南京大学
NANJING UNIVERSITY

人工智能导论

对抗搜索

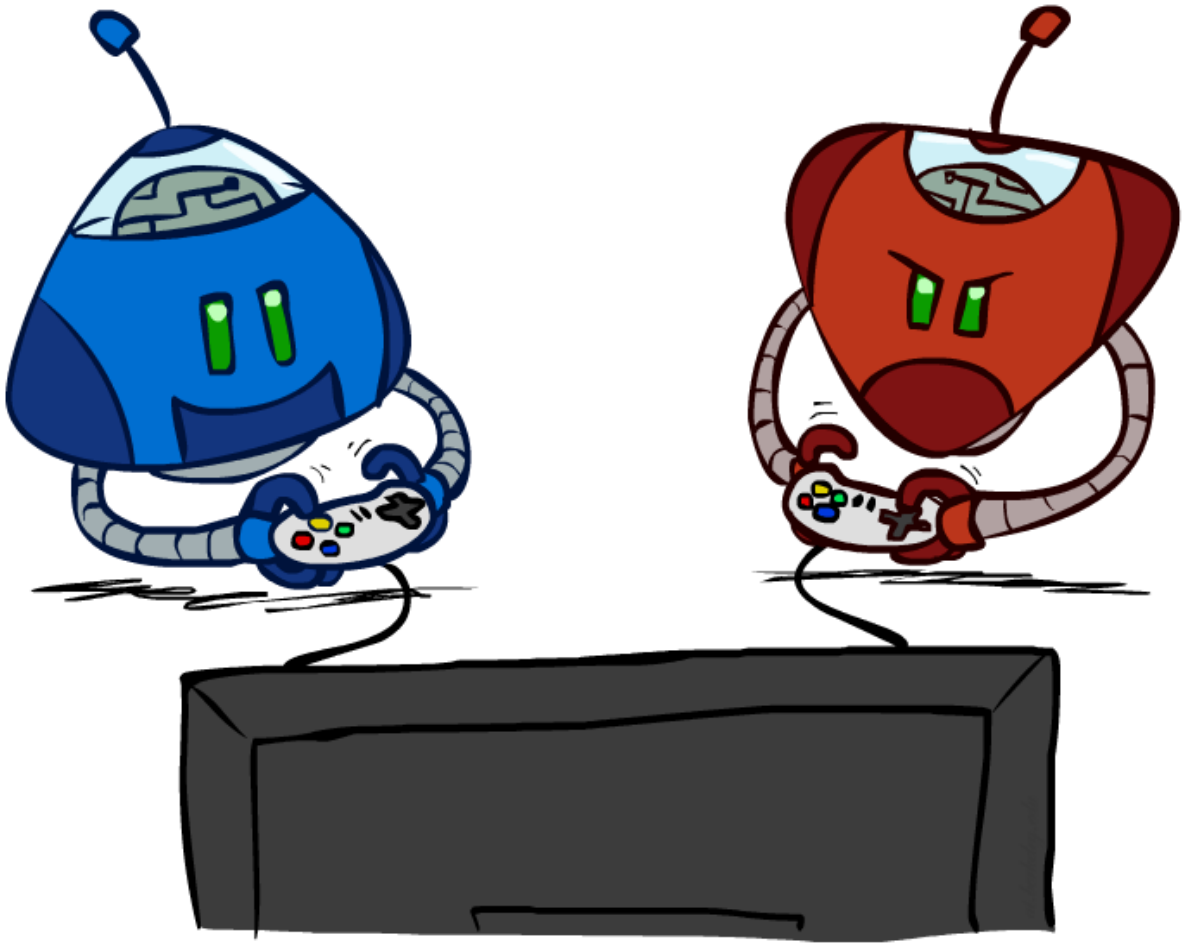
郭兰哲

南京大学 智能科学与技术学院

https://www.lamda.nju.edu.cn/guolz/IntroAI/fall2025/index.html

Email: guolz@nju.edu.cn

对抗搜索



提纲

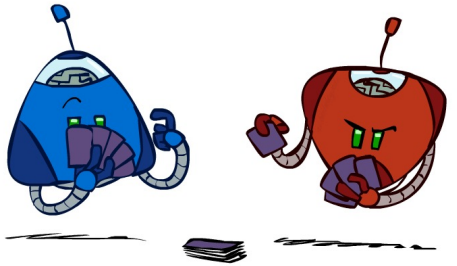
- 对抗博弈
  - 双人零和博弈
- 确定性搜索
  - 最大最小搜索
  - Alpha-beta 剪枝
- 基于模拟的搜索
  - 蒙特卡洛树搜索

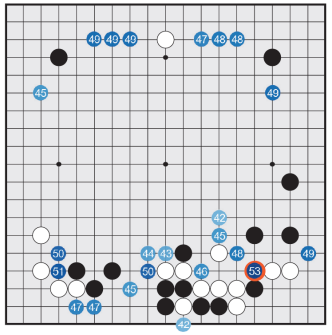
提纲

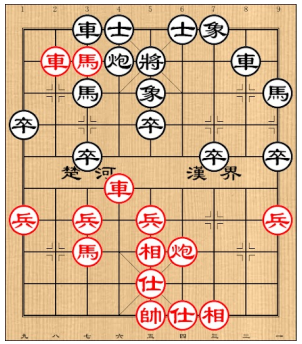
- 对抗博弈
  - 双人零和博弈
- 确定性搜索
  - 最大最小搜索
  - Alpha-beta 剪枝
- 基于模拟的搜索
  - 蒙特卡洛树搜索

博弈

对抗搜索（Adversarial Search）也称为博弈搜索（Game Search）



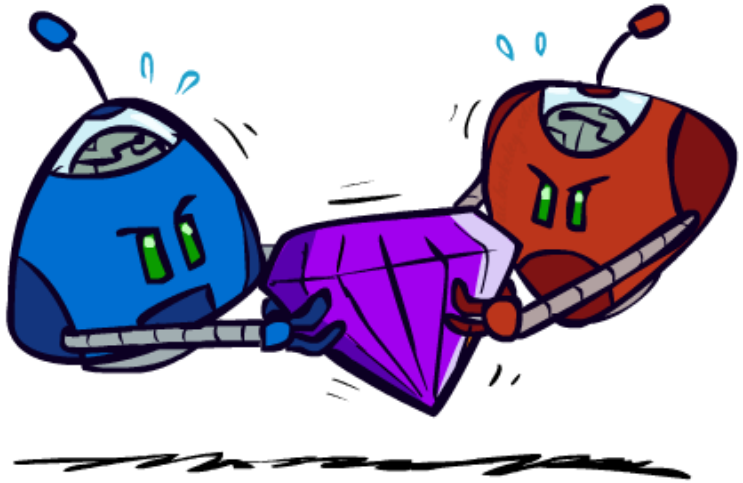




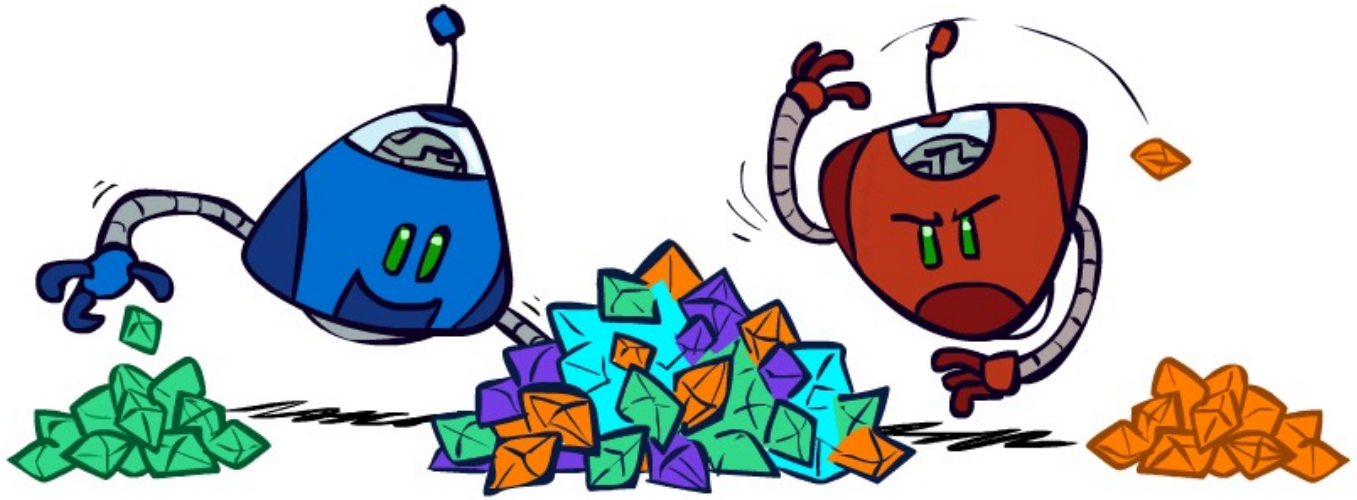
博弈的种类：

- ✓ 确定的、有随机性的
- ✓ 是否有完整信息？
- ✓ 几个玩家？
- ✓ 是不是零和博弈？

零和博弈



一个玩家赢了，则对手一定输了



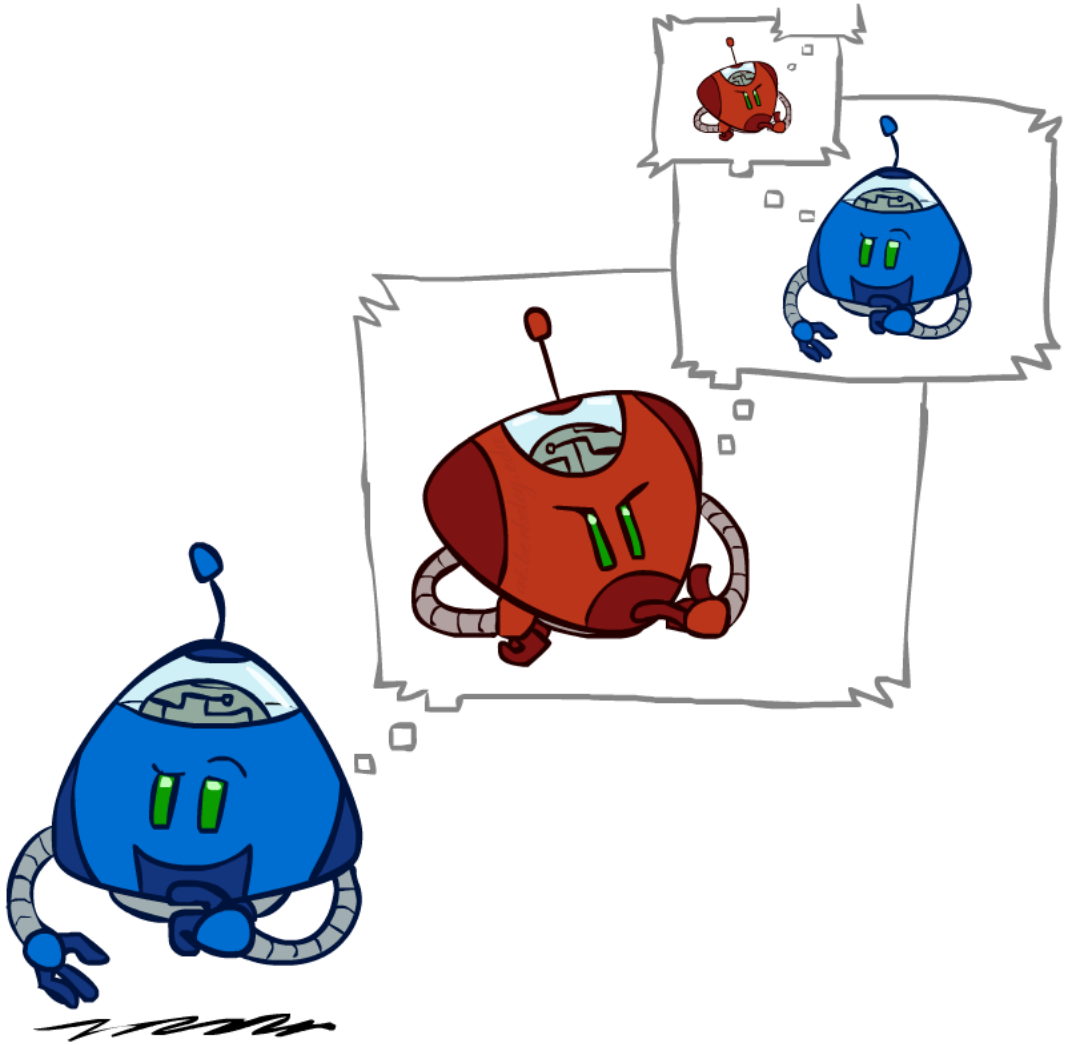
你可能赚了，但我也不亏

双人零和博弈

我们考虑**信息确定、全局可观察、竞争对手轮流行动、输赢收益零和**假设下的**双人博弈**问题

VS.

对抗搜索



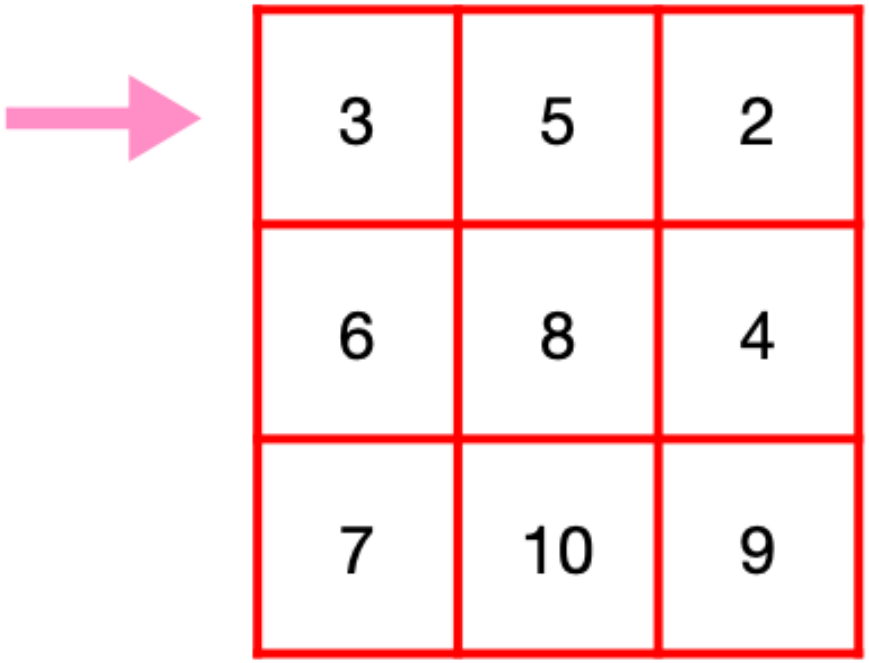
Two-Step Game

- Two step game: 首先，Alice选择第 i 行，然后，Bob选择第 j 列
- 结果：Alice输（Bob赢）第 i 行第 j 列的元素值

3	5	2
6	8	4
7	10	9

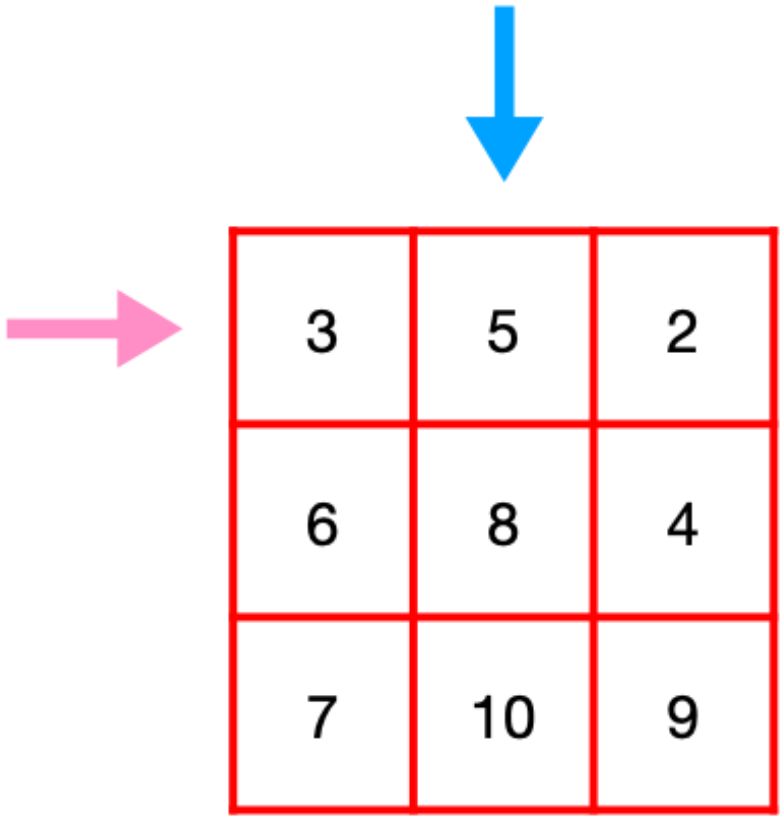
Two-Step Game

- Two step game: 首先，Alice选择第 i 行，然后，Bob选择第 j 列
- 结果：Alice输（Bob赢）第 i 行第 j 列的元素值



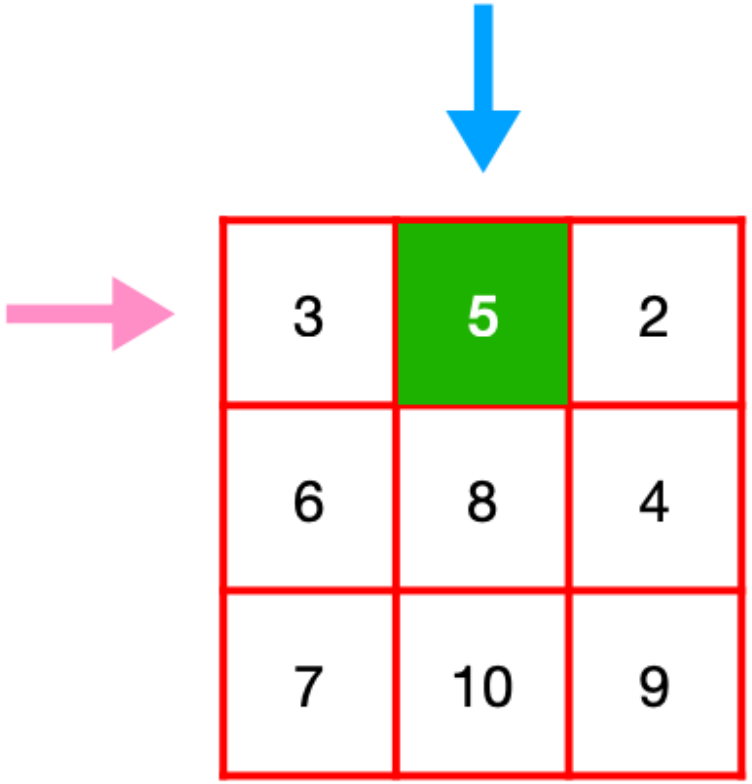
Two-Step Game

- Two step game: 首先，Alice选择第 i 行，然后，Bob选择第 j 列
- 结果：Alice输（Bob赢）第 i 行第 j 列的元素值



Two-Step Game

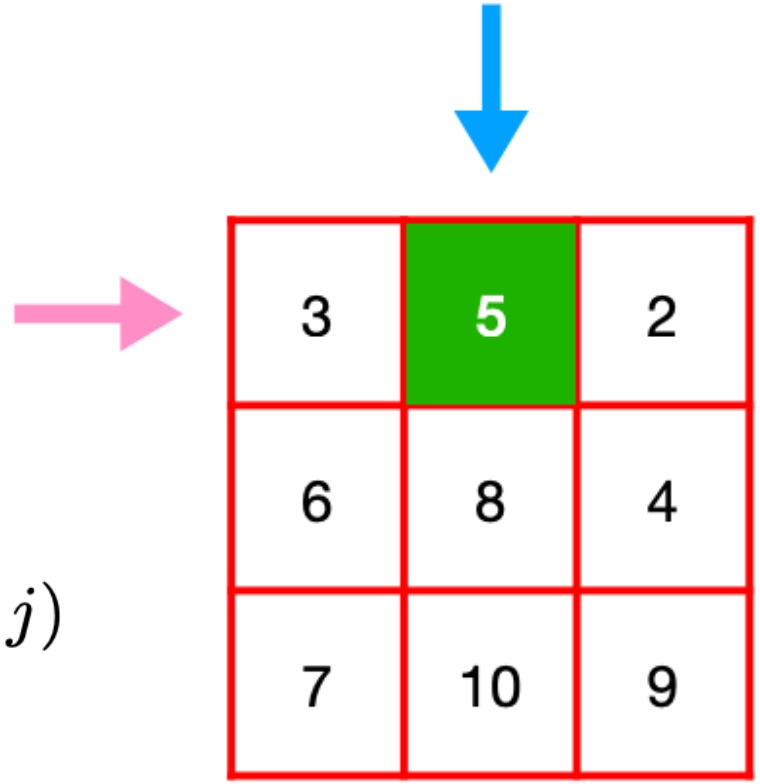
- Two step game: 首先，Alice选择第 i 行，然后，Bob选择第 j 列
- 结果：Alice输（Bob赢）第 i 行第 j 列的元素值



Two-Step Game

- Two step game: 首先，Alice选择第 i 行，然后，Bob选择第 j 列
- 结果：Alice输（Bob赢）第 i 行第 j 列的元素值
- A MinMax Game

$$U(i^*, j^*) = \min_i \max_j U(i, j)$$



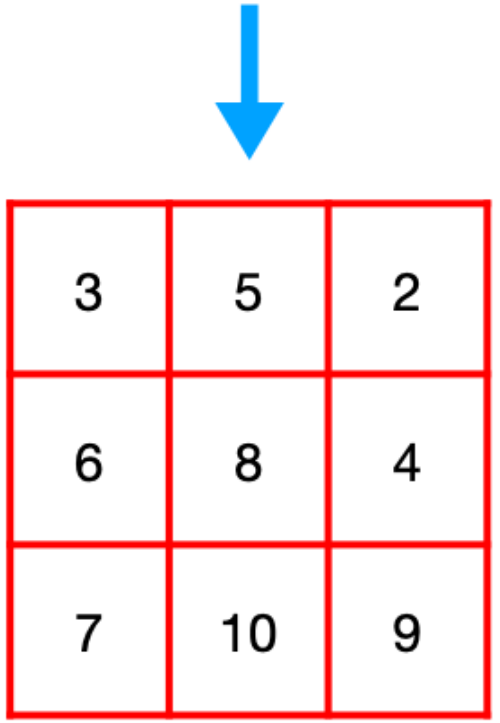
Two-Step Game

- Two step game: 首先，**Bob**选择第 j 列，然后， **Alice**选择第 i 行
- 结果：**Alice**输（**Bob**赢）第 i 行第 j 列的元素值

3	5	2
6	8	4
7	10	9

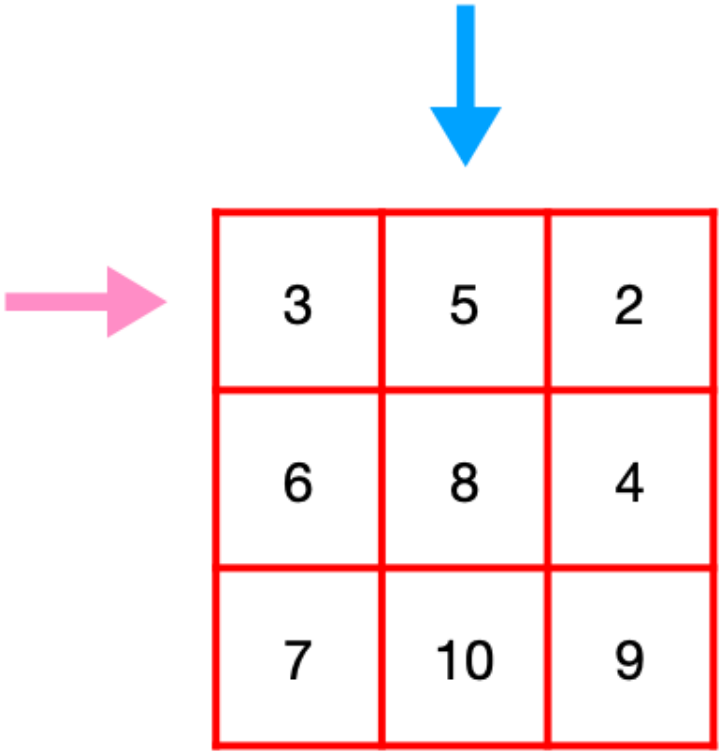
Two-Step Game

- Two step game: 首先，**Bob**选择第 j 列，然后， **Alice**选择第 i 行
- 结果：**Alice**输（**Bob**赢）第 i 行第 j 列的元素值



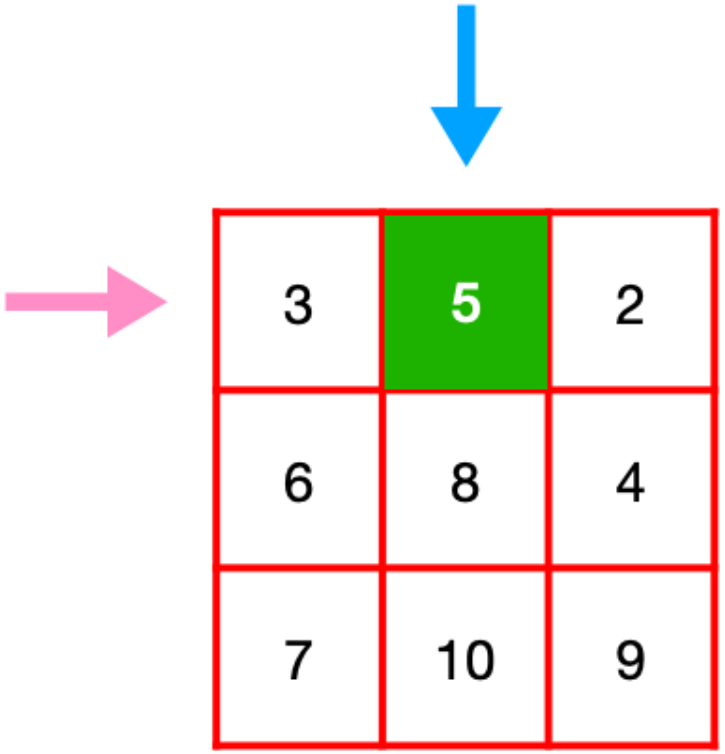
Two-Step Game

- Two step game: 首先，**Bob**选择第 j 列，然后， **Alice**选择第 i 行
- 结果：**Alice**输（**Bob**赢）第 i 行第 j 列的元素值



Two-Step Game

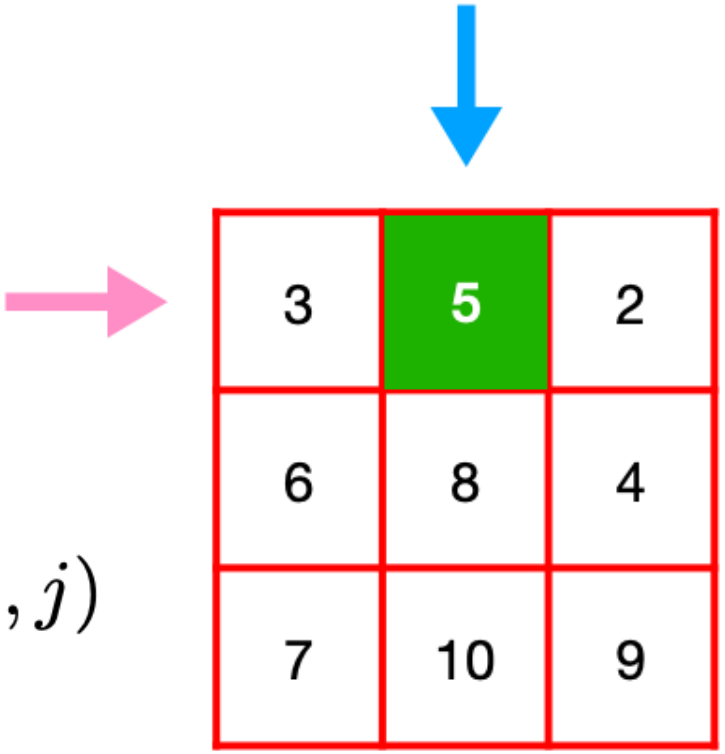
- Two step game: 首先，**Bob**选择第 j 列，然后， **Alice**选择第 i 行
- 结果：**Alice**输（**Bob**赢）第 i 行第 j 列的元素值



Two-Step Game

- Two step game: 首先，**Bob**选择第 j 列，然后， **Alice**选择第 i 行
- 结果：**Alice**输（**Bob**赢）第 i 行第 j 列的元素值
- A MaxMin Game

$$U(i^*, j^*) = \max_j \min_i U(i, j)$$



Quiz : MinMax = MaxMin?

