3188 人工智能导论
2024年春季 Note 5

作者（其他所有注释）：尼基尔·夏尔马
作者（贝叶斯网络注释）：乔希·胡格和杰基·梁， 由王蕾吉娜编辑
作者（逻辑注释）：亨利·朱， 由考佩林编辑
致谢（机器学习和逻辑注释）：部分章节改编自教科书《人工智能：一种现代方法》。

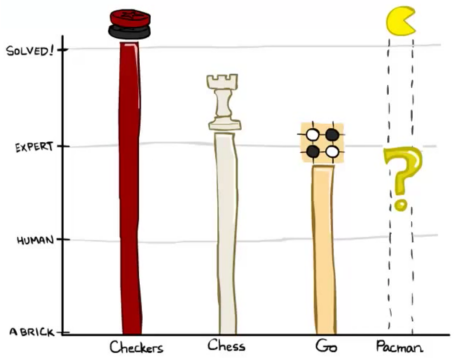
最后更新时间：2023年8月26日

博弈

在第一篇笔记中，我们讨论了搜索问题以及如何高效且最优地解决这些问题——通过使用强大的广义搜索算法，我们的智能体能够确定最佳可行计划，然后简单地执行该计划以达成目标。现在，让我们转换话题，考虑这样的场景：我们的智能体有一个或多个对手，这些对手试图阻止它们达成目标。我们的智能体不能再运行我们已经学过的搜索算法来制定计划，因为我们通常无法确定地知道对手会如何针对我们制定计划并对我们的行动做出反应。相反，我们需要运行一类新的算法，这类算法能够给出对抗搜索问题的解决方案，这类问题更常见的叫法是博弈。

游戏有许多不同的类型。游戏中的行动可能会产生确定性的结果，也可能会产生随机（概率性）的结果，可以有任意数量的玩家，并且可能是零和游戏，也可能不是。我们要介绍的第一类游戏是确定性零和游戏，即行动是确定性的，我方的收益直接等同于对手的损失，反之亦然。思考这类游戏最简单的方式是将其定义为一个单一的变量值，一方团队或智能体试图使其最大化，而另一方团队或智能体则试图使其最小化，这实际上使它们处于直接竞争状态。在吃豆人游戏中，这个变量就是你的分数，你试图通过快速高效地吃豆子来使其最大化，而幽灵则试图通过先吃掉你来使其最小化。许多常见的家庭游戏也属于这类游戏：

- 国际跳棋——第一个国际跳棋计算机玩家于1950年创建。从那时起，国际跳棋就成为了一种已解决的游戏，这意味着给定双方玩家都采取最优策略的情况下，任何局面都可以被确定地评估为对任何一方是赢、输还是平局。
- 国际象棋——1997年，深蓝成为首个在六局比赛中击败人类国际象棋冠军加里·卡斯帕罗夫的计算机程序。深蓝被设计为使用极其复杂的方法，每秒可评估超过2亿个棋局。目前的程序甚至更强大，尽管没有那么具有历史意义。
- 围棋——围棋的搜索空间比国际象棋大得多，因此大多数人认为在未来几年内围棋计算机程序都无法击败人类世界冠军。然而，谷歌开发的阿尔法围棋在2016年3月以4比1的比分历史性地击败了围棋冠军李世石。



上述所有世界冠军级别的智能体至少在某种程度上都使用了我们即将介绍的对抗搜索技术。与返回全面计划的普通搜索不同，对抗搜索返回一种策略，该策略会根据我们的智能体及其对手的某种配置简单地推荐最佳可行行动。我们很快就会看到，这类算法具有通过计算产生行为的美妙特性——我们运行的计算在概念上相对简单且具有广泛的通用性，但它能自然而然地在同一团队的智能体之间产生协作，以及对对手智能体进行 “智胜”。

