黑白棋游戏实验报告

王一栋（151220113、646842131@qq.com）

(南京大学 计算机科学与技术系, 南京 210093)

摘 要: 理解并介绍了 MiniMax 搜索的实现；修改了 MiniMaxDecider 类，加入了 AlphaBeta 剪枝，并且比较引入剪枝带来的速度变化；改进了 othello.OthelloState 类中的 heuristic 函数；理解了 MTDDecider 类，介绍它与 MiniMaxDecider 类的异同。

关键词: 人工智能；MiniMax 搜索；AlphaBeta 剪枝；MTD-f 算法

中图法分类号: TP301 **文献标识码**: A

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。本次实验下载了开源的 Othello 人机对战源程序，理解并对其部分 AI 算法进行了修改。

2 实验代码及介绍

2.1 理解并介绍MiniMax搜索的实现

MiniMax搜索用一个boolean变量maximize来判断是处于最大值层还是最小值层，depth是最大搜索深度。

在decide函数里，value表示当前节点评分初值（若在最大层则为负无穷，反之正无穷），并用一个list存储最优动作。获取当前节点的可操作动作，遍历每个动作（action），用newState表示当前状态经过action之后的新状态，对新状态进行递归评分（minMaxRecursor), 在这里minMaxRecursor返回的是评分的绝对值，我们要根据maximize的值来决定newValue的正负号。若新值比旧值更优，则把value赋为新值，并清空旧的bestAction, 若新值不小于旧值，则在 bestAction里加上当前动作（若新值旧值相同，则不清空旧的bestAction，只在其中加上当前action即可），因为value相等，所以bestAction里的动作等价，最后随机选择一个动作执行即可。

heuristic函数在OthelloState中定义，若玩家1获胜，则返回“正无穷”（用5000替代），否则返回“负无穷”（用－5000替代），若无玩家获胜，则用当前状态两个玩家的占据角落数，拥有棋子数，当前状态可下步数，稳定的棋子数目来对当前状态评分。

minmax算法的真正核心是估值函数minMaxRecursor，若当前状态已被访问，则返回该状态的key，若当前状态表示游戏结束或达到最大搜索深度，则利用heuristic函数计算并返回当前状态的评分。如果不满足上述三个条件，继续递归评分，注意每次递归时都用！maximize将maximize改变。

2.2 修改 MiniMaxDecider 类，加入 AlphaBeta 剪枝，并且比较引入剪枝带来的速度变化

带 alphabeta 剪枝的 minmax 搜索函数如下：

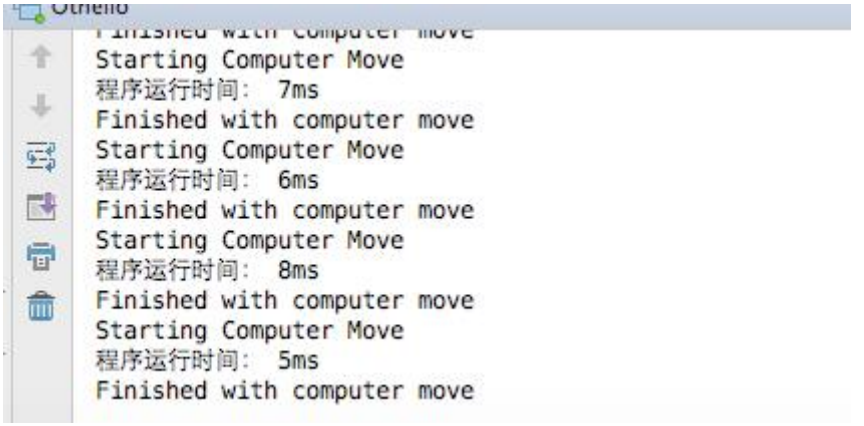
```
public float alphaBetaMinMaxRecursor(State state, int depth, boolean maximize,float alpha,float beta) {
    // Has this state already been computed?
    if (computedStates.containsKey(state))
        // Return the stored result
        return computedStates.get(state);
    // Is this state done?
    if (state.getStatus() != Status.Ongoing)
        // Store and return
        return finalize(state, state.heuristic());
    // Have we reached the end of the line?
    if (depth == this.depth)
        //Return the heuristic value
        return state.heuristic();

    // If not, recurse further. Identify the best actions to take.
    float value = maximize ? Float.NEGATIVE_INFINITY : Float.POSITIVE_INFINITY;
    int flag = maximize ? 1 : -1;
    List<Action> test = state.getActions();
    for (Action action : test) {
        // Check it. Is it better? If so, keep it.
        try {
            State childState = action.applyTo(state);
            float newValue = this.alphaBetaMinMaxRecursor(childState,  depth: depth + 1, !maximize,alpha,beta);
            //Record the best value
            if (flag * newValue > flag * value)
                value = newValue;
            if(maximize){
                if(value>alpha) alpha=value;
                if(alpha>=beta) break;
            }else{
                if(value<beta) beta=value;
                if(alpha>=beta) break;
            }
        } catch (InvalidActionException e) {
            //Should not go here
            throw new RuntimeException("Invalid action!");
        }
    }
    // Store so we don't have to compute it again.
    return finalize(state, value);
}
```

