2_3) || (flag2_1 && !flag2_3)) &&
judge_next(hang,lie,Same[i+4],i+4,color)))
                {
                    Evaluated_Board[hang][lie].Direction[color-1].Rush_3+=1;
                }
    }
}
```

（1）处理冲四和活四的残留情况

- 这里我们首先处理了在四连中提到过的冲四和活四的遗留情况，即形如：



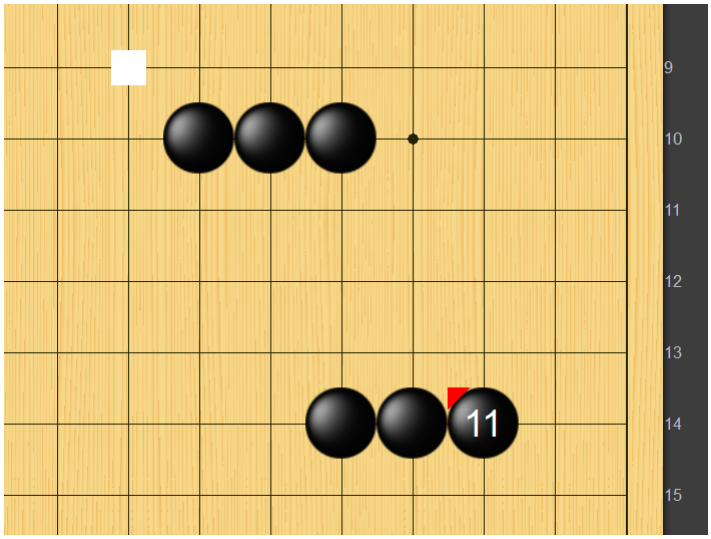
从上向下依次为活四、冲四、冲四(咱不考虑judge_next)。

（2）计算子条件，为判断活三冲三做准备

flag1_1,flag1_2,flag_1_3,flag2_1,flag2_2,flag2_3为一些子条件，目的是为了让后续的条件判断有条理一些

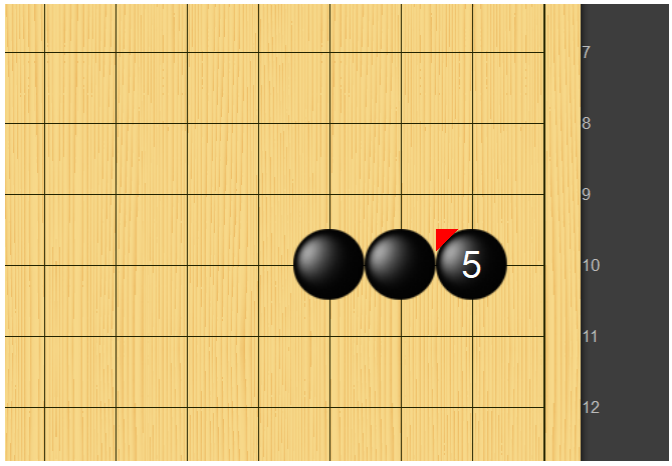
以下暂且设i为右方，i+4对应为左方：

- ```
  bool flag1_1=(Same_Empty[i]>2 || (Same_Empty[i]==2 &&
  Same_Empty_Same[i]==0))? true : false;
  ```
  -
  ```



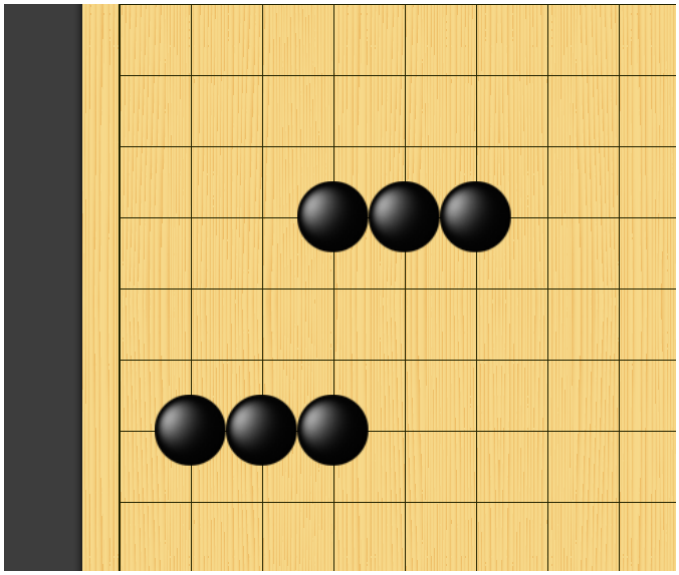
从上向下依次对应Same_Empty[i]>2 和 (Same_Empty[i]==2 && Same_Empty_Same[i]==0)

