# 搜索问题

局部搜索/对抗搜索

# 模拟退火搜索
# Simulated annealing search

- 思想: 为避免陷入局部最优，允许一些“坏”的移动，但是要逐渐减少“坏”移动的频率

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"
  local variables: current, a node
                   next, a node
                   T, a "temperature" controlling prob. of downward steps

  current ← MAKE-NODE(INITIAL-STATE[problem])
  for t ← 1 to ∞ do
      T ← schedule[t]
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← VALUE[next] – VALUE[current]
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(Δ E/T)
```

# 模拟退火的特点

- 可以证明，如果温度 $T$ 下降的速度足够慢，那么模拟退火找到全局最优的概率将逼近1
- 广泛用于超大规模集成电路设计，机场调度等

# 集束搜索

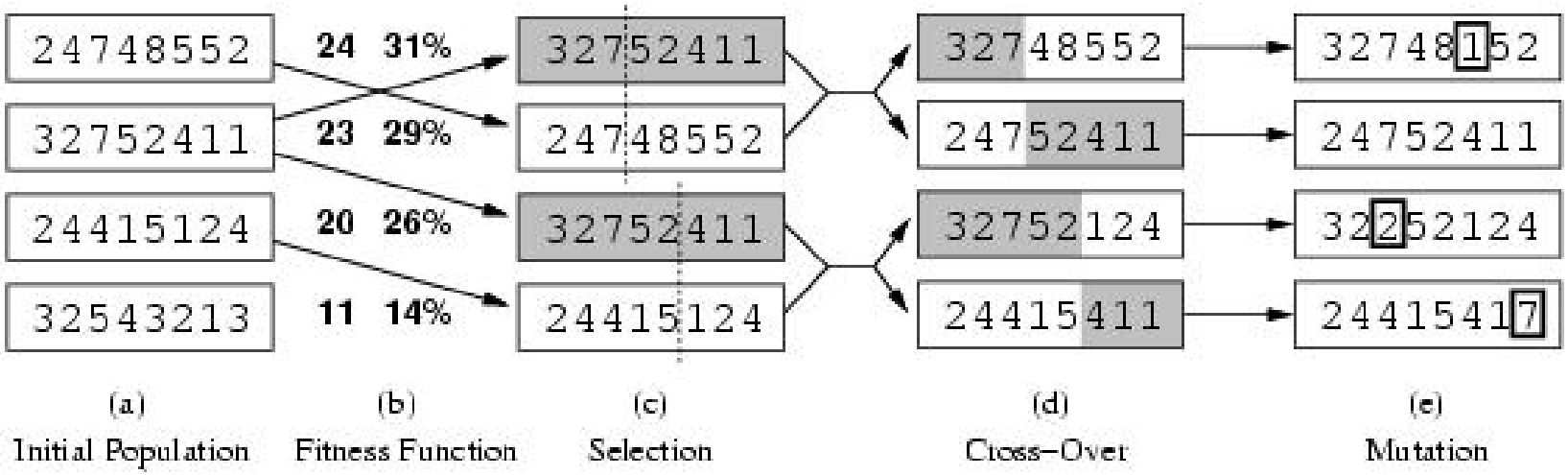
# Local beam search

- 追踪 $k$ 个状态，而不是仅仅一个
- 第一步：从随机产生的 $k$ 个状态开始
- 第二步：在每一轮，产生这 $k$ 个状态的所有孩子状态
- 第三步：若任何新的状态是目标，则停止；否则，从这些新的状态中选择 $k$ 个最好的状态然后重复第二步.

# 遗传算法
# Genetic algorithms

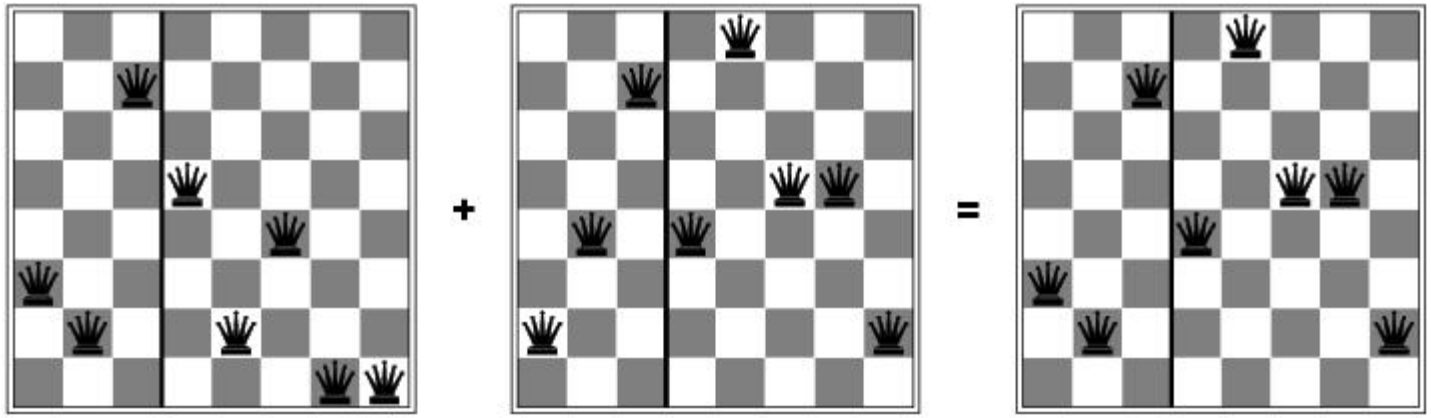
- 思想：一个孩子状态由两个父状态组合产生
- 第一步：从 $k$ 个随机生成的状态开始 (population)
- 第二步：一个状态被表述为一个字符串（通常是0/1 串）
- 第三步：采用评估函数 (fitness function). 好的状态得到高的分值
- 第四步：通过选择、交叉和变异来生成下一组状态（population)

# Genetic algorithms




- Fitness function: number of non-attacking pairs of queens (min = 0, max = $8 \times 7/2 = 28$)
- $24/(24+23+20+11) = 31\%$
- $23/(24+23+20+11) = 29\%$ etc

# Genetic algorithms



# 遗传算法

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population ← empty set
    for i = 1 to SIZE(population) do
      x ← RANDOM-SELECTION(population, FITNESS-FN)
      y ← RANDOM-SELECTION(population, FITNESS-FN)
      child ← REPRODUCE(x, y)
      if (small random probability) then child ← MUTATE(child)
      add child to new_population
    population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

---

```
function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

# 效果对比

表 1　和回溯法的时间对比表( $t$ /s)

| 皇后数 | 8 | 10 | 15 | 30 | 50 | 100 | 500 | 1 000 | 1 500 |
|---|---|---|---|---|---|---|---|---|---|
| 文献[4]方法 | 0.132 | 0.48 | 113.609 | | | | | | |
| 本文的方法 | 0 | 0 | 0.528 1 | 1.366 | 2.03 | 27.055 | 5 106.07 | 52 470.6 | 228 667 |
| 回溯法 | 0 | 0.47 | 578.04 | | | | | | |

