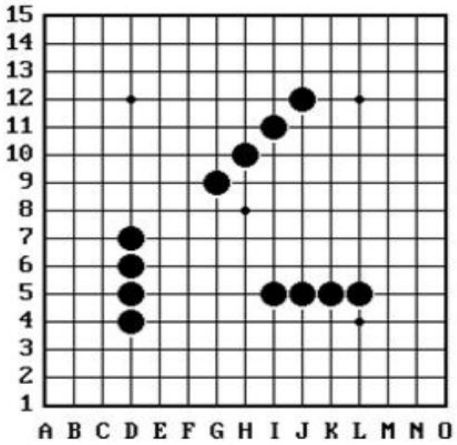
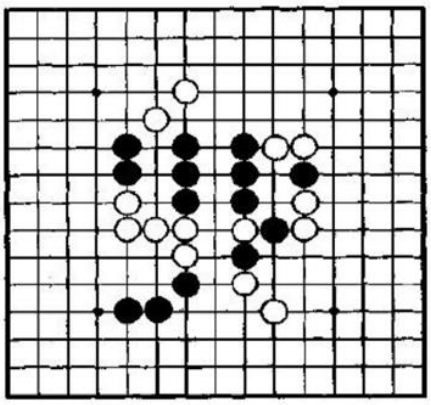
# 对抗搜索算法实现五子棋 AI

## 一、问题描述

五子棋是一种两人对弈的纯策略型棋类游戏，通常双方分别使用黑白两色的棋子，下在棋盘直线与横线的交叉点上，先形成 5 子连线者获胜。



本次实验我实现了一个人机对弈的五子棋 AI。AI 设计中的关键函数的设计方法都在报告中进行了详细的阐释，并且**介绍了普通版本、改进版本等多个版本**。报告中粘贴的都为简化的伪代码，详细代码可以参考源码，源码中注释齐全。

我使用了 Python 语言进行源代码的编写，采用极大极小值搜索算法和 α－β 剪枝算法进行搜索，同时使用 tkinter 库完成了图形化演示界面。程序的具体使用方法见第七条。

## 二、问题求解思路简析

五子棋是一种确定性的、完整信息的、双方轮流行动的零和游戏。游戏中，人类玩家走出一步，然后 AI 思考并回应一步，直到一方获胜或者平局才结束。游戏有一张棋盘、黑色方和白色方。

因此，在代码中，可以使用一个二维数组来表示整张棋盘，二维数组中每个位置的值表示这个地方有黑子/白子/没有子。伪代码表示的程序基本结构如下：

```
gobang_board = [SIDE_LEN][SIDE_LEN]  # 二维数组表示棋盘，此处伪码
HUMAN, AI, EMPTY = -1, 1, 0  # 棋盘棋子

player = HUMAN    # 人先走

while True:
    if player == HUMAN:
```

```
            r, c = get_move_human()
        elif player == AI:
            r, c = get_move_ai()
        # 在棋盘上下一步棋
        move(player, (r, c))
        # 判断是否获胜
        if game_win():
            # 进行获胜相应显示
            return
        # 交替行动
        player = -player
```

那么，问题的关键就在于如何实现 get_move_ai 函数，也就是说当 AI 在面对一个棋局时，如何选择走出哪一步。

AI 怎么知道下在哪个地方最合适呢？**显然只能通过计算来完成**。我们可以想办法获取所有可能下棋子的位置，然后通过一个合适的评估函数来为每一个位置进行评分，最后从所有位置中选择评分对 AI 最有利的那个位置下棋就可以了。因此，问题就转变为编写一个合适的**评估函数**，以及一个有效的**生成函数**。这两个地方都是难点所在。

此外，我们人在下棋的时候，通常会稍稍往后面几步棋考虑一下，以便让我们做出的选择能够不仅仅局限于眼前的利益，可以考虑的深远一些，从而走出更有利的招数。所谓走一步想七步，我们的 AI 如果也能和人一样多往后面想几步的话，棋力也会有很大的提升。所以，我们还需要一个搜索函数，从而实现多层的搜索和选择。

# 三、对抗搜索简介

对抗搜索也称为博弈搜索，在人工智能领域可以定义为：**有完整信息的、确定性的、轮流行动的、两个游戏者的零和游戏**（如象棋）。

游戏：意味着处理互动情况，互动意味着有玩家会参与进来（一个或多个）；

确定性的：表示在任何时间点上，玩家之间都有有限的互动；

轮流行动的：表示玩家按照一定顺序进行游戏，轮流出招；

零和游戏：意味着游戏双方有着相反的目标，换句话说：在游戏的任何终结状态下，所有玩家获得的总和等于零，有时这样的游戏也被称为严格竞争博弈；

关于零和，也可以这样来理解：自己的幸福是建立在他人的痛苦之上的，二者的大小完全相等，因而双方都想尽一切办法以实现“损人利己”。零和博弈的结果是一方吃掉另一方，一方的所得正是另一方的所失，整个社会的利益并不会因此而增加一分。

