作业2:黑白棋游戏

张祎扬 （181840326 181840326@smail.nju.edu.cn)

(南京大学 匡亚明学院 南京 210046)

摘　要：本次试验借用黑白棋游戏为载体，通过对MinMax极小极大值算法的探索和改进，展示了博弈问题中常见的搜索算法和其搜索性能。

关键词：黑白棋、MinMax、$\alpha-\beta$剪枝、博弈

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。

本实验使用java语言编写黑白棋的游戏框架，通过以下几个任务对博弈问题中常用的MinMax极小极大值算法进行了探索和改进，深入理解了算法的具体内容和性能优劣以及提升方法。

2 实验内容

2.1 任务1

阅读源代码MiniMaxDecider.java，理解并介绍MiniMax搜索的实现

在MinMaxDecider类的开始定义了几个全局变量。`boolean maximize`用来指示当前状态是max还是min，true表示max，false表示min。`depth`表示搜索树的深度。`computedStates`是一个map，用来存放键值对，以记录搜索过的状态，防止重复。然后对这些变量进行初始化。

下面按照调用顺序来介绍这些方法。

float finalize(State state,float value)

未定义，直接返回函数的参数`value`.

float miniMaxRecursor(State state,int depth, boolean maximize)

这个方法用递归的方法来求每个状态的最大最小值。

在课本上，min和max是分别实现的，而程序中使用一个开关将两者合并为一个函数，关键就在于这个`boolean`型变量`maximize`.`flag`变量由`maximize`的值定义。`int flag = maximize ? 1 : -1;`语句根据`maximize`的值给`flag`赋值，当`maximize`为true（也就是在max结点）时设置`flag`为1，当`maximize`为`false`（也就是在min结点）时设置`flag`为-1.这样做就把最大值和最小值的比较统一起来，实际比较的时候比较的是`value`和`flag`的乘积。当求最大值当时候，value为正，只需通过比较找到value的最大值即可。当求最小值的时候，因为`value*flag`是一个负数，而要找value的最小值也就是求`value*flag`的最大值。这样就把两种情况统一起来，在比较的时候不需要分类，可以直接同时找到max和min值，**这是一个很妙的方法**。

接下来分析代码：

第一个`if`语句判断当前状态是不是已经搜索过（是不是已经在map中），如果是的话直接返回通过键查找的值，避免重复计算。

第二个`if`语句判断游戏游戏是否结束，如果结束则返回由当前状态通过**启发式函数**计算的分数。

第三个`if`语句判断搜索深度是否达到最大，如果是的话就返回**启发式函数**对当前状态的计算分数。

接下来是递归的主体部分。`List<Action> test`是对当前局面的所有可能产生的动作。通过一个循环遍历这些动作，用`childState`表示对当前局面应用action以后的新局面。用`newValue`来表示新局面的minmax值（递归，depth+1，maximize值取反）。然后再通过一个`if`语句对两个value进行比较，从而更新最优的value值。**在这里，根据上文对flag的分析，最优意味着当前是max结点就取最大值，当前如果是min结点就取最小值。**`try-catch`语句的目的是异常处理。

方法返回最优的value值。

Action decide(State state)

bestActions是一个变长列表，用来存放最佳的动作。

主体部分对当前状态的可用动作列表进行循环。在每一次循环中，`newState`表示应用该动作后的新局面，`newValue`表示新局面的启发函数计算值，第一个`if`语句的比较条件是`>`，这保证了取得最优解，第二个`if`语句的比较条件是`>=`，这考虑了多个最优解的情况（即minmax值相同）。同样地，`try-catch`语句的目的是异常处理。

最终用`Collections.shuffle(bestActions);`语句来随机排列bestActions列表中元素的顺序，在返回列表中的第一个元素，这本质上等价于如果有多个最优解，则随机返回任意一个。

最终该方法返回一个最佳动作并执行它。

总的来说，minmaxRecursor方法用递归的方式求解各结点的minmax值，decide方法对当前状态的所有可行动作调用minmaxRecursor方法，并返回minmax最优的动作。（由于flag的巧妙设置，这个最优表现为在max结点求最大值，而在min结点求最小值）。

2.2 任务2

2.2.1

修改MiniMaxDecider类，加入AlphaBeta剪枝，并且比较引入剪枝带来的速度变化（尝试不同的搜索最大深度）

根据书[1]上的介绍，极大极小搜索是深度优先的，所以任何时候只需考虑树中某条单一路径上的结点。$\alpha-\beta$剪枝的名称取自描述这条路径上的回传值的两个参数：

