Othello 实验报告

宋磊 181220049 974534426@qq.com

（南京大学 人工智能学院）

1 对 MiniMax 搜索实现的理解与介绍

1.1 变量介绍

Maximize 为控制此次为 Min 搜索或是 Max 搜索的开关变量，为 1 时，进行 Max 搜索，为 0 时进行 Min 搜索，具体在后面的 1.3 对 Min 和 Max 函数的合并中详细介绍。

Depth 对搜索的最大深度进行了限制。

ComputedStates 用 hash table 存储了已经计算过的状态的值，防止重复计算。

1.2 函数介绍

1.2.1 decide()

Decide()函数为决策函数，通过循环执行合法动作，达到新的状态，对新的状态递归调用 miniMaxRecursor()函数，返回对该状态的估值，之后如果估值优于当前最好值，则进行更新。

对于估值相同的状态，程序选择对所有最好状态对应的 action 存储在 bestActions 中，之后通过 Collections.shuffle()随机打乱 bestActions 后再进行选择，这样估值相同的状态不会受到执行动作的顺序的影响，都等概率被选择。

下图中是对动作的循环并执行，对新状态进行估值，检查更新，如果有异常出现，则抛出异常。

```
for (Action action : state.getActions()) {
    try {
        // Algorithm!
        State newState = action.applyTo(state);
        float newValue = this.miniMaxRecursor(newState, depth: 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!");
    }
}
```

下图实现了对最好状态的随机选择。

```
// If there are more than one best actions, pick one of the best randomly
Collections.shuffle(bestActions);
return bestActions.get(0);
```

1.2.2 miniMaxRecursor()

函数先对已经计算过的状态判重、对已经结束游戏的状态进行结束、对已经达到搜索最大深度时，返回对当前状态的估值（state.heuristic()函数返回对当前状态的估值）。之后像 decide()函数一样，遍历每一个 action，

得到新的状态，对新的状态进行递归搜索，检查是否需要更新 action 和 value。

1.2.3 finalize()

返回 value。

1.3 对Min和Max函数的合并

与课本上分别实现 Min 和 Max 函数不同，程序中通过一个开关即 maximize 变量将两个函数合并，根据 maximize 为 0 或为 1 对 value 和 flag 赋不同的初始值。

```
float value = maximize ? Float.NEGATIVE_INFINITY : Float.POSITIVE_INFINITY;
List<Action> bestActions = new ArrayList<~>();
// Iterate!
int flag = maximize ? 1 : -1;
```

在比较 newValue 和当前最好 value 时，两边同乘 flag，使得在 flag 为 1（Max 函数）时，若 newValue>=value 才进行更新，flag 为-1（Min 函数）时，若 newValue<=value 进行更新。

```
if (flag * newValue >= flag * value) bestActions.add(action);
```

2 加入 AlphaBeta 剪枝后的比较

在深度较小时，加入剪枝后速度没有明显变化，深度加深后，带来明显速度提升。在深度增加到 10 后，剪枝前，白棋第一步搜索用了大概一分钟时间，第二步已经搜索不出来；但加入剪枝后，第一步只用了一秒多，近 60 倍的提升，大概黑白双方可以各走 6 步。

虽然这个数据与电脑性能等因素有关，不是十分准确，但可以看出，剪枝带来了明显的速度提升，并且预测在步数逐渐增多时，由于可以选择的行动更多，剪枝对性能提升将会更大。

如图为相关代码：

```
State childState = action.applyTo(state);
float newValue = this.miniMaxRecursor(childState, alpha, beta, depth: depth + 1, !maximize);
//Record the best value
if (flag * newValue > flag * value)
    value = newValue;
if(maximize){
    if(newValue >= beta)
        return newValue;
    alpha = (alpha > value) ? alpha : value;
}
if(!maximize){
    if(newValue <= alpha)
        return newValue;
    beta = (beta < value) ? beta : value;
}
```

