博弈搜索（对抗搜索）

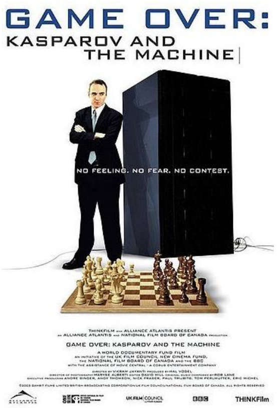
1 博弈：

如何应对巨大状态空间？

博弈的形式化。

2 Minimax算法

3 Alpha-Beta算法



Review

- Breadth-first f(n)=depth(n)
- Depth-first f(n)=-depth(n)
- Depth limited
- Iterative deepening
- Uniform-cost f(n)=g(n)
- Bidirectional
- Greedy best-first f(n)=h(n)
- A* f(n)=g(n)+h(n)

review

- Hill climbing： steepest ascent /stochastic /first-choice /random restart
- Simulated annealing (allowing bad moves)
- Local beam (keeping k states)
- Genetic algorithm (combining two states to generate successors)

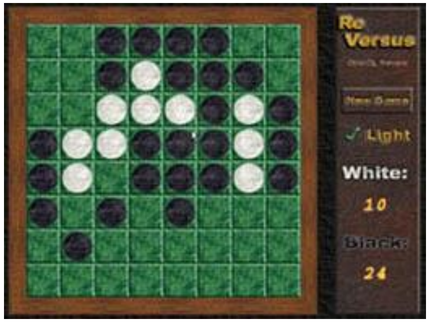
寻找满意的后继

博弈

Game*

- Adversarial(对抗) search problems
  - often known as games(博弈).
  - Game theory(博弈论/对策论)

- For centuries
  - humans have used games like Chess or Go to exert their intelligence.



- Recently
  - there has been great success in building game programs that challenge human supremacy.

Game

- Today's topic about games
  - Two-player, turn-taking
  - fully observable, deterministic
  - zero-sum(零和, 1 + (-1) = 0)
  - time-constrained

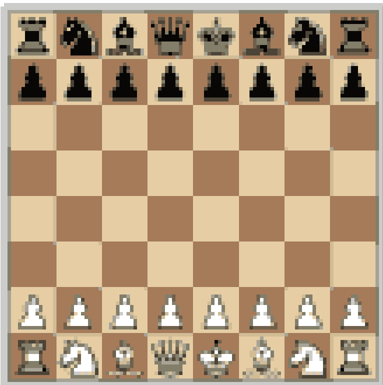
Games are a form of multi-agent environment.

- multi-agent environment
  - What do other agents do and how do they affect our success?
  - Cooperative vs. competitive multi-agent environments.
  - Competitive multi-agent environments give rise to adversarial search a.k.a. games

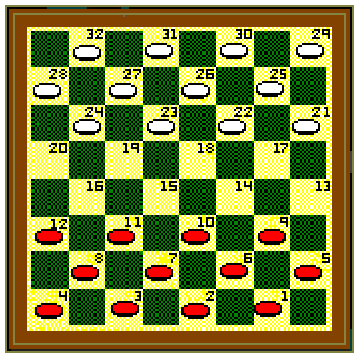
Why study games?

- Why study games? *
  - Games are fun!
  - Historical role in AI
  - Studying games teaches us how to deal with other agents trying to foil our plans
  - Huge state spaces – Games are hard!

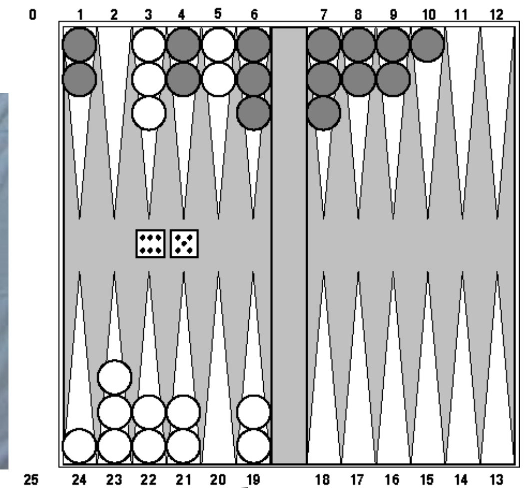
Type of games*



The board set for play



Red to play



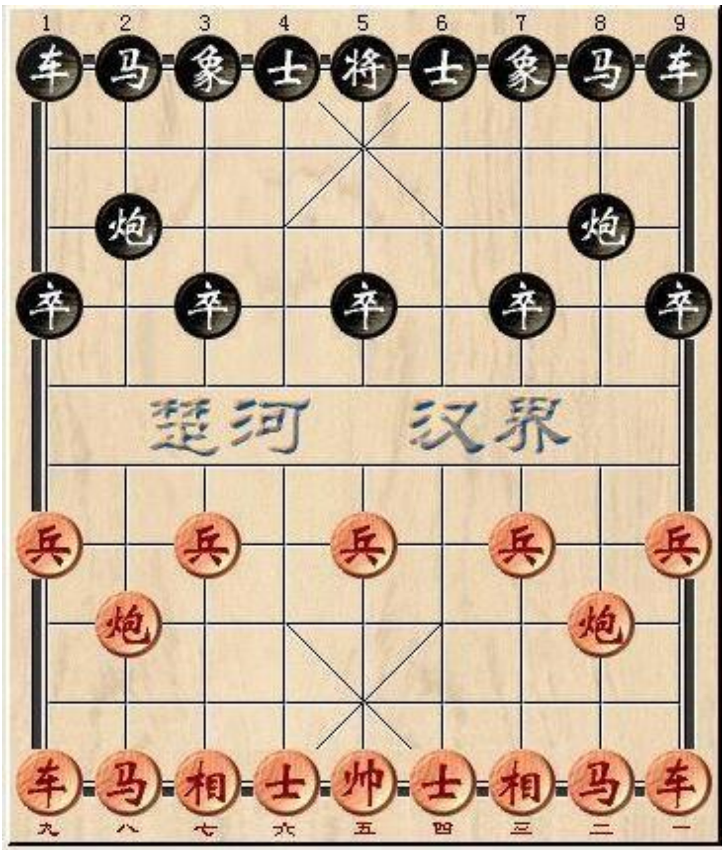
	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly 西洋双陆棋
imperfect information		bridge, poker, scrabble nuclear war

The state-of-the-art of game playing

计算机博弈：永不停歇的挑战！

- 2001年“Deep Fritz” 击败了除了克拉姆尼克之外的所有排名世界前十位的棋手
- 2002年10月“Deep Fritz”与克拉姆尼克在巴林交手，4:4
- 2003年1月“Deep Junior”与卡斯帕罗夫在纽约较量，3:3
- 中国首届“人机大战”，诸宸0:2败于笔记本电脑“紫光之星”
- 2016，AlphaGo

Game playing has a huge state space.
How: Chinese Chess



state space

三十二个棋子
十四种不同的棋子
九列十行

Game playing has a huge state space.

- In general, the branching factor and the depth of terminal states are large.
  - Chess (国际象棋):
    - Number of states: ~10^{40}
    - Branching factor: ~35
    - Number of total moves in a game: ~100
  - 中国象棋10^{161}、
  - 围棋更复杂10^{768}

Game playing has a huge state space.

- 国际象棋搜索树大约有35^{100}或者10^{154}个结点。如果考虑完整的搜索策略，就是用亿次机来处理，也得花天文数字计的时间。博弈要求在无法计算出最优决策的情况下也要给出某种决策。如何尽可能地利用好时间。
- 搜索树的深度影响性能。

How to deal with the huge state space? (what are secrets?)

- Many game programs are based on
  - alpha-beta + iterative deepening + huge databases + ...
- The methods are general, but their implementation is dramatically improved by many specifically tuned-up enhancements (e.g., the evaluation functions).
- Go (圍棋) has too high a branching factor for search techniques. Go software must rely on huge databases and pattern-recognition techniques.
- Search is very important.

games{5.1}

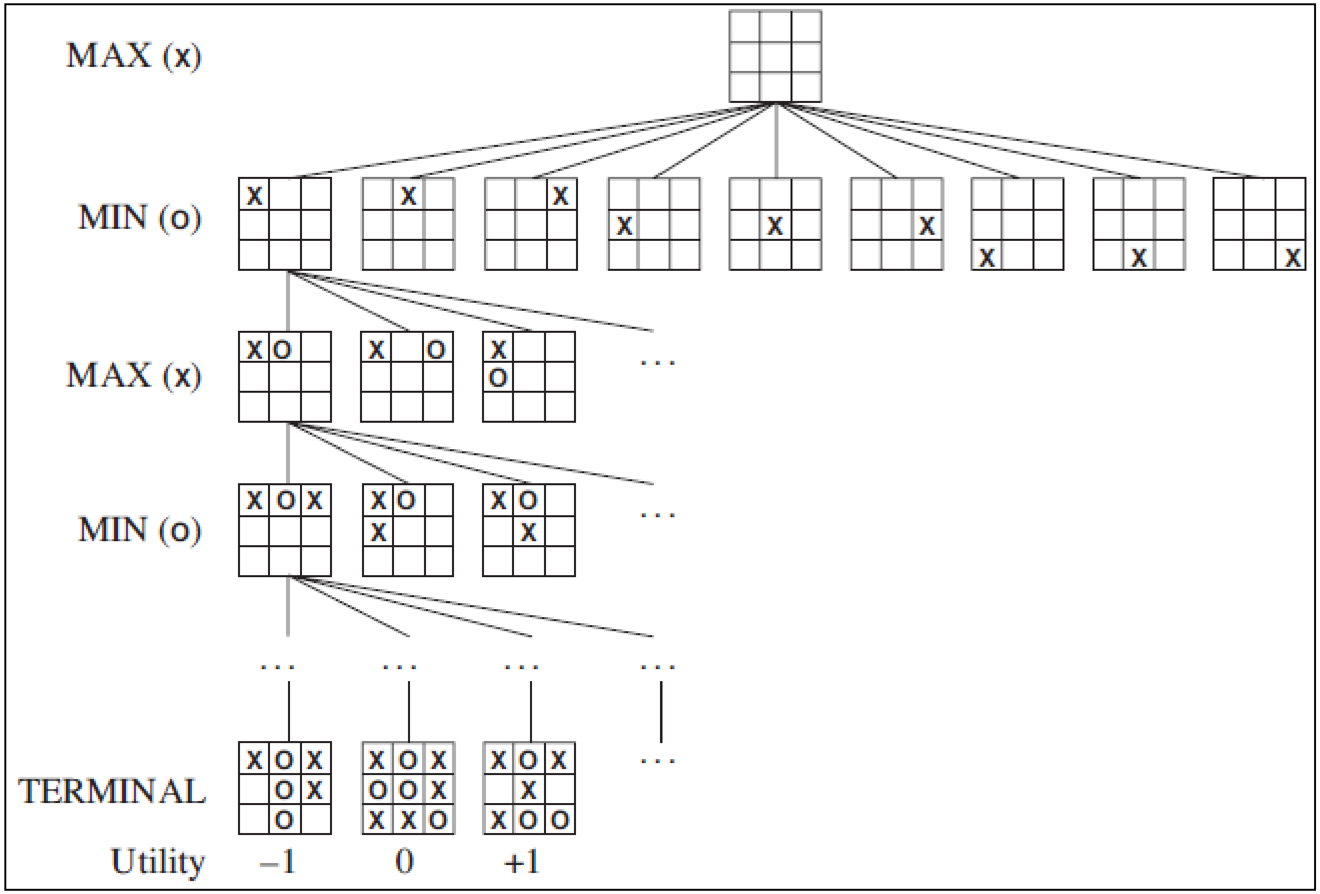
- 考虑两人博弈：MAX和MIN，（为什么这样命名）。MAX先行，两人轮流出招，直到游戏结束。给胜者加分，给败者罚分。
- 游戏可以形式化成一类搜索问题，含有特定的组成部分。

games{5.1}

1. S_0：**初始状态**，规范游戏开始时的情况。
2. PLAYER(s)：定义此时该谁行动。
3. ACTIONS(s)：此状态下的合法移动集合。
4. RESULT(s,a)：**转移模型**，定义行动的结果。
5. TERMINAL-TEST(s)：**终止测试**，游戏结束返回真，否则返回假。游戏结束的状态称为**终止状态**。
6. UTILITY(s,p)：**效用函数**，定义游戏者p在终止状态s下的数值。**零和博弈**是指所有棋手的收益之和在每个棋局实例中都相同。国际象棋中是0+1，1+0或½+½。

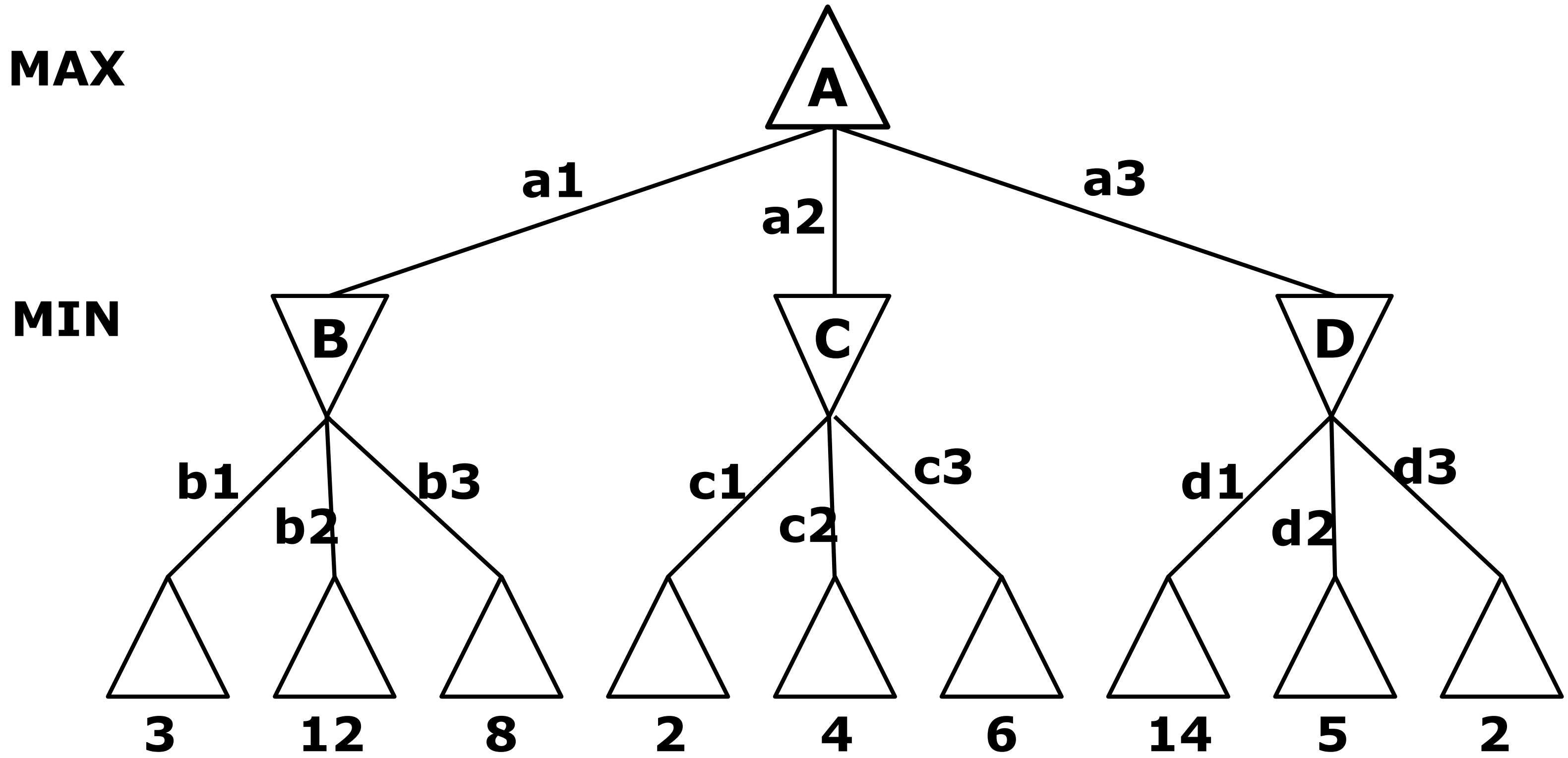
初始状态、ACTIONS函数和RESULT函数定义了游戏的**博弈树**——其中结点是状态，边是移动。