在 decide 函数中：

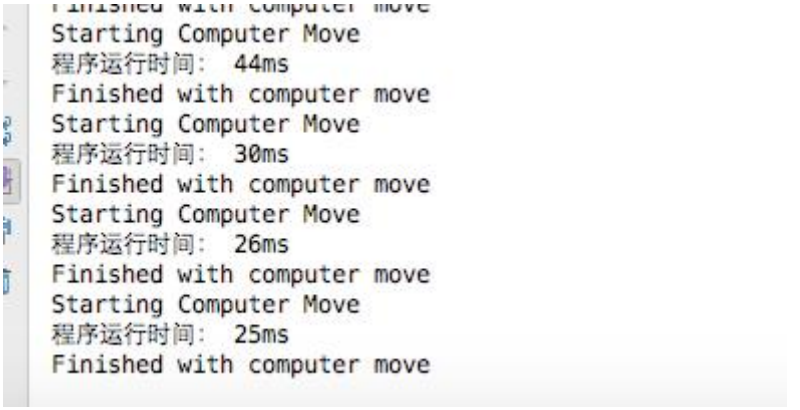
```
for (Action action : state.getActions()) {
    try {
        // Algorithm!
        State newState = action.applyTo(state);
        float newValue = this.alphaBetaMinMaxRecursor(newState,  depth: 1, !this.maximize,alpha,beta);
        // Better candidates?
        if (flag * newValue > flag * value) {
            value = newValue;
            bestActions.clear();
        }
        // Add it to the list of candidates?
        if (flag * newValue >= flag * value) bestActions.add(action);
        if (maximize) {
            if (value > alpha) alpha = value;
        } else {
            if (value < beta) beta = value;
        }
    } catch (InvalidActionException e) {
        throw new RuntimeException("Invalid action!");
    }
}
```

minmax 搜索速度如下图

深度为 2:



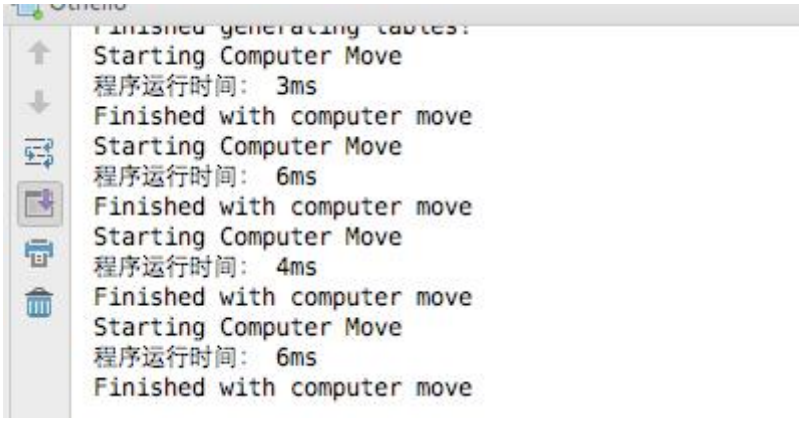
深度为 4:



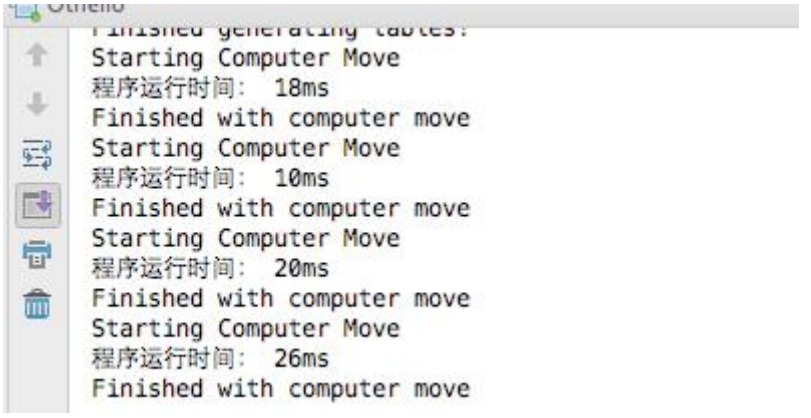
深度为 6:

带 alpha beta 剪枝的 minmax 搜索速度如下图

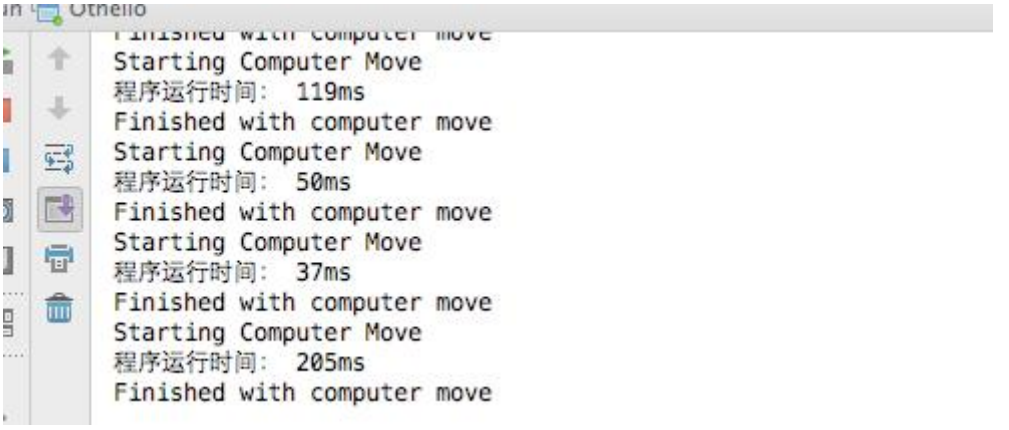
深度为 2:


