报告题目：黑白棋游戏

作者姓名：王东宇（学号：141220099、email 地址：141220099@smail.nju.edu.cn）

(南京大学 计算机科学与技术系，南京 210093)

1 引言

黑白棋，又称反棋（Reversi）、奥赛罗棋（Othello）等，游戏使用围棋的棋盘棋子，在 8*8 的棋盘上，黑白双方分别落棋，翻动对方的棋子。本次实验通过阅读及理解源代码，来了解关于对抗博弈的各种思想方法，并在此基础上进行优化改进，加强加深对博弈思想的应用与理解。

2 对抗博弈的方法实现

博弈搜索目前广泛研究的是确定的、二人、零和、完备信息的博弈搜索。也就是说，没有随机因素的博弈在两个人之间进行，在任何一个时刻，一方失去的利益即为另一方得到的利益，不会出现“双赢”的局面，而且在任何时刻，博弈的双方都明确的知道每一个棋子是否存在和存在于哪里。

2.1 MiniMax搜索

在对抗博弈中，先手的一方明显追求的是局面的最优化，但同时它应考虑到后手的一方则会选择局面的最差值。因此先手行棋时应选择子节点的极大值，而每个子节点的极大值是由后手来决定的，故每个子节点的极大值都将是它们各自的子节点的极小值，这就形成了极大极小的博弈过程。每个局面的评价由启发式函数 Heuristic 给出，考虑因素包括边界、行动力、占角等，具体将在 2.3 中进行介绍。

MiniMax 搜索算法需要实现极大值和极小值，课本中分别实现了 Max 函数和 Min 函数，而在源代码中利用了是否追求最大 maximize 这个布尔量作为开关，若追求最大，则利用 maximize 初始化 value 为 Float.NEGATIVE_INFINITY 并设置 flag=1，这样在 flag * newValue > flag * value 时便是选择了较大的 newValue 值，从而实现取极大值；若追求最小，则利用 maximize 初始化 value 为 Float.POSITIVE_INFINITY 并设置 flag=-1，这样在 flag * newValue > flag * value 时便是选择了较小的 newValue 值，从而实现取极小值。无论是取极大值还是取极小值，newValue 都是绝对值较大的那一个。

```
float value = maximize ? Float.NEGATIVE_INFINITY : Float.POSITIVE_INFINITY;
List<Action> bestActions = new ArrayList<>();
// Iterate!
int flag = maximize ? 1 : -1;
```

```
float newValue = this.miniMaxRecursor(childState,  depth: 1, !maximize);
//Record the best value
                if (flag * newValue > flag * value)
                    value = newValue;
```

2.2 AlphaBeta剪枝

在进行 MiniMax 递归搜索时加入 Alpha 和 Beta 参数，表示上下界，只有在界内范围的 newValue 值才有可能继续进行递归，否则将被剪枝。

若直接在源代码的 miniMaxRecursor 中加入 Alpha 和 Beta 参数，剪枝其实是从第三层(根节点为第 0 层)才开始。要使得剪枝从第二层开始，则应同时在 decide 函数中遍历第一层时就维护 Alpha 和 Beta 参数。

使用变量 mode 来对 MiniMax 搜索算法进行选择：

```
/** The mode to be chosen
 *  1-MiniMax without pruning , 2-MiniMax with pruning at depth 3 , 3-MiniMax wtih pruning at depth 2
 */
private final int mode = 3;
```

在 miniMaxRecursor 中加入 Alpha 和 Beta 参数：

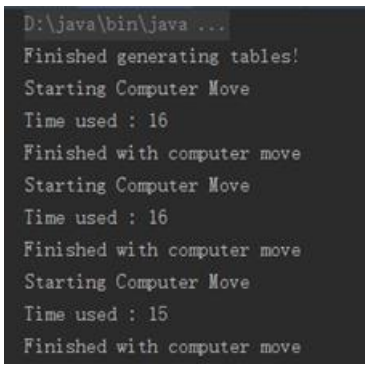
```
//add alpha-beta pruning
if (flag == 1) {
    if (mode == 2) {
        if (value >= beta) return value;
    }
    else {
        if (value > beta) return value;
    }
    alpha = alpha > value ? alpha : value;
}
else {
    if (mode == 2) {
        if (value <= alpha) return value;
    }
    else {
        if (value < alpha) return value;
    }
    beta = beta < value ? beta : value;
}
```

decide 函数中维护 Alpha 和 Beta 参数：

```
if (mode == 3 && flag == 1) {
    alpha = alpha > value ? alpha : value;
}
else if (mode == 3 && flag == -1) {
    beta = beta < value ? beta : value;
}
else {}
```