games{5.1}



MINIMAX

optimal decisions in games {5.2}



- MAX选择哪一步？a1/a2/a3

optimal decisions in games {5.2}

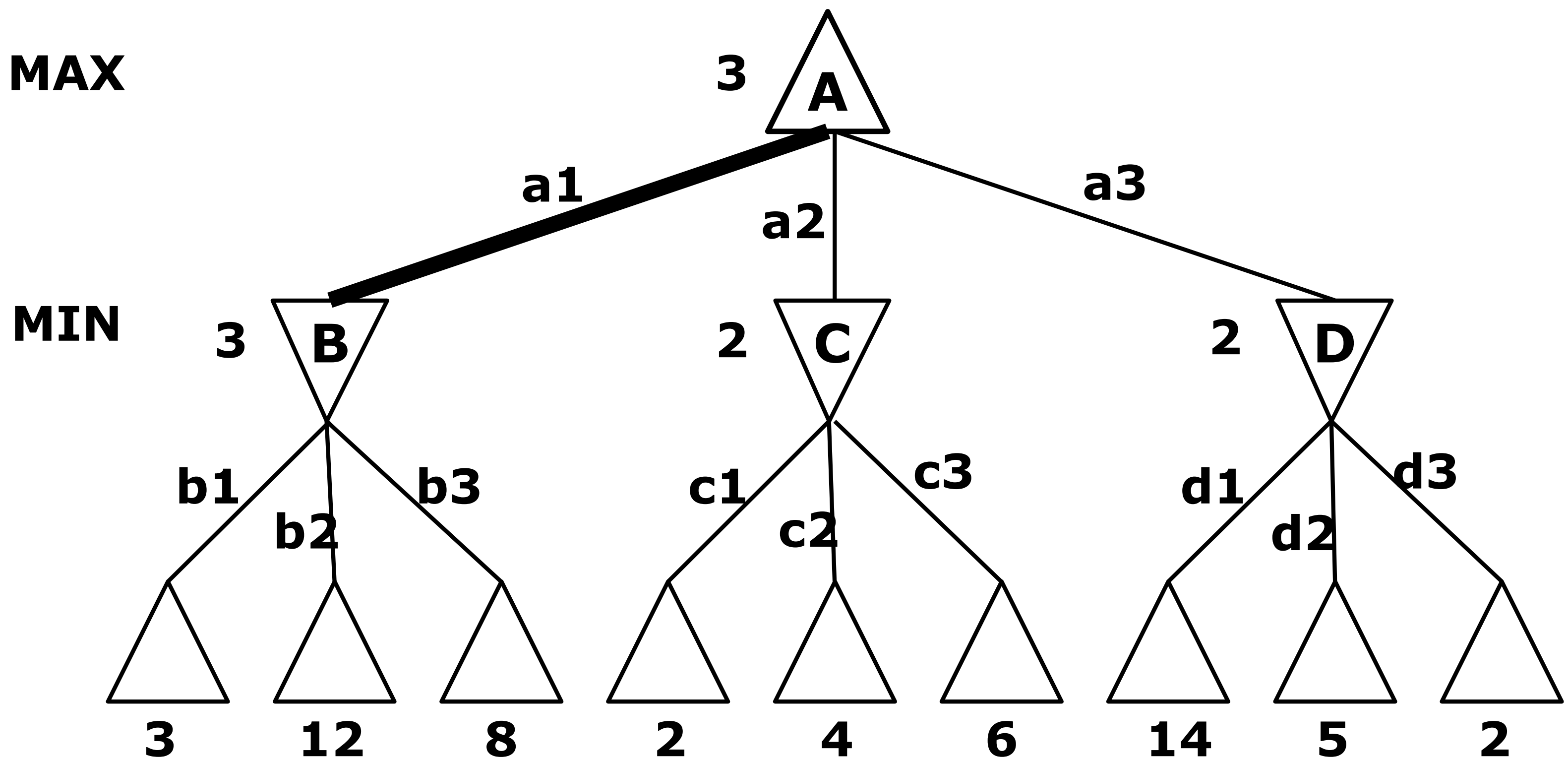
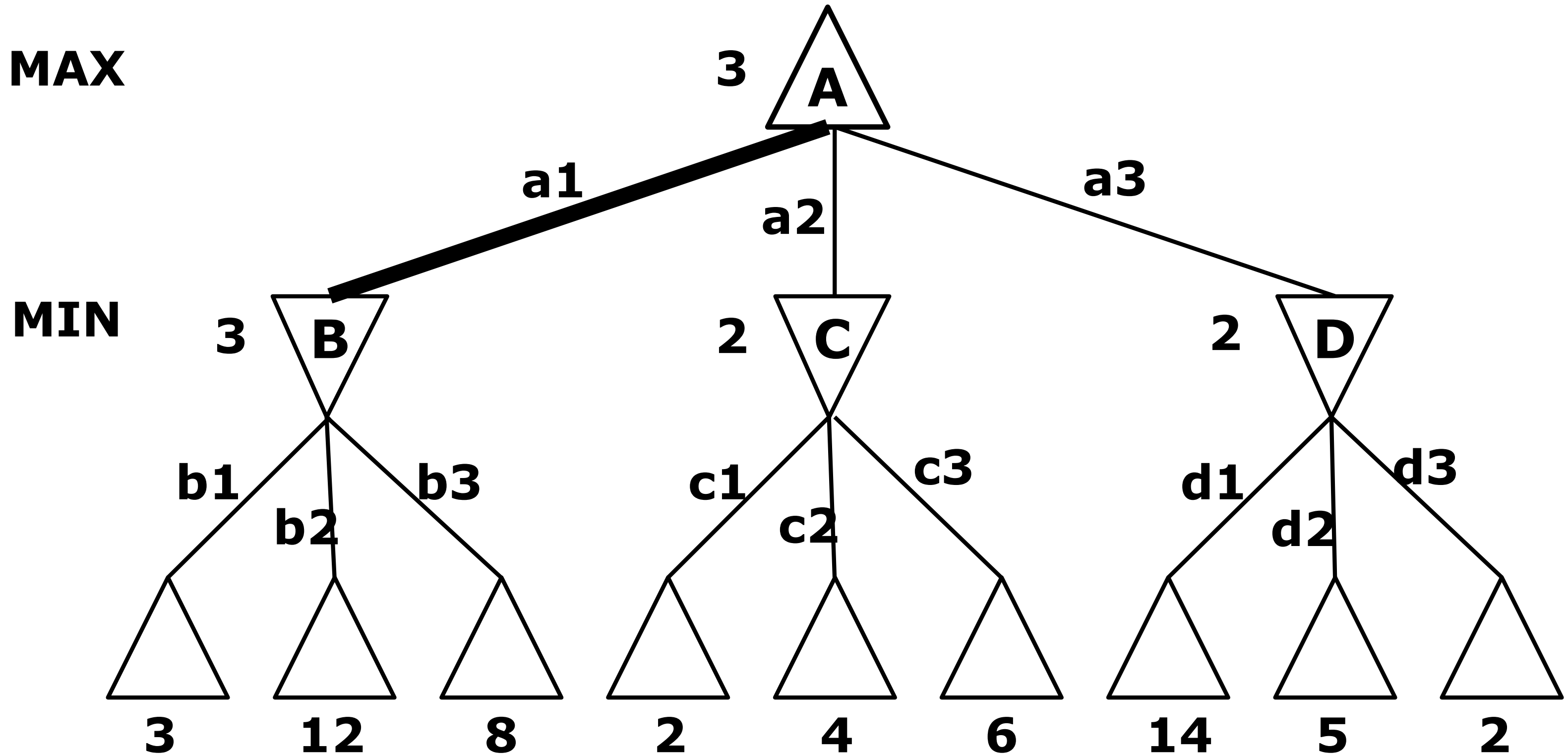


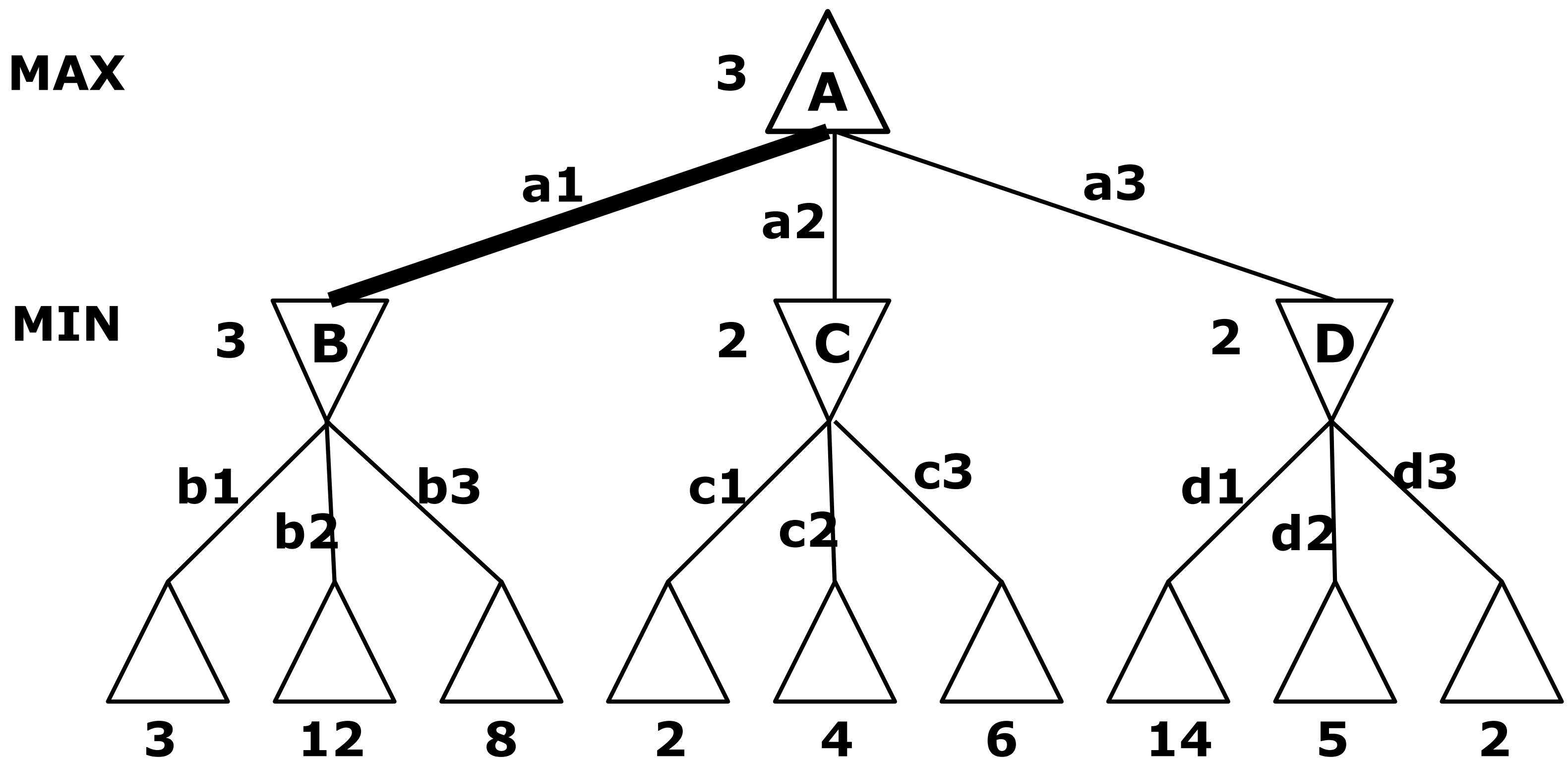
Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

optimal decisions in games {5.2}



$\text{MINIMAX}(s) =$

optimal decisions in games {5.2}



$$
\textsc{Minimax}(s) =
\begin{cases}
\textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\
\max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\
\min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min}
\end{cases}
$$

the minimax algorithm {5.2.1}

function MINIMAX-DECISION(*state*) **returns** *an action*
  return $\arg\max_{a\ \in\ \text{ACTIONS}(s)}$ MIN-VALUE(RESULT(*state*, *a*))

---

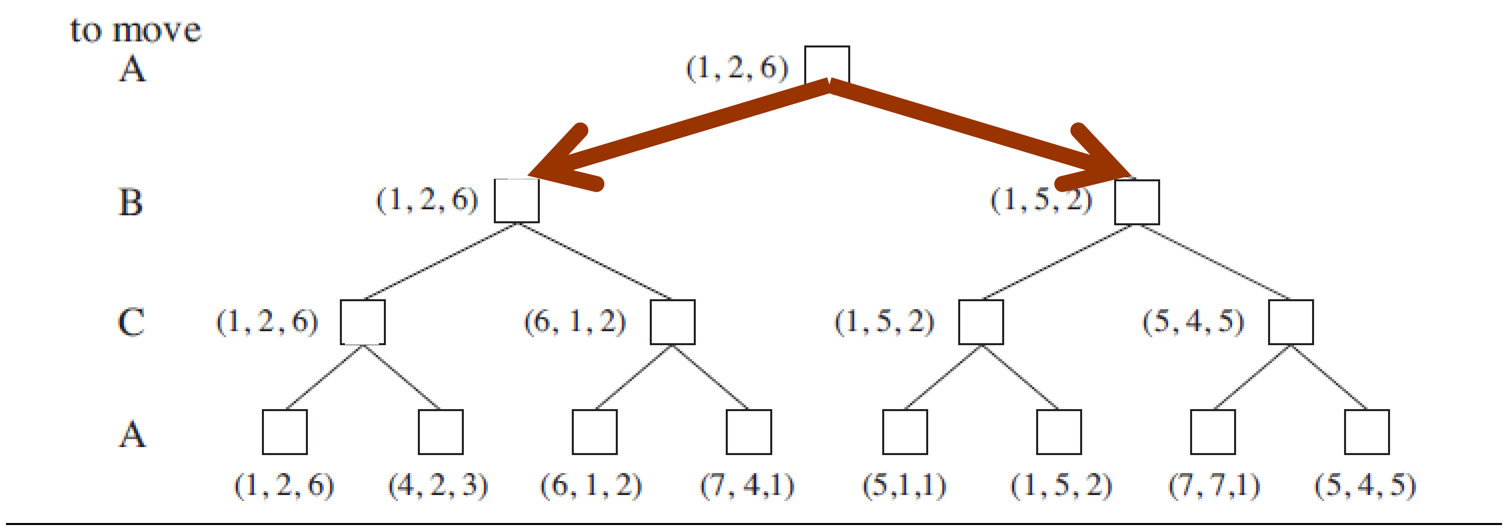
function MAX-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each *a* **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