3	5	2
6	8	4
7	10	9

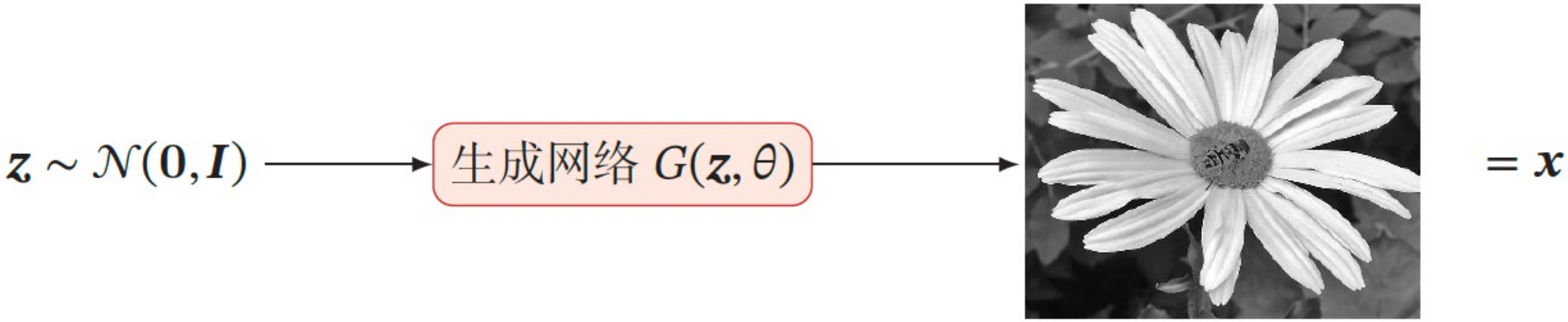
2	5	4
3	1	6
7	2	3

Theorem:

$$\max_j \min_i U(i,j) \leq \min_i \max_j U(i,j)$$

生成对抗网络(Generative Adversarial Networks, GAN)

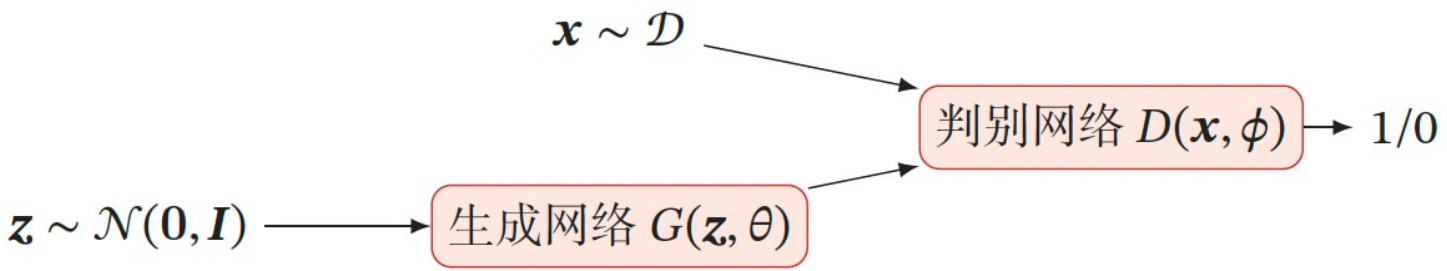
是否可以构建出一个模型$G: \mathcal{Z} \to \mathcal{X}$，使之可以生成符合数据分布$p(x)$的样本?



生成对抗网络(Generative Adversarial Networks, GAN)

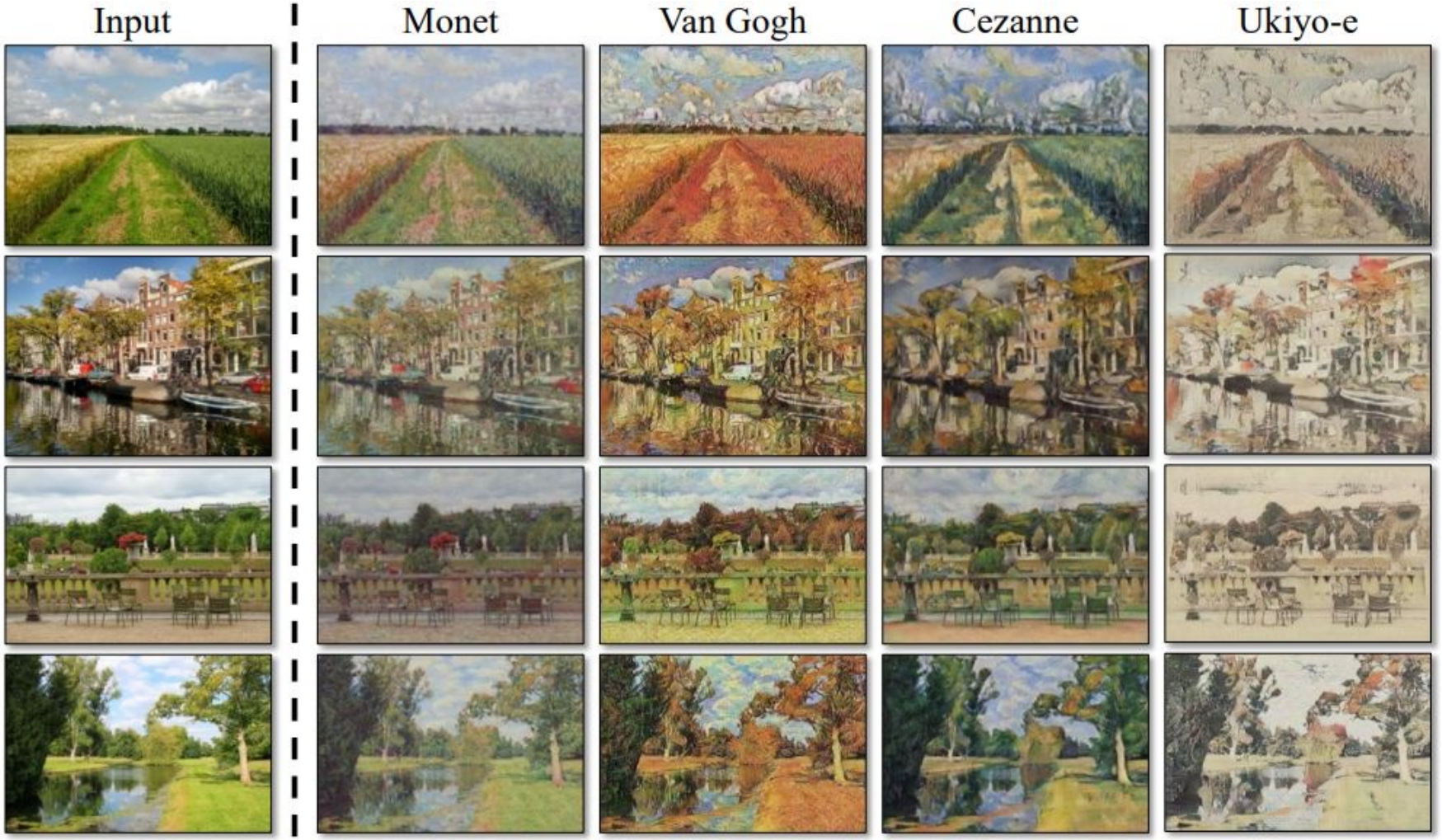
- 生成对抗网络[Goodfellow et al., 2014]通过**对抗训练和最大最小优化**的方式使得生成网络产生的样本服从真实数据分布

- 判别网络(Discriminator)：目标是尽量准确地判断一个样本是来自于真实数据还是由生成网络产生
- 生成网络(Geneartor)：目标是尽量生成判别网络无法区分来源的样本

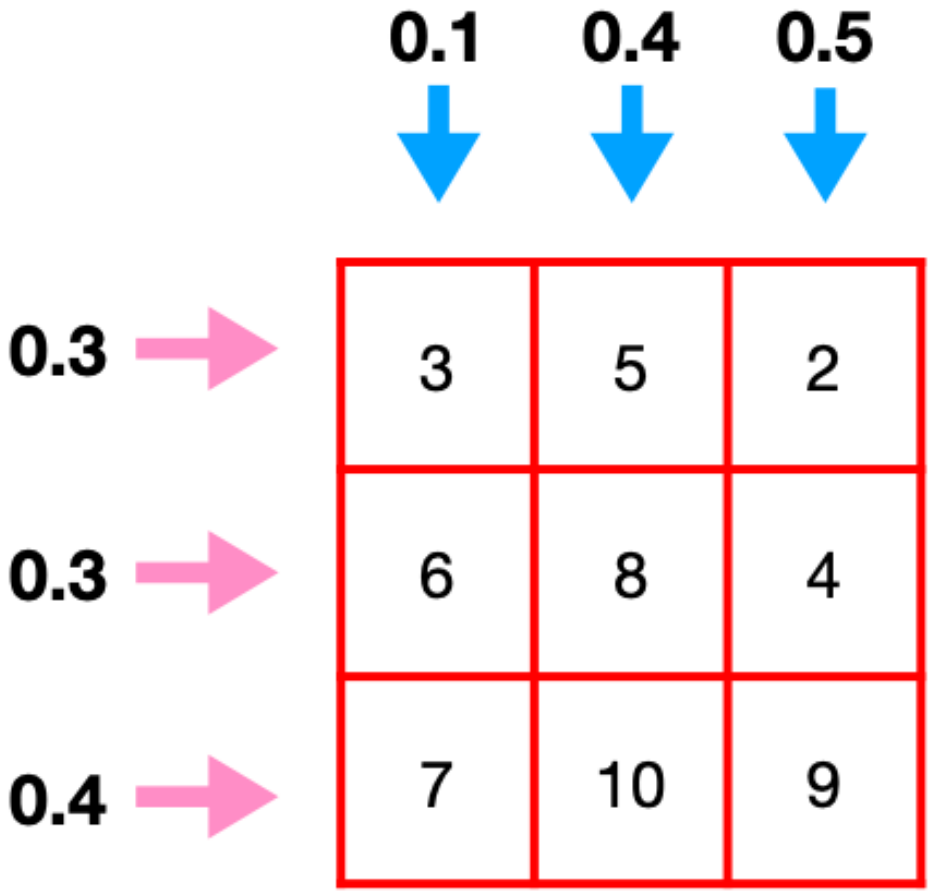


生成对抗网络的典型应用

- **图像到图像(Image-to-Image)**



确定性策略和非确定性策略



提纲

- 对抗博弈
  - 双人零和博弈
- 确定性搜索
  - 最大最小搜索
  - Alpha-beta 剪枝
- 基于模拟的搜索
  - 蒙特卡洛树搜索

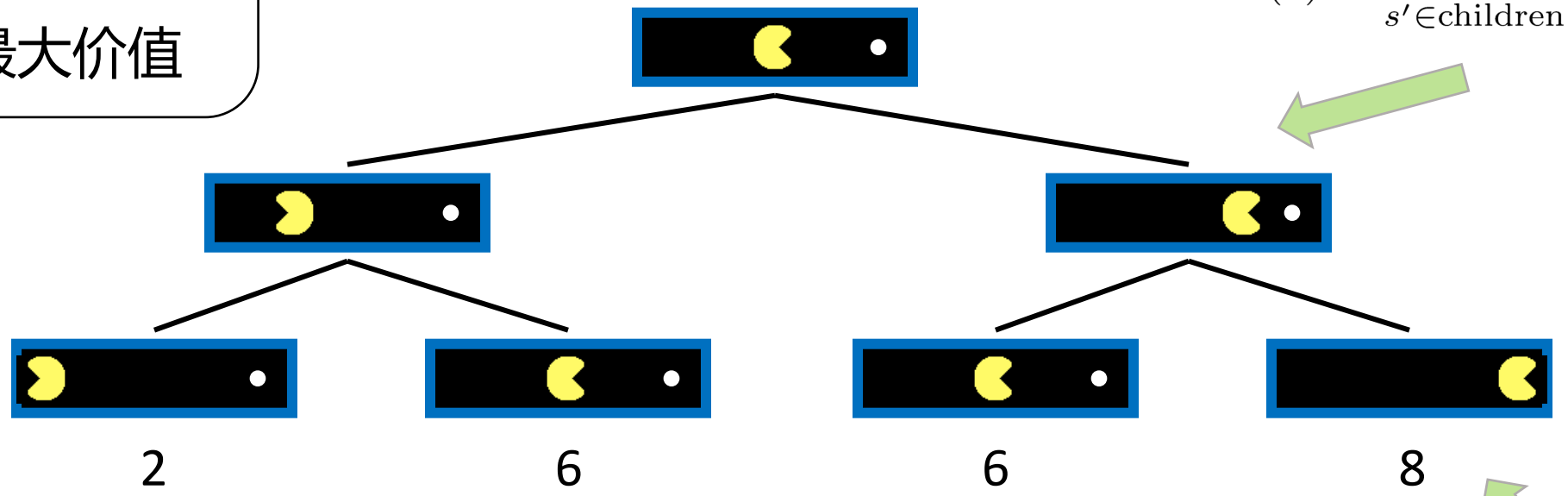
单一Agent搜索树

状态的价值 V

从当前状态出发能获得的最大价值

Non-Terminal States:

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



Terminal States:

$$V(s) = \text{known}$$

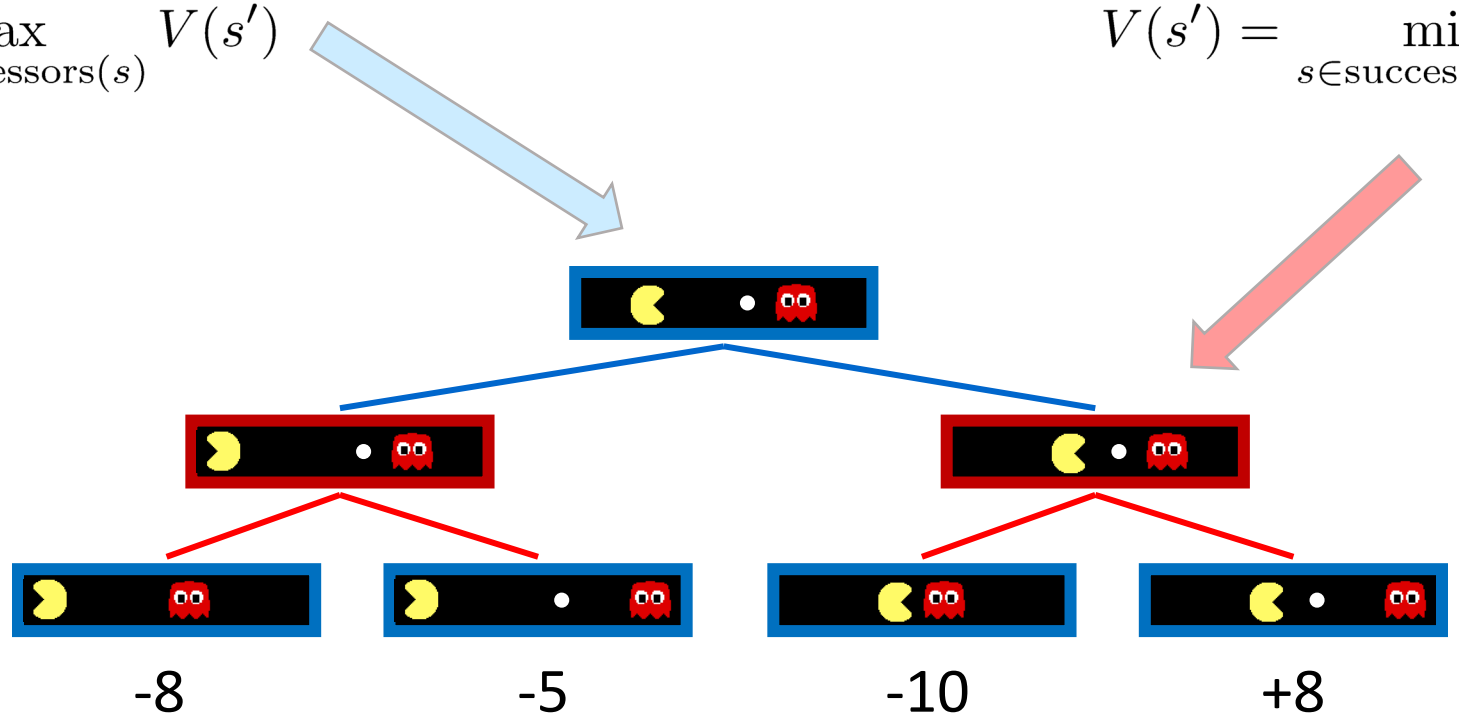
博弈搜索树

States Under Agent's Control:

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

States Under Opponent's Control:

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

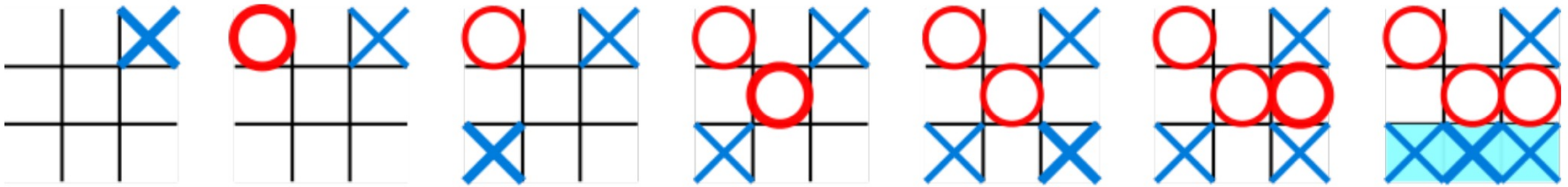


Terminal States:

$$V(s) = \text{known}$$

多步搜索

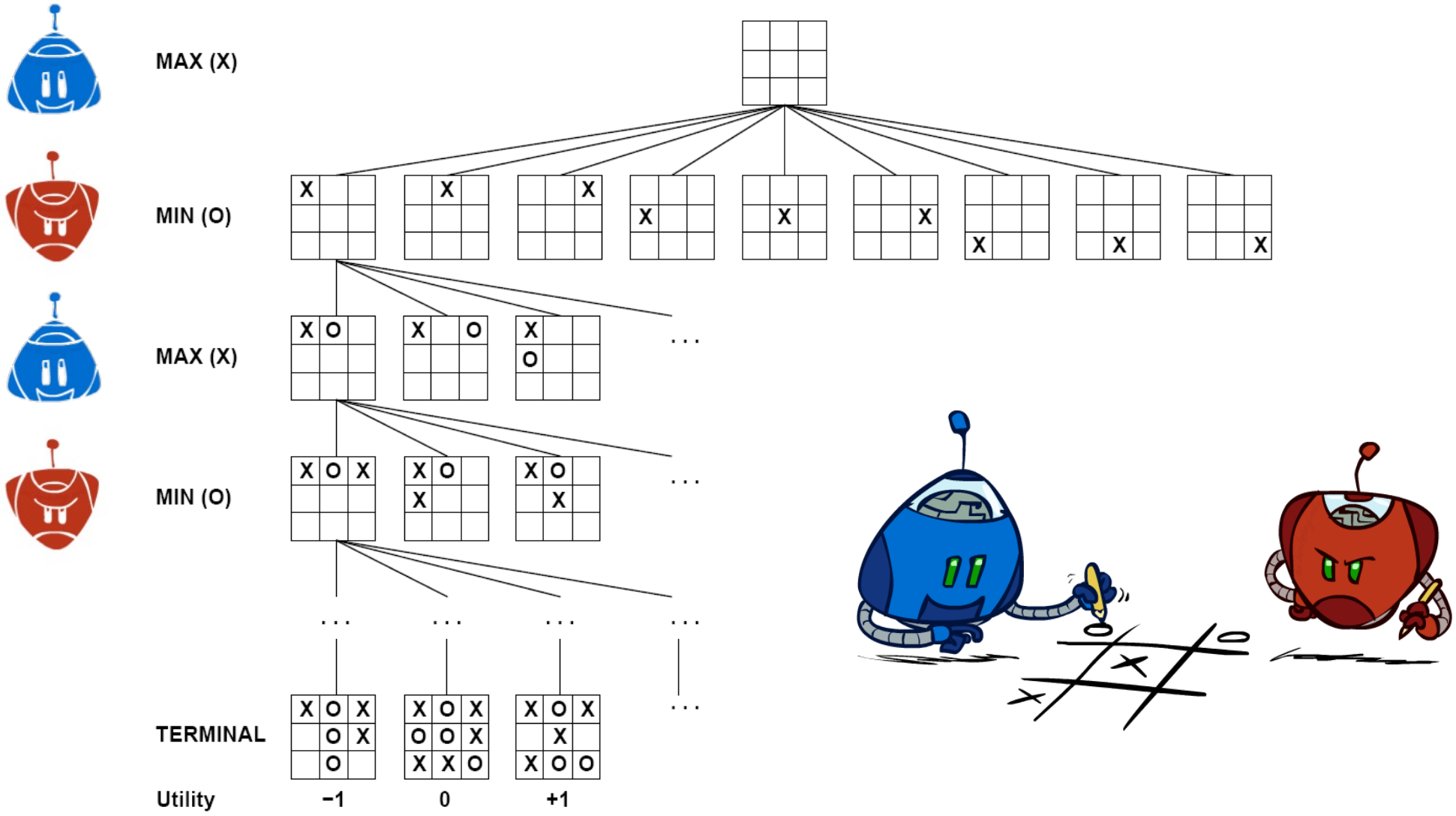
两人轮流在一有九格方盘上划加字或圆圈, 谁先把三个同一记号排成横线、直线、斜线, 即是胜者



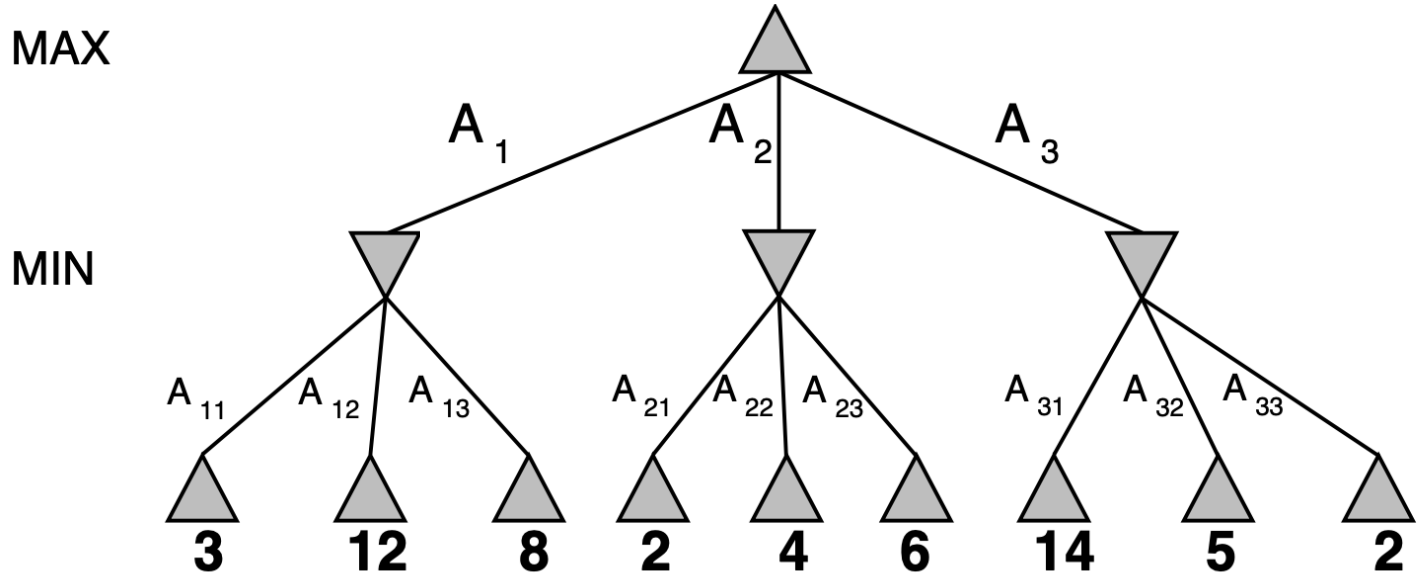
问题定义

- **状态**：状态s包括当前的游戏局面和当前行动的玩家
- **动作**：给定状态s，动作指的是$player(s)$在当前局面下可以采取的操作a，记动作集合为$actions(s)$
- **状态转移**：给定状态s和动作$a \in actions(s)$，状态转移函数$result(s,a)$决定了在s状态采取a动作后所得后继状态
- **终局状态检测**：终止状态检测函数$terminal_test(s)$用于测试游戏是否在状态s结束
- **终局得分**：终局得分$utility(s,p)$表示在终局状态s时玩家p的得分

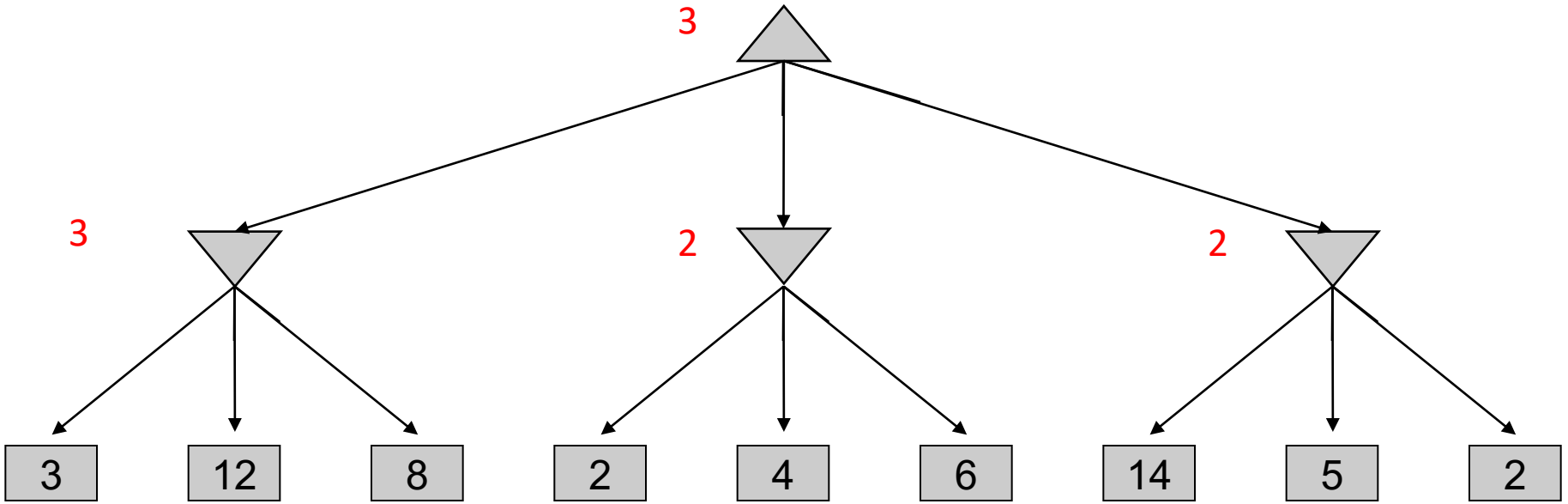
搜索树



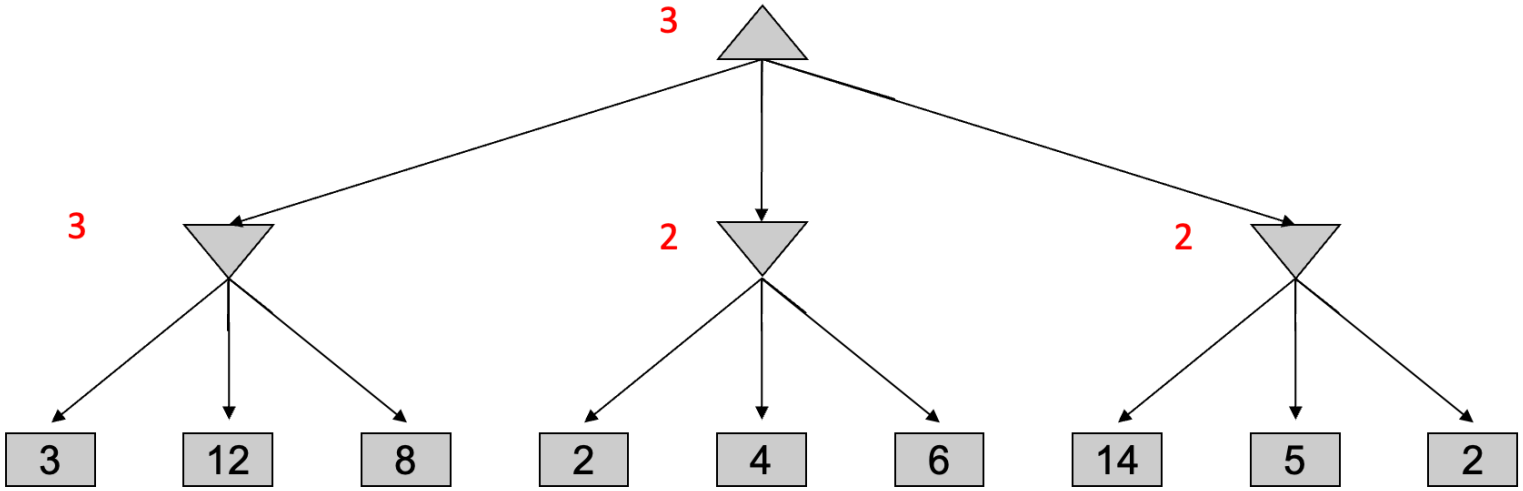
最优策略


MAX
MIN
A 1
A 2
A 3
A 11
A 12
A 13
A 21
A 22
A 23
A 31
A 32
A 33
3
12
8
2
4
6
14
5
2

最优策略



最优策略

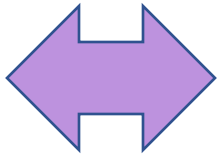


给定一棵博弈树，最优策略可以通过检查每个节点的极小极大值来决定，记为minimax(n)

$$\text{minimax(s)} = \begin{cases} \text{utility}(s), & \text{if terminal\_test}(s) \\ \max_{a \in \text{actions}(s)} \text{minimax}\big(\text{result}(s,a)\big), & \text{if player}(s) = MAX \\ \min_{a \in \text{actions(s)}} \text{minimax}\big(\text{result}(s,a)\big), & \text{if player}(s) = MIN \end{cases}$$

