Lecture5 对抗搜索

1. 介绍

背景引入

- 对抗搜索问题等价于游戏
- 它们经常发生在有**多个智能体**比赛的环境下
- 存在**对手**，我们不能控制它对抗我们的行为
- 游戏 vs 搜索：最优解不是一系列行动，而是一种策略
    - 如果对手做了 a，那么智能体做 b，否则如果对手做了 c，那么智能体做 d，等等
- 如果是硬编码的（即，用规则实现），则会显得冗长而脆弱
- 不过，好的是，游戏被建模为**搜索**问题，并使用**启发式评估函数**

游戏

- 游戏对于 AI 来说是很大的挑战
- 游戏对于 AI 来说非常有趣，因为它们很难解决
    - 对于象棋，它的平均分支因子是 35，那么暴力搜索有 35^{100} 个节点，约等于 10^{154} 个
- 即使最优决策是不可行的，也需要做出一些决策

游戏的类型

	确定的	不确定的
完整的信息	象棋、围棋、跳棋、黑白棋	西洋棋、大富翁
不完整的信息	战棋、盲井字棋	桥牌，扑克，拼字游戏

- 我们最感兴趣的是确定性博弈，完全可观察的环境，零和博弈，其中两个主体交替行动

零和游戏 Zero-Sum Games

- 对抗：完全的比赛
- 智能体对于每一步棋的结果有不同的评判标准
- **一个智能体最大化单一的评判标准，它的对手会企图使它最小化**
- 每一个玩家（智能体）的行动被称为一个“ply”

嵌入式思考

- 嵌入式思维或逆向推理
- 当一个智能体判断它需要如何行动时
    - 如何决定，它会思考一系列可能的行动
    - 它也需要考虑它的对手
    - 当然它的对手也会考虑如何行动
    - 每个智能体都将想象对手对它们行为的反应

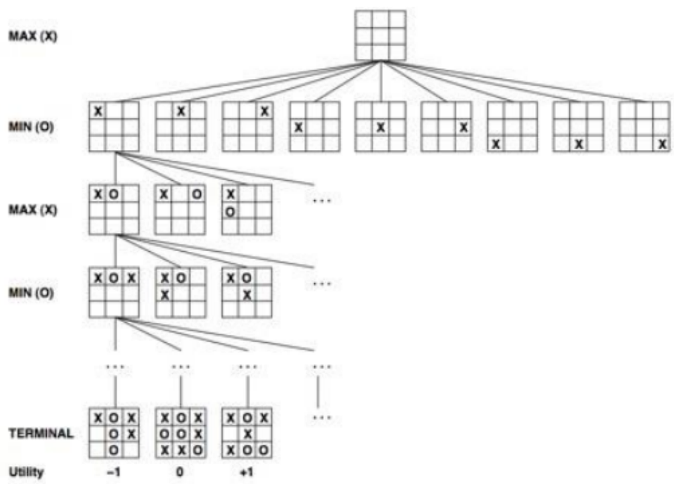
- 这就需要一种嵌入式的思考

2. 对抗搜索问题

问题形式化

- 初始状态：游戏开始的状态
- 玩家 $Player(s)$：确定哪个玩家在状态 s 的时候可以行动
    - 通常是轮流的
- 行动 $Actions(s)$：返回在状态 s 下一系列合理的行动
- 转移函数 Transition Function：$S \times A \rightarrow S$：确定行动的结果
- 终局测试 Terminal Test：当游戏结束的时候，返回 True，否则返回 False
    - 游戏结束的状态叫做终止状态 terminal state
- 效用（目标函数）$Utility(s, p)$：目标函数评估在游戏进入终止状态 s 时，玩家 p 的得分
    - 对于棋类游戏，评估函数的输出可以是
        - 赢：+1
        - 输：0
        - 平：1/2

Minimax 算法



- 两个玩家：Max 玩家和 Min 玩家
- 玩家轮流行动
- Max 先行动，最大化结果
- Min 后行动，最小化结果
- 计算每个节点的极大极小值，这是针对最优对手的最佳可实现效用（目标函数）
- 极小极大值 ≡ 对抗最佳策略时的最佳可达到收益