---

function MIN-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow \infty$
  for each *a* **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a)))
  return v

optimal decisions in multiplayer games {5.2.2}

- 极小极大思想推广到多人博弈中
- 例如若博弈有三个人*A*，*B*和*C*参与，则每个结点都与一个向量〈v_A, v_B, v_C〉相关联。对于终止状态，这个向量代表着从每个人角度出发得到的状态效用值。最简单的实现方法就是让函数UTILITY返回一个效用值向量

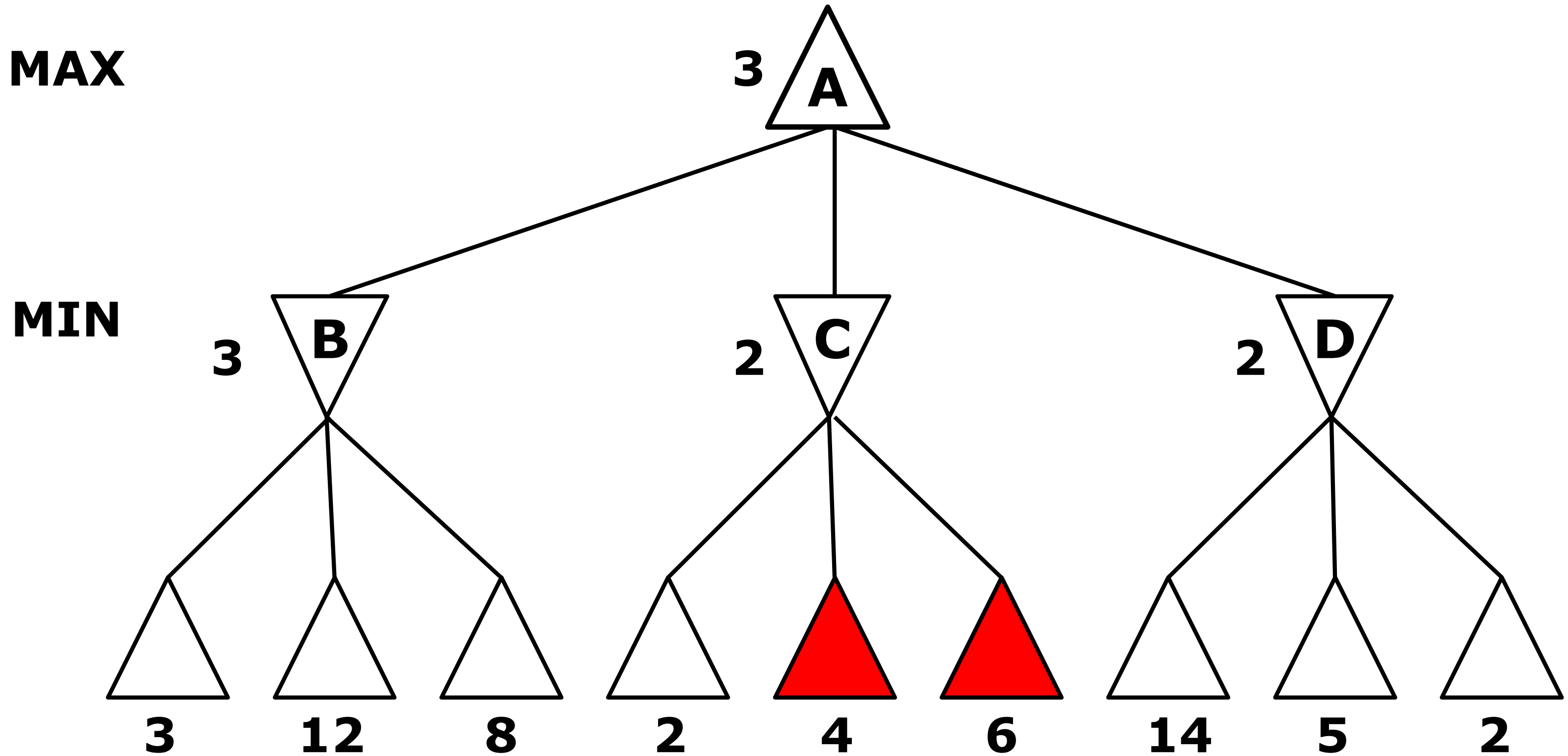


optimal decisions in multiplayer games {5.2.2}

- 对于非终止状态，结点n的回传值就是该选手在结点n选择的后继者的效用值向量。例如状态X, 回传值是〈v_A=1, v_B=2, v_C=6〉。
- 选手之间出现**联盟**。联盟不断建立或者解散。
- 如果游戏是非零和的，合作也可能发生在两人游戏中。例如，假设终止状态的效用值向量是〈v_A=1000, v_B=1000〉，并且1000对于两个选手都是最高的可能效用值。

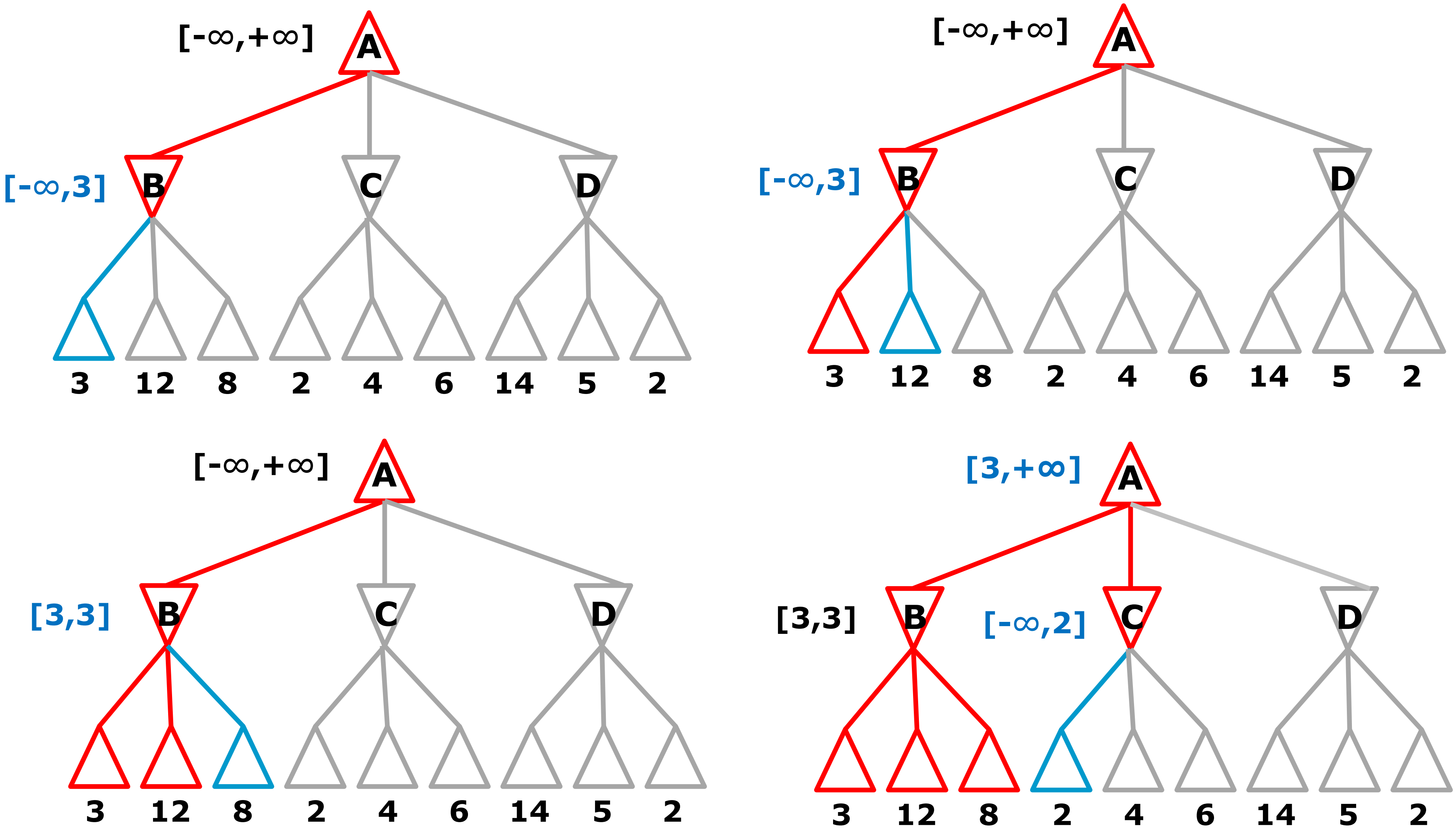
ALPHA-BETA

Alpha-Beta Pruning {5.3}



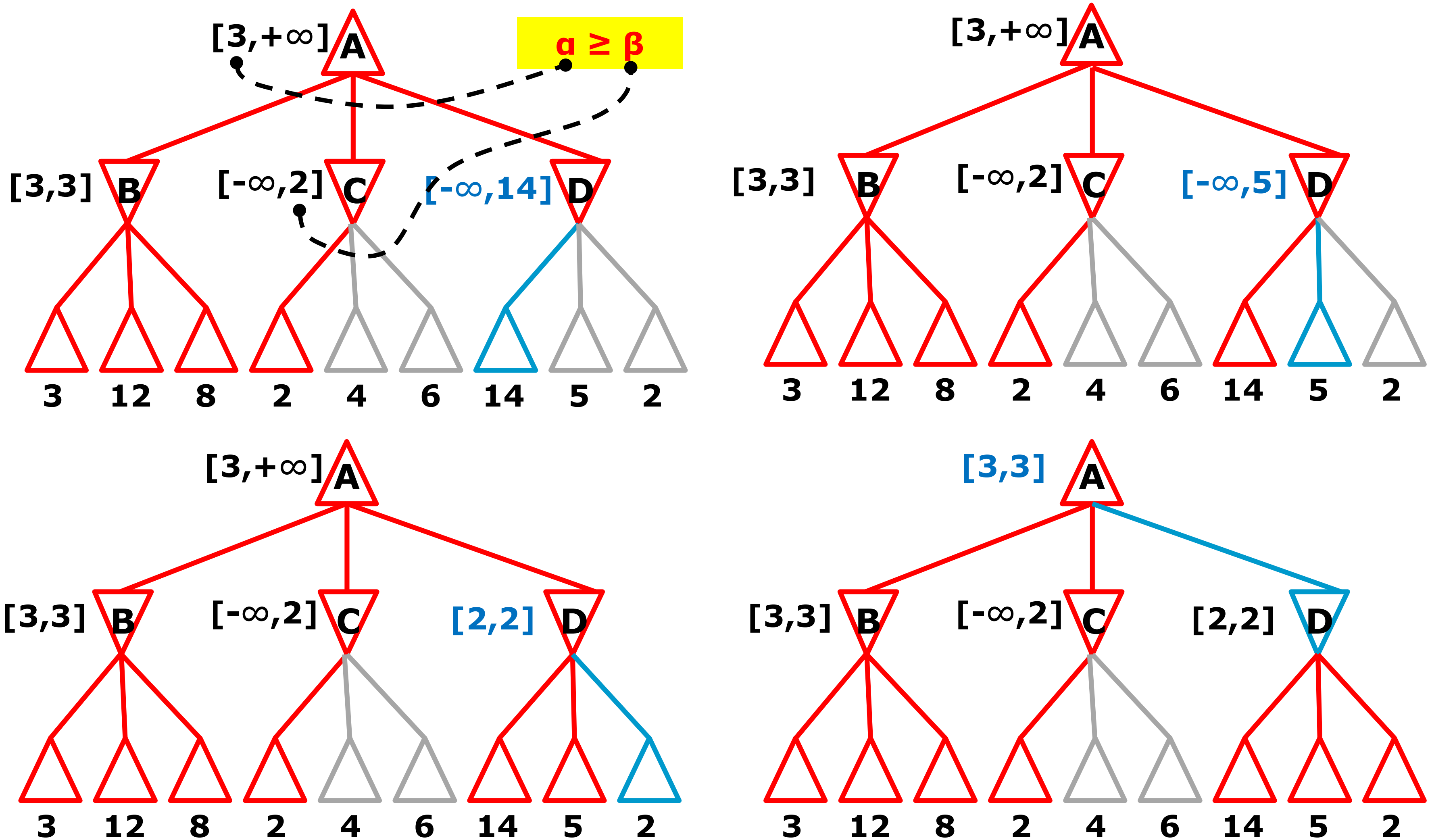
$$\text{MINIMAX}(root) = \max(\min(3, 12, 8), \min(2, x, y), \min(14, 5, 2))$$
$$= \max(3, \min(2, x, y), 2)$$
$$= \max(3, z, 2) \qquad \text{where } z = \min(2, x, y) \leq 2$$
$$= 3$$

Alpha-Beta Pruning {5.3}



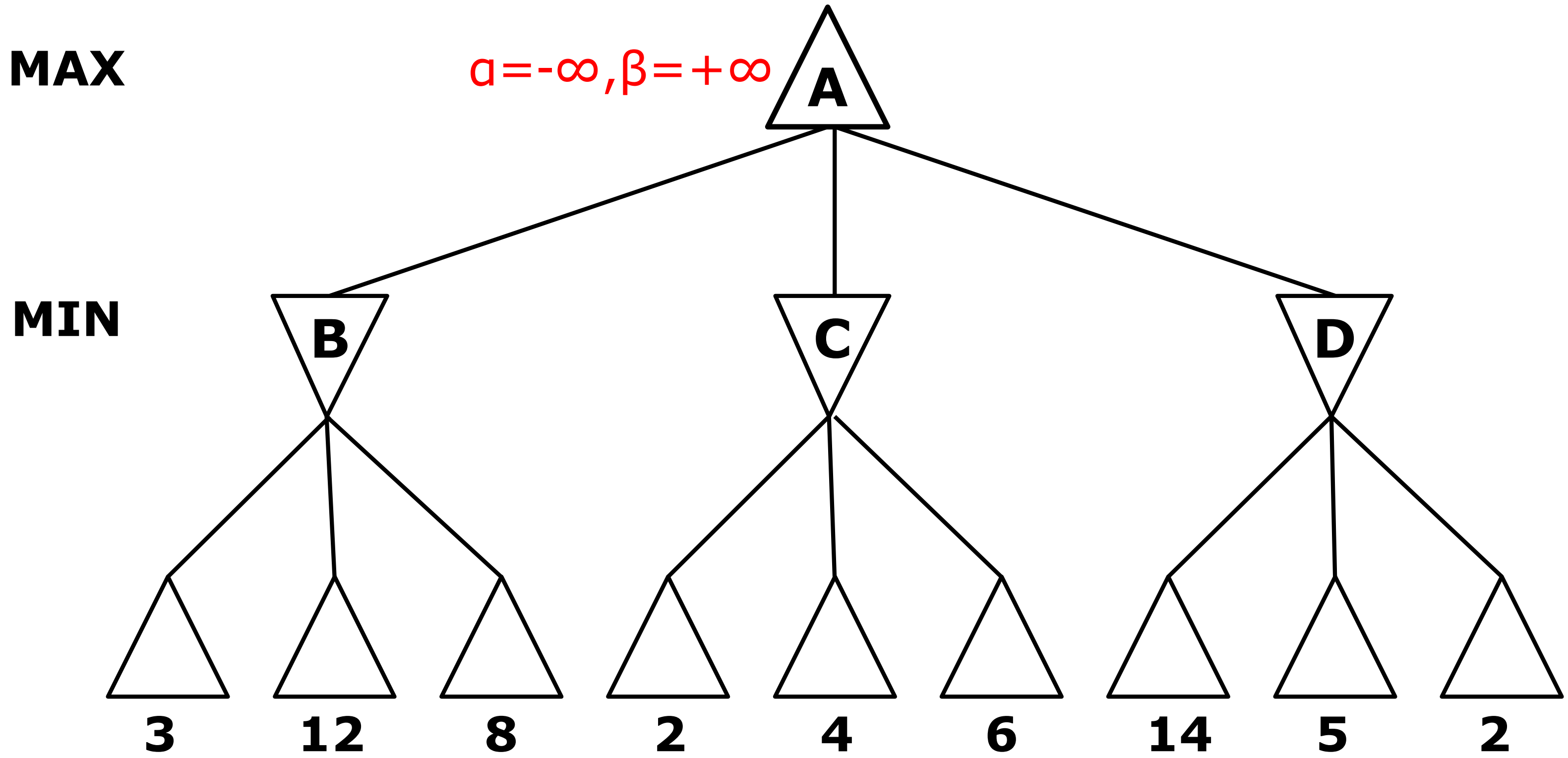
each node is marked with a range of its value

