作业 2: 黑白棋游戏

孙嘉欣（201240005、201240005@smail.nju.edu.cn）

(南京大学 匡亚明学院大理科班)

摘 要: 介绍 MiniMax 搜索的实现;应用 AlphaBeta 剪枝并比较速度;理解 heuristic 函数并改进;阅读并理解 MTDDecider 类并与 MiniMaxDecider 类比较异同.

关键词: MiniMax 搜索;Alpha-Beta 剪枝;heuristic 函数;MTD(f)算法

1 任务 1：MiniMax 搜索的实现

程序中的 MiniMax 搜索不同之处在于使用一个开关将两者合并为一个函数。

其第一步，构造运行 MiniMax 算法的决策者 MiniMaxDecider：用一个布尔变量 maximize 记录当前一步是取最大还是最小，值 depth 设置搜索深度和一个 Hashmap 记录已经计算过的状态（避免重复计算）。

再调用 decide 函数进行决策，展开当前局面下所有可能下一步局面（搜索树第一层），需根据得分 value 从中选择一步（最大化选最高分，最小化选最低分），故设置一个值 value 来记录得分，列表 List 记录下得分最佳的所有局面。若当前要最大化得分则初设 value 为 Float 中的最小值 NEGATIVE_INFINITY，若当前要最小化则设 value 为 Float 中的最大值 POSITIVE_INFINITY，从而好与每个局面得分比较；因为可能有同分的局面，所以先将所有最大/小得分局面保存到 List。遍历当前局面中的所有可能 Action,每步得到一个新的状态 State，当前树深度为 1，送到 miniMaxRecursor 函数继续递归向下搜索，并告诉函数下一步是最大最小化与这一步相反。

miniMaxRecursor 函数中若发现该状态搜索过了，则直接从 Hashmap 中找到对应的值返回；若为终局，则通过启发式函数得到局面得分，将局面与得分保存到 Hashmap,并返回得分；若到达设定深度被截断，则通过启发式函数返回得分，此时并未走到终局，局面还有后续，所以不记录到 Hashmap，这样才能使其他路径中更快走到该局面时继续向后搜索，而不是直接查得分，戛然而止。若非以上三种情况，则暂时无法得到得分，需继续向下展开，同样遍历所有子局面，将深度加 1，最大最小化调换，递归调用 MiniMaxRecursor 得到得分。记录最佳得分时，此处体现了使用开关将最大最小化合并的巧妙，再比较前在得分前乘上一个系数（1 或-1），使最大化时得分不变，最小化时得分取负，这样无论是取最大值还是最小值都可以通过一个条件 flag * newValue > flag * value 来比较，乘上系数后更大/小的得分都会比当前记录值更大，从而统一了更新得分的代码。最终，将向下递归得到的更大/小得分与局面保存到 Hashmap，并返回得分。另外，若遍历所有子局面时发现非法走子则捕获、并抛出异常。

这样，通过交替开关最大最小化的开关，miniMaxRecursor 函数可从深度为 1 的可能局面递归向下搜索，得到第一层所有局面的得分。decide 函数遍历比较第一层所有局面的得分，同样通过乘上对应系数的方法，巧妙统一比较得分的代码，此时若得分更大/小，需要更新，则更新 value，并清除 List 里保存的旧 Action；接着得分若更大/小或相等,则把 Action 保存到 List,这时不仅会保存更新后的 Action,遇到得分相同的 Action，不需更新 value，会继续添加到列表中。若遍历所有子局面时发现非法走子也会捕获、并抛出异常。