找到 Max 的最佳策略

- 对于游戏树进行 DFS 搜索
- 最优叶节点可以出现在树的任何深度
- Minimax原则：计算处于一种状态下的效用，假设双方玩家从此时起到游戏结束都处于最佳状态
- 一旦发现终端节点，就向上传播极大极小值

Minimax 值传递

For a state s minimax(s) =

$$\begin{cases} Utility(s) & \text{if Terminal-test(s)} \\ max_{a \in Actions(s)} \text{ minimax(Result(s,a))} & \text{if Player(s) = Max} \\ min_{a \in Actions(s)} \text{ minimax(Result(s,a))} & \text{if Player(s) = Min} \end{cases}$$

- 如果状态是终端节点，那么评估函数输出是 $Utility(s)$
- 如果状态是一个 Max 的节点，评估函数输出的是它的子节点中效用的最大的
- 如果状态是一个 Min 的节点，评估函数输出的是它的子节点中效用的最小的

伪代码

```
// 找到utility最高的下一个状态
function DECISION(state):
    return STATE:
    <child, _> = MAXIMIZE(state)
    return child
```

```
function MAXIMIZE(state):
    return Tuple of <State, Utility>

    if TERMINAL-TEST(state):
        return <NULL, Utility(state)>

    <maxChild, maxUtility> = <NULL, -∞>

    for child in state.children():
        <_, utility> = MINIMIZE(child)
        if utility > maxUtility:
            <maxChild, maxUtility> = <child, utility>

    return <maxChild, maxUtility>
```

```
function MINIMIZE(state):
    return Tuple of <State, Utility>

    if TERMINAL-TEST(state):
        return <NULL, Utility(state)>

    <minChild, minUtility> = <NULL, +∞>

    for child in state.children():
        <_, utility> = MAXIMIZE(child,)
        if utility < minUtility:
```

```
            <minChild, minUtility> = <child, utility>

    return <minChild, minUtility>
```

示例

算法分析

- 最优（对手最优）和完全（有限的树）
- DFS 的时间复杂度 $O(b^m)$
- DFS 的空间复杂度 $O(bm)$

不过这还有问题

- 井字棋
  - $b \approx 5, \ m = 9$
  - $5^9 = 1953125$
- 棋类
  - $b \approx 35$
  - $d \approx 100$（游戏树的平均深度）
  - $b^d = 35^{100} \approx 10^{154}$ 个节点

- 围棋
    - 分支因子 b 开始的时候甚至达到了 361（19x19 的棋盘）

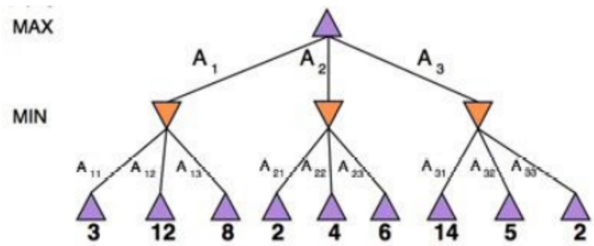
解决时间复杂度过大

- 问题：在真正的游戏中，我们的时间是有限的，不可能搜索到叶节点的
- 为了在合理的时间内实际运行，Minimax 只能搜索到一定深度
- 如果搜索的层度越深，选择的结果可能有很大的不同
- 解决时间复杂度的问题
    - 在非终端节点的地方，使用的一种局面评估函数替换终端效用
    - 使用 Iterative Deepening Search（IDS）
    - 使用剪枝：删除树的大部分

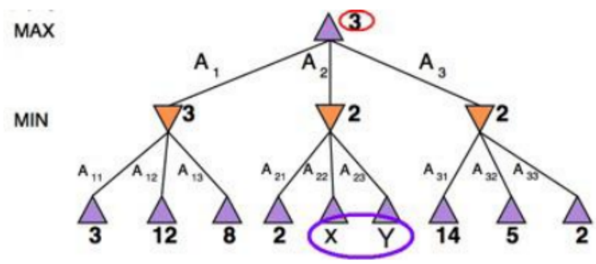
Minimax 算法 α-β 剪枝

引入

在一个 2-ply 的游戏搜索树中



哪个是不需要的?