这里的(Same_Empty[i]==2 && Same_Empty_Same[i]==0)意味着它离边界有两个空位，为什么一定要强调边界呢，因为边界既意味着一种对进一步延展的限制，又意味着不会被禁手困扰，如图黑11落下后，当下一步黑棋继续落在黑11的右方时，一定能形成活四，因为坐标14处是边界不会落子，不可能形成长连禁手，这就与Same_Empty[i]>2等价了。这里是很容易忽视的细节。

- ```
  bool flag1_2=(Same_Empty[i]==1 && Same_Empty_Same[i]==0)? true : false;
  ```
  - 

    这里对应的情况就是离边界刚好有一个空位。

- ```
  bool flag1_3=(Same_Empty[i+4]>1 || (Same_Empty[i+4]==1 &&
  Same_Empty_Same[i+4]==0))? true : false;
  ```
  -



从上向下依次对应Same_Empty[i+4]>1 和 (Same_Empty[i+4]==1 && Same_Empty_Same[i+4]==0)

为什么在讨论右方时要加入左方的条件呢，因为左边的限制对右方的延展同样非常重要。

这两条条件满足其一的话意味着左方至少能有一个延展空间。

flag2_1,flag2_2,flag2_3与上面是完全镜像的，故不再赘述。

（3）下面根据flag判断活三和冲三

首先是活三，对活三的要求比较严格，即有一个方向能够延展两个位置，而同时对头方向能够延展一个位置，同时需要保证落下当前子之后，能够延展两个位置的方向上的第一个位置不会因为禁手而被限制。

```
(flag1_1 && flag1_3 && judge_next(hang,lie,Same[i],i,color)) （另一条件是镜像的）
```

然后是冲三，对冲三要求略松，即落子后能变成冲4，那么只要有一个方向能够安全延展就可。

```
(((flag1_2 && flag1_3) || (flag1_1 && !flag1_3)) &&
judge_next(hang,lie,Same[i],i,color) （另一个条件是镜像的）
```

