人工智能——博弈树搜索

饶洋辉
数据科学与计算机学院,
中山大学
raoyangh@mail.sysu.edu.cn
http://sdcs.sysu.edu.cn/node/2471

博弈树搜索

- 博弈树
- MiniMax策略
- Alpha Beta 剪枝
- 评价函数

- Next: CSP
- Readings: Chap 6.1, 6.2, 6.3

*Slides based on those of Sheila McIlraith

搜索问题的泛化

- 至今我们考虑的搜索问题都假设智能体对环境有完全的控制
  - 除非智能体做出改变环境的行为，否则状态不会改变
- 这种假设并不总是合理的
  - 可能存在利益与你的智能体相违背的其他智能体
- 在这种情况下，我们需要通过扩展搜索的视角（泛化）来处理不是由我们的智能体所控制的状态的变化

博弈的主要特点

- 每个玩家都有他们自身的利益取向
- 每个玩家都会根据自身的利益来改变世界（状态）
- 难点：你如何行动取决于你认为对方会如何行动，当对方如何行动又取决于他们认为你会如何行动

博弈的特征

- 两个玩家
- 离散的：游戏的状态或决策可以映射为离散的值
- 有限的：游戏的状态或可以采取的行动的种类是有限的

博弈的特征

- 零和博弈：完全的竞争
  - 游戏的一方赢了，则另一方输掉了同等的数量
  - 有些游戏并不具备这个特征
- 确定性：没有不确定的因素
  - 没有骰子，没有随机抽取的扑克牌，没有抛硬币等
- 完整的信息：任何层面的状态都是可观察的
  - 比如：没有隐藏的卡牌

博弈的示例：剪刀，石头，布

- 剪刀可以剪布，布可以包石头，石头可以砸剪刀
- 可以用矩阵表示：玩家1选择一行，玩家2选择一列
- 每一格表示各个玩家结算的分数（玩家1的分数/玩家2的分数）
- 1：赢了，0：平局，-1：输了
- 所以这个游戏是零一博弈

Player II

Player I	R	P	S
R	0/0	-1/1	1/-1
P	1/-1	0/0	-1/1
S	-1/1	1/-1	0/0

两玩家零和博弈的扩展

- 剪刀石头布是简单的一次性（one shot）的博弈
  - 每一方只有一次动作
  - 博弈论中：属于策略或范式博弈
- 许多博弈是有多步的
  - 轮流：玩家是交替行动的
  - 比如，象棋、跳棋等
  - 在博弈论中：属于扩展形式的博弈
- 我们专注于扩展形式的博弈
  - 扩展形式的博弈中才会出现需要计算的问题

两玩家零和博弈的定义

- 两个玩家 A（Max）和B（Min）
- 状态集合S（游戏状态的有限集合）
- 一个初始状态I $\in S$（游戏）
- 终止位置T $\in S$（游戏的终止状态：游戏结束的状态）
- 后继函数（一个接收状态为输入，返回通过某些动作可以到达的状态的函数）
- 效益（Utility）或收益（payoff）函数V：T $\rightarrow \boldsymbol{R}$.（将终止位置映射到实数的函数，表示每个终止位置对玩家A有多有利和对玩家B有多不利）

两玩家零和博弈的直观介绍

- 玩家交替行动（从玩家A，或玩家Max开始）
  - 当到达某个终止状态$t \in T$时游戏结束
- 一个游戏状态：一个（状态-玩家）对
  - 告诉当前是哪个状态，轮到哪个玩家行动
- 效益函数和终止状态代替原来的目标状态
  - 玩家A或Max希望最大化终止状态的效益
  - 玩家B或Min希望最小化终止状态的效益
- 另一种解读
  - 在终止状态t是，玩家A或Max获得了$V(t)$的收益，玩家B或Min获得了$-V(t)$的收益
  - 这就是为何称为“零和”