显然，五子棋是确定性的、有完整信息的、轮流行动的零和游戏，因此非常

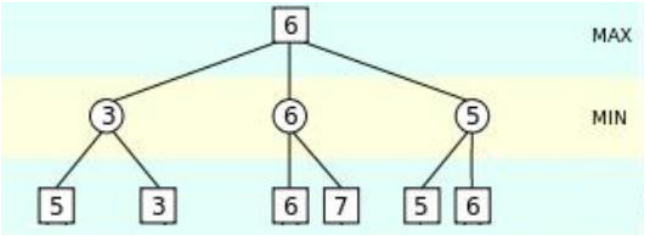
适合采用对抗搜索的算法进行搜索。

假设我们已经写好了评估函数和生成函数，其中评估函数能够根据当前局面给出一个得分，得分越高表示局面对 AI 越有利，越低表示对 AI 越不利；生成函数可以生成下一步下棋的可选位置。那么，我们要怎么选择出最优的下棋位置呢？

## 1、极大极小值搜索

如果我们只考虑一层搜索，其实实现方法很简单：**使用生成函数生成所有可能下棋的位置，并使用评估函数对每个位置进行评分。然后，从中选择得分最高的位置即可**，因为这个位置对 AI 最有利。这实际上就是一个极大值搜索过程。同时，我们可以考虑另一种情况：如果下一步是 HUMAN 走，那么我们就要从所有位置中选择评分最低的那个位置下棋，因为这个位置对 AI 最不利，相对来说，也就是对 HUMAX 最有利的位置。（当然，HUMAN 方是由人操控的，但是我们这里假设双方都按照自己认为的最佳着法行棋）。我们称为 AI 的 MAX 层，即 AI 要保证自己下棋的评分最大化；HUMAN 为 MIN 层，即 HUMAN 要保证 AI 的下棋评分最小，也就是对自己最有利。

上面是只有一层的搜索，如果要考虑多层搜索，第一层是 AI 下棋，第二层是玩家下棋，第三层是 AI 下棋，第四层是玩家下棋，依次类推。假设每一层有 50 个可选择的位置，每个位置看做树的一个节点，那么第一层是根节点下面的子节点，有 50 个节点，第二层是第一层下面的子节点，就有 50×50 个节点，第三层就有 50×50×50 个节点，依次类推，这样会形成一个巨大的博弈树。我们要做的就是搜索这棵树，找到对于 AI 最有利的下棋位置。




假设一个两层的博弈树，如上图，最上面一层是树的根节点，这里 MAX 表示会选取下一层子节点中评分最高的。第二层的 MIN 表示会选取下一层子节点中评分最低的。第三层是叶子节点，只需要计算评分。注意：只有在叶子节点时才会计算评分，在树的中间层，对于 AI 来说暂时是无法知道哪一个节点是最有利的。

极大极小值搜索的伪代码如下：其中的 get_max 和 get_min 分别对应着上述的极大值搜索和极小值搜索。

```
def min_max(gobang_board, player, depth):
    # 返回当前局面对于 player 而言的最佳下棋位置
    moves = generate.generate(gobang_board, player)
    best_move_pos_list = []
```

```
    if player == const.MAX_P:
        max_score = const.SCORE_MIN
        for rc_pos in moves:
            下一步棋
            # 因为已经走了一步，故此层为 min 层，取孩子们的最小值
            对孩子使用 get_min(极小值搜索)计算得分
            如果孩子的得分比 max_score 大，更新得分
    else:
        min_score = const.SCORE_MAX
        for rc_pos in moves:
            下一步棋
            # 因为已经走了一步，故此层为 max 层，取孩子们的最大值
            对孩子使用 get_max(极大值搜索)计算得分
            如果孩子的得分比 min_score 小，更新得分

    best_move_pos = best_move_pos_list[random]# 若有多个最优位置，随机选择
    return best_move_pos
```

## 2、α－β剪枝算法

极大极小值搜索算法的缺点就是当博弈树的层数变大时，需要搜索的节点数目会指数级增长。比如上面每一层的节点为 50 时，六层博弈树的节点就是 50 的 6 次方，运算时间会非常漫长。

在上面的例子中，我们会计算所有叶子节点的评分，但这个不是必要的。如果可以去掉那些一定不会选择的叶子节点，那么可以极大地优化函数的效率。如何实现呢？

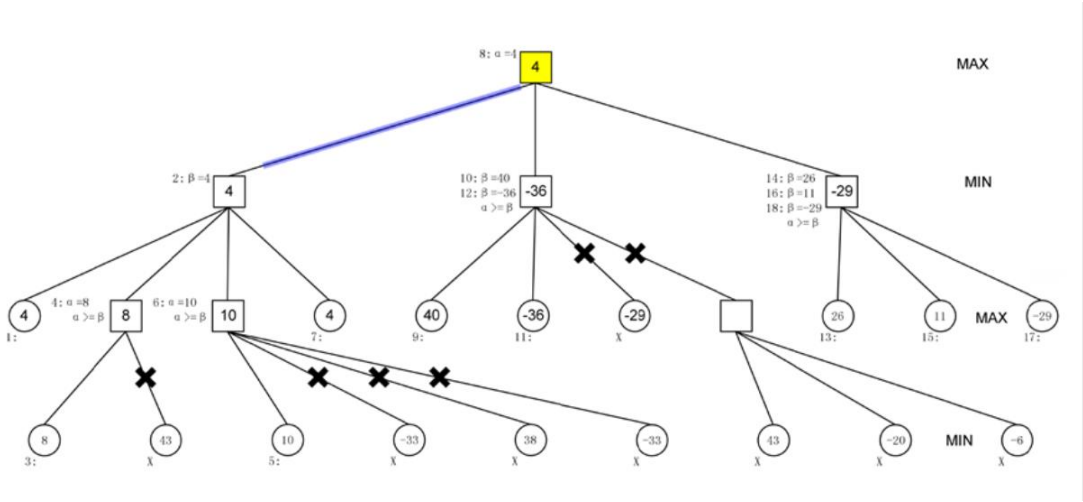
Alpha-Beta 剪枝就是用来**将搜索树中不需要搜索的分支裁剪掉**，以提高运算速度。基本的原理是：