$$
\begin{aligned}
\text{Minimax(root)} &= \max(\min(3,12,8), \min(2,X,Y), \min(14,5,2)) \\
&= \max(3, \min(2,X,Y), 2) \\
&= \max(3, Z, 2) \quad \text{ where } Z = \min(2,X,Y) \le 2 \\
&= 3
\end{aligned}
$$

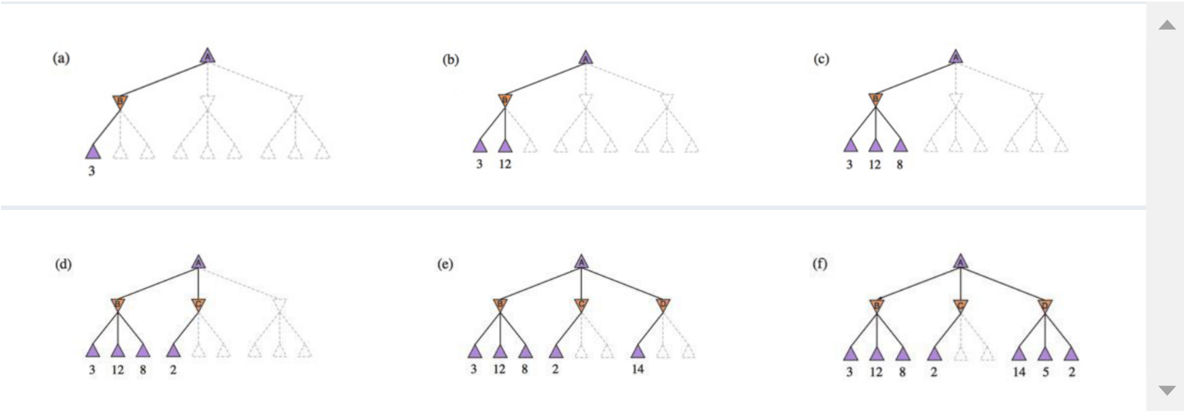
- Minimax 算法的选择与 X 和 Y 的值无关

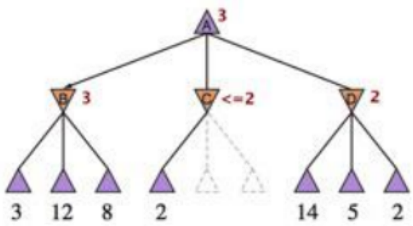
剪枝策略

- 就像 Minimax 算法一样，进行 DFS 搜索
- 参数：注意保持两个边界
    - α：Max 层在计算的子节点上的最大值（Max 输出的当前最小边界）
    - β：Min 层在计算子节c点上的最小值（Min 输出的当前最大边界）

- 初始化：$\alpha = -\infty, \beta = \infty$
- 传播：将 α, β 的值向下传播搜索用于剪枝
    - 更新通过向上传播终端节点的效用来更新 α, β
    - 仅在 Max 层更新 α，在 Min 层更新 β
- 剪枝：当 $\alpha \geq \beta$ 的时候，进行剪枝

示例





伪代码

```
// 找到utility最高的下一个状态
function DECISION(state):
    return STATE:
    <child, _> = MAXIMIZE(state, -∞, +∞)
    return child
```

```
function MAXIMIZE(state, α, β):
    return Tuple of <State, Utility>

    if TERMINAL-TEST(state):
        return <NULL, Utility(state)>

    <maxChild, maxUtility> = <NULL, -∞>

    for child in state.children():
        <_, utility> = MINIMIZE(child, α, β)
        if utility > maxUtility:
            <maxChild, maxUtility> = <child, utility>

        if maxUtility ≥ β:
```

```
            break

        if maxUtility > α
            α = maxUtility

    return <maxChild, maxUtility>
```

```
function MINIMIZE(state, α, β):
    return Tuple of <State, Utility>

    if TERMINAL-TEST(state):
        return <NULL, Utility(state)>

    <minChild, minUtility> = <NULL, +∞>

    for child in state.children():
        <_, utility> = MAXIMIZE(child, α, β)
        if utility < minUtility:
            <minChild, minUtility> = <child, utility>

        if minUtility ≤ α:
            break

        if minUtility < β:
            β = minUtility

    return <minChild, minUtility>
```