Nim问题：非正式的描述

1. 开始时有一定数量的几堆火柴
2. 每一回合每个玩家可以移走任意数目的火柴
3. 移走最后的火柴的玩家则输

- 在II-Nim问题中，每个人有两堆火柴，每堆有2根火柴

S =			
	(_ , _)-A	(_ , i)-A	(_ , ii)-A
	(i , _)-A	(i , i)-A	(i , ii)-A
	(ii , _)-A	(ii , i)-A	(ii , ii)-A
	(_ , _)-B	(_ , i)-B	(_ , ii)-B
	(i , _)-B	(i , i)-B	(i , ii)-B
	(ii , _)-B	(ii , i)-B	(ii , ii)-B

根据对称性，一些状态是等价的（比如(_,ii)-A和(ii,_)-A）。可以把这些等价的状态合并为1个（即规定左边的火柴堆不多于右边的火柴堆）

Nim问题：非正式的描述

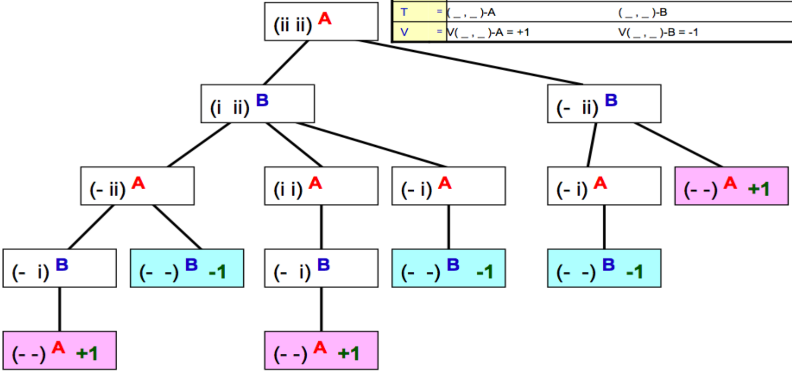
II-Nim

S	=	a finite set of states (note: state includes information sufficient to deduce who is due to move)	(_ , _)-A (_ , i)-A (_ , ii)-A (i , i)-A (i , ii)-A (ii , ii)-A (_ , _)-B (_ , i)-B (_ , ii)-B (i , i)-B (i , ii)-B (ii , ii)-B	
I	=	the initial state	(ii , ii)-A	
Succs	=	a function which takes a state as input and returns a set of possible next states available to whoever is due to move	Succs(_,i)-A = { (_,_)-B } Succs(_,ii)-A = { (_,_)-B , (_,i)-B } Succs(i,i)-A = { (_,i)-B } Succs(i,ii)-A = { (_,i)-B (_,ii)-B (i,i)-B} Succs(ii,ii)-A = { (_,ii)-B , (i,ii)-B }	Succs(_,i)-B = { (_,_)-A } Succs(_,ii)-B = { (_,_)-A , (_,i)-A } Succs(i,i)-B = { (_,i)-A } Succs(i,ii)-B = { (_,i)-A , (_,ii)-A (i,i)-A } Succs(ii,ii)-B = { (_,ii)-A , (i,ii)-A }
T	=	a subset of S. It is the terminal states	(_ , _)-A	(_ , _)-B
V	=	Maps from terminal states to real numbers. It is the amount that A wins from B.	V(_ , _)-A = +1	V(_ , _)-B = -1

Nim问题：非正式的描述

II-Nim Game Tree

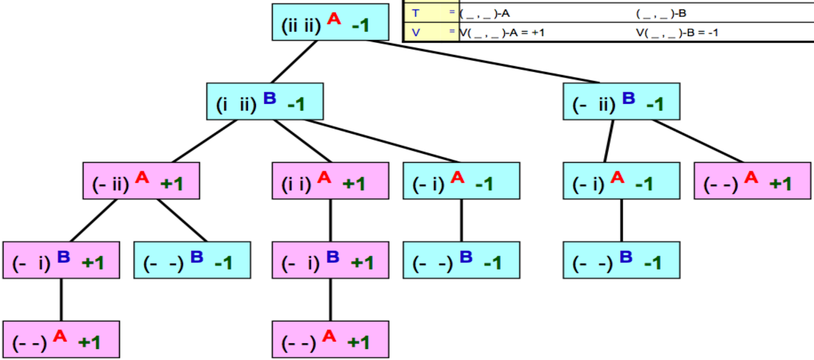
S	=	(_ , _)-A (_ , i)-A (_ , ii)-A (i , i)-A (i , ii)-A (ii , ii)-A (_ , _)-B (_ , i)-B (_ , ii)-B (i , i)-B (i , ii)-B (ii , ii)-B	
I	=	(ii , ii)-A	
Succs	=	Succs(_,i)-A = { (_,_)-B }	Succs(_,i)-B = { (_,_)-A }
		Succs(_,ii)-A = { (_,_)-B , (_,i)-B }	Succs(_,ii)-B = { (_,_)-A , (_,i)-A }
		Succs(i,i)-A = { (_,i)-B }	Succs(i,i)-B = { (_,i)-A }
		Succs(i,ii)-A = { (_,i)-B (_,ii)-B (i,i)-B}	Succs(i,ii)-B = { (_,i)-A , (_,ii)-A (i,i)-A }
		Succs(ii,ii)-A = { (_,ii)-B , (i,ii)-B }	Succs(ii,ii)-B = { (_,ii)-A , (i,ii)-A }
T	=	(_ , _)-A	(_ , _)-B
V	=	V(_ , _)-A = +1	V(_ , _)-B = -1