同时这里需要注意，就像四连不包含所有冲四、活四的情况一样，三连也不包含所有冲三、活三的情况。剩余的情况需要保留到二连、一连处去讨论。

5：二连

```
else if(Same[i]+Same[i+4]==1)
{
    int flag=0;
    //活四冲四判断
```

```
    //形如上或下"oxxoxx" / "xxoxxo"为冲四
    //形如上加下"xxoxxoxx" 为活四
    if(Same_Empty[i]==1 && Same_Empty_Same[i]==2)
    {
        if(judge_next(hang,lie,Same[i],i,color))
        {
            flag++;       //检验“oxxoxx”右边那个o
        }
    }
    if(Same_Empty[i+4]==1 && Same_Empty_Same[i+4]==2)
    {
        if(judge_next(hang,lie,Same[i+4],i+4,color))
        {
            flag++;       //检验“xxoxxo”左边那个o
        }
    }

    if(flag==2)
    {
        Evaluated_Board[hang][lie].Direction[color-1].Living_4+=1;
    }
    else if(flag==1)
    {
        Evaluated_Board[hang][lie].Direction[color-1].Rush_4+=1;
    }
    else
    {
        //活三判断
        bool flag1_1=(Same_Empty[i]==1 && Same_Empty_Same[i]==1)? true : false;
        bool flag1_2=(Same_Empty_Same_Empty[i]>1 || (Same_Empty_Same_Empty[i]==1
&&
Same_Empty_Same_Empty_Same[i]==0))? true : false;
        bool flag1_3=(Same_Empty[i+4]>1 || (Same_Empty[i+4]==1 &&
Same_Empty_Same[i+4]==0))? true : false;

        bool flag2_1=(Same_Empty[i+4]==1 && Same_Empty_Same[i+4]==1)? true :
false;
        bool flag2_2=(Same_Empty_Same_Empty[i+4]>1 ||
(Same_Empty_Same_Empty[i+4]==1 &&
Same_Empty_Same_Empty_Same[i+4]==0))? true : false;
        bool flag2_3=(Same_Empty[i]>1 || (Same_Empty[i]==1 &&
Same_Empty_Same[i]==0))? true : false;

        if((flag1_1 && flag1_2 && flag1_3 &&
judge_next(hang,lie,Same[i],i,color)) ||
           (flag2_1 && flag2_2 && flag2_3 &&
judge_next(hang,lie,Same[i+4],i+4,color)))
        {
            Evaluated_Board[hang][lie].Direction[color-1].Living_3+=1;
        }
        else if( (flag1_1 && (flag1_2 || flag1_3) &&
judge_next(hang,lie,Same[i],i,color)) ||
                 (flag2_1 && (flag2_2 || flag2_3) &&
judge_next(hang,lie,Same[i+4],i+4,color)) )
        {
```

```
            Evaluated_Board[hang][lie].Direction[color-1].Rush_3+=1;
        }
    }

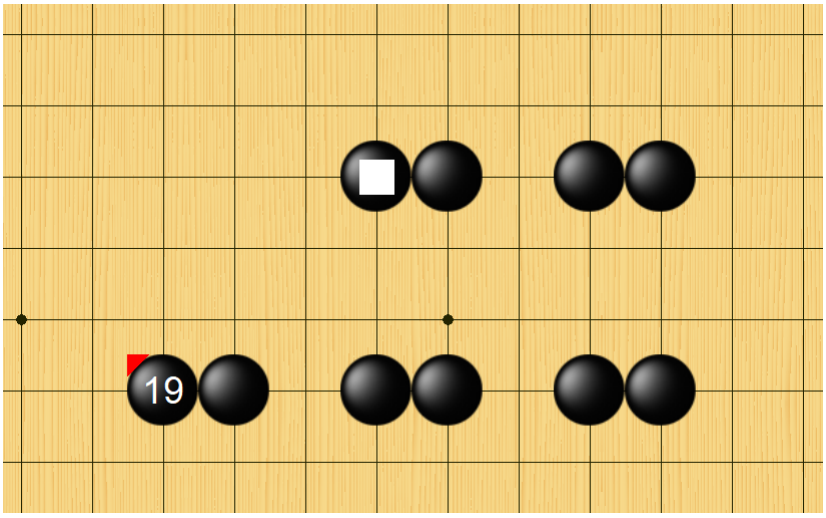
    //判断活2和冲2（计算价值用）
    bool flag1_1=(Same_Empty[i]>2 || (Same_Empty[i]==2 &&
Same_Empty_Same[i]==0))? true : false;
    bool flag1_2=(Same_Empty[i]>3 || (Same_Empty[i]==3 &&
Same_Empty_Same[i]==0))? true : false;
    bool flag1_3=(Same_Empty[i+4]>1 || (Same_Empty[i+4]==1 &&
Same_Empty_Same[i+4]==0))? true : false;

    bool flag2_1=(Same_Empty[i+4]>2 || (Same_Empty[i+4]==2 &&
Same_Empty_Same[i+4]==0))? true : false;
    bool flag2_2=(Same_Empty[i+4]>3 || (Same_Empty[i+4]==3 &&
Same_Empty_Same[i+4]==0))? true : false;
    bool flag2_3=(Same_Empty[i]>1 || (Same_Empty[i]==1 &&
Same_Empty_Same[i]==0))? true : false;

    if( (flag1_1 && flag1_2 && flag1_3 && judge_next(hang,lie,Same[i],i,color)
&&
judge_next(hang,lie,Same[i]+1,i,color)) ||
        (flag2_1 && flag2_2 && flag2_3 &&
judge_next(hang,lie,Same[i+4],i+4,color) &&
                            judge_next(hang,lie,Same[i+4]+1,i+4,color)) )
    {
        Evaluated_Board[hang][lie].Direction[color-1].Living_2+=1;
    }
    else if( (flag1_1 && (flag1_2 || flag1_3) &&
judge_next(hang,lie,Same[i],i,color) &&
                             judge_next(hang,lie,Same[i]+1,i,color)) ||
             (flag2_1 && (flag2_2 || flag2_3) &&
judge_next(hang,lie,Same[i+4],i+4,color) &&
                             judge_next(hang,lie,Same[i+4]+1,i+4,color)) )
    {
        Evaluated_Board[hang][lie].Direction[color-1].Rush_2+=1;
    }
}
```

（1）处理冲四和活四的残留情况

如图：



从上到下依次为冲四和活四

(2) 处理冲三和活三的残留情况

这里的流程也是先计算一些子条件，然后将这些子条件组合起来，具体怎么组合可以在棋盘上摆一摆，结合给出的式子细品。哥们写到这刚知道期末考试提前了，蚌埠住了，不细解释了。