数据来源

http://wenku.baidu.com/view/1cae7308bb68a98271fefa51.html

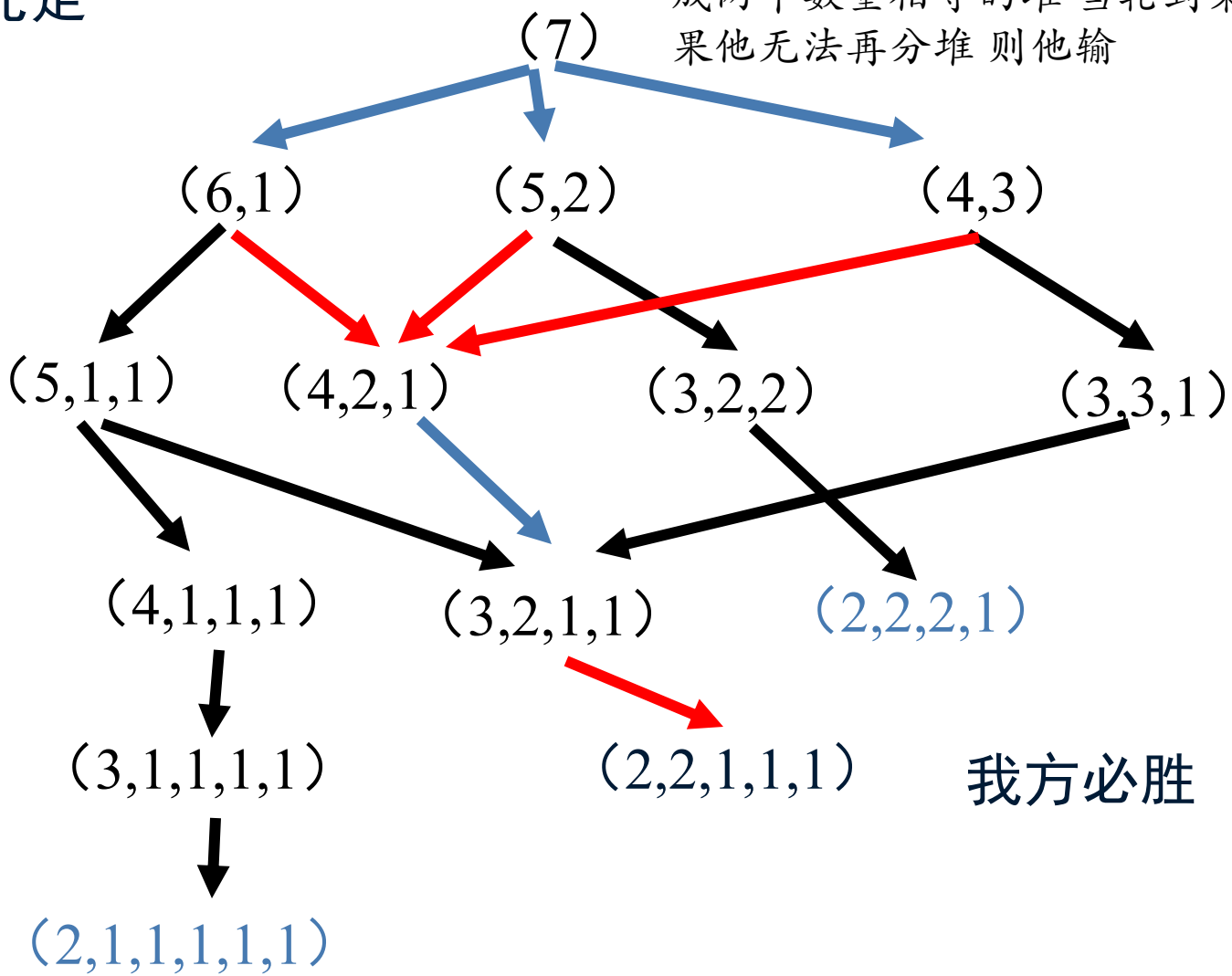
# 讨论

- 遗传算法看起来很美
  - 模拟进化
- 不能保证找到最优解
  - 本质上仍是近似算法
- 进化策略的选择存在较多优化空间
-

# 分钱币游戏

有一定数量的钱币 两个人轮流分 但不能分成两个数量相等的堆 当轮到某一个人时 如果他无法再分堆 则他输

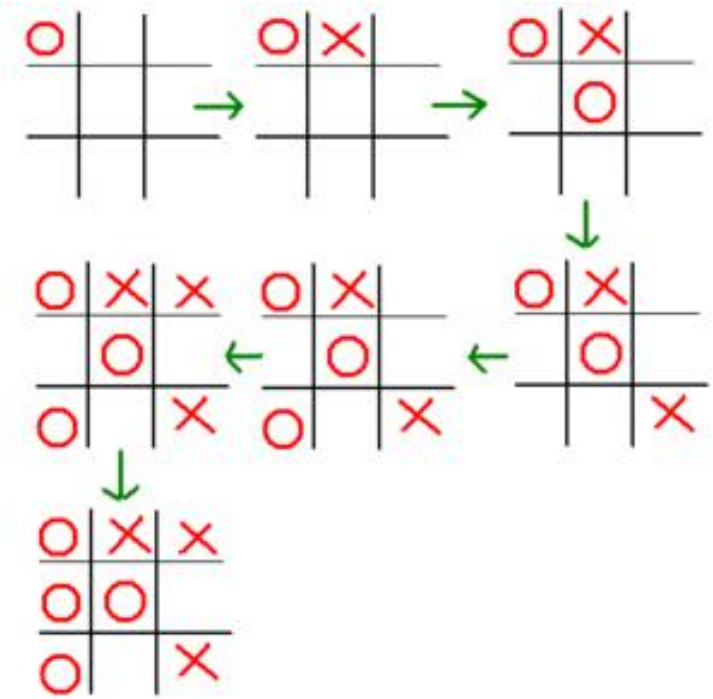
对方先走




我方必胜

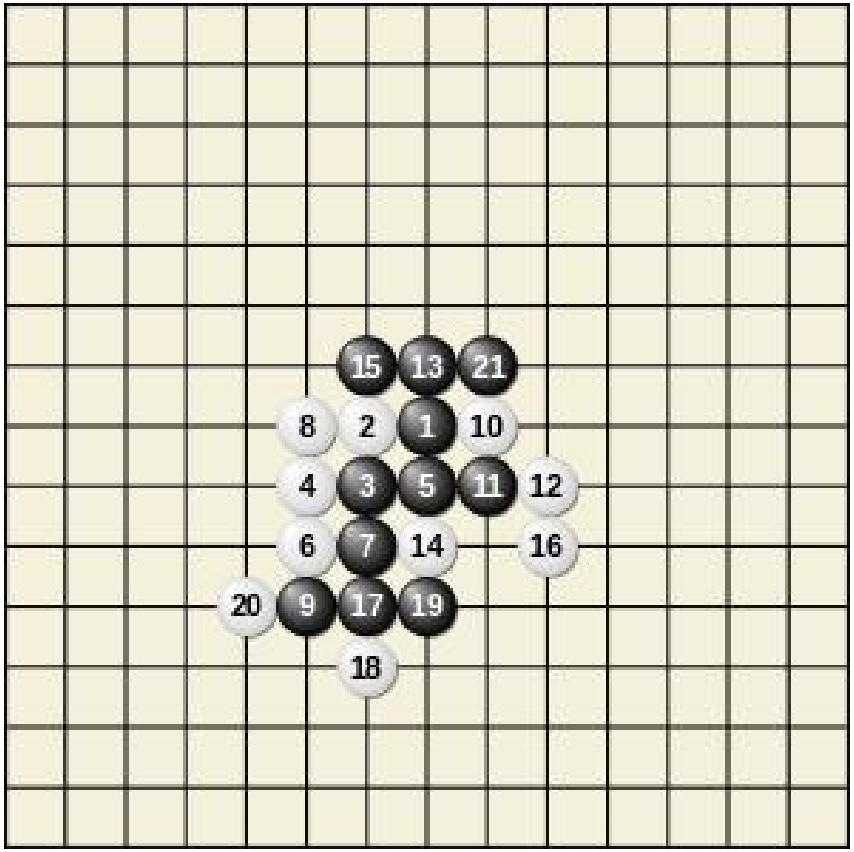
# 两人对抗游戏

- **井字棋 (Tic-tac-toe)**
  - 两个玩家，一个打圈(◯)，一个打叉(✗)，轮流在3乘3的格上打自己的符号，最先以横、直、斜连成一线则为胜



# 两人对抗游戏

- 五子棋 (Gomoku)
  - 先在横线、直线或斜对角线上形成5子连线者获胜



# 中国象棋

- 一盘棋平均走50步，总状态数约为10的161次方。
- 假设1毫微秒走一步，约需10的145次方年。
- 结论：不可能穷举。

# 游戏的问题定义

- s: 游戏中的特定状态（所有玩家都能获得）
  - $s_0$：初始状态
- Player(s): 在状态s时应该行动的玩家（此处讨论的游戏是玩家轮流行动的，如棋牌类）
- Action(s): 状态s时合法的动作集合
- Result(s, a)：状态s接受到动作a时产生的目标状态
- Terminal-Test(s)：测试s是否目标状态
- Utility(s,p)：效用函数，表示对玩家p而言，状态s的价值。

# 游戏中的搜索问题

- "无法预测的"对手 → 需要为对手的每个可能的落子选择一个动作
- 搜索空间过大（如五子棋、象棋）、时间限制 → 导致不可能找到目标，必须采用近似算法
- 对于两个玩家而言，它们的评估函数分值往往是相等的，但对两个玩家的意义正好相反。因此这类搜索问题被称为对抗搜索

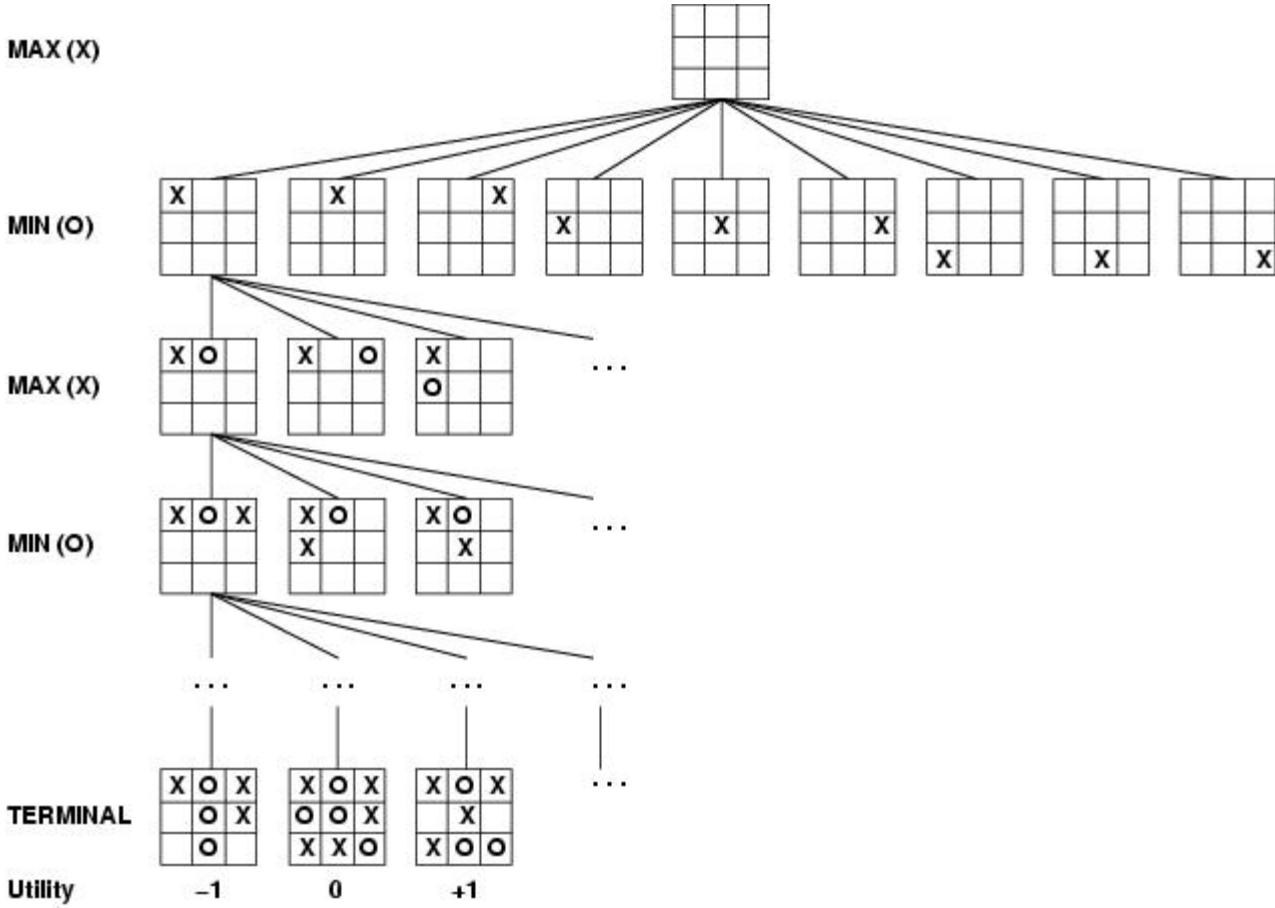
# 博弈树搜索（Game tree search）

tic-tac-toe游戏

player 1: MAX
player 2: MIN

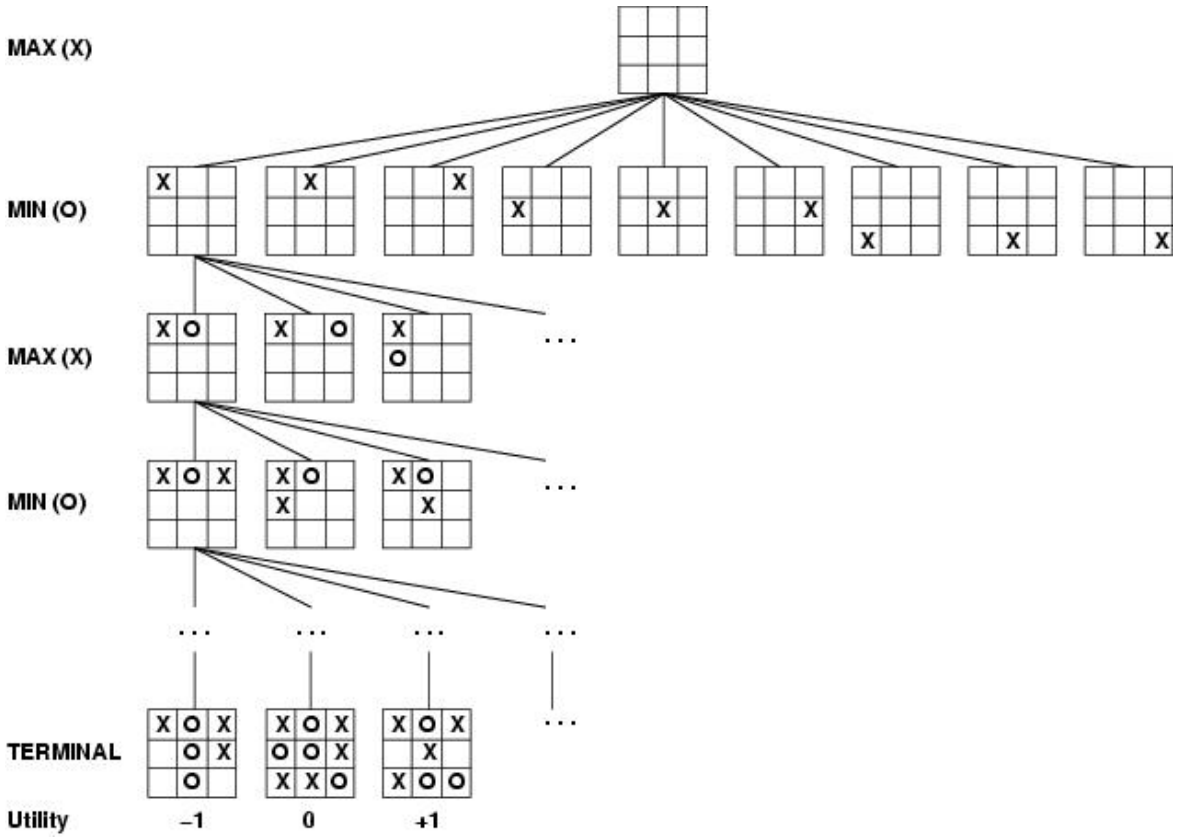
任一玩家达成
三连即获胜

搜索树的节点数量
大概为 9！=362880



# 博弈树

- 博弈问题
  - 双人
  - 一人一步
  - 双方信息完备
  - 零和
    - 一方得分为V
    - 另一方得分为-V

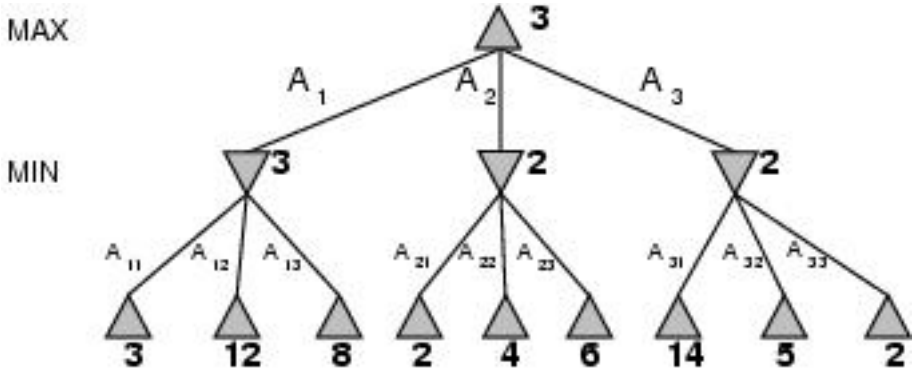


# Minimax算法

- 该算法的目标是寻找完美的策略，后续我们会介绍当时间或计算资源受限时，如何寻找不完美的策略
- 我们用MAX，MIN来表示两个玩家，他们交替行动
- 和传统搜索问题相比，每个玩家的行动策略必须考虑对手的行动