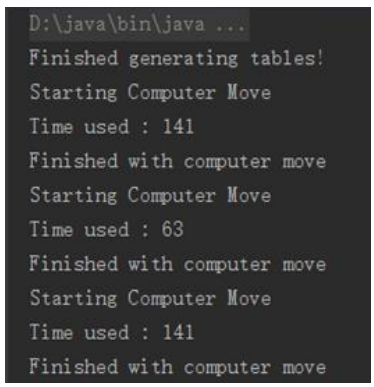
比较剪枝带来的速度变化 (Time used 为电脑进行一次决策的时间)：

(1) 最大深度为 3 (模式 mode 依次为 1，2，3)：



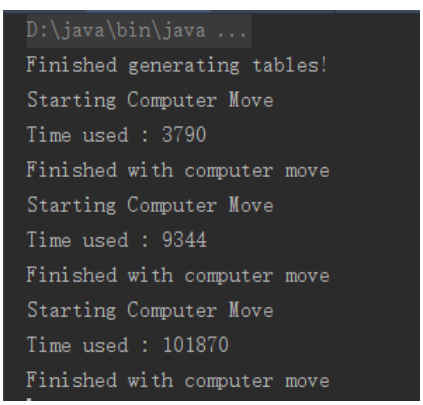
搜索最大深度较小，速度基本相等

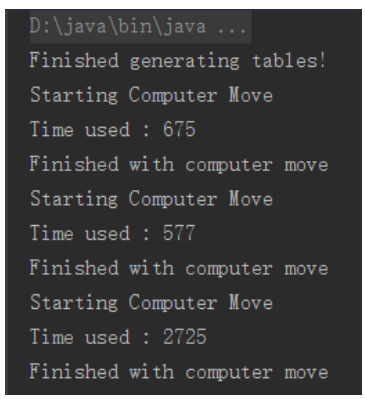
(2) 最大深度为 6 (模式 mode 依次为 1，2，3)：

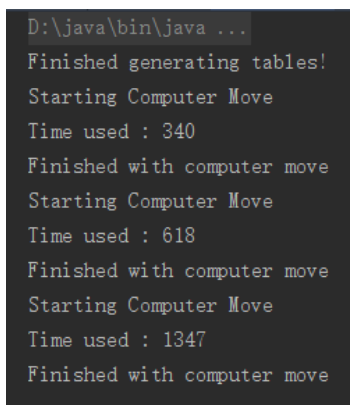


剪枝后速度明显加快，且从第二层比从第三层开始剪枝要有效一些

(3) 最大深度为 8 (模式 mode 依次为 1，2，3)：







剪枝后速度明显加快，且从第二层比从第三层开始剪枝明显加快

2.3 Heuristic启发函数

heuristic 启发函数用于对当前局面进行评价，给出评价值。考虑因素包括：

(1) 棋子数 pieceDifferential：旗子数目越多，局面越占优势，权重为 1；

(2) 可选择位置 moveDifferential：可选择的位置越多，则限制对方的可能性越大，局面越占优势，权重为 8；

(3) 角落位置 cornerDifferential：角落既是稳定子，也能通过与其它子的配合增加自己的稳定子，作用极大，权重为 300；

(4) 可翻转子数 stabilityDifferential：大致判断当前局面下，下一步能将对手几个子翻转过来以及自己将有几个子会被对手翻转过来，权重为 1；