(3)判断活2和冲2

同上

6：一连

一连实际上就是只有欲落之子一个，但这并不妨碍它落下后形成活四、冲四、活三、冲三乃至活二、冲二多种形状，它要处理的残留情况时最多的：

```
//单独一子
else if(Same[i]+Same[i+4]==0)
{
    //活四冲四判断
    //形如“oxoxxx”且中间o可落子位活四
    //若有两个冲四（两边对称）则刚好为禁手
    bool flag=false;
    if(Same_Empty[i]==1 && Same_Empty_Same[i]==3)
    {
        if(judge_next(hang,lie,Same[i],i,color))
        {    //需要判断的点是“oxoxxx”中间的o
            Evaluated_Board[hang][lie].Direction[color-1].Rush_4+=1;
            flag=true;
        }
    }
    if(Same_Empty[i+4]==1 && Same_Empty_Same[i+4]==3)
    {
        if(judge_next(hang,lie,Same[i+4],i+4,color))
        {
            Evaluated_Board[hang][lie].Direction[color-1].Rush_4+=1;
            flag=true;
```

```
        }
    }
    if(!flag)
    {
        //有冲四了就不用判断三了
        //活三冲三判断
        //形如“ooxoxxoo”或“ooxoxxo”或“oxoxxoo”的且中间o可落子为活三

        bool flag1_1=(Same_Empty[i]==1 && Same_Empty_Same[i]==2)? true : false;
        bool flag1_2=(Same_Empty_Same_Empty[i]>1 || (Same_Empty_Same_Empty[i]==1
&&
Same_Empty_Same_Empty_Same[i]==0))? true : false;
        bool flag1_3=(Same_Empty[i+4]>1 || (Same_Empty[i+4]==1 &&
Same_Empty_Same[i+4]==0))? true : false;

        bool flag2_1=(Same_Empty[i+4]==1 && Same_Empty_Same[i+4]==2)? true :
false;
        bool flag2_2=(Same_Empty_Same_Empty[i+4]>1 ||
(Same_Empty_Same_Empty[i+4]==1 &&
Same_Empty_Same_Empty_Same[i+4]==0))? true : false;
        bool flag2_3=(Same_Empty[i]>1 || (Same_Empty[i]==1 &&
Same_Empty_Same[i]==0))? true : false;

        if( (flag1_1 && flag1_2 && flag1_3 &&
judge_next(hang,lie,Same[i],i,color)) ||
            (flag2_1 && flag2_2 && flag2_3 &&
judge_next(hang,lie,Same[i+4],i+4,color)) )
        {
            Evaluated_Board[hang][lie].Direction[color-1].Living_3+=1;
            flag=true;
        }
        else if( (flag1_1 && (flag1_2 || flag1_3) &&
judge_next(hang,lie,Same[i],i,color)) ||
                 (flag2_1 && (flag2_2 || flag2_3) &&
judge_next(hang,lie,Same[i+4],i+4,color)) )
        {
            Evaluated_Board[hang][lie].Direction[color-1].Rush_3+=1;
            flag=true;
        }
    }

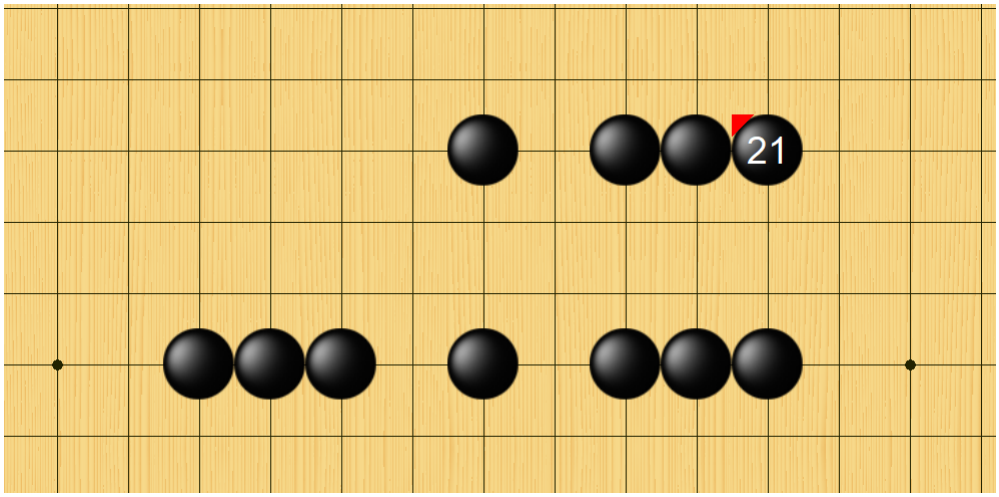
    if(!flag)
    {
        bool flag1_1=(Same_Empty[i]==1 && Same_Empty_Same[i]==1 &&
Same_Empty_Same_Empty[i]>=1)? true : false;
        bool flag1_2=(Same_Empty_Same_Empty[i]>1 || (Same_Empty_Same_Empty[i]==1
&&
Same_Empty_Same_Empty_Same[i]==0))? true : false;
        bool flag1_3=(Same_Empty[i+4]>1 || (Same_Empty[i+4]==1 &&
Same_Empty_Same[i+4]==0));

        bool flag2_1=(Same_Empty[i+4]==1 && Same_Empty_Same[i+4]==1 &&
Same_Empty_Same_Empty[i+4]>=1)? true : false;
```

```
        bool flag2_2=(Same_Empty_Same_Empty[i+4]>1 ||
(Same_Empty_Same_Empty[i+4]==1 &&
Same_Empty_Same_Empty_Same[i+4]==0))? true : false;
        bool flag2_3=(Same_Empty[i]>1 || (Same_Empty[i]==1 &&
Same_Empty_Same[i]==0));

        if( (flag1_1 && flag1_2 && flag1_3 &&
judge_next(hang,lie,Same[i],i,color) &&
                            judge_next(hang,lie,Same[i]+2,i,color)) ||
            (flag2_1 && flag2_2 && flag2_3 &&
judge_next(hang,lie,Same[i+4],i+4,color) &&
                            judge_next(hang,lie,Same[i+4]+2,i+4,color)) )
        {
            Evaluated_Board[hang][lie].Direction[color-1].Living_2+=1;
        }
        else if( (flag1_1 && (flag1_2 || flag1_3) &&
judge_next(hang,lie,Same[i],i,color) &&
                             judge_next(hang,lie,Same[i]+2,i,color)) ||
                  (flag2_1 && (flag2_2 || flag2_3) &&
judge_next(hang,lie,Same[i+4],i+4,color) &&
                             judge_next(hang,lie,Same[i+4]+2,i+4,color)) )
        {
            Evaluated_Board[hang][lie].Direction[color-1].Rush_2+=1;
        }
    }
}
```