当一个 MIN 层节点的 α 值 ≤ β 值时 ，剪掉该节点的所有未搜索子节点；

当一个 MAX 层节点的 α 值 ≥ β 值时 ，剪掉该节点的所有未搜索子节点。

其中 α 值是该层节点当前最有利的评分，β 值是父节点当前的 α 值，根节点因为是 MAX 层，所以 β 值 初始化为正无穷大(+∞)。

初始化节点的 α 值，如果是 MAX 层，初始化 α 值为负无穷大(-∞)，这样子节点的评分肯定比这个值大。如果是 MIN 层，初始化 α 值为正无穷大(+∞)，这样子节点的评分肯定比这个值小。




Alpha-beta 基于这样一种朴素的思想：**时时刻刻记得当前已经知道的最好选择**，如果从当前格局搜索下去，不可能找到比已知最优解更好的解，则停止这个格局分支的搜索（剪枝），回溯到父节点继续搜索。

Alpha-beta 算法可以看成变种的 Minimax，基本方法是从根节点开始采用深度优先的方式构造格局树，在构造每个节点时，都会读取此节点的 alpha 和 beta 两个值，其中 alpha 表示搜索到当前节点时已知的最好选择的下界，而 beta 表示从这个节点往下搜索最坏结局的上界。由于我们假设对手会将局势引入最坏结局之一，因此当 beta 小于 alpha 时，表示从此处开始不论最终结局是哪一个，其上限价值也要低于已知的最优解，也就是说已经不可能此处向下找到更好的解，所以就会剪枝。

程序中，我的 Alpha-beta 算法的伪码如下：

```
def alpha_beta(gobang_board, player, depth):
    # 返回当前局面对于 player 而言的最佳下棋位置，本函数由 minmax 函数修改而来
    count_pruning = 0   # 记录剪枝次数
    alpha, beta = const.SCORE_MIN, const.SCORE_MAX
    moves = generate.generate(gobang_board, player)
    best_move_pos = (None, None)

    if player == const.MAX_P:
        for rc_pos in moves:
            计算这一步的得分
            if score > alpha:
                更新 alpha 值和 best_move
                if 获胜:
                    break   # 如果已经找到获胜点，则退出
            # 剪枝算法不能考虑 score=alpha 的情况，因为 score 的计算不完全
            # elif score == beta:
            #     best_move_pos_list.append(rc_pos)
    else:
```

```
        for rc_pos in moves:
            β剪枝，此处实现类似于上面，不再赘述

    return best_move_pos
```

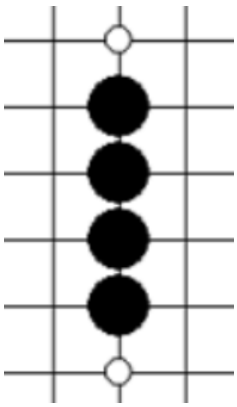
# 四、评估函数设计

五子棋的评估函数是整个问题中最难的部分。人类在下棋的时候，如何判断场上形势是有利还是不利？一般情况下我们是通过评估场上的棋型来判断局势。AI 同理，最常用的评估函数写法是，统计场上的各类棋型的数目，对每种棋型赋予不同的分值，最后计算得到当前棋盘的分数，从而判断场上局面如何。

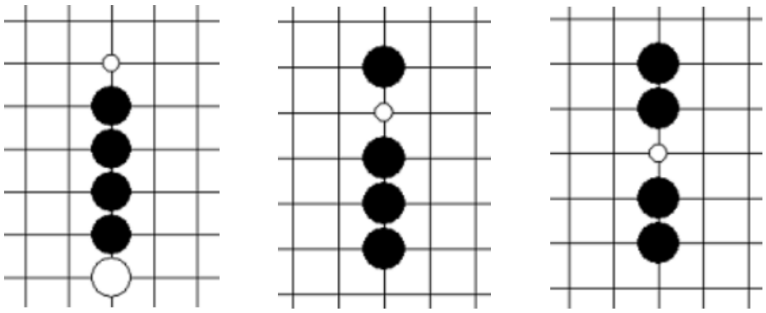
## 1、棋型简介

通常意义下的棋型有以下几种：

- 连五：顾名思义，五颗同色棋子连在一起，不需要多讲。
- 活四：有两个连五点（即有两个点可以形成五），图中白点即为连五点。稍微思考一下就能发现，活四出现的时候，如果对方不能立刻连五，那么对方已经输了。

