实验报告

————alpha-beta 剪枝

姓名：张子玉　　学号：19335276　　日期：2021.9.16

摘要：

编写一个中国象棋博弈程序，要求用 alpha-beta 剪枝算法，可以实现人机对弈。棋局评估方法可以参考已有文献，要求具有下棋界面，界面编程也可以参考网上程序，但正式实验报告要引用参考过的文献和程序。

本次实验使用 JavaScript 来实现，有以下几点原因：

1. HTML+JavaScript 使得界面设计简单，可以把重心放在对算法的学习上
2. 不用搭建环境
3. 编写过程中设计的都是基础语法，没有语法方面的障碍

导言

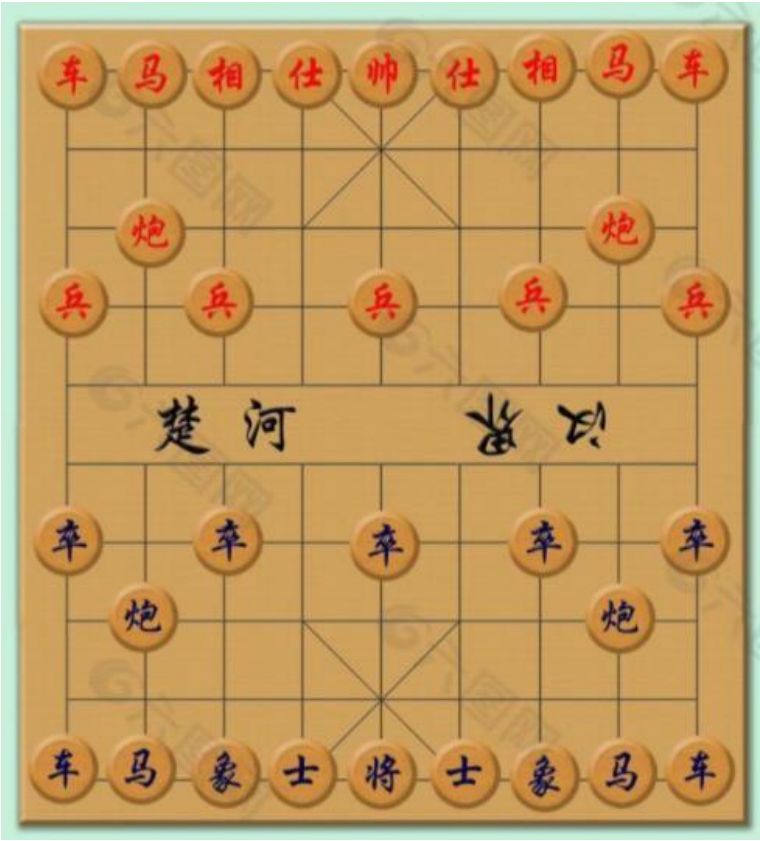
零和博弈

零和博弈（zero-sum game），又称零和游戏，与非零和博弈相对，是博弈论的一个概念，属非合作博弈。指参与博弈的各方，在严格竞争下，一方的收益必然意味着另一方的损失，博弈各方的收益和损失相加总和永远为“零”，双方不存在合作的可能。

中国象棋

中国象棋是一个古老而经典的零和博弈问题。这同时也是一个比较复杂的博弈问题，没有通用的有效解法。在古代博弈问题通常就是人与人之间的博弈，而解决复杂博弈问题的能力也一定程度上反映出了人的智慧。实现计算机能够自动在中国象棋中像人一样思考和操作是解决本问题的关键。

中国象棋有红黑两方，红方先走。在没有让子的情况下，红黑两方各有五个兵(卒)，两个砲(炮)，两个車，两个马，两个相(象)，两个仕(士)，一个帅(将)。象棋棋盘有被分为对称的两半，中间由"楚河 汉界"隔开，整个棋盘共有十行九列个交叉点，每个交叉点可以可以给一个棋子落子。初始棋盘布局如下：



中国象棋基本规则:

- 象棋盘由九道直线和十道横线交叉组成。棋盘上共有九十个交叉点，象棋子应摆放和活动在这些交叉点上。
- 棋盘中间直线断开处，称为“河界”；河界内应标注“楚河 汉界”，两端划有斜交叉线的地方，称为“九宫”。
- 九道直线，红棋方面从右到左用中文数字一至九来标识；黑棋方面从右到左用阿拉伯数字 1 至 9 来标识。
- 棋子共有三十二个，分为红、黑两组，每组十六个，各分七种，其名称和数目如下：红棋子：帅一个，车、马、炮、相、士各两个，兵五个。黑棋子：将一个，车、马、炮、象、士各两个，卒五个。
- 子力价值：原则上一车相当于双马、双炮或一马一炮，马炮等值，车、马、炮称为“强子”；仕（士）、相（象）等值，称为“弱子”。过河兵（卒）价值浮动。
- 对局时，由执红棋的一方先走，双方轮流各走一着，直至分出胜、负、和，对局即终了。双方各走一着，称为一个回合。
- 帅（将）每着只许走一格，只能在“九宫”内前、后、左、右移动。任何一方走之后，都不准造成帅、将在同一直线上直接对面。
- 仕（士）每着只许沿“九宫”斜线走一格，可进可退。
- 相（象）不能越过“河界”，每着斜走两格，可进可退，即俗称“相 （象）走田字”。当“田”字中心有棋子（无论何方）占据，俗称“塞相（象）眼”，则不许进或退。
- 马每着走一直（或一横）一斜，可进可退，即俗称“马走日字”。如果在先直（横）的那个交叉点有棋子（无论何方）占据，俗称“蹩马腿”，则不许进或退。
- 车每一着可以直进、直退、横走，不限格数，但不可隔子而行。
- 炮的走法同车一样；吃子时必须隔一个棋子（无论何方）跳吃，即俗称“炮打隔子”。

- 兵（卒）在过“河界”前，每着只许向前直走一格；过“河界”后，每着可向前直走或横走一格，但不能后退。
- 行棋方将某个棋子从一个点走到另一个点，即为一着。走一着棋时，如果己方棋子能够走到的点有对方棋子存在，就可以把对方棋子吃掉而占领该点。
- 一方棋子攻击对方的帅（将），并在下一着能将其吃掉，称为“将军”，或简称“将”。
- 被“将军”方必须立即“应将”，如果无法“应将”或不“应将”，即被“将死”。
- 轮到行棋的一方无子可走，即被“困毙”。
- 一方行棋后形成帅（将）直接对面、或主动送吃帅（将），或在被“将军”时误走它子，没有“应将”， 听任对方吃帅（将），均属“自杀”。
- 单方“长将”判负。
- 形成双方均无取胜可能的简单局势，或棋局出现待判局面，符合“棋例”中“不变作和”的有关规定，或符合“自然限着”的回合规定，即在连续 60 回合中（可根据比赛等级酌减）均未吃过棋子，为和棋。

实验将尽量完整无误地实现上述中国象棋规则。

极小化极大算法

极小化极大算法，是一种找出失败的最大可能性中的最小值的算法。

局面估价函数：我们给每个局面（state）规定一个估价函数值 f，评价它对于己方的有利程度。
Max 局面：假设这个局面轮到己方走，有多种决策可以选择，其中每种决策都导致一种子局面（sub-state）。由于决策权在我们手中，当然是选择估价函数值 f 最大的子局面，因此该局面的估价函数值等于子局面 f 值的最大值。
Min 局面：假设这个局面轮到对方走，在最坏的情况下，对方当然是选择估价函数值 f 最小的子局面，因此该局面的估价函数值等于子局面 f 值的最小值。
通常可以设定叶子结点局面的估价值。