3 理解并改进 heuristic 函数

3.1 理解

heuristic 函数在每次搜索时达到一定深度后，对当前局面进行估计，找到一个估值最高的 action 并执行。平衡了搜索的时间成本和结果的准确性。

othello.OthelloState 中的 heuristic 函数首先对当前状态的胜负进行判断，如果胜利，则对结果加上一个很

大的正数，如果失败，则加上一个很大的负数，使得一定选择该 action 或不选择该 action。

如果没有分出胜负，则将 winconstant 设为 0，由其他项共同决定该状态的好坏。

3.2 改进策略介绍

3.2.1 关于角的策略

在黑白棋中，角上的点是十分重要的，因为他永远不会被两个对方的棋子夹住，相应的，与他相邻的己方的棋子，如果同时与边上的己方棋子相邻，则也是稳定的棋子，如图所示（图来自知乎），这 26 个黑棋一定是稳定的。

原算法中已经考虑了角上的点，并且赋予了极大的权值，但没有考虑与他相邻的也是稳定的点，例如图中，左上角的得分应该高于右下角，因为稳定的点更多，但原算法中，对该方面考虑得分相同，所以在该方面进行改进。

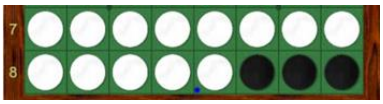
具体改良如下图，countCornerAround()函数前两个参数是角的位置，后两个是搜索的方向，返回值为角上稳定点的个数，例如上图左上角返回 16，右下角返回 10，如果是对方的器，则返回负值。

```
private float cornerAroundDifferential() {
    float diff = 0;
    diff += countCornerAround( row: 0,  col: 0,  x_dir: 1,  y_dir: 1);
    diff += countCornerAround( row: 0,  col: dimension-1,  x_dir: -1,  y_dir: 1);
    diff += countCornerAround( row: dimension-1,  col: 0,  x_dir: 1,  y_dir: -1);
    diff += countCornerAround( row: dimension-1,  col: dimension-1,  x_dir: -1,  y_dir: -1);

    return diff;
}
```

countCornerAround()函数如下，首先判断角上的棋子是黑棋还是白棋，之后对与角上相同颜色的棋子进行计数，max_last_row 为上一行的连续的该颜色棋子的最大数量，下一行连续棋子如果超过 max_last_row，则该棋子很容易被翻过，不计为稳定子（举例：如第二张图中，第 8 行白棋最大连续数为 5，第 7 行为 8，但第 7 行稳定子数量不能记为 8，因为后三个子很容易被黑棋翻掉，不计为稳定子）。

```
private int countCornerAround(int row, int col, int x_dir, int y_dir) {
    short corner = getSpotOnLine(hBoard[row], (byte)col);
    if(corner == 0)
        return 0;
    int cnt = 0, max_last_row = dimension, max_now_row = 0;
    for(int i = row; i < dimension && i >= 0 && getSpotOnLine(hBoard[i], (byte)col) == corner; i += y_dir) {
        max_now_row = 0;
        for(int j = col; Math.abs(col-j) < max_last_row && j >= 0 && getSpotOnLine(hBoard[i], (byte)j) == corner; j += x_dir) {
            cnt += 1;
            max_now_row += 1;
        }
        max_last_row = max_now_row;
    }
    return (corner == 2) ? cnt : -1*cnt;
}
```



另外，星位和 C 位（左上角 B1、A2、B2 及其他角上对应位置）是十分危险的点，因为占了这些点后，极其容易使角被对方占领。所以要避免走这些点。

关于星位和 C 位判断代码如下，corner_x 和 corner_y 为四个角的坐标，x，y 为方向数组，通过这样的方法，可以使用(corner_x[i]+x[i][j], corner[i]+y[i][j])得到坐标，而不是用 12 行 geiSpotOnline 获得星位和 C 位上的棋子，减少重复代码行数，便于调试。在角上没有棋子时，自身占的星位和 C 位越多，得分越低，所以这个函数的返回值的权值为负（见下面的权值）。

```
float diff = 0;
int[] corner_x = {0, dimension-1, 0, dimension-1};
int[] corner_y = {0, 0, dimension-1, dimension-1};
int[][] x = {{1, 0, 1},
             {-1, 0, -1},
             {1, 0, 1},
             {-1, 0, -1}};
int[][] y = {{0, 1, 1},
             {0, -1, -1},
             {0, -1, -1},
             {0, -1, -1}};

for(int i = 0; i < 4; i++) {
    int c_x = corner_x[i], c_y = corner_y[i];
    if(getSpotOnLine(hBoard[c_x], (byte)c_y) == 0) {
        for(int j = 0; j < 3; j++) {
            short piece = getSpotOnLine(hBoard[c_x + x[i][j]], (byte)(c_y + y[i][j]));
            if(piece != 0)
                diff += piece == 2 ? 1 : -1;
        }
    }
}
return diff;
```

