中山大学数据科学与计算机学院

计算机科学与技术专业-人工智能

本科生实验报告

（2018-2019 学年秋季学期）

课程名称：**Artificial Intelligence**

教学班级	计科 2 班	专业（方向）	计算机科学与技术
学号	16337341	姓名	朱志儒

实验题目

博弈树搜索

实验内容

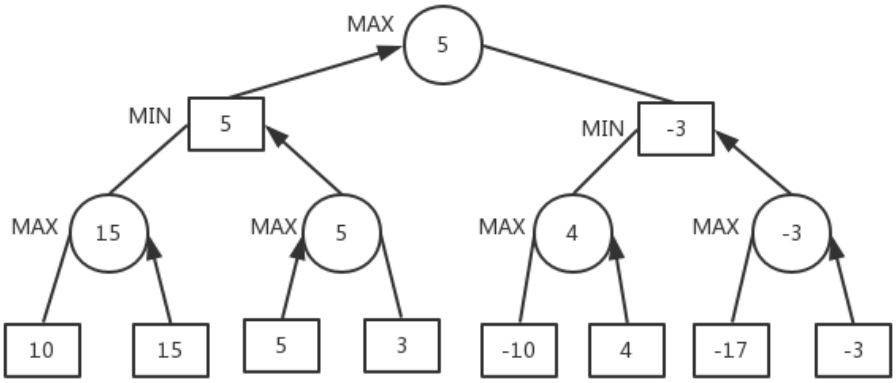
· 算法原理

1) 博弈树

对于任一种博弈竞赛，我们都可以将其构成一个博弈树，它类似于问题求解搜索中使用的搜索树和状态图。博弈树中的每个节点代表某一个棋局，每个分支代表走一步棋，根节点代表棋局最初始的状态，叶子节点表示对弈结束时的棋局。在叶子节点对应的棋局中，比赛的结果可能是赢、输或和局。从根节点开始，比赛双方轮流扩展节点，两个玩家的行动逐层交替出现，根据特定的评价函数，每个节点均有一个评价值，以表示该节点的优劣得分。

2) Minimax 算法

对博弈树进行深度优先搜索获得当前棋局之后所有可能的结果, 玩家双方均会选择对自己最有利的走法，也就是说，对于玩家一方 A 而言，A 会在可选的选项中选择最大化其优势的走法，对方则会选择使 A 优势最小化的走法。从博弈树来看，每一层轮流从子节点中选取最大值-最小值-最大值-最小值...，如下图所示。



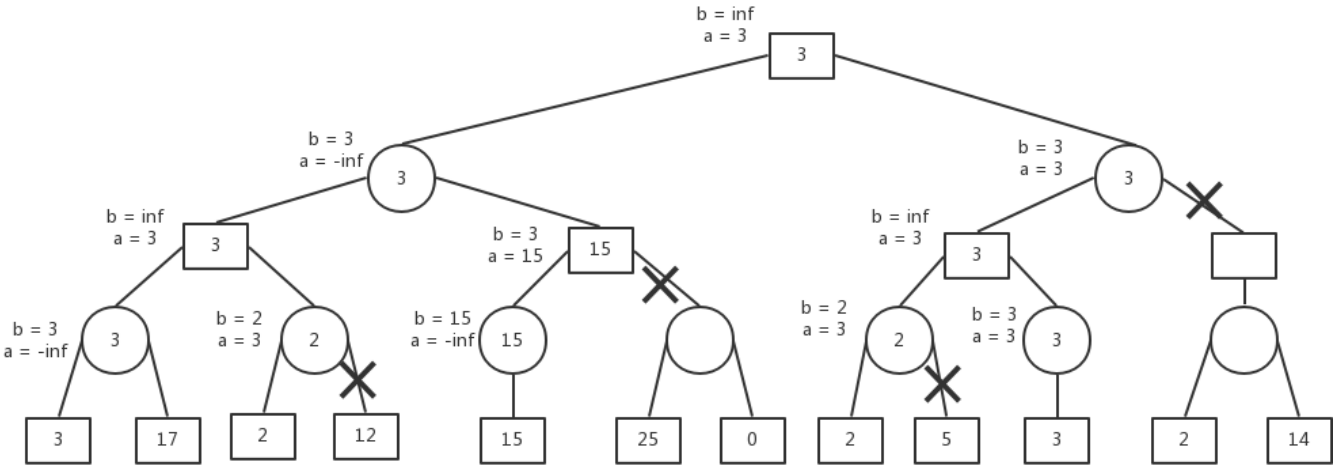
深度优先搜索的深度限制为 4 层时，图中第四层为叶子节点；第三层为 A 根据第四层的棋局估计值推出该层节点的棋局估计值，选取的是子节点中的最大值，即最大化自己的优势；第二层为对方根据第三层的棋局估计值推出该层节点的棋局估计值，选取的是子节点中的最小值，即最小化 A 的优势；第一层为 A 根据第二层棋局估计值推出该层节点的棋局估计值，选取的是子节点中的最大值，即最大化自己的优势。从而 A 就可以得到下一步最大化优势的走法。