博弈过程中，可以选择向后看几步。例如，往后看两步（也就是只直接计算往后数两步的估价值），那么，情况就是这样的：先手预测了自己走完这一步的所有可能局面，并同时预测了对手的所有应对方案（也就是对手会选择让下一步局面估价函数最小的走法，所以先手这一步应该选择，让后手不管走么走估价函数都尽量大的走法），综合这些信息，选择了一个最优的局面，后手也是如此。
需要的计算量是随着向后看的步数的增加而呈指数级增长的。但是，这些状态中其实是包含很多不必要的状态的，所以我们可以进行剪枝。

Alpha-Beta 剪枝

Alpha-beta 剪枝的名称来自计算过程中传递的两个边界，这些边界基于已经看到的搜索树部分来限制可能的解决方案集。其中，Alpha 表示目前所有可能解中的最大下界，Beta 表示目前所有可能解中的最小上界。
因此，如果搜索树上的一个节点被考虑作为最优解的路上的节点（或者说是这个节点被认为是有必要进行搜索的节点），那么它一定满足以下条件（N 是当前节点的估价值）：$\alpha \leq N \leq \beta$
在我们进行求解的过程中，alpha 和 beta 会逐渐逼近。如果对于某一个节点，出现了 $\alpha > \beta$

的情况，那么，说明这个点一定不会产生最优解了，所以，我们就不再对其进行扩展（也就是不再生成子节点），这样就完成了对博弈树的剪枝。

实验过程

下面将重点描述搜索部分，一些细节部分的实现，例如，实现将帅不能对面，单方长将作负，双方不变作和等规则，鉴于本次实验的主要目的是编写 alpha-beta 剪枝，在此不再赘述。

1. 界面设计

首先，我们需要将整个棋盘表示出来，中国象棋有 10 行 9 列，很自然地想到可以用 10×9 矩阵表示棋盘。但是这里为了快速判断棋子是否走出边界，使用 16x16 矩阵来表示棋盘。使用一个辅助数组，表示棋盘中哪些位置是真实的，哪些是虚拟的：

```
// 辅助数组，用于判断棋子是否在棋盘上
var IN_BOARD_ = [
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
  0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
  0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
  0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
  0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
  0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
  0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
  0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
  0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
  0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
];
```

为了减少计算量，我们使用长度为 256 的一维数组来表示棋盘。这样，一个位置只需要一个变量。

同时，为了实现一维数组和二维数组之间的位置转换，设置函数：

`function COORD_XY(x,y)` 实现输入二维数组的 x、y 坐标，返回在一维数组中的下标

`function RANK_Y(sq)` 实现输入在一维数组中的位置，返回在二维数组中的行数

`function FILE_X(sq)` 实现输入在一维数组中的位置，返回在二维数组中的行数

函数 `function IN_BOARD(sq)`实现输入位置返回该位置是否在真实棋盘中。

接下来，我们要表示棋子，每种棋子的基础值如下：

```
// 棋子编号
var PIECE_UNKNOWN = -1;
var PIECE_KING = 0;   // 将
var PIECE_ADVISOR = 1;  // 士
var PIECE_BISHOP = 2; // 象
var PIECE_KNIGHT = 3; // 马
var PIECE_ROOK = 4;   // 车
var PIECE_CANNON = 5; // 炮
var PIECE_PAWN = 6;   // 卒
```

若为红方的棋子，则在基础值上+8；若为黑方的棋子，则在基础值上+16。这样，红方棋子&8=8；黑方棋子&16=16。例如，红方的象为 10，二进制为 0000 1010，0000 1010&0000 1000=0000 1000

参考 FEN 格式串（一种使用 ASCII 码字符描述国际象棋局面的标准），这里我们也使用字符串来表示棋盘局面。

我们以初始局面来进行解说：

```
"rnbakabnr/9/1c5c1/p1p1p1p1p/9/9/P1P1P1P1P/1C5C1/9/RNBAKABNR w"
```

中国象棋共有十行，每行都用一个字符串表示，行间使用正斜杠分割。棋子用字母表示，如果没有棋子，则用数字表示出相邻连续的空位数。

字母表示棋子：

红方	字母	黑方	字母
帅	K	将	k
仕	A	士	a
相	B	象	b
马	N	马	n
车	R	车	r
炮	C	炮	c
兵	P	卒	p

最后一个字母（与前面的字符串之间用空格隔开）表示轮到哪一方走，“w”表示红方，“b”表示黑方。

2. 棋子走法

由起点和终点来表示走法，由于棋盘的大小是 256，所以起点和终点的位置都是不超过 255 的整数，可以将两者压缩到一个整数中表示，令走法=起点+终点<<8，函数 function MOVE(sqSrc,sqDst) 实现走法的表示，function SRC(mv) 获取起点，function DST(mv)获取终点。

中国象棋中对各种棋子的走法都有规定，使用辅助数组，辅助数组标识出棋子的合法位置或合法走法。

下面以对将（帅）走法为例进行说明:

将（帅）只能在九宫之内活动，设置辅助数组，判断是否在九宫内，九宫位置=1:

```
// 辅助数组，用于判断是否在九宫
var IN_FORT_ = [
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
];
```

将（帅）移动的时候，可上可下，可左可右，每次走动只能按竖线或横线走动一格，由此可知，终点和起点满足下列等式之一:

pcDst = sqSrc - 16

pcDst = sqSrc - 1

pcDst = sqSrc + 1

pcDst = sqSrc + 16

设置 16x16x2 的一维数组 LEGAL_SPAN 来辅助判断，在该数组中，(-16 + 256)、(-1 + 256)、(1 + 256)、(16 + 256) 四个位置是 1，其他位置都不是 1。若走法合法，则满足 LEGAL_SPAN[sqDst - sqSrc + 256] == 1

有了能检验走法是否合法的方法后，便可以生成走法了，仍然以将（帅）为例，下面为伪代码:

```
1. for i<-0 to 3 {       // 将的4个方向
2.    sqDst <- sqSrc + KING_DELTA[i]    // 得到一个可能的终点位置
3.    if 该位置不位于九宫中
4.      // 该走法不合法，执行下一轮循环
```

```
    continue
  pcDst <- 终点位置的棋子;       // 如果终点位置没有棋子，那么 pcDst=0
  if pcDst 不是本方棋子
    走法合法，保存到步骤数组中
```

3. 极大值极小值搜索

搜索时先设定一个参数 MINMAXDEPTH，生成固定的最多 MINMAXDEPTH 层的博弈搜索树。

根节点的层次是 MINMAXDEPTH，每往下搜索一层，层次减 1。当层次为 0 时，到达叶子节点，不再向下搜索，而是调用评估函数计算估值。

伪代码如下：

```
// MAX 点搜索
function maxAlphaSearch(depth, alpha, beta) //alpha 为所有祖先的最大 alpha 值 beta 为所有祖先的最小 beta 值
    if depth=0
        return 估价函数评估结果
    moves <- 生成当前局面全部走法;
    value <- 0;
    遍历 moves 中的每一个走步 move
        执行 move
        value <- minSearch(depth - 1, alpha, beta)
        撤销 move
        if value > alpha        // 寻找子结点最大的估价值
            alpha <- value
            if depth = MAXDEPTH   // 如果回到了根节点，需要记录根节点的最佳走法
                记录根节点的最佳走法
            if alpha >= beta    //beta 剪枝
                return alpha
    return alpha            // 返回当前节点的最优值

// MIN 点搜索
function minBetaSearch(depth, alpha, beta) //alpha 为所有祖先的最大 alpha 值 beta 为所有祖先的最小 beta 值
    if depth=0
        return 估价函数评估结果
    moves <- 生成当前局面全部走法;
    value <- 0;
    遍历 moves 中的每一个走步 move
        执行 move
        value <- maxSearch(depth - 1, alpha, beta)
        撤销 move
```

```
        if value < beta         // 寻找子结点最小的估价值
            beta <- value
            if depth == MAXDEPTH   // 如果回到了根节点，需要记录根节点的最佳走法
                记录根节点的最佳走法
            if alpha >= beta    //alpha 剪枝
                return beta
    return beta             // 返回当前节点的最优值
```

