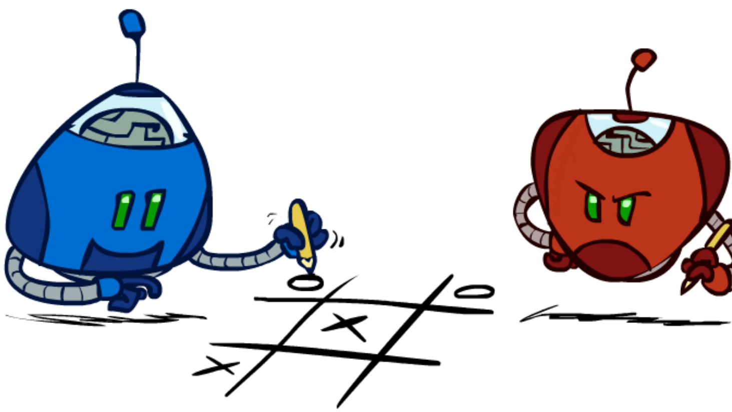
人工智能导论：对抗性的搜索 (Adversarial Search)

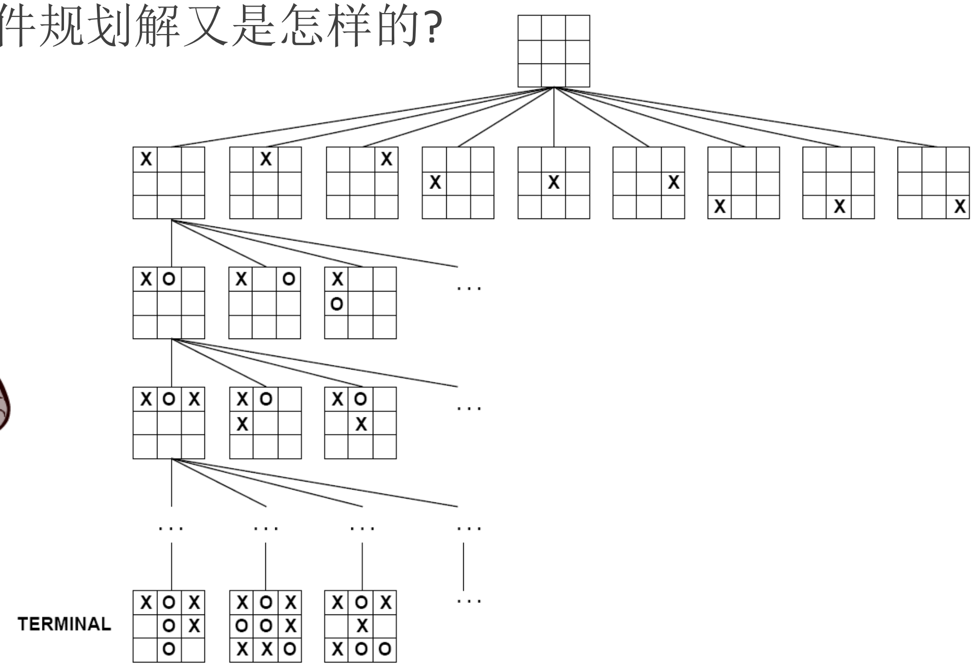
海南大学

井字游戏

如何用与或搜索 （AND-OR search）来求解井字游戏?

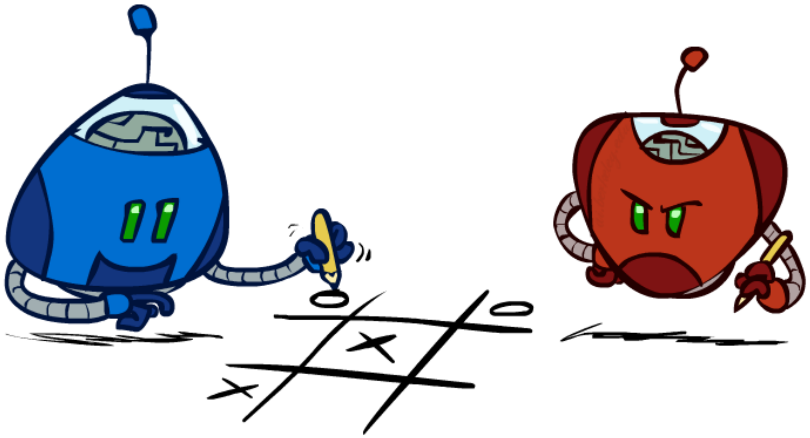
- 如果你想赢得这个游戏，你的条件规划解是怎样的?
- 如果你想至少打平这个游戏，那么条件规划解又是怎样的?





内容提纲

- 历史和展望
- 零和游戏 (最小最大值)
- 评估函数
- 搜索效率 (α-β 剪枝)
- 机遇博弈 (期望最大值)



计算机游戏/比赛的当前水平

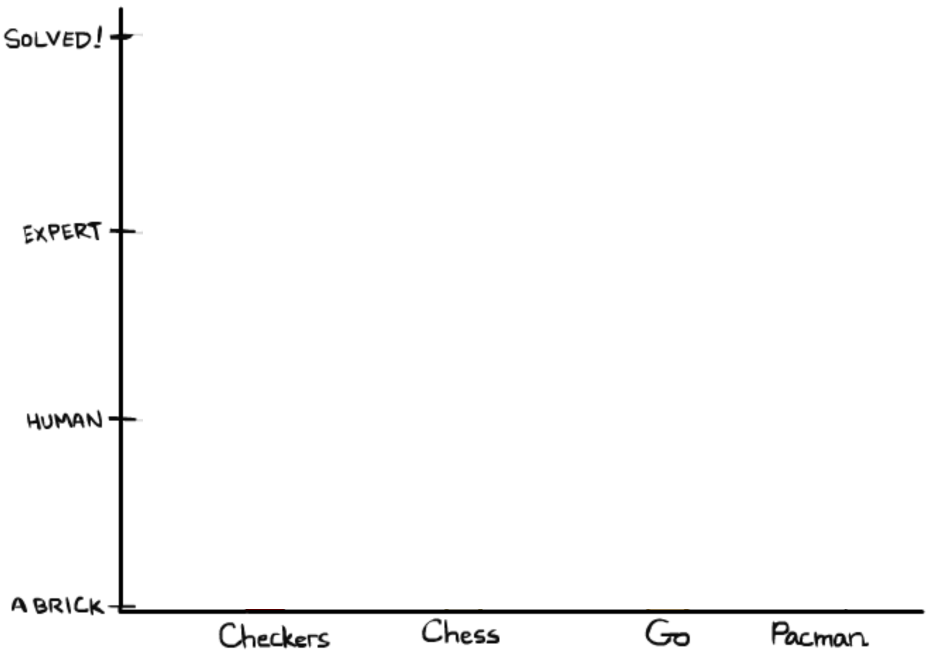
国际跳棋
- 1950年，第一个计算机跳棋程序
- 1994年，计算机击败人类冠军(Marion Tinsley)
- 2007年，游戏搜索问题被解决。总共有39万亿个终局状态

国际象棋
- 1945-1960年，计算机国际象棋程序
- 1997年，象棋机器深蓝击败人类冠军

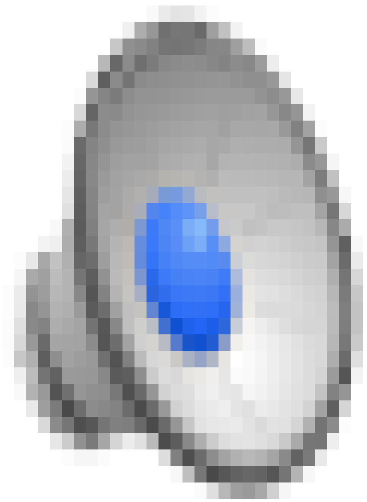
围棋
- 1968年，计算机围棋程序出现
- 2005-2014年，蒙特卡罗树搜索提高了程序性能；当前水平能击败高水平的业余选手，和部分职业选手
- 2016年，Google’s alphaGO 在5盘比赛中击败人类高手

吃豆子（Pacman）

人工智能的博弈水平



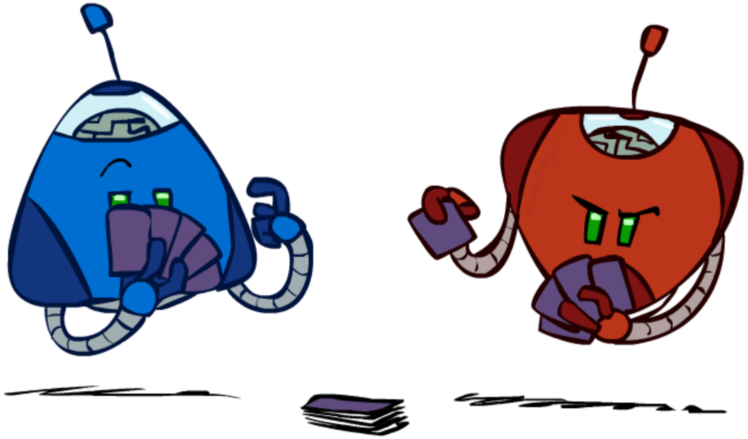
视屏展示：神秘的Pacman智能体



游戏类型

描述角度

- 环境变换确定的，或不确定的？
- 信息完全可观察的？
- 一个，两个，或多个玩家？
- 轮流的，或是即时的？
- 零和的？



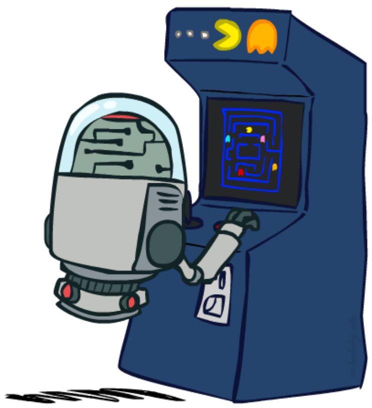
算法的目的是计算一个依情况而定的行动计划（策略），为每一个可能的事件推荐一个相应的行动。

人工智能所研究的“标准的”游戏比赛（games）

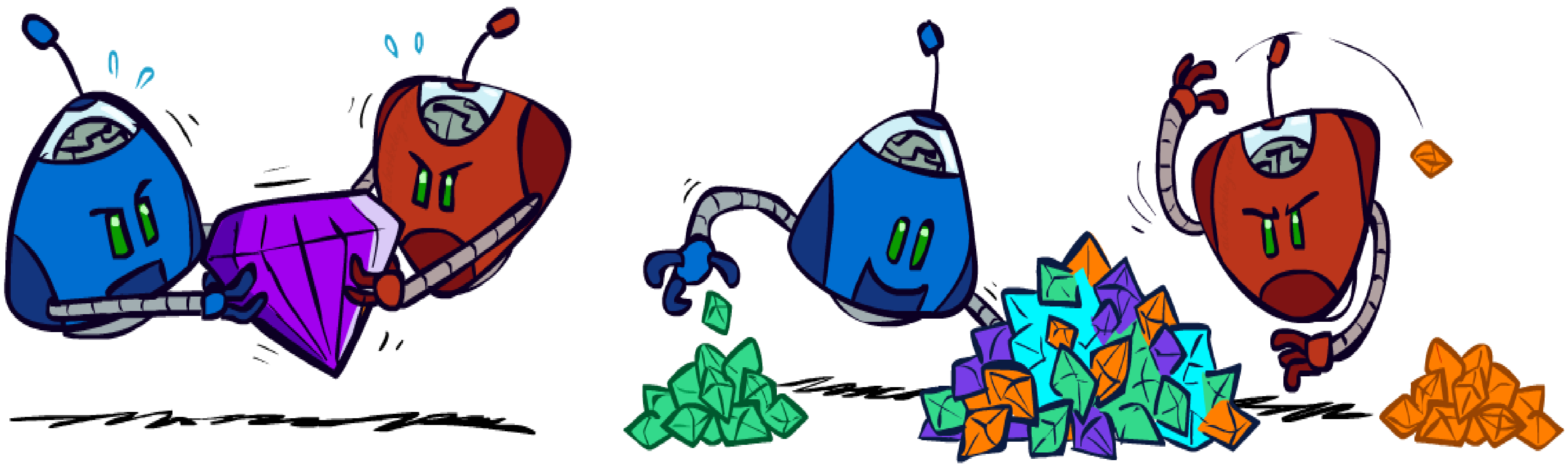
标准的游戏是，确定的，全局可观察的，轮流行动的，两人的，零和的。

游戏问题的模型建立：

- 初始状态: s_0
- 玩家: Player(s) 显示在当前状态轮到哪一个玩家行动
- 行动: Actions(s) 当前轮次的玩家可能的移动
- 状态转换模型: Result(s,a)
- 终局状态检测: Terminal-Test(s) 是否是终局
- 终局得分: Utility(s,p) 玩家 p 的得分
  - 或只用 Utility(s) 代表游戏一开始时首先移动的玩家的得分



