数据结构与算法课程实习作业报告

马超凡* 吴熙楠 高玮伯 金佳伟 易佳怡

摘要：我们的算法总体策略是通过决策树来分析接下来移动后棋盘的情况，然后通过不同时期的估值函数对棋盘情况进行分析评分，对特殊情况有特殊的处理办法，在决策树中我们使用了多种方法来增加搜索深度。在热身赛上不会出现报错，尽管总体胜率一直不会很高。

关键字：大小，位置，特殊情况

1 算法思想

1.1 总体思路

利用我方对棋盘局面定义之估值函数评估棋盘分数，并使用 Minimax 演算法(最大最小值法)，假定双方都会选择对己方最有利的情况，即为选择对己方分数最高之棋局，使用深度优先搜索，搜索一定深度内的可能移动，在递归调用完成时同时更新节点之分数，每个节点分为终止节点、min 节点和 max 节点，终止节点代表深度达到最大深度之节点，其值即为该局面之分数，min 节点代表此时由敌方做决策，该节点分数为所有子节点分数之最小值，max 节点代表此时由我方做决策，该节点分数为所有子节点分数之最小值。最后在父节点找

出对我方分数最高的移动并输出，同时搭配 alpha-beta 剪枝，并使用启发式规则，在前期考虑落子时不考虑在敌方领域落子，其余时期在 minimax 中若选择落子在敌方领域，要求此时分数需高于操作前之棋局分数，否则不考虑此移动。

前期估值函数（在合成等级为 4 之前定义为前期）：

首先对棋盘内基本棋子赋分求和，并满足等级高一级，分数增加一倍还要多，来达到优先合成等级高的数的目的。同时先手时实现尽量多吃对方；后手时保守一点，保我方发育。总体采用的思路为见缝插针，“吃了就跑”的游击战术。

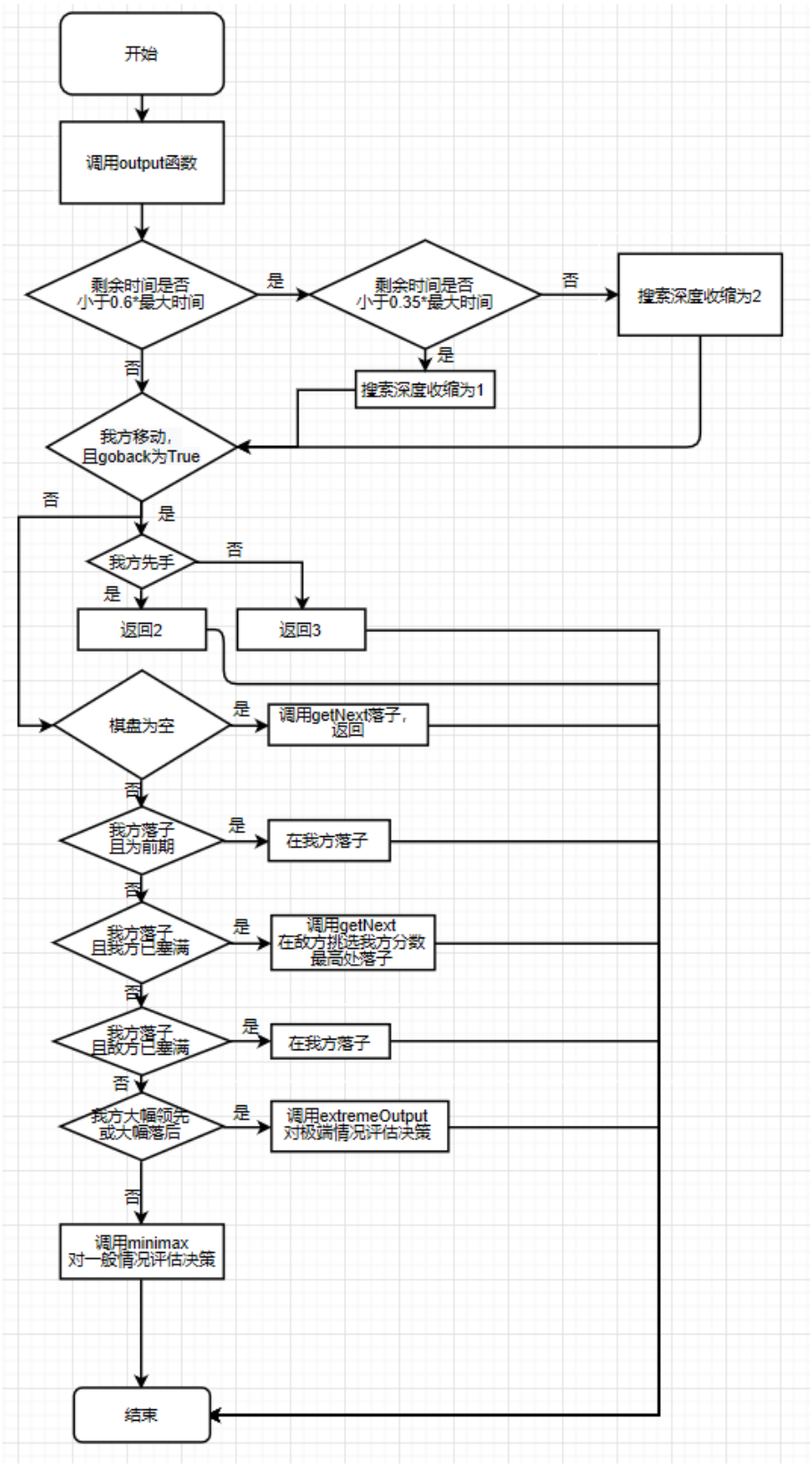
中期估值函数（在合成等级为 4 之后进入中期）：

中期估值函数需要满足以下要求，由棋子等级决定总体上的棋盘得分值，再由位置信息对棋盘得分进行调整。开始我们设想在边角位置养出一个非常大的数字，在距离边界线最远的地盘上按升序排列，摆好阵形来配合最大棋子的合成，因此当时的位置依赖主要是在最边界的那一列为单调排列会获得一定程度的加分，若最大棋子在预定好的位置则获得决定性的加分，如果棋子在对面棋盘则减去相应分数，因为这样会破坏已经形成的规则棋盘，经过实战检验后发现，这样做限制了棋盘 bot 的灵活性，一味防守通常是让对面进攻我们到死，因此我们更改了实现方式。

最终的中期估值函数也是保持原有的基础分方式，在位置依赖关系上进行了调整，改为如果棋子附近存在与他大小相近的棋子（只看比他大的最小的和比他小的最大的，这两个）那么会获得一定的加分，并且棋子等级越高加分越高，如果差距过大会失去分数。另外，如果我方棋子有被对面棋子吃掉的风险，那么也会失去很多分数。最后，为了实现阻碍对方合成的落子，还有一部分是遍历对面棋子所有等级为一的，如果他附近有许多等级比他高的，那么棋盘得分也会有所增加。

特殊情况函数，考虑三类较特殊情况：①我们处于领先优势的时候（然而似乎这个很少被用到），比如我方率先合成 7 级以上的子，且对方未合成与我们同样大小的子，我们将转移视线开始扰乱对面合成。②我们处于落后劣势的时候，比如我方落后对面两级以上的时候，我们将会抓住时机往对面塞不容易合成可以干扰对面的子，使得对方快速达到不能走的状态，然后乘机发育自己。③当我们的落子不太妙的时候，并且刚好有一个 2 落在边界线上，我们考虑往对面落一个 2，然后下一步移动吃掉，再返回。

1.2 算法流程图


开始
调用output函数
剩余时间是否
小于0.6*最大时间
是
剩余时间是否
小于0.35*最大时间
否
搜索深度收缩为2
否
是
搜索深度收缩为1
我方移动，
且goback为True
否
是
我方先手
否
是
返回2
返回3
棋盘为空
是
调用getNext落子，
返回
否
我方落子
且为前期
是
在我方落子
否
我方落子
且我方已塞满
是
调用getNext
在敌方挑选我方分数
最高处落子
否
我方落子
且敌方已塞满
是
在我方落子
否
我方大幅领先
或大幅落后
是
调用extremeOutput
对极端情况评估决策
否
调用minimax
对一般情况评估决策
结束