Alpha-Beta Pruning {5.3}



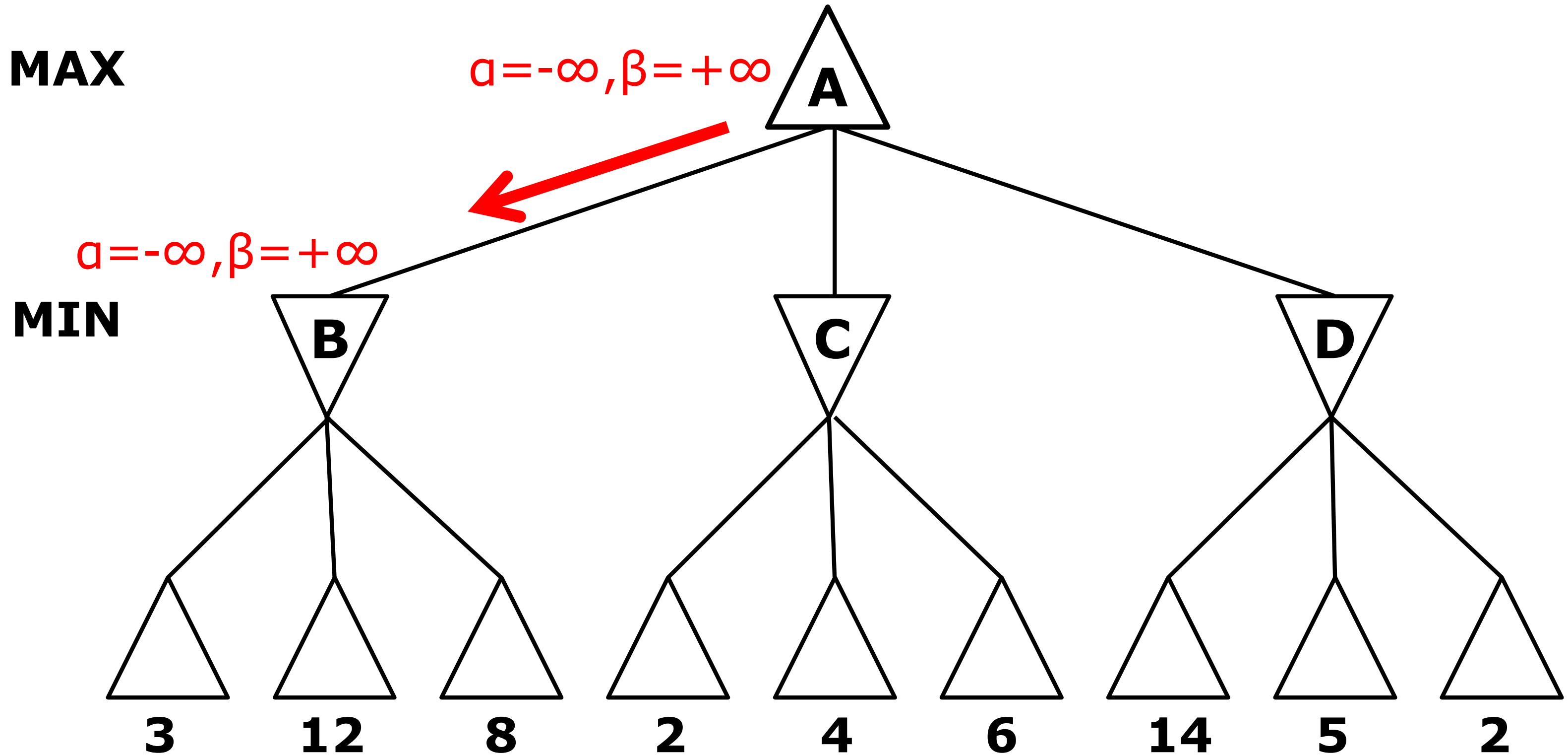
each node is marked with a range of its value

Alpha-Beta Pruning {5.3}



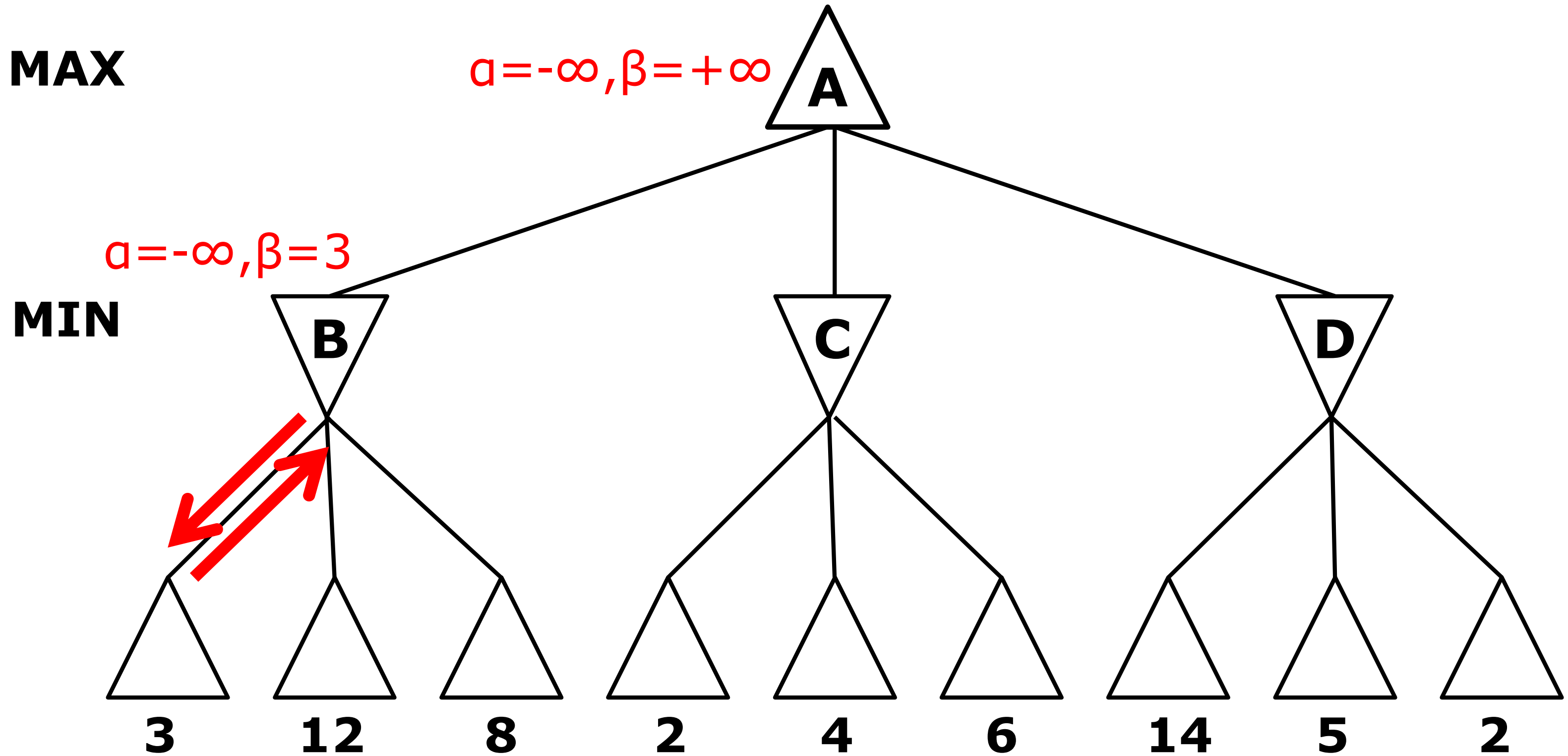
- α = 到目前为止路径上发现的**MAX**的最佳值（即极大值）
- β = 到目前为止路径上发现的**MIN**的最佳值（即极小值）

Alpha-Beta Pruning {5.3}



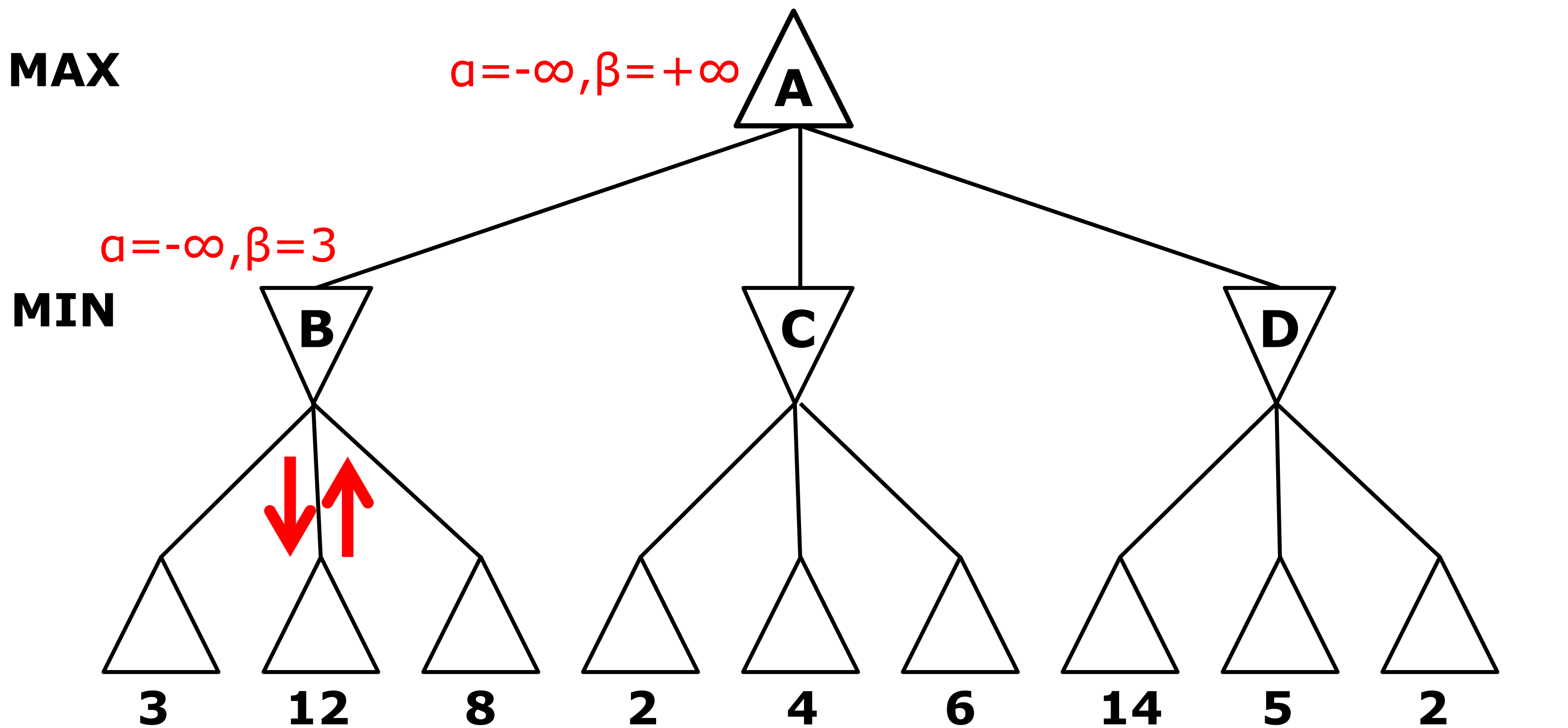
- α = 到目前为止路径上发现的**MAX**的最佳值（即极大值）
- β = 到目前为止路径上发现的**MIN**的最佳值（即极小值）

Alpha-Beta Pruning {5.3}



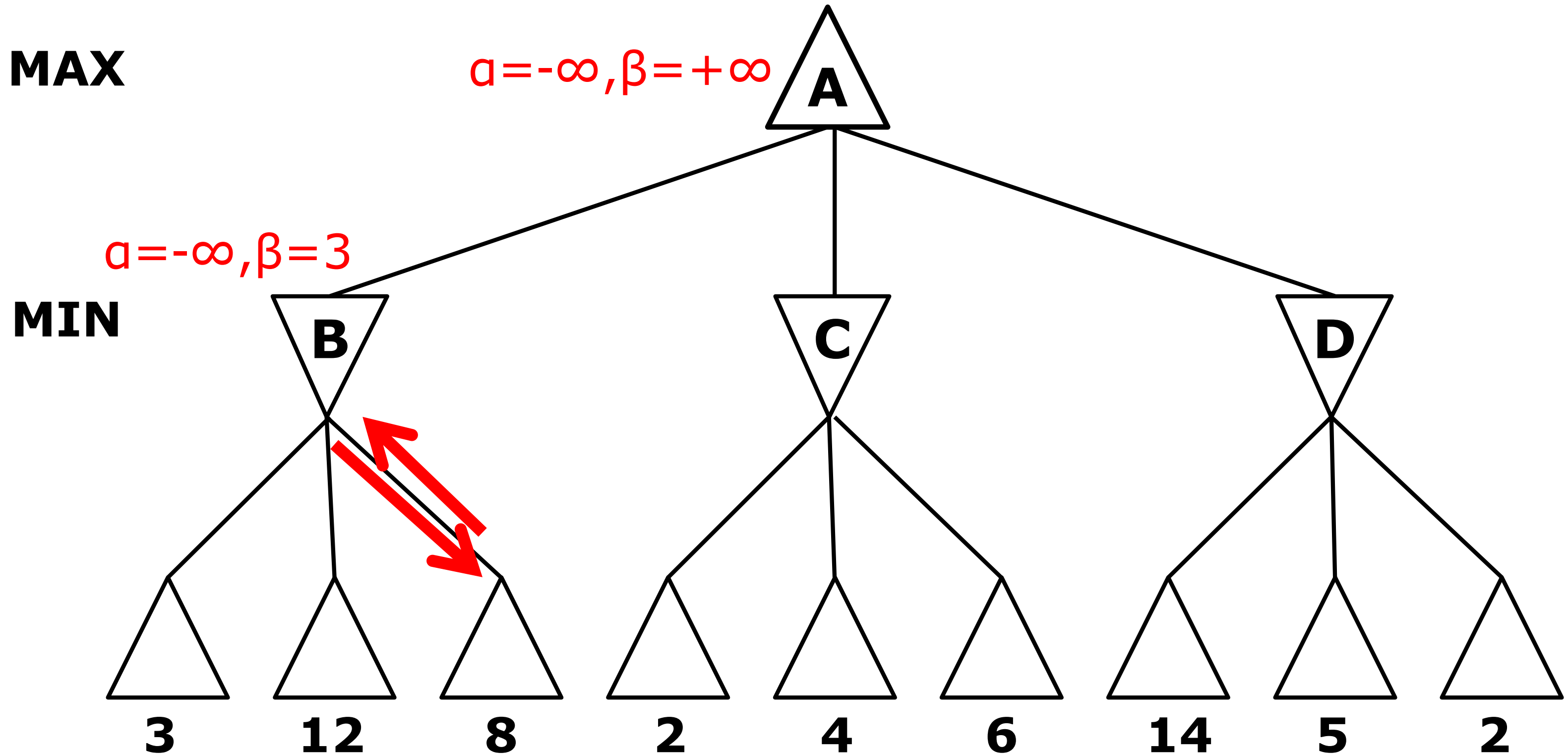
- α = 到目前为止路径上发现的MAX的最佳值（即极大值）
- β = 到目前为止路径上发现的MIN的最佳值（即极小值）

