Game-Playing & Adversarial Search

Zahoor Tanoli (PhD)

COMSATS Attock

Overview

- **Minimax Search with Perfect Decisions**
  - Impractical in most cases, but theoretical basis for analysis

- **Minimax Search with Cut-off**
  - Replace terminal leaf utility by heuristic evaluation function

- **Alpha-Beta Pruning**
 - The fact of the adversary leads to an advantage in search!

Types of Games

	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	battleship Kriegspiel	bridge, poker, scrabble nuclear war

Not Considered: Physical games like tennis,
Football, ice hockey, etc.
(but see "robot soccer" http://www.robocup.org/)

Typical assumptions

- **Two agents whose actions alternate**

- **Utility values for each agent are the opposite of the other**
  - This creates the adversarial situation

- **Fully observable environments**

- **In game theory terms:**
  - "Deterministic, turn-taking, zero-sum games of perfect information"

Game tree (2-player, deterministic, turns)
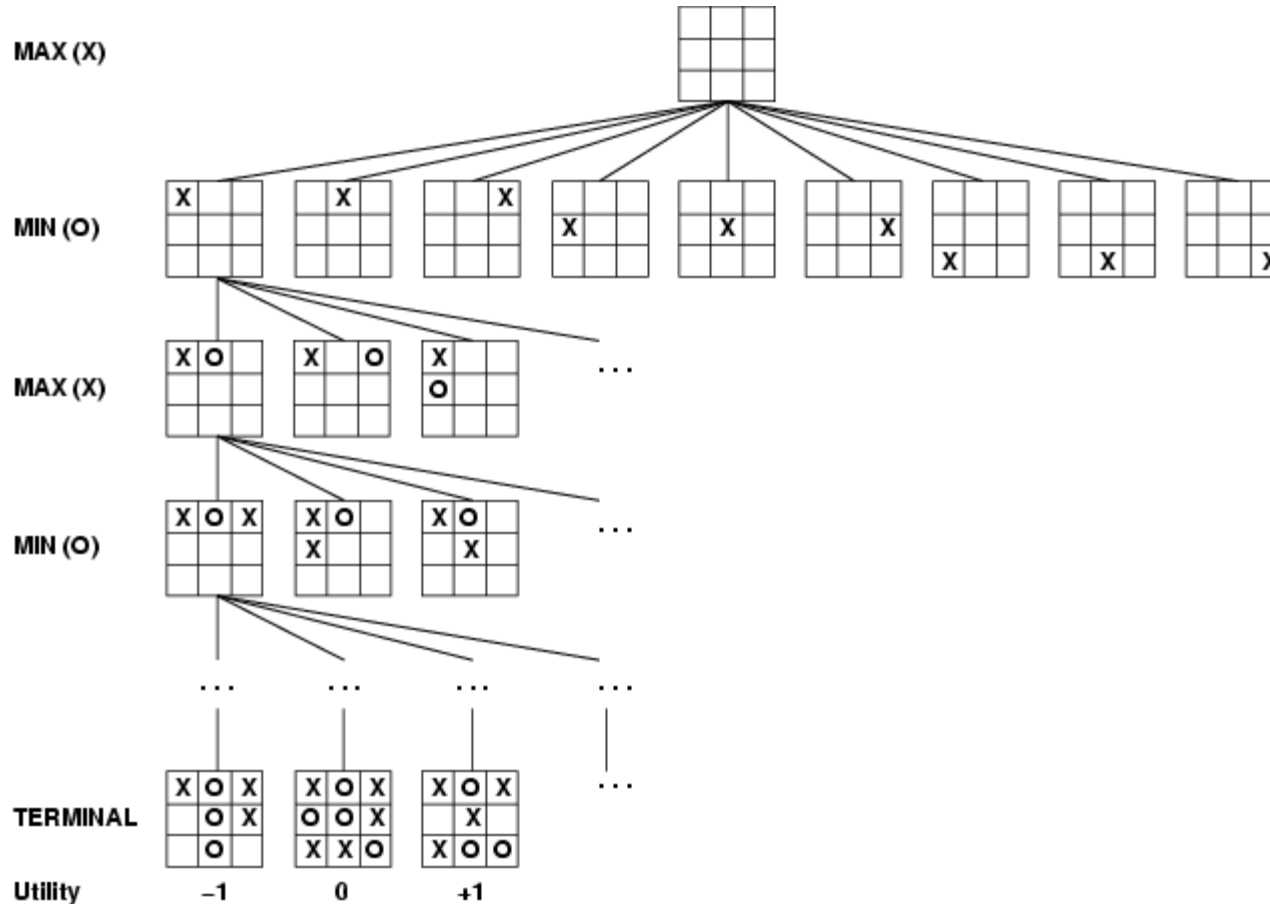
How do we search this tree to find the optimal move?

Search versus Games

- **Search – no adversary**
  - Solution is (heuristic) method for finding goal
  - Heuristics and CSP techniques can find *optimal* solution
  - Evaluation function: estimate of cost from start to goal through given node
  - Examples: path planning, scheduling activities

- **Games – adversary**
  - Solution is strategy
    - strategy specifies move for every possible opponent reply.
  - Time limits force an *approximate* solution
  - Evaluation function: evaluate "goodness" of game position
  - Examples: chess, checkers, Othello, backgammon

Games as Search

- **Two players: MAX and MIN**

- **MAX moves first and they take turns until the game is over**
  - Winner gets reward, loser gets penalty.
  - "Zero sum" means the sum of the reward and the penalty is a constant.

- **Formal definition as a search problem:**
  - **Initial state:** Set-up specified by the rules, e.g., initial board configuration of chess.
  - **Player(s):** Defines which player has the move in a state.
  - **Actions(s):** Returns the set of legal moves in a state.
  - **Result(s,a):** Transition model defines the result of a move.
  - (**2nd ed.: Successor function:** list of (move,state) pairs specifying legal moves.)
  - **Terminal-Test(s):** Is the game finished? True if finished, false otherwise.
  - **Utility function(s,p):** Gives numerical value of terminal state s for player p.
    - E.g., win (+1), lose (-1), and draw (0) in tic-tac-toe.
    - E.g., win (+1), lose (0), and draw (1/2) in chess.
- **MAX uses search tree to determine next move.**

An optimal procedure: The Min-Max method

Designed to find the optimal strategy for Max and find best move:

- **1. Generate the whole game tree, down to the leaves.**

- **2. Apply utility (payoff) function to each leaf.**

- **3. Back-up values from leaves through branch nodes:**
  - a Max node computes the Max of its child values
  - a Min node computes the Min of its child values

- **4. At root: choose the move leading to the child of highest value.**
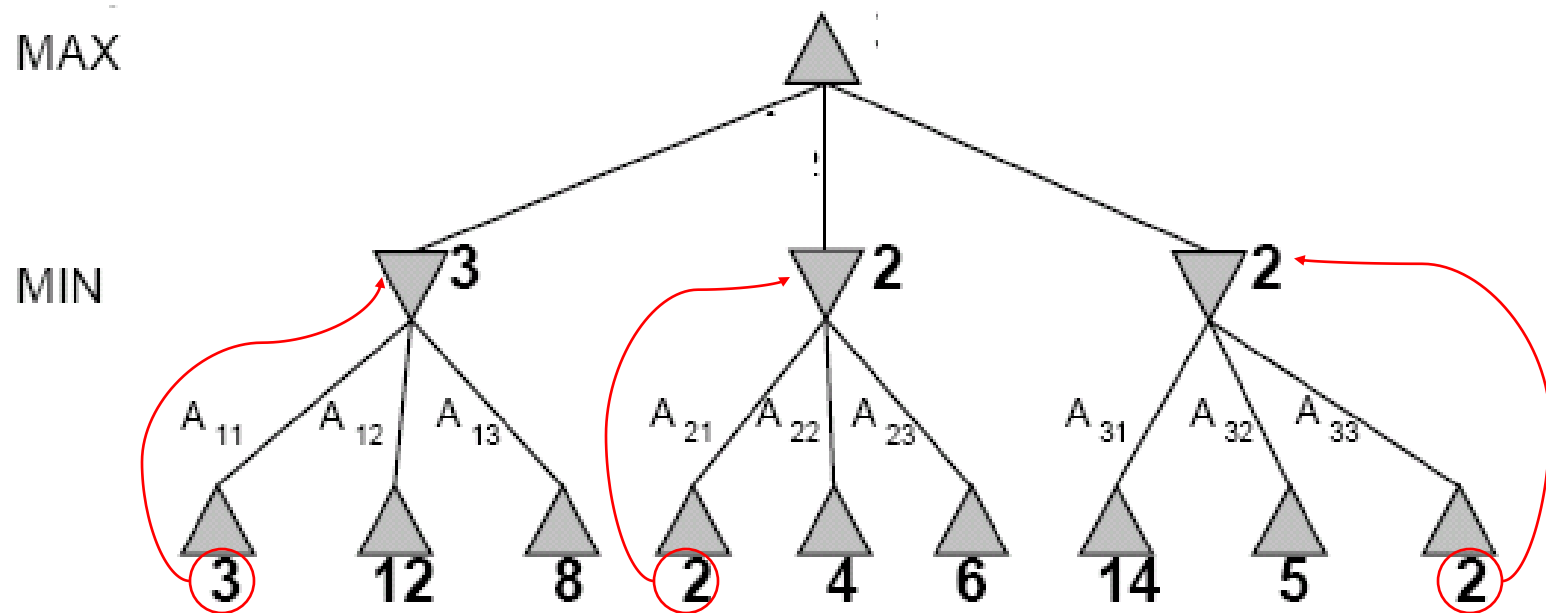
Game Trees
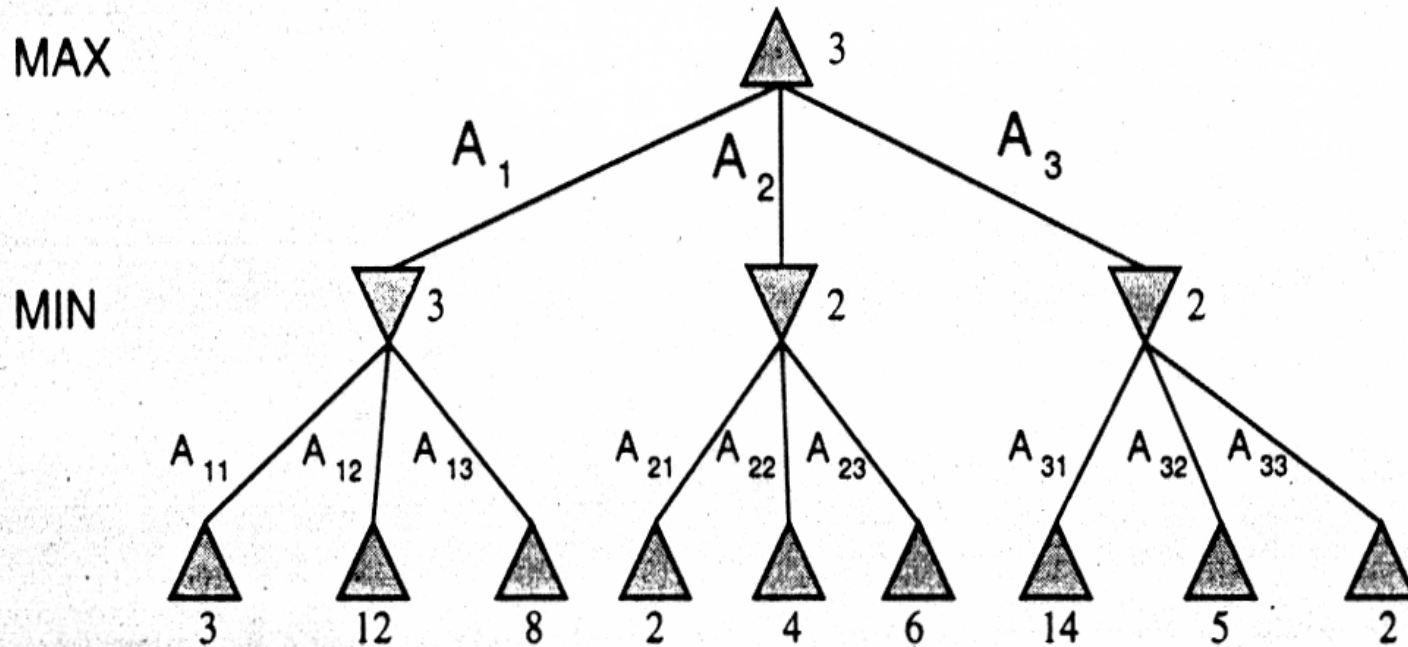
Figure 5.2 A two-ply game tree as generated by the minimax algorithm. The △ nodes are moves by MAX and the ▽ nodes are moves by MIN. The terminal nodes show the utility value for MAX computed by the utility function (i.e., by the rules of the game), whereas the utilities of the other nodes are computed by the minimax algorithm from the utilities of their successors. MAX's best move is A_1, and MIN's best reply is A_{11}.
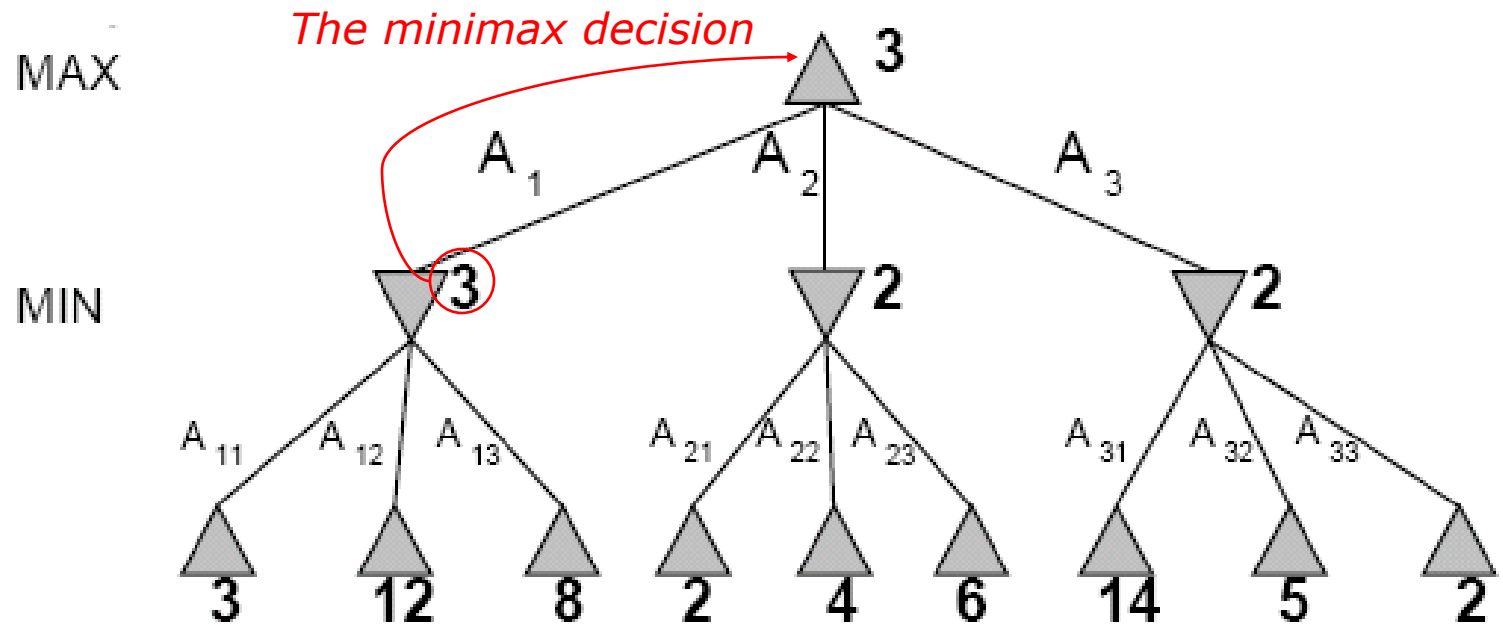
Two-Ply Game Tree

