实验 2：黑白棋游戏

王琛然（151220104、17721502736@163.com）

(南京大学 计算机科学与技术系, 南京 210093)

摘 要: 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。本文主要对黑白棋的几种算法：MiniMaxDecider 实现的 MiniMax 搜索、加入 AlphaBeta 剪枝的 MiniMax 搜索、改进启发式函数的 MiniMax 搜索和 MTDDecider 算法进行理解与介绍，并对几种算法进行效率分析。

关键词: MiniMax 算法、AlphaBeta 算法、MTDDecider 算法

中图法分类号: TP301 **文献标识码**: A

1 实验要求

1. 阅读源代码 MiniMaxDecider 类，理解并介绍 MiniMax 搜索算法
2. 加入 AlphaBeta 剪枝，与原始 MiniMax 搜索算法比较
3. 改进启发式函数 heuristic 函数
4. 理解 MTDDecider 类，与 MiniMaxDecider 类比较

2 MiniMax

2.1 原始MiniMaxDecider类理解

MiniMaxDecider 类中主要有两个函数：decide 和 miniMaxRecursor。Decide 用来决策，判断下一步如何走；miniMaxRecursor 函数在 decide 中被调用，递归并利用启发式函数向下直到叶子节点，然后回溯并与父节点的值进行比较，其本质和 decide 函数相同，都是对下一步进行决策。

2.1.1 decide 函数

在 decide 函数中，不同于课本中分别实现的 Min 和 Max 函数，而是利用 boolean 类型的 maximize 作为开关控制状态：当 maximize = 0 时，表示对手先下，此时需要使对手移动到有极小值的状态，即为 Min 函数；当 maximize = 1 时，表示电脑先下，此时需要时自己移动到有极大值的状态，即为 Max 函数。

首先得到当前状态可以进行下一步的所有动作并遍历，新建状态 newState 并将所进行的动作附加于新状态，对当前状态调用 miniMaxRecursor 函数，限定深度为 1，并选择与当前状态相反的状况（即若当前状态寻找极大值，则下一层递归寻找极小值；反之亦然，以此类推），返回递归计算的结果

与当前状态的 value 进行比较，若选择优于当前选择，则将 newValue 赋予 value，清空 bestActions 数组并添加当前所选动作；若当前选择更优，则直接添加当前动作到 bestActions 数组。

跳出循环后，若 bestAction 数组中有不止一个行为，则从中任选一个并返回该动作

```java
public Action decide(State state) {
    // Choose randomly between equally good options
    float value = maximize ? Float.NEGATIVE_INFINITY : Float.POSITIVE_INFINITY;
    float alpha = Float.NEGATIVE_INFINITY;
    float beta = Float.POSITIVE_INFINITY;
    List<Action> bestActions = new ArrayList<Action>();
    // Iterate!
    int flag = maximize ? 1 : -1;

    long startTime = System.currentTimeMillis();
    for (Action action : state.getActions()) {
        try {
            // Algorithm!
            State newState = action.applyTo(state);
            float newValue = this.miniMaxRecursor(newState, alpha: 1, !this.maximize);
            // 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);
        } catch (InvalidActionException e) {
            throw new RuntimeException("Invalid action!");
        }
    }
    // If there are more than one best actions, pick one of the best randomly
    Collections.shuffle(bestActions);
```

2.1.2　miniMaxRecursor 函数

miniMaxRecursor 函数本质同 decide 函数，均为决策下一步行为。

首先对当前状态进行判断，若当前状态已经存在，则直接返回当前状态（状态存在 computedStets 中）；若当前状态是结束状态，则返回当前状态的值；若深度达到上限，则调用启发式函数选择最优值。

```java
// 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();
```

当状态不属于上述任何一类，则递归向下找到最优动作：对当前状态所有可行动作集合遍历，新建状态 childState 并将当前动作赋予新状态，向下一层进行递归直至受限深度，并由上述状态可知，当到达受限深度时，需要调用启发式函数找到最优局面，然后再回溯到父节点并与父节点的值进行比较，选择更优的值。

```java
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);
        //Record the best value
        if (flag * newValue > flag * value) {
            value = newValue;
        }
    } catch (InvalidActionException e) {
                        //Should not go here
        throw new RuntimeException("Invalid action!");
    }
}
```

跳出循环后，直接返回当前的值，回到上一层递归。

2.2 AlphaBeta剪枝算法

2.2.1 算法介绍与设计

根据 AlphaBeta 剪枝算法的定义，若当前节点没有优于父节点的值，则可以不进行后续的搜索，直接剪枝。在 miniMaxRecursor 函数的调用中添加 alpha 和 beta 两个参数来进行搜索剪枝运算。alpha 表示 Max 节点子节点搜索的最大值，beta 表示 Min 节点子节点搜索的最小值。

```
float newValue = this.miniMaxRecursor(newState, alpha, beta,  depth: 1, !this.maximize);
```