3.2.2 关于边的策略

边上的点虽然不像角上的点一样稳定，但也是不容易被翻，所以也考虑到 heuristic 函数中，但要减少边的权值，要小于角的权值，这里实现时只是简单对边上的点计数，对与上面的方法重复计数的点，不做处理，对结果影响不大。

代码如下，对边上的点枚举计数。

```
private float edgeDifferential() {
    float diff = 0, tmp = 0;
    for(int i = 1; i < dimension-1; i++)
    {
        tmp = getSpotOnLine(hBoard[0], (byte)i);
        if (tmp != 0) diff += tmp == 2 ? 1 : -1;
        tmp = getSpotOnLine(hBoard[dimension-1], (byte)i);
        if (tmp != 0) diff += tmp == 2 ? 1 : -1;
    }
    for(int i = 1; i < dimension-1; i++)
    {
        tmp = getSpotOnLine(hBoard[i], (byte)0);
        if (tmp != 0) diff += tmp == 2 ? 1 : -1;
        tmp = getSpotOnLine(hBoard[i], (byte)(dimension-1));
        if (tmp != 0) diff += tmp == 2 ? 1 : -1;
    }
    return diff;
}
```

3.2.3 权重

最后的启发式函数如图，对角上的点赋予 300 权值；角上稳定不会被吃掉的点为 200 权值；星位和 C 位为负权值，因为占了这个位置后，很容易被对方占到角；对边赋予 50 权值，边的重要性没有角大，其他三个为原本的启发式函数。

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

3.3 实验结果

新写出的 heuristic 函数与其他同学未改良的程序进行对战时，可以比较容易取得胜利，在深度稍微加深时，也可以胜利，可以看出 heuristic 函数有一定的改进。

4 MTDDecider 类

MTD(f)算法通过使用长度为零的窗口多次调用 AlphaBetaWithMemory 来找到最优解，每次调用这个函数，都会返回 minimax 值的界限，并记录这个值、对应深度和上下界类型(通过 saveAndReturnState 函数记录)，当区间下界大于等于上界时，就找到了最优解。

4.1 代码分析

iterative_deepening()函数从根状态开始迭代加深搜索最优解，根据 USE_MTDF 的值判断使用 MTDF 还是 Negamax，不断调用并对值进行更新。

```
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;
```

MTDF()函数递归地调用不同边界的 alpha-beta 计算最优解，他的第二个参数 firstGuess 是猜测值，形参对应的实参使用之前找到最好的解，即上图中的 a.value，使猜测值距离目标值近，算法更加高效。主代码中多次调用 Negamax()函数(通过对 AlphaBetaWithMemory()取负实现)，这个函数返回一个界限，限制了可能最优解的范围，每一次调用，都会更新这个界限，使界限越来越小，最终在下界大于等于上界时，找到最优解。

AlphaBetaWithMemory()函数进行零窗口[beta - 1, beta]进行搜索，深度较深(大于 4)时，最优解可能在当前深度附近，代码会对 depth 和这个深度附近(代码中为 depth-2)的深度都进行搜索，如下图：

```
if (depth > 4) {
    depthsToSearch = new int[2];
    depthsToSearch[0] = depth - 2;
    depthsToSearch[1] = depth;
} else {
    depthsToSearch = new int[1];
    depthsToSearch[0] = depth;
}
```

之后递归调用 AlphaBetaWithMemory 找最优解，并将结果通过 saveAndReturnState()函数返回(这个函数只是简单记录当前节点是否为上下界或准确值以及对应的值和深度)并记录。

4.2 MiniMaxDecider的比较

MTD 算法时对 minimax 的改进，保留了搜索的思想，极大极小值等。

但 MTD 使用零窗口搜索，并且进行 alphabeta 剪枝，通过置换表判重提高效率。MTD 算法搜索的窗口越小，截断就越多，搜索就越高效，所以使用了零窗口的搜索；这个算法会对相同的局面进行多次搜索，所以需要对已经搜索过的状态进行记录，代码中使用哈希表(代码中为 transpositionTable)记录。