CS 188 人工智能导论

2024年春季 笔记5

作者（所有其他笔记）：Nikhil Sharma

作者（贝叶斯网络笔记）：Josh Hug和Jacky Liang，由Regina Wang编辑

作者（逻辑笔记）：Henry Zhu，由Peyrin Kao编辑

致谢（机器学习和逻辑笔记）：部分章节改编自教科书Artificial Intelligence: A Modern Approach.

最后更新：2023年8月26日

游戏

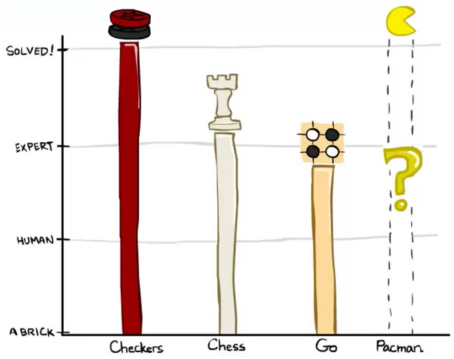
在第一笔记中，我们讨论了搜索问题以及如何有效且最优地解决它们--使用强大的通用搜索算法，我们的代理可以确定最佳计划，然后简单执行以到达目标。现在，让我们换个思路，考虑我们的代理有一个或多个**对手**尝试阻止他们达到目标的情况。我们的代理不能再运行我们已经学到的搜索算法来制定计划，因为我们通常无法确定对手将如何制定计划并回应我们的行动。相反，我们需要运行一个新的算法类来解决**对抗性搜索问题**，更常见的是称之为**游戏**。

有很多不同类型的游戏。游戏的动作结果可以是确定性的或**随机的**（概率性的），可以有任意数量的玩家，并且可能或可能不是**零和**。我们将讨论的第一类游戏是**确定性零和游戏**，即动作是确定性的，我们的收益直接等同于对手的损失，反之亦然。最简单的理解这些游戏的方法是将其定义为一个单变量值，一个团队或代理试图最大化，而对方团队或代理试图最小化，有效地将其置于直接竞争中。在吃豆人游戏中，该变量是你的得分，你通过快速有效地吃豆子试图最大化得分，而幽灵则试图通过先吃掉你来最小化得分。许多常见的家庭游戏也属于这类游戏：

· 跳棋- 第一个跳棋电脑玩家创建于1950年。从那时起，跳棋已成为一个**已解决的游戏**，这意味着任何位置都可以确定性地评估为胜、负或和局，只要双方都采取最佳行为。

· 国际象棋- 1997年，深蓝成为第一个在六局比赛中击败人类国际象棋冠军加里·卡斯帕罗夫的电脑代理。深蓝被设计用于每秒评估超过两亿个位置的极为复杂的方法。当前的程序甚至更好，尽管不那么具有历史性。

· 围棋- 围棋的搜索空间比国际象棋大得多，因此大多数人认为围棋电脑代理在未来几年内都无法击败人类世界冠军。但是，谷歌开发的AlphaGo在2016年3月历史性地以4比1击败围棋冠军李世石。



上述所有世界冠军代理至少在某种程度上都使用了我们即将讨论的对抗性搜索技术。不同于返回全面计划的常规搜索，对抗性搜索返回的是**策略**或**方针**，仅根据我们代理和其对手的一些配置推荐最佳移动。我们很快会看到，这些算法具有通过计算产生行为的美丽特性--我们运行的计算在概念上相对简单且广泛通用，但本质上会在同一团队的代理之间产生合作，并在对抗代理中产生“超前思考”。

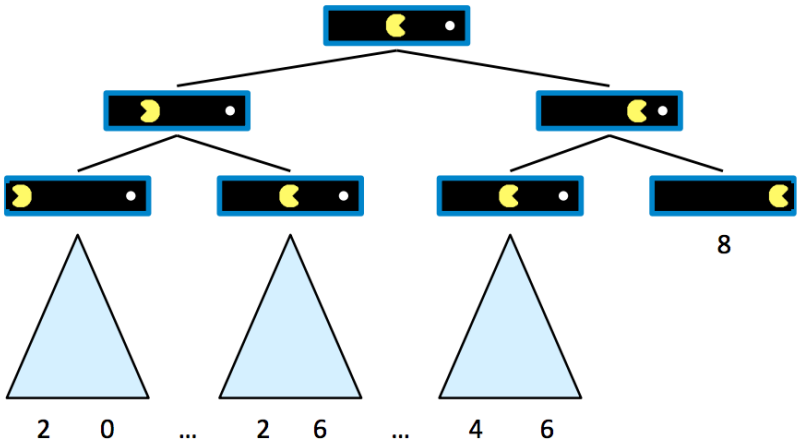
标准游戏公式包括以下定义：

- Initial state, s_0
- Players, $Players(s)$ denote whose turn is
- Actions, $Actions(s)$ available actions for the player
- Transition model $Result(s,a)$
- Terminal test, $Terminal-test(s)$
- Terminal values, $Utility(s,player)$

