

程序报告

学号：2213041

姓名：李雅帆

一、问题重述

1. 黑白棋

黑白棋 (Reversi) 是一个经典的策略性游戏, 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)。它使用 8x8 的棋盘, 由两人执黑子和白子轮流下棋, 最后子多方为胜方, 黑方先行, 双方交替下棋。

2. 游戏规则:

棋局开始时黑棋位于 E4 和 D5, 白棋位于 D4 和 E5, 如图所示。



(1) 黑方先行, 双方交替下棋。

(2) 一步合法的棋步包括:

- ① 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;
- ② 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;
- ③ 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。

3. 实验目的:

要求用 python 语言, 使用 『蒙特卡洛树搜索算法』 实现 miniAlphaGo for Reversi, 算法部分需要自己实现, 不要使用现成的包、工具或者接口。

二、设计思想

1.采用的方法

采用了蒙特卡洛树搜索（MCTS）算法来实现 AI 玩家。MCTS 是一种基于树搜索的启发式算法，用于解决决策问题，特别是在没有完全信息的情况下，例如博弈游戏。

下面是该算法的主要步骤：

（1）选择：从根节点开始，利用一定策略（通常是 Upper Confidence Bound，UCB）选择一个子节点，直到找到一个未完全扩展的节点或者达到终止条件。

（2）扩展：对于选中的未完全扩展的节点，根据可能的行动扩展出新的子节点。

（3）模拟：对于新扩展的节点（或达到终止条件的节点），使用随机策略模拟游戏的进行，直到达到终止状态。

（4）反向传播：根据模拟的结果，更新选中节点及其所有祖先节点的统计信息，如访问次数、胜利次数等。

（5）重复以上步骤，直到达到最大迭代次数或其他停止条件。

该算法通过反复模拟游戏来评估每个可能的行动，并根据模拟结果来动态调整搜索策略，从而最终选择出最优的行动。

2.MCTS 算法的伪代码如下：

```
function MCTS(root, max_iterations):
    for i from 1 to max_iterations:
        leaf_node = select_policy(root)
        blackwin, whitewin = stimulate_policy(leaf_node)
        back_propagate(leaf_node, blackw=blackwin, whitew=whitewin)
    return best_action(root)

function select_policy(node):
    while node is not terminal and not fully_expanded(node):
        if not fully_expanded(node):
            return expand(node)
        else:
            node = ucb(node)
    return node

function stimulate_policy(node):
    while not terminal(node):
        action = random.choice(legal_actions(node.state))
        node = transition(node, action)
    return result(node)

function back_propagate(node, blackw, whitew):
    while node is not None:
        node.visit += 1
        node.blackwin += blackw
        node.whitewin += whitew
        node = node.parent
```

3.方法的改进和优化方向:

(1) UCB 参数调整: UCB 算法的参数对搜索结果有重要影响,可以通过调整参数来优化搜索性能,例如尝试不同的探索参数。

(2) 启发式策略:可以引入更复杂的启发式策略来指导搜索,例如基于领域知识的策略,以提高搜索效率和性能。

(3) 并行化:将 MCTS 算法并行化可以加快搜索速度,可以尝试利用多线程或分布式计算来实现并行化。

(4) 剪枝和加速技术:可以引入一些剪枝和加速技术来减少搜索空间,例如 Alpha-Beta 剪枝等。

(5) 深度学习引导:可以结合深度学习方法,利用神经网络来指导搜索,以提高搜索的准确性和效率。

三、代码内容

1.Node 类:定义了一个节点类,用于表示搜索树中的每个节点。节点包含了游戏状态、访问次数、胜利次数等信息,以及指向父节点和子节点的引用。

```
class Node:
def __init__(self, state, color, parent=None, action=None):
self.visit = 0
self.blackwin = 0
self.whitewin = 0
self.reward = 0.0
self.state = state
self.children = [] # 子节点
self.parent = parent # 父节点
self.action = action
self.color = color

def add_child(self, new_state, action, color):
child_node = Node(new_state, parent=self, action=action, color=color)
self.children.append(child_node)
```