3) Alpha-beta 剪枝

Alpha-beta 剪枝建立在 Minimax 算法的基础上，但它减少了 Minimax 算法搜索树的节点数。

对于 MIN 层的节点，如果估计出其倒推值的上确界 beta 小于或等于其 MAX 层父节点的估计倒推值的下确界 alpha，即beta ≤ alpha，则不必再扩展该 MIN 层节点的其余节点，因为其余节点的估计值对MAX层父节点的倒推值没有任何影响, 这个过程称为Alpha剪枝。

对于 MAX 层的节点，如果估计出其倒推值的下确界 alpha 大于或等于其 MIN 层父节点的估计倒推值的上确界 beta，即beta ≤ alpha，则不必再扩展该 MAX 层节点的其余节点，因为其余节点的估计值对 MIN 层父节点的倒推值没有任何影响了，这个过程称为 Beta 剪枝。



如上图所示的 Alpha-beta 剪枝过程，方框为 MAX，圆圈为 MIN。初始设置 a 为负无穷大，b 为正无穷大，深度优先搜索到达左下角的节点，第四层 MIN 层取子节点中的最小值 3，3 小于值为正无穷的 b，则 b 改为 3；回溯到第三层 MAX 层，3 大于值为负无穷的 a，则 a 改为 3，b 不变还是为正无穷；深度优先搜索右子树到第五层，返回值为 2；回溯到第四层 MIN 层节点，2 小于值为正无穷的 b，则 b 改为 2，而此时 a 为 3，就有 b＜a，所以进行剪枝，不用搜索第四层 MIN 节点的右子节点，第四层 MIN 层节点返回 2；回溯到第三层 MAX 层节点，2 小于 3，a 不变，返回值为 3；回溯到第二层 MIN 层节点，3 小于值为正无穷的 b，则 b 改为 3；深度优先搜索到第五层节点，返回值为 15；回溯到第四层 MIN 层节点，没有其他的子节点，所以返回值为 15；回溯到第三层 MAX 层节点，15 大于值为负无穷的 a，则 a 改为 15，此时 b 为 3，就有 b ＜ a，所以进行剪枝，不用搜索第三层 MAX 层节点的右子节点，第三层 MAX 层节点返回值为 15；回溯到第二层 MIN 层节点，3＜15，所以其返回值为 3；回溯到根节点，3 大于值为负无穷的 a，则 a 改为 3，再深度优先搜索根节点的右子节点，以同样的方式进行 Alpha-beta 剪枝，最后结果如图。

4) 评价函数

a) 基于位置特征的估计值

黑白棋和围棋相似，有“金边银角草肚皮”的说法，棋子在四个角的优势特别大，因为在四角的棋子无法被翻转；而在四角的邻近位的优势最小，因为这些位置容易让对方占角或是被对方翻转大量的棋子；四条边上的其他位置的优势也比较大，因为迅速占边可以比较容易地获得边界稳定子的优势;而在棋盘的中心位置优势较低。棋盘的所有位置的权重值如下:

20	-3	11	8	8	11	-3	20
-3	-7	-4	1	1	-4	-7	-3
11	-4	2	2	2	2	-4	11
8	1	2	-3	-3	2	1	8
8	1	2	-3	-3	2	1	8
11	-4	2	2	2	2	-4	11
-3	-7	-4	1	1	-4	-7	-3
20	-3	11	8	8	11	-3	20

计算我方所有棋子权重的和与对方所有棋子权重的和，再相减就可以得到基于位置特征的估计值。

b) 基于黑白子比例的估计值

在黑白棋中，如果我方棋子比对方棋子数量多，则说明我方占优，如果对方棋子比我方棋子数量多，则说明对方占优。在实际对战的过程中，这项估计值的参考价值并不是特别大，因为黑白子比例与谁将下子关系很大，比如说，在我方棋子数目远小于对方棋子数目的情况下，我方着子后可能翻转对方大量的棋子，从而会逆转局势。所以黑白子比例的估计值在总估计值中占比较低。

c) 基于边界稳定子的估计值

在黑白棋中，边界稳定子是棋盘估计值的总要指标，拥有更多的边界稳定子，既能保证我方最少的棋子数，又能辅助我方翻转大量对方的棋子，形成成片稳定子的作用。对于边界

稳定子的估计值的计算，从棋盘的一个角开始，权重分别为：

1	1	1	2	3	4	6	7

这样设计权重将鼓励占边，形成成片的边界稳定子，加大我方的优势。

边界稳定子的估计值在总估计值中占比较大。

d) 基于行动力的估计值

在黑白棋中，行动力是指棋盘上某一方的可下子位置个数，行动力较高时，可保证之后的若干步都有较好的下法。与黑白子比例的估计值相似，行动力估计值采用比例算法，但需要考虑两种特殊的情况：①如果我方没有地方下子，那么设置特低的行动力估计值；②如果对方没有地方下子，那么设置特高的行动力估计值。这样就可以避免我方处于无子可下的糟糕局面，而倾向于选择使得对方无子可下的优势局面。行动力估计值在总估计值中占比交较高。

整个棋局估计值

在计算整个棋局估计值时需要考虑上述 4 种不同的估计值，它们的占比如下：

位置特征	黑白子比例	边界稳定子	行动力
0.02	0.2	6	1

· 伪代码

1) Minimax 算法

```
1.  function minimax(node, depth, maxplayer)
2.      if depth = 0 or node.child = None
3.          return node 的棋局估计值
4.      if maxplayer
5.          bestvalue := INT_MIN
6.          for child in node.child
7.              v := minimax(child, depth - 1, False)
8.              bestvalue := max(bestvalue, v)
```

```
        return bestvalue
    else
        bestvalue := INT_MAX
        for child in node.child
            v := minimax(child, depth - 1, True)
            bestvalue := min(bestvalue, v)
        return bestvalue
```

2) Alpha-beta 剪枝

```
function alphabetapruning(node, depth, alpha, beta, maxplayer)
    if depth = 0 or node.child = None
        return node 的棋局估计值
    if maxplayer
        v := INT_MIN
        for child in node.child
            v:=max(v, alphabetapruning(child, depth-1, alpha, beta, False))
            alpha := max(alpha, v)
            if beta <= alpha
                break
    else
        v := INT_MAX
        for child in node.child
            v:=min(v, alphabetapruning(child, depth - 1, alpha, beta, True))
            beta := min(v, beta)
            if beta <= alpha
                break
    return v
```