零和游戏，以及和通常游戏的区别

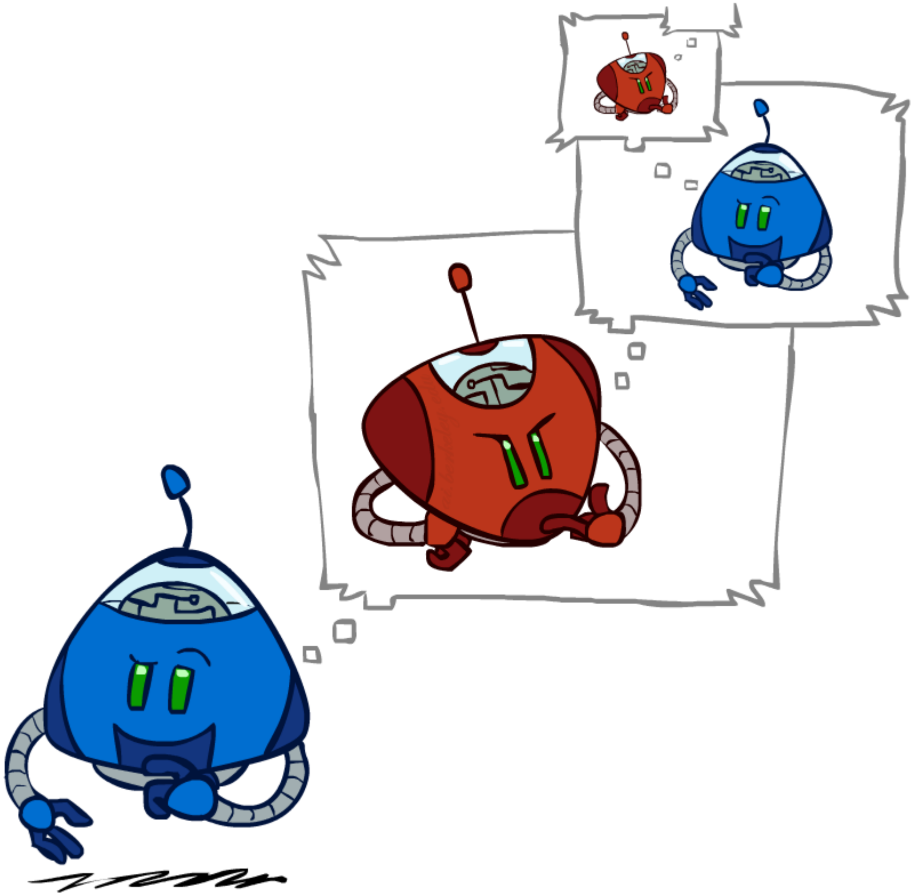


零和游戏

- 智能体竞争实现相反的利益
- 一方 ***最大化这个利益***, 另一方 ***最小化它***

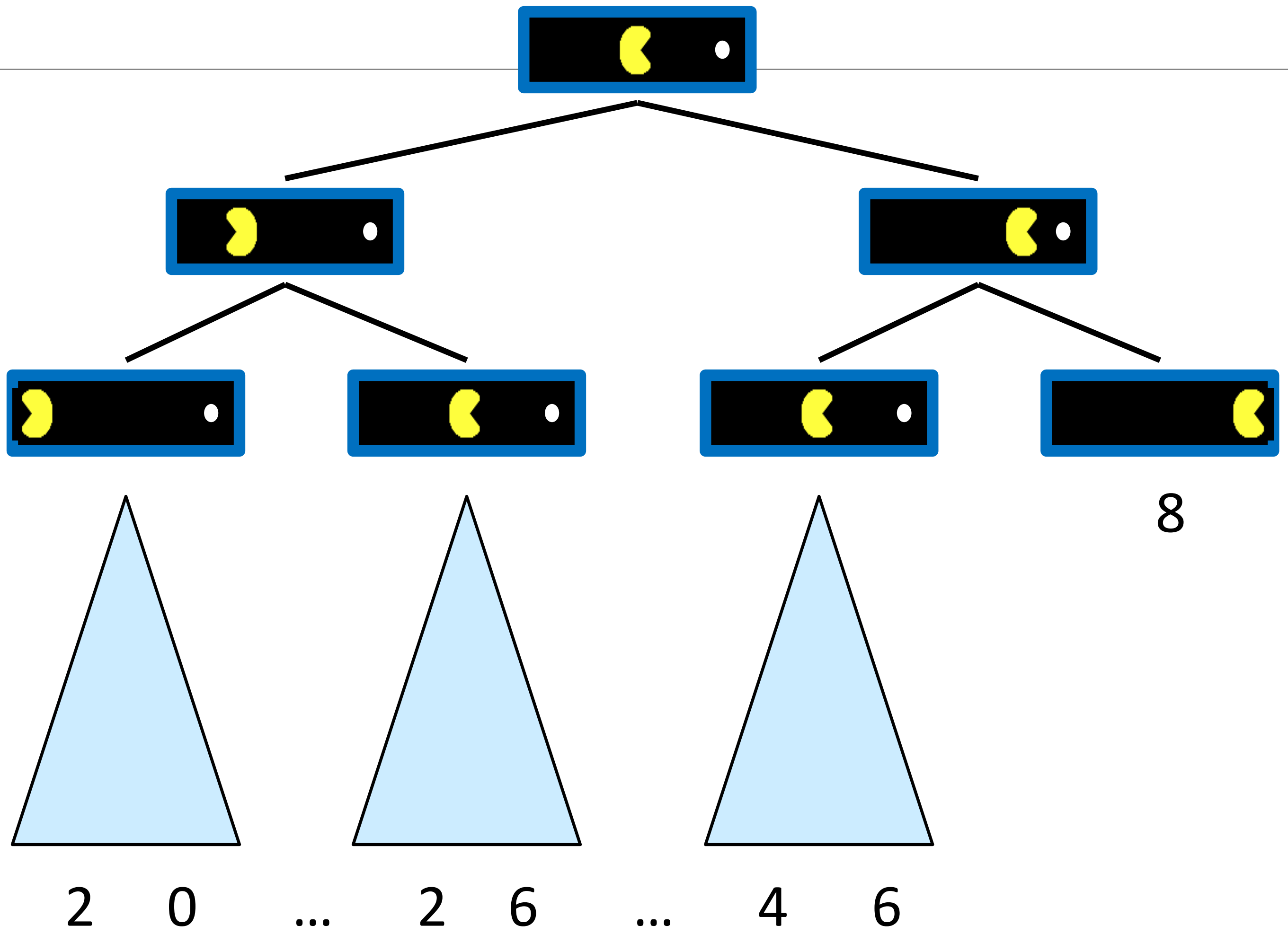
通常游戏

- 智能体有 ***独立的*** 利益
- 合作, 竞争, 联盟等相互关系, 都有可能



对抗性搜索

单一智能体搜索树

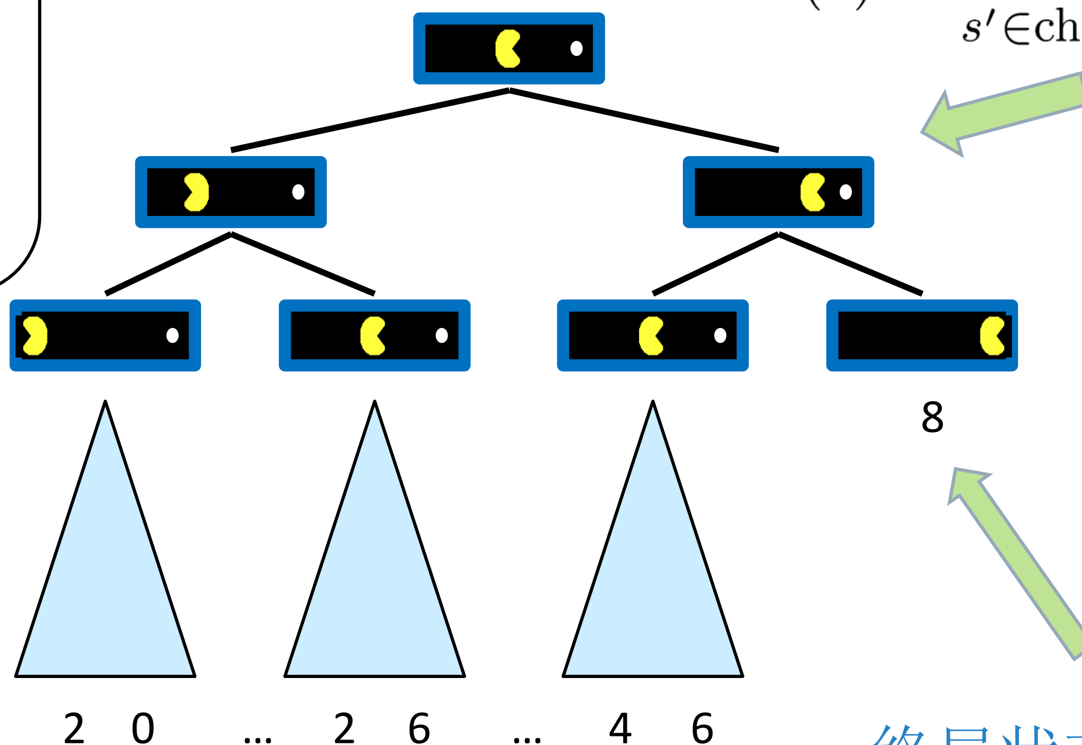


状态值（中间和终局状态值）

一个状态的值: 从这个状态往下走可能达到的最大利益值（结果）

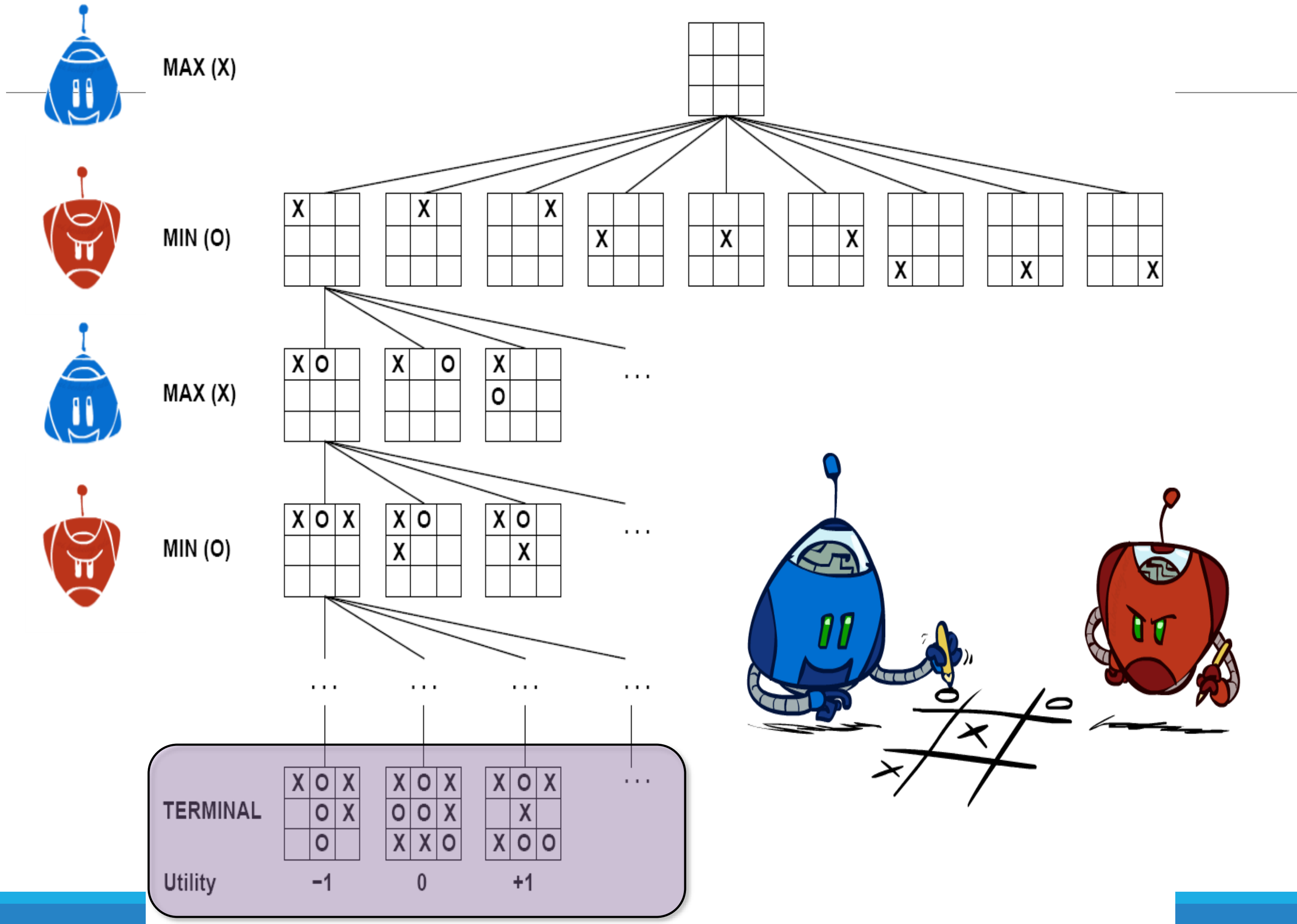
中间状态:

$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$



终局状态的值已知

井字游戏搜索树



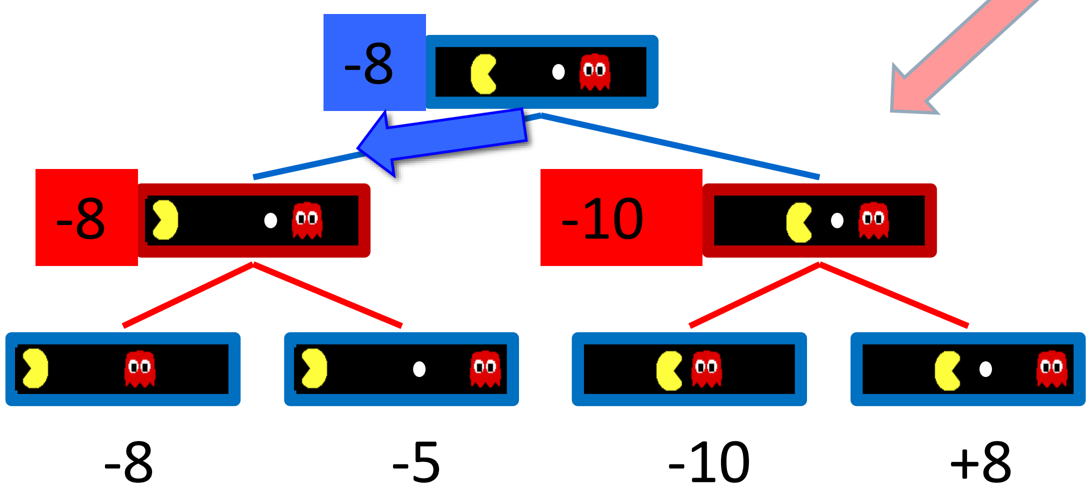
最小最大值（Minimax values）

MAX 节点: 智能体控制下的节点状态

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

MIN 节点: 对手控制下的节点状态

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



终局状态的值是已知的

最小最大算法(Minimax) 实现

深度优先搜索

Function 最小最大-决策(s) returns 一个行动

return 行动 a in Actions(s)，它能导致最大的

最小-值(Result(s,a))的返回值

Function 最大-值(s) returns 一个值

If 终局-检测(s) then return Utility(s)

