

程序报告

一、问题重述

用蒙特卡洛树算法编写黑白棋AIplayer

规则如下：

1. 黑方先行，双方交替下棋。
2. 一步合法的棋步包括：
 - 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；
 - 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；
 - 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。

二、设计思路

蒙特卡洛树一般在以下场景中使用：

智能体对环境没有完整的认识，事先并不知道每种行动的优劣，只有他们尝试过了之后才能确定。

程序的思路大致为：

对于当前节点，判断其是否为叶节点。

- 不是
找到当前节点value最大的子节点，再次判断
- 是
判断该节点的被访问次数是否是0。
 - 是
计算其value并且迭代累加回去。
 - 不是
枚举当前节点所有可能的动作并添加到树中。
令第一个新节点成为当前节点。
计算其value值并向上累加。

在本程序中，分别编写了node类和mcts类来实现此算法。

具体逻辑在代码注释中有所表现。

三、代码内容

mctsNode.py

```
1  from math import log, sqrt
2
3
4  class Node:
5      def __init__(self, board, parent, color, action):
6          """
7              构造
8
9              :param board:    当前棋盘状态
10             :param parent:   父节点
11             :param color:    执子方
12             :param action:   来到此node的action
13             :return:
14             """
15             self.parent = parent
16             self.board = board
17             self.color = color
18             self.prevAction = action
19             self.children = [] # 子节点列表
20             self.visit_times = 0 # 被访问次数
21             self.unVisitActions = list(board.get_legal_actions(color)) # 目前的
22             # 棋盘状态下合法的下法
23
24             self.isOver = self.gameOver(board) # 是否双方都没有合法操作
25
26             if (self.isOver == 0) and (len(self.unVisitActions) == 0): # 我方无
27             # 合法操作，但是对方有合法操作
28                 self.unVisitActions.append("noway")
29
30             self.reward = {'X': 0, 'O': 0}
31             self.bestVal = {'X': 0, 'O': 0}
32
33     @staticmethod
34     def gameOver(board):
35         """
36             如果双方都没有合法落子了，游戏结束
37
38             :param board:    当前棋盘状态
39             :return:         true/false 是否游戏无法继续
40             """
41             l1 = list(board.get_legal_actions('X'))
42             l2 = list(board.get_legal_actions('O'))
43             return len(l1) == 0 and len(l2) == 0
44
45     def calcBestVal(self, balance, color):
46         """
47             如果双方都没有合法落子了，游戏结束
48
49             :param balance:   参数
50             :param color:     当前执子
51             :return:
52             """
53         if self.visit_times == 0:
```

```

52         print("-----")
53         print("oops!visit_times==0!")
54         self.board.display()
55         print("-----")
56     # 公式
57     c1 = self.reward[color] / self.visit_times
58     c2 = balance * sqrt(2 * log(self.parent.visit_times) /
self.visit_times)
59     self.bestVal[color] = c1 + c2

```

mcts.py

```

1  from mctsNode import Node
2  from copy import deepcopy
3  from func_timeout import FunctionTimedOut, func_timeout
4  import random
5  import math
6
7
8  class MonteCarlo:
9      # uct方法的实现
10     # return: action(string)
11     def search(self, board, color):
12         """
13         构造
14
15         :param board:    当前棋盘状态
16         :param color:    执子方
17         """
18
19         # actions是当前所有的合法落子
20         actions = list(board.get_legal_actions(color))
21         # 特殊情况: 只有一种落子, 那么直接返回这一种。
22         if len(actions) == 1:
23             return list(actions)[0]
24
25         # 创建根节点
26         newBoard = deepcopy(board)
27         root = Node(newBoard, None, color, None)
28
29         # 考虑时间限制
30         try:
31             # 测试程序规定每一步在60s以内
32             func_timeout(59, self.whileFunc, args=[root])
33         except FunctionTimedOut:
34             pass
35
36         # 返回能够得到bestValue的那个action
37         return self.best_child(root, math.sqrt(2), color).prevAction
38
39     def whileFunc(self, root):
40         """
41         构造
42
43         :param root:    根节点
44         """
45         while True:

```

```

46         # mcts four steps
47         # selection, expansion
48         expand_node = self.tree_policy(root)
49         # simulation
50         reward = self.default_policy(expand_node.board,
expand_node.color)
51         # Backpropagation
52         self.backup(expand_node, reward)
53
54     @staticmethod
55     def expand(node):
56         """
57         输入一个节点，在该节点上拓展一个新的节点，使用random方法执行Action，返回新增的
节点
58
59         :param node:    待拓展的节点
60         :return:        拓展出来的节点
61         """
62
63         # 在有效且没被访问过的落子中随机选择一个，并从列表中删除此落子
64         action = random.choice(node.unVisitActions)
65         node.unVisitActions.remove(action)
66
67         # 执行action，得到新的board
68         newBoard = deepcopy(node.board)
69         # 如果还有落子机会
70         if action != "noway":
71             #####
72             #此处因为pycharm疯狂报错说_move是私有的函数所以都改为了move#
73             #####
74             newBoard.move(action, node.color)
75         else:
76             pass
77
78         newColor = 'x' if node.color == 'o' else 'o'
79         newNode = Node(newBoard, node, newColor, action)
80         node.children.append(newNode)
81
82         return newNode
83
84     @staticmethod
85     def best_child(node, balance, color):
86         # 对每个子节点调用一次计算bestValue
87         for child in node.children:
88             child.calcBestVal(balance, color)
89
90         # 对子节点按照bestValue降序排序
91         sortedChildren = sorted(node.children, key=lambda x:
x.bestVal[color], reverse=True)
92
93         # 返回bestValue最大的元素
94         return sortedChildren[0]
95
96     def tree_policy(self, node):
97         """
98         根据exploration/exploitation算法返回最好的需要expand的节点
99         注意如果节点是叶子结点（棋局结束）直接返回。
100

```

```

101         :param node:        当前需要开始搜索的节点（例如根节点）
102         :return:            还有未展开的节点，那么返回这个展开的节点；都被展开了，返回value最好的child
103         """
104         retNode = node
105         # 如果棋局还没有结束
106         while not retNode.isOver:
107             if len(retNode.unvisitActions) > 0:
108                 # 还有未展开的节点，那么返回这个展开的节点
109                 return self.expand(retNode)
110             else:
111                 # 都被展开了，返回value最好的child
112                 retNode = self.best_child(retNode, math.sqrt(2),
retNode.color)
113
114         return retNode
115
116     @staticmethod
117     def default_policy(board, color):
118         """
119         蒙特卡罗树搜索的Simulation阶段
120         输入一个需要expand的节点，随机操作后创建新的节点，返回新增节点的reward。
121         注意输入的节点应该不是子节点，而且是有未执行的Action可以expand的。
122
123         基本策略是随机选择Action。
124
125         :param board:        当前棋盘
126         :param color:        当前执子
127         :return:              赢家和赢了多少
128         """
129         newBoard = deepcopy(board)
130         newColor = color
131
132         def gameOver(board1):
133             l1 = list(board1.get_legal_actions('x'))
134             l2 = list(board1.get_legal_actions('o'))
135             return len(l1) == 0 and len(l2) == 0
136
137         while not gameOver(newBoard):
138             actions = list(newBoard.get_legal_actions(newColor))
139             if len(actions) == 0:
140                 action = None
141             else:
142                 action = random.choice(actions)
143
144             if action is None:
145                 pass
146             else:
147                 newBoard.move(action, newColor)
148
149             newColor = 'x' if newColor == 'o' else 'o'
150
151         # 0黑 1白 2平局，diff是二者相差的棋子数目
152         winner, diff = newBoard.get_winner()
153         diff /= 64
154         return winner, diff
155
156     @staticmethod