(1)先处理冲四和活四的残留情况



从上到下依次为冲四和活四

(2)在没有残留的四的情况下，处理活三和冲三的残留情况

仍然是先计算各个子条件flag，然后组合起来进行判断。

(3)在没有残留的三的情况下，处理冲二和活二的残留情况

仍然是先计算各个子条件flag，然后组合起来进行判断。

7: judge_next()

> 接下来我们解决上面的遗留问题：如何递归地判断：
> 当前子落下后，原先的空位是否仍然可下

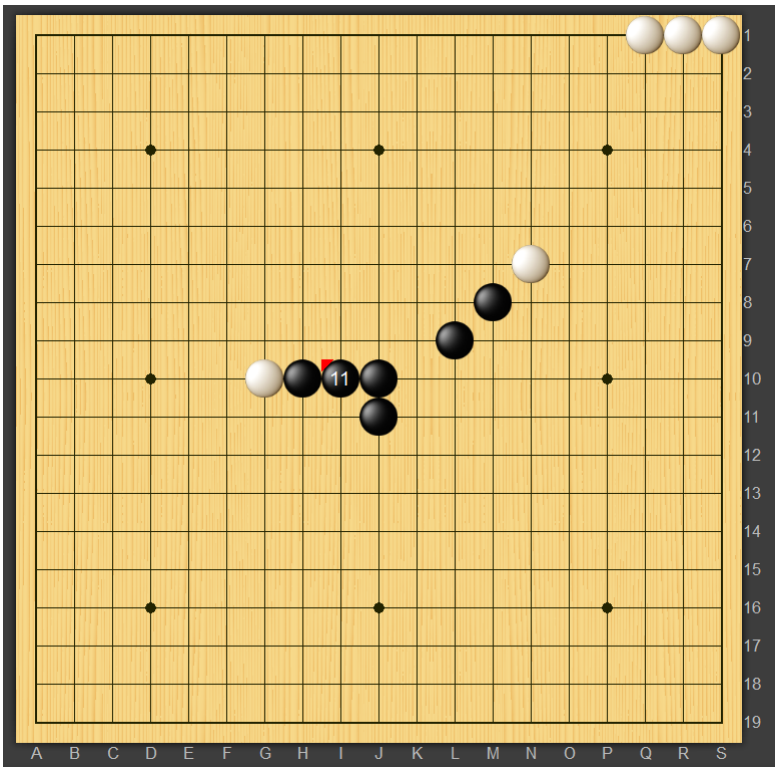
```
bool judge_next(int hang,int lie,int distance,int direction,int color){
    if(color==WHITE)
    {          //如果是白棋显然没有禁手问题
        return true;
    }
    int next_hang,next_lie;
    distance++;     //因为相隔一个，坐标数要加二
    if(direction>=4)
    {
        distance=-distance;        //方向相反，加减应随之相反
    }
    //计算关键点坐标
    switch(direction%4)
    {
        case 0:
            next_hang=hang-distance;
            next_lie=lie;
            break;
        case 1:
            next_hang=hang-distance;
            next_lie=lie+distance;
            break;
        case 2:
            next_hang=hang;
            next_lie=lie+distance;
            break;
        case 3:
            next_hang=hang+distance;
            next_lie=lie+distance;
            break;
        default:
            printf("传入的方向有误");
            break;
    }
    bool if_just_evaluate=false;
    if(aRecordBoard[hang][lie]!=NONE)
    {
        if_just_evaluate=true;
    }

    aRecordBoard[hang][lie]=color;         //将刚才那个子落了

    bool flag = next_just_check_ban(next_hang,next_lie,color);
```

```
    if(!if_just_evaluate)
    {
        //适应判断局面分的需要，在判断局面分时，原本这里就有子，不能擦除
        aRecordBoard[hang][lie]=NONE;         //恢复
    }
    if(flag){
        return true;
    }
    else{
        return false;
    }
}
```