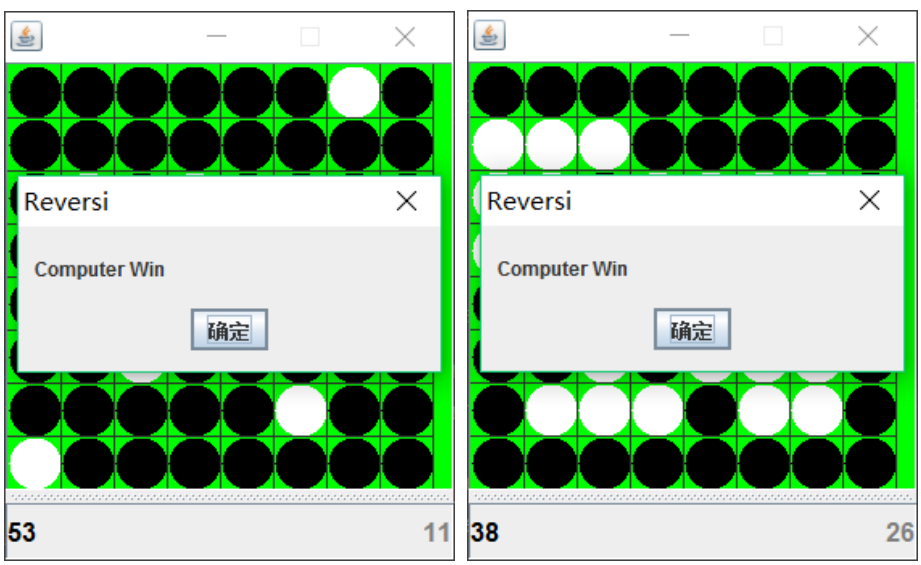
新增因素：

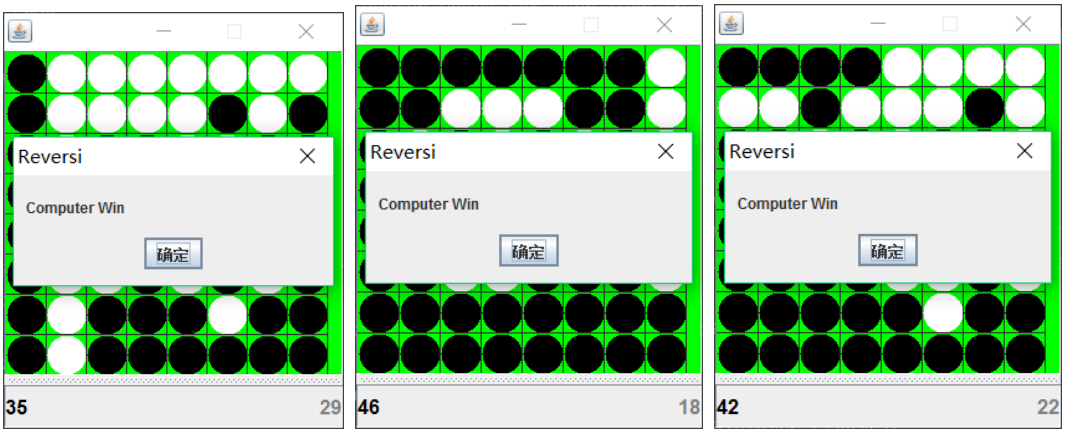
(5) 星位 X-Square XSquareDifferential：占据星位通常情况下会导致失去角落位置，造成重大损失，因此将被罚分；但同时有时占据星位可作为战术的一个部分，有利于最终赢得比赛。故权重设置为 1；

(6) 稳定子：无论棋局如何发展，都不会被翻转过来的棋子，通常和角落的棋子有关系。因时间和能力有限，未能完成。

对局情况：由于通过手工点击棋盘进行操作较为繁琐，因此将源代码中需要手动点击棋盘的代码更换为电脑行动，这样便形成了电脑 vs 电脑的情况。这时在 OthelloState 类中进行判断，并选择不同的启发式函数即可；

改进后的启发式函数作为先手(黑棋)：胜率几乎为 100%





改进后的启发式函数作为后手(白棋)：胜率接近 70%

2.4 MTDDecider

在 MTDDecider 类中综合运用了多种博弈过程中所需要的搜索方法及辅助手段，这些方法依次调用，高层的函数依赖于底层函数的正确实现，底层函数给予高层函数以便利和支持。依据函数调用链，这些方法自底向上下面将一一进行介绍：

(1) 具有记忆功能的 AlphaBeta 搜索：参数包括当前的状态 state，Alpha 和 Beta 值，深度 depth 和极大极小布尔量。由于设置了时间，因此在传入的 depth 过大时，应检查是否超时。此方法的记忆功能由转置表 transpositionTable 来实现，表中存放着棋盘状态及状态所对应的信息结点 node。

