黑白棋游戏实验报告

周韧哲 (181220076、zhourz@smail.nju.edu.cn)

(南京大学 计算机科学与技术系，南京 210093)

摘 要: 完成了黑白棋游戏的四个任务

关键词: 极小极大算法，AlphaBeta 剪枝，启发式函数，MTD(f)算法

1 引言

介绍极小极大算法与 MTD(f)算法，在极小极大算法中添加了 AlphaBeta 剪枝，改进了启发式函数。

2 具体实现

2.1 任务一：介绍MiniMax搜索

在 MiniMaxDecider.java 中,将 min 和 max 的判断合并为一个布尔变量 maximize,maximize 为 1 则当前状态下在最大化 value,为 0 则当前状态下在最小化 value。Depth 为向下搜索的最大深度，已经计算过的 state 和其 value 则存储在 computedStates 中。

MiniMaxDecider(boolean maximize, int depth)：初始化各数据结构。

Decide(State state)：Minimax 算法的主要框架。若此时为最大化，则将 value 初始化为负无穷大；否则为正无穷大。bestActions 用来最佳 value 的动作列表，变量 flag 为 1 或-1，对应于最大化或最小化，通过 flag 完成了 min 与 max 的合并:先获得当前 state 下所有能做的动作列表,然后分别执行这些动作进入下一个 state，通过函数 miniMaxRecursor()来迭代计算下一个状态的 value 即 newvalue,若 flag * newValue > flag * value(显然若无 flag 变量则需把 min 和 max 分开判断)，说明此 state 更优，更新 value 为 newvalue,并把 bestActions 清空，将该动作加入 bestActions 中(此时 flag * newValue >= flag * value)，可以看出 bestActions 中的动作是当下 value 最高且相同的一系列动作。故当所有可执行动作做完后，bestActions 中存储了 value 最佳的动作集合，最后随机选取一个动作即可。

MiniMaxRecursor(State state, int depth, boolean maximize):递归计算 state 的 value。如果 state 已经计算过则直接返回该 state 的 value;如果该 state 下游戏结束，返回 state 的 value(finalize(state, state.heuristic()))；如果搜索深度达到了最大深度，则返回启发式函数给出的 value 的估计；若都不是，则进一步向下搜索。Test 为当前 state 下可执行的动作列表，依次执行，在每一个动作执行中，调用 miniMaxRecursor()递归求得 childState 的 value,最后返回最佳 value 即可。

Finalize(State state, float value)：返回 value。

2.2 任务二：在MiniMax中加入AlphaBeta剪枝

参照讲义上的伪代码，将 MiniMaxRecursor(State state, int depth, boolean maximize)中这部分改成如下所示：

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

        if (maximize) {
            if (value>=beta) return value;
            alpha = Math.max(alpha,value);
        }
        else {
            if (value<=alpha) return value;
            beta = Math.min(beta,value);
        }

    } 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);
```

并在 Decide(State state)中调用 miniMaxRecursor()时将 alpha,beta 分别初始化为负无穷大与正无穷大：

```
float newValue = this.miniMaxRecursor(newState, depth: 1, !this.maximize,Float.NEGATIVE_INFINITY, Float.POSITIVE_INFINITY);
```

然后在 Othello.java 的 computerMove()中调用 System.currentTimeMillis()来打印时间差从而比较两者速度差别：当最大搜索深度较小时，alphabeta 剪枝带来的改进效果几乎没有，逐渐加深最大搜索深度，当深度超过 4 时，明显可以感受到 alphabeta 剪枝带来的整体的速度提升。(以下为不同深度下剪枝与未剪枝的速度对比)

Depth	2	3	4	5	6	7	8	9
Pruning	0~4ms	1~15ms	3~30ms	30~150ms	30~200ms	100~1500ms	200~5500ms	>400ms, sometimes more than 80000ms
Not pruning	0~4ms	1~15ms	3~35ms	50~1500ms	200~5000ms	300~20000ms	>1000ms, sometimes more than 150000ms	>8000ms, sometimes more than 200000ms

2.3 任务三：改进启发式函数

在 heuristic()中，winconstant 是得分，初始化为 0。若当前状态下，playerone 赢，则置为 5000；若 playertwo 赢，则置为-5000，最后返回的是 winconstant 与各种 Differential 的加权和。其中，pieceDifferential()为两玩家总棋子数之差，系数为 1；moveDifferential()为两玩家可移动的棋子数之差，系数为 8；cornerDifferential()为两玩家在四个顶点处的棋子数之差，系数为 300；stabilityDifferential()为水平(hBoard)、竖直(vBoard)、两对角线(dBoard1,dBoard2)两玩家可翻转的棋子数之差，系数为 1。显然，总棋子数对游戏结果会有影响，可移动的棋子数也对游戏结果会有影响，顶点处棋子由于不会被翻转，也对游戏结果有影响，可翻转的棋子数也对游戏结果有一定影响。

故我的改进思路是：加上边界处棋子的影响，边界处棋子越多，则越不容易被翻转，对最后游戏结果的影响也较大。加入以下代码，从而获得边界处两玩家棋子数之差：