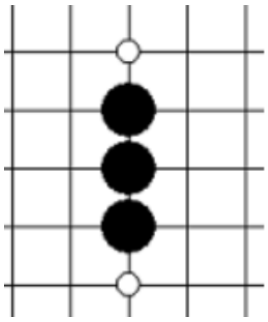
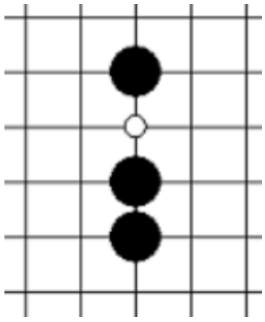


- 冲四：有一个连五点，如下面三图，均为冲四棋型。图中白点为连五点。相对比活四来说，冲四的威胁性就小了很多，因为这个时候，对方只要跟着防守在那个唯一的连五点上，冲四就没法形成连五。

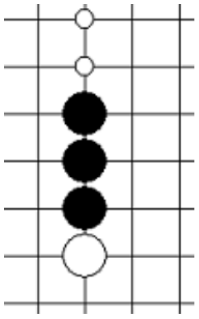
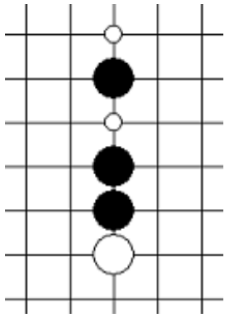
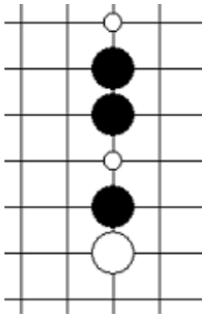


- 活三：可以形成活四的三，如下图，代表两种最基本的活三棋型。图中白点

为活四点。活三棋型是我们进攻中最常见的一种，因为活三之后，如果对方不以理会，将可以下一手将活三变成活四，而我们知道活四是已经无法单纯防守住了。所以，当我们面对活三的时候，需要非常谨慎对待。在自己没有更好的进攻手段的情况下，需要对其进行防守，以防止其形成可怕的活四棋型。

- 眠三：只能够形成冲四的三，如下各图，分别代表最基础的六种眠三形状。图中白点代表冲四点。眠三的棋型与活三的棋型相比，危险系数下降不少，因为眠三棋型即使不去防守，下一手它也只能形成冲四，而对于单纯的冲四棋型，我们知道，是可以防守住的。

- 活二和眠二同理，而且非常灵活。下面不再列出棋型图片。

在代码中，使用列表表示棋型。如[1,1,1,1,1]表示连五棋型。因为每种棋型可能都有多种变式，故每种棋型都存为一个列表，列表中存放该种棋型的全部变式。三元组中其他两个元素是与棋型有关的一些信息，存储于此是为了节约不必要的计算耗费，提高效率。部分表示方法如下：

```
# 棋型
SHAPE_LIST = [
    # 各棋型存储为三元组，分别为
    # ([棋型列表],列表元素个数,列表倒中最后一个 1 的后面一位的下标)
    # 每种不对称棋型都有两种情况
    # 连五
    [
        ([1, 1, 1, 1, 1], 5, 5),
    ],
    # 活四
    [
        ([0, 1, 1, 1, 1, 0], 6, 5),
```

```
        ],
        .......     # 其他棋型，这里不再列举
    ]
```

## 2、统计棋型的方法

我们知道，棋型是针对棋盘上的某一行/某一列/某条斜线上的棋子而进行的。如何记录棋盘上的棋形个数？一个很直观的方法是，棋盘上有 15 条水平线，15 条竖直线，不考虑长度小于 5 的斜线，有 21 条从左上到右下的斜线，21 条从左下到右上的斜线。因此，我们可以想办法把整个棋盘划分为上述的一条条线，然后对每一条线分别对黑棋和白棋查找是否有符合的棋型，并且统计棋型的数目即可。

那么，如何统计一条线上的棋型数目呢？由于前面我们使用一维向量表示各种棋型，因此可以在每一条线的检测过程中，遍历一次棋型列表，查找某一棋型是否为这条线的子列表。如果是的话，说明存在该棋型，那么对应的棋型种类的计数个数加一即可。对于整张棋盘，先将棋盘划分为一条条线，然后对每一条线进行统计即可。伪代码如下：

```
def count_line_type(line, who):
    # 传入一行棋盘,检测该数组中出现的棋型数目
    # who 参数是指目前统计哪位棋手的棋型,而不是当前谁下
    for each_type in SHAPE_LIST[type_name]:
        if each_type 包含于 line:
            count_chess_type[type_name] += 1
```

```
def count_board_type(gobang_board):
    # 统计整个棋盘的棋型种类
    # 将整个棋盘横竖斜进行划分,形成一个列表,列表每个元素都是棋盘上的一行/列/斜
    gobang_list = convert_board_to_list(gobang_board)
    for each_line in gobang_list:
        count_line_type(each_line, const.HUMAN)
        if count_chess_type_human[FIVE] > 0:  # 如果统计到连五,直接返回
            return
        count_line_type(each_line, const.AI)
        if count_chess_type_ai[FIVE] > 0:
            return
```

