第 5 章　对 抗 搜 索

本章讨论在有其他 Agent 计划与我们对抗时，该如何预先规划的问题。

5.1 博　　弈

第 2 章介绍了**多 Agent 环境**，其中每个 Agent 需要考虑到其他 Agent 的行动及其对自身的影响。其他 Agent 的不可预测性可能导致该 Agent 问题求解过程中的**偶发性**，正像我们在第 4 章中所讨论的。本章讨论竞争环境，竞争环境中每个 Agent 的目标之间是有冲突的，这就引出了**对抗搜索**问题——通常被称为**博弈**。

数学中的**博弈论**，是经济学的一个分支，把多 Agent 环境看成是博弈，其中每个 Agent 都会受到其他 Agent 的“显著”影响，不论这些 Agent 间是合作的还是竞争的[1]。人工智能中“博弈”通常专指博弈论专家们称为**有完整信息的**、确定性的、轮流行动的、两个游戏者的**零和游戏**（如国际象棋）。术语中，这是指在确定的、完全可观察的环境中两个 Agent 必须轮流行动，在游戏结束时效用值总是相等并且符号相反。例如下国际象棋，一个棋手赢了，则对手一定是输了。正是 Agent 之间效用函数的对立导致了环境是对抗的。

从人类文明产生以来，博弈就和人类智慧如影随形——有时甚至到了令人担忧的程度。对于人工智能研究人员来说，博弈的抽象特性使得博弈成为感兴趣的研究对象。博弈游戏中的状态很容易表示，Agent 的行动数目通常受限，而行动的输出都有严谨的规则来定义。体育游戏如台球和冰球，则有复杂得多的描述，有更大范围的可能行动，也有不够严谨的规则来定义行动的合法性。所以除了足球机器人，体育游戏目前并没有吸引人工智能领域的很大兴趣。

与第 3 章中讨论的大多数玩具问题不同，博弈因为难于求解而更加令人感兴趣。例如国际象棋的平均分支因子大约是 35，一盘棋一般每个棋手走 50 步，所以搜索树大约有 35^{100} 或者 10^{154} 个结点（尽管搜索图“只可能”有大约 10^{40} 个不同的结点）。如同现实世界，博弈要求具备在无法计算出最优决策的情况下也要给出某种决策的能力。博弈对于低效率有严厉的惩罚。在其他条件相同的情况下，只有一半效率的 A*搜索意味着运行两倍长的时间，于是只能以一半效率利用可用时间的国际象棋程序就很可能被击败。所以，博弈在如何尽可能地利用好时间上产生了一些有趣的研究结果。

我们从最佳招数的定义和寻找它的搜索算法开始。接着讨论时间有限时如何选择好的招数。**剪枝**允许我们在搜索树中忽略那些不影响最后决定的部分，启发式的**评估函数**允许在不进行完全搜索的情况下估计某状态的真实效用值。5.5 节讨论诸如西洋双陆棋这类包含概率因素的游戏；我们也讨论桥牌，它包含不完整信息，桥牌中每个人都不能看到所有的

1　包含非常多 Agent 的环境通常被视为**经济系统**，而不是博弈。

牌。最后我们看看最高水平的博弈程序如何与人类对手抗衡以及未来的发展趋势。

首先考虑两人参与的游戏：MAX 和 MIN，马上就会讨论这样命名的原因。MAX 先行，两人轮流出招，直到游戏结束。游戏结束时给优胜者加分，给失败者罚分。游戏可以形式化成含有下列组成部分的一类搜索问题。

- S_0：**初始状态**，规范游戏开始时的情况。
- PLAYER(s)：定义此时该谁行动。
- ACTIONS(s)：返回此状态下的合法移动集合。
- RESULT(s,a)：**转移模型**，定义行动的结果。
- TERMINAL-TEST(s)：**终止测试**，游戏结束返回真，否则返回假。游戏结束的状态称为**终止状态**。
- UTILITY(s,p)：**效用函数**（也可称为目标函数或收益函数），定义游戏者 p 在终止状态 s 下的数值。在国际象棋中，结果是赢、输或平，分别赋予数值+1，0，或 1/2。有些游戏可能有更多的结果，例如双陆棋的结果是从 0 到+192。**零和博弈**是指在同样的棋局实例中所有棋手的总收益都一样的情况。国际象棋是零和博弈，棋局的收益是 0+1，1+0 或 $1/2+1/2$。“常量和”可能是更好的术语，但称为零和更传统，可以将这看成是下棋前每个棋手都被收了 1/2 的入场费。