初始化 v = -∞

for each a in Actions(s):

v = max(v, 最小-值(Result(s,a)))

return v

Function 最小-值(s) returns 一个值

If 终局-检测(s) then return Utility(s)

初始化 v = +∞

for each a in Actions(s):

v = min(v, 最大-值(Result(s,a)))

return v

$$V(s) = \max_{s' \in \mathrm{successors}(s)} V(s')$$

$$V(s') = \min_{s \in \mathrm{successors}(s')} V(s)$$

另一种实现方法

Function 最小最大-决策(s) returns 一个行动

return 行动 a in Actions(s)， 它能导致最大的 value(Result(s,a)) 的返回值

↕

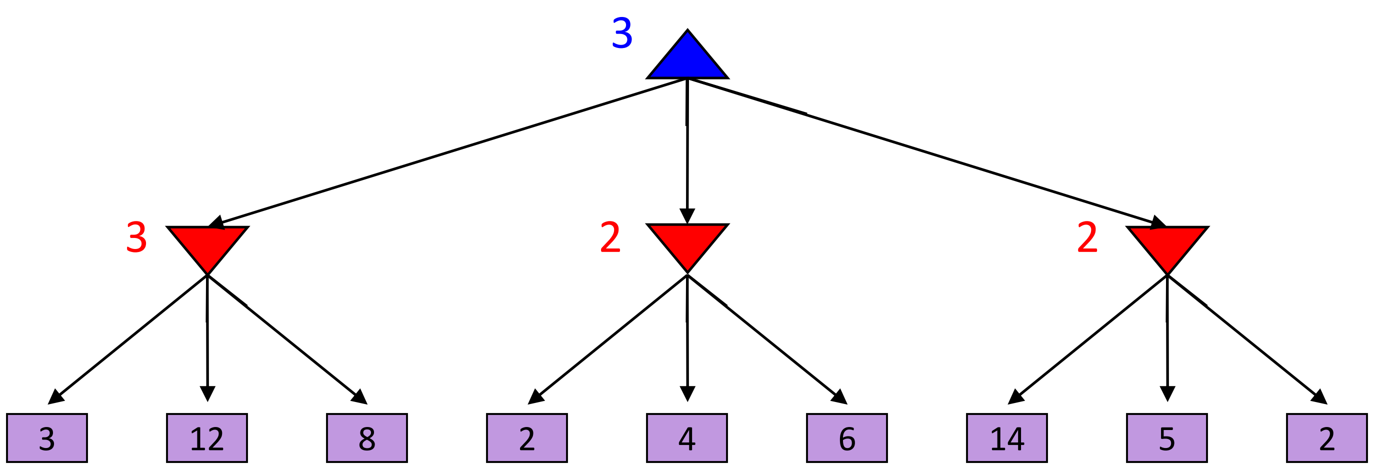
function value(s) returns 一个值

If 终局-检测(s) then return Utility(s)

if Player(s) = MAX then return $\max_{a \text{ in Actions}(s)}$ value(Result(s,a))

if Player(s) = MIN then return $\min_{a \text{ in Actions}(s)}$ value(Result(s,a))

最小最大值（Minimax）举例



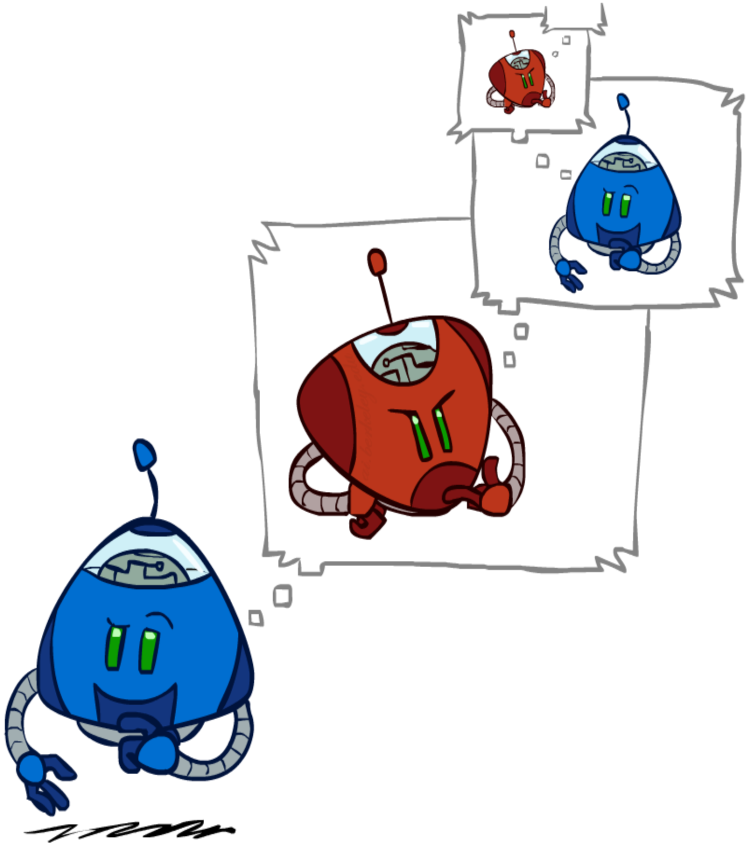
Minimax 的效率

Minimax 的效率?

- 深度优先穷尽搜索
- 时间复杂度: $O(b^m)$
- 空间复杂度: $O(bm)$

举例: 国际象棋, , $b \approx 35$, $m \approx 100$

- 找到准确解，不可行
- **有必要**探索整棵树吗? 人是如何下象棋的，是如何思考的？



资源有限 vs 庞大的搜索空间

资源局限

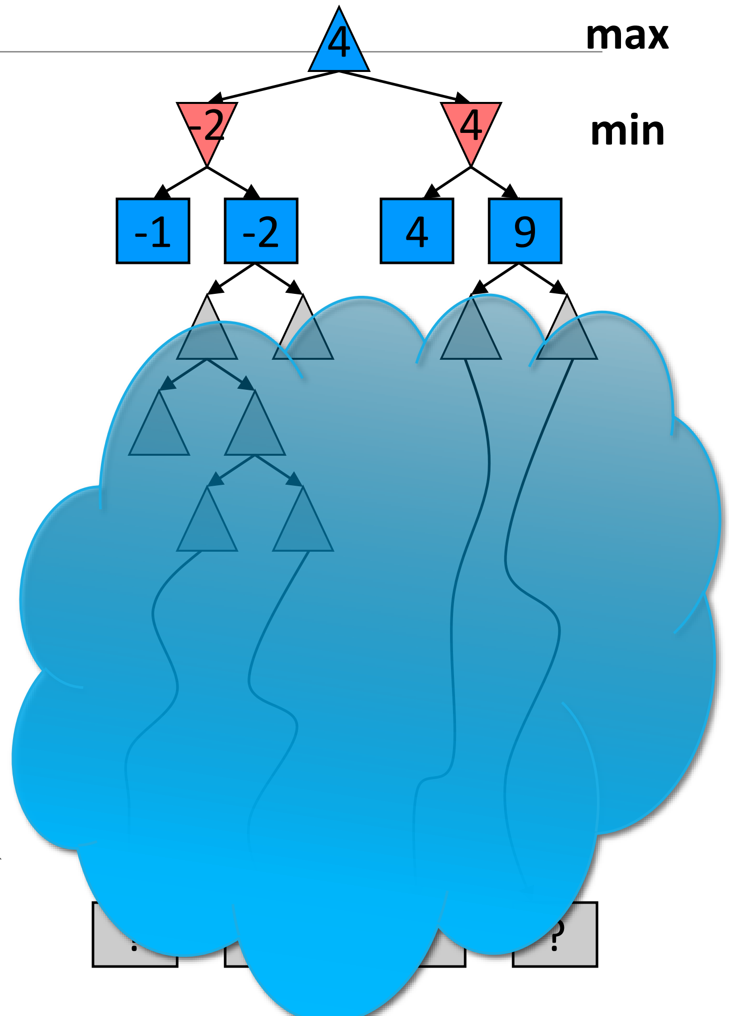
问题: 现实中, 几乎不能搜索到叶节点!

办法之一: 有界搜索加预测
- 搜索只到预定深度层次
- 使用 ***评估函数*** 预测搜索边界节点的值

失去了最优解的保证

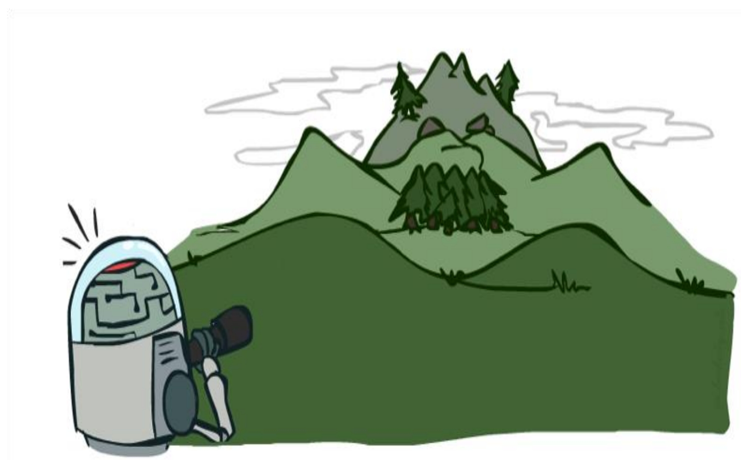
搜索的层次越多结果就越不相同

例如:
- 假设计算时间有100 秒, 每秒能探索1万个节点
- 所以每步能检查1百万个节点
- 在国际象棋中, b=~35 ， 搜索大致能达到搜索树的第4层 – 还是不够好

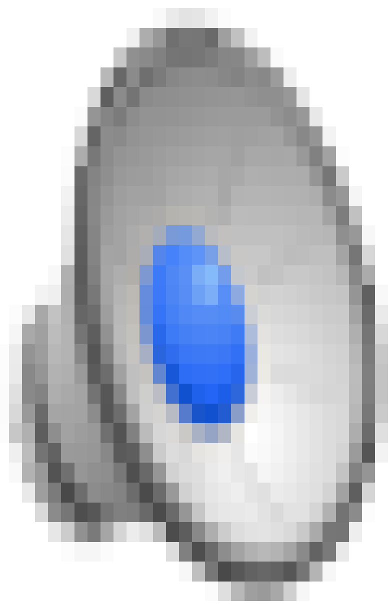


探索深度的重要性

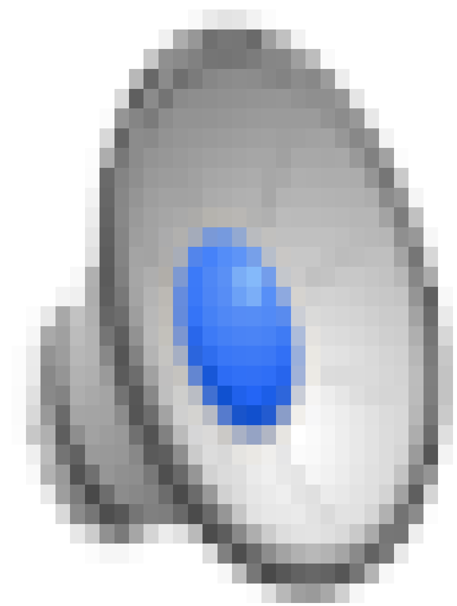
- 评估函数总是不完美的（不准确的）
- 通常，越深的搜索 => 越好的表现
- 或者说, 更深的搜索能够弥补相对不准确的评估函数
- 评估函数设计复杂度和计算复杂度之间的权衡



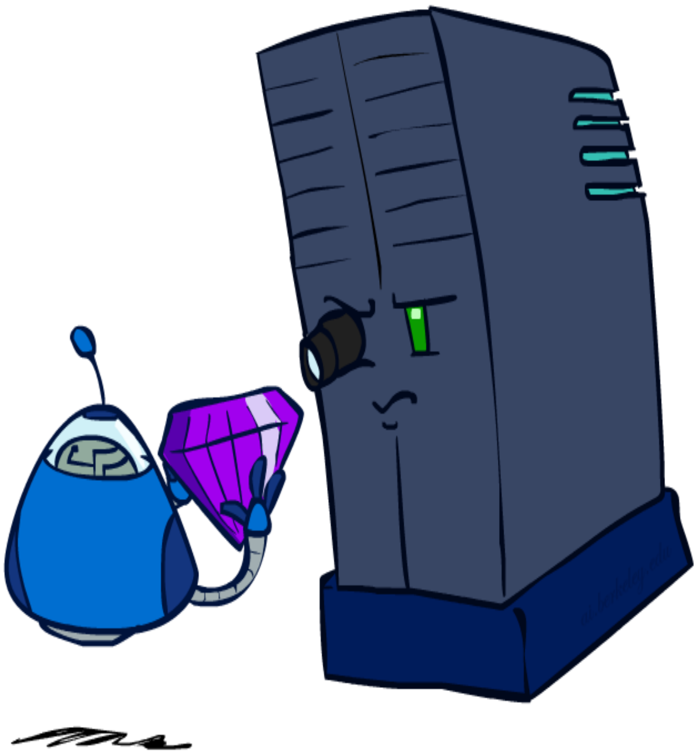
视屏演示：有限搜索深度 (深度=2)