最小最大搜索

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



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

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

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

最小最大搜索

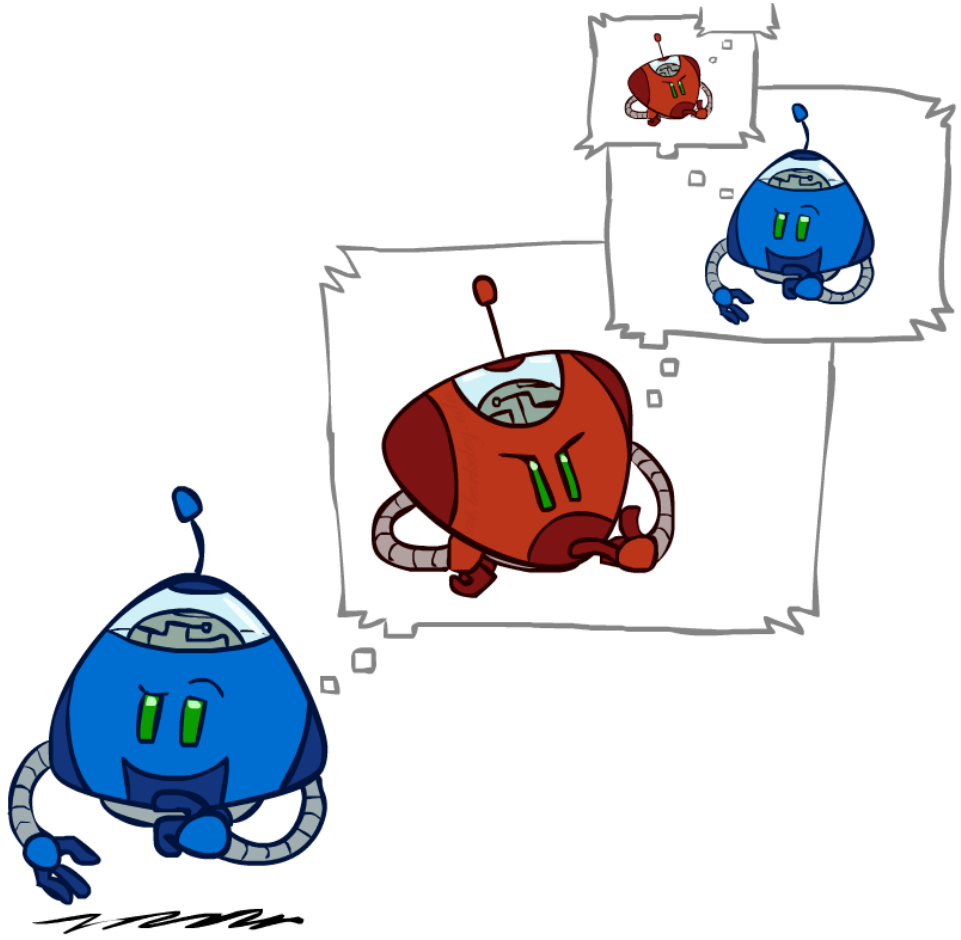
```
def value(state):
    if the state is a terminal state: return the state's utility
    if the agent is MAX: return max-value(state)
    if the agent is MIN: return min-value(state)
```

```
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
```

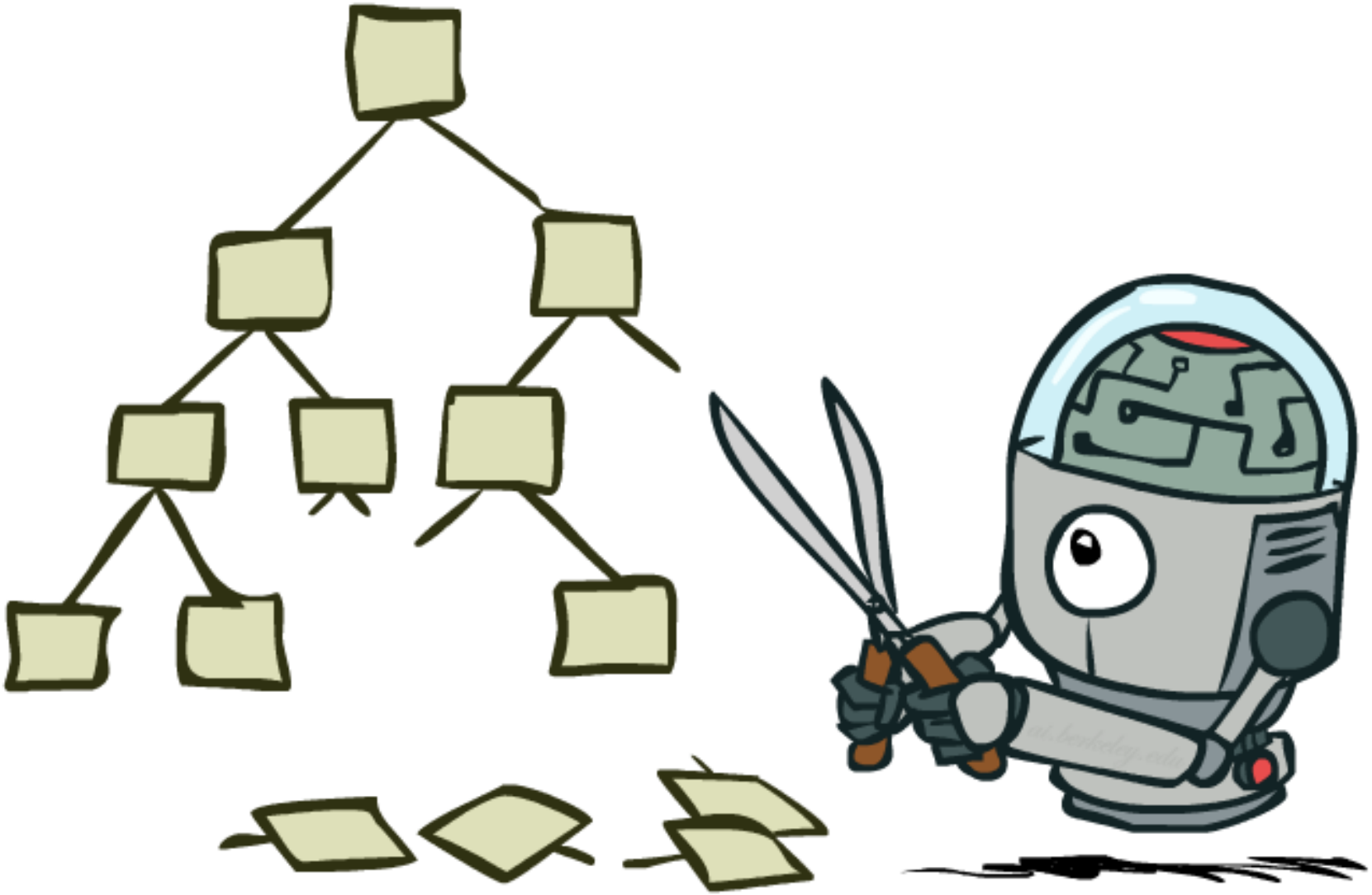
性能分析

- 时间和空间?
  - 和 DFS 类似
  - 时间复杂度: $O(b^m)$
  - 空间复杂度: O(bm)



- Example: For chess, $b \approx 35$, $m \approx 100$
  - 精确的搜索几乎是不可行的

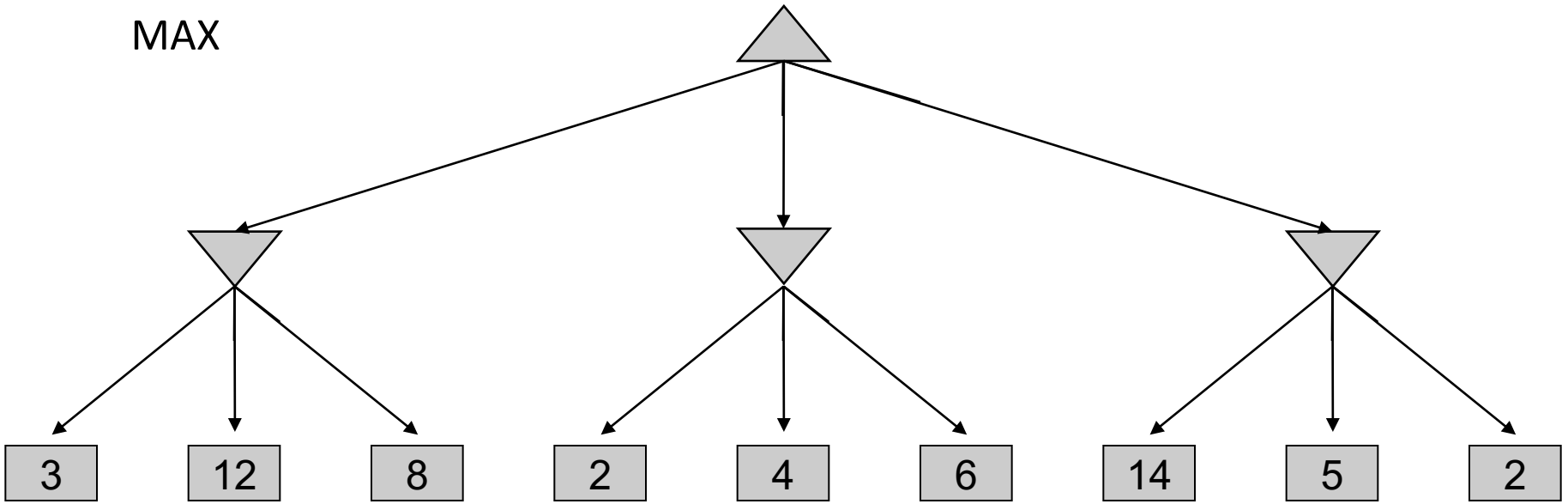
树剪枝



提纲

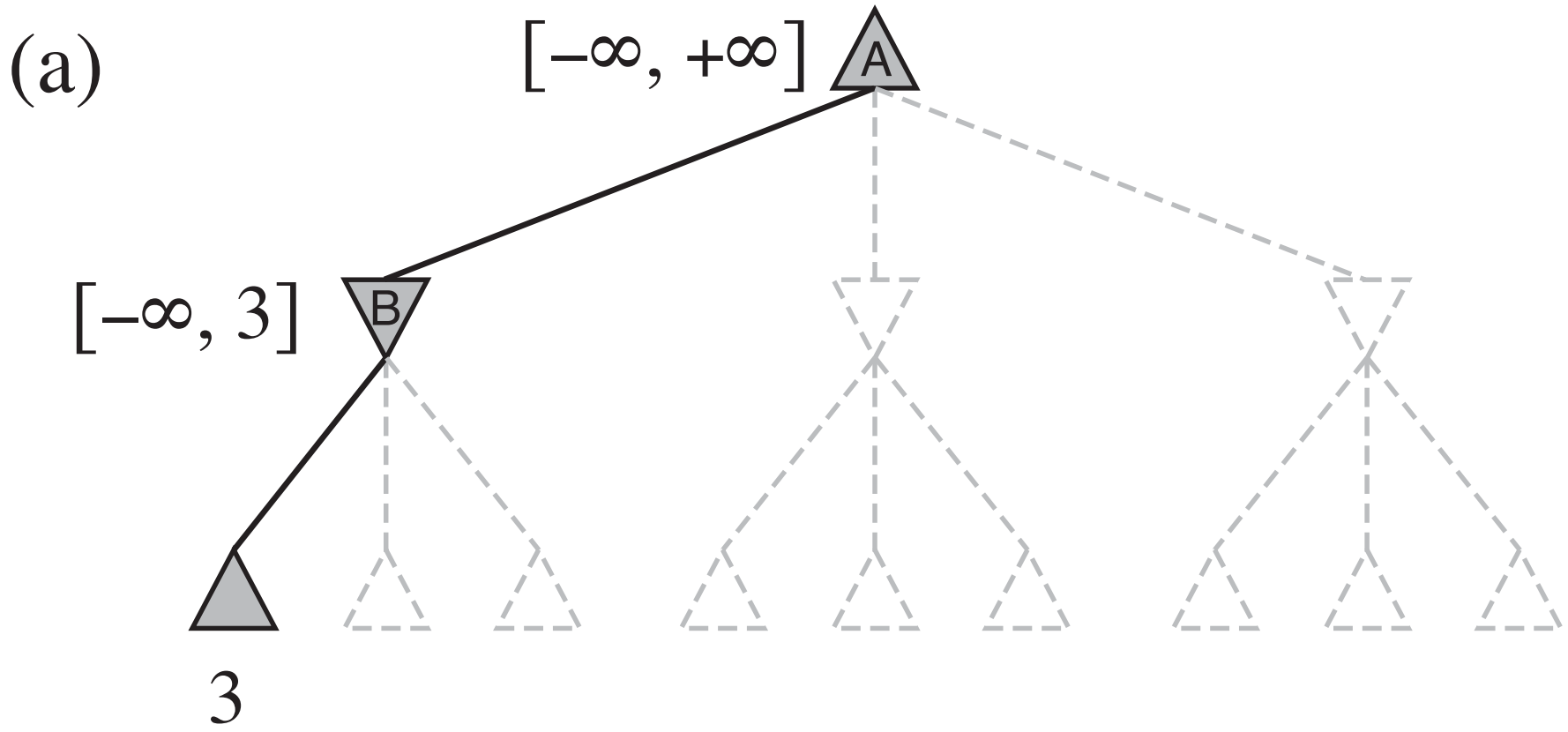
- 对抗博弈
  - 双人零和博弈
- 确定性搜索
  - 最大最小搜索
  - Alpha-beta 剪枝
- 基于模拟的搜索
  - 蒙特卡洛树搜索

树剪枝



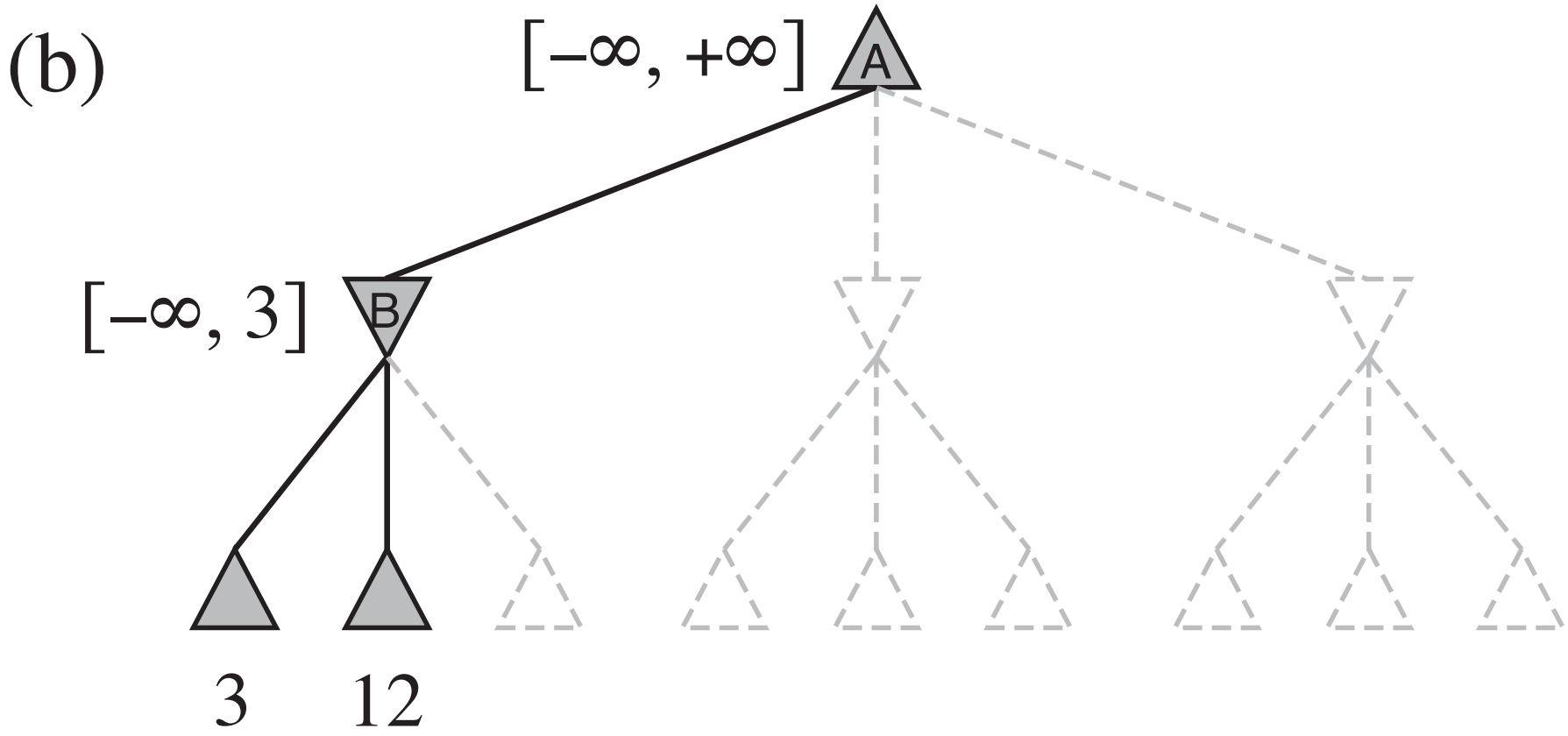
树剪枝

Vector:[alpha, beta]



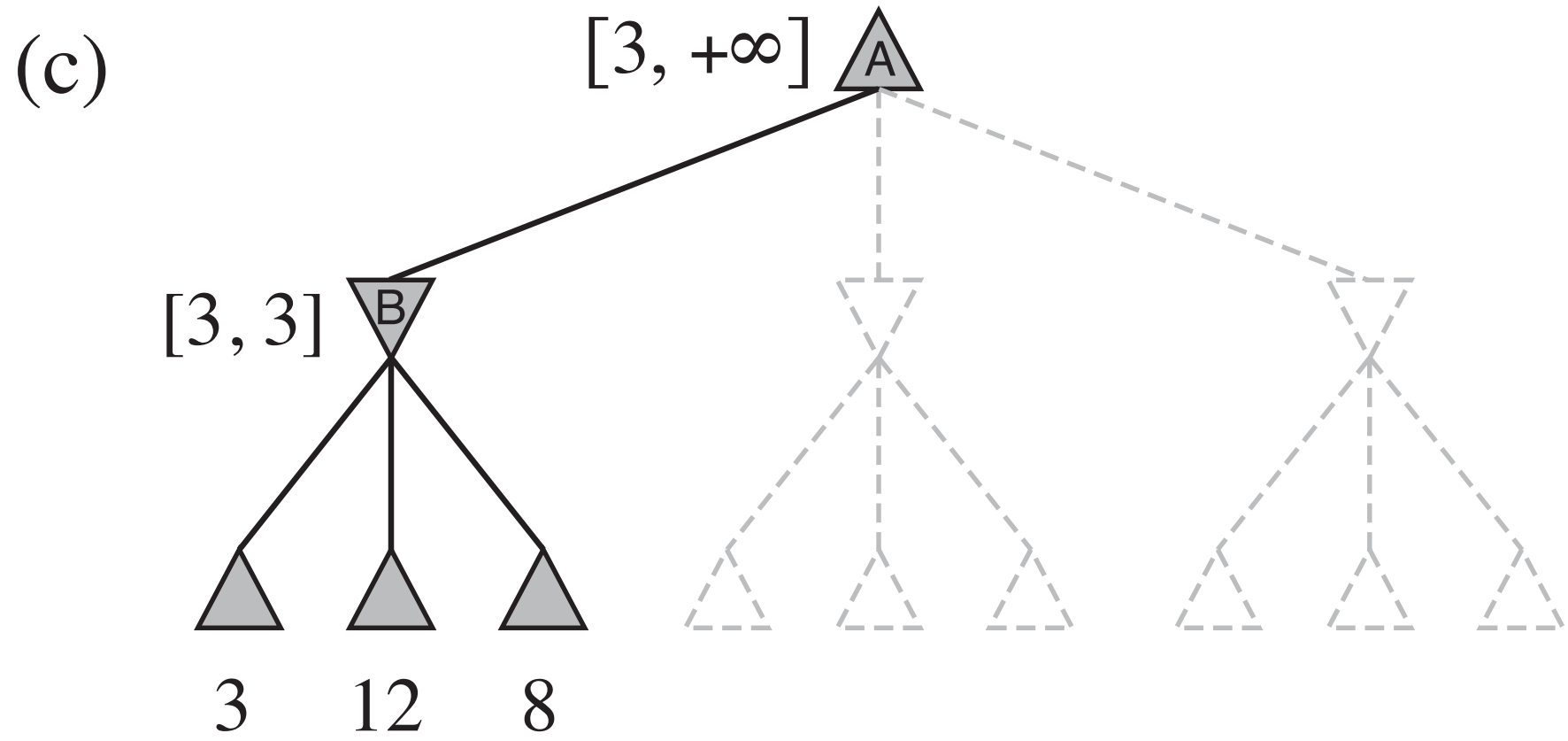
树剪枝

Vector:[alpha, beta]



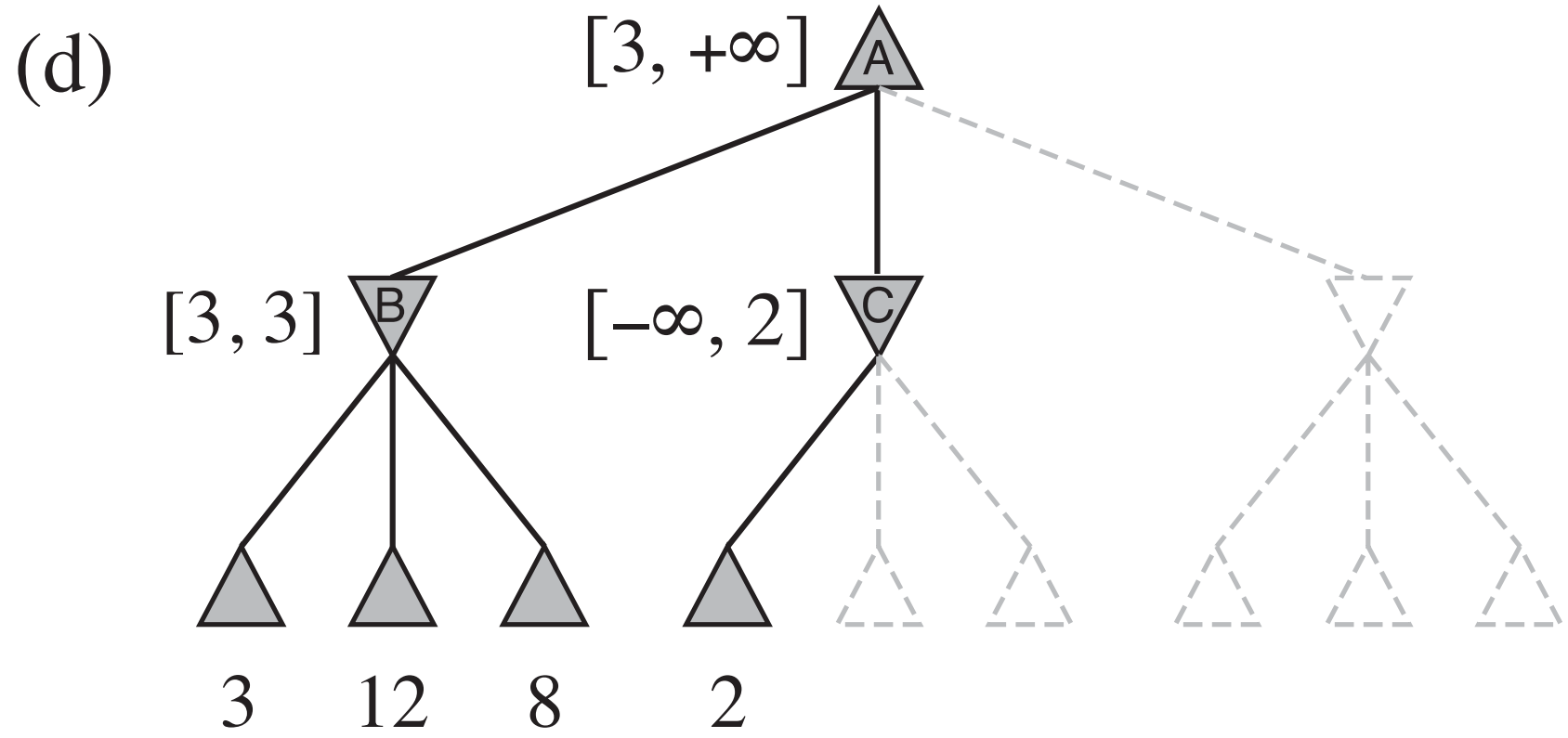
树剪枝

Vector:[alpha, beta]



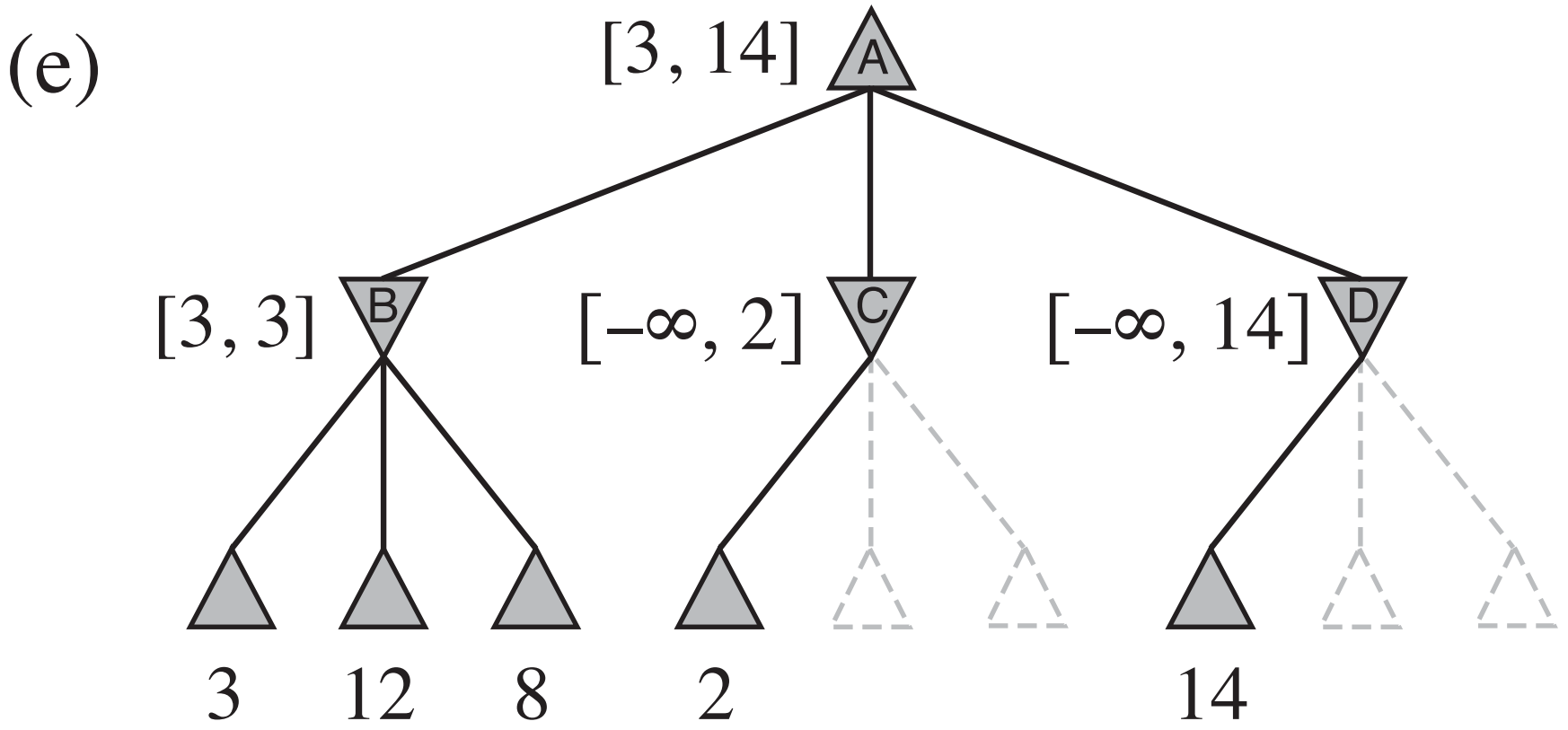
树剪枝

Vector:[alpha, beta]