极小极大

我们将考虑的第一个零和博弈算法是 **极小极大**，该算法运行的动机假设是我们面对的对手表现得最优，并且总是会做对我们最不利的行动。为了介绍这个算法，我们首先必须形式化 **终端效用** 和 **状态值** 的概念。状态值是控制该状态的代理可获得的最佳分数。为了了解这意味着什么，请观察以下简单的吃豆人游戏板：

假设吃豆人从10分开始，每走一步扣1分，直到他吃掉豆子，此时游戏到达 **终端状态**并结束。我们可以开始为这个板块构建一个 **游戏树**，其状态的子级是继任状态，就像普通搜索问题的搜索树一样：



从这棵树中可以明显看出，如果吃豆人直接走到豆子，他将以8分结束游戏，而如果他在任何时候回溯，他将以较低的分数结束。现在我们已经生成了有几个终端和中介状态的游戏树，我们准备形式化这些状态的值的含义。

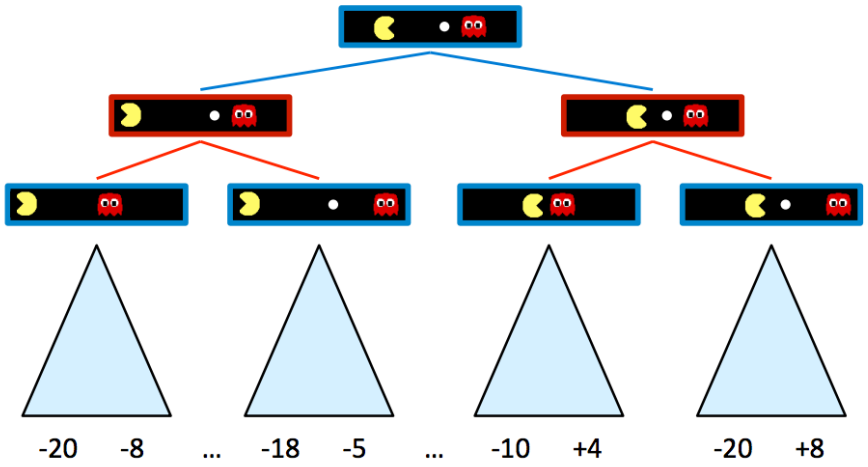
状态的值被定义为代理从该状态可以实现的最佳可能结果 (**效用**)。我们稍后会更具体地形式化效用的概念，但现在简单地认为代理的效用是其获得的分数或点数。终端状态的值称为 **终端效用**，总是已知的确定值并且是固有的游戏属性。在我们的吃豆人例子中，最右边终端状态的值仅仅是8，这是吃豆人通过直接走到豆子获得的分数。在这个例子中，非终端状态的值被定义为其子状态值的最大值。定义V(s)作为定义状态s值的函数，我们可以总结上述讨论：

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

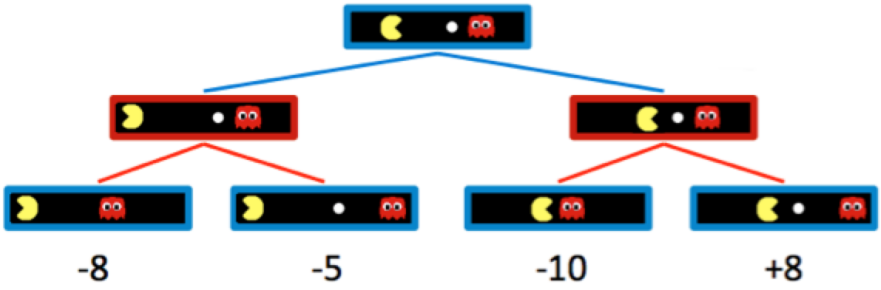
这建立了一个非常简单的递归规则，从中可以看出，根节点的右子节点的值应该是 8，而根节点的左子节点的值应该是 6，因为这是代理在初始状态下向右或向左移动时可以获得的最大可能分数。因此，通过运行这样的计算，一个代理可以确定最佳的移动方向为向右，因为右子节点的值比初始状态的左子节点的值大。