```
private float borderDifferential(){
    float differential=0;
    for(int i = 0; i < dimension; i++) {
        for (int j = 0; j < dimension; j++) {
            short border = getSpotOnLine(hBoard[i], (byte) j);
            if (border != 0){
                differential += border == 2 ? 1 : -1;
            }
        }
    }
    return differential;
}
```

在 heuristic()中最后返回值加上 borderDifferential(),系数为 200。

2.4 任务四：介绍MTDDecider类

MTDDecider 类使用 MTD-F 算法来进行搜索。MTD-F 算法的思路是：首先给定一个猜测值 g,一个上界 upperbound 和一个下界 lowerbound，最优解包括在上界和下界中。在上下边界构成的范围不断缩小的过程中，多次调用 AlphaBeta 搜索,每次调用时都使用极小的窗口,返回一个最小值的边界，更新上边界或下边界,当下边界的值大于等于上边界时,搜索完成[1] [2] 。

SearchNode 为搜索节点，存储了 EntryType，值 value，与深度 depth。变量 USE_MTDF 来指示是否使用 MTDF()。transpositionTable 为置换表，已经搜索过的节点保存在置换表中。在搜索过程中，很多结点虽然是经过不同的路径到达的,但其 state 是一样的,置换表保存了已搜索结点 state 与其 searchNode。若发现一样的 state，则直接从置换表中获得 state 的 searchNode 即可，这样可以减少对节点的重复搜索。

iterative_deepening(State root)在限制深度和时间内搜索，USE-MTDF 指示使用 MTDF 搜索或 AlphaBeta 剪枝搜索，返回最优动作。由 MTDF 框架代码：

```
g = firstGuess
upperbound = WIN
lowerbound = LOSE
while (lowerbound < upperbound)
    if (g == lowerbound)      beta = g + 1
    else      beta = g
    g = -AlphaBetaWithMemory(root, beta - 1, beta, depth, -flag)
    if (g < beta)      upperbound = g
    else      lowerbound = g
return g
```

了解到 MTDF(State root, int firstGuess, int depth)初始化 lowerbound 与 upperbound 为 LOSE 与 WIN，在下界小于上界的限制内，通过 AlphaBetaWithMemory 迭代搜索当前 state 的 value，每次调用 AlphaBetaWithMemory 都使用极小的窗口,返回一个最小值的边界，并根据边界 beta 更新 lowerbound 与 upperbound 令区间缩小，逐渐使其收敛，搜索完成最终返回当前 state 的 value。

AlphaBetaWithMemory(State state, int alpha, int beta,int depth, int color)在深度与时间限制内使用 AlphaBeta 剪枝向下搜索，同时使用置换表 transpositionTable 来减小重复搜索的开销。因为节点的 type 有三个类型：EXACT_VALUE, LOWERBOUND, UPPERBOUND;若当前 node 深度>=depth 且不为空，且其 type 是 EXACT_VALUE，即节点分值已知，且 Alpha <=结点分值<=Beta，为准确值，直接返回；否则，若 depth 为 0，且该状态下游戏未结束，则其为新 node，赋值后加入 transpositionTable 中，并返回其 value；否则，以深度 4 作为分界线，若深度大于 4，则用 depth 与 depth-2 进行搜索；否则，只用 depth 进行搜索：获取可执行的动作，循环对 state 执行动作，并调用 AlphaBetaWithMemory()递归求得孩子节点的 value，将孩子节点加入 transpositionTable 中并返回 bestValue，其中 bestValue 为孩子节点中 value 的最大值，且若 bestValue 大于 alpha，则更新 alpha。

MTD(f)算法与 MiniMax 算法的比较:

共同点：两者都运用了 AlphaBeta 剪枝，在达到限制深度时使用启发式函数给出当前局面评分。

不同点：MTD(f)算法是 MiniMax 算法的改进版本，加入了置换表来减小重复搜索带来的开销，利用空窗的 AlphaBeta 搜索提高了剪枝率，极大增强了搜索的效率。

3 结束语

本文详细介绍了 Assignment2 中对任务的实现，介绍了极小极大算法与 MTD(f)算法，在极小极大算法中添加了 AlphaBeta 剪枝，改进了启发式函数。

References:

[1] Zobrist, Albert L.：A New Hashing Method with Application for Game Playing. ICGA Journal, vol. 13, no. 2, pp. 69-73, 1990

附中文参考文献:

[2] 邹竞. 基于 MTD(f)的中国象棋人机博弈 算法的设计与优化.计算机与数字工程.2008,Vol. 36 No. 938