$\alpha=$到目前为止路径上发现的MAX的最佳（即极大值）选择

$\beta=$到目前为止路径上发现的MIN的最佳（即极小值）选择

搜索中不断更新α和β的值，并且当某个节点的值比目前的MAX的α或者MIN的β值更差的时候裁剪此结点剩下的分支（终止递归调用）。

同时书上给出了一段完整算法的伪代码(以MAX-VALUE为例)

```
function MAX-VALUE(state,alpha,beta)returns a utility value
  if TERMINAL-TEST(state)then return UTILITY(state)
  v<- -infinity
  for each a in ACTIONS(state)do
    v<- MAX(v,MIN-VALUE(RESULT(s,a),alpha,beta))
    if v>= beta then return v
    alpha<- MAX(alpha,v)
  return v
```

参考这段代码对任务1中的源代码MiniMaxDecider.java进行修改，加入$\alpha-\beta$剪枝，主要是对miniMaxRecursor方法中的for循环进行了修改，修改后的代码如下：

```
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+1,!maximize,alpha,beta);
        //Record the best value
        if (flag * newValue > flag * value){
          value = newValue;
          //alpha-beta cut
          if((maximize) && (value>alpha)){
            if(value>=beta)
              return finalize(state,value);
            alpha=value;
          }else if((!maximize) && (value<beta)){
            if(value<=alpha)
              return finalize(state,value);
            beta=value;
          }
        }
      } catch (InvalidActionException e) {
         //Should not go here
        throw new RuntimeException("Invalid action!");
      }
```

2.2.2

接下来对修改后的代码和修改前的代码进行对比测试，不断增加搜索最大深度，比较速度变化

Depth	without cut	Alpha-beta cut
2	几乎没有延迟，在人下棋的同时电脑可以做出反应	没有延迟
3	比之前有了一点延迟，但几乎可以忽略不计，电脑还是迅速作出反应	没有延迟
4	延迟还是可以忽略不计	没有延迟
5	在有些步骤有接近半秒的延迟	没有延迟
6	有半秒左右的延迟	几乎没有延迟
7	刚开始的步骤有一秒左右延迟，在某些步骤可能反应时间增加至好几秒，再往后增加至十多秒	偶尔出现一秒不到的延迟
8	第一步就有好几秒延迟，到前五步就已经有十几秒的延迟了。同时电脑负载很重，风扇声音很大，发热。没有继续进行到游戏结束。	基本没有延迟
9	第一步有五秒延迟，第二步有29秒延迟，没有继续下去。	偶尔出现好几秒的延迟
10	没有进行下去	延迟基本都在十秒以内

从剪枝前与剪枝后的对比可以看出，当搜索深度比较小的时候，搜索速度本身就很快，几乎感觉不到延迟，所以没有什么太大的区别。而当搜索深度大于7以后，假如没有加入$\alpha-\beta$剪枝，搜索时间将迅速增加。（根据书[2]上解释，min-max实际上仍然是一种深度优先搜索，所以搜索的数量是呈指数级增长的，我们确实也可以从实际执行中发现，随着深度达到一定程度之后，等待电脑走棋的时间将快速变长）。当加入$\alpha-\beta$剪枝算法以后，在同样深度下的表现明显优于原来的算法，在深度较大时差异更为明显。当原算法所导致的时间延迟已经难以正常游戏时，剪枝后的算法仍然将时间的延迟保持在可以接受的范围内。

由此可见，引入剪枝带来的性能提升是很大的。

2.3 任务3

2.3.1

理解othello.OthelloState类中的heuristic函数

在`heuristic`函数中，定义了一个int型变量winconstant，表示得分。如果赢了则得分5000，如果输了则得分-5000。函数的返回值是最终计算的总得分，除了`winconstant`以外，还有四项，分别是`pieceDifferential()`,`8*moveDifferential()`,`300*cornerDifferential()`,`stabilityDifferential()`.

pieceDifferential()

在`hBorad`中逐行循环，对每一行棋子差值求和，返回两个玩家之间棋子的差值，权重为1分。

moveDifferential()

通过每一个`dimension`的循环，求玩家1和玩家2的可移动棋子的数量的差值，权重为8分。

cornerDifferential()

