

数据结构与算法课程实习作业报告

白纯迪 吕东澍 宋思潼 杨轩 刘行健*

摘要：利用评估函数对棋盘局面作出评估，通过负极大值算法对博弈树进行深度搜索，获取合法的最佳下棋决策和合并决策。运用 **alpha-beta** 剪枝提高搜索效率，进而增加搜索深度。通过本地和天梯比赛调整评估函数的评价指标、权重和搜索深度等参数，以提高胜率。

关键字：线性结构 负极大值算法 **alpha-beta** 剪枝 评估函数 面向对象

1	算法思想.....	2
1.1	总体思路.....	2
1.2	算法流程图.....	3
1.3	算法运行时间复杂度分析	6
2	程序代码说明.....	6
2.1	数据结构说明	8
2.2	函数说明	9
2.3	程序限制	13
3	实验结果.....	14
3.1	测试数据	14
3.2	结果分析	16
4	实习过程总结.....	17
4.1	分工与合作	17
4.2	经验与教训	21
4.3	建议与设想	32
5	致谢	32
6	参考文献.....	32

1 算法思想

1.1 总体思路

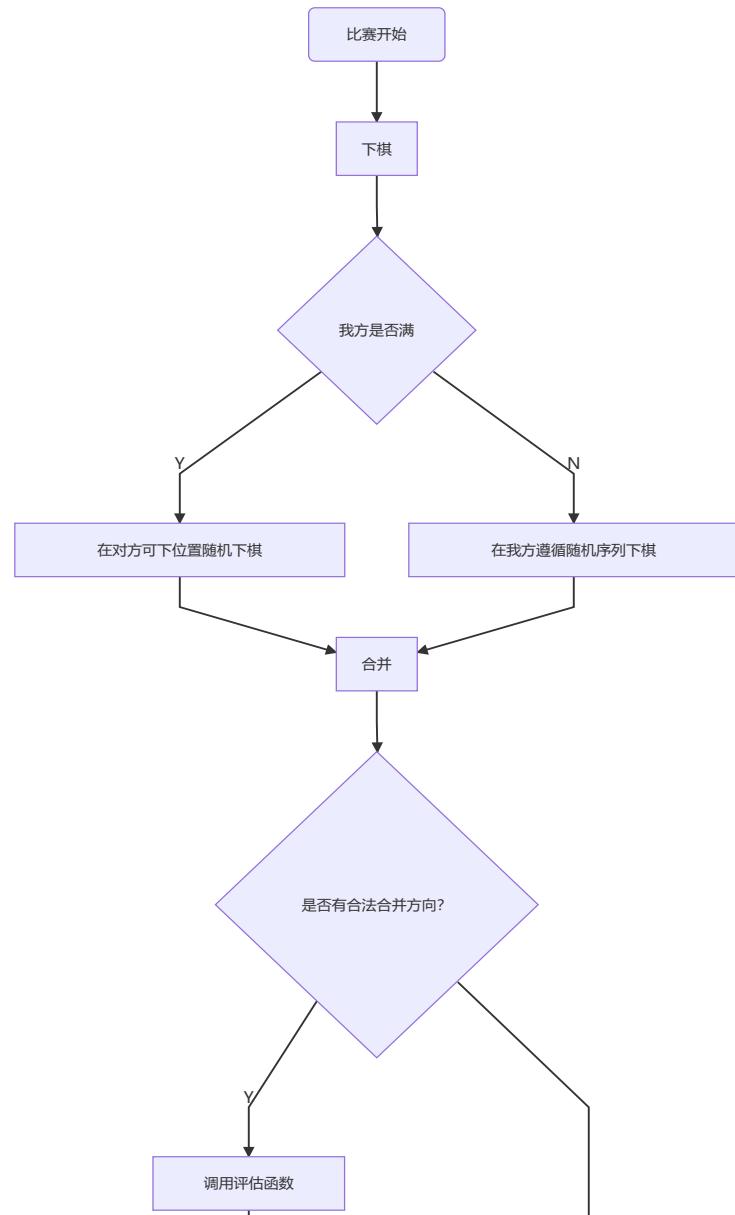
算法的总体思路是通过对棋局各方面性质的评价来估算双方棋盘各自的局面值，通过负极大值算法对博弈树进行深度优先搜索，在最优化对方决策的情况下，获得能使 n 步以后己方与对方局面值差值最大的己方决策。

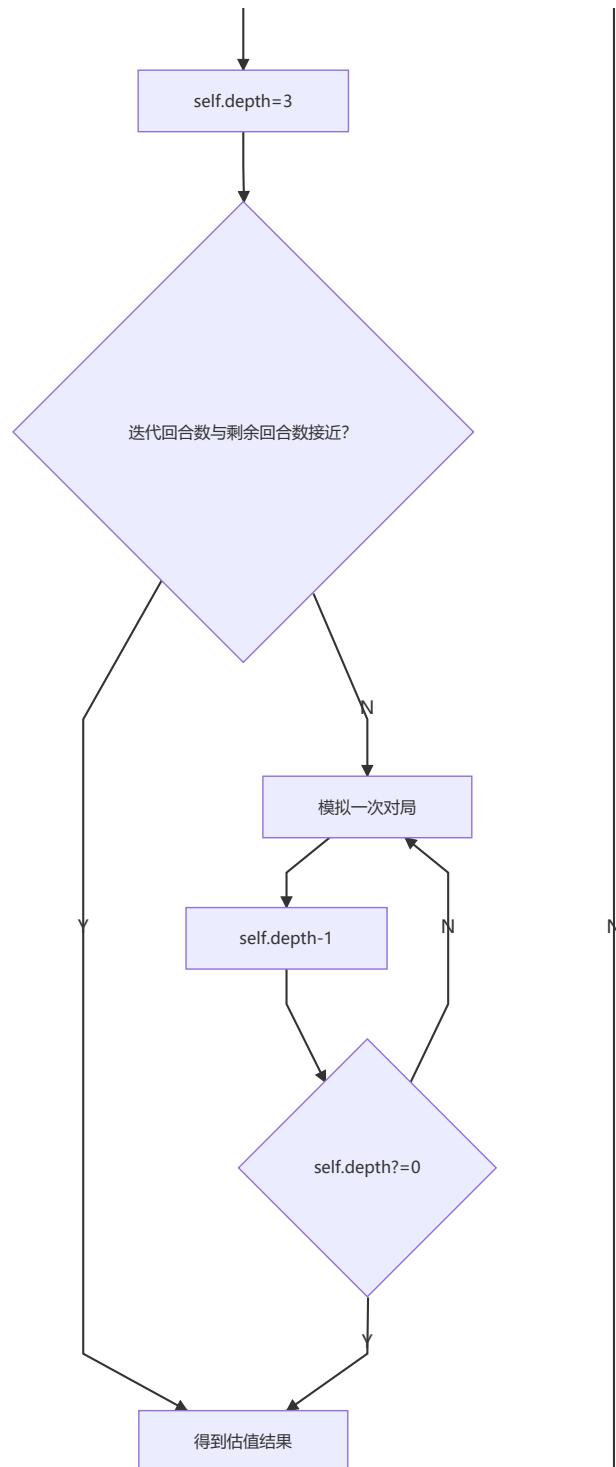
在数据结构上，采用栈来缓存做出变化前的棋盘（即搜索树父节点处的棋盘），采用列表来存储对各种可能的决策，并未用到“四叉树类”和“树节点”。总之，虽然算法本身是一种树形递归算法，但在代码实现时并没有采用树形结构，而是采用了简单的线性栈结构，原因在于，虽然一种局面下对应有四种决策分枝，但实际上可以用简单的列表遍历来实现子节点分枝的遍历，用栈来顺序存取父节点处的棋盘，所以实际上不需要创建树和节点类也可实现博弈树的功能。

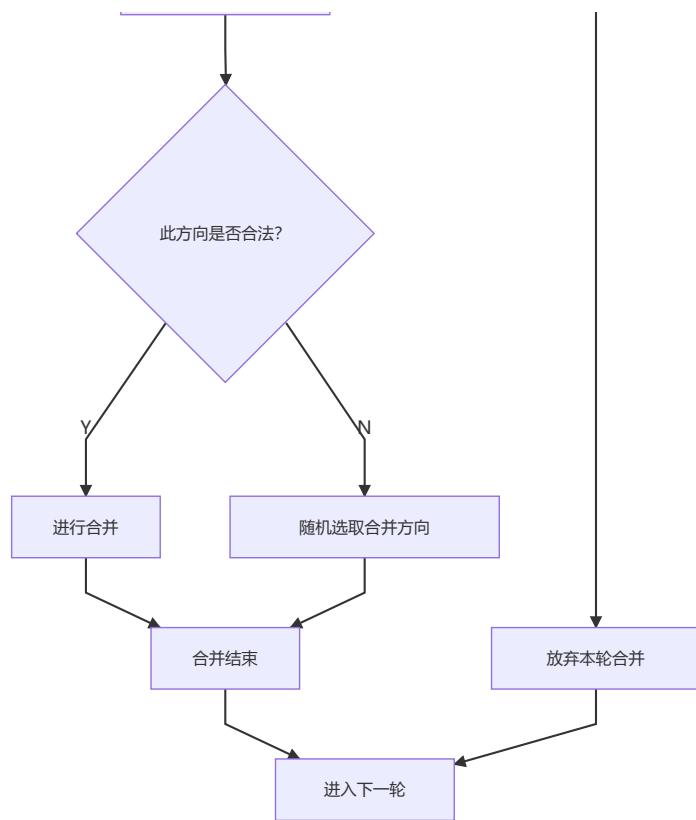
主要的算法策略是通过负极大值算法对不同决策分枝进行搜索，获取某一子层的双方局面值。搜索算法中应用 **alpha-beta** 剪枝以提高搜索效率。需要做出的决策分为两类：决策下棋位置和决策合并方向。