在统计过程中有许多地方可以进行优化。如：找到连五、活四等必赢棋型后，可以直接退出查找，因为后续的查找已经没有必要进行；统计一行的过程中，如果找到棋型，需要做个标记，避免同一处棋子被重复统计为多个棋型。

```
    # 连五 活四
    for type_name in range(const.FIVE, const.LFOUR+1):
        for each_type_tuple in const.SHAPE_LIST[type_name]:
            each_type = each_type_tuple[0]
```

```
            length = each_type_tuple[1]
            for i in range(len_of_line-length+1):
                if line[i:i+length] == each_type:
                    count_chess_type[type_name] += 1
                    return  # 连五活四为必赢棋型,不需要继续检测
```

## 3、统计棋型方法的改进

上面的评估函数存在一个非常严重的问题：将棋盘进行划分时，有许多划分的线都为空，没有必要进行统计。因此，有两种方法进行统计：

1、对于棋盘上每个不为空的点，生成其横、纵、左上-右下、左下-右上四条线，进行统计。

这种方法的优点是，当棋盘上的棋子数很少时，效率极高。但是有一个实现难点，难以保证棋盘上每条线最多仅被记录一次。我设想过每拆解一条线以后，在列表中进行搜索，如果有同样的线，则不将这条线插入列表。但这种解法有一个问题：如果棋盘上真的存在两条相同的线，那么最终仅计算了一次，显然不妥。因此，最终没有采用这个优化方法。

2、在划分棋盘的过程中，检查该条线上是否有棋子。如果一颗棋子都没有，那么就放弃这条线。

这种方法的实现仅需要在原代码上进行少量改动，非常方便。程序中便采用了这种方法。伪码如下：

```
def convert_board_to_list(gobang_board):
    # 本方法对上述两个方法都进行改进
    # 扫描棋盘，取出有子的边，并返回边组成的列表
    # gobang_board = np.array(gobang_board)
    gobang_list = []
    # 横
    for i in gobang_board:
        if i.any() != const.EMPTY:   # 关键！！当这条线不全为空时，再加入
            gobang_list.append(list(i))
    # 竖
    for i in gobang_board.T:
        if i.any() != const.EMPTY:
            gobang_list.append(list(i))
    # 斜-左上到右下 和 斜-右上到左下
    # 代码较为复杂，不在此处粘贴
    return gobang_list
```

## 4、计算得分的方法

计算分数时必须考虑的一点是：下一步是哪位行棋。对于同样的棋型，如冲四，如果下一步是冲四方行动，那么可以直接形成连五获胜；但是如果下一步是另一方行动，只需要简单的堵一下即可破解。因此，行棋者对于局面的评估非常重要。

假设已经考虑了行棋者，那么对于已经统计好的棋型，如何计算评分呢？我记录了一个棋型分值列表，按照下棋经验对每种棋型赋予了对应的分值。如下：

```
    # 棋型对应分值,注意顺序必须匹配
    SHAPE_SCORE = [
        99999,  # 连五
        8000,   # 活四
        4000,   # 冲四  双冲四/双活三相当于活四
        4000,   # 活三
        300,    # 眠三
        300,    # 活二
        50,     # 眠二
    ]
    BONUS_SCORE = 1000  # 行棋者的加分
```

连五表示已经获胜，故得分最高。活四的获胜概率极高，故分值也很高。同时，冲四、活三等必防棋型的分值也很高，并且两个以上的冲四或活三的分值都相当于一个活四。同时，对行棋者赋予一定加分。评分时先统计好各棋型的数目，再按照上述规则进行计分即可。

需要注意的是，计分过程中需要进行一些特殊处理和优化处理。如，统计到连五时，直接返回预设值，因为该棋型已经获胜。统计到连四棋型并且下一步是该玩家行棋时，也相当于获得胜利，也可以返回。统计到冲四、活三等必防棋型时，需要加上行棋者的额外得分，以便 AI 对这些棋型引起重视。最后，统计到活二等小收益棋型时，不再对行棋者加分，防止对评分造成太大影响。伪码如下：

```
def cal_score(player):
    # 根据棋型统计结果计算得分,返回值为(human_score,ai_score)
    # 注意,必须传入下一步是谁行棋,便于加上行棋手加分
    sh, sa = 0, 0  # score_human,score_ai
    ch, ca = count_chess_type_human, count_chess_type_ai
    if 连五 或 活四/冲四并且下一步是该选手行棋:
        return 预设值（预设值足够大，确保获胜很明显）
    if 活四 冲四 活三:
        计算棋型得分，加上行棋者附加分
        return 得分情况
    # 前面使用许多的 return，是因为眠三、眠二棋型太多，不便统计
    # 因此，能不统计就不统计
    # 其他棋型的统计
    if 其他棋型:
        计算棋型得分，注意不能加行棋者额外分
        return 预设值（预设值足够大，确保获胜很明显）
```

最后，对上述各方法进行汇总，即可写出 evaluate 函数。首先将棋盘划分为一条条线，然后统计每条线的棋型种类，再根据统计结果计算得分，最后求出 AI 得分和 HUMAN 得分之差，即为最终的局面得分。该得分反应了局势对于 AI 的有利程度。代码如下：

```
def evaluate(gobang_board, player):
    # 评估函数,返回当前局面对 ai 的有利性,ai 越有利,返回越大
    # evaluate 函数必须知道下一步是谁行棋
    count_board_type(gobang_board)
    score_human, score_ai = cal_score(player)
    return score_ai-score_human
```

