程序报告

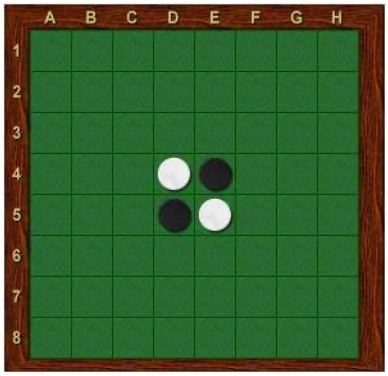
学号：2213924　　姓名：申宗尚

一、问题重述

黑白棋(reversi),也叫苹果棋，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，变为己方棋子，故又称“翻转棋”。棋子双面为红、绿色的成为“苹果棋”。它使用 8*8 的棋盘，由两人执黑子和白子轮流下棋，最后子多方为胜。

游戏规则：

棋局开始时黑棋位于 E4 和 D5，白棋位于 D4 和 E5，如下图所示



游戏规则：

(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。

d.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。

e.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。

(3)棋局持续下去，直到棋盘填满或者双方都无合法棋步可下。

(4)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。

实验要求：

使用 python 语言，用蒙特卡洛树搜索算法实现 mini alphago 的程序。

由于作业平台已经完成对游戏规则(board.py, game.py)的大体实现，故我们需要编写算法，实现 AI 玩家(AI player)类的设计，并在其中实现蒙特卡洛树搜索算法即可。

二、设计思想

蒙特卡洛树搜索（Monte Carlo Tree Search，MCTS）是一种启发式搜索算法，通常用于解决具有极大状态空间的游戏或决策问题。它通过模拟随机样本来构建一棵搜索树，并根据模拟结果来指导搜索。

1.树结构：

搜索过程被组织成一棵树，称为“蒙特卡洛搜索树”（Monte Carlo Search Tree）。

每个节点表示游戏中的一个状态，可能是当前状态或者是在之前的某一步中产生的状态。

树的根节点代表当前游戏状态。

2.四个主要阶段：

选择（Selection）：从根节点开始，根据一定策略选择一个未完全探索的节点，直到达到叶子节点。

扩展（Expansion）：对于选定的叶子节点，根据游戏规则扩展其子节点，即生成可能的后继状态。

模拟（Simulation）：针对新生成的节点或已有节点的扩展部分，通过随机模拟（例如随机游戏走子）快速得出一个结果。

反向传播（Backpropagation）：根据模拟的结果，更新沿着选择路径的所有节点的统计信息，如访问次数和收益，以便下一次选择能够更好地引导搜索。

3.节点数据结构：

每个节点包含两个基本统计信息：

访问次数（Visit Count）：该节点被访问的次数。

价值（Value）：该节点对应的状态的平均收益或估计值。

其他可能的信息包括子节点列表、动作列表等。

4.选择策略：

常用的选择策略是 UCT（Upper Confidence Bound for Trees），结合访问次数和价值的加权值，以平衡探索未知节点和利用已知信息的权衡。

5.停止条件：

可以是达到预定的搜索时间、达到固定的迭代次数或者满足特定的终止条件（例如游戏结束）

三、代码介绍

总览：

在代码中，我们实现了三个类，分别是：

1.Node 类：表示 MCTS 中的节点。它包含了当前游戏状态的信息，如棋盘状态、当前落子颜色等，以及与 MCTS 相关的统计信息，如访问次数、奖励等。每个节点可以有多个子节点，代表着不同的游戏走法。

2.MonteCarlo_Search 类：实现了 MCTS 的搜索算法。它包含了根节点 root，以及 MCTS 中的一些参数和方法。在 search 方法中，通过调用 search_by_mcts 方法执行 MCTS 算法搜索最佳落子位置。search_by_mcts 方法利用了 build_montecarlo_tree 方法构建 MCTS 树，并在超时时间内返回搜索结果。

3.AIPlayer 类：表示一个基于 MCTS 的人工智能玩家。get_move 方法中创建了 MonteCarlo_Search 的实例，并调用其 search 方法来获取最佳落子位置。