若当前状态为 Max，则需要在子节点搜索最大值，所以当子函数的返回值大于 alpha 的值时，更新 alpha 的值，此时，beta 值是父节点 Min 所限定的最小值，所以若 alpha>beta 则表示该节点的值都大于父节点期望的最小值，无论如何都不会被父节点选中，所以可以直接剪掉该节点的所有分支；若当前状态为 Min，则需要在子节点搜索最小值，所以当子函数的返回值小于 beta 的值时，更新 beta 的值，此时 alpha 是父节点 Max 所限定的最大值，所以若 alpha>beta，则表示该节点的值都小于父节点所期望更大的值，所以父节点也不会选择该节点，可以直接剪枝。

```
if(USE_ALPHA) {
    if (flag == 1 && value > alpha)
        alpha = value;
    else if (flag == -1 && value < beta)
        beta = value;
    if (alpha > beta)
        break;
}
```

2.2.2 运行效果比较

用 System.currentTimeMillis()函数来获得当前时间，在开始搜索前获得开始时间，搜索结束后获得结束时间，二者相减获得运行时间。

当深度为 6 时：

不使用 AlphaBeta 剪枝：

```
Starting Computer Move
Total time: 410
Finished with computer move
Starting Computer Move
Total time: 602
Finished with computer move
Starting Computer Move
Total time: 285
Finished with computer move
Starting Computer Move
Total time: 291
Finished with computer move
Starting Computer Move
Total time: 453
Finished with computer move
Starting Computer Move
Total time: 550
Finished with computer move
Starting Computer Move
Total time: 334
Finished with computer move
Starting Computer Move
Total time: 735
Finished with computer move
Starting Computer Move
Total time: 534
```

使用 AlphaBeta 剪枝：

```
Finished generating tables!
Starting Computer Move
Total time: 115
Finished with computer move
Starting Computer Move
Total time: 65
Finished with computer move
Starting Computer Move
Total time: 63
Finished with computer move
Starting Computer Move
Total time: 73
Finished with computer move
Starting Computer Move
Total time: 119
Finished with computer move
Starting Computer Move
Total time: 47
Finished with computer move
Starting Computer Move
Total time: 51
Finished with computer move
Starting Computer Move
Total time: 71
Finished with computer move
Starting Computer Move
Total time: 73
```

深度为 4 时：

不使用 AlphaBeta 剪枝：

```
Starting Computer Move
Total time: 11
Finished with computer move
Starting Computer Move
Total time: 81
Finished with computer move
Starting Computer Move
Total time: 22
Finished with computer move
Starting Computer Move
Total time: 32
Finished with computer move
Starting Computer Move
Total time: 18
Finished with computer move
Starting Computer Move
Total time: 15
Finished with computer move
Starting Computer Move
Total time: 57
Finished with computer move
```

使用 AlphaBeta 剪枝：

```
Starting Computer Move
Total time: 46
Finished with computer move
Starting Computer Move
Total time: 40
Finished with computer move
Starting Computer Move
Total time: 15
Finished with computer move
Starting Computer Move
Total time: 26
Finished with computer move
Starting Computer Move
Total time: 12
Finished with computer move
Starting Computer Move
Total time: 6
Finished with computer move
Starting Computer Move
Total time: 7
Finished with computer move
```

深度为 2 时：

不使用 AlphaBeta 剪枝：

```
Starting Computer Move
Total time: 5
Finished with computer move
Starting Computer Move
Total time: 4
Finished with computer move
Starting Computer Move
Total time: 3
Finished with computer move
Starting Computer Move
Total time: 3
Finished with computer move
Starting Computer Move
Total time: 3
Finished with computer move
Starting Computer Move
Total time: 1
Finished with computer move
Starting Computer Move
Total time: 1
Finished with computer move
```

使用 AlphaBeta 剪枝：

```
Starting Computer Move
Total time: 3
Finished with computer move
Starting Computer Move
Total time: 4
Finished with computer move
Starting Computer Move
Total time: 2
Finished with computer move
Starting Computer Move
Total time: 2
Finished with computer move
Starting Computer Move
Total time: 2
Finished with computer move
Starting Computer Move
Total time: 1
Finished with computer move
Starting Computer Move
Total time: 1
Finished with computer move
```