- 【1】 在选择下棋位置的决策时，我们采取较为防守的策略，只要我方棋盘未满，即在我方棋盘处的指定位置下棋，若我方棋盘已满，则随机选择对方棋盘处的空位下棋。这样选择的原因是，在大多数的情况下，选择在对方棋盘下棋的弊大于利；在少部分情况下，如“对方棋盘有较大同级别棋子有合并趋势”“对方棋盘将近填满”，我们也曾尝试编写函数来进攻对方棋盘以达到给对方添堵的目的，但由于函数的考虑并不足够周到，往往结果是为对方“送子”，或者并没有在最终达到“添堵”的目的，所以便放弃了这一想法
- 【2】 在选择合并方向的决策时，我们通过负极大值算法获取合并 3 步（包括对方合并）以后使得己方与对方局面值差值最大的合并方向。在搜索算法的实现中，会先验证该合并方向是否合法，若不合法则直接剪枝，遍历下一种决策。这样即可保证最终做出的决策一定是合法的。其中，合并 3 步是综合考虑搜索深度和超时因素得出的最优搜索深度。在后期优化的过程中，我们曾考虑过“即时优势”与“即时危机”的情况，编写相应的函数来进行即时进攻与防守。这种“即时决策”有别于深度搜索得出的最优决策，目的是为了防止通过深度搜索得出的最优决策使得当前的棋子陷于更大的危机或者放弃了当前更大的局面优势。但由于“即时决策”仍然面临着考虑不够周全的问题，其“掐断式”写法更是使得大部分情况下的递归搜索和评估函数失去了意义，算法变得目光短浅，胜率一直“居低不上”，所以最终还是去掉了即时决策的编写，回归最朴素的搜索算法。

1.2 算法流程图







1.3 算法运行时间复杂度分析

由于场上的棋子数、空格数的数量级与棋盘总个数相等，因此可认为 N 为棋盘总格数。

1.figureValue 函数：

```
def figureValue(self, isFirst): # 将对象方棋子的级别加权求和，得到棋子级别大小和数量
    提供的 value
    selfside = isFirst
    selfscore = self.board.getScore(selfside)
    figurevalue = 0
    for i in selfscore:
        if i >= 7: # 级别太低也没有加的必要
            figurevalue += 3 ** i
        elif i >= 5:
            figurevalue += 2 ** i
        elif i >= 3:
            figurevalue += i * 2
    return figurevalue
```

计算每个棋子的 value 并加入到 figurevalue 中，时间复杂度为 $O(N)$ 。

因此总的时间复杂度为 $O(N)$ 。

2.emptyCellsValue 函数：

```
def emptyCellsValue(self, isFirst): # 计算对象方空位数所提供的 value
    selfSide = isFirst
    selfEmptyCellsNums = len(self.board.getNone(selfSide)) # 对象方空位数
    emptyCellsValue = selfEmptyCellsNums**0.5
    return emptyCellsValue
```

取出对象方棋盘内空格数的时间复杂度为 $O(1)$ ，计算 emptyCellsvalue 的时间复杂度为 $O(1)$ 。

因此总的时间复杂度为 $O(1)$ 。

3.monotonicityValue 函数：

```
def monotonicityValue(self, isFirst): # 单调性 value 评估，仅考虑对象方棋盘内的棋子，且对方棋子入侵时，其所在的行与列的 value 贡献均为 0
    selfSide = isFirst
    monotonicityvalue = 0
    # 遍历对象方棋盘每一行
    for row in range(4):
        .....
    # 遍历对象方棋盘每一列，同上
    for col in range((1 - isFirst) * 4, (1 - isFirst) * 4 + 4):
        .....
    return monotonicityvalue
```

遍历对象方棋盘上的格子，时间复杂度为 $O(N)$ 。

因此总的时间复杂度为 $O(N)$ 。

对于总的评估函数：

```
def estimateValue(self, isFirst): # 计算先手或后手方的棋面 value
    monotonicityWeight = 0.23 # 单调性权重
    emptyCellsWeight = 0.21 # 空格数权重
    figureWeight = 0.42 # 棋子级别权重 0.42

    value = monotonicityWeight * self.monotonicityValue(isFirst) + \
            emptyCellsWeight * self.emptyCellsValue(isFirst) + \
            figureWeight * self.figureValue(isFirst)
    return value
```

调用了函数 1~3，因此时间复杂度为其中最高的 $O(N)$ 。

由于 minmax 搜索与剪枝函数搜索深度为 3，每层搜素取四个方向，每个方向调用一次评估函数。因此，算法的总时间复杂度为 $O(N^3)$ 。

2 程序代码说明

2.1 数据结构说明

【1】 扩充了 Player 类的属性：

`self.board`: 记录当前操作时的棋盘，为 `Chessboard` 类
`self.best_position` 和 `self.best_direction`: 记录得出的最佳决策
`self.depth`: 记录搜索深度
`self.another`: 记录轮到己方下棋时在己方棋盘的指定下棋位置
`self.current_round`: 记录当前回合数，在递归搜索时需要更新
`self.prev_stack`: 记录“父棋盘”的存储栈

```
class Player:
    def __init__(self, isFirst, array):
        # 初始化
        self.isFirst = isFirst
        self.array = array
        self.board = None
        self.best_direction = None
        self.best_position = None
        self.depth = 3
        self.another = None # 轮到己方下棋时，若下在己方棋盘则应该下的位置
        self.current_round = 0 # 表示当前进行的回合数
        self.prev_stack = Stack() # 用于回溯之前的棋盘，即用于cancel move
```

将这些随递归搜索而改变的量存储为 `Player` 类的属性，既可以避免变量作用域的标注与分辨，又可以减少向功能函数传入的参数的数量，使得程序更加简洁明了。

【2】 应用栈来存取“父棋盘”：

在递归搜索时，由于调用 `add` 方法和 `move` 方法会改变棋盘的状态，所以每次在对棋盘做出相应改变前，都要将当前棋盘存入栈中。在此分枝搜索结束，遍历下一种决策前，`pop` 出栈顶的“父棋盘”并赋值给当前棋盘 `self.board`，以达到恢复棋盘状态（即“取消变化”）的目的。

```
for direction in directionList:
    current_board = self.board.copy()
    if self.board.move(isFirst, direction): # 此合并方向合法时make move
        self.prev_stack.push(current_board)
        # 进行递归
        if isFirst: # 本层player为先手方，即下一步是另一方选择方向进行合并
            value = -self.get_ChoiceValue(depth - 1, -beta, -alpha, -player, not isFirst, 'direction')
        else: # 本层player为后手方，即下一步是另一方选择位置下棋
            self.current_round += 1 # 下一步就是下一回合了
            value = -self.get_ChoiceValue(depth - 1, -beta, -alpha, -player, not isFirst, 'position')

    self.board = self.prev_stack.pop() # cancel move
```

2.2 函数说明

【1】choose 接口：为了尽量简化 output 接口里的代码，设置两个 choose 接口分别返回最终的下棋位置决策和合并方向决策。

```

def choose_position(self):
    # 优先进攻，其次下在自己这里，其次下在对方那里（具体下在哪个位置？需要决策！）
    self.best_position = self.another
    if self.best_position != ():
        return self.best_position
    else: # self.another可能为() (己方棋盘满了)
        available = self.board.getNone(not self.isFirst) # 对方的允许落子点
        if not available: # 整个棋盘已满
            return None
        else:
            from random import choice
            return choice(available)

def choose_direction(self):
    self.best_direction = None
    self.get.ChoiceValue(self.depth, -1000000000, 1000000000, -1, self.isFirst, 'direction')
    if self.best_direction is not None:
        return self.best_direction
    # 若迭代深搜得出的最佳方向是非法的，则随机选择
    from random import shuffle
    directionList = [0, 1, 2, 3]
    shuffle(directionList)
    for direction in directionList:
        if self.board.move(self.isFirst, direction):
            return direction