而在 MonteCarlo_Search 类中，具体完成了以下对于四个主要阶段的实现：

1.选择(Select)：

```
def select(self):
    current_node = self.root
    while not current_node.isLeaf:
        if random.random() > self.epsilon:
            current_node = current_node.get_best_child()
        else:
            current_node = random.choice(current_node.children)
        self.epsilon *= self.gamma
    return current_node
```

```
def get_best_child(self):
    if len(self.children) == 0:
        self.best_child = None
    else:
        sorted_children = sorted(self.children, key=lambda child: child.value[self.color], reverse=True)
        self.best_child = sorted_children[0]
    return self.best_child
```

在 MonteCarlo_Search 类的 Select 方法中，从根节点开始向下选择，直到选择到叶节点(未扩展)为止，单步选择策略为：使用 epsilon 参数作为探索率（在本代码中设置为 0.3），通过随机数是否大于 epsilon，实现探索，其他情况，使用 Node 类的 get_best_child 方法，选择 value 值最多的子节点，以此类推。

2.扩展(Expand)：

```
def expand(self, node: Node):
    if len(node.actions) == 0:
        board = deepcopy(node.board)
        color = 'X' if node.color == 'O' else 'O'
        child = Node(board=board, color=color, parent=node, pre_action="none", root_color=self.color)
        node.add_child(child)
        return node.best_child
    for action in node.actions:
        board = deepcopy(node.board)
        board._move(action=action, color=node.color)
        color = 'X' if node.color == 'O' else 'O'
        child = Node(board=board, color=color, parent=node, pre_action=action, root_color=self.color)
        node.add_child(child=child)
    return node.best_child
```

在 MonteCarlo_Search 类的 Expand 方法中，对于 Select 到的叶子节点，分析其合法行动，如果无合法行动，则直接生成子节点（变更为对手颜色），若有合法行动，则对每个合法行动均生成一个子节点（变更为对手颜色），实现节点的扩展。

3.模拟(Simulation)：

```
def simulation(self, node: Node):
    board = deepcopy(node.board)
    color = node.color
    while not self.game_over(board=board):
        actions = list(board.get_legal_actions(color=color))
        if len(actions) != 0:
            board._move(random.choice(actions), color)
        color = 'X' if color == 'O' else 'O'
    winner, difference = board.get_winner()
    return winner, difference
```

在 MonteCarlo_Search 类的 Simulation 方法中，对于一个需要模拟的节点，先通过 deepcopy 创造模拟的棋盘，然后通过循环和随机选择可行行动，随机执行到游戏结束，获取赢家和赢子数作为返回值，实现模拟。

4.反向传播(Backpropagation)：

```
def backpropagation(self, node: Node, winner, difference):
    while node is not None:
        node.visit_count += 1
        if winner == 0:
            node.reward['O'] -= difference
            node.reward['X'] += difference
        elif winner == 1:
            node.reward['X'] -= difference
        elif winner == 2:
            pass
        if node is not self.root:
            node.parent.visit_count += 1
            for child in node.parent.children:
                child.get_value()
            node.parent.visit_count -= 1
        node = node.parent
```

```
def get_value(self):
    if self.visit_count == 0:
        return
    for color in ['X', 'O']:
        self.value[color] = self.reward[color] / self.visit_count + \
                            Node.coefficient * math.sqrt(
            math.log(self.parent.visit_count)*2 / self.visit_count)
```

在 MonteCarlo_Search 类的 Backpropagation 方法中，由当前节点向上对于所有节点根据 winner 和 difference，更新 visit 和 reward 值，同时使用 node 类的 get_value 方法，根据 UCB 公式计算节点价值，从而实现对于整颗蒙特卡洛树的反向传播。

```
def build_montecarlo_tree(self):
    while 1==1:
        current_node = self.select()
        if current_node.isOver:
            winner, difference = current_node.board.get_winner()
        else:
            if current_node.visit_count:
                current_node = self.expand(current_node)
            winner, difference = self.simulation(current_node)
        self.backpropagation(node=current_node, winner=winner, difference=difference)
```

在 build_montecarlo_tree 方法中，构建 MCTS 树。它循环执行 MCTS 的四个步骤：选择、扩展、模拟和反向传播。

```
def search(self):
    if len(self.root.actions) == 1:
        return self.root.actions[0]

    return self.search_by_mcts()

def search_by_mcts(self):
    try:
        func_timeout(timeout=3, func=self.build_montecarlo_tree)
    except FunctionTimedOut:
        pass

    return self.root.get_best_reward_child().preAction
```

