作业 2: 黑白棋游戏

季千焜（221300066、qkjiai@smail.nju.edu.cn）

(南京大学 人工智能学院, 南京 210093)

1 读源代码 MiniMaxDecider.java

MiniMaxDecider 类有以下几个重要的成员变量和函数：

1. 成员变量：

- maximize：表示当前是最大化还是最小化的角色。作为一个开关在 decide 和 miniMaxRecursor 函数中发挥作用
- depth：表示搜索的深度，即搜索树的层数。
- computedStates：用于存储已经计算过的状态和对应的评估值（得分）。
- DEBUG：用于生成搜索空间的图形表示，可以在调试时使用。

2. 构造函数：

- MiniMaxDecider(boolean maximize, int depth)：初始化 MiniMaxDecider 对象，设置最大化或最小化的角色以及搜索的深度。

3. Decide 决策函数：

- Action decide(State state)：根据传入的状态返回下一步的动作。首先设置 value 和 flag 两个变量，maximize 为 true 时 value 值为负无穷，flag 值为 1；反之 value 值为正无穷，flag 值为-1，以此来决定是选择执行 Max 函数还是 Min 函数。

此外还定义了变量 bestActions，用来存储最优动作（可能会有多个最优动作所以采用 List 型）。然后对当前局面所有可能的动作进行遍历，得到新的局面，根据新的局面用 miniMaxRecursor 函数计算对应的得分，并与当前的最优得分比较，如果前者大于后者，即 Max 函数中新局面得分大于最高得分/Min 函数中新局面得分小于最低得分时，对最优得分进行更新，并清空 bestActions。第二次和第一次同理，是为了将所有能得到最优得分的动作都加入 bestActions 中。遍历完所有可能的动作以后，利用 Collections 类的 shuffle 函数随机打乱 bestActions 中动作的顺序，再返回其中的第一个元素，即随机抽取一个 bestActions 中的动作执行。

4. miniMaxRecursor 计算给定局面最优得分函数：

- float miniMaxRecursor(State state, int depth, boolean maximize)：MiniMax 算法的实现。根据当前状态、搜索深度及 Max 和 Min 的交替，递归地搜索博弈树，并返回当前分支的最优得分（如果游戏结束或递归搜索达到限定的深度，则返回当前局面由启发式函数 heuristic 计算得出的值）。

5. finalize 结束函数：

- private float finalize(State state, float value)：将状态和对应的评估值存储到 computedStates 中，并返回评估值。

总结：MiniMaxDecider 类通过递归搜索博弈树，使用 alpha-beta 剪枝来提高搜索效率，并使用 HashMap 来避免重复计算状态。它可以根据给定的最大化或最小化角色和搜索深度，实现 MiniMax 搜索。

2 修改 MiniMaxDecider 类，加入 AlphaBeta 剪枝

在 miniMaxRecursor 方法中，需要添加了两个参数 alpha 和 beta 并分别初始化为正无穷和负无穷，用于

表示当前搜索路径上的 Alpha 和 Beta 边界。在每次递归调用时，根据当前的最大化或最小化状态更新 Alpha 和 Beta 的值，并进行剪枝判断。如果 Alpha 大于等于 Beta，则可以提前终止搜索，因为我们已经找到了一个更好的剪枝点，对于最大化的情况，要更新 Alpha 的值为当前的最大值，对于最小化的情况，要更新 Beta 的值为当前的最小值。在每次递归调用时，需要将当前的 Alpha 和 Beta 值传递给下一层递归调用，以确保正确的剪枝。

修改后的核心代码如下：

```
public float miniMaxRecursor(State state, int depth, boolean maximize, float alpha, float beta) {
    // Has this state already been computed?
    if (computedStates.containsKey(state))
        // Return the stored result
        return computedStates.get(state);
    // Is this state done?
    if (state.getStatus() != Status.Ongoing)
        // Store and return
        return finalize(state, state.heuristic());
    // Have we reached the end of the line?
    if (depth == this.depth)
        // Return the heuristic value
        return state.heuristic();

    // If not, recurse further. Identify the best actions to take.
    float value = maximize ? Float.NEGATIVE_INFINITY : Float.POSITIVE_INFINITY;
    int flag = maximize ? 1 : -1;
    List<Action> test = state.getActions();
    for (Action action : test) {
        // Check it. Is it better? If so, keep it.
        try {
            State childState = action.applyTo(state);
            float newValue = this.miniMaxRecursor(childState, depth + 1, !maximize, alpha, beta);
            // Record the best value
            if (flag * newValue > flag * value)
                value = newValue;
            // Alpha-Beta pruning
            if (maximize) {
                alpha = Math.max(alpha, value);
                if (alpha >= beta)
                    break;
            } else {
                beta = Math.min(beta, value);
                if (beta <= alpha)
                    break;
            }
        } catch (InvalidActionException e) {
```

```
                // Should not go here
                throw new RuntimeException("Invalid action!");
            }
        }
        // Store so we don't have to compute it again.
        return finalize(state, value);
    }
```