MAX

MIN

3   12   8   2   4   6   14   5   2
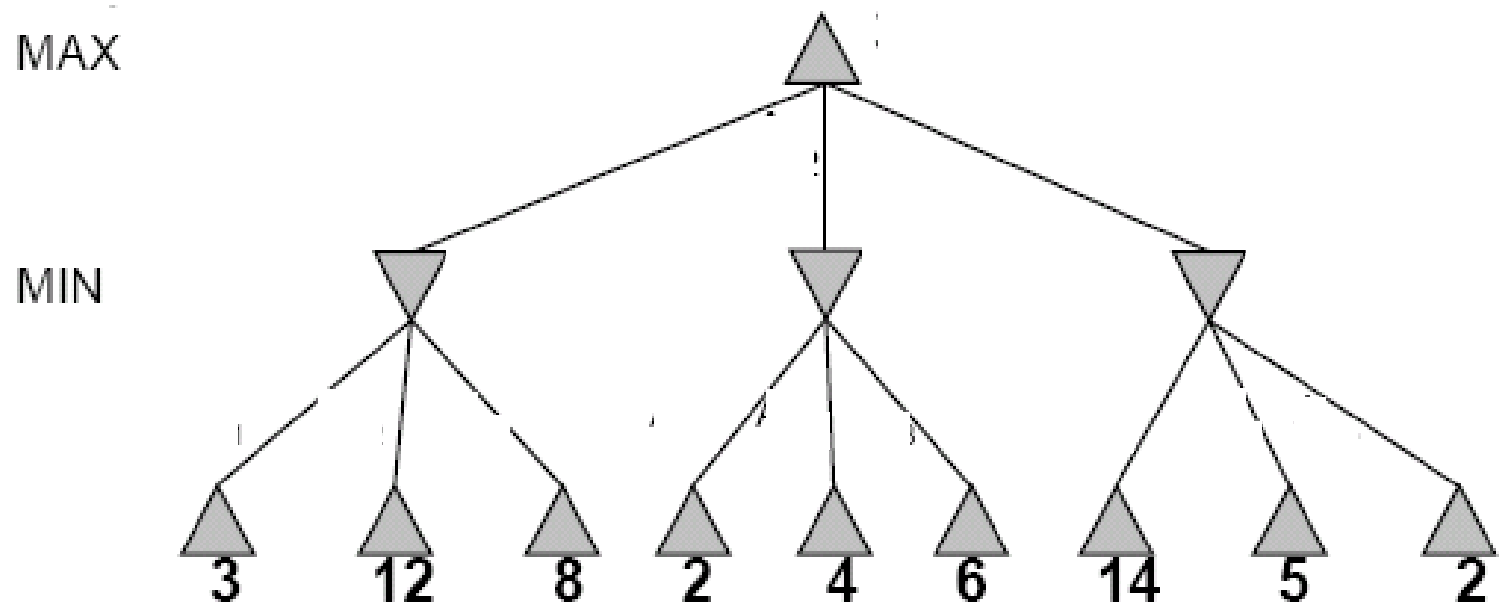
Two-Ply Game Tree

Two-Ply Game Tree

Minimax maximizes the utility for the worst-case outcome for max

The minimax decision

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

Properties of minimax

- ## Complete?
  - Yes (if tree is finite).

- ## Optimal?
  - Yes (against an optimal opponent).
  - Can it be beaten by an opponent playing sub-optimally?
    - No. (Why not?)

- ## Time complexity?
  - $O(b^m)$

- ## Space complexity?
  - O(bm)   (depth-first search, generate all actions at once)
  - O(m)   (backtracking search, generate actions one at a time)

Game Tree Size

- **Tic-Tac-Toe**
    - b ≈ 5 legal actions per state on average, total of 9 plies in game.
        - "ply" = one action by one player, "move" = two plies.
    - 5^9 = 1,953,125
    - 9! = 362,880 (Computer goes first)
    - 8! = 40,320 (Computer goes second)
    - → **exact solution quite reasonable**

- **Chess**
    - b ≈ 35 (approximate average branching factor)
    - d ≈ 100 (depth of game tree for "typical" game)
    - b^d ≈ 35^{100} ≈ 10^{154} nodes!!
    - → **exact solution completely infeasible**

- **It is usually impossible to develop the whole search tree.**

(Static) Heuristic Evaluation Functions

- **An Evaluation Function:**
  - Estimates how good the current board configuration is for a player.
  - Typically, evaluate how good it is for the player, how good it is for the opponent, then subtract the opponent's score from the player's.
  - Often called "static" because it is called on a static board position.
  - Othello: Number of white pieces - Number of black pieces
  - Chess: Value of all white pieces - Value of all black pieces

- Typical values from -infinity (loss) to +infinity (win) or [-1, +1].

- If the board evaluation is X for a player, it's -X for the opponent
  - "Zero-sum game"

Alpha-Beta Pruning
Exploiting the Fact of an Adversary

- **If a position is provably bad:**
    - It is NO USE expanding search time to find out exactly how bad

- **If the adversary can force a bad position:**
    - It is NO USE expanding search time to find out the good positions that the adversary won't let you achieve anyway

- **Bad = not better than we already know we can achieve elsewhere.**

- **Contrast normal search:**
    - ANY node might be a winner.
    - ALL nodes must be considered.
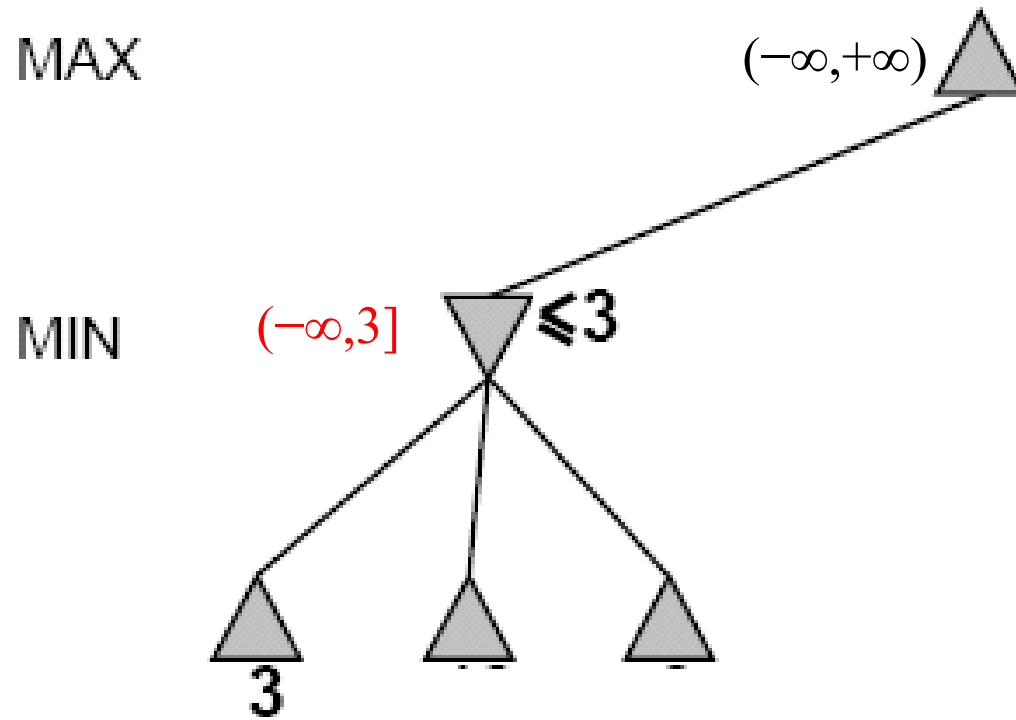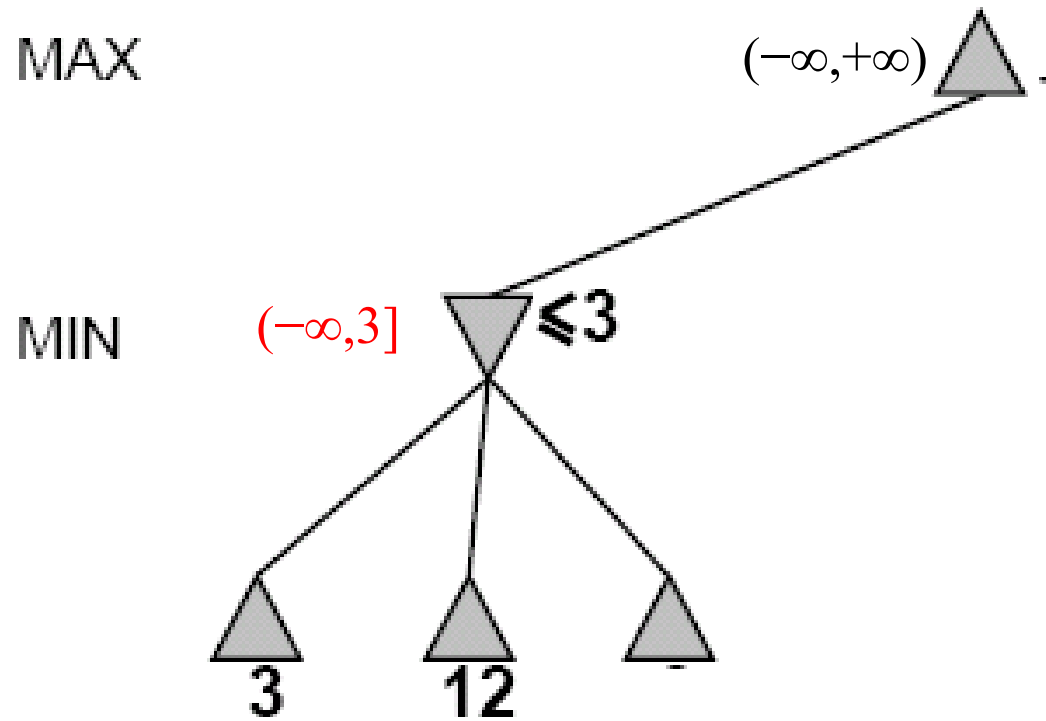   - (A* avoids this through knowledge, i.e., heuristics)