1.3 算法运行时间复杂度分析

设棋盘共 r 行，c 列：

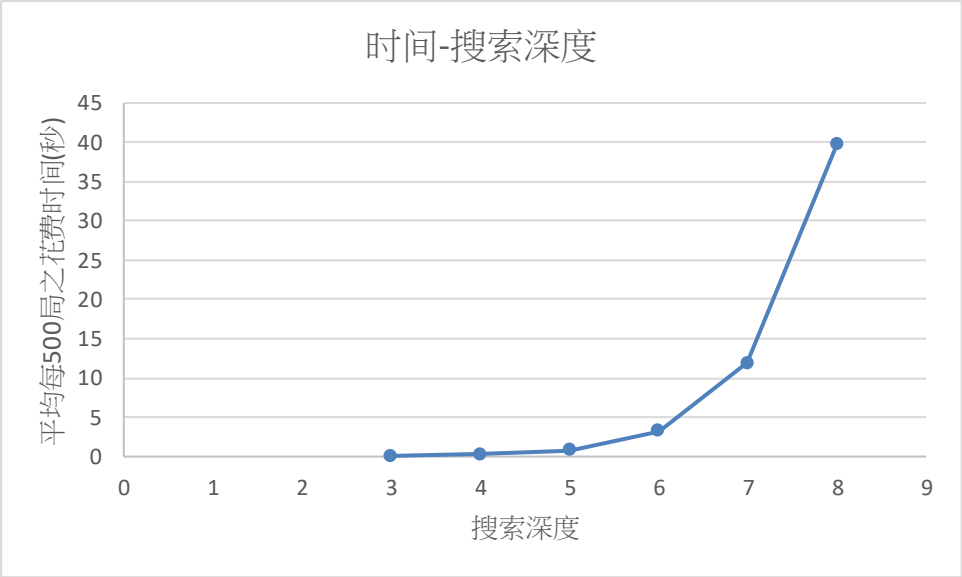
基础分计算的复杂度为 O（r*c），遍历棋子得到位置关系的复杂度为 O（r*c），遍历对方棋子部分的复杂度为 O（r*c），判断是否被吃的部分复杂度为（r*c）。

故总的算法复杂度为 O（r*c），但是在实际开销中，由于循环较多，计算步骤较复杂，时间开销较大，这限制了决策树的深度，但是里面关于一些位置依赖的算法相当于一些更大的基础分的征兆，一定意义上可以理解为增加决策树的深度。

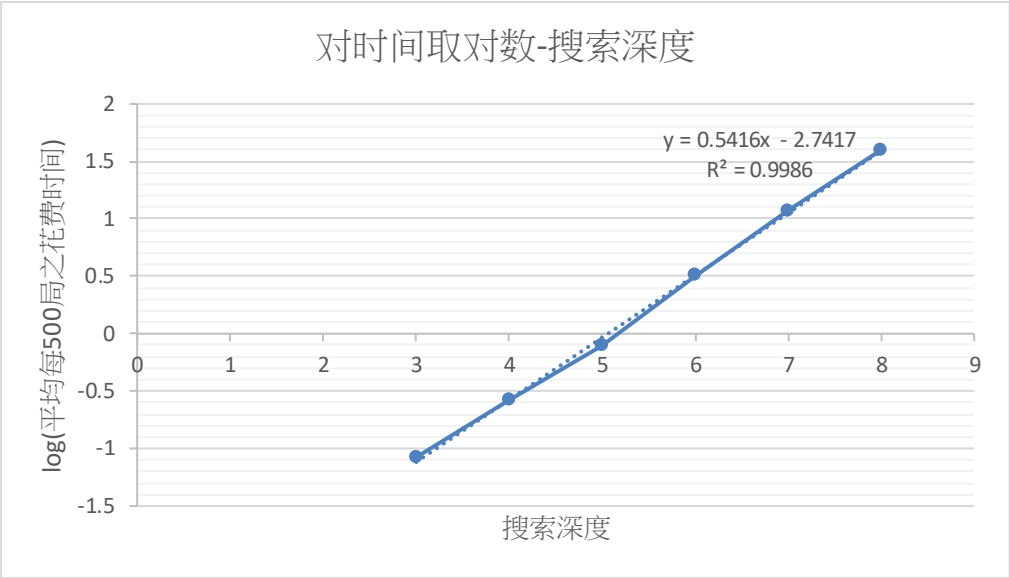
Minimax 演算法加上 alpha-beta 剪枝和启发式规则减少考虑在敌方领域落子之时间复杂度:

若无剪枝，则 Minimax 演算法时间复杂度与深度 d 为指数关系，约为 O(e^d)，因为每个子节点大约都有数量接近的子节点。考虑剪枝后，情况较为复杂，因此使用实验来评估。在本地使用非常简单为 O(1)的计分函数来评估棋盘，并利用此演算法对不同棋局重复运行多次且计算花费之平均时间以得出对搜索深度的时间复杂度。

搜索深度	3	4	5	6	7	8
平均每 500 局之花费时间(秒)	0.084	0.2625	0.7879	3.222	11.9	39.75



搜索深度	3	4	5	6	7	8
log(平均每 500 局之花费时间)	-1.07521	-0.58087	-0.10352	0.508224	1.075547	1.599337



由以上图表可看出，花费时间与深度为指数关系，若深度为 d，时间复杂度约为 O(e^0.54d)，经过有效地剪枝后，时间复查度由未剪枝前的 O(e^d)优化为 O(e^0.54d)

2 程序代码说明

2.1 数据结构说明

Minimax 演算法采用树的基本数据结构实现，树的基本单位为节点 Node，并自定义了一个类 Node。

相较于课上所学的树，此节点上存储信息更多，并且可链接多个子节点，以棋局做为标志 key，将深度、mode(下棋或移动)、父节点、子节点和 min 或 max 节点均存储于该节点，以方便调用，该节点的深度若为最大深度则该节点为终止节点，其值为该棋盘的分数，否则试其为 min 或 max 节点将值设为正或负无穷，同时存储其 alpha 值和 beta 值以利 alpha-beta 剪枝。

```
43      def addChild(self, key):  # 添加子节点
44          if key not in self.child:
45              self.child.append(key)
46
47      def updateValue(self, newval):
48          self.value = newval
49
50      def updateAlpha(self, newAlpha):
51          self.alpha = newAlpha
52
53      def updateBeta(self, newBeta):
54          self.beta = newBeta
```

```python
class Node:  # 树的节点
    def __init__(self, key, minormax: bool, depth, mode, parent, isFirst, early, searchDepth):  # minormax = True 表示min
        self.key = key  # 棋盘作为key
        self.depth = depth  # 深度
        self.mode = mode
        self.parent = parent
        self.isFirst = isFirst
        if self.depth == searchDepth:  # 到达最大深度则返回棋盘分数
            self.value = chessboardPoint(key, isFirst, early)
        else:  # 还没到最大深度 若为min 先令此数为正无穷，max为负无穷，随者搜索过程调整
            self.value = infinity if minormax else -1 * infinity
        self.minormax = minormax  # 我方操作则max(False) 敌方操作则min(True)
        self.child = []
        self.alpha = -1 * infinity  # alpha 起始为负无穷
        self.beta = infinity  # beta 起始为正无穷
```

Player 类：

在 Player 类中，除了先后手及随机序列为，增加了搜索树最大深度和 goback，goback 为 True 代表上一局向中间移动攻击对手，这一局需退回来以确保安全。

```python
class Player:
    def __init__(self, isFirst: bool, array) -> None:
        self.isFirst = isFirst
        self.array = array
        self.searchDepth = 3  # 树的搜索深度 (start from 0) (可再调整)
        self.goback = False
```