# 五、生成函数

生成函数是五子棋 AI 实现中另一个极度重要的函数，生成函数的效率直接影响了整个 AI 的运行效率。生成的可能着法越精简，便越能减少不必要的评估时间。由于搜索深度每加深一层，生成的节点数目都会指数式增长，故生成函数的效率非常重要。

那么，AI 到底有哪些可能的着法呢？一个最简单的生成函数的写法就是：返回棋盘上所有的空位置。代码如下：

```
def generate_2021_4_18(gobang_board):
    # 根据当前局面，生成可以落子的位置
    moves = []
    for r in range(const.SIDE_LEN):
        for c in range(const.SIDE_LEN):
            if gobang_board[r][c] == const.EMPTY:
                moves.append((r, c))
    return moves
```

容易知道上面的生成函数生成了许多的冗余节点。因为有很多孤立的位置不可能是最优的着法。而且，下棋的前期，棋盘上的绝大多数位置都为空，这将导致大量的冗余运算。所以，我们必须对生成函数进行优化。

根据人们下棋的经验可知，一般情况下，我们只会在棋盘上有子的位置的周围一圈下子。因此，生成函数可以进行优化：遍历棋盘上的点，仅当该点的周围有棋子时，才将其加入着法列表。这个简单的改进极大地提高了效率。代码如下：

```
def generate_2021_4_21(gobang_board):
    # 根据经验可知，下棋的地方一般都会在有子的地方附近，而不会去无子的地方开疆拓土
    # 因此，本函数仅检查有子部分的周围一圈
    # 本函数改进后大大减少了生成的位置数目
    moves = set()
    drct_list = ((-1, -1), (-1, 0), (-1, 1), (0, 1), (1, 1),
```

```
                (1, 0), (1, -1), (0, -1))  # 一个子的周围一圈位置

    for r in range(const.SIDE_LEN):
        for c in range(const.SIDE_LEN):
            if gobang_board[r][c] != const.EMPTY:
                for each_drct in drct_list:
                    dr, dc = each_drct
                    if 0 <= r+dr < const.SIDE_LEN and 0 <= c+dc < const
.SIDE_LEN and gobang_board[r+dr][c+dc] == const.EMPTY:
                        # 若在范围内且为空，则插入 moves 集合
                        moves.add((r+dr, c+dc))

    moves = list(moves)
    return moves
```

# 六、函数性能优化

## 1、评估函数优化

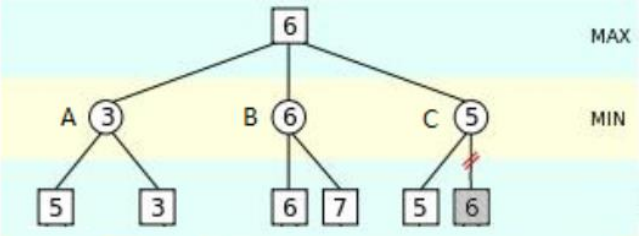
上面的评估函数已经进行了很大的优化，但是还有进一步的改进空间。我们知道，每下完一步棋，棋型都可能改变。而棋型的改变只可能与刚刚下的那个位置有关。因此，我们在每下一颗棋子之前，可以检查该位置的横、纵、左上-右下、右上-左下四条方向，记录棋型数目；再在该位置下子之后，再对该位置的四个方向进行棋型统计。两次统计的差值，就是下子之后新增的棋型数目。

我们再用两个列表记录整个棋盘的棋型数目，每次仅将新增的棋型加到列表上去，不仅可以完成棋型的统计，而且可以极大地简化每次搜索的数目，从而可以极大优化统计效率。

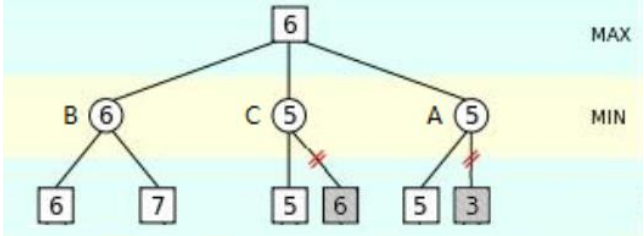
## 2、生成函数优化——启发式评估

前面的生成函数实际上已经进行了一些优化，性能有较大的改进。但是，生成函数还有很大的优化空间。

影响 alpha beta 剪枝效率的关键，是要让评分高的位置更早的被搜索到，这样可以更快的进行剪枝。如下图是剪枝中举的例子，在这个结构下，只有节点 C 的第二个子节点被剪枝了。




如果生成函数生成的节点能够按照近似最优解的顺序排序，那么将使 alpha-beta 剪枝的效率大大提高。因此，我们引入启发式评估的方法。通过启发式评估后，我们可以先预估节点 A，B，C 的评分，假设和实际情况一样，得到评分是节点 B > 节点 C > 节点 A，在生成博弈树时，通过调用子节点的前后顺序，就可以更快的进行剪枝。下图就是上图博弈树重新按照子节点的预估评分进行排序后的结果。可以看到节点 C 和 节点 A 的第二个子节点都被剪枝了，加快了搜索效率。