# Minimax算法中的搜索树

- 正放的三角形节点表示玩家MAX行动（第1,3层），倒放的三角形表示玩家MIN



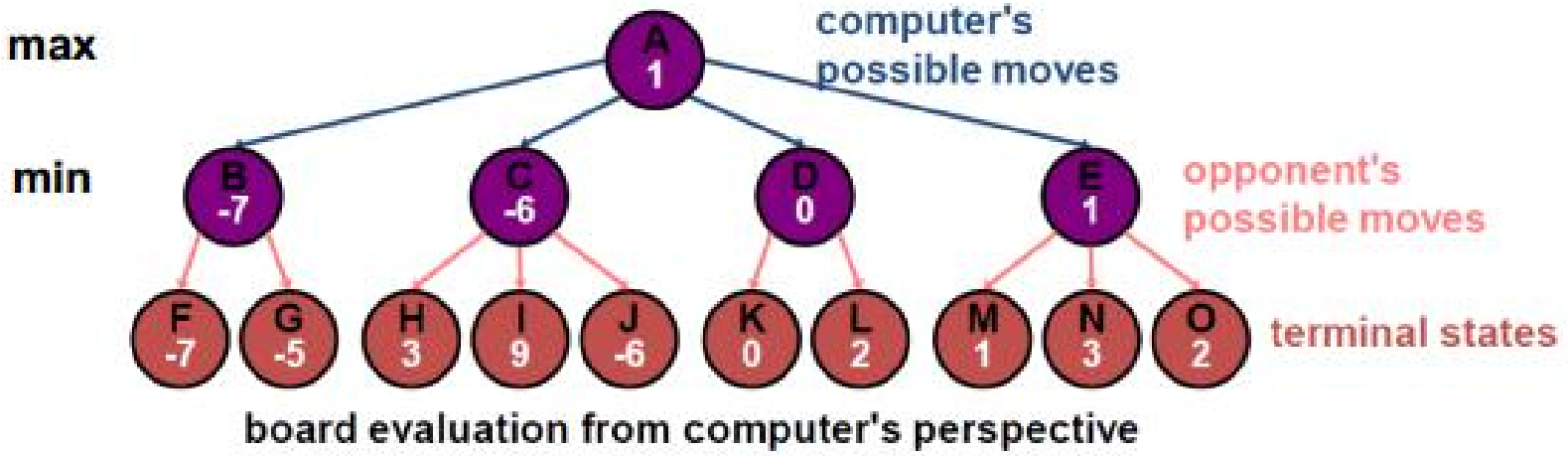

- 图中展示的是两个玩家各走一步的情形
- 叶节点标出的是MAX的效用函数值
- 假设每个节点都会采取对自己最优的行动方案
  注意，一个状态对MAX玩家的效用值越高，对MIN则越低，反之亦然
- 中间节点标出的是minimax 值

**Minimax**(*node*) =

- Utility(*node*) if Terminal-Test(*node*) is True
- $\max_{action}$ **Minimax**(Succ(*node, action*)) if *player* = MAX
- $\min_{action}$ **Minimax**(Succ(*node, action*)) if *player* = MIN

# 极小极大值原理

- 假设两位玩家都按照最佳策略行动
  - computer 假设在其行动以后，其对手会选择效用值最小的状态来移动
  - computer在选择其最佳行动方案时要同时考虑自己以及对手的最佳行动




从max的角度显示效用值

# Minimax 算法

- 采用递归方式来计算每个节点的minimax值
  - 从任意节点开始递归，直到叶节点为止(此时Terminal-test返回真值)
  - 将叶节点的minimax值逐层传递回去，得到中间节点的minimax值
- 若搜索树的深度为m，每个节点平均有b个子节点，则总的节点个数在 $b^m$ 量级
  - 对于实际的游戏，这样的时间成本太高，无法实用
- Minimax算法仅是一个理论上的分析，并且是后续高效算法的基础

# Minimax 算法

```
function MINIMAX-DECISION(state) returns an action
    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
```

# 深度受限的Minimax算法

```
01 function minimax(node, depth, maximizingPlayer)
02     if depth = 0 or node is a terminal node
03         return the heuristic value of node

04     if maximizingPlayer
05         bestValue := −∞
06         for each child of node
07             v := minimax(child, depth − 1, FALSE)
08             bestValue := max(bestValue, v)
09         return bestValue

10     else    (* minimizing player *)
11         bestValue := +∞
12         for each child of node
13             v := minimax(child, depth − 1, TRUE)
14             bestValue := min(bestValue, v)
15         return bestValue
```

# Minimax 的性能

完整性？？ Yes,仅当博弈树是有限时

最优性？？ Yes,遇到一个聪明的对手。Otherwise??

时间复杂度？？ $O(b^m)$

空间复杂度？？ O(bm) (深度优先搜索)

For chess, b≈35, m≈100 for “reasonable" games

→寻找精确解时完全不可行的

But do we need to explore every path?

# 最优策略

- 最优解是导致取胜的终止状态的一系列招数
- 但，在游戏中MIN还有发言权
- 在对手不犯错误时，最优策略能够导致至少不比任何其它策略差的结果。

# α - β 剪枝

- 若与一个聪明的对手对弈，则博弈树上的一些分枝绝不会发生

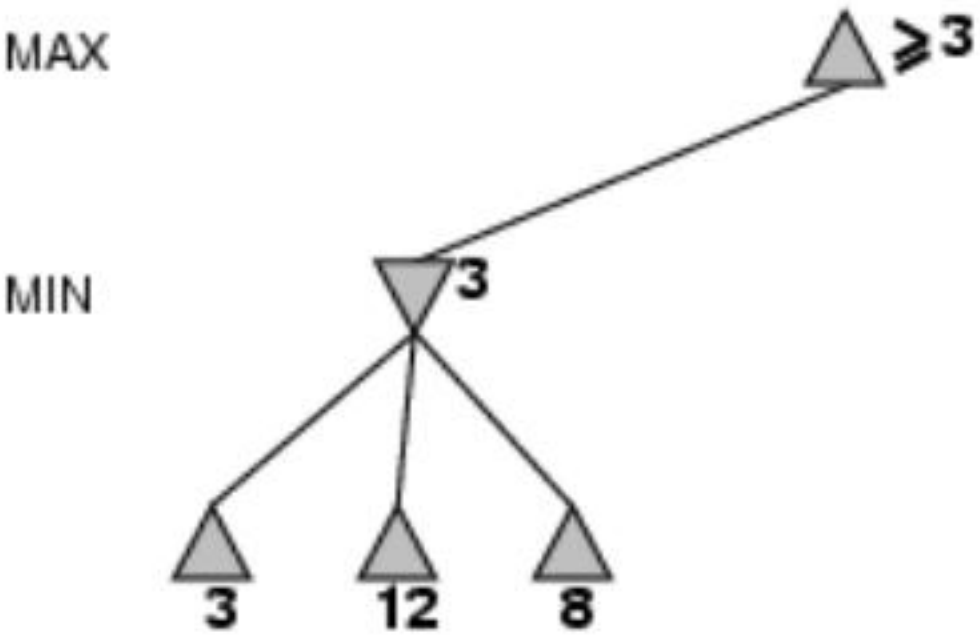
- "If you have an idea that is surely bad, don't take the time to see how truly awful it is."
  -- Pat Winston

- 剪枝能消除搜索树的很大一部分分枝

# α-β pruning

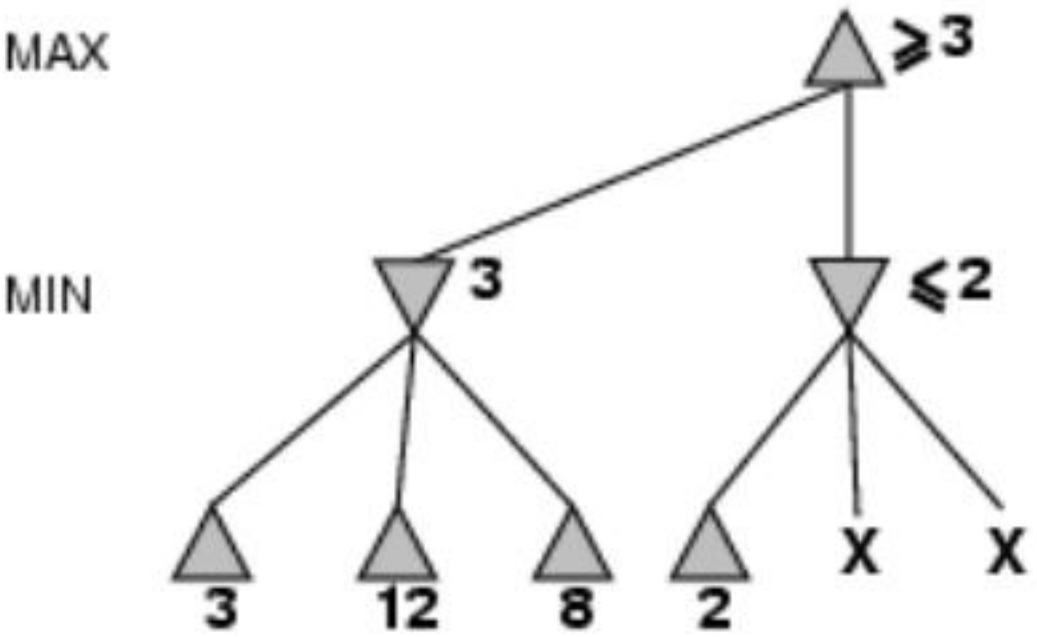
- 动机：minimax搜索须遍历指数级的结点，效率太低
- pruning：剪枝掉一部分节点（子树），保证结果依然是正确的（得到最佳的minimax值）
  - 剪枝意味着程序不去处理、展开这些节点
- α-β pruning 大约剪掉一半的节点

