中山大學
SUN YAT-SEN UNIVERSITY

《人工智能实验》

实验报告

Lab 10

博弈树搜索

学 院 名 称：	数据科学与计算机学院
专业（班级）：	17 级计算机科学与技术
学 生 姓 名：	薛伟豪
学 号：	17341178
联 系 方 式：	15013041671

Lab 10：博弈树搜索

1. 算法原理

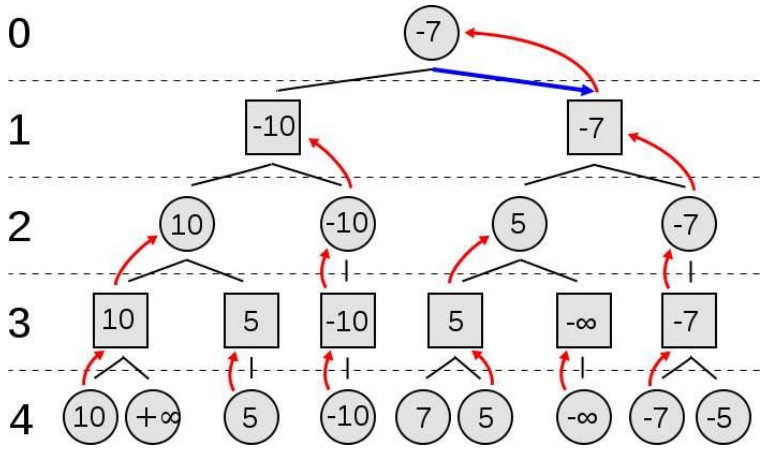
1.1. MiniMax策略

在博弈过程中，任何一方都希望自己取得胜利，因此当某一方当前有多个行动方案可供选择时，总会挑选对自己最为有利而对对方最为不利的行动方案。对于任一种博弈竞赛，我们都可以将其构成一个博弈树。以下棋为例，博弈树中的每个节点代表某一个棋局状态，每个分支代表走一步棋，根节点代表最初始的棋局状态，叶子节点表示对弈结束时的棋局状态。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，每个节点均有一个评价值，该评价值通过特定的评价函数和该节点对应的棋局状态算出，以表示该节点的优劣得分。

MiniMax策略简单来讲可以归纳为以下三点：

- PlayA和PlayB的行动逐层交替。
- A和B的利益关系对立，即假设A要使分数更大，B就要使分数更小。
- A和B均采取最优策略。

如图所示为一棵博弈树，圆形表示MAX节点，希望评估值越大越好；方形表示MIN节点，希望评估值越小越好：



图中第4层为叶子节点，第3层会根据第4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，即最小化对方的优势；第2层会根据第3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2层会根据第3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，即最小化对方

的优势；第1层会根据第2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。

实际编程的时候，是往下不断生长节点，然后动态更新每个父节点的预估值。但是，当层数增多的时候，博弈树会变得很庞大，从而每次建树需要进行很长时间。为了优化这种情况的发生，我们需要进行Alpha-beta剪枝。

1.2. Alpha-beta剪枝

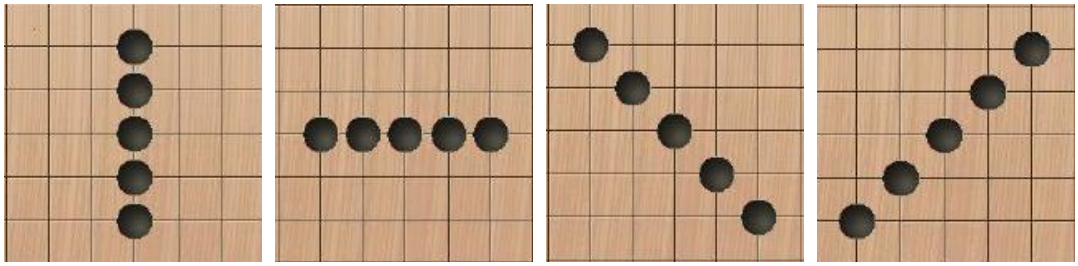
Alpha-beta剪枝建立在Minimax算法的基础上，但它减少了Minimax算法搜索树的节点数。我们对博弈树的叶子结点进行赋值，再从底往上更新。每个结点的α值默认为-∞，β值默认为∞，若是max结点，则根据子节点更新自己的α值，若是min结点，则根据子节点更新自己的beta值。