2.2 函数说明

possibleMove 函数：

①在 Node 类底下定义此函数，输入当前回合数、先后手和是否为前期，输出一个列表

②功能：输出所有需要考虑的下一步。

③具体操作为：若 mode 为移动，则考虑四个方向是否能移动，若能移动则添加移动后棋局至列表，最后输出列表。若 mode 为落子，分前期和非前期讨论，为前期则输出列表仅有落子在己方后的棋盘，非前期时，如果能在己方落子，将落子在己方后的棋盘添加至列表后，考虑敌方所有空位，若落子在该空位能使棋局分数比落子前高，则将落子在该空位后的棋局添加至列表，最后输出列表。

```
def possibleMove(self, currentRound, isFirst, early) -> list:  # 输出下一步的所有可能
    output = []
    if self.mode == "position":  # 下棋
        if self.minormax:  # 敌方下棋
            if early:  # 前期只考虑在自己的领域下棋
                copyBoard = self.key.copy()
                if len(copyBoard.getNone(not isFirst)) == 0:
                    pass
                else:
                    copyBoard.add(not isFirst, self.key.getNext(not isFirst, currentRound))
                    output.append(copyBoard)
                return output

            elif self.key.getNext(not isFirst, currentRound) != ():  # 若敌方可在敌方领域下棋则添加
                copyBoard = self.key.copy()  # 复制棋盘后添加下一步位置
                copyBoard.add(not isFirst, self.key.getNext(not isFirst, currentRound))
                output.append(copyBoard)
            for position in self.key.getNone(isFirst):  # 我方空位
                copyBoard = self.key.copy()
                copyBoard.add(isFirst, position)
                if chessboardPoint(board=copyBoard, isFirst=not isFirst, early=early) > chessboardPoint(
                        board=self.key, isFirst=not isFirst, early=early):
                    output.append(copyBoard)  # 敌方在我方空位落子 分数要上升才考虑
```

```
else:  # 我方下棋
    if early is True:  # 前期只考虑在自己的领域下棋
        copyBoard = self.key.copy()  # 复制棋盘后添加下一步位置
        copyBoard.add(isFirst, self.key.getNext(isFirst, currentRound))
        output.append(copyBoard)
        return output

    elif self.key.getNext(isFirst, currentRound) != ():  # 若我方可在我方领域下棋则添加
        copyBoard = self.key.copy()  # 复制棋盘后添加下一步位置
        copyBoard.add(isFirst, self.key.getNext(isFirst, currentRound))
        output.append(copyBoard)
    for position in self.key.getNone(not isFirst):  # 敌方空位
        copyBoard = self.key.copy()
        copyBoard.add(not isFirst, position)
        if chessboardPoint(board=copyBoard, isFirst=isFirst, early=early) > chessboardPoint(board=self.key,
                                                                                            isFirst=isFirst,
                                                                                            early=early):

            output.append(copyBoard)  # 我方在敌方空位落子 分数要上升才考虑
        output.append(copyBoard)
```

```
else:  # 移动
    if self.minormax:  # 敌方移动
        for direction in range(4):  # 上下左右
            copyBoard = self.key.copy()
            if copyBoard.move(not isFirst, direction):  # 如果可移动 则添加到output
                output.append(copyBoard)
    else:  # 我方移动
        for direction in range(4):  # 上下左右
            copyBoard = self.key.copy()
            if copyBoard.move(isFirst, direction):  # 如果可移动 则添加到output
                output.append(copyBoard)
return output
```

newmode 函数：

①输入 mode、我方先后手、minormax(相当于我方或敌方下棋)，输出字串 direction 或 position 代表下一步之模式。

②功能：返回下轮决策是落子或移动。

③具体操作为：若 mode、我方先后手、minormax 处于某些状态则返回 direction，反之则返回 position。

```
122    def newmode(mode, isFirst, minormax) -> str:  # 返回下一模式
123        if (mode, isFirst, minormax) in [('position', True, True), ("direction", True, False),
124                                         ("direction", False, True), ('position', False, False)]:
125            return "direction"
126        else:
127            return 'position'
```

minimax 函数：

①输入棋局、当前回合数、minormax(相当于我方或敌方下棋)、mode、先后手、父节点、当前深度、是否为前期和最大搜索深度，输出决策(移动方向或落子位置)。

②功能：利用 minimax 演算法搭配剪枝，返回对我方最优的决策。

③具体操作为：利用递归调用实现深度优先搜索，若深度为最大搜索深度，则为终止节点，为递归函数结束条件，则以该棋局为 node 新增节点，并返回分数。若深度为 0 即创建父节点，考虑所有移动或落子可能(排除落子在敌方后分数随即降低之情况)，对所有可能之棋局调用 minimax 函数后，找出输出分数最高的操作，输出该操作。若深度不等于 0 或最大搜索深度，则先创立以该棋局为 key 之节点并继承父节点之 alpha 和 beta 值，若此时没有下一步可移动或落子则这一步不操作，那么直接对此棋局调用 minimax 函数，但是改为另一方动作，并更新此节点之 alpha 或 beta 值为该 minimax 函数之输出值并输出。若可操作，则逐一进行可能操作，并对操作后棋局调用 minimax 函数，将此节点之 alpha 或 beta 值(min 节点对应 beta，max 节点对应 alpha)更新为 minimax 函数之输出值，若 beta 值小于 alpha 值，则停止循环，循环结束后输出 alpha 或 beta 值(即为该节点之分数值)。

```
def minimax(nowBoard, currentRound, minormax: bool, mode, isFirst, parent, depth,
            early, searchDepth):  # minimax tree, depth starts from 0
    newRound = currentRound + 1 if newmode(mode, isFirst,
                                           minormax) == 'position' and mode is not 'position' else currentRound
    # 计算下一轮是否要加1
    if depth == searchDepth:  # 搜到底层 直接返回棋盘分数
        node = Node(key=nowBoard, depth=depth, minormax=minormax, mode=mode, parent=parent, isFirst=isFirst,
                    early=early, searchDepth=searchDepth)
        if minormax:  # 本步敌方移动，父节点为max，更新beta
            node.updateBeta(node.value)
        else:  # 本步己方移动，父节点为min，更新alpha
            node.updateAlpha(node.value)
        return node.value

    elif depth == 0:  # 顶层节点  为了方便决策 直接返回下旗位置或移动方向
        point = -1 * infinity
        output = ()
        node = Node(key=nowBoard, depth=0, minormax=minormax, mode=mode, parent=None, isFirst=isFirst,
                    early=early, searchDepth=searchDepth)  # 先制造节点


    elif depth == 0:  # 顶层节点  为了方便决策 直接返回下旗位置或移动方向
        point = -1 * infinity
        output = ()
        node = Node(key=nowBoard, depth=0, minormax=minormax, mode=mode, parent=None, isFirst=isFirst,
                    early=early, searchDepth=searchDepth)  # 先制造节点
        if mode == 'position':  # 我方下棋
            if node.key.getNext(isFirst, currentRound) != ():  # 若我方可在我方领域下棋则添加
                copyBoard = nowBoard.copy()  # 复制棋盘后添加下一步位置
                copyBoard.add(isFirst, node.key.getNext(isFirst, currentRound))
                point = minimax(nowBoard=copyBoard, currentRound=newRound, minormax=not minormax, isFirst=isFirst,
                                mode=newmode(mode, isFirst, minormax), depth=1, parent=node, early=early,
                                searchDepth=searchDepth)
                output = node.key.getNext(isFirst, currentRound)

            for position in nowBoard.getNone(not isFirst):  # 敌方空位
                copyBoard = nowBoard.copy()
                copyBoard.add(not isFirst, position)
                if chessboardPoint(board=copyBoard, isFirst=isFirst, early=early) > chessboardPoint(nowBoard,
                                                                                                    isFirst, early):
                    # 我方在敌方空位落子 分数要上升才考虑

                    if minimax(nowBoard=copyBoard, currentRound=newRound, minormax=not minormax, isFirst=isFirst,
                               mode=newmode(mode, isFirst, minormax), depth=1, parent=node,
                               early=early, searchDepth=searchDepth) > point:
                        point = minimax(nowBoard=copyBoard, currentRound=newRound, minormax=not minormax,
                                        isFirst=isFirst,
                                        mode=newmode(mode, isFirst, minormax), depth=1, parent=node,
                                        early=early, searchDepth=searchDepth)
                        output = position
            return output

        else:  # 我方移动
            if len(node.possibleMove(currentRound=currentRound, early=early, isFirst=isFirst)) == 1:  # 只有一个可能移动方向
                for direction in range(4):
                    copyBoard = nowBoard.copy()
                    if copyBoard.move(isFirst, direction):
                        return direction
```

```
        for direction in range(4):  # 上下左右
            copyBoard = nowBoard.copy()
            if copyBoard.move(isFirst, direction) and minimax(nowBoard=copyBoard, currentRound=newRound,
                                                              minormax=not minormax, isFirst=isFirst,
                                                              mode=newmode(mode, isFirst, minormax), depth=1,
                                                              parent=node,
                                                              early=early, searchDepth=searchDepth) > point:
                point = minimax(nowBoard=copyBoard, currentRound=newRound, minormax=not minormax, isFirst=isFirst,
                                mode=newmode(mode, isFirst, minormax), depth=1, parent=node,
                                early=early, searchDepth=searchDepth)
                output = direction  # 添加进字典
        return output
```

```
    else:  # 不是树的底层或顶层,当我方操作（max）时返回下界alpha，反之返回上界beta
        node = Node(key=nowBoard, depth=depth, minormax=minormax, mode=mode, parent=parent, isFirst=isFirst,
                    early=early, searchDepth=searchDepth)  # 先制造节点
        node.updateAlpha(node.parent.alpha)
        node.updateBeta(node.parent.beta)
        if len(node.possibleMove(currentRound=currentRound, isFirst=isFirst, early=early)) is not 0:  # 如果有下一步 添加到子节点
            if minormax:  # min,返回beta值
                for i in node.possibleMove(currentRound=currentRound, isFirst=isFirst, early=early):
                    node.updateBeta(min(node.beta, minimax(nowBoard=i, depth=depth + 1, minormax=not minormax,
                                                           mode=newmode(mode, isFirst, minormax), isFirst=isFirst,
                                                           currentRound=newRound, parent=node, early=early,
                                                           searchDepth=searchDepth)))
                    if node.beta <= node.alpha:  # beta小于等于alpha时剪枝
                        break
                return node.beta
            else:  # max，返回alpha值
                for i in node.possibleMove(currentRound=currentRound, isFirst=isFirst, early=early):
                    node.updateAlpha(max(node.alpha, minimax(nowBoard=i, depth=depth + 1, minormax=not minormax,
                                                             mode=newmode(mode, isFirst, minormax), isFirst=isFirst,
                                                             currentRound=newRound, parent=node, early=early,
                                                             searchDepth=searchDepth)))
                    if node.beta <= node.alpha:
                        break
                return node.alpha
        else:  # 没有下一步
            if minormax:  # min
                node.updateBeta(minimax(nowBoard=nowBoard.copy(), depth=depth + 1, currentRound=newRound,
                                        minormax=not minormax, mode=newmode(mode, isFirst, minormax), isFirst=isFirst,
                                        parent=node, early=early, searchDepth=searchDepth))
                return node.beta
            else:  # max
                node.updateAlpha(minimax(nowBoard=nowBoard.copy(), depth=depth + 1, currentRound=newRound,
                                         minormax=not minormax, mode=newmode(mode, isFirst, minormax), isFirst=isFirst,
                                         parent=node, early=early, searchDepth=searchDepth))
                return node.alpha
```