最终将得到的 Action 列表随机打乱，取第一个，保证了在最佳行动多于一个时，可随机选择一个决策。

2 任务 2：引入 Alpha-Beta 剪枝

2.1 Alpha-Beta剪枝

在第一步，构造运行 MiniMax 算法的决策者 MiniMaxDecider 时，如图 1，增加了一个开关，用于控制是

否引入 Alpha-Beta 剪枝，方便后续两者的速度对比。

```
// Whether to use alpha-beta cut?
private boolean alpha_beta;

/**
 * Initialize this MiniMaxDecider.
 * @param maximize Are we maximizing or minimizing on this turn? True if the former.
 * @param depth    The depth to which we should analyze the search space.
 * @param alpha_beta Whether to use alpha-beta cut on this turn?
 */
public MiniMaxDecider(boolean maximize, int depth, boolean alpha_beta) {
    this.maximize = maximize;
    this.depth = depth;
    computedStates = new HashMap<State, Float>();
    //Use alpha-beta cut?
    this.alpha_beta=alpha_beta;
}
```

Fig.1　The on-off switch of Alpha-Beta cut
图 1　Alpha-Beta 剪枝开关

对照教材上的伪代码，在实际执行 MiniMax 算法的函数 miniMaxRecursor 中，增添两个系数 alpha 和 beta，其他递归搜索部分不变，唯一改动在向下递归搜索获得更大/小得分并更新得分后，加入 Alpha-Beta 剪枝.

```
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: depth + 1, !maximize, alpha, beta);
        //Record the best value
        if (flag * newValue > flag * value) {
            value = newValue;

            //Alpha-beta cut
            if(alpha_beta) {
                if (maximize) {
                    if (value >= beta) return finalize(state, value);
                    alpha = Math.max(value, alpha);
                }
                else {
                    if (value <= alpha) return finalize(state, value);
                    beta = Math.min(value, beta);
                }
            }
        }
```

Fig.2　Alpha-Beta cut
图 2　Alpha-Beta 剪枝

如图,若 Alpha-Beta 剪枝的开关打开，则先判断此时是取最大还是最小，取最大时若当前得分大于等于 Beta（即到目前为止路径上发现的 MIN 的最佳 (即极小值）选择）则直接剪枝，将局面与得分保存到 Hashmap 后返回得分，否则尝试更新 Alpha（即到目前为止路径上发现的 MAX 的最佳 (即极大值）选择）；取最小时反之若当前得分小于等于 Alpha 则剪枝，将局面与得分保存到 Hashmap 后返回得分，否则尝试更新 Beta。Decide 函数中展开树的第一层，递归调用 miniMaxRecursor 函数时，传入最初的 Alpha 和 Beta，即 Float 最小值 NEGATIVE_INFINITY 和 Float 最大值 POSITIVE_INFINITY，以此向下一路更新。

2.2 速度变化对比

```
long startTimeMillis = System.currentTimeMillis();//Timer

OthelloAction action = (OthelloAction) computerPlayer.decide(board);
try {
    board = action.applyTo(board);
    //System.out.println(board);
} catch (InvalidActionException e) {
    throw new RuntimeException("Invalid action!");
}
repaint();
actions = board.getActions();
//耗时
long costTime=System.currentTimeMillis()-startTimeMillis;
totalTime+=costTime;
maxTime=Math.max(maxTime,costTime);
moveCount+=1;
System.out.println(costTime+"ms av."+(totalTime/moveCount)+"ms max."+maxTime+"ms");

System.out.println("Finished with computer move");
```

Fig.3 Timekeeping
图 3 用时统计

如图 2，在电脑落子的代码部分加入统计耗时的代码，逐渐加深深度，每种情况人机随手下了 3 盘，比较电脑落子用时：

搜索深度为 2 时，未加 alpha-beta 剪枝平均每步耗时 4ms，最大耗时 16ms(±1ms)；加 alpha-beta 剪枝平均每步耗时 3ms，最大耗时 16ms(±2ms)，速度提高不明显。

搜索深度为 3 时，未加 alpha-beta 剪枝平均每步耗时 10ms(±3ms)，最大耗时 50ms(±8)ms；加 alpha-beta 剪枝平均每步耗时 7ms(±1ms)，最大耗时 40s(±9s)。