由于这两种算法的大部分逻辑是一样的。可以合并这两种算法，合并后就是负极大值搜索算法，负极大值搜索伪代码如下：

```
function minmaxSearch(depth) //返回值不再表示对先手的有利程度 而是表示对当前走步的玩家的有利程度
    if depth=0
        return 估价函数评估结果(对当前走步的玩家的有利程度)
    valueBest <- 负无穷大          // 初始最优值
    moves <- 生成当前局面全部走法
    value <- 0
    遍历 moves 中的每一个走步 move
        执行 move
        value <- minmaxSearch(depth - 1) //对下一步走步玩家的有利程度的相反值(也就是不利程度)就是对当前走步玩家的有利程度
        撤销 move
        if value > valueBest        // 寻找对当前走步玩家估价值最大的决策
            valueBest <- value
            if depth=MAXDEPTH   // 如果回到了根节点，需要记录根节点的最佳走法
                记录根节点的最佳走法
    return valueBest            // 返回当前节点的最优值
```

极大值极小值搜索的关键就是估价函数。本实验中，估价函数的设置参考国际象棋程序中，常用两方子力价值和之差作为局面估价函数值，在此之上，加入对绝对位置的考虑。

参考知名中国象棋博弈程序 ElephantEye 的估价函数设计，对每个棋子的子力评估值可以看做是关于这个棋子种类和其绝对位置的二元函数。

ElephantEye 的估价函数设计具体如下：

```
[ // 帅
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  1,  1,  1,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  2,  2,  2,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0, 11, 15, 11,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
```

```
], [  // 仕
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0, 20,  0, 20,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0, 23,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0, 20,  0, 20,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
```

```
], [  // 相
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,  20,   0,   0,   0,  20,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,  18,   0,   0,   0,  23,   0,   0,   0,  18,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,  20,   0,   0,   0,  20,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
```

```
], [  // 马
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,  90,  90,  90,  96,  90,  96,  90,  90,  90,   0,   0,   0,   0,
   0,   0,   0,  90,  96, 103,  97,  94,  97, 103,  96,  90,   0,   0,   0,   0,
   0,   0,   0,  92,  98,  99, 103,  99, 103,  99,  98,  92,   0,   0,   0,   0,
   0,   0,   0,  93, 108, 100, 107, 100, 107, 100, 108,  93,   0,   0,   0,   0,
   0,   0,   0,  90, 100,  99, 103, 104, 103,  99, 100,  90,   0,   0,   0,   0,
   0,   0,   0,  90,  98, 101, 102, 103, 102, 101,  98,  90,   0,   0,   0,   0,
   0,   0,   0,  92,  94,  98,  95,  98,  95,  98,  94,  92,   0,   0,   0,   0,
   0,   0,   0,  93,  92,  94,  95,  92,  95,  94,  92,  93,   0,   0,   0,   0,
   0,   0,   0,  85,  90,  92,  93,  78,  93,  92,  90,  85,   0,   0,   0,   0,
   0,   0,   0,  88,  85,  90,  88,  90,  88,  90,  85,  88,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
```

```
], [  // 车
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,206,208,207,213,214,213,207,208,206,  0,  0,  0,  0,
  0,  0,  0,206,212,209,216,233,216,209,212,206,  0,  0,  0,  0,
  0,  0,  0,206,208,207,214,216,214,207,208,206,  0,  0,  0,  0,
  0,  0,  0,206,213,213,216,216,216,213,213,206,  0,  0,  0,  0,
  0,  0,  0,208,211,211,214,215,214,211,211,208,  0,  0,  0,  0,
  0,  0,  0,208,212,212,214,215,214,212,212,208,  0,  0,  0,  0,
  0,  0,  0,204,209,204,212,214,212,204,209,204,  0,  0,  0,  0,
  0,  0,  0,198,208,204,212,212,212,204,208,198,  0,  0,  0,  0,
  0,  0,  0,200,208,206,212,200,212,206,208,200,  0,  0,  0,  0,
  0,  0,  0,194,206,204,212,200,212,204,206,194,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
], [  // 炮
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,100,100, 96, 91, 90, 91, 96,100,100,  0,  0,  0,  0,
  0,  0,  0, 98, 98, 96, 92, 89, 92, 96, 98, 98,  0,  0,  0,  0,
  0,  0,  0, 97, 97, 96, 91, 92, 91, 96, 97, 97,  0,  0,  0,  0,
  0,  0,  0, 96, 99, 99, 98,100, 98, 99, 99, 96,  0,  0,  0,  0,
  0,  0,  0, 96, 96, 96, 96,100, 96, 96, 96, 96,  0,  0,  0,  0,
  0,  0,  0, 95, 96, 99, 96,100, 96, 99, 96, 95,  0,  0,  0,  0,
  0,  0,  0, 96, 96, 96, 96, 96, 96, 96, 96, 96,  0,  0,  0,  0,
  0,  0,  0, 97, 96,100, 99,101, 99,100, 96, 97,  0,  0,  0,  0,
  0,  0,  0, 96, 97, 98, 98, 98, 98, 98, 97, 96,  0,  0,  0,  0,
  0,  0,  0, 96, 96, 97, 99, 99, 99, 97, 96, 96,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
```

```
, [  // 兵
 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
 0,  0,  0,  9,  9,  9, 11, 13, 11,  9,  9,  9,  0,  0,  0,  0,
 0,  0,  0, 19, 24, 34, 42, 44, 42, 34, 24, 19,  0,  0,  0,  0,
 0,  0,  0, 19, 24, 32, 37, 37, 37, 32, 24, 19,  0,  0,  0,  0,
 0,  0,  0, 19, 23, 27, 29, 30, 29, 27, 23, 19,  0,  0,  0,  0,
 0,  0,  0, 14, 18, 20, 27, 29, 27, 20, 18, 14,  0,  0,  0,  0,
 0,  0,  0,  7,  0, 13,  0, 16,  0, 13,  0,  7,  0,  0,  0,  0,
 0,  0,  0,  7,  0,  7,  0, 15,  0,  7,  0,  7,  0,  0,  0,  0,
 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
```

由于帅是无价的，所以上面帅的权值仅仅表示帅的位置分值。可以看到马在整个棋盘的估价函数在 95 上下浮动，车在 210 上下浮动，炮在 95 上下浮动。兵过河前分数较小，过河后分数明显增加。

ElephantEye 的估价函数有着一个很重要的优点：简单且容易维护。这个估价函数可以拆分成每个棋子的子力加权和，且每个棋子子力只和其类型和位置有关。因此，其不需要在极大值极小值搜索的叶子节点才扫描整个棋盘计算局面估价函数值，其可以在走步的过程中O(1)根据当前移动(和吃掉棋子)维护局面估价函数值。