```

【2】get.ChoiceValue 接口：递归搜索树。

通过调用该函数，为 self.best_direction 赋值；函数返回本层决策的最大 value 值。

```

# 运用negamax算法和alpha-beta剪枝获取这一步决策的value值
def get.ChoiceValue(self, depth, alpha, beta, player, isFirst, mode):
    player 参数标注当前玩家：-1 为己方，1 为对方
    isFirst 参数标注当前玩家是否为先手：True 为先手，False 为后手
    由于递归搜索时为双方交替决策，故下一层递归调用时需取为-not player 和 not isFirst
    • 递归主体：按当前需要做出决策的类型（mode）进行分类
    1. 当这一步需要做出的决策是合并方向时：

```

Step 1: 构造列表存放所有可能的方向进行遍历。

这里方向的存放顺序与先后手有关，保证在遍历得到的局面值相同时优先选择向己方棋盘的“后方”合并，以达到防守的目的。

```

if isFirst:
    directionList = [0, 1, 3, 2]
else:
    directionList = [0, 1, 2, 3]

for direction in directionList:

```

Step 2: 判断该 direction 是否合法，以保证后续得到的最优方向是合法方向中的最优决策。

```

for direction in directionList:
    current_board = self.board.copy()
    if self.board.move(isFirst, direction): # 此合并方向合法时make move
        self.prev_stack.push(current_board)
        # 进行递归

else: # 此合并方向非法时, 继续尝试下一个方向
    self.board = current_board # if判断时移动了盘面, 应及时恢复盘面
    continue

```

Step 3: 合并方向合法时, 进行递归。根据本轮 `player` 是先手还是后手来决定下一层递归调用时的 `mode` 类型。由于为负极值算法, 本层取下一层的相反数作为本层该节点处的 `value` 值。

需要注意的是, 若本轮 `player` 为后手方, 则下一轮为下一回合的开始, 需更新 `self.current_round`。

```

# 进行递归
if isFirst: # 本层player为先手方, 即下一步是另一方选择方向进行合并
    value = -self.get.ChoiceValue(depth - 1, -beta, -alpha, -player, not isFirst, 'direction')
else: # 本层player为后手方, 即下一步是另一方选择位置下棋
    self.current_round += 1 # 下一步就是下一回合了
    value = -self.get.ChoiceValue(depth - 1, -beta, -alpha, -player, not isFirst, 'position')

```

Step 4: 本分枝深度搜索结束后, 恢复棋盘, 并选出本层决策的最大 `value` 值返回。若本层为第一层, 则 `value` 值最大的决策即为所选的最佳决策。

```

self.board = self.prev_stack.pop() # cancel move

# 进行alpha-beta剪枝
if value >= beta:
    if depth == self.depth:
        self.best_direction = direction
    return beta
if value > alpha:
    if depth == self.depth: # 回合数为现实回合时选择best direction
        self.best_direction = direction
    alpha = value

```

最终 `return alpha`

2. 当这一步需要做出的决策是下棋位置时:

由于选择下棋位置的方法是以本方指定下棋位置为先, 本方棋满后随机选择对方棋盘空位下棋, 故直接将双方的决策各执行一遍后, 又回到了选择合并方向的决策上。

```

another = self.board.getNext(isFirst, self.current_round)
if another != ():
    self.board.add(isFirst, another)
else:
    available = self.board.getNone(not isFirst)
    if available:
        from random import choice
        self.board.add(isFirst, choice(available))
isFirst = not isFirst # 交换下棋方
another = self.board.getNext(isFirst, self.current_round)
if another != ():
    self.board.add(isFirst, another)
else:
    available = self.board.getNone(not isFirst)
    if available:
        from random import choice
        self.board.add(isFirst, choice(available))
isFirst = not isFirst # 交换下棋方

# 双方下棋结束，当前盘面为双方add后的盘面

```

之后代码部分与 mode=='direction' 时完全相同。

- 递归结束条件：到达所定深度或当前回合数接近总回合数（500）
当当前回合数接近总回合数剪掉搜索枝可以避免搜索到 500 回合以后的棋盘 array 缺失的 bug。

```

def get.ChoiceValue(self, depth, alpha, beta, player, isFirst, mode): # isFirst表示当前player是否为先手(bool)
    if depth == 0:
        return self.get_BoardValue(player)
    if self.current_round >= len(self.array) - 3: # 迭代到的回合数快超过总回合数时，剪掉搜索枝
        return self.get_BoardValue(player)

```

【3】 get_BoardValue 接口：返回某层的双方局面值之差（己方-对方）

```

def get_BoardValue(self, player): # player=-1时为己方, player=1时为对方
    isFirst = self.isFirst if player == -1 else (not self.isFirst) # 表示当前玩家是先手or后手
    selfvalue = self.estimateValue(isFirst)
    enemyvalue = self.estimateValue(not isFirst)
    return selfvalue - enemyvalue

```

【4】 estimateValue 接口：返回某方棋盘的局面值。isFirst 表是否为先手方，与是否为己方无关。

```

def estimateValue(self, isFirst): # 计算先手或后手方的棋面value
    monotonicityWeight = 0.23 # 单调性权重
    emptyCellsWeight = 0.21 # 空格数权重
    figureWeight = 0.42 # 棋子级别权重0.42

    value = monotonicityWeight * self.monotonicityValue(isFirst) + \
            emptyCellsWeight * self.emptyCellsValue(isFirst) + \
            figureWeight * self.figureValue(isFirst)
    return value

```

【5】 三个评价指标接口：

- monotonicityValue: 单调性评价指标

```

# 单调性value评估，仅考虑对象方棋盘内的棋子，且对方棋子入侵时，其所在的行与列的value贡献均为0
def monotonicityValue(self, isFirst):
    selfSide = isFirst
    monotonicityvalue = 0
    # 遍历每一行
    for row in range(4):
        # 引入单调性因子，表征单行或者单列的单调性好坏
        monotonicityFactor = 0
        # 列数从对象方棋盘的最左列到最右列
        col = (1 - isFirst) * 4
        while col < (1 - isFirst) * 4 + 3:
            # 当前格子的级别
            currentValue = self.board.getValue((row, col))
            # 下一格的坐标
            nextrow = row
            nextcol = col + 1
            # 有内鬼，则终止交易，转至下一行
            if selfSide != self.board.getBelong((row, col)) or \
                selfSide != self.board.getBelong((nextrow, nextcol)):
                monotonicityFactor = 0
                break # 只是跳出while循环
            # 无内鬼，计算下一格的级别，单调性因子变化
            else:
                nextValue = self.board.getValue((nextrow, nextcol))
                '''此表达式可以更改'''
                monotonicityFactor += nextValue - currentValue
            # 列数加一，继续下一格
            col += 1
        # 此行的单调性因子计算完毕，加至value中
        '''此表达式可以更改'''
        monotonicityvalue += abs(monotonicityFactor)

    # 遍历对象方棋盘每一列
    for col in range((1 - isFirst) * 4, (1 - isFirst) * 4 + 4):
        monotonicityFactor = 0
        row = 0
        while row < 3:
            currentValue = self.board.getValue((row, col))
            nextrow = row + 1
            nextcol = col
            if selfSide != self.board.getBelong((row, col)) or \
                selfSide != self.board.getBelong((nextrow, nextcol)):
                monotonicityFactor = 0
                break
            else:
                nextValue = self.board.getValue((nextrow, nextcol))
                monotonicityFactor += nextValue - currentValue
            row += 1
        monotonicityvalue += abs(monotonicityFactor)
    return monotonicityvalue

```

2. emptyCellsValue: 空格数评价指标

```

def emptyCellsValue(self, isFirst): # 计算对象方空位数value
    selfSide = isFirst
    selfEmptyCellsNums = len(self.board.getNone(selfSide)) # 对象方空位数
    '''此处计算式可更改，但是应有：在空格数较少时，value对空格数的导数较大，空格数较大时，value对空格数的导数较小'''
    emptyCellsvalue = selfEmptyCellsNums ** 0.5 # 更改为幂函数
    return emptyCellsvalue

```

3. figureValue: 棋子级别评价指标

```
def figureValue(self, isFirst): # 将对象方棋子的级别加权求和, 得到棋子级别大小和数量提供的value
    selfside = isFirst
    selfscore = self.board.getScore(selfside)
    figurevalue = 0
    for i in selfscore:
        if i >= 7: # 级别太低也没有加的必要
            figurevalue += 3 ** i
        elif i >= 5:
            figurevalue += 2 ** i
        elif i >= 3:
            figurevalue += i * 2
    return figurevalue
```

2.3 程序限制

目前没有发现程序报错或做出违法行为的现象。

在开发与优化过程中已被解决的 bug 有：

1. array 数组越界问题：原因是递归搜索树搜索到了 500 回合以后的棋盘状态。

改进方法：剪掉 497 回合后的搜索枝

```
if self.current_round >= len(self.array) - 3: # 迭代到的回合数快超过总回合数时, 剪掉搜索枝
    return self.get_BoardValue(player)
```