搜索深度为 4 时，未加 alpha-beta 剪枝平均每步耗时 56ms(±7ms)，最大耗时 182ms(±48)ms；加 alpha-beta 剪枝平均每步耗时 22ms(±11ms)，最大耗时 111s(±11s)，速度逐渐拉开差距。

搜索深度为 6 时，未加 alpha-beta 剪枝平均每步耗时 2s 左右，最大耗时 5s 左右，甚至达到 10s，卡顿明显；加 alpha-beta 剪枝平均每步耗时 150ms 左右，最大耗时 480ms 左右，对弈基本流畅，两者速度差距明显，达到 10 倍以上。

搜索深度为 7 时，未加 alpha-beta 剪枝从第三步开始耗时升到 4s 以上，且一路攀升，平均每步耗时半分钟左右，最大耗时接近 2 分钟，中间基本处于长时间转圈不动的状态，直到最后几步；加 alpha-beta 剪枝平均每步耗时 1.2s 左右，最大耗时 5.6s 左右，还能保证基本对弈，可见加入了 alpha-beta 剪枝后对大范围多分枝的搜索速度能有大幅度提升。

3 任务 3：heuristic 函数

3.1 原有heuristic函数的理解

原 heuristic 函数的得分包括 5 个部分：胜负分 winconstant、棋子数之差 pieceDifferential、行动力之差 moveDifferential（合法的可能棋步数量差）、所占角之差 cornerDifferential、稳定值之差 stabilityDifferential（水平、竖直与两个对角线四个方向上能翻转的棋子数差值之和），每个部分赋予一定的分值权重，5 个部分的评分相加得到 heuristic 函数的最终得分。

胜负最重要，因此胜方加 5000 分，使一旦搜索到胜负，就对局面的评判起决定性影响。棋子数多少决定最终胜负，但采取“多子策略”，每步棋都试图翻转最大数量的棋子是贪心的错误策略，因此还要考虑其他

的策略。考虑其他当棋手拥有大量的可能棋步时，他就拥有好的行动力，没有行动力就只能让步，行动力越小选择余地越小，对局面的控制力越小，因此考虑行动力之差，并赋予 8 分的权重。角一旦被占就无法被翻转，能以此为中心控制相邻的两条边和一条斜线，在角周围的区域占极大优势，是非常关键的位置，因此给角数量差赋予 300 分的权重。稳定性是黑白棋翻转中的重要指标，heuristic 函数中从四个方向分别统计从该方向可翻转的棋子数，再累加，以此作为评价稳定性的指标。

实际代码实现中，黑白棋棋局大小为 8*8，框架中将棋盘分为水平、竖直与两个对角线四个方向分别保存在 4 个棋盘中方便查询。每个单方向棋盘中保存该方向的每条线上的棋子，每条线通过一个 16 位的 short 型数值表示出该条线上的棋子摆放情况，每 2 位代表一个位置，每种位置有 00-空,01-墙,10-P1,11-P2 四种情况，即可表示出每条线上 8 个位置的表示情况。因为棋盘中斜线上有的位置实际小于等于 8（比如顶角的斜线上只有 1 个位置），因此引入 01-墙的情况，从而使每条线都可以被统一表示为有 8 个位置的 16 位数。游戏一开始就递归枚举出每条线上所有可能情况，共 14748 种，将每种情况对应的双方可翻转子数差、可落子位置差都计算出来，做成查询表。heuristic 函数直接拿着当前局面对应查表，累和后得到对应的棋子数之差，行动力之差和稳定值之差。

3.2 启发式函数的改进

从四个顶角开始向两边延申的棋子一样无法被翻转，因此统计占角数量时，顺带统计相邻的两边延申出那些同色棋子，给这些棋子赋予 30 分的权重，为了方便放在一个函数，不再只返回个数差，而是将角和相邻棋子的数量差加权统计后直接作为 cornerValue 返回。

```
return this.pieceDifferential() +
    8 * this.moveDifferential() +
     this.cornerValueDifferential() +
    1 * this.stabilityDifferential() +
    winconstant;
```

