几种智能算法在黑白棋程序中的应用*

柏爱俊
0511@USTC

2007 年 10 月

目录

1 引言 **2**

2 博弈树搜索 **3**

2.1 评估函数 4
2.2 极小极大搜索 4
2.3 Alpha-Beta 搜索 6
2.4 结合历史表和置换表的 Alpha-Beta 搜索 9
2.5 试验结果 10

3 强化学习 **12**

3.1 基本概念和标准模型 12
3.2 常用的学习算法 13
3.2.1 动态规划 13
3.2.2 蒙特卡罗方法 13
3.2.3 λ-时序差分 14
3.3 在黑白棋中的应用 15
3.4 试验结果 17

4 结论与展望 **17**

4.1 本文主要工作 17
4.2 存在的问题 18

* 中国科学技术大学大学生研究计划结题报告

插图

1 开局时黑棋的一步着子 . . . 3
2 极小极大博弈树 . . . 4
3 NegaMax 表示的极小极大博弈树 . . . 6
4 Alpha-Beta 搜索博弈树 . . . 7
5 强化学习的标准模型 . . . 12
6 主体与环境交互的过程 . . . 12
7 学习曲线 . . . 17
8 开局时黑方判断自己赢的概率的变化过程 . . . 18

List of Algorithms

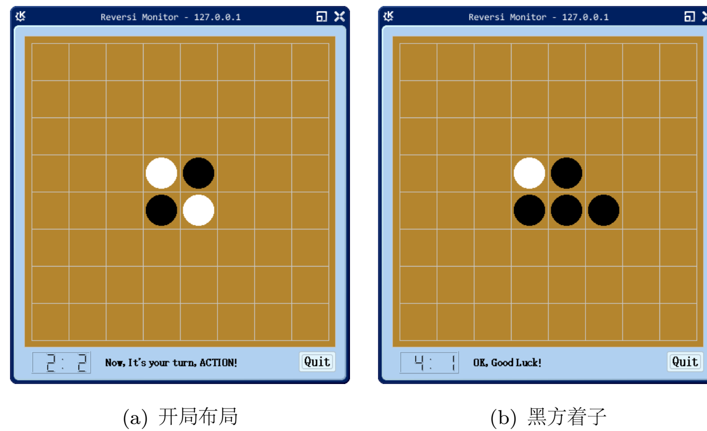
1 MinMax . . . 5
2 Max . . . 5
3 Min . . . 6
4 NegaMax . . . 7
5 AlphaBeta . . . 8
6 AlphaBeta(with history) . . . 9
7 Dynamic Programming . . . 14
8 Monte Carlo . . . 15
9 λ Temporal Difference . . . 16
10 AlphaBeta(with hash, history) . . . 20
11 Reversi Learning . . . 21
12 ChooseAction . . . 21

1 引言

暑假期间，我有幸在陈老师的实验室参加了大学生研究计划，受益颇多。在球队方面，我主要负责 2D 仿真机器人主动视觉模块的工作，不过这部分由于工作理论性不是很强，可写的东西比较少，而不是很适合作为报告的题目。自己在做球队工作的同时，出于学习的需要，尝试了几种人工智能的算法在黑白棋程序中的应用，所以就决定介绍一下这方面的内容。

黑白棋（Reversi、 Othello），也称翻转棋，是一个经典的策略性游戏。它使用 8×8 的棋盘，由双方轮流下棋，棋子分正反两面，分别为黑色和白色。下棋过程中如果一方的棋子把对方的棋子“夹住”了，就可以把被夹住的棋子翻转过来，成为自己的棋子。一步合法的着子，至少翻转一枚对方的棋子（图 1是开局时黑棋的一种走法）。如果某一方无子可下，就要停步一次，让对方先走。当双方都无子可下时，比赛结束，子多方为胜方。

受机器人足球比赛平台的启发，我采用了 C/S 的结构，一个 Server 可以下面连接几个 Client，这些 Client 可以是用于实际下棋的 Agent，也可以是用于观看比赛的 Monitor，Agent 可以是一个下棋



(a) 开局布局 (b) 黑方着子

图 1: 开局时黑棋的一步着子

的程序，也可以是一个与人交互的 Monitor。利用这个平台可以实现人与人比赛，机器与人比赛以及机器与机器比赛。附件包括了整个平台的程序，代码和文档。下面介绍一下下棋程序中主要用到的算法。

下棋的经历告诉我们，当下棋过程中的某一步有好几种走法可供选择时，棋手就要作出决策，选择对自己最有利的走法，那么如何判断是否有利呢？最简单的办法就是找一个的评估函数，这个函数对每一种局面给出一个估值，估值越高表明对自己越有利。用 $V(S)$ 表示对状态 S 的评估，如果评估函数足够好，能充分反应当前状态对自己的利弊，那么显然状态 S 下，应该采取的最佳行动

$$A(S) \;=\; \arg\max_a V(S'), \qquad S'\text{为 } S \text{ 下执行行动 } a \text{ 所到达的下一个状态} \tag{1}$$

不过这样的函数肯定具有非常复杂的形式，不太可能解析表示出来。为了提高程序的棋力，又要弥补这一不足，一般有两种解决方案：

1. 采用多步搜索，在搜索过程的终止状态再调用评估函数；
2. 利用机器学习的方法学习一个较好的评估函数。

首先介绍基于搜索的算法。

2 博弈树搜索

下棋是一个典型的双 agent（智能体，又称主体）、信息完全、零和博弈过程 [1, pages 119]，每个博弈者完全熟悉环境以及自己和对方所有可能的移动方式及影响。博弈者的策略搜索过程可以用树结构表示，称之为博弈树，树上的节点为棋盘可能出现的状态，父节点通过一步行动派生出的所有后继节点为其子节点。为了从众多可供选择的行动方案中选出一个对自己最为有利的行动方案，就需要对当前的情况以及将要发生的情况进行分析，通过某搜索算法从中选出最优的走步。在博弈问题中，每一个格局可供选择的行动方案都有很多，因此会生成十分庞大的博弈树，试图通过直到终局的搜索而得到最好的一步棋是不可能的，因此只能往前搜索有限的步数，最基本的搜索策略称为极小极大搜索 (MinMax Search)。

2.1 评估函数

一个简化的版本中，主要采用了基于棋格表和行动力（mobility）的估值。棋格表其实就是棋盘上不同位置的权值表，一方的行动力是其可以着子的位置的总数。如果用 S 表示棋盘的状态（状态可定义为每个格子的棋子有无和颜色），那么基于以上模型可知

$$V(S) = \sum_{i=1}^{8}\sum_{j=1}^{8}\omega[i,j]\cdot s[i,j] + \mathrm{Mobility}(my_color) \tag{2}$$