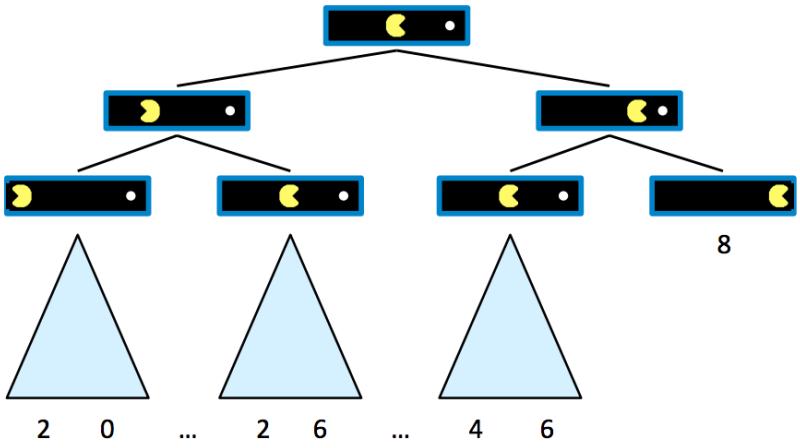
标准博弈形式由以下定义组成：

- 初始状态， s_0
- 玩家，Players(s) 表示轮到谁行动
- 行动，Actions (s) 表示玩家可用的行动
- 转移模型 Result(s,a)
- 终端测试，终端测试
- 终端值，效用（状态，玩家）

极小极大算法

我们要考虑的第一个零和博弈算法是极小极大算法，它基于这样一个假设运行：我们面对的对手行为最优，并且总是会做出对我们最不利的行动。为了介绍这个算法，我们必须首先形式化终端效用和状态值的概念。一个状态的值是控制该状态的智能体所能获得的最优分数。为了理解这意味着什么，观察下面这个极其简单的吃豆人游戏棋盘：

假设吃豆人从10分开始，每移动一步就失去1分，直到它吃到小球，此时游戏进入终端状态并结束。我们可以如下开始为这个棋盘构建一个游戏树，其中一个状态的子节点就像普通搜索问题的搜索树一样是后继状态：



从这棵博弈树可以明显看出，如果吃豆人直接走向豆子，他将以8分的成绩结束游戏，而如果他在任何时候折返，最终得到的分数会更低。既然我们已经生成了一个包含多个终结状态和中间状态的博弈树，那么我们就可以正式定义这些状态中任何一个状态的值的含义了。

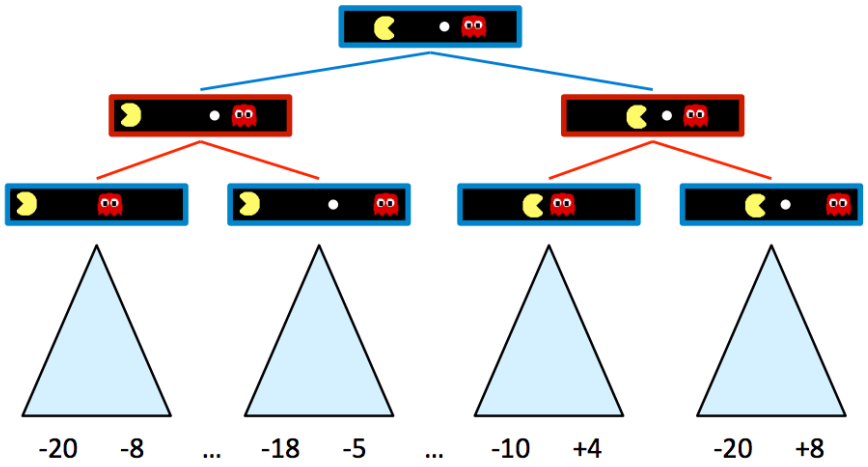
一个状态的值被定义为智能体从该状态所能获得的最佳可能结果（效用）。我们稍后会更具体地形式化效用的概念，但目前简单地将智能体的效用看作它获得的分数或点数就足够了。终结状态的值，称为终结效用，总是某个确定的已知值，并且是游戏的固有属性。在我们的吃豆人示例中，最右边终结状态的值就是8，即吃豆人直接走向豆子所获得的分数。同样，在这个示例中，非终结状态的值被定义为其子状态值中的最大值。将定义状态 s 值的函数定义为 $V(s)$，我们可以总结上述讨论：

$$\begin{aligned} \forall \text{non-terminal states}, \quad V(s) &= \max_{s' \in successors(s)} V(s') \\ \forall \text{terminal states}, \quad V(s) &= \text{known} \end{aligned}$$