在搜索到可以取胜/取负的时候，此时不能简单的使用上述的棋子估价函数。这是因为可能有很多种取胜的走步，这时 AI 程序应该选择最短的可以到达取胜局面的分数，而不是直接根据棋力价值大小来走步。残局的棋力价值在几种走步中都差不多了，这样评估会导致 AI 程序变成几乎是随机选择走步，会导致取胜步数增加甚至很长时间无法将死对方。类似地，在 AI 程序所有决策值的估价都是取负局面时，应该采取到失败局面步数最长的决策，以拖延时间以增加发现对手破绽的机会。

因此，这里需要特别定义取胜/取负的分数。这个估价显然需要将搜索距离考虑进去，程序中定义取负的估价值为：distance * 16 - 无穷大(取胜的估价值为此相反数)。这样就可以通过距离指导 AI 程序选择步数尽量少的取胜走步，选择步数尽量多的取负走步。

极大值极小值搜索效率比较低。这是因为像中国象棋的这类棋类游戏，根据相关参考文献，其基本分支数为 30-40，也就是说每一步都有可能有 30-40 种走法。而搜索树的节点数关于深度是指数增长的，当深度达到 5-6 时，就可以产生 40 的 6 次方个结点，计算机是难以在几秒的时间内处理那么多数据的。而且这个时间复杂度增长会非常快(算法时间复杂度与访问到的结点数成正比，很快就可以达到天文数字)，导致即使用超级计算机，也难以求解很深的深度。

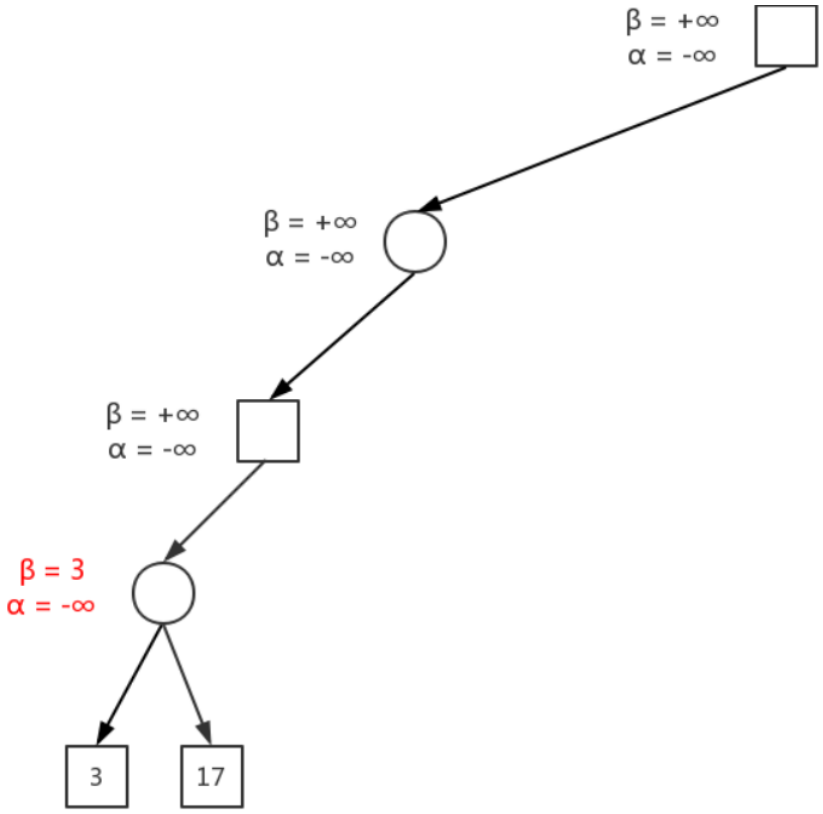
因此，我们下面进行 alpha-beta 剪枝，及时剪掉一些无用分枝，以此来提高算法的效率。

4. alpha-beta 剪枝

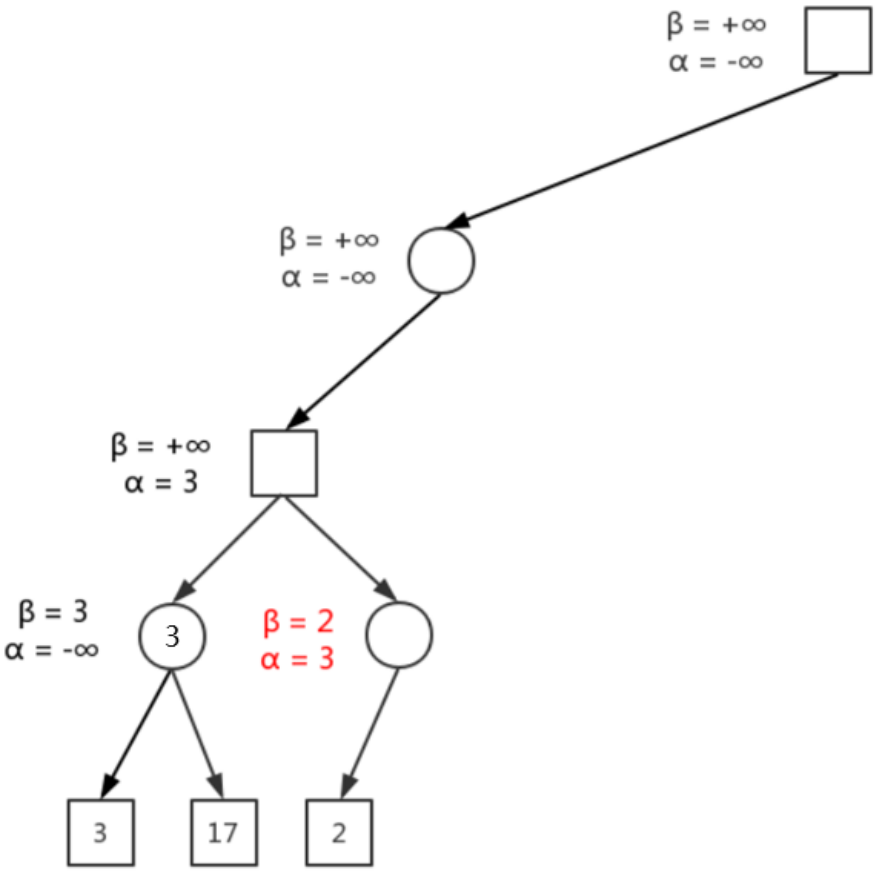
我们用一个简单的例子来讲解 alpha-beta 剪枝的过程：

正方形代表 max 节点，圆形代表 min 节点，假设每次往后看 4 步。alpha 表示目前所有可能解中的最大下界，beta 表示目前所有可能解中的最小上界。当 max 子节点的值>max 节点的 alpha 时，更新 alpha；当 min 子节点的值>min 节点的 beta 时，更新 beta。可以看到，随着求解过程的进行，alpha 和 beta 的值会逐渐逼近，当出现 alpha>beta 的情况时，进行剪枝。alpha 初始化为-∞，beta 初始化为+∞。

新生成的子节点的 alpha、beta 值继承自父节点。



第一次搜索到叶子节点，节点中最小值为 3<+∞，所以更新 min 节点 beta=3，min 节点值=3



该 min 节点搜索完毕，回溯到 max 父节点，由于子节点 min 节点=3>-∞，所以更新 max 节点的 alpha=3，由 max 节点再向下探索，并将 alpha、beta 遗传给子节点，所以新的 min 节点的初始 alpha=3，beta=+∞，探索其第一个子节点，值为 2<+∞，更新 beta=2，此时 alpha>beta，所以该节点不用再继续搜索，令该子节点的值=2。