算法首先利用 state 到转置表中进行查找匹配，若找到 node 且 node.depth>=depth (>=的原因后面会有说明)，则说明在之前的搜索过程已经对当前状态进行过搜索，所以直接返回 node.value 即可代表当前状态的评价值。未找到，则判断当前局面是否需要进行搜索，若 depth=0 或者游戏已经不在进行，则调用 heuristic 函数对当前局面直接进行判断，返回评价值的同时构建 node 结点存入转置表中，记录下状态。当需要进行搜索时，若当前 depth 较小，系统搜索所有结果所花时间较少，可直接进行搜索；不然，先对 depth-2 的深度进行搜索，这样能够花费较少的时间的同时使得转置表多增加一些状态信息，从而在 depth 的深度搜索过程能够有更多的 cache 命中。

搜索过程依次进行递归调用，Alpha-Beta 剪枝，产生新的 value 值，并对动作集合 actions 进行排序，供其它函数调用。

(2) MTDF 搜索：参数包括根节点状态 root，初始值 firstGuess 和深度 depth。初始时算法首先规定好上下界，并将初始值设置在上下界之间。然后不断以零窗口[beta-1，beta]作为 Alpha-Beta 对 AlphaBeta 搜索函数进行调用，并以返回值来不断更新上下界，直到下界大于上界，从而达到趋近最优解的评价的目的。由于调用的 AlphaBeta 搜索中存在着转置表，这样使得 MTDF 的搜索速度能够非常的迅速，效率较一般的搜索算法有大幅的提高。

(3) 遍历深化搜索：参数仅包括根节点状态 root，算法顾名思义，就是从 root 出发不断地对深度进行增加，多次进行遍历搜索。由于搜索的过程是通过调用 MTDF 搜索或 AlphaBeta 搜索来完成，这其中都会存在着转置表对状态进行记录，因此虽然搜索过程的深度在不断的增加，但有了前面深度的搜索的缓存，效率会大大提高。同时，在较低层进行搜索完成之后会对 actions 集合进行排序，这样使得下一次对 actions 集合进行遍历时，就有可能会先搜索到一些好的分支，而其它大量的分支将会被剪掉，无疑又是一次效率的提示。

搜索结束的条件是超时或者深度达到最大深度。由于源代码将最大深度设置为 64，这说明即使搜索完整个棋盘也不会达到最大深度，故结束的条件只有超过初始的设置时间。由于调用完 AlphaBeta 函数后默认 actions 集合的第一个 action 为最优值，因此在超时异常中，我们判断第一个 action 是否比其它的最近一层探索过的 action 都要好，若成立则我们 reset 所有 action 为上一层的估计值，将上一层的估计值作为选择最优解的标准；不然说明最近一层的第一个 action 较差，我们只 reset 最近一层探索过的其它 action 为上一层的估计值，这样在选择最优解时我们就能够了解到下一层的第一个 action 的估计值较差，就会避免去选择它作为最优解。

与 MiniMaxDecider 类的异同：

(1)相同之处：

a) 两者都在搜索的过程中运用到了 AlphaBeta 剪枝，提高了效率；

(2)不同点：两者的不同点主要是 AlphaBeta 剪枝搜索算法的不同

a) MTDDecider 的搜索过程使用到转置表进行状态记录，而 MiniMaxDecider 则每次都是重新计算；

b) MTDDecider 搜索时会对 actions 集合进行排序，这样可以提高先搜到较好分支而剪去大量较差分支

的概率，而 MiniMaxDecider 的剪枝效率则和分支的好坏分布有着较大的关系；

c) MTDDecider 进行剪枝搜索时用的是 NegaMax 算法，该算法分别将-beta 和-alpha 代替原先的 alpha 和 beta 作为参数，这样使得程序员不需要在意当前是求极大值还是极小值，只需将 value 值与参数 alpha 进行比较即可更新，便于代码的编写，而 MiniMaxDecider 中的算法为 MiniMax 算法，需要考虑当前的求值来决定与 alpha 或 beta 进行比较，然后更新。

3 结束语

本次实验由于代码量较少，阅读和理解代码所占比重较大，因此调试基本无从说起。单从源代码去理解算法难免到处碰壁，导致灰心丧气。但只要静下心来一点点琢磨，也能一点点地发现代码中的乐趣，也更能体会到茅塞顿开的喜悦。

致谢 在此,我向提供我支持与帮助的王洪和王鸿基舍友表示感谢.

References:

[1] http://blog.163.com/xialingge2006@126/blog/static/10282290200939113615999/

[2] http://www.xqbase.com/computer/advanced_intro2.htm

[3] http://blog.csdn.net/u012501320/article/details/25098401

附中文参考文献:

[4] 人工智能 一种现代的方法(第三版)