Search vs. Adversarial Search 搜索与对抗搜索

Search 搜索	Adversarial Search 对抗搜索
Single agent 单智能体	Multiple agents 多智能体
Solution is (heuristic) method for finding goal. 解是寻找目标的（启发式）方法	Solution is strategy (strategy specifies move for every possible opponent reply). 解是策略（指定对每个可能对手回应的行动策略）
Heuristics can find optimal solution. 启发式法可以找到最优解	Time limits force an approximate solution. 时间受限被迫执行一个近似解
Evaluation function: estimate of cost from start to goal through given node. 评价函数：给定节点从起始到目标的代价估计	Evaluation function: evaluate “goodness” of game position. 评价函数：评估博弈局势的“好坏”

特定环境

deterministic, perfect information, turn taking, two players, zero sum

确定、完全可观察、轮流、双人、零和（Ps. 通常是有时间约束的）

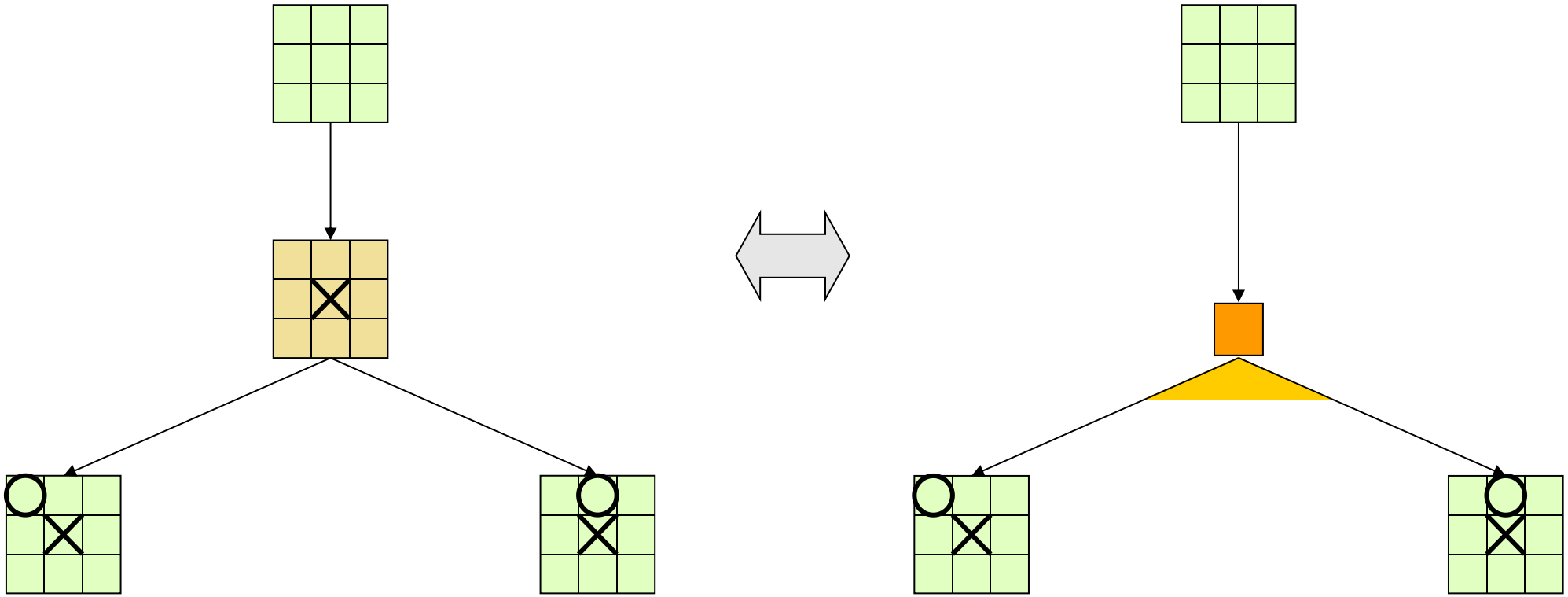
- Calling the two players:
  将两个玩家称为：**MAX，MIN.**
- **MAX** moves first, and then they take turns moving, until the game is over.
  MAX先走棋，然后轮流走棋，直到博弈结束。
- At game end 博弈结束时
  - winner: award points
    胜者：奖励点数
  - loser: give penalties.
    败者：给予处罚

Formally Defined as a Search Problem 形式化定义为搜索问题

S_0	▪ Initial state, specifies how the game is set up at the start. 初始状态，指定博弈开始时的设定。
PLAYER(*s*)	▪ Defines which playerhas the move in a state. 定义哪个玩家在某状态下动作。
ACTIONS(*s*)	▪ Returns the set of legal moves in a state. 返回某个状态下的合法动作。
RESULT(*s*, *a*)	▪ Transition model, defines the result of a move. 转换模型，定义一步动作的结果。
TERMINAL-TEST(*s*)	▪ Terminal test, *true*when the game is over and *false* otherwise. 终止检测，博弈结束时为*true*，否则为*false*。
UTILITY(*s*, *p*)	▪ Utility function, defines the value in states for a player *p*. 效用函数，定义在状态*s*、玩家为*p*的值。

相关说明

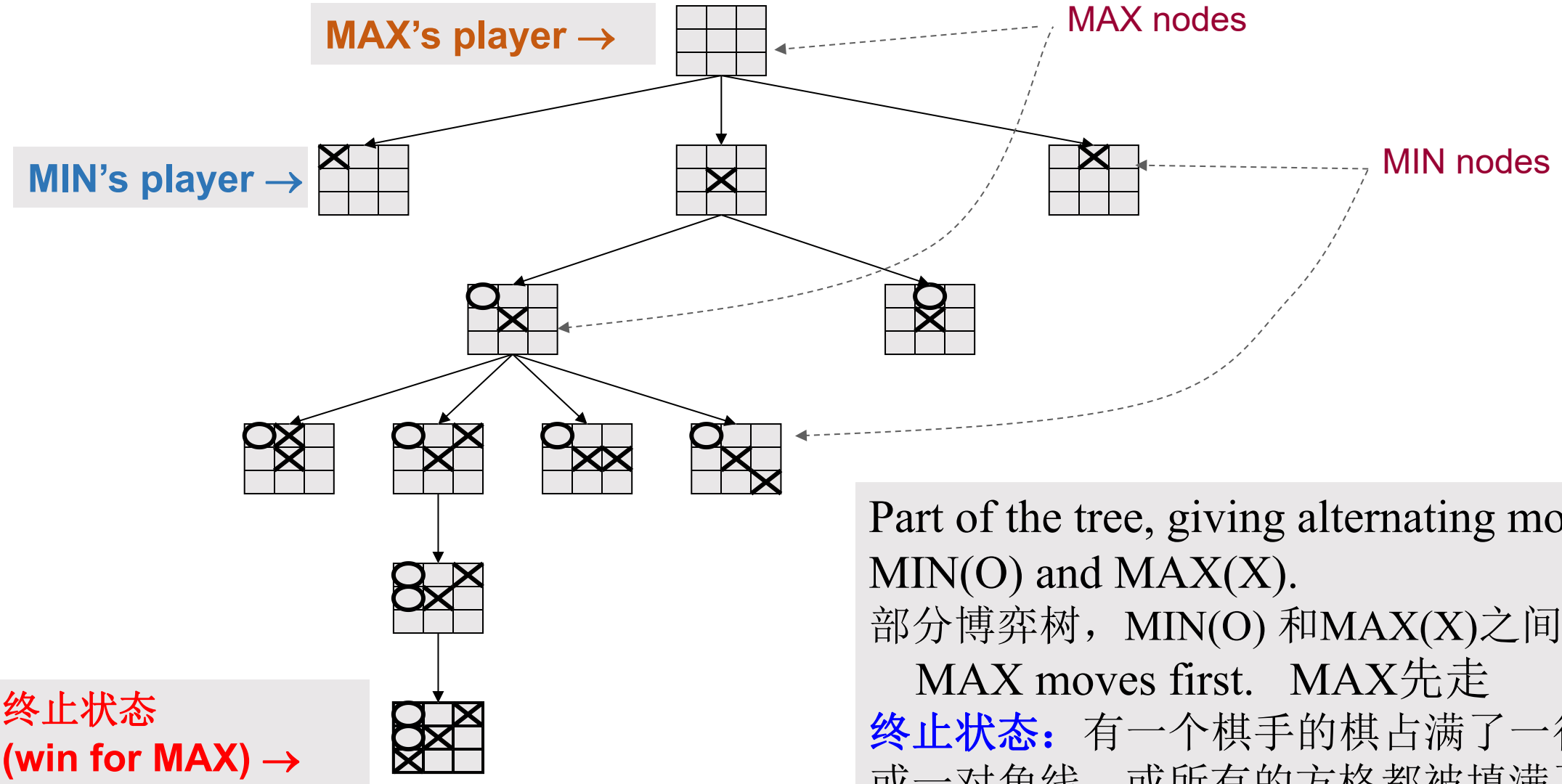
- MAX的**不确定性**是由另一个智能体MIN（对手）的行为引起的。



相关说明

- MAX的不确定性是由另一个智能体MIN（对手）的行为引起的。
- 双方均要对方失败。
- 不存在保证MAX胜利的计划，原因是MAX的后继状态由MIN的行为决定（对MIN也是同样）
- 在每次换手时，有一个特定的时间限制
- 状态空间是巨大的：在规定时间内，仅一个空间的小片段可以探究。

Example: Game Tree of Tic-tac-toe 井字棋的博弈树



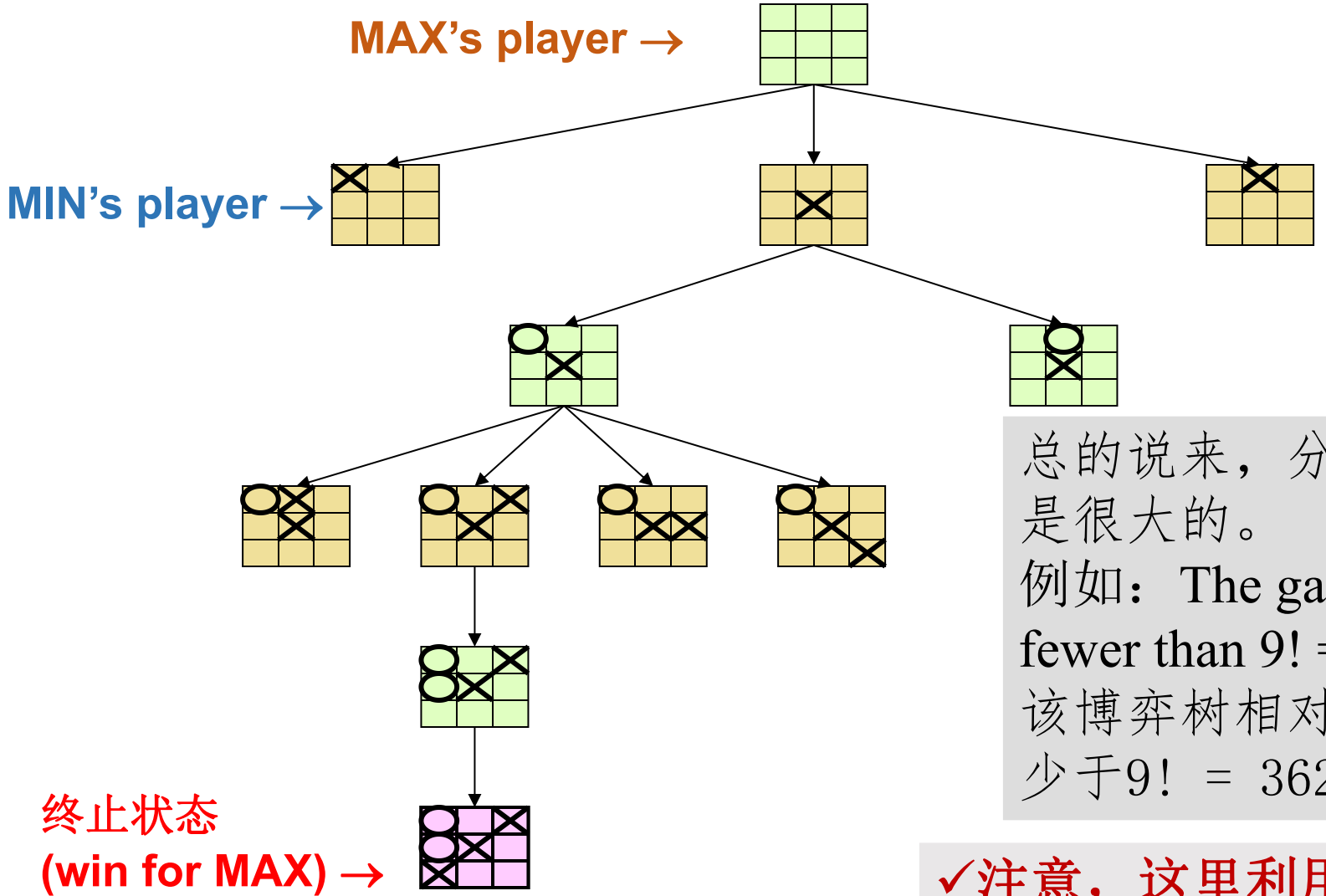
Part of the tree, giving alternating moves by MIN(O) and MAX(X).

部分博弈树，MIN(O) 和MAX(X)之间交替走子

MAX moves first. MAX先走

终止状态：有一个棋手的棋占满了一行，一列，或一对角线，或所有的方格都被填满了。

Example: Game Tree of Tic-tac-toe 井字棋的博弈树



总的说来，分支因子和终止状态的深度是很大的。

例如：The game tree is relatively small, fewer than 9! = 362,880 nodes.

该博弈树相对较小，

少于9！ = 362,880个节点

✓注意，这里利用对称性来减少分支因子。

Optimal Solution 最优解

- In normal search 普通搜索
  - The optimal solution would be a sequence of actions leading to a goal state(terminal state) that is a win.
    最优解将是导致获胜的目标状态（终端状态）的一系列动作。
- In adversarial search 对抗搜索
  - Both of MAX and MIN could have an optimal strategy.
    MAX和MIN都会有一个最优策略。
    - ✓In initial state, MAXmust find a strategy to specify MAX's move,
      在初始状态，MAX必须找到一个策略来确定MAX的动作，
    - ✓then MAX's moves in the states resulting from every possible response by MIN, and so on.
      然后MAX针对MIN的每个合理的对应采取相应的动作，以此类推。

Minimax Theorem 最小最大定理

For every two-player, zero-sum game with finitely many strategies, there exists a value V and a mixed strategy for each player, such that

对于两个玩家、具有有限多个策略的零和博弈，每个玩家存在一个值V和一个混合策略，使得：

(a) Given player 2's strategy, the best payoff possible for player 1 is V,

给定玩家2的策略，则玩家1可能的最好收益是V，

(b) Given player 1's strategy, the best payoff possible for player 2 is –V.

给定玩家1的策略，则玩家2可能的最好收益是-V。

- For a zero sum game, the name **minimax** arises because each player *minimizes the maximum payoff* possible for the other, he also *minimizes his own maximum loss*.

对于零和博弈来说，其名称**minimax**的由来是因为每个玩家会使对手可能的**最大收益变得最小**，还会使自己的**最大损失变得最小**。

Optimal Solution in Adversarial Search 对抗搜索的最优解

- Given a game tree, the optimal strategy can be determined from the minimax valueof each node, write as MINIMAX(*n*).

  给定一棵博弈树，则最优策略可以由每个节点的minimax值来确定，

  记作 **MINIMAX(n)**。

- Assume that both players play optimally from there to the end of the game.