上述为 beta 剪枝。alpha 剪枝：若任一极大值层节点的 α 值大于或等于它任一先辈极小值层节点的 β 值，即 α（后继层）≥β（先辈层），则可以中止该极大值层中这个节点以下的搜索。这个 MAX 节点的最终倒推值就确定为这个 α 值。对称的，beta 剪枝：若任一极小值层节点的 β 值小于或等于它任一先辈极大值层节点的 α 值，即 α（先辈层）≥β（后继层），则可中止该极小值层中这个节点以下的搜索。该节点最终的倒推值就确定为这个 β 值。

在负极大值搜索的基础上加入 alpha-beta 剪枝后，构成 alpha-beta search，伪代码如下：

```
1. function alphaBetaSearch(alpha, beta, depth) //返回值表示对当前走步
   的玩家的有利程度
2.     if depth=0
3.         return 估价函数评估结果(对当前走步的玩家的有利程度)
4.     moves <- 生成当前局面全部走法
5.     value <- 0
6.     遍历 moves 中的每一个走步 move
7.         执行 move
8.         value <- -alphaBetaSearch(-beta, -
   alpha, depth - 1);   // 递归调用,对下一步走步玩家的有利程度的相反值
```

```
    (也就是不利程度)就是对当前走步玩家的有利程度 递归调用时上下界取相反数并互换
9.          撤销 move
10.         if value >= beta    // 得到一个大于或等于上界的值，终止对当前节点的搜索
11.             return beta//返回之前搜到的分数(之前搜到的分数是当前搜索的上界
12.         if value > alpha  // 发现比之前得到的下界大
13.             alpha <- vl//更新下界
14.             if depth=MAXDEPTH   // 如果回到根节点，需要记录最佳走法
15.                 记录最佳走法
16.     return alpha    // 返回当前节点的最优值
```

通过上面的 alpha-beta 剪枝，可以有效地裁剪搜索树，避免无效的搜索，在规定的时间内可以搜更深的搜索层次，可能找到对 AI 程序更有利的局面。

5. 历史表排序

alpha-beta 搜索依赖于搜索每种走法的顺序。如果总是先去搜索最坏的着法，那么 Beta 截断就不会发生，因此该算法就如同极大极小搜索一样，效率非常低。如果程序总是能挑最好的着法来首先搜索，就能不断发生 Beta 截断，大大提高搜索效率。

根据相关参考资料，在国际象棋 AI 程序设计中，可以通过历史表这样的数据结构启发搜索顺序，提高剪枝效率。

历史表用于存储好的走法，可以看成一种 map(key-value 映射)的抽象数据结构。其 key 值表示走法，而 value 表示当前走法的权重。

如果一种走法在不同层很多结点都被搜索过，并且是好的走法，那么其在历史表中就会有更大的权重。根据相关参考资料，历史表是有效的，这是因为，在下棋中，出色的走法往往是关键几种少数的走法（比如已知的几个不错的走法，都是将红车移动到某一位置。如果遇到一个走法，也是将红车移动到这一位置，那么猜测这一走法也是个不错的走法）。根据以前的搜索信息可以维护出历史表，进而较早地找到这些走法，然后根据这些优秀的走法限制出上下界用于 alpha-beta 剪枝。

什么样的走法算是比较好的走法呢？本次实验中，我们选择了以下两类走法：

1、产生 Beta 截断的走法

2、不能产生 Beta 截断，但它是所有 PV 走法(也就是 Beta > vl > vlAlpha 的走法)中最好的走法。

历史表 historyTable[]是一个大小为 4096 的数组，遇到一个比较好的走法 mv 时，给历史表中的该走法增加一个 value 值：

historyTable[mv] = historyTable[mv] + value

对于 value 的取值，我们参照国际象棋的设计，这里选择为深度的平方：

value = depth * depth

获取历史表的方法：

```
1. Position.prototype.historyIndex = function(mv) {
2.   return ((this.squares[SRC(mv)] - 8) << 8) + DST(mv);
```

```
3. }
```

更新历史表的方法：

```
1. Search.prototype.setBestMove = function(mv, depth) {
2.   this.historyTable[this.pos.historyIndex(mv)] += depth * depth;
3. }
```

历史表排序优化后的 alpha-beta 搜索算法，会先生成所有的走步，然后将这些走步按照历史表当前权重进行排序。根据权重从大到小的顺序进行当前层的搜索。在搜索完当前节点后，更新历史表。

加入历史表后的 alpha-beta 搜索：

```
1. function alphaBetaSearch(down, up, depth) //返回值表示对当前走步的玩家的有利程度
2.     if depth=0
3.         return 估价函数评估结果(对当前走步的玩家的有利程度)
4.     moves <- 生成当前局面全部走法
5.     将 moves 按照历史表中走法的权重从大到小进行排序
6.     value <- 0
7.     按顺序遍历 moves 中的每一个走步 move
8.         执行 move
9.         value <- -alphaBetaSearch(-up, -down, depth - 1)   // 递归调用,对下一步走步玩家的有利程度的相反值(也就是不利程度)就是对当前走步玩家的有利程度 递归调用时上下界取相反数并互换
10.        撤销 move
11.        if value >= up   // 得到一个大于或等于上界的值，终止对当前节点的搜索
12.            return up//返回之前搜到的分数(之前搜到的分数是当前搜索的上界)
13.        if value > down  // 发现比之前得到的下界大
14.            down <- vl//更新下界
15.            if depth == MAXDEPTH   // 如果回到根节点，需要记录最佳走法
16.                记录最佳走法
17.    用当前节点找到的较好走法更新历史表
18.    return down   // 返回当前节点的最优值
```

6. 迭代加深搜索

如我们之前所说的，在搜索过程中，要指定一个固定的深度。深度设置的过小，无法充分利用电脑性能，不能搜索到较好的解；深度设置的过大，可能导致求解时间过长，用户需要等待很久，影响游戏体验。

同时，同一局游戏中使用一个固定的搜索深度是存在问题的。刚开局时走法较多，不需

要过深的搜索，而在进入残局后，棋子数较少，可行的走法也相对较少，这时搜索树的宽度较少，而应当搜索得更深。若限制搜索深度为固定值，则难以满足这样的动态需求。

所以我们不指定搜索深度，而是指定搜索时间。设置搜索时间为 1000 毫秒，从深度 depth=1 开始搜索，如果没有超出时间限制，则进入更深一层的搜索。伪代码如下：

```
1. for i <-1 to limit_depth
2.  alphaBetaSearch(-∞, +∞, i) // 搜索深度为 i
3.  if 超过搜索时间
4.          break
```

迭代加深不仅充分利用了时间资源，还能充分发挥历史表的作用：前一层搜索结束后，将在历史表中积累大量数据，这将有效减少更深一层的搜索时间。

7. 检查重复局面 & Zobrist 校验码

为实现单方长将判负和双方长将作和的规则，需要检查过去的几个局面是否重复。判断局面是否重复，可以通过比对每个位置棋子是否相同来判断，但这样耗时大。Zobrist 校验码则可以快速判断棋子是否重复：通过对每个局面的计算得到一个校验码，只要校验码是相等的，局面就是一样的。

zobrist 校验码通过为每一个棋子在棋盘每一个位置产生一个随机数来实现，整个局面的校验码就是棋盘上剩余棋子在棋盘相应位置随机数的异或运算。

在程序中，数组 PreGen_zobristKeyTable[]存储这些随机数。

注意，由于对于两个局面，如果棋子种类及数量完全一样但走棋方不同，这两个局面是不一样的，所以程序中也为走棋方设立一个随机数 PreGen_zobristKeyPlayer，并将其异或到局面的效验值上去。