其中：

$$s[i,j] = \begin{cases} 1 & color[i,j] = my_color \\ -1 & color[i,j] = opp_color \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

$color[i,j]$ 表示棋盘上第 i 行第 j 列（简记为 (i,j)）处的颜色，可以是己方颜色（my_color）、对方颜色（opp_color）或空白。$\omega[i,j]$ 即为 (i,j) 处的权值，是经验数据。Mobility(my_color) 是己方的行动力大小。

2.2 极小极大搜索

我们假设参与博弈的两个人为 $Black$ 和 $White$，从 $Black$ 的角度看，$Black$ 每一步行动总是争取自己的最大收益，称为 Max 过程，对应博弈树上的节点称为 Max 节点；由于 $Black$ 对 $White$ 的棋力并不清楚，所以只能认为 $White$ 的每一步总是使自己的收益最小，称为 Min 过程，对应博弈树上的节点称为 Min 节点；于是下棋其实就是 Max 和 Min 交替进行的过程（当然，一方停步时例外）。考虑前面的单步搜索模型，我们假设此时轮到 $Black$ 下，$Black$ 发现自己所有可能的行动派生的子节点中评估值最大的节点为 S_i'，那么最大收益就为 $V(S_i') - V(S)$，对应的行动 a_i 就是单步搜索所能找到的最优解。但如果 $Black$ 向前搜索两步呢？他也许会发现 S_i' 所派生的子节点中的最小值为 $V(S_i'')$ 小于某个其他节点所派生的子节点中的最小值 $V(S_j'')$，不妨设 S_j' 为同层节点中所派生的子节点的最小值中最大的节点，那么他转而应该选择 S_j' 对应的行动 a_j 作为自己的最佳行动，因为选择 a_j 的收益最差不会少于 $V(S_j'') - V(S)$，而这已经是最好的情况了。如果搜索继续加深，那么选择会越来越“明智”。

一个实际的例子参见图 2 [2]。

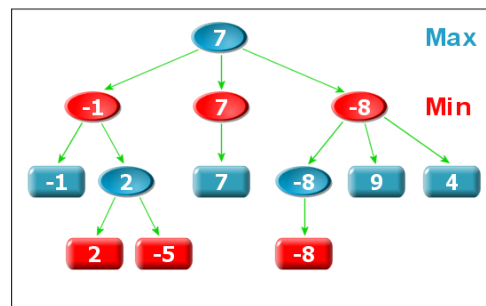


图 2: 极小极大博弈树

可以看出 [3]:

1. 自己可以下的节点为 Max 节点，对方可以下的为 Min 节点；
2. 叶子节点调用评估函数，并将其值向上传播；
   (a) Max 节点被赋值成其所有子节点中的最大值；
   (b) Min 节点被赋值成其所有子节点中的最小值；
3. 根节点在其子节点中选择值最大的节点所对应的动作作为最佳行动。

算法的实现类似于树的深度优先遍历，Max （Min）节点取后继节点中的最大（小）值作为自己的估值返回给父结点，用伪码表示为请见算法 1、2、3。

```
  Input: board board and max search depth depth
  Output: best move best_move
1 best_val ← −∞;
2 foreach move do
3 |   MakeMove(board, move, my_color);
4 |   val ← Min(board, depth − 1);
5 |   UnMakeMove(board, move, my_color);
6 |   if val > best_val then
7 |   |   best_val ← val;
8 |   |   best_move ← move;
9 return best_move;
```

算法 1 MinMax

```
   Input: board board and remain depth depth
   Output: maximum of its children notes
 1 max ← −∞;
 2 if depth ≤ 0 then
 3 |   return Evaluate(board);
 4 if !CanMove(board, my_color) then
 5 |   if !CanMove(board, opp_color) then
 6 |   |   ▷ Game Over
 7 |   |   return Evaluate(board);
 8 |   ▷ I stop for one time, opponent will take it
 9 |   return Min(board, depth);
10 foreach move do
11 |   MakeMove(board, move, my_color);
12 |   val ← Min(board, depth − 1);
13 |   UnMakeMove(board, move, my_color);
14 |   max = max(val, max);
15 return max;
```

算法 2 Max

```
Input: board board and remain depth depth
Output: minimum of its children notes
1  min ← ∞;
2  if depth ≤ 0 then
3  |  return Evaluate(board);
4  if !CanMove(board, opp_color) then
5  |  if !CanMove(board, my_color) then
6  |  |  return Evaluate(board);
7  |  return Max(board, depth);
8  foreach move do
9  |  MakeMove(board, move, opp_color);
10 |  val ← Max(board, depth − 1);
11 |  UnMakeMove(board, move, opp_color);
12 |  min = min(val, min);
13 return min;
```

算法 3 Min

代码中的 Evaluate 函数就是前面定义的 $V(S)$。

为了便于理解，把 Min，Max 函数分开了写，但也有更好的实现方法，这就是负极大搜索（NegaMax），见算法 4。图 3是图 2的 NegaMax 版本 [2]。

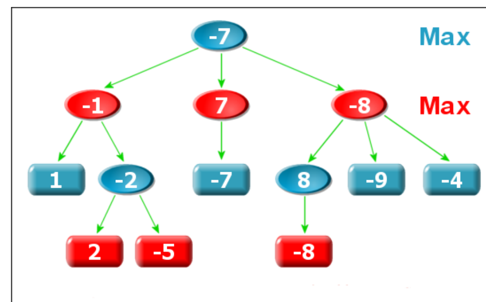


图 3: NegaMax 表示的极小极大博弈树

对于一般的极小极大搜索，即使每一步只有很少的下法，搜索位置的数目也会随着搜索深度呈指数级增长。在大多数的中局棋形中，平均每一步都有十种走法，如果向前搜索的深度为九步，那么为了进行一步着子，访问的棋局数就达到了十亿个，一般的电脑是很难胜任的。是否有办法在不减小搜索深度的前提下将需要搜索的节点减小一些？

