作业 2：黑白棋游戏 实验报告

丁云翔（191250026、191250026@smail.nju.edu.cn）

摘 要： 理解并介绍 MiniMax 搜索的实现；修改 MiniMaxDecider 类，加入 AlphaBeta 剪枝，并且比较引入剪枝带来的速度变化；理解 heuristic 函数并尝试改进；阅读并理解 MTDDecider 类并介绍与 MiniMaxDecider 类的异同。

关键词： MiniMax 搜索，AlphaBeta 剪枝，heuristic 函数，MTD 算法。

1 任务 1

MiniMaxDecider.java 中的 MiniMaxDecider 类中有一个 boolean 型成员变量 maximize，此变量作为一个开关在 decide 和 miniMaxRecursor 函数中发挥作用，将 Max 和 Min 函数合并为一个函数。其余三个成员变量中 int 型变量 depth 表示搜索的深度；Map 型变量 computedStates 存有已经计算过得分的局面和相应的得分，避免重复计算；boolean 型变量 DEBUG 未使用。

类中还有 4 个函数，构造函数 MiniMaxDecider 负责初始化对象的成员变量；decide 函数和 miniMaxRecursor 函数是实际进行搜索的部分，接下来将详细介绍；finalize 函数直接返回传入的参数 value。

decide 函数是决策函数，它可以根据传入的状态返回下一步的动作。首先根据类成员变量 maximize 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 函数的功能。还定义了 List 型变量 bestActions，用来存储存储最优动作（可能会有多个最优动作所以采用 List）。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，并将得分乘以 flag 后与当前的最优得分乘以 flag 后比较，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。第二次比较的条件与第一次比较基本一致，除了将大于改为了大于等于，如果条件成立，就把对应的 action 加入 bestActions，此举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。

miniMaxRecursor 函数是计算给定局面的最优得分的函数，它可以根据传入的局面、搜索的深度以及控制实现 Max 函数功能还是 Min 函数功能的变量，返回局面对应的最优得分。每次调用该函数时，传入的参数 maximize 都是调用该函数的函数中 maximize 取反之后的值，以实现随着搜索深度的增加，Max 和 Min 交替进行。如果传入的局面在 computedStates 中，即已经计算过了，则直接返回对应的得分；如果游戏结束了或者递归搜索达到了限定的深度，则返回当前局面由启发式函数 heuristic 计算得出的值；如果是其他情况则操作与 decide 函数相近，只不过不牵涉到 bestActions，只计算最优得分。最后返回传入参数中的局面对应的最优得分。

通过使用以上的变量和方法，MiniMax 搜索得以实现。

2 任务 2

在 MiniMaxDecider 类加入 AlphaBeta 剪枝，相关代码主要有以下几处，首先在 decide 函数中加入对 alpha 和 beta 两个 float 型变量的定义和初始化，其中 alpha 初始化为负无穷，beta 初始化为正无穷；然后在 miniMaxRecursor 函数的定义和调用处都加上 alpha 和 beta；最后再在 miniMaxRecursor 函数中加入剪枝的核心代码，如下图所示：

```
try {
    State childState = action.applyTo(state);
    float newValue = this.miniMaxRecursor(childState,  depth: depth + 1, !maximize, alpha, beta);
    //Record the best value
    if (flag * newValue > flag * value){
        value = newValue;
    }
    //加入alpha-beta剪枝
    if(maximize){
        if(value >= beta)
            return finalize(state, value);
        if(value > alpha)
            alpha = value;
    }
    else{
        if(value <= alpha)
            return finalize(state, value);
        if(value < beta)
            beta = value;
    }

} catch (InvalidActionException e) {
```

因为默认的搜索深度为 2 比较小，未剪枝时的搜索速度就比较快了，所以剪枝后速度对比不明显。所以我将搜索深度增大到 7，通过进行游戏并记录搜索时间进行比较，计时发现未采用剪枝时，游戏中间几步每步的搜索时间约为每步 7 到 12 秒；而采用剪枝时，游戏中间几步的搜索时间约为每步 1 到 4 秒，相比与未采用剪枝时所用的时间，速度的提升还是很明显的。