Starting Computer Move
程序运行时间: 3ms
Finished with computer move
Starting Computer Move
程序运行时间: 6ms
Finished with computer move
Starting Computer Move
程序运行时间: 4ms
Finished with computer move
Starting Computer Move
程序运行时间: 6ms
Finished with computer move

深度为 4:


Starting Computer Move
程序运行时间: 18ms
Finished with computer move
Starting Computer Move
程序运行时间: 10ms
Finished with computer move
Starting Computer Move
程序运行时间: 20ms
Finished with computer move
Starting Computer Move
程序运行时间: 26ms
Finished with computer move

深度为 6:


Othello
Starting Computer Move
程序运行时间: 119ms
Finished with computer move
Starting Computer Move
程序运行时间: 50ms
Finished with computer move
Starting Computer Move
程序运行时间: 37ms
Finished with computer move
Starting Computer Move
程序运行时间: 205ms
Finished with computer move

通过以上六图可以发现当搜素深度不深时，两个算法性能几乎相同，但是当搜素深度达到6时，搜索时间相差几十倍。因此alpha-beta剪纸减少了很多不必要的搜索，节省了大量的搜索时间。

2.3 介绍 othello.OthelloState 类中的 heuristic 函数，并对其改进

heuristic函数在OthelloState中定义，若玩家1获胜，则返回“正无穷”（用5000替代），否则返回“负无穷”（用－5000替代），若无玩家获胜，则用当前状态两个玩家的占据角落数，拥有棋子数，当前状态可下步数，稳定的棋子数目来对当前状态评分。

我发现heuristic函数中中占据角落数给的权重最大，是因为当你占有四个角落中的一个时，那个角落上的棋子是不可能被反转的，这个棋子又不可能是一个孤立的棋子，因此与其相连的各个棋子也都安全了，所以这个游戏本质上是一个占角的游戏。

通过不断的玩游戏，我发现当游戏处于初期的时候，几乎不可能有稳定的棋子，所以修改了heuristic函数，让其在棋子数目过小的时候不考虑稳定子，提升了程序运行的速度。

```
@Override
public float heuristic() {
    //System.out.printf("%f %f %f %f\n",this.pieceDifferential(), this.moveDi
    Status s = this.getStatus();
    int winconstant = 0;
    switch (s) {
    case PlayerOneWon:
        winconstant = 5000;
        break;
    case PlayerTwoWon:
        winconstant = -5000;
        break;
    default:
        winconstant = 0;
        break;
    }
    if(this.pieceDifferential()>20) {
        return this.pieceDifferential() +
                8 * this.moveDifferential() +
                300 * this.cornerDifferential() +
                1 * this.stabilityDifferential() +
                winconstant;
    }
    else{
        return this.pieceDifferential() +
                8 * this.moveDifferential() +
                300 * this.cornerDifferential() +
                winconstant;
    }
}
```

2.4 介绍MTDDecider类与MiniMaxDecider类的异同

MTDDecider类和MiniMaxDecider类都是基于深度优先搜索的算法，MTDDecider类是MiniMaxDecider类的改进。

MTDDecider类在iterative_deepening中运用了迭代深化搜索。即在深度优先搜索中.通过逐渐地提高深度限制（maxdepth），这种动态提升深度到方法可以充分的利用规定的搜索时间，返回一个目前可以接受的较优的解，由于每次迭代都会记录节点的信息，所以不会大量重复计算。Iterative_deepening()函数将上次迭代获得的博弈值传给MTDF()函数作为初始窗口位置,并从该函数取得当前迭代的博弈值。AlphaBetaWithMemory()函数使用Alpha-Beta算法,并使用双置换表存储已搜索结点的结果并用alpha-beta剪枝。

AlphaBetaWithMemory()会返回该步操作的猜测值firstGuess,在MTDF函数中，不断循环判断猜测值和beta，g和beta之间的关系，并修改beta，猜测值和g，即修改窗口大小。

3 结束语

本次实验通过阅读并理解 Othello 人机对战源程序，加深了对 MinMax 算法和 Alpha-Beta 剪枝的理解。除此之外，还学习了 MTD-f 算法。

致谢 在此,我们向对本文的工作给予支持和建议的同学和老师，特别是南京大学计算机科学与技术系的俞杨教授表示感谢。

References:

[1] Stuart J. Russell, Peter Norvig. Artificial Intelligence: A Modern Approach (3rd edition), Pearson, 2011.