视屏演示：有限搜索深度 (深度=10)

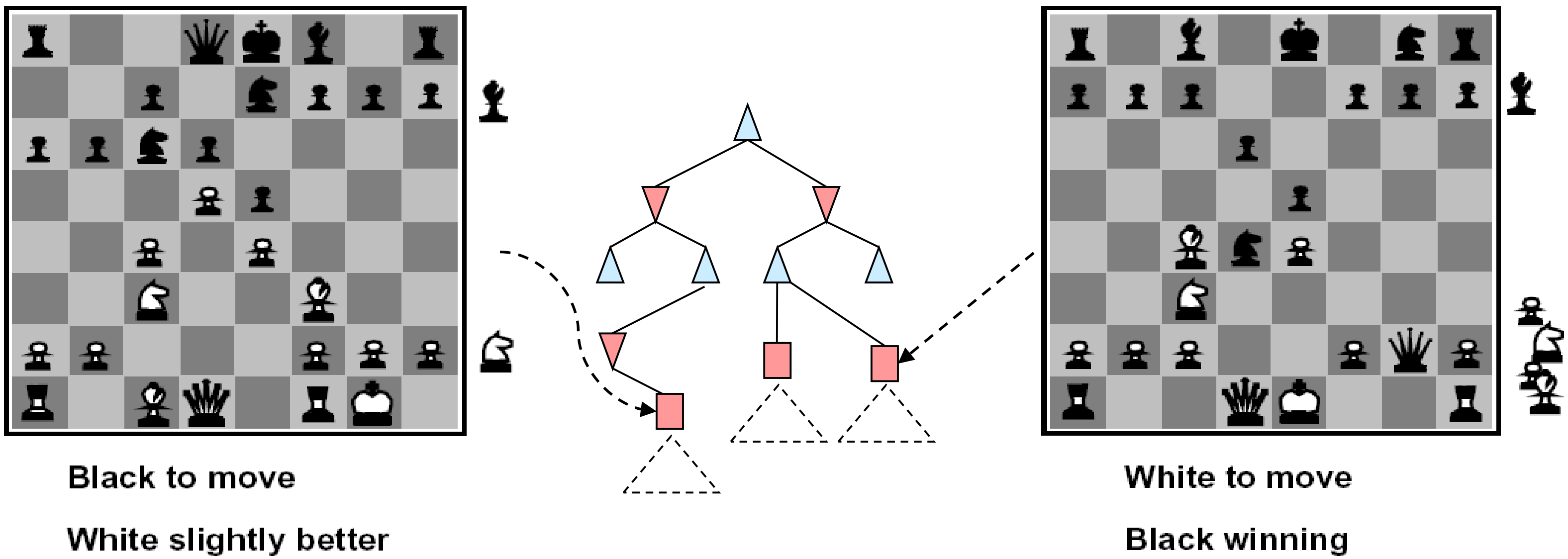


评估函数



评估函数

在一个限定深度搜索中，**用来给非终局**节点**状态**估值。



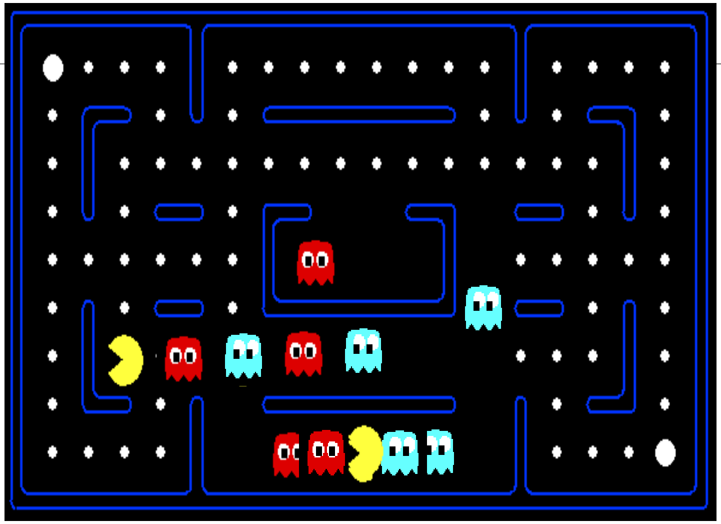
理想函数: 返回这个节点状态的实际最小最大值

实践中: 特征函数值的加权线性和:

- EVAL(s) = $w_1 f_1(s) + w_2 f_2(s) + ... + w_n f_n(s)$
- 例如 $w_1 = 9$, $f_1(s)$ = (白皇后数量 – 黑皇后数量), 等。

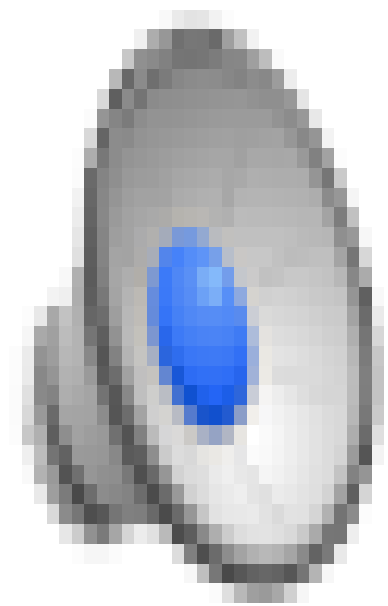
评估函数应作用在 ***沉寂*** 状态节点, 即其后继状态的评估值相对变化不大。

Pacman游戏状态评估

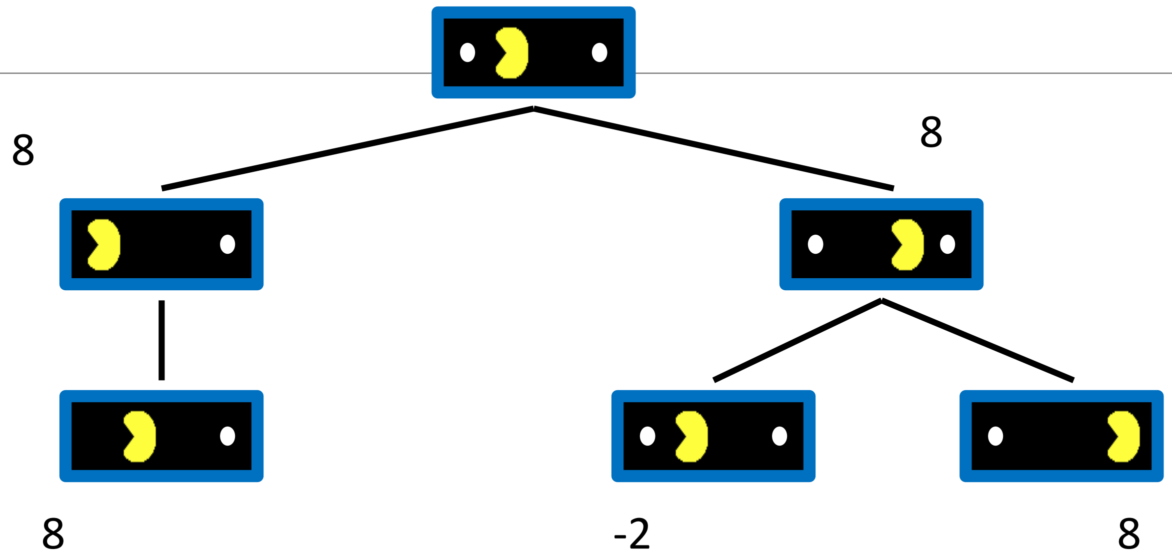


评估函数值应反映所处的状态局面。

评估函数实例演示：犹豫的情况(深度=2)



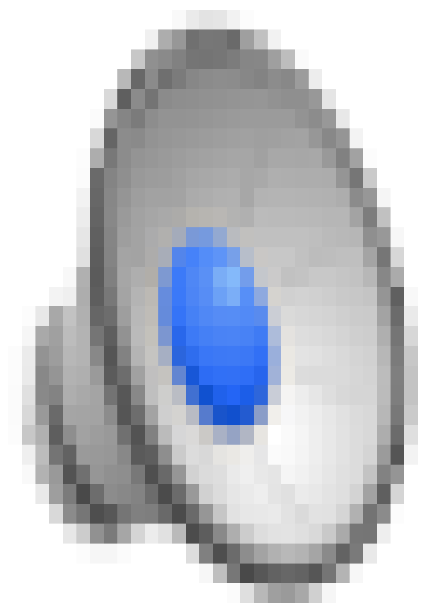
Pacman 为什么会犹豫？



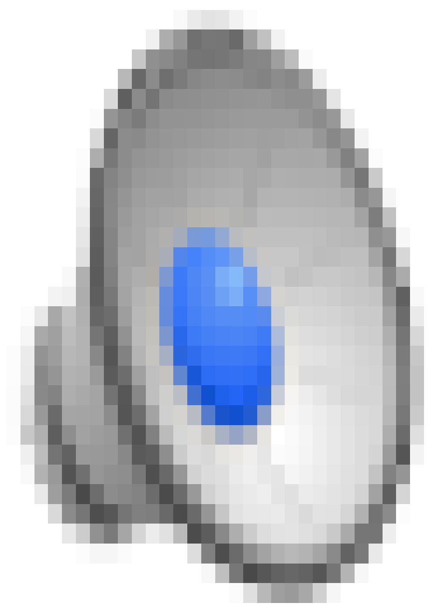
智能体重新规划时可能会面临的处境

- 在当前两步搜索的情况下，向左吃掉豆子的得分和向右移动的是一样的（后面可能会吃掉豆子）
- 得分是用最小最大值方法得到的
- 所以这个时候等待和吃掉豆子的选择结果似乎一样； 它有可能向右移动，然后下一轮重新规划时，可能又走回来了！

调整评估函数，解决这个问题 (深度=2)



演示：共同的评估函数可自动形成合作

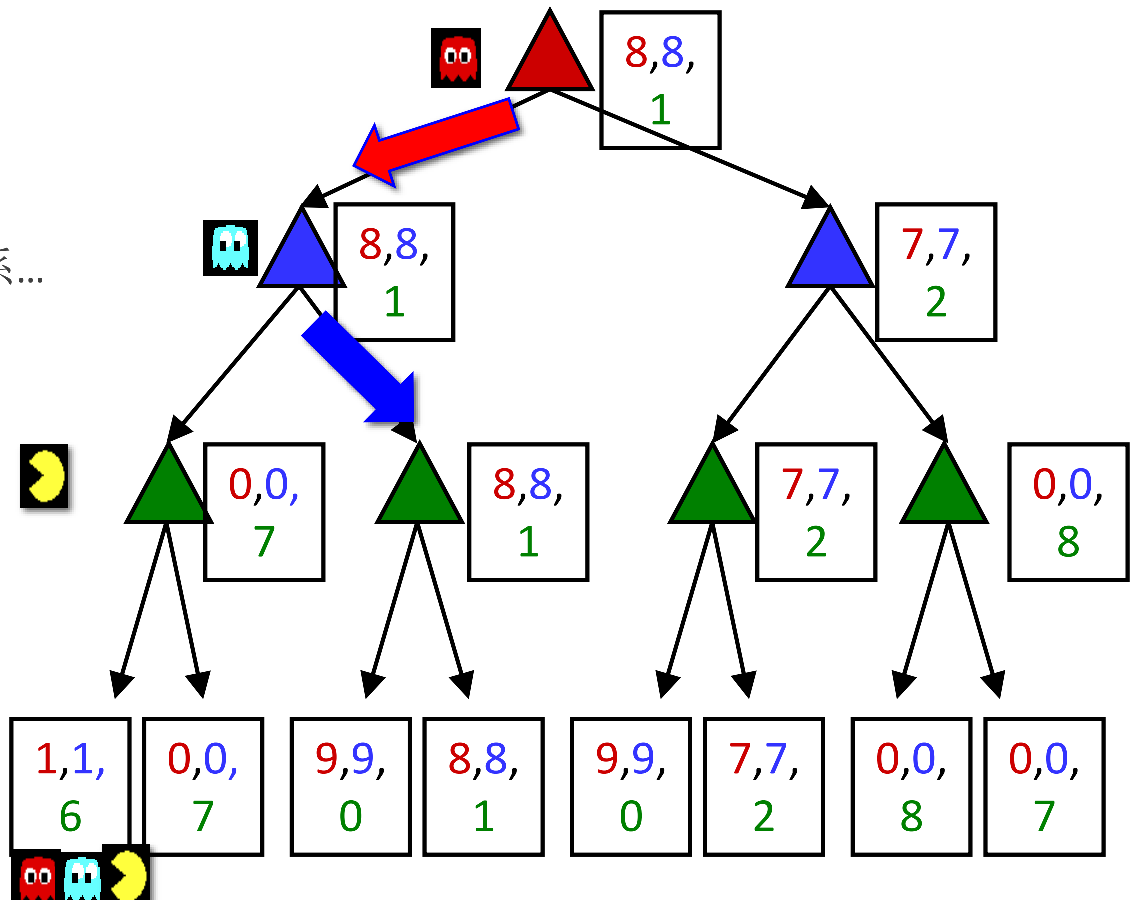


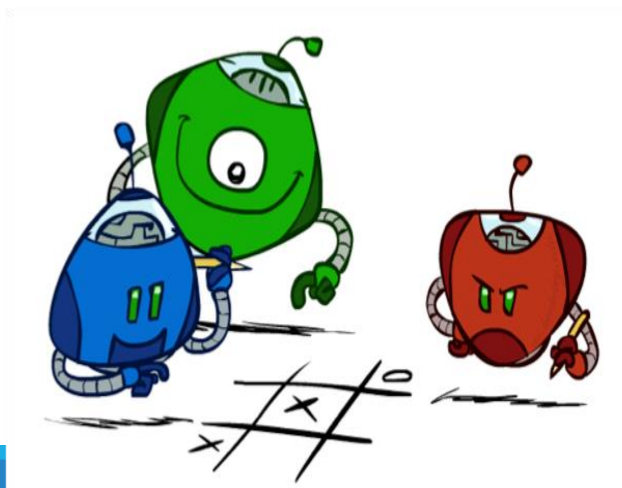
普遍化的最小最大值法（minimax）

如果不是零和游戏，或有多个玩家?

