

程序报告

姓名：秦嘉俊 学号：3210106182

一、问题重述

黑白棋 (Reversi)，又称翻转棋，是一个经典的策略性游戏。

一般棋子双面为黑白两色，故称“黑白棋”。因为行棋之时将对方棋子翻转，则变为己方棋子，故又称“翻转棋”(Reversi)。

黑白棋使用 8x8 大小的棋盘，由两人执黑子和白子轮流下棋，最后子多方为胜方。

1.1 黑白棋规则

1. 黑方先行，双方交替下棋。
2. 一步合法的棋步包括：
 - (a) 在一个空格处落下一个棋子，并且翻转对手一个或多个棋子；
 - (b) 新落下的棋子必须落在可夹住对方棋子的位置上，对方被夹住的所有棋子都要翻转过来，可以是横着夹，竖着夹，或是斜着夹。夹住的位置上必须全部是对手的棋子，不能有空格；
 - (c) 一步棋可以在数个（横向，纵向，对角线）方向上翻棋，任何被夹住的棋子都必须被翻转过来，棋手无权选择不去翻某个棋子。
3. 如果一方没有合法棋步，也就是说不管他下到哪里，都不能至少翻转对手的一个棋子，那他这一轮只能弃权，而由他的对手继续落子直到他有合法棋步可下。
4. 如果一方至少有一步合法棋步可下，他就必须落子，不得弃权。
5. 棋局持续下去，直到棋盘填满或者双方都无合法棋步可下。
6. 如果某一方落子时间超过 1 分钟或者连续落子 3 次不合法，则判该方失败。

1.2 任务

使用 Python 算法实现 MCTS(Monte Carlo Tree Search, 即蒙特卡洛树搜索) 实现 Mini AlphaGo for Reversi, 并提供简单的图形界面用于人机博弈交互。

二、设计思想

2.1 蒙特卡洛树搜索

对搜索算法进行优化以提高搜索效率基本上是在解决如下两个问题：优先扩展哪些节点以及放弃扩展哪些节点，综合来看也可以概括为如何高效地扩展搜索树。如果将目标稍微降低，改为求解一个近似最优解，则上述问题可以看成是如下探索性问题：算法从根节点开始，每一步动作为选择（在非叶子节点）或扩展（在叶子节点）一个孩子节点。可以用执行该动作后所收获奖励来判断该动作优劣。奖励可以根据从当前节点出发到达目标路径的代价或游戏终局分数来定义。算法会倾向于扩展获得奖励较高的节点。

算法事先不知道每个节点将会得到怎样的代价（或终局分数）分布，只能通过采样式探索来得到计算奖励的样本。由于这个算法利用蒙特卡洛法通过采样来估计每个动作优劣，因此它被称为蒙特卡洛树搜索（Monte-Carlo Tree Search）算法 [Kocsis 2006]

2.1.1 选择 (Section)

选择指算法从搜索树的根节点开始，向下递归选择子节点，直至到达叶子节点或者到达具有还未被扩展过的子节点的节点 L 。这个向下递归选择过程可由 UCB 算法来实现，在递归选择过程中记录下每个节点被选择次数和每个节点得到的奖励均值。

我们将节点分成三类：

未访问：还没有评估过当前局面

未完全展开：被评估过至少一次，但是子节点（下一步的局面）没有被全部访问过，可以进一步扩展

完全展开：子节点被全部访问过

我们找到目前认为「最有可能走到的」一个未被评估的局面（双方都很聪明的情况下），并且选择它。什么节点最有可能走到呢？最直观的想法是直接看节点的胜率（赢的次数/访问次数），哪个节点最大选择哪个，但是这样是不行的！因为如果一开始在某个节点进行模拟的时候，尽管这个节点不怎么好，但是一开始随机走子的时候赢了一盘，就会一直走这个节点了。因此我们创造了一个函数：

$$UCT(v_i, v) = \frac{Q(v_i)}{N(v_i)} + c \sqrt{\frac{\log N(v)}{N(v_i)}}$$

其中 $Q(v)$ 是该节点赢的次数， $N(v)$ 是该节点模拟的次数， C 是一个常数。因此我们每次选择的过程如下——从根节点出发，遵循最大最小原则，每次选择己方 UCT 值最优的一个节点，向下搜索，直到找到一个「未完全展开的节点」，根据我们上面的定义，未完全展开的节点一定有未访问的子节点，随便选一个进行扩展。