# α-β pruning 例子



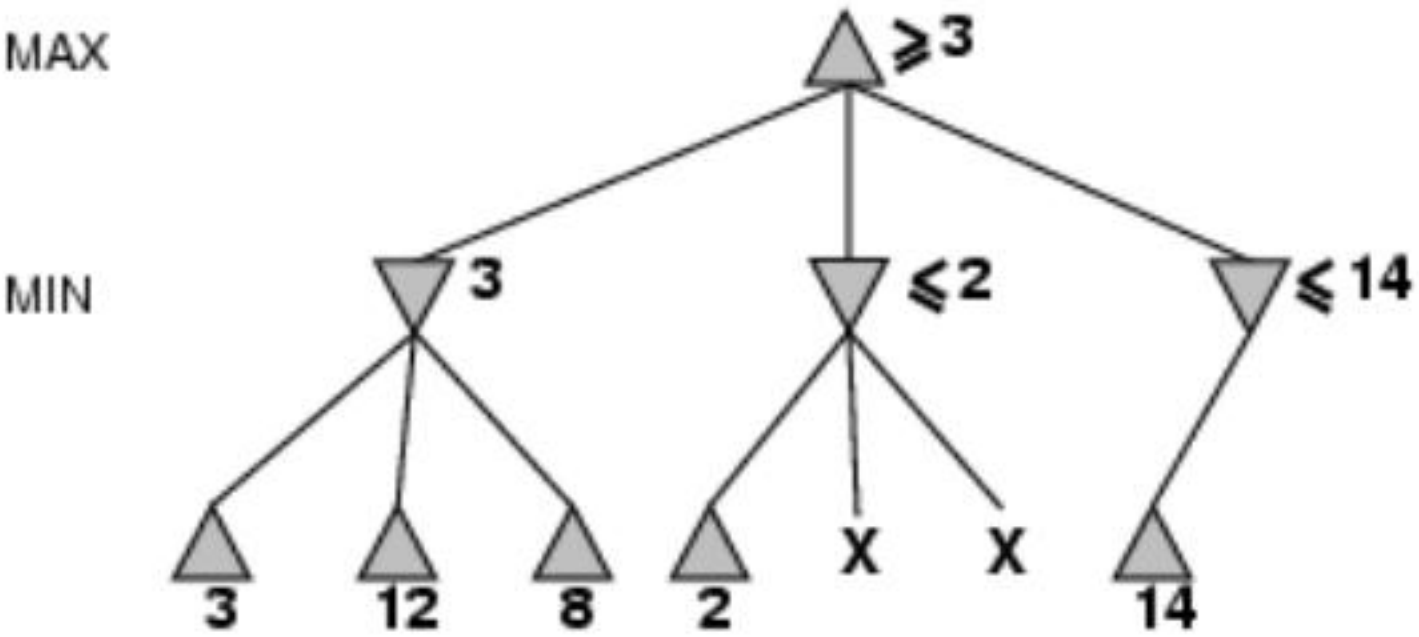

对于根节点，已经得到其一个子节点的值为3，那么作为一个MAX节点，根节点的值必然 >=3

# α-β pruning 例子



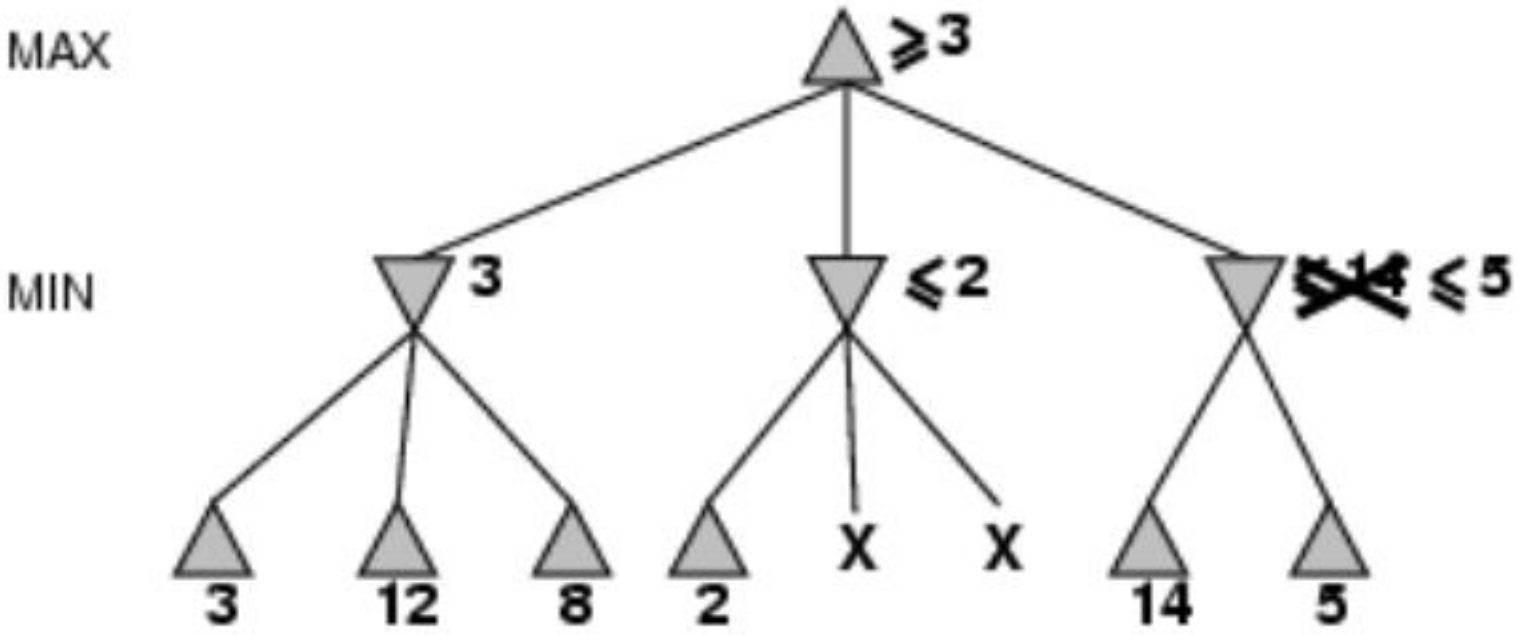

对于根节点的第二个子节点，已经得到其一个子节点的值为2，那么作为一个MIN节点，它的值必然 <=2（那么，我们就不需要计算两个X节点的值了，它们对根节点的值不会有影响）