初始状态、ACTIONS 函数和 RESULT 函数定义了游戏的**博弈树**——其中结点是状态，边是移动。图 5.1 给出了井字棋的部分博弈树。在初始状态 MAX 有九种可能的棋招。游戏轮流进行，MAX 下 X，MIN 下 O，直到到达了树的终止状态即一位棋手的标志占领一行、一列、一对角线或所有方格都被填满。叶结点上的数字是该终止状态对于 MAX 来说的效用值；值越高对 MAX 越有利，而对 MIN 则越不利（这也是棋手命名的原因）。

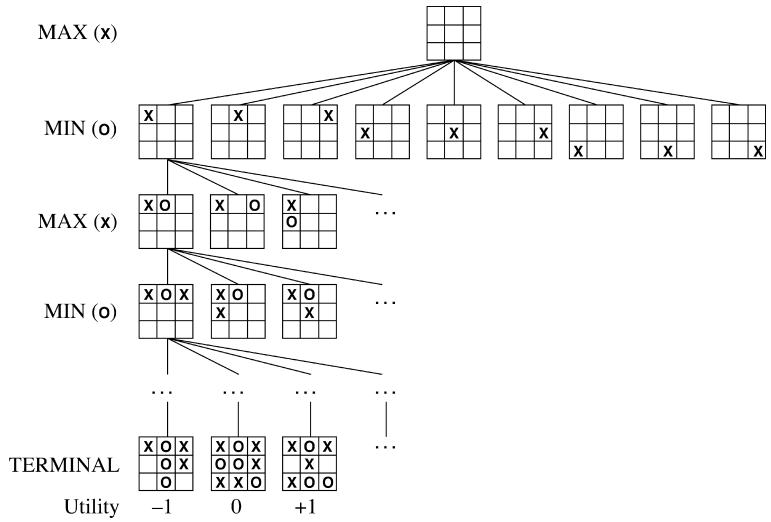


图 5.1 井字棋游戏的（部分）搜索树

最上面的结点是初始状态，MAX 先走棋，放置一个 X 在空位上。图显示了搜索树的一部分，给出 MIN 和 MAX 的轮流走棋过程，直到到达终止状态，所有终止状态都按照游戏规则被赋予了效用值

5.2　博弈中的优化决策

在一般搜索问题中，最优解是到达目标状态的一系列行动——终止状态即为取胜。在对抗搜索中，MIN 在博弈中也有发言权。因此 MAX 必须找到应急**策略**，制定出 MAX 初始状态下应该采取的行动，接着是 MIN 行棋，MAX 再行棋时要考虑到 MIN 的每种可能的回应，依此类推。这有些类似于 AND-OR 搜索算法（图 4.11），MAX 类似于 OR 结点，MIN 类似于 AND 结点。粗略地说，当对手不犯错误时最优策略能够得到至少不比任何其他策略差的结果。我们将从寻找最优策略开始。

即使是井字棋这样简单的游戏，也很难在一页画出它的整个博弈树，所以讨论如图 5.2 所示的更简单游戏。在根结点 MAX 的可能行棋为 a_1，a_2 和 a_3。对于 a_1，MIN 可能的对策有 b_1，b_2 和 b_3，等等。这个特别的游戏在 MAX 和 MIN 各走一步后结束（按照博弈的说法，这棵博弈树的深度是一步，这包括两个单方招数，每个单方招数称为**一层**）。终止状态的效用值范围是从 2 到 14。

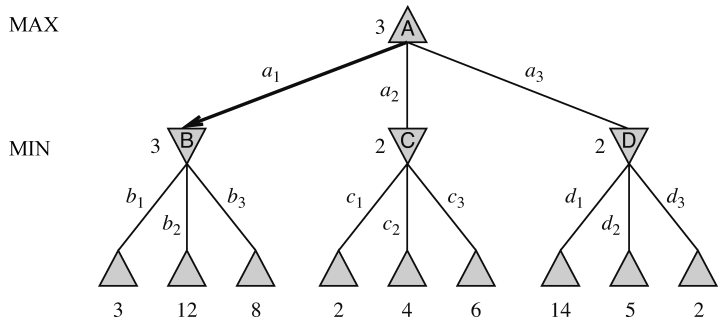


图 5.2　两层博弈树

△结点是“MAX 结点”，代表轮到 MAX 走，▽结点是“MIN 结点”。终止结点显示 MAX 的效用值；其他结点标的是它们的极小极大值。MAX 在根结点的最佳行棋是 a_1，因为它指向有最高的极小极大值的后继，而 MIN 此时的最佳行棋是 b_1，因为它指向有最低的极小极大值的后继