Another Alpha-Beta Example
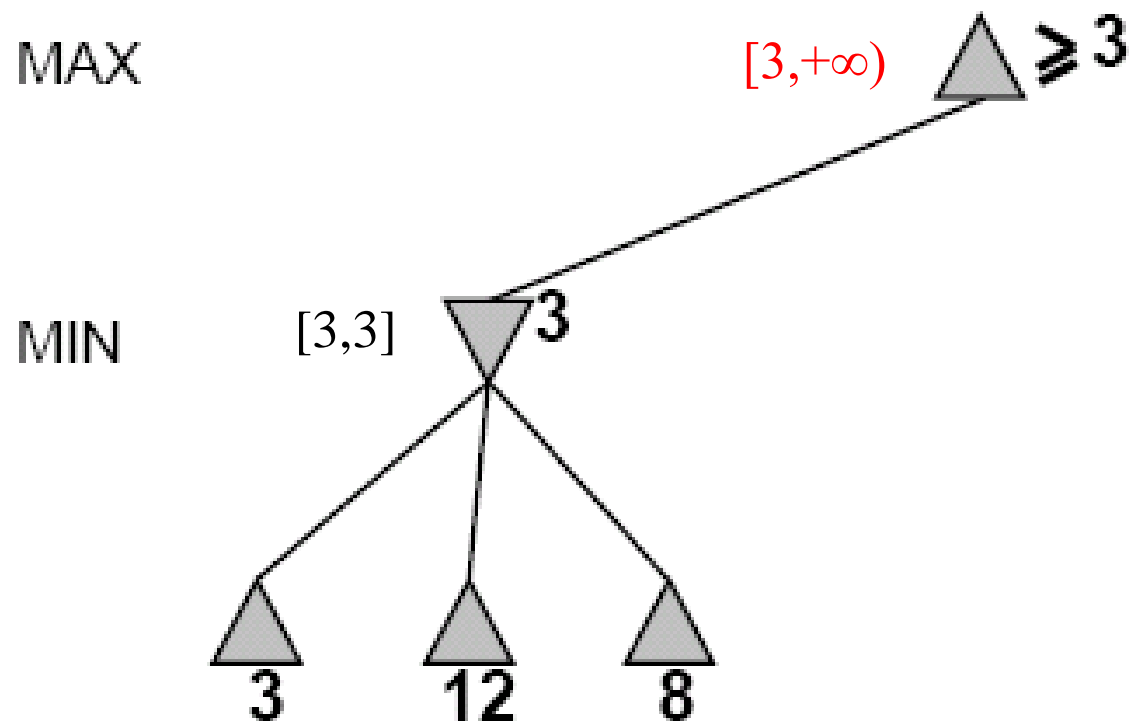
Do DF-search until first leaf
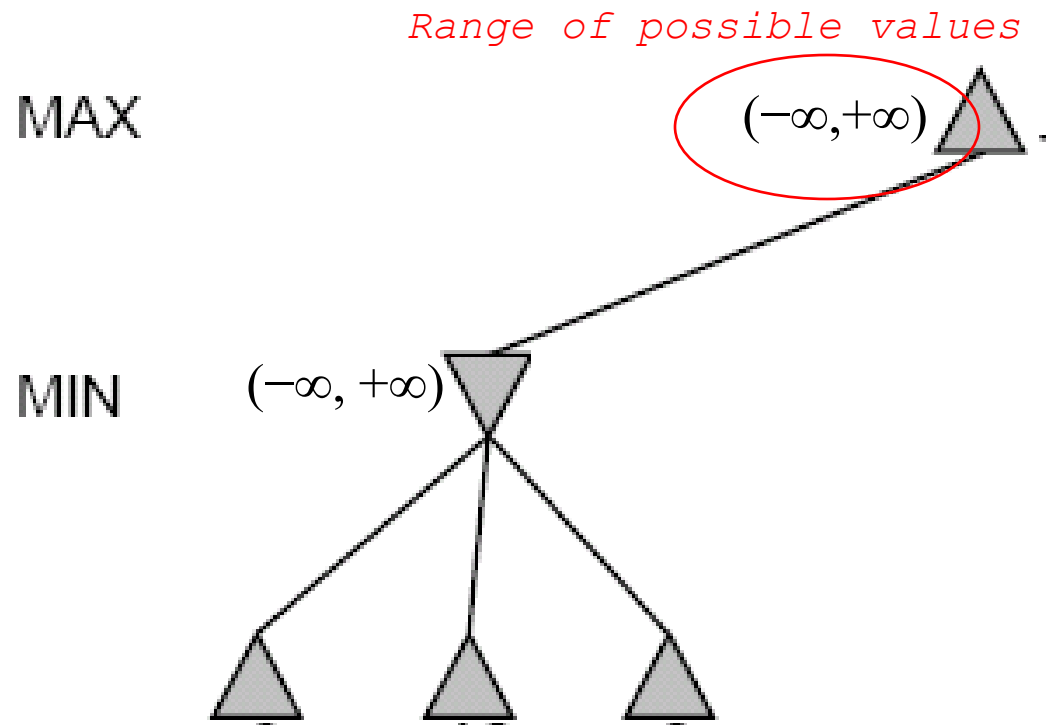
Range of possible values

MAX $(-\infty,+\infty)$

MIN $(-\infty, +\infty)$
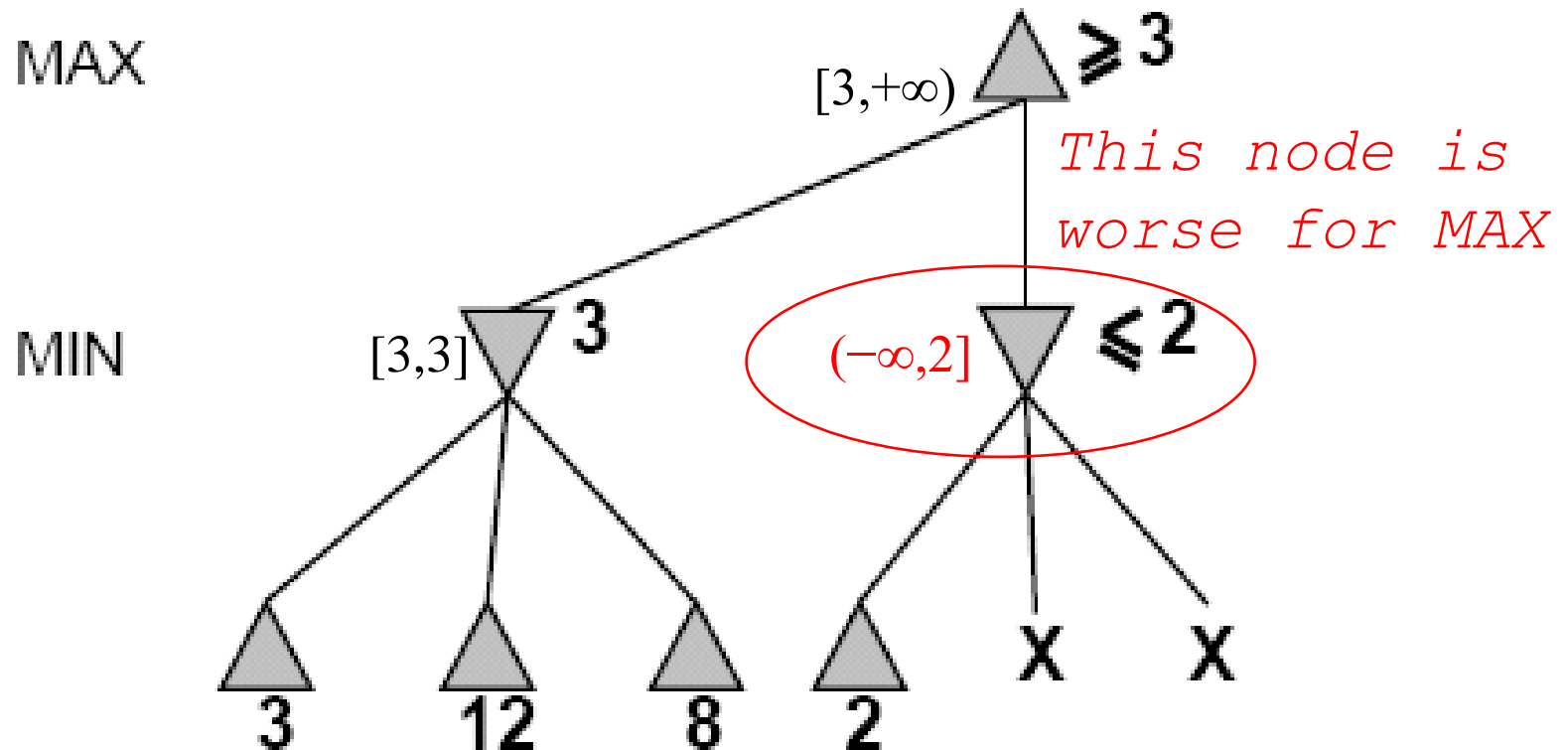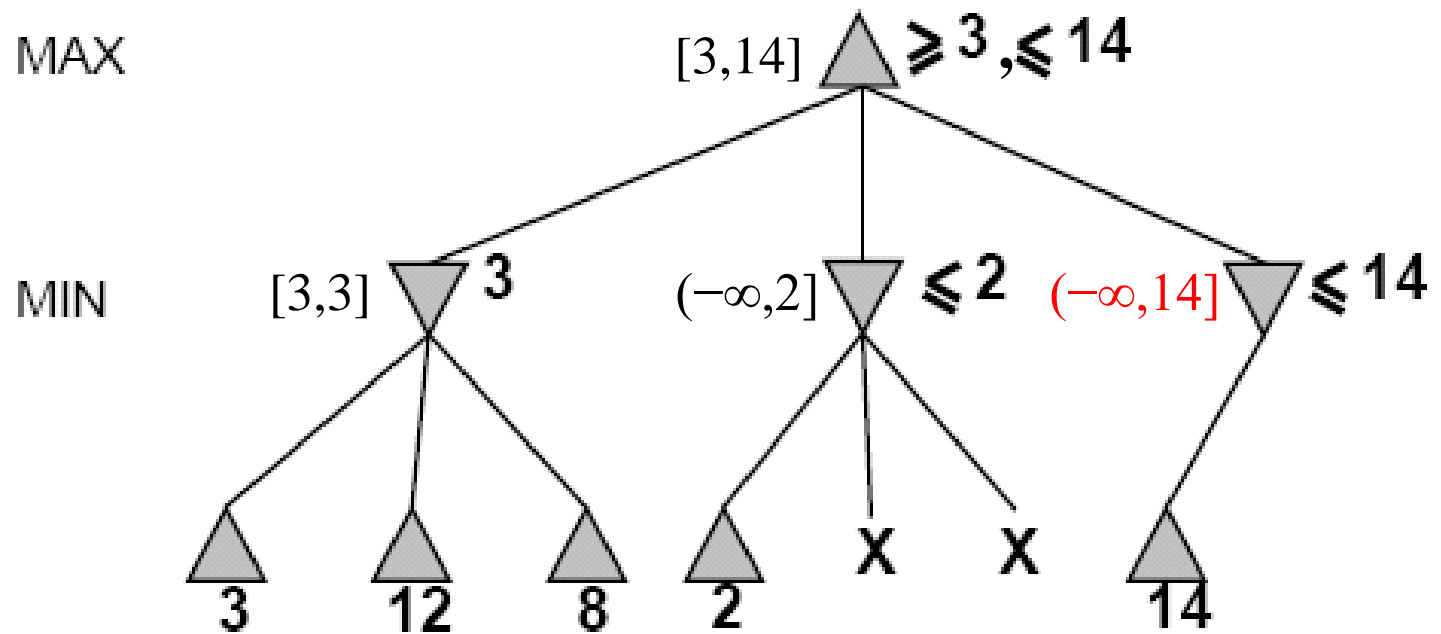
Alpha-Beta Example (continued)

Alpha-Beta Example (continued)

Alpha-Beta Example (continued)

MAX   [3,+∞)   ≥3

MIN   [3,3]   3

3   12   8

Alpha-Beta Example (continued)

Alpha-Beta Example (continued)