Alpha-Beta Pruning {5.3}



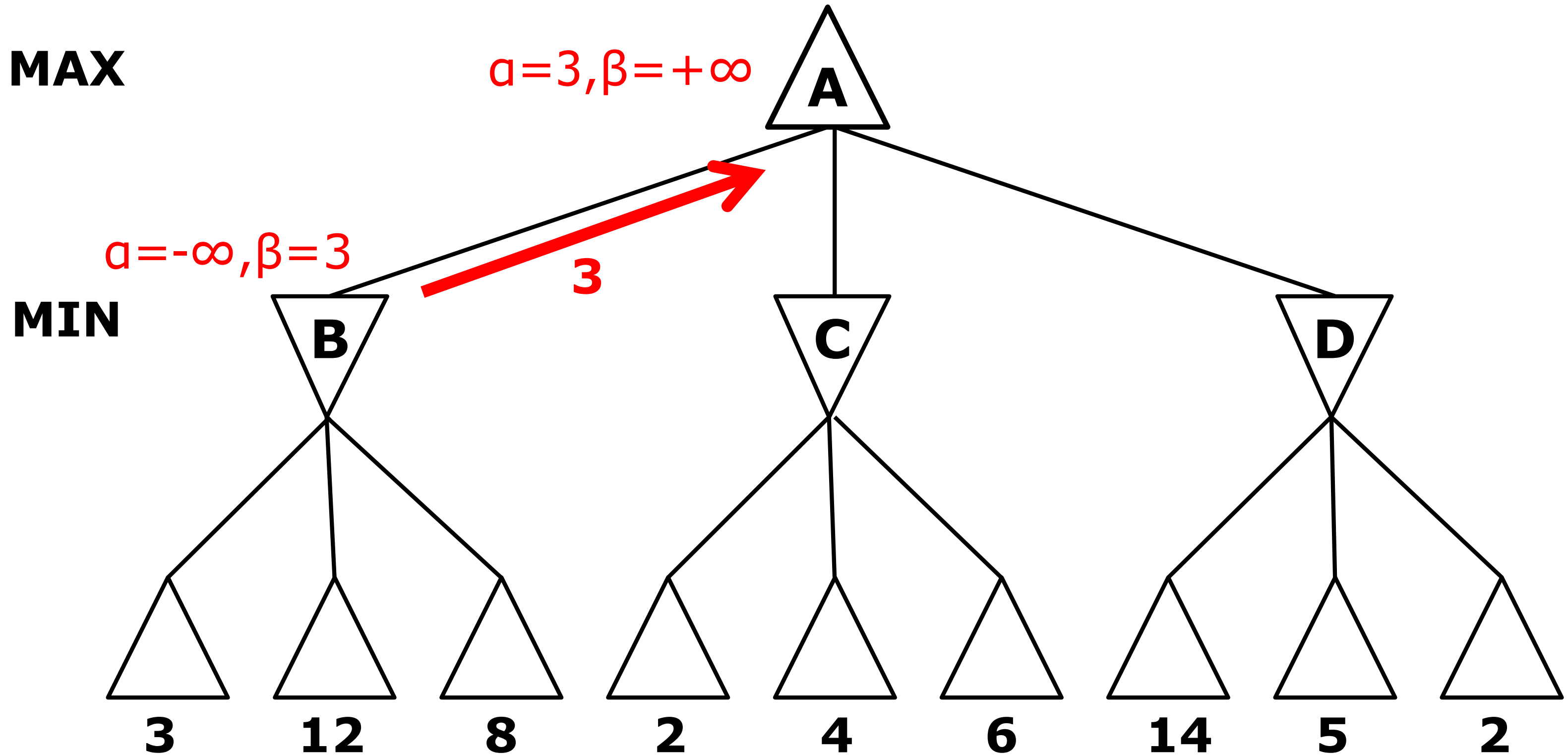
- α = 到目前为止路径上发现的MAX的最佳值（即极大值）
- β = 到目前为止路径上发现的MIN的最佳值（即极小值）

Alpha-Beta Pruning {5.3}



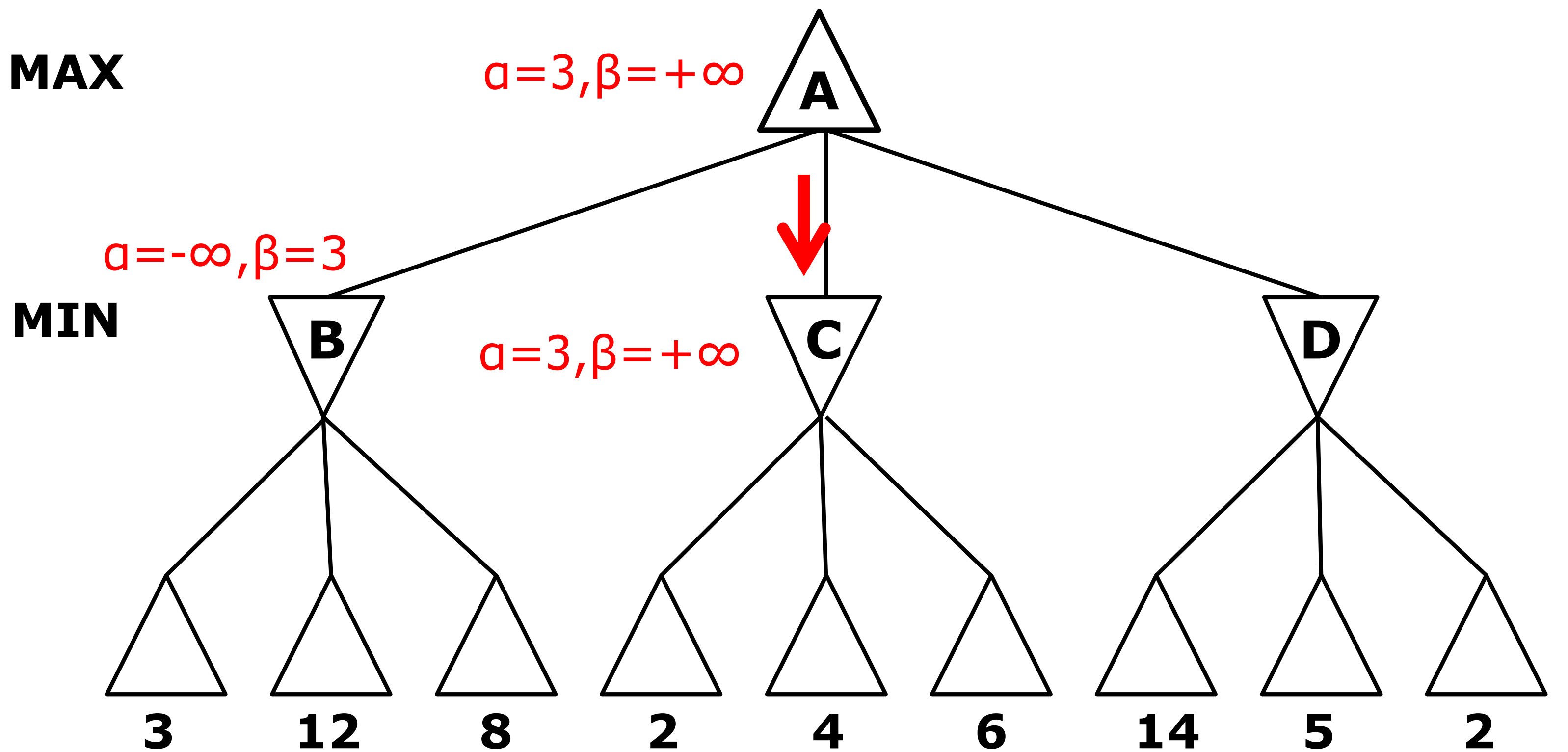
- α = 到目前为止路径上发现的MAX的最佳值（即极大值）
- β = 到目前为止路径上发现的MIN的最佳值（即极小值）

Alpha-Beta Pruning {5.3}



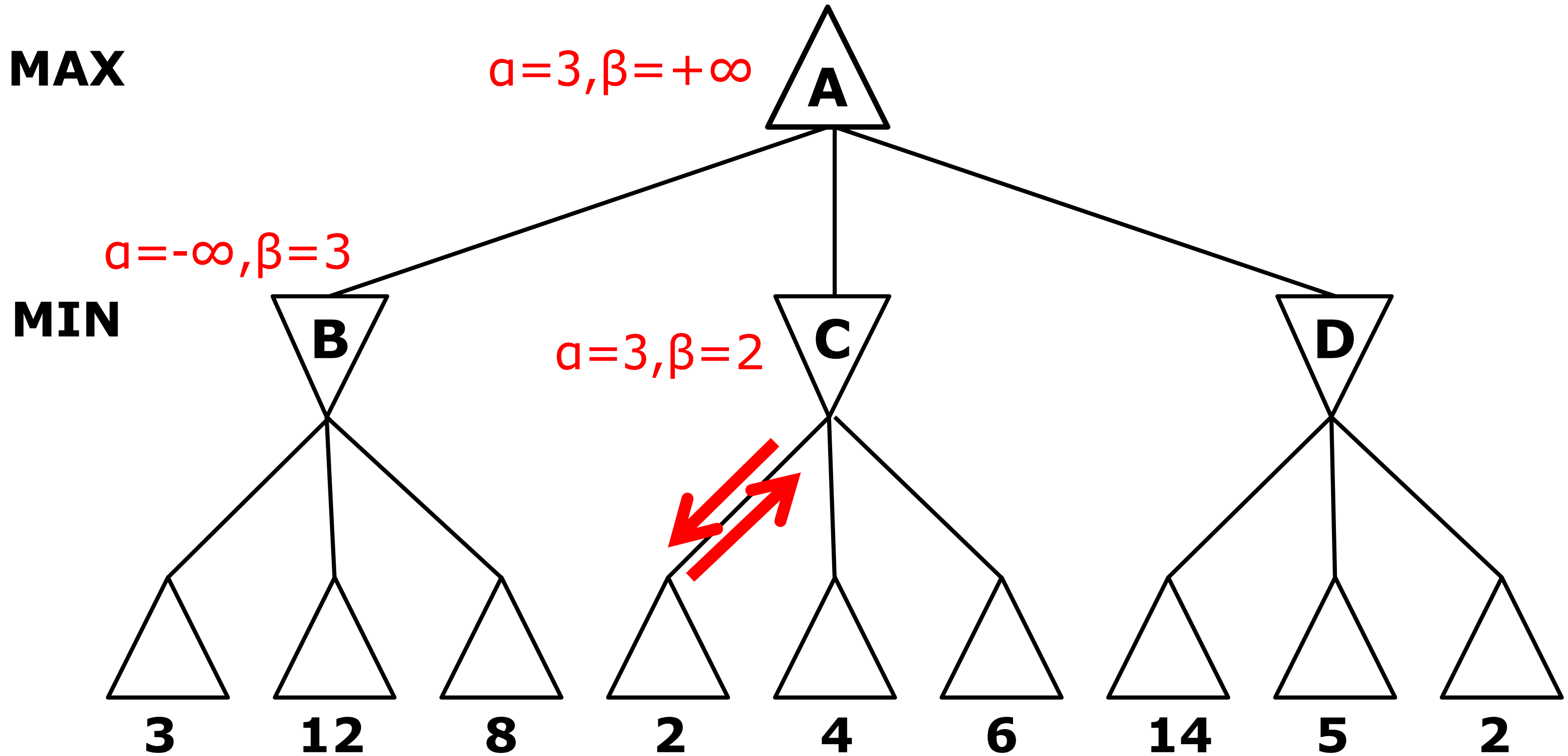
- α = 到目前为止路径上发现的MAX的最佳值（即极大值）
- β = 到目前为止路径上发现的MIN的最佳值（即极小值）

Alpha-Beta Pruning {5.3}



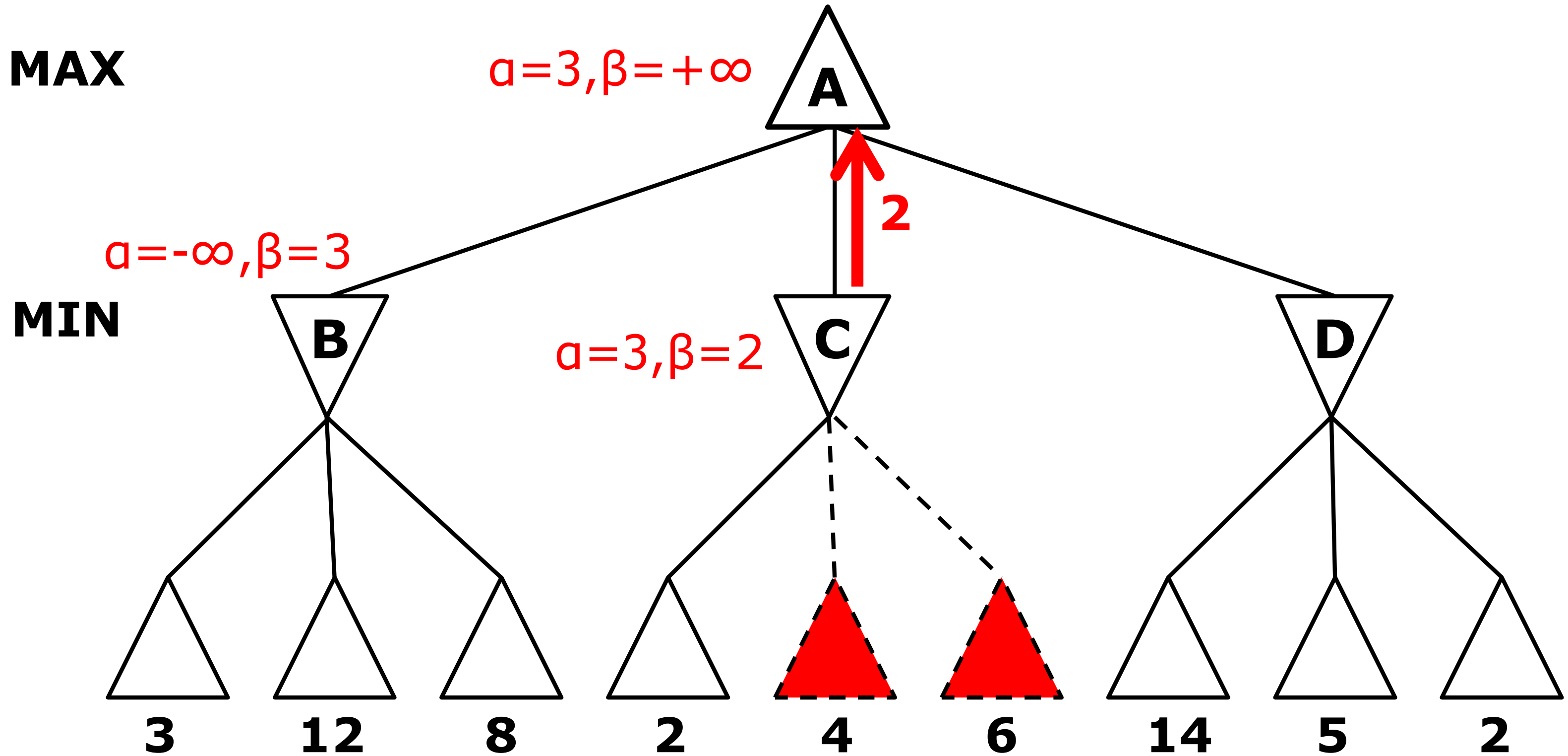
- α = 到目前为止路径上发现的MAX的最佳值（即极大值）
- β = 到目前为止路径上发现的MIN的最佳值（即极小值）