给定一棵博弈树，最优策略可以通过检查每个结点的**极小极大值**来决定，记为 MINIMAX(n)。假设两个游戏者始终按照最优策略行棋，那么结点的极小极大值就是对应状态的效用值（对于 MAX 而言）。显然地，终止状态的极小极大值就是它的效用值自身。更进一步，对于给定的选择，MAX 喜欢移动到有极大值的状态，而 MIN 喜欢移动到有极小值的状态。所以得到如下公式：

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & s \text{ 为终止状态} \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & s \text{ 为 MAX 结点} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & s \text{ 为 MIN 结点} \end{cases}$$

我们将这些定义应用于图 5.2 中的博弈树。底层终止结点的效用值即为它们的效用函数值。第一个 MIN 结点为 B，其三个后继的值分别是 3、12 和 8，所以它的极小极大值是 3。类似地，可以得出其他两个 MIN 结点的极小极大值都是 2。根是 MAX 结点，其后继结点分的极小极大值分别为 3、2 和 2，所以它的极小极大值是 3。可以确定在根结点的**极小**

极大决策：对于 MAX 来说 a_1 是最优选择，因为它指向有最高的极小极大值的终止状态。

对 MAX 的最挂行棋进行求解时做了 MIN 也按最佳行棋的假设——尽可能最大化 MAX 的最坏情况。如果 MIN 不按最佳行棋行动怎么办？这种情况下显然（习题 5.7）MAX 可以做得更好。可能有一些策略在对付非最优化对手方面做得比极小极大策略好，但是用这些策略对付最优化对手则会得到更差的结果。

5.2.1 极小极大算法

极小极大算法（图 5.3）从当前状态计算极小极大决策。它使用了简单的递归算法计算每个后继的极小极大值，直接实现上面公式的定义。递归算法自上而下一直前进到树的叶结点，然后随着递归回溯通过搜索树把极小极大值**回传**。例如，在图 5.2 中，算法先递归到三个底层的叶结点，对它们调用 UTILITY 函数得到效用值分别是 3、12 和 8。然后它取最小值 3 作为回传值返回给结点 B。通过类似的过程可以分别得到 C 和 D 的回传值均为 2。最后在 3、2 和 2 中选取最大值 3 作为根结点的回传值。

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

图 5.3 极小极大值决策算法

返回最佳可能行棋对应的行动，即在假设对手行棋是为了使效用值最小的前提下，能够导致最佳效用值的行动。函数 MAX-VALUE 和 MIN-VALUE 遍历整个博弈树一直到叶结点，以决定每个状态的回传值。$\text{argmax}_{a \in S} f(a)$计算找出集合 S 中有最大 $f(a)$值的 a

极小极大算法对博弈树执行完整的深度优先探索。如果树的最大深度是 m，在每个结点合法的行棋有 b 个，那么极小极大算法的时间复杂度是 $O(b^m)$。一次性生成所有的后继的算法，空间复杂度是 $O(bm)$，而每次生成一个后继的算法（参见原书第 87 页），空间复杂度是 $O(m)$。当然对于真实的游戏，这样的时间开销完全不实用，不过此算法仍然可以作为对博弈进行数学分析和设计实用算法的基础。

5.2.2 多人博弈时的最优决策

许多流行的游戏都允许多个参加者。让我们来看一看如何把极小极大思想推广到多人博弈中。从技术观点上看这很自然，但由此也产生了一些有趣的新概念问题。

首先需要用向量值替换每个结点上的单一效用值。例如若博弈有三个人 A、B 和 C 参

与，则每个结点都与一个向量〈v_A, v_B, v_C〉相关联。对于终止状态，这个向量代表着从每个人角度出发得到的状态效用值（在两人的零和博弈中，由于效用值总是正好相反所以二维向量可以简化为单一值）。最简单的实现方法就是让函数 UTILITY 返回一个效用值向量。