# α-β pruning 例子



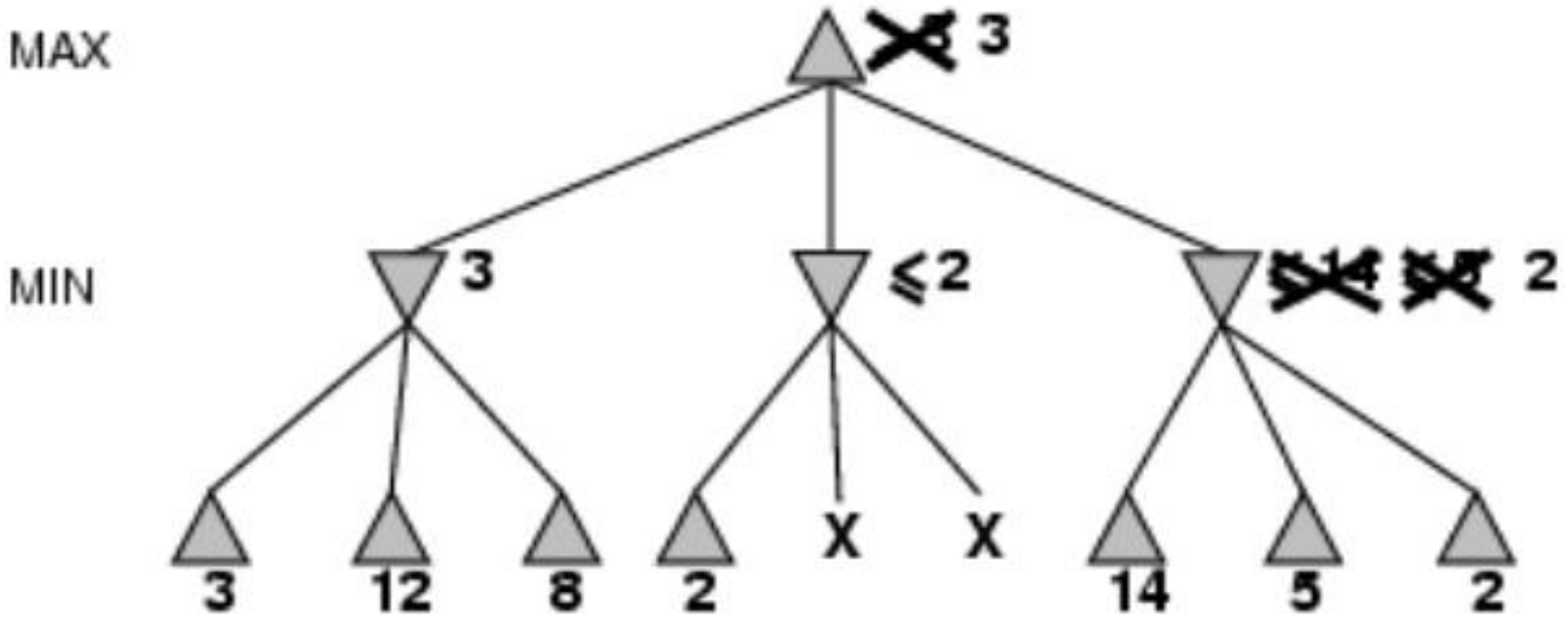

对于根节点的第三个子节点，已经得到其一个子节点的值为14，那么作为一个MIN节点，它的值必然 <=14

# α-β pruning 例子




对于根节点的第三个子节点，继续计算其子节点的值，发现它的值被更新为<=5

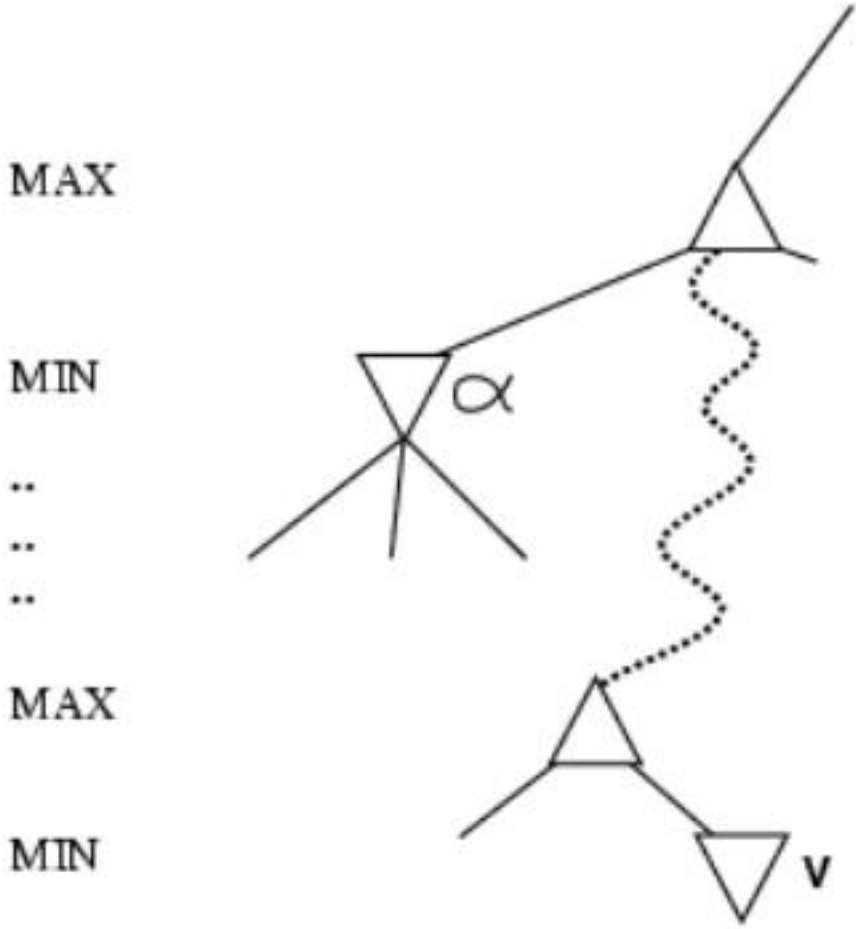
# α-β pruning 例子




对于根节点的第三个子节点，继续计算其子节点的值，发现它的值被更新为 2

# 为什么叫 α-β?

- α 是对MAX而言当前找到的最优选择的值
- 若 $v$ 比 α 差, MAX将抛弃 $v$，以及从v展开的子树
- 类似地， β 是对MIN而言的最优值



# α-β剪枝技术

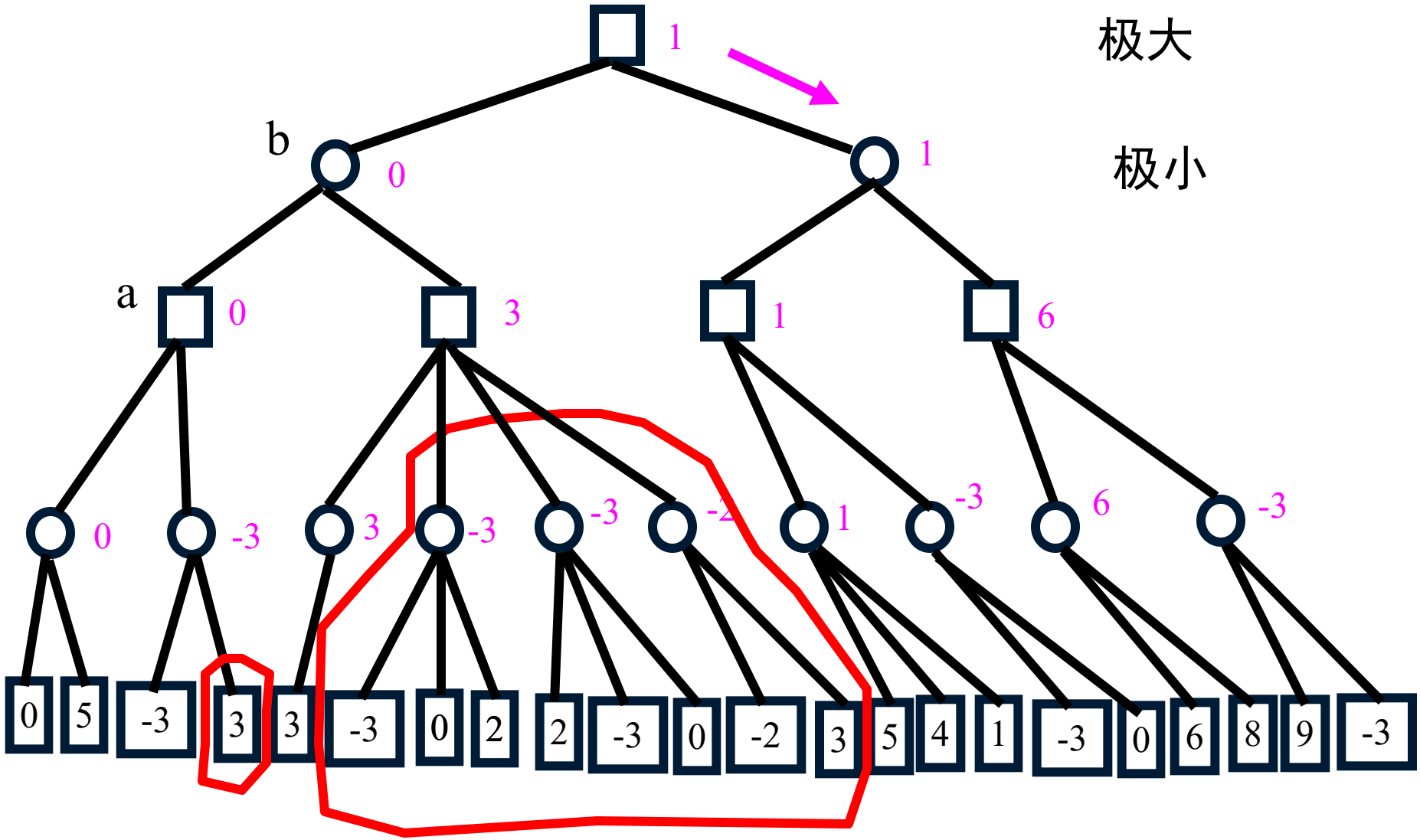
- 对于一个MAX节点来说，它取值称为α值
- 对于一个MIN节点来说，它取值称为β值

- β剪枝：任何MAX节点x的α值如果不能降低其父节点的β值，则对节点x以下的分枝可停止搜索。
- α剪枝：任何MIN节点x的β值如果不能升高其父节点的α值，则对节点x以下的分枝可停止搜索。

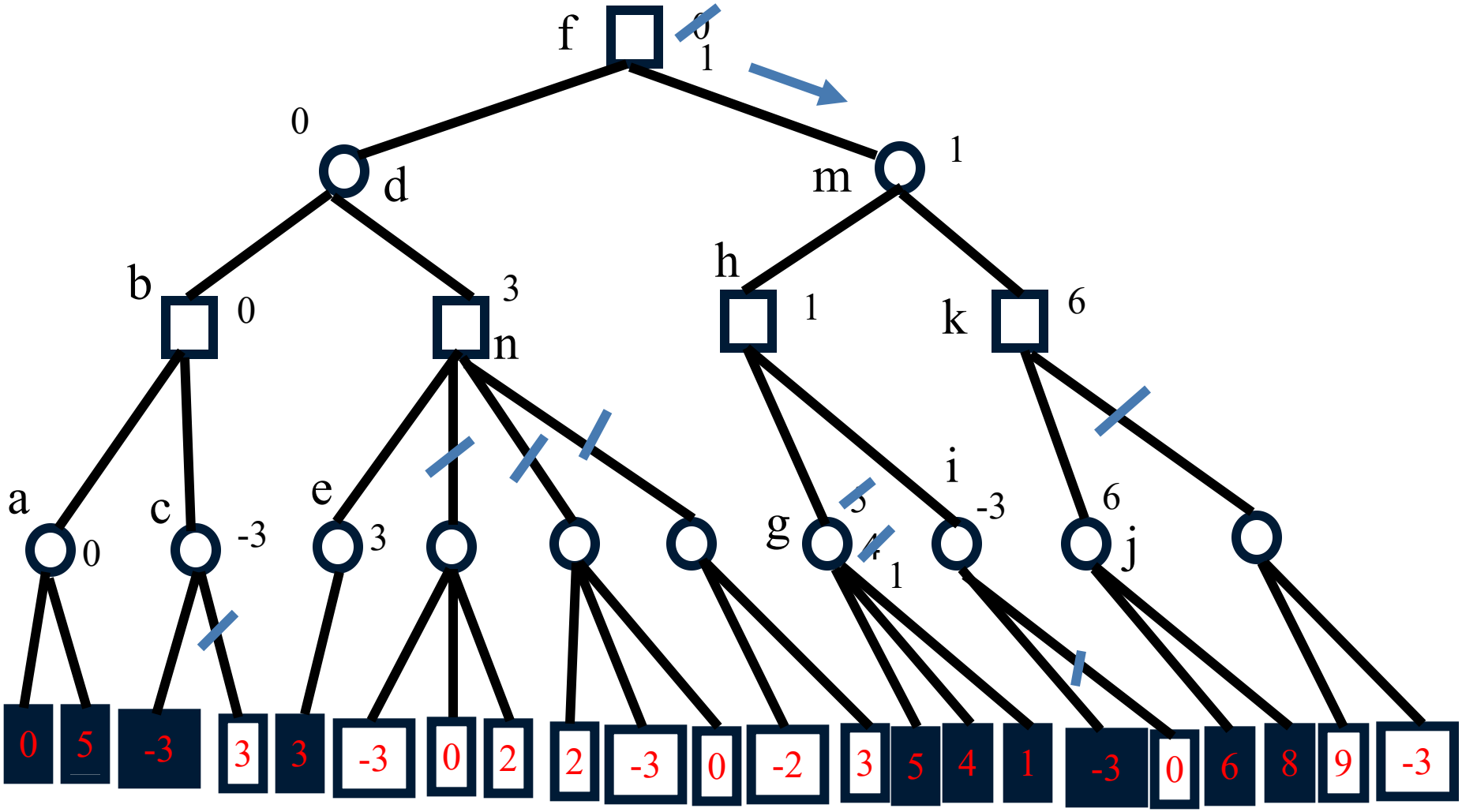
# α-β剪枝

- 极大节点的下界为α。
- 极小节点的上界为β。
- 剪枝的条件：
  - 后辈节点的β值≤祖先节点的α值时， α剪枝
  - 后辈节点的α 值≥祖先节点的β值时， β剪枝
- 简记为：
  - 极小≤极大，剪枝
  - 极大≥极小，剪枝

# 极小极大过程



# α-β剪枝过程



# α-β剪枝算法

**function** ALPHA-BETA-SEARCH(*state*) **returns** *an action*
  **inputs**: *state*, current state in game

  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  **return** the *action* in SUCCESSORS(*state*) with value $v$

---

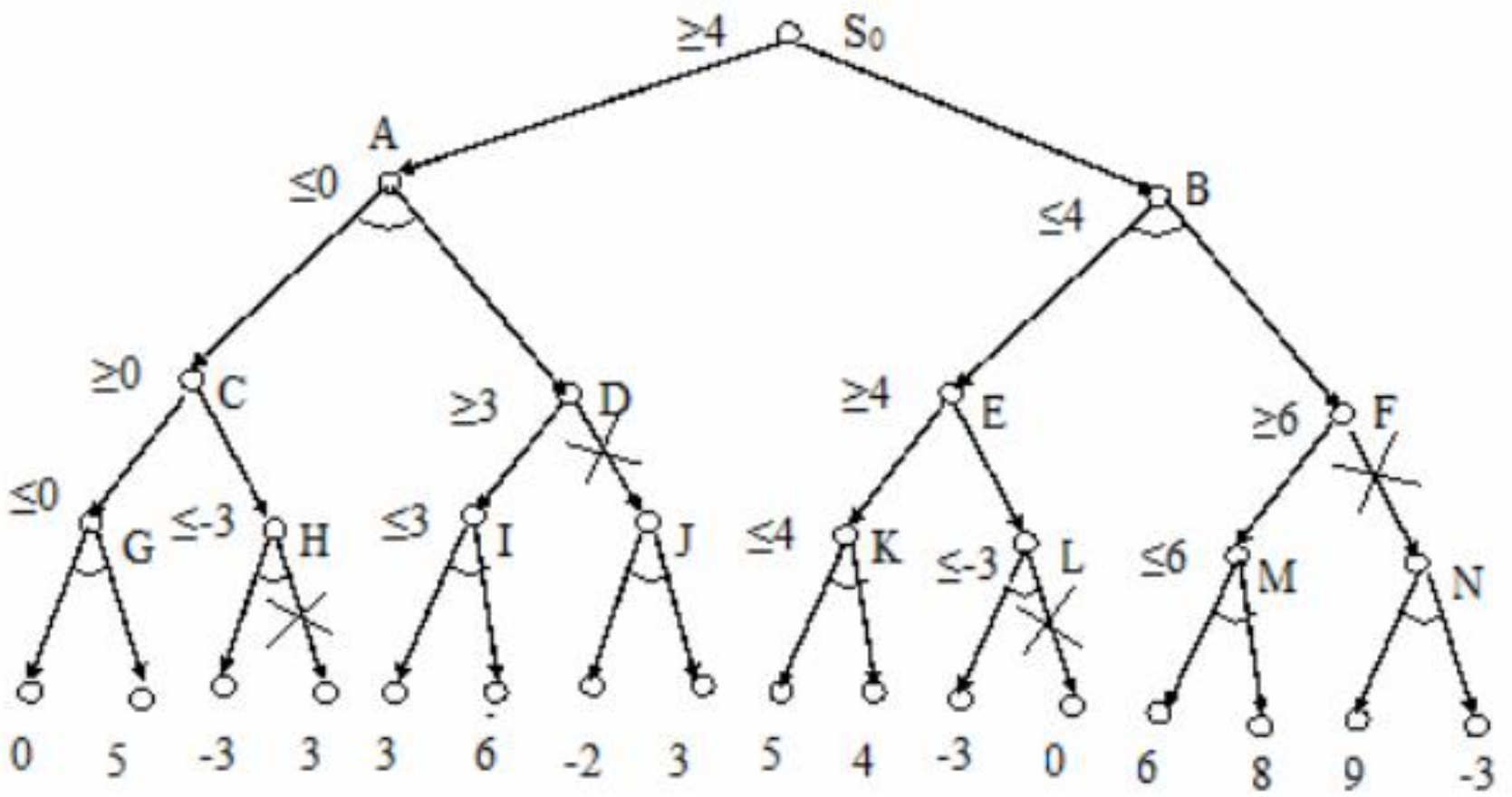
**function** MAX-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  **inputs**: *state*, current state in game
    $\alpha$, the value of the best alternative for MAX along the path to *state*
    $\beta$, the value of the best alternative for MIN along the path to *state*

  **if** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  **for** $a, s$ in SUCCESSORS(*state*) **do**
    $v \leftarrow$ MAX($v$, MIN-VALUE($s, \alpha, \beta$))
    **if** $v \geq \beta$ **then return** $v$
    $\alpha \leftarrow$ MAX($\alpha, v$)
  **return** $v$