我们直接结合一个例子来解释这个函数：



假设黑11为欲落子，它显然构成了连三，为了判断它是否是冲三需要判断在黑11落子后（k，10）位是否可下，此时调用了judge_next函数，调用方式为：

```
if(judge_next(hang, lie, Same[i], i, color))
    ......
```

此时方位为右方，所以i实际上等于2，Same[i] = 1，hang和lie即为黑9的坐标，color为黑色。

进入judge_next函数后：

- 首先判断颜色是否为白色——如果是白色则无禁手，直接返回true，而当前为黑色因此继续执行。
- 然后需要根据(hang,lie) 和 距离(distance = Same[i] = 1) 和 方位(direction = i = 2)计算出即将判断的点(k,10)的坐标，具体的计算方式只是一点技巧问题，不细嗦了，最终得到(k,10)的坐标(next_hang, next_lie)。
- 然后我们需要判断(hang,lie)处是否已经存在子，若已存在则我们后续不需要擦除，而若不存在我们将先在该处落子，等判断完毕再给它擦除掉。此时黑11还未落子，因此if_just_evaluate = false。

- 接下来我们先拟在黑11处落下黑子，然后调用next_just_check_ban(next_hang,next_lie,color)函数来判断(next_hang,next_lie)处是否为禁手。这个next_just_check_ban跟check_ban_and_evaluate函数基本相同，唯一的区别是前者只判断禁手而不评估价值，因此其相比check_ban_and_evaluate省去了计算冲三活二冲二等无关禁手而有关价值的内容，同时也不重置Remake_Evaluated_Board，具体可以见代码。
    - 在next_just_check_ban(next_hang,next_lie,color)函数中，我们为了判断(J,11) , (K,10) , (L,9)，(M,8)四子是否构成冲四，又必须判断(K,10)落下黑子后，(I,12)是否成为禁手，因此又将调用judge_next函数，这是一个递归的过程：
        - 在此次的judge_next函数中，我们显然最终又会调用next_just_check_ban函数，但递归的终点一定会出现next_just_check_ban函数不用调用judge_next即可判断明白禁手的情况，因此这个递归不是无限的。这里省略这个过程，直接快进到此次judge_next返回true的时候。

      由于judge_next返回了true，因此我们判定(J,11) , (K,10) , (L,9)，(M,8)四子构成了冲四，同样的方式我们判断出(H,10),(I,10),(J,10),(K.10)四子构成冲四，因此此时冲四和冲四形成了禁手，说明在黑11已落下的情况下，(K,10)是禁手，因此next_just_check_ban函数返回了false。

  judge_next函数接收到false后，首先将刚才拟落下的黑11擦除，再向check_ban_and_evaluate返回来false，check_ban_and_evaluate根据这个false判断出(H,10),(I,10),(J,10)三子并不是冲三，整个过程结束。

> 去理解一个递归方法显然是一件非常痛苦的事，得细品一下慢慢理解.....