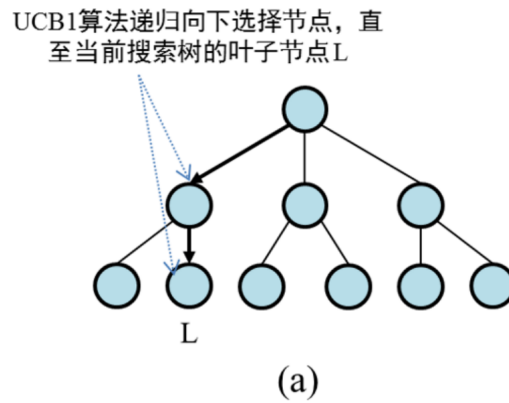


图 1: 选择

2.1.2 扩展 (Expansion)

如果节点 L 不是一个终止节点（或对抗搜索的终局节点），则随机扩展它的一个未被扩展过的后继边缘节点 M.

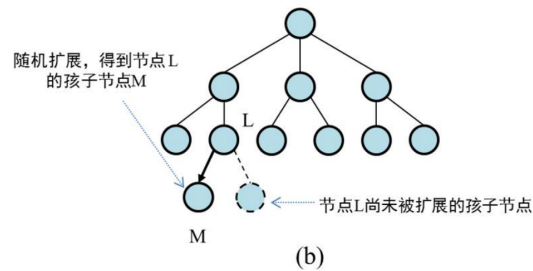


图 2: 扩展

2.1.3 模拟 (Simulation)

从节点 M 出发，模拟扩展搜索树，直到找到一个终止节点。模拟过程使用的策略和采用 UCB 算法实现的选择过程并不相同，前者通常会使用比较简单的策略，例如使用随机策略。

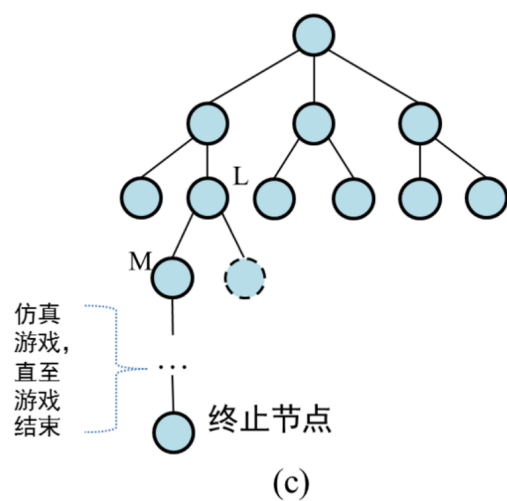


图 3: 模拟

2.1.4 反向传播 (Back Propagation)

用模拟所得结果（终止节点的代价或游戏终局分数）回溯更新模拟路径中 M 以上（含 M ）节点的奖励均值和被访问次数。

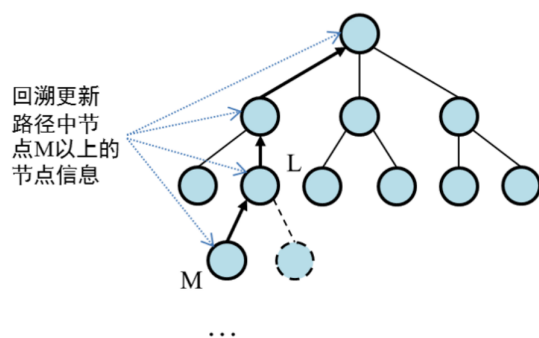


图 4: 反向传播

2.2 黑白棋中的策略

传统黑白棋策略包括以下几类：

1. 贪心策略

每一步走子都选择使得棋盘上子最多的一步，而不考虑最终的胜负；

2. 确定子策略

某些子一旦落子后就再也不会被翻回对方的子，最典型的是四个角上的子，这类子被称为确定子 (Stable Discs)。每一步走子都选择使得棋盘上己方的确定子最多的一步。