3 理解 othello.OthelloState 类中的 heuristic 函数并进行改进

heuristic 函数是 OthelloState 类中的一个评估函数，用于评估当前游戏状态的好坏程度。该函数返回一个浮点数作为评估值。该函数的计算过程如下：

1. 首先，根据当前游戏状态的 Status（游戏状态）来确定一个胜利常数 winconstant。如果当前状态是 PlayerOneWon，将 winconstant 设置为 5000；如果当前状态是 PlayerTwoWon，将 winconstant 设置为 -5000；否则，将 winconstant 设置为 0。
2. 然后，计算以下四个部分的评估值，并将它们相加：
   pieceDifferential()：棋子差异，即玩家和对手的棋子数量之差。
   moveDifferential()：可移动差异，即玩家和对手的可移动位置数量之差。
   cornerDifferential()：角落差异，即玩家和对手占据角落的棋子数量之差。
   stabilityDifferential()：稳定性差异，即玩家和对手的稳定棋子（不能被翻转的棋子）数量之差。
3. 最后，将以上四个部分的评估值乘以相应的权重（1、8、100、1），并与 winconstant 加和得到总评估值再返回。

改进如下：

添加了一个新的因素边缘棋子差异（edgeDifferential()）。表示玩家和对手占据边缘位置的棋子数量之差的加权值。边缘包括如下两种：

1. 顶角：如果一方已经占据了一个顶角上的点，那这个顶角在行和列方向上相邻的顶点如果也是同一方的，那也将是不可翻转的顶点，加权值赋值为 2。
2. 最边缘的行与列：最外行与列上的棋子只在横竖中一个方向的翻转受影响，故加权值赋值为 1。

将改进后五个部分的评估值乘以相应的权重（1、8、100、1、400），重新计算总评估值完成改进。

4 阅读并尽量理解 MTDDecider 类，介绍它与 MiniMaxDecider 类的异同

MTDDecider 类是一个实现了 Decider 接口的类，用于进行游戏决策。它基于 MTD(f)算法来计算最佳行动。

MTDDecider 类中的主要函数是 decide 函数，它接收一个 State 对象作为参数，并返回一个 Action 对象作为决策结果。decide 函数使用 iterative_deepening 函数来进行迭代加深搜索，逐渐增加搜索深度，直到达到最大深度或超过规定的搜索时间。

在 iterative_deepening 函数中使用了 AlphaBetaWithMemory 函数来进行搜索。AlphaBetaWithMemory 函数是一个实现了 Alpha-Beta 剪枝和置换表查找的搜索算法。它通过递归地搜索博弈树的各个节点，并根据当前搜索深度和剪枝条件来确定是否继续搜索该节点的子节点。同时，它还利用置换表来缓存已经计算过的状

态，以避免重复计算。

在 AlphaBetaWithMemory 函数中，还使用了 saveAndReturnState 函数来保存当前搜索状态的计算结果，并将其存储在置换表中。这样，在后续的搜索中，如果遇到相同的状态，就可以直接从置换表中获取之前计算的结果，而不需要重新计算。

此外，在 MTDDecider 类中还定义了一些辅助类和常量，用于存储搜索过程中的统计信息和状态值。例如，SearchNode 类用于存储置换表中的节点信息，SearchStatistics 类用于存储搜索过程中的统计数据，LOSE 和 WIN 常量分别表示输和赢的状态值。

与 MiniMaxDecider 类相比，MTDDecider 类有以下几点异同：

异：

1. MTDDecider 类使用了迭代加深搜索（iterative deepening search）来逐渐增加搜索深度，而 MiniMaxDecider 类则是固定搜索深度。这样做的好处是可以在有限的时间内进行更深的搜索，以获得更好的结果。

2. MTDDecider 类使用了置换表（transposition table）来缓存重复的状态，以避免重复计算。这对于提高搜索效率至关重要。而 MiniMaxDecider 类则没有使用置换表。

3. MTDDecider 类使用了 MTD(f)算法来进行搜索，而 MiniMaxDecider 类使用的是传统的 MiniMax 算法。MTD(f)算法是一种优化了的搜索算法，它通过不断调整上下界来逼近最佳解，从而减少搜索空间。

同：

计算评估值 value 的核心方法比较相似，即使用 AlphaBetaWithMemory 函数和 miniMaxRecursor 函数对 value 的计算方法比较相似，都运用了 maximize 开关、一个存储历史状态的集合、递归搜索、alpha-beta 剪枝和启发式函数等。