Alpha-Beta Pruning {5.3}



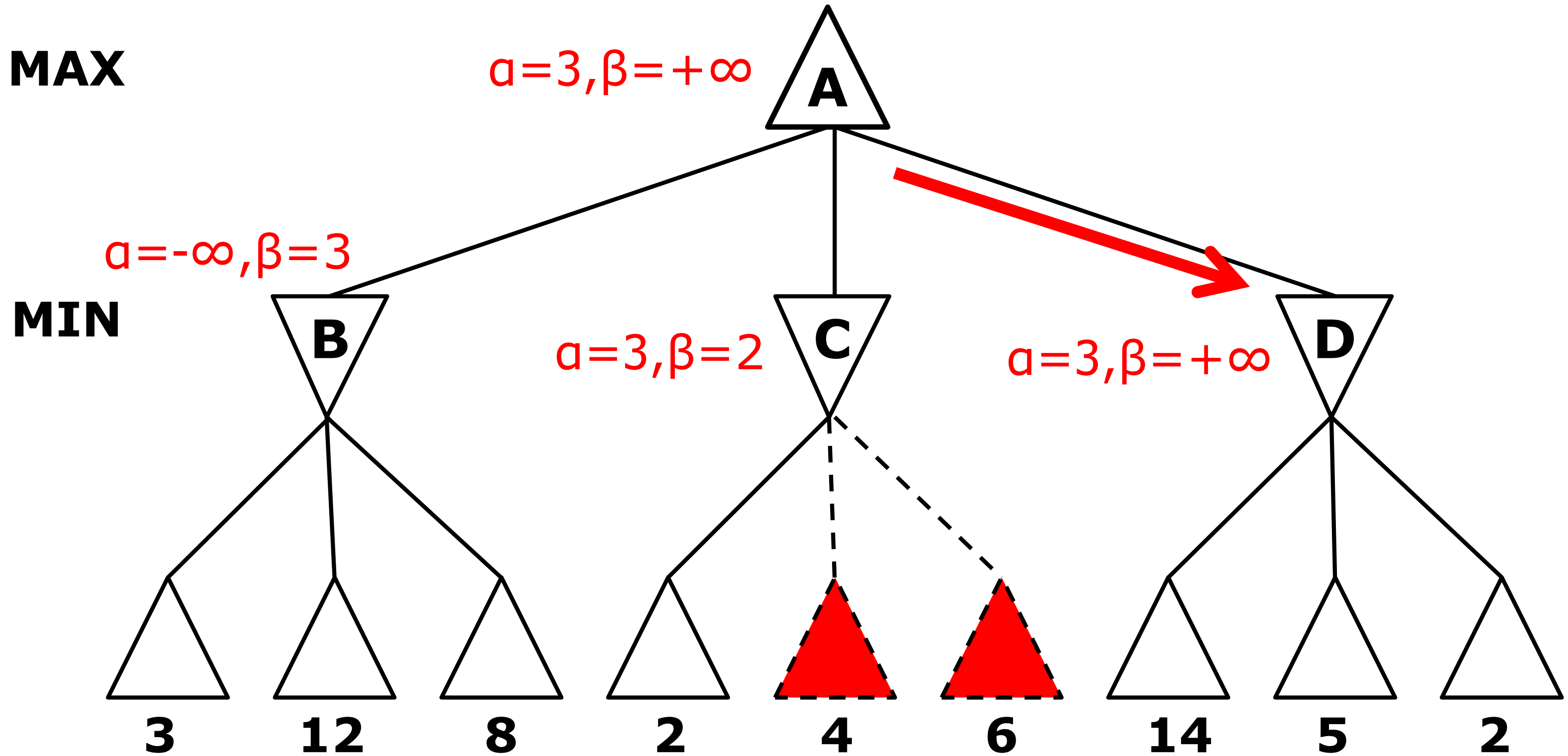
- α = 到目前为止路径上发现的MAX的最佳值（即极大值）
- β = 到目前为止路径上发现的MIN的最佳值（即极小值）

Alpha-Beta Pruning {5.3}



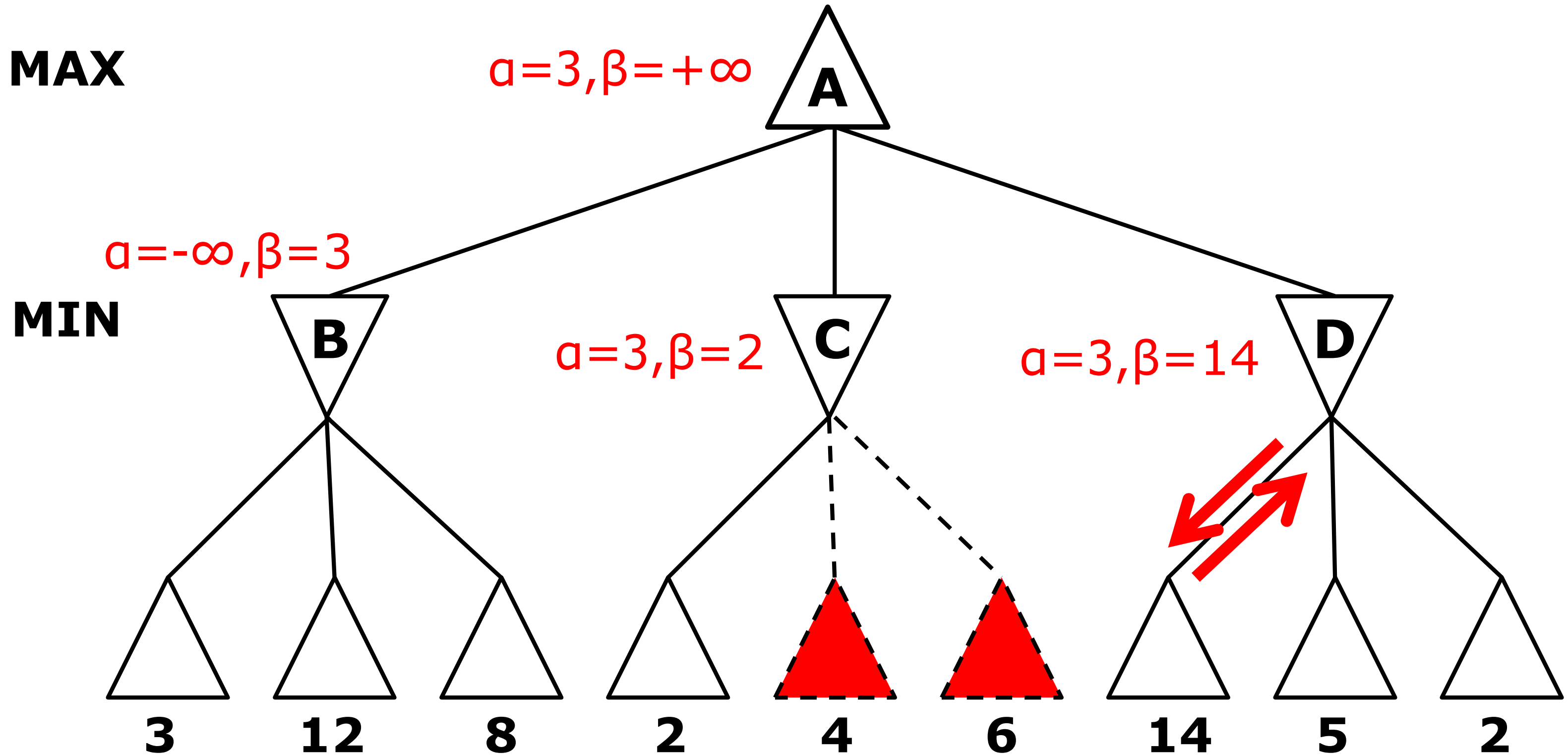
- α = 到目前为止路径上发现的MAX的最佳值（即极大值）
- β = 到目前为止路径上发现的MIN的最佳值（即极小值）

Alpha-Beta Pruning {5.3}



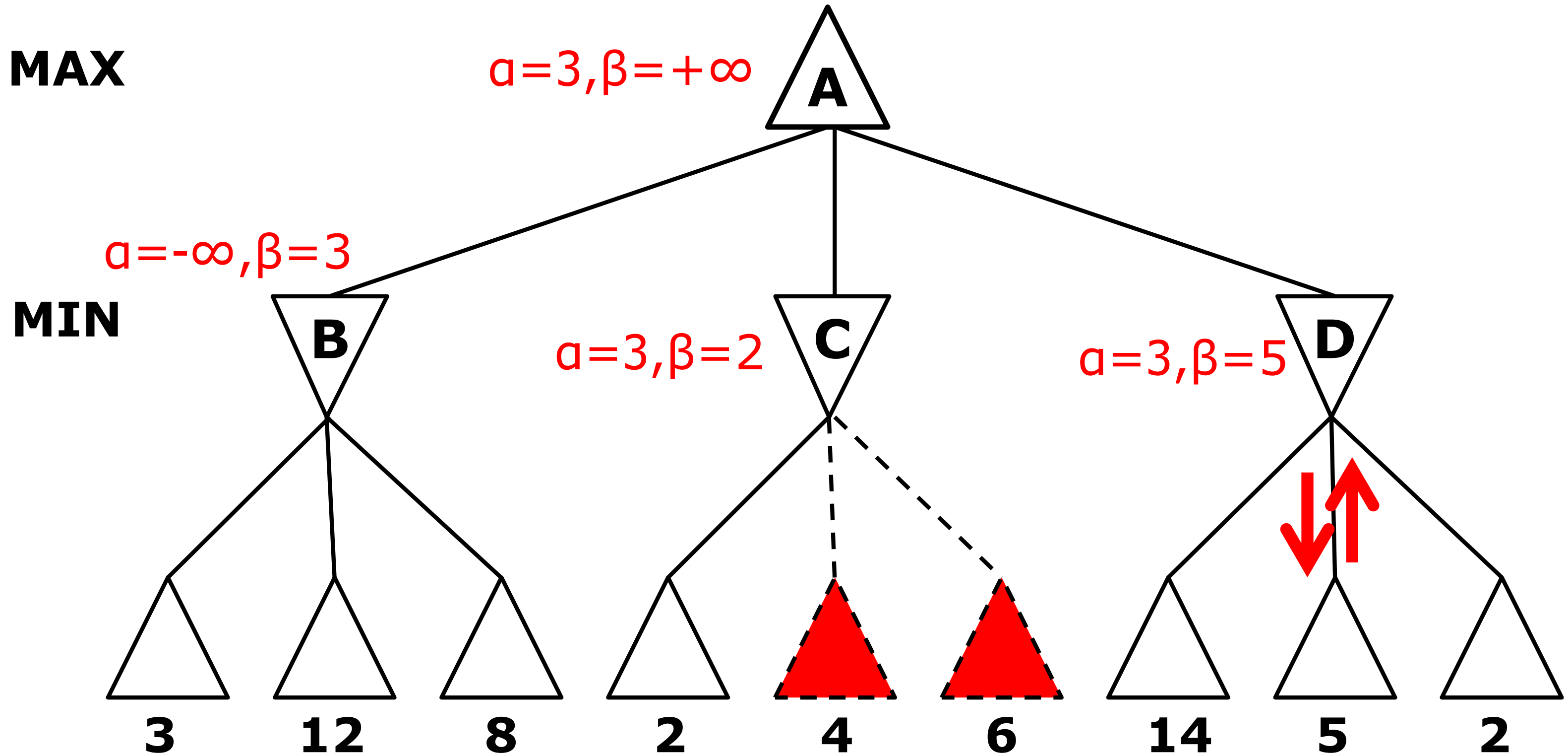
- α = 到目前为止路径上发现的MAX的最佳值（即极大值）
- β = 到目前为止路径上发现的MIN的最佳值（即极小值）

Alpha-Beta Pruning {5.3}



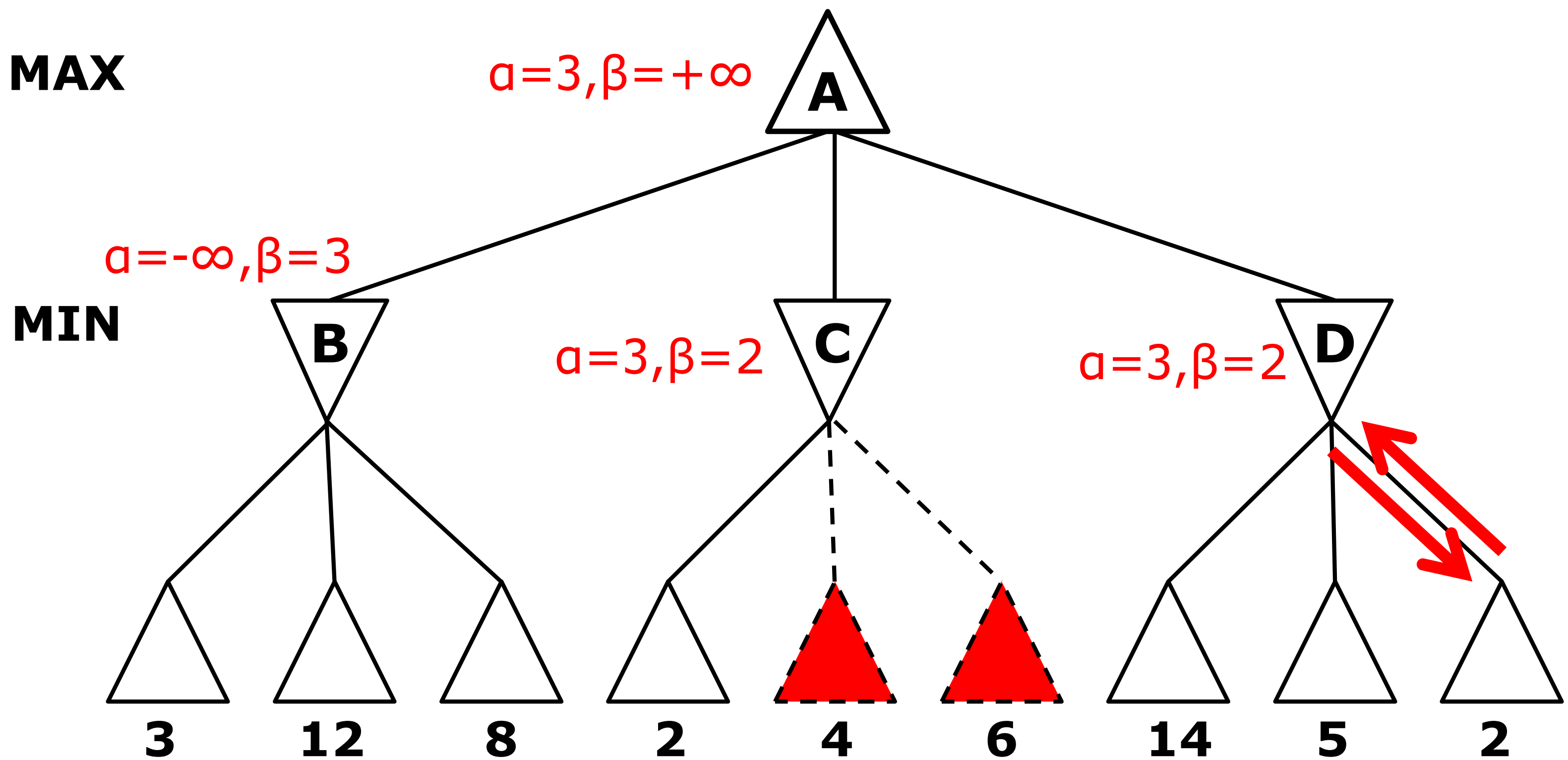
- α = 到目前为止路径上发现的MAX的最佳值（即极大值）
- β = 到目前为止路径上发现的MIN的最佳值（即极小值）