在 search 方法中，通过 func_timeout，设置蒙特卡洛树生成的时间限制，以及返回最终决定。

```
class AIPlayer:
    def __init__(self, color: str, model_save_path="./results/model.pth"):
        self.color = color.upper()
        self.comments = "请稍后，{}正在思考".format("黑棋(X)" if self.color == 'X' else "白棋(O)")
        self.model_save_path = model_save_path

    def get_move(self, board):
        print(self.comments)
        model = MonteCarlo_Search(board, self.color, model_save_path=self.model_save_path)
        action = model.search()
        return action
```

在 AIPlayer 类中，通过调用蒙特卡洛树搜索方法，给出最终的 action 决定。

从而，代码实现了一个基于 MCTS 的智能玩家，能够在给定棋盘状态下，利用 MCTS 算法搜索出最佳的落子位置

四、代码内容

```
from copy import deepcopy
import csv
import torch
from func_timeout import func_timeout, FunctionTimedOut
import math
import os.path
import random

class Node:
    coefficient = 2


    def __init__(self, board, color, root_color, parent=None, pre_action=None):
        self.board = board
        self.color = color.upper()
        self.root_color = root_color
        self.parent = parent
        self.children = []
        self.best_child = None
        self.get_best_child()
        self.preAction = pre_action
        self.actions = list(self.board.get_legal_actions(color=color))
        self.isOver = self.game_over()
        self.reward = {'X': 0, 'O': 0}
        self.visit_count = 0
        self.value = {'X': 1e5, 'O': 1e5}
        self.isLeaf = True
        self.best_reward_child = None
        self.get_best_reward_child()


    def game_over(self):
        black_list = list(self.board.get_legal_actions('X'))
        white_list = list(self.board.get_legal_actions('O'))
        game_is_over = len(black_list) == 0 and len(white_list) == 0
        return game_is_over

    def get_value(self):
        if self.visit_count == 0:
            return
        for color in ['X', 'O']:
            self.value[color] = self.reward[color] / self.visit_count + \
                                Node.coefficient * math.sqrt(
```

```
            math.log(self.parent.visit_count)*2 / self.visit_count)

    def add_child(self, child):
        self.children.append(child)
        self.get_best_child()
        self.get_best_reward_child()
        self.isLeaf = False

    def get_best_child(self):
        if len(self.children) == 0:
            self.best_child = None
        else:
            sorted_children = sorted(self.children, key=lambda child: child.value[self.color],
reverse=True)
            self.best_child = sorted_children[0]
        return self.best_child

    def get_best_reward_child(self):
        if len(self.children) == 0:
            best_reward_child = None
        else:
            sorted_children = sorted(self.children, key=lambda child: child.reward[
                                                                self.color] /
child.visit_count if child.visit_count > 0 else -1e5,
                                     reverse=True)
            best_reward_child = sorted_children[0]
        self.best_reward_child=best_reward_child
        return self.best_reward_child


class MonteCarlo_Search:
    def __init__(self, board, color, model_save_path):
        self.root = Node(board=deepcopy(board), color=color, root_color=color)
        self.color = color
        self.experience = {"state": [], "reward": [], "color": []}
        self.max_experience = 10000000000
        self.trans = {"X": 1, "O": -1, ".": 0}
        self.learning_rate = 0.3
        self.epsilon = 0.3
        self.gamma = 0.999

        self.model = None
        self.model_save_path = model_save_path
        self.device =torch.device("cpu")
```

```
def get_experience(self):
    queue = []
    for child in self.root.children:
        queue.append(child)
    while len(queue) > 0:
        if len(self.experience) == self.max_experience:
            break
        if not queue[0].isLeaf:
            self.add_experiences(queue[0])
            for child in queue[0].children:
                queue.append(child)
        queue.pop(0)

def add_experiences(self, node: Node):

    if len(self.experience["reward"]) == self.max_experience:
        return

    experience = self.get_state(node)
    self.experience["state"].append(experience)
    reward = node.reward["X" if node.color == "O" else "O"] / node.visit_count