```

```

157     def backup(node, reward):
158         """
159         回溯，将reward加回去
160
161         :param node:        当前节点
162         :param reward:
163         :return:
164         """
165
166         newNode = node
167         # 节点不为none时（根节点的parent是none）
168         while newNode is not None:
169             # 被访问次数增加
170             newNode.visit_times += 1
171
172             if reward[0] == 0:
173                 newNode.reward['X'] += reward[1]
174                 newNode.reward['O'] -= reward[1]
175             elif reward[0] == 1:
176                 newNode.reward['X'] -= reward[1]
177                 newNode.reward['O'] += reward[1]
178             elif reward[0] == 2:
179                 pass
180
181             newNode = newNode.parent

```

main.py

```

1  from mcts import MonteCarlo
2
3
4  class AIPlayer:
5      """
6      AI 玩家
7      """
8
9      def __init__(self, color):
10         """
11         玩家初始化
12
13         :param color: 下棋方, 'X' - 黑棋, 'O' - 白棋
14         """
15
16         self.color = color
17
18     def get_move(self, board):
19         """
20         根据当前棋盘状态获取最佳落子位置
21
22         :param board: 棋盘
23         :return: action 最佳落子位置, e.g. 'A1'
24         """
25         if self.color == 'X':
26             player_name = '黑棋'
27         else:
28             player_name = '白棋'

```

```

29         print("请等一会, 对方 {}-{} 正在思考中...".format(player_name,
self.color))
30
31         # -----请实现你的算法代码-----
32         -----
33         # 计算在当前棋盘我方的最好策略。
34         mcts = MonteCarlo()
35         action = mcts.search(board, self.color)
36         # -----
37         -----
38         return action

```

四、实验结果

与RandomPlayer对打

试运行（给AI设限5秒内给出结果）

```

1  from game import Game
2  from player import HumanPlayer, RandomPlayer
3  from AIPlayer import AIPlayer
4
5  # 黑棋初始化
6  black_player1 = AIPlayer("X")
7  black_player2 = HumanPlayer("X")
8  black_player3 = RandomPlayer("X")
9
10 # AI玩家白棋初始化
11 white_player = AIPlayer("O")
12
13 # 游戏初始化, 第一个玩家是黑棋, 第二个玩家是白棋
14 game = Game(black_player3, white_player)
15
16 # 开始下棋
17 game.run()

```

结果:

```

1  A B C D E F G H
2  1 . . . . . . .
3  2 . . . . . . .
4  3 . . . . . . .
5  4 . . . O X . . .
6  5 . . . X O . . .
7  6 . . . . . . .
8  7 . . . . . . .
9  8 . . . . . . .
10 统计棋局: 棋子总数 / 每一步耗时 / 总时间
11 黑   棋: 2 / 0 / 0
12 白   棋: 2 / 0 / 0
13
14 请等一会, 对方 黑棋-X 正在思考中...
15  A B C D E F G H
16  1 . . . . . . .