现在我们要引入一个新的游戏板，其中有一个对抗的幽灵要阻止吃豆人吃掉豆子。

游戏规则规定两个代理轮流移动，从而形成一个游戏树，其中两个代理在他们“控制”的树层上轮流切换。一个代理控制一个节点意味着该节点对应于该代理的回合，因此这是他们决定行动并相应改变游戏状态的机会。以下是基于新的两个代理游戏板生成的游戏树：



蓝色节点对应的是吃豆人控制的节点，吃豆人可以决定采取什么行动，而红色节点对应的是幽灵控制的节点。注意，所有幽灵控制节点的子节点都是幽灵从其父节点的状态向左或向右移动的节点，吃豆人控制节点也是如此。为简化起见，让我们将这个游戏树截短为一个深度为2的树，并分配伪造的值给终端状态如下：



显然，添加由幽灵控制的节点改变了吃豆人认为的最佳移动方向，并且新的最佳移动通过极小极大算法来确定。极小极大算法不再在树的每一级上最大化子节点的效用，而是仅对吃豆人控制的节点进行最大化，同时对幽灵控制的节点进行最小化。因此，上述两个幽灵节点的值分别是 $\min(-8, -5) = -8$ 和 $\min(-10, +8) = -10$。相应地，吃豆人控制的根节点的值是 $\max(-8, -10) = -8$。由于吃豆人想要最大化他的分数，他会向左移动并获得 -8，而不是尝试吃掉豆子并得分 -10。这是计算产生行为的一个典型例子——尽管吃豆人希望得到 $+8$ 的分数，但通过极小极大算法，他“知道”表现最佳的幽灵不会让他得到。因此，为了最优地行动，吃豆人被迫对冲他的赌注，反常地远离豆子以减小失败的幅度。我们可以总结极小极大算法为状态赋值的方法如下：

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

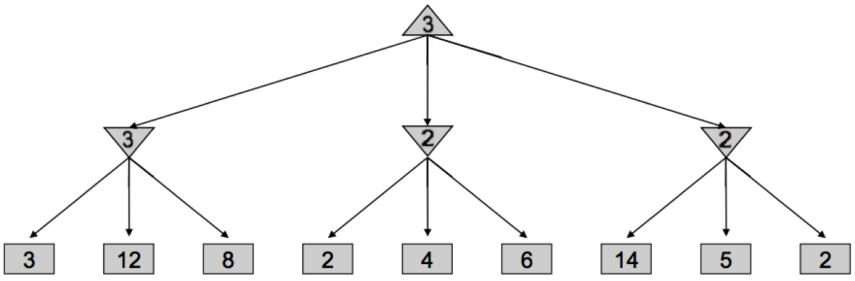
在实现中，极大极小算法行为类似于深度优先搜索，按照DFS的顺序计算节点值，起始于最左端的终端节点，逐步向右进行。具体来说，它进行**后序遍历**游戏树。极大极小算法的伪代码简洁直观，如下所示。注意，极大极小算法将返回一个动作，该动作对应于根节点指向其值的子节点的分支。

```
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
```

Alpha-Beta剪枝

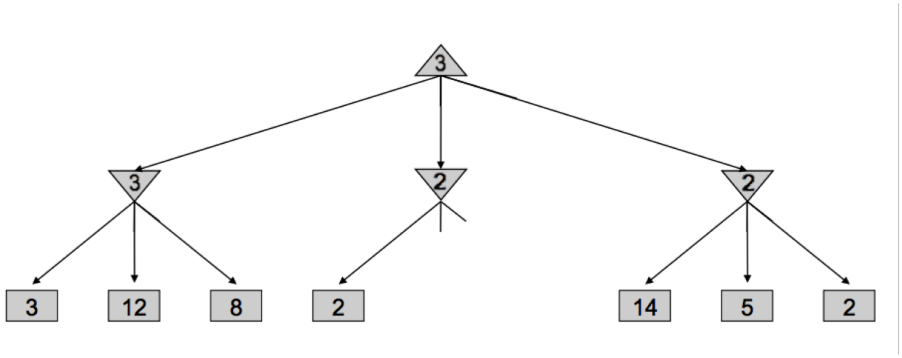
极大极小算法看起来几乎完美 - 它简单、最优且直观。然而，它的执行与深度优先搜索非常相似，时间复杂度相同，都是$O(b^m)$。回想一下b是分支因子，m是找到终端节点的近似树深度，这对于许多游戏而言，运行时间过长。例如，国际象棋的分支因子$b \approx 35$，树深度$m \approx 100$。为了缓解此问题，极大极小算法有一项优化 -**alpha-beta剪枝**。

