HW2：黑白棋实验报告

匡亚明学院 张子谦 191240076 191240076@smail.nju.edu.cn

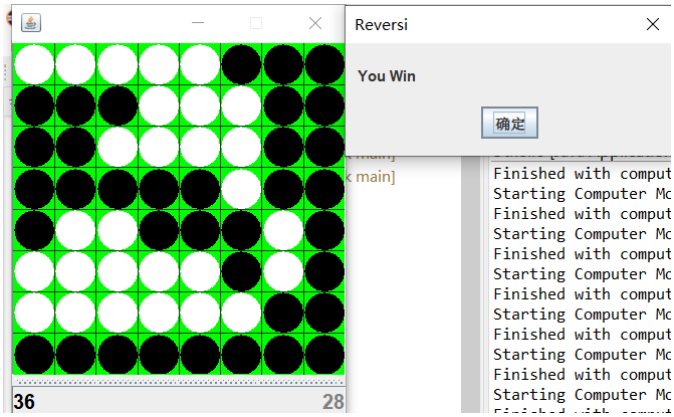
本人承诺该实验全程由本人独立自主完成，无抄袭或给予他人抄袭行为，代码已上传至本人github

本实验在阅读Brain Rose所著《A Minute to Learn... A Lifetime to Master》(译为《黑白棋指南》，已附于文件)第一部分后完成，相关概念策略基于此书基础进行吸收理解后在实验中进行阐述

摘要：黑白棋是经典的零和博弈问题，双方按照规则轮流落子、翻转棋子，直到一方棋盘上无子或者双方都无子可下。本实验在原有框架代码基础上，对MiniMax算法进行理解并优化，增加了$\alpha-\beta$剪枝。同时在对规则与一般策略的理解基础上对heuristic函数进行了修正改进，使得Agent对战胜率得到提升。最后阅读MTDDecider相关部分代码，理解MTD(f)算法。

1. 引言

黑白棋AI建立在经典的MiniMax博弈算法上，随着搜索层数加深，Agent构造出更全面的搜索树，并能选取确定深度下的最优解。在未改进heuristic函数前，人机对战中，搜索层数设置为2时，可以在大部分对局中获胜，当层数提升为5时，在5局对战中只有一局取得胜利（如下图所示）。



2. 实验内容

Task1: MiniMax搜索的实现

为避免Copy-Paste，程序使用maximize开关将Max-Value,Min-Value进行合并。 下用伪代码辅以注释介绍MiniMax的实现。

```
public Action decide(State state) {
    // set value
    if maximize set value -oo;
    else set value +oo;
    List bestActions;
    // set flag
    int flag = maximize ? 1 : -1;
    //Iterate
    for (Action action : state.getActions()) {
        //execute this action
        State newState = action.applyTo(state);
        if(maxmize)
```

```
            value = max(value,MiniMax(newState,minmize))
        else
            value = min(value,MiniMax(newState,maxmize))
        float newValue = this.miniMaxRecursor(newState, 1, !this.maximize);
        refresh value according to newValue;
        add the actions to bestActions;
        return random best action;
    }
```

decide() 函数先确定这一步是maxmize得分还是minmize得分，然后遍历当前情况下所有可行步，调用minimax函数，考虑执行该步骤后的得分情况，获取最高（低）分，将得分最高的所有情况置入bestActions，如果多个步得分相同最高，则从中随机选取。

```
public float miniMaxRecursor(State state, int depth, boolean maximize) {
    // Has this state already been computed?
    if (state has been computed before || finished || reach the depth)
        return corresponding value -- heuristic score or final score;
    // set value
    if maximize set value -oo;
    else set value +oo;
    //set flag
    int flag = maximize ? 1 : -1;
    List test =available actions;
    for (Action action : test) {
        //execute this action
        State childState = action.applyTo(state);
        //recursive call
        if(maxmize)
            value = max(value,MiniMax(newState,depth+1,minmize))
        else
            value = min(value,MiniMax(newState,depth+1,maxmize))
        refresh value
        }
        Store it to computedlist
        return value;
    }
```

MiniMax搜索部分大部分内容与decide()函数内容相似。

首先进行判断，如果当前state已经计算过则直接返回，避免多余的时间开销，如果达到深度，则调用heuristic函数评估当前局面，如果游戏结束则直接数黑白棋子查看是否获胜。之后根据maxmize or minimize一层层回溯赋予节点value。