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。

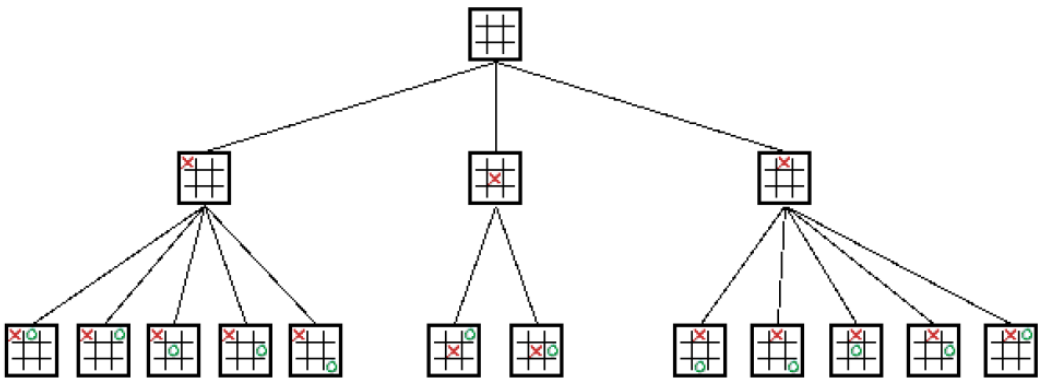
对于MAX层的节点，如果估计出其倒推值的下确界alpha大于或等于其MIN层父节点的估计倒推值的上确界beta，即beta≤alpha，则不必再扩展该MAX层节点的其余节点，因为其余节点的估计值对MIN层父节点的倒推值没有任何影响了，这个过程称为Beta剪枝。

1.3. 五子棋问题

五子棋是一种两人对弈的纯策略型棋类游戏，通常双方分别使用黑白两色的棋子，下在棋盘直线与横线的交叉点上，先形成5子连线者获胜。



我们需要构建一棵极大极小博弈树，这里使用一张井字棋的搜索示意图来进行说明。



上图很清晰的展示了对局可能出现的所有情况（已经去除了等价的情况），如果让这个图延展下去，我们就相当于穷举了所有的下法，如果我们能在知道所有下法的情况下，对这些下法加以判断，我们的AI自然就可以选择具有最高获胜可能的位置来下棋。

极大极小博弈树就是一种选择方法，由于五子棋以及大多数博弈类游戏是无法穷举出所有可能的步骤的（状态会随着博弈树的扩展而呈指数级增长），所以通常我们只会扩展有限的层数，而AI的智能高低，通常就会取决于能够扩展的层数，层数越高，AI了解的信息就越多，就越能做出有利于它的判断。

为了让计算机选择那些获胜可能性高的步骤走，我们就需要一个对局面进行打分的算法，越有利，算法给出的分数越高。在得到这个算法过后，计算机就可以进行选择了，在极大极小博弈树上的选择规则是这样的：

- AI会选择子树总具有最高估值叶子节点的路径
- USER会选择子树中具有最小估值叶子节点的路径。

这样的原则很容易理解，作为玩家，我所选择的落点一定要使自己的利益最大化，而相应的在考虑对手的时候，也不要低估他，一定要假设他会走对他自己最有利，也就是对我最不利的那一步。