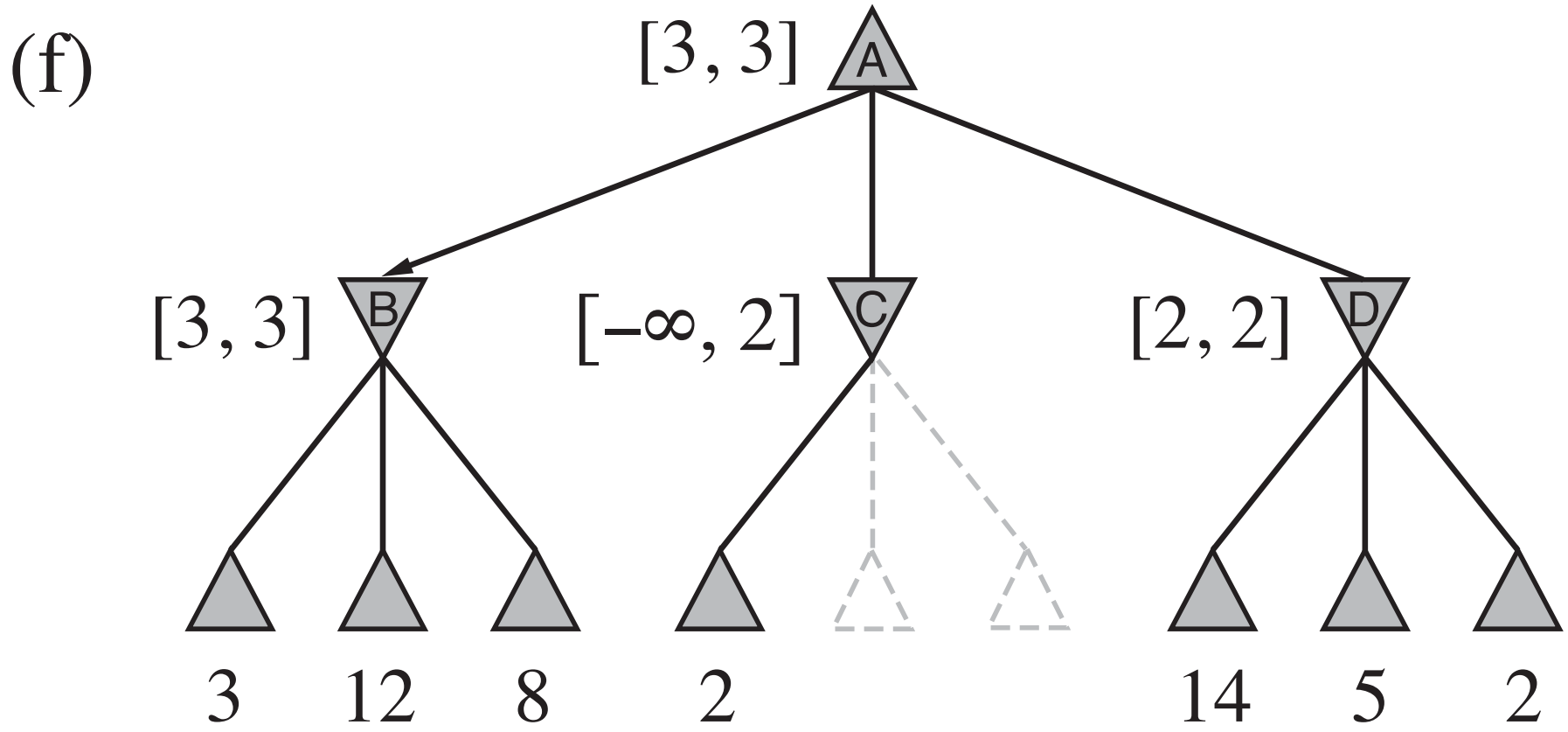
树剪枝

Vector:[alpha, beta]

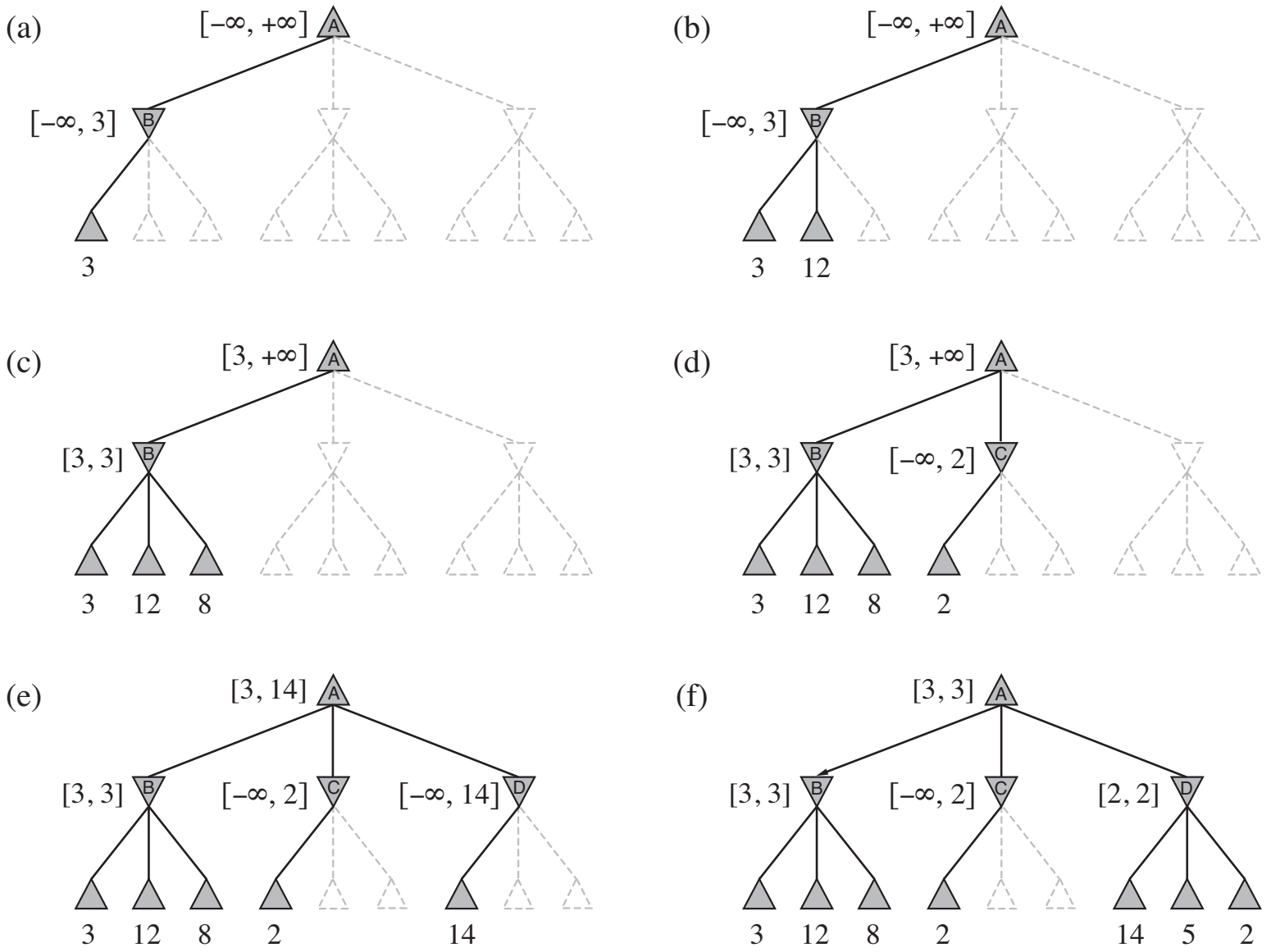


树剪枝

Vector:[alpha, beta]



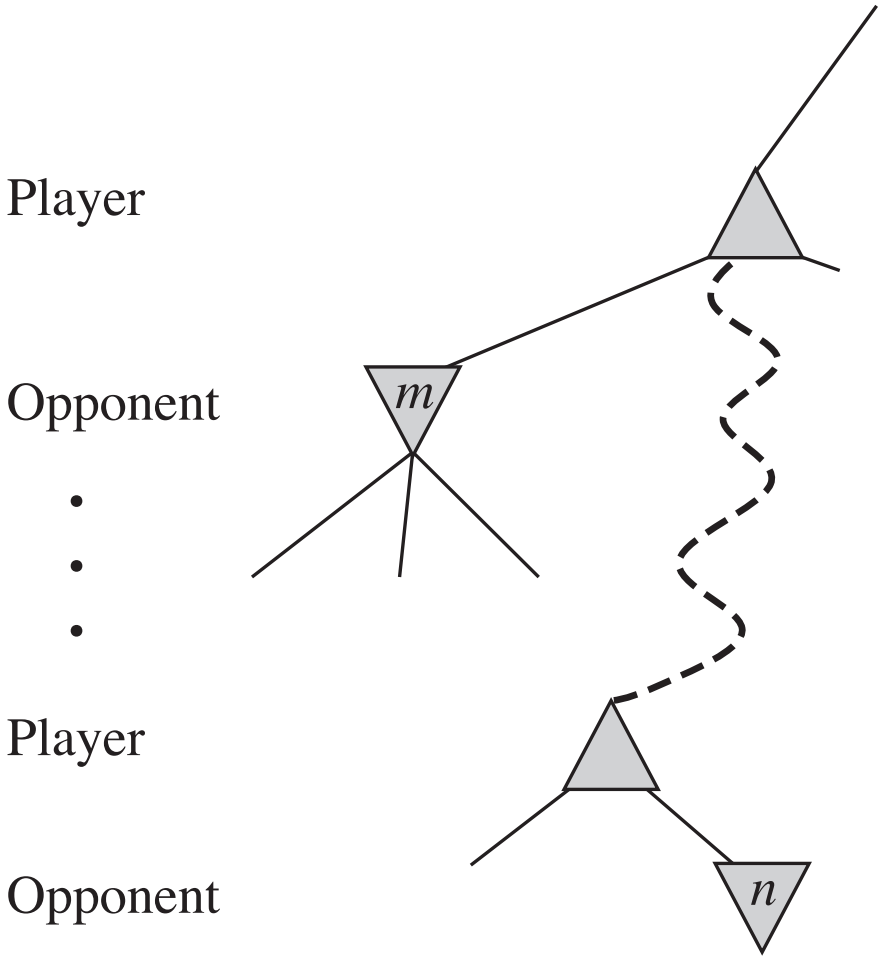
树剪枝


(a)
[−∞, +∞]
A
[−∞, 3]
B
3
(b)
[−∞, +∞]
A
[−∞, 3]
B
3
12
(c)
[3, +∞]
A
[3, 3]
B
3
12
8
(d)
[3, +∞]
A
[3, 3]
B
[−∞, 2]
C
3
12
8
2
(e)
[3, 14]
A
[3, 3]
B
[−∞, 2]
C
[−∞, 14]
D
3
12
8
2
14
(f)
[3, 3]
A
[3, 3]
B
[−∞, 2]
C
[2, 2]
D
3
12
8
2
14
5
2

Alpha-Beta剪枝

α:目前为止路上发现的 MAX的最佳（即极大值）选择，即α=“至少”

β:目前为止路径上发现的MIN的最佳（即极小值）选择，即β=“至多”



如果对于玩家来说， m好于n，

那么我们永远不会在博弈中到达n

Alpha-Beta剪枝

- 对于MAX节点，如果其孩子结点（MIN结点）的收益大于当前的α值，则将α值更新为该收益；

  对于MIN结点，如果其孩子结点（MAX结点）的收益小于当前的β值，则将β值更新为该收益。

  根结点（MAX结点）的α值和β值分别被初始化为$-\infty$和$+\infty$

- 随着搜索算法不断被执行，每个结点的α值和β值不断被更新。大体来说，每个结点的$[\alpha, \beta]$从其父结点提供的初始值开始，取值按照如下形式变化：α逐渐增加、β逐渐减少。不难验证，如果一个结点的α值和β值**满足$\alpha > \beta$**的条件，则该结点尚未被访问的后续结点就会被剪枝，因而不会被访问