Fig.4 modified heuristic method
图 4 修改的 heuristic 函数

原函数中对稳定性的评估是通过四个方向分别统计从该方向可翻转的棋子数再累加，但一个棋子可能从多个方向被翻转，因而被重复统计，以此为指标并不合理。但是已原本的框架通过查表速度很快，破坏已有的查表搜索速度可能会下降。

4 任务 4：MTDDecider 类介绍

4.1 MTDDecider类使用的算法

MTDDecider 类使用了 MTD（f）算法（Memory-enhanced Test Driver）+带有 Alpha-Beta 剪枝和置换表的负极大值算法(negaMax)。

negaMax 算法是 MiniMax 算法的变体，黑白棋作为一种零和游戏，基于原理：max(a, b) = -min(-a, -b)，在几个节点中选择得分最小的节点相当于将这些节点的得分乘以-1 后取得分最大的节点。如同在 MiniMax 算法中一样，Alpha-Beta 剪枝也能减少 negaMax 算法所搜索的节点数。alpha 和 beta 分别代表搜索树的某个层级中的节点估值下界和上界，negaMax 算法中某个节点的子节点的 alpha 和 beta 取值为需要将该节点的取值取负，并调换（即-beta 和-alpha）。原本基础的 Alpha-Beta 剪枝中初始设置根节点的 alpha 和 beta 为理论最小值和理论最大值，MTDDecider 类通过 MTD（f）算法设置了更精确的节点估值上下界来获取更多的剪枝，优化搜索效率。

MTD(f)算法使用零大小的搜索窗口对节点估值进行试探：选取区间上的一点进行零宽窗口搜索试探，搜索只有两种结果——估值在窗口之上或在窗口之下，若存在比下界值要大的值，将新返回的值设为新的下界，否则设为新的上界，从而缩小估值的取值区间。在确定新的估值区间后，再次进行试探，因为多次搜索试探过程中，局面估值往往会出现在前一次搜索的返回值附近，因此选取前一次搜索返回值做为新试探值。通过一系列的零窗口搜索反复试探，就可以不断地缩小估值区间，使边界从初始设置的（-∞,+∞）逐渐收敛为一

个确定的值，最终找到局面估值。这个反复试探使窗口收敛的过程，采用迭代加深的深度优先搜索多次调用 MTD（f）算法来完成。由于需要对同一局面多次进行搜索，MTD（f）算法中又引入了置换表，存储并检索以前在内存中搜索的树部分，以减少重复搜索之前搜过的节点的开销，AlphaBetaWithMemory 就是带有储存了之前搜索结果的置换表的 Alpha-Beta 搜索变体。

```
function MTDF(root, f, d) is
    g  := f
    upperBound  := +∞
    lowerBound  := −∞

    while lowerBound < upperBound do
        β  := max(g, lowerBound + 1)
        g  := AlphaBetaWithMemory(root, β − 1, β, d)
        if g < β then
            upperBound  := g
        else
            lowerBound  := g

    return g
```

Fig.5　MTD(f)
图 5　MTD（f）算法

伪代码（图 5）：

f 是对局面估值的试探值，第一次给予的猜测为 0，后续为前一次搜索的返回值。d 是迭代加深深度优先搜索的深度限制。

4.2　　与MiniMaxDecider类的异同

MTDDecider 类与 MiniMax 类的不同：MTDDecider 类使用 negaMax 算法取代了 MiniMax 类中的 MiniMax 算法，前者是后者的简化，并加入了 Alpha-Beta 剪枝，又使用了 MTD（f）算法来优化剪枝。

相同：negaMax 算法和 MiniMax 算法使用的是同样的搜索树，每个节点代表了棋盘上的一种布局，节点到子节点的转移代表玩家的一种可能下法，使用 heuristic 函数作为局面的评估。

References:

[1]　Negamax - Wikipedia
[2]　MTD(f) - Wikipedia