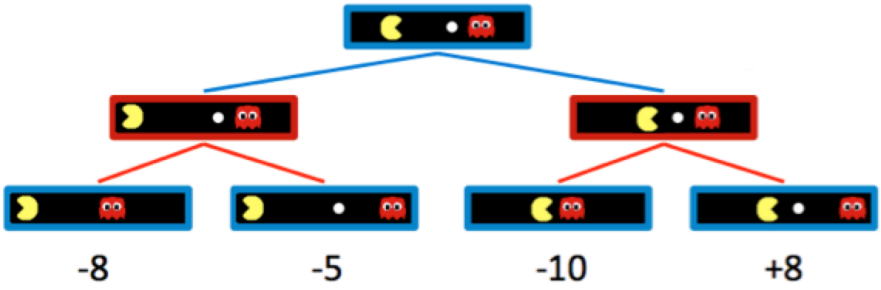
这建立了一个非常简单的递归规则，由此可知根节点的直接右子节点的值将为8，根节点的直接左子节点的值将为6，因为这些分别是智能体从起始状态向右或向左移动时可以获得的最大可能分数。由此可见，通过运行这样的计算，智能体可以确定向右移动是最优的，因为右子节点的值大于起始状态的左子节点的值。

现在让我们引入一个新的游戏棋盘，上面有一个敌对的幽灵，它想阻止吃豆人吃到小球。

游戏规则规定，两个智能体轮流行动，从而形成一个游戏树，在这个游戏树中，两个智能体在它们“控制”的树层上交替行动。一个智能体控制一个节点仅仅意味着该节点对应于轮到它行动的状态，因此这是它们决定行动并相应地改变游戏状态的机会。以下是由上述新的双智能体游戏棋盘产生的游戏树：



蓝色节点对应于Pacman控制的节点，Pacman可以决定采取什么行动，而红色节点对应于幽灵控制的节点。请注意，幽灵控制节点的所有子节点都是幽灵从其父节点状态向左或向右移动后的节点，Pacman控制的节点则相反。为了简单起见，我们将这个游戏树截断为深度为2的树，并为终端状态分配如下欺骗值：



自然而然地，添加由幽灵控制的节点会改变吃豆人认为的最优移动，而新的最优移动是通过极小极大算法确定的。极小极大算法并非在树的每个层级上对节点的子节点的效用进行最大化，而是仅在由吃豆人控制的节点的子节点上进行最大化，同时在由幽灵控制的节点的子节点上进行最小化。因此，上面的两个幽灵节点的值分别为 $\min(-8,-5)=-8$ 和 $\min(-10,+8)=-10$。相应地，由吃豆人控制的根节点的值为 $\max(-8,-10)=-8$。由于吃豆人想要最大化他的分数，他会向左走并获得 -8 的分数，而不是试图去吃豆子并获得 -10 的分数。这是通过计算产生行为的一个典型例子——尽管吃豆人想要如果他最终处于最右边的子状态就能获得的 +8 分，但通过极小极大算法他"知道"一个表现最优的幽灵不会让他得到它。为了最优地行动，吃豆人被迫权衡利弊，并且违反直觉地远离豆子以最小化他失败的程度。我们可以将极小极大算法为状态赋值的方式总结如下：

$$\begin{aligned}
\forall \text{ agent-controlled states}, \quad V(s) &= \max_{s' \in successors(s)} V(s') \\
\forall \text{ opponent-controlled states}, \quad V(s) &= \min_{s' \in successors(s)} V(s') \\
\forall \text{ terminal states}, \quad V(s) &= \text{known}
\end{aligned}$$

