人工智能导论作业 2：黑白棋游戏

石睿 (211300024、211300024@nju.edu.cn)

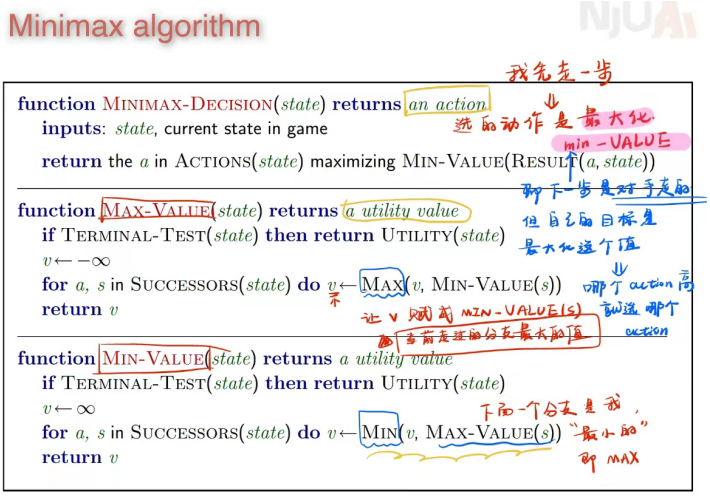
(南京大学 人工智能学院，南京 210093)

摘 要: 基于黑白棋游戏，理解 minimax、alpha-beta 和 MTDD 算法，并做出优化

关键词: minimax 算法；alpha-beta 剪枝；MTDD 算法

中图法分类号: TP301 **文献标识码:** A

1 阅读 MiniMaxDecider.java，理解并介绍 MiniMax 搜索的实现



MiniMaxDecider.java 文件中实现了接口 Decider，其中成员属性相较普通的 minimax 算法增加了 boolean maximize（指示当前向下递归到哪个玩家了，我方－使用“max”算法；对手－使用“min”算法）、conputedStates（以一个 HashMap 记录当前棋盘状态和此时计算出来的 value 值。以备之后不断递归的时候出现了重复状态，可以直接返回 value（调用实现的 heuristic 函数，在第三部分详细阐述）而不用重复计算。）

但在后续代码阅读的过程中发现在不断递归得到 new state 的时候，并没有把 new state 加入到 computedStates 之中，即此 hashmap 只定义却并没有实际应用。可能的原因是出现完全重复棋盘状态的概率太低，而且不断递归需要存储的所有棋盘状态实在过大啦，用巨大的空间开销来换取一点时间开销，有点得不偿失。所以代码后续并没有使用 computedStates，而是所有状态都递归到限定的深度结束。

实现上面伪代码中第一部分函数－－－函数 Action

其中较为巧妙的是依据成员属性 maximize（在构造函数中已经被初始化），来对 value 和 flag 进行赋值，进而通过一个函数，同时实现不同开局的情况的整合【避免了不良的程序设计范式“copy－paste”的出现】。也可以用于后续递归的时在我方决策和对手决策之间的不断转换（只需要取个反即可啦）

举个例子，如果当前是我方进行搜索，maximize=1\flag=1,目标为在对手 mini 所有下一个状态所得的 value 的情况下找到最大的 value。newValue 就是对手尽量 mini 我方收益，flag*newValue>flag*value 的时候（此时 flag 为 1），对 value 赋予更大的 newValue，就是我方挑选出对手最小化之后的值的最大值。

值得注意的是>和>=符号的使用，bestActions.clear()只在 flag*newValue>flag*value 的时候才进行，因为可能会有多个操作最后得到的 value 都是最好的，所以在 value 值相同的时候要保留原来的“最好操作”，最后随机选择一个“最好操作”来返回。而在原来 value 值没有这么好的时候，对原先所有的“最好操作”进行清空，更新为本次操作。

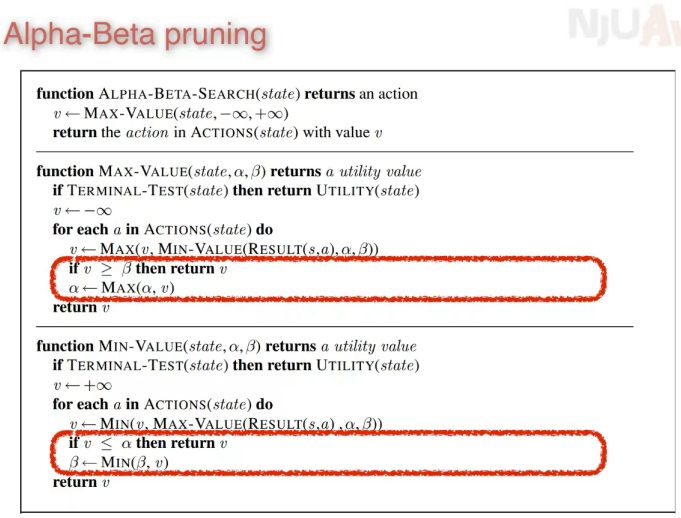
```
for (Action action : state.getActions()) {
    try {
        // Algorithm!
        State newState = action.applyTo(state);//使用当前循环到的action来往下走一步的状态
        float a=Float.NEGATIVE_INFINITY;
        float b=Float.POSITIVE_INFINITY;
        //和ppt类似，此时把min和max合并成一个miniRecursor函数来进行递归求解值
        //newValue是当前状态，依据当前做决定的是我还是对手来选择不同的recursor进行求解
        float newValue = this.miniMaxRecursor(newState, depth: 1, !this.maximize,a,b);
        // Better candidates?
        //即选出所有可走的步骤的最大值（我方出牌）、最小值（对方出牌）
        //【使用乘以（-1）巧妙合并！】
        if (flag * newValue > flag * value) {
            value = newValue;
            bestActions.clear();
            //【因为当前这个值是最好的，所以原来所有的最好情况下的动作现在变的不好啦！】
            //tip:只有在 值是大于符号的时候才加入，因为如果等于的时候清空，之后就没法随便从所有可行的动作中
            //     随机挑选出一个动作啦！
            //要把原来认为是“最好”的动作清空。之后再把这个新的最好的动作加入！
        }
        // Add it to the list of candidates?
        if (flag * newValue >= flag * value)
            bestActions.add(action);
```