2. 棋盘状态复原的问题：只有深拷贝得到的对象才能在原对象发生改变后保持不变。

```
current_board = self.board.copy()
if self.board.move(isFirst, direction): # 此合并方向合法时make move
    self.prev_stack.push(current_board)
```

第一步 current_board 需要赋值为 self.board 的深拷贝对象，而不是 self.board, 否则后续栈中的“父棋盘”均会随 self.board 的改变而改变，与 self.board 的指向是完全一致的。

3 实验结果

3.1 测试数据

实验环境说明：

- 硬件配置：Intel(R) Core(TM) i7-8580U CPU @1.8GHz 1.99GHz/8.00G (CPU/内存)
- 操作系统：Window10/64位 (名称/版本)
- Python 版本：3.8.2 (版本号)

3.1.1 测试方法：

(1) 关于迭代深度的测试

对于迭代深度的测试比较简单，即在确保不超时的情况下尽可能将其调为最大。

(2) 关于评估函数的测试

对于评估函数考量参数及其权重的测试，本组主要采取控制变量法，即控制其他参数及其权重不变，调整其中某一个参数的权重，并与调整之前的代码进行热身赛，以此来确定最适合本组算法的评估函数参数及其权重。

(3) 关于整体算法稳定性的测试

对于算法在运行过程中可能会出现的包括但不限于超时，调用非法等问题，需要对算法进行大量的实战测试，与不同的组的测试代码进行对战，以此来检测代码的稳定性与实战能力。

3.1.2 测试结果：

(1) 迭代深度测试

注：由于代码最初有部分函数在 minimax 与 $\alpha - \beta$ 剪枝之前进行，得出的迭代深度实际上是伪迭代，所以会出现极其不可思议的数据。

迭代深度	6	7	8	9	10	11
测试结果	正常运行	正常运行	1.5%超时	78%超时	100%超时	100%超时

测试结论：最佳迭代深度为 depth=7

(2) 评估函数测试

本组算法最初的评估函数共有 6 个参数，分别为平滑性，单调性，最大棋子级别，空格数，孤岛数，对方棋子在我方的评估。综合考量后，由于孤岛难以出现，且对整体算法贡献不大，故取消这一项。

根据参考文献，初步确定各项权重如下：

Smooth weight: 0.02

Mono weight: 0.26

Empty weight: 0.54

Max weight: 0.20

首先调节平滑性权重：（与调节之前的对战）

平滑性权重	胜场数	负场数	胜率
0.02	33	67	33%
0.04	31	69	31%
0.06	39	61	39%
0.08	49	51	49%
0.10	47	53	47%
0.12	58	42	58%
0.14	52	48	52%
0.16	50	50	50%
0.18	41	59	41%
0.20	38	62	38%
0.22	42	58	42%
0.24	45	55	45%
0.26	35	65	35%
0.28	29	71	29%
0.30	21	79	21%
0.32	25	75	25%
0.34	12	88	12%
0.36	13	87	13%
0.38	5	95	5%
0.40	8	92	8%

同样方法调节其他权重，最终得出最佳权重如下：

Smooth weight: 0.12

Mono weight: 0.23

Empty weight: 0.22

Max weight: 0.42

Enemy weight: 0.17

(4) 稳定性测试（包括热身赛的结果）：

在本地进行大量测试没有问题之后，我们转战天梯热身赛

与当时天梯热身赛排名前十的代码对战，全部以 0-10 战败，但没有报错

对战其他队伍的战况如下

我方	我方胜利	对方胜利	对方	胜利原因	失败原因
我方	3	97	Test1	得分统计	得分统计
我方	8	92	微波炉 4 号	得分统计	得分统计
我方	12	88	微波炉 2 号	得分统计	得分统计
我方	23	77	Emm	得分统计	得分统计
我方	19	81	京都人形	得分统计	得分统计
我方	100	0	Test2	对方报错	得分统计
我方	0	100	0.3.3.3	得分统计	得分统计
我方	15	85	Kiiii	得分统计	得分统计

我方	0	100	Player526	得分统计	得分统计
我方	77	23	四季平安	对方报错	得分统计
我方	19	81	D3	得分统计	得分统计
我方	0	100	对战试试	得分统计	得分统计
我方	0	100	二龙戏珠	得分统计	得分统计

经过测试发现，我方算法稳定性较好，但竞争力极差。

测试之后，我们在算法中加入了 `contest` 和 `athena` 函数来提高算法的竞争力，但效果并不明显，反而导致新增函数覆盖评估函数的情况，导致评估失效。

所以，我们对算法进行了反璞归真式的处理，重新编辑代码，只留下搜索和评估，同时重整了搜索的代码。接下来的测试，更多的使用了与官方的 `bot` 进行对战处理。

我方	我方胜利	对方胜利	对方	胜利原因	失败原因
大换血	20	0	Bot D	得分统计	
大换血	40	0	Bot C	得分统计	
大换血	7	23	Bot B	得分统计	得分统计
大换血	0	40	Bot A		得分统计
大换血	50	0	8848hh	得分统计	

经过这一次优化，我方算法竞争力明显提升，我们又针对评估函数的空格数一项进行了优化，使得其权重分配更加合理。最终版本战绩如下：

我方	我方胜利	对方胜利	对方	胜利原因	失败原因
最终版	12	8	大换血	得分统计	得分统计
最终版	19	41	Bot B	得分统计	得分统计
最终版	2	9	Bot A	得分统计	得分统计

最终版本稳定性较高，竞争力较之前有了显著提升。

3.2 结果分析

3.2.1 测试阶段

在测试阶段，评估函数被覆盖，导致无法对战况做出正确的判断，添加的撤回机制，进攻机制并不能根据形势做出正确的判断，竞争力较差。算法在迭代深度为 7 时稳定不超时，在迭代深度为 8 是不稳定偶尔超时。由于评估函数被覆盖，迭代均为伪迭代，时间开销主要用于各个函数以及权重计算。

3.2.2 热身赛阶段

此阶段添加了 `contest` 和 `athena` 函数增强了对我方的保护，但由于评估函数的问题并没有修改，效果仍然不理想，战绩较差。伪迭代深度甚至可以达到 12，时间开销仍然没有用于迭代评估。

3.2.3 天梯实战阶段

此阶段进行了大换血的更改，删除了没有用处的函数，更改了评估函数的迭代机制，效果理想，战绩有显著提升。此时经过测试发现，迭代深度为 3 时，算法稳定不超时，迭代深度超过 3 时，有超时风险。时间开销主要用于迭代计算评估局面，以做出正确的选择。

4 实习过程总结

4.1 分工与合作

【1】 分工情况:

刘行健——负责除评估函数接口以外的算法和代码编写；后期主要负责 debug 和评估函数的优化

杨轩——负责评估函数的研究和编写

宋思潼——负责分析攻守策略，编写相应功能函数

吕东澍——负责程序测试和比赛结果的分析与记录，调整评估函数权重

白纯迪——负责搜索树算法的意见指导；后期负责分析攻守策略

【2】 工作阶段:

第一阶段：写出整体的代码框架：[←](#)

DDL：5.20 晚(尽早，写完后及时共享)[←](#)

【1】 负极大值搜索算法（博弈树）及 alpha-beta 剪枝 —— 刘行健，白纯迪[←](#)

分工方法：两人同时进行，不断讨论，最后整合优化[←](#)

【2】 评估函数 —— 杨轩[←](#)

初期任务：写出单人版 2048 的评估函数[←](#)

第二阶段：优化算法 —— 宋思潼，吕东澍（其他三人参与讨论，尤其是【2】）[←](#)

DDL：初步定 5.28（左右）开始天梯赛竞技[←](#)

(6.2 课上为冠亚军争夺战)[←](#)

【1】 优化算法效率[←](#)

【2】 关于进攻和防守的决策：评估函数的优化 —— 添加更多条件分支及更改权重[←](#)

进攻：在对方棋盘落棋的位置；有棋子在对方棋盘时的合并方向[←](#)

防守：对方棋子在我方棋盘时的情况[←](#)

【3】 检查添加【2】后的非法行为等异常情况[←](#)

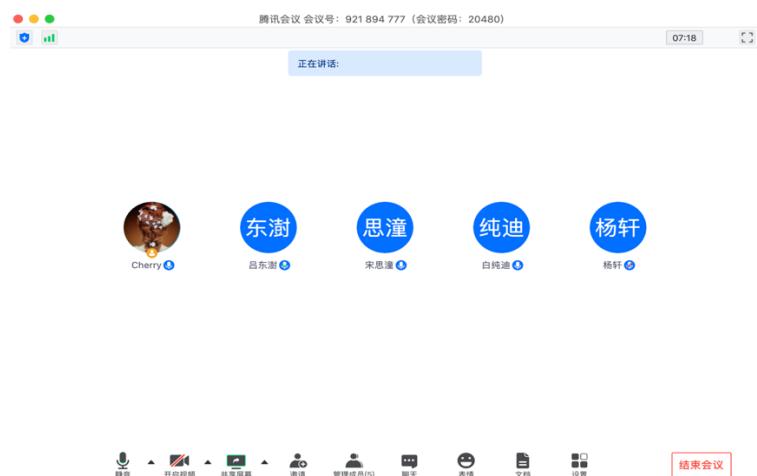
【4】 本地运行平台进行初步调试：[←](#)