Nim问题：非正式的描述

II-Nim Game Tree

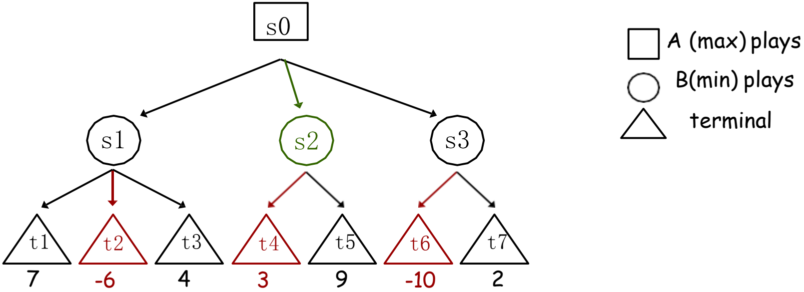
S =	(_ , _)-A (_ , i)-A (_ , ii)-A (i , i)-A (i , ii)-A (ii , ii)-A (_ , _)-B (_ , i)-B (_ , ii)-B (i , i)-B (i , ii)-B (ii , ii)-B	
I =	(ii , ii)-A	
Succs =	Succs(_,i)-A = { (_,_)-B } Succs(_,ii)-A = { (_,_)-B , (_,i)-B } Succs(i,i)-A = { (_,i)-B } Succs(i,ii)-A = { (_,i)-B (_,ii)-B (i,i)-B} Succs(ii,ii)-A = { (_,ii)-B , (i,ii)-B }	Succs(_,i)-B = { (_,_)-A } Succs(_,ii)-B = { (_,_)-A , (_,i)-A } Succs(i,i)-B = { (_,i)-A } Succs(i,ii)-B = { (_,i)-A , (_,ii)-A (i,i)-A } Succs(ii,ii)-B = { (_,ii)-A , (i,ii)-A }
T =	(_ , _)-A	(_ , _)-B
V =	V(_ , _)-A = +1	V(_ , _)-B = -1



MiniMax策略

- 假设对方能总是做出最优的行动
  - 己方总是做出能最小化对方获得的收益的行动
  - 通过最小化对方的收益，可以最大化己方的收益
- 注意到如果已经知道在某些情况下对方无法做出最优的行动，那么可能存在比MiniMax更好的策略（也就是说，有其他的策略可以获得更多的收益）

MiniMax策略的收益



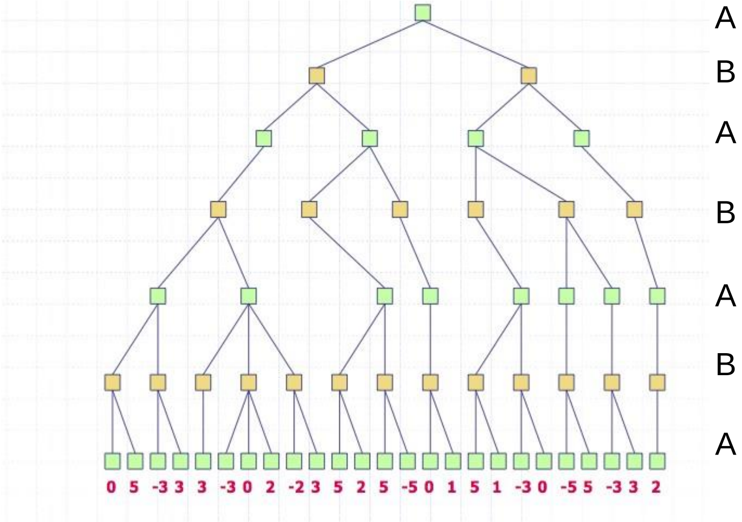
终止状态具有一个效益值（V）。对于在非终止状态时的“效益值”，我们可以通过假设每个玩家都做出对自己最优的行动来计算得到。