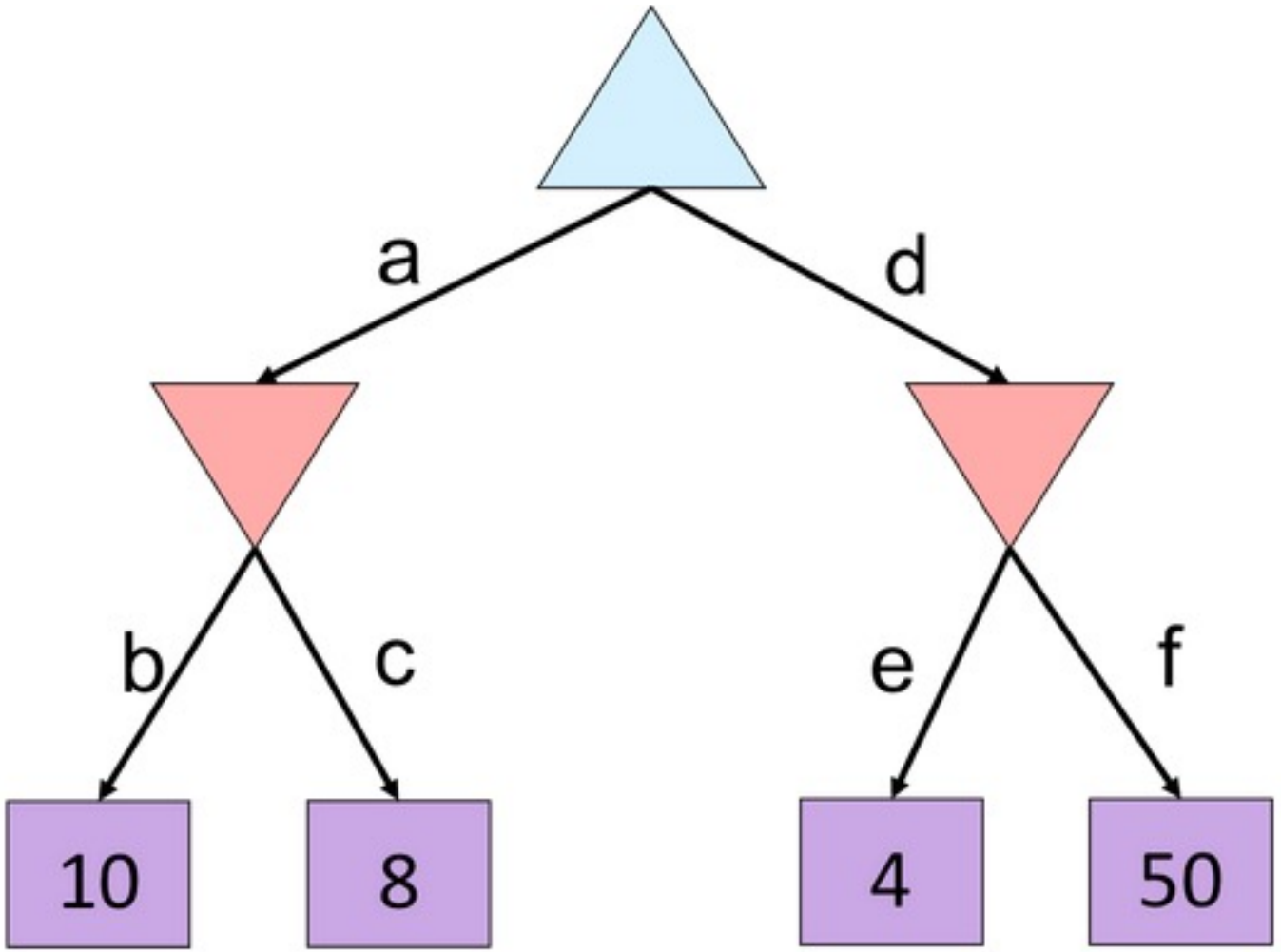
Alpha-Beta剪枝

α:目前为止路上发现的 MAX的最佳（即极大值）选择
β:目前为止路径上发现的MIN的最佳（即极小值）选择

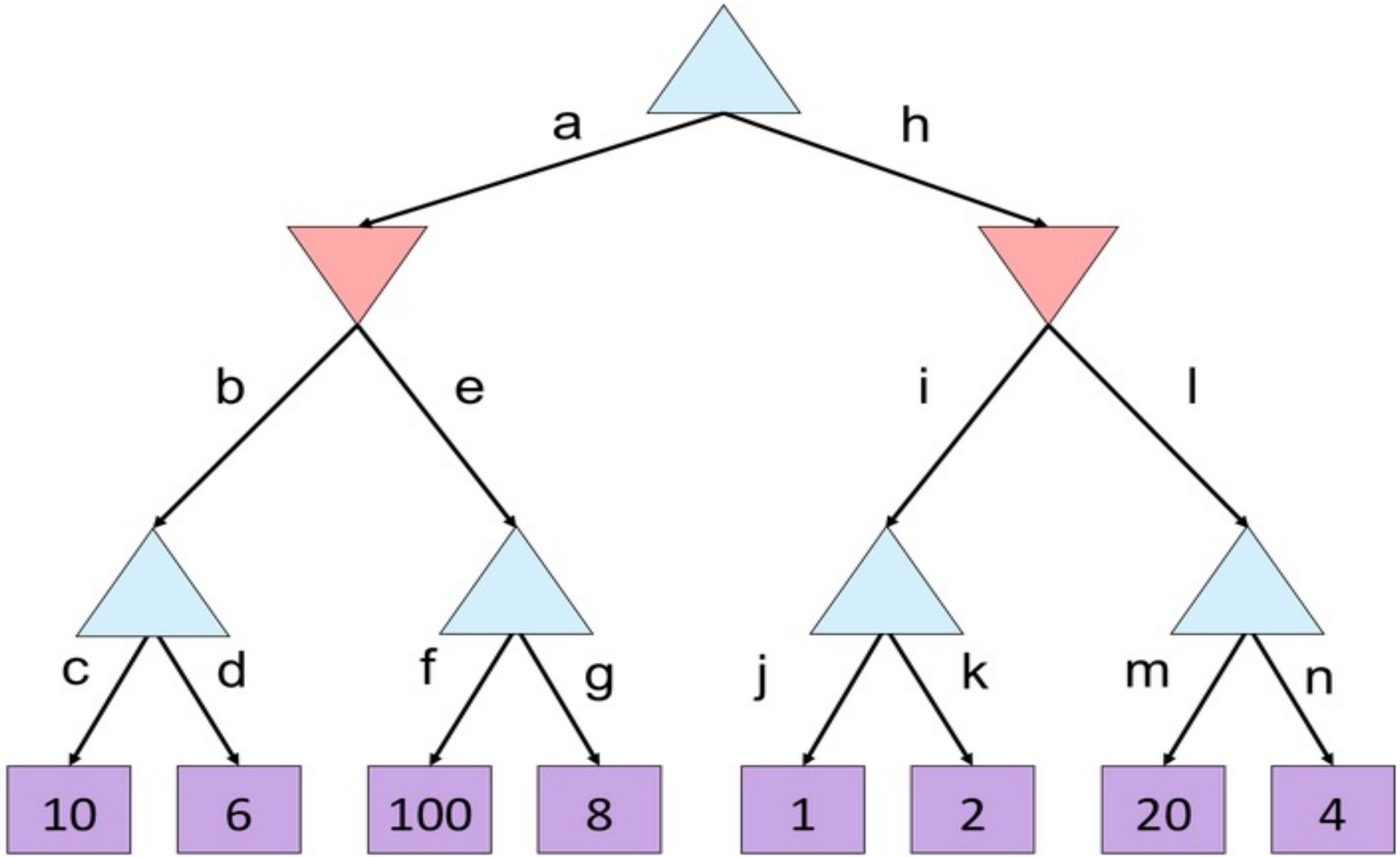
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
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
```

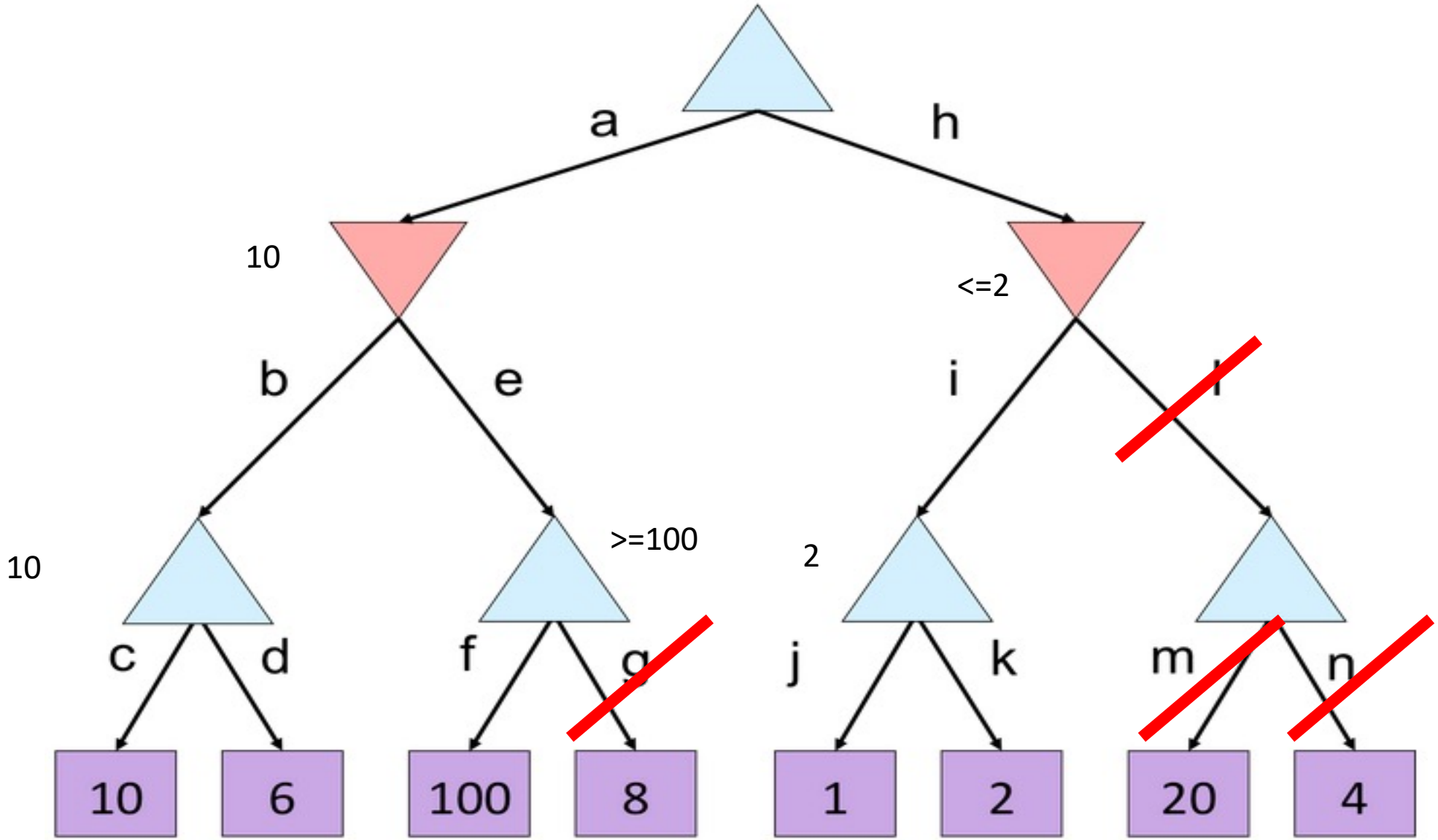
Quiz1: Alpha-Beta剪枝



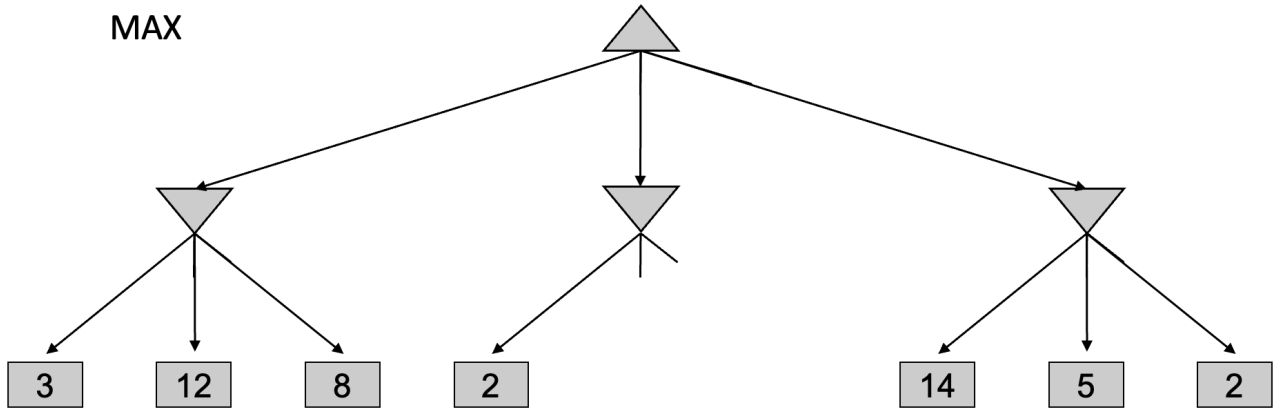
Quiz2: Alpha-Beta剪枝



Quiz2: Alpha-Beta剪枝



思考



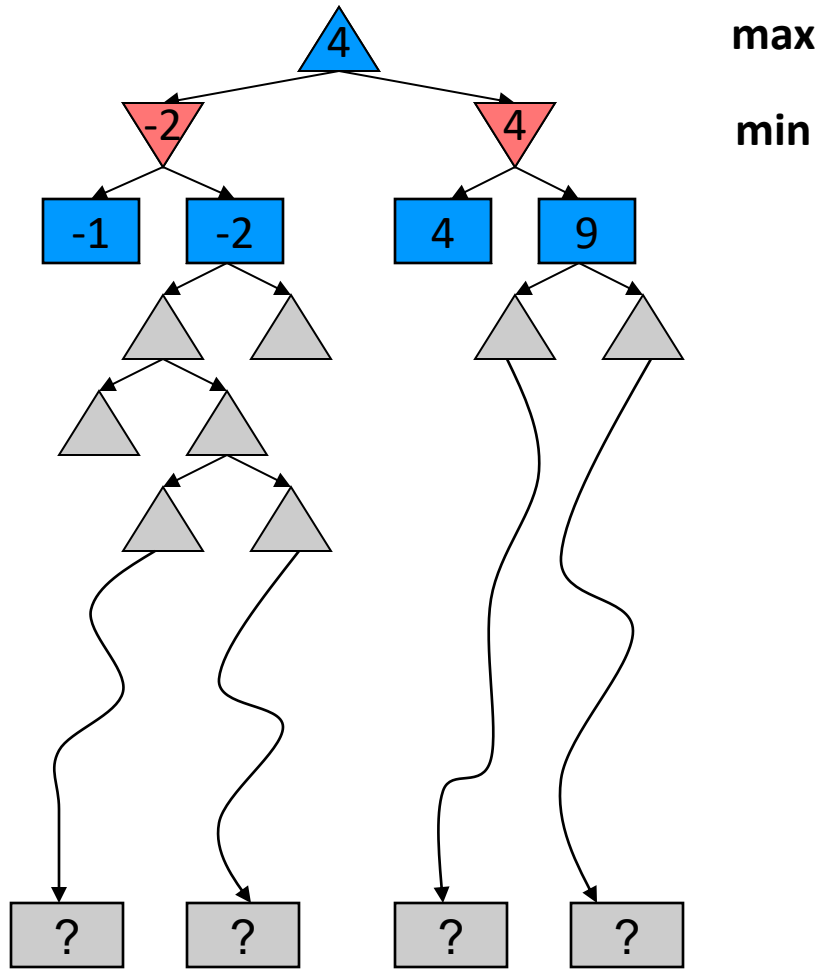
启发? **搜索顺序很重要**

可以设计方案对后继状态进行排序，

例如，对于象棋，可以设计排序规则：吃子>威胁>前进>后退

资源受限

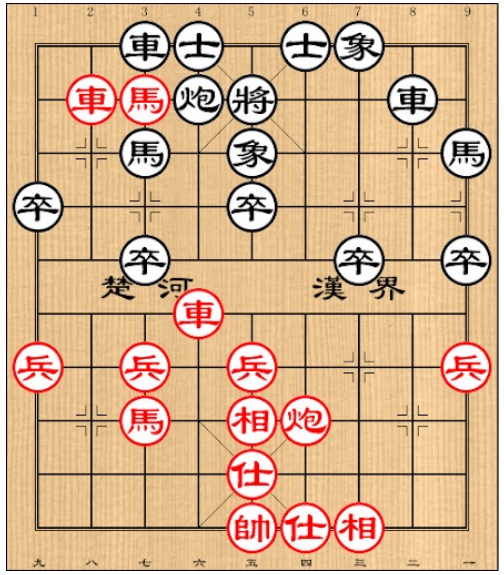
尽管alpha-beta剪枝能够避免搜索完整的空间，但是仍然要**搜索部分空间直至终止状态**，这样的搜索深度也是不现实的

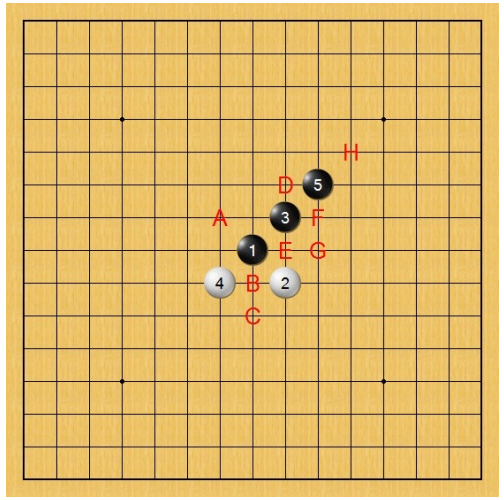


一个可行的思路：

参考启发式搜索，设计评估函数用于搜索中的状态，有效地把非终止节点变成终止节点

评估函数





如何设置评估函数?

以象棋为例，要考虑兵的数目、车的数目、马的数目等等

兵1分，马3分，车5分...

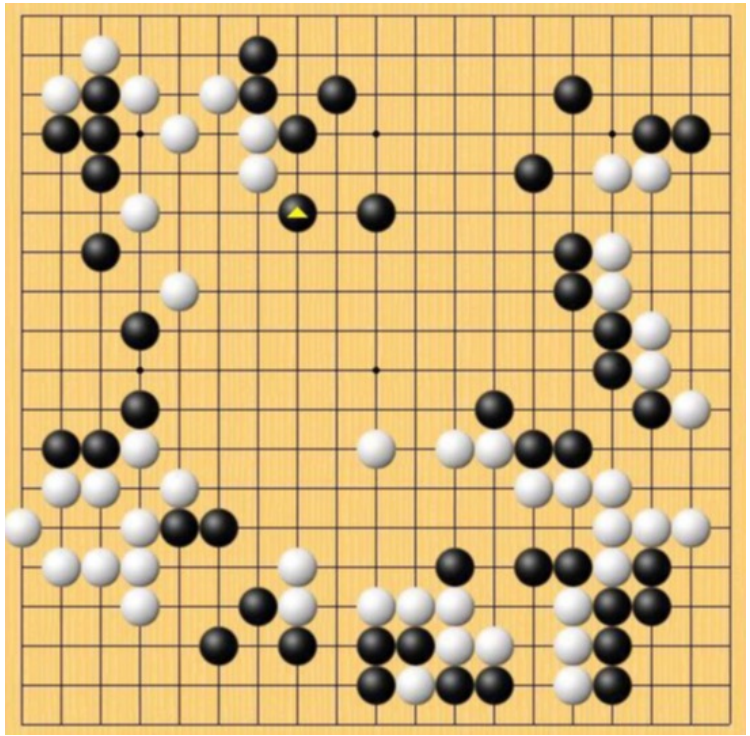
形式化描述：**加权线性函数**

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

w_i表示权重，f_i是棋局的某个特征

评估函数

Alpha-beta搜索用于围棋会面临什么挑战?



- **分支因子大**

围棋分支因子开始时为361，搜索层数受限

- **评估函数难设置**

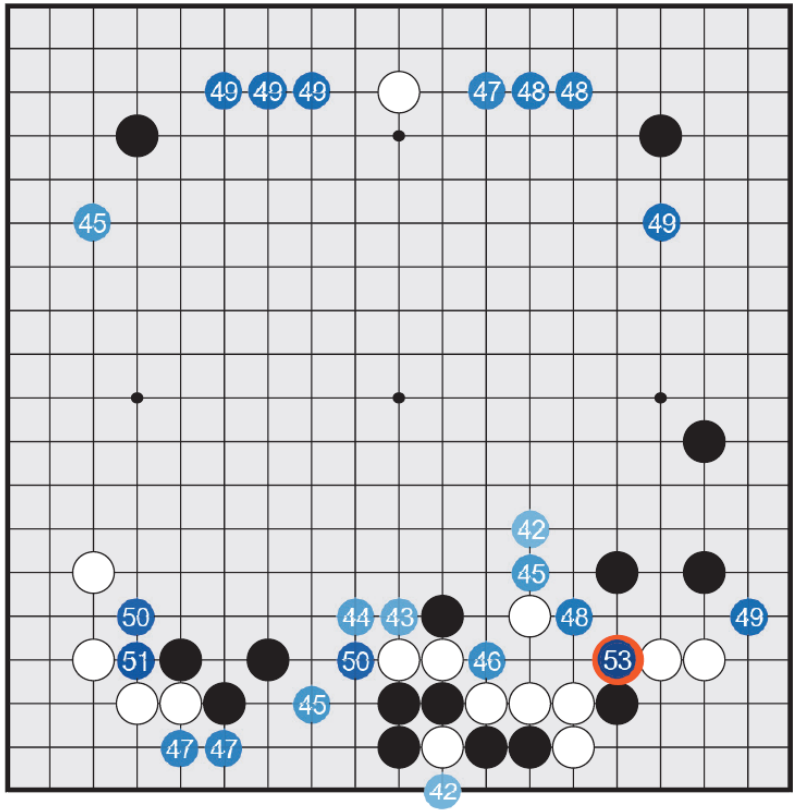
现代围棋程序基本不采用alpha-beta搜索

提纲

- 对抗博弈
  - 双人零和博弈
- 确定性搜索
  - 最大最小搜索
  - Alpha-beta 剪枝
- 基于模拟的搜索
  - 蒙特卡洛树搜索

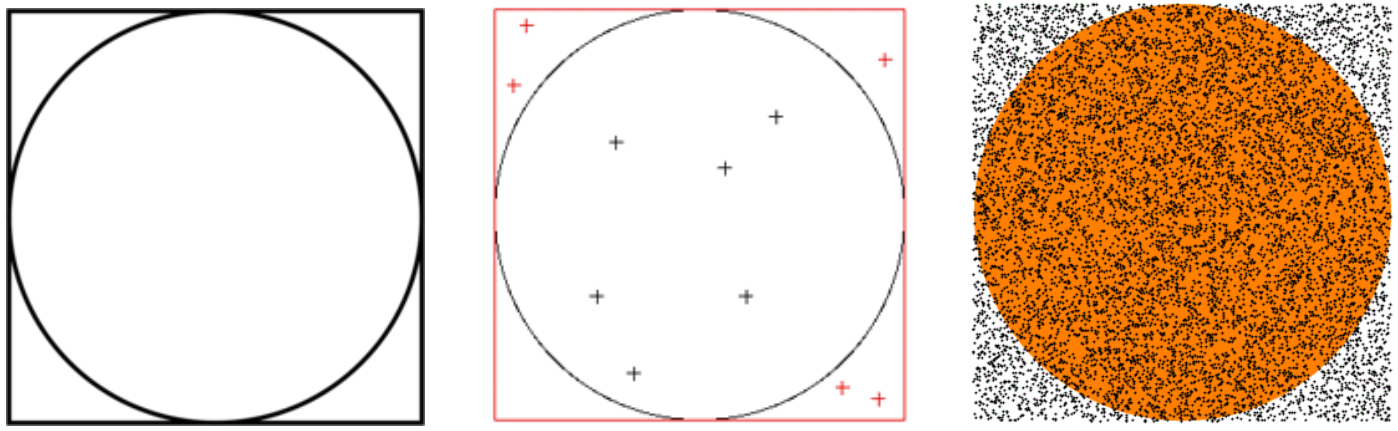
状态评估

如果不采用评价函数，有没有其他办法评估状态的好坏?



蒙特卡洛方法

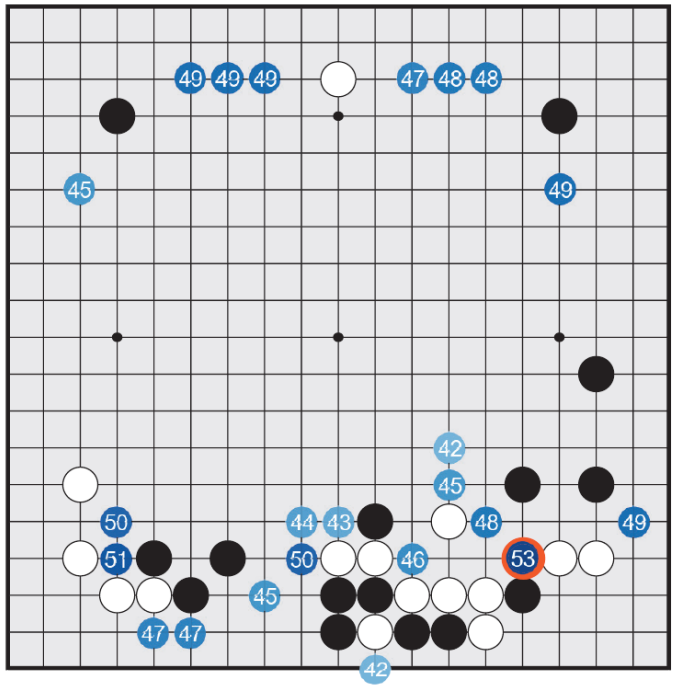
- 蒙特卡洛方法（Monte-Carlo methods）是一类广泛的计算算法
  - 依赖于重复随机抽样来获得数值结果
- 例如，计算圆的面积



$$\text{Circle Surface} = \text{Square Surface} \times \frac{\#\text{points in circle}}{\#\text{points in total}}$$

蒙特卡洛方法

围棋对弈：估计当前状态下的胜率

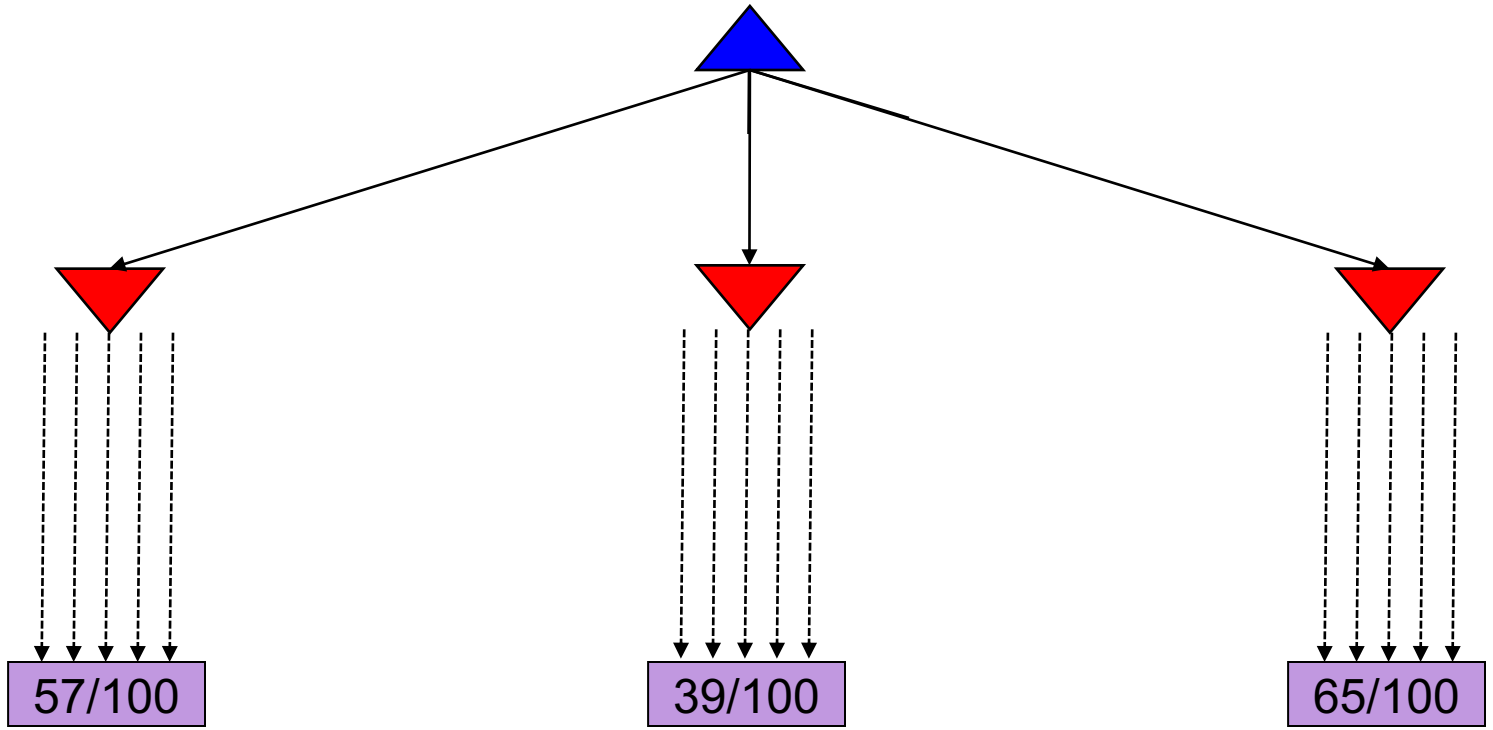


从当前状态出发，做随机模拟

$$\text{Win Rate}(s) = \frac{\#\text{win simulation cases started from } s}{\#\text{simulation cases started from } s \text{ in total}}$$

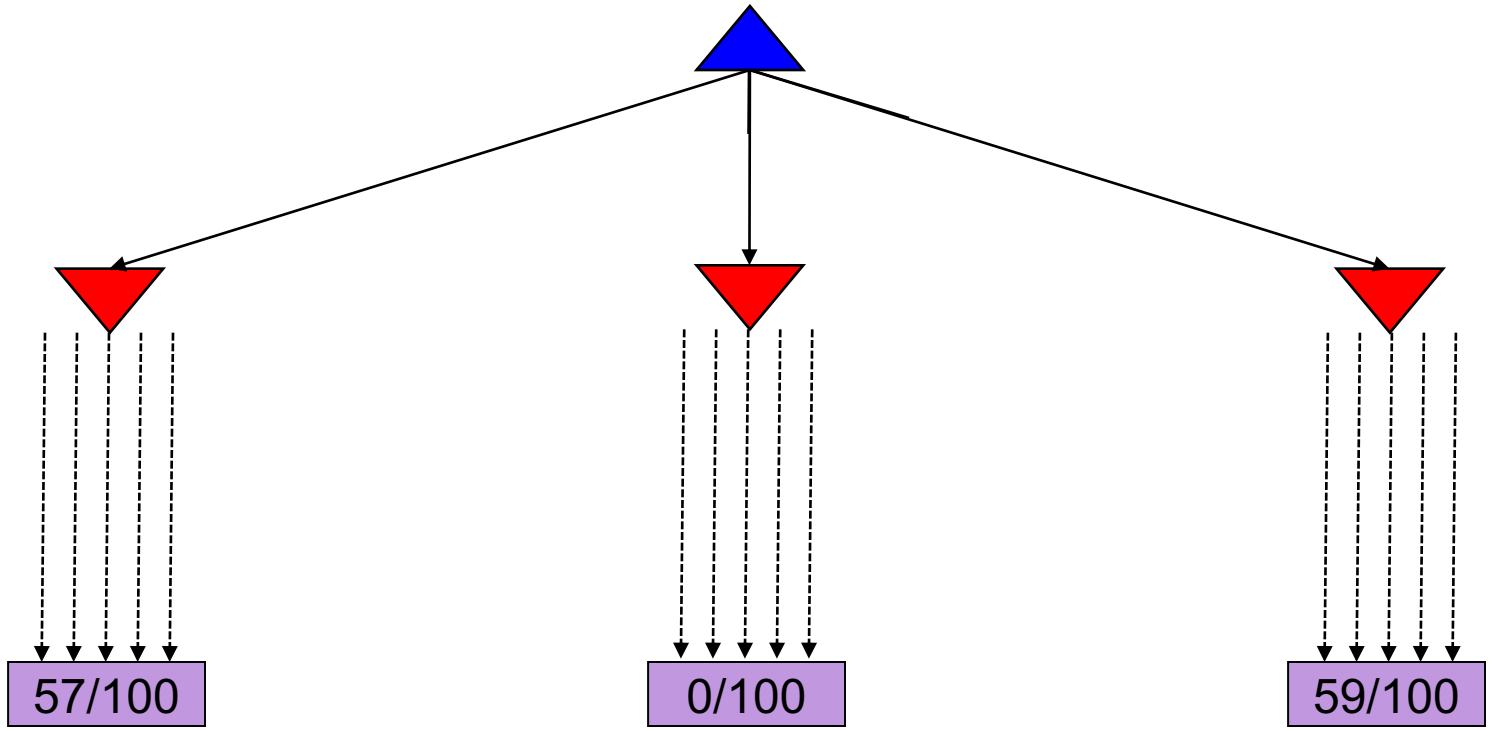
蒙特卡洛搜索

模拟：从一个节点出发，进行大量模拟，记录赢的次数



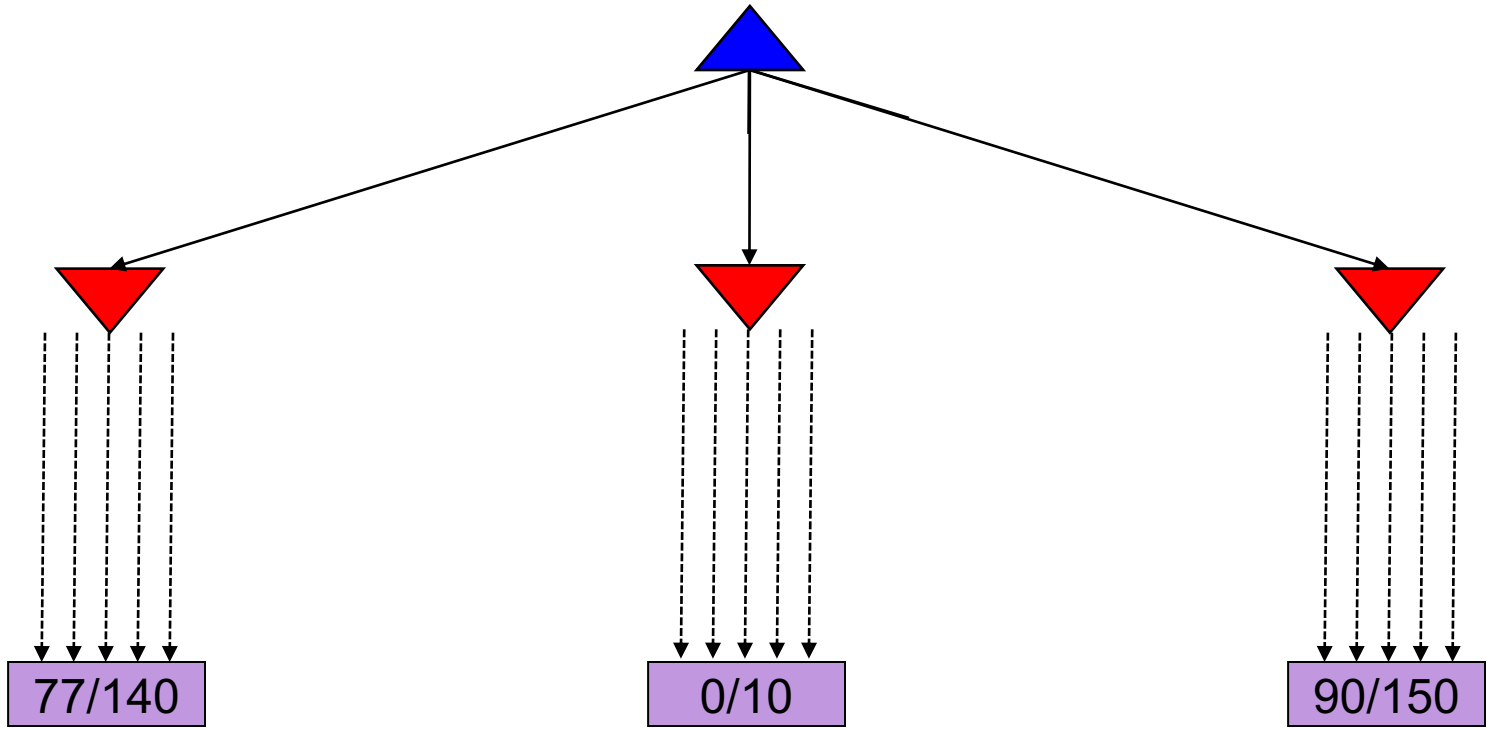
蒙特卡洛搜索

模拟：从一个节点出发，进行大量模拟，记录赢的次数



蒙特卡洛搜索

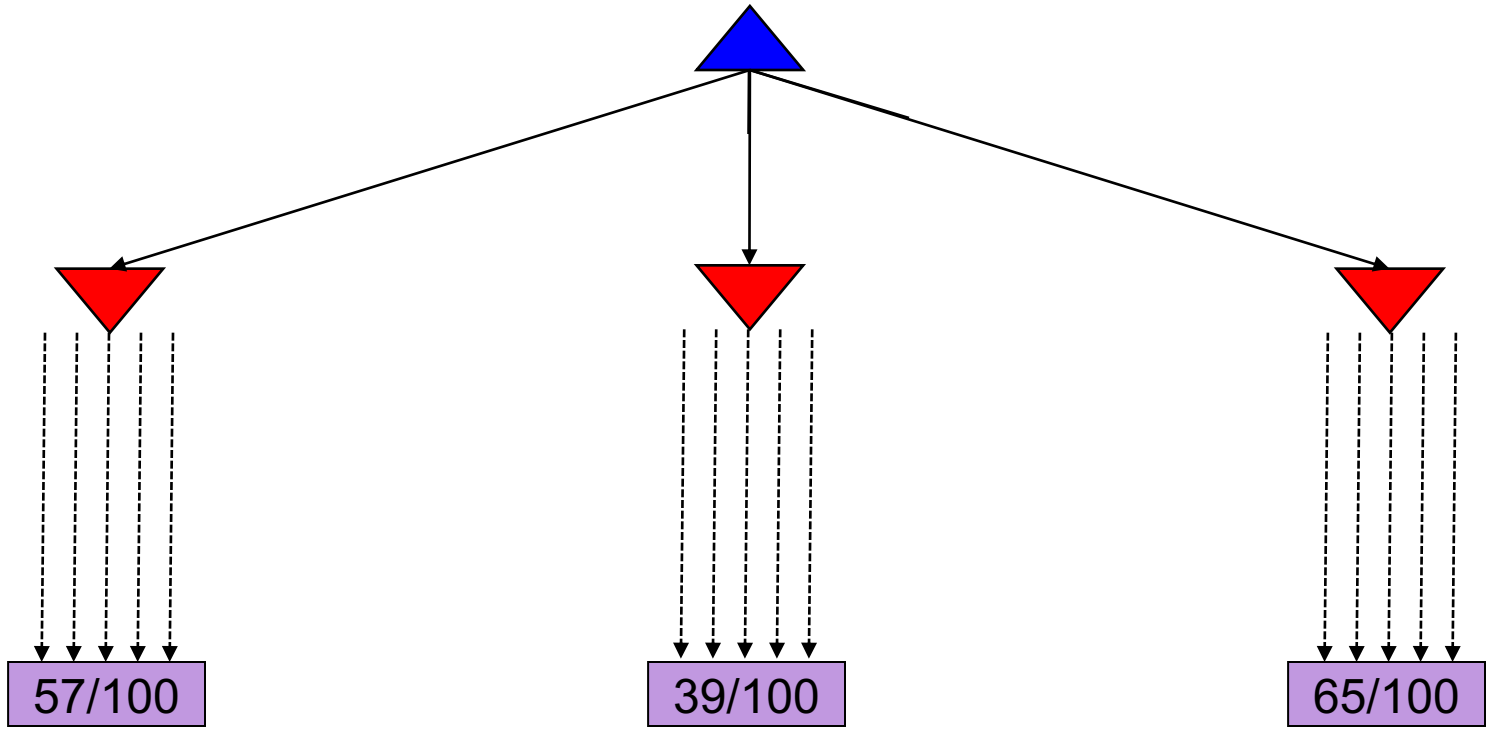
模拟：从一个节点出发，进行大量模拟，记录赢的次数



蒙特卡洛搜索

资源是有限的，选择哪些节点进行模拟？

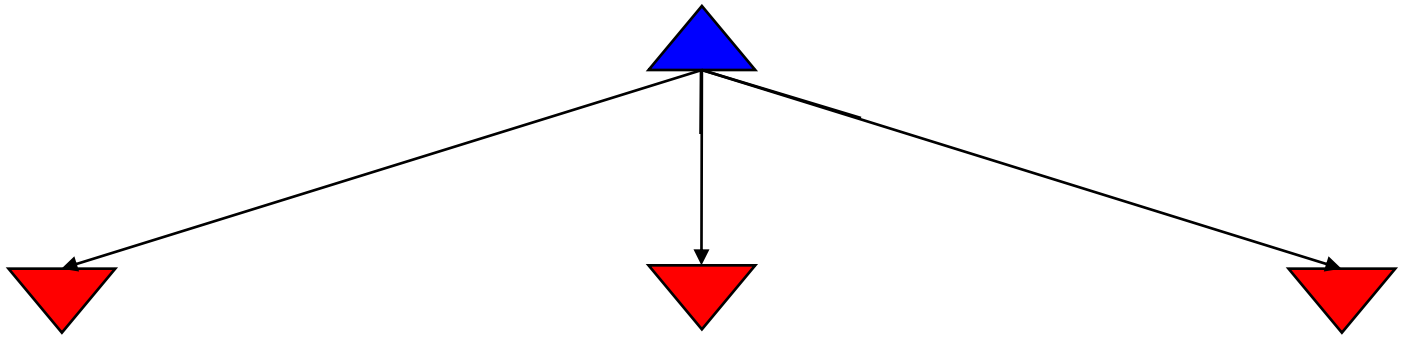
选择更有希望的节点进行模拟



蒙特卡洛搜索

如何评估节点的价值?

大量的模拟



先有鸡还是先有蛋?

多臂老虎机

多臂老虎机 （Multi-Armed Bandits）

赌博机有 K 个摇臂，每次转动一个赌博机摇臂，
赌博机则会随机吐出一些硬币

如何在有限次数的尝试中使收益最大化?

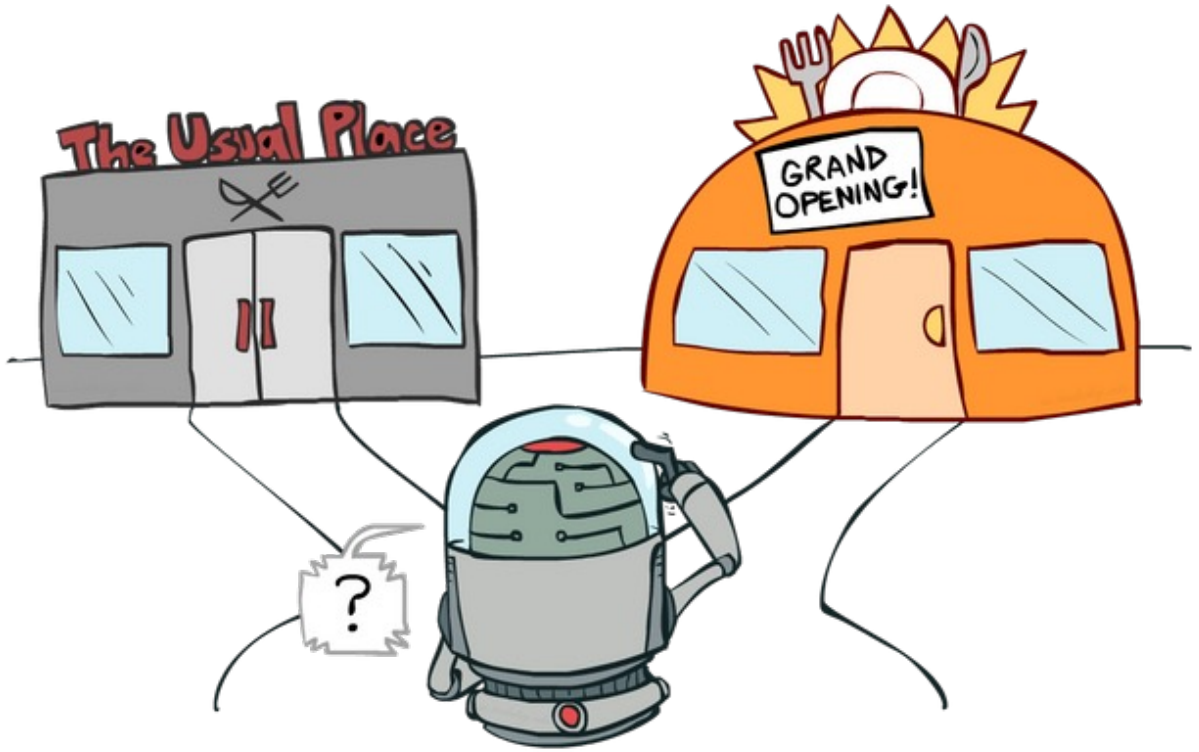
探索与利用：

- 探索(Exploration)：估计不同摇臂的优劣 (奖赏期望的大小)
- 利用(Exploitation)：选择当前最优的摇臂

探索-利用

基于目前策略获取已知最优收益还是尝试不同的决策

- Exploitation：执行能够获得已知最优收益的决策
- Exploration：尝试更多可能的决策，不一定会是最优收益



探索-利用窘境

如何在有限次数的尝试中使收益最大化?

- **仅探索（Exploration-only)**

每个摇臂摇动 T / K 次

不足：浪费次数在收益较差的摇臂上

- **仅利用（Exploitation-only)**

1. 每个摇臂摇动一次，记录收益
2. 剩余的T-K次全部用在收益最大的摇臂上

不足：一次估计的结果不可靠

探索利用窘境

Exploration-Exploitation Dilemma

ϵ-贪心算法

ϵ-贪心算法：在探索与利用之间进行平衡的搜索算法

在第t步，ϵ-贪心算法按照如下机制来选择摇动赌博机：

- 以 **$1-\epsilon$ 的概率**，选择在过去$t-1$次摇动赌博机所得**平均收益最高的摇臂**进行摇动；
- 以 **ϵ 的概率**，**随机选择一个摇臂**进行摇动。

不足：没有考虑每个摇臂被探索的次数

输入： 摇臂数 K;
奖赏函数 R;
尝试次数 T;
探索概率 ϵ.

过程：

$r = 0$;
$\forall i = 1, 2, \ldots K : Q(i) = 0,\ \text{count}(i) = 0$;
for $t = 1, 2, \ldots, T$ **do**
  if rand()$< \epsilon$ **then**
    $k =$ 从 $1, 2, \ldots, K$ 中以均匀分布随机选取
  else
    $k = \arg\max_i Q(i)$
  end if
  $v = R(k)$;
  $r = r + v$;
  $Q(k) = \frac{Q(k) \times \text{count}(k) + v}{\text{count}(k) + 1}$;
  $\text{count}(k) = \text{count}(k) + 1$;
end for

输出： 累积奖赏 r

Softmax

- Softmax：基于当前已知的摇臂平均奖赏来对探索与利用折中
  - 若某个摇臂当前的平均奖赏越大，则它被选择的概率越高
  - 概率分配使用Boltzmann分布：

$$P(k) = \frac{e^{\frac{Q(k)}{\tau}}}{\sum_{i=1}^{K} e^{\frac{Q(i)}{\tau}}}$$

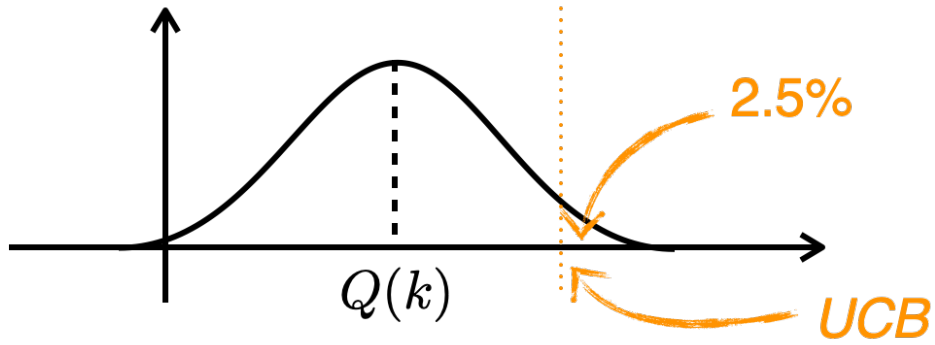
ϵ-贪心与Softmax算法都有一个折中参数(ϵ, τ)，算法性能孰好孰坏取决于具体应用问题

上限置信区间（Upper-Confidence Bound）

上限置信区间算法（Upper Confidence Bounds, UCB）： 为每个动作的奖励期望计算一个估计范围，优先采用估计范围上限较高的动作

假设每个摇臂的均值为$Q(k)$，估计的偏差为$\delta(k)$

每次根据$Q(k)+\delta(k)$选择摇臂



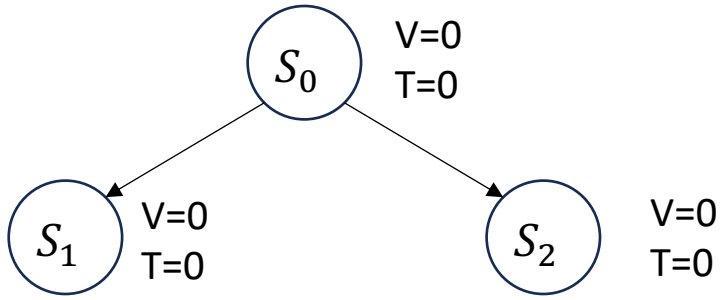
$$UCB(k) = Q_k + C * \sqrt{{ln\ T}/{T_k}}$$

蒙特卡洛搜索

- **选择（selection）**：从根节点开始，按照某种选择策略（通常为UCB）向下选择子节点，直至到达叶子结点
- **扩展（expansion）**：为所选节点生成一个新的子节点 （注意：MCTS有不同实现方式，有的是每次随机生成一个子节点，有的是生成所有后继节点再进行选择）
- **模拟（simulation）**：从新生成的节点出发出发，模拟扩展搜索树，获得模拟结果
- **回溯（backpropagation）**：基于模拟结果自底向上更新路径的节点的奖励均值和被访问次数（注意：只有胜方的获胜次数和模拟次数都会增加，败方节点只增加模拟次数）

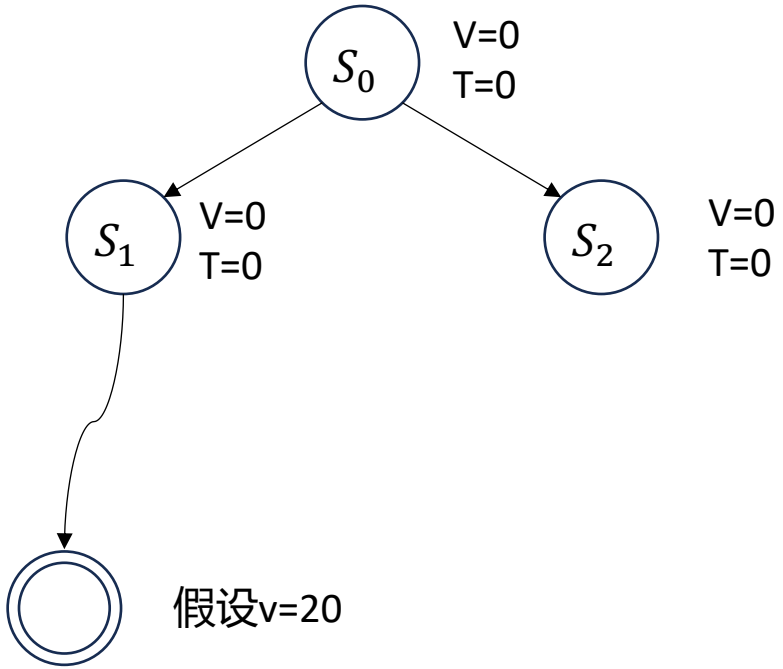
蒙特卡洛搜索

$$UCB(k) = Q_k + 2 * \sqrt{{ln\ T}/{T_k}}$$



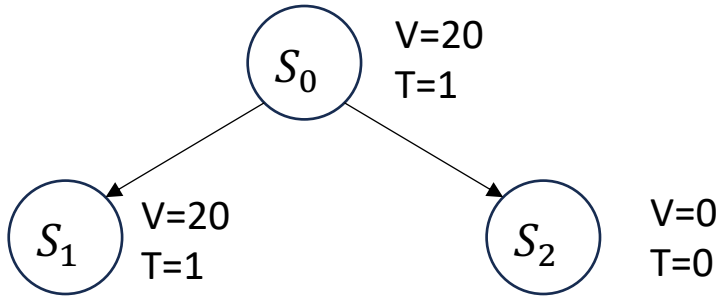
蒙特卡洛搜索

$$UCB(k) = Q_k + C * \sqrt{ln\,T / T_k}$$



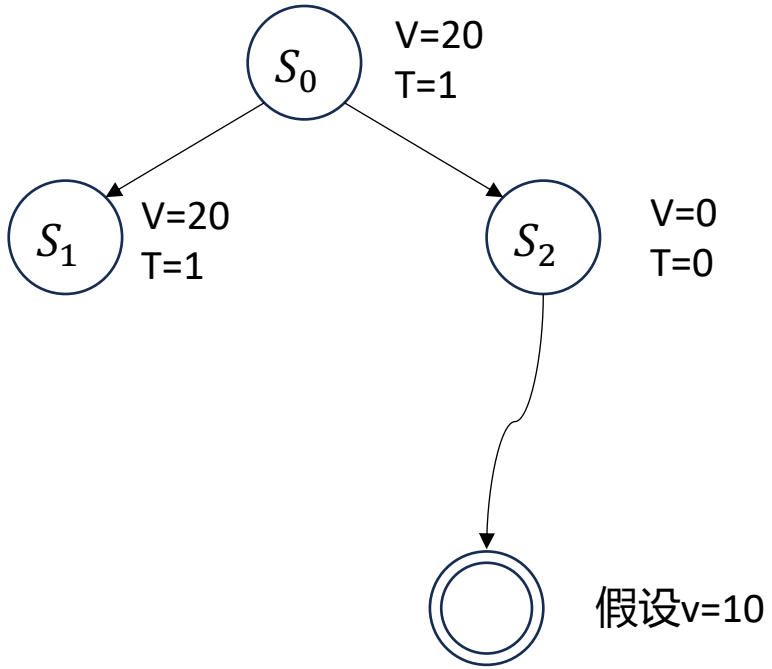
蒙特卡洛搜索

$$UCB(k) = Q_k + C * \sqrt{ln\,T / T_k}$$



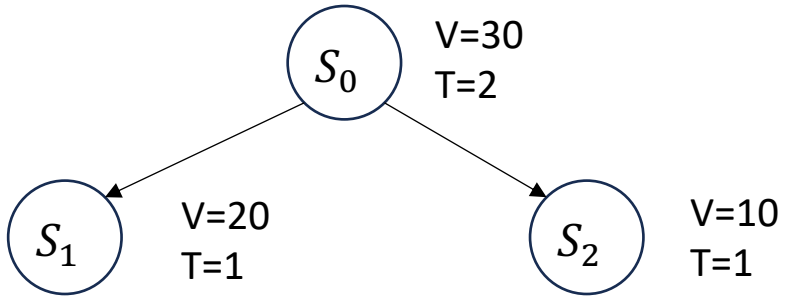
蒙特卡洛搜索

$$UCB(k) = Q_k + C * \sqrt{ln\,T / T_k}$$



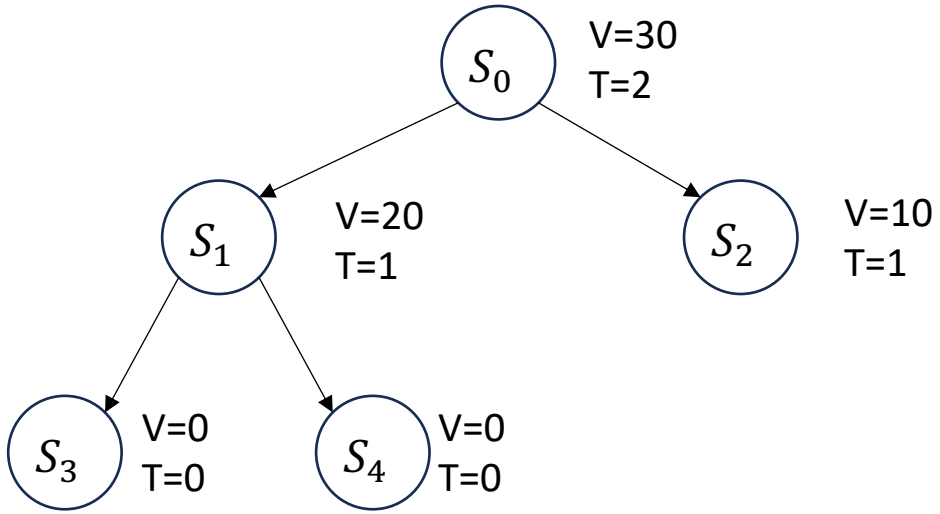
蒙特卡洛搜索

$$UCB(k) = Q_k + C * \sqrt{ln\,T / T_k}$$



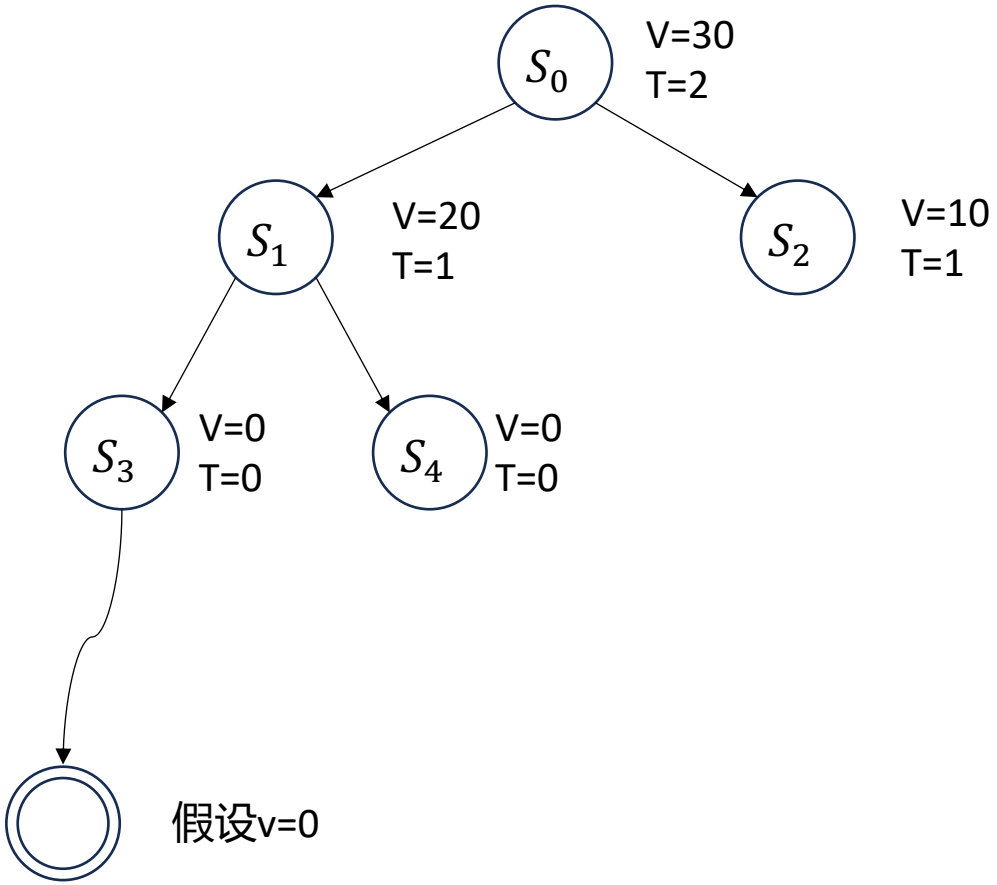
蒙特卡洛搜索

$$UCB(k) = Q_k + C * \sqrt{ln\,T / T_k}$$



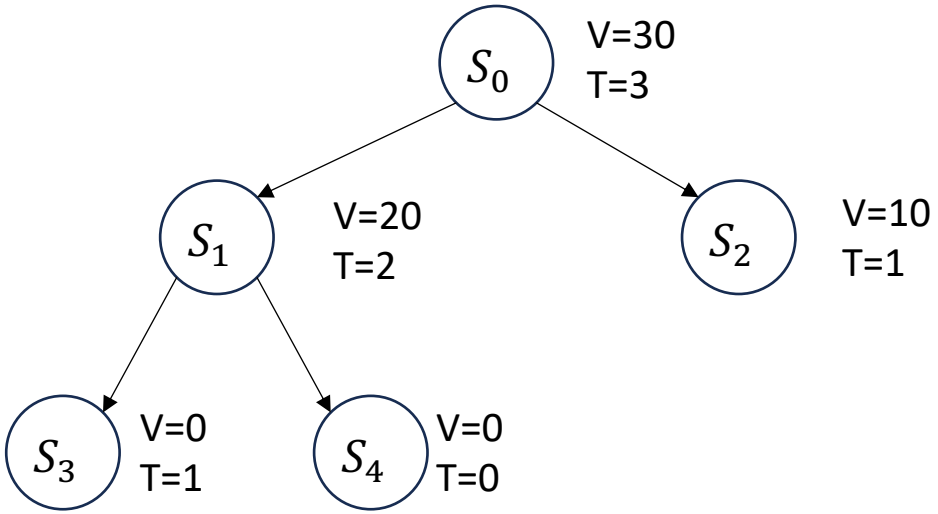
蒙特卡洛搜索

$$UCB(k) = Q_k + C * \sqrt{ln\,T / T_k}$$



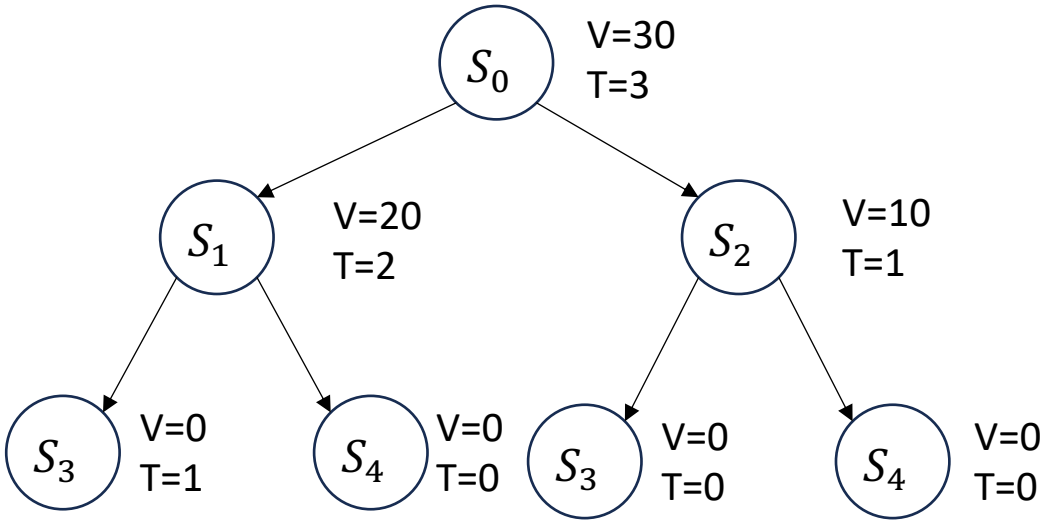
蒙特卡洛搜索

$$UCB(k) = Q_k + C * \sqrt{{ln\ T}/{T_k}}$$



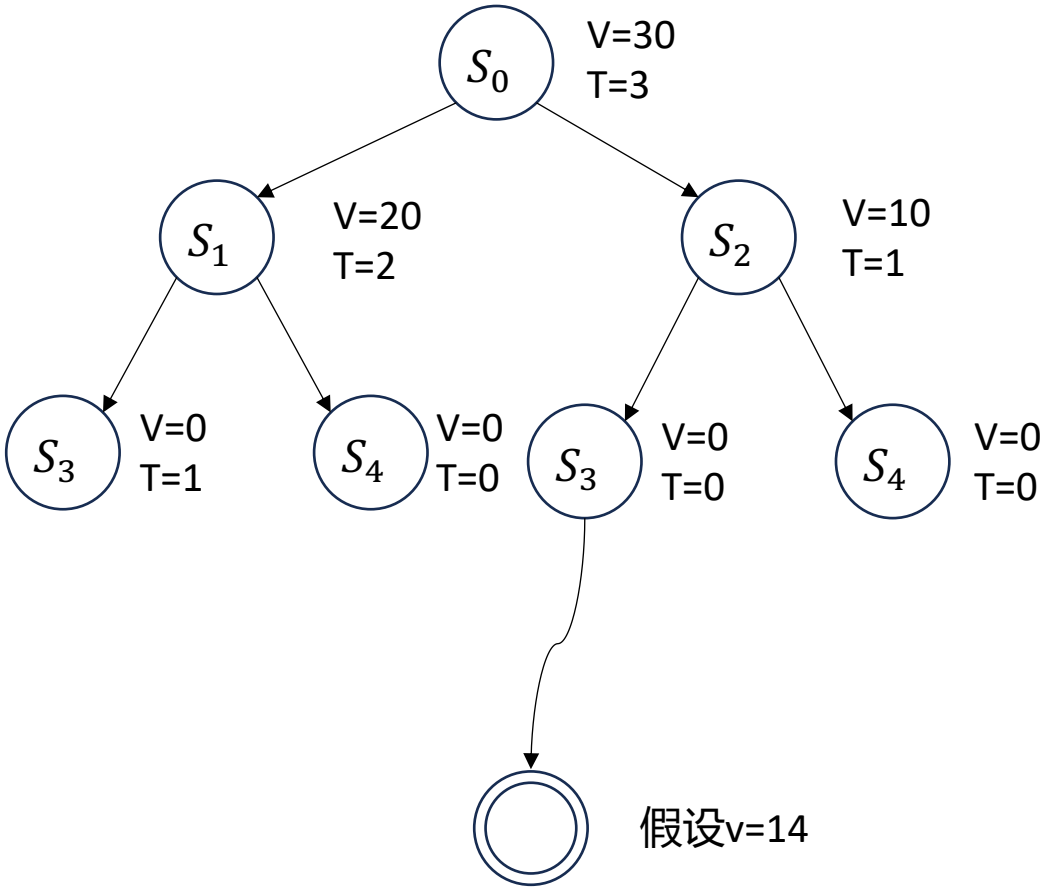
蒙特卡洛搜索

$$UCB(k) = Q_k + C * \sqrt{{ln\ T}/{T_k}}$$



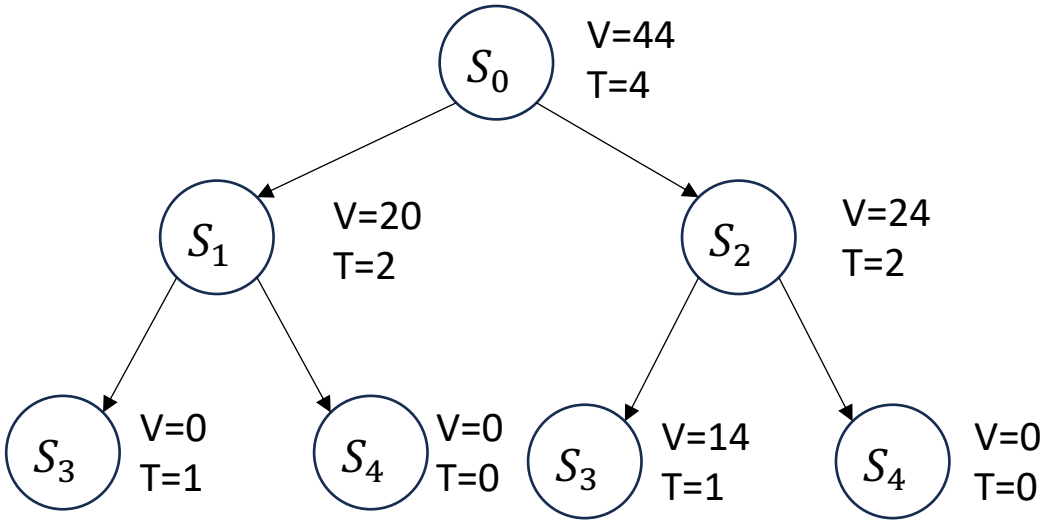
蒙特卡洛搜索

$$UCB(k) = Q_k + C * \sqrt{ln\,T / T_k}$$

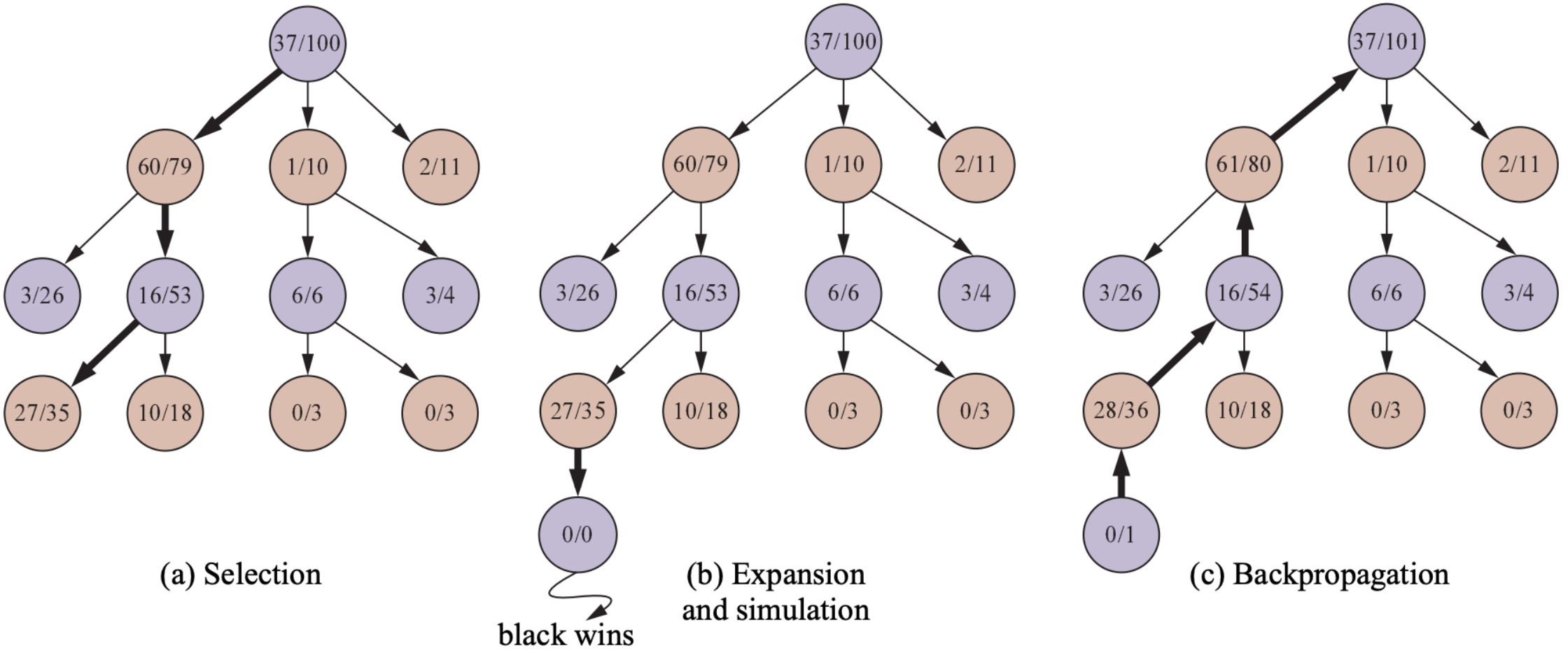


蒙特卡洛搜索

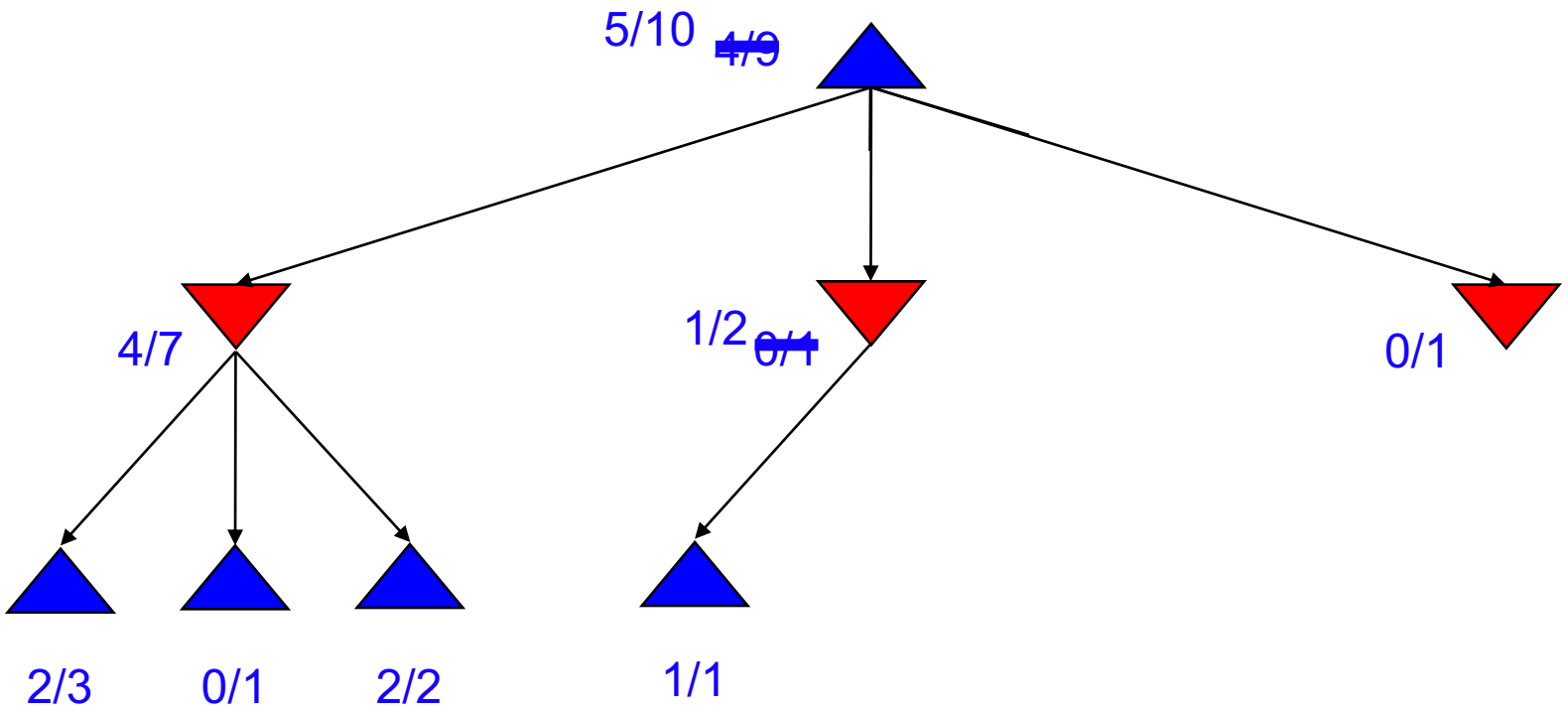
$$UCB(k) = Q_k + 2 * \sqrt{ln\,T / T_k}$$



蒙特卡洛搜索

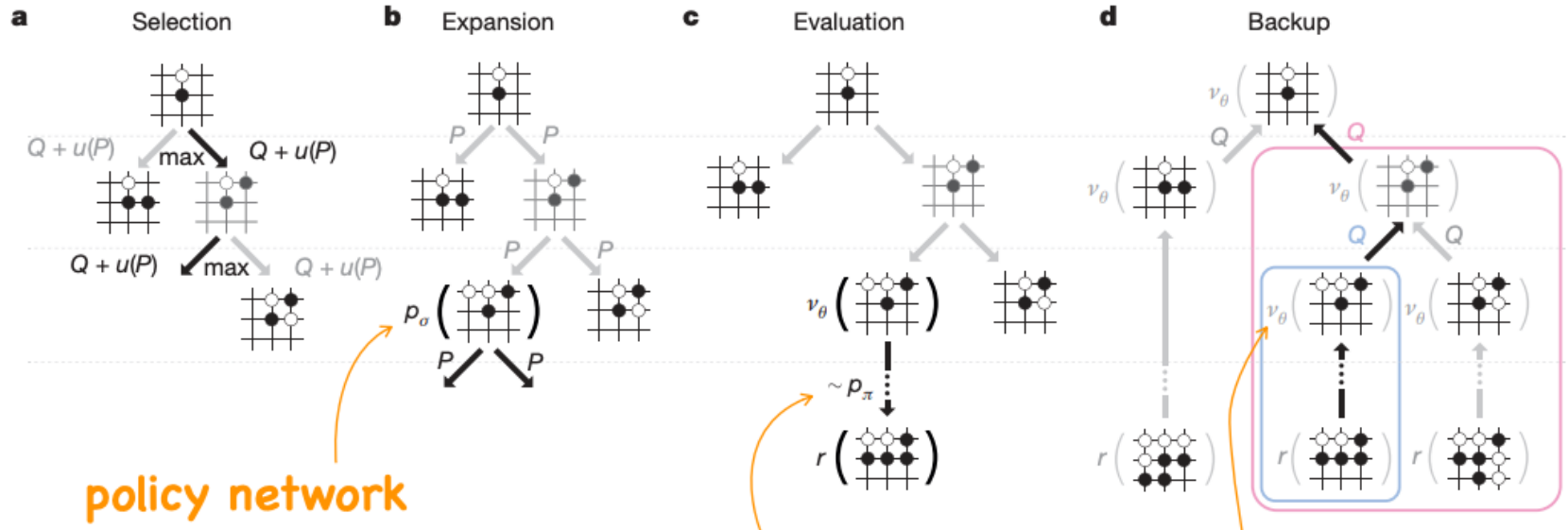

37/100
60/79
1/10
2/11
3/26
16/53
6/6
3/4
27/35
10/18
0/3
0/3
(a) Selection
37/100
60/79
1/10
2/11
3/26
16/53
6/6
3/4
27/35
10/18
0/3
0/3
0/0
black wins
(b) Expansion
and simulation
37/101
61/80
1/10
2/11