事实证明，在搜索了某些节点后就可以不用搜索同层的其他节点了。这就是下面要介绍的 Alpha-Beta 搜索。

2.3 Alpha-Beta 搜索

为了得出一般性的结论，我们先看一个具体的例子。请考虑图 4中的博弈树（注意这里没有表示成 NegaMax） [2]。 我们假设按照重左到右的顺序搜索同层节点。A 是 *Max* 节点，它将被赋值成 B、

```
  Input: board board, side to move color and remain depth depth
  Output: maximum of its children notes
1 max ← −∞;
2 sign[my_color] = 1;
3 sign[opp_color] = −1;
4 if depth ≤ 0 then
5 |  return sign[color] · Evaluate(board);
6 if !CanMove(board, color) then
7 |  if !CanMove(board, my_color + opp_color − color) then
8 |  |  return sign[color] · Evaluate(board);
9 |  return -NegaMax(board, my_color + opp_color − color, depth);
10 foreach move do
11 |  MakeMove(board, move, color);
12 |  val ← -NegaMax(board, my_color + opp_color − color, depth − 1);
13 |  UnMakeMove(board, move, color);
14 |  max = max(val, max);
15 return max;
```

算法 4 NegaMax

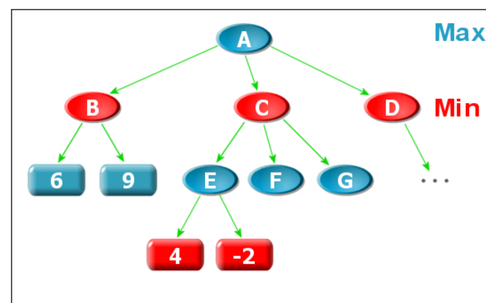


图 4: Alpha-Beta 搜索博弈树

C、D 中的最大值。为了知道 A 的值，首先需要知道 B 的值，为此我们搜索 B 的子节点，并将 B 赋值为 6，因为 B 是 *Min* 节点。然后开始决定 C 的值，我们首先搜索 E，E 的两个子节点的值分别为 4 和 -2，E 是一个 *Max* 节点，所以 E 的值为 4。因为 C 是 *Min* 节点，所以我们知道 C 的值肯定小于或等于 4，所以 C 不会被 A 选中，因为 A 是 *Max* 节点，而现在已经有一个子节点 B 的值大于 4。所以就没有必要在搜索 F 和 G 了。也即发生了剪枝。

算法实现时为每个节点引入两个值：α 和 β，分别表示该节点估值允许的下限和上限，两者构成了一个区间，这个区间被称为 α-β 窗口，窗口的长度定义为 $\beta - \alpha - 1$。初始时，根结点的 $\alpha = -\infty$，$\beta = \infty$，其他节点的 α，β 值是深度优先搜索过程中从它的父节点继承过来的。对上面的例子，当搜索完 B 之后，我们知道 A 最终的估值肯定不会小于 B 的值（因为 A 是 *Max* 节点），这个下限就是 A 的 α 值，$\alpha = 6$。继续进行深度优先搜索，当搜索完 E 之后，C 的上限也就确定了（C 是 *Min* 节点），此时 C 的 β 值是 4，注意到 C 的 α 也是 6（这是从 A 那里继承的），由前面的讨论知，此时 C 发生了剪枝，事实上，当 $\beta \le \alpha$ 时，就会发生剪枝。

我们看到，α 和 β 都是在深度优先搜索过程中得到更新的，对于 *Max* 节点，如果子节点的返回

值大于 α，就把 α 更新成该返回值。对于 Min 节点，过程恰好相反，β 被更新成比它更小的子节点的返回值。Max 节点的 α 值不会减小，Min 节点的 β 值不会增加，当某个结点的 $\beta \leq \alpha$ 时，就会发生剪枝。Max 节点发生剪枝的叫 β 剪枝，Min 节点发生的叫 α 剪枝。根据子节点的返回值同 α，β 的关系，可以把子节点分为三类（以 Max 过程为例，S 代表对应的 Max 节点，S' 代表某个子节点）：

1. **非常差的节点** $V(S') \leq \alpha$，S' 肯定不会被选中，需要继续搜索下一个子节点；
2. **非常好的节点** $V(S') \geq \beta$，发生了 β 剪枝；
3. **主要步骤节点** $\alpha < V(S') < \beta$，不能发生剪枝，但有可能被选中，还需要继续搜索。

Alpha-Beta 搜索的伪码参见算法 5。

```
Input: board board, alpha α, beta β, side to move color and remain depth depth
Output: useful maximum of its children notes
1  max ← −∞;
2  sign[my_color] = 1;
3  sign[opp_color] = −1;
4  if depth ≤ 0 then
5  |   return sign[color] · Evaluate(board);
6  if !CanMove(board, color) then
7  |   if !CanMove(board, my_color + opp_color − color) then
8  |   |   return sign[color] · Evaluate(board);
9  |   return -AlphaBeta(board, −β, −α, my_color + opp_color − color, depth);
10 foreach move do
11 |   MakeMove(board, move, color);
12 |   val ←-AlphaBeta(board, −β, −α, my_color + opp_color − color, depth − 1);
13 |   UnMakeMove(board, move, color);
14 |   if val > α then
15 |   |   if val ≥ β then
16 |   |   |   return val;
17 |   |   α = max(val, α);
18 |   max = max(val, max);
19 return max;
```

算法 5 AlphaBeta

观察 Alpha-Beta 搜索的算法会发现，剪枝的多少很大程度上依赖于搜索子节点的顺序，如果 Max 节点的子节点按其值从大到小有序，Min 节点的子节点按其值从小到大有序，那么剪枝会最早发生，从而使要访问的节点数大幅减少。但完全有序是不可能实现的，因为不可能在搜索子节点之前就知道父节点的值（否则就不需要搜索了）。但可以利用搜索的历史信息对子节点进行预排序，下一节介绍这一方法。

2.4 结合历史表和置换表的 Alpha-Beta 搜索

前面提到的 Alpha-Beta 搜索，虽然是对 MinMax 搜索的提高，但也只是很小的提高，因为每一层节点的顺序并没有特殊处理，从而浪费了较多的剪枝机会。我们注意到，每次搜索时，会遇到三类节点，虽然这样划分很粗糙，但稳定性却很好，就是说不太可能出现某一次搜索时某个节点被划分成**非常差的节点**，而下次搜索时遇到同样的节点却成了**非常好的节点**，所以我们可以引入历史表机制，记录这种划分，保证每次搜索时先搜索**非常好的节点**，再搜索**主要步骤节点**，最后搜索**非常差的节点**，这样发生剪枝的情况就大大增加了。

历史表定义为 $history[2,\ 65,\ 65]$，举例来说，$historye[Black,\ 15,\ 1]$ 表示 $Black$ 在下棋过程中的第 15 步期望最好的一步，$history[Black,\ 15,\ 2]$ 是期望次好的一步，依此类推。初始时，$history$ 被赋值成了一个经验的有序序列。