MiniMax策略

- 构建完整的博弈树（每个叶子节点都表示终止状态）
  - 根节点表示起始状态，边表示可能的行动，之类的
  - 每个叶子节点（终止状态）都标记了对应的效益值
- 反向传播效益值$U(n)$
  - 每个叶子节点t的U(t)值是预定义好的（算法输入的一部分）
  - 假如节点n是一个Min节点：
    - $U(n) = \min\{U(c): c$是n的子节点$\}$
  - 假如节点n是一个Max节点：
    - $U(n) = \max\{U(c): c$是n的子节点$\}$

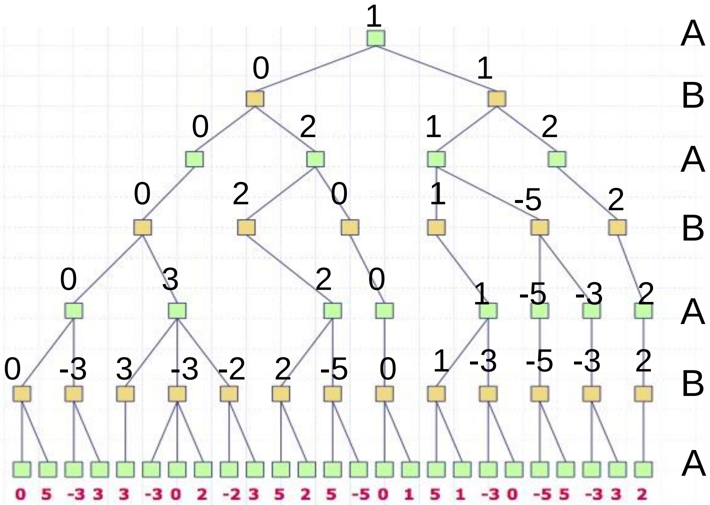
MiniMax策略

- 练习：计算出下面博弈树中的每个节点的理论效益值

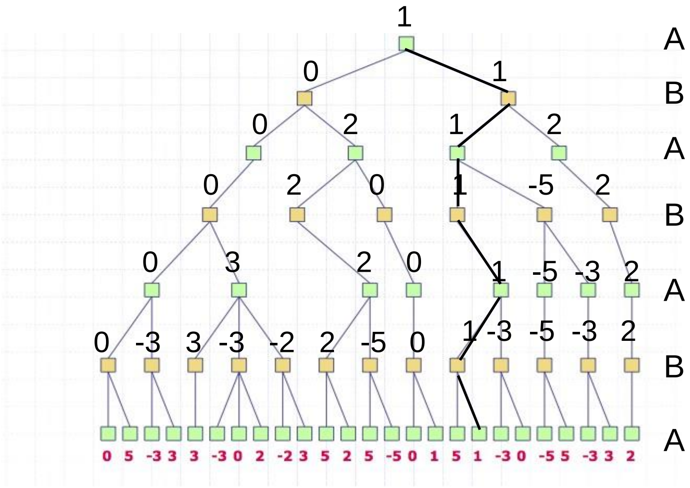


MiniMax策略

- 问题：假如每个玩家都按照对自己最优的策略行动，博弈树中的哪条路径会是游戏进行的过程？



MiniMax策略


1
A
0 1
B
0 2 1 2
A
0 2 0 1 -5 2
B
0 3 2 0 1 -5 -3 2
A
0 -3 3 -3 -2 2 -5 0 1 -3 -5 -3 2
B
A
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