【2】根据提炼后的信息判断禁手

我们已经得到了子落在该点可以形成的各种形状数，据此判断禁手就很容易了：

judge_if_banhand函数是在对手落子后判断其是否下了禁手的，因此里面的禁手种类划分非常细，目的是能很明确的说出对手下了什么禁手。

```
bool judge_if_banhand(int hang,int lie,int color,bool print){
    if(Value_Board[hang][lie].direction[color-1].Too_long==true){
        if(print)
            printf("长连禁手！\n");
        return false;
    }
    if(Value_Board[hang][lie].direction[color-1].Living_3>=2 &&
Value_Board[hang][lie].direction[color-1].Living_4==0 && Value_Board[hang]
[lie].direction[color-1].Rush_4==0){
        if(print)
            printf("双活三禁手！\n");
        return false;
    }
    if(Value_Board[hang][lie].direction[color-1].Living_4+Value_Board[hang]
[lie].direction[color-1].Rush_4>=2 && Value_Board[hang][lie].direction[color-
1].Living_3==0){
        if(print)
            printf("双活四/冲四禁手！\n");
        return false;
    }
```

```
    if(Value_Board[hang][lie].direction[color-1].Living_4+Value_Board[hang]
[lie].direction[color-1].Rush_4==1 && Value_Board[hang][lie].direction[color-
1].Living_3>=2){
        if(print)
            printf("四三三(三)禁手！\n");
        return false;
    }
    if(Value_Board[hang][lie].direction[color-1].Living_4+Value_Board[hang]
[lie].direction[color-1].Rush_4==2 && Value_Board[hang][lie].direction[color-
1].Living_3>=1){
        if(print)
            printf("四四三(三)禁手！\n");
        return false;
    }
    if(Value_Board[hang][lie].direction[color-1].Living_4+Value_Board[hang]
[lie].direction[color-1].Rush_4==3 && Value_Board[hang][lie].direction[color-
1].Living_3==1){
        if(print)
            printf("四四四三禁手！\n");
        return false;
    }
    //注意43不是禁手
    return true;
}
```

而当我们自己在做决策时，我们实际并不关心某个点可能形成哪种禁手，而只关心其是不是禁手，因此在我们做决策过程中判断禁手不需要划分这么细，参考next_just_check_ban函数的最后：

```
//判断禁手
if(Living_3==1 && (Living_4+Rush_4)==1)
{
    //先考虑特殊情况，即四三不是禁手。
    return true;
}

if(Living_3+Living_4+Rush_4>=2)
{
    //除了一个四一个三以外，其余情况下只要和大于等于2，一定是禁手。
    return false;
}
else
{
    return true;
}
```

【3】根据提炼后的信息评估该点价值

```
//对应每种形状的价值
#define Value_Right_5 50000
#define Value_Living_4 4320
#define Value_Rush_4 720
#define Value_Living_3 720
#define Value_Rush_3 100
#define Value_Living_2 120
#define Value_Rush_2 20

void evaluate_value(int hang, int lie){
    int i;
    int value[2]={0,0};
    for(i=0;i<2;i++){
        value[i]+=Evaluated_Board[hang][lie].Direction[i].Right_5*Value_Right_5;
        value[i]+=Evaluated_Board[hang]
[lie].Direction[i].Living_4*Value_Living_4;
        value[i]+=Evaluated_Board[hang][lie].Direction[i].Rush_4*Value_Rush_4;
        value[i]+=Evaluated_Board[hang]
[lie].Direction[i].Living_3*Value_Living_3;
        value[i]+=Evaluated_Board[hang][lie].Direction[i].Rush_3*Value_Rush_3;
        value[i]+=Evaluated_Board[hang]
[lie].Direction[i].Living_2*Value_Living_2;
        value[i]+=Evaluated_Board[hang][lie].Direction[i].Rush_2*Value_Rush_2;
        Evaluated_Board[hang][lie].Score[i]=value[i];
    }
    Evaluated_Board[hang][lie].All_score = value[0] + value[1];
}
```

这里将五连的价值定义为50000，活4价值定义为4320是为了确保其他值无论如何相加都无法超越这两种形状的价值，因为这两种形状是必胜的形状。其它价值主要是一些经验值，值得说明的是为什么给冲二赋了20的值：

这虽然看起来没什么用，但对于前几步落子还是有用的，它确保了你的落子任何情况下不会脱离“主战场”。

底下的for循环分别对应该点对白棋的价值和对黑棋的价值，将对两者的价值相加得到该点的总价值。

(二）由各单点价值得出局面整体价值

上面我们判断了单个点的禁手情况以及价值评估，但实际进行决策时我们需要得到的是某个局面整体的价值。

- 评估局面时，我们首先对棋盘上每颗子调用check_ban_and_evaluate方法得到其构成的形状信息，这里同时回答了一个问题：为什么judge_next函数中有一个名为if_just_evaluate的布尔变量，就是为了这里判断那些已经有落子的点的价值时，放置在judge_next的末尾将落子抹去(都是debug的痛苦回忆了……咋都de不出bug，一打印棋盘发现全盘的落子都被抹去了)。
- 得到每个落子的形状信息后，可以将其各种形状数加起来得到总价值，注意这里要除以对应形成该形状的棋子数，例如：

- 对一个活4而言，它由四颗棋子组成，而这四颗棋子中的每颗棋子在check_ban_and_evaluate中都判断出了living_4的存在，因此在累加其价值时计入了4个living_4的价值，但实际上只有一个活4，因此需要将这个总值 / 4，活三/冲三与其同理，需要将总值/3。
- 对白棋而言，局面总价值为白棋总价值 - 黑棋总价值； 对黑棋而言，局面总价值为黑棋总价值 - 白棋总价值。