分析

- 最坏时间复杂度（Minimax 没有剪去任何一个枝，最好的选择在游戏树的最右边）$O(b^m)$
- 最好时间复杂度（最好的选择在游戏树的最左边）$O(b^{\frac{m}{2}})$（实践上）

实时决策 Real-time decisions

- Minimax：生成整个游戏搜索空间
- α-β 剪枝：修剪了树的很大一部分
- 但是 α-β 剪枝要一直计算到叶节点，这是不现实的实时操作
- 解决方案
  - 限制搜索深度
  - 在非终端节点的地方（但是要返回的时候），使用的一种局面评估函数替换终端效用，评判该节点的局面情况

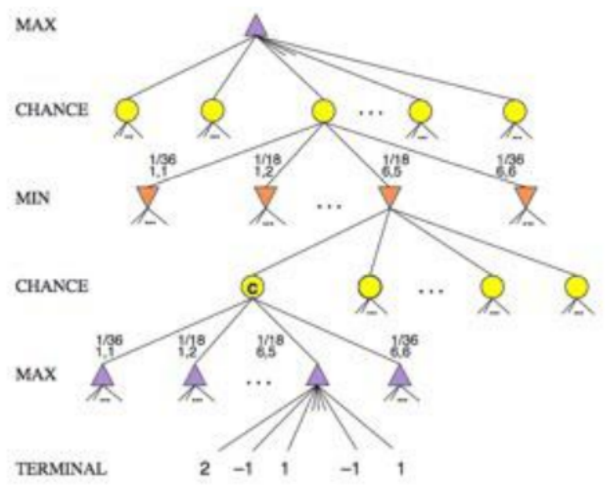
Utility(s)

- 一种评价状态 s 的表现
  - 示例
    - 黑白棋：白色棋子的数量与黑色棋子的数量
    - 象棋：所有白色棋子的值与所有黑色棋子的值

- 一个理想的计算函数将以与真正效用函数相同的方式对终端状态进行排序，但必须要快

随机博弈游戏 Stochastic Games

- 包含一个随机元素（掷色子）
- 包含机会节点
- 示例：西洋双陆棋，一种结合技巧和运气的古老棋盘游戏
- 游戏的目标是，每个玩家都要在对手之前将自己所有的棋子移出棋盘



Expectiminimax 算法

考虑随机的情况，使用 Minimax 来处理一些随机情况

- If state is a Max node then return the highest Expectiminimax-Value of Successors(state)
- If state is a Min node then return the lowest Expectiminimax-Value of Successors(state)
- If state is a chance node then return average of Expectiminimax-Value of Successors(state)

对于状态 s 来说，$Expectiminimax(s)$ 等于

$$\begin{cases} Utility(s) & \text{if Terminal-test(s)} \\ max_{a \in Actions(s)}\ \text{Expectiminimax(Result(s,a))} & \text{if Player(s)} = \text{Max} \\ min_{a \in Actions(s)}\ \text{Expectiminimax(Result(s,a))} & \text{if Player(s)} = \text{Min} \\ \sum_r P(r)\ \text{Expectiminimax(Result(s,r))} & \text{if Player(s)} = \text{Chance} \end{cases}$$

- r 表示所有的随机事件（例如，掷骰子）
- $Result(s, r)$：当随机事件是 r 的时候，状态 s 的评估

总结

- 对于 AI 来说，游戏是一个令人兴奋和有趣的话题
- 由于状态空间巨大，设计对抗性搜索代理颇具挑战性
- 我们只是触及了这个话题的皮毛
- 进一步探讨的主题包括部分可观察的游戏（如桥牌、扑克等纸牌游戏）
- AI 将游戏建模为一个搜索问题
- Minimax 算法在给定对手最优策略的情况下选择最优的走法
- Minimax 算法的效用一直延伸到叶节点，这是不现实的，因为游戏时间有限

- Alpha-Beta 剪枝可以减少游戏树的搜索，从而允许在时间限制下搜索更深
- 在实践中，剪枝、启发式的评估函数、节点重新排序和 IDS 算法都是有效的