MiniMax策略的深度优先实现

- 我们希望能构建整个博弈树并且记录每个玩家决策所需的值
- 但是博弈树的大小是指数增长的
- 之后我们会看到，其实知道整个博弈树并不必要
- 为了解决博弈树太大的问题，我们需要找到深度优先搜索算法来实现MiniMax
- 通过深度优先搜索我们可以找到MAX玩家的下一步动作（对于MIN玩家也是类似）
- 这样就可以避免记录指数级大小的博弈树，只需要计算我们需要的动作

MiniMax策略的深度优先实现

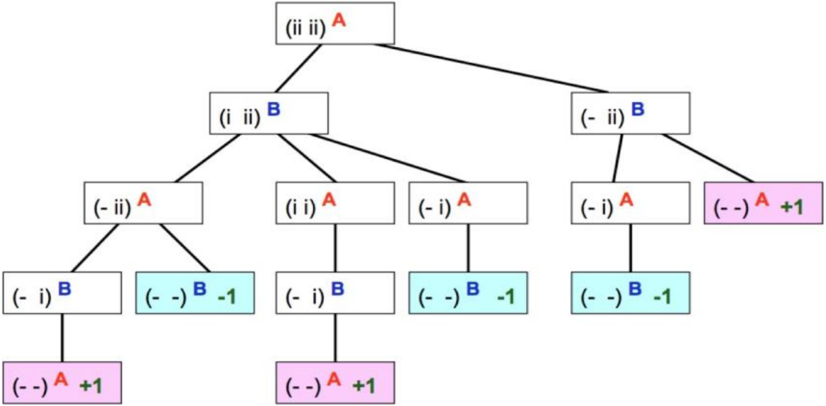
```
DFMiniMax(n, Player)  //return Utility of state n given that
                      //Player is MIN or MAX

If n is TERMINAL
Return V(n)  //Return terminal states utility
             //(V is specified as part of game)

//Apply Player’s moves to get successor states.
ChildList = n.Successors(Player)
If Player == MIN
    return minimum of DFMiniMax(c, MAX) over c ∈ChildList  Else //Player is MAX
    return maximum of DFMiniMax(c, MIN) over c ∈ ChildList
```

- 这个算法要能运行要求博弈树的深度是有限的
- 深度优先搜索的优点是：空间复杂度低

剪枝

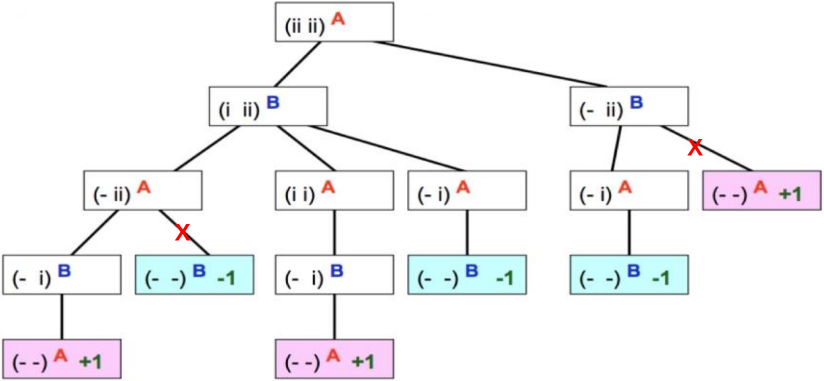
- MiniMax的决策没有必要计算整个博弈树
- 假设使用深度优先来生成博弈树
  - ✓ 只要计算节点n的一部分子节点就可以确定在MiniMax策略中我们不会考虑走到节点n了
  - ✓ 如果已经确定节点n不会被考虑，那么也就不用继续计算n的子节点了
- 有两种类型的剪枝：
  - ✓ 对Max节点的剪枝 (α-cuts)
  - ✓ 对Min节点的剪枝 (β-cuts)