MAX $\quad$ [3,14] $\quad \geqslant$ 3 ,$\leqslant$ 14

MIN $\quad$ [3,3] $\quad$ 3 $\qquad (-\infty,2] \quad \leqslant$ 2 $\quad (-\infty,14] \quad \leqslant$ 14
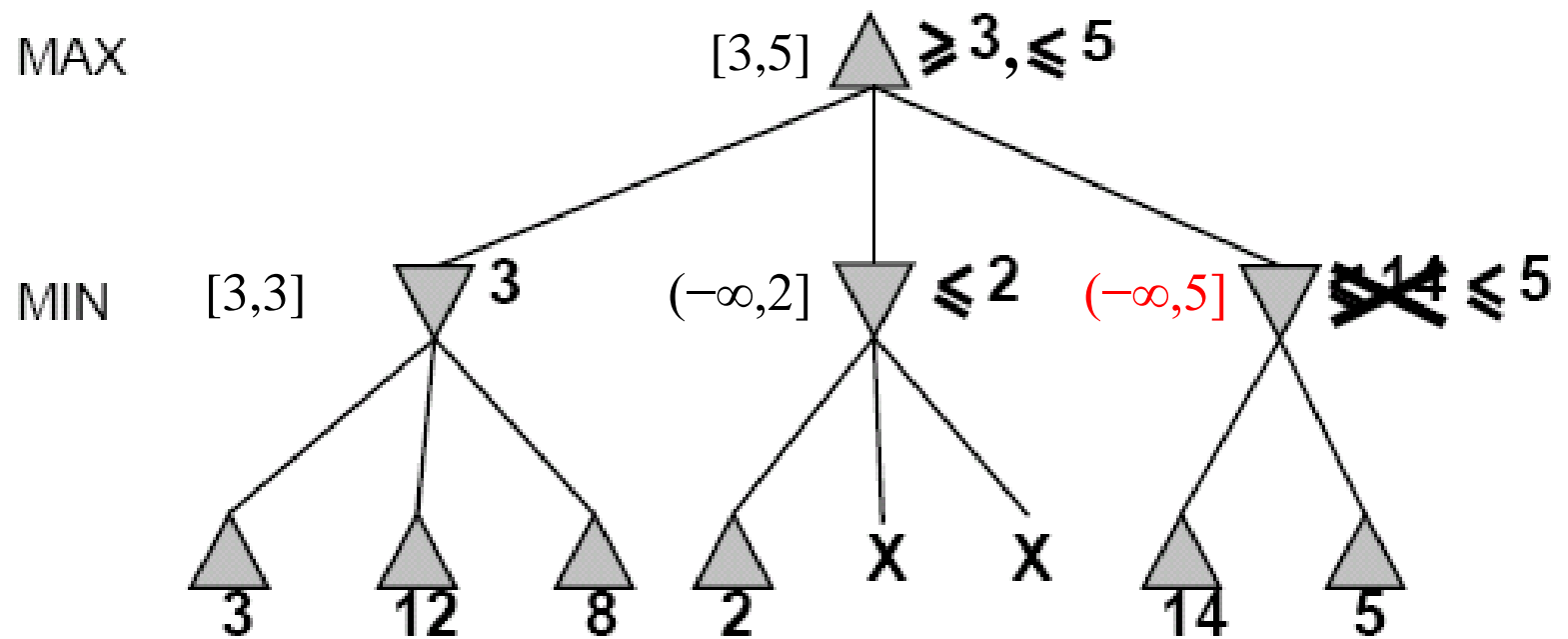
3 $\qquad$ 12 $\qquad$ 8 $\qquad$ 2 $\qquad$ X $\qquad$ X $\qquad$ 14
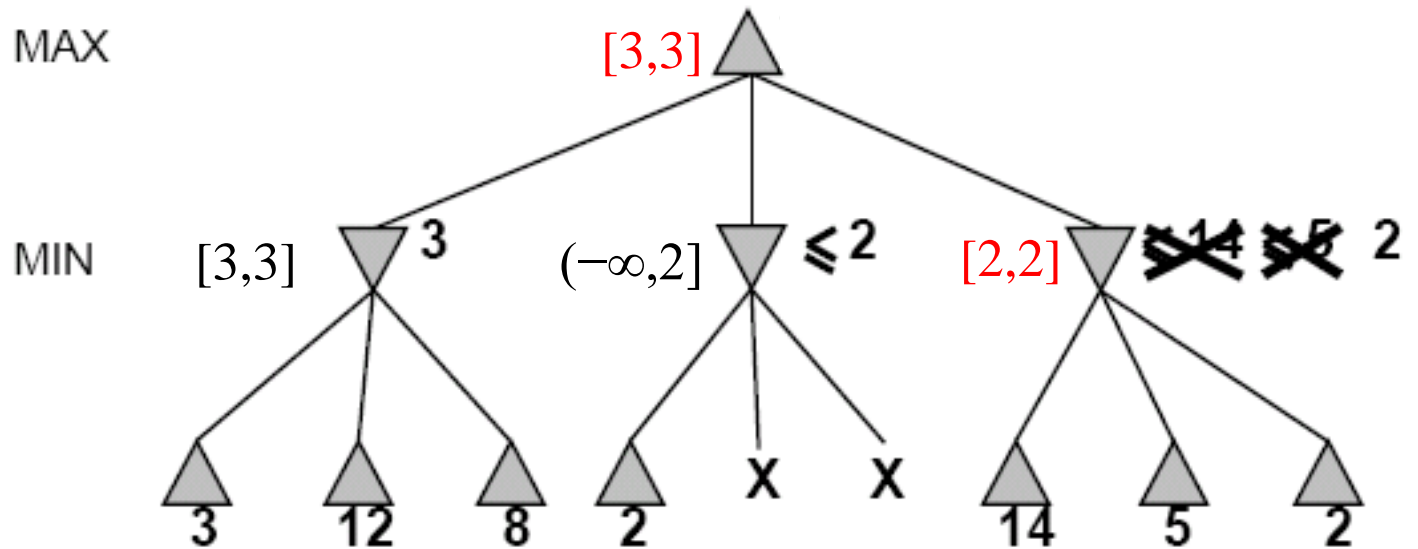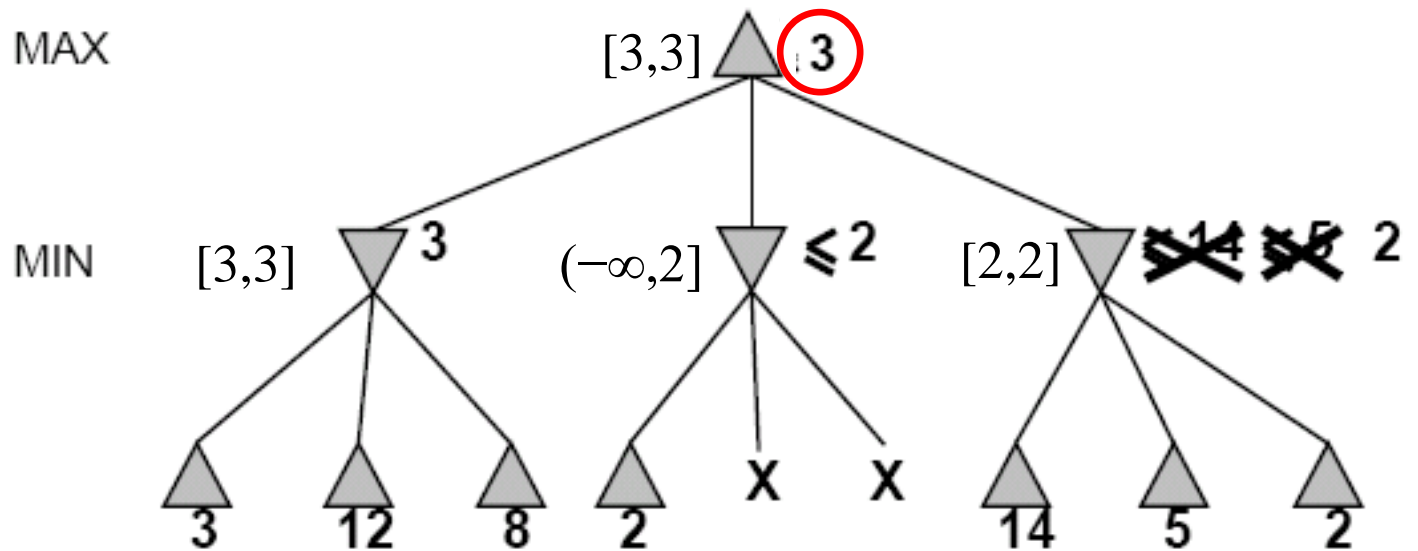
Alpha-Beta Example (continued)

Alpha-Beta Example (continued)

Alpha-Beta Example (continued)

Alpha-beta Algorithm

- **Depth first search**
  - only considers nodes along a single path from root at any time

α = **highest-value choice found at any choice point of path for MAX**
(initially, α = −infinity)

β = **lowest-value choice found at any choice point of path for MIN**
(initially, β = +infinity)

- **Pass current values of α and β down to child nodes during search.**
- **Update values of α and β during search:**
  - MAX updates α at MAX nodes
  - MIN updates β at MIN nodes
- **Prune remaining branches at a node when $\alpha \geq \beta$**

When to Prune

- **Prune whenever $\alpha \geq \beta$**

  - **Prune below a Max node whose alpha value becomes greater than or equal to the beta value of its ancestors.**
    - **Max nodes update alpha based on children's returned values.**

  - **Prune below a Min node whose beta value becomes less than or equal to the alpha value of its ancestors.**
    - **Min nodes update beta based on children's returned values.**

Pseudocode for Alpha-Beta Algorithm

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

Pseudocode for Alpha-Beta Algorithm

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 ACTIONS(*state*) with value *v*

---

function MAX-VALUE(*state*, α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for *a* in ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v,MIN-VALUE(Result(s,a), α, β))
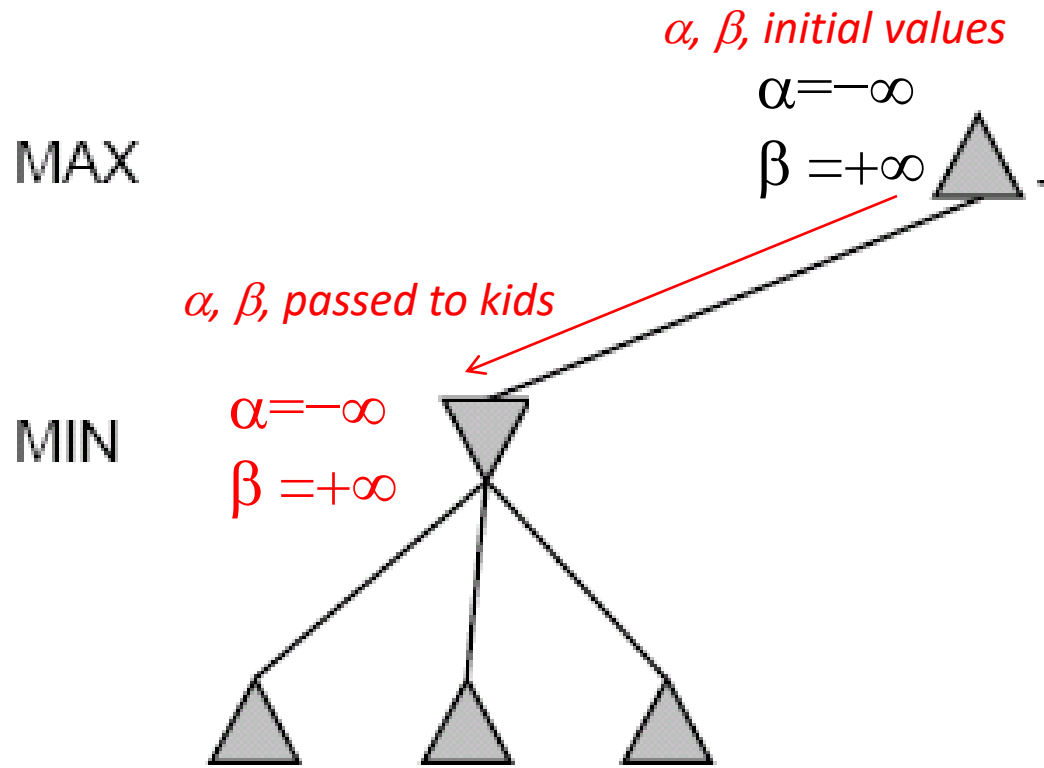   **if** $v \geq \beta$ **then return** v
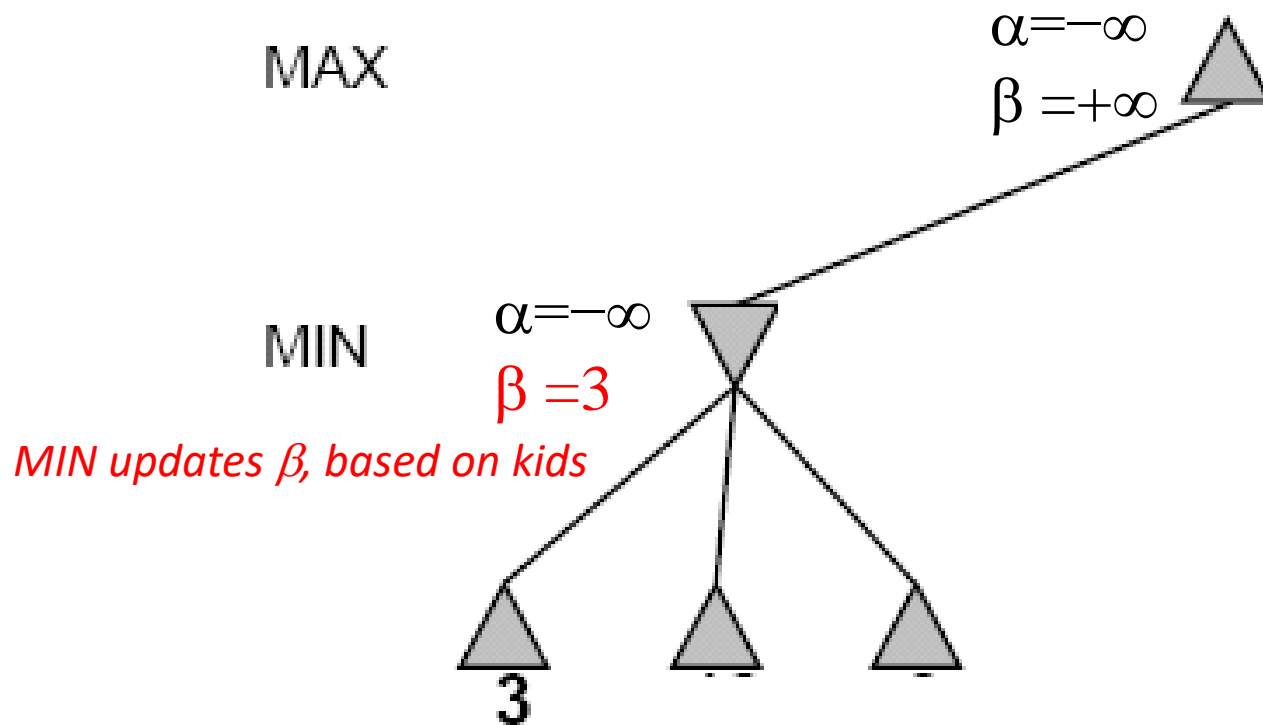 $\alpha \leftarrow$ MAX(α,v)
  return v