3. 位置优先策略

考虑到角点的重要性，把棋盘上的每一个子都赋予一个优先级，每一步从可走子里选择优先级最高的一个子。

4. 机动性策略 (mobility)

黑白棋每一步的可走子都是有限的，机动性策略是指走子使得对手的可走子较少，从而逼迫对手不得不走出差的一步 (bad move)，使得自己占据先机。

5. 消失策略 (evaporation, less is more)

在棋盘比试的前期，己方的子越少往往意味着局势更优。因此在前期可采用使己方的子更少的走子。

6. 奇偶策略 (parity)

走子尽量走在一行或一列有奇数个空子的位置。以上只列举了一些常见的黑白棋策略或原则，事实上还有很多更为复杂的策略，此处不进行列举。

但在实现中，贪心策略的效果并不理想，局部最优反而可能带来后续的问题。其他策略大都不利于实现，因此这里我们主要采用位置优先策略：即给每个位置赋予优先级，当需要落子时我们优先考虑优先级高的位置。具体的优先级如下：

1	9	2	4	4	2	9	1
9	10	8	7	7	8	10	9
2	8	3	5	5	3	8	2
4	7	5	6	6	5	7	4
4	7	5	6	6	5	7	4
2	8	3	5	5	3	8	2
9	10	8	7	7	8	10	9
1	9	2	4	4	2	9	1

优先级这样指定的主要原因是角、边不容易被吃子，而其他位置容易被吃。同时综合考虑了限制对手行动力、扩大自己稳定子，得出的上文的优先级。我们这里将这种优先级的做法用于我们模拟的过程。

三、代码内容

3.1 Game 部分

在 Game.py 中，我新创建的一个类 SimulationGame，它的实现方式基本和 Game 相同，主要用于蒙特卡洛树搜索中模拟部分。当我们要模拟后续结果时，就可以直接通过调用 SimulationGame.run() 得到答案。代码如下：(SimulateGame 的其他方法和 Game 相同，这里不再重复)

```
1      class SimulateGame(object):
2          """
3          用于蒙特卡洛树 Simulate 时进行测试。
4
5          """
6
7          def __init__(self, black_player, white_player, board=Board(),
8                      current_player=None):
9              self.board = deepcopy(board) # 棋盘
10             # 定义棋盘上当前下棋棋手，先默认是 None
11             self.black_player = black_player # 黑棋一方
12             self.white_player = white_player # 白棋一方
13             self.black_player.color = "X"
14             self.white_player.color = "O"
15             self.current_player = current_player
16
17         def run(self):
18             """
19             运行游戏
20             :return:
21             """
22             # 定义统计双方下棋时间
23             total_time = {"X": 0, "O": 0}
24             # 定义双方每一步下棋时间
25             step_time = {"X": 0, "O": 0}
26             # 初始化胜负结果和棋子差
27             winner = None
28             diff = -1
29
30             # 游戏开始
31             # print('\n=====开始游戏!=====\n')
32             # 棋盘初始化
33             # self.board.display(step_time, total_time)
34             while True:
```

```

33         # 切换当前玩家,如果当前玩家是 None 或者白棋 white_player, 则返回
           黑棋 black_player;
34         # 否则返回 white_player。
35         self.current_player = self.switch_player(self.
           black_player, self.white_player)
36         start_time = datetime.datetime.now()
37         # 当前玩家对棋盘进行思考后, 得到落子位置
38         # 判断当前下棋方
39         color = "X" if self.current_player == self.
           black_player else "O"
40         # 获取当前下棋方合法落子位置
41         legal_actions = list(self.board.get_legal_actions(
           color))
42         # print("%s合法落子坐标列表: "%color,legal_actions)
43         if len(legal_actions) == 0:
44             # 判断游戏是否结束
45             if self.game_over():
46                 # 游戏结束, 双方都没有合法位置
47                 winner, diff = self.board.get_winner() #
                   得到赢家 0,1,2
48                 break
49             else:
50                 # 另一方有合法位置,切换下棋方
51                 continue
52
53         board = deepcopy(self.board._board)
54         action = action = self.current_player.get_move(self.
           board)
55         if action is None:
56             continue
57         else:
58             self.board._move(action, color)
59             if self.game_over():
60                 winner, diff = self.board.get_winner() #
                   得到赢家 0,1,2
61                 break
62         return winner, diff