  假设两个玩家博弈自始至终都发挥得很好。

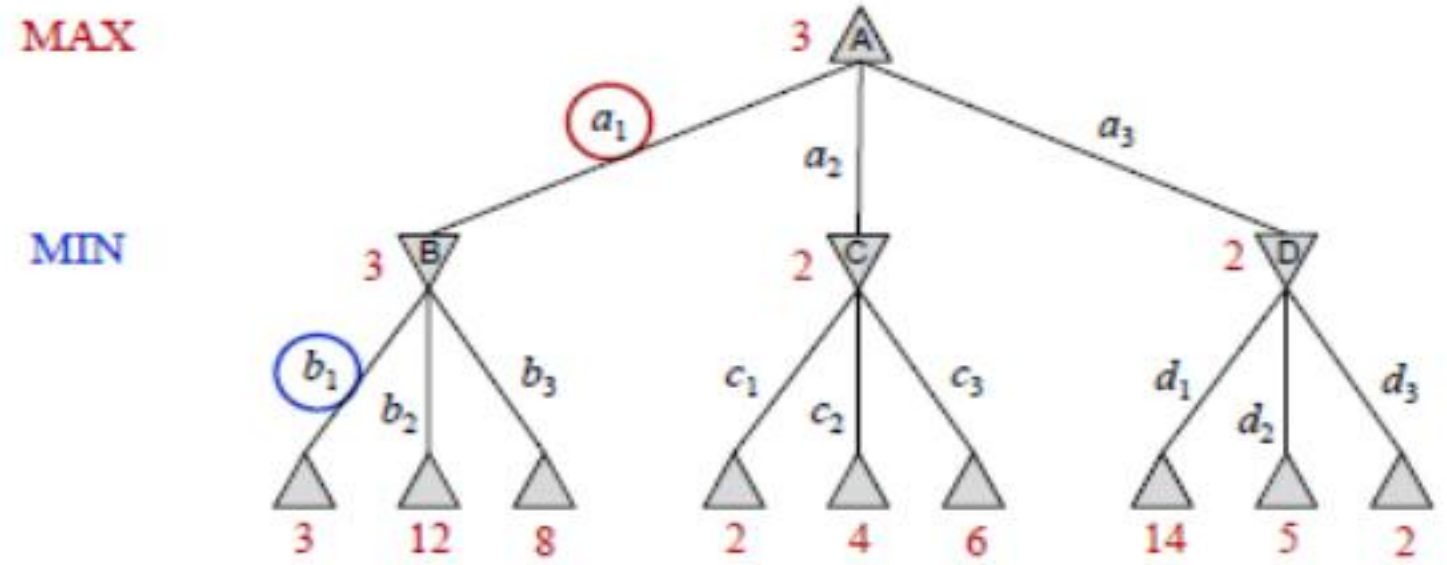
function MINIMAX(s) **returns** an action
if TERMINAL-TEST(s) **then return** **UTILITY(s)**
If PLAYER(s) = MAX **then return** $\max_{a \in \text{ACTIONS}(s)}$ MINIMAX(RESULT(s, a))
if PLAYER(s) = MIN **then return** $\min_{a \in \text{ACTIONS}(s)}$ MINIMAX(RESULT(s, a))

The minimax value of a terminal state is just its utility.

MAX prefers to move to a state of maximum value, MIN prefers a state of minimum value.

终端状态的**minimax**值只是其效用。**MAX**倾向于移动到一个最大值状态，**MIN**则倾向于一个最小值状态。

Minimax Decision--A Two-player Game Tree 一个双人玩家的博弈树



MAX's best move at root is a_1(with the highest minimax value)

根节点处MAX的最佳移动是a_1（具有最高的minimax值）

MIN's best reply at B is b_1(with the lowest minimax value)

B节点处MIN的最佳应对是b_1（具有最低的minimax值）

博弈中的优化决策：评估函数

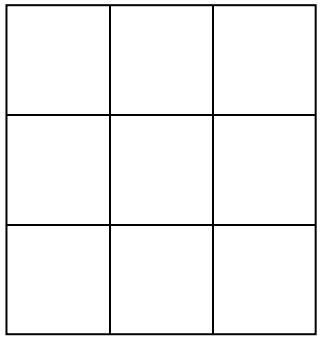
- 用函数e(s)代表状态s下的值；
- e(s) 是MAX状态的启发值，是对MAX有利状态的估计；
- e(s)>0 代表对MAX有利，值越大越好；
- e(s)<0 代表对MAX不利；
- e(s)=0 代表对MAX、MIN势均力敌。

例子： Tic-tac-Toe 井字棋

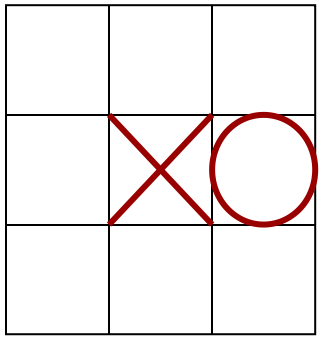
e(s) = 在所有空位放上MAX后成3子一线的个数

减去-

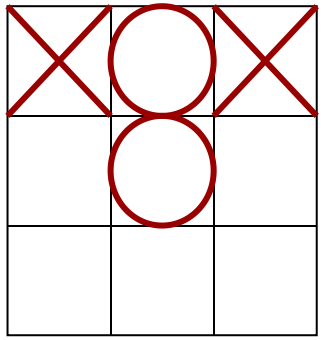
在所有空位放上MIN后成3子一线的个数



8–8 = 0



6–4 = 2



3–3 = 0

评估函数的组成

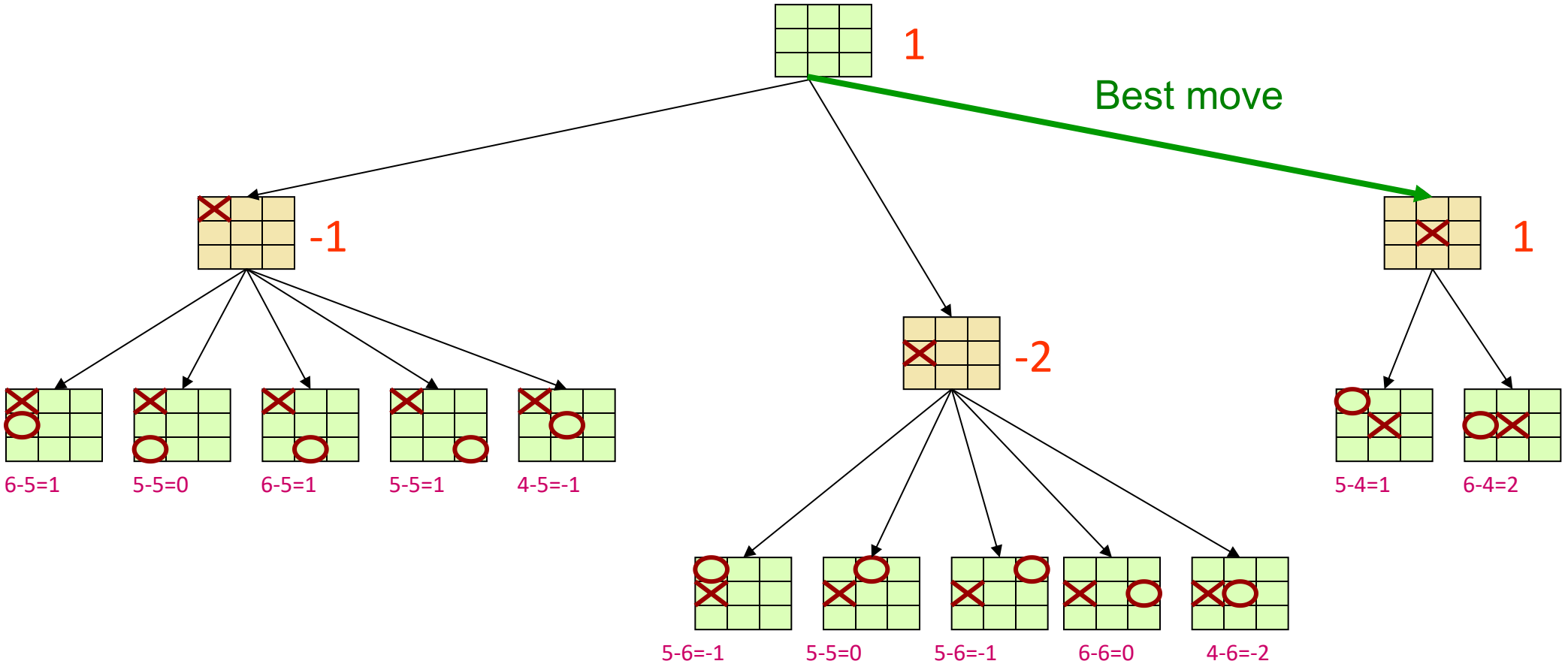
- 各“特征值”的总和：

$$e(s)=\sum_{i=1}^{n} w_i f_i(s)$$

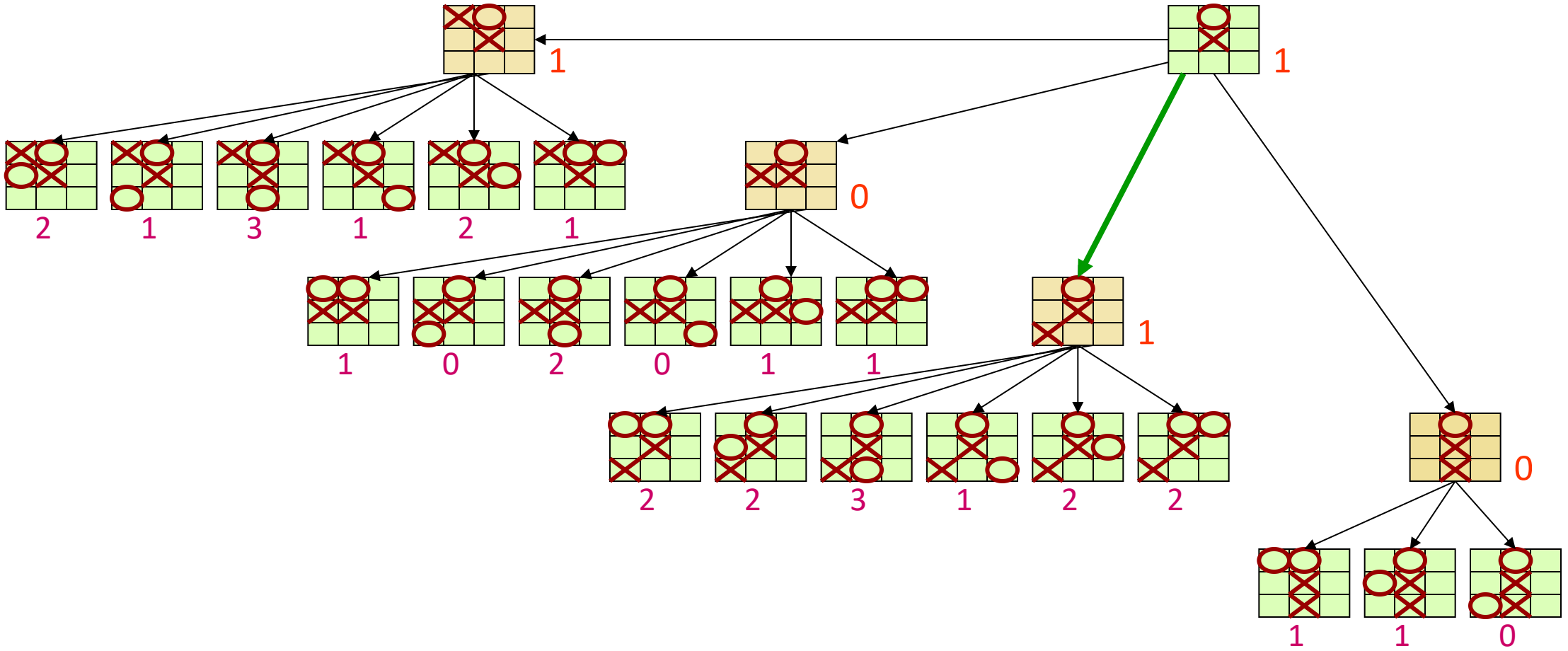
- 特征值包括：
  - 每种势态的数值
  - 可能移动的数值
  - 势态的控制数值

评估值：回退计算

井字棋二层（horizon = 2）的搜索树



下一步：



为什么采用回退值？

1.对每一个非叶子结点N的评估值是由它的子结点倒推求得，是MAX能搜索到的深度h回推得到**最安全状态**下最佳走步（假设MIN是足够好的棋手）。

2.如果从一开始e就是可信赖的，那么实际状态的值是要好于评估函数的值的。

极大极小算法思想 Minimax Algorithm

- 轮到**MAX**走时，从当前状态扩展博弈树到所设置深度h（根据允许时间要求所能的深度）；
- 计算每个叶子节点的评估值；
- 根据不同的叶子节点采用不同的方法**倒推**计算各节点值：
  - 对于MAX节点选取其后继节点中最大的值为其评估值；
  - 对于MIN节点选取其后继节点中最小的值为其评估值。
- 选择移动到具有最大倒推值的**MIN**节点。

Game Playing (for MAX)

重复直到达到终止条件（胜、负、平）：

1. 用MINIMAX算法选择移动；
2. 执行移动；
3. 观察MIN的移动。

注意：每次循环产生的深度为h的大博弈树仅为一次移动而进行；所有的下次循环将被再次重复。

（其实一个深度h减2的子树可以被再次利用）。

Minimax Algorithm 最小最大算法

```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game
  v←MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← -∞
  for a,s in SUCCESSORS(state) do
    v ← MAX(v,MIN-VALUE(s) )
  return v
```

```
function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a,s in SUCCESSORS(state) do
    v ← MIN(v,MAX-VALUE(s) )
  return v
```

Properties of Minimax Decision 最小最大决策的性质

☐The minimax algorithm performs a depth-firstexploration of the game tree.