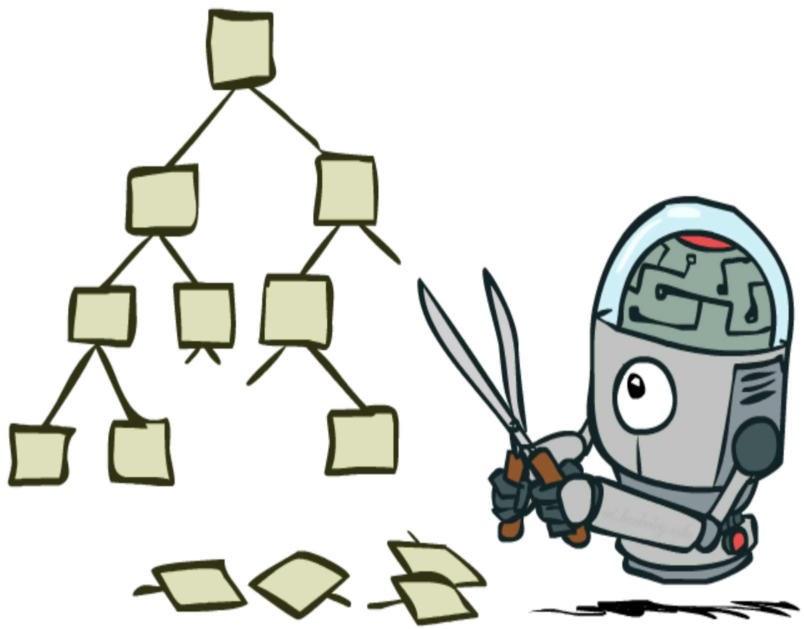
普遍情况:

- 终局（叶结点）的值是 一组值
- 中间状态节点的值也是 一组值
- ***每个玩家最大化自己的利益值***
- 能够动态产生协作和竞争等关系...

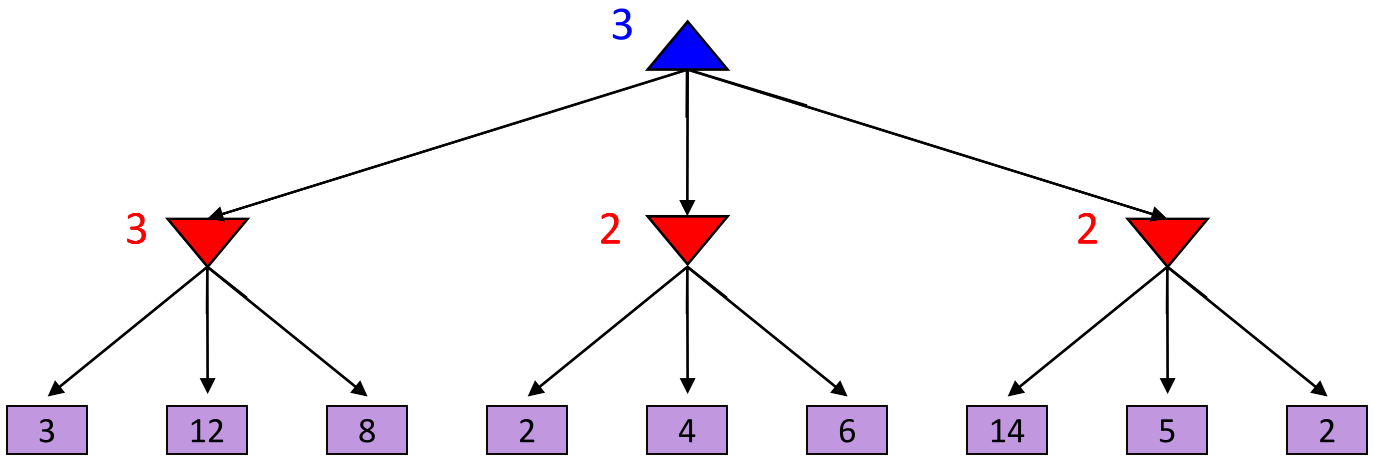




博弈（游戏）树的剪枝
(Game Tree Pruning)