```

3.2 PosFirstPlayer 类

PosFirstPlayer, 即采用位置优先策略的玩家, 在初始化时我们将位置按照优先级归类, 并共同构成一个 list. 当需要落子时, 我们按照优先级从低到高的顺序依次迭代, 如果能在高优先级中找到一个可以落子的位置, 我们就直接落子于此。否则继续遍历较低的优先级, 直到能够落子或者遍历结束无子可下为止。

```
1      class PosFirstPlayer(object):
2          def __init__(self, color):
3              """
4              PosFirstPlayer
5              :param roxanne_table: 从上到下依次按落子优先级排序
6              :param color: 执棋方
7              """
8
9              self.roxanne_table = [
10                 ['A1', 'H1', 'A8', 'H8'],
11                 ['C3', 'F3', 'C6', 'F6'],
12                 ['C4', 'F4', 'C5', 'F5', 'D3', 'E3', 'D6', 'E6'],
13                 ['A3', 'H3', 'A6', 'H6', 'C1', 'F1', 'C8', 'F8'],
14                 ['A4', 'H4', 'A5', 'H5', 'D1', 'E1', 'D8', 'E8'],
15                 ['B3', 'G3', 'B6', 'G6', 'C2', 'F2', 'C7', 'F7'],
16                 ['B4', 'G4', 'B5', 'G5', 'D2', 'E2', 'D7', 'E7'],
17                 ['B2', 'G2', 'B7', 'G7'],
18                 ['A2', 'H2', 'A7', 'H7', 'B1', 'G1', 'B8', 'G8']
19             ]
20             self.color = color
```

PosFirstPlayer 中选择落子位置的方法 `select` 和 `get_move` 与题目给出的类的实现完全相同, 这里不再重复。

3.3 蒙特卡洛树的构建

3.3.1 基本内容

经过之前部分的分析, 我们可以发现对于蒙特卡洛搜索树的构建, 我们需要 `N`(记录目前模拟比赛次数), `Q`(记录目前模拟比赛胜利次数), `parent`(记录当前节点的上一节点, 即从哪个节点更新过来的, 这个属性在反向传播中用到), `color`(目前状态到下一状态需要放置什么颜色的棋子), `child`(记录有哪些后继节点). 对应初始化代码如下:

```
1      class Tree:
2          """
3          蒙特卡洛树的节点
```



```

4      """
5
6      def __init__(self, color, parent=None):
7          self.N = 0      # 总局数
8          self.Q = 0      # 胜利次数
9          self.parent = parent # 用于回溯
10         self.color = color
11         self.child = dict() # 这里使用 dictionary 以落子位置为 key, value 为
                               Tree 这个实例

```

3.3.2 下一次的顏色

当我们在扩展时，我们需要更新当前节点的后继。而当前节点的后继的 color 属性应该和当前节点相反，因此我们使用 Oppo 方法来实现。

```

1 def Oppo(self):
2     return 'X' if self.color == 'O' else 'O'