返回玩家1和玩家2占据顶角的棋子个数的差值，权重为300分。

stabilityDifferential()

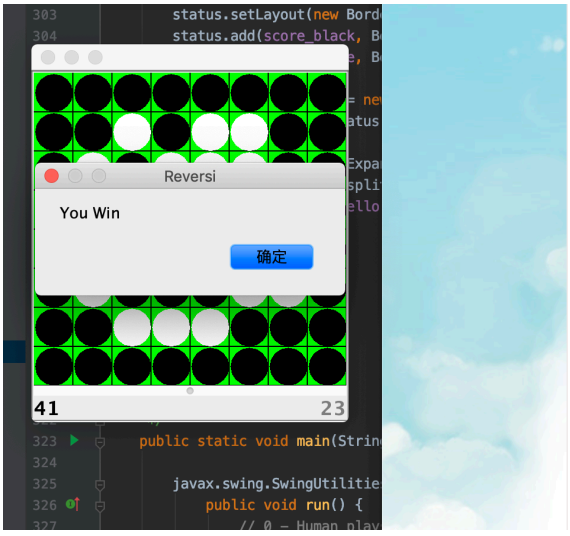
分别对每一行、每一列、每一条对角线求玩家1和玩家2可以翻转的棋子的差值，权重为1分。

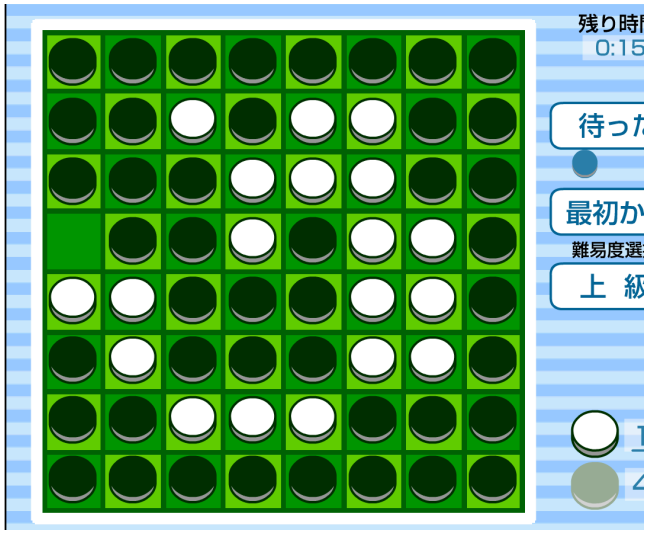
最后`heuristic`函数的返回值就是把以上各个量按照权重求和。再加上关于输赢的`winconstant`,可以注意到`winconstant`的值特别大，因为胜负结果显然是最重要的。其次`cornerDifferential()`的权重相比其他的也很大，这是因为四个角落的位置对于胜利是至关重要的，如果失去了四个角，就已经奠定了输局。这是因为四个角落的棋子无法被反转，一旦占据就是永久占据。并且无论是从横排还是竖排甚至对角线来看，顶角的棋子都位于棋盘的最外缘，很容易直接反转一整排，对结果的影响是巨大的。

2.3.2

尝试改进函数`heuristic()`

在改进之前，我想先测试一下现有函数的强大程度，但是由于我本人真的很菜（从来没有赢过电脑），我决定去网上开一个黑白棋小游戏（**我相信作为游戏推出的算法应该是比较强大的**），让网上的游戏代替我本人来和java项目下棋。**我直接在4399小游戏中选择了变态难度**，结果如下,23:41