从 FEN 串始化棋局时，如果 FEN 串是红方走棋，那么局面校验码不用异或 PreGen_zobristKeyPlayer；如果是黑方走棋，局面校验那需要异或 PreGen_zobristKeyPlayer。此后只要切换走棋方，都会异或 PreGen_zobristKeyPlayer。

8. 项目框架

核心代码由 3 个文件组成：chessboard.js、position.js、search.js

前端 web 设计由 2 个文件组成：index.css、index.html

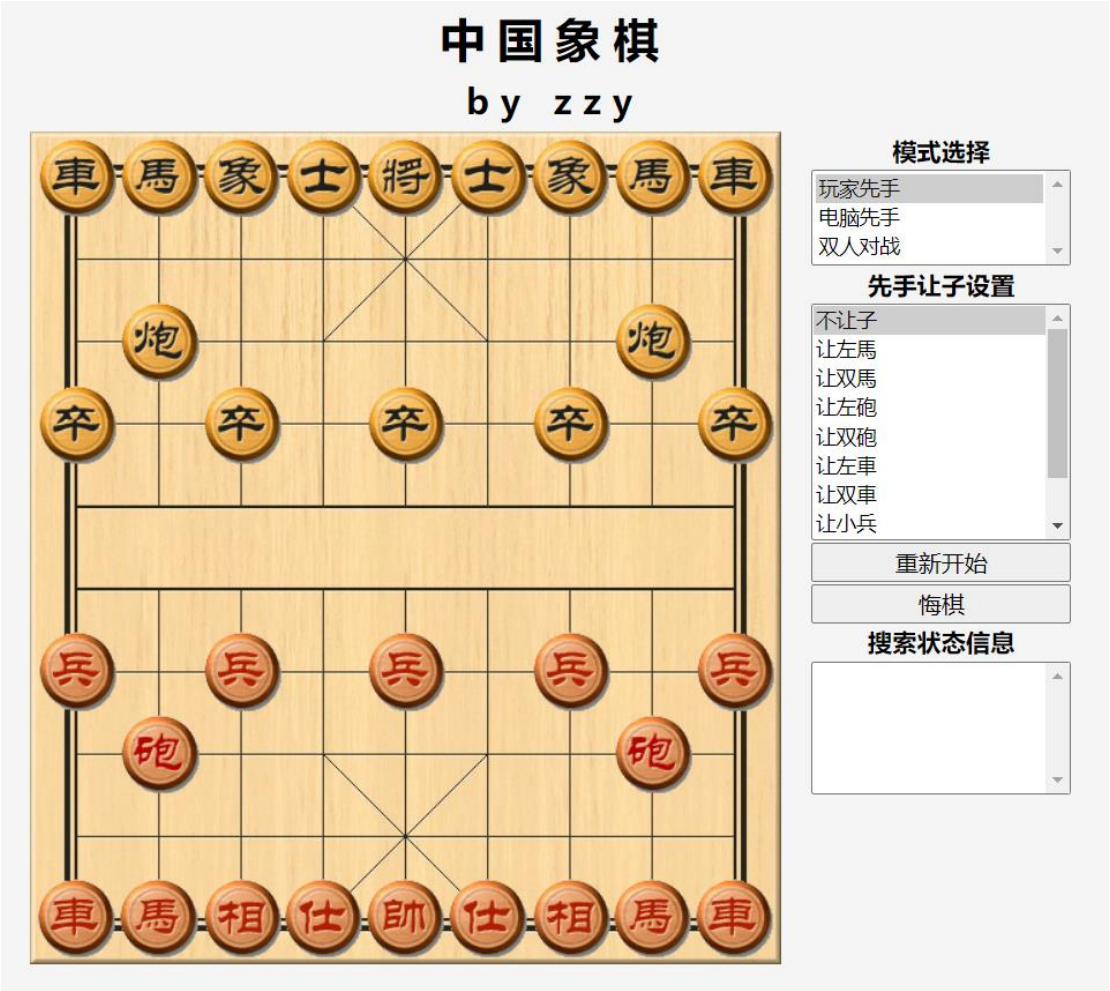
chessboard 表示一个棋盘，主要功能是初始化棋局，显示棋盘、棋子，响应棋盘上的点击事件。position 存储了一维棋局数组，并定义了很多对该数组进行操作的方法，包括：生成走法、校验走法的合法性、判断特定局面等。search 中实现了对博弈树的搜索。