分析如图所示（已经加入了 alpha-beta 剪枝后的代码）

实现伪代码第二、第三部分的函数－－函数 miniMaxRecursor（被 Action 调用）

延续 Action 函数的“开关”思想，在不同的情况下给 value 赋值+∞和－∞，并对递归的时候的 maximize 进行取反，即传入的是!maxmize，从而进入对手决策阶段

2 修改 minimaxDecide 类，加入 alpha-beta 剪枝，分析不同方式带来的速度变化



在弄清楚代码结构之后，可以在 miniMaxRecursor 之中引入 alpha-beta 剪枝。同时利用上原框架代码中引入的 maxmize 来判断此时是谁在进行决策。

alpha-beta 的具体思想可用一个例子来说明 max(min(3,8,12),min(2,x,y),min(4,1,z))，在第一个情况分析完成时，alpha 就是 3，而在第二个情况分析的时候，min 一定会小于 2，而在外层 max 的作用下，min(2,x,y)一定不会被选择，所以可以直接返回 2 而不用搜索剩下的 x 和 y（反正这种情况最终也不会被选择）

具体思想如上图伪代码所示，在 miniMaxRecursor 之中加入了如下代码，就可以实现啦。

```
try {
    State childState = action.applyTo(state);
    float newValue = this.miniMaxRecursor(childState, depth + 1, !maximize,a,b);
    //巧妙：对maxmize进行取反，就相当于进入了另外一个对手的对局啦！
    //Record the best value
    //【引入的剪枝，只需要修改a和b的值就可以啦！
    if(maximize ? newValue>=b : newValue<=a)//max和min的合并
        return newValue;
    if(maximize && (a<=newValue))
        a=newValue;
    if(!maximize && (newValue<=b))
        b=newValue;

     //if (flag * newValue > flag * value)
        //value = newValue;
```

以下通过对于不同算法的棋盘初始状态进行相同的两步操作，进行重复实验后取平均值所得的时间如下（各做了 5 次，取平均值。不太会自动化测试 o(┬﹏┬)o）

单位:ns	**Minimax 算法**	**Minimax+alpha-beta 剪枝**
depth=2	$4.218*10^6$	$3.171*10^6$
depth=3	$1.555*10^7$	$6.347*10^6$
depth=4	$2.359*10^7$	$1.32*10^7$
depth=5	$9.161*10^7$	$1.817*10^7$

说明：

• 在 depth<=4 的时候，两种算法的所用时间相差不大，而在 depth>5 的时候，此时搜索次数指数增加，而剪枝可以在得不到更好结果的情况下主动结束本次搜索，进而可以大幅减少运行时间呢

3 理解并优化 heuristic 函数

```
* // Bit-boards representing valid moves
   private short[] p1MoveBoard, p2MoveBoard;
*/
return this.pieceDifferential() +
  8 * this.moveDifferential() +
 300 * this.cornerDifferential() +
  1 * this.stabilityDifferential() +
  winconstant;
```

框架代码中的启发式函数（即伪代码中得到的 ultility value 的函数）有五部分组成

• peiceDifferential：以行遍历棋盘，计算当前双方有用的棋子数量，并返回其差值

• moveDifferential：遍历棋盘上每一个位置，计算当前双方分别可以进行的落子方式，返回其差值

• cornerDifferential：只查找棋盘上的四个对角，返回双方分别占据的个数之差（非常重要！因为四个对角上的棋子不能被翻转颜色，进而对稳定当前我方局势有着重要的作用）

• stabilityDifferential：遍历所有的水平线、竖直线、左对角线、右对角线，返回双方各自拥有的所有不能被翻转的棋子之差（和八皇后的思想相同，以四种可以翻转棋子的方式进行遍历-各存放在一个数据结构中。非常重要！维持我方状态的稳定性-保持我方优势）