Alpha-Beta Pruning {5.3}



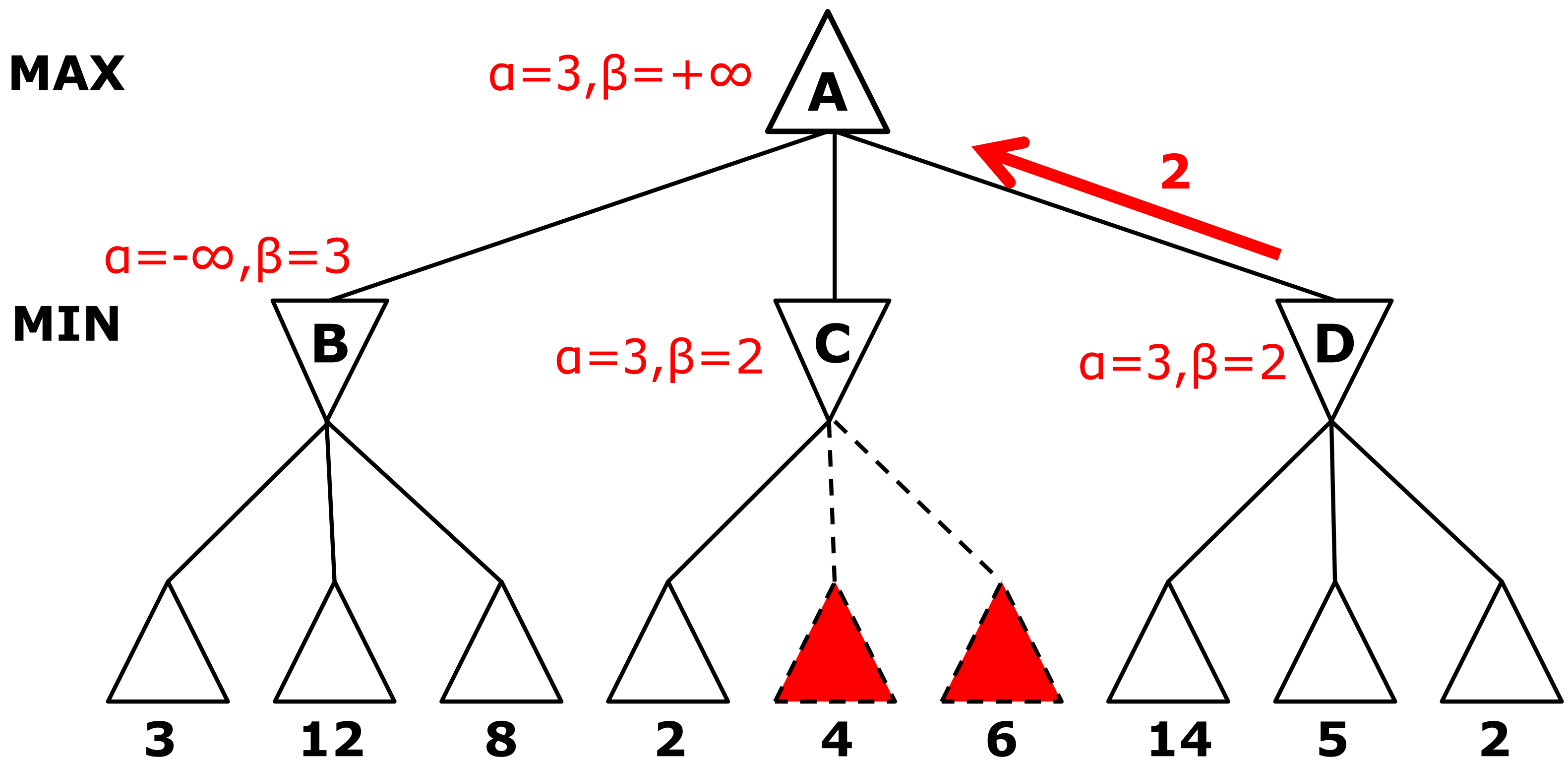
- α = 到目前为止路径上发现的MAX的最佳值（即极大值）
- β = 到目前为止路径上发现的MIN的最佳值（即极小值）

Alpha-Beta Pruning {5.3}



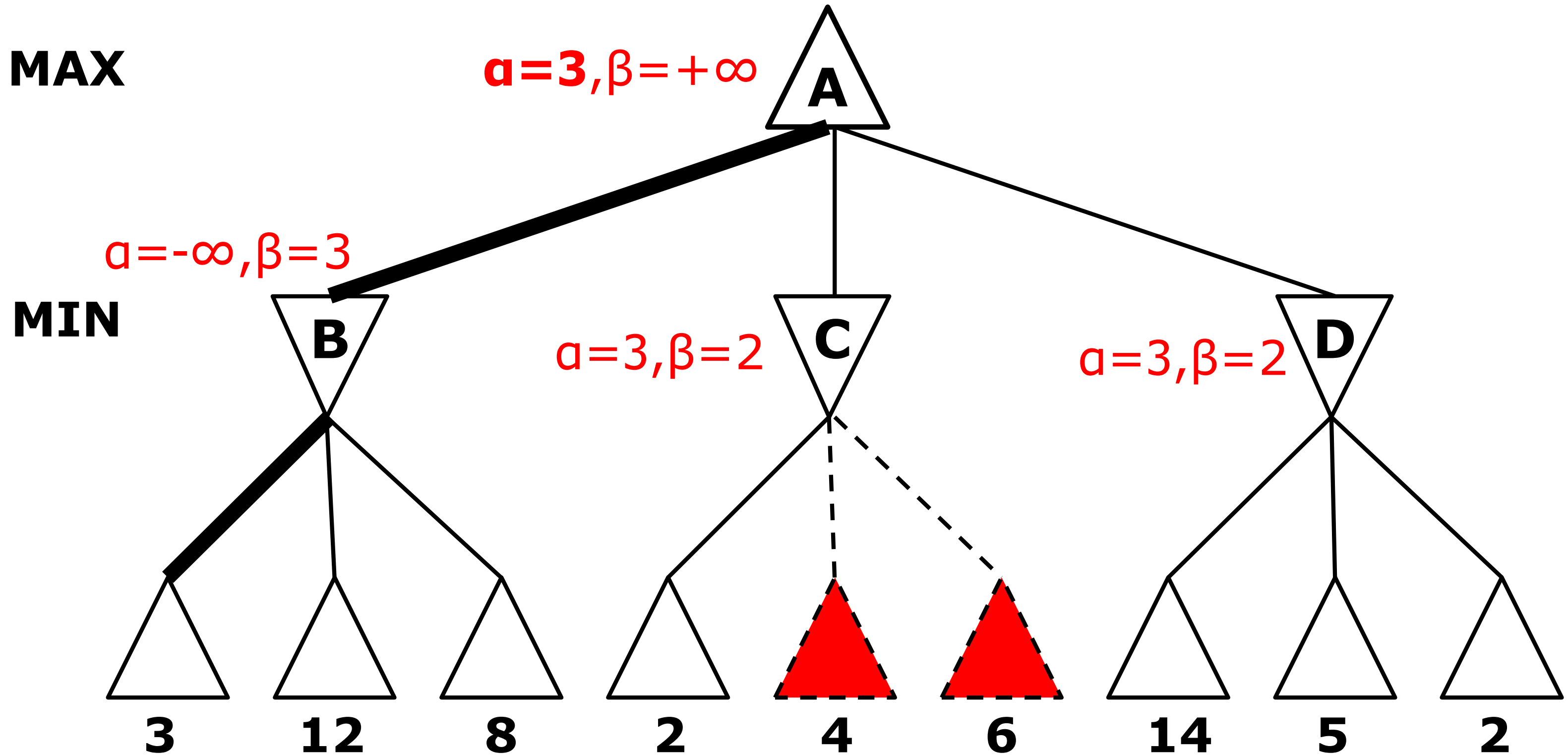
- α = 到目前为止路径上发现的MAX的最佳值（即极大值）
- β = 到目前为止路径上发现的MIN的最佳值（即极小值）