最小最大算法表现为博弈树的深度优先探索。

- Time complexity 时间复杂性 $O(b^m)$
- Space complexity 空间复杂性

- ***O*(*bm*)** --The algorithm generates all actions at once 算法同时生成所有动作
- ***O*(*m*)** --The algorithm generates actions one at a time 算法一次生成一个动作

➢ Where *b*--The branching factor (legal moves at each point)
分支因子（每个点的合法走子）

m--The maximum depth of any node
任一节点的最大深度

minimax算法

优点:

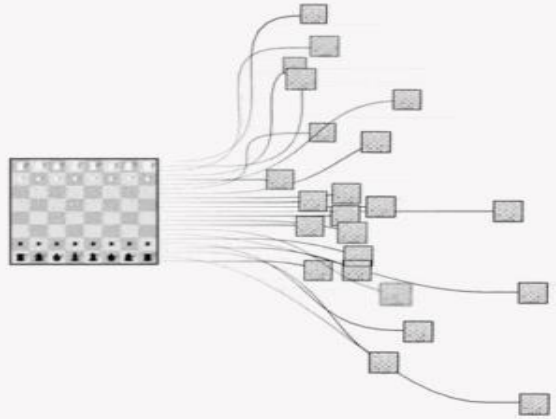
- 算法是一种简单有效的对抗搜索手段
- 在对手也“尽力而为”前提下，算法可返回最优结果

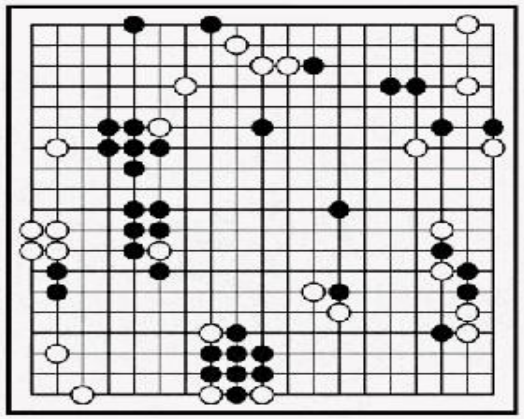
缺点:

- 如果搜索树极大，则无法在有效时间内返回结果

改善:

- 使用alpha-beta pruning算法来减少搜索节点
- 对节点进行采样、而非逐一搜索 (i.e., MCTS)



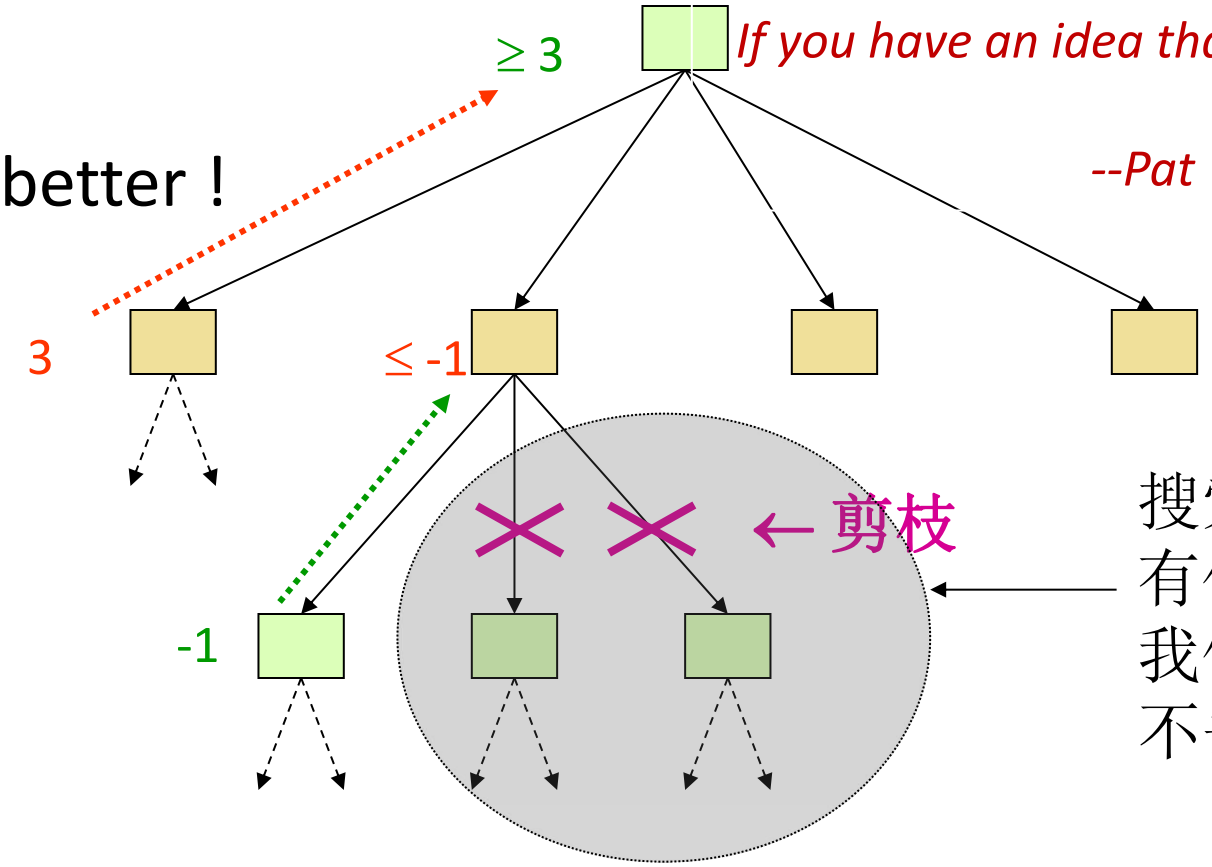


枚举当前局面之后每一种下法，然后计算每个后续局面的赢棋概率，选择概率最高的后续局面。

我们可以做的更好么？

Number of game states is exponential in depth of the tree.
博弈状态的量随着树的深度呈现指数式增长。

Yes ! Much better !



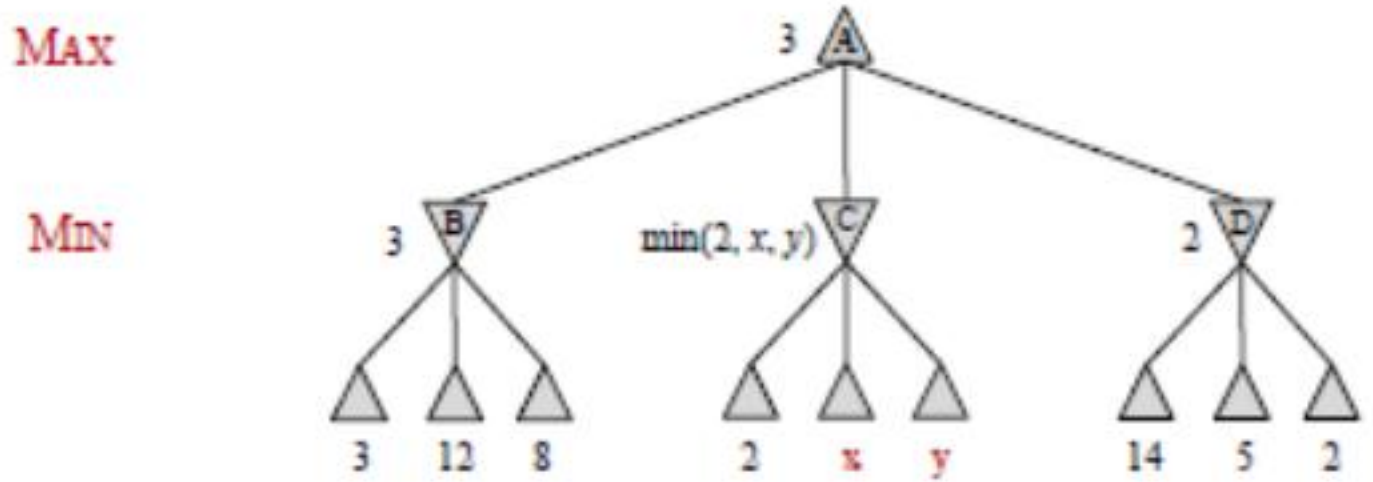
If you have an idea that is surely bad, don't take the time to see how truly awful it is.
--Pat Winston (Director, MIT AI Lab,1972-1997)

搜索树的这一部分对倒推求值没有任何影响。
我们把这种对不需要生成的节点不去生成的方法称为**剪枝**

Overview 概述

- The trick to solve the problem:
  解决该问题的技巧
- Compute correct minimax decision without looking at every node in game tree.
  计算正确的minimax决策而不考虑博弈树的每个节点。
- That is, use“pruning” to eliminate large parts of the tree.
  就是说，采用“剪枝”方法来消除该树的大部分。
- What is alpha–beta pruning
  什么是alpha–beta剪枝
- It is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm.
  是一种搜索算法，旨在削减由minimax算法评价的节点数量。

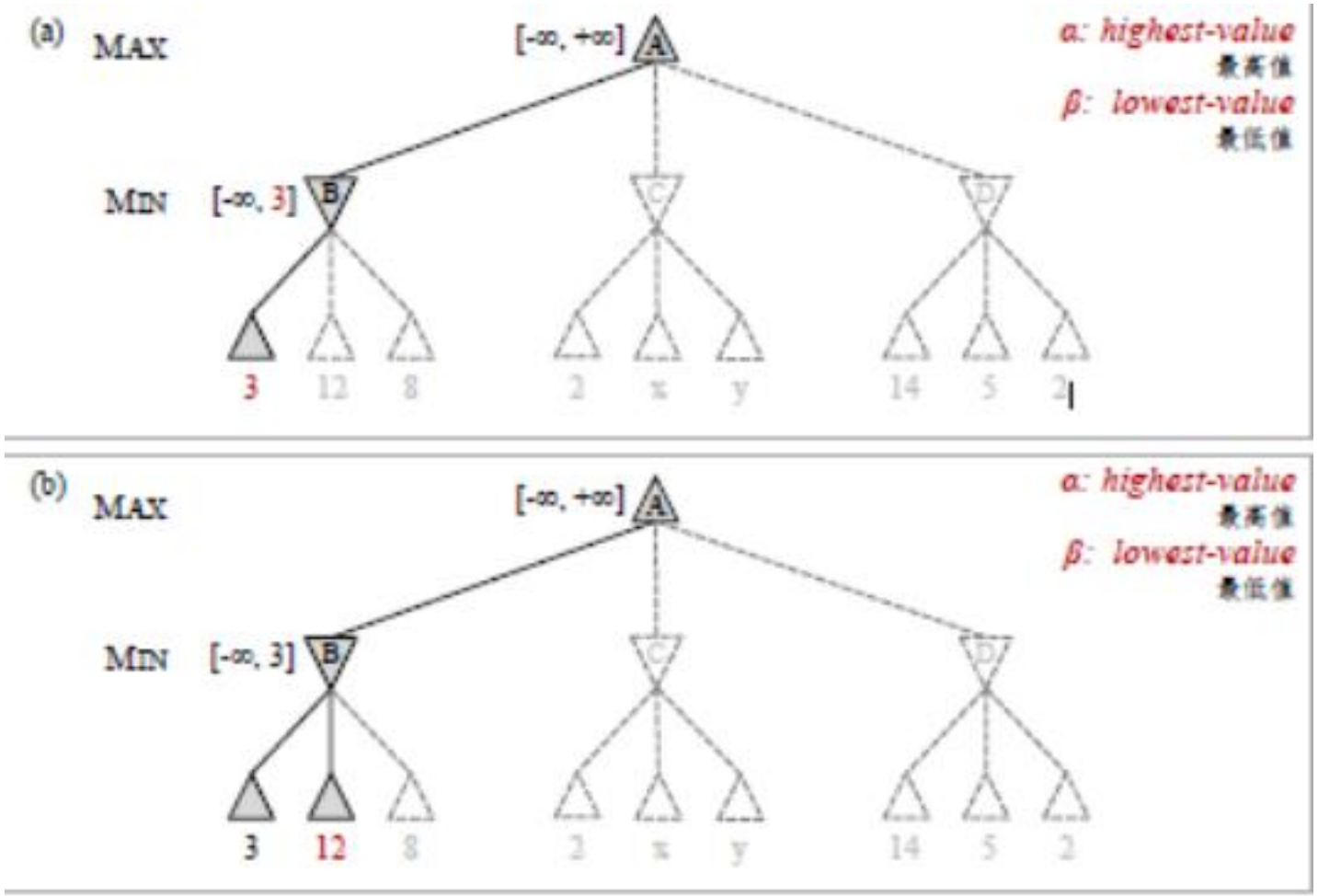
Example: Game Tree Using Minimax 采用Minimax的博弈树



- The value of root node is given by: 根节点的值由如下方法得出：

$$\begin{aligned}\text{MINIMAX}(root) &= \max(\min(3, 12, 8), \min(2, x, y), \min(14, 5, 2)) \\ &= \max(3, \min(2, x, y), 2) \\ &= \max(3, z, 2) \text{ where } z = \min(2, x, y) \le 2 \\ &= 3.\end{aligned}$$

Example: Game Tree Using Alpha-Beta Pruning
采用Alpha-Beta剪枝的博弈树



Initial value: 初始值：

$A[\alpha=-\infty, \beta=+\infty]$

(a) The 1st leaf below B has the value 3. Hence, B, as a MIN node,

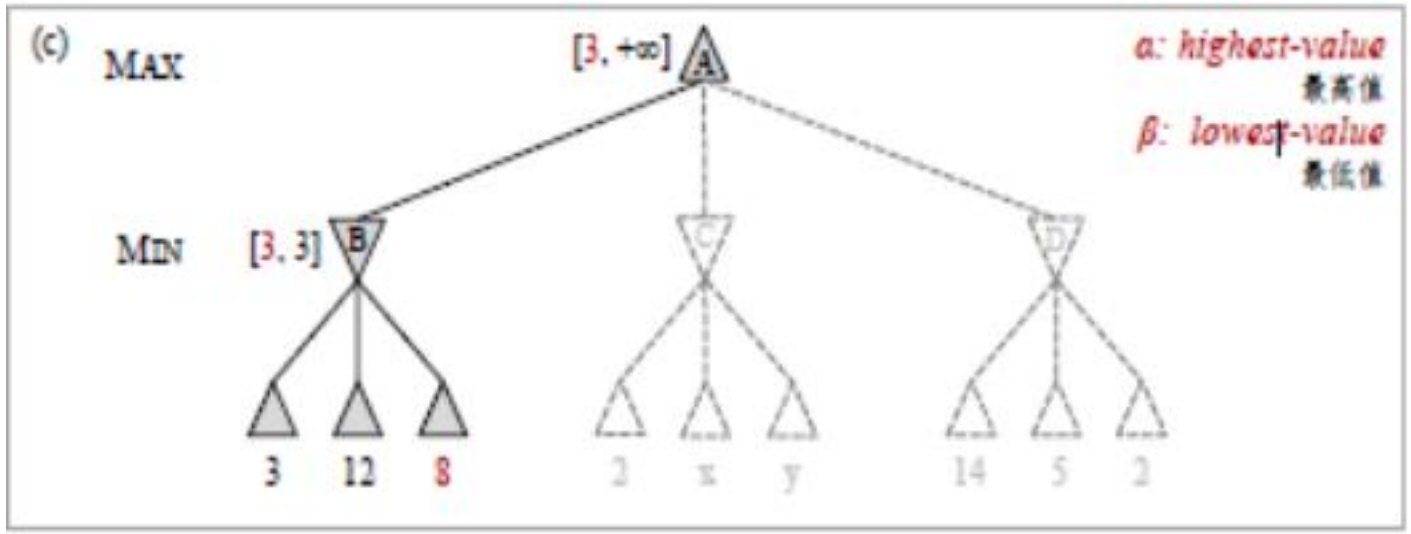
$B[\beta=3]$.

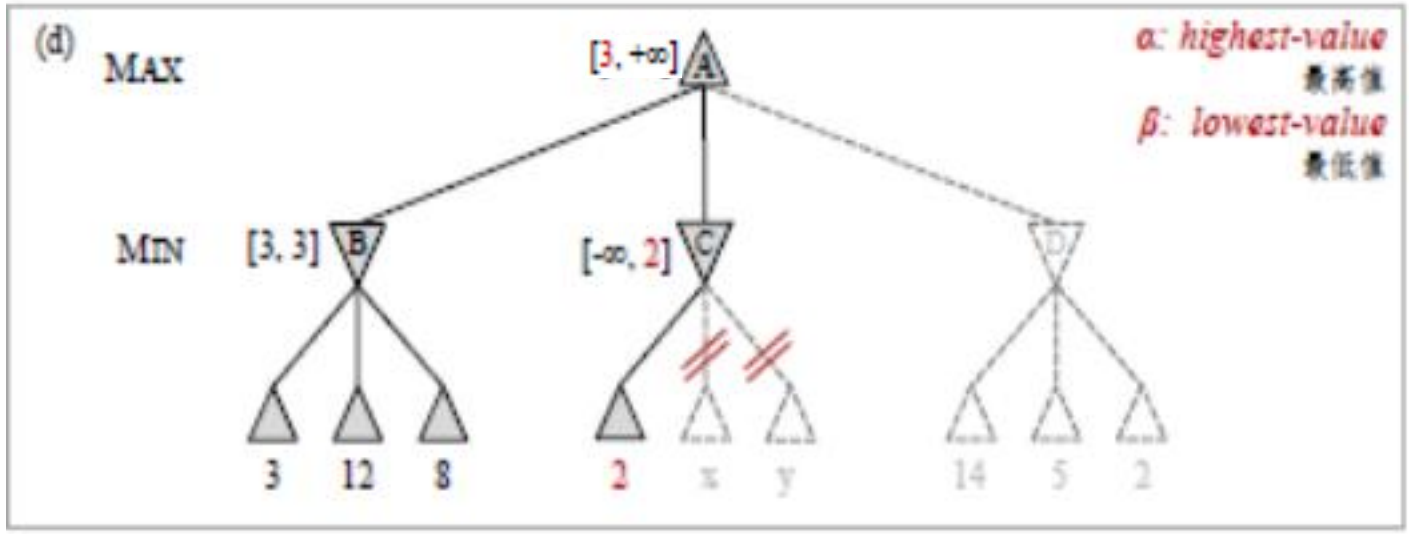
(b) The 2nd leaf below B has the value 12;MIN would avoid this move,

still,

$B[\beta=3]$.