3/26
16/54
6/6
3/4
28/36
10/18
0/3
0/3
0/1
(c) Backpropagation

蒙特卡洛搜索



AlphaGO

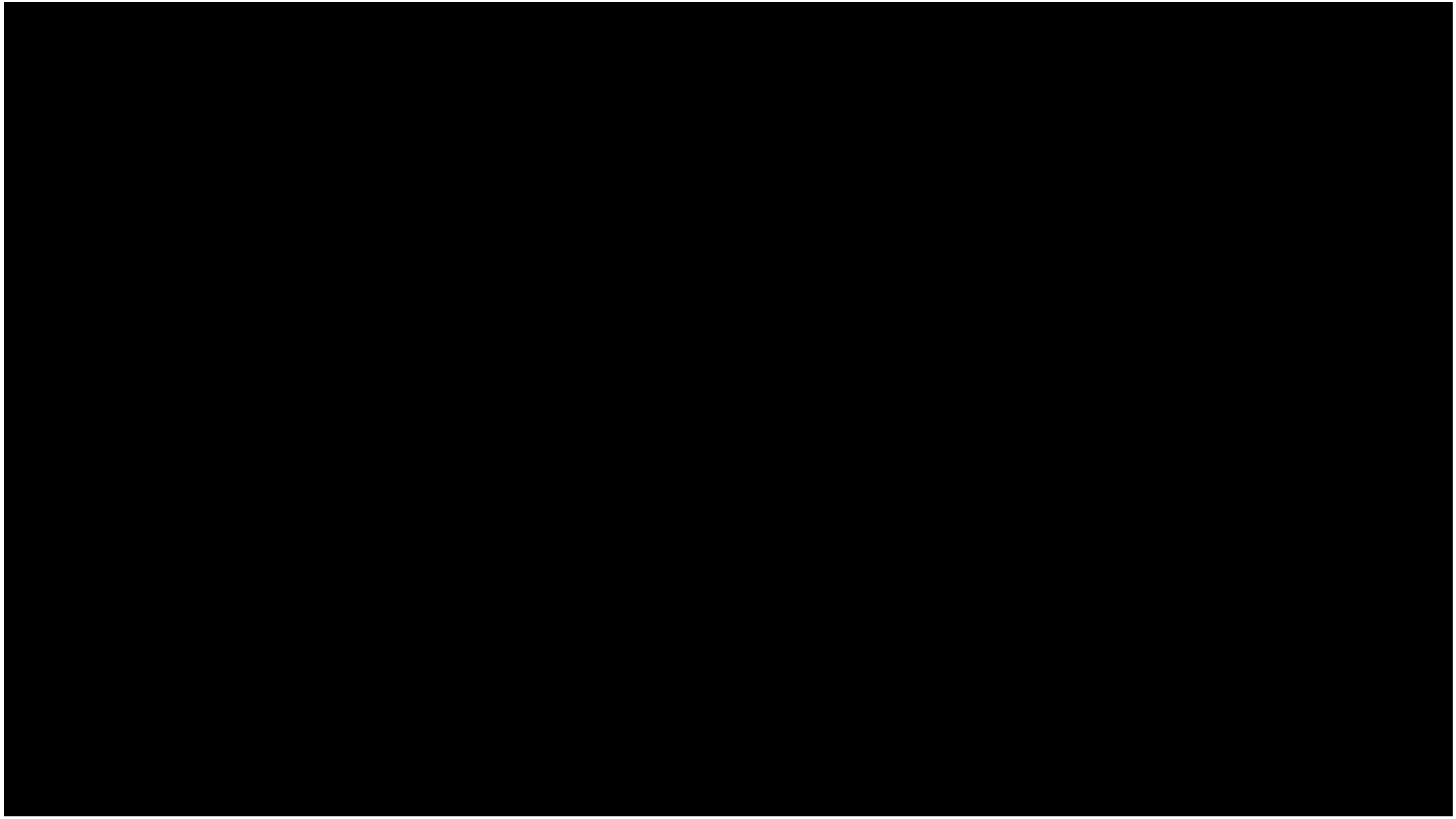
综合使用蒙特卡洛搜索、神经网络、强化学习等技术



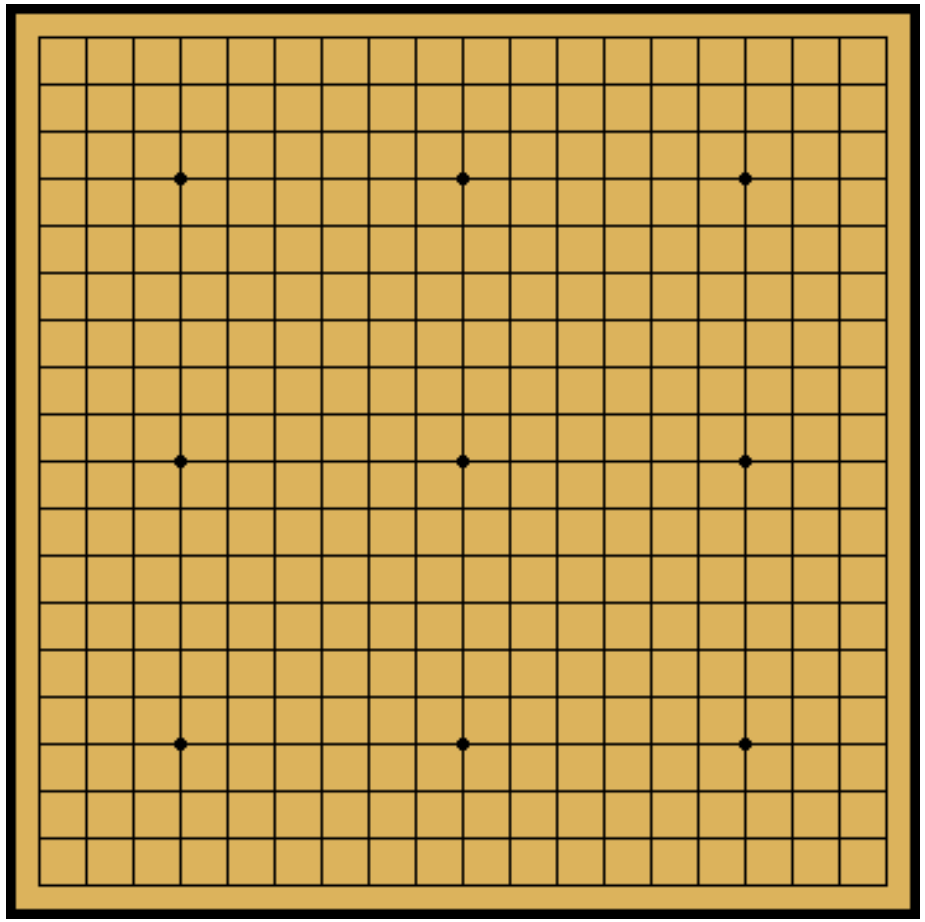
训练策略网络进行节点扩展

训练神经网络直接评估当前节点，无需模拟

AlphaGO

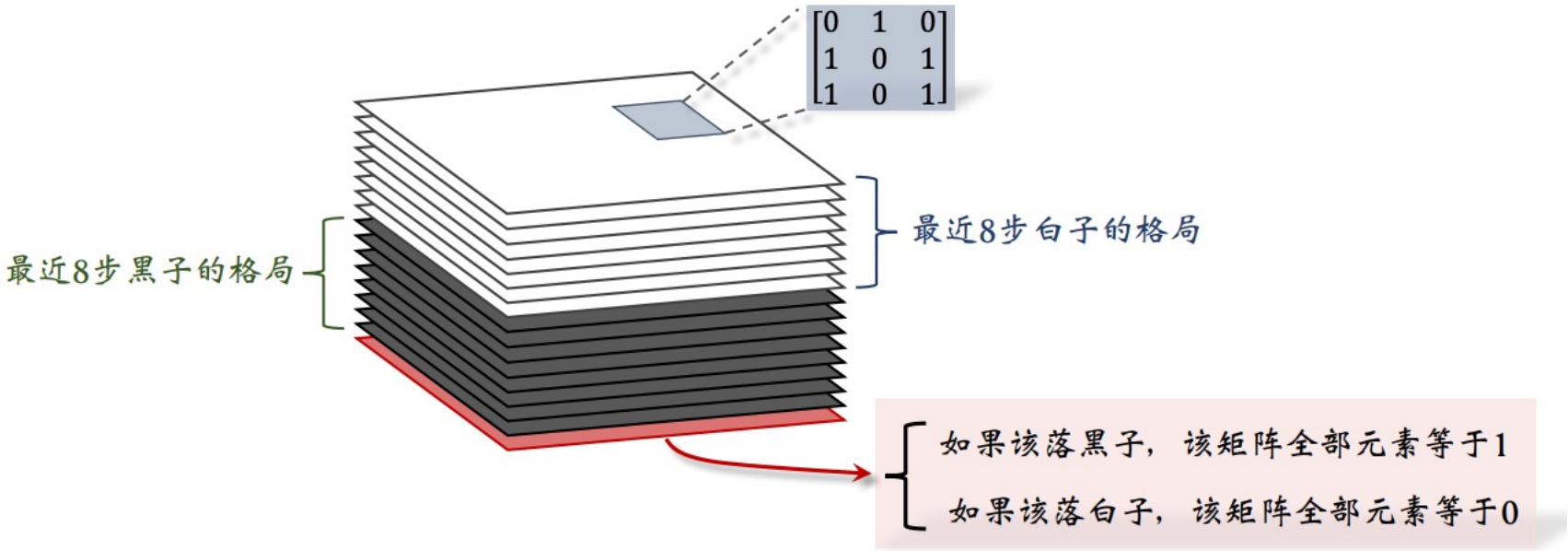


围棋



- 围棋棋盘：$19\times19=361$
- State：19×19的矩阵
- Action: $\mathcal{A} \subset \{1, 2, 3, \cdots, 361\}$

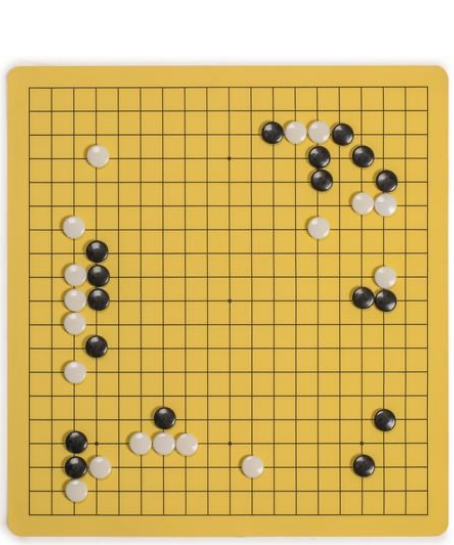
AlphaGo眼中的围棋



- AlphaZero使用的是19*19*17的张量表示一个状态
- 17：最近八步棋盘上黑子的位置，最近八步棋盘上白子的位置，以及当前该哪一方下棋（如果是黑棋，矩阵全为1，否则全为0）

策略网络(AlphaZero)

- 策略网络：$\pi(a|s;\theta)$，输入是状态，输出是每个动作的概率



state

19×19×17 tensor

Conv

feature

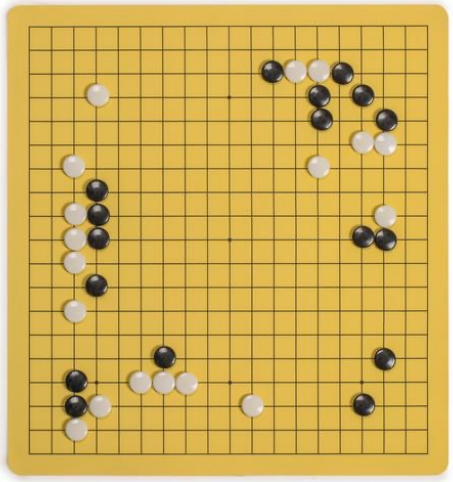
Dense

$\pi(1|s,\boldsymbol{\theta})$
$\pi(2|s,\boldsymbol{\theta})$
$\pi(3|s,\boldsymbol{\theta})$

$\pi(359|s,\boldsymbol{\theta})$
$\pi(360|s,\boldsymbol{\theta})$
$\pi(361|s,\boldsymbol{\theta})$

Probability distribution over the 361 actions

策略网络(AlphaGo)



state

19×19×48 tensor

Conv

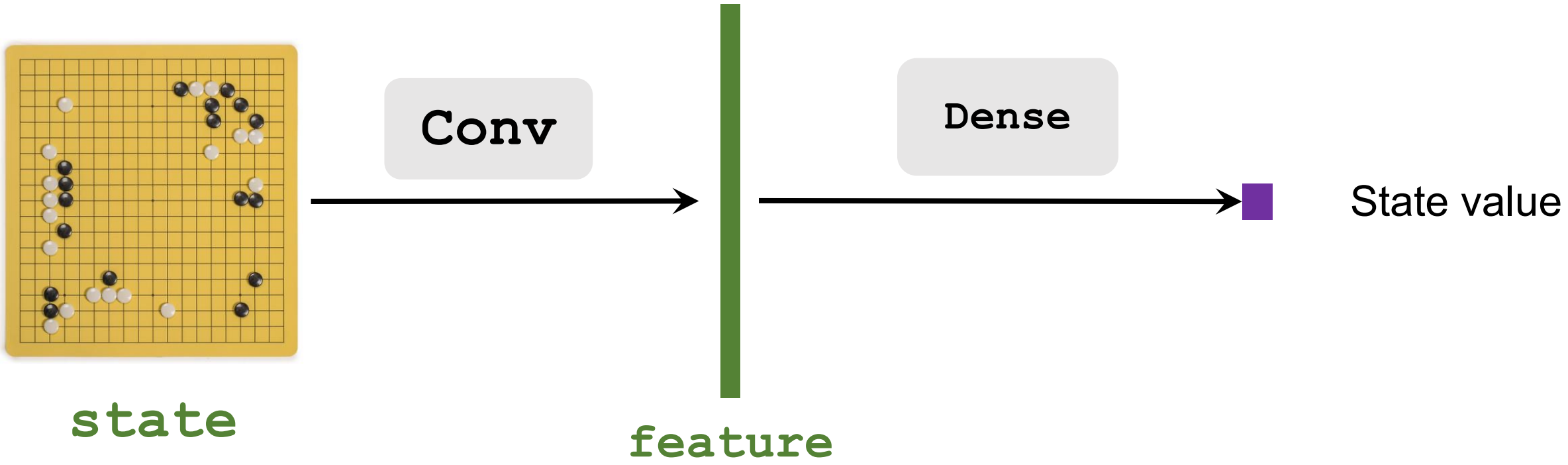
$\pi(1|s,\boldsymbol{\theta})$
$\pi(2|s,\boldsymbol{\theta})$
$\pi(3|s,\boldsymbol{\theta})$

$\pi(359|s,\boldsymbol{\theta})$
$\pi(360|s,\boldsymbol{\theta})$
$\pi(361|s,\boldsymbol{\theta})$

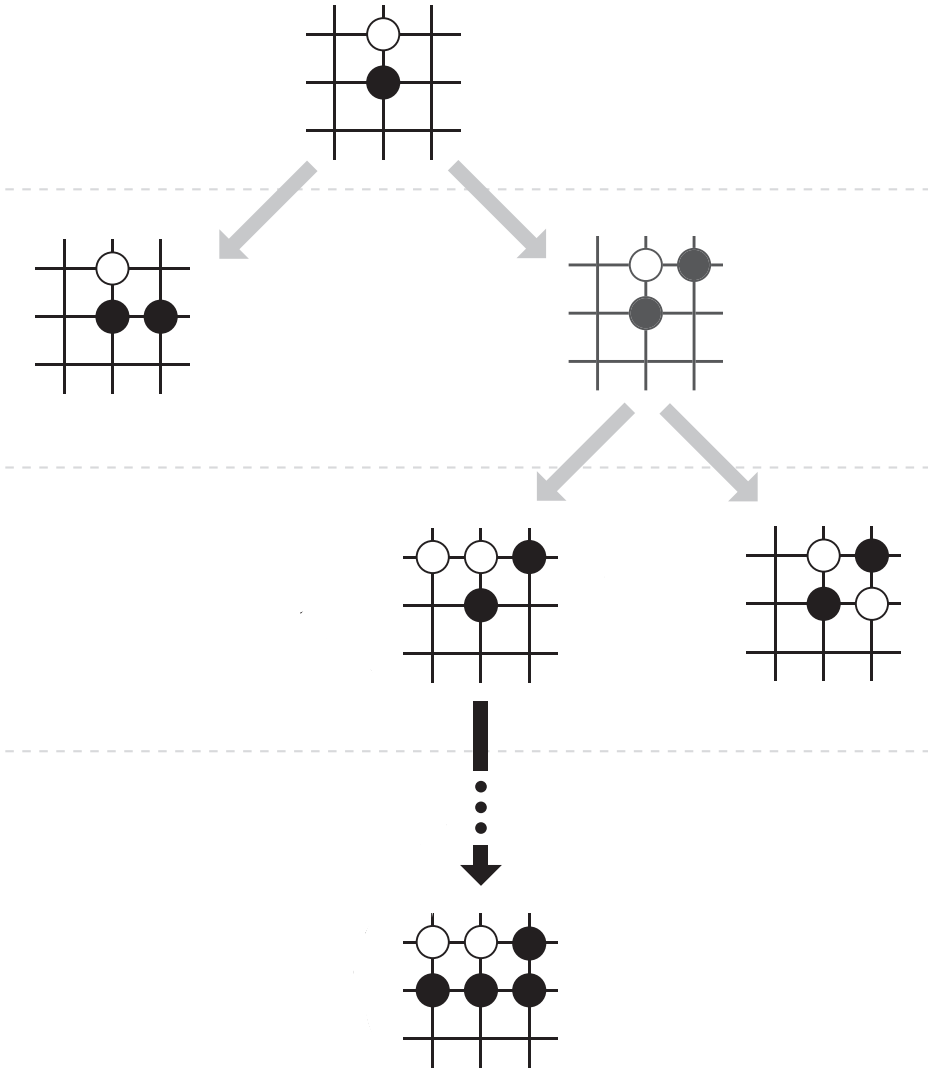
Probability distribution over the 361 actions

价值网络

- 价值网络：$v(s;w)$，对状态价值函数的估计，输入是状态，输出是一个实数，表示当前状态的好坏

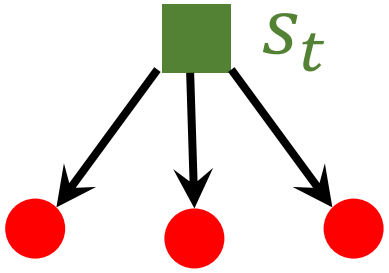


蒙特卡罗搜索



1. 选择(Selection)
2. 扩展(Expansion)
3. 评估(Evaluation)
4. 回溯(Backup)

第一步：选择

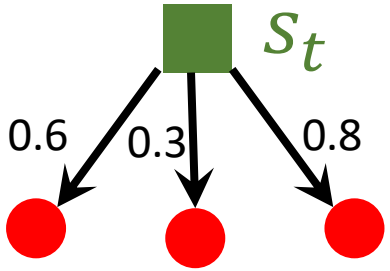


给定状态s_t, 找出胜算较大的动作，只搜索这些好的动作

$$\text{score}(a) = Q(a) + \eta \cdot \frac{\pi(a \mid s_t; \boldsymbol{\theta})}{1 + N(a)}$$

- $Q(a)$：蒙特卡洛搜索模拟带来的动作价值估计
- $\pi(a \mid s_t; \theta)$: 策略网络的输出
- $N(a)$: s_t下动作a已经被访问的次数

第一步：选择

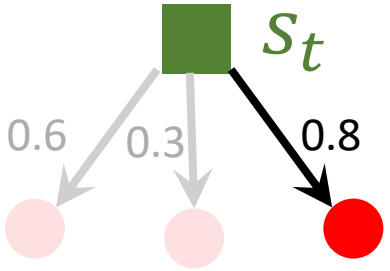


给定状态s_t, 找出胜算较大的动作，只搜索这些好的动作

$$\text{score}(a) = Q(a) + \eta \cdot \frac{\pi(a \mid s_t; \boldsymbol{\theta})}{1 + N(a)}$$

- $Q(a)$：蒙特卡洛搜索模拟带来的动作价值估计
- $\pi(a \mid s_t; \theta)$: 策略网络的输出
- $N(a)$: s_t下动作a已经被访问的次数

第一步：选择

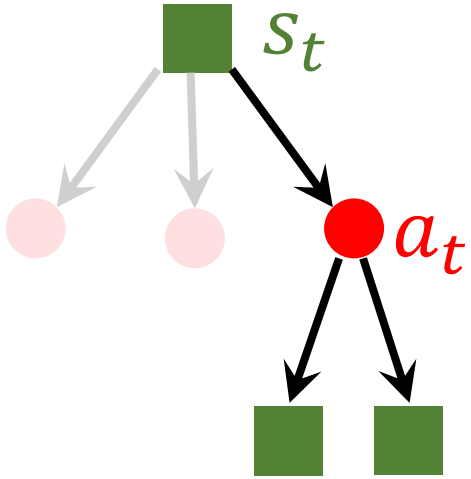


给定状态s_t，找出胜算较大的动作，只搜索这些好的动作

$$\text{score}(a) = Q(a) + \eta \cdot \frac{\pi(a \mid s_t; \boldsymbol{\theta})}{1 + N(a)}$$

- $Q(a)$：蒙特卡洛搜索模拟带来的动作价值估计
- $\pi(a \mid s_t; \theta)$：策略网络的输出
- $N(a)$：s_t下动作a已经被访问的次数

第二步：扩展

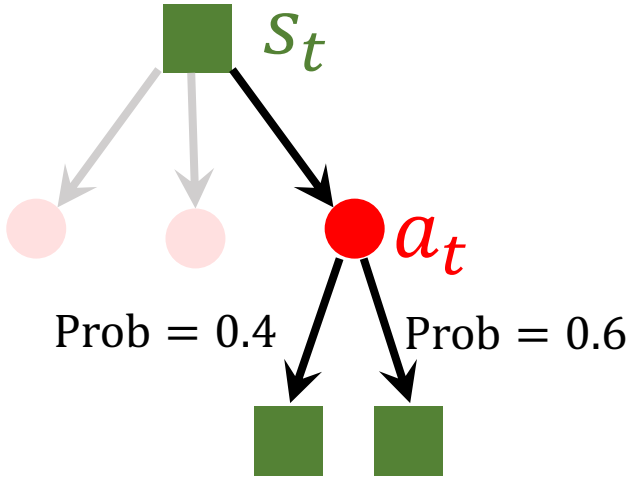


对手接下来会做什么动作?

AlphaGo用自己的策略网络去模拟对手，根据策略网络采样一个动作

$$a'_t \sim \pi(\cdot \mid s'_t; \boldsymbol{\theta})$$

第二步：扩展

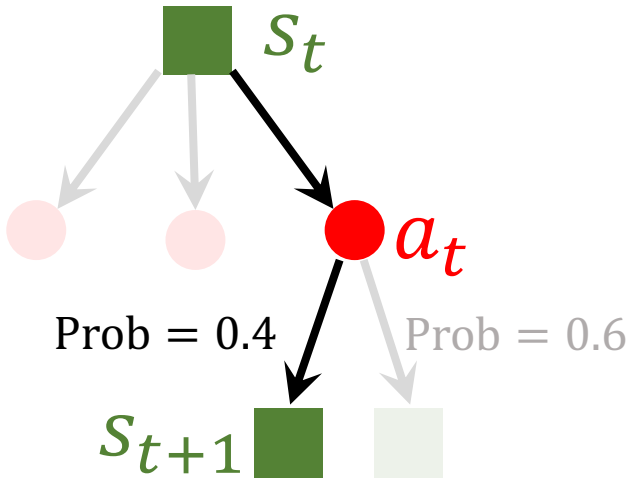


对手接下来会做什么动作?

AlphaGo用自己的策略网络去模拟对手，根据策略网络采样一个动作

$$a'_t \sim \pi(\cdot \mid s'_t; \boldsymbol{\theta})$$

第二步：扩展

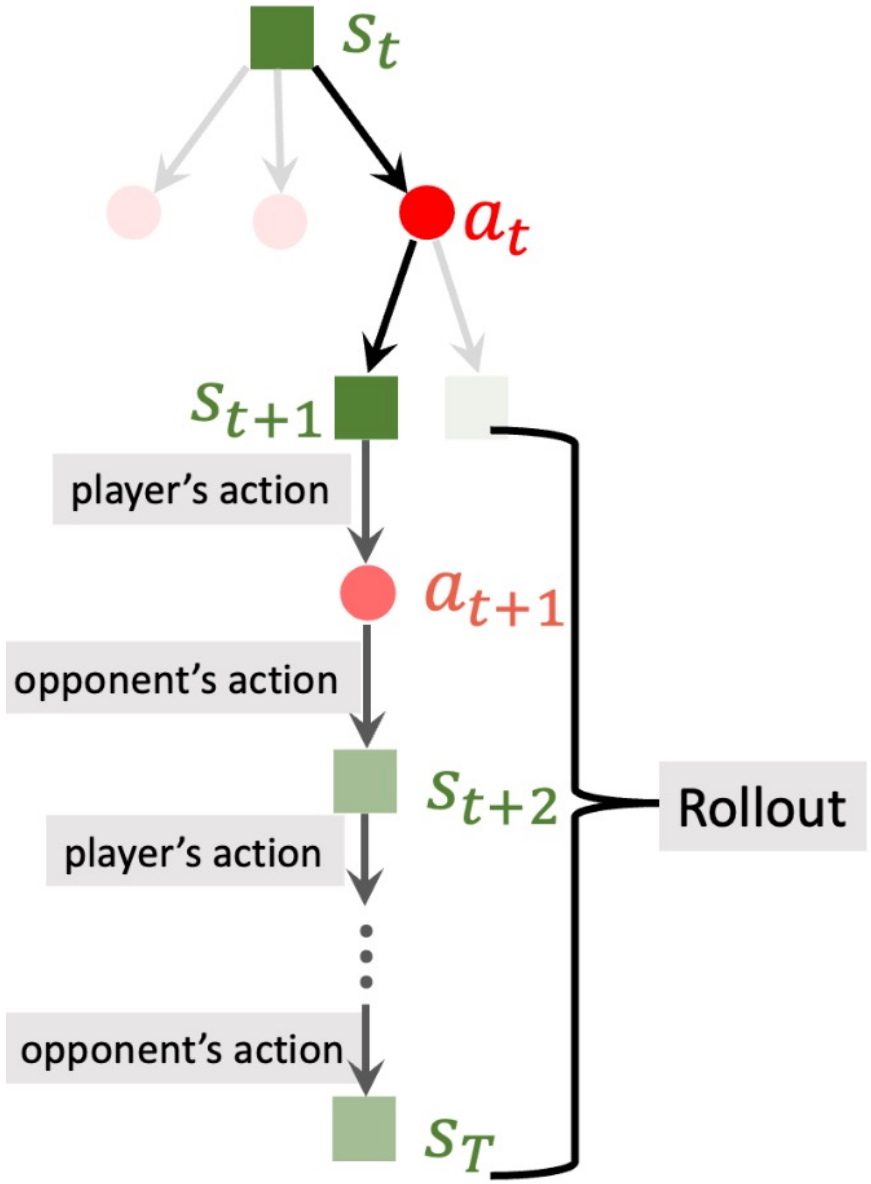


对手接下来会做什么动作?

AlphaGo用自己的策略网络去模拟对手，根据策略网络采样一个动作

$$a'_t \sim \pi(\cdot \mid s'_t; \boldsymbol{\theta})$$

第三步：模拟

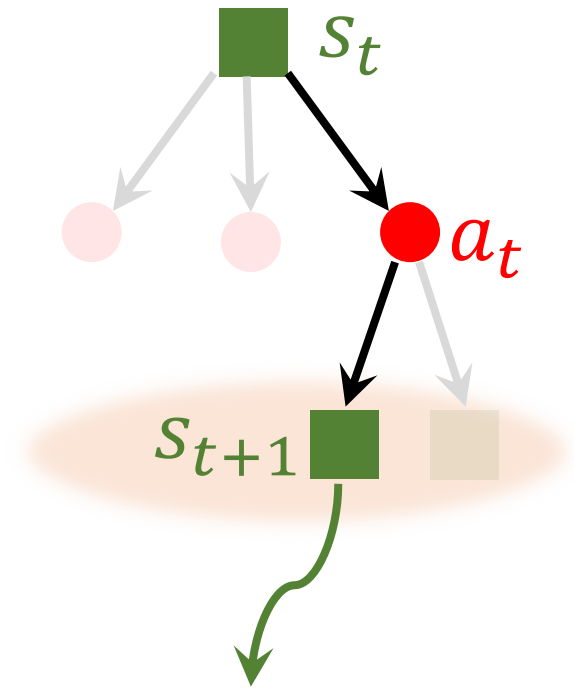


策略网络自我博弈，一直到分出胜负为止

- Player's action: $a_k \sim \pi(\cdot \mid s_k; \boldsymbol{\theta})$.
- Opponent's action: $a'_k \sim \pi(\cdot \mid s'_k; \boldsymbol{\theta})$.

- 游戏结束，获得奖赏r_T
  - Win: $r_T = +1$.
  - Lose: $r_T = -1$.

第三步：模拟



Record $V(s_{t+1})$

$$V(s_{t+1}) = \frac{1}{2} v(s_{t+1}; \mathbf{w}) + \frac{1}{2} r_T$$

策略网络自我博弈，一直到分出胜负为止

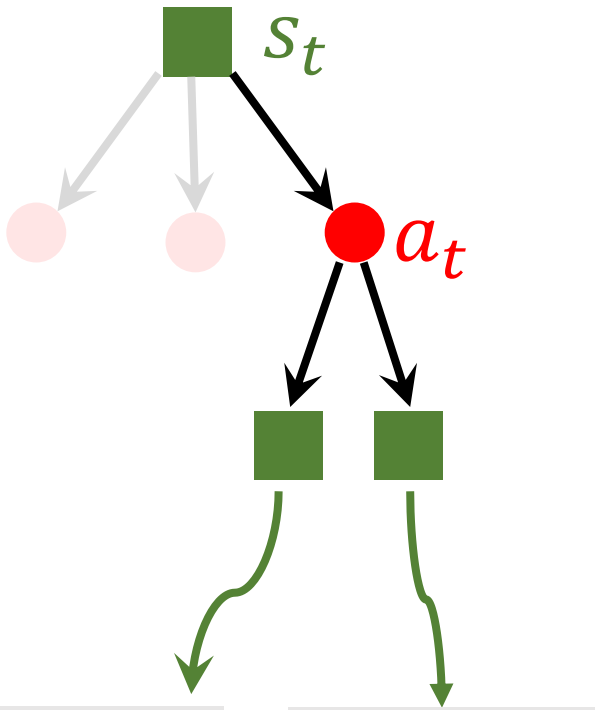
- Player's action: $a_k \sim \pi(\cdot \mid s_k; \boldsymbol{\theta})$.
- Opponent's action: $a'_k \sim \pi(\cdot \mid s'_k; \boldsymbol{\theta})$.

- 游戏结束，获得奖赏r_T
  - Win: $r_T = +1$.
  - Lose: $r_T = -1$.

- 此外，还可以通过价值网络评判s_{t+1}的好坏

- $v(s_{t+1}; \mathbf{w})$: output of the value network.

第四步：回溯

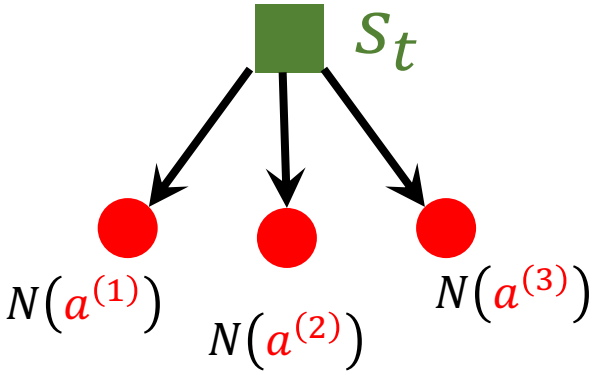


- 模拟过程会重复很多次，得到多个记录
- 把所有的记录做个平均，得到$Q(a_t)$

Records:
- $V_1^{(1)}$,
- $V_2^{(1)}$,
- $V_3^{(1)}$,
- $V_4^{(1)}$,
- ⋮

Records:
- $V_1^{(2)}$,
- $V_2^{(2)}$,
- $V_3^{(2)}$,
- $V_4^{(2)}$,
- ⋮

MCTS的决策



- $N(a)$: 记录每个动作在模拟过程中被选择的次数
- 真正的决策:

$$a_t = \underset{a}{\operatorname{argmax}}\, N(a)$$

策略网络的训练

- AlphaGo版本

1. 随机初始化策略网络，从人类棋谱中学习策略网络
2. 让两个策略网络自我博弈，改进策略网络
3. 基于已经训练好的策略网络，训练价值网络

价值网络的训练

- 让训练好的策略网络做自我博弈，记录状态回报二元组 (s_t, g_t)

- 让价值网络$v(s_t; w)$去拟合回报g_t

AlphaZero版本

- AlphaZero和AlphaGo2016版本最大的区别在于训练策略网络的方式，不再从人类棋谱学习，而是向MCTS学习
- 用MCTS控制两个玩家对弈，每走一步棋，MCTS需要做成千上万次模拟，并记录下每个动作被选中的次数$N(a)$
- 假设当前状态是s_t，执行MCTS，完成很多次模拟，得到361个整数：$N(1), \cdots, N(361)$，表示每个动作被选中的次数，归一化：

$$\boldsymbol{p}_t = \text{normalize}\left(\left[N(1),\ N(2),\ \cdots,\ N(361)\right]^T\right)$$

- 更新策略网络：让$\pi(\cdot\,|s_t, \theta)$尽量接近p_t

AlphaGo

ARTICLE

doi:10.1038/nature16961

Mastering the game of Go with deep neural networks and tree search

David Silver[1]*, Aja Huang[1]*, Chris J. Maddison[1], Arthur Guez[1], Laurent Sifre[1], George van den Driessche[1], Julian Schrittwieser[1], Ioannis Antonoglou[1], Veda Panneershelvam[1], Marc Lanctot[1], Sander Dieleman[1], Dominik Grewe[1], John Nham[2], Nal Kalchbrenner[1], Ilya Sutskever[2], Timothy Lillicrap[1], Madeleine Leach[1], Koray Kavukcuoglu[1], Thore Graepel[1] & Demis Hassabis[1]