剪枝



- 假设叶子节点（终止节点）的取值只有1和-1，并且使用深度优先搜索实现MiniMax策略。我们应该在哪里进行剪枝呢？

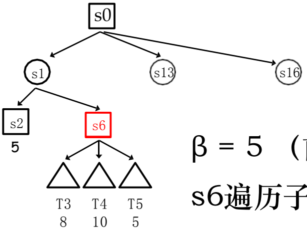
剪枝

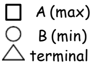


- 如果有的状态已经快到当前玩家胜利的结局了，那就不用继续评估其他子节点了，这样可以把博弈树剪枝很多

对Max节点的剪枝 (α-cuts)

- 在Max节点n：
- 设β是n被遍历过的兄弟节点中的最低值（n左边的兄弟节点就是已经被遍历过了）
- 设α是n被遍历过的子节点中的最高值（随着子节点的遍历而改变）





β = 5 （前面只遍历了一个兄弟节点）

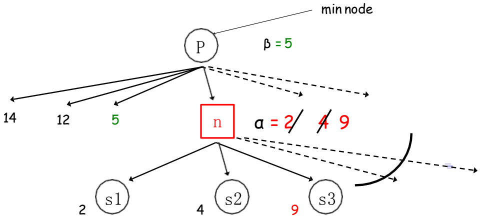
s6遍历子节点的时候alpha值的变化:

α= 8; α= 10; α= 10

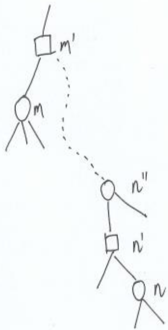
对Max节点的剪枝 (α-cuts)

在Max节点n的时候，如果α值变得 ≥ β的时候，就可以停止遍历n的子节点了

前面的Min节点不会来到通过n节点的父母来到n节点这个状态的，因为Min一定会选择n节点的值更小的兄弟节点



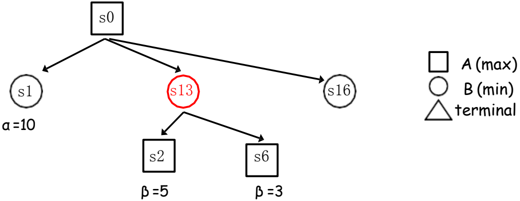
Alpha-beta剪枝的泛化



- 如果U (m) ⩾ U (n), 那么n节点可以被剪枝
- 使用归纳法来证明
- Base case: m' = n'. 显然n节点可以被剪枝
- 接下来是归纳法的步骤:
- Case 1: U (n') > U (n). 那么n节点可以剪枝
- Case 2: U (n') = U (n). 那么 U (m) ⩾ U (n) = U (n') ⩾ U (n''). 根据归纳法，n''节点可以剪枝，所以n节点也可以剪枝

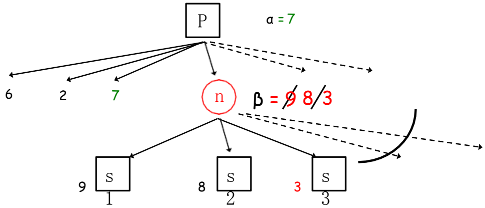
对Min节点的剪枝 (β-cuts)

- 在Min节点n:
- 设α 到现在为止n节点的兄弟节点中值最高的 (在评估节点n时是固定的)
- 设β是到现在为止节点n的子节点中值最低的 (changes as children examined)



对Min节点的剪枝 (β-cuts)

- 如果β 变得 ⩽ α 那么可以停止扩展n的子节点
- Max节点一定不会选择n节点，因为它会优先选择n节点值更高的兄弟节点



- 放大一点来说，在Min节点 n，如果β 变得 ⩽ 某个Max祖先节点的α值，那么n节点的扩展就可以停止了

Alpha-beta剪枝的实现

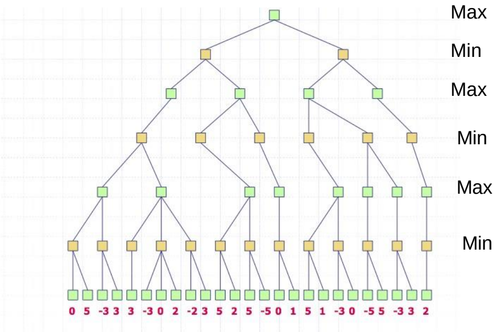
```
AlphaBeta(n,Player,alpha,beta) //return Utility of state
If n is TERMINAL
    return V(n) //Return terminal states utility  ChildList =
n.Successors(Player)
If Player == MAX
 for c in ChildList
    alpha = max(alpha, AlphaBeta(c,MIN,alpha,beta))
    If beta <= alpha
        break  return alpha
Else //Player == MIN
          for c in ChildList
    beta = min(beta, AlphaBeta(c,MAX,alpha,beta))
    If beta <= alpha
        break  return beta
```

