

# 程序报告

学号：

姓名：

## 一、问题重述

(简单描述对问题的理解，从问题中抓住主干，必填)

本问题主要是实现蒙特卡洛树搜索算法来下黑白棋。和黑白棋相关的函数已经写好，主要需要实现的就是蒙特卡洛树搜索算法，给出当前最优的下法。

## 二、设计思想

(所采用的方法，有无对方法加以改进，该方法有哪些优化方向(参数调整，框架调整，或者指出方法的局限性和常见问题)，伪代码，理论结果验证等... 思考题，非必填)

设计思想：主要是实现蒙特卡洛树搜索。蒙特卡洛树搜索主要有四个步骤，分别是选择、拓展、模拟和反向传播。

优化方向：蒙特卡洛树搜索能修改的两个参数分别是时间和  $c$ ，可以进行细节上的调整。

## 三、代码内容

(能体现解题思路的主要代码，有多个文件或模块可用多个"====="隔开，必填)

笔者创建了两个类，分别是蒙特卡洛树节点 MCTreeNode 和蒙特卡洛树搜索 UCTSearch。

首先介绍 MCTreeNode 下的成员函数。

```
def __init__(self, parent, action, color):
    self.action = action
    self.parent = parent
    self.children = []
    self.winNum = 0
    self.visitNum = 0
    self.color = color
```

首先是节点的创建，包括下棋位置 action、父节点(上一步节点) parent、子节点(下一步节点) children、当前节点分数 winNum、当前节点的访问次数 visitNum 和当前节点的阵营 color。

```
def UCB1(self, c):
    UCBlst = np.empty(len(self.children))
    for index, child in enumerate(self.children):
        UCB = child.winNum / child.visitNum + c * math.sqrt(2 * math.log(self.visitNum) / child.visitNum)
        UCBlst[index] = UCB
    return self.children[np.argmax(UCBlst)]
```

上面的 UCB1 函数计算所有子节点的 UCB 值，并返回 UCB 值最大的子节点。笔者计算之后先放在一个 ndarray 中(即 UCBlst)，然后调用 argmax 函数返回 UCB 最大的子节点。

```

def Expand(self, actionList):
    exploredAction = [child.action for child in self.children]
    unexploredAction = []
    for action in actionList:
        if action not in exploredAction:
            unexploredAction.append(action)
    if len(unexploredAction) != 0:
        randomAction = random.choice(unexploredAction)
        if self.color == 'X':
            color = 'O'
        else:
            color = 'X'
        newNode = MCTreeNode(self, randomAction, color)
        newNode.visitNum = newNode.winNum = 0
        self.children.append(newNode)
        return newNode
    else:
        return None

```

Expand 函数实现了蒙特卡洛树搜索中的拓展。actionList 是所有当前节点能下的位置，首先将当前节点所有子节点的 action 放在 exploredAction 中，然后将未探索节点的 action 放在 unexploredAction 中，如果有节点未被探索，就随机选一个未被探索的节点进行创建；若都被探索过就直接返回 None。

```

def BackPropagate(self, value):
    startNode = self
    while startNode:
        startNode.visitNum += 1
        if startNode.color == 'X':
            startNode.winNum += value
        elif startNode.color == 'O':
            startNode.winNum -= value
        startNode = startNode.parent

```

BackPropagate 函数实现了反向传播，得到最终分数 value 之后就反向传播，注意黑棋是加上 value 而白棋是减去 value，然后 visitNum 加一，向上回溯 parent。

---

下面主要介绍 UCTSearch 类中的函数。

```

def __init__(self, board, color, c):
    self.board = board
    self.color = color
    if color == 'X':
        self.root = MCTreeNode(None, None, 'O') # dummy head
    else:
        self.root = MCTreeNode(None, None, 'X')
    self.c = c