接下来，我们实现关键的局面评分步骤。直接分析整个棋面是一件很复杂的事情，为了让其具备可分析性，我们可以将其进行分解，分解成易于我们理解和实现的子问题。

对于一个二维的期面，五子棋不同于围棋，五子棋的胜负只取决于一条线上的棋子，所以根据五子棋的这一特征，我们就来考虑将二维的棋面转换为一维的，下面是一种简单的思考方式，对于整个棋盘，我们只需要考虑四个方向即可，按照四个方向来将棋盘转换为一维向量的集合。这样一来，我们可以评估每条线的状态，并进行汇总，作为我们棋面的得分：

$$evaluateVaule = \sum_{i} evaluateLine(LineState[i])$$

接下来我们所要做的就是评价每一条线状态，根据五子棋的规则，我们可以很容易穷举出各种可能出现的基本棋型，我们首先为这些基本棋型进行识别和评价，并且统计每个线状态中出现了多少种下面所述的棋型，并据此得出评价值，得到如下图所示的静态估值表：

棋型标注	棋型示意	静态估值
AI_ZERO	□□□	0
AI_ONE	□X□	10
AI_ONE_S	□XO	1
AI_TWO	□XX□	100
AI_TWO_S	□XXO	10
AI_THREE	□XXX□	1000
AI_THREE_S	□XXXO	100
AI_FOUR	□XXXX□	10000
AI_FOUR_S	□XXXXO	1000
AI_FIVE	XXXXX	100000
备注：□表示空位，X表示待评估方棋子，O表示待评估方对方棋子		

1.4. 五子棋问题的进一步优化

注意到，如果我们搜索到第四层，那么对于这个11*11的棋盘，总共需要搜索：120+120*119+120*119*118+120*119*118*117=198849120个状态节点！搜索如此多的状态节点的开销是非常庞大的，因此我们需要想办法减少需要搜索的状态节点。

我们可以采取以下方法来减少需要搜索的状态节点：

- 利用此前提到的alpha-beta剪枝对博弈树进行剪枝。
- 每次搜索仅搜索落子点周围2*2格范围内存在棋子的位置，这样可以避免搜索一些明显无用的节点，而且可以大幅度提升整体的搜索速度。
- 避免对必胜/负局面进行搜索，当搜索过程中出现了必胜/负局面的时候直接返回不再搜索，因为此时继续搜索是没有必要的，直接返回当前棋局的估价值即可。
- 加入随机化AI的下棋方式，普通的AI算法对于给定的玩家下棋方式会给出固定的回应，这就导致玩家获胜一次之后只要此后每次都按此方式下棋，都能够获胜。为了避免这种情况，可以在AI选择下子位置的时候，在估值相差不多的几个位置中随机挑选一个进行放置，以此增加AI 的灵活性。
- 规划搜索顺序，有很多有价值的下子点存在于更靠近棋盘中央的地方，如果从棋盘中央向外搜索的话，则能够提高α-β剪枝的效率，让尽可能多的分支被排除。

2. 伪代码

2.1. MiniMax策略

```
Function MiniMax(node, depth)
1.  if node is a terminal node or depth = 0
2.      return the heuristic value of node
3.  if the adversary is to play at node
4.      let α := +∞
5.      for each child of node
6.          α := min(α, minimax(child, depth-1))
7.  else if we are to play at node
8.      let α := -∞
9.      for each child of node
10.         α := max(α, minimax(child, depth-1))
11. return α
```

2.2. Alpha-beta剪枝算法

```
Function AlphaBetaPruning(node, depth, alpha, beta)
1.    if node is a terminal node or depth = 0
2.        return the heuristic value of node
3.    if the adversary is to play at node
4.        let α := +∞
5.        for each child of node
6.            α := min(α, minimax(child, depth-1, alpha, beta))
7.            beta := min(α, beta)
8.            if beta <= alpha
9.                break
10.   else if we are to play at node
11.       let α := -∞
12.       for each child of node
13.           α := max(α, minimax(child, depth-1, alpha, beta))
14.           alpha := max(α, alpha)
15.           if alpha >= beta
16.               break
17.   return α
```