概念上，alpha-beta剪枝是这样的：如果你在通过查看节点n的后继者来确定其值，当知道节点n的值最多等于其父节点的最优值时，立即停止查看。让我们通过一个例子来解析这一复杂的表述。考虑下图所示的游戏树，方形节点对应于终端状态，下指三角形对应于最小化节点，上指三角形对应于最大化节点：



让我们演示minimax是如何生成这棵树的——它通过遍历值为3， 12和8的节点，并给左边的最小化节点分配值 min(3,12,8)=3。然后给中间的最小化节点分配值 min(2,4,6)=2，接着给右边的最小化节点分配值 min(14,5,2)=2，最后给根节点的最大化节点分配值 max(3,2,2)=3。然而，如果我们考虑这个情况，我们可以意识到，一旦我们访问的值为2的中间最小化节点的子节点时，我们就不需要再看中间最小化节点的其他子节点了。为什么？因为我们已经看到一个值为2的中间最小化节点的子节点，

我们知道无论其他子节点的值是什么，中间最小化节点的值最多只能是2。现在既然这点已经确定，让我们进一步思考，根节点的最大化节点正在决定是选择左边最小化节点的值3还是选择一个值≤2的值，根节点一定会选择左边最小化节点的返回值3而不是中间最小化节点的返回值，无论剩余子节点的值如何。这正是为什么我们可以**剪枝**搜索树，而不看中间最小化节点的剩余子节点：



实现这种剪枝可以将我们的运行时间减少到$O(b^{m/2})$，有效地使我们的“可解”深度翻倍。在实践中，通常要少很多，但总体上可以使我们至少再深入一到两层。这仍然非常重要，因为考虑到三步棋比考虑两步棋的玩家更有可能获胜。这个剪枝正是最小最大算法结合alpha-beta剪枝所做的，并实现如下：

α: MAX's best option on path to root
β: MIN's best option on path to root

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

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

花点时间将其与普通的极大极小算法伪代码进行比较，请注意，我们现在可以提前返回，而无需搜索所有后续节点。

评估函数

虽然alpha-beta剪枝可以帮助增加我们可以合理运行minimax的深度，但这通常仍然远远不够，无法完成大多数游戏的搜索树。因此，我们转向**评估函数**，这些函数接收一个状态并输出该节点的真实minimax值的估算值。通常，这被简单解释为好的评估函数会给“更好”的状态分配更高的值。评估函数广泛应用于**深度限制的minimax**，在这里，我们将位于最大可解深度的非终端节点视为终端节点，赋予它们人为终端效用值，这些值是由精心选择的评估

函数确定的。因为评估函数只能产生非终端效用值的估计值，这在运行minimax时取消了最佳游戏行为的保证。

在为运行minimax的代理设计评估函数时，通常会投入大量的思考和实验，评估函数越好，代理的行为就越接近最佳。此外，在使用评估函数之前更深入地搜索也通常会给我们更好的结果——在游戏树中更深入地计算它们可以减轻对最优性的影响。这些函数在游戏中的作用与启发式搜索问题中的作用非常相似。

最常见的评价函数设计是线性组合**特征**。

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

每个 $f_i(s)$对应从输入状态 s中提取的一个特征，每个特征都分配一个对应的**权重** w_i。 特征只是我们可以提取并赋予数值的游戏状态的一些元素。例如，在跳棋游戏中，我们可以构造一个具有4个特征的评价函数: 代理方的兵数、代理方的王数、对手的兵数和对手的王数。然后，我们会根据它们的重要性选择适当的权重。在我们的跳棋示例中，为代理方的兵/王选择正权重，为对手的兵/王选择负权重最合理。此外，我们可能会决定，由于王在跳棋中比兵更有价值，对应于代理方/对手王的特征应该具有比兵特征更大的权重。下面是一个可能的评价函数，它符合我们刚刚构思的特征和权重:

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

正如您所看到的，评价函数设计可以非常自由形式，也不一定都是线性函数。例如基于神经网络的非线性评价函数在强化学习应用中非常常见。最重要的是记住，评价函数尽可能频繁地为更好的位置提供更高的分数。这可能需要大量的微调和实验，以评估使用具有多种不同特征和权重的评价函数的代理的性能。