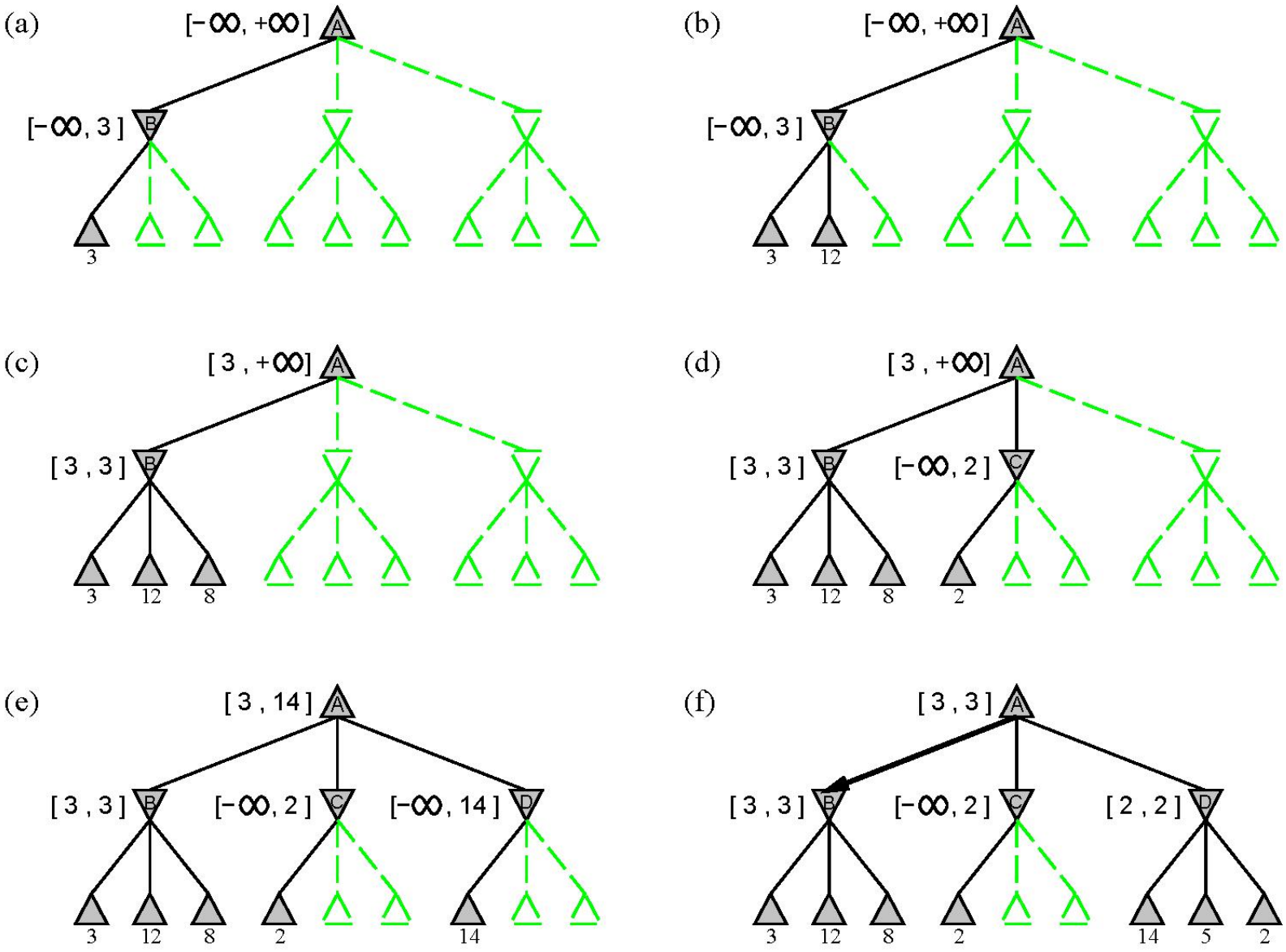
# α-β剪枝算法（续）

```
function MIN-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(s, α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

# α-β剪枝案例



# 例子


(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

# α-β搜索的效率

- 效率很大程度上取决于检查后继的顺序; 所以尝试检查可能较好的后继是值得的。极小极大值搜索的时间复杂度是$O(b^m)$
- 最坏情况:
  - 没有分枝需要修剪
  - 相比于穷举搜索没有改进
- 最好情况:
  - 每个玩家的最佳行动都先被检查
- 在实践中，性能接近于最好情况，而不是最坏情况，但要依实际情况而定

# α–β的性能

剪枝不影响最终结果

好的行动顺序能提高剪枝的效率

With “perfect ordering," time complexity = $O(b^{d/2})$
doubles solvable depth

不幸的是, $35^{50}$ 也是有可能的

# 处理搜索树中的重复状态

- 在游戏中重复状态的频繁出现往往是因为调换—导致同样棋局的不同行棋序列的排列
  - 例如，[$a_1$, $b_1$, $a_2$, $b_2$]和[$a_2$, $b_2$, $a_1$, $b_1$]都结束于同样的棋局。
- 解决办法：第一次遇见某棋局时将对它的评价存储在哈希表中（该哈希表称为调换表）。

# 计算资源的限制

- 实际游戏中要求每个玩家必须在规定时间内行动
- 假设我们每一步允许花费100秒，每秒平均探索$10^4$个节点
  - 每一步可以（最多）探索$10^6$个节点
- 国际象棋，一步约有35个分支，搜索到叶子节点（即满足Terminal_Test为真）的深度约为100，意味着 $35^{100}=10^{154}$ 节点

应如何修改Minimax算法?

**Minimax**(*node*) =

- Utility(*node*) if Terminal-Test(*node*) is True
- $\max_{action}$ **Minimax**(Succ(*node, action*)) if *player* = MAX
- $\min_{action}$ **Minimax**(Succ(*node, action*)) if *player* = MIN

# 启发式 Minmax (H-Minimax)

- 通过使用截断测试 (cutoff test)，在尚未到达叶子结点时，即停止搜索，并返回一个启发式的Minmax值

**H-Minimax**(*node, d*) =

- Eval(*node*) if Cutoff-Test(*node, d)* is True
- $\max_{action}$ **H-Minimax**(Succ(*node, action*), d + 1) if *player* = MAX
- $\min_{action}$ **H-Minimax**(Succ(*node, action*), d + 1) if *player* = MIN

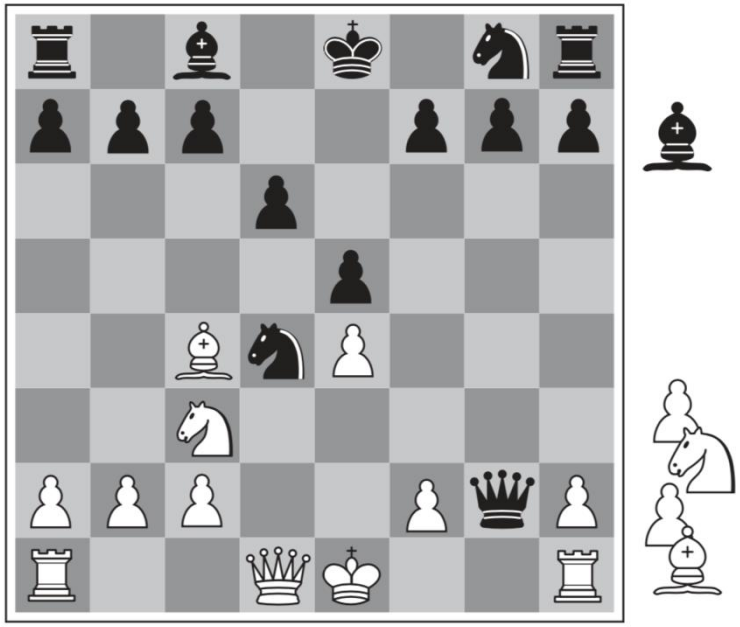
d: 当前搜索深度
Eval: 评估函数 (Evaluation function)

# 评估函数 Eval

- 评估函数给出了在当前状态s下，玩家p能从游戏中期望获得的效用值Utility(s, p)的一个估计值
- 评估函数的设计原则

1. 用Eval对所有结束状态排序，可以得到与Utility一样的排序结果
2. 计算量小（我们之所以用Eval就是减少计算量）
3. 对于非结束状态，Eval给出的估计值与从该状态出发赢得游戏的概率强相关

黑棋比白棋多了1马和2兵



(a) White to move

# 评估函数 Eval

- 一种常用的方案是各种特征的线性加权组合

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

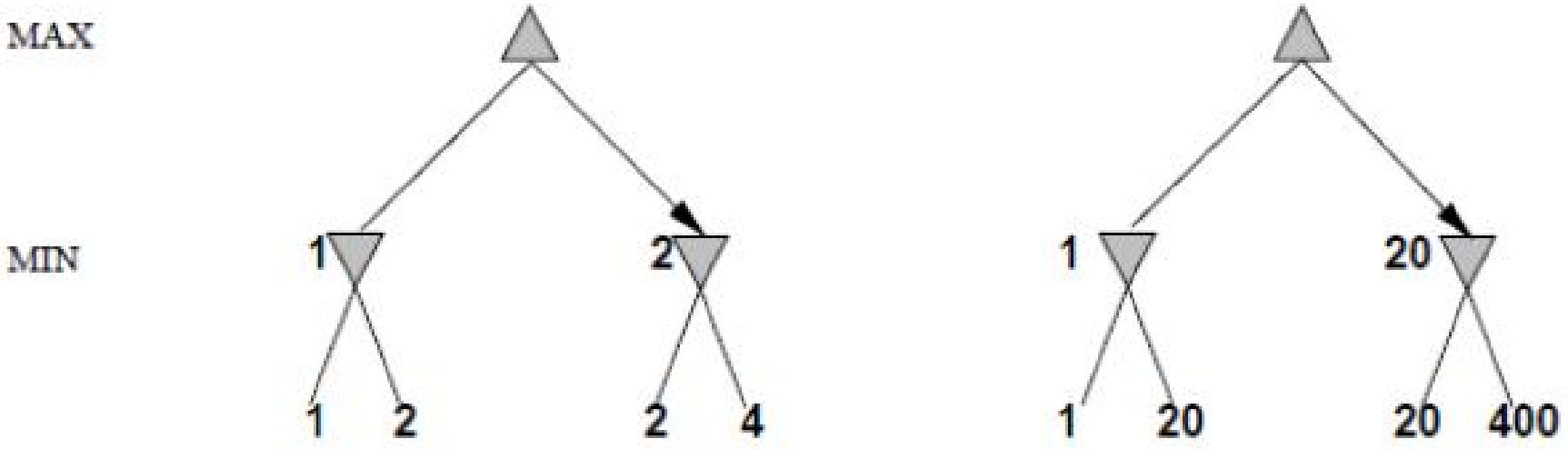
- 对于象棋
  - $w_k$ 可以是每个棋子的物料价值 **(material value)**
    - pawn (兵) = 1, knight (马) = 3, rook (车) = 5, queen (皇后) = 9
  - $f_k(s)$ 表示该棋子（在状态s下的）权重
    - 比如 bishop(象）在残局 中更重要

- 评估函数可以是经验设计，也可以通过机器学习的方式得到

# More on 评价函数

- 评价函数评估当前局面配置的好坏
- 一个线性的评价函数是关于特征$f_1, f_2, f_3$的加权和
  - More important features get more weight
- 对弈的质量直接依赖于评价函数的质量
- 为了构建一个好的评价函数，必须:
  - 利用行业知识提取好的特征
  - 选择或学习更好的权重

# 题外话：精确的评价函数并不重要




Behavior is preserved under any monotonic（单调的） transformation of EVAL

# 截断搜索 Cut-off Search

- 每一步都搜索到一个固定的深度
  - 这个深度被称为地平线（horizon）
  - 截断搜索能看到地平线内会发生的威胁
- 地平线效应（horizon effect）
  - 看不到地平线外的威胁
  - 在8层的搜索中(即搜索4个回合)，就可能得不到在第5步有杀棋的信息