3 任务 3

othello.OthelloState 类中的 heuristic 函数计算当前局面的分值，对 playerOne（以下简称 p1）有利的分值为正数，对 playerTwo（以下简称 p2）有利的分值为负数。首先计算了一个额外的分值 winconstant，如果 p1 获胜，winconstant 为 5000；如果 p2 获胜，winconstant 为-5000；如果还没有玩家获胜，则 winconstant 为 0。再通过给出的公式 this.pieceDifferential() + 8 * this.moveDifferential() + 300 * this.cornerDifferential() + 1 * this.stabilityDifferential() + winconstant，其中前四个函数返回值分别是 p1 现有的棋子数减 p2 现有的棋子数，p1 当前可以落子的位置数减 p2 当前可以落子的位置数，p1 占有的位于棋盘四角的棋子数减去 p2 占有的位于棋盘四角的棋子数，p1 可翻转的棋子数减去 p2 可翻转的棋子数（包括横、竖和两个对角线方向的和），在计算时分别给这四个返回值乘以给定的系数 1、8、100、1，再计算这四项与 winconstant 的和，得到最后的总分并返回。

我对 heuristic 函数做了如下改进：添加了以下两个考虑的因素，一是在考虑顶角的基础上进行拓展，如果某一方已经占据了一个顶角，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，以此类推，沿边界方向延伸的同色棋子均属于此类拓展点，这类拓展点的棋子数量也可以纳入启发式函数考虑的范围，鉴于其重要性略低于顶点，我给予这类拓展点 225 的系数；二是最边界的行和列上的棋子数量（除顶点和上述拓展点），因为行/列边界上的棋子在列/行方向和对角线方向上是无法翻转的了，所以他的稳定性只受一个方向上的影响，所以可以纳入启发式函数的考虑范围，相比于顶点处的棋子在所有方向上都具有稳定性，有 300 的系数，我给予在行/列和对角线稳定的边界点 150 的系数。代码上的改进主要体

现在添加计算这些因素的函数，以及在最后计算返回值的求和式中加入上述函数及系数。

4 任务 4

阅读关于 MTDDecider 类中所使用的 MTD 算法的介绍：“MTD-f 算法只使用零窗口进行搜索，来搜索是否存在比下界值要大的值，如果是将新返回的值设为新的下界，否则设为新的上界，通过一系列的零窗口搜索，使原先初始的最优值边界从（-∞,+∞）逐渐收敛为一个确定的值，而这个值就是当前局面的最优值。同时 MTD-f 算法通过使用置换表技术，把已经搜索过的节点保存在内存中，来减少了对节点重复搜索的开销。”再结合 MTDDecider 类中代码的实现，我对该算法有了大致的理解。该算法中有以下五个主要的函数。decide 函数的功能是根据传入的局面调用 iterative_deepening 函数返回一个最优的动作。iterative_deepening 函数是一个有搜索时间限制的迭代深化搜索函数，以传入的局面为根节点对当前所有可能的动作进行遍历并根据 USE_MTDF 的值选择 MTDF 函数或 AlphaBetaWithMemory 函数来计算新局面的分数，即使达到了搜索的时间限制，已经搜索完成的迭代深度的结果也可以使用，搜索完成后得到一个最佳动作的集合，并随机返回其中的一个动作。MTDF 函数进行迭代搜索，在迭代中调用了 AlphaBetaWithMemory 函数，并返回当前深度下 minimax 的最佳估计值。AlphaBetaWithMemory 函数可以近似看作一个加入了置换表和运用了 alpha-beta 剪枝的 miniMaxRecursor 函数，置换表的作用是当当前局面在置换表 transpositionTable 中时，即以前曾经搜索过，则直接取出对应的值并返回，避免了对重复局面进行搜索的开销。并且，该函数对不同的 depth 做了不同的处理，对于小于等于 4 的 depth，直接按 depth 进行搜索；对于大于 4 的 depth，如果搜索时间不够则抛出异常；如果搜索时间够则分为深度分别为 depth – 2 和 depth 的两轮搜索。在遍历动作的循环中对得分的计算类似于运用了 alpha-beta 剪枝的 miniMaxRecursor 函数。最后的返回值是调用 saveAndReturnState 函数得到的。saveAndReturnState 函数主要功能是根据条件修改 EntryType 的值，并将当前状态对应的节点加入 transpositionTable，并返回当前节点对应的 value。

与 MiniMaxDecider 类的异同：相同的地方在于计算 value 的核心方法比较相似，即 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；不同的地方在于 MTDDecider 类在 MiniMaxDecider 类的基础上做了很多改进，使用了置换表来存储历史状态，存储的信息更多，并且采用了零窗口进行搜索，使用有搜索时间限制的迭代深化搜索，可以返回在有限搜索时间内最准确的搜索结果，适用性更强。