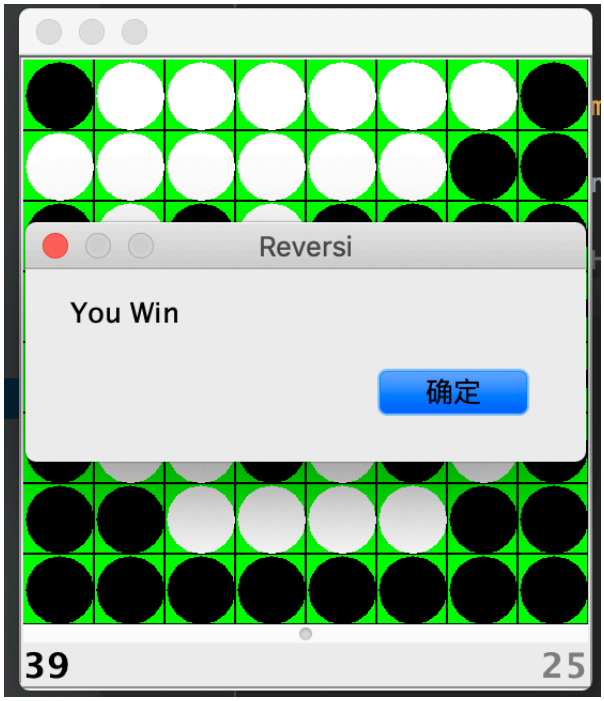




接下来我对`heuristic()`函数进行改进。根据我对黑白棋规则的理解，因为占据四个角落可以获得绝对优势，这已经在算法中实现了，而**占据角落的条件是**角落的斜对角的棋子必须是对方的棋子，所以我实现一个函数用来判断四个内角的棋子的差值。

```
//imitate the function cornerDifferential()
private float subcornerDifferential(){
    float diff = 0;
    short[] subcorners = new short[4];
    subcorners[0] = getSpotOnLine(hBoard[1],(byte)1);
    subcorners[1] = getSpotOnLine(hBoard[1],(byte)(dimension-2));
    subcorners[2] = getSpotOnLine(hBoard[dimension-2],(byte)1);
    subcorners[3] = getSpotOnLine(hBoard[dimension-2],(byte)(dimension-2));
    for(short subcorner : subcorners) if (subcorner != 0) diff += subcorner == 2 ? 1 : -1;
    return diff;
}
```

因为这个棋子最好想方设法让对方占据，所以我给它赋权重-200（没有顶角重要）结果是39:25，（似乎好了一点）

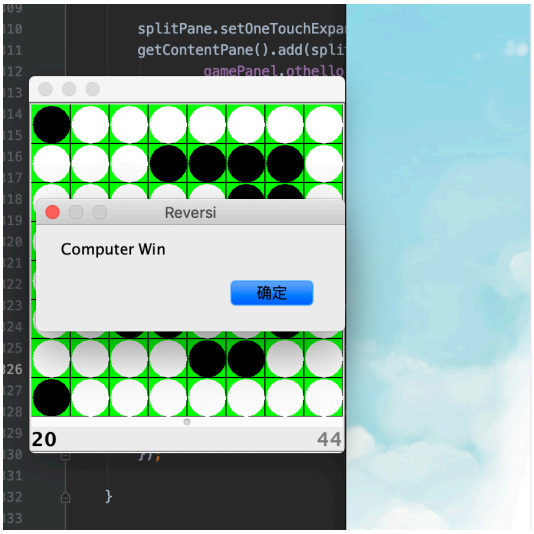


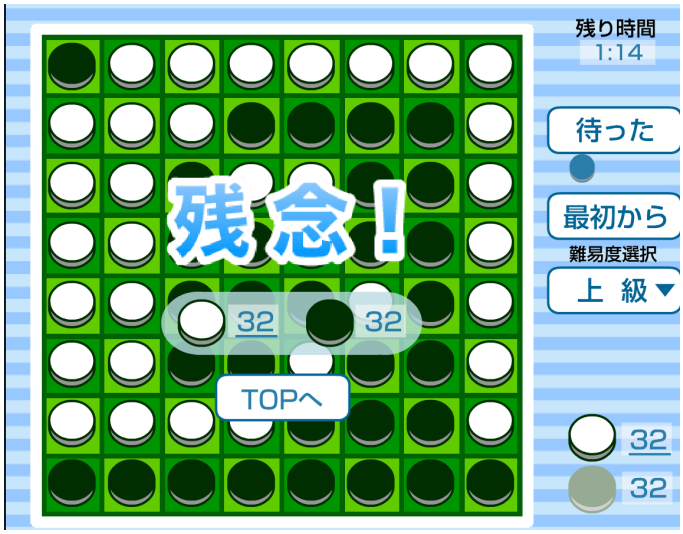
再考虑一下游戏规则，我觉得占据边上的位置也很重要，因为边上的棋子在横、竖、对角线四个方向上只有一种可能被翻转，它翻转的概率很小，所以权重应该大于普通的棋子，我给它赋权重50.