2.AIPlayer 类:实现了 AI 玩家类,包含了初始化、反转颜色、判断是否完全扩展、判断是否终止状态、模拟策略、回传信息、UCB 算法、扩展节点、选择策略、MCTS 算法和获取最佳落子方法。

```
class AIPlayer:
"""
AI 玩家
"""
```

```

def __init__(self, color):
    """
    玩家初始化

    :param color: 下棋方, 'X' - 黑棋, 'O' - 白棋
    """
    self.color = color

def reverse_color(self, color):
    if color == 'X':
        return 'O'
    else:
        return 'X'

def if_fully_expanded(self, node):
    cnt_max = len(list(node.state.get_legal_actions(node.color)))
    cnt_now = len(node.children)
    return cnt_max <= cnt_now

def if_terminal(self, state):
    if state is None:
        return True
    action_black = list(state.get_legal_actions('X'))
    action_white = list(state.get_legal_actions('O'))
    return len(action_white) == 0 and len(action_black) == 0

def stimulate_policy(self, node):
    if node is None:
        return 0, 0
    board = copy.deepcopy(node.state)
    color = copy.deepcopy(node.color)
    cnt = 0
    while not self.if_terminal(board):
        actions = list(board.get_legal_actions(color))
        if len(actions) == 0:
            color = self.reverse_color(color)
        else:
            action = random.choice(actions)
            board._move(action, color)
            color = self.reverse_color(color)
        cnt += 1
    if cnt > 20:
        break

```

```

        return board.count('X'), board.count('O')

def back_propagate(self, node, blackw, whitew):
    while node is not None:
        node.visit += 1
        node.blackwin += blackw
        node.whitewin += whitew
        node = node.parent
    return 0

def ucb(self, node, uct_scalar=0.0):
    def exploit(node):
        return node.blackwin / (node.blackwin + node.whitewin) if node.color == 'O' else
node.whitewin / (node.blackwin + node.whitewin)

    def explore(node):
        return math.sqrt(2.0 * math.log(node.parent.visit) / float(node.visit))

    max_score = -float('inf')
    max_nodes = []

    for child in node.children:
        score = exploit(child) + uct_scalar * explore(child)
        if score > max_score:
            max_score = score
            max_nodes = [child]
        elif score == max_score:
            max_nodes.append(child)

    if not max_nodes:
        return None # Handle empty sequence

    return random.choice(max_nodes)

def expand(self, node):
    available_actions = list(node.state.get_legal_actions(node.color))
    actions_already_taken = [child.action for child in node.children]

    if len(available_actions) == 0:
        return node.parent

    action = random.choice(available_actions)
    while action in actions_already_taken:
        action = random.choice(available_actions)

```

```

        new_state = copy.deepcopy(node.state)
        new_state._move(action, node.color)
        new_color = self.reverse_color(node.color)
        node.add_child(new_state, action=action, color=new_color)

    return node.children[-1]

def select_policy(self, node):
    while node is not None and not self.if_terminal(node.state):
        if not self.if_fully_expanded(node):
            return self.expand(node)
        else:
            node = self.ucb(node)
    return node

def MCTS(self, root, max_iterations=100):
    for _ in range(max_iterations):
        leaf_node = self.select_policy(root)
        blackwin, whitewin = self.stimulate_policy(leaf_node)
        self.back_propagate(leaf_node, blackw=blackwin, whitew=whitewin)

    return self.ucb(root).action if self.ucb(root) is not None else None

def get_move(self, board):
    """
    根据当前棋盘状态获取最佳落子位置
    :param board: 棋盘
    :return: action 最佳落子位置, e.g. 'A1'
    """
    if self.color == 'X':
        player_name = '黑棋'
    else:
        player_name = '白棋'
    print("请等一会，对方 {}-{} 正在思考中...".format(player_name, self.color))

    # -----请实现你的算法代码-----
    action = None
    root_board = copy.deepcopy(board)
    root = Node(state=root_board, color=self.color)
    action = self.MCTS(root)
    # -----

    return action