可以看出当搜索深度较深的时候，AlphaBeta 剪枝算法优势比较明显，运算时间大幅度减少，但在小深度的情况下效果不明显，因为在深度较小的时候，递归的次数减少，所以剪枝的效果不明显。

2.3 Heuristic函数

2.3.1 函数介绍

在搜索到达限定深度时，需要用启发式函数来选择当前最优局面，在启发式函数中，首先定义一号玩家和二号玩家的初始值，由于一号玩家为先手，所以认为其为 Max 方，设定定值 winconstant = 5000，能够返回一个正值；二号玩家为后手，认为其为 Min 方，设定定值 winconstant = -5000，返回负值；缺省为 0。

```
switch (s) {
case PlayerOneWon:
    winconstant = 5000;
    break;
case PlayerTwoWon:
    winconstant = -5000;
    break;
default:
    winconstant = 0;
    break;
}
```

启发式函数分为四个部分：pieceDifferential 是棋盘上双方棋子之差；moveDifferential 是棋盘上双方可行棋步（可以下棋的位置）之差；cornerDifferential 是双方占据四个角落的数量之差（player1 – player2）；stabilityDifferential 是双方稳定不会改变的棋子数量之差；差值前面的系数是该差值的权重，由于在行棋策略中，占据角落最为重要，当一方占据多个角落时，另一方输棋的概率较大，所以占据的角落数量之差权重最大，设为 300，当该方为 Max 方，当对方占据角落数量多于本方，角落的数量之差为负数，函数返回值较小，则在决策时不会选择该局面；当本方占据角落数量对于对方，角落的数量之差为正数，函数返回值大，则在决策时更倾向于选择该局面；反之亦然；其次为可行棋步之差，同理有 cornerDifferential。最后再加上定值 winconstant，总之在本方为 Max 方时，使 heuristic 函数返回值尽可能的大；当本方为 Min 方时，使 heuristic 函数返回值尽可能的小。

```
return this.pieceDifferential() +
    8 * this.moveDifferential() +
  300 * this.cornerDifferential() +
    1 * this.stabilityDifferential() +
    winconstant;
```

2.3.2 函数改进

根据行棋策略可知，在边上的棋子也比较重要，所以可以考虑在边上不是角落的棋子数量，但由于其重要性低于角落的棋子，所以其权重应小于角落的棋子权重，可以设为 10：

```
private float sideDifferential() {
    float diff = 0;
    for(int i = 0; i<8; i++) {
        short[] sides = new short[8];
        for (int j = 0; j < dimension; j++) {
            sides[j] = getSpotOnLine(hBoard[i], (byte) j);
        }
        for (short side : sides) if (side != 0) diff += side == 2 ? 1 : -1;
    }
    return diff;
}
```

除此之外，星位（与角斜向相邻的地方）是很危险的局面，不能够去占据的，权重可以设为 15，并且由于不能占据，所以和上面角落和边的 diff 变化状态相反：

```
private float X_squareDifferential() {
    float diff = 0;
    short[] Xs = new short[4];
    Xs[0] = getSpotOnLine(hBoard[1], (byte)1);
    Xs[1] = getSpotOnLine(hBoard[1], (byte)(dimension - 2));
    Xs[2] = getSpotOnLine(hBoard[dimension - 2], (byte)1);
    Xs[3] = getSpotOnLine(hBoard[dimension - 2], (byte)(dimension - 2));
    for (short X : Xs) if (X != 0) diff += X == 2 ? -1 : 1;
    return diff;
}
```

最终，修改过的 heuristic 函数为：

```
return this.pieceDifferential() +
    8 * this.moveDifferential() +
  300 * this.cornerDifferential() +
        10 * this.sideDifferential() +
        15 * this.X_squareDifferential() +
    1 * this.stabilityDifferential() +
    winconstant;
```

3 MTD 算法

3.1 算法介绍

算法采用循环迭代的方式，采用空窗口进行搜索，搜索开始时，上下界范围较大，多次调用 MTDF 算法或 AlphaBeta 剪枝算法来完成搜索，调用结果返回真实的极小极大值的上下界，然后修改 MTD(f)中的上下界，随着搜索过程的不断进行，上下界范围不断缩小，向真实值逼近。当下边界的值大于或等于上边界时，搜索完成。同时，与置换表结合使用，重复利用搜索中已经生成过的节点，减少不必要的计算，提高效率。除此之外，MTD 算法中需要对真实值进行初始的猜测估计，初始预测值 f irstguess 的好坏会影响上下界逼近的速度。