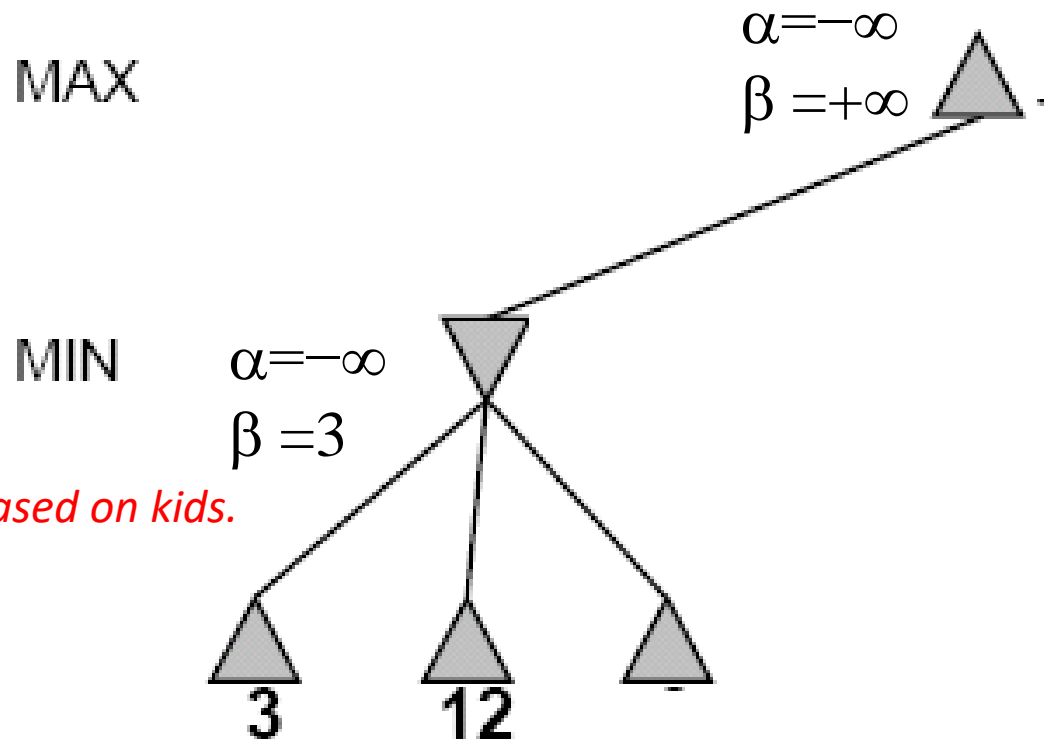(MIN-VALUE is defined analogously)

Alpha-Beta Example Revisited

Do DF-search until first leaf

α, β, initial values

$\alpha = -\infty$

MAX

$\beta = +\infty$

α, β, passed to kids

$\alpha = -\infty$

MIN

$\beta = +\infty$

Alpha-Beta Example (continued)

MAX $\quad \alpha = -\infty$
$\beta = +\infty$

MIN $\quad \alpha = -\infty$
$\beta = 3$

MIN updates β, based on kids

3

Alpha-Beta Example (continued)

MAX

$\alpha = -\infty$

$\beta = +\infty$

MIN

$\alpha = -\infty$

$\beta = 3$

MIN updates β, based on kids.
No change.

3   12

Alpha-Beta Example (continued)

MAX updates α, based on kids.

$\alpha=3$

$\beta=+\infty$

MAX

MIN

3

3 is returned as node value.

3   12   8

Alpha-Beta Example (continued)

MAX

$\alpha=3$
$\beta =+\infty$

$\alpha, \beta, passed\ to\ kids$

$\alpha=3$
$\beta =+\infty$

MIN

3

3   12   8

Alpha-Beta Example (continued)

MAX

MIN

$\alpha=3$
$\beta =+\infty$

MIN updates β,
based on kids.

$\alpha=3$
$\beta =2$

3

3   12   8   2

Alpha-Beta Example (continued)

MAX

$\alpha=3$
$\beta =+\infty$

MIN

3

$\alpha=3$
$\beta =2$

$\alpha \geq \beta,$
so prune.

3   12   8   2   X   X

Alpha-Beta Example (continued)

MAX

MIN

MAX updates α, based on kids.
No change.

$\alpha=3$
$\beta =+\infty$

3

$\leqslant 2$

2 is returned
as node value.

3   12   8   2   X   X

Alpha-Beta Example (continued)

Alpha-Beta Example (continued)

MAX

$\alpha=3$
$\beta=+\infty$

MIN updates β, based on kids.

$\alpha=3$
$\beta=14$

MIN

3

≤ 2

3   12   8   2   X   X   14

Alpha-Beta Example (continued)

Alpha-Beta Example (continued)

MAX

$\alpha = 3$

$\beta = +\infty$

2 is returned as node value.

MIN

3

≤ 2

2

X   X

3   12   8   2   14   5   2

Alpha-Beta Example (continued)