现在考虑非终止状态。考虑在图 5.4 中的博弈树上标为 X 的结点。此时，轮到游戏者 C 选择行棋。它有两种选择，导致的终止状态的效用值向量分别是：〈v_A=1, v_B=2, v_C=6〉和〈v_A=4, v_B=2, v_C=3〉。6 比 3 大，所以 C 选择第一种走法。这意味着如果在状态 X，后继的招数会走到效用值向量为〈v_A=1, v_B=2, v_C=6〉的终止状态。所以，X 的回传值就是这个向量。结点 n 的回传值就是该选手在结点 n 选择的后继者的效用值向量。任何玩过诸如强权外交游戏这样的多人博弈的人很快会意识到这比双人游戏要复杂得多。多人博弈通常会涉及在游戏选手之间出现正式或者非正式**联盟**的情况。随着游戏的进行，联盟不断建立或者解散。我们该如何理解这种行为？联盟是否是各选手选择最优策略的自然结果？结果可能确实是这样。例如，假设 A 和 B 相对比较弱，而 C 比较强。对于 A 和 B 而言，它们一起进攻 C 比等 C 逐个消灭它们要好，这通常是最优策略。这样，博弈从纯自私的行为变成合作。当然，一旦 C 在联合攻击下被削弱，联盟就会失去价值，于是 A 或者 B 就会破坏协议。某些情况下，明确的联盟仅仅是使本将要发生的事情具体化。另外一些情况下，毁约会损害社会声誉，也可能会有直接利益，选手要在这两方面之间寻求平衡。在 17.5 节会有更详细的讨论。

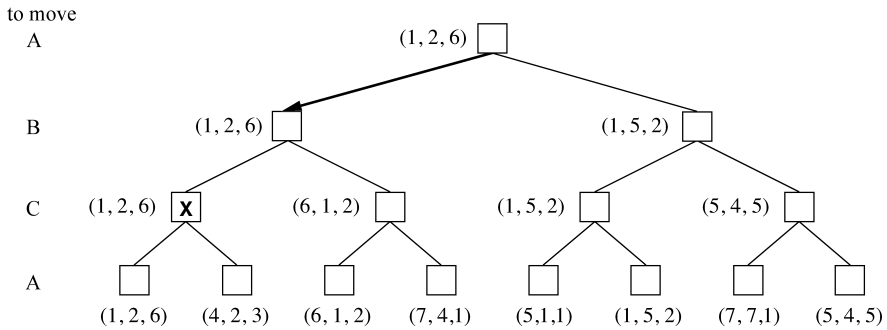


图 5.4　三人参与（A, B, C）的博弈树中的前三层
每个结点上标有从每个选手角度出发的值。根结点标示了最佳步骤

如果游戏是非零和的，合作也可能发生在两人游戏中。例如，假设终止状态的效用值向量是〈v_A=1000, v_B=1000〉，并且 1000 对于两个选手都是最高的可能效用值。那么双方的最优策略就是尽一切可能来到达此状态，即双方会自动合作来达到共同渴望的目标。

5.3　α-β剪枝

极小极大值搜索的问题是必须检查的游戏状态的数目是随着博弈的进行呈指数级增长。不幸的是，指数增长无法消除，不过我们还是可以有效地将其减半。这里的技巧是可能不需要遍历博弈树中每一个结点就可以计算出正确的极小极大值。于是，借用第 3 章中的**剪枝**思想尽可能消除部分搜索树。这种特别技术称为**α-β剪枝**。将此技术应用到标准的极小极大搜索树上，会剪掉那些不可能影响决策的分支，仍然返回和极小极大算法同样的

结果。

再来看图 5.2 中的两层博弈树。重新观察最优决策的计算过程，特别注意此过程中在每个结点的已知信息。图 5.5 解释了每一步骤。结果发现可以在不计算评价其中两个叶结点的情况下就确定极小极大决策。