以 $Black$ 为例，用 $step$ 表示下棋过程的第几步（也即棋盘上已有的棋子数加一），其值从 5 变化到 64 。当前搜索深度为 $depth$ 时，对应的全局步数为 $step + depth$，如果某一步 $move$ 是**非常好的节点**，那么就用 $\text{FloatMove}(Black,\ step + depth,\ move)$ 将 $move$ 插入到本层子节点排序为第一的位置；如果某一步 $move$ 是**主要步骤节点**，那么就用 $\text{AdvantageMove}(Black,\ step + depth,\ move)$ 将 $move$ 和它前面的一个位置对换。这样简单处理后，便使同层节点大致有序了，从而提高了算法的效率。

伪码描述请看算法 6。(为节省排版空间略去了重复的部分)

```
Input: board board, alpha α, beta β, side to move color and remain depth depth
Output: useful maximum of its children notes
1  max ← −∞;
2  sign[my_color] = 1;
3  sign[opp_color] = −1;
4  ⋮
5  foreach move ∈ history do
6  |  MakeMove(board, move, color);
7  |  val ←-AlphaBeta(board, −β, −α, my_color + opp_color − color, depth − 1);
8  |  UnMakeMove(board, move, color);
9  |  if val > α then
10 |  |  if val ≥ β then
11 |  |  |  FloatMove(color, step + max_depth − depth, move);
12 |  |  |  return val;
13 |  |  AdvantageMove(color, step + max_depth − depth, move);
14 |  |  α = max(val, α);
15 |  max = max(val, max);
16 return max;
```

算法 6 AlphaBeta(with history)

另外我们还有一个历史信息没有利用好，那就是以前出现过的节点。事实上无论是一次搜索还是相邻的几次搜索之间，会出现很多的相同节点，利用好这些节点保留的信息，可以节省很多时间。

为此，我们引入置换表（Transposition Table），置换表里面记录了所遇到的节点的编码[1]和相关信

[1] 节点的编码定义为棋盘的 128 位二进制表示，前 64 位表示白子的分布，后 64 位表示黑子的分布，共占 4 个字节

息，这些信息包括了行动方（即节点对应的是 *Black* 走还是 *White* 走)、评估值、最佳行动等，这些都是历史信息。

对这些记录定义了两种匹配类型：

1. **完全匹配** 不仅节点的编码一样，剪枝情况[2]和行动方也一样；
2. **部分匹配** 节点的编码和行动方一样，但剪枝情况不同。

当搜索到某个节点时，发现置换表中已经有了这个节点的记录，如果与当前节点完全匹配，就可以直接返回表中保存的评估值，而不用再继续搜索下去；否则如果是部分匹配，虽然评估值没什么用，不过节点历史上的最佳行动确是很有价值的信息，可以列为本次搜索优先考虑的行动。利用置换表可以更进一步的提高算法的效率。

其实置换表实现成一个大的 hash 表，hash 函数构造为：

$$HashKey(S) \ = \ \sum_{i\ =\ 1}^{8}\sum_{j\ =\ 1}^{8} mask[\ s[i,\ j]+1,\ i,\ j] \tag{4}$$

其中 *mask* 是一个随机数表，为每个格子的每个状态准备了一个随机数。事实上由于搜索过程中棋盘总是渐变的，相邻状态的棋盘，变化的棋子数不是很多，所以可以用前一状态的 $hash_key$ 计算当前的 $hash_key$。为此，定义了一个数组 *flip*， *flip* 满足：

$$flip[old,\ new,\ i,\ j] \ = \ mask[new,\ i,\ j] \ - \ mask[old,\ i,\ j] \tag{5}$$

则，

$$HashKey(S) \ = \ HashKey(S') \ + \ \sum_{i\ =\ 1}^{8}\sum_{j\ =\ 1}^{8} flip[\ s[i,\ j]+1,\ s'[i,\ j]+1,\ i,\ j] \tag{6}$$

这样只要用一个全局变量 $hash_key$ 能随时跟踪 S 就行了。每次棋盘发生变化时，如位置 $(i,\ j)$ 从状态 *old* 变成了 *new*，更新一下 $hash_key$：

$$hash_key \ = \ hash_key \ + \ flip[old,\ new,\ i,\ j] \tag{7}$$

计算得出的 $hash_key$ 虽然会有冲突，但考虑到处理冲突要会有开销，而且解决冲突后得到的记录实时性不好，有可能是很久以前的记录，于搜索不益，所以就没有处理 $hash_key$ 的冲突。hash 表中只存储最新的记录，如果有冲突，新记录直接覆盖旧记录。

这部分的伪码请见算法 10。(为节省排版空间略去了重复的部分，附在文后)

2.5 试验结果

上面介绍的算法由简单到复杂，使搜索深度逐步加深，从而提高了程序的棋力。我进行了一些测试 [3]，下面是统计数据[4]。

表 1统计了不同算法最大深度时的一些数据。其中 max_depth 为可以承受（须要在一分钟内走出一步棋）的最大搜索深度， *score* 为比赛的结果， *nodes* 为最大深度时共搜索过的节点数，

[2]剪枝情况分三种，即没有发生剪枝（NO_BOUND)、发生 α 剪枝（UP_BOUND)、发生 β 剪枝（LOW_BOUND)

[3]测试环境是 Ubuntu Linux 2.6.22-12-386, AMD x86_64 2211MHz, 1011M RAM

[4]Hash 表示只使用置换表的 Alpha-Beta 搜索，History 表示只使用历史表的 Alpha-Beta 搜索，HashHistory 表示使用历史表和置换表的 Alpha-Beta 搜索

$total_time$ 为最大深度时一场比赛花费在搜索上的时间（单位：秒），$exact$ 为 hash 表完全匹配的结点数，$part$ 为 hash 表部分匹配的结点数。测试的算法搜索深度只设了一个初始值，不随比赛过程而变化，为了使测试结果较公正，每个算法测试时对手都选了目前最强的一个版本的程序（使用了历史表和置换表的 Alpha-Beta 搜索和动态的搜索深度）。

	max_depth	*score*	*nodes*	*total_time*	*exact*	*part*
MinMax	5	4:60	339,7402	106.867	0	0
AlphaBeta	8	12:51	908,3234	251.215	0	0
Hash	8	19:45	375,7319	97.786	8,3714	4,7596
History	9	11:53	1217,4733	353.825	0	0
HashHistory	10	23:41	1407,1178	371.428	60,1814	32,1421