```

3.3.3 UCB 函数

由前文的分析可知，对一个节点能否作为我们的选择，关键就在于他的 UCB 值。因此我们在 Tree 这个类中实现计算函数 UCB 以便算出这个值。

```

1 def UCB(self):
2     return self.Q / self.N + 2 * sqrt(2 * log(self.parent.N) / self.N)

```

3.4 蒙特卡洛树搜索

对于蒙特卡洛树搜索，我们单独实现了一个类来完成所有的搜索过程。

3.4.1 初始化

对一个 MCTS 类，我们只需要两个属性 simulate_black 和 simulate_white. 它们都是采用位置优先策略的玩家的实例，在蒙特卡洛树搜索中模拟双方进行比赛。

```

1     class MCTS:
2         """
3         实现 MCTS 蒙特卡洛树搜索
4         """
5
6         def __init__(self):

```

```

7         self.simulate_black = PosFirstPlayer('X')
8         self.simulate_white = PosFirstPlayer('O')
9         self.begin_time = time()

```

3.4.2 UCTSearch

每次决策都要执行 UCTSearch 来为我们这步下在哪里提供信息。UCTSearch 的过程和蒙特卡洛树的过程相同，从根节点出发，首先调用 Select 函数选出目前最有可能获胜的方法，并将这个点进行扩展，随后利用 SimulatePolicy 进行模拟比赛，得到结果后算出选出的点的分数，并最后通过 BackPropagte 进行反向传播。

需要注意的是在模拟出答案后，正如前文所说 `v.color` 表示的是 `v` 这个状态将要放的颜色，也就是说 `v` 是处于 `oppo(color)` 这个颜色，因此在得到答案后我们需要 16 到 18 行的代码进行调整。

此外我们在最开始设置了循环，并读取时间，以保证每次落子的搜索时间不会超过 59 秒。

```

1 def UCTSearch(self, root, board):
2     while time() - self.begin_time < 59: # 每次落子最多考虑 59 秒
3         Saved_board = deepcopy(board)
4         v_select = self.Select(root, Saved_board)
5         self.Expand(v_select, Saved_board)
6         winner, diff = self.SimulatePolicy(v_select, Saved_board)
7
8         if winner == 2:
9             score = 0.5
10        elif winner == 0:
11            score = 1
12        else:
13            score = 0
14
15        if v_select.color == 'X': # v 是叶子节点，.color 表示这个状态 将要 放 什么颜色才能转到下一个状态，即 v 这个状态
16                                # 是刚刚放完 v.parent.color 而更新过来的。因此若 v 父亲放的黑色，则 v 应该属于白色方
17
18            score = 1 - score
19        self.BackPropagte(v_select, score)

```

3.4.3 Select

Select 阶段我们按照原理部分的过程，对当前节点的后继，我们找到其中 UCB 值最大的后继并进行进一步的搜索直到叶子节点。需要注意的是如果存在叶子节点没有进行过模拟比赛，那么我们直接选择这个节点，避免出现一直选择某几种方法，一开始就走错了路，但其他节点从未被选过的情况。

```
1 def Select(self, node, board):
2
3     max_UCB = -666
4     choice = None
5     if len(node.child) == 0:
6         return node
7     for i in node.child.keys():
8         son = node.child[i]
9         if son.N ≤ 1:
10             choice = i
11             break
12     else:
13         tmp = son.UCB()
14         if tmp > max_UCB:
15             max_UCB = tmp
16             choice = i
17     board._move(choice, node.color)
18     return self.Select(node.child[choice], board)
```

3.4.4 Expand

对于当前状态，所有下一步能下的点都作为可能的后继状态，我们都在搜索树中创造其对应的实例，这就是扩展。

```
1 def Expand(self, node, board):
2     """
3     扩展搜索树
4     """
5
6     childs = board.get_legal_actions(node.color)
7     for i in childs:
8         node.child[i] = Tree(node.Oppo(), node)
```