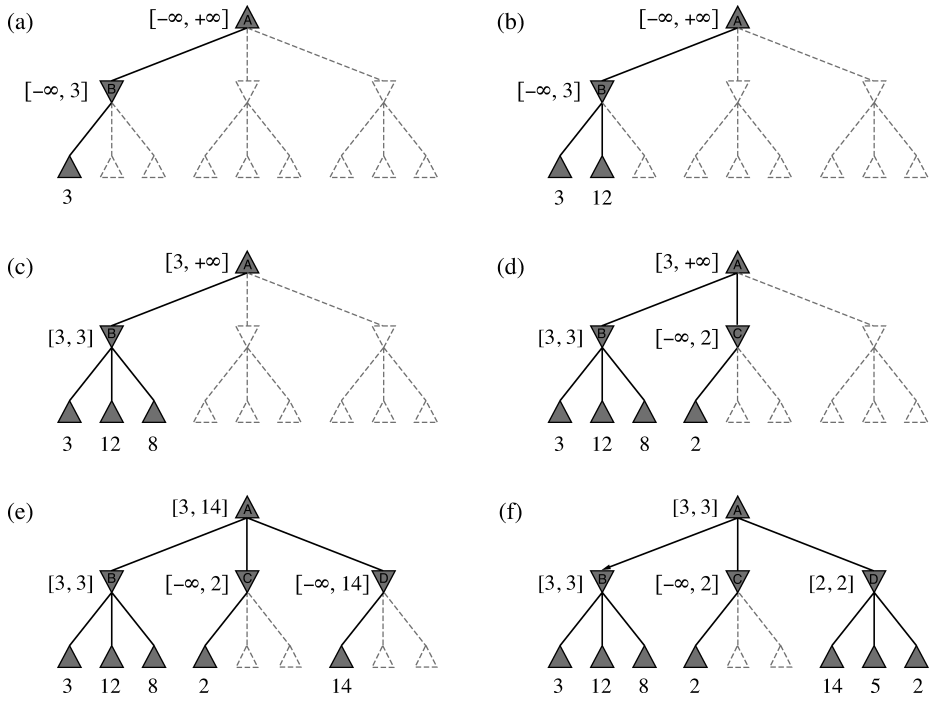


图 5.5　图 5.2 中博弈树的最优决策过程

每一结点上标出了可能的取值范围。（a）*B* 下面的第一个叶结点值为 3。因此作为 MIN 结点的 *B* 值*至多*为 3。（b）*B* 下面的第二个叶结点值为 12。MIN 不会用这招，所以 *B* 的值仍然至多为 3。（c）*B* 下面的第三个叶子值为 8；此时已经观察了 *B* 的所有后继，所以 *B* 的值就是 3。现在可以推断根结点的值至少为 3，因为 MAX 在根结点有值为 3 的后继。（d）*C* 下面的第一个叶结点值为 2。因此 *C* 这个 MIN 结点的值至多为 2。不过已经知道 *B* 的值是 3，所以 MAX 不会选择 *C*。这时再考察 *C* 的其他后继已经没有意义了。这就是α-β剪枝的实例。（e）*D* 下面的第一个叶结点值为 14，所以 *D* 的值*至多*为 14。这比 MAX 的最佳选择（即 3）要大，所以继续探索 *D* 的其他后继。还要注意现在知道根的取值范围，根结点的值至多为 14。（f）*D* 的第二个后继值为 5，所以我们又必须继续探索。第三个后继值为 2，所以 *D* 的值就是 2 了。最终 MAX 在根结点的决策是走到值为 3 的 *B* 结点

还可以把这个过程看作是对 MINIMAX 公式的简化。假设图 5.5 中的 *C* 结点的两个没有计算的子结点的值是 x 和 y。根结点的值计算如下：

$$\begin{aligned}\text{MINIMAX}(root) &= \max(\min(3, 12, 8), \min(2, x, y), \min(14, 5, 2)) \\ &= \max(3, \min(2, x, y), 2) \\ &= \max(3, z, 2) \qquad \text{其中 } z = \min(2, x, y) \leqslant 2 \\ &= 3\end{aligned}$$

即，根结点的值以及因此做出的极小极大决策与被剪枝的叶结点 x 和 y 无关。

α-β剪枝可以应用于任何深度的树，很多情况下可以剪裁整个子树，而不仅仅是剪裁叶结点。一般原则是：考虑在树中某处的结点 n（见图 5.6），选手选择移动到该结点。如果选手在 n 的父结点或者更上层的任何选择点有更好的选择 m，那么在实际的博弈中就永远不会到达 n。所以一旦发现关于 n 的足够信息（通过检查它的某些后代），能够得到上述结