表 1: 几种搜索算法的实验数据统计

表 2是对表 1的计算分析，从中可以看出它们之间的性能差异。其中 ray 是平均每个节点搜索的子节点数，$nodes = 30 \times \sum_{i=0}^{max_depth} ray^i$（乘以 30 是因为一次比赛中如果不考虑停步的影响共搜索了 30 次），可以根据 $ray \leftarrow [(nodes/30)(ray-1)+1]^{1/(max_depth+1)}$ 迭代求解，ray 越小说明剪枝效果越明显；$time_per_node$ 是平均每个节点所花费的时间（不包括递归搜索子节点的时间，单位：微秒），$time_per_node = toatl_time/nodes$，$time_per_node$ 反应了剪枝和置换表综合影响的效果，剪枝越多 $time_per_node$ 越小，置换表记录匹配越多 $time_per_node$ 也越小；$exact_rate$ 和 $part_rate$ 分别表示完全匹配和部分匹配的节点数占总共访问过节点数的比率，一般 ray 越小，可能匹配的节点数百分比会越大。

	ray	*time_per_node*	*exact_rate*	*part_rate*
MinMax	10.04	31.46	0.0%	0.0%
AlphaBeta	4.70	27.66	0.0%	0.0%
Hash	4.19	25.23	2.23%	1.27%
History	4.07	29.06	0.0%	0.0%
HashHistory	3.57	26.39	4.27%	2.28%

表 2: 几种搜索算法的性能比较

结合表 2，从平均每个节点搜索的子节点数角度，可以看出 AlphaBeta 搜索明显优于 MinMax 搜索，只使用了历史表和只使用了置换表的 AlphaBeta 搜索效果都有提高，其中使用了历史表的效果提高较大，而效果提高最好的是既使用历史表又使用置换表的 AlphaBeta 搜索。从平均每个节点花费的时间角度看，只使用置换表的搜索所花费最小，这时因为一些得到完全匹配的节点基本不花时间，得到部分匹配的节点有可能比较容易发生剪枝而使所花时间较少；使用了历史表的搜索所花时间比只使用置换表的搜索的花费多，说明历史表的维护成本较大；MinMax 搜索花费时间最大，这与它不发生剪枝是分不开的。综合考虑，既使用历史表又使用置换表的 AlphaBeta 搜索性能最好[5]。

[5]从表 2还可以看出，置换表匹配的节点数占总共访问过节点数的比率太小，这是下一步要努力的方向之一。

3 强化学习

前面介绍了基于搜索的方案，也取得了不错的效果。不过要想进一步提高，就很难了，主要是因为程序中用到的评估函数过于简单，不能很好地描述下棋过程中遇到的错综复杂的状态。于是，我决定利用机器学习的方法，学习到一个较客观的评估函数。我在实验室参加了强化学习的讨论组，所以自然想到用强化学习（Reinforcement Learning）的方法。

3.1 基本概念和标准模型

强化学习要解决的问题是这样的：一个能感知环境的自治主体，怎样通过学习选择能达到其目标的最优动作 [4, pages 263]。图 5是强化学习的标准模型，主体可以观察环境的状态（state），并能做出一组动作（action）改变环境的状态，主体每次执行一个动作之后会得到一个回报（reward），由回报函数给出，它对主体从不同状态选取的不同动作赋予一个数值，对那些对能完成目标的动作赋予正回报，其他动作赋予零回报或负回报。学习的目标是一个最优策略（policy），这个策略能够使主体从任何初始状态选择恰当的动作序列，使主体随时间的累积回报达到最大。 强化学习的主要思想是“与环

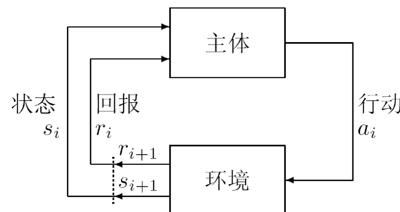


图 5: 强化学习的标准模型

境交互”和“试错” [7]。图 6是主体与环境交互的过程。

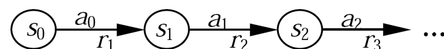


图 6: 主体与环境交互的过程

可以用马尔可夫决策过程（Markov decision process, MDP）定义强化学习的一般形式：主体可感知到其环境的不同状态集合 S，并且有它可执行的动作集合 A。在每个离散时间步 t，主体感知到当前状态 s_t，选择当前动作 a_t 并执行它。环境响应此主体，给出回报 $r_{t+1} = r(s_t,\ a_t)$，并产生一个后继状态 $s_{t+1} = \delta(s_t,\ a_t)$。这里的函数 r 和 δ 是环境的一部分，主体并不知道。在 MDP 中，$r(s_t,\ a_t)$ 和 $\delta(s_t,\ a_t)$ 只依赖于当前状态和动作，而与过去的状态和动作无关。状态转移函数 δ 依赖于状态转换模型 $T:\ S \times A \times S \rightarrow [0,\ 1]$，$T(s,\ a,\ s')$ 表示状态 s 下执行行动 a 到达状态 s' 的概率，满足 $\sum_{s'} T(s,\ a,\ s') = 1$，也就是说 $\delta(s_t,\ a_t)$ 是一个随机变量，服从某个由环境给出的分布 [6, page 13]。在后面的讨论中，为了简化问题，我假设 $\delta(s_t,\ a_t)$ 是确定的。

主体的任务是学习一个策略 $\pi:\ S \rightarrow A$，它基于当前观察到的状态 s_t 选择下一步动作 a_t，即 $\pi(s_t) = a_t$。定义由状态 s_t 开始的长远回报为：

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \ldots + r_T \tag{8}$$

其中，回报序列 r_{t+i} 是通过由状态 s_t 开始并重复使用策略 π 来选择动作执行生成的。对于有终任务（Episode task，即任务有终止状态），T 是从当前到任务完成所需要的时间，对于持续任务，$T = \infty$。如考虑有终任务目标状态下回报为 0，这有终任务和持续任务的长远回报可以统一写为 $R_t = \sum_{i=0}^{\infty} r_{t+i+1}$。考虑时间越远即时回报对当前影响越小，可以加上一个折扣因子 γ（$0 \leq \gamma \leq 1$），折扣长远回报为：

$$R_t = \sum_{i=0}^{\infty} \gamma^i r_{t+i+1} \tag{9}$$

我们可以把主体从状态 s 开始的长远回报作为策略 π 下状态 s 的评价：

$$\begin{aligned} V^\pi(s) &= R_t \\ &= \sum_{i=0}^{\infty} \gamma^i r_{t+i+1} \end{aligned} \tag{10}$$