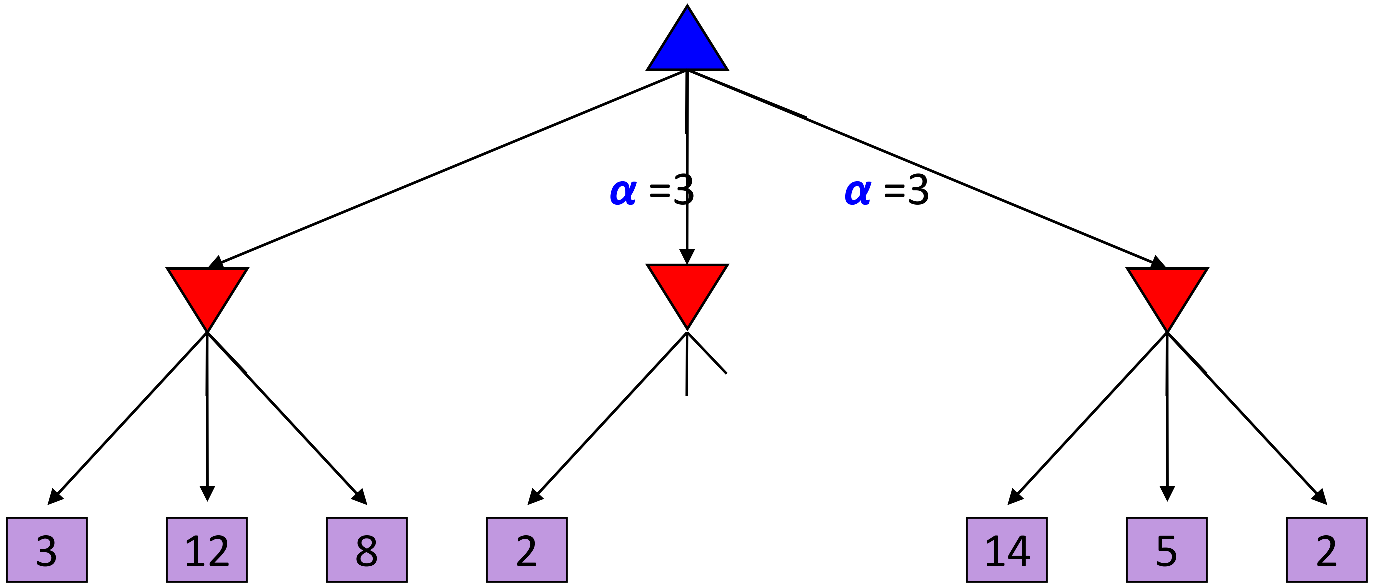


最小最大法举例



Alpha-Beta 剪枝例子

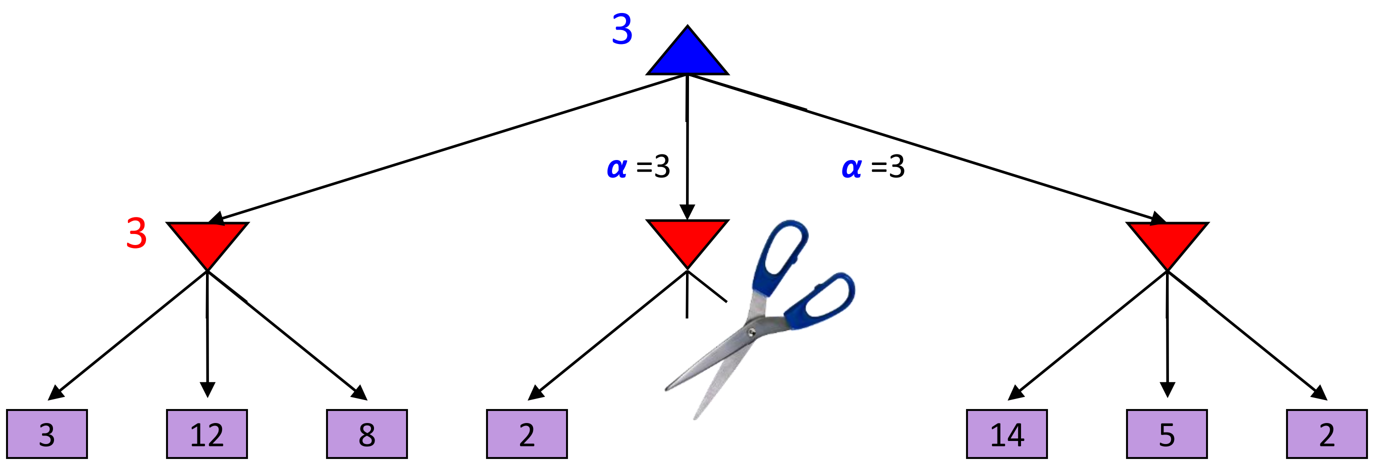
α = 在从当前MIN节点到根节点路径上所有MAX节点中最大的值



节点产生的顺序与结果有关: 可能导致不同的可被剪掉的节点数量

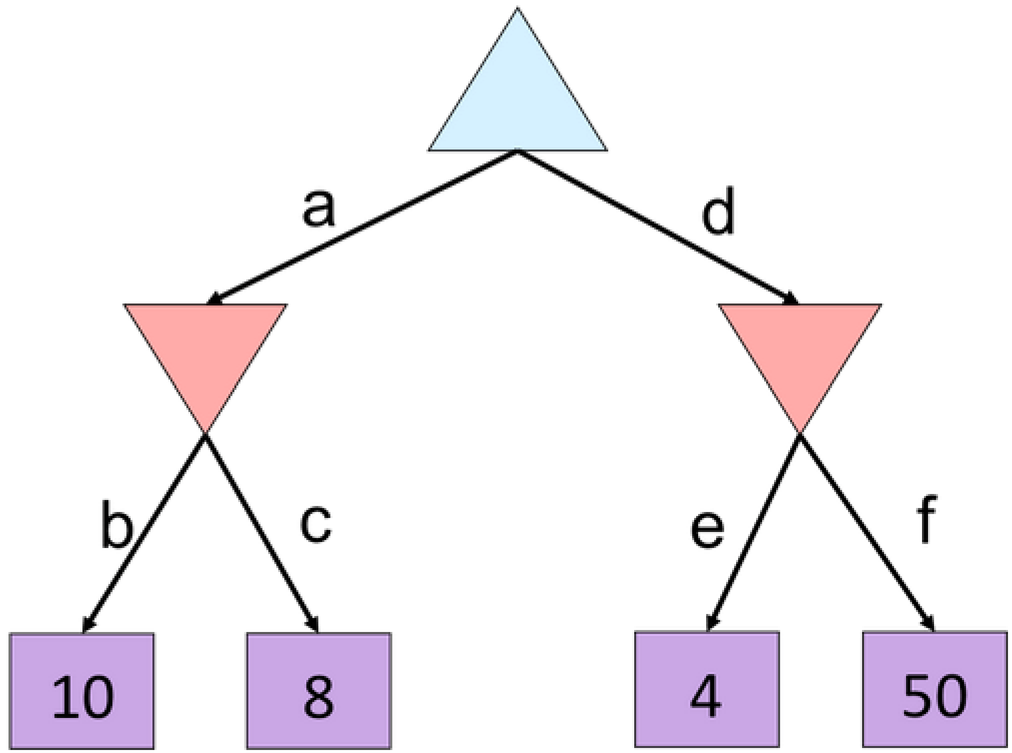
Alpha-Beta 剪枝例子

α = 在当前路径上所有MAX节点中最大的值

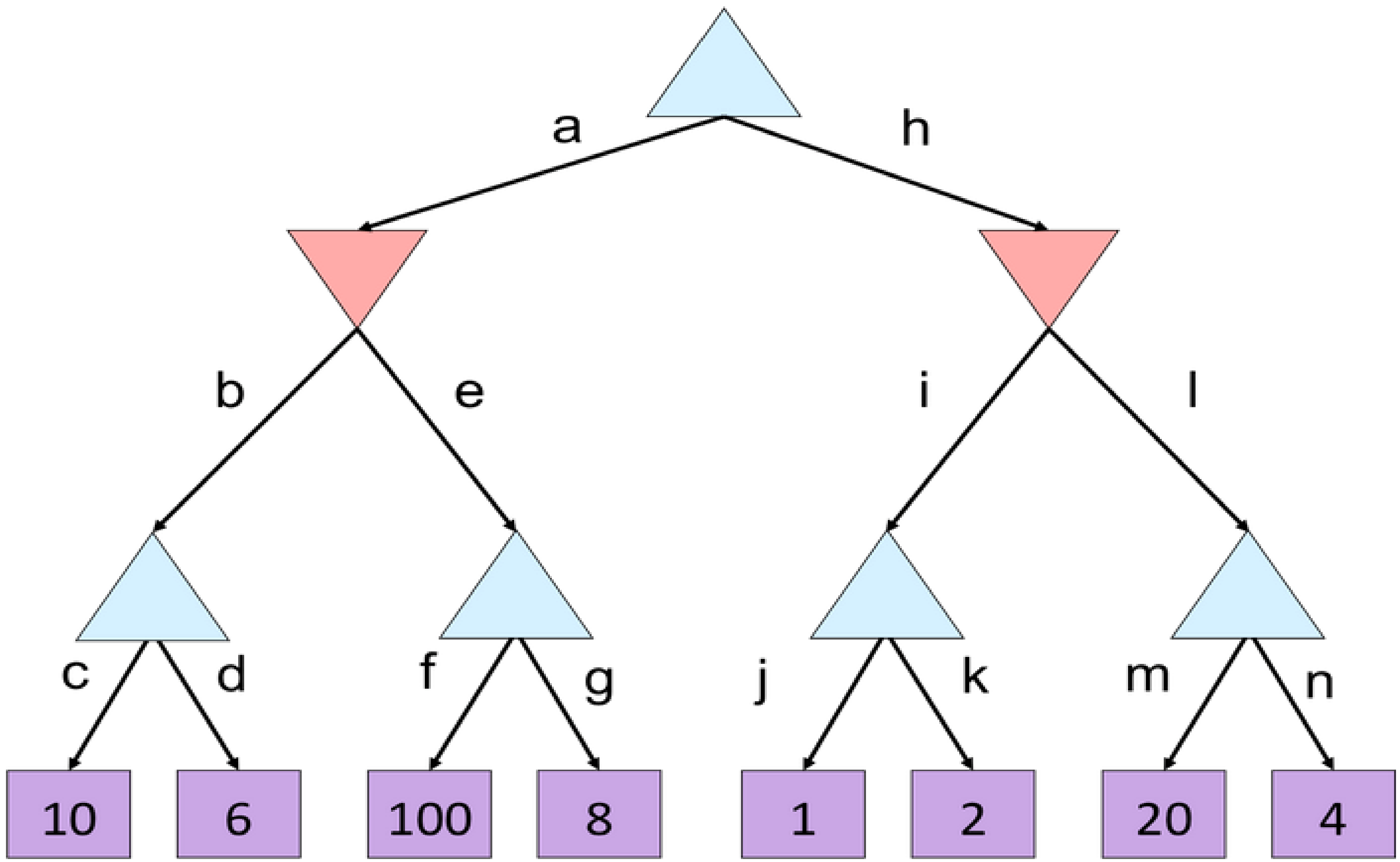


节点产生的顺序与结果有关: 可能导致不同的可被剪掉的节点数量

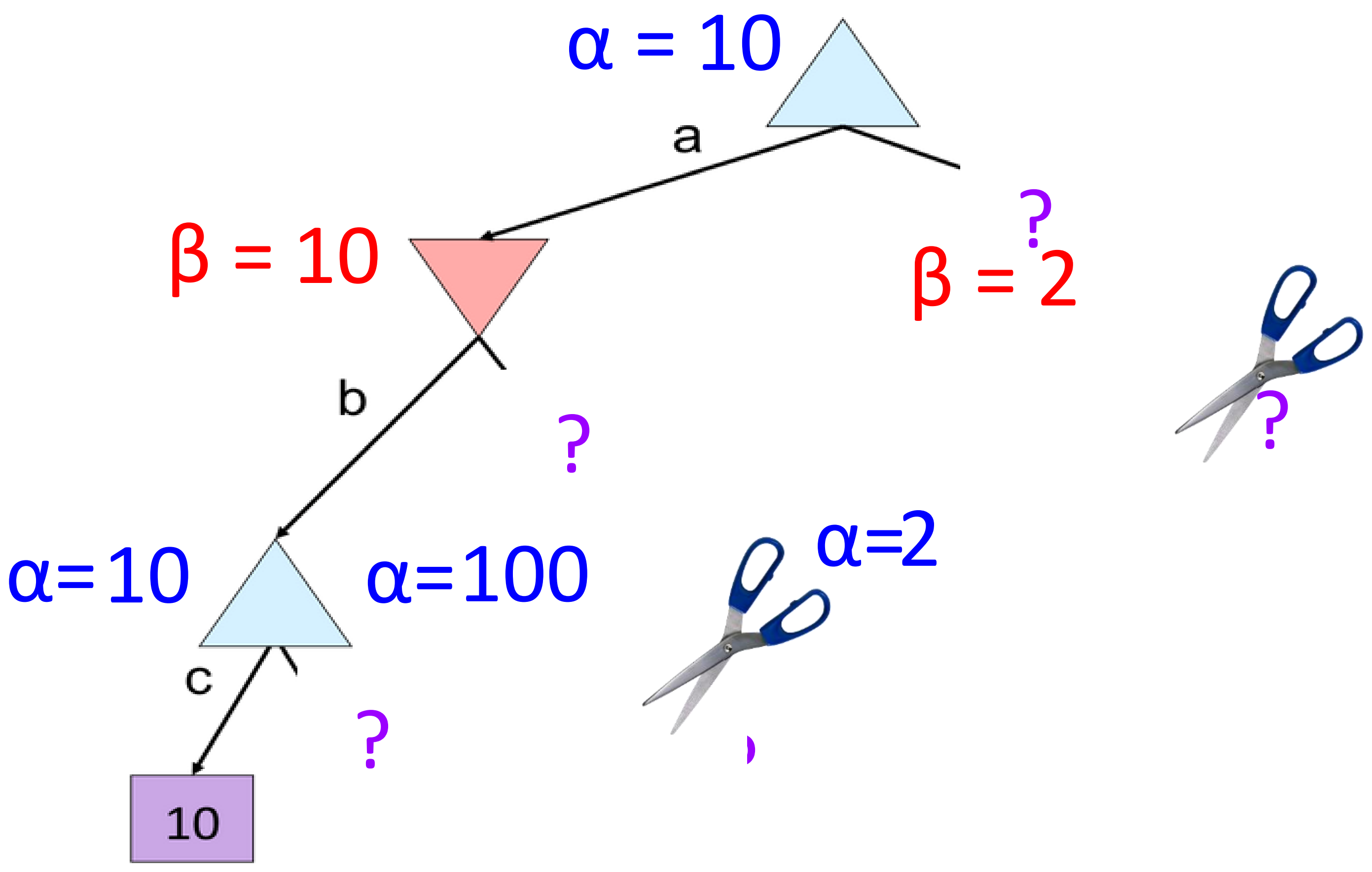
Alpha-Beta 举例



Alpha-Beta 举例 2



Alpha-Beta 举例 2



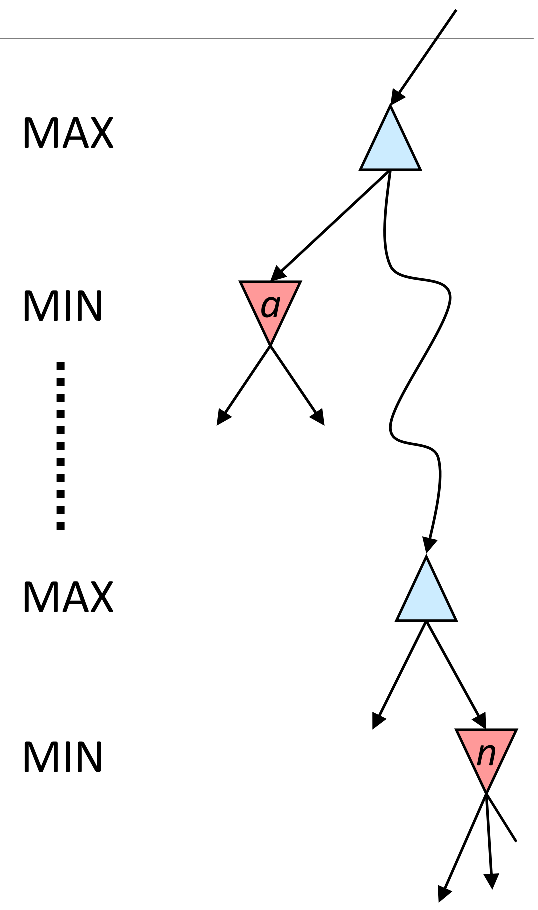
Alpha-Beta 剪枝

假定修剪MIN节点的子节点

- 假设正在计算节点 *n* 的 最小-值
- *n 节点*的值在检查其子节点的过程中逐渐减小
- 令 ***α*** 是从根节点到当前MIN节点路径上的(任何一个)MAX节点所能取到的最大值
- 如果 *n* 的当前值比 ***α*** 的小，那么路径上的MAX 分支节点将会避开这条路径，所以我们可以剪掉(不去检查) *n* 的其他子节点

对MAX 节点的子节点剪枝操作是对称的

- 令 **β** 是从根到当前MAX节点路径中的 MIN 节点所能达到的最小值



Alpha-Beta 剪枝算法实现

```
def alpha-beta-search(state) return 一个行动
  v <- max-value(state, -∞, +∞)
  return 导致值为 v 的行动
```

α: MAX节点的最大值，当前路径上
β: MIN节点的最小值，当前路径上

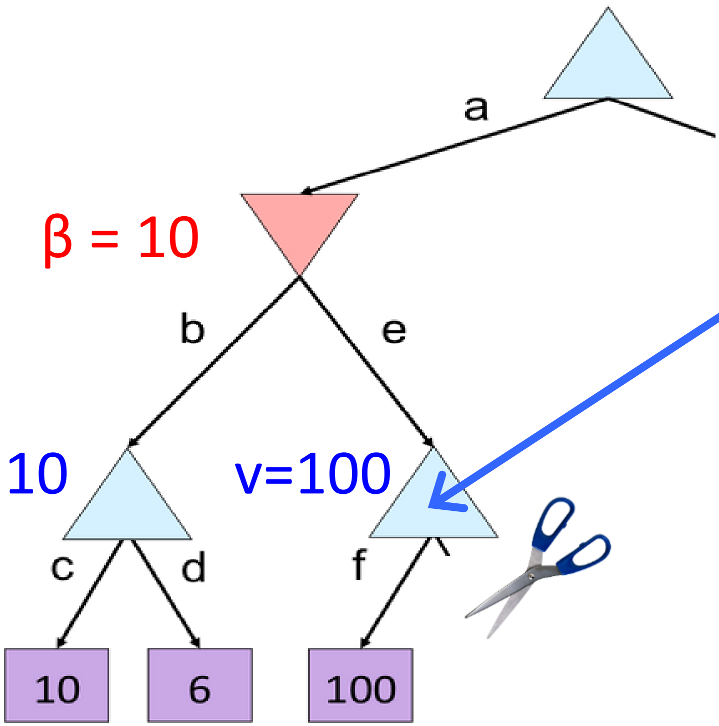
```
def max-value(state, α, β):
  if 终局-检测(state) return Utility(state)
    初始化 v = -∞
    for each a in ACTIONS(state) do
        v = max(v, min-value(Result(s,a),
            α, β))
        if v ≥ β return v //剪枝
        α = max(α, v)
    return v
```

```
def min-value(state , α, β):
  if 终局-检测(state) return Utility(state)
    初始化 v = +∞
    for each a in ACTIONS(state) do
        v = min(v, max-value(Result(s,a),
            α, β))
        if v ≤ α return v //剪枝
        β = min(β, v)
    return v
```

Alpha-Beta 举例 2

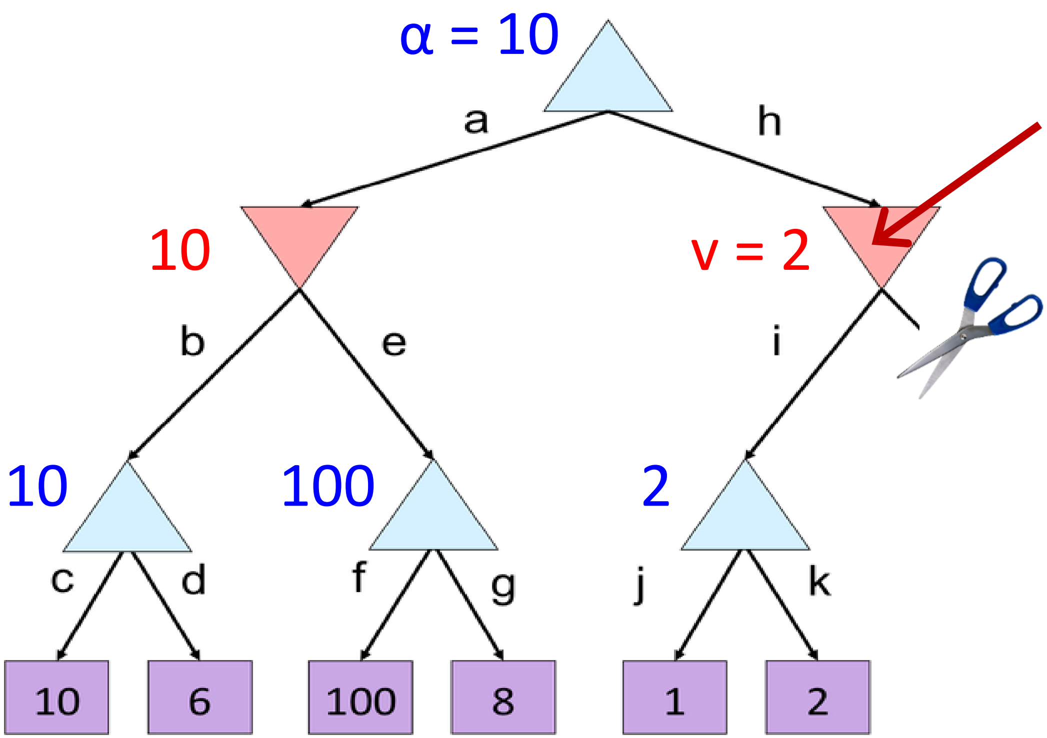
α: MAX's best option on path to root

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
```

Alpha-Beta 举例 2



α: MAX's best option on path to root

β: MIN's best option on path to root