• winconstant：依据当前对局状态赋值。如果产生了赢家（游戏结束），我方胜利为 5000，对手胜利为-5000

现在已有的各个启发式函数的比例应该是已经经过了自动化测试以及自动对抗产生的较优的比例啦。因为没有进行自动化测试，所以我也只能给出一个可能的修改思路 o(╥﹏╥)o

注意到四个启发式函数是以加权的方式合并在一起的，而在尝试并理解黑白棋的游戏规则过程中，我发现当前所拥有的棋子数只是昙花一现，对方如果有良好的布局，可以一下子就把很多自己一方的棋子翻转过来，所以 **potential option1**：pieceDifferential 的比例可以再减少。

同时，当看到 corneDifferencial 函数和 stabilityDifferential 函数并进行了几句游戏对抗，就意识到“不可翻转”的棋子对于稳定当前己方优势起到极为重要的作用！所以 **potential option 2**: 可以再增加 cornerDifferential 和 stabilityDifferential 的比例，把“维持当前优势的状态”这一特性的重要性放大。

依据两种可能的 potential option 可以做出的修改如下所示

以及，**potential option 3**: 在对局已经分出胜负的情况下，有两种情况。如果搜索到了当前我可以胜利，那么此时 winconstant 应该占绝对因素，即可以再扩大我方胜利时候的 winconstant（此时起主导因素的是 winconstant）。如果搜索到了当前我最后会失败，此时 winconstant 的比例应该不占绝对因素，即在我会输的情况下，尽量让我和对方的差距减小，提高前面三个启发式函数的占比。

```
return this.pieceDifferential() +//【op1:本行的比例可以缩小】
          8 * this.moveDifferential() +
         300 * this.cornerDifferential() +//【op2:本行的比例还是可以增大】
          1 * this.stabilityDifferential() + //【op2:本行的比例还是可以增大】
          winconstant;//【op3:可以在不同的情况下，给予不同的权重】
    }
```

4 理解 MTDDecider 类

总体上来看：真正运行 MTDDDecider 类的函数是 iterative_deepening。如果定义了 USE_MTDD 这个“宏”，就使用 MTDD 算法得到 value 的值（也和 alpha-beta 剪枝有相似之处）。否则就是用 AlphaBetaWithMemory 来得到 value 的值。

MTDDecider 和 MinimaxDecider 之间的相似之处：

两者在 alpha-beta 剪枝的判断方式上基本相同，而且都是通过的得到 value 的值并且有条件的更新来达到 minimax 算法。

MTDDecider 和 MinimaxDecider 之间的不同之处：

1、在原来给出的 minimax+alpha-beta 剪枝的算法之中，depth（搜索深度）是设定死的。而在 iterative_deepening 之中，通过了一个 for 循环，从深度 1 到 maxdepth 不断增加搜索深度，来进行 value 的赋值。【深度增加的搜索】

2、在搜索深度不断增加的过程中，如果“宏”USE_MTDF 已经定义了，就引入 MTDF 函数中来迭代得到 value 的值，而这种得到 value 的方式和原来的 minimax-alpha-beta 大为不同。

```
if (USE_MTDF)//是不是要使用启发式搜索算法,是-进入 MTDF 启发式算法当中
                    //当前递归调用的时候搜索的深度是 d（d 从 1 一直增加到 maxdepth 呢！）
                    //相似：在 MTDF 中迭代，通过修正上下限，来让 value 尽可能精确
                        value = MTDF(n, (int) a.value, d);//只能获得一个粗略的 beta 值，需要一轮一轮迭代获得
更精确的值！
                    else {//不是-直接进行 ab 剪枝，可以获得精确地 value 值
                        int flag = maximizer ? 1 : -1;
                        value = -AlphaBetaWithMemory(n, -beta , -alpha, d - 1, -flag);
                    }
```

以下为 MTDF 函数的部分截取，在循环中，限制了 alpha-beta 的大小差距（为 1），再调用 AlphaBetaWithMemory 函数进行递归啦（此时传入的 alpha 和 beta 分别是 beta-1 和 beta）。所以这样两者的差距越小，就可以产生更多的剪枝，速度更快。但同时更快的剪枝也就意味着会丢失更多的信息（最优解可能更容易被一次剪枝剪掉），所以需要不断迭代来缩小 value 的上下限来获得更加准确的 value 值！

```
while (lowerbound < upperbound) {//不断循环，直到得到比较精确的g的值（修改上下限来让g更加精确）
    if (g == lowerbound) {
        beta = g + 1;
    } else {
        beta = g;
    }
    // Traditional NegaMax call, just with different bounds
    g = -AlphaBetaWithMemory(root, beta - 1, beta, depth, -flag);
    if (g < beta) {
        upperbound = g;
    } else {
        lowerbound = g;
    }
}
```

其他一些细节上的相似和不同：在 iterative_deepening 中定义了 ActionValuePair，既存储了 action 和对应的 value，可以在到达相同状态的时候直接返回；在 MTDF 中的 g 一开始被赋予了 firstGuess 的参数，通过这个命名可以看出，这也使用了和 alpha-beta 剪枝相同的思想，如果拥有良好搜索的先后顺序，那么就可以大大缩小搜索范围呢！