V^π 称为策略 π 下的状态评估函数函数，也叫策略 π 的值函数。

若对于所有状态 s，有 $V^\pi(s) \geq V^{\pi'}(s)$，则称 $V^\pi \geq V^{\pi'}$。使 V^π 达到最大的策略 π 为最优策略，用 π^* 表示，π^* 的值函数记为 V^*。显然最优策略下：

$$\pi^*(s_t) = \arg\max_a V^*(s_{t+1}) \tag{11}$$

所以，只要学习到 V^* 就可以得到最优策略 π^*。

下面介绍几种常用的学习算法。

3.2 常用的学习算法

3.2.1 动态规划

我们希望用 V 通过学习过程逐步逼进 V^*，记 $V_1, V_2, \ldots, V_k, \ldots$ 为逼进序列。注意到贝尔曼迭代式（Bellman equation）[8]：

$$V^*(s_t) = r_{t+1} + \gamma V^*(s_{t+1}) \tag{12}$$

所以可以利用下面这个迭代式实现学习：

$$V_{k+1}(s) \leftarrow \max_a (r_{t+1} + \gamma V_k(s')), \quad s' = \delta(s, a) \tag{13}$$

容易用动态规划方法实现，见算法 7

3.2.2 蒙特卡罗方法

从算法 7可以看出，动态规划需要事先知道状态转移函数 δ 和回报函数 r，即需要一个环境的完整模型，很多问题事先很难给出环境的完整模型（如机器人足球，博弈问题），这时可以用蒙特卡罗（Monte Carlo）方法实现学习。

蒙特卡罗方法把交互过程中得到的即时回报的平均作为值函数的估计。蒙特卡罗方法仅适用于有终任务，当一个任务完成后，计算出此任务状态序列的每个状态 s 的长期回报 $R_{s,\ a}$，回报函数可以由环境给出也可以编制到程序内部，用多次执行任务的平均值更新值函数。伪码见算法 8。

其中 $visits[s]$ 记录了状态 s 在整个学习过程中被访问的次数，利用它实现对评价估计求平均。

```
1  foreach s do
2  |  V[s] ← 0;
3  repeat
4  |  s ← Initial( );
5  |  repeat
6  |  |  a ← arg max_a V[δ(s, a)];
7  |  |  Execute(a);
8  |  |  r ← Reward(s, a);
9  |  |  s′ ← Observe( );
10 |  |  V[s] ← r + γV[s′];
11 |  |  s ← s′;
12 |  until s = terminal;
13 until end;
```

算法 7 Dynamic Programming

3.2.3 λ-时序差分

蒙特卡罗方法在与环境的交互过程中学习，不需要完整的环境模型，但更新较慢，只有完成了任务，才能对值函数更新。有些问题的回报函数多数状态下为零，只有在终止状态才不为零（如机器人足球中射门了才有回报，黑白棋中比赛结束了才知道回报），称为稀有回报函数，这样的问题用蒙特卡罗方法学习效果依然不好。时序差分（Temporal Difference）结合了蒙特卡罗方法和动态规划的思想，和蒙特卡罗方法一样不需要完整的环境模型；同时也像动态规划一样，随时更新评价估计，而不需要等任务完成 [6]。一个简单的时序差分学习按下式进行迭代：

$$V(s_t) \leftarrow V(s_t) + \alpha [r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{14}$$

在更新评价时，不是让它直接等于即时回报和后续评价之和，而是引入学习率 α 逐步调整，可以设定为一个 0 到 1 之间的常数，也可以如下式 [6]：

$$\alpha = \frac{1}{1 + visits[s_t]} \tag{15}$$

在学习过程中根据状态 s 的访问次数增多而不断减小。

每步行动产生的影响不仅是当前状态，也可能影响以后的状态，虽然迭代公式考虑了这种影响，把即时回报和后续评价作为评价更新的依据，但这种更新对回报函数大多数为零的问题显得较慢。如果每次行动之后不仅对当前状态更新，也对之前的很多步更新，则可以较快得完成学习，为此引入一个和回溯有关的参数 λ 来调整这种影响，得到 λ-时序差分（TD(λ)）。

记时刻 t 的评价更新差值为：

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{16}$$

把迭代公式改写为:

$$V(s) \leftarrow V(s) + \alpha \delta_t e_t(s) \tag{17}$$

```
1  foreach s do
2  |  V[s] ← 0;
3  |  visits[s] ← 0;
4  repeat
5  |  s ← Initial( );
6  |  repeat
7  |  |  (S, A) ← GenerateEpisode(s, π);
8  |  |  foreach (s_t, a_t) ∈ (S, A) do
9  |  |  |  R ← Σ_{i≥0} γ^i r(s_{t+i+1}, a_{t+i+1});
10 |  |  |
11 |  |  |  R ← (V[s_t] × visits[s_t] + R) / (visits[s_t] + 1);
12 |  |  |
13 |  |  |  V[s_t] ← R;
14 |  |  |  visits[s_t] ← visits[s_t] + 1;
15 |  until s = terminal;
16 until end;
```

算法 8 Monte Carlo

上式的 $e_t(s)$ 称为回溯权重，离当前状态 s_t 越远的状态 s， $e_t(s)$ 应越小。 $e(s)$ 在每步行动后根据参数 λ 进行衰减，一种衰减方式为：

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) + 1 & s = s_t \\ \gamma\lambda e_{t-1}(s) & s \neq s_t \end{cases} \tag{18}$$

算法 9给出了 λ 时序差分算法。

3.3 在黑白棋中的应用

前面介绍了强化学习的基本思想和方法，但还不能直接应用到黑白棋问题中，因为黑白棋是一个多智能体系统（MAS），环境的改变是下棋双方共同作用的结果，而强化学习只能处理一个主体与环境交互的问题。为了应用强化学习，我们需要把对手看成环境的一部分，这样一来环境就是未知的了，所以就不能用动态规划方法，又因为黑白棋具有稀有回报函数，所以选择用 λ-时序差分实现学习算法。

学习的目标是这样一个函数 P，对于给定的状态 s， $P(s)$ 给出 s 的行动方从 s 开始到比赛结束赢的概率。状态编码成一个向量，包括 65 个分量， $s[0]$ 标志当前的行动方， $s[0] = 0.0$ 表示是白子走， $s[0] = 1.0$ 表示是黑子走。另外 64 个分量对应棋盘上的 64 个位置， $s[k] = 0.0$ 表示第 k 个位置上是白棋， $s[k] = 1.0$ 表示是黑棋， $s[k] = 0.5$ 表示没有棋子。 $V(s)$ 定义为[6]：

$$V(s) = \begin{cases} P(s) & s[0] = my_color \\ 1.0 - P(s) & s[0] = opp_color \end{cases} \tag{19}$$