接下来遍历可行步，根据当前节点应maximize还是minimize 得分，调用minimax函数，根据得分情况更新value。

需要注意，在本该将已计算过局面放入哈希表的finalize()函数中并没有进行这一操作。

Task2: $\alpha-\beta$ 剪枝的实现

为方便对比，添加is_pruning变量作为开关确定是否进行剪枝。

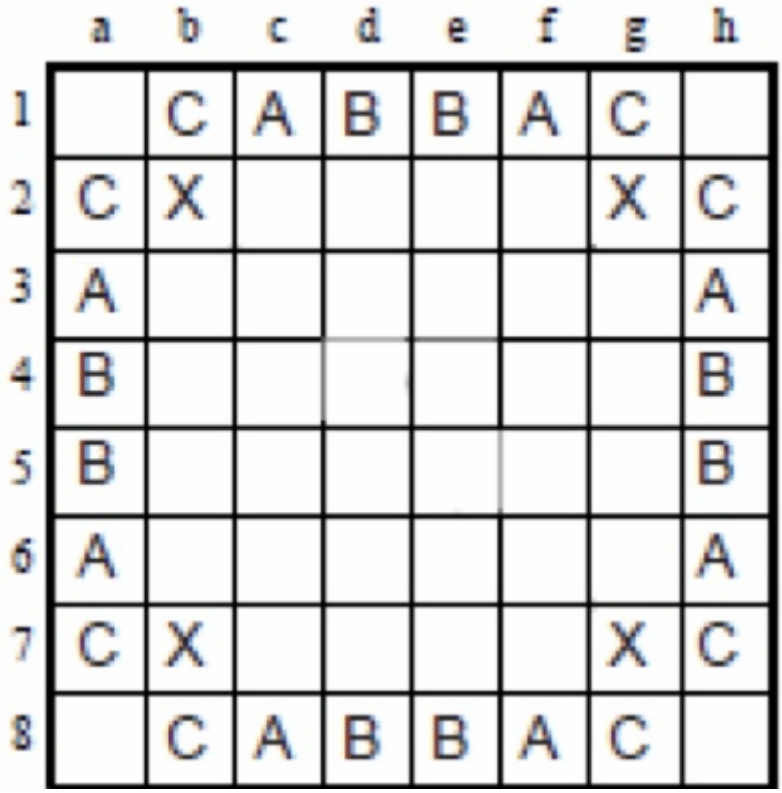
```
if(is_pruning) {
//alpha-beta pruning
    if(maximize) {
        if(alpha<value) alpha = value;
     }
    else {
    if(beta>value)  beta = value;
    }
    if(alpha>=beta)
        return finalize(state,value);
}
```

递归层数	未剪枝总耗时/ms	剪枝后总耗时/ms
4	572	239
5	2585	1012
6	22993	1737

通过调用System.currentTimeMillis()函数获得执行一次decide所需时间，并得出总耗时，递归层数为2、3时因为决策树规模不大，所以效率提升不明显，当递归层数增长为4-6时可见通过剪枝后，耗时显著减少，效率提升很大。且随着递归层数加深，人机对战中获胜可能性也在减小。

Task3: heuristic 函数理解与改进

在进行改进前先对黑白棋进行一些基本介绍，棋盘划分如下图所示。作出以下定义：



角：途中(a,1);(h,1);(a,8);(h,8)四个点

X点/星位：图中标识X位置。

C点/外星位：图中标识C位置。

A点：图中标识A位置。

B点：图中标识B位置。

确定子：永远无法被翻转的棋子。

边界子：与一个或多个空格相邻的棋子。

内部子：完全被其他棋子围住的棋子。

发散手：产生多个新边界子的棋步。

凝聚手：产生很少边界子的棋步。

行动力：合法棋步的个数。

稳定边：由某一条边界ABBA位置全部为同色棋子所占构成的边。

黑白棋最基本的策略便是占角，因为角五个方向被边界所束，天然地无法被翻转，因此构成确定子，由某一个角向两条边延伸出的棋子也因角的保护自然构成确定子，在一定条件下，其向对角线方向延伸出的棋子也成为稳定子，因此一旦占住一个角，常常可能增加很多确定子，从而构建优势。