· 关键代码

1) Alpha-beta 剪枝

对于每种不同的棋局，AI 均可能有多个落子点，在不同地方落子棋局的估计值也将不同，而棋局的估计值需要通过使用 Minimax 算法搜索博弈树获得，为减少访问的节点而使用 Alpha-beta 剪枝。

```
double alphabetapruning(Node &root, char ai, char player, int mode, int dept
h, double alpha, double beta) {
```

```
2.      //mode=0 时表示 MAX 层节点，mode=1 时表示 MIN 层节点
3.      char color = mode == 1 ? ai : player, opp = color == '@' ? 'O' : '@';
4.      auto avaiplaces = show_places(root.board, color);   //得到可下子的位置
5.      double v;
6.      if (depth == limit) {
7.          for (int i = 0; i < avaiplaces.size(); ++i) {
8.              Node newnode = Node(root.board, mode);      //新建子节点
9.              newnode.action = avaiplaces[i];           //记录该节点下子的位置
10.             move(newnode.board, avaiplaces[i], color);  //下子后棋盘发生变化
11.             int oppmode = mode == 1 ? 0 : 1;            //进入下一种 mode
12.             auto places =show_places(newnode.board,opp);//得到对方可下子的位置
13.             if (places.size() != 0)
14.                 //如果对方还有可下子的位置，则递归搜索
15.                 newnode.score = alphabetapruning(newnode, ai, player, oppmod
    e, depth - 1, alpha, beta);
16.             else
17.                 //如果对方没有地方下子，则评估当前棋局
18.                 newnode.score = evaluate(newnode.board, color);
19.             root.children.push_back(newnode);           //加入新的子节点
20.         }
21.         int index;
22.         double max = -100000.0;
23.         //得到估价值最高的走法
24.         for (int i = 0; i < root.children.size(); ++i)
25.             if (root.children[i].score > max) {
26.                 index = i;
27.                 max = root.children[i].score;
28.             }
29.         //返回估价值最高的走法
30.         return index;
31.     }
32.     if (mode == 0) {
33.         //MAX 层节点
34.         v = -100000.0;
35.         for (int i = 0; i < avaiplaces.size(); ++i) {
36.             Node newnode = Node(root.board, mode);      //新建子节点
37.             newnode.action = avaiplaces[i];             //记录该节点下子的位置
38.             move(newnode.board, avaiplaces[i], color);  //下子后棋盘发生变化
39.             int oppmode = mode == 1 ? 0 : 1;            //进入下一种 mode
40.             auto places=show_places(newnode.board, opp);//得到对方可下子的位置
41.             if (depth != 1 && places.size() != 0) {
42.                 //未到达深度限制且对方有地方下子，则递归搜索并更新 v 和 alpha 值
43.                 v = max(v, alphabetapruning(newnode, ai, player, oppmode, de
    pth - 1, alpha, beta));
```

```
                alpha = max(alpha, v);
                if (beta <= alpha)   //alpha 剪枝
                    break;
            }
            else {
                //到达深度限制或对方无子可下，则评估当前棋局并更新 v 值
                newnode.score = evaluate(newnode.board, ai);
                v = max(v, newnode.score);
            }
        }
    }
    else {
        //MIN 层节点
        v = 100000.0;
        for (int i = 0; i < avaiplaces.size(); ++i) {
            Node newnode = Node(root.board, mode);      //新建子节点
            newnode.action = avaiplaces[i];;            //记录该节点下子的位置
            move(newnode.board, avaiplaces[i], color);  //下子后棋盘发生变化
            int oppmode = mode == 1 ? 0 : 1;            //进入下一种 mode
            auto places=show_places(newnode.board, opp);//得到对方可下子的位置
            if (depth != 1 && places.size() != 0) {
                //未到达深度限制且对方有地方下子，则递归搜索并更新 v 和 beta 值
                v = min(v, alphabetapruning(newnode, ai, player, oppmode, de
pth - 1, alpha, beta));
                beta = min(beta, v);
                if (beta <= alpha)   //beta 剪枝
                    break;
            }
            else {
                //到达深度限制或对方无子可下，则评估当前棋局并更新 v 值
                newnode.score = evaluate(newnode.board, ai);
                v = min(v, newnode.score);
            }
        }
    }
    //回溯时清空占用的内存
    root.children.clear();
    return v;
}
```