- 当AlphaBeta(n, Player, alpha, beta) 被调用的时候，alpha 是n的祖先Max节点中alpha值的最大值，而beta是n节点的祖先Min节点的beta值的最小值
- 第一次调用：AlphaBeta(START-NODE, Player, -infinity, +infinity)

一步步运行Alpha-beta剪枝

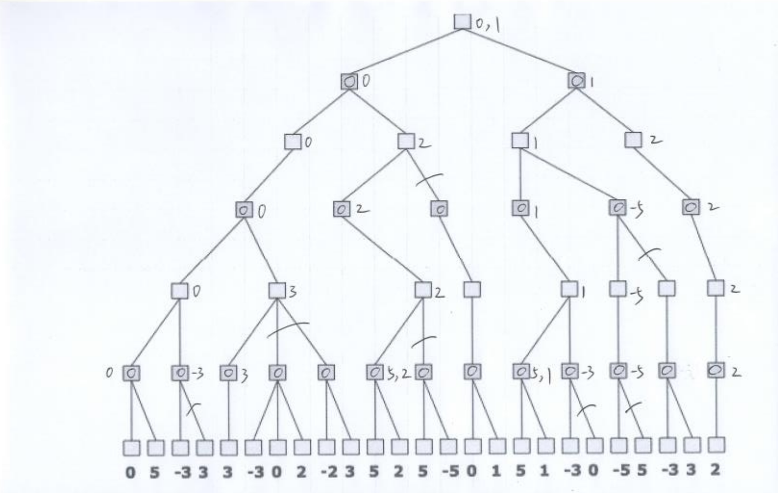
- 记录Max节点alpha值的变化和Min节点beta值的变化
- Max节点如果alpha值大于任何祖先Min节点的beta值，就进行alpha剪枝
- Min节点如果beta值小于任何祖先Max节点的alpha值，就进行beta剪枝

示例



假设从左到右扩展节点，哪部分的计算可以忽略（剪枝）呢?

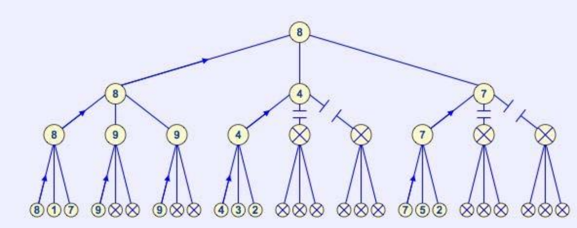
示例



Alpha-beta剪枝的有效性

- 没有剪枝的话，需要扩展O(b^D)个节点，与普通的MiniMax算法一样
- 但是，如果节点遍历的顺序是最优的（即最优的动作被优先遍历），使用alpha-beta剪枝需要遍历的节点数是 O($b^{D/2}$).
- 这意味着我们理论上可以搜索决策树的两倍深度
- 在Deep Blue程序中, alpha-beta剪枝使每个节点的分支系数由35降为6

最优情况的一个示例



设博弈树的宽度是B（图中是B=3）。第一层的有效分支因子是B，第二层的有效分支因子是 1. 第三层的有效分支因子是 B. 以此类推….

实际操作中的问题

真实得游戏很难把整个博弈树都枚举出来

- e.g., 象棋得分支因子大约是35，那么博弈树就会
  有 2,700,000,000,000,000 个节点
- 即使使用alpha-beta剪枝也收效甚微

必须限制搜索树的深度

- 实际游戏中根本无法扩展到终止节点
- 因此需要启发式地计算（非终止节点）叶子节点的值
- 这样的启发式方法被称为评价函数

评价函数：基本要求

- 必须使得终止节点的排序和原来的效用函数一致
- 计算不能太耗时
- 对于非终止节点，评价函数需要与这个节点实际能获得胜利的概率强相关.