```

```
17 2 . . . . .
18 3 . . . X . . .
19 4 . . . X X . .
20 5 . . . X O . .
21 6 . . . . .
22 7 . . . . .
23 8 . . . . .
24 统计棋局： 棋子总数 / 每一步耗时 / 总时间
25 黑 棋： 4 / 0 / 0
26 白 棋： 1 / 0 / 0
27
28 请等一会，对方 白棋-O 正在思考中...
29 A B C D E F G H
30 1 . . . . .
31 2 . . . . .
32 3 . . . X . . .
33 4 . . . X X . .
34 5 . . O O O . .
35 6 . . . . .
36 7 . . . . .
37 8 . . . . .
38 统计棋局： 棋子总数 / 每一步耗时 / 总时间
39 黑 棋： 3 / 0 / 0
40 白 棋： 3 / 5 / 5
41
42 请等一会，对方 黑棋-X 正在思考中...
43 A B C D E F G H
44 1 . . . . .
45 2 . . . . .
46 3 . . . X . . .
47 4 . . . X X . .
48 5 . . O X O . .
49 6 . . X . . . .
50 7 . . . . .
51 8 . . . . .
52 统计棋局： 棋子总数 / 每一步耗时 / 总时间
53 黑 棋： 5 / 0 / 0
54 白 棋： 2 / 5 / 5
55
56 .....
57 .....
58 .....
59
60 请等一会，对方 白棋-O 正在思考中...
61 A B C D E F G H
62 1 X O O O O O O
63 2 X O O O O O O
64 3 X O O O O O O
65 4 X X O O O O O
66 5 X X O X X X X
67 6 X O X X O X X
68 7 O X X X X X .
69 8 O X X O O X X
70 统计棋局： 棋子总数 / 每一步耗时 / 总时间
71 黑 棋： 28 / 0 / 0
72 白 棋： 34 / 5 / 145
73
74 请等一会，对方 白棋-O 正在思考中...
```



```

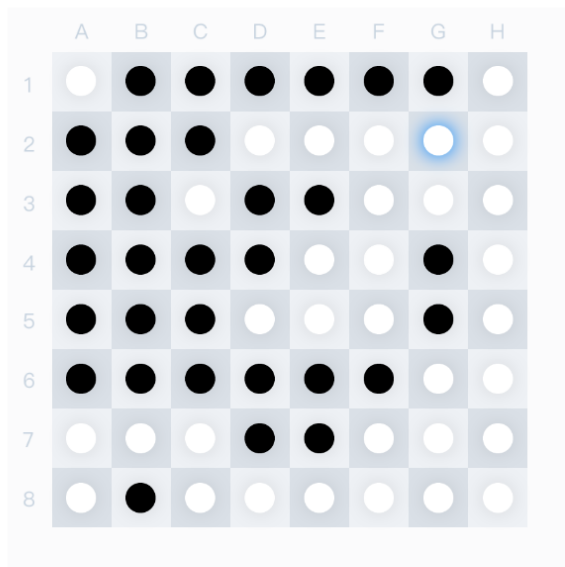
75      A B C D E F G H
76  1 X 0 0 0 0 0 0 0
77  2 X 0 0 0 0 0 0 0
78  3 X 0 0 0 0 0 0 0
79  4 X X 0 0 0 0 0 0
80  5 X X 0 X 0 X 0 X
81  6 X 0 X X 0 0 0 X
82  7 0 0 0 0 0 0 0 .
83  8 0 X X 0 0 X X X
84  统计棋局： 棋子总数 / 每一步耗时 / 总时间
85  黑    棋： 19 / 0 / 0
86  白    棋： 44 / 5 / 150
87
88  请等一会，对方 黑棋-X 正在思考中...
89      A B C D E F G H
90  1 X 0 0 0 0 0 0 0
91  2 X 0 0 0 0 0 0 0
92  3 X 0 0 0 0 0 0 0
93  4 X X 0 0 0 0 0 0
94  5 X X 0 X 0 X 0 X
95  6 X 0 X X 0 0 X X
96  7 0 0 0 0 0 0 0 X
97  8 0 X X 0 0 X X X
98  统计棋局： 棋子总数 / 每一步耗时 / 总时间
99  黑    棋： 21 / 0 / 0
100 白    棋： 43 / 5 / 150
101
102
103  =====游戏结束!=====
104
105      A B C D E F G H
106  1 X 0 0 0 0 0 0 0
107  2 X 0 0 0 0 0 0 0
108  3 X 0 0 0 0 0 0 0
109  4 X X 0 0 0 0 0 0
110  5 X X 0 X 0 X 0 X
111  6 X 0 X X 0 0 X X
112  7 0 0 0 0 0 0 0 X
113  8 0 X X 0 0 X X X
114  统计棋局： 棋子总数 / 每一步耗时 / 总时间
115  黑    棋： 21 / 0 / 0
116  白    棋： 43 / 5 / 150
117
118  白棋获胜！
119
120  Process finished with exit code 0

```

与mo平台的高级对打

(给AI设限59秒)

结果：



棋局胜负: 白棋赢

先后手: 黑棋先手

棋局难度: 高级

当前棋子: 白棋

当前坐标: G2



64 / 64



确定

五、总结

是否达到目标预期: 是

可能改进的方向: 提高算法性能, 不知为何

实验过程中遇到的困难: 对于pycharm的不熟悉, 对于蒙特卡洛树算法的理解不够深入

收获: 由于pycharm无微不至甚至奇奇怪怪的报错了解了更多关于python语言的规范知识。