2) 估价函数

分别计算基于位置特征的估计值、基于黑白子比例的估计值、基于边界稳定子的估计值、

基于行动力的估计值，然后将这些估计值加权求和得到整个棋局的估计值。

```
double evaluate(char board[8][8], char color) {
    int sideVal[9] = { 1, 1, 1, 2, 3, 4, 6, 7 };
    int mystonecount = 0, opstonecount = 0;
    double score = 0, rateeval = 0, moveeval = 0, sidestableeval = 0, corner
eval = 0;
    char opp = color == '@' ? 'O' : '@';
    //计算位置特征估计值
    for (int i = 0; i < 8; ++i)
        for (int j = 0; j < 8; ++j)
            if (board[i][j] == color) {
                score += square_weights[i][j];
                mystonecount++;
            }
            else if (board[i][j] == opp) {
                score -= square_weights[i][j];
                opstonecount++;
            }
    //计算黑白子比例估计值
    if (mystonecount > opstonecount)
        rateeval = 100.0 * mystonecount / (mystonecount + opstonecount);
    else if (mystonecount < opstonecount)
        rateeval = -100.0 * opstonecount / (mystonecount + opstonecount);
    else
        rateeval = 0;
    //计算行动力估计值
    int mymove = show_places(board, color).size();
    int opmove = show_places(board, opp).size();
    //如果我方没有地方下子，那么设定特低的行动力估计值
    if (mymove == 0)
        moveeval = -450;
    //如果对方没有地方下子，那么设定特高的行动力估计值
    else if (opmove == 0)
        moveeval = 150;
    else if (mymove > opmove)
        moveeval = (100.0 * mymove) / (mymove + opmove);
    else if (mymove < opmove)
        moveeval = -(100.0 * opmove) / (mymove + opmove);
    else
        moveeval = 0;
    //计算边界稳定点估计值
    int myside = 0, opside = 0, myconer = 0, opconer = 0;
    int corner_pos[4][2] = { {0, 0}, {0, 7}, {7, 0}, {7, 7} };
```

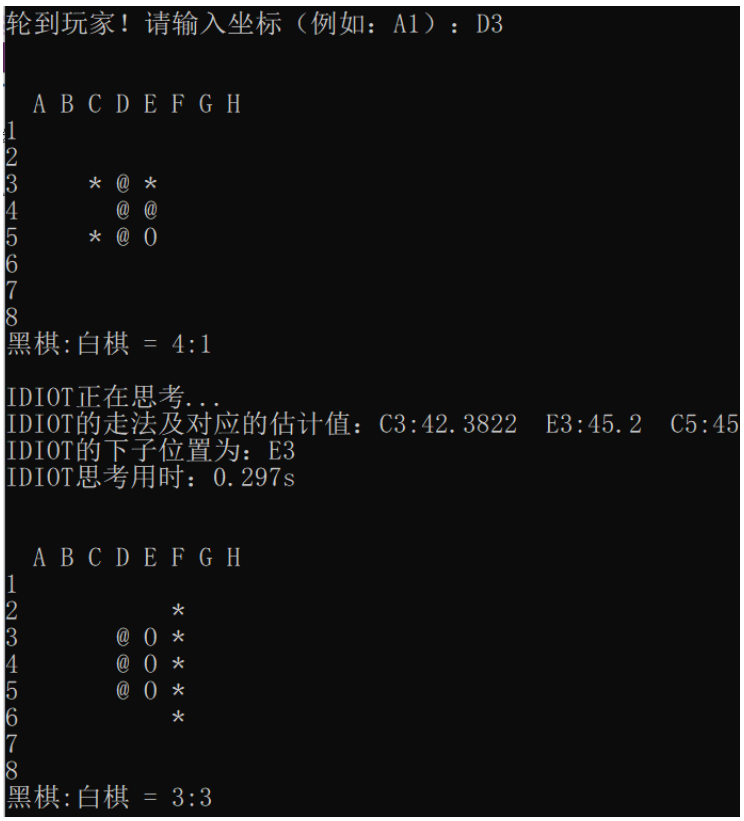
```
    for (int i = 0; i < 4; ++i)
        if (board[corner_pos[i][0]][corner_pos[i][1]] == color) {
            myconer++;
            for (int j = 0; j < 8; ++j)
                if (board[corner_pos[i][0]][j] == color)
                    myside += sideVal[i];
                else
                    break;
            for (int j = 0; j < 8; ++j)
                if (board[j][corner_pos[i][1]] == color)
                    myside += sideVal[i];
                else
                    break;
        }
        else if (board[corner_pos[i][0]][corner_pos[i][1]] == opp) {
            opconer++;
            for (int j = 0; j < 8; ++j)
                if (board[corner_pos[i][0]][j] == opp)
                    opside += sideVal[i];
                else
                    break;
            for (int j = 0; j < 8; ++j)
                if (board[j][corner_pos[i][1]] == opp)
                    opside += sideVal[i];
                else
                    break;
        }
    sidestableeval = 2.5 * (myside - opside);
    cornereval = 25 * (myconer - opconer);
    //计算整个棋局估计值
    return score * 0.02 + moveeval * 1 + sidestableeval * 6.0 + cornereval *
    8.0 + rateeval * 0.2;
}
```