Example: Game Tree Using Alpha-Beta Pruning 采用Alpha-Beta剪枝的博弈树





(c) The 3rd leaf below B has a value of 8; so exactly MIN node *B*[*β*=3].

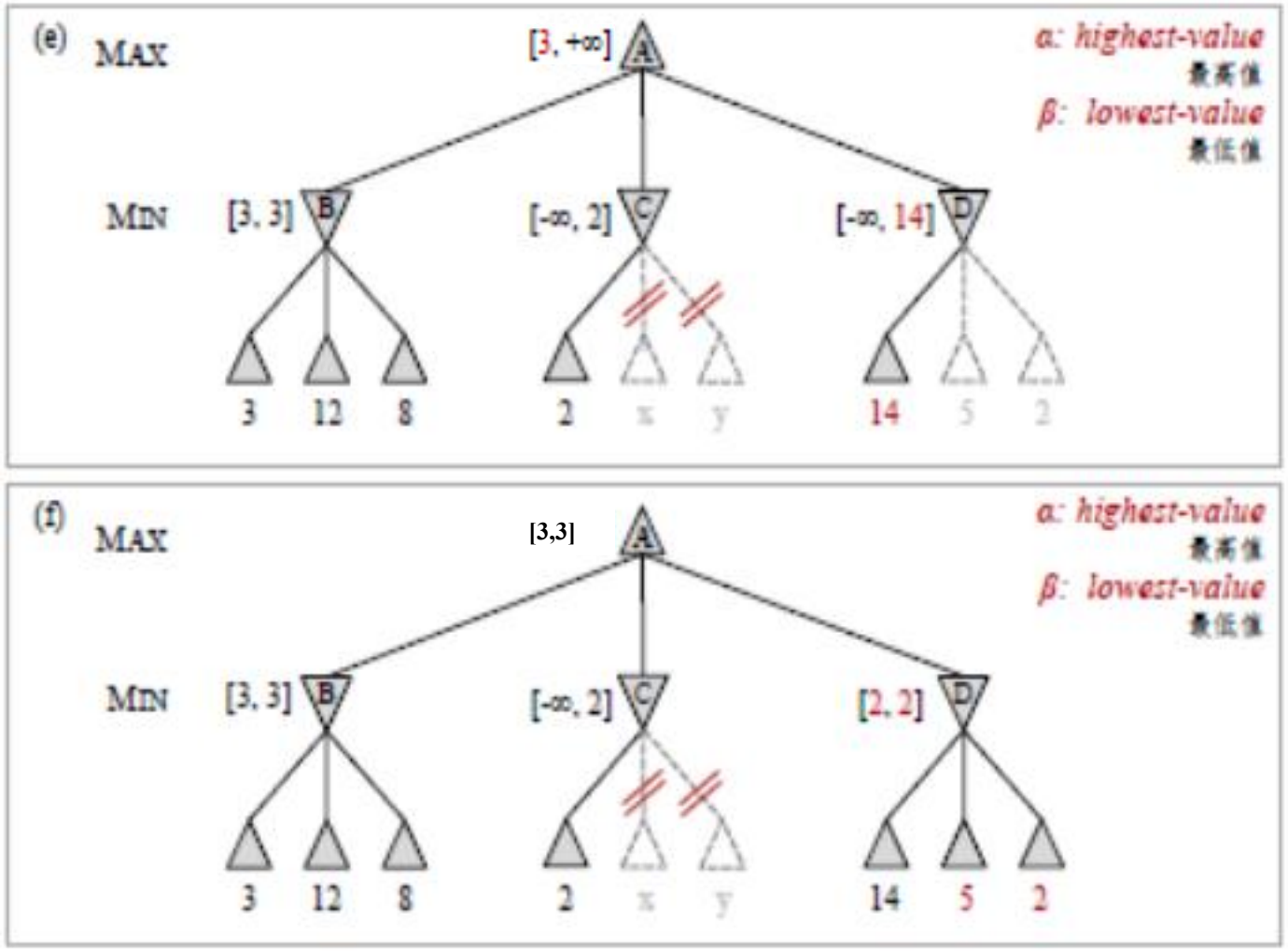
Now, we can infer B[α=3], because MAX has A[α≥3].

*B*下面第三个叶节点的值为8；故MIN节点正是*B*[*β*=3]。现在，因为MAX为*A*[*α*≥3]，我们能够推出*B*[*α*=3]。

(d) The 1st leaf below *C* has the value 2, hence, as a MINnode *C*[*β*=2], and *B*[*β*=3]>*C*[*β*=2], so MAX would never choose *C*. Therefore just prune all successor of *C*(*α*–*β* pruning).

C下面第一个叶节点的值为2，因此，由于MIN节点*C*[*β*=2], 且*B*[*β*=3]>*C*[*β*=2], 故MAX将不会选择C，所以只需剪掉C的所有后继节点(*α*–*β*pruning)。

Example: Game Tree Using Alpha-Beta Pruning 采用Alpha-Beta剪枝的博弈树



(e) The 1st leaf below D is 14, $D[\beta \leq 14]$, so we need to keep exploring D's successor states. We now have bounds on all of root's successors, so $A[\beta \leq 2]$.

D下面第一个叶节点为14，$D[\beta \leq 14]$，故我们需要不断搜索D节点的后继状态。到此我们已经遍布了根节点的所有后继节点，故$A[\beta \leq 2]$。

(f) The 2nd successor of D is worth 5, so keep exploring. The 3rd successor is worth 2, so $D[\beta = 2]$. MAX's decision at the root keeps $A[\beta = 2]$.

D的第2个后继节点的值等于5，故不断搜索，第3个后继节点等于2，故$D[\beta = 2]$。

根节点MAX的抉择保持$A[\beta = 2]$。

Why Called Alpha-Beta 为何称其为Alpha-Beta

- Alpha–beta pruning gets its name from the following two parameters:

  Alpha-Beta剪枝从如下两个参数得到其名称：

  ✓α: highest-value we have found so far at any point along the path for MAX.

  α：沿着MAX路径上的任意选择点，迄今为止我们已经发现的最高值。

  ✓β: lowest-value we have found so far at any point along the path for MIN.

  β：沿着MIN路径上的任意选择点，迄今为止我们已经发现的最低值。

- Alpha–beta search respectively:

  Alpha-Beta搜索依次完成如下动作：

  ✓updates the values of $\alpha$and βas it goes along, and

  边搜索边更新α和β的值，并且

  ✓prunes the remaining branches at a node as soon as the value of the current node is known to be worse than the current αor βvalue for MAX or MIN.

  一旦得知当前节点的值比当前MAX或MIN的α或β值更差，则在该节点剪去其余的分枝。

Alpha-Beta 搜索算法

```
function ALPHA-BETA-SEARCH(state) returns an action
v← MAX-VALUE(state, −∞, +∞)
return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
if TERMINAL-TEST(state) then return UTILITY(state)
v← −∞
for each a in ACTIONS(state) do
v← MAX(v, MIN-VALUE(RESULT(state, a), α, β))
if v ≥ β then return v    // a（后辈层）≥β（先辈层），β剪枝
else α←MAX(α, v)
return v

function MIN-VALUE(state, α, β) returns a utility value
if TERMINAL-TEST(state) then return UTILITY(state)
v← +∞
for each a in ACTIONS(state) do
v← MIN(v, MAX-VALUE(RESULT(state, a), α, β))
if v ≤ α then return v    // a（先辈层）≥β（后辈层），α剪枝
else β←MIN(β, v)
return v
```

General Principle of Alpha-Beta Pruning α - β 剪枝的一般原则

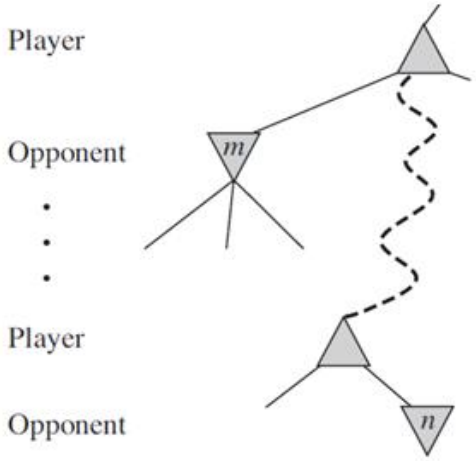
- Alpha–beta pruning can be applied to trees of any depth, and often possible to prune entire subtrees rather than just leaves.

  α - β剪枝可被用于任意深度的树，并且常常可以剪去整个子树而不仅仅是叶节点。

- The general principle: 一般原则：
  - Consider a node *n* somewhere in the tree, such that Player has a choice of moving to that node.

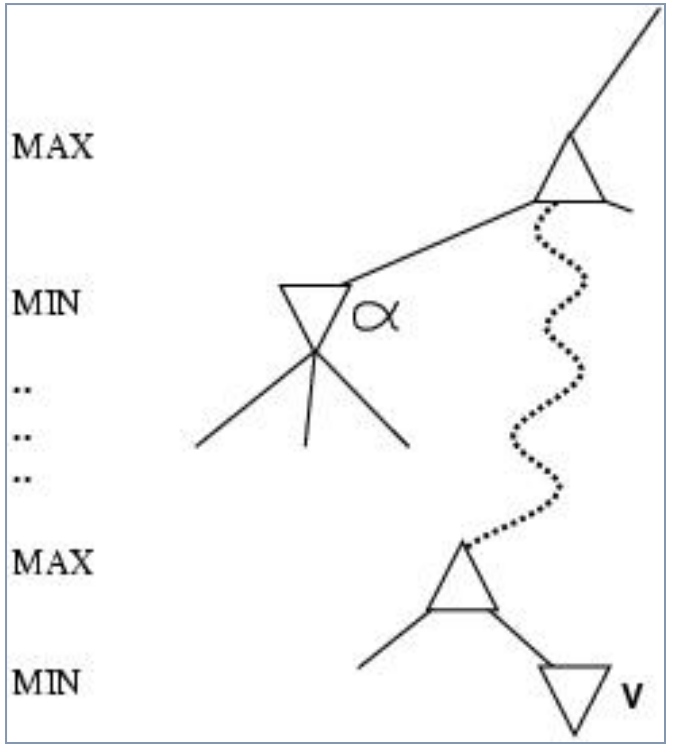
    设某个节点n位于树的某处，于是玩家选择移向那个节点。
  - If Player has a better choice *m* at parent node of *n*, or at any choice point further up, then *n will never be reached* in actual play.

    若玩家在位于n的父节点或更上层处有更好的选择m，则在实战中 完全没必要抵达n。



α－β是什么？

- α是MAX至今为止的路径上所有选择点中发现的最好选择的值，即是最大值。
  如果v比α差，MAX会避免它，即发生剪枝。
- 类似的，β是给MIN记录的最好结果即是最小值。
  - 如果v比β差，MIN会避免它，即发生剪枝。



如何做的更好？

- 假设博弈树的分支因子为b

  则极大极小算法检测$O(b^h)$个节点，最坏情况下α-β剪枝也是一样的。

  - 以下情况下α-β剪枝比极大极小算法优异：

    a. 一个MAX节点的MIN孩子们是按降序排列的。

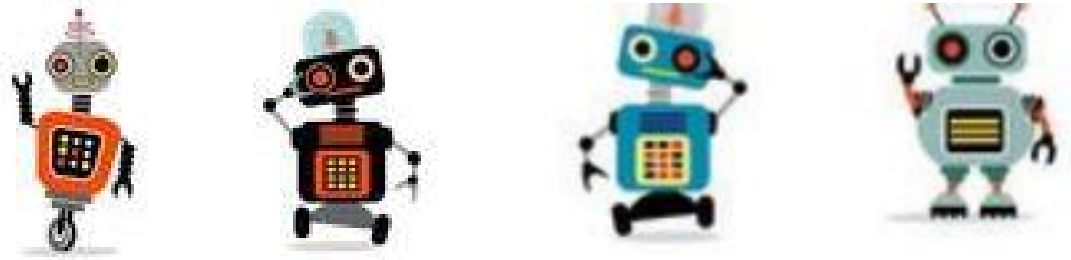
    b. 一个MIN节点的MAX孩子们是按升序排列的。

  - 这种情况下α-β剪枝算法检测$O(b^{h/2})$个节点。[Knuth and Moore, 1975]
  - 但这需要一个**神谕**（如果我们知道节点的完美排序，我们就不需要搜索博弈树了）。

- 如果节点是按随机排序的，那么α-β剪枝算法检测$O(b^{3h/4})$个节点。

- **启发式极小极大值：节点的启发式排序**

  根以下的节点排序依照前一次循环所得到的倒推值进行。



其他改进方法

- 采用适应深度限制+循环加深
- 扩大搜索范围：保留k>1条的路径，代替仅保留一条，并且在大于设置深度的叶子节点下扩展博弈树（帮着对付水平线效应——指当前的后继都是差不多的状态）。
- 特别情况扩展：在设置深度h时，如果一个节点明显地比其他的节点好，则沿着这个移动继续扩展几步。
- 用对照表法对付重复状态。
- 另外还有诸如当 $\alpha \geq \beta$ 虽然不成立，但 α 不比 β 小多少时，仍然采用剪枝，特别是在开局初期。

Algorithm of AlphaGo AlphaGo的算法

- Deep neural networks 深度神经元网络
  - value networks: used to evaluate board positions
    价值网络：用于评估棋局
  - policy networks: used to select moves.
    策略网络：用于选择走子
- Monte-Carlo tree search (MCTS) 蒙特卡罗树搜索 (MCTS)
  - Combines Monte-Carlo simulation with value networks and policy networks.
    将蒙特卡罗仿真与价值和策略网络相结合
- Reinforcement learning 强化学习
  - used to improve its play.
    用于改进它的博弈水平。

Source: Mastering Go with deep networks and tree search
Nature, Jan. 28, 2016

About Monte-Carlo Methods 关于蒙特卡罗方法

- Monte-Carlo methods are a broad class of computational algorithms that rely on *repeated random sampling* to obtain numerical results.
  蒙特卡罗方法是一大类计算算法，它凭借重复随机采样来获得数值结果。
- They tend to follow a particular pattern:
  它们往往遵循如下特定模式：
  - define a domain of possible inputs;
    定义一个可能的输入域；
  - generate inputs randomly from a probability distribution over the domain;
    从该域的一个概率分布随机地生成输入；
  - perform a deterministic computation on the inputs;
    对该输入进行确定性计算；
  - aggregate the results.
    将结果聚合。

Example: Approximating π by Monte-Carlo Method 用蒙特卡罗方法估计π

- Given that circle and square have a ratio of areas that is $\pi/4$, the value of π can be approximated using a Monte-Carlo method:

  鉴于圆形与正方形面积之比为$\pi/4$，则π的值可采用蒙特卡罗方法近似得出：

  a) Draw a square on the ground, then inscribe a circle within it.