The game of Go has long been viewed as the most challenging of classic games for artificial intelligence owing to its enormous search space and the difficulty of evaluating board positions and moves. Here we introduce a new approach to computer Go that uses 'value networks' to evaluate board positions and 'policy networks' to select moves. These deep neural networks are trained by a novel combination of supervised learning from human expert games, and reinforcement learning from games of self-play. Without any lookahead search, the neural networks play Go at the level of state-of-the-art Monte Carlo tree search programs that simulate thousands of random games of self-play. We also introduce a new search algorithm that combines Monte Carlo simulation with value and policy networks. Using this search algorithm, our program AlphaGo achieved a 99.8% winning rate against other Go programs, and defeated the human European Go champion by 5 games to 0. This is the first time that a computer program has defeated a human professional player in the full-sized game of Go, a feat previously thought to be at least a decade away.

All games of perfect information have an optimal value function, $v^*(s)$, which determines the outcome of the game, from every board position or state s, under perfect play by all players. These games may be solved by recursively computing the optimal value function in a search tree containing approximately b^d possible sequences of moves, where b is the game's breadth (number of legal moves per position) and d is its depth (game length). In large games, such as chess ($b \approx 35$, $d \approx 80$)[1] and especially Go ($b \approx 250$, $d \approx 150$)[1], exhaustive search is infeasible[2,3], but the effective search space can be reduced by two general principles. First, the depth of the search may be reduced by position evaluation: truncating the search tree at state s and replacing the subtree below s by an approximate value function $v(s) \approx v^*(s)$ that predicts the outcome from state s. This approach has led to superhuman performance in chess[4], checkers[5] and othello[6], but it was believed to be intractable in Go due to the complexity of the game[7]. Second, the breadth of the search may be reduced by sampling actions from a policy $p(a|s)$ that is a probability distribution over possible moves a in position s. For example, Monte Carlo rollouts[8] search to maximum depth without branching at all, by sampling long sequences of actions for both players from a policy p. Averaging over such rollouts can provide an effective position evaluation, achieving superhuman performance in backgammon[8] and Scrabble[9], and weak amateur level play in Go[10].

Monte Carlo tree search (MCTS)[11,12] uses Monte Carlo rollouts to estimate the value of each state in a search tree. As more simulations are executed, the search tree grows larger and the relevant values become more accurate. The policy used to select actions during

policies[13–15] or value functions[16] based on a linear combination of input features.

Recently, deep convolutional neural networks have achieved unprecedented performance in visual domains: for example, image classification[17], face recognition[18], and playing Atari games[19]. They use many layers of neurons, each arranged in overlapping tiles, to construct increasingly abstract, localized representations of an image[20]. We employ a similar architecture for the game of Go. We pass in the board position as a 19×19 image and use convolutional layers to construct a representation of the position. We use these neural networks to reduce the effective depth and breadth of the search tree: evaluating positions using a value network, and sampling actions using a policy network.

We train the neural networks using a pipeline consisting of several stages of machine learning (Fig. 1). We begin by training a supervised learning (SL) policy network p_σ directly from expert human moves. This provides fast, efficient learning updates with immediate feedback and high-quality gradients. Similar to prior work[13,15], we also train a fast policy p_π that can rapidly sample actions during rollouts. Next, we train a reinforcement learning (RL) policy network p_ρ that improves the SL policy network by optimizing the final outcome of games of self-play. This adjusts the policy towards the correct goal of winning games, rather than maximizing predictive accuracy. Finally, we train a value network v_θ that predicts the winner of games played by the RL policy network against itself. Our program AlphaGo efficiently combines the policy and value networks with MCTS.

ARTICLE

doi:10.1038/nature24270

Mastering the game of Go without human knowledge

David Silver[1]*, Julian Schrittwieser[1]*, Karen Simonyan[1]*, Ioannis Antonoglou[1], Aja Huang[1], Arthur Guez[1], Thomas Hubert[1], Lucas Baker[1], Matthew Lai[1], Adrian Bolton[1], Yutian Chen[1], Timothy Lillicrap[1], Fan Hui[1], Laurent Sifre[1], George van den Driessche[1], Thore Graepel[1] & Demis Hassabis[1]

A long-standing goal of artificial intelligence is an algorithm that learns, *tabula rasa*, superhuman proficiency in challenging domains. Recently, AlphaGo became the first program to defeat a world champion in the game of Go. The tree search in AlphaGo evaluated positions and selected moves using deep neural networks. These neural networks were trained by supervised learning from human expert moves, and by reinforcement learning from self-play. Here we introduce an algorithm based solely on reinforcement learning, without human data, guidance or domain knowledge beyond game rules. AlphaGo becomes its own teacher: a neural network is trained to predict AlphaGo's own move selections and also the winner of AlphaGo's games. This neural network improves the strength of the tree search, resulting in higher quality move selection and stronger self-play in the next iteration. Starting *tabula rasa*, our new program AlphaGo Zero achieved superhuman performance, winning 100–0 against the previously published, champion-defeating AlphaGo.