```
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
```

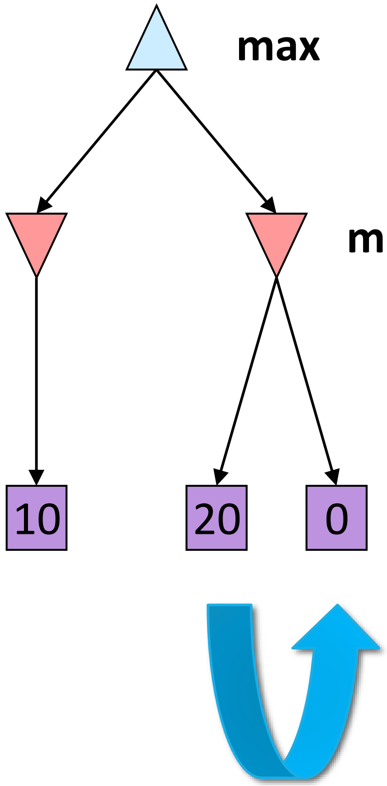
Alpha-Beta 剪枝属性

剪枝 ***不影响*** 根节点的最小最大值的计算！

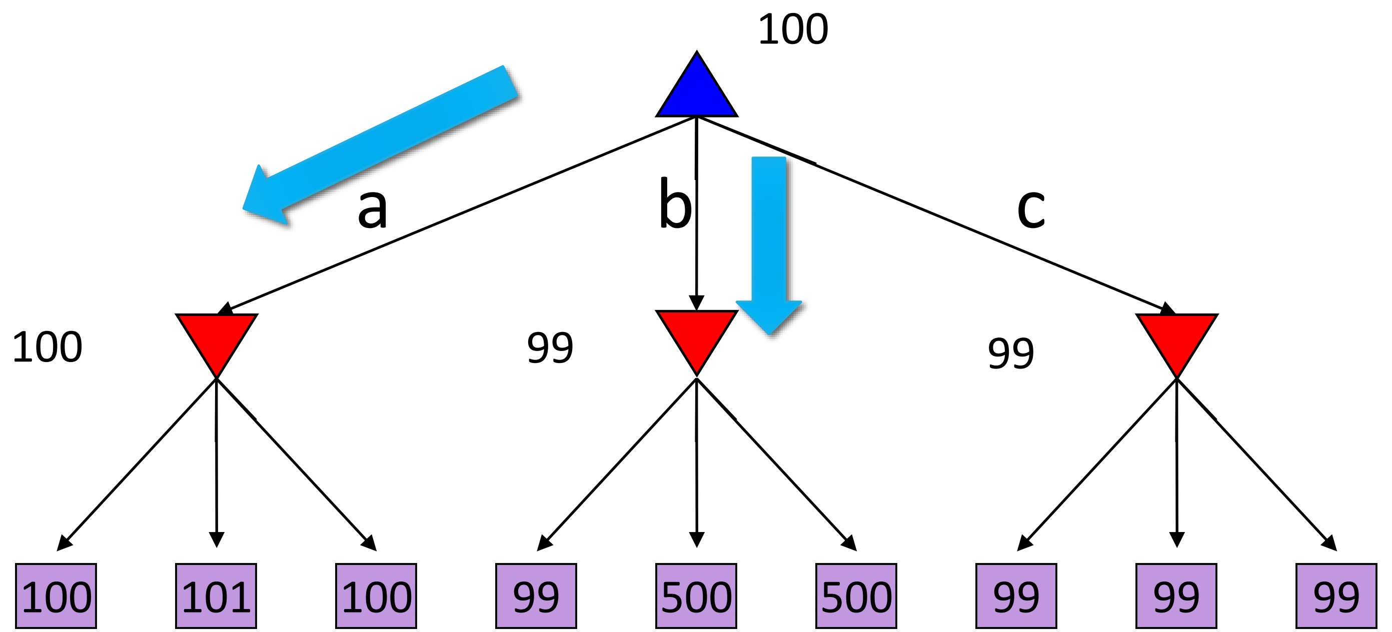
好的子节点的排序能提高剪枝的有效性

假定是 “完美的排序”:

- 时间复杂度能降到 $O(b^{m/2})$
- 搜索深度能提高一倍!



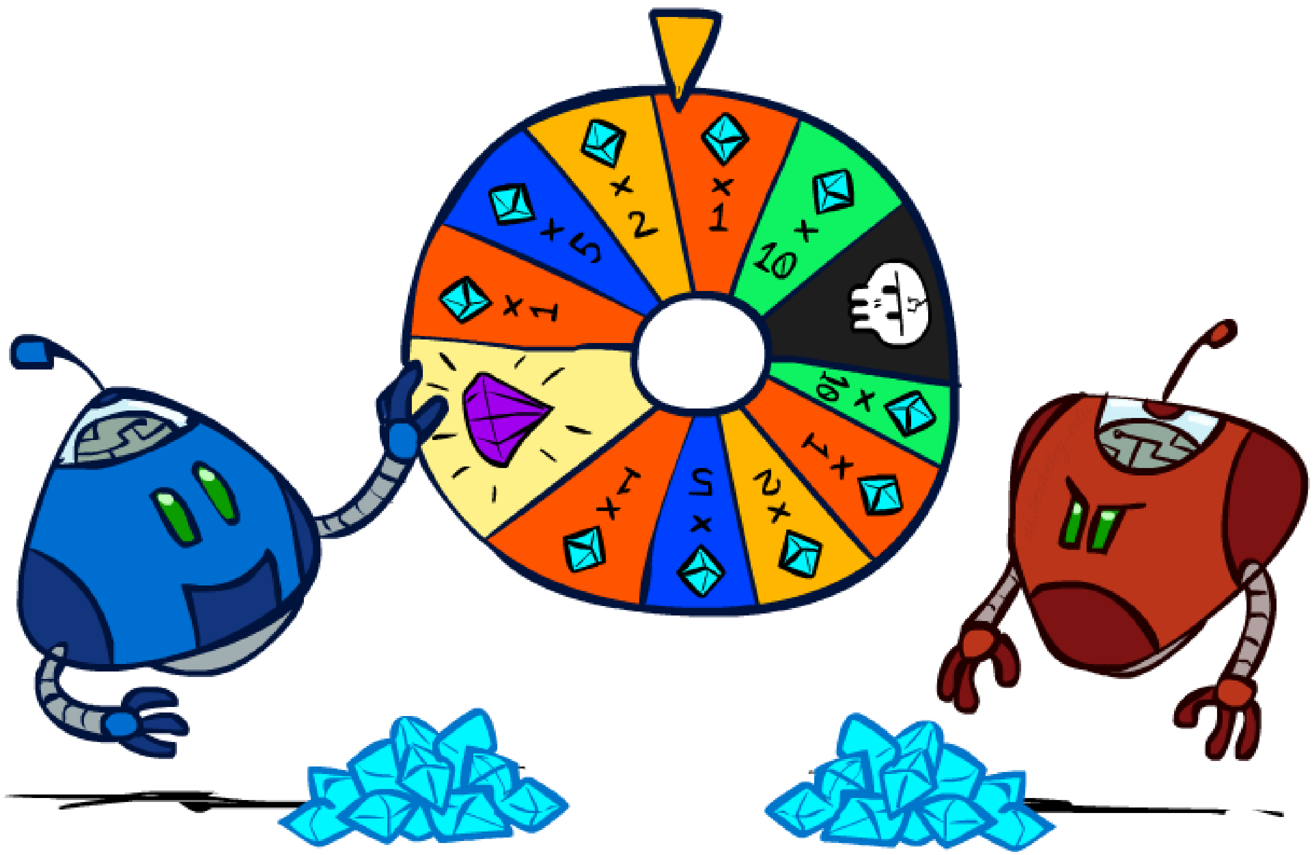
最小最大值 重新思考



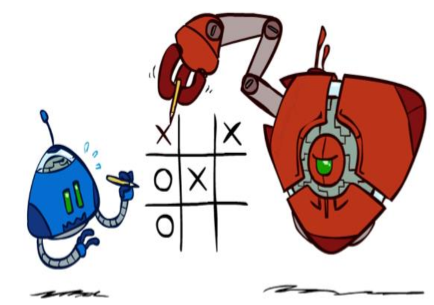
把叶结点值当成准确值。

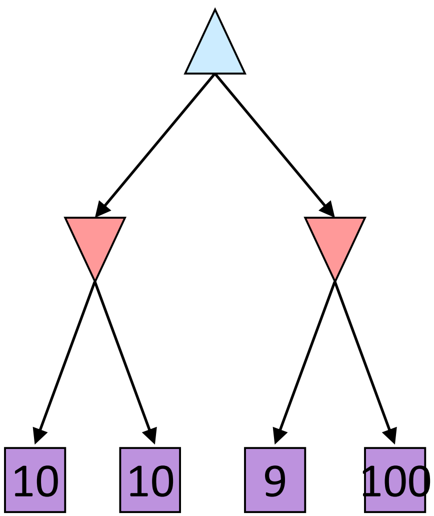
实际它们是不确定情况下的估计值，也许实际情况中，走b路径比走a更好。

不确定结果的游戏



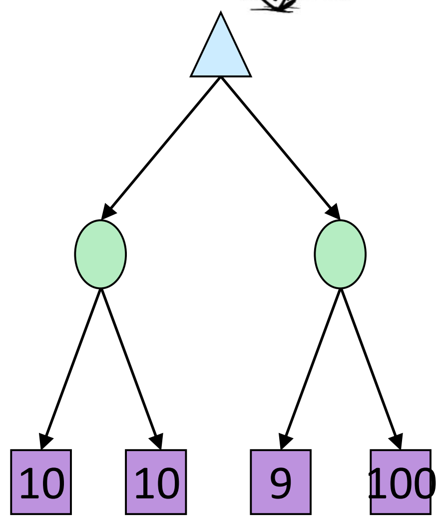
搜索树中可能有机遇节点





井字游戏

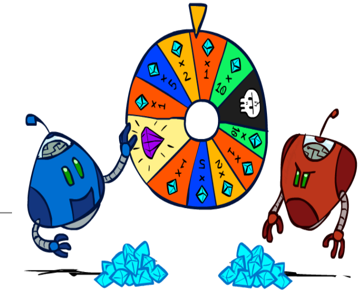
最小最大值(Minimax)

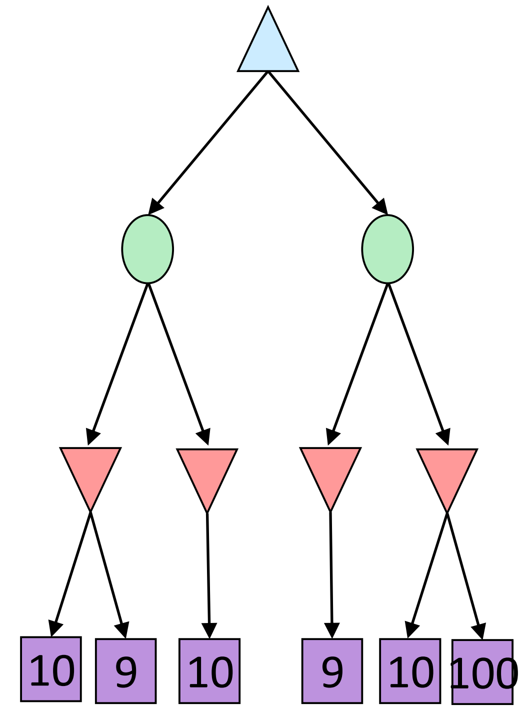


投资游戏

期望最大值

(Expectimax)





大富翁游戏

期望最小最大值(Expectiminimax)

最小最大值

Function 决策(s) returns 一个行动

return Actions(s) 中的行动 a ,它的 value(Result(s,a)) 最大

⇕

function value(s) returns a value

If 终局-检测(s) then return Utility(s)

if Player(s) = MAX then return $\max_{a \text{ in Actions}(s)}$ value(Result(s,a))

if Player(s) = MIN then return $\min_{a \text{ in Actions}(s)}$ value(Result(s,a))

期望最小最大值(Expectiminimax)

Function 决策(s) returns 一个行动

return Actions(s) 里的一个行动 a，
它的 value(Result(s,a)) 是最大的

function value(s) returns a value

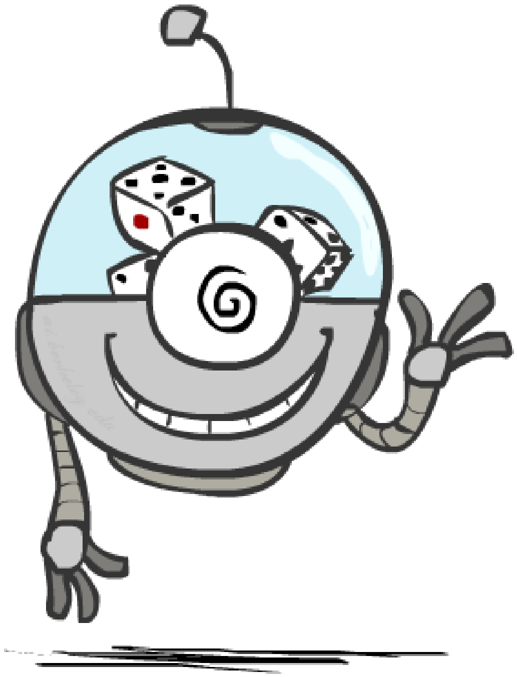
if 终局-检测(s) then return Utility(s)

if Player(s) = MAX then return $\max_{a \text{ in } Actions(s)}$ value(Result(s,a))

if Player(s) = MIN then return $\min_{a \text{ in } Actions(s)}$ value(Result(s,a))

if Player(s) = CHANCE then return $\text{sum}_{a \text{ in } Actions(s)}$ Pr(a) * value(Result(s,a))

概率 Probabilities

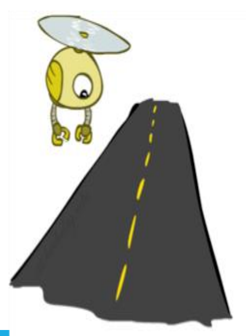
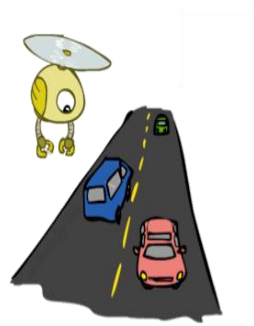


提示: 期望值

一个随机变量的期值是概率分布到对应输出结果的加权平均值

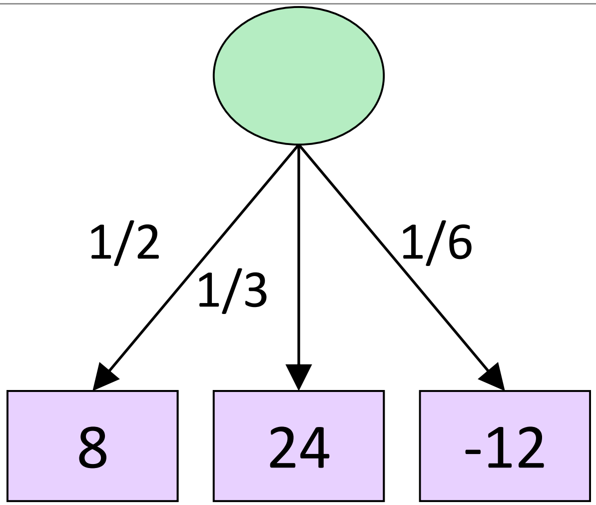
举例: 去机场路上花费的时间?

时间:	20 min		30 min		60 min	
	x	+	x	+	x	35 min
概率:	0.25		0.50		0.25	

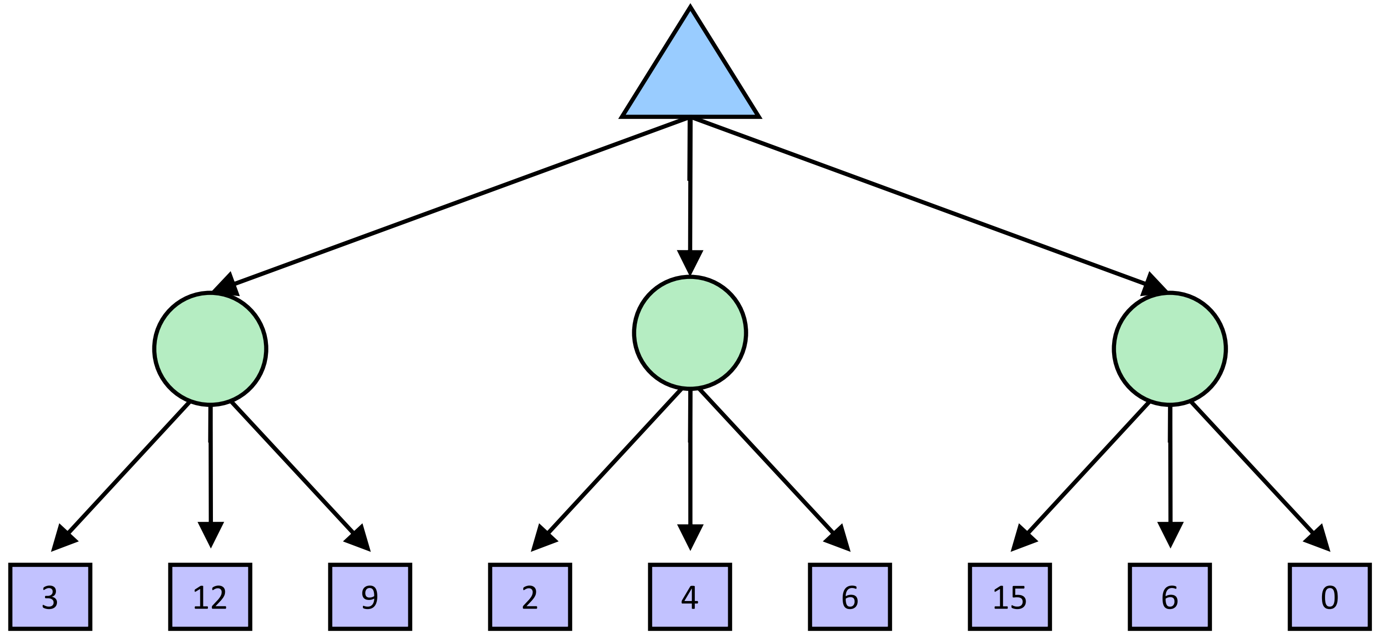

计算期望最小最大值的代码

sum$_{\text{a in Action(s)}}$
Pr(a) * value(Result(s,a))

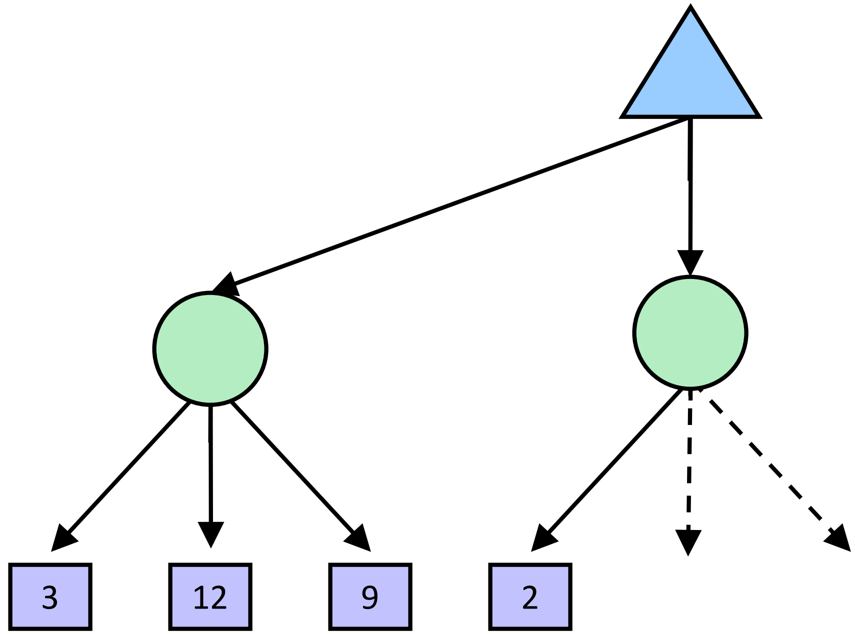


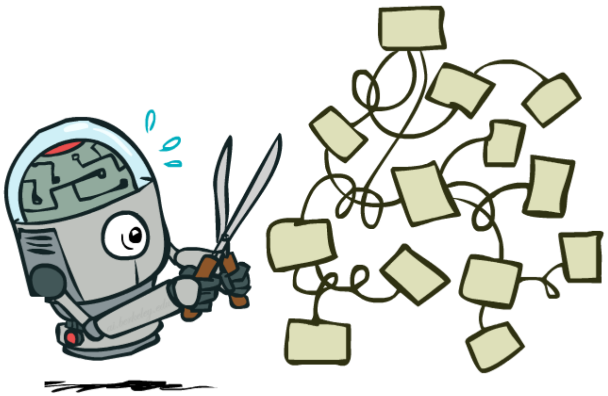
v = (1/2) (8) + (1/3) (24) + (1/6) (-12) = 10

Expectimax Example



Expectimax Pruning?





举例: 十五子跳棋Backgammon

骰子增加了行动分支数 *b*: 两个骰子有21 种不同的结果组合

- Backgammon ≈ 20 合法行动 （b=20）
- b 有时可能达到4000，当两个骰子数相同时候（该游戏的一个规则）

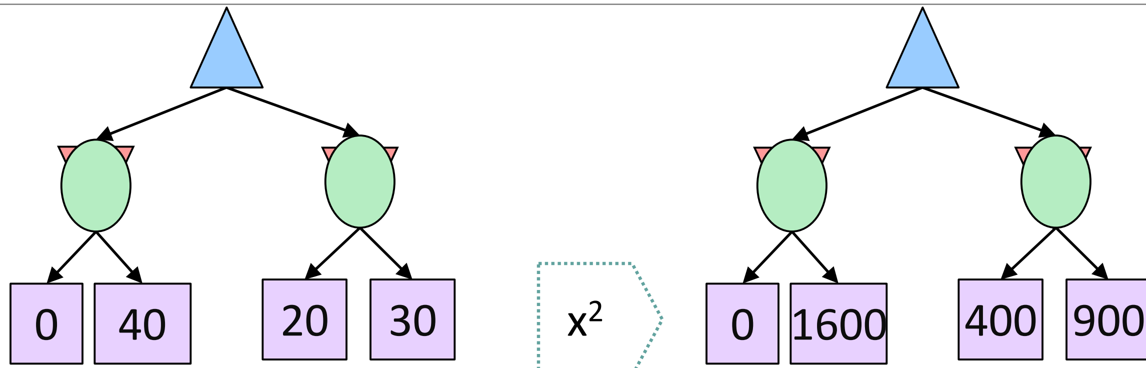
Image: Wikipedia

当搜索深度增加, 走到一个给定的搜索节点的概率在变小

- 所以此时搜索的意义变小
- 因此限制搜索深度变得可行
- 但是剪枝变得比以前有所复杂…

历史上的人工智能程序: TDGammon 仅用步长为2的搜索 + 非常好的评估函数 + 增强学习法，成就了世界冠军级的游戏水平。

Utility值与决策的关系



x>y => f(x)>f(y)

f(x) = Ax+B where A>0