  先画出一个正方形，然后在其中画一个圆弧。

  b) Uniformly scatter some objects of uniform size over the square.

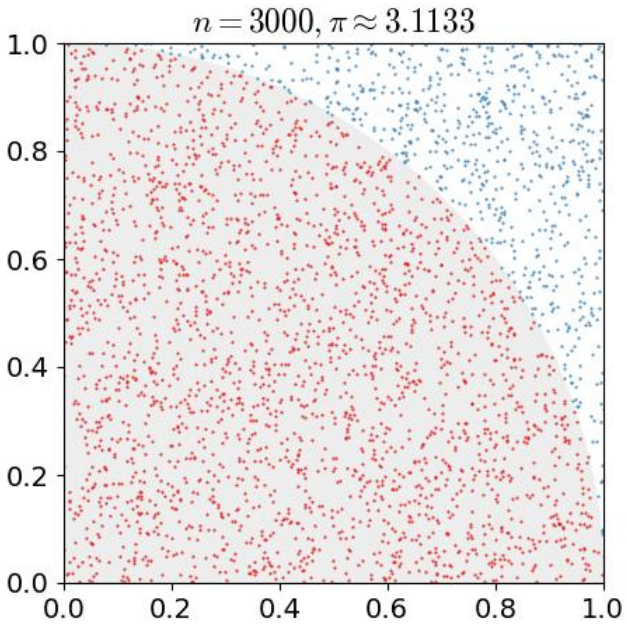
  将尺寸大小一致的小颗粒散落在正方形上。

  c) Count the number of objects inside the circle and the square.

  计算圆形和正方形中小颗粒的数量和总的数量。

  d) The ratio of the two counts is an estimate of the ratio of the two areas, which is $\pi/4$. Multiply the result by 4 to estimate π.

  两个数量之比为两个面积的估算，即$\pi/4$。结果乘以4得出π。



Monte-Carlo Tree Search (MCTS) 蒙特卡罗树搜索 (MCTS)

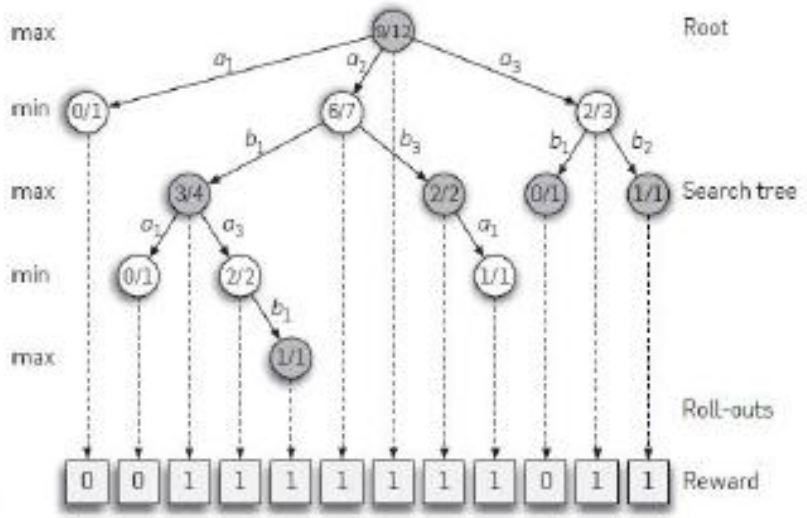
- MCTS combines Monte-Carlo simulation with game tree search.
  MCTS将蒙特卡罗仿真与博弈树搜索相结合。
- Like minimax, each node corresponds to a single state of game.
  和minimax一样，每个节点对应于一个的博弈状态。
- Unlike minimax, the values of nodes are estimated by Monte-Carlo simulation.
  不同于minimax，节点的值通过蒙特卡罗仿真来估值。

$W(a)/N(a)$ = the value of action a 动作a的值

where: 其中

$W(a)$ = the total reward 总的奖励

$N(a)$ = the number of simulations 仿真的数量



Source: Communications of the ACM, Mar. 2012, 55(3), pp. 106-113

对抗搜索：蒙特卡洛树搜索

利用(exploitation)与探索(exploration)在游戏博弈树上的有机协调 推荐阅读材料

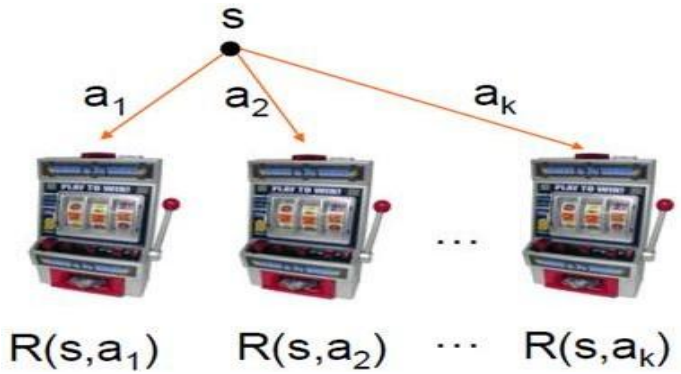
- David Silver, et.al., Mastering the game of Go with Deep Neural Networks and Tree Search, ***Nature***, 529:484-490,2016
- Cameron Browne, et.al., Survey of Monte Carlo Tree Search Methods, ***IEEE Transactions on Computational Intelligence and AI in Games***, 4(1):1-49,2012
- Sylvain Gelly, Levente Kocsis, Marc Schoenauer, et al., The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions, ***Communications of the ACM***, 55(3):106-113,2012
- Levente KocsisCsaba Szepesvari, Bandit Based Monte-Carlo Planning, ***ECML*** 2006
- Auer, P., Cesa-Bianchi, N., & Fischer, P. , Finite-time analysis of the multi-armed bandit problem, ***Machine learning***, 47(2), 235-256, 2002

蒙特卡洛规划（Monte-Carlo Planning）

- 单一状态蒙特卡洛规划： 多臂赌博机 (multi-armed bandits)
- 上限置信区间策略 (Upper Confidence Bound Strategies, UCB)
- 蒙特卡洛树搜索 (Monte-Carlo Tree Search)
  - UCT (Upper Confidence Bounds on Trees)

单一状态蒙特卡洛规划： 多臂赌博机 (multi-armed bandits)

- 单一状态， k种行动（即有k个摇臂）
- 在摇臂赌博机问题中，每次以随机采样形式采取一种行动a, 好比随机拉动第 k个赌博机的臂，得到$R(s,a_k)$ 的回报。
- 问题：下一次需要拉动哪个赌博机的臂，才能获得最大回报呢？



多臂赌博机 (multi-armed bandits)

- 多臂赌博机问题是一种序列决策问题，这种问题需要在**利用(exploitation)**和**探索(exploration)**之间保持平衡。
- 利用**(exploitation)**：保证在过去决策中得到最佳回报
- 探索(**exploration**)：寄希望在未来能够得到更大回报

多臂赌博机 (multi-armed bandits)

- 如果有k个赌博机，这k个赌博机产生的操作序列为$X_{i,1}, X_{i,2}, \ldots (\mathrm{i} = 1, \ldots, \mathrm{K})$。在时刻$t = 1,2 \ldots$，选择第$\mathrm{i}_t$个赌博机后，可得到奖赏$\mathrm{X}_{\mathrm{i}t,\ t}$，则在$n$次操作$\mathrm{i}_1, \ldots, \mathrm{i}_n$后，可如下定义悔值函数：

$$R_n = \max_{i=1,\ldots,k} \sum_{t=1}^{n} X_{i,t} - \sum_{t=1}^{n} X_{I_{t,t}}$$

- 悔值函数表示了如下意思：在第t次对赌博机操作时，假设知道哪个赌博机能够给出最大奖赏（虽然在现实生活中这是不存在的），则将得到的最大奖赏减去实际操作第i_t个赌博机所得到的奖赏。将n次操作的差值累加起来，就是悔值函数的结果。

- 很显然，一个良好的多臂赌博机操作的策略是在不同人进行了多次玩法后，能够让悔值函数的方差最小。

上限置信区间 (Upper Confidence Bound, UCB)

- 在多臂赌博机的研究过程中，上限置信区间（Upper Confidence Bound, UCB）成为一种较为成功的策略学习方法，因为其在探索-利用（exploration-exploitation）之间取得平衡。
- 在UCB方法中，使 $X_{i,T_i(t-1)}$ 来记录第 i 个赌博机在过去 $t-1$ 时刻内的平均奖赏，则在第 t 时刻，选择使如下具有最佳上限置区间的赌博机：

$$I_t = \max_{i \in \{1,\dots,k\}} \{\overline{X_{i,T_i(t-1)}} + c_{t-1,T_i(t-1)}\}$$

其中 c_{ts} 取值定义如下：

$$c_{ts} = \sqrt{\frac{2\ln t}{s}}$$

$T_i(t) = \sum_{s=1}^{t} \prod(I_s = i)$ 为在过去时刻（初始时刻到 t 时刻）过程中选择第 i 个赌博机的次数总和。

Zhaoman, AI,
Dept. of AI, CS, CUG

上限置信区间 (Upper Confidence Bound, UCB)

也就是说，在第t时刻，UCB算法一般会选择具有如下最大值的第j个赌博机：

$$UCB = \bar{X}_j + \sqrt{\frac{2In\,n}{n_j}} \text{ 或者 } UCB = \bar{X}_j + C \times \sqrt{\frac{2In\,n}{n_j}}$$

X_j是第j个赌博机在过去时间内所获得的平均奖赏值，n_j是在过去时间内拉动第j个赌博机臂的总次数，n是过去时间内拉动所有赌博机臂的 总次数。C 是一个平衡因子，其决定着在选择时偏重探索还是利用。

从这里可看出UCB算法如何在探索-利用（exploration-exploitation）之间寻找平衡：既需要拉动在过去时间内获得最大平均奖赏的赌博机，又希望去选择拉动臂次数最少的赌博机。

上限置信区间 (Upper Confidence Bound, UCB)

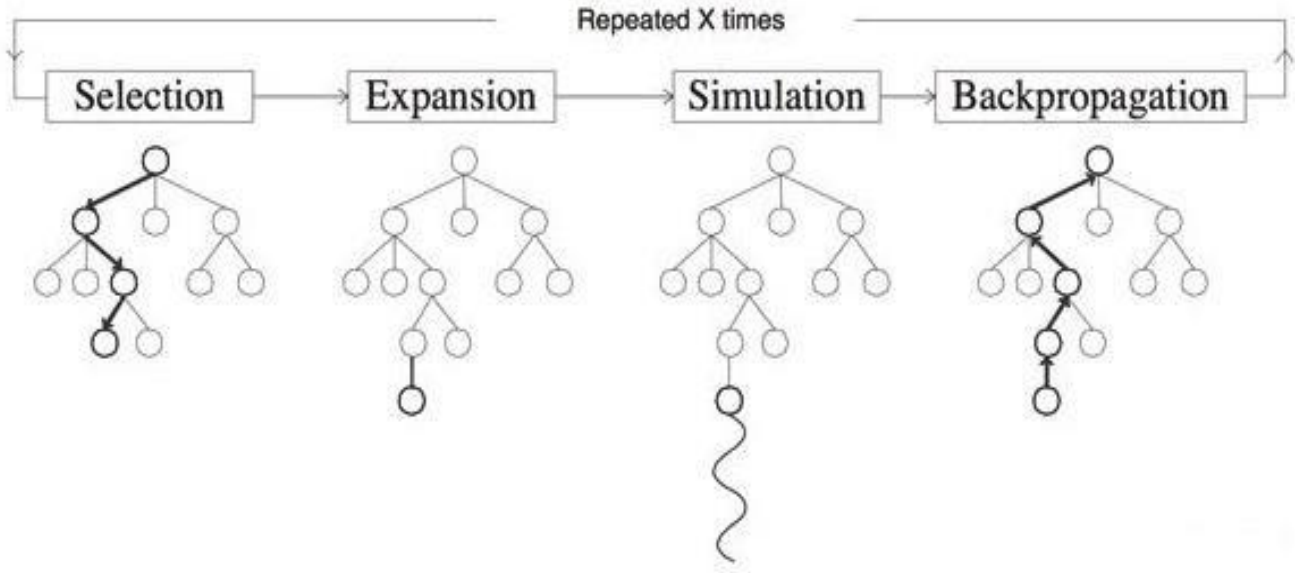
- UCB算法描述

Deterministic policy: UCB1.
Initialization: Play each machine once.
Loop:

- Play machine j that maximizes $\bar{x}_j + \sqrt{\frac{2 \ln n}{n_j}}$, where $\bar{x}_j$ is the average reward obtained from machine j, n_j is the number of times machine j has been played so far, and n is the overall number of plays done so far.

蒙特卡洛树搜索

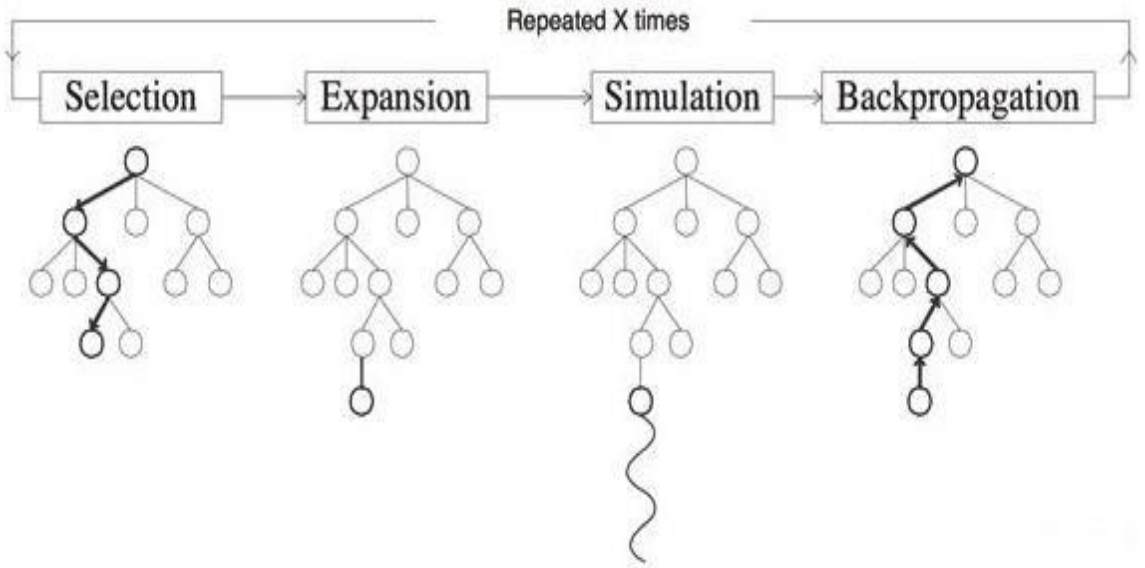
- 将上限置信区间算法UCB应用于游戏树的搜索方法，由Kocsis和Szepesvari在2006年提出
- 包括了四个步骤：选择(selection)，扩展(expansion)，模拟(simulation)，反向传播(Back-Propagation)