# 地平线效应

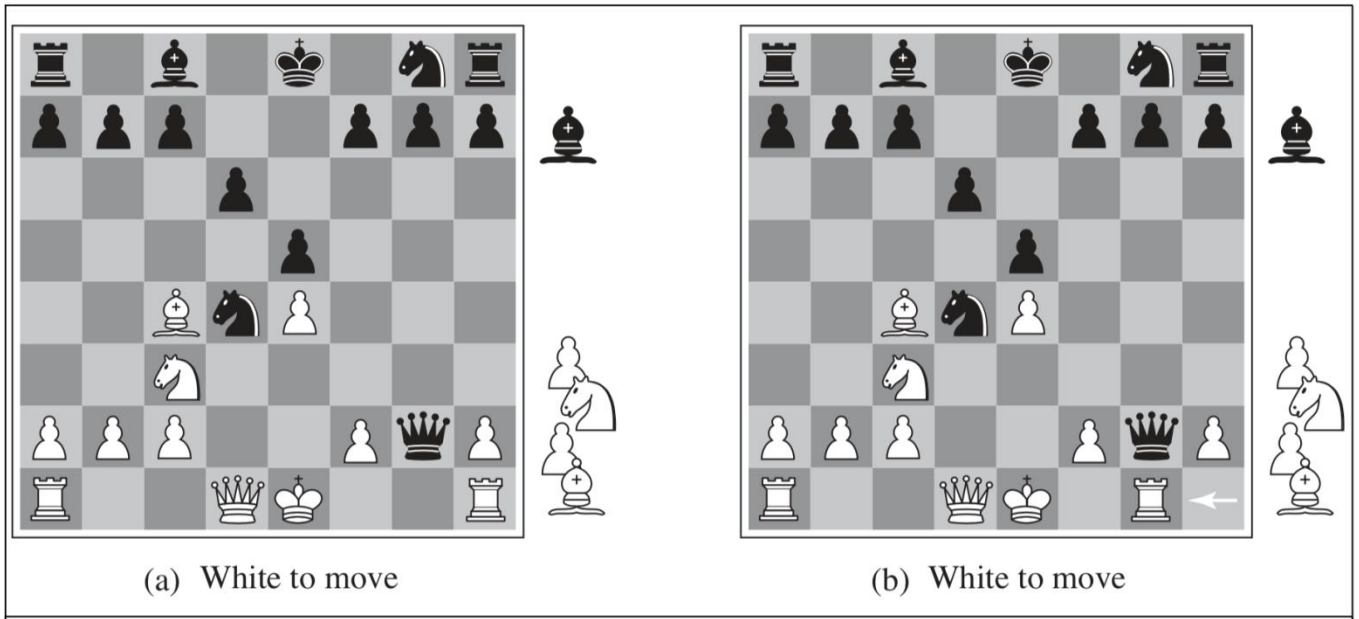



**Figure 5.8** Two chess positions that differ only in the position of the rook at lower right. In (a), Black has an advantage of a knight and two pawns, which should be enough to win the game. In (b), White will capture the queen, giving it an advantage that should be strong enough to win.

状态(a)和状态(b)的区别在于白方车的位置。
注意到状态(b)是一个很不稳定的状态（白车会抓住黑皇后）

# 静态搜索 Quiescence Search

- 静态（quiescent state）：评估值不发生剧烈变化
- 到达地平线后，检查状态是否quiescent
  - 如果是，则评估并返回
  - 如果不是，继续搜索

# 截断搜索

- 用if CUTOFF-TEST(state, depth) then return EVAL(state)代替α–β算法中TERMINAL-TEST所在的两行，或
- 使用迭代搜索：当时间用完时，程序就返回目前完成最深的搜索所选择的招数。
  - 由于评价函数的近似性，上述方法可能导致错误。

# 截断搜索（续）

- 需要更为复杂的截断测试，评价函数应该只用于那些静止的棋局（静态棋局）。
- 非静止的棋局可以进一步扩展直到静止的棋局，这种额外的搜索称为静态搜索。
- 单步延伸是避免地平线效应的一种策略

这样做可能超过深度限制，但由于单步延伸很少，不会增加太多开销。

# 国际象棋游戏系统

| 算法 | 搜索深度 | 人类水平 |
| --- | --- | --- |
| H-Minimax 加上 静态搜索 | 双方共走4步 | 新手 |
| 加上 alpha-beta 剪枝 | 双方共走10步 | 高手 |
| IBM 深蓝 | 双方共走14步 | 卡斯帕罗夫（棋王） |

# 对待有限的时间

- 在实际游戏中，通常对每一步有时间限制T
- 我们如何考虑这个问题?

  – 所以，我们可以设置一个保守的深度有限，以保证在T时间内决定一次行动
  –但是，搜索可能提前结束，更多搜索的机会被浪费了

# 对待有限的时间

- 在实践中, 迭代深入深度优先搜索(IDS) 被很好地使用
  – 运行alpha-beta search 以深度限制逐渐增加的方式
  – 当时间T快运行完时，返回最后一次完整的α–β 搜索的结果 (i.e., the deepest search that was completed)

# 现今一些确定性的游戏

Chess(国际象棋）：Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply（层、厚度）.

– Kasparov lost the match 2 wins to 3 wins and 1 tie
– Deep Blue played by “brute force” (i.e., raw power from computer speed and memory); it used relatively little that is similar to human intuition and cleverness
– Used minimax, alpha-beta, sophisticated heuristics

# 现今一些确定性的游戏

Checkers（西洋跳棋）：Chinook, the World Man-Machine Checkers Champion.

- Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994.
- In 2007, checkers was solved: perfect play leads to a draw.

Chinook cannot ever lose

使用了一个提前计算好的存有443,748,401,247个不多于8个棋子的棋局数据库，使它的残局(endgame)走棋没有缺陷

50 machines working in parallel on the problem

# 现今一些确定性的游戏

Go（围棋）：2016年以前，人类冠军拒绝与计算机比赛，因为计算机是个小学生棋手。In go, b > 300（棋盘为 19x19），所以大多数程序使用基于模式识别的方法来提供一个貌似可行的解。Go has became a new benchmark for Artificial Intelligence (人工智能新的试金石)

# AlphaGo: 第一次打败人类in 19x19 Go

- Google DeepMind

  –深度神经网络、监督学习、强化学习、Monte-Carlo simulation

2016年3月， AlphaGo vs 围棋世界冠军、职业九段棋手李世石，4比1获胜

# 人类唯一的一次胜利

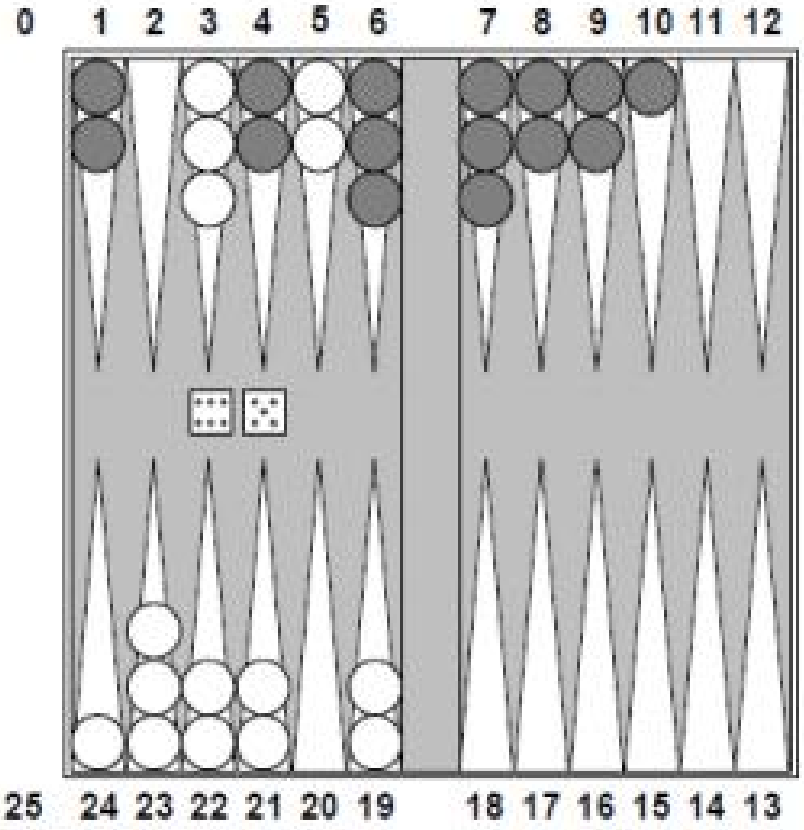

新浪围棋棋谱
谷歌Alph 九段 0
李世石 九段 0
谷歌李世石世界人机大战4局
* 日期 ： 2016-03-13
* 地点 ： 首尔四季酒店
* 用时 ： 120
* 读秒 ： 60秒 3次
* 贴目 ： 7.5
* 结果 ： 白中盘胜
* 猜下一手
黑 白 黑白 不用暗示
* 解说