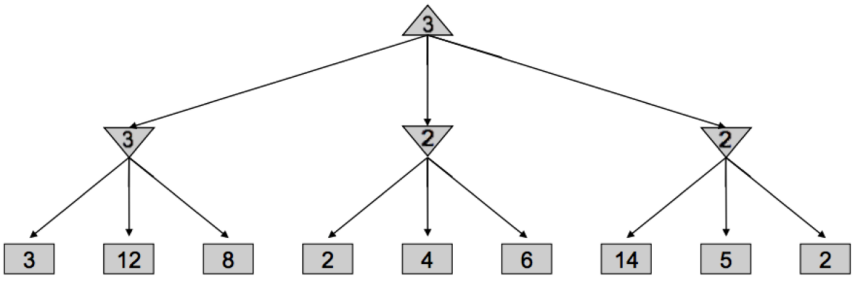
在实现过程中，极小化极大算法的行为类似于深度优先搜索，按照与深度优先搜索相同的顺序计算节点值，从最左边的终端节点开始，然后逐步向右进行。更准确地说，它对博弈树进行后序遍历。下面给出了极小化极大算法的伪代码，既简洁又直观。请注意，极小化极大算法将返回一个动作，该动作对应于根节点从其获取值的子节点的分支。

```
def value(state):
    if the state is a terminal state: return the state's utility
    if the agent is MAX: return max-value(state)
    if the agent is MIN: return min-value(state)

def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v

def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
```

α-β剪枝

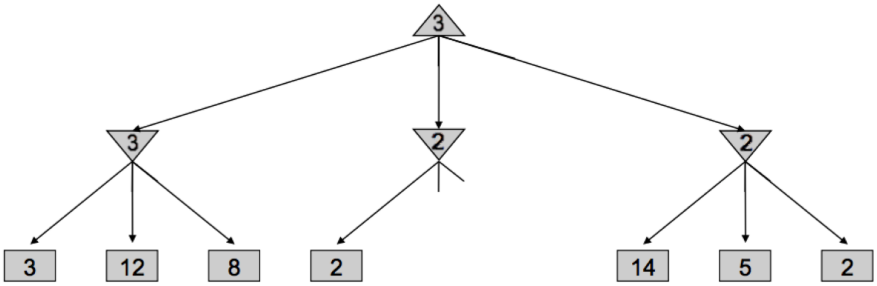
极小化极大算法看起来近乎完美——它简单、最优且直观。然而，它的执行过程与深度优先搜索非常相似，并且时间复杂度相同，糟糕的是 $O\left(b^m\right)$ 。回想一下 b 是分支因子， m 是可以找到终端节点的近似树深度，这对于许多游戏来说会产生过长的运行时间。例如，国际象棋的分支因子是 $b \approx 35$ ，树深度是 $m \approx 100$ 。为了帮助缓解这个问题，极小化极大算法有一个优化——alpha-beta剪枝。

从概念上讲，α-β剪枝是这样的：如果你试图通过查看节点 n 的后继节点来确定该节点的值，一旦你知道 n 的值最多只能等于 n 父节点的最优值，就停止查看。让我们通过一个例子来解开这个棘手陈述的含义。考虑以下博弈树，方形节点对应终态，向下指向三角形对应极小化节点，向上指向三角形对应极大化节点：



让我们逐步了解极小极大算法是如何得出这棵树的——它首先遍历值为3、12和8的节点，并将值 $\min(3,12,8)=3$ 赋给最左边的极小化节点。然后，它将 $\min(2,4,6)=2$ 赋给中间的极小化节点，并将 $\min(14,5,2)=2$ 赋给最右边的极小化节点，最后才将 $\max(3,2,2)=3$ 赋给根节点处的极大化节点。然而，如果我们思考这种情况，就会意识到一旦我们访问了值为2的中间极小化节点的子节点，我们就不再需要查看中间极小化节点的其他子节点了。为什么呢?