- 在最小最大值决策中，评估函数单调递增变化不影响决策结果
  - 只要好的状态结果有高的评估值
- 期望最小最大值，决策结果不受评估值的正仿射变换影响
  - 获胜概率高的状态，其对应的评估值也应该大

总结

博弈（游戏） 当找到最优解是不可能的时候，需要行动上的决策
- 限制深度搜索，和近似评估函数

博弈（游戏） 决策存在有效率的搜索计算
- Alpha-beta 剪枝

在博弈游戏上的探索产生了重要的研究思想
- 增强式学习 Reinforcement learning (国际跳棋)
- 迭代加深 Iterative deepening (国际象棋)
- 合理的元推理 Rational metareasoning (奥赛罗棋)
- 蒙特卡罗树搜索 Monte Carlo tree search (围棋)
- 经济学中关于部分信息可知的博弈方法 (纸牌)

视屏游戏中的决策问题，仍是巨大的挑战！(搜索空间庞大)
- $b = 10^{500}$, $|S| = 10^{4000}$, $m = 10{,}000$

Definition 3.2.1 (Normal-form game) *A (finite, n-person)* normal-form game *is a tuple (N, A, u), where:*

- *N is a finite set of n players, indexed by i;*

action

- *$A = A_1 \times \cdots \times A_n$, where A_i is a finite set of* actions *available to player i. Each vector $a = (a_1, \ldots, a_n) \in A$ is called an* action profile*;*

action profile

utility function

- *$u = (u_1, \ldots, u_n)$ where $u_i : A \mapsto \mathbb{R}$ is a real-valued* utility *(or* payoff*)* function *for player i.*

	Heads	Tails
Heads	$1, -1$	$-1, 1$
Tails	$-1, 1$	$1, -1$

gure 3.6: Matching Pennies game.

Definition 3.2.4 (Mixed strategy) *Let (N, A, u) be a normal-form game, and for any set X let $\Pi(X)$ be the set of all probability distributions over X. Then the set of* mixed strategies *for player i is $S_i = \Pi(A_i)$.*

Definition 3.2.5 (Mixed-strategy profile) *The set of* mixed-strategy profiles *is simply the Cartesian product of the individual mixed-strategy sets, $S_1 \times \cdots \times S_n$.*

Definition 3.2.7 (Expected utility of a mixed strategy) *Given a normal-form game (N, A, u), the expected utility u_i for player i of the mixed-strategy profile $s = (s_1, \ldots, s_n)$ is defined as*

$$u_i(s) = \sum_{a \in A} u_i(a) \prod_{j=1}^{n} s_j(a_j).$$

Definition 3.3.3 (Best response) *Player i's* best response *to the strategy profile s_{-i} is a mixed strategy $s_i^* \in S_i$ such that $u_i(s_i^*, s_{-i}) \geq u_i(s_i, s_{-i})$ for all strategies $s_i \in S_i$.*

Definition 3.3.4 (Nash equilibrium) *A strategy profile $s = (s_1, \ldots, s_n)$ is a* Nash equilibrium *if, for all agents i, s_i is a best response to s_{-i}.*

Intuitively, a Nash equilibrium is a *stable* strategy profile: no agent would want to change his strategy if he knew what strategies the other agents were following.

	Heads	Tails
Heads	$1, -1$	$-1, 1$
Tails	$-1, 1$	$1, -1$

gure 3.6: Matching Pennies game.

		Bob	
		One	**Two**
Alice	**One**	-1, 1	1, -1
	Two	1, -1	0, 0