[6]虽说 $V(s)$ 定义为状态 s 开始的长期回报，但对于回报函数不易确定的问题，也可以有其他定义，只要 $V(s)$ 能反映状态 s 的评价就行了

```
1  foreach s do
2  │ V[s] ← 0;
3  │ e[s] ← 0;
4  repeat
5  │ s ← Initial( );
6  │ S ← ∅;
7  │ repeat
8  │ │ a ← π(s);
9  │ │ Execute(a);
10 │ │ r ← Reward(s, a);
11 │ │ s′ ← Observe( );
12 │ │ S ← S ∪ s;
13 │ │ δ ← r + γV[s′] − V[s];
14 │ │ e[s] ← e[s] + 1;
15 │ │ foreach s ∈ S do
16 │ │ │ V[s] ← V[s] + αδe[s];
17 │ │ │ e[s] ← γλe[s];
18 │ │ s ← s′;
19 │ until s = terminal;
20 until end;
```

算法 9 λ Temporal Difference

$V(s)$ 表示状态 s 下，无论是自己走还是对手走，自己赢的概率，所以 $V(s)$ 可以作为对状态的评价，从而可以成为决策的依据。

对于比赛结束的前一个状态，$P(s)$ 的值是确定的，学习过程中（$\lambda=0,\ \gamma=1$）此值按下式向前传播：

$$P(s) = \begin{cases} P(s) + \alpha(P(s') - P(s)) & s[0]=s'[0] \\ P(s) + \alpha(1.0 - P(s') - P(s)) & s[0] \neq s'[0] \\ 1.0 & s' \text{ 表明 } s[0] \text{ 赢了} \\ 0.5 & s' \text{ 表明双方和局} \\ 0.0 & s' \text{ 表明 } s[0] \text{ 输了} \end{cases} \tag{20}$$

为了应用 λ-时序差分，还需要解决回溯权重 $e(s)$ 的表示问题，事实上不可能为每个状态记录一个回溯权重，因为状态空间太大（达 3^{64}）。程序中使用了一个较简单的模型，不要求为每个状态记录回溯权重，只把当前状态的回溯权重设为 1.0，一次任务中前面状态的回溯权重依据和当前状态的距离呈指数递减。

为了表示 $V(s)$，首先要表示 $P(s)$，对于状态空间较小的问题可以直接建立 $P(s)$ 的查找表，但对于状态空间较大的问题，往往由于建立查找表所需存储空间太大，而变得行不通。不过可以用神经网络替代查找表，借助神经网络的容错性和泛化能力可以取得很好的效果。

把当前状态 s 编码作为网络的输入，把每个 $P(s)$ 的更新值作为网络的训练值。程序中采用了 $65\times130\times1$ 的网络，利用反向传播算法实现训练。网络初始时所有权值和阈值都设为零。

基于 λ-时序差分学习过程的伪码参见算法 11（由于排版空间不够，附在文后）。

3.4 试验结果

学习程序中，选取 $\lambda = 0.7$， $\alpha = 0.1$， $\gamma = 1.0$，选取 $sigmoid$ 函数作为网络的阈值函数。由于学习的程序较快，让学习的程序跟其他程序比赛来学习的话，会因为对手较慢和平台较慢的原因而拖慢学习过程，所以我就选择让学习的程序自己跟自己下棋来进行学习（我还不能肯定这样是否可行）[7]。

定义一次比赛过程中的学习误差为这次比赛中所有网络训练值与实际输出差的平方和，学习误差关于学习次数的曲线可以看作学习曲线，反映了学习逐渐收敛的情况，图 7 是学习了 5000 场比赛的学习曲线。 曲线波动很大，这与误差的定义有关，误差定义时并没有涵盖全部状态的误差，而仅是计

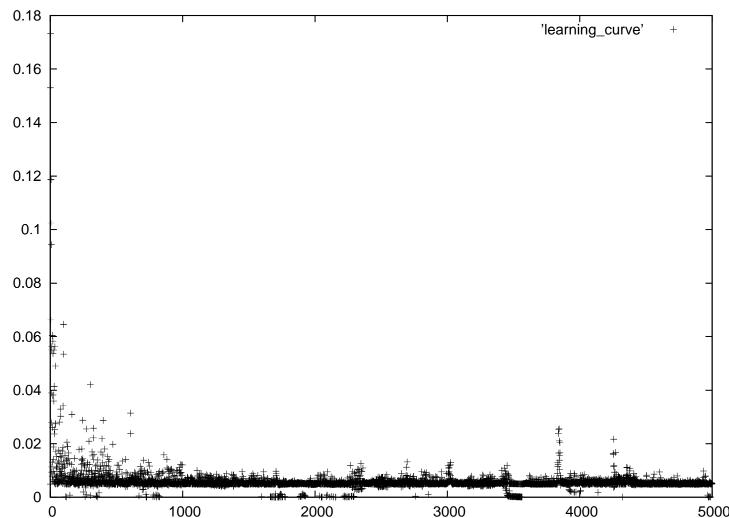


图 7: 学习曲线

算了一场比赛遇到的状态，不同场比赛遇到的状态变化比较大，所以误差波动也较大，不过误差减小的趋势还是可以体现的。观察曲线，还发现学习了 5000 次，误差还没有减小为零，这说明学习过程还没有收敛，还需要进一步学习。

图 8 是开局时黑方判断自己赢的概率的变化过程（一共学习了 5000 场比赛)，可以看出这个概率逐渐收敛到 0.5 附近。

4 结论与展望

4.1 本文主要工作

本文以黑白棋为实例初步介绍了人工智能在下棋问题中的应用，提出了解决这一问题的两个基本方案，即博弈树搜索和机器学习。在介绍博弈树搜索时，先后介绍了极小极大搜索、AlphaBeta 搜索以及 AlphaBeta 搜索的两个改进 (包括历史表和置换表的应用)，通过试验和分析证明，对 AlphaBeta 搜索的改进取得了明显效果。而后，尝试了机器学习的方法，首先介绍了强化学习的标准模型和常用算法，并结合黑白棋问题的实际，选择了 λ-时序差分的算法，并给出了值函数在问题空间了的具体定义，能很好的作为策略选择的依据，通过试验基本证明了学习方法的可行性，取得了一定的效果。