180好手出棋，AlphaGo屏幕弹出了认输的提示框。本局黑23碰，白26略缓，此后一直被动，黑69撑得过满。李世石凭借用生命算出的白78神之一手，逼迫谷歌AlphaGo阵脚大乱，终于扳回一城。15日12时双方进行五番棋第五局较量。欢迎关注新浪围棋微信订阅号“棋道经纬”sinachess

# AlphaGo被逼出了bug

2017年5月， AlphaGo vs 排名世界第一的柯洁
以3比0的总比分获胜

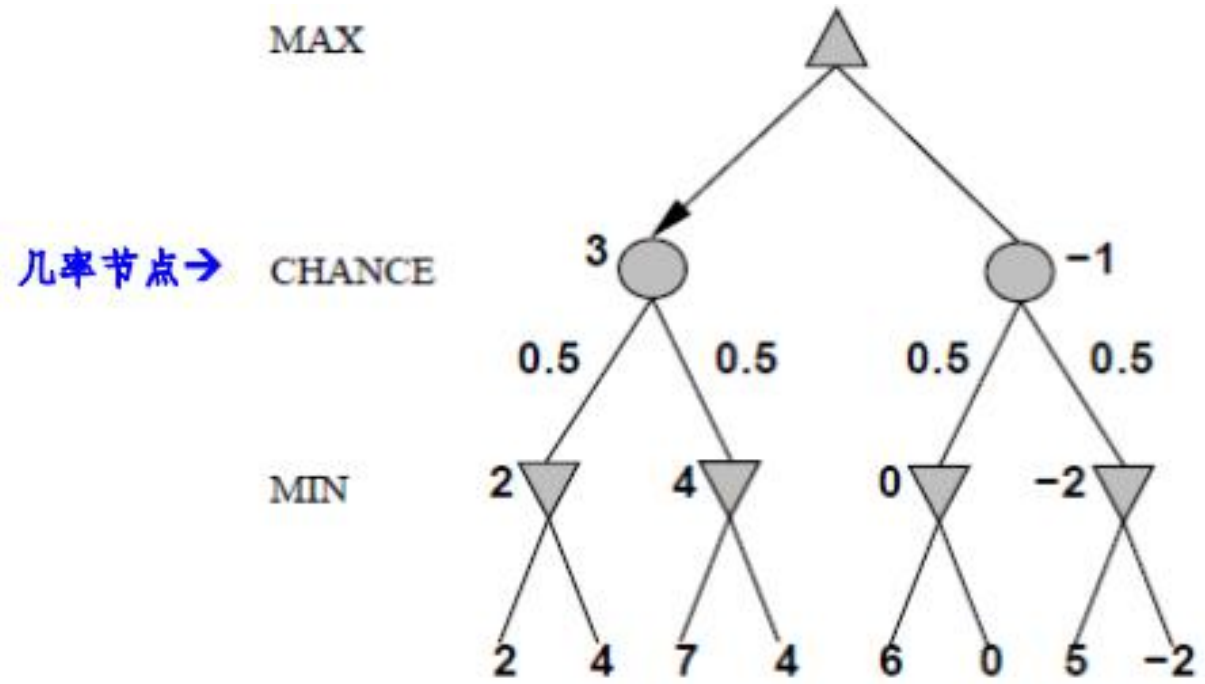

# 包含几率因素的游戏

- 西洋双陆棋



# 非确定性游戏概述

在非确定性的游戏中，几率因素是由掷骰子，抽牌等引起的。 抛硬币游戏的简化示例：

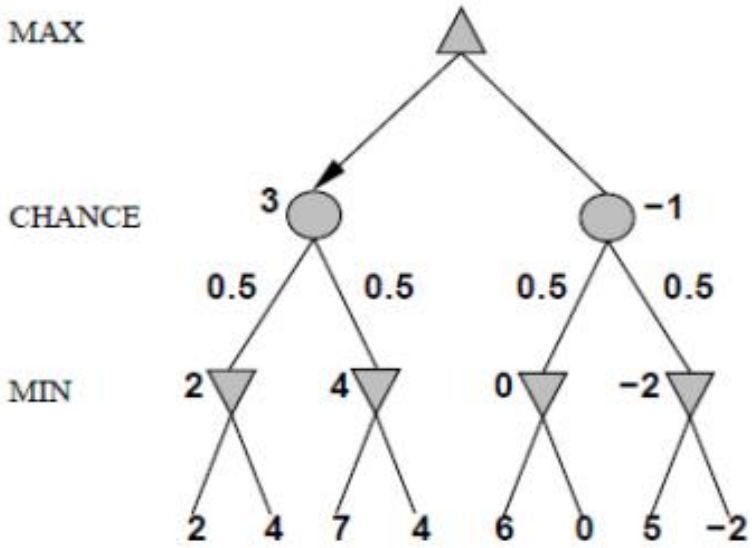


# 非确定性游戏概述

- 权重取决于事件发生的概率

- 将极小极大值推广为期望极小极大值

- Choose move with highest expected value



# 期望效用最大化

- 为什么我们要计算平均效用值? 为什么不计算极小极大值?

- 期望效用最大化原则: 一个智能体基于其给定的知识库，会根据期望效用最大化来选择行动方式

- 决策的一般性原则
- 经常作为理性的定义
- 我们会在本课程中反复看到该观点!

# 期望极小极大值算法

EXPECTIMINIMAX 类似于MINIMAX，多考虑一个几率节点

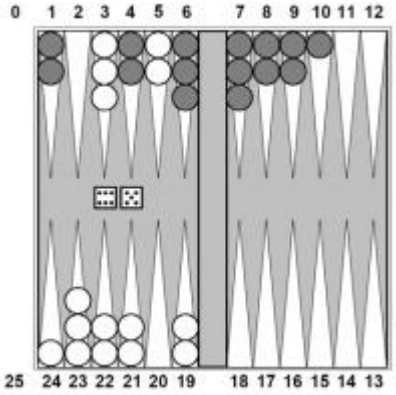
if state is a Max node then
    return the highest EXPECTIMINIMAX-VALUE of SUCCESSORS(state)

if state is a Min node then
    return the lowest EXPECTIMINIMAX-VALUE of SUCCESSORS(state)

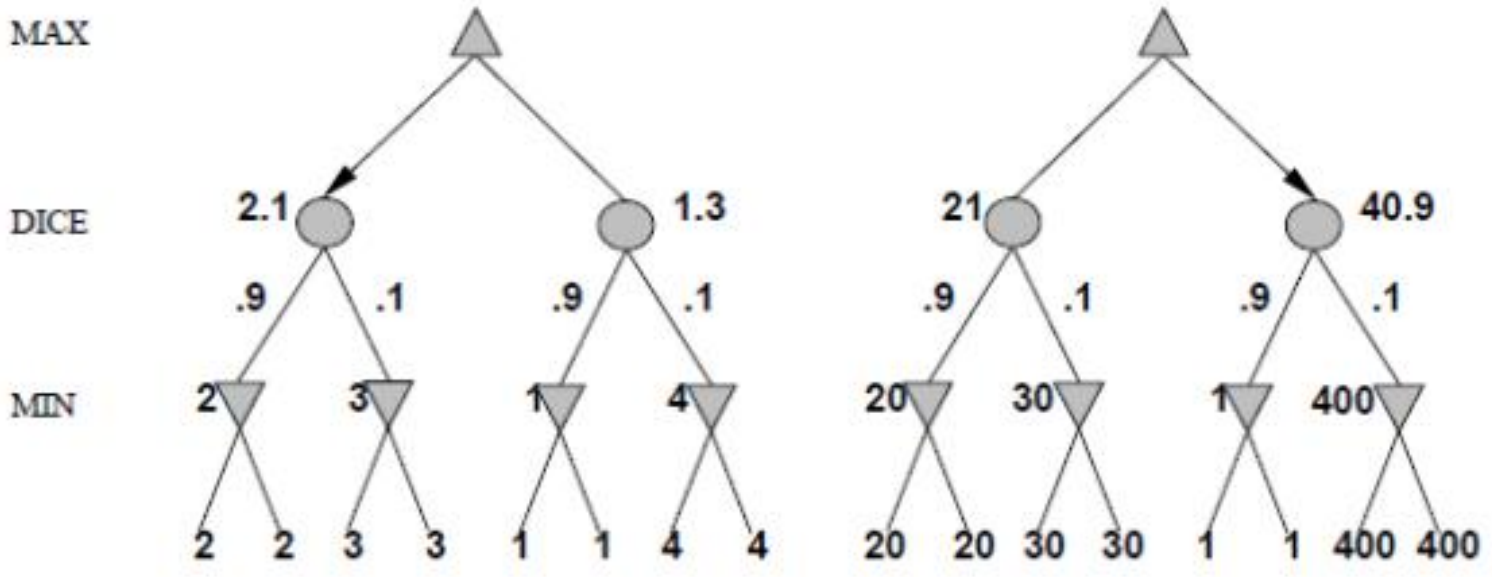
if state is a chance node then
    return average of EXPECTIMINIMAX-VALUE of SUCCESSORS(state)

# 随机的Two-Player

- 掷骰子增加分枝b: 两个骰子有21种可能的掷法
  - 西洋双陆棋≈20种合法行动
  - Depth 4 = 20 x $(21 \times 20)^3$=1.2 x $10^9$
- 当深度增加时，到达指定节点的概率会收窄
  - 此时再向前搜索的价值会减少
  - 所以限定搜索深度是OK的
  - 但是剪枝不太可能实现...
- TDGammon uses depth-2 search + very good eval function + reinforcement learning: world-champion level play



# 题外话：精确的评价函数的重要性




Behaviour is preserved only by positive linear transformation of EVAL

Hence EVAL should be proportional to the expected payoff

评价函数应该是棋局的期望效用值的正线性变换

# 不完整信息的游戏

E.g., card games, where opponent's initial cards are unknown
Typically we can calculate a probability for each possible deal
Seems just like having one big dice roll at the beginning of the Game

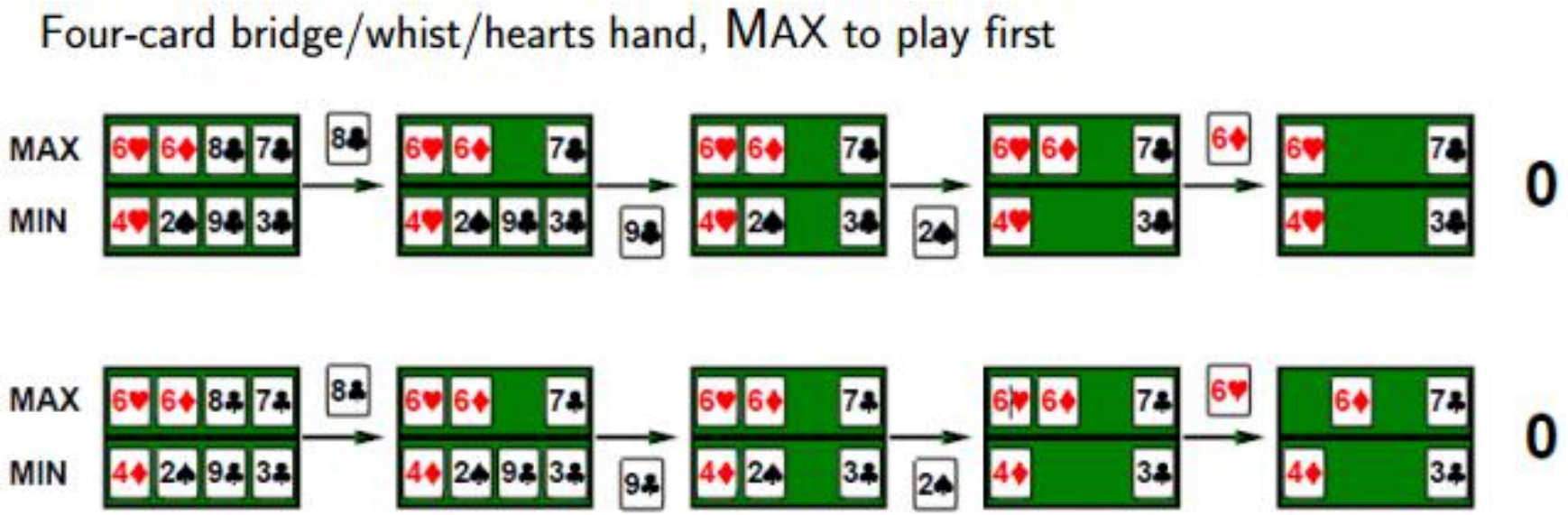
Idea: compute the minimax value of each action in each deal, then choose the action with highest expected value over all Deals

在评价一个有未知牌的给定行动过程时，首先计算出每副可能牌的出牌行动的极小极大值，然后再用每副牌的概率来计算得到对所有发牌情况的期望值。

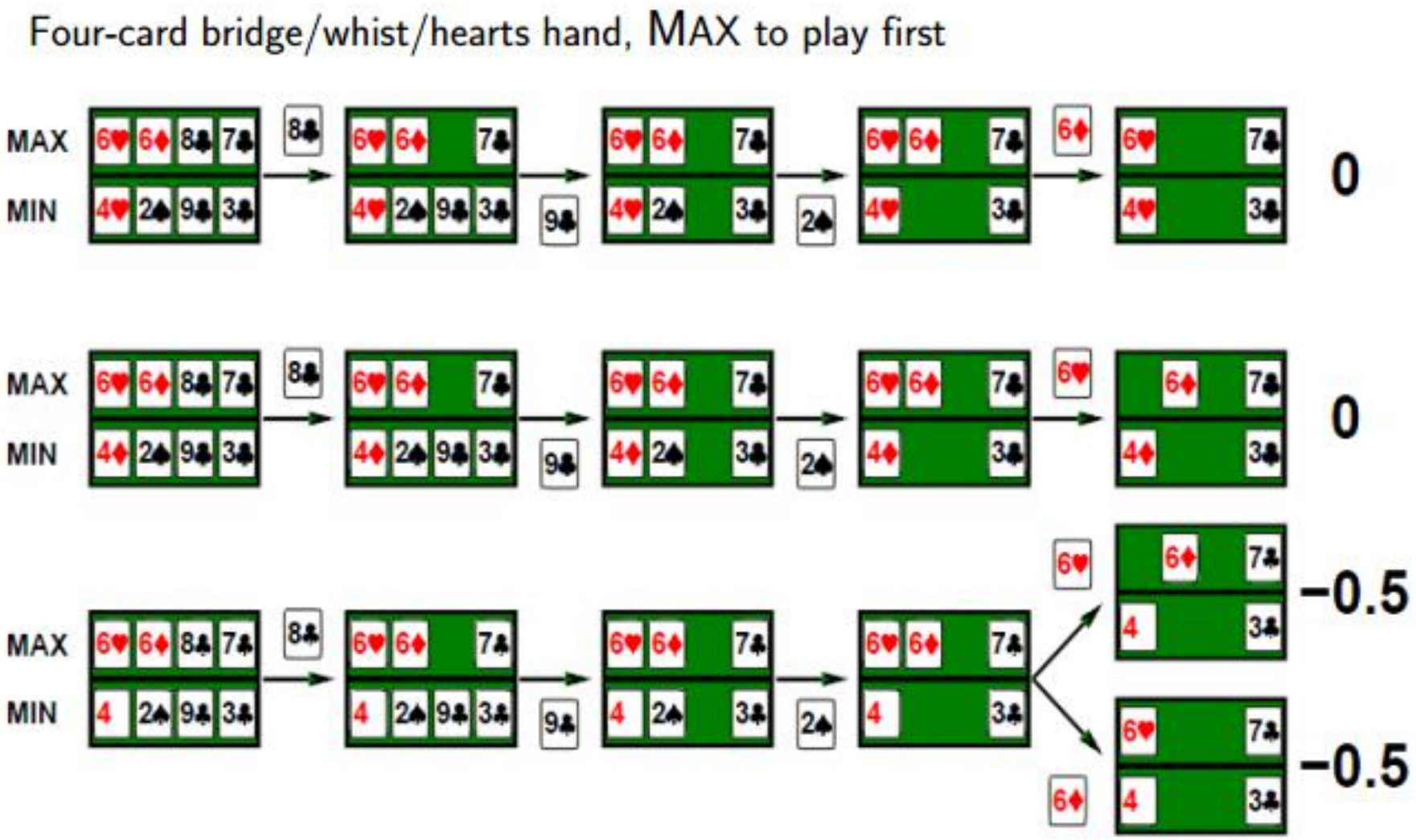
# Example

Four-card bridge/whist/hearts hand, MAX to play first

# Example


Four-card bridge/whist/hearts hand, MAX to play first
MAX
MIN
0
MAX
MIN
0

# Example


Four-card bridge/whist/hearts hand, MAX to play first
MAX
MIN
0
0
−0.5
−0.5

# 牌类游戏

- 考虑未知牌的所有可能应对策略 假设应对策略s发生的概率是p(s)

$$\arg\max_{a} \sum P(s)MINIMAX(RESULT(s,a))$$

- 若使用蒙特卡洛近似，随机采样N个样本，设组合s在样本中出现的概率与p(s)成正比：

$$\arg\max_{a} \frac{1}{N} \sum_{i=1}^{n} P(s)MINIMAX(RESULT(s,a))$$

# 合理的分析

*所以从直觉上说用所有可能状态的平均效用值来评价一次行动的价值is WRONG

在局部观察中，一个行动的价值取决于信度状态

这样可以在信度状态中生成和搜索博弈树

以下行为可以帮助生成信度状态：

- 打出一张牌来刺探对手
- 给合伙人发信号
- 靠演技来最小化信息披露

# 小结

- 游戏展示了AI的很多重要特点
- 完美（最优决策）是不可能达到的，必须采取近似方法