Max calculates the same node value, and makes the same move!
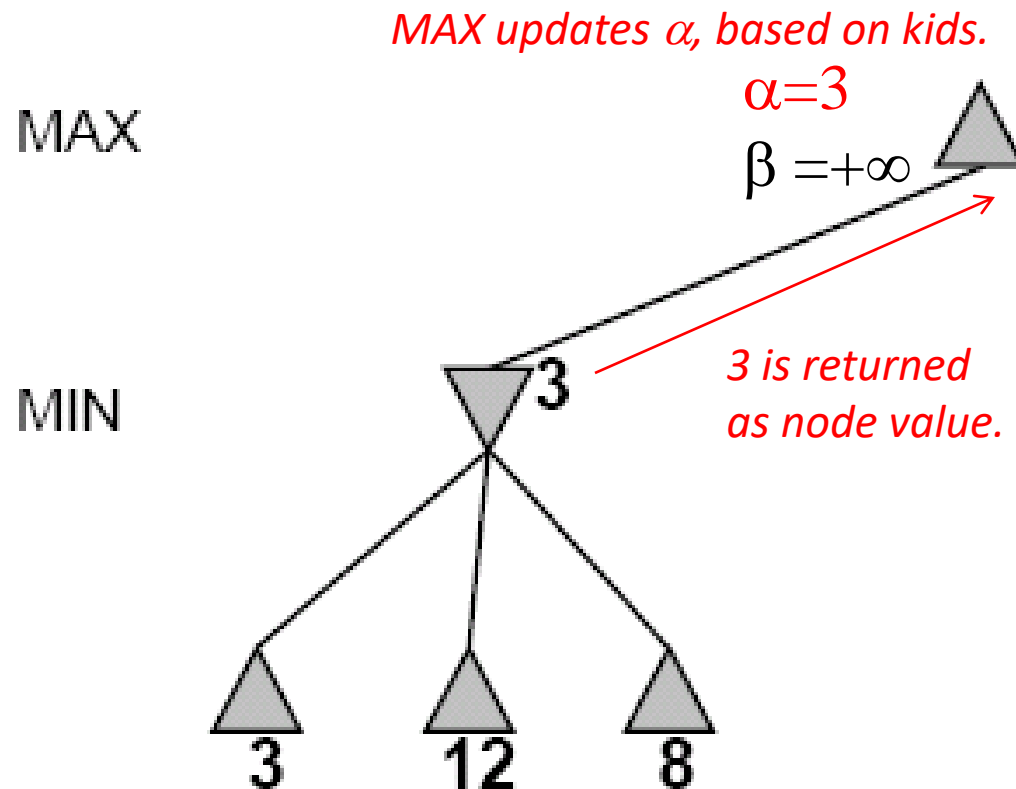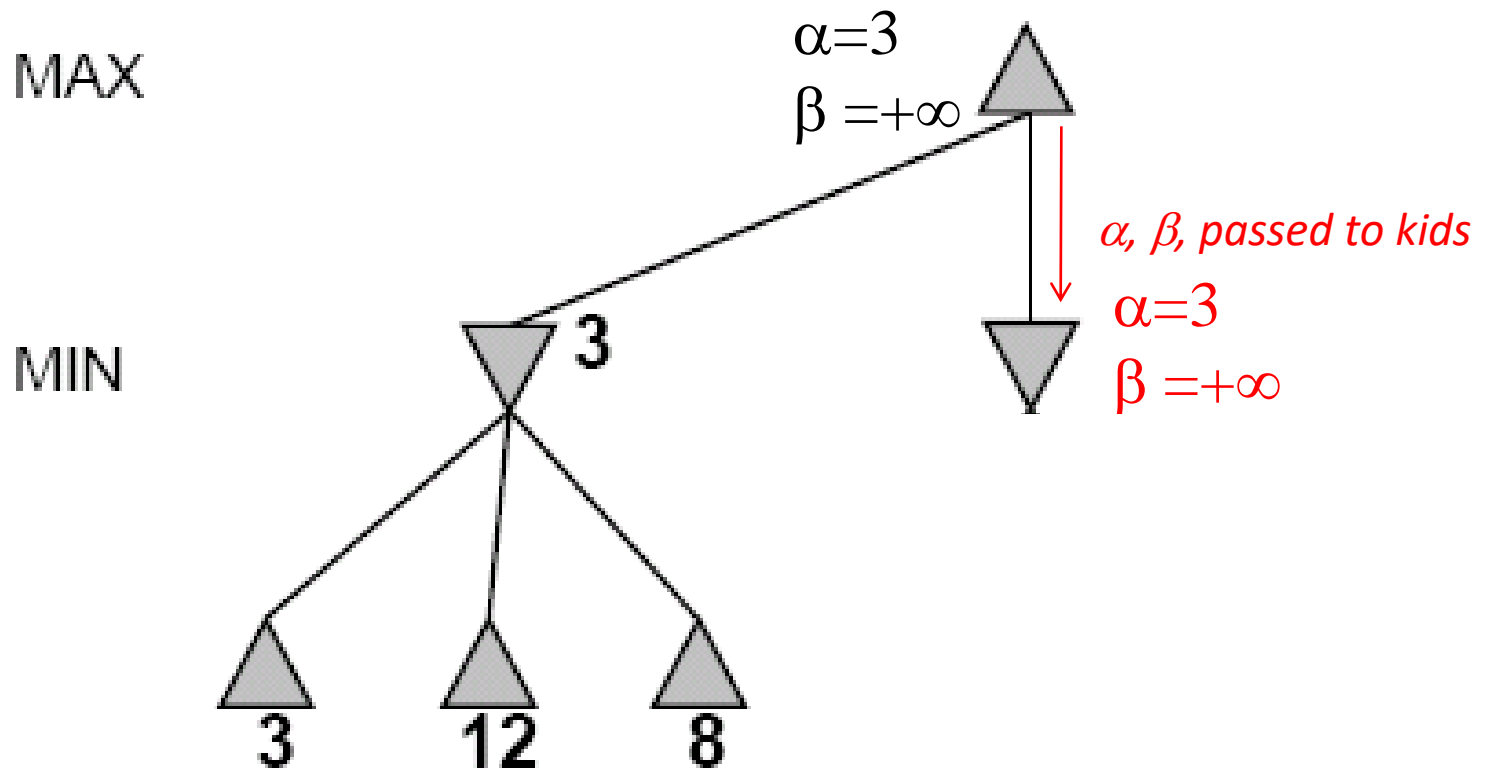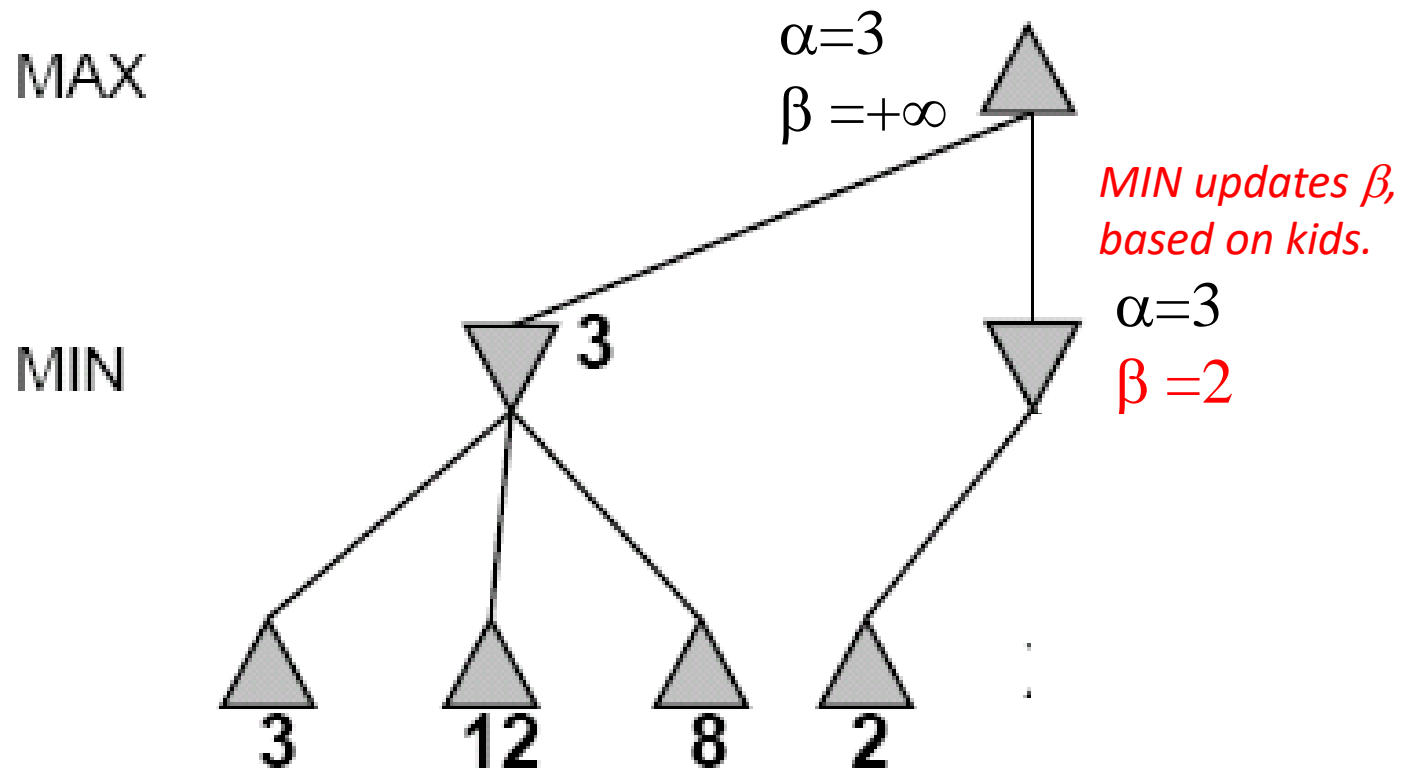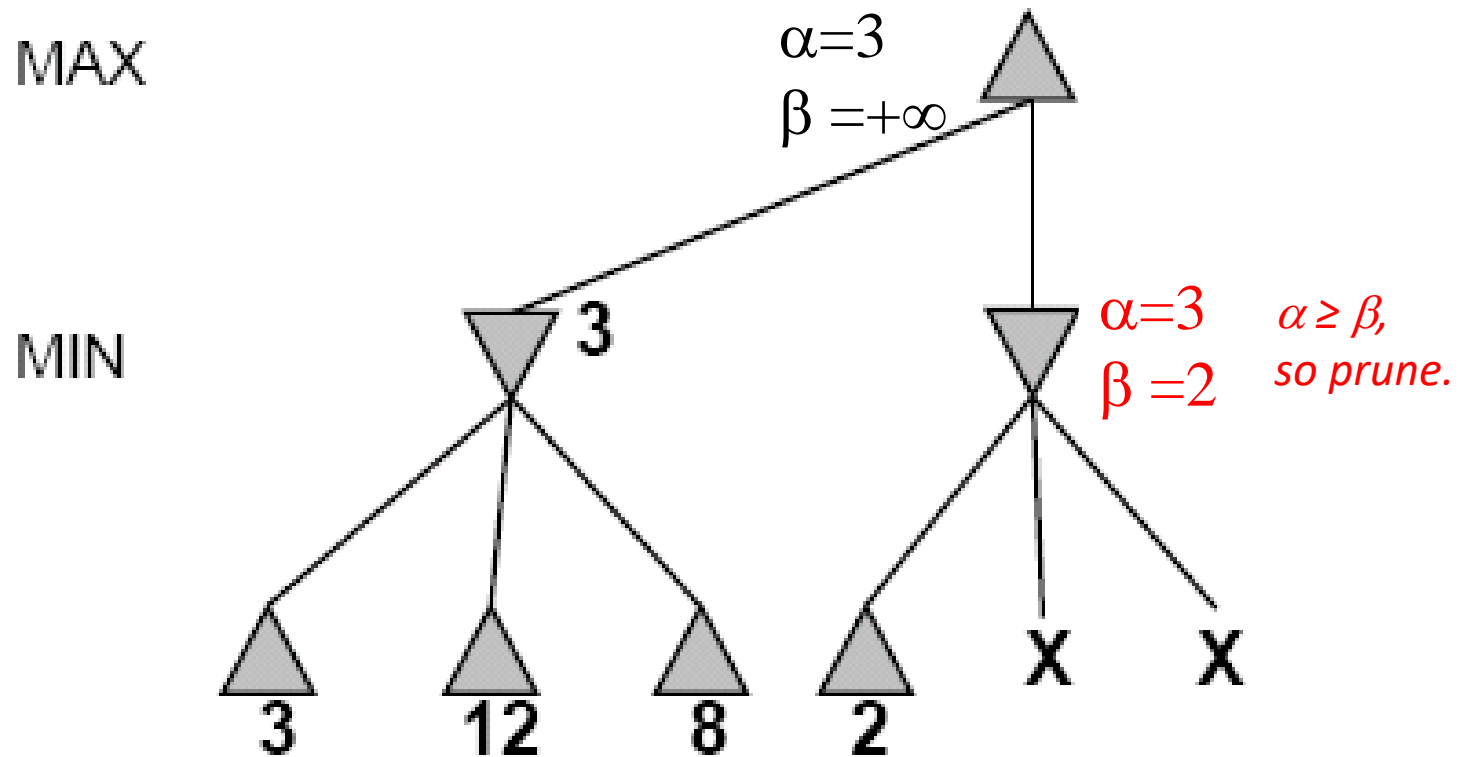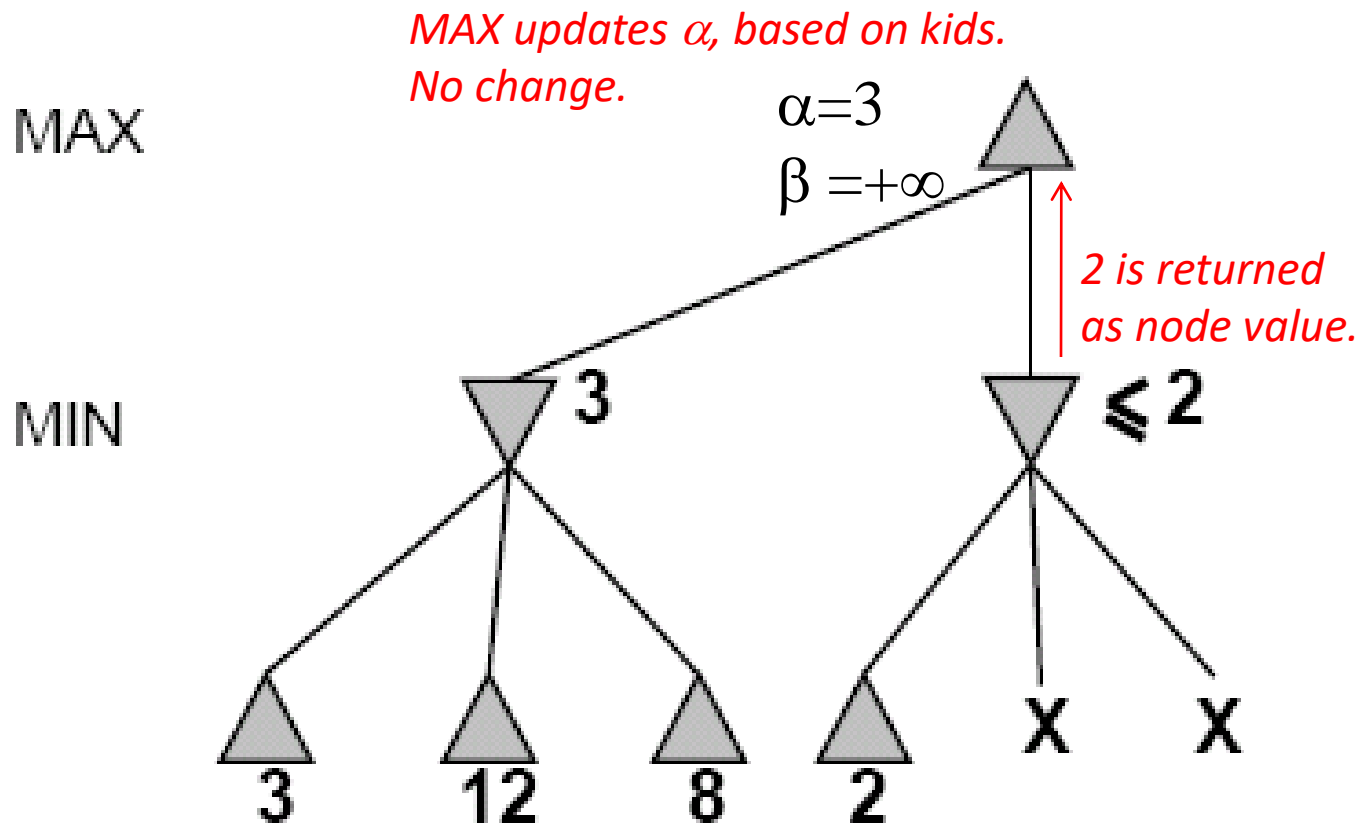