3. 重要代码展示

- **Alpha-beta剪枝**

```
/*************************************************
参数说明:
state 待转换的状态
type  当前层的标记: MAX MIN
depth 当前层深
alpha 父层alpha值
beta  父层beta值
*************************************************/
int minMax(ChessBoard state, int x, int y, int type, int depth, int alpha, int beta) {
    ChessBoard newState(state);
    newState.placePiece(x, y, nextType(type));
    int weight = 0;
    int max = -INF;     //下层权值上界
    int min = INF;      //下层权值下界

    if (depth < MAX_DEPTH) {
        //已输或已胜则不继续搜索
        if (evaluatePiece(newState, x, y, nextType(type)) >= AI_FIVE) {
            if (type == MIN_NODE)
                return AI_FIVE;     //我方胜
            else
                return -AI_FIVE;    //我方负
        }
```

```
        int i, j;
        for (i = 0; i < BOARD_SIZE; ++i) {
            for (j = 0; j < BOARD_SIZE; ++j) {
                if (newState.chessBoard[i][j].type == EMPTY && canSearch(newState, i, j)) {
                    weight = minMax(newState, i, j, nextType(type), depth + 1, min, max);
                    if (weight > max)
                        max = weight;   //更新下层上界
                    if (weight < min)
                        min = weight;   //更新下层下界
                    // alpha-beta
                    if (type == MAX_NODE) {
                        if (max >= alpha)
                            return max;
                    }
                    else {
                        if (min <= beta)
                            return min;
                    }
                }
                else
                    continue;
            }
        }
        if (type == MAX_NODE)
            return max; //最大层给出最大值
        else
            return min; //最小层给出最小值
    }
    else {
        //评估我方局面
        weight = evaluateState(newState, MAX_NODE);
        // 评估对方局面
        weight -= type == MIN_NODE ? evaluateState(newState, MIN_NODE)*10 : evaluateState(newState, MIN_NODE);
        return weight;      //搜索到限定层后给出权值
    }
}
```

- **评价一个方向上的棋子**

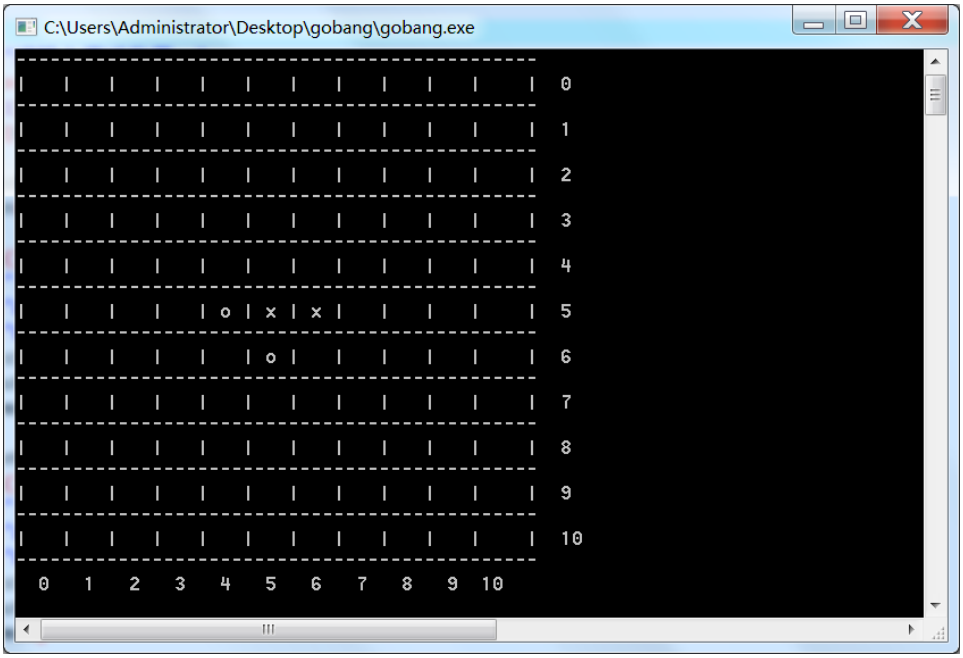
```
//以center作为评估位置进行评价一个方向的棋子
int evaluateLine(int line[], bool ALL) {
    int value = 0;      //估值
    int cnt = 0;        //连子数
    int blk = 0;        //封闭数
    for (int i = 0; i < BOARD_SIZE; ++i) {
        if (line[i] == AI_MY) {     //找到第一个己方的棋子,还原计数
            cnt = 1;
            blk = 0;
            //看左侧是否封闭
            if (line[i - 1] == AI_OP)
                ++blk;
            //计算连子数
            for (i = i + 1; i < BOARD_SIZE && line[i] == AI_MY; ++i, ++cnt);
            //看右侧是否封闭
            if (line[i] == AI_OP)
                ++blk;
            //计算评估值
            value += getValue(cnt, blk);
        }
    }
    return value;
}
```