通过人机对战和人人对战来枚举特殊情况，优化评估函数[←](#)

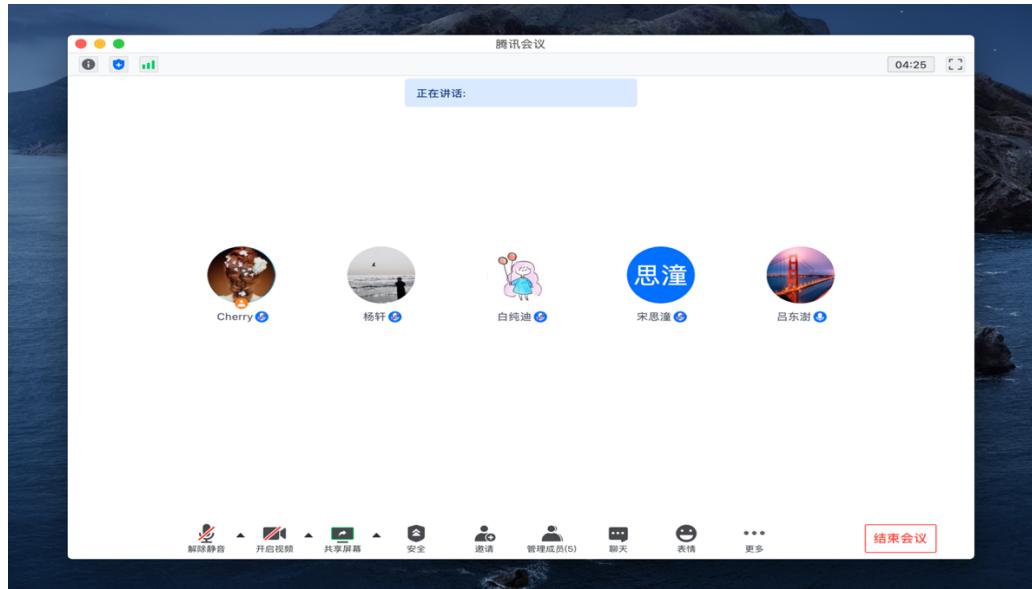
【5】 进行代码竞技场热身赛尝试（不是天梯赛）[←](#)

【3】 合作方式

1. 阶段性组会与会议记录



(第一次组会)



(第二次组会)

2. gitee 合作

gitee 开源软件 企业版 高校版 博客 我的码云 搜索项目

35 次提交 1 个分支 0 个标签 0 个发行版 1 位贡献者

master + Pull Request + Issue 文件 Web IDE 克隆/下载

c CherryLIU01 最后提交于 5 天前 删除文件 模块代码文件夹/keep

player文件夹 最终版代码。修改评估函数，排版代码，添加详细注释 5 天前

调试报告文件夹/比赛报告整合 新建调试报告文件夹，现有比赛报告表格文件一个 17 天前

.gitignore Initial commit 19 天前

README.en.md Initial commit 19 天前

README.md update README.md. 5 天前

README.md

2048大作业

player文件夹

内含各代player.py文件，为调试过程中的个版本代码
命名方式: player x (x为数字, 表示第x代)
每一代都有重大区别, 比如新增函数、修改评估函数、代码逻辑bug修改等
然而这9代代码都被废弃了, 因为有一些根源性的问题一直没有解决。终极版在天梯

调试报告文件夹

1. 比赛报告: 为调参或加函数或改逻辑bug时进行的各种比赛, 包括在本地与各种ai的比赛 和 在竞技场中的比赛 命名规则: 比赛报告_日期 内容: 胜率比较与分析; 报错/违规/超时的发现与代码修正 复盘结果最好有局部截图分析
2. 策略报告: 分析比赛策略, 进行算法改进的心路历程 命名规则: 策略分析_日期 内容: 通过分析比赛想出一些策略 (函数) 来进行进攻或防守
3.

天梯player文件夹

为投入到天梯小组赛区的代码合集

参与贡献

master		TEAMWORK2048_major_tasks / player文件夹	新建文件	新建文件夹	上传文件	克隆/下载
C CherryLIU01 最后提交于 5天前 最终版代码 。修改评估函数，排版代码，添加详细注释						
...						
上天梯的代码		最终版代码。修改评估函数，排版代码，添加详细注释				5天前
player 1.py		新建player文件夹，里面存放了各代player总代码				17天前
player 2.py		新建player文件夹，里面存放了各代player总代码				17天前
player 3.py		新建player文件夹，里面存放了各代player总代码				17天前
player 4.py		update player文件夹/player 4.py.				15天前
player 5.py		新增revoke (进攻后撤退机制) 和invade (进攻机制)，评估函数权重做出调整				9天前
player 6.py		新增contest (根据形势撤退机制)				9天前
player 7.py		在评估函数中加了一个selectvalue项，没什么用，别看了				8天前
player 8.py		新增athena机制，并完善了contest机制，更正了一些逻辑细节				8天前
player 9.py		新增athena_cut机制；对评估函数进行改动				7天前

3. 微信群合作





4.2 经验与教训

虽然说小组最终版的代码十分朴素，但其实我们组在开发的过程中有一段痛心的血泪史。首先，在1.1中讲过，我们后期优化时重点侧重了对于“即时危机”和“即时优势”的细化，编写了一些掐断式的决策函数，却由于在有限的时间内未能将函数优化得十分周到，反而导致算法的目光变得短浅，胜率还不如以往的“简朴版”。在“繁琐版”天梯经过一段时间的低靡之后，我们决定“返璞归真”将代码更换为以前的简朴版，排名这才提高了不少。以及，我们最初版的评估函数评估了棋盘状态的许多方面，但由于评估方面太多反而导致某些方面的权重为0时貌似最优，故删去了一些次要的评估方面，留下最关键的几个。

下面展示的是我们在后期优化过程中编写的函数以及编写函数时的心路历程。

4.2.1 contest 函数

```

def contest(self):
    # 分为先后手，判断我方棋盘位置
    judgelist = [] #存放危险位置棋子

    if self.isFirst:#先手
        for col in range(1, 8):
            for row in range(0, 4):
                judgeyou = (row, col)
                judgeme = (row, col - 1)
                # 按照从左往右第一个出现两方棋子交界处
                if self.board.getValue(judgeyou) != 0 and (not self.board.getBelong(judgeyou)) and \
                    self.board.getValue(judgeme) != 0 and self.board.getBelong(judgeme):
                    # 前方没有危险 pass
                if self.board.getValue(judgeyou) != self.board.getValue(judgeme):
                    pass
                # 前方有危险
                elif self.board.getValue(judgeyou) == self.board.getValue(judgeme):
                    if row != 0: # 上方有可以存放棋子格子
                        judgeyouup = (row - 1, col)
                        if self.board.getValue(judgeyouup) != 0 and (
                            not self.board.getBelong(judgeyouup)): # 得到上方路径上第一棋子 如果一直没有棋子 则也无必要
                            judgelist.append(self.board.getValue(judgeyouup))
                    if row != 3: # 下方有可以存放棋子格子
                        judgeyoudown = (row + 1, col)
                        if self.board.getValue(judgeyoudown) != 0 and (not self.board.getBelong(judgeyoudown)):
                            judgelist.append(self.board.getValue(judgeyoudown))
                    if col != 7: # 右方有可以存放棋子路径
                        judgeyouright = (row, col + 1)
                        if self.board.getValue(judgeyouright) != 0 and (not self.board.getBelong(judgeyouright)):
                            judgelist.append(self.board.getValue(judgeyouright))

                    self.best_direction = 3
                    for value in judgelist:
                        if value == self.board.getValue(judgeme) + 1: # judgelist中只要有一个value满足这个条件就得往左撤退
                            self.best_direction = 2
                        else:
                            continue
                    # 若没有value满足这个条件，则可以往右吃

    else:
        #我方为后手
        for col in range(6, -1, -1): # 从6到0
            for row in range(0, 4):
                judgeyou = (row, col)
                judgeme = (row, col + 1)
                # 按照从左往右第一个出现两方棋子交界处
                if self.board.getValue(judgeyou) != 0 and self.board.getBelong(judgeyou) and \
                    self.board.getValue(judgeme) != 0 and (
                    not self.board.getBelong(judgeme)): # 这里我改了，注意这是我方后手， judgeme的应该是False才行
                    # 前方没有危险
                    if self.board.getValue(judgeyou) != self.board.getValue(judgeme):
                        pass
                    # 前方有危险
                    elif self.board.getValue(judgeyou) == self.board.getValue(judgeme):
                        if row != 0: # 上方有可以存放棋子格子
                            judgeyouup = (row - 1, col)
                            if self.board.getValue(judgeyouup) != 0 and self.board.getBelong(
                                judgeyouup): # 得到上方路径上第一棋子 如果一直没有棋子 则也无必要
                                judgelist.append(self.board.getValue(judgeyouup))
                        if row != 3: # 下方有可以存放棋子格子
                            judgeyoudown = (row + 1, col)
                            if self.board.getValue(judgeyoudown) != 0 and self.board.getBelong(judgeyoudown):
                                judgelist.append(self.board.getValue(judgeyoudown))
                        if col != 0: # 左方有可以存放棋子路径
                            judgeyouleft = (row, col - 1)
                            if self.board.getValue(judgeyouleft) != 0 and self.board.getBelong(judgeyouleft):
                                judgelist.append(self.board.getValue(judgeyouleft))

                    self.best_direction = 2
                    for value in judgelist:
                        if value == self.board.getValue(judgeme) + 1: # judgelist中只要有一个value满足这个条件就得往右撤退
                            self.best_direction = 3
                        else:
                            continue
                    # 若没有value满足这个条件，则可以往左吃