index.html 中通过实例化一个 chessboard 对象，来调用各种函数进行响应。

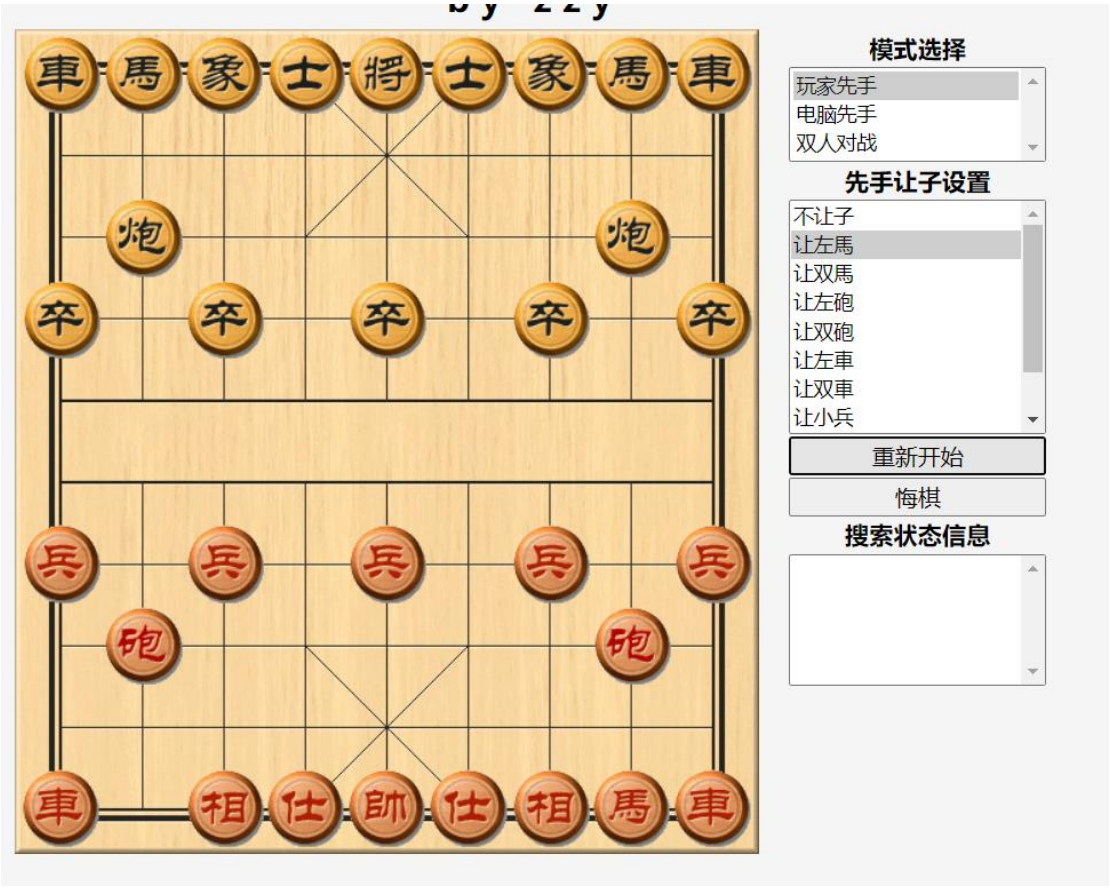
结果分析

1. 网页显示

本次实验结果为 web 形式，双击 index.html 即可查看结果（无需编译）
打开网页后，可以看到初始界面如下：



棋盘和棋子的图片使用的是 JavaScript 中国象棋程序（1） - 界面设计 - royhoo - 博客园 (cnblogs.com)中提供的图片。
游戏可以选择模式：玩家先手、电脑先手、双人对战。可以选择先手让子设置：不让子、让左马、让双马、让左炮、让双炮、让左车、让双车、让小兵、让左半、让九子
黑方为电脑方，红方为我方。
若要更换模式或者让子设置，则选好后点击重新开始。
例如，选择让左马：



可以看到，先手是我方，让对方左马，则左马棋子消失。
游戏还设置了悔棋功能，即可撤销上步走法，重新再走。

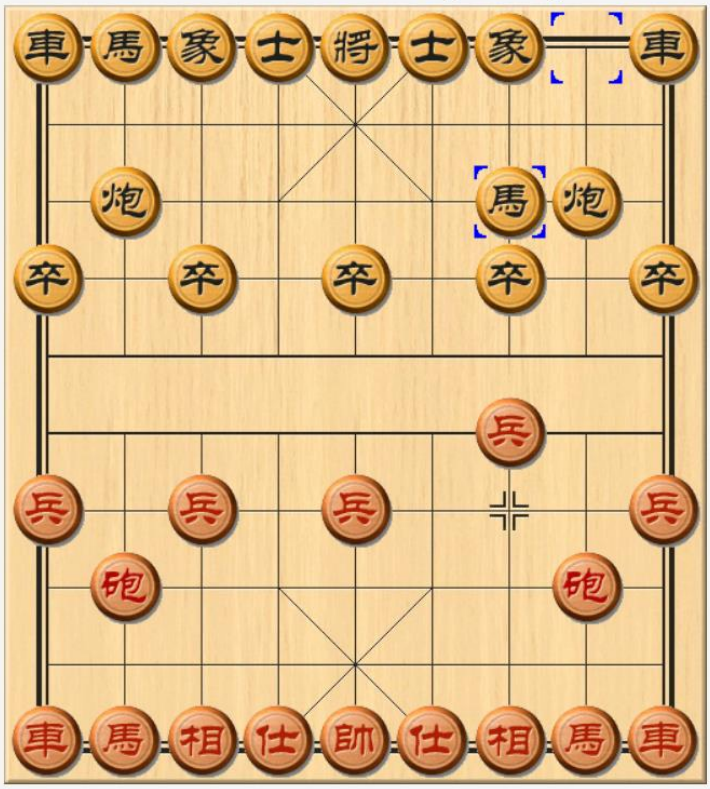
可以看到，右侧界面中还设有搜索状态信息，这里将显示电脑每一步走法时的搜索情况，实时更新，显示信息包括：搜索深度、搜索点数、优势估计。