同时代码如下：

```
 private float edgeDifferential(){
   float difference = 0;
   difference += pointTable[hBoard[0] + 32768];  //the first line
   difference += pointTable[hBoard[dimension-1] + 32768];  //the last line
   difference += pointTable[vBoard[0] + 32768];  //the first column
   difference += pointTable[vBoard[dimension-1] + 32768];  //the last column
   return difference;
 }
```

同时由于上一次的结果不是很理想，我把顶角斜对面的棋子的权重改为-100，然后给边缘的棋子赋权重50.我并没有对自己的算法期望太高，觉得提升最多体现在棋子数量的悬殊上。不过**万万没想到的是**，它竟然赢了4399小游戏.....





这给了我莫大的信心，事实证明：

1. 我比电脑菜得多。
2. 我对`heuristic()`函数的优化起到了积极正面的作用。优化主要体现在增加边缘的棋子的权重，以及尽量使顶角斜对角的位置放置对方的棋子。

最终`heuristic()`函数的返回值是：

```
return this.pieceDifferential() +
  8 * this.moveDifferential() +
  300 * this.cornerDifferential() +
  1 * this.stabilityDifferential() +
  (-100) * this.subcornerDifferential() +
  50 * this.edgeDifferential() +
  winconstant;
```

2.4 任务4

阅读并尽量理解MTDDecider类，介绍它与MiniMaxDecider类的异同

在MTDDecider类中,`decide()`函数的返回值是`iterative_deepening()`函数。

`iterative_deepening()`函数对搜索树进行迭代加深地搜索，根据`boolean`型变量`USE_MTDF`来决定使用`MTDF()`函数或者`AlphaBetaWithMemory()`函数。

`AlphaBetaWithMemory()`函数用$\alpha-\beta$剪枝的方法递归地搜索并返回最佳的得分。首先检查当前状态是否已经搜索过，这就需要用get方法看它在不在转换表中。接下来对搜索开始状态和结束状态进行检查。函数的主体部分是一个for循环，其中递归调用自身，并且加入了$\alpha-\beta$算法，**检查bestValue是否在上下界之内**，如果不在就剪掉该枝。

`MTDF()`函数主要使用$\alpha-\beta$剪枝算法，不断地改变上下界，来递归地计算根结点的minmax值。下面分析`MTDF()`的伪代码：

```
private int MTDF(State root,int firstGuess,int depth){
  int g=firstGuess;
  int upper = + infinity;
  int lower = - infinity;
  while(lowerbound<upperbound){
    beta = max(g,lowerbound+1);
    update g with AlphaBetaWithMemory();
    if(g<beta)
      upperbound=g;
    else
      lowerbound=g;
  }
  return g;
}
```

这其中，firstGuess是猜测的值，`MTDF()`函数用最快的方式不断缩小范围的上界和下界之间的距离，当上界等于下界时找到最优值。（while循环的条件是上界大于下界）同时用$\alpha-\beta$剪枝算法搜索返回的值来更新上界或下界。

与MiniMaxDecider类的异同

相同之处：都采用minimax极小极大值算法，用极小极大值来评估局面。同时在每一条路径上其实都是深度优先搜索，并且可以通过`depth`参数控制搜索深度。

不同之处：MTDDecider类在MiniMaxDecider基础上加入了$\alpha-\beta$剪枝算法，大大提升了效率和速度。同时在MTDDecider中还有一个非常重要的`TranspositionTable`,也就是转换表，用来存储所有搜索过的状态，来避免重复搜索计算。从函数名`AlphaBetaWithMemory()`也可以看出这是有“记忆”的$\alpha-\beta$剪枝算法，就是在于它能记住（存储）搜索过的状态，从而避免重复的计算和搜索。

3 结束语

这篇实验报告记录了我在MinMax搜索算法的基础上探索$\alpha-\beta$剪枝算法，优化对局面的启发式评估函数的过程。在这一过程中，我对基本的MinMax算法有了较为深刻的理解，同时在这个算法的基础上进行拓展，加入$\alpha-\beta$剪枝，对算法的性能进行了优化，并且得到了显著的提升。同时，通过对游戏规则的理解，我改进了启发式函数，使得对局面的评估更为合理，也有助于提高博弈的胜率。

致谢 感谢匡亚明学院脑科学与人工智能方向的部分同学在群里的一些讨论，给予我如何测试改进后的`heuristic()`函数的启发。

References:

[1]《人工智能：一种现代的方法》5.3 $\alpha-\beta$剪枝

[2]《人工智能：一种现代的方法》5.3 $\alpha-\beta$剪枝