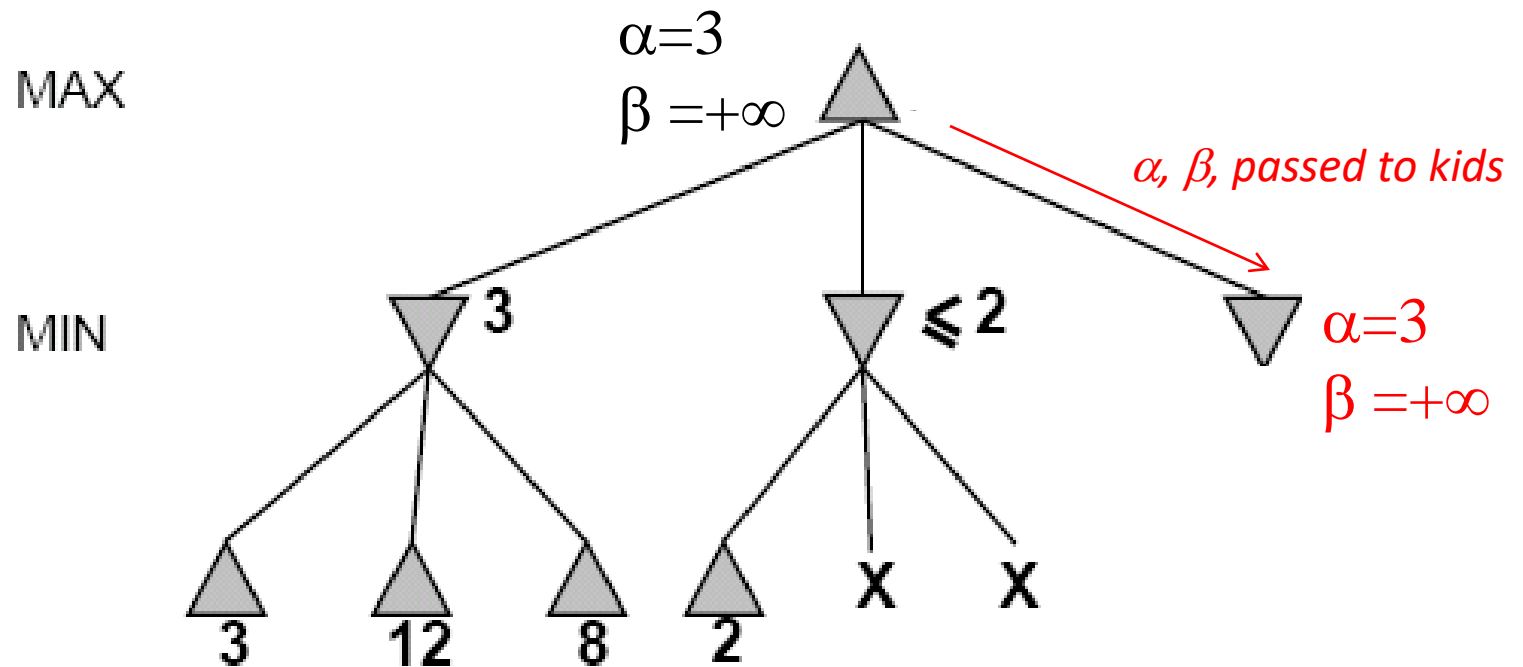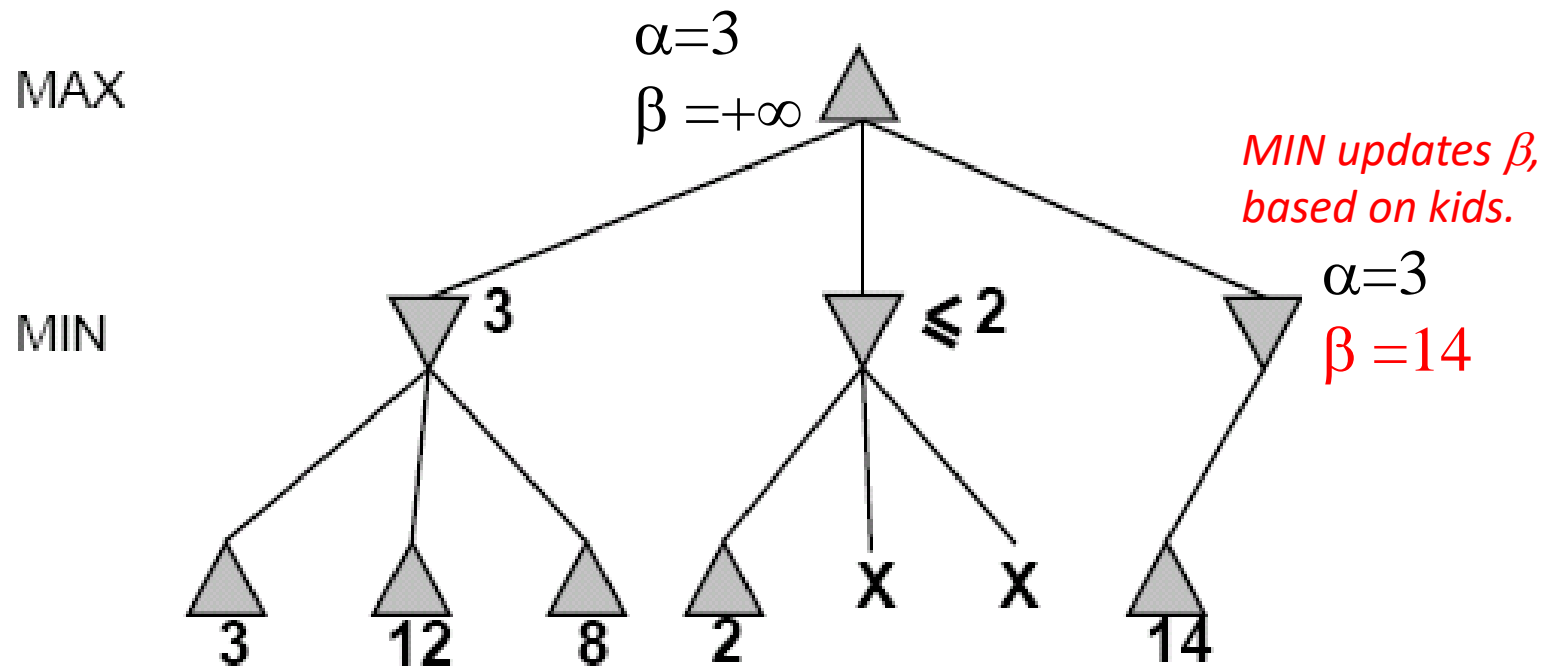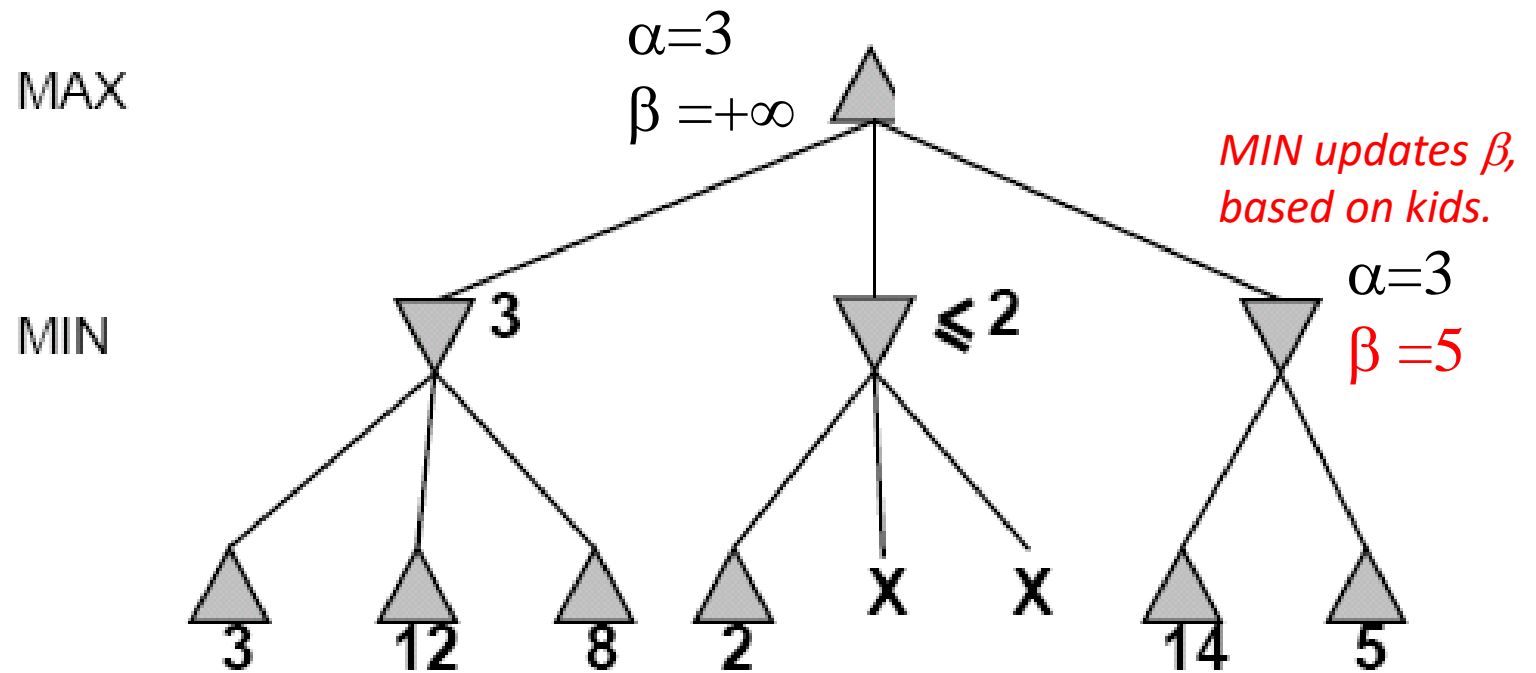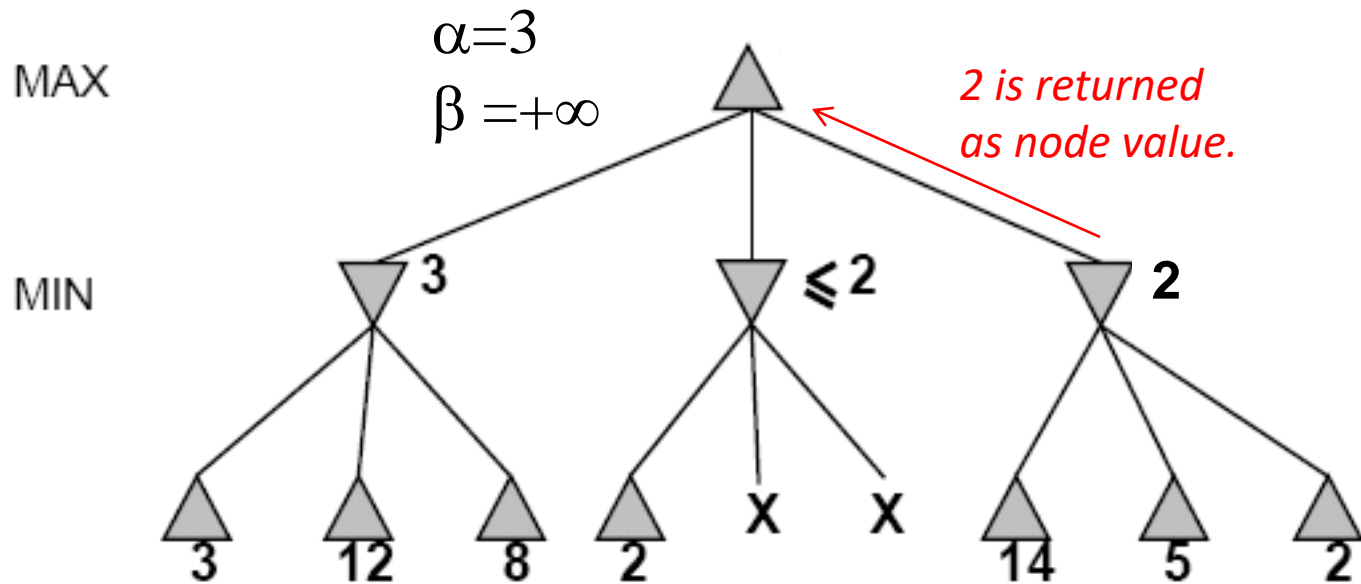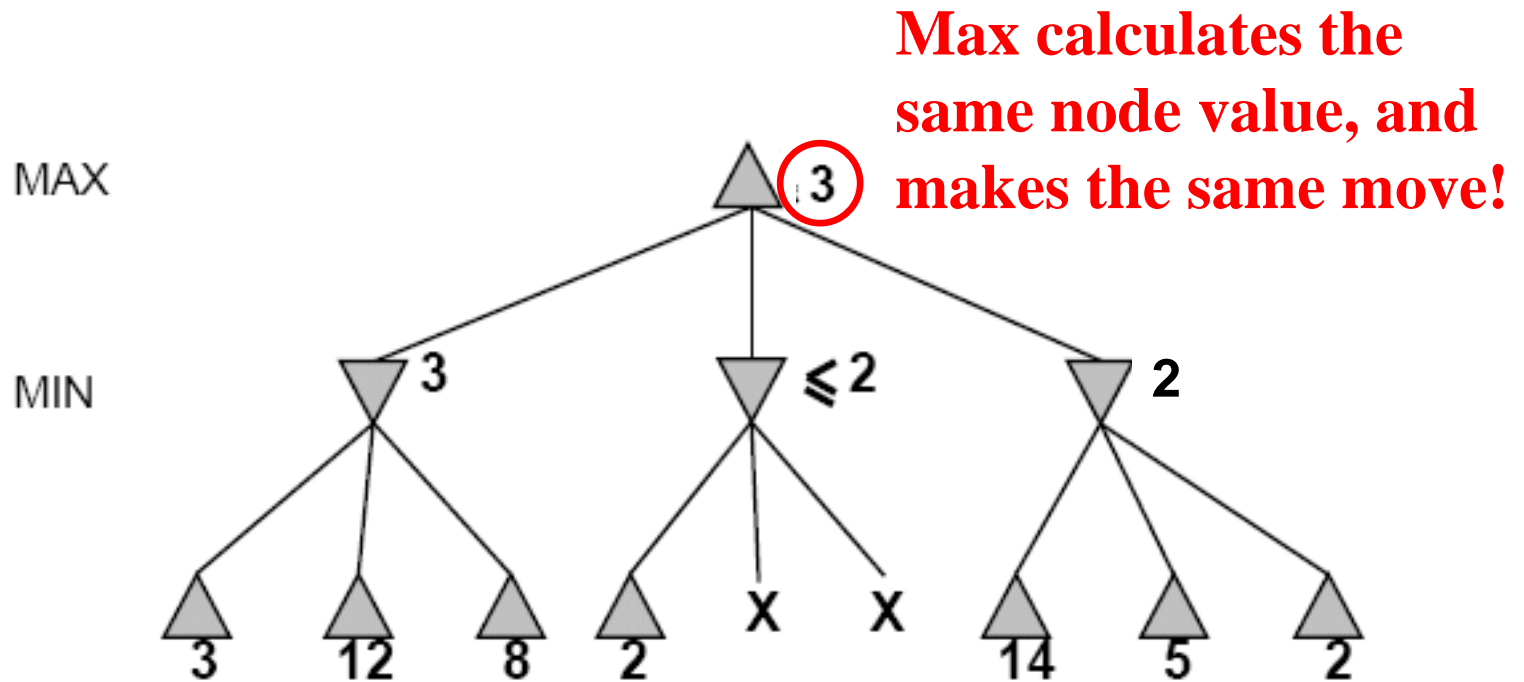
Effectiveness of Alpha-Beta Search