3.4.5 SimulatePolicy

假定两个都采用位置优先策略的玩家进行模拟比赛，并将结果返回。

```
1 def SimulatePolicy(self, node, board):
2     """
3     模拟比赛
4     这里我们利用 SimulateGame 进行模拟
5     """
6
7     if node.color == 'X':
8         current_player = self.simulate_white
9     else:
10        current_player = self.simulate_black
11    return SimulateGame(self.simulate_black, self.simulate_white, board,
        current_player).run()
```

3.4.6 BackPropagte

从当前叶子节点开始，沿着 `v.parent` 不断向上更新总局数和胜利次数。

```
1 def BackPropagte(self, node, score):
2     """
3     反向传播，更新父节点的数据
4     """
5     v = node
6     while v is not None:
7         v.Q += score
8         v.N += 1
9         v = v.parent
10        score = 1 - score
```

3.5 AIPlayer 类的实现

AIPlayer 即采取我们蒙特卡洛树搜索的人工智能玩家。

实验框架给出的部分这里不再重复，我们这里主要修改了他的 `run` 方法。每次决策我们先建一个根节点表示当前状态，随后就调用 `UCTSearch` 方法进行蒙特卡洛树搜索，结束后我们检查哪个后继状态走的最多，最后选择那个状态并落子。

需要注意的是，之所以走的越多的状态就是我们认为的最佳选择，是因为蒙特卡洛树搜索算法的核心就是，越优秀的节点，越有可能走，反过来就是，走得越多的节点，越优秀。

```
1     def get_move(self, board):
```

```

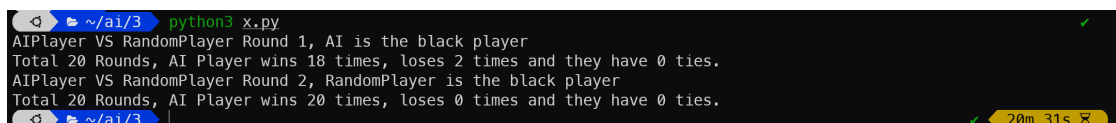
2      """
3      根据当前棋盘状态获取最佳落子位置
4      :param board: 棋盘
5      :return: action 最佳落子位置, e.g. 'A1'
6      """
7      if self.color == 'X':
8          player_name = '黑棋'
9      else:
10         player_name = '白棋'
11     print("请等一会, 对方 {}-{} 正在思考中...".format(player_name, self.color))
12
13     # -----请实现你的算法代码-----
14     Searching = MCTS()
15     root = Tree(color=self.color)
16     Searching.UCTSearch(root, deepcopy(board))
17
18     choice = root
19     max_N = -666
20     # 注意这里 走的越多 说明这个方案 越好
21     for i in root.child.keys():
22         son = root.child[i]
23         if son.N > max_N:
24             max_N = son.N
25             choice = i
26     action = choice
27     # -----
28
29     return action

```

四、实验结果

4.1 AIPlayer VS RandomPlayer

共 20 次测试, AIPlayer(黑) 16:2:2 RandomPlayer, RandomPlayer(黑) 2:1:17 AIPlayer



```

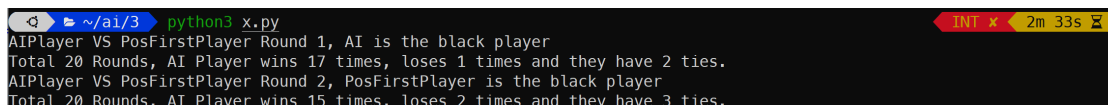
~/ai/3 python3 x.py
AIPlayer VS RandomPlayer Round 1, AI is the black player
Total 20 Rounds, AI Player wins 18 times, loses 2 times and they have 0 ties.
AIPlayer VS RandomPlayer Round 2, RandomPlayer is the black player
Total 20 Rounds, AI Player wins 20 times, loses 0 times and they have 0 ties.