Much progress towards artificial intelligence has been made using supervised learning systems that are trained to replicate the decisions of human experts[1–4]. However, expert data sets are often expensive, unreliable or simply unavailable. Even when reliable data sets are available, they may impose a ceiling on the performance of systems trained in this manner[5]. By contrast, reinforcement learning systems are trained from their own experience, in principle allowing them to exceed human capabilities, and to operate in domains where human expertise is lacking. Recently, there has been rapid progress towards this goal, using deep neural networks trained by reinforcement learning. These systems have outperformed humans in computer games, such as Atari[6,7] and 3D virtual environments[8–10]. However, the most challenging domains in terms of human intellect—such as the game of Go, widely viewed as a grand challenge for artificial intelligence[11]—require a precise and sophisticated lookahead in vast search spaces. Fully general methods have not previously achieved human-level performance in these domains.

AlphaGo was the first program to achieve superhuman performance in Go. The published version[12], which we refer to as AlphaGo Fan, defeated the European champion Fan Hui in October 2015. AlphaGo Fan used two deep neural networks: a policy network that outputs move probabilities and a value network that outputs a position eval-

trained solely by self-play reinforcement learning, starting from random play, without any supervision or use of human data. Second, it uses only the black and white stones from the board as input features. Third, it uses a single neural network, rather than separate policy and value networks. Finally, it uses a simpler tree search that relies upon this single neural network to evaluate positions and sample moves, without performing any Monte Carlo rollouts. To achieve these results, we introduce a new reinforcement learning algorithm that incorporates lookahead search inside the training loop, resulting in rapid improvement and precise and stable learning. Further technical differences in the search algorithm, training procedure and network architecture are described in Methods.

Reinforcement learning in AlphaGo Zero

Our new method uses a deep neural network f_θ with parameters θ. This neural network takes as an input the raw board representation s of the position and its history, and outputs both move probabilities and a value, $(\boldsymbol{p}, v) = f_\theta(s)$. The vector of move probabilities $\boldsymbol{p}$ represents the probability of selecting each move a (including pass), $p_a = \Pr(a|s)$. The value v is a scalar evaluation, estimating the probability of the current player winning from position s. This neural network combines the roles of both policy network and value network[12] into a single architecture.

博弈AI的发展现状

- **跳棋：** 1990年战胜人类冠军，使用alpha-beta搜索和存有390000亿个残局的数据库表现趋于完美

- **国际象棋：** IBM的深蓝国际象棋程序，1997年击败世界冠军Garry Kasparov，每步棋搜索最多至300亿个棋局，常规搜索深度是14步，某些情况下搜索深度可以达到40层，评估函数考虑了超过8000个特征

- **围棋：** AlphaGO，采用蒙特卡洛搜索+深度强化学习，AlphaZero，无需人类棋谱数据进行训练

博弈AI的发展现状

- **星际争霸：** DeepMind 团队基于多智能体深度强化学习推出的AlphaStar在星际争霸II中达到了人类大师级的水平，并且在《星际争霸II》的官方排名中超越了99.8%的人类玩家

	GO	MOBA
Action Space	$250^{150} \approx 10^{360}$ (250 pos available, 150 decisions per game in average)	10^{1500} (10 options, 1500 actions per game)
State Space	$3^{360} \approx 10^{170}$ (361 pos, 3 states each)	10^{20000} (10 heroes, 2000+pos * 10+states)

- **DOTA2：** OpenAI推出的“OpenAI Five”击败世界冠军

- **王者荣耀：** 腾讯推出的觉悟AI，可以击败97%的玩家，并且多次击败顶尖职业团队

大模型能胜任博弈AI任务吗？

给定王者荣耀对局中的实时盘面信息，作为作为主玩家的助手，给出决策建议。# 盘面信息<game_state> </game_state> # 思考的要点：盘面理解（英雄信息、发育状态、兵线态势、防御塔状态、野区资源、局面感知、视野情况等等）阵容与策略（英雄/阵容特点、强势弱势期、个人责任、风险收益平衡）实时动态时机观察（局势顺逆僵持、双方动向意图、交战情况、资源取舍、战术选择、追击逃跑）特殊场景（顺风、逆风、关键资源抢夺）、特殊英雄机制与特殊战术# 建议选择思考给出决策意见后，再从不互斥的候选选项中选出最接近的1-2个建议。思考过程放入<think> </think>，行动建议使用","分隔放入<answer> </answer>。候选选项有<action_candidates> </action_candidates>

Game State after Rendering:

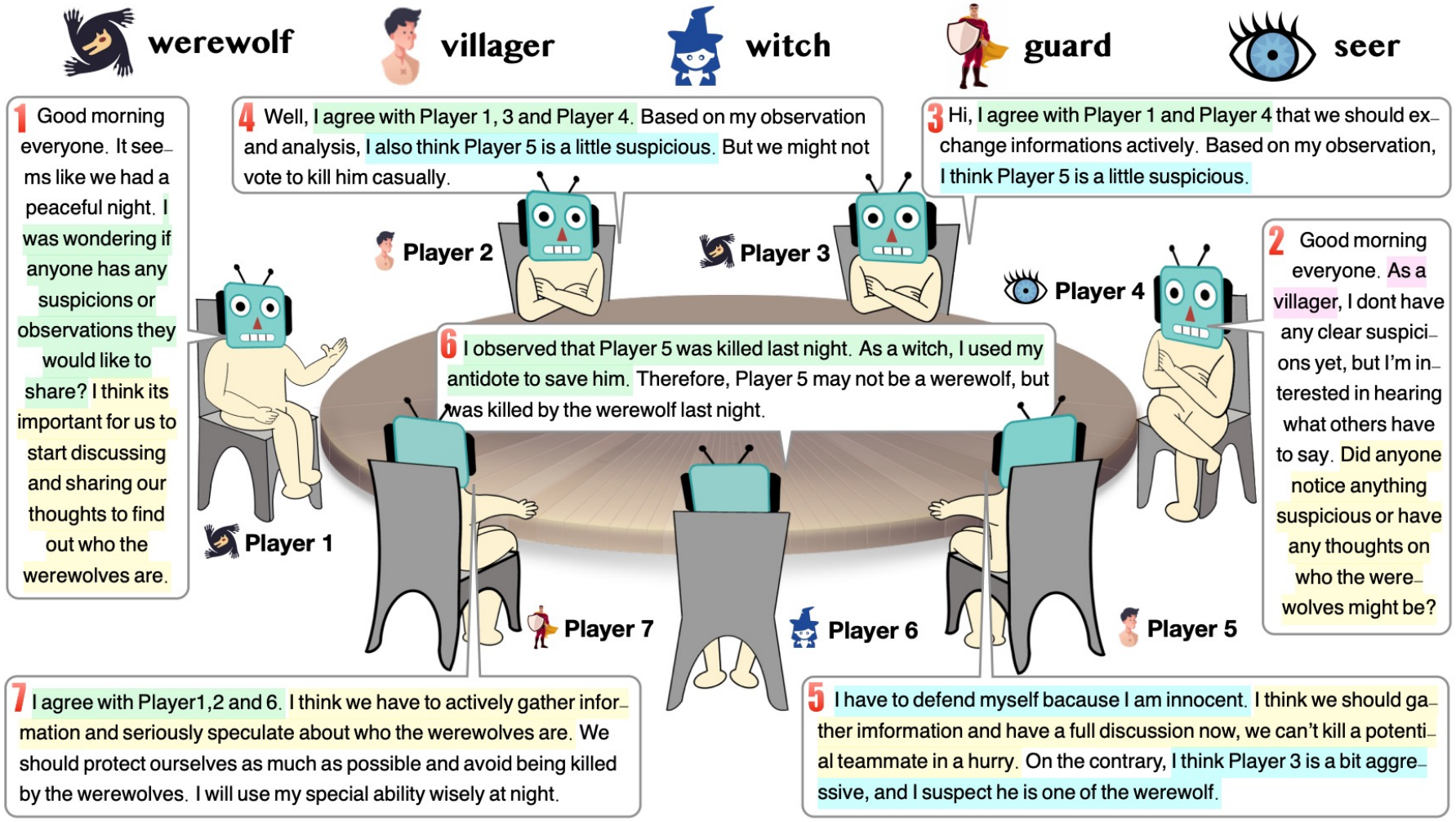
Model Response:

<think>
我需要分析当前盘面，帮助玩家扁鹊做出最佳决策。首先，观察到暗影主宰已刷新在主宰龙坑，且正受到攻击，这可能是由我方铠在处理。由于主宰对团队增益重大，优先处理它能扭转局势。其次，敌方马可波罗在我方蓝区，但距离较远，且扁鹊当前位置到主宰龙坑较近，移动方便。最后，确保没有其他紧急威胁，如敌方英雄接近或需要防守的塔，因此建议立即前往主宰龙坑协助。
</think>

<result>
协助铠拿下暗影主宰，获取团队增益。
</result>

Think in Games: Learning to Reason in Games via Reinforcement Learning with Large Language Models. https://arxiv.org/pdf/2508.21365

大模型能胜任博弈AI任务吗？



Exploring Large Language Models for Communication Games: An Empirical Study on Werewolf. https://arxiv.org/pdf/2309.04658

大模型能胜任博弈AI任务吗?

A Survey on Large Language Model Based Game Agents. https://arxiv.org/pdf/2404.02039

本章小结

- 双人零和博弈：两个Rational Agent之间的游戏
- 最大最小搜索：类似于DFS，通过递归实现
- Alpha-Beta剪枝：减去不会影响上层节点的分支
- 蒙特卡洛搜索：通过模拟判断节点的价值

搜索部分小结

- 掌握常见的无信息搜索算法，能够编程实现DFS, BFS, UCS
- 掌握常见的启发式搜索算法，能够编程实现A*搜索
- 掌握常见的局部搜索算法，能够编程实现爬山搜索
- 了解博弈搜索算法的基本思想，能够综合运用，解决现实博弈问题，如黑白棋、五子棋、象棋等

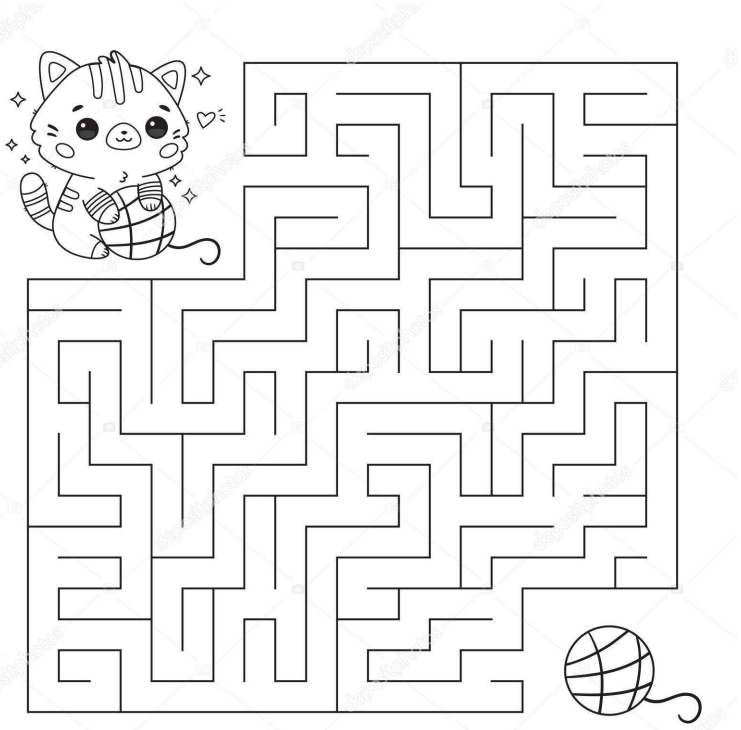
前往下一站

Search

Reinforcement Learning

确定环境

从起点到终点动作序列



非确定环境

学会一个策略，能够在任意状态下找到最佳动作