2. 测试

一开始电脑响应慢，且一开始的优势估计比较小，在 0 附近波动：

中国象棋

by zzy



模式选择

玩家先手
电脑先手
双人对战

先手让子设置

不让子
让左馬
让双馬
让左砲
让双砲
让左車
让双車
让小兵

重新开始

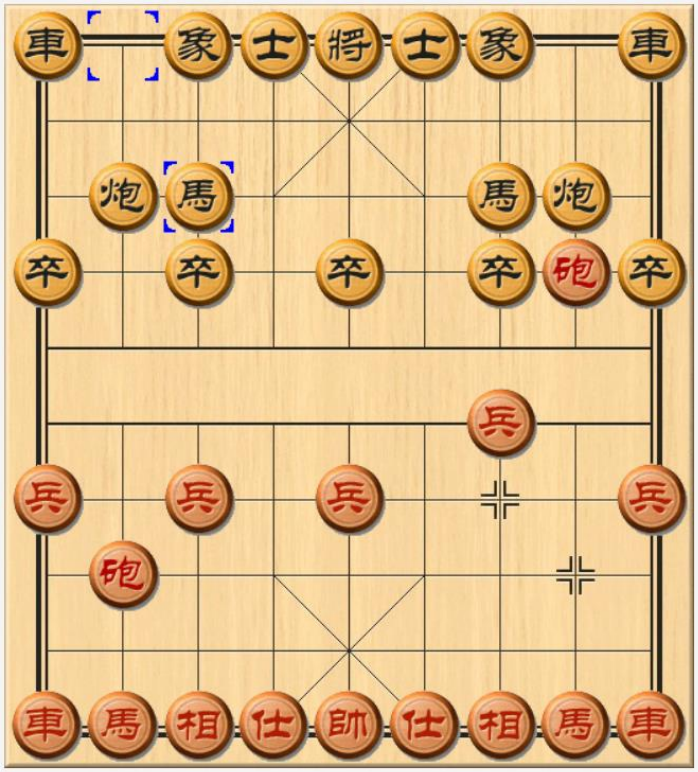
悔棋

搜索状态信息

搜索深度：6
搜索点数：684239
优势估计：-2

中国象棋

by zzy



模式选择

玩家先手
电脑先手
双人对战

先手让子设置

不让子
让左馬
让双馬
让左砲
让双砲
让左車
让双車
让小兵

重新开始

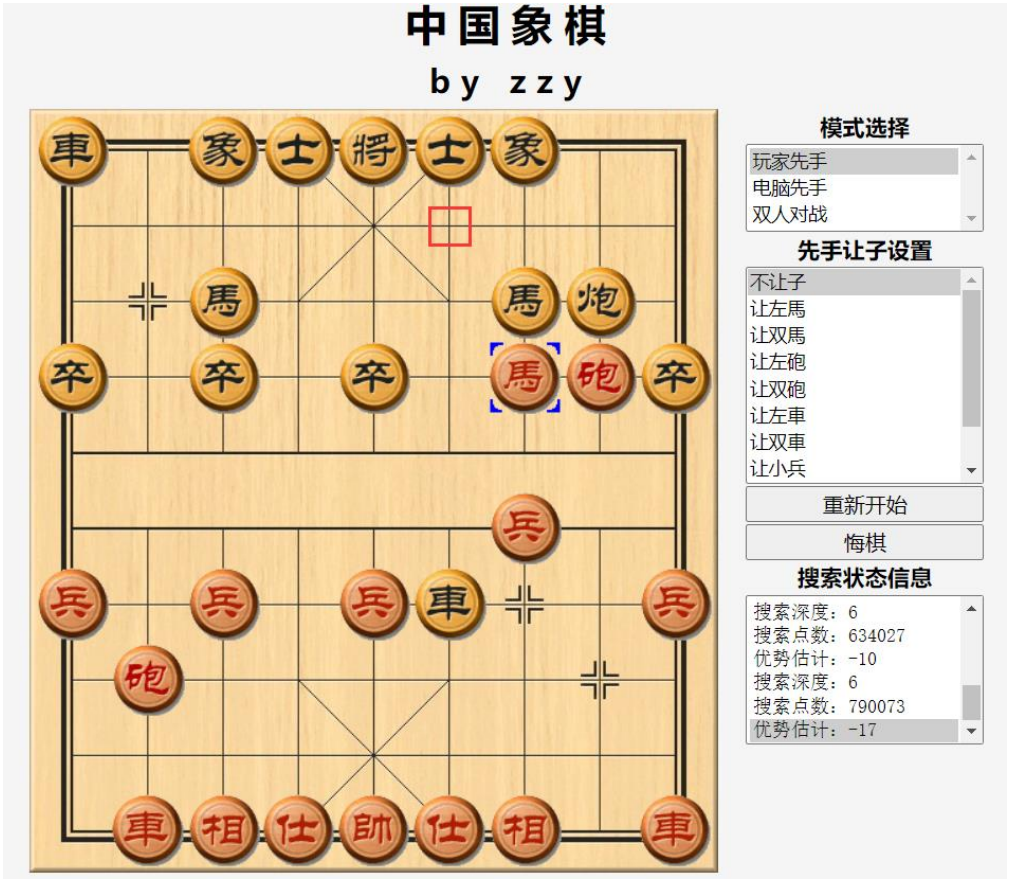
悔棋

搜索状态信息

搜索深度：6
搜索点数：684239
优势估计：-2
搜索深度：6
搜索点数：759697
优势估计：3

当玩家出现失误时，优势估计会大大增长：

下面测试一下“蹩马腿”：



此时，马无法走到红框选中的位置，因为马走的时候，是先向上走一步，再向左上走一步。而此时，上方有棋子挡住了，所以无法走。
当我方要输时，优势估计将变得无限大：

输时，将弹出“请再接再厉！”：

3. 性能

由于本次实验的特殊性，无法测试比较多的次数，下棋是一个过程，无法将全部结果都量化，只能通过时间和搜索信息进行一些分析。下面的结论来自于我在自己的普通笔记本电脑上，使用 Microsoft Edge 浏览器测试十次左右的效果。

电脑响应速度：

开局的时候电脑响应较慢，平均 1、2 秒给出响应。这是由于开局的时候可能的下棋方式较多，所以电脑几乎每次都是耗尽了规定的搜索时间来完成走步。而到了后期，尤其是胜负已经快分出来的时候，此时电脑速度变快。

上网查找了别的一些在线下象棋的网站，发现自己的程序下棋速度还是有待提高的。可能是一开始的时候电脑不需要一定找到最优解，只需找到较好的解即可，也无需预判之后的很多步。所以可以考虑把搜索限制时间按照优势估计来进行调整（因为优势估计值就反映了胜负趋势，当值在 0 附近时，胜负难见分晓；当优势估计值很大或很小时，胜负趋势明确）：当优势估计值在 0 附近时，搜索限制时间设置的较小；当优势估计值不在 0 附近时，搜索限制时间设置的较大。

搜索深度 & 搜索节点：

开局的时候，电脑一般搜索 6 层左右，搜索节点数在 600000-700000 之间浮动；到了后

期，电脑一般搜索 4 层左右，搜索节点数在 10000 左右。若快分出胜负了，那么搜索层数和搜索节点数将变得较小。

下棋水平：

（这里对于下棋水平的检测不是专业的，找来的测试者的水平无法给出一个标准的测量方式，只是个人主观判断棋力强弱。因此下述结论可能带有较强的主观判断，不是很准确）

这里我找了 3 个不同象棋水平的人来体验。对于象棋水平较差和中等的人来说，本程序可以轻松战胜他们。对于象棋水平较强的人来说，程序有大概率可以战胜他们。

结论

遇到的问题及解决：

1. 象棋规则
   开始本次实验的第一个问题就是我不会下象棋。先进行了象棋的学习，再开始的本次实验。不过个人象棋水平还是很低，在测试程序的时候，有些阻碍。
2. 搜索时间的设置
   对搜索时间的设置，我尝试了多种数值。一开始采用的是参考的博客的 4000ms，但是测试的时候发现问题就是，电脑响应过慢，影响到了游戏体验，而且会由于电脑搜索过程较深，可能下出来的棋子让人难以看出意图（可能原因是此时电脑棋力过高，高于普通水平太多，所以影响到了普通玩家的体验）。
   也有尝试过时间更短的，例如 500ms 等，但是考虑到为了让人工智能显得更智能（棋力相比普通玩家稍高些），最终调整到了 1000ms。
3. 估价函数的选择
   开始的时候对这部分内容比较迷茫，个人感觉估价函数应该考虑到棋子的重要性、棋子的攻击力（由于棋子走法不同，所以认为攻击力有差别）、局面情况、棋子之间的相对位置等。
   最终选择了多个博客都选择了的 ElephantEye 的估价函数。

不足之处：

本程序在算法和时间复杂度方面还是有很多不足之处，可以考虑从以下方面改进：

1.设置多种棋力水平供选择
   可以在电脑的棋力上做更多改进，例如，设置棋力水平供玩家选择，可以选择高、中、低等。这样玩家可以根据自身棋力水平或者游戏的个人乐趣所求，来选择。

2.充分利用玩家思考时间。
   本程序中，电脑在玩家行动后才会进行搜索，而玩家思考的过程中，电脑就空闲了。而在真人的对决中，对方在思考的时候，己方也会在思考，思考对方可能会下哪一步，下了的后的对策。让电脑利用玩家思考时间搜索，一是更符合真人对决的情况，二是等到玩家落子时响应时的搜索就可以减少一些计算量，总体上也可以搜索更多结点和更深深度，以达到减少响应时间和提高棋力的效果。

3.考虑并行计算。
   现在的计算机一般也有多个 CPU 核，目前实现的算法还只利用了一个线程来搜索，最

多只能利用到一个 CPU 核的性能。可以考虑设计并行算法利用到多个 CPU 核并行搜索，以提高下棋的性能。

4.存在水平线效应问题。

本程序中，叶子节点都是直接调用估价函数返回估价值。但在叶子节点之下，局面可能产生剧烈动荡，但仅仅使用子力价值作为评估函数返回值并不能很好的反映局面的真实动态情况。例如，叶子节点是一个吃子走法，它的估价值应当较高，但如果是一个换子走法（即下一步对手又吃回来），那这叶子节点可能是一个平手的走法(甚至被吃掉后是个更差的走法)。这种现象称为水平线效应。

根据相关参考资料，克服水平线效应，可采取空步裁剪、静态搜索等方法实现。

5.优化 alpha-beta 搜索

alpha-beta 搜索还可以继续优化，例如，可以采用主要变例搜索(VPS)以提高搜索性能。

个人心得：

这次实验中，我从朴素的极大值极小值搜索开始一步步优化搜索的过程，先是将极大值极小值搜索合并成了负极大值搜索，化简了代码，后面加上了 alpha-beta 剪枝，迭代加深搜索，历史表排序，Zobrist 校验码检查重复状态等优化。

在本次实验中我更进一步理解了博弈搜索问题、极大值极小值搜索、alpha-beta 剪枝原理。同时也通过本次实验的具体问题，对理论进行了实践，了解到相关估价函数的设计，也明白了这样设计的优缺点。之后根据相关参考资料，也了解到后续可以改进的方向，对如何有效地解决博弈搜索问题也有了更深入的认识。

主要参考文献

象棋巫师 - 象棋百科全书 (xqbase.com)

www.cnblogs.com/royhoo/p/6426394.html

https://m.xiangqi.okinfo.org/