要实现这一点，就需要对每一个可以下的位置进行评分的预估，让预估分高的位置排在前面。我采用的预估评分方法是：对于一个空的位置，分别下白棋或黑棋，获取这个点四个方向能够形成的棋型，进行打分，再将两分相减，得到该点的最终评分，作为排序的依据。同时，为每个点加上棋盘位置得分，使得在多个评分相同的情况下，AI 优先选择靠近棋盘中央的位置下子。实践证明，该启发式方法的效果很好。代码如下：

```
def evaluate_point(gobang_board, player, rc_pos):
    # 传入一个位置，大概估计该位置适不适合 player 下在此处
    # 判断方法：两个人都在此处下一次，结果取两人之差
    r, c = rc_pos

    gobang_board[r][c] = player  #先在该处下一个子
    count_point_type(gobang_board, rc_pos)   #统计棋型
    score_human, score_ai = cal_score(-player)  #计算得分
    score_1 = score_ai-score_human

    gobang_board[r][c] = -player  #再以对手在该处下一个子
    count_point_type(gobang_board, rc_pos)   #统计棋型
    score_human, score_ai = cal_score(player)  # 计算得分
    score_2 = score_ai-score_human
    gobang_board[r][c] = const.EMPTY
```

```
    return score_1-score_2//2  #计算两分之差。这里进行了一些优化
```

同时，可以设置一个节点数上限。排序后，最多仅选出不超过上限个节点当作可能的着法，那些效果不好的点直接舍弃。这样可以进一步减少生成的可能着发个数。

改进后的生成函数，先找出所有周围有子的空点备用，然后对于每个空点进行启发式评估，计算得分，按照得分进行排序。最终选择出不超过上限个点。伪码如下：

```
def generate(gobang_board, player=const.AI):
    # 本函数在选择节点时加入下子点评估，按照得分对可能位置进行排序，并且仅保留前 LIMIT_GENERATE_NUM 个
    # 评估时需要知道当前是谁下
    LIMIT = const.LIMIT_GENERATE_NUM
    moves = set()

    取得所有周围有子的空点，加入 moves 集合

    moves = list(moves)
    score_and_num_list = []  # 二元组，第一个元素存得分，第二个元素存下标
    for i in range(len(moves)):
        r, c = moves[i]
        # 得分中加入位置得分，便于 ai 尽量往中间走
        score = evaluate.evaluate_point(
            gobang_board, player, (r, c))+const.POS_SCORE[r][c]
        score_and_num_list.append((score, i))

    # 使用极大极小思想对生成位置排序
    if_reverse = True if player == const.MAX_P else False
    score_and_num_list.sort(reverse=if_reverse)   # 按照分数进行排序
    final_moves = []
    for i in range(min(LIMIT, len(score_and_num_list))):    # 保证生成不超过 LIMIT 个
        which = score_and_num_list[i][1]  # 取下标
        final_moves.append(moves[which])
    return final_moves
```