实验结果及分析

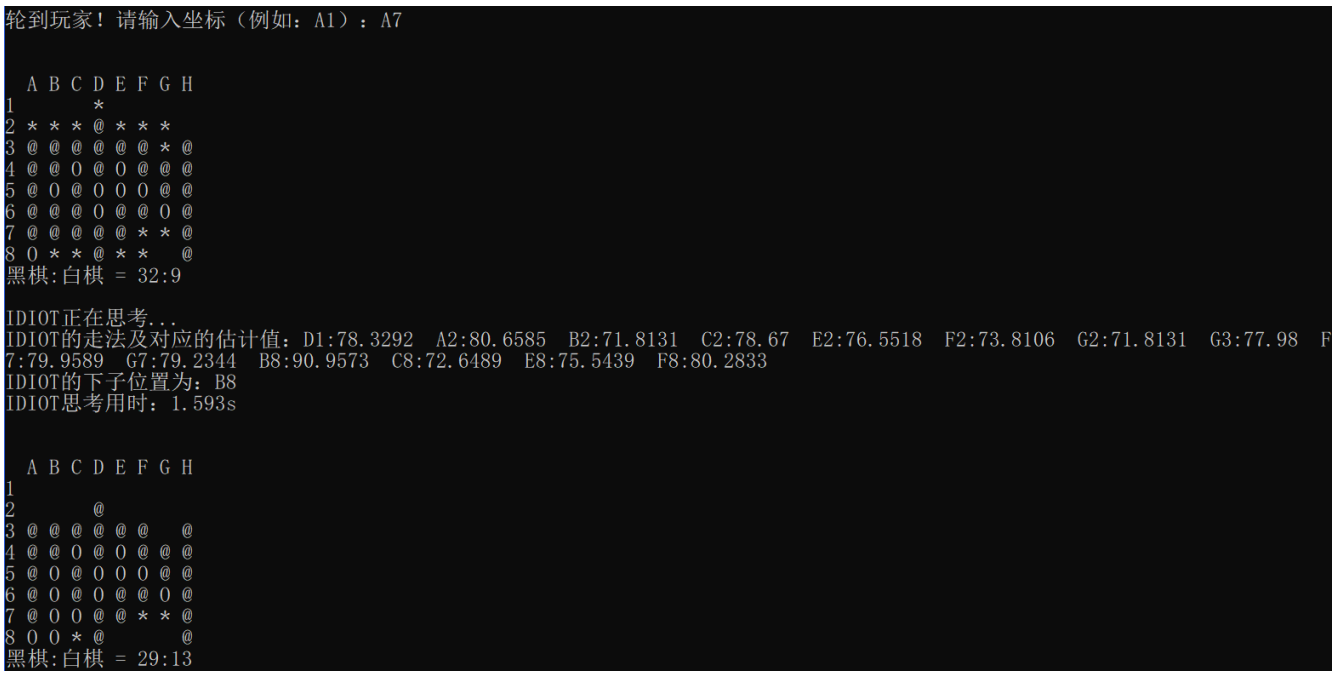
· 实验结果展示

玩家为黑棋，AI 为白棋，搜索深度为 4，AI 选择估计值最大的点落子。棋盘中“@”为黑棋，“O”为白棋，“*”为可落子位置。

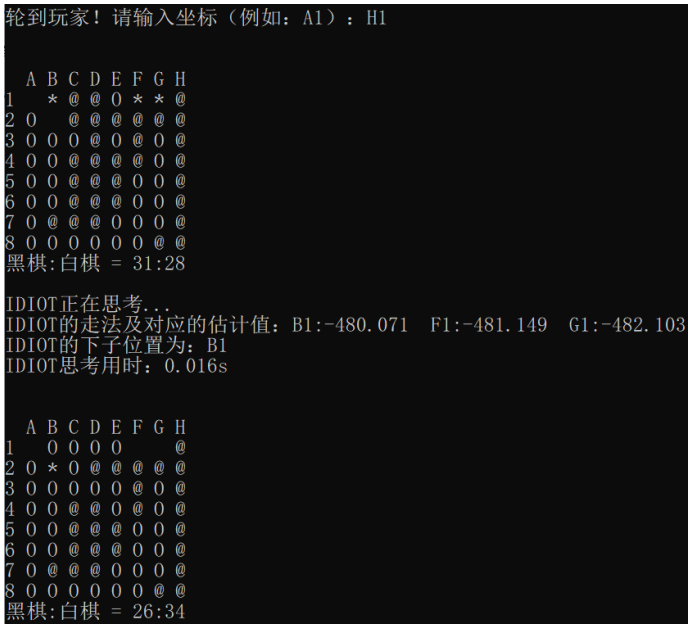
刚开局，黑棋落子后，AI 落子如图所示：



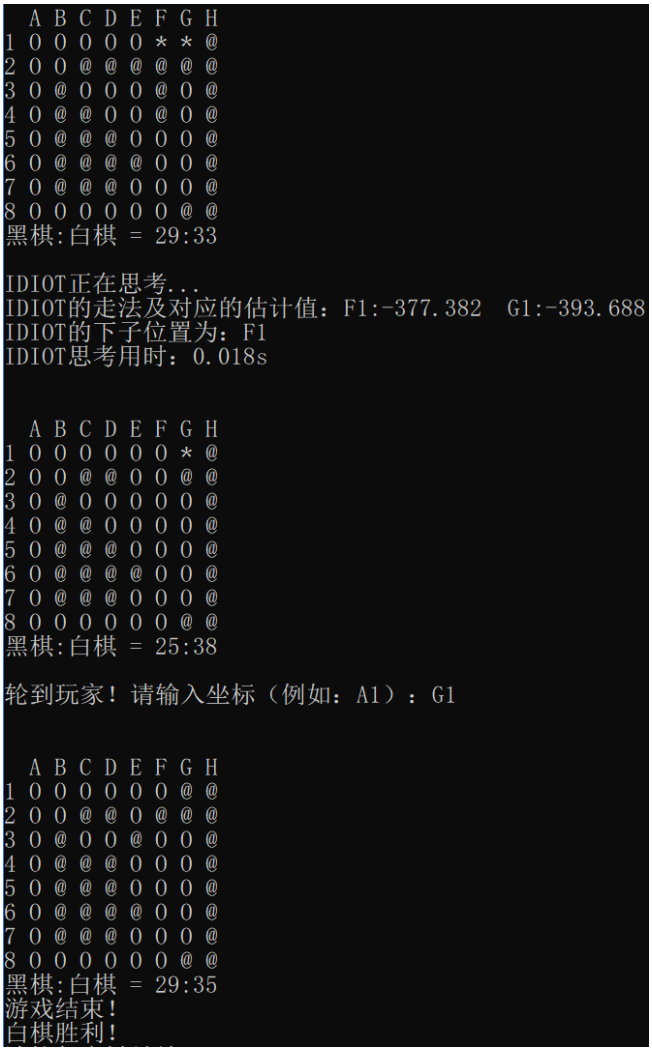
比赛中期黑棋优势较大，AI 处于下风，如图所示：



比赛后期，白棋逆转，AI 处于上风，如图所示：



比赛的最后，AI 执白棋反败为胜，如图所示：



· 评测指标展示

如下图所示，AI 可落子的点为 B1、F1、G1，它们的估计值分别是-480.071、-481.149、-482.103，这些估计值是通过分别计算白棋的位置特征估计值、黑白子比例估计值、边界稳定子估计值、行动力估计值，再加权求和得到的。显然 AI 选择在 B1 落子对白棋的优势较大，局势由原来的黑棋：白棋 ＝ 31：28 变为 26：34，白棋从原来的劣势转变为现在的优势。