```

4.2.1.1 心路历程

1. 函数作用：

在修改完评估函数后复盘战局，我们发现常常会出现这样的情况：我方棋子当与对方接壤且级别一致时常常会不顾一切地向前进攻，而这样的结果往往是“螳螂捕蝉，另一只蝉在背后”，为了解决这样的问题，我编写了 `contest` 函数，企图先通过 `contest` 的判断避免 AI 执行评估函数得出来的“愚蠢结果”。

2. 函数实现：

初版 `contest` 函数的优先级高于评估函数，首先根据先后手不同选择不同的遍历顺序找到接壤处，如果存在交锋的“危险”，将继续判断对方棋子相邻三个方向的棋子级别是否会对我们的进攻造成威胁，如果有威胁，则选择后退；如果前方尚且没有危险则迅速出击。

在对增加了 `contest` 的 AI 进行复盘的时候，虽然胜率略有增加，但是这个策略仅在后期起着比较大的作用。而前期双方接壤时，对我们造成威胁的棋子不仅仅是相邻三个方向上的，而是三个方向路径上的，针对这种情况，我对代码进行了修改，得到了第二版的 `contest` 函数（如上）。

再次复盘后我们发现，的确避免了这种情况。但是修改 `contest` 前后，AI 的策略发生了极大的变化，从相对积极转变为了相对保守，在这样的情况下，我们的胜率并没有明显的提升。我们最终版本的代码也没有选用 `contest` 函数，再次修改评估函数之后，我们放弃了 `contest` 函数。现在看来，`contest` 函数编写的背景是在我们评估函数存在瑕疵的情况下，并且这种人眼的“短视”存在极大的“鼠目寸光”的危险，不如效能比较好的 AI 的多步棋局评估。

4.2.2 athena 函数

```

def athena(self):
    # 先手棋局
    if self.isFirst and self.best_direction == 3:
        mylist = [None] * 4 # 我方边界处棋子级别列表
        yourlist = [None] * 4 # 对方边界（未必接壤）处棋子级别列表
        current_board = self.board.copy()
        self.board.move(self.isFirst, self.best_direction)
        for row in range(0, 4):
            finished = False
            for col in range(7, -1, -1):
                if self.board.getValue((row, col)) != 0 and self.board.getBelong((row, col)) and (not finished):
                    mylist[row] = self.board.getValue((row, col))#按照边界填充我方列表
                    finished = True
        for row in range(0, 4):
            finished = False
            for col in range(0, 8):
                if self.board.getValue((row, col)) != 0 and (not self.board.getBelong((row, col))) and (not finished):
                    yourlist[row] = self.board.getValue((row, col))#按照边界填写对方列表
                    finished = True

        self.board = current_board # 棋盘恢复
        current_board = self.board.copy()
        for i in range(0, 4):
            if mylist[i] == yourlist[i] and (mylist[i] is not None):
                if self.board.move(self.isFirst, 2):
                    self.best_direction = 2 # 如果向左合法
                    self.board = current_board # 棋盘恢复
                    return
                self.board = current_board # 棋盘恢复
                current_board = self.board.copy()
                if self.board.move(self.isFirst, 1):
                    self.best_direction = 1 # 优先级暂时按照从上到下 先看一下情况考虑是否要递归调用
                    self.board = current_board # 棋盘恢复
                    return
                self.board = current_board # 棋盘恢复
                current_board = self.board.copy()
                if self.board.move(self.isFirst, 0):
                    self.best_direction = 0
                    self.board = current_board # 棋盘恢复
                    return
                self.board = current_board
                return

```

4.2.2.1 心路历程

1. 函数说明：

评估函数做出的决定常常出现“送子观音”的情况，为了避免我方辛辛苦苦合成的棋子成为对方的间谍，开发 `athena`（处女神）函数，判断评估函数做出决定后的棋局是否存在成为“送子观音”的危险，进行有效规避。

2. 函数实现：

在此仅以先手方为例，代码如上，根据评估函数做出决定后的棋局分别得到我方和对方的边界棋子列表，一旦出现成为“送子观音”的风险，则选择其他合法方向进行合并。

`athena` 函数其实和 `contest` 函数的核心思想基本一致，都是希望避免当前的局面对我方不利，我们希望达成一个当前局面的优先级大于迭代几局的结果，但是事后反思发现评估函数的作用正是在此，突破人脑的局限，对多层棋局进行模拟，所以这也是我们最终版代码并没有选用 `athena` 函数的原因。

4.2.3 invade 函数

```

def invade(self):
    lst = [(0, 4), (1, 5), (2, 6), (3, 7)] # 用于转换先后手时列数不同的辅助列表
    # 对方即将合成较大棋子时若可以阻挠则选取其路径上的空格
    value = 5 # 用于判断棋子级别是否达到标准
    position = None
    x = 1 if self.isFirst else 0 # 用于选取对方棋盘
    for row in range(4): # 遍历对方的每一行
        valueList = [self.board.getValue((row, column)) for column in range(lst[0][x], lst[3][x] + 1)]
        # 若对方某一行中有两个等级大于等于5的棋子，且中间为空格，则取出该空格的位置
        # 若存在不止一个这样的位置，则选取等级更大时的空格位置
        if valueList[0] == valueList[2] and valueList[1] == 0 and valueList[0] >= value:
            value = valueList[0]
            position = (row, lst[1][x])
        elif valueList[1] == valueList[3] and valueList[2] == 0 and valueList[1] >= value:
            value = valueList[0]
            position = (row, lst[2][x])
        elif valueList[0] == valueList[3] and valueList[1] == 0 and valueList[2] == 0 and valueList[0] >= value:
            value = valueList[0]
            position = (row, lst[1][x])
    for column in range(lst[0][x], lst[3][x] + 1): # 遍历对方的每一列
        valueList = [self.board.getValue((row, column)) for row in range(4)]
        # 若对方某一列中有两个等级大于等于5的棋子，且中间为空格，则取出该空格的位置
        # 若存在不止一个这样的位置，则选取等级更大时的空格位置
        if valueList[0] == valueList[2] and valueList[1] == 0 and valueList[0] >= value:
            value = valueList[0]
            position = (1, column)
        elif valueList[1] == valueList[3] and valueList[2] == 0 and valueList[1] >= value:
            value = valueList[0]
            position = (2, column)
        elif valueList[0] == valueList[3] and valueList[1] == 0 and valueList[2] == 0 and valueList[0] >= value:
            value = valueList[0]
            position = (1, column)
    if position != None:
        return position