- **Worst-Case**
  - branches are ordered so that no pruning takes place. In this case alpha-beta gives no improvement over exhaustive search

- **Best-Case**
  - each player's best move is the left-most child (i.e., evaluated first)
  - in practice, performance is closer to best rather than worst-case
  - E.g., sort moves by the remembered move values found last time.
  - E.g., expand captures first, then threats, then forward moves, etc.
  - E.g., run Iterative Deepening search, sort by value last iteration.

- **In practice often get $O(b^{(d/2)})$ rather than $O(b^d)$**
  - this is the same as having a branching factor of sqrt(b),
    - $(sqrt(b))^d = b^{(d/2)}$, i.e., we effectively go from b to square root of b
  - e.g., in chess go from b ~ 35 to b ~ 6
    - this permits much deeper search in the same amount of time
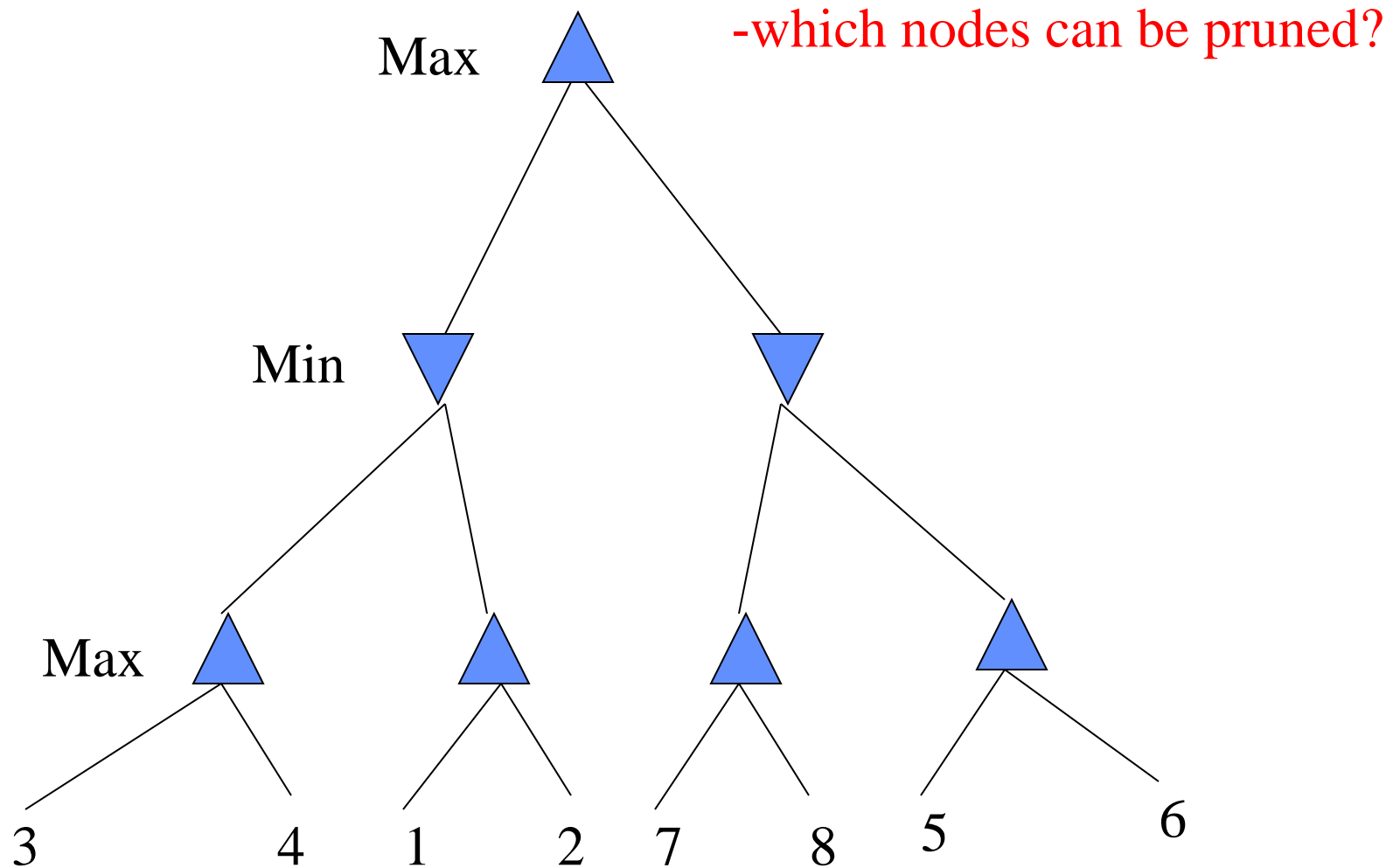
Final Comments about Alpha-Beta Pruning

- **Pruning does not affect final results**

- **Entire subtrees can be pruned.**

- **Good move *ordering* improves effectiveness of pruning**

Example

-which nodes can be pruned?

Answer to Example

Max

Min

Max

-which nodes can be pruned?

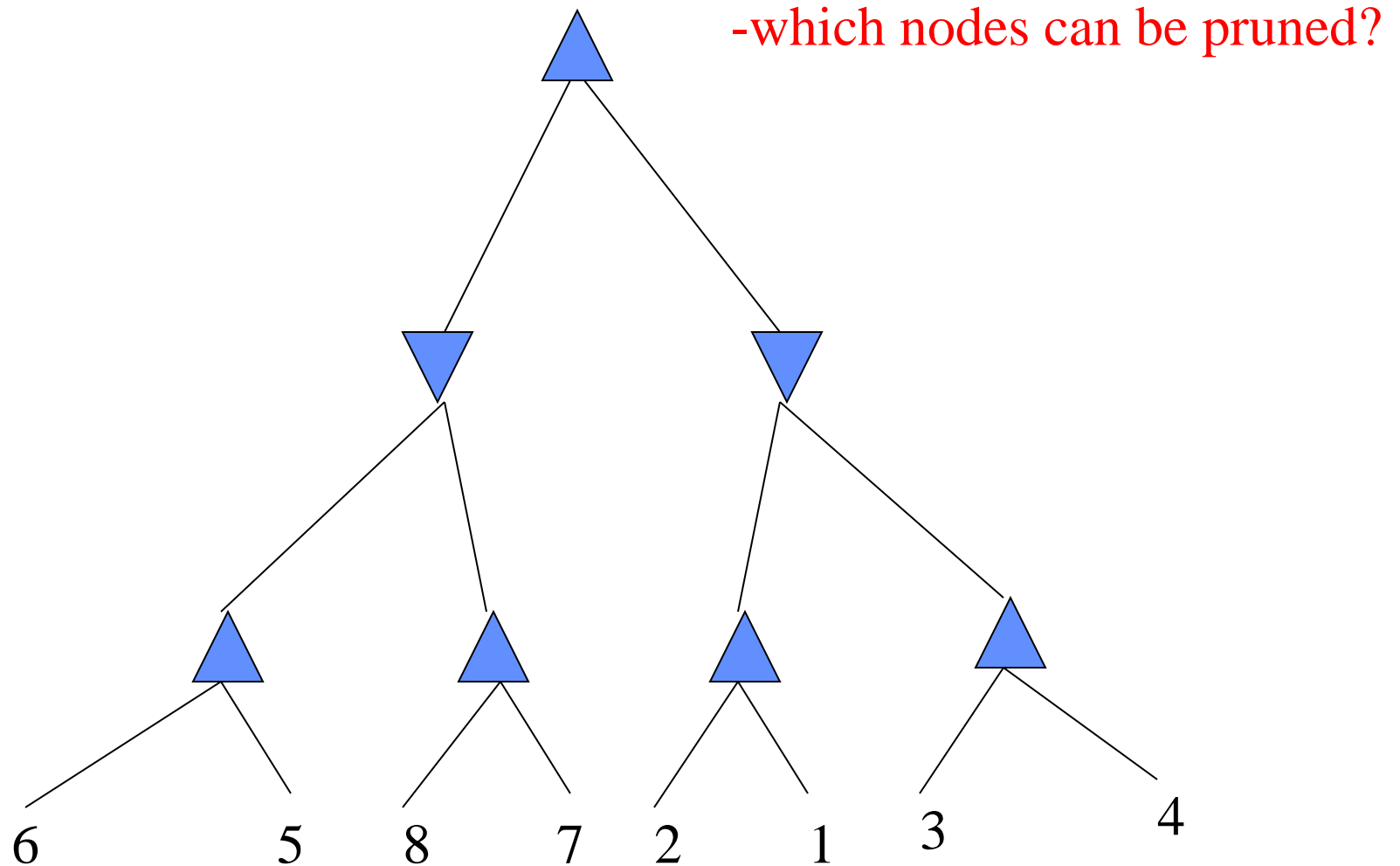3   4   1   2   7   8   5   6
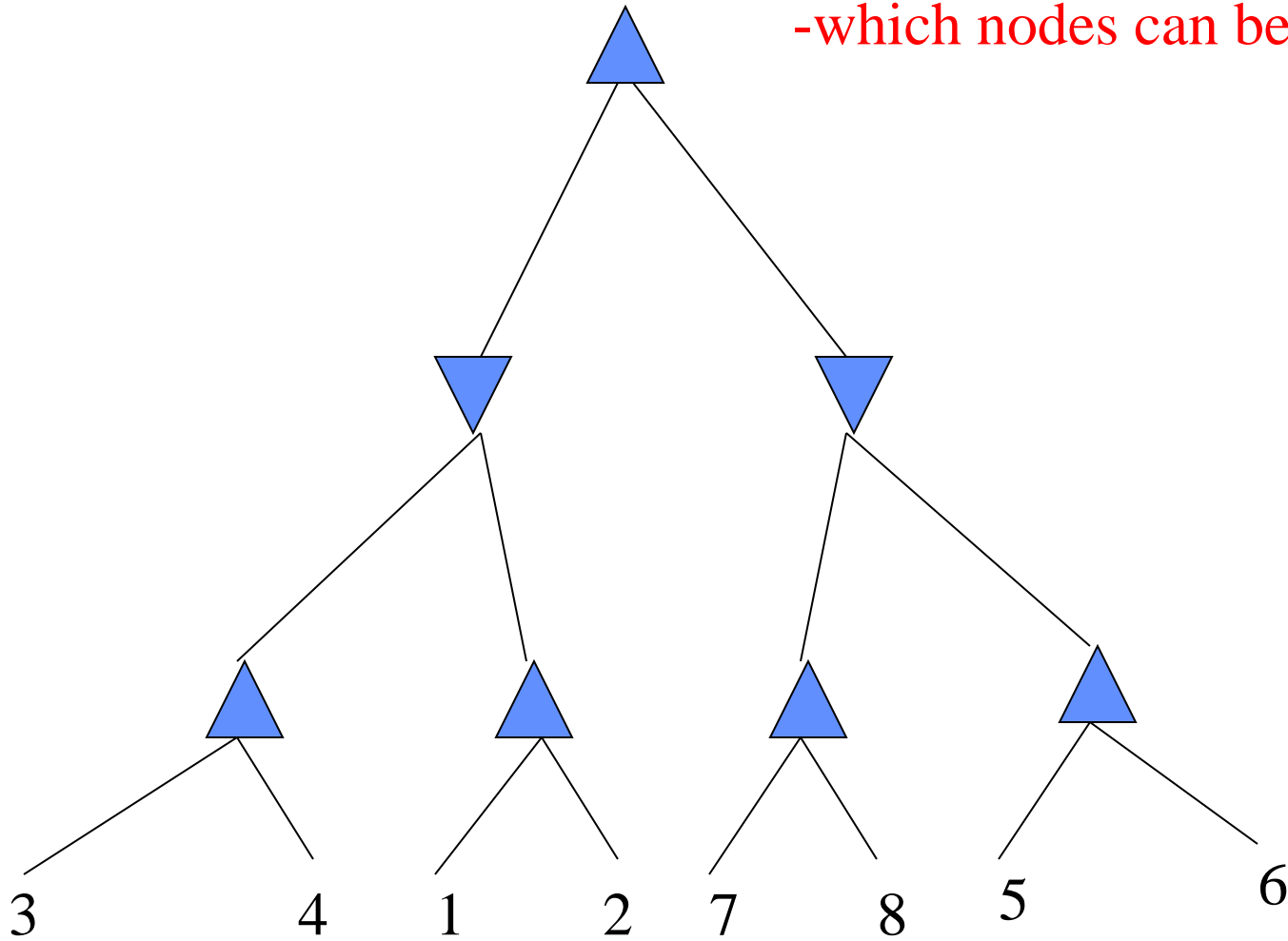