根据规则，一方想要占得角的位置，必须要求对方在星位，外星位至少有一枚棋子，才能进行翻转。因而这两种位置在某个角未被占领时格外危险，尽量不应下在这两处。而相应的，如果己方在A,B点有棋子，那么当对方下到星位或外星位时，变更加容易发动攻击，同时由于处在边界位置，其只在边界两边会受到进攻，形成稳定边后，具有较强的防守能力，因而这两个位置也具有较高的优先级。

而为了迫使对方下到星位或外星位以帮助自己抢占角，就需要对对方的行动力进行削减，迫使对方下到危险位置。因此在大部分情况下要增加凝聚手，减少发散手。

在此理论基础上对原heuristic函数进行分析：

```
pieceDifferential()
moveDifferential()
cornerDifferential()
stabilityDifferential()
winconstant
```

第一个函数为双方棋子数差值，由于在终局前，行动力优先级显著高于棋子数差值，且过多的棋子容易形成边界子，进而构成墙，增加对方行动力，削减自身行动力，因而权重设置为1；

第二个函数为双方行动力差值，权重设置为8；

第三个函数为双方占角数差值，权重设置为300；

第四个函数为双方可翻转棋子的差值，权重设置为1。需注意，这里的stability并非前文所述确定子，而是考虑在当前局面下能否通过某一步合法操作进行翻转的棋子。此函数实质所反映的是近似于边界子与内部子的性质。

winconstant标识是否取得胜利，value设置为5000；

在之前分析的基础上增加了以下函数：

```
starDifferential()
cDifferential()
aDifferential()
bDifferential()
blockDifferential()
```

分别判断是否下到星位或外星位（在对应角未被占领的情况下），A位，B位，是否能构成稳定边；

分别给予权重，并在对战中进行调整，最终为 -60，-30，30，20，100；

并对原函数进行调整，stabilityDifferential权重调整为6，cornerDifferential权重调整为400；同时考虑到进入终局阶段，此时翻转棋子数更加重要，设置pieceDifferential()乘以系数 $1+e^{棋子数-56}$；

改进前，与C.Reversi对战，搜索深度5层时只能战胜Level1-4，在面对对方棋子弃角楔入时不具有足够的防范意识，同时对于占AB位进攻角的意识不够强；改进后可以战胜level1-8，分析对局时注意到，即使进行了改进，对于偶数理论，迫敌筑墙等更为高级的技巧缺少防范，由于时间问题，本人并未对相关策略进行深入研读，暂时无法进一步进行改进。

Task4: MTDDecider算法理解

在介绍MTD(f)算法前，先介绍PVS算法。在$\alpha-\beta$剪枝中，我们称对于一个节点的value， 如有$\alpha<value<\beta$，则该节点为PV（Principle Variation）。在过程中我们所关注的是区间 $[\alpha,\beta]$ ，也称为搜索窗口，窗口的宽度很大程度上决定了搜索的耗时程度。如果窗口宽度很小，那么很容易剪裁掉搜索树中的大部分，从而减少搜索时间，特别的，当窗口变为 $[\beta-1,\beta]$时，我们称为 *零宽窗口*（极小窗口）。因此假设之前已经找到最佳的估值为α，然后应用零宽窗口，快速辨别此假设的正确性。如果搜索得到的value值小于等于上界α，则说明估值正确。否则则会再进行一次常规窗口搜索。

而MTD(f)则完全使用零窗口搜索，对于一段区间$[lowerbound,upperbound]$ 进行搜索，则先进行一次guess,猜值为value, 置β 值为value， 对窗口 $[\beta-1,\beta]$ 进行剪枝(AlphaBetaWithMemory)，如果 新的 $value<\beta$ ，则更新估值上界， $upperbound=guess$，否则更新估值下界 $lowerbound=guess$ 。将新得到的value作为新的窗口，继续搜索，不断重复直到 $lowerbound>=upperbound$。

同时应用置换表技术，将已搜索过的节点存储，减少重复搜索的时间开销。

3. 结束语

本次实验从MiniMax博弈算法出发，探讨了$\alpha-\beta$ 剪枝和 MTD(f) 两种提高搜索速度的方法，并通过对heuristic改进提升Agent的水平。通过对比，剪枝显著的提升了搜索的速度，同时在测试中，也发现改进heuristic函数后的Agent在对战中水平也得到了提高。但为了让其有更加优秀的表现，需要对黑白棋策略有进一步的研究，对更多技巧有深入本质的理解。

References：课件 Lecture 4

《A Minute to Learn... A Lifetime to Master》 -- Brain Rose