L. Kocsis and C. Szepesvari, Bandit based Monte-Carlo Planning, ***ECML***, 2006:282–293

蒙特卡洛树搜索

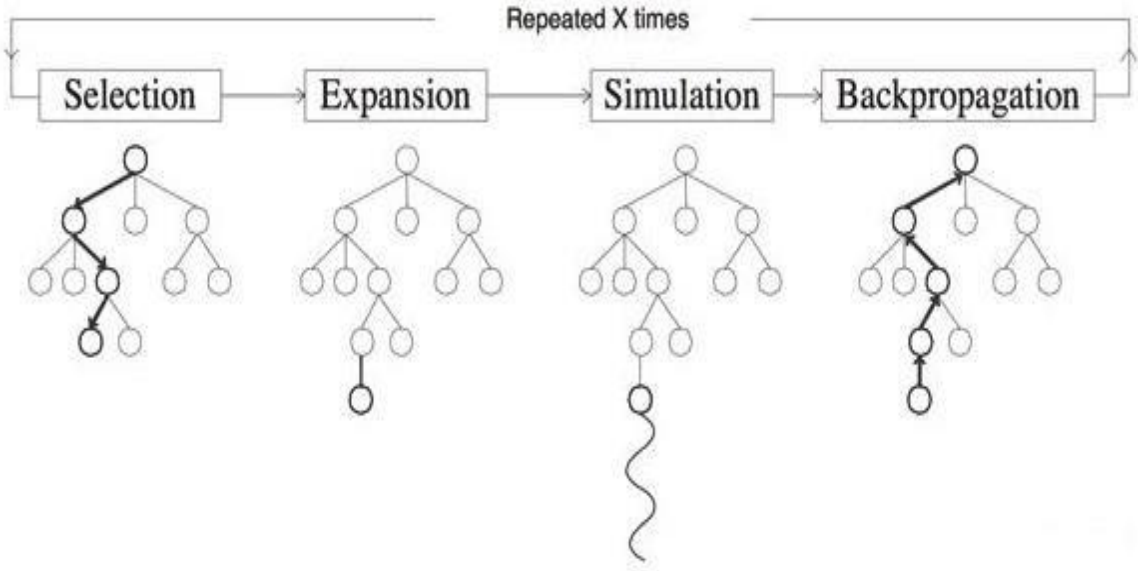
- **选择：**
  - 从根节点 R 开始，向下递归选择子节点，直至选择一个叶子节点L。
  - 具体来说，通常用UCB1（Upper Confidence Bound，上限置信区间）选择最具“潜力”的后续节点



$$UCB = X_j + \sqrt{\frac{2lnn}{n_j}}$$

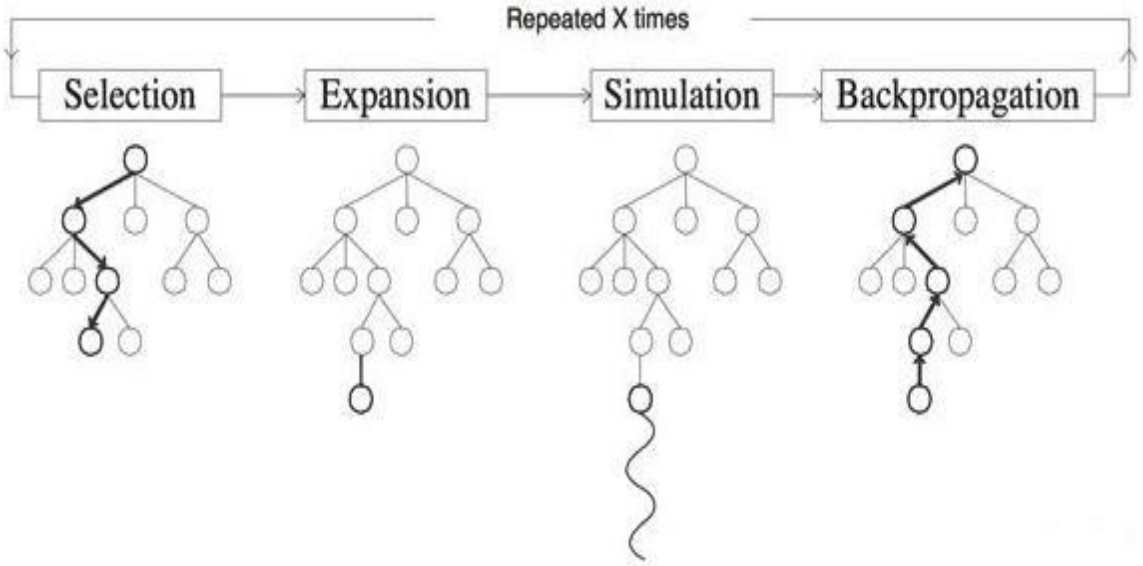
蒙特卡洛树搜索

- 扩展：
  - 如果 L 不是一个终止节点（即博弈游戏），则随机创建其后的一个未被访问节点，选择该节点作为后续子节点C。



蒙特卡洛树搜索

- **模拟：**
  - 从节点 C出发，对游戏进行模拟，直到博弈游戏结束。

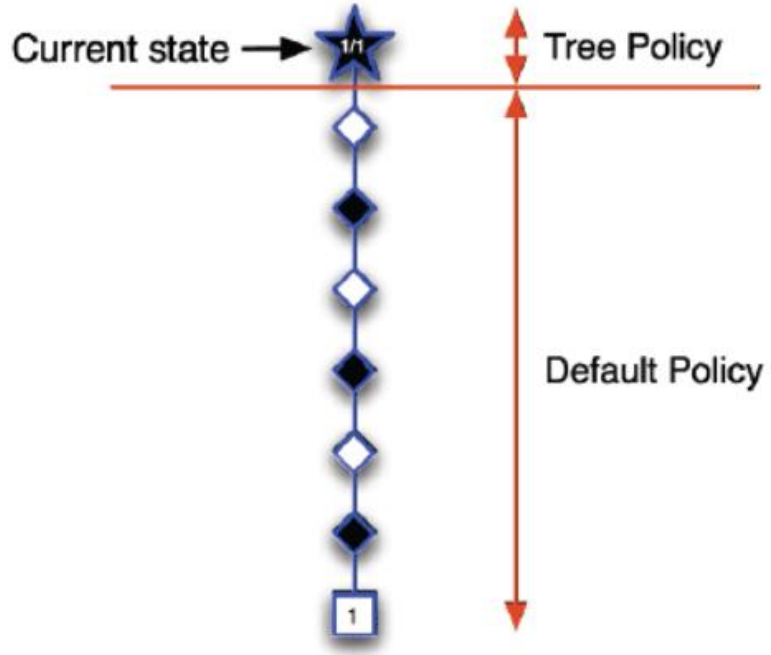


- **反向传播**
  - 用模拟所得结果来回溯更新导致这个结果的每个节点中获胜次数和访问次数。

蒙特卡洛树搜索

两种策略学习机制：

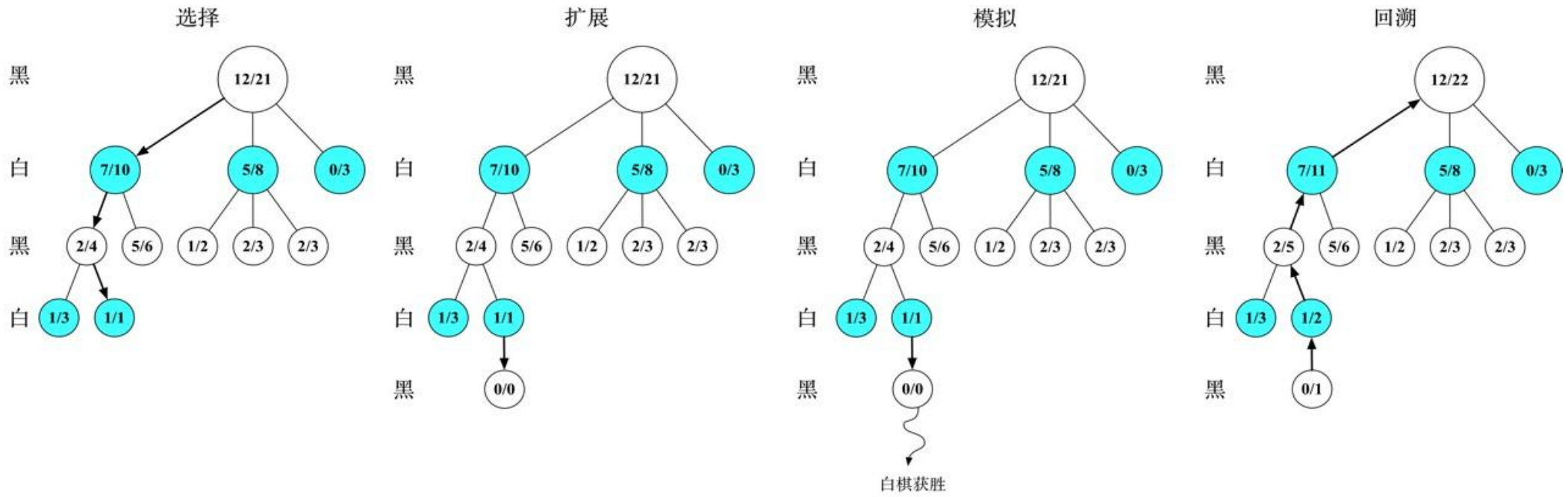
- 搜索树策略：从已有的搜索树中选择或创建一个叶子结点（即蒙特卡洛中选择和拓展两个步骤）。搜索树策略需要在利用和探索之 间保持平衡。
- 模拟策略：从非叶子结点出发模拟游戏，得到游戏仿真结果。



蒙特卡洛树搜索：例子

- 以围棋为例，假设根节点是执黑棋方。
- 图中每一个节点都代表一个局面，每一个局面记录两个值 A/B：

A：该局面被访问中黑棋胜利次数。对于黑棋表示己方胜利次数，对于白棋表示己方失败次数(对方胜利次数)　　B：该局面被访问的总次数。



蒙特卡洛树搜索：例子

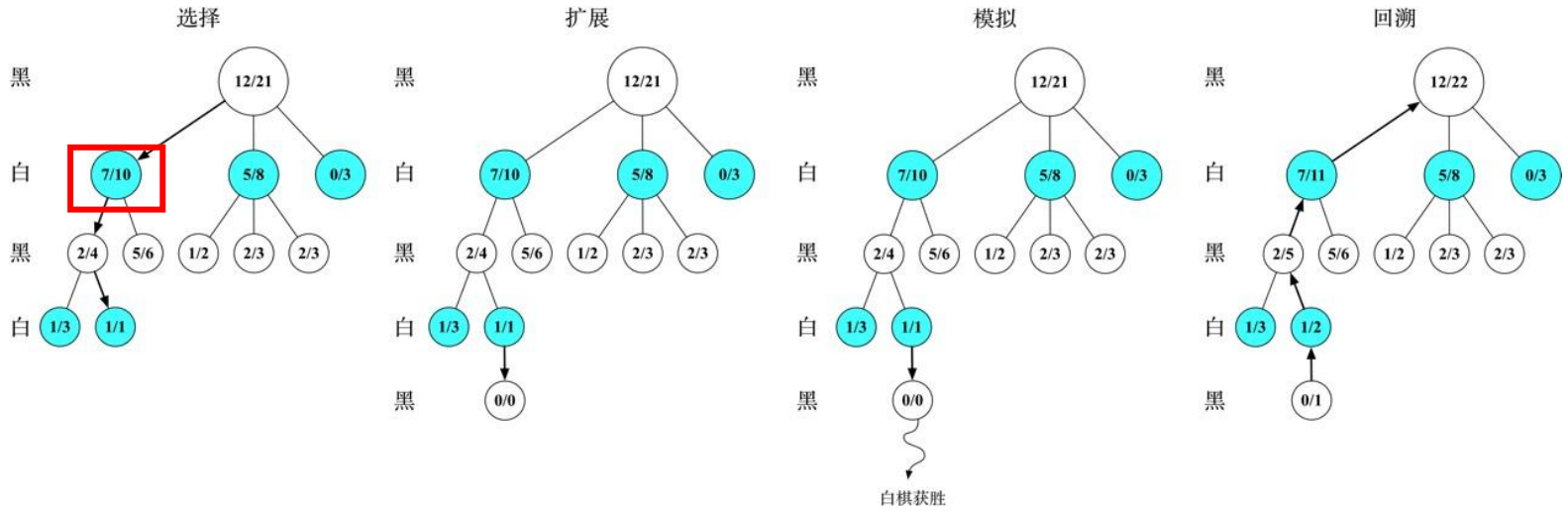
蒙特卡洛树搜索四个步骤：假设根结点由黑棋行棋，为了选择根节点的后续节点，需要由UCB1公式来计算根节点后续节点如下值，取一个值最大的节点作为后续节点：