```

首先是初始化，需要当前棋局状态 `board`，当前的阵营 `color` 和超参数 `c`。另外创建了蒙特卡洛树的根，这是一个 `dummy head`，主要是方便之后的选择最优下法。

```
def SelectPolicy(self, currentNode, board):
    selectNode = currentNode
    if selectNode.color == 'X':
        color = 'O'
    else:
        color = 'X'
    actionList = list(self.board.get_legal_actions(color))
    while actionList != None:
        expandNode = selectNode.Expand(actionList)
        if expandNode != None:
            board._move(expandNode.action, expandNode.color)
            return expandNode
        elif len(selectNode.children) != 0:
            selectNode = selectNode.UCB1(self.c)
            board._move(selectNode.action, selectNode.color)
            actionList = list(self.board.get_legal_actions(selectNode.color))
        else:
            return selectNode
```

上面 `SelectPolicy` 函数实现选择，首先通过调用 `get_legal_actions` 函数来得到当前能走的位置放在 `actionList` 中，然后进行节点的拓展。如果拓展得到了未探索的节点 `expandNode`，那么就移动棋子并返回当前拓展节点；如果得到 `None` 说明所有节点都被探索过，就选择 UCB 值最高的节点、移动棋子并继续向下探索。

```
def SimulatePolicy(self, board, color):
    if color == 'X':
        color = 'O'
    else:
        color = 'X'
    actionList = list(board.get_legal_actions(color))
    if len(actionList) != 0:
        selectAction = random.choice(actionList)
        board._move(selectAction, color)
        return self.SimulatePolicy(board, color)
    else:
        return board.get_winner()
```

上面 `SimulatePolicy` 函数实现模拟，当得到一个拓展节点，就进行随机下子，直到棋局结束，然后返回 `get_winner()` 函数。

```

def Search(self):
    start_time = datetime.datetime.now()
    end_time = datetime.datetime.now()
    value = 0
    while (end_time - start_time).seconds < 1:
        board = copy.deepcopy(self.board)
        currentNode = self.SelectPolicy(self.root, board)
        winner, diff = self.SimulatePolicy(board, currentNode.color)
        if (self.color == 'X' and winner == 0):
            value = diff
        elif (self.color == 'O' and winner == 1):
            value = -diff
        else:
            value = 0
        currentNode.BackPropagate(value)
        end_time = datetime.datetime.now()

    if len(self.root.children) != 0:
        return self.root.UCB1(self.c).action
    else:
        return None

```

Search 函数是实现蒙特卡洛树搜索的最主要的函数。在时间没结束之前都不断进行蒙特卡洛搜索：首先复制当前状态为 board，然后选择节点 currentNode，然后进行模拟，最后得到的胜利的棋子数作为得分做反向传播。当时间截止之后返回根节点下 UCB 最高的子节点的动作。

#### 四、实验结果

(实验结果，必填)

=====

测试详情 隐藏棋盘 ^

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
|   | A | B | C | D | E | F | G | H |
| 1 | ● | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| 2 | ● | ● | ○ | ● | ○ | ● | ● | ○ |
| 3 | ● | ○ | ● | ○ | ● | ● | ● | ○ |
| 4 | ● | ● | ● | ● | ● | ○ | ○ | ○ |
| 5 | ● | ● | ● | ● | ○ | ○ | ● | ○ |
| 6 | ● | ● | ● | ● | ● | ● | ○ | ○ |
| 7 | ● | ● | ● | ● | ● | ● | ● | ○ |
| 8 | ○ | ● | ● | ● | ● | ● | ● | ○ |

棋局胜负: 黑棋赢

先后手: 黑棋先手

棋局难度: 高级

当前棋子: 白棋

当前坐标: H7

←

64 / 64

→

确定

## 五、总结

（自评分析（是否达到目标预期，可能改进的方向，实现过程中遇到的困难，从哪些方面可以提升性能，模型的超参数和框架搜索是否合理等），**思考题，非必填**）

本次实验基本达到了预期目标，在自己电脑中跑 RandomPlayer 胜率都比较高。但在实验中还是遇到了很多困难，主要还是自己蒙特卡洛树搜索的流程认识不够深入，特别是反向传播函数，笔者一直没有搞清楚传播的值的正负，另外用 Python 写树也是第一次，也存在很多问题。未来的提升方向主要是调整参数等。