```

```

# 选取对方空格列表中的孤岛
emptyList = self.board.getNone(self.isFirst)
directionList = [(-1, 0), (1, 0), (0, -1), (0, 1)] # 计算空格的上下左右位置时的辅助列表
islandList = [] # 孤岛列表
for cell in emptyList:
    if cell[0] <= 2 and cell[0] >= 1 and cell[1] <= lst[2][x] and cell[1] >= lst[1][x]: # 当空格在对方领域中间时
        num = 0 # 计算该空格周围的棋子数
        for i in range(4):
            newrow = cell[0] + directionList[i][0]
            newcol = cell[1] + directionList[i][1]
            newpos = (newrow, newcol)
            if self.board.getValue(newpos) != 0 and self.board.getValue(newpos) != 1:
                num += 1
        if num == 4:
            islandList.append(cell)
    elif ((cell[0] == 0 or cell[0] == 3) and cell[1] <= lst[2][x] and cell[1] >= lst[1][x]
          or cell[0] <= 2 and cell[0] >= 1 and (cell[1] == lst[0][x] or cell[1] == lst[3][x])): # 当空格在对方领域非常角的边界时
        num = 0 # 计算该空格周围的棋子数
        for i in range(4):
            newrow = cell[0] + directionList[i][0]
            newcol = cell[1] + directionList[i][1]
            newpos = (newrow, newcol)
            if (newpos[0] <= 3 and newpos[0] >= 0 and newpos[1] <= lst[3][x] and newpos[1] >= lst[0][x]) # newpos在对方领域内
                and self.board.getValue(newpos) != 0 and self.board.getValue(newpos) != 1:
                num += 1
        if num == 3:
            islandList.append(cell)
    elif ((cell[0] == 0 or cell[0] == 3) and (cell[1] == lst[0][x] or cell[1] == lst[3][x])): # 当空格在对方领域的角落时
        num = 0 # 计算该空格周围的棋子数
        for i in range(4):
            newrow = cell[0] + directionList[i][0]
            newcol = cell[1] + directionList[i][1]
            newpos = (newrow, newcol)
            if (newpos[0] <= 3 and newpos[0] >= 0 and newpos[1] <= lst[3][x] and newpos[1] >= lst[0][x]) # newpos在对方领域内
                and self.board.getValue(newpos) != 0 and self.board.getValue(newpos) != 1:
                num += 1
        if num == 2:
            islandList.append(cell)
    if islandList != []:
        return islandList[0]
    else:
        return None

```

4.2.3.1 心路历程

在优化的过程中，我们认为，当对方有较大的棋子即将合并而中间有空格时，下在这个空额可以阻碍对方合成更大的棋子；以及当对方有一个孤岛（周围均为对方棋子的空格）时，如果下在这个空格的位置，无论对方接下来往哪个方向进行合并，都能保持我们的棋子在一定时间在其内部，从而对对方的合并造成阻碍。`Invade` 函数是对我们一开始持续了很久的相对保守的策略的进行的一次尝试，加入了 `invade` 函数之后，大部分时候决策仍然继承先前的保守，但在能对对方造成较大阻碍时则采取了进攻的姿态。然而写出 `invade` 函数之后，我们发现其有效的时刻并不是很多，尤其是孤岛的情形，很少出现。对当前局面进行预判的方式在后来也被证明是不如搜索树评估的，于是最后我们放弃了 `invade` 函数。

4.2.4 最初版评估函数

此函数为客观评价函数，即输入参数 阵营(isFirst) ，输出该阵营在此棋面下的 value，不附带有主观性，敌我均可用。将 isFirst 代表的阵营称为对象方，另一方则称为对方（这也深刻体现了该函数的客观性）。isFirst 为 bool 值，isFirst = True 相当于 isFirst = 1，isFirst = False 相当于 isFirst = -1

4.2.4.1 总函数：estimate

```

def estimateValue(self, isFirst): # 计算先手或后手方的棋面value
    smoothnessWeight = 0.1 #####平滑性权重
    monotonicityWeight = 0.1 #####单调性权重
    emptyCellsWeight = 0.1 #####空格数权重
    islandsWeight = 0.1 #####孤岛数权重
    figureWeight = 0.1 #####棋子级别权重
    invadeWeight = 0 #####入侵对方时的权重
    value = smoothnessWeight * self.smoothnessValue(isFirst) + \
            monotonicityWeight * self.monotonicityValue(isFirst) + \
            emptyCellsWeight * self.emptyCellsValue(isFirst) + \
            islandsWeight * self.islandsValue(isFirst) + \
            figureWeight * self.figureValue(isFirst) + \
            invadeWeight * self.invadeValue(isFirst) #####总value
    return value

```

4.2.4.1.1 构建思路：

计算当前棋盘上各种参数（如空格数、棋子级别等）贡献的 value，加权求和得到最终的 value

4.2.4.2 棋子级别函数：figurevalue

```

def figureValue(self, isFirst): # 将对象方棋子的级别加权求和 得到 棋子级别大小和数量提供的value
    import math
    selfSide = isFirst
    selfChessmanList = self.board.getScore(selfSide) # 对象方所有棋子级别按升序排成的列表
    figurevalue = 0
    for level in selfChessmanList: # 对每个棋子计算其value并加入到figurevalue中
        '''此处计算式可更改 但是应有value(2*level)>2*value(level)'''
        figurevalue += math.pow(2.2, level)
    return figurevalue

```

4.2.4.2.1 构建思路：

考虑对象方棋子本身贡献的 value 时，要同时考虑棋子的最高级别是多少和各级别棋子的数量。由于高级别的棋子贡献更大，所以在计算 value 时，应保证一个高级别的棋子的 value 应比多个低级别的棋子的 value 要大。因此选择每个棋子贡献的 value 为棋子级别 level 的 n 次方。（n>2）

4.2.4.3 空格数函数: emptyCellsValue

```

def emptyCellsValue(self, isFirst): # 计算对象方空位数所提供的value
    import math
    selfSide = isFirst
    selfEmptyCellsNums = len(self.board.getNone(selfSide)) # 对象方空位数
    '''此处计算式可更改，但是应有：在空格数较少时，value对空格数的导数较大，空格数较大时，value对空格数的导数较小'''
    if selfEmptyCellsNums != 0:
        emptyCellsValue = math.log(selfEmptyCellsNums, 2) # 这里有定义域问题，如果selfEmptyCellsNums=0则会报错，故增加一种情况。
        return emptyCellsValue
    return 0

```

4.2.4.3.1 构建思路：

对象方棋盘上的空格数越多，对象方棋子的机动性就越大，越不容易被卡位。当空位较多时，增加一个空位带来的 value 增量应该比较小，因为整体的机动性变化不明显。而当空位较少时，增加一个空位带来的 value 增量应该比较大，因为这时候每个空位都很关键。因此选择对数函数，使得空位数较少时，value 对空位数的导数较大，空位数较大时，value 对空位数的导数较小。

4.2.4.4 孤岛函数: islandValue

```

def islandValue(self, isFirst):
    import math
    selfSide = isFirst
    selfEmptyCellsPosList = self.board.getNone(selfSide) # 获得对象方所有空格位置的列表
    islandNums = 0 # 孤岛数计数
    directionList = [(-1, 0), (1, 0), (0, -1), (0, 1)] # 计算空格的上下左右位置时的辅助列表
    for cell in selfEmptyCellsPosList: # 考虑每一个对象方空格
        isIsland = True # 用于判断空格是否为孤岛
        i = 0 # 计数器
        # 遍历此空格上下左右的格子
        while i <= 3 and isIsland:
            newrow = cell[0] + directionList[i][0] # 空格周围格子所在的行
            newcol = cell[1] + directionList[i][1] # 空格周围格子所在的列
            newpos = (newrow, newcol) # 空格周围格子的坐标

            if newrow < 0 or newrow > 3:
                # 行数不在对象方棋盘范围内
                pass
            elif (newcol - (1 - isFirst) * 4) < 0 or (newcol - (1 - isFirst) * 4) > 3:
                # 列数不在对象方棋盘范围内
                pass
            elif self.board.getValue(newpos) <= 1: # 若 空格旁的这个格子 级别小于等于1，则其必定是对象方的'2'或者空格
                isIsland = False

            i += 1

        if isIsland:
            islandNums += 1
    '''此处可更改表达式，但应保证value为负值'''
    islandValue = -math.pow(islandNums, 1.2)
    return islandValue

```

4.2.4.4.1 构建思路:

对于对象方棋盘，若一个空格 周围没有 其他空格 或者 级别为 1 的棋子 时，将其称为孤岛，它卡在高级别棋子当中，当级别为 1 的棋子落在其上时，会对整体的平滑性与单调性造成较大影响。因此它的存在会使得总 `value` 下降。

4.2.4.5 平滑性函数: `smoothnessValue`

```

def smoothnessValue(self, isFirst): # 计算对方棋子的平滑性，即同一方向上相邻棋子级别差异是否明显，不平滑带来的value应为负值
    selfSide = isFirst
    smoothnessvalue = 0 # 初始化平滑性value
    directionList = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    # 遍历棋盘上每一个格子（因为对方棋子可能在对方棋盘内有分布）
    for row in range(0, 4):
        for col in range(0, 8):
            # 如果此格为对方的棋子，则要考虑其与邻近的对方棋子贡献的平滑性value
            if selfSide == self.board.getBelong((row, col)) and \
                self.board.getValue((row, col)) != 0:
                # level代表此格棋子的级别
                level = self.board.getValue((row, col))
                # 遍历此格棋子上下左右的格子
                for i in range(4):
                    newrow = row + directionList[i][0]
                    newcol = col + directionList[i][1]
                    # 如果此方向的格子超界，则不用考虑此方向
                    if (newrow) < 0 or (newrow) > 3 or \
                        (newcol) < 0 or (newcol) > 7:
                        pass
                    # 如果此方向的格子上为对方棋子，则要考虑平滑性影响
                    elif selfSide == self.board.getBelong((newrow, newcol)) and \
                        self.board.getValue((newrow, newcol)) != 0:
                        newLevel = self.board.getValue((newrow, newcol))
                        # deltaLevel代表两棋子的级别差
                        deltaLevel = abs(level - newLevel)
                        '''此处表达式可以修改，但应保证smoothnessvalue非正...'''
                        smoothnessvalue -= deltaLevel ** 2
    return smoothnessvalue