左1：7/10对应的局面奖赏值为：$\frac{7}{10}+\frac{\sqrt{\log(21)}}{10}=1.252$

左2，5/8对应的局面奖赏值为：$\frac{5}{8}+\frac{\sqrt{\log(21)}}{8}=1.243$

左3，0/3对应的局面评估分数为：$\frac{0}{3}+\frac{\sqrt{\log(21)}}{3}=1.007$

黑棋会选择局面7/10进行行棋。



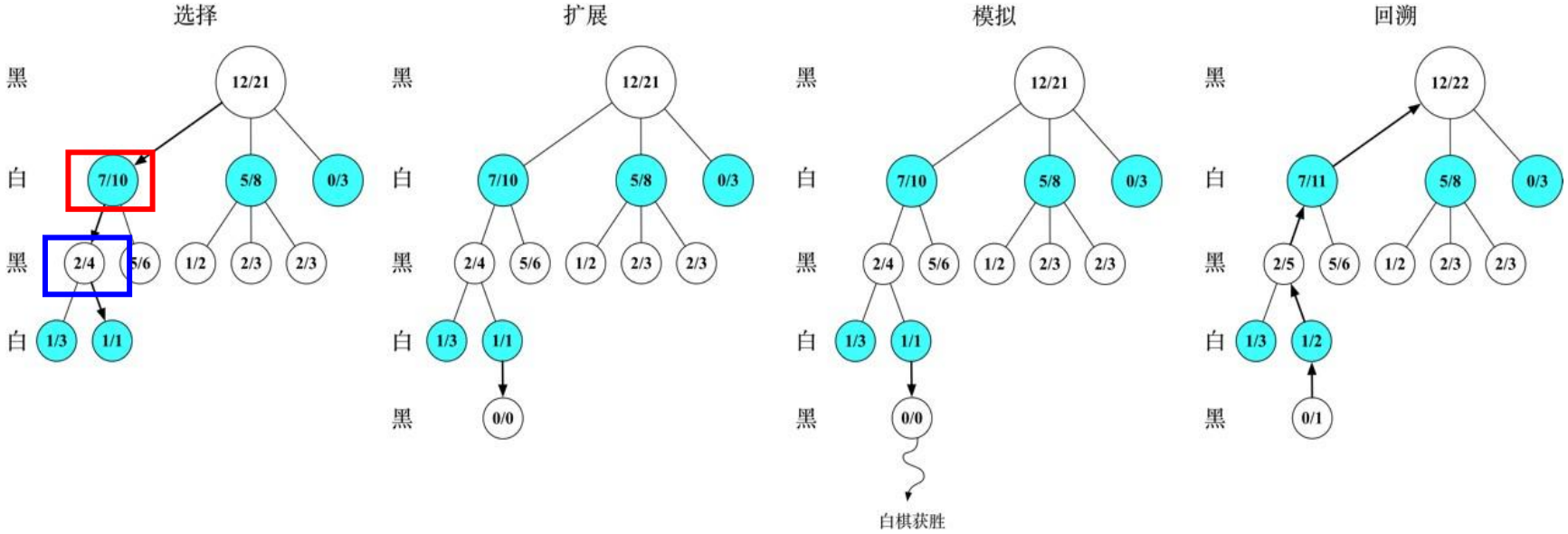
蒙特卡洛树搜索：例子

在节点7/10，由白棋行棋，评估该节点 下面的两个局面，由UCB1公式可得（注 意：此时A记录的是白棋失败的次数， 所以第一项为1-A/B）：

左1，2/4对应的局面奖赏为 $1-\frac{2}{4}+\frac{\sqrt{\log(21)}}{4}=1.372$

左2，5/6对应的局面奖赏为 $1-\frac{5}{6}+\frac{\sqrt{\log(21)}}{6}=0.879$

白棋会选择局面2/4进行行棋。



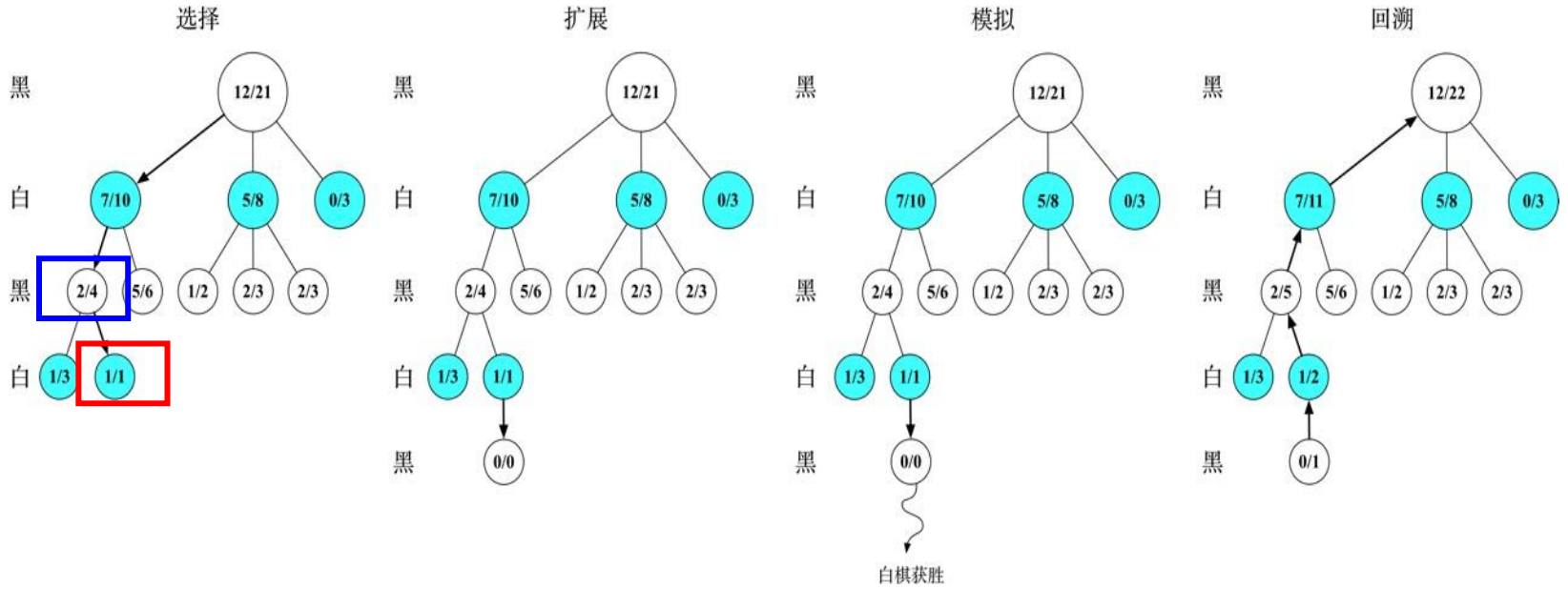
蒙特卡洛树搜索：例子

在节点2/4，黑棋评估下面的两个局面，由UCB1公式可得：

左1，1/3对应的局面奖赏为：$\frac{1}{3}+\frac{\sqrt{\log(21)}}{3}=1.341$

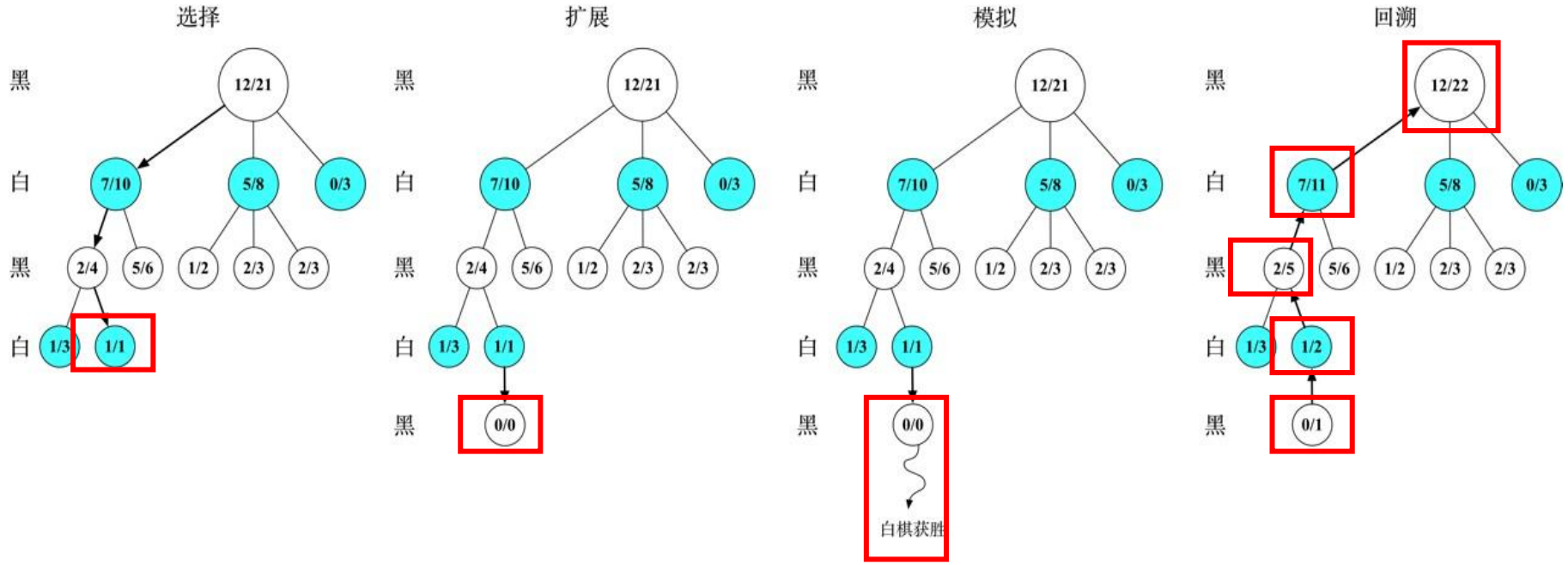
左1，1/1对应的局面奖赏为：$\frac{1}{1}+\frac{\sqrt{\log(21)}}{1}=2.745$

黑棋会选择局面1/1进行行棋。此时已经到达叶子结点，需要进行扩展。



蒙特卡洛树搜索：例子

随机扩展一个新节点。由于该新节点未被访问，所以初始化为0/0，接着在该节点下进行模拟。假设经过一系列仿真行棋后，最终白棋获胜。根据仿真结果来更新该仿真路径上每个节点的A/B值，该新节点的A/B 值被更新为0/1，并向上回溯到该仿真路径上新节点的所有父辈节点，即所有父辈节点的A不变，B值加1。



蒙特卡洛树搜索算法

function MONTE-CARLO-TREE-SEARCH(state) returns an action

    tree ← NODE(state)

while Is-Time-REMAINING( ) do

    leaf ← SELECT(tree)

    child ← EXPAND(leaf)

    result ← SIMULATE(child)

    BACK-PROPAGATE(result, child)

return Actions(state) whose node has highest number of playouts

//首先初始化一棵棋谱树，然后重复 SELECT / EXPAND / SIMULATE / BACK-PROPAGATE 循环，直到时间耗尽，最后返回下棋次数最多的节点。

蒙特卡洛树搜索算法 (Upper Confidence Bounds on Trees , UCT)

$\boldsymbol{S}$	状态集
$\mathrm{A}(\mathrm{s})$	在状态s能够采取的有效行动的集合
$\boldsymbol{s}(\boldsymbol{v})$	节点v所代表的状态
$\boldsymbol{a}(\boldsymbol{v})$	所采取的行动导致到达节点v
$\boldsymbol{f}:\ S\times \mathrm{A}\rightarrow S$	状态转移函数
$\boldsymbol{N}(\boldsymbol{v})$	节点v被访问的次数
$\boldsymbol{Q}(\boldsymbol{V})$	节点v所获得的奖赏值
$\boldsymbol{\Delta}(\boldsymbol{v},\boldsymbol{p})$	玩家p选择节点v所得到的奖赏值

蒙特卡洛树搜索

算法 基于最小最大搜索的蒙特卡洛树搜索算法

函数： UCTSearch
输入： 当前状态 s_0
输出： 玩家 MAX 行动下，当前最优动作 a^*

```
1 v0 ← create_node(s0)
2 while 未达到最大迭代次数 do
3 |  vl ← SelectPolicy(v0)
4 |  sr ← SimulatePolicy(vl.state)
5 |  BackPropagate(vl, sr)
6 end
7 a* ← UCB1(v0, 0)
```

函数： SelectPolicy
输入： 选择的起始结点 v_0
输出： 选择步骤的结束结点 v

```
1 v ← v0
2 while not terminal_test(v.state) do
3 |  if v 存在未被扩展的子结点 then
4 |  |  return Expand(v)
5 |  else
6 |  |  v ← UCB1(v, Cp)
7 |  end
8 end
```

函数： SimulatePolicy
输入： 状态 s_0
输出： 模拟的终止状态 s

```
1 s ← s0
2 while not terminal_test(s) do
3 |  a ←从 actions(s) 中随机采样
4 |  s ← result(s, a)
5 end
```

函数： BackPropagate
输入： 反向传播更新的起始结点 v，终局状态 s_t

```
1 while v is not null do
2 |  v.N ← v.N + 1
3 |  v.Q ← v.Q−utility(st, player(v.state))
4 |  v ← v.parent
5 end
```

函数： Expand
输入： 结点 v
输出： 未被扩展的后继结点 v'

```
1  a ← actions(v.state) 中随机的未探索动作
2  v' ← create_node(result(v.state, a))
3  v'.N ← 0
4  v'.Q ← 0
5  v'.parent ← v
6  v'.children ← ∅
7  v.children ← v.children∪{v'}
```

函数： UCB1
输入： 结点 v，超参数 c
输出： 置信上限最大的动作 a^*

$$a^* \leftarrow \operatorname{argmax}_{v' \in v.children} \frac{v'.Q}{v'.N} + c\sqrt{\frac{2\ln v.N}{v'.N}}$$

蒙特卡洛树搜索

- 蒙特卡洛树搜索是一种基于树结构的蒙特卡洛方法，所谓的蒙特卡洛树搜索就是基于蒙特卡洛方法在整个2^N（N等于决策次数，即树深度）空间中进行启发式搜索，基于一定的反馈寻找出最优的树结构路径（可行解）。概括来说就是，MCTS是一种确定规则驱动的**启发式随机搜索**算法。
- 上面这句话其实阐述了MCTS的5个主要核心部分：
  - **树结构：**树结构定义了一个可行解的解空间，每一个叶子节点到根节点的路径都对应了一个解（solution），解空间的大小为2^N（N等于决策次数，即树深度）
  - **蒙特卡洛方法：**MSTC不需要事先给定打标样本，随机统计方法充当了驱动力的作用，通过随机统计实验获取观测结果。
  - **损失评估函数：**有一个根据一个确定的规则设计的可量化的损失函数（目标驱动的损失函数），它提供一个可量化的确定性反馈，用于评估解的优劣。从某种角度来说，MCTS是通过随机模拟寻找损失函数代表的背后“真实函数”。
  - **反向传播线性优化：**每次获得一条路径的损失结果后，采用反向传播（Backpropagation）对整条路径上的所有节点进行整体优化，优化过程连续可微
  - **启发式搜索策略：**算法遵循损失最小化的原则在整个搜索空间上进行启发式搜索，直到找到一组最优解或者提前终止

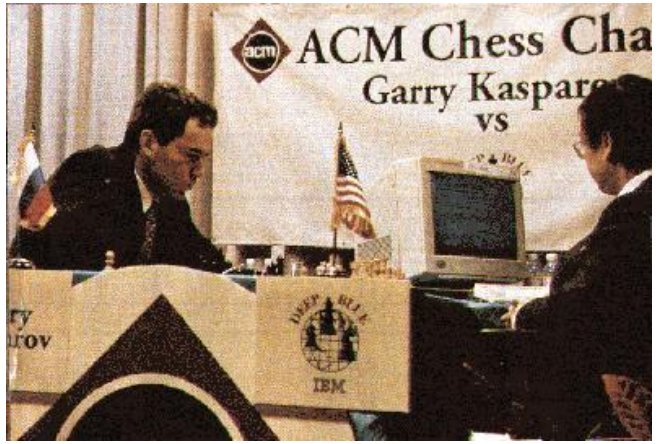
Game Playing Algorithms Today 博弈算法的进展

Computers are better than humans 计算机优于人类	
Checkers 西洋跳棋	Solved in 2007 2007年已解决
Chess 国际象棋	IBM Deep Blue defeated Kasparov in 1997 IBM深蓝于1997年战胜了卡斯帕罗夫
Go 围棋	Google AlphaGo beat Lee Sedol, a 9 dan professional in Mar. 2016 谷歌AlphaGo于2016年3月战胜了9段职业棋手李世乭
Computers are competitive with top human players **计算机与顶级人类玩家媲美**	
Backgammon 西洋双陆棋	TD-Gammon used reinforcement learning to learn evaluation function TD-Gammon使用了强化学习方法来得到评价函数
Bridge 桥牌	Top systems use Monte-Carlo simulation and alpha-beta search 顶级的系统使用蒙特卡罗仿真和alpha-beta搜索

博弈程序发展现状

- 西洋跳棋: Chinook 在1994年打败了人类冠军 Marion Tinsley 。
- 黑白棋： The Logistello software 在1997年6：0完败世界冠军。
- 国际象棋: Deep Blue 1997打败了人类冠军 Garry Kasparov.
- 围棋: 2016年7月18日，GoRatings公布最新世界排名，AlphaGo以3612分，超越3608分的柯洁成为新的世界第一。
- 2017年10月19日国际学术期刊《自然》（Nature）**阿尔法元100：0战胜哥哥阿尔法狗**

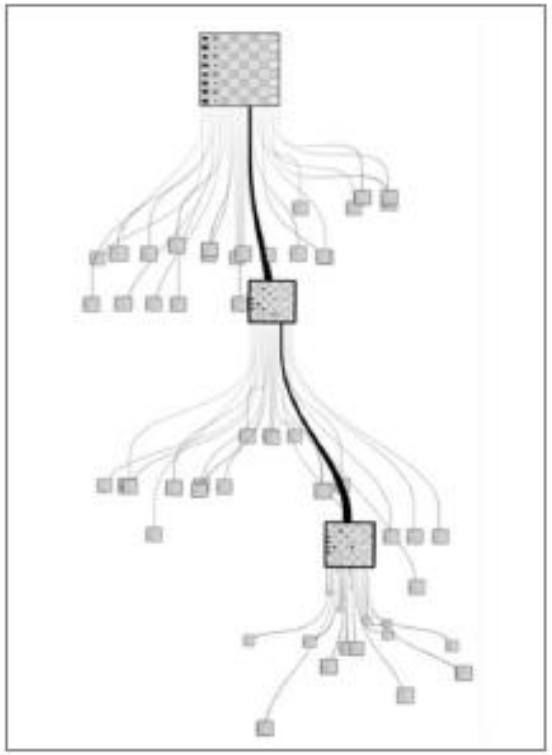




扩展阅读：Go vs. Chess 围棋与国际象棋

- Go has long been viewed as one of most complex game and most challenging of classic games for AI.