论，我们就可以剪裁它。

记住极小极大搜索是深度优先的，所以任何时候只需考虑树中某条单一路径上的结点。α-β剪枝的名称取自描述这条路径上的回传值的两个参数：

α = 到目前为止路径上发现的 MAX 的最佳（即极大值）选择

β = 到目前为止路径上发现的 MIN 的最佳（即极小值）选择

α-β搜索中不断更新α 和β 的值，并且当某个结点的值分别比目前的 MAX 的α 或者 MIN 的β 值更差的时候剪裁此结点剩下的分支（即终止递归调用）。完整算法参见图 5.7。建议读者把此算法应用于图 5.5 中的树。

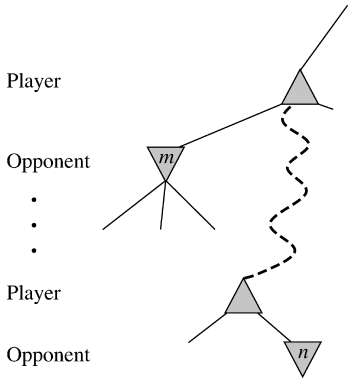


图 5.6　α-β剪枝的一般情况
如果对选手而言 *m* 比 *n* 好，那么行棋就不会走到 *n*

```
function ALPHA-BETA-SEARCH(state) returns an action
  v←MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v←−∞
  for each a in ACTIONS(state) do
    v←MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α←MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v←+∞
  for each a in ACTIONS(state) do
    v←MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β←MIN(β, v)
  return v
```

图 5.7　α-β搜索算法
注意算法和图 5.3 中的极小极大算法相似，除了在 MIN-VALUE 和 MAX-VALUE 中修改α 和β 的值（还有用来传递这些参数的记录）

5.3.1　行棋排序

α-β剪枝的效率很大程度上依赖于检查后继状态的顺序。例如在图 5.5（e）和（f）中，根本不能剪掉 *D* 的任何后继（从 MIN 的角度），因为首先生成的是最差的后继。如果 *D* 的第三个后继先生成，就能够剪掉其他两个。这意味着应首先检查可能最好的后继。

如果能够这样做[1]，那么α-β算法只需检查 $O(b^{m/2})$个结点来做出决策（*m* 是对的最大深度），而不是极小极大算法的 $O(b^m)$。这意味着有效分支因子不是 *b* 而是$\sqrt{b}$ ——对于国际象棋而言不是 35 而是 6。换种说法，在同样的时间里α-β算法比极小极大算法向前预测大

1　显然，这很难做到；否则，用排序函数就可以下一盘好棋了！

约两倍的步数。如果后继状态采用随机顺序而不是最佳优先的顺序，那么要检查的总结点数大约是 $O(b^{3m/4})$。对于国际象棋，有一些相当简单的排序函数（如吃子优先，然后是威胁、前进、后退）可以使得检查的总结点数为 $O(b^{m/2})$的两倍。

增加动态行棋排序方案，如先试图采用以前走过的最好行棋，可能让我们非常接近理论极限。以前的行棋可能是上一步棋——面临同样的棋局威胁——也可能来自当前行棋的上一次搜索过程。从当前行棋获得信息的一种方法是迭代深入搜索。首先，搜索一层并记录最好行棋路径。接着搜索更深一层，此时使用记录的路径来导引行棋排序。在第 3 章已经看到，指数级增长的博弈树上的迭代深入搜索只增加了常数级别的搜索时间，可能从行棋排序做得更多。最好行棋称为**绝招**，先走绝招称为绝招启发式。

第 3 章提到，搜索树中的重复状态会使搜索代价呈指数级增长。在博弈中，重复的状态频繁出现是因为**换位**——不同行棋序列导致同样棋局。例如，白棋走 a_1，黑棋用 b_1 应对，白棋在棋盘另一边的不相关的一招 a_2，黑棋走 b_2 来应对。于是序列$[a_1, b_1, a_2, b_2]$和$[a_2, b_2, a_1, b_1]$都到达同样棋局。第一次遇到某棋局时把该棋局的评估值存储在哈希表里很有价值，这样当它后来再出现时不需要重新计算。存储以前见过的棋局的哈希表一般被称为**换位表**；它本质上和图搜索中的 explored 表相同（参见 3.3 节）。使用换位表可以取得很好的动态效果，在国际象棋中有时可能把到达的搜索深度扩大一倍。另一方面，如果可以每秒钟评价上百万个结点，那么在换位表中就不太可能保存所有评价了。选择保留有价值的结点而摒弃其他结点，则有许多不同的策略。