综上，我们写出代码如下：

```
int evaluate_jumian_value(int color){
    //优先级是对方活四>对方冲四>己方活四>对方活3
    int Sef_Value_Living_4=100000;
    int Rival_Value_Living_4=10000000;
    int Sef_Value_Rush_4=720;
    int Rival_Value_Rush_4=1000000;
    int Sef_Value_Living_3=720;
    int Rival_Value_Living_3=50000;
    int Sef_Value_Rush_3=480;
    int Rival_Value_Rush_3=720;
    int Sef_Value_Living_2=480;
    int Rival_Value_Living_2=480;
    int Sef_Value_Rush_2=20;
    int Rival_Value_Rush_2=100;
    //这里关于己方活三，活二，对面冲三，等的赋值还有待商榷

    int i,j;
    int Right_5[2]={0,0};
    int Living_4[2]={0,0};
    int Rush_4[2]={0,0};
    int Living_3[2]={0,0};
    int Rush_3[2]={0,0};
    int Living_2[2]={0,0};
    int Rush_2[2]={0,0};

    for(i=0;i<SIZE;i++)
    {
        for(j=0;j<SIZE;j++)
        {
            if(aRecordBoard[i][j]==BLACK)
            {
                check_ban_and_value(i,j,BLACK);
                Living_4[0]+=Evaluated_Board[i][j].Direction[0].Living_4;
                Rush_4[0]+=Evaluated_Board[i][j].Direction[0].Rush_4;
                Living_3[0]+=Evaluated_Board[i][j].Direction[0].Living_3;
                Rush_3[0]+=Evaluated_Board[i][j].Direction[0].Rush_3;
                Living_2[0]+=Evaluated_Board[i][j].Direction[0].Living_2;
                Rush_2[0]+=Evaluated_Board[i][j].Direction[0].Rush_2;
            }
            if(aRecordBoard[i][j]==WHITE)
            {
                check_ban_and_value(i,j,WHITE);
                Living_4[1]+=Evaluated_Board[i][j].Direction[1].Living_4;
                Rush_4[1]+=Evaluated_Board[i][j].Direction[1].Rush_4;
                Living_3[1]+=Evaluated_Board[i][j].Direction[1].Living_3;
```

```
                Rush_3[1]+=Evaluated_Board[i][j].Direction[1].Rush_3;
                Living_2[1]+=Evaluated_Board[i][j].Direction[1].Living_2;
                Rush_2[1]+=Evaluated_Board[i][j].Direction[1].Rush_2;
            }
        }
    }

    int value=0;
    if(color==BLACK)
    {
        value+=Living_4[0]*Sef_Value_Living_4/4;
        value+=Rush_4[0]*Sef_Value_Rush_4/4;
        value+=Living_3[0]*Sef_Value_Living_3/3;
        value+=Rush_3[0]*Sef_Value_Rush_3/3;
        value+=Living_2[0]*Sef_Value_Living_2/2;
        value+=Rush_2[0]*Sef_Value_Rush_2/2;

        value-=Living_4[1]*Rival_Value_Rush_4/4;
        value-=Rush_4[1]*Rival_Value_Rush_4/4;
        value-=Living_3[1]*Rival_Value_Living_3/3;
        value-=Rush_3[1]*Rival_Value_Rush_3/3;
        value-=Living_2[1]*Rival_Value_Living_2/2;
        value-=Rush_2[1]*Rival_Value_Rush_2/2;
    }
    else
    {
        value+=Living_4[1]*Sef_Value_Living_4/4;
        value+=Rush_4[1]*Sef_Value_Rush_4/4;
        value+=Living_3[1]*Sef_Value_Living_3/3;
        value+=Rush_3[1]*Sef_Value_Rush_3/3;
        value+=Living_2[1]*Sef_Value_Living_2/2;
        value+=Rush_2[1]*Sef_Value_Rush_2/2;

        value-=Living_4[0]*Rival_Value_Rush_4/4;
        value-=Rush_4[0]*Rival_Value_Rush_4/4;
        value-=Living_3[0]*Rival_Value_Living_3/3;
        value-=Rush_3[0]*Rival_Value_Rush_3/3;
        value-=Living_2[0]*Rival_Value_Living_2/2;
        value-=Rush_2[0]*Rival_Value_Rush_2/2;
    }
    return value;
}
```

这里值得注意的是

```
//优先级是对方活四>对方冲四>己方活四>对方活3
```

原因是当我们评估局面价值时，总是以某一方为主视角，且在该方刚落完子后评估局面，这意味着此时下一步轮对方下，因此此时如果对方有冲四或者活四是必赢的(对应自己必输)，这意味着即使己方此时有活四或者冲四也没有用(慢了一步)。