如何设计评价函数

- 利用状态中的特征值：如象棋中，棋盘上白兵的数量，黑兵的数量，白皇后的数量等等
- 这些特征值综合到一起，可以定义出状态的类别：特征值一样的状态分为一类
- 这些类别的状态里，有的能走向胜利，有的会导致平局，有的会跌向失败
- e.g., 比方说，根据经验某个类别里的状态会由72%的状态获得胜利，有20%的状态会输，有8%的状态会平局
- 那么对于这个状态的评价应该是效用函数的期望 $0.72 \cdot 1 + 0.20 \cdot (-1) + 0.08 \cdot 0 = 0.52$.
- 但是，状态有太多的类别了

如何设计评价函数

- 大多数的评价函数会分别计算各个特征的数值贡献，之后再进行结合
- e.g., 象棋中，每个兵评价为1，马或象评价为3，车评价为5，皇后评价为9
- 数学上，一个加权评价函数为

$$Eval(s) = w_1 \cdot f_1(s) + \cdots + w_n \cdot f_n(s) = \sum_{i=1}^{n} w_i \cdot f_i(s)$$

- Deep Blue用了超过8000个特征
- 这里要考虑一个很强的假设：所有特征的贡献都是独立于其他特征的
- 这个假设不一定成立，因此有时也会用非线性的组合

如何设计评价函数

- 这些特征和权重都不是象棋规则的一部分
- 它们是由人类下棋的经验而来的
- 假如没有这方面的经验，则评价函数里的权重可以通过机器学习的技巧估计得到

Alpha-beta剪枝：井字棋

- 定义X_n为只有n个X而没有O的行，列或对角线的数量
- 同样O_n是只有n个O而没有X的行，列或对角线的数量
- 效用函数对于任何$X_3 = 1$的状态都赋值为+1
- 并且对于任何$\mathrm{O}_3 = 1$的状态都赋值为-1
- 其他所有的终止状态效用都为0
- 对于非终止状态，我们使用线性评价函数

$$Eval(s) = 3X_2(s) + X_1(s) - (3O_2(s) + O_1(s)).$$

Alpha-beta剪枝：井字棋

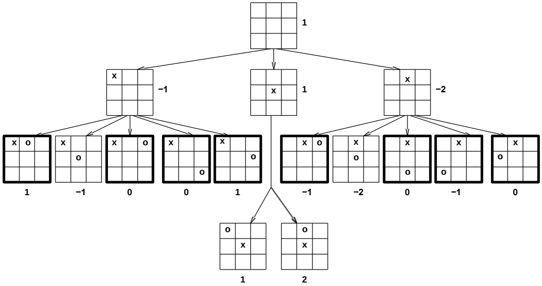
从空白棋盘开始建立博弈树，把树扩展到第2层

在博弈树第二层的每个节点上标记当前的评价函数值

使用MiniMax算法，标出第0和1层的节点的评价值

在最优节点最先生成的假设下，把第二层会被alpha-beta剪枝的节点圈出来

Alpha-beta剪枝：井字棋

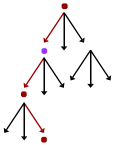


对于大型搜索的问题

- 问题：即使在标准搜索中，也无法把树扩展到终止节点
  - 我们往往无法期待A*能通过扩展所有的边界节点来达到目标节点
  - 因此我们经常通过限制往前看的能力，并且在不知道路径是否能达到目标的情况下就做出行动
  - 这种方法被称为在线或实时搜索

- 在这种情况下，我们使用启发式函数不仅是为了引导搜索，也为了得到可以前进的动作
  - 一般情况下，这种方法是无法保证最优性的，但是可以极大地降低时间和空间消耗

实时搜索

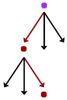


1.运行A* (或其他更好的搜索算法) 直到一定要进行动作的时候或内存不足的时候

Note: 这个时候搜索树还没扩展到叶子节点

2.使用评价函数$f(n)$ 决定要沿着哪条路径

选看起来好的哪条路径 (比方说左图中红色那条路径).



3.沿着看起来好的路径(红色路径)做出一个动作

4.重启搜索算法，把当前状态作为根节点构建搜索树(我们可以把第一次搜索的路径保存起来).