    self.experience["reward"].append(reward)
    self.experience["color"].append(node.color)

def get_state(self, node):
    new_state=node.board._board
    return new_state



def search(self):
    if len(self.root.actions) == 1:
        return self.root.actions[0]

    return self.search_by_mcts()

def search_by_mcts(self):
    try:
        func_timeout(timeout=3, func=self.build_montecarlo_tree)
    except FunctionTimedOut:
        pass

    return self.root.get_best_reward_child().preAction

def build_montecarlo_tree(self):
```

```
        while 1==1:
            current_node = self.select()
            if current_node.isOver:
                winner, difference = current_node.board.get_winner()
            else:
                if current_node.visit_count:
                    current_node = self.expand(current_node)
                winner, difference = self.simulation(current_node)
            self.backpropagation(node=current_node, winner=winner, difference=difference)

    def select(self):
        current_node = self.root
        while not current_node.isLeaf:
            if random.random() > self.epsilon:
                current_node = current_node.get_best_child()
            else:
                current_node = random.choice(current_node.children)
            self.epsilon *= self.gamma
        return current_node

    def simulation(self, node: Node):
        board = deepcopy(node.board)
        color = node.color
        while not self.game_over(board=board):
            actions = list(board.get_legal_actions(color=color))
            if len(actions) != 0:
                board._move(random.choice(actions), color)
            color = 'X' if color == 'O' else 'O'
        winner, difference = board.get_winner()
        return winner, difference

    def game_over(self, board):
        b_list = list(board.get_legal_actions('X'))
        w_list = list(board.get_legal_actions('O'))
        is_over = len(b_list) == 0 and len(w_list) == 0  # 返回值 True/False
        return is_over

    def expand(self, node: Node):
        if len(node.actions) == 0:
            board = deepcopy(node.board)
            color = 'X' if node.color == 'O' else 'O'
            child = Node(board=board, color=color, parent=node, pre_action="none",
root_color=self.color)
            node.add_child(child)
```

```
            return node.best_child
        for action in node.actions:
            board = deepcopy(node.board)
            board._move(action=action, color=node.color)
            color = 'X' if node.color == 'O' else 'O'
            child = Node(board=board, color=color, parent=node, pre_action=action,
root_color=self.color)
            node.add_child(child=child)
        return node.best_child

    def backpropagation(self, node: Node, winner, difference):
        while node is not None:
            node.visit_count += 1
            if winner == 0:
                node.reward['O'] -= difference
                node.reward['X'] += difference
            elif winner == 1:
                node.reward['X'] -= difference
            elif winner == 2:
                pass
            if node is not self.root:
                node.parent.visit_count += 1
                for child in node.parent.children:
                    child.get_value()
                node.parent.visit_count -= 1
            node = node.parent



class AIPlayer:
    def __init__(self, color: str, model_save_path="./results/model.pth"):
        self.color = color.upper()
        self.comments = "请稍后，{}正在思考".format("黑棋(X)" if self.color == 'X' else "白棋
(O)")
        self.model_save_path = model_save_path

    def get_move(self, board):
        print(self.comments)
        model = MonteCarlo_Search(board, self.color, model_save_path=self.model_save_path)
        action = model.search()
        return action
```

五、实验结果

与高级人机对弈，获胜，领先棋子 34 颗

六、总结

本次解答基本达到了目标预期，手写了一颗 MonteCarlo 树，通过搜索达到了给出黑白棋最优落子点的要求，本次实验难点和关键如下：

1、选择、扩展、模拟、反向传播四大关键方法的构建：通过查阅书本、ppt、网上资料，在对于 MCTS 基础知识有充分理解的基础上进行了代码实现

2、对于数据、UCB 的构建和利用： 由于黑白棋游戏特殊之处（给出胜子数），故不同于单纯使用胜局数进行 value 值的计算，可以采用胜子数进行计算，由于胜子数高的行动一般也是优秀的行动，有效的提高了 MCTS 的准确度

3、对于探索过程的设计： 由于 MCTS 的特殊性，若不涉及探索过程，容易陷入死局，故设置了 epsilon 参数=0.3，在该概率下使用随机的方法进行选择，从而避免进入死胡同，提高准确度和置信度。

总的来说，这次实验提高了我对于 MCTS、黑白棋、**python 程序语言编写**的熟练度。