3.1.1 decide 函数：（iterative_deepening）

首先在受限深度内迭代循环，并每一重循环中，都初始化 alpha 和 beta 进行空搜索。根据 USE_MTDF 的值判断是否进行 MTD 算法，若不进行 MTD 算法调用，则直接调用 AlphaBeta 剪枝算法，并根据调用函数的返回值修改当前动作的 value。由于下一层的搜索是对手搜索，所以 alpha 和 beta 要做对应更改。

```
for (d = 1; d < maxdepth; d++) {
    int alpha = LOSE; int beta = WIN; int actionsExplored = 0;
    for (ActionValuePair a : actions) {
        State n;
        try {
            n = a.action.applyTo(root);

            int value;
            if (USE_MTDF)
                value = MTDF(n, (int) a.value, d);
            else {
                int flag = maximizer ? 1 : -1;
                value = -AlphaBetaWithMemory(n, -beta , -alpha, depth: d - 1, -flag);
            }
            actionsExplored++;
            // Store the computed value for move ordering
            a.value = value;
```

3.1.2 MTD 算法

若进行 MTD 算法，根据 iterative_deepening 函数可知，初始预测值即为动作的初始 value，初始化上下界。进入 while 循环，循环跳出条件为：下界大于或等于上界，即已将上下界逼近于真实值。根据下界改变 beta 的值，向下一层调用 AlphaBeta 剪枝算法，返回值为新的边界，根据返回值对上下界范围进行修改，

```java
private int MTDF(State root, int firstGuess, int depth)
        throws OutOfTimeException {
    int g = firstGuess;
    int beta;
    int upperbound = WIN;
    int lowerbound = LOSE;

    int flag = maximizer ? 1 : -1;

    while (lowerbound < upperbound) {
        if (g == lowerbound) {
            beta = g + 1;
        } else {
            beta = g;
        }
        // Traditional NegaMax call, just with different bounds
        g = -AlphaBetaWithMemory(root, alpha: beta - 1, beta, depth, -flag);
        if (g < beta) {
            upperbound = g;
        } else {
            lowerbound = g;
        }
    }

    return g;
}
```

3.1.3 AlphaBeta 剪枝算法

在进行算法主体之前，对当前状态进行判断。在置换表中查找是否已存在当前状态，若存在则直接取出，减少计算；若当前已到达受限深度或搜素已经结束，则调用启发式函数选择最优值并返回保存。

为了减少搜索时间，当搜索深度大于 4 的时候，则分为两部分：浅深度搜素和长深度搜索

```java
int[] depthsToSearch;
if (depth > 4) {
    depthsToSearch = new int[2];
    depthsToSearch[0] = depth - 2; // TODO: this should be easily adjustable
    depthsToSearch[1] = depth;
} else {
    depthsToSearch = new int[1];
    depthsToSearch[0] = depth;
}
```

对当前的状态遍历，新建状态并将动作和动作的值都赋予新状态，向下一层进行 AlphaBeta 剪枝运算，更改 MinMax 状态，并用递归运算的返回值更新动作的值。若新值大于最优值，则替换最优值；若最优值大于 alpha 值，则更新 alpha 的值。若最优值已经大于 beta 的值，则表示该分支都不会存在比当前更优的值，即可剪枝。跳出循环后，保存当前状态并返回。

```java
for (int i = 0; i < depthsToSearch.length; i++) {
    for (ActionValuePair a : actions) {
        int newValue;
        try {
            State childState = a.action.applyTo(state);
            // Traditional NegaMax call
            newValue = -AlphaBetaWithMemory(childState, -beta, -alpha,
                    depth: depthsToSearch[i] - 1, -color);
            // Store the value in the ActionValuePair for action ordering
            a.value = newValue;
        } catch (InvalidActionException e) {
            throw new RuntimeException("Invalid action!");
        }
        if (newValue > bestValue)
            bestValue = newValue;
        if (bestValue > alpha)
            alpha = bestValue;
        if (bestValue >= beta)
            break;
    }
    // Sort the actions to order moves on the deeper search
    Collections.sort(actions, Collections.reverseOrder());

}
return saveAndReturnState(state, alpha, beta, depth, bestValue, color);
```

4 Reference

致谢 在此,向对人工智能课程老师和助教表示感谢.

[1] 一看就懂的 Alpha-Beta 剪枝算法详解 https://blog.csdn.net/baixiaozhe/article/details/51872495

[2] 基于 MTD(f)算法的六子棋程序设计研究 https://wenku.baidu.com/view/a2a13b3bcc7931b764ce1507.html

[3] 极大极小树的剪枝算法 1 alpha-beta 剪枝 https://blog.csdn.net/u012501320/article/details/25075323

[4] 黑白棋天地 http://www.soongsky.com/othello/strategy/notation.php?text