5.4 不完美的实时决策

极小极大算法生成整个博弈的搜索空间，而α-β算法允许我们剪裁掉其中的一大部分。然而，α-β算法仍然要搜索部分空间直到终止状态。这样的搜索深度也是不现实的，因为要在合理的时间内确定行棋——典型地最多只有几分钟的时间来决策。Claude Shannon 发表论文《设计计算机国际象棋程序》（1950），文中提出应该尽早截断搜索，应将启发式**评估函数**用于搜索中的状态，有效地把非终止结点转变为终止结点。换言之，建议按两种方式对极小极大算法或α-β算法进行修改：用估计棋局效用值的启发式评估函数 EVAL 取代效用函数，用决策什么时候运用 EVAL 的**截断测试**取代终止测试。因此得到如下的启发式极小极大值，s 为状态，d 为最大深度：

$$\text{H-MINIMAX}(s, d) = \begin{cases} \text{EVAL}(s) & \text{如果 CUTOFF-TEST}(s,d)\text{为真} \\ \max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d+1) & s \text{ 为 MAX 结点} \\ \min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d+1) & s \text{ 为 MIN 结点} \end{cases}$$

5.4.1 评估函数

与第 3 章中启发式函数返回对目标距离的估计一样，对于给定的棋局，评估函数返回对游戏的期望效用值的估计。估计的思想早在 Shannon 之前就有。数百年来，人类与计算

机程序相比搜索的力量更加受限，国际象棋棋手（其他游戏也一样）找到了一些判断棋局价值的方法。显而易见的是博弈程序的性能严重依赖于评估函数的质量。不准确的评估函数可能引导 Agent 走向失败。如何设计好的评估函数呢？

首先，评估函数对终止状态的排序应该和真正的效用函数的排序结果一样：赢状态的评估值一定要好于平局，而平局一定要好于输的状态。否则，使用它的 Agent 可能会出昏招，即使它可以向前一直看到游戏结束。第二，评估函数的计算本身不能花费太长时间（总观点是为了更快地搜索）。第三，对于非终止状态，评估函数应该和取胜几率密切相关。

先讨论“取胜几率”的含义。国际象棋不是几率博弈：我们确定知道当前状态，没有骰子。不过如果搜索必须在一些非终止状态截断，那么算法对这些状态的最后结果必然是不确定的。这种不确定性引入的原因是计算局限性，而不是信息受限。在计算能力有限的情况下，评估函数对给定状态进行评估，它能做的就是尽可能猜测最后的结果。

下面把上述思想具体化。大多数评估函数都要考虑状态的不同**特征**参数——例如在国际象棋中，包括白兵的数目、黑兵的数目、白后的数目、黑后的数目，等等。这些特征一起定义了状态的各种类别或者等价类：同一分类中的状态对所有特征具有相同的值。例如，某分类是包含两兵对一兵的残局。一般来说，任何给定的分类都会包含制胜的状态，导致平局的状态和会导致失败的状态。评估函数无法知道到底是在哪种状态，不过它可以为每个结果返回一个值，反映出这些状态的比例。例如，假设经验告诉我们某类中 72%的状态是制胜的（效用值+1），20%是会输的（0），而其他 8%是平局（1/2）。那么该类中状态的合理评价是**期望值**：$(0.72 \times +1) + (0.20 \times 0)+(0.08 \times ½) = 0.76$。总体上每个分类确定一个期望值，帮助产生任一状态的评估函数。对于终止状态，评估函数无须返回实际的期望值，因为状态的排序保持不变。