由于我们已经看到中间最小化器的一个子节点值为2，所以我们知道无论其他子节点的值是多少，中间最小化器的值最多为2。既然已经确定了这一点，让我们再进一步思考——根节点的最大化器正在比较左最小化器返回的3和 ≤ 2 的值，无论其其余子节点的值如何，它肯定会优先选择左最小化器返回的3而不是中间最小化器返回的值。这正是我们可以修剪搜索树，而无需查看中间最小化器其余子节点的原因：



实施这样的剪枝可以将我们的运行时间缩短至与 $O\left(b^{m/2}\right)$ 相当的水平，有效地使我们的"可解决"深度翻倍。在实际情况中，通常会减少很多，但一般来说至少可以使搜索再深入一到两个层次成为可能。这仍然相当重要，因为能够提前思考三步的玩家比只能提前思考两步的玩家更有优势赢得比赛。这种剪枝正是带有α-β剪枝的极小化极大算法所做的，实现如下：

α：到达根节点路径上MAX的最佳选项

 β：到达根节点路径上MIN的最佳选项

```
def max-value(state, α, β):
    初始化 v = −∞
    对于状态的每个后继状态:
        v = max(v, value(successor, α, β))
        如果 v ≥ β 返回 v
        α = max(α, v)
    返回v
```

```
def min-value(state, α, β):
    初始化 v = +∞
    对于状态的每个后继状态:
        v = min(v, 后继状态的价值(α, β))
        如果 v ≤ α 返回 v
        β = min(β, v)
    返回v
```

花些时间将此与普通极小极大算法的伪代码进行比较，并注意到我们现在几乎可以在不遍历每个后继节点的情况下返回。

评估函数

尽管α-β剪枝有助于增加我们可行地运行极小极大算法的深度，但对于大多数游戏来说，这通常仍远远不足以深入搜索树的底部。因此，我们转向评估函数，即那些接受一个状态并输出该节点真正极小极大值估计的函数。通常，这可以简单地理解为，一个好的评估函数会给"更好"的状态赋予比"更差"的状态更高的值。评估函数在深度受限的极小极大算法中被广泛使用，在这种算法中，我们将位于最大可解深度的非终端节点视为终端节点，根据精心选择的评估函数为它们赋予模拟终端效用。

由于评估函数只能得出非终端效用值的估计值，因此在运行极小极大算法时无法保证最优玩法。
在设计运行极小极大算法的智能体时，通常会在评估函数的选择上投入大量的思考和实验，评估函数越好，智能体的行为就越接近最优。此外，在使用评估函数之前深入搜索树也往往能给我们带来更好的结果——将计算更深地埋入博弈树中可以减轻最优性的折损。这些函数在博弈中的作用与启发式方法在标准搜索问题中的作用非常相似。
评估函数最常见的设计是特征的线性组合。

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + ... + w_n f_n(s)$$

每个 $f_i(s)$ 对应于从输入状态 s 中提取的一个特征，并且为每个特征分配一个相应的权重 w_i。特征仅仅是游戏状态的一些元素，我们可以从中提取并赋予一个数值。例如，在跳棋游戏中，我们可能构建一个具有4个特征的评估函数：己方棋子数量、己方王棋数量、对方棋子数量和对方王棋数量。然后，我们会大致根据它们的重要性选择合适的权重。在我们的跳棋示例中，为己方的棋子/王棋选择正权重，为对方的棋子/王棋选择负权重是最合理的。此外，我们可能会决定，由于在跳棋中王棋比棋子更有价值，所以与己方/对方王棋对应的特征所应赋予的权重，其绝对值应大于与棋子相关的特征的权重。以下是一个可能符合我们刚刚讨论出的特征和权重的评估函数：

$$Eval(s) = 2 \cdot agent\_kings(s) + agent\_pawns(s) - 2 \cdot opponent\_kings(s) - opponent\_pawns(s)$$

如你所见，评估函数的设计可以相当自由，也不一定非得是线性函数。例如，基于神经网络的非线性评估函数在强化学习应用中非常常见。需要牢记的最重要一点是，评估函数要尽可能频繁地为更好的位置给出更高的分数。这可能需要进行大量的微调，并使用具有多种不同特征和权重的评估函数对智能体的性能进行实验。