Alpha-Beta Pruning {5.3}



- α = 到目前为止路径上发现的MAX的最佳值（即极大值）
- β = 到目前为止路径上发现的MIN的最佳值（即极小值）

Alpha-Beta Pruning {5.3}

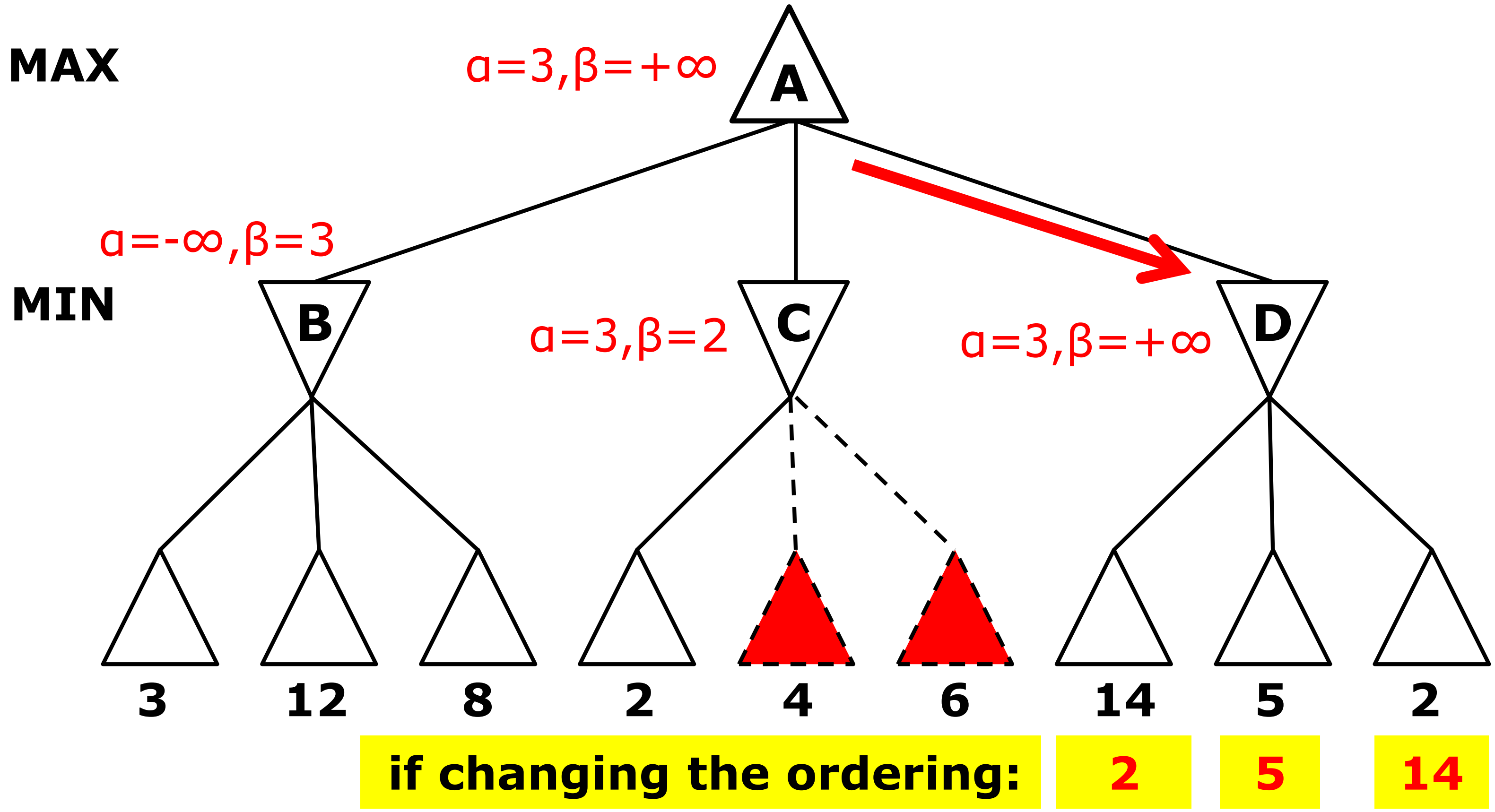


- α = 到目前为止路径上发现的MAX的最佳值（即极大值）
- β = 到目前为止路径上发现的MIN的最佳值（即极小值）

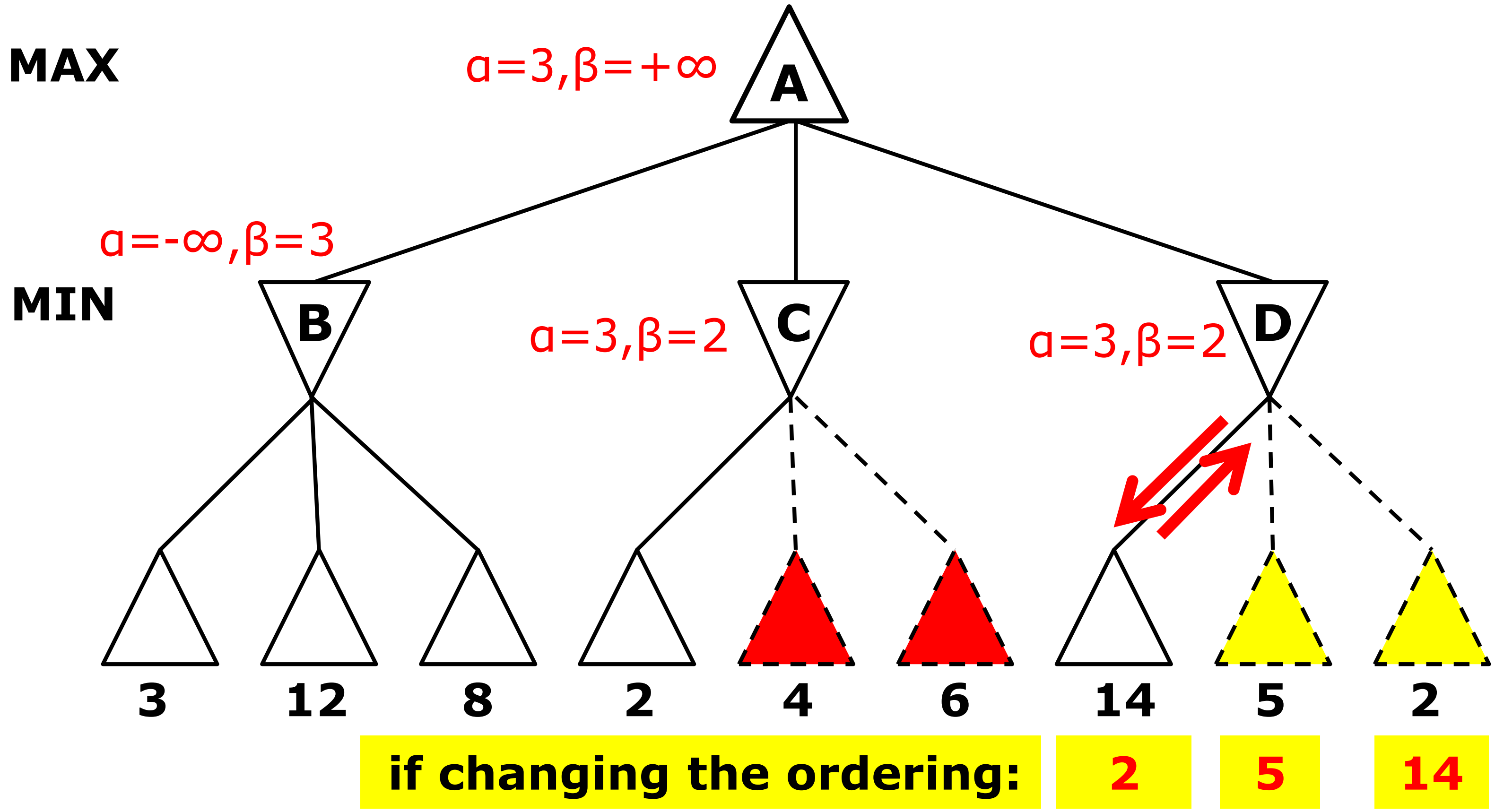
```
function Alpha-Beta-Search(state) returns an action
    v ← Max-Value(state,- ∞,+ ∞)
    return the action in Actions(state) with value v

function Max-Value(state, α,β) returns a utility value
    if Terminal-Test(state) then return Utility(state)
    for each a in Actions(state) do
        α= Max(α,Min-Value(Result(s,a), α,β)
        if α ≥ β  then return α
     return α

function Min-Value(state, α,β) returns a utility value
    if Terminal-Test(state) then return Utility(state)
    for each a in Actions(state) do
        β= Min(β,Max-Value(Result(s,a), α,β)
        if α ≥ β  then return β
     return β
```


MAX
MIN
A
B
C
D
α=3,β=+∞
α=-∞,β=3
α=3,β=2
α=3,β=+∞
3
12
8
2
4
6
14
5
2
if changing the ordering:
2
5
14

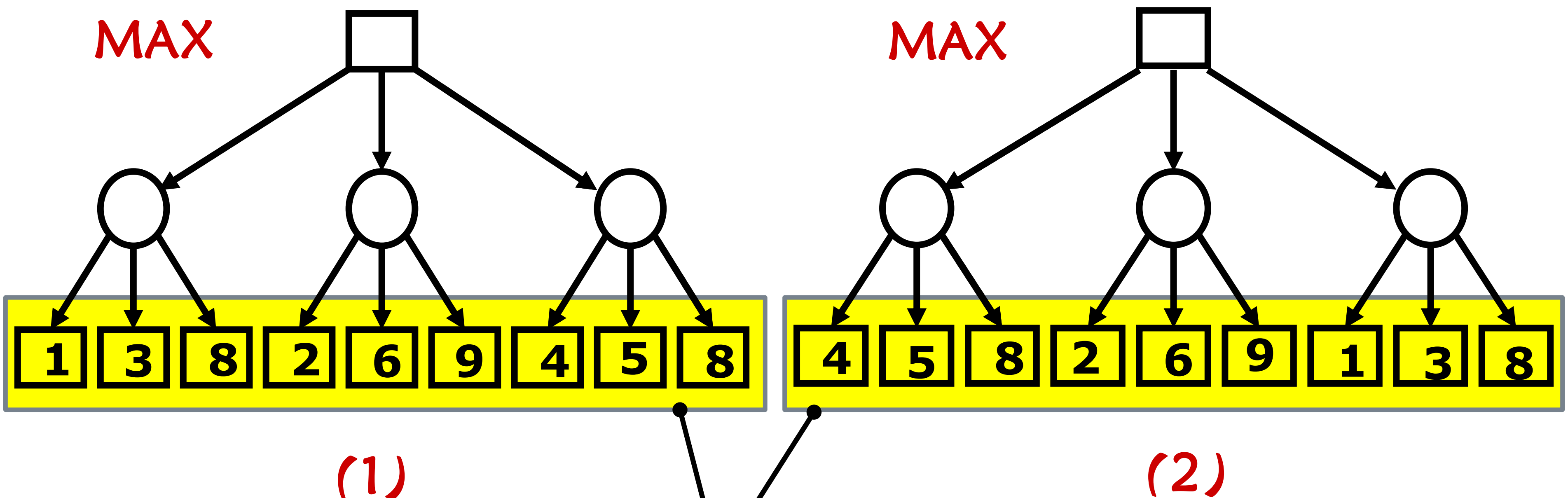
move ordering{5.3}



It is better if the MAX children of a MIN node are ordered in increasing backed up values

move ordering {5.3.1}

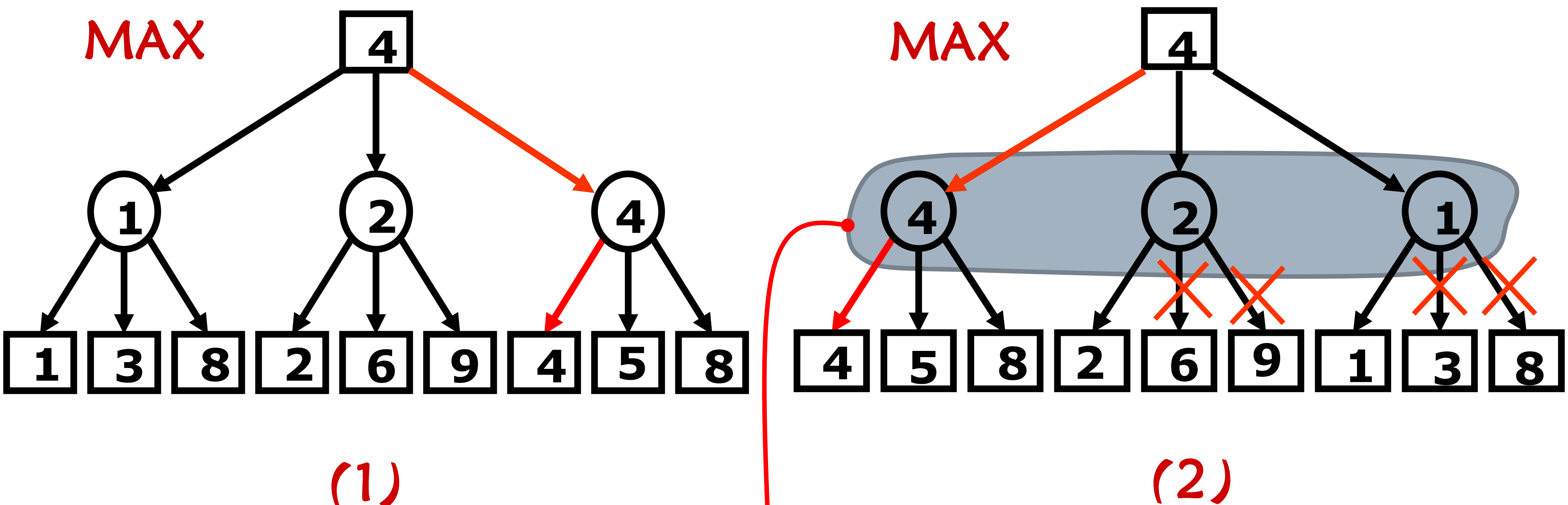
- find the best move:
- Which nodes are pruned?



The MAX children of MIN nodes are ordered in increasing backed up values

move ordering {5.3.1}

- find the best move:
- Which nodes are pruned?



It is better if the MIN children of a MAX node are ordered in decreasing backed up values

move ordering {5.3.1}

- Assume a game tree of uniform branching factor b
- Minimax examines $O(b^m)$ nodes, so does alpha-beta in the worst-case

- The gain for alpha-beta is maximum when:
  - The MIN children of a MAX node are ordered in decreasing backed up values
  - The MAX children of a MIN node are ordered in increasing backed up values
  - then alpha-beta examines $O(b^{m/2})$ nodes

- But this requires an oracle (if we knew how to order nodes perfectly, we would **not need to search the tree**)

- If nodes are ordered at random, then the average number of nodes examined by alpha-beta is ~$O(b^{3m/4})$

We often can not reach to the terminal nodes within limited time!

imperfect real-time decisions {5.4}

- 应该尽早截断搜索
- 用估计棋局效用值的启发式**评估函数**EVAL取代效用函数，用决策什么时候运用EVAL的**截断测试(cutoff test)**取代终止测试

$$\text{H-MINIMAX}(s,d) = \begin{cases} \text{EVAL}(s) & \text{if CUTOFF-TEST}(s,d) \\ \max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MIN}. \end{cases}$$

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

evaluation functions {5.4.1}

- 评估函数必须和取胜几率相关：算法在一个非终止状态截断，评估函数只能猜测最后的结局，获胜的几率多大，平局的几率多大，失败的几率多大。
- 定义状态的类别，例如两兵对一兵的残局类别，同一类别中，有致胜的状态、导致平局的状态，导致失败的状态。评估算法无法知道会导致哪个结局。

evaluation functions {5.4.1}

- 某类中72%的状态是致胜的（效用值+1），20%是会输的（0），而其他8%是平局（½）。那么该类中状态的合理评价是**期望值**：$(0.72 \times +1) + (0.20 \times 0)+(0.08 \times ½) = 0.76$。总体上每个分类确定一个期望值，帮助产生任一状态的评估函数。----------类别过多，难以统计。

evaluation functions {5.4.1}

- 多数评估函数会分别计算每个特征的影响，然后把它们组合起来找到总数值
- 国际象棋各个棋子的**子力价值**估计：兵值1分，马和象值3分，车值5分，后值9分。其他特征诸如“是否好兵阵”和“王是否安全”可能值半个兵。评估函数可以是特征值的**线性组合**：

$$\textsc{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$$

evaluation functions {5.4.1}

- 线性组合基于过强的假设：每个特征的贡献是独立的。

- 采用非线性组合。两个象的价值>单个象的价值的两倍；残局中的象棋力值高于开始时的棋力值。

- 棋力值和组合权值来自经验规律，经验规律可通过机器学习获得。机器学习的结果：一个象值三个兵。

Cutoff search{5.4.2}

- 如何截断，以满足时间限制？

- 直接的方法：设置固定的深度限制，深度值设定必须符合游戏规则许可的时间。

- 更好的方法：迭代加深，直到时间用完。

Cutoff search{5.4.2}

- 评估函数是不准确的，截断可能导致错误，特别是在非静态棋局下（评估值出现大的摇摆）：
- 达到深度限制后，对于评估函数认为导致获胜的状态，只要再往前一步就可能会发现毫无悬念地通向失败。
- 有很好吃招的棋局对于只统计棋力的评估函数来说就不是静态的

Cutoff search{5.4.2}

- 评估函数只适用于静态棋局——评估值没有大的摇摆。
- 非静态棋局可以进一步扩展直到变为静态棋局。这种额外的搜索称为**静态搜索**

Cutoff search{5.4.2}

- **地平线效应**：对手招数导致我方严重损失并且从理论上基本无法避免时。我方的搜索深度为8，这个深度内，大部分选择都会导致象被吃掉，但有一个选择能通过牺牲两个兵而保住象，我方选择保住象的走法。但实际上超出搜索深度8，你会发现象一定会被吃掉，那么两个兵是白白牺牲的。实际上象被吃掉是不可避免的，只是超出了我方能看到的地平线

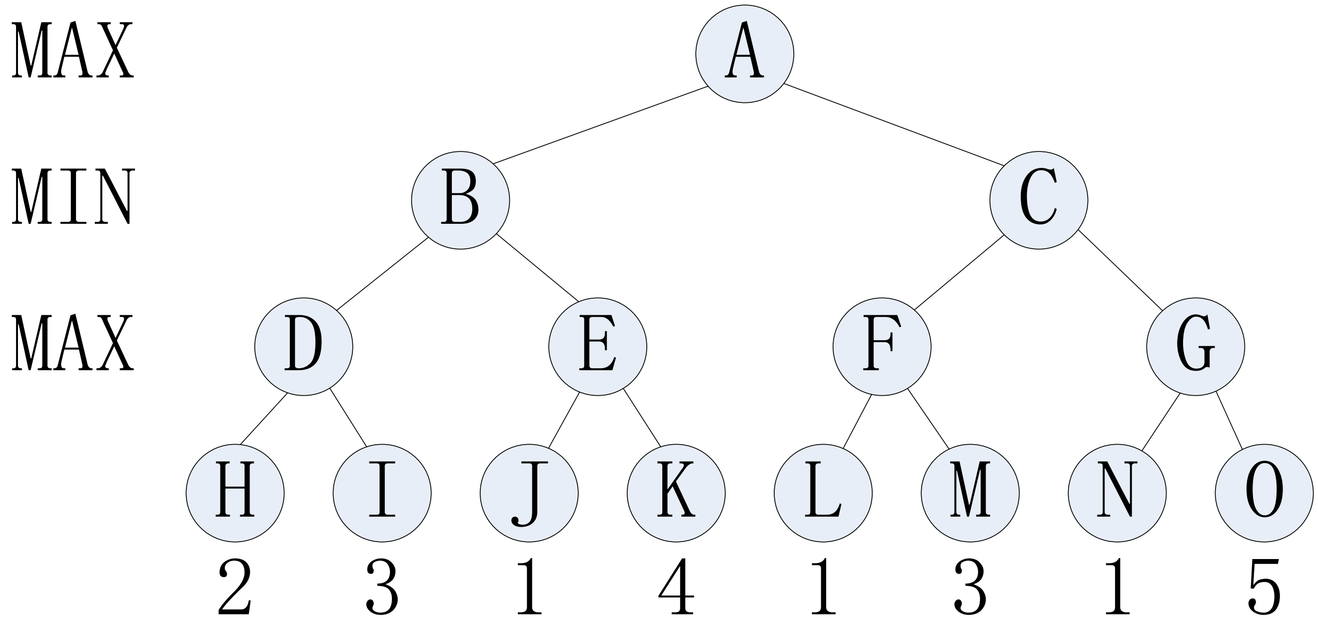
向前剪枝

- 除了截断搜索，还可使用向前剪枝。大多数人在下象棋的时候，对每个棋局只考虑部分行棋。

- 一种向前剪枝的方法是**柱搜索**：在每一层，只考虑最好的n步行棋（称为“柱”）可能，并不是考虑所有行棋招数。但无法保证最佳的行棋不被裁剪掉。

excercise

- exercise:



结点	B	C	D	E	F	G
从父节点获得的（α，β）值						
返回值						

Thanks!

next: Chapter 6 CSP