  围棋一直被视为最复杂的博弈之一、而且是最具挑战性的AI经典博弈。

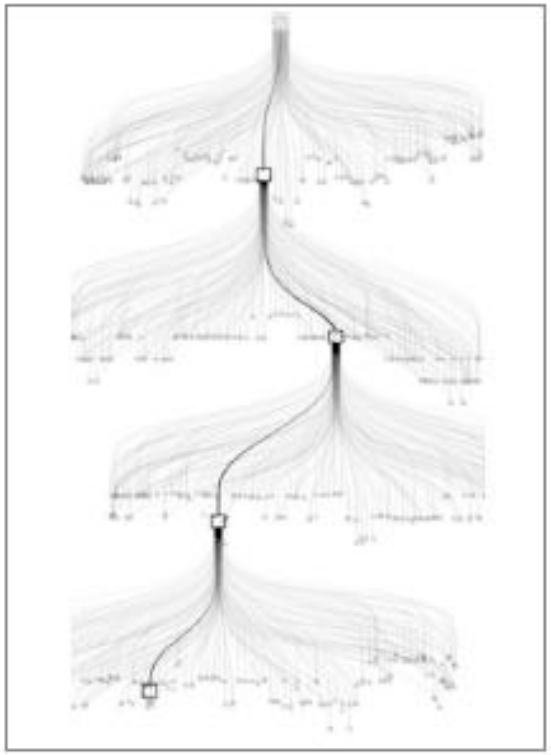


Chess (b ≈ 35, d ≈ 80)

8 x 8 = 64, possible games ≈ 10^{120}

Go (b ≈ 250, d ≈ 150)

19 x 19 = 361, possible games ≈ 10^{170}



Algorithm of AlphaGo AlphaGo的算法

- Deep neural networks 深度神经元网络
  - value networks: used to evaluate board positions
    价值网络：用于评估棋局
  - policy networks: used to select moves.
    策略网络：用于选择走子
- Monte-Carlo tree search (MCTS) 蒙特卡罗树搜索 (MCTS)
  - Combines Monte-Carlo simulation with value networks and policy networks.
    将蒙特卡罗仿真与价值和策略网络相结合
- Reinforcement learning 强化学习
  - used to improve its play.
    用于改进它的博弈水平。

Source: Mastering Go with deep networks and tree search
Nature, Jan. 28, 2016

AlphaGo算法解读

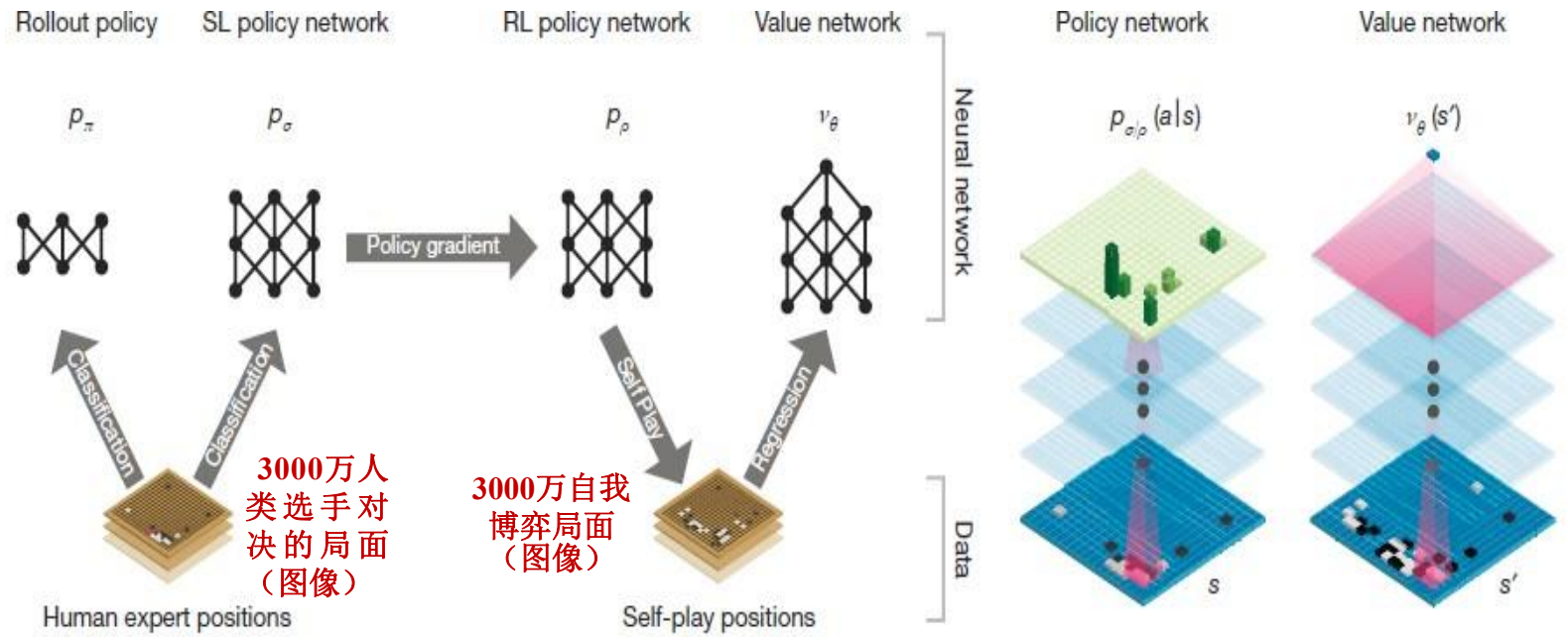


Figure 1 | Neural network training pipeline and architecture. a, A fast rollout policy p_π and supervised learning (SL) policy network p_σ are trained to predict human expert moves in a data set of positions. A reinforcement learning (RL) policy network p_ρ is initialized to the SL policy network, and is then improved by policy gradient learning to maximize the outcome (that is, winning more games) against previous versions of the policy network. A new data set is generated by playing games of self-play with the RL policy network. Finally, a value network v_θ is trained by regression to predict the expected outcome (that is, whether the current player wins) in positions from the self-play data set. b, Schematic representation of the neural network architecture used in AlphaGo. The policy network takes a representation of the board position s as its input, passes it through many convolutional layers with parameters σ (SL policy network) or ρ (RL policy network), and outputs a probability distribution $p_\sigma(a|s)$ or $p_\rho(a|s)$ over legal moves a, represented by a probability map over the board. The value network similarly uses many convolutional layers with parameters θ, but outputs a scalar value $v_\theta(s')$ that predicts the expected outcome in position s'.

- 将每个状态（局面）均视为一幅图像
- 训练策略（*policy*）网络和价值（*value*）网络
- $p_{\sigma,p}(a|s)$表示当前状态为s（局面）时，采取行动a后所得到的概率；$v_\theta(s')$表示当前状态为s'时，整盘棋获胜的概率。

David Silver, et.al., Mastering the game of Go with Deep Neural Networks and Tree Search, ***Nature***, 529:484-490,2016

AlphaGo算法解读：策略网络的训练

- **基于监督学习来先训练策略网络**
  - Idea: 执行监督学习 (SL) 以预测人类棋步
  - 给定状态s ，预测移动的概率分布$a, p_{\sigma,p}(a|s)$
  - 在 3000 万个位置上进行了训练，预测人类动作的准确率为 57%
  - 还训练了一个更小、更快的推出策略网络（准确率为 24%）
- **再基于强化学习来训练策略网络**
  - Idea: 使用强化学习 (RL) 对策略网络进行微调
  - 将 RL 网络初始化为 SL 网络
  - 网络的两个快照相互对弈，更新参数以最大化预期最终结果
  - RL 网络在 80% 的时间内战胜 SL 网络，在 85% 的时间内战胜开源 Pachi Go 程序

AlphaGo算法解读：价值网络的训练

- Idea: 训练网络进行位置评估
- 给定状态s'，估计$v_\theta(s')$，即两名玩家从位置 s 开始并遵循所学策略的预期对局结果
- 通过最小化实际结果与预测结果之间的均方误差来训练网络
- 在从不同自我对弈游戏中抽取的 30M 个位置上进行训练

AlphaGo算法解读：蒙特卡洛树搜索中融入了策略网络和价值网络

在通过深度学习得到的策略网络和价值网络帮助之下，如下完成棋局局面的选择和搜索。给定节点v_0，将具有如下最大值的节点v选择作为v_0的后续节点

$$\frac{Q(v)}{N(v)}+\frac{P(v|v_0)}{1+N(v)}$$

- 这里$P(v|v_0)$ 的值由策略网络计算得到。
- 在模拟策略阶段(default policy)，AlphaGo不仅考虑仿真结果，而且考虑价值网络计算结果。
- 策略网络和价值网络是离线训练得到的。

AlphaGo算法解读：蒙特卡洛树搜索中融入了策略网络和价值网络

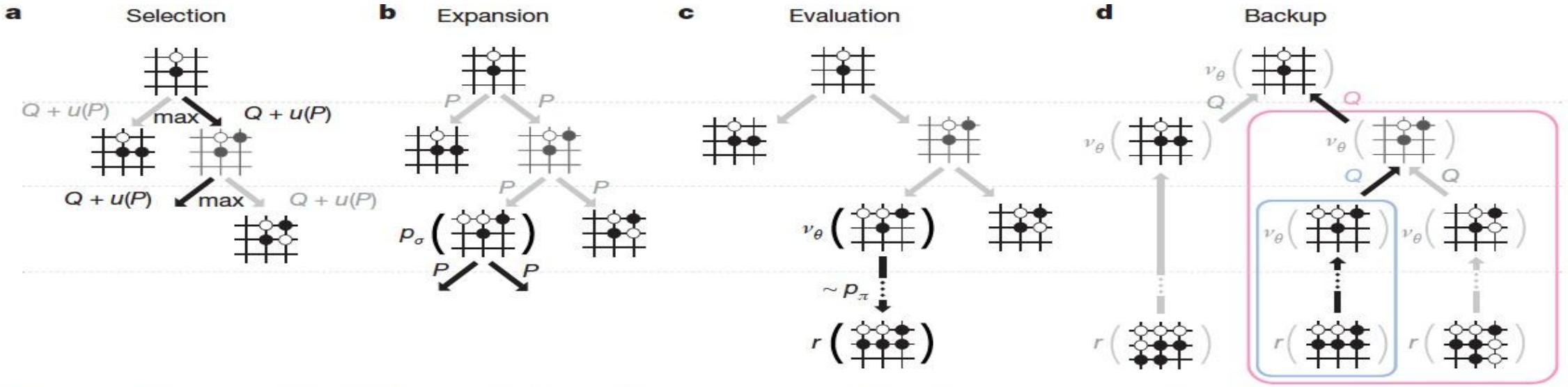


Figure 3 | Monte Carlo tree search in AlphaGo. **a**, Each simulation traverses the tree by selecting the edge with maximum action value Q, plus a bonus $u(P)$ that depends on a stored prior probability P for that edge. **b**, The leaf node may be expanded; the new node is processed once by the policy network p_σ and the output probabilities are stored as prior probabilities P for each action. **c**, At the end of a simulation, the leaf node is evaluated in two ways: using the value network v_θ; and by running a rollout to the end of the game with the fast rollout policy p_π, then computing the winner with function r. **d**, Action values Q are updated to track the mean value of all evaluations $r(\cdot)$ and $v_\theta(\cdot)$ in the subtree below that action.

$$a_t = \underset{a}{\operatorname{argmax}}(Q(s_t,a) + u(s_t,a))$$

$$N(s,a) = \sum_{i=1}^{n} 1(s,a,i)$$

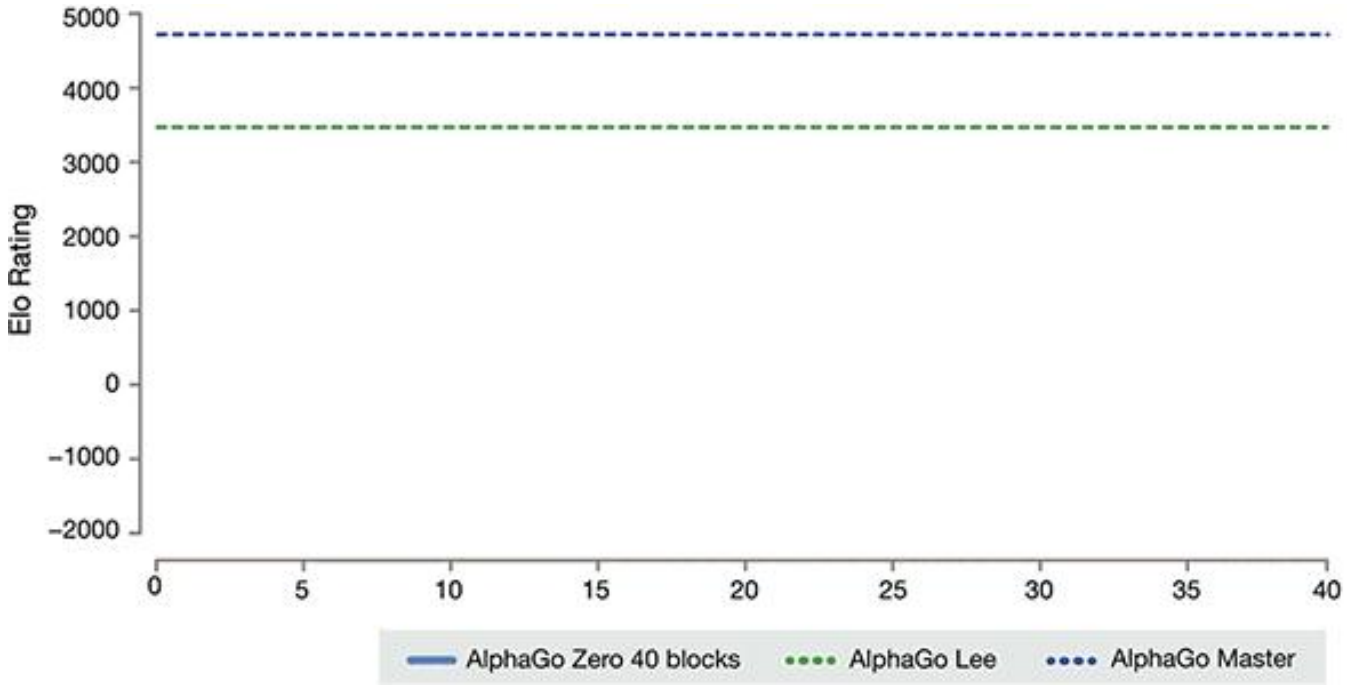
$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{n} 1(s,a,i) V(s_L^i)$$

价值网络计算

策略网络计算

$$u(s,a) \propto \frac{P(s,a)}{1+N(s,a)}$$

AlphaGo算法解读：AlphaGo Zero （一张白纸绘蓝图）



- 不需要人类选手对决的棋面进行训练
- 策略网络和价值网络合并
- 深度残差网络

经过40天训练后，Zero总计运行约2900万次自我对弈，得以击败AlphaGo Master，比分为89 比11

Mastering the game of Go without human knowledge, ***Nature***, volume 550, pages 354– 359 (19 October 2017)