```

3. MCTS 算法主要分为四个步骤:

- (1) 选择: 从根节点开始, 根据一定的策略选择一个节点, 直到找到一个未完全扩展的节点或达到终止条件。
- (2) 扩展: 对于选中的未完全扩展的节点, 根据可能的行动扩展出新的子节点。
- (3) 模拟: 对新扩展的节点或达到终止条件的节点, 使用随机策略模拟游戏进行, 直到达到终止状态。
- (4) 反向传播: 根据模拟的结果, 更新选中节点及其所有祖先节点的统计信息。

```
def MCTS(self, root, max_iterations=100):
    for _ in range(max_iterations):
        leaf_node = self.select_policy(root)
        blackwin, whitewin = self.stimulate_policy(leaf_node)
        self.back_propagate(leaf_node, blackw=blackwin, whitew=whitewin)

    return self.uctb(root).action if self.uctb(root) is not None else None
```

4.get_move 方法: 根据当前棋盘状态, 创建根节点, 然后运行 MCTS 算法来选择最佳落子位置。

```
def get_move(self, board):
    """
    根据当前棋盘状态获取最佳落子位置
    :param board: 棋盘
    :return: action 最佳落子位置, e.g. 'A1'
    """
    if self.color == 'X':
        player_name = '黑棋'
    else:
        player_name = '白棋'
    print("请等一会, 对方 {}-{} 正在思考中...".format(player_name, self.color))
    # -----请实现你的算法代码-----
    action = None
    root_board = copy.deepcopy(board)
    root = Node(state=root_board, color=self.color)
    action = self.MCTS(root)
    # -----
    return action
```

代码的主要思路是利用 MCTS 算法进行搜索, 通过模拟游戏的进行来评估每个可能的行动, 并根据模拟结果动态调整搜索策略, 最终选择出最优的行动。

四、实验结果

接口测试

✓ 接口测试通过。

用例测试

展示棋盘

测试点	状态	时长	结果
对手对弈	✓	46s	黑棋获胜, 领先棋子数: 6

接口测试

✓ 接口测试通过。

用例测试

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: 白棋后手

棋局难度: 高级

当前棋子: 白棋

当前坐标: D7

五、总结

代码达到了实现基于蒙特卡洛树搜索（MCTS）算法的 AI 玩家的目标，并提供了一个有效的方法来选择最佳落子位置，通过了平台的测试。

然而，还有一些改进的方向：

1.改进模拟策略：当前的模拟策略是随机选择动作，可能不够精确，可以尝试改进模拟策略，例如使用启发式算法或深度学习模型来更准确地模拟游戏结果。

2.优化 UCB 算法参数: 当前的 UCB 算法使用固定的参数值 (默认为 0.0), 可以尝试通过实验调整该参数以获得更好的性能。

3.优化节点扩展策略：节点扩展时使用的随机选择动作的方法可能导致性能不稳定。可以尝试使用更智能的方法来选择扩展动作，例如根据启发式函数或预训练模型来选择动作。

4.性能优化: 对于大型棋盘或深度搜索树, 当前实现可能存在性能瓶颈。可以尝试使用更高效的数据结构、算法或并行化技术来提升性能。

5.超参数调优：当前的 MCTS 算法中的迭代次数（max_iterations）是一个超参数，可以通过实验调优来找到最佳值。

6.框架搜索：目前的实现是基于原始的 MCTS 算法，可能存在更先进的变体或改进方法，可以进一步研究和尝试。

在实现过程中，可能会遇到一些困难：

1.性能问题：MCTS 算法在大规模搜索空间中可能会遇到性能问题，尤其是在每次迭代时需要进行大量的模拟和回传操作。

2.参数调优：确定合适的 UCB 参数和迭代次数可能需要进行大量的实验和调优，需要花费一定的时间和计算资源。

3.代码调试：由于 MCTS 算法涉及到大量的递归和迭代操作，调试代码可能会比较复杂，需要仔细检查每个步骤的实现是否正确。