[7]测试环境是 Ubuntu Linux 2.6.22-12-386, AMD x86_64 2211MHz, 1011M RAM

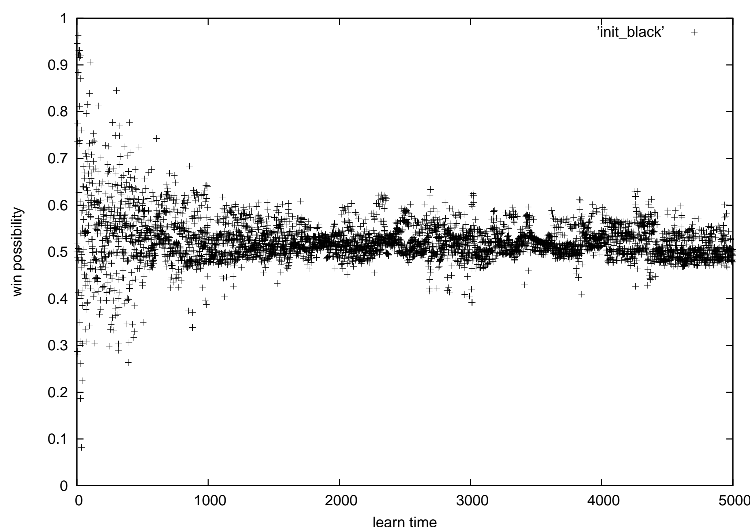


图 8: 开局时黑方判断自己赢的概率的变化过程

4.2 存在的问题

基于搜索的算法效果较好，但目前发现置换表匹配的节点数较少，不能很好的发挥作用，表现为使用置换表的程序最多（要求一分钟之内能走处一步棋）只能搜索 8 层，而使用历史表的程序却能搜到 9 层，这是出乎我的意料的，初步断定应该是置换表了关键字冲突比较多造成的，下一步希望能把 $hash_key$ 的计算方法调整得好一点。基于强化学习的算法，目前棋力没有基于搜索的高，主要是因为学习过程中波动较大，不太稳定，一些参数还需要做进一步的调整。另外网络的结构还可以优化，状态的编码还可以细化，学习模型还待改善。

最后，我希望把学习到的评估函数用到搜索算法里面，这样就可以两个方法各取所长了，效果应该会比较好。

参考文献

[1] Nils J. Nilsson: Artifficial Intelligence: A New Synthesis,
China Machine Press, Beijing, 2000

[2] Hamed Ahmadi Nejad: An Introduction to Game Tree Algorithms,
http://ce.sharif.edu/ ahmadinejad/gametree/

[3] Bruce Moreland: Min-Max Search: An obvious place to start,
http://www.seanet.com/ brucemo/topics/minmax.html

[4] Tom M. Mitchell: Machine Learning,
China Machine Press, Beijing, 2003

[5] Richard S. Sutton and Andrew G. Barto: Reinforcement Learning: An Introduction,
MIT Press, Cambridge, MA, 1998

[6] 宋志伟， 基于逻辑马尔可夫决策过程的关系强化学习研究，
中国科学技术大学博士学位论文

[7] 宋志伟，陈小平： 仿真机器人足球中的强化学习，
机器人，25(7):761-766, S, 2003

[8] Mance E. Harmon and Stephanie S. Harmon: Reinforcement Learning: A Tutorial,
Wright-Patterson AFB, OH, 45433

致谢

首先感谢陈小平教授，是您给我参与到实验室研究活动的机会。能够在您的实验室完成我的大学生研究计划是我莫大的荣幸，您在工作中对我的指导，您表现出的渊博知识，以及您对学术、对科研、对人生的态度都给我留下了深刻的印象，再次向您表示最诚挚的感谢和敬意。

还要感谢实验室的宋老师，感谢您对我的悉心指导和启发关怀，是您让我领会到做研究可以是一件很快乐的事。

实验室 2D 组的吴锋、石轲、章宗长和刘腾飞师兄对我参与研究的整个过程都给予了很大的帮助，在此向他们表示诚挚的谢意。

同时，还要感谢台运方、王文奎和张昊迪同学，你们陪同我一起参加了大研计划，一起学习、研究、分享快乐，共同走过了一段美好、充实的时光。

最后，深深地感谢我的父母，是你们从背后默默支持我走到现在。

Input: board *board*, alpha α, beta β, side to move *color* and remain depth *depth*
Output: useful maximum of its children notes

```
1  ⋮
2  best_move = 0;
3  ⋮
4  switch MatchType(color, alpha, beta) do
5  |   case EXACTLY
6  |   |   return MatchEvaluate( );
7  |   case PARTLY
8  |   |   best_move ← MatchBestMove( );
9  if best_move ≠ 0 then
10 |   MakeMove(board, best_move, color);
11 |   val ← -AlphaBeta(board, −β, −α, my_color + opp_color − color, depth − 1);
12 |   UnMakeMove(board, best_move, color);
13 |   if val > α then
14 |   |   if val ≥ β then
15 |   |   |   FloatMove(color, step + max_depth − depth, move);
16 |   |   |   return val;
17 |   |   AdvantageMove(color, step + max_depth − depth, move);
18 |   |   α = max(val, α);
19 |   max = max(val, max);
20 foreach move ∈ history ∧ move ≠ best_move do
21 |   ⋮
22 |   if val > α then
23 |   |   if val ≥ β then
24 |   |   |   FloatMove(color, step + max_depth − depth, move);
25 |   |   |   ▷ Because we use nega-method, no LOW_BOUND wanted
26 |   |   |   Insert(color, val, move, UP_BOUND);
27 |   |   |   return val;
28 |   |   AdvantageMove(color, step + max_depth − depth, move);
29 |   |   α = max(val, α);
30 |   Insert(color, max, move, NO_BOUND);
31 |   max = max(val, max);
32 return max;
```

算法 10 AlphaBeta(with hash, history)

```
1  repeat
2      s ← Initial( );
3      S ← ∅;
4      while true do
5          if CanMove(s, Black) then
6              a ← ChooseAction(Black, s, S);
7              Execute(a);
8          if CanMove(s, White) then
9              a ← ChooseAction(White, s, S);
10             Execute(a);
11         if !CanMove(s, Black) ∧ !CanMove(s, White) then
12             break;
13 until end;
```

算法 11 Reversi Learning

```
   Input: side to move color, current state s and state list S
   Output: the best action a
1  mark[Black] = 1.0;
2  mark[White] = 0.0;
3
4  a ← arg max_a V[δ(s, a)];
5  max ← V[δ(s, a)];
6  S ← S ∪ s;
7  δ_m ← max − V(s);
8  δ_o ← −δ_m;
9  flag ← mark[color];
10 s_0 ← s;
11 foreach s ∈ S do
12     e ← λ^{dist(s, s_0)};
13     if s[0] = flag then
14         train ← P(s) + αδ_m e;
15     else
16         train ← P(s) + αδ_o e;
17     TrainNetwork(s, train);
18 return a;
```

算法 12 ChooseAction