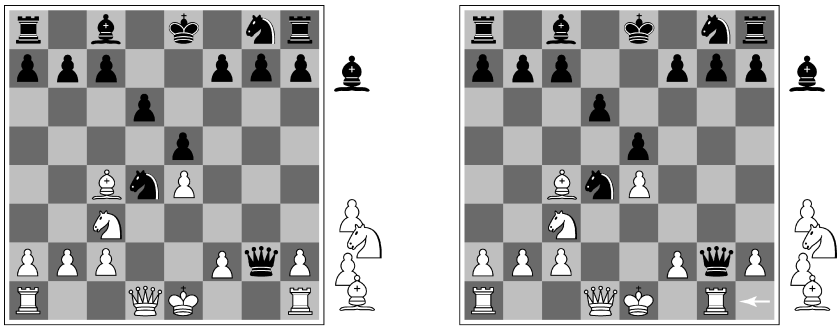
在实际应用中，这种分析往往有太多的分类，因此需要太多的经验去估计所有的取胜可能。所以，大多数评估函数会分别计算每个特征的影响，然后把它们组合起来找到总数值。例如，国际象棋的入门书中给出各个棋子的**子力价值**估计如下：兵值 1 分，马和象值 3 分，车值 5 分，后值 9 分。其他特征诸如“是否好兵阵”和“王是否安全”可能值半个兵。这些特征值简单地加在一起就得到了对棋局的评估。

如果评估比对方多大概一个兵的棋力时，实际上会有较大的胜面，如果评估比对方多三个兵的棋力时基本上是必胜的，如图 5.8（a）所示。评估函数可如下表示，这在数学上称为**加权线性函数**：

$$\mathrm{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$$

其中 w_i 是权值，f_i 是棋局的某个特征。对于国际象棋来说，f_i 可能是棋盘上每种棋子的数目，w_i 可能是每种棋子的价值（如兵为 1，象为 3，等等）。

把特征值这样加起来的方法看起来是合理的，不过实际上这是以很强的假设为基础：每个特征的贡献独立于其他特征的值。例如给象赋予 3 分忽略了象在残局中能够发挥更大作用的事实，象在残局中有更大的发挥空间。因此，当前国际象棋或其他博弈程序也使用非线性的特征组合。例如，两象的价值比单个象价值的两倍要略大一些，象在残局中棋力值更高（即当下棋步数很大并且剩余棋子数很少时）。



(a) White to move (b) White to move

图 5.8 两个几乎相同的棋局，只有右下角车的位置不同
在（a）中，黑方多一个马两个兵，应该取胜。在（b）中白方会吃掉黑方皇后，从而确立几乎必胜的优势

机敏的读者会发现特征和权值并不是属于国际象棋规则！它们来自于几个世纪以来人们下棋的经验。有些博弈很难总结经验规律，这时可以利用第 18 章的机器学习技术来确定评估函数的权值。要指出的是，机器学习也证实了国际象棋中的一个象确实值三个兵。

5.4.2 截断搜索

下一步则是修改 ALPHA-BETA-SEARCH，当适合截断搜索时调用启发式函数 EVAL。实现时用下面一行程序替换图 5.7 中提到 TERMINAL-TEST 的两行代码：

```
if CUTOFF-TEST(state, depth) then return EVAL(state)
```

内存中还必须安排记录一些信息，这样 *depth* 在每一次递归调用时可以逐渐增加。最直接的控制搜索次数的方法是设置固定的深度限制，这样 CUTOFF-TEST(*state*, *depth*)当 *depth* 大于固定深度 *d* 时返回 true（同 TERMINAL-TEST 一样，对于所有终止结点它也返回 true）。根据游戏规则许可的时间来决定深度 *d*。更好的方法是使用迭代深入（参见第 3 章）。当时间用光时，程序返回目前最深的完整搜索所选择的招数。而且，迭代深入同样可以帮助行棋排序。

由于评估函数的近似本质，这种方法可能会导致错误。重新考虑国际象棋中基于子力优势的简单评估函数。假设程序在搜索图 5.8（b）的棋局时到达了深度限制，此时黑方有一马两兵的优势。程序会报告这个状态的启发式函数值，从而认为这个状态会导致黑方获胜。而其实下一步白方就可以毫无意外地吃掉黑方皇后。因此，这个棋局实际是白棋赢，需要向前多看一步才能预测。

显然我们需要更加复杂的截断测试。评估函数只适用于那些**静态**棋局——即，评估值不会很快出现大的摇摆变化的棋局。例如在国际象棋中，有很好吃招的棋局对于只统计棋力的评估函数来说就不是静态的。非静态棋局可以进一步扩展直到变为静态棋局。这种额外的搜索称为**静态搜索**；有时它只考虑某些类型的棋招，诸如吃子能够快速消解棋局的不确定性。

地平线效应更难消除。这是指对手招数导致我方严重损失并且从理论上基本无法避免时。考虑图 5.9 中的棋局，可以看出黑象无路可逃。例如，白车可以走 h1,a1,a2 从而吃掉