```

图 5: AIPlayer VS RandomPlayer

4.2 AIPlayer VS PosFirstPlayer

共 20 次测试, AIPlayer(黑) 17:2:1 PosFirstPlayer, PosFirstPlayer(黑) 2:3:15 AIPlayer

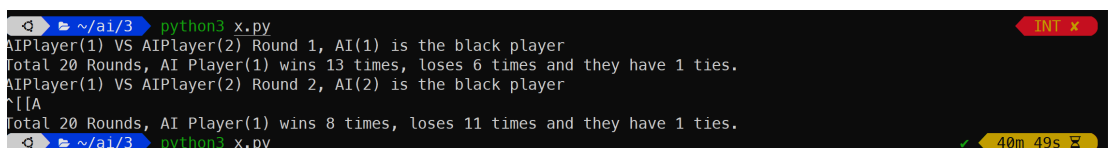


```
~/ai/3 python3 x.py
AIPlayer VS PosFirstPlayer Round 1, AI is the black player
Total 20 Rounds, AI Player wins 17 times, loses 1 times and they have 2 ties.
AIPlayer VS PosFirstPlayer Round 2, PosFirstPlayer is the black player
Total 20 Rounds, AI Player wins 15 times, loses 2 times and they have 3 ties.
```

图 6: AIPlayer VS PosFirstPlayer

4.3 AIPlayer1 VS AIPlayer2

共 20 次测试, AIPlayer1(黑) 13:1:6 AIPlayer2, AIPlayer2(黑) 11:1:8 AIPlayer1



```
~/ai/3 python3 x.py
AIPlayer(1) VS AIPlayer(2) Round 1, AI(1) is the black player
Total 20 Rounds, AI Player(1) wins 13 times, loses 6 times and they have 1 ties.
AIPlayer(1) VS AIPlayer(2) Round 2, AI(2) is the black player
^[[A
Total 20 Rounds, AI Player(1) wins 8 times, loses 11 times and they have 1 ties.
```

图 7: AIPlayer1 VS AIPlayer2

可以看出我们的人工智能算法还是有比较不错的性能的, 同时在两个 AI 对战时, 谁先下谁胜率高, 这也符合我们的认知。

五、总结

蒙特卡洛树是一个非常重要且经典的搜索算法, 在这次亲身实践中我更好地理解了算法的本质, 熟悉了算法的过程和细节。同时在实现算法之后能够自己与自己写的 AI 对弈, 这对于实验的趣味性是很大的提升。

在完成本次实验的过程中, 我的算法在基本实现后基本达到预期, 但在与系统测试的 AI 对战时, 结果始终不如人意 (只能战胜初级难度), 于是我开始寻求优化, 在每次进行 UCTSearch 时, 我发现我们可以将时间上限调高到 59s(规则要求单步思考超过一分钟即判负), 从而尽可能多地模拟比赛, 以期找到最优的下法。此外, 我们还可以更改优先级矩阵, 通过数学方法寻求更好的优先级顺序。

但在经过反复调参、优化后发现自己的 AI 离战胜中级难度还是有一定距离 (初级, 高级可过, 但中级一直卡住), 这让我对算法本身产生了怀疑, 想起来学长学姐们说的 AB 剪枝, 或许使用这种方法可能有更好的效果。但无论如何, 我已经感受到蒙特卡洛的魅力和作用。

在本次实验之外, 我认为可以考虑给算法加入机器学习的方法。本身黑白棋的规模相对围棋来说较小, 棋盘仅有 8*8, 而且在本地运行时我发现有些局的结果十分类似。如果我们

能让我们的人工智能不断地自我对弈，并加以记忆和学习，这样相信我们的黑白棋程序能变得更加强大。

最后，期待下次实验带给我新的惊喜。