前期：

两个辅助函数：

Ⅰ，getourpower 函数：参数为当前棋盘状况 board，是否先手 isFirst，以及行数 row。功能：找出第 row 行我方最靠近对方的数。实现方法：由于前期棋盘状况较简单，并且采用见缝插针，吃了就跑的战术，简化了计算量，从对方分界线的一列开始遍历，利用 board.getBelong 函数，返回第一个我方棋子。称为“攻击棋”

Ⅱ，getdanger 函数：与 getourpower 函数相反来求第 row 行对方最靠近我方的数。详情参

见Ⅰ。得到危险棋。

Ⅲ，评价棋盘函数：ee（early evaluate）：输入棋盘 board，与 isFirst 先后手，返回一个得分（int）。

具体实现：遍历场面上所有我方棋子，累加（2**等级 - 1），这样赋分可以实现等级高一级，分数增加不止一倍，此为基本分。利用前面的两个辅助函数，在我方攻击棋有机会吞并对方处于分界线上的棋子时，额外加上对应的基本分；在我方已经吞并了对方棋子时，即对方棋盘上（主要是分界线上）有我方的棋子时，额外加上 2 倍的对应基本分。反过来危险的时候，即对方达成上述条件时，额外扣相应的分。另外，考虑我方最大等级的棋子是否在棋盘的角落处，如果是，则额外加基本分的 3/4，来达到尽量进入中期时提供好一点的发育环境。

```
def getourpower(board, isFirst, row):
    board1 = board.copy()
    if isFirst:
        for i in range(5):
            if board1.getValue((row, 4 - i)) != 0 and board1.getBelong((row, 4 - i)):
                return board1.getValue((row, 4 - i))
        else:
            return 0
    else:
        for i in range(5):
            if board1.getValue((row, 3 + i)) != 0 and not board1.getBelong((row, 3 + i) ):
                return board1.getValue((row, 3 + i))
        else:
            return 0


def getdanger(board, isFirst, row):
    board1 = board.copy()
    if not isFirst:
        for i in range(5):
            if board1.getValue((row, 4 - i)) != 0 and board1.getBelong((row, 4 - i)):
                return board1.getValue((row, 4 - i))
        else:
            return 0
    else:
        for i in range(5):
            if board1.getValue((row, 3 + i)) != 0 and not board1.getBelong((row, 3 + i)):
                return board1.getValue((row, 3 + i))
        else:
            return 0
```

```
def ee(board, isFirst):  # early evaluate
    score = 0
    lst1 = board.getScore(isFirst)
    len1 = len(lst1)
    for i in range(len1):
        score += 2 ** lst1[i] - 1
    else:
        if isFirst:
            # 先只考虑是否进攻
            for i in range(4):
                if getourpower(board, isFirst, i) != 0 and getourpower(board, isFirst, i) == board.getValue((i, 4)) and board.getValue((i,3)) == 0 and not board.getBelong((i,4)):
                    score += 2**getourpower(board, isFirst, i) - 1
                if (board.getBelong((i,4)) or board.getValue((i,4)) == 0) and getourpower(board, isFirst, i) == getdanger(board, isFirst, i) and (board.getValue((i,3)) !=0 or board.getBelong((i,4))):
                    score -= 2**getdanger(board, isFirst, i) - 1
                if board.getBelong((i, 4)) and board.getValue((i,4)) != getdanger(board, isFirst, i):
                    score += 2**board.getValue((i, 4)) - 1
                if not board.getBelong((i, 3)) and board.getValue((i,3)) != getourpower(board, isFirst, i):
                    score -= 2 * (2**board.getValue((i, 3)) - 1)
                if not board.getBelong((i, 2)):
                    score -= 2 * (2**board.getValue((i, 2)) - 1)
            if (board.getValue((0,0)) == lst1[len1 - 1] or board.getValue((3,0)) == lst1[len1 - 1]) and lst1[len1 - 1] > 1:
                score += 2**lst1[len1 - 1] / 4 * 3 - 1
        else:  # 后手，保守发育
            for i in range(4):
                if getourpower(board, isFirst, i) != 0 and getourpower(board, isFirst, i) == board.getValue((i, 3)) and board.getValue((i,4)) == 0 and not board.getBelong((i,3)):
                    score += 2**getourpower(board, isFirst, i) - 1
                if (!board.getBelong((i,3)) or board.getValue((i,3)) == 0) and getourpower(board, isFirst, i) == getdanger(board, isFirst, i) and (board.getValue((i,4)) != 0 or !board.getBelong((i,3))):
                    score -= 2**getdanger(board, isFirst, i) - 1
                if board.getBelong((i, 4)) and board.getValue((i,4)) != getourpower(board, isFirst, i):
                    score -= 2 * (2**board.getValue((i, 4)) - 1)
                if board.getBelong((i, 5)):
                    score -= 2 * (2**board.getValue((i, 5)) - 1)
                if not board.getBelong((i, 3)) and board.getValue((i,3)) != getdanger(board, isFirst, i):
                    score += 2**board.getValue((i, 3)) - 1
            if (board.getValue((0,7)) == lst1[len1 - 1] or board.getValue((3,7)) == lst1[len1 - 1]) and lst1[len1 - 1] > 1:
                score += 2**lst1[len1 - 1] / 4 * 3 - 1
        return score
```

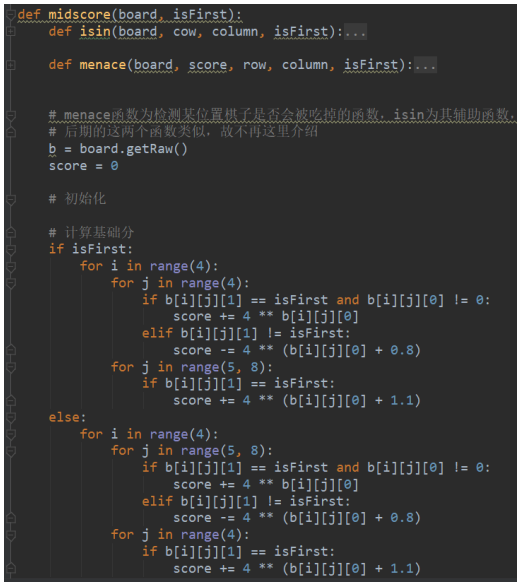
中期估值函数：

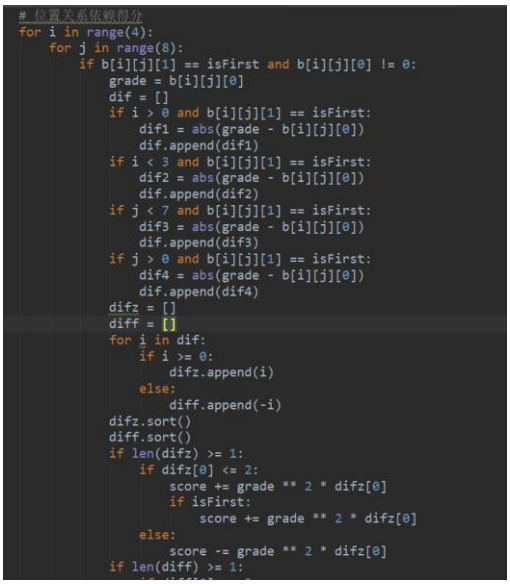
①输入棋盘 Chessboard，与 isFirst 先后手，返回一个得分（int）。

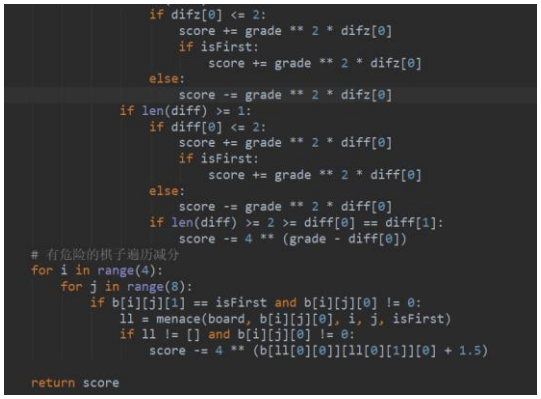
②功能：对棋盘的场面局势进行分析，对 isFirst 更有利则返回更高的分数。

算法策略：遍历棋盘，分析处理棋盘。中期估值函数通过对数字得分的精妙设计，实现了优先和成大子，其次注意单调性排序，注意防止我方棋子被吃，和在对方棋盘落子能取得一定优势后选择向对方落子。

③具体操作为：遍历场面上所有我方棋子，把 4**等级数（以确保优先合成等级高的棋子） 加起来得到基础分。如果棋子在对方棋盘获得则计算时等级提升 1.1 并且加 40 分（在不耽误大子合成的情况下，优先向对面棋盘落子）。然后遍历每个棋子四周比他大的最小的和比他小的最大的，若和他本身的相差小于等于二，加上他本身一半的等级分，否则减去他本身一半的等级分（在不耽误大子合成的情况下，优先棋子单调排列）。遍历场面上所有对方等级为 1 的棋子，若四周平均等级比他高 2，则总得分加上 64。（不耽误大子合成的情况下在可以造成一定阻碍时向对面棋盘落子）遍历场面上所有我方棋子，通过 menace 函数检测他是否有被吃掉的危险，如果有则减去他的等级分。（在不耽误大子合成的情况下避免我方棋子被吃掉）返回经过这一系列操作之后的总得分，即为中期棋盘估值。







特殊情况函数：

①isin 函数：判断某棋子是否处于己方的棋盘内，输入棋子位置，输出 bool 值。

②menace 函数：判断此处的我方棋子是否有被敌方棋子吃掉的危险，输入棋子的级别和

位置，若有危险，返回威胁棋子位置组成列表[[row,column]]；否则返回[]。

③ast 函数：为确保函数在遍历的时候不会出现行数或者列数超出范围的情况，进行检验，返回 bool 值。

④search 函数：给定搜索级别值，从给定行列出发逐行搜索，返回搜索到第一个符合的棋子位置(row,column)，否则返回空的元组。其中使用了 ast 函数来确保搜索不会超出范围。

⑤trouble1 函数：给对面捣乱的函数，在领先情况下使用，若对方存在两个较大级别的棋子快要合并的情况，在其中一个棋子旁边插入一个 2 搅乱计划，返回插入 2 的位置(cow,column)。其中使用 search 函数以及 menace 函数来确保搜寻结果以及不会被反吃的情况。

⑥trouble2 函数：同样是给对面捣乱的函数，在落后情况下使用，在 trouble1 可行的情况下优先使用 trouble1,否则，对于落后情况尽量牵制对方，往对方扔 2，返回 2 的位置(cow,column)，尽量达到将对面堵死然后发育自己的计划。

⑦search2 函数：给定某棋子的位置以及级别，查看周围是否有可合并项，返回可合并项位置及方向组成的列表[[cow,column,diraction]]。

⑧total 函数：给定棋子位置，返回除开空位周围四个棋子的级别之和。

⑨chance1 函数：对于可能存在的对面棋子位于边界线并且刚好可以被我方棋子吃掉不被反吃的情况，返回方向 choice;否则返回 choice=-1。其中调用了 menace 函数来确保不会有被反吃的情况。

⑩chance2 函数：若我方为先手且我方有 2 落在边界线上，在对面空位走个 2 且合并后不会被对方吃掉的情况，返回落子位置和移动方向[(row,column),diraction]。

⑪extremeOutput 函数：对于落后领先的特殊情况特殊处理，先判断是否满足领先以及可以使用 chance2 的情况，如果是，则打包带走落子与移动方向;如果不是的话，那么就判断是否为领先情况，如果是且可以使用 trouble1 则调用，否则调用中期函数，落后情况调用 trouble2 同理，其余情况还是按照一般情况处理。

```
def extremeOutput(self, currentRound, board, mode):
    z=True
    alist = board.getScore(self.isFirst)
    yourLlist = board.getScore(not self.isFirst)  # 敌方序列
    if alist[-1] >= 7 and yourLlist[-1] < alist[-1] and self.chance2(board)!=[]:
        pass
    else:
        z=False
    if not z:
        if mode == 'position':  # 给出己方下棋的位置
            if len(yourLlist) is not 0 and (alist[-1] + 2 <= yourLlist[-1]) and self.trouble2(board)!=():  # 大幅落后
                return self.trouble2(board)
            elif len(yourLlist) is not 0 and alist[-1] >= 8 and yourLlist[-1] < alist[-1] and self.trouble1(board)!=():  # 大幅领先 可以考虑7或8
                return self.trouble1(board)
            else:
                return minimax(nowBoard=board, currentRound=currentRound, minormax=False, mode=mode,
                               isFirst=self.isFirst, parent=None, depth=0, early=False,searchDepth=self.searchDepth)
        elif mode == 'direction':  # 给出己方合并的方向
            if alist[-1] >= 7 and yourLlist[-1] < alist[-1] and int(self.chance1(board)) in [0,1,2,3]:
                if board.move(self.isFirst,self.chance1(board)):
                    return 3
                else:
                    return minimax(nowBoard=board, currentRound=currentRound, minormax=False, mode=mode,
                                   isFirst=self.isFirst, parent=None, depth=0, early=False,searchDepth=self.searchDepth)
            else:
                return minimax(nowBoard=board, currentRound=currentRound, minormax=False, mode=mode,
                               isFirst=self.isFirst, parent=None, depth=0, early=False,searchDepth=self.searchDepth)
    else:
        if mode=='position':
            return self.chance2(board)[0]
```

```
            return int(self.chance2(board))[1]
        else:
            return None
# 判断棋子是否为我方棋子，返回bool
def isin(self, board, cow, column):
    if self.isFirst:
        return board.getBelong((cow, column))
    else:
        return not board.getBelong((cow, column))

# 判断此处的我方棋子是否有被敌方棋子吃掉的危险，若有危险，返回威胁棋子位置组成列表[[row,column]]；否则返回[]
def menace(self, board, score, row, column):
    l = []
    d = 0
    x=board.getDecision(not self.isFirst)
    while d < 4:
        a = row
        b = column
        s = 0
        if d == 0 and row != 0:  # 我方棋子不处于第一行，向上搜索
            # 若搜索到空位且未达到顶端且此处棋子还属于敌方，则继续搜索
            while s == 0 and a != 0 and not self.isin(board, a-1, b):
                a -= 1
                if x!=(a,b):
                    s = board.getValue((a, b))
                else:
                    s=1
            if score == s:
                l.append([a, b])
            else:
                ----
            # 若搜索到空位且未达到底端且棋子还属于敌方，则继续搜索
            while s == 0 and a != 3 and not self.isin(board, a+1, b):
                a += 1
                if x!=(a,b):
                    s = board.getValue((a, b))
                else:
                    s=1
            if score == s:
                l.append([a, b])
            else:
                pass
        elif d == 2 and column != 0:  # 我方棋子不属于第一列，向左搜索
            # 若搜索到空位且未达到最左端且棋子还属于敌方，则继续搜索
            while s == 0 and b != 0 and not self.isin(board, a, b-1):
                b -= 1
                if x!=(a,b):
                    s = board.getValue((a, b))
                else:
                    s=1
            if score == s:
                l.append([a, b])
            else:
                pass
        elif d == 3 and column != 7:  # 我方棋子不属于第八列，向右搜索
            # 若搜索到空位且未达到最右端且棋子还属于敌方，则继续搜索
            while s == 0 and b != 7 and not self.isin(board, a, b+1):
                b += 1
                if x!=(a,b):
                    s = board.getValue((a, b))
                else:

            if score == s:
                l.append([a, b])
            else:
                pass
        else:
            pass
        d += 1
    return l

def ast(self,board,cow,column):
    column+=1
    if self.isFirst:
        if int(column % 4 + 4)<8 and \
                int(column % 4 + 4)>=4 and int(cow - 1 + column / 4)>=0 and int(cow - 1 + column / 4)<4:
            return True
        else:
            return False
    else:
        if int(column % 4 )<4 and \
                int(column % 4 )>=0 and int(cow  + column / 4)>=0 and int(cow + column / 4)<4:
            return True
        else:
            return False

# 给定搜索级别值，从给定行列出发逐行搜索，返回搜索到第一个符合的棋子位置(row,column)
def search(self, board, score, cow, column):
    s = ()
    while s == () and self.ast(board,cow,column):
        if self.isFirst:
            column += 1
            if board.getValue((int(cow - 1 + column / 4), int(column % 4 + 4))) == score:
                s = (int(cow - 1 + column / 4), int(column % 4 + 4))
            else:
                pass
        else:
            column += 1
            if board.getValue((int(cow + column / 4), int(column % 4))) == score:
                s = (int(cow + column / 4), int(column % 4))
            else:
                pass
    return s

# 若对方存在两个较大级别的棋子快要合并的情况，在其中一个棋子旁边插入一个2搅乱计划，返回插入2的位置(cow,column)
def trouble1(self, board):
    s = max(board.getScore(not self.isFirst))  # 取得敌方棋子级别的最大值
    l = []
    a = ()
    k = []
    b = ()
    c = []
    # 搜索包含级别至少为3
    while s > 2 and l == []:
        if self.isFirst and (s in board.getScore(not self.isFirst)) and self.search(board, s, 0, 4)!=():
            a = self.search(board, s, 0, 4)  # 搜索满足条件的第一个棋子
            l = self.menace(board, s, a[0], a[1])  # 查看附近是否可能存在可以合并的棋子
        elif not self.isFirst and (s in board.getScore(not self.isFirst)) and self.search(board, a, 0, 0)!=():
            a = self.search(board, a, 0, 0)  # 搜索满足条件的第一个棋子
            l = self.menace(board, s, a[0], a[1])  # 查看附近是否可能存在可以合并的棋子
        else:
            pass
```

```
            pass
        s -= 1
    if len(l)>0:
        for i in l:
            if abs(i[0] - a[0]) > 1 or abs(i[1] - a[1]) > 1:  # 查看这两个棋子之间是否有空位
                k.append(i)
            else:
                pass
    else:
        k=[]
    if k == []:
        b = ()
    else:
        c = k[0]
        if c[0] == a[0]:
            if c[1] > a[1]:
                b = (a[0], a[1] + 1)
            else:
                b = (a[0], a[1] - 1)
        else:
            if c[0] > a[0]:
                b = (a[0] + 1, a[1])
            else:
                b = (a[0] - 1, a[1])
    return b

# 对于落后情况尽量牵制对方，往对方扔2，返回2的位置(cow,column)
def trouble2(self, board):
    b = self.trouble1(board)
    s = []
    c = 0
    if b != ():
        return b
    else:
        l = board.getNone(not self.isFirst)  # 取得对方空位列表
        a = len(l)
        if a>0:
            for i in range(0, a):
                s.append(self.total(board, l[i][0], l[i][1]))  # 往对方周围除开空位级别数和最大的地方扔2
            c = s.index(max(s))
            b = l[c]
        else:
            b=()
        return b

# 查看周围是否有可合并项，返回可合并项位置及方向组成的列表[[cow,column,diraction]]
def search2(self, board, score, row, column):
    l = []
    d = 0
    while d < 4:
        a = row
        b = column
        s = 0
        if d == 0 and row != 0:
            a -= 1
            s = board.getValue((a, b))
            if score == s:
                l.append([a, b, 0])
            else:
                pass
        elif d == 1 and row != 3:
            s = board.getValue((a, b))
            if score == s:
                l.append([a, b, 1])
            else:
                pass
        elif d == 2 and column != 0:
            b -= 1
            s = board.getValue((a, b))
            if score == s:
                l.append([a, b, 2])
            else:
                pass
        elif d == 3 and column != 7:
            b += 1
            s = board.getValue((a, b))
            if score == s:
                l.append([a, b, 3])
            else:
                pass
        else:
            pass
        d += 1
    return l

# 返回除开空位周围四个棋子的级别之和
def total(self, board, row, column):
    d = 0
    a = 0
    while d < 4:
        s = 0
        if d == 0 and row > 0:
            row -= 1
            s = board.getValue((row, column))
            a += s
        elif d == 1 and row < 3:
            while s == 0 and row != 3:
                row += 1
                s = board.getValue((row, column))
            a += s
        elif d == 2 and column > 0:
            while s == 0 and column != 0:
                column -= 1
                s = board.getValue((row, column))
            a += s
        elif d == 3 and column < 7:
            while s == 0 and column != 7:
                column += 1
                s = board.getValue((row, column))
            a += s
        else:
            pass
        d += 1
    return a

#对于可能存在的对面棋子位于边界线并且刚好可以被我方棋子吃掉不被反吃的情况，返回方向choice;否则返回choice=-1
def chance1(self,board):
    d=0
    l=[]
    a=0
    choice=-1
    if self.isFirst:
```

```
            while column>=0 and s==0:
                a=board.getValue((d,column))
                if a!=0 and self.isin(board,d,column):
                    s=a
                else:
                    pass
                column-=1
            l.append(s)
            d+=1
        #判断是否满足可以被我方吃且吃后不会被反吃的情况
        if (l[0]==board.getValue((0,4)) and self.menace(board,l[0]+1,0,4)==[] and not self.isin(board,0,4) and
            l[1]!=board.getValue((1,4)) and l[2]!=board.getValue((2,4)) and l[3]!=board.getValue((3,4))) or \
           (l[1]==board.getValue((1,4)) and self.menace(board,l[1]+1,1,4)==[] and not self.isin(board,1,4) and
            l[0]!=board.getValue((0,4)) and l[2]!=board.getValue((2,4)) and l[3]!=board.getValue((3,4))) or \
           (l[2]==board.getValue((2,4)) and self.menace(board,l[2]+1,2,4)==[] and not self.isin(board,2,4) and
            l[0]!=board.getValue((0,4)) and l[1]!=board.getValue((1,4)) and l[3]!=board.getValue((3,4))) or \
           (l[3]==board.getValue((3,4)) and self.menace(board,l[3]+1,3,4)==[] and not self.isin(board,3,4) and
            l[0]!=board.getValue((0,4)) and l[1]!=board.getValue((1,4)) and l[2]!=board.getValue((2,4))):
            choice=3
        else:
            pass
        return choice
    else:
        return choice
#若我方为先手且我方有2落在边界线上，在对面空位走个2且合并后不会被对方吃掉的情况\
#返回落子位置和移动方向[(row,column),diraction]
def chance2(self,board):
    k=[]
    d=0
    l=[]
    a=0

    k=[]
    d=0
    l=[]
    a=0
    if self.isFirst:
        while d<4:
            column=3
            s=0
            while column>=0 and s==0:
                a=board.getValue((d,column))
                if a!=0:
                    s=a
                else:
                    pass
                column-=1
            l.append([s,column])
            d+=1
        if l[0][0]==1 and l[0][1]==3 and self.menace(board,2,0,4)==[] and board.getValue(0,4)==0 and not board.getBelong(0,4) \
           and l[1][0]!=board.getValue(1,4) and l[2][0]!=board.getValue(2,4) and l[3][0]!=board.getValue(3,4):
            k=[(0,4),3]
        elif l[1][0]==1 and l[1][1]==3 and self.menace(board,2,1,4)==[] and board.getValue(1,4)==0 and not board.getBelong(1,4) \
           and l[0][0]!=board.getValue(0,4) and l[2][0]!=board.getValue(2,4) and l[3][0]!=board.getValue(3,4):
            k=[(1,4),3]
        elif l[2][0]==1 and l[2][1]==3 and self.menace(board,2,2,4)==[] and board.getValue(2,4)==0 and not board.getBelong(2,4) \
           and l[1][0]!=board.getValue(1,4) and l[0][0]!=board.getValue(0,4) and l[3][0]!=board.getValue(3,4):
            k=[(2,4),3]
        elif l[3][0]==1 and l[3][1]==3 and self.menace(board,2,3,4)==[] and board.getValue(3,4)==0 and not board.getBelong(3,4) \
           and l[1][0]!=board.getValue(1,4) and l[2][0]!=board.getValue(2,4) and l[0][0]!=board.getValue(0,4):
            k=[(3,4),3]
        else:
            k=[]
    else:
        pass
    return k
```

Output 函数：

①在 Player 类底下定义，输入回合数、棋局、mode，输出落子位置或移动方向

②功能：对棋盘局面进行分析，直接输出决策或是调用 minimax 函数或 extremeOutput 函数返回决策

③具体操作为：若剩余时间过少，则减少最大搜索深度以避免超时，若我方棋盘为空，在我方落子。若 goback 是 True 则先手返回 2，后手返回 3(上局吃掉边界上的棋子，要返回原位)。若 mode 为落子且为前期，在我方落子。若 mode 为落子且敌方已被塞满，在我方落子。若 mode 为落子且我方被塞满，考虑在敌方所有空位落子，选取其中分数最高的落子。对于其余情况，分为四种可能:前期、大幅领先(我方最大等级超过 7 且对方最大等级小于我方)、大幅落后(我方最大等级小于敌方最大等级-1)、一般情况。若大幅领先或落后，调用 extremeOutput 函数返回决策。若为前期或一般情况，调用 minimax 函数返回决策。

```
def output(self, currentRound: int, board, mode: str):
    if board.getTime(self.isFirst) <= 0.6 * maxTime:
        self.searchDepth = 2
    elif board.getTime(self.isFirst) <= 0.35 * maxTime:
        self.searchDepth = 1
    alist = board.getScore(self.isFirst)
    yourLlist = board.getScore(not self.isFirst)  # 敌方序列

    if len(alist) is 0:  # 棋盘空着
        return board.getNext(self.isFirst, currentRound)

    elif mode is 'direction' and self.goback:  # 吃完之后回去
        if self.isFirst and board.move(True, 2):  # 先手
            return 2
        elif not self.isFirst and board.move(False, 3):  # 后手
            return 3



    elif mode is 'position' and alist[-1] <= 4 and len(alist) != 0 and len(
            board.getNone(self.isFirst)) is not 0:  # 前期落子
        return board.getNext(self.isFirst, currentRound)

    elif mode is 'position' and len(board.getNone(not self.isFirst)) == 0:  # 敌方已经被塞满
        return board.getNext(self.isFirst, currentRound)

    elif mode is 'position' and len(board.getNone(self.isFirst)) == 0:  # 我方被塞满 敌方还没满
        if len(yourLlist) == 0:
            return board.getNone()[0]
        pointDict = {}
        for position in board.getNone(not self.isFirst):  # 敌方空位
            copyBoard = board.copy()
            copyBoard.add(not self.isFirst, position)
            pointDict[chessboardPoint(copyBoard, self.isFirst, early=False)] = position
            return pointDict[max(pointDict.keys())]  # 返回最高的分数的下其位置

    if mode in ['position', 'direction']:
        if len(alist) is 0:  # 棋盘空着
            pass
        else:  # 棋盘非空
            if alist[-1] < 4 and len(alist) != 0:  # 前期
                return minimax(nowBoard=board, currentRound=currentRound, minormax=False, mode=mode,
                               isFirst=self.isFirst, parent=None, depth=0, early=True, searchDepth=self.searchDepth)
            elif len(yourLlist) is not 0 and (alist[-1] + 2 <= yourLlist[-1]):  # 大幅落后
                return self.extremeOutput(currentRound=currentRound, board=board, mode=mode)
            elif len(yourLlist) is not 0 and alist[-1] >= 7 and yourLlist[-1] < alist[-1]:  # 大幅领先 可以考虑7或8
                return self.extremeOutput(currentRound=currentRound, board=board, mode=mode)
            else:  # 一般情况
                return minimax(nowBoard=board, currentRound=currentRound, minormax=False, mode=mode,
                               isFirst=self.isFirst, parent=None, depth=0, early=False,
                               searchDepth=self.searchDepth)
    else:  # 下一步没得动
        return None
```

2.3 程序限制

程序运行基本不会出错，但是如果进一步加深搜索深度，判断条件过多可能超时。
如果在前期，我方几乎被塞满导致接近无法落子可能会出错，在 minimax 函数中，前期决策树并没有考虑在敌方落子，因为一般情况下，前期双方领域都很空旷所以几乎不可能发生被卡死的情况，因此在设计时似乎没有考虑这个问题，但是在实战中也没有发生过。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：intel® Core(TM) i7-8565U CPU @ 1.80GHz 1.99GHz　内存 8.00GB
- 操作系统：Win10 家庭版
- Python 版本：Python3.82

测试方式：

1、 对中期函数采用了样例输入使其输出分值的情况来调整参数。
2、 在提交热身赛之前组员们由于设备问题远程操纵进行 debug。
3、 在提交到热身赛后通过观察局势复盘来发现问题，对各种函数与参数不断进行改进优化。

```
class Board:...



...


def midscore(board, isFirst):...



l1=[4,1,0,0,      -1,0,0,0]
l2=[1,4,3,0,      -1,-4,0,0]
l3=[4,3,1,0,      -3,0,0,0]
l4=[1,0,0,0,      -3,-5,-2,0]
l=[]
l.append(l1)
l.append(l2)
l.append(l3)
l.append(l4)

b=Board(l)

print(midscore(b,True))

```

上图为中期测试的函数，Board 类为自己写的简略的 Chessboard，通过这种方式对中期函数的参数进行调整，最终确定了最适合比赛的参数结果。

中期函数初始版参加了 370 场比赛，其中胜出 67 场，失败 303 场。

中期函数最终版参加的 410 场比赛中，胜出 87 场，失败 320 场，平局 3 场。比赛均为先后手交替，先手胜出 47 场，后手胜出 40 场。在失败的比赛中，只有 7 场是因为超时而失败。

测试结果：代码运行正常，稳定性也很好，就是竞争力不强。

3.2 结果分析

中期函数初始版：在我们复盘的比赛中，发现一直保守的行为并不能有效获得比赛胜利，有时候一个小棋子占领了最大棋子应该在的地方，导致堵塞，逐渐死亡，而且也很难招架住对方的进攻。这造成了我们应该保护的棋子没有在应该的位置，出现两个较大的相同等级的棋子之后不能有效的进行合成，反倒成为一种阻碍。经过组员讨论，我们放弃了这种猥琐发育，不去进攻，发育也发育不好的策略。

特殊情况处理初始版：最开始的时候人脑并没有考虑出太多的特殊情况，也并没有考虑到多种情况并存的情况，在利用天梯调试的时候不断发现出问题，然后改进，最初没有 ast 函数，然后时常在调用特殊函数时出现超出范围的情况。当时还设计了一种专门深入对面内部的战法，但最终经过讨论决定还是稍微保守一点，于是舍弃了深入敌方内部的函数，并且删除了一些无用的判断。

中期函数最终版：经过改进的中期函数胜率有了提升，其可以实现在变化移动中逐渐壮大自己，但依旧存在把自己堵死的状况，对对面的进攻也较为有效，在有好机会的适合也能实现向对面棋盘落子阻挡对面发育。这一种相对比较激进的算法的稳定发挥也让我们的代码可以去打比赛了。

特殊函数最终版：删除了深入敌方内部的函数，删去了许多无用或者可以被简化的判断，删去了后手时偷吃对面的情况，尽可能做到简化，同时增加了对于多种情况并存的判

断，在面对更复杂情况时也能做出判断。

算法的运行时间可以控制，基本不会超时，中期函数会花费大量的时间，特殊情况处理还不够完善，冗杂的判断条件还是过多。

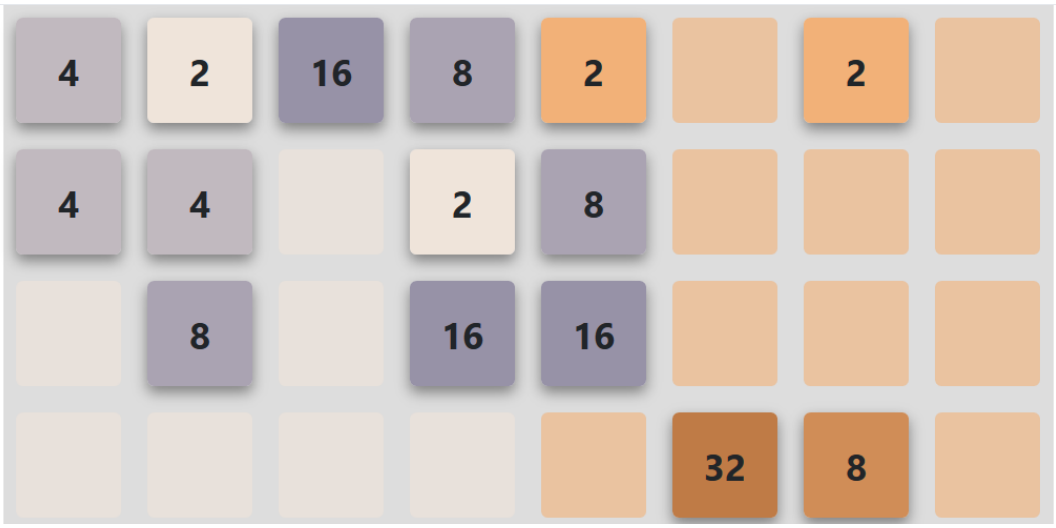
3.3 经典战局（可选）

在这一局我方为先手，上演一把欲擒故纵。

1、暴露破绽，在坐标（2，3）处的 4 去吸引对方来吃

2、对方向左移动，吃掉我方诱饵 4

3、我方露出獠牙，向上移动，将对方两个 8 全部吃掉。真是一出欲擒故纵的好戏！

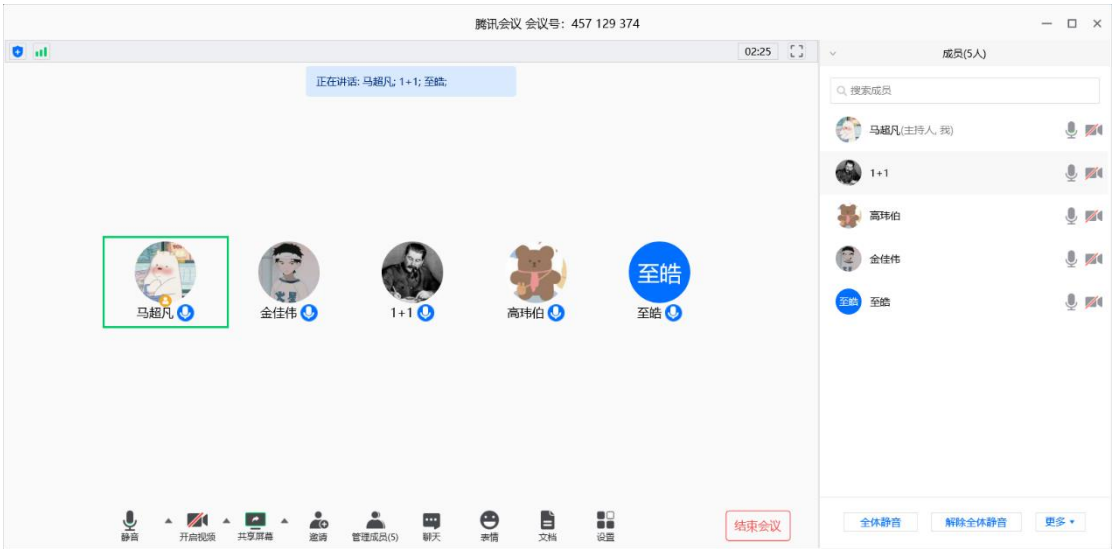


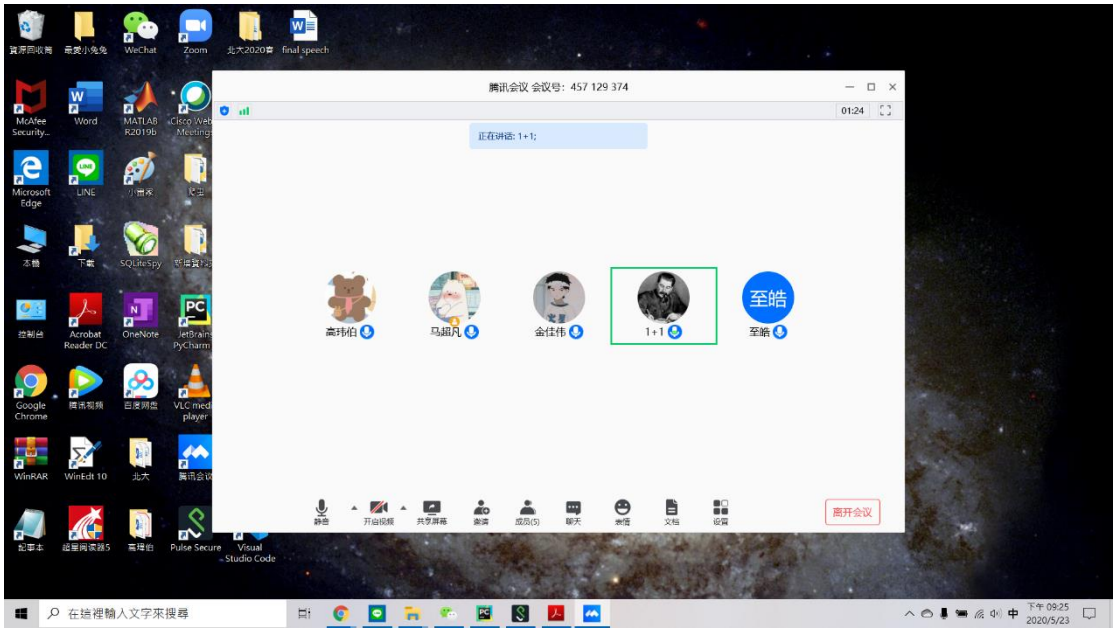
4 实习过程总结

4.1 分工与合作

小组分工：
决策树的实现：高玮伯，易佳怡
前期评估函数：金佳伟
中期评估函数：马超凡
特殊情况处理：吴熙楠

小组交流的方式主要为腾讯会议与微信群，由于篇幅问题，只列出两次会议的截图，微信群聊天记录不再列出。





第一次会议在 5 月 14 日晚，大家在会议之前准备了自己的想法方案，准备好在会议上进行讨论。会议决定了我们的总体方略，会议之后要求同学们对各个部分进行细化，以方便进行下一次会议的分工完成。

第二次会议于 5 月 16 日晚，大家都拿出了自己的细化方案，在会议中我们进行讨论交流，在各个方面选出来大家认为最好的细化策略。在会议上大家进行了分工，在接下来一个星期的时间里去完成各自的代码。

第三次会议于 5 月 23 日晚，大家都拿出来各自的代码，由于设备原因开始远程 dubug，将各自的代码修改完毕后组合成我们最开始的总代码。

之后在微信群里，大家不断发现问题，解决问题，讨论交流，改进各自部分的代码。在天梯赛结束后，大家在微信群里讨论共同完成实习报告。

4.2 经验与教训

1、 这次我们的天梯赛没有取得一个比较好的成绩，现在反思下来应该是我们的估值函数过于复杂，导致花费的时间过多，使得决策树的层数非常浅。这是我们最开始的理解和讨论没有充分建立在实验的基础上造成的，希望小组成员都能够吸取这个教训。

2、 我们这次大作业实行了比较多的线上会议，效果还是非常好的，交流是必要的，群策群力也是必要的，这样我们才能够有比较好并且全面的方案实施，在进行充分的讨论交流之后我们才能更好的开展接下来的工作。

3、 另外，这次中期函数依赖于数值的选取，需要把数值设置的非常精妙，这虽然看起来比较简单，但在实际操作的适合会有很多问题，数值往往难以抉择。不要过分依赖数值的选取是一个教训。

4、 考虑情况较少，对于复杂战局墨守陈规，有大家任务都重的原因，但应该考虑情况更细致。

5、 Minimax 函数每次调用时都要重新制造节点 Node，可能会太过消耗时间，也许可以在制造节点时保存更完善的信息，让下一回合时，能够继续使用其中的部份节点。

4.3 建议与设想

这次实习作业的规则公平性做的非常好，基本上保证了随机因素的最小。但是实习作业临近期末，大家都比较忙，可能没有过多的时间去精致打磨这个实习作业，建议就是希望能够早点开展这个竞赛，尽量不要放在最后。

此外，这学期的课程都是线上授课非常可惜，虽然线上上课时间和空间上相对比较自由，但是却没能体会到良好的课堂气氛，尤其是最后的决赛，如果能在现场看两方代码激烈对战，一定非常刺激且欢乐，希望下学期能继续举办大作业决赛，让我们能去参观。

希望以后的学弟学妹们要有扎实的数理基础。陈斌老师的数算课虽然任务量较大，但是一学期下来是非常有收获的，一定要认真去学，积极参与每一次的活动。

5 致谢

感谢陈斌老师这一学期的指导！
感谢各位助教和技术组的成员为我们做的工作！
感谢我们组的每一位同学对组长的忍耐！
感谢所有组员们都有在 DDL 前完成自己份内的工作！
感谢所有参赛的同学，你们的参与让我们看到了自己的不足及能改进的地方！
感谢我们的电脑，他在这几周不眠不休的努力，非常辛苦！
感谢互联网，让我们即使在五个不同城市也能合作，也让课程能在线上顺利进行！
感谢各位大佬的指导！

6 参考文献

课程网站：http://gis4g.pku.edu.cn/course/pythonds/
https://en.wikipedia.org/wiki/Minimax
https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning
http://programmermagazine.github.io/201407/htm/focus3.html
https://fu-sheng-wang.blogspot.com/2017/02/ai-14-adversarial-search.html
http://blog.codinglabs.org/articles/2048-ai-analysis.html