# 七、图形界面以及程序使用方法

图形界面使用 tkinter 库编写，较为简单，编写过程不再赘述。下面介绍程序使用方法：

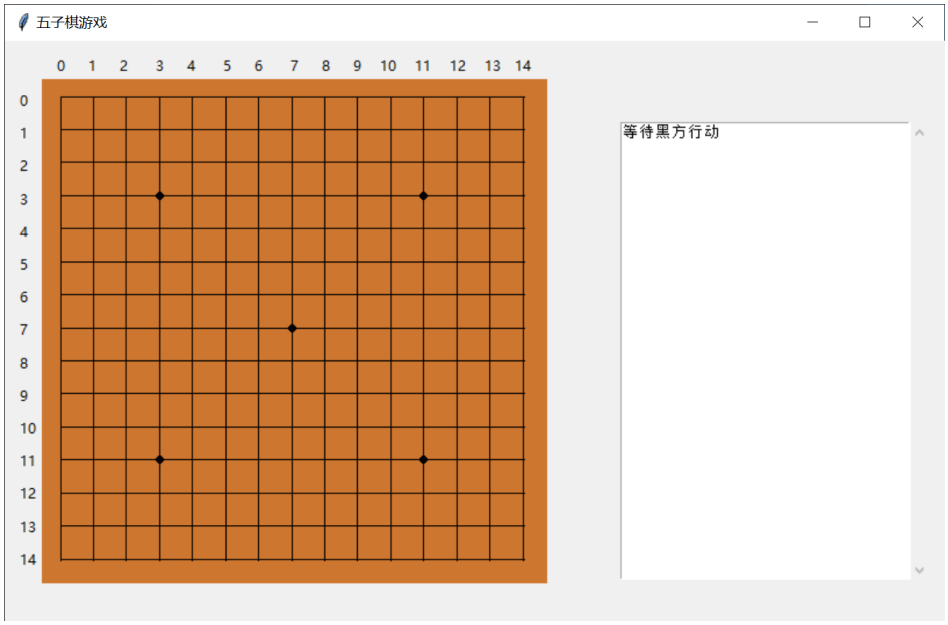
## 1、文件目录结构

文件中共有 5 个 py 文件。功能分别如下：

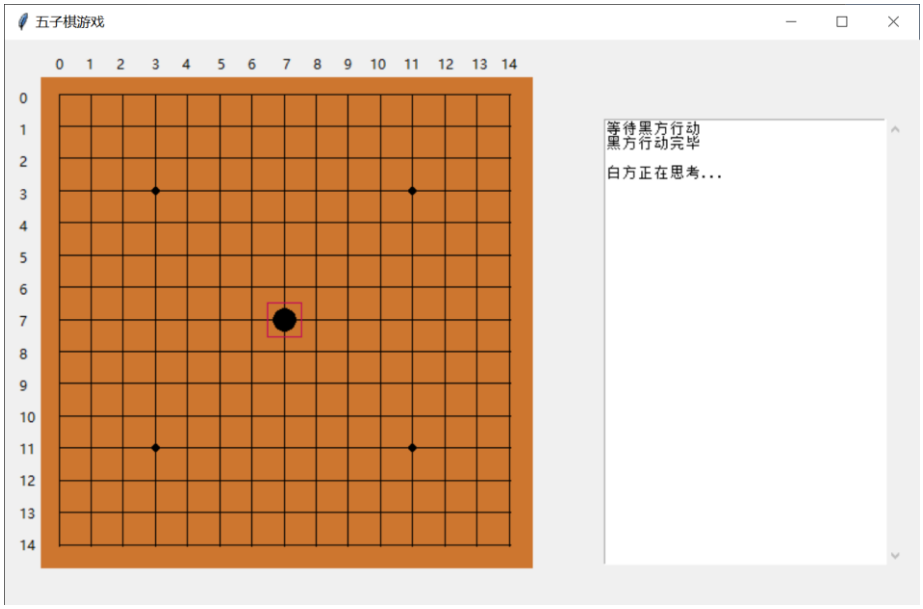
- AI_action.py：主要存放搜索算法，包括极大极小值搜索和 alpha-beta 剪枝算法。
- const.py：存放程序中需要使用到的常量。
- evaluate.py：主要存放评估函数。
- generate.py：主要存放生成下一步可能着发的函数。
- main.py：存放图形界面的所有代码以及 main 函数。**运行程序时，直接运行 main.py 即可。**

## 2、游戏玩法

运行程序后，出现如下窗口：




窗口左边是游戏界面，右边是提示信息。当提示“等待黑方行动”时，我们可以通过点击棋盘上的点进行下棋。

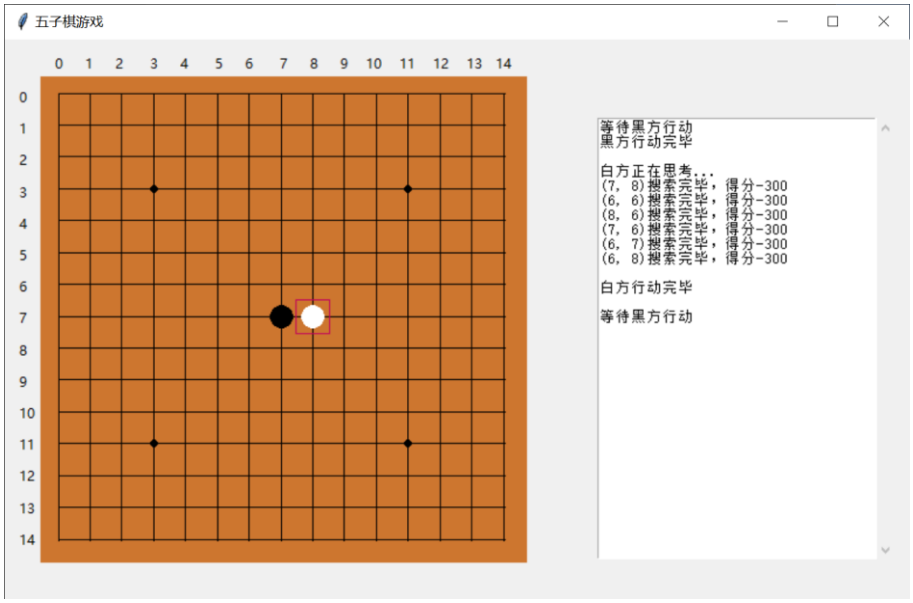


刚下的棋子旁边会有一个红色的边框。黑色行动完毕后，白色（AI）会进行思考，搜索过程中 cmd 界面出现搜索信息。

```
human的棋型: [0, 0, 0, 0, 0, 1, 0]
ai   的棋型: [0, 0, 0, 0, 0, 0, 0]

当前为第 2 层，正在搜索 (5, 7)
α= -300 , β= 100000
当前为第 1 层，正在搜索 (6, 8)
α= -300 , β= 100000
(6, 8) 的分数:  -300
已剪枝 5 次
human的棋型: [0, 0, 0, 0, 0, 1, 0]
ai   的棋型: [0, 0, 0, 0, 0, 0, 0]
```

搜索完成后，白色自动行动一步，并且窗口出现一些提示消息。




当某一方获胜后，棋盘不再能进行点击，并且右侧出现白方获胜的提示。