```

4.2.4.5.1 构建思路:

在对象方棋盘内，考虑一行或者一列，当相邻的棋子级别之差较小时，它们就较容易合并，此时这一行或者一列的平滑性较好。因此两相邻的棋子级别差越大，它俩贡献的 `value` 就越小。

4.2.4.6 进攻函数：invadeValue

```

def invadeValue(self, isFirst): # 当对象方棋子在对方棋盘上有分布时，考虑其贡献的value
    selfSide = isFirst
    enemySide = not isFirst
    invadevalue = 0
    directionList = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    # 对对方的棋盘遍历,
    for row in range(4):
        for col in range(isFirst * 4, 4 + isFirst * 4):
            # 找出对方棋盘中的对象方棋子
            if selfSide == self.board.getBelong((row, col)):
                # level 为该棋子的级别
                level = self.board.getValue((row, col))

                # 对该棋子四个方向的格子进行遍历
                for i in range(4):
                    newrow = row + directionList[i][0]
                    newcol = col + directionList[i][1]
                    # 如果此方向的格子超界，则不用考虑此方向
                    if newrow < 0 or newrow > 3 or \
                        newcol < 0 or newcol > 7:
                        pass

                    # 如果此方向的格子中为对方的等级别棋子，则要考虑入侵带来的value，若对象方为先手，则对象方可以选择吃掉对方棋子，故而贡献的value>0，反之<0
                    elif enemySide == self.board.getBelong((newrow, newcol)) and \
                        self.board.getValue((newrow, newcol)) == level:
                        # 对象方为先手手次定了value的正负
                        '''此处的表达式可更改，但应保证先手手的value正负号相反'''
                        invadevalue += (isFirst - 0.5) * level
                        # 对'new'棋子的四个方向进行遍历，若存在对方的高一级别的棋子'newnew'：对象方的棋子吃掉'new'后，会被'newnew'吃掉，value减小；对方的棋子'new'吃掉对象方棋子，value减小
                        # 因此无论如何value均减小
                        for j in range(4):
                            newnewrow = newrow + directionList[j][0]
                            newnewcol = newcol + directionList[j][1]
                            if newnewrow < 0 or newnewrow > 3 or \
                                newnewcol < 0 or newnewcol > 7:
                                pass
                            elif enemySide == self.board.getBelong((newnewrow, newnewcol)) and \
                                self.board.getValue((newnewrow, newnewcol)) == level + 1:
                                '''此处表达式可更改，但应保证value要减小，且减小量要比 对象方为先手时的value增量 大'''
                                invadevalue -= level

# 如果此方向为对方空格，则对方的等级别棋子可能会隔着空格击杀，此时value会减小
elif enemySide == self.board.getBelong((newrow, newcol)) and \
    self.board.getValue((newrow, newcol)) == 0:
    stop = False
    # 沿该方向前进，若出界，则停下；若碰到对方空格，则继续前进；若碰到对方等级别棋子，则停下并且value减小；其他情况下均停下
    while not stop:
        newrow = newrow + directionList[i][0]
        newcol = newcol + directionList[i][1]
        # 出界
        if newrow < 0 or newrow > 3 or \
            newcol < 0 or newcol > 7:
            stop = True
        # 碰到对方等级别棋子
        elif enemySide == self.board.getBelong((newrow, newcol)) and \
            self.board.getValue((newrow, newcol)) == level:
            '''此处的表达式可修改，但应保证value减小'''
            invadevalue -= 0.5 * level
            stop = True
        # 碰到对方空格，前进四
        elif enemySide == self.board.getBelong((newrow, newcol)) and \
            self.board.getValue((newrow, newcol)) == 0:
            stop = False
        # 其他情况，停下
        else:
            stop = True

return invadevalue

```

4.2.4.6.1 构建思路：

当对象方棋子在对方棋盘内有分布时，依据情况的不同来计算其贡献的 value: (1) 其周围存在有对方的等级别棋子时，若对象方为先手，则对象方可以选择吃掉对方棋子，故而贡献的 value>0，反之<0。 (2) 其他的一些特殊情况另外考虑。

4.2.4.7 单调性函数：monotonicityValue

```

def monotonicityValue(self, isFirst): # 单调性value评估，仅考虑对象方棋盘内的棋子，且对方棋子入侵时，可认为其所在的行与列的value贡献均为0
    selfSide = isFirst
    monotonicityvalue = 0
    # 遍历对象方棋盘每一行
    for row in range(4):
        # 引入单调性因子，表征单行或者单列的单调性好坏
        monotonicityFactor = 0
        # 列数从对象方棋盘的最左列到最右列遍历，即遍历这一行的所有对象方格子
        col = (1 - isFirst) * 4
        while col < (1 - isFirst) * 4 + 3:
            # 当前格子的级别
            currentValue = self.board.getValue((row, col))
            # 下一格的坐标
            nextrow = row
            nextcol = col + 1
            # 有内鬼，则终止交易，转至下一行
            if selfSide != self.board.getBelong((row, col)) or \
                selfSide != self.board.getBelong((nextrow, nextcol)):
                monotonicityFactor = 0
                break # 只是跳出while循环
            # 无内鬼，计算下一格的级别，单调性因子变化
            else:
                nextValue = self.board.getValue((nextrow, nextcol))
                '''此表达式可以更改'''
                monotonicityFactor += nextValue - currentValue
            # 列数加一，继续下一格
            col += 1
        # 此行的单调性因子计算完毕，加至value中
        '''此表达式可以更改'''
        monotonicityvalue += abs(monotonicityFactor)
    
```

```

# 遍历对象方棋盘每一列，同上一种情况
for col in range((1 - isFirst) * 4, (1 - isFirst) * 4 + 4):
    monotonicityFactor = 0
    row = 0
    while row < 3:
        currentValue = self.board.getValue((row, col))
        nextrow = row + 1
        nextcol = col
        if selfSide != self.board.getBelong((row, col)) or \
            selfSide != self.board.getBelong((nextrow, nextcol)):
            monotonicityFactor = 0
            break
        else:
            nextValue = self.board.getValue((nextrow, nextcol))
            monotonicityFactor += nextValue - currentValue
        row += 1
    monotonicityvalue += abs(monotonicityFactor)
return monotonicityvalue

```

4.2.4.7.1 构建思路：

当一行或者一列内的棋子级别是单调增加或者减少时，其越容易进行合并，因此会贡献 value。对于对象方棋盘内的每一行和每一列进行单调性评估，仅当该列或行的棋子均为对象方棋子且级别单调变化时，才计算 value 贡献。

4.3 建议与设想

建议：增大最大思考时间，比如从 $5\text{s} \rightarrow 10\text{s}$ ，这样可以使得决策更优化，比赛的观赏性更强。

寄语：希望选修这门课的学弟学妹们，多在数算上下功夫，就一定能获得更多的知识与回报，课程虽卷，但最终一定会收获满满。

大作业设想：参考贪吃蛇大作战进行智能对战。

5 致谢

感谢陈斌老师一学期以来干货满满的教授，让我们对 `python` 语言的掌握更加熟练，对算法有了更深刻的理解。

感谢各位助教一学期以来不断为我们答疑解惑，提供新知。

感谢技术组的各位大佬，为我们提供了人性化的平台和接口，为大作业比赛的建设辛勤付出。

感谢你群各位大佬，从你们的交流中我们学到了很多新的知识，获得思路的启发。

感谢远在元培的数科大佬，向我们传授了麻将程设的经验，并提供了珍贵的参考文献。

感谢在另一个小组里的室友，我们虽然是竞争对手，但却可以进行深入的探讨，做到双赢。

感谢腾讯为我们提供了交流合作的平台。

感谢**让我们能够获取更多的海外学术知识（危）。

感谢 `python` 之父。

感谢北大。

6 参考文献

<https://www.cnblogs.com/mumuxinfei/p/4415352.html>

<https://blog.csdn.net/wenjianmuran/article/details/90633418>

https://blog.csdn.net/qq_32767041/article/details/80634244

<https://blog.csdn.net/wsh596823919/article/details/80753087>

https://blog.csdn.net/qq_44516149/article/details/106399289

<https://www.cnblogs.com/mumuxinfei/p/4415352.html>

<https://www.zhihu.com/question/54217135>

参考书籍：《快乐写游戏 轻松学编程：PC 游戏编程》 by 陈其