- **评价当前棋局的某一方**

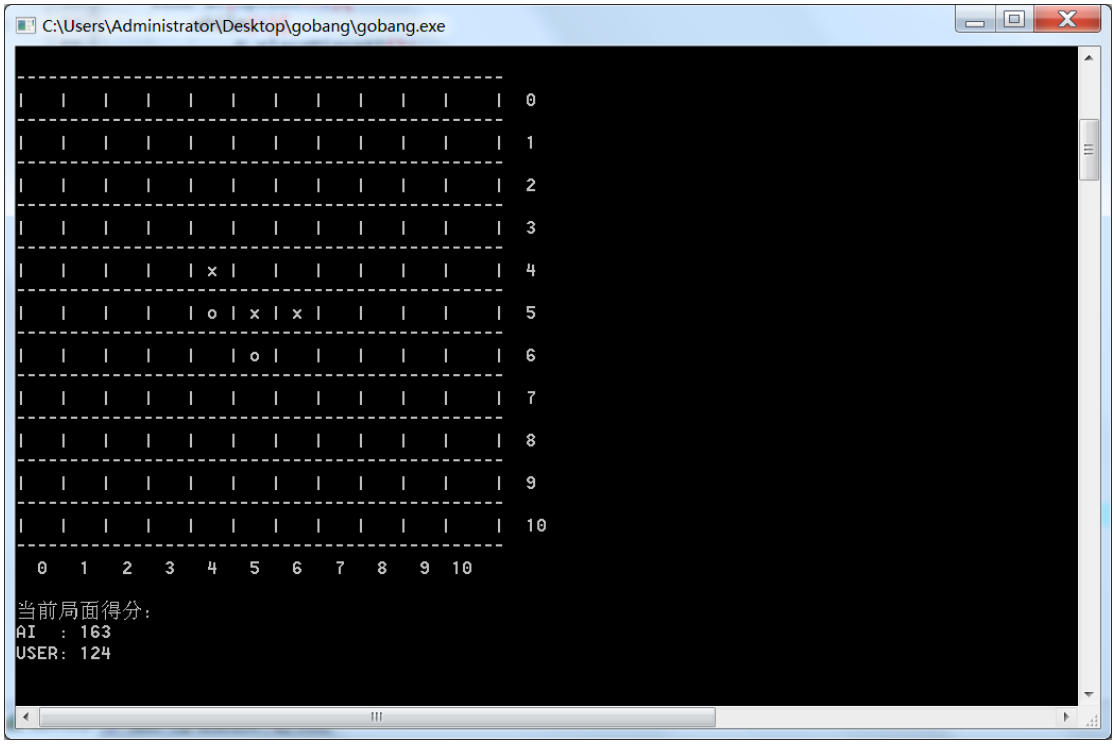
```
//评价一个棋面上的一方
int evaluateState(ChessBoard state, int type) {
    int value = 0;
    // 分解成线状态
    int line[6][17];
    int lineP;

    for (int p = 0; p < 6; ++p)
        line[p][0] = line[p][16] = AI_OP;

    //从四个方向产生
    for (int i = 0; i < BOARD_SIZE; ++i) {
        //产生线状态
        lineP = 1;
        for (int j = 0; j < BOARD_SIZE; ++j) {
            line[0][lineP] = getPieceType(state, i, j, type);                      /* | */
            line[1][lineP] = getPieceType(state, j, i, type);                      /* - */
            line[2][lineP] = getPieceType(state, i + j, j, type);                  /* \ */
            line[3][lineP] = getPieceType(state, i - j, j, type);                  /* / */
            line[4][lineP] = getPieceType(state, j, i + j, type);                  /* \ */
            line[5][lineP] = getPieceType(state, BOARD_SIZE - j - 1, i + j, type); /* / */
            ++lineP;
        }
        //估计
        int special = i == 0 ? 4 : 6;
        for (int p = 0; p < special; ++p) {
            value += evaluateLine(line[p], true);
        }
    }
    return value;
}
```

4. 实验结果展示

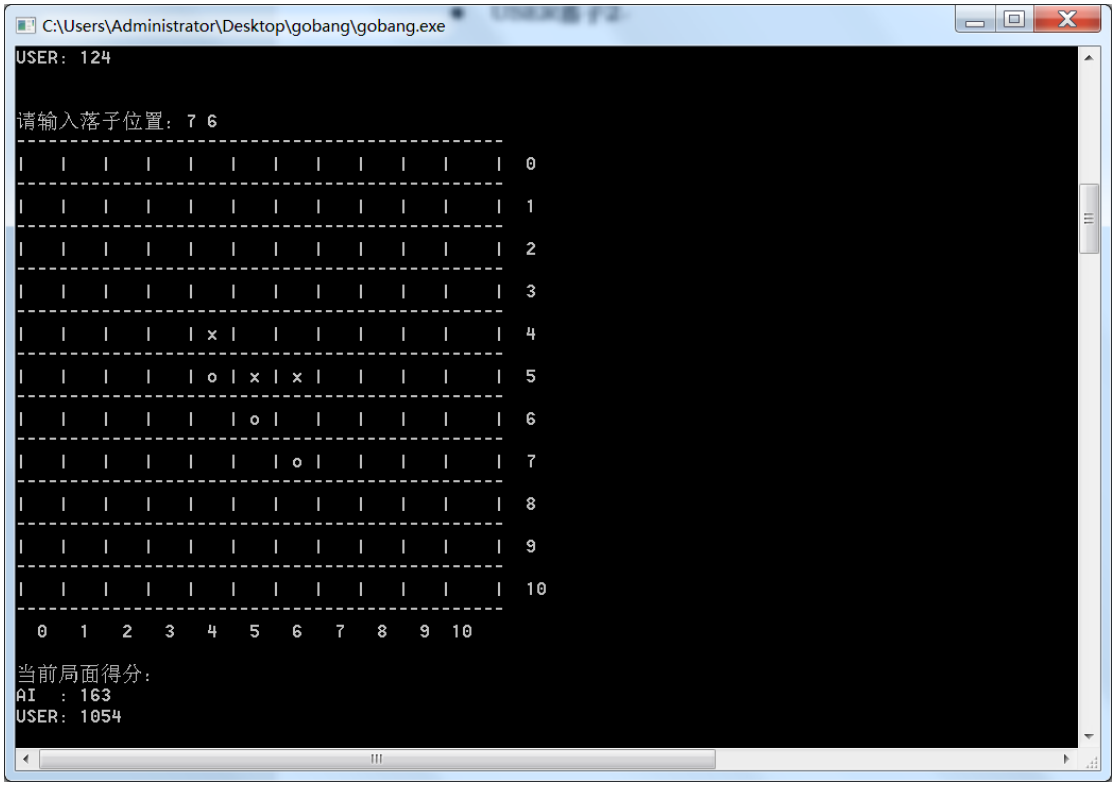
- **初始状态**



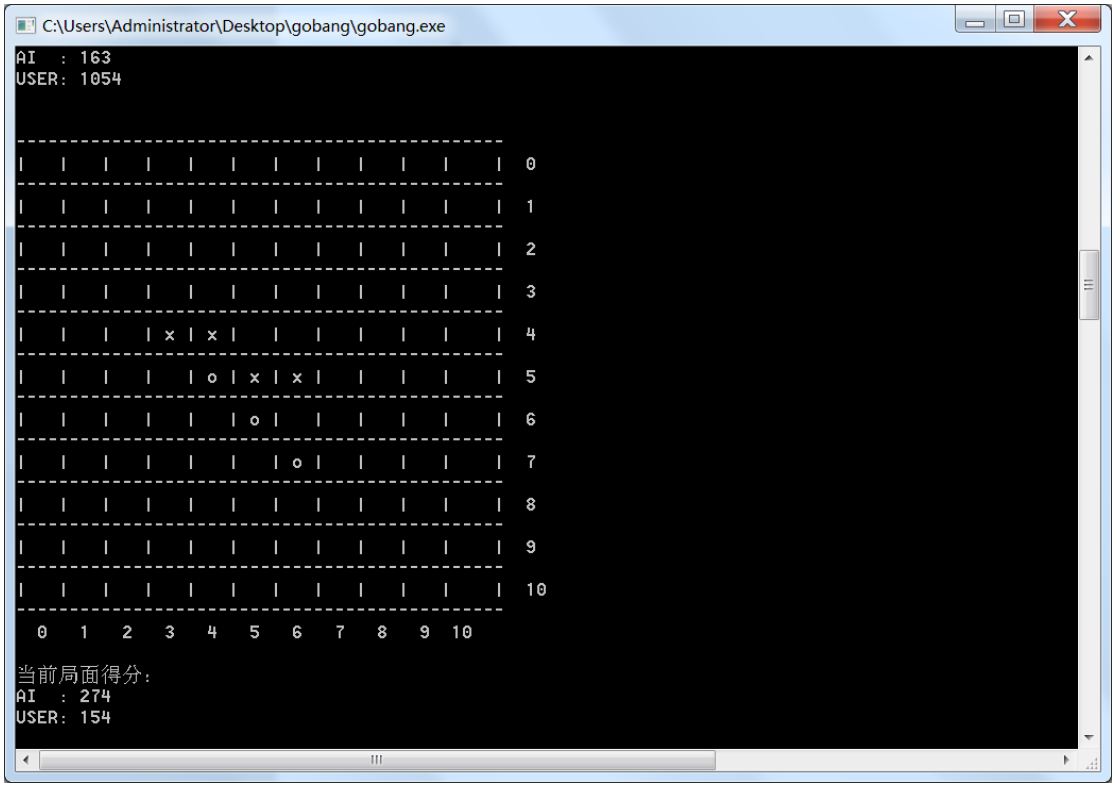
- **机器落子1**



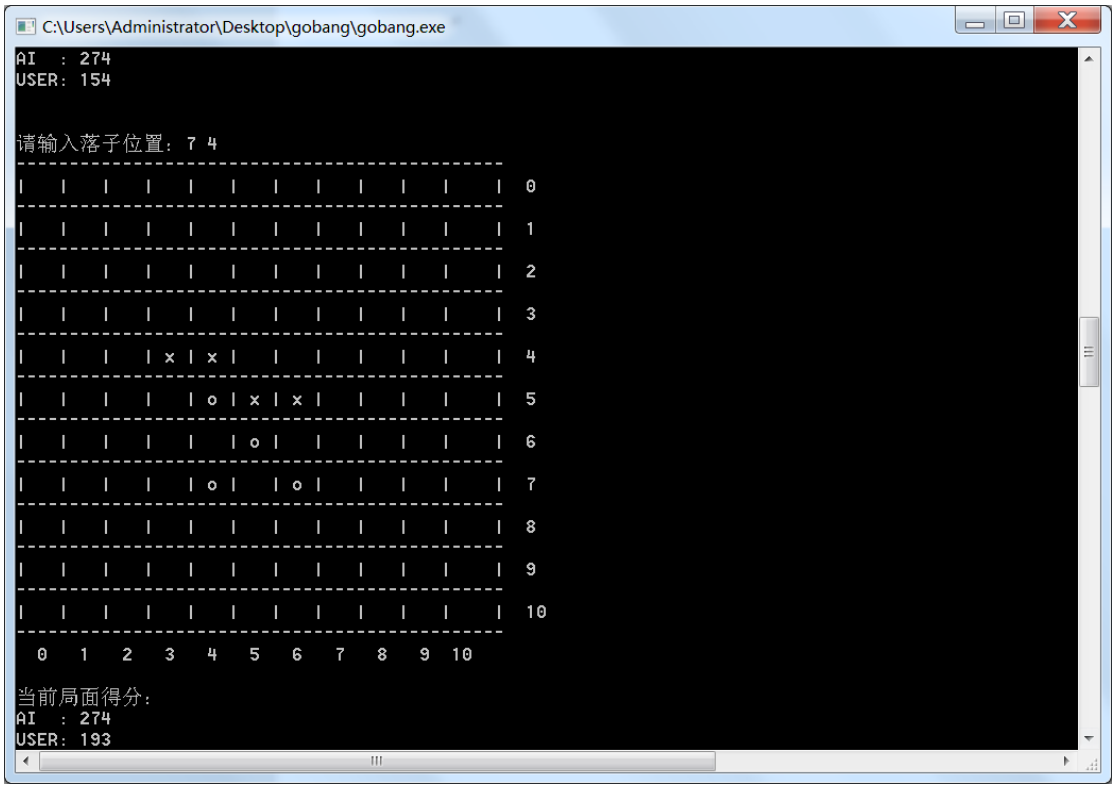
- **USER落子2**



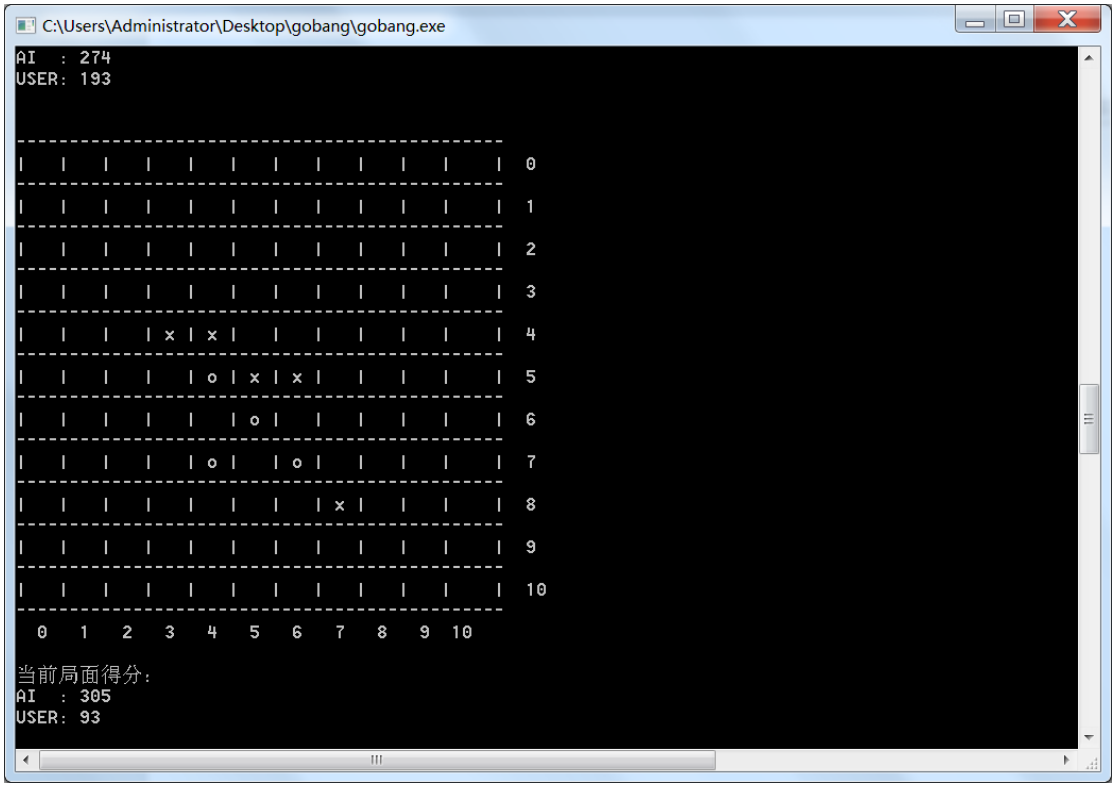
- **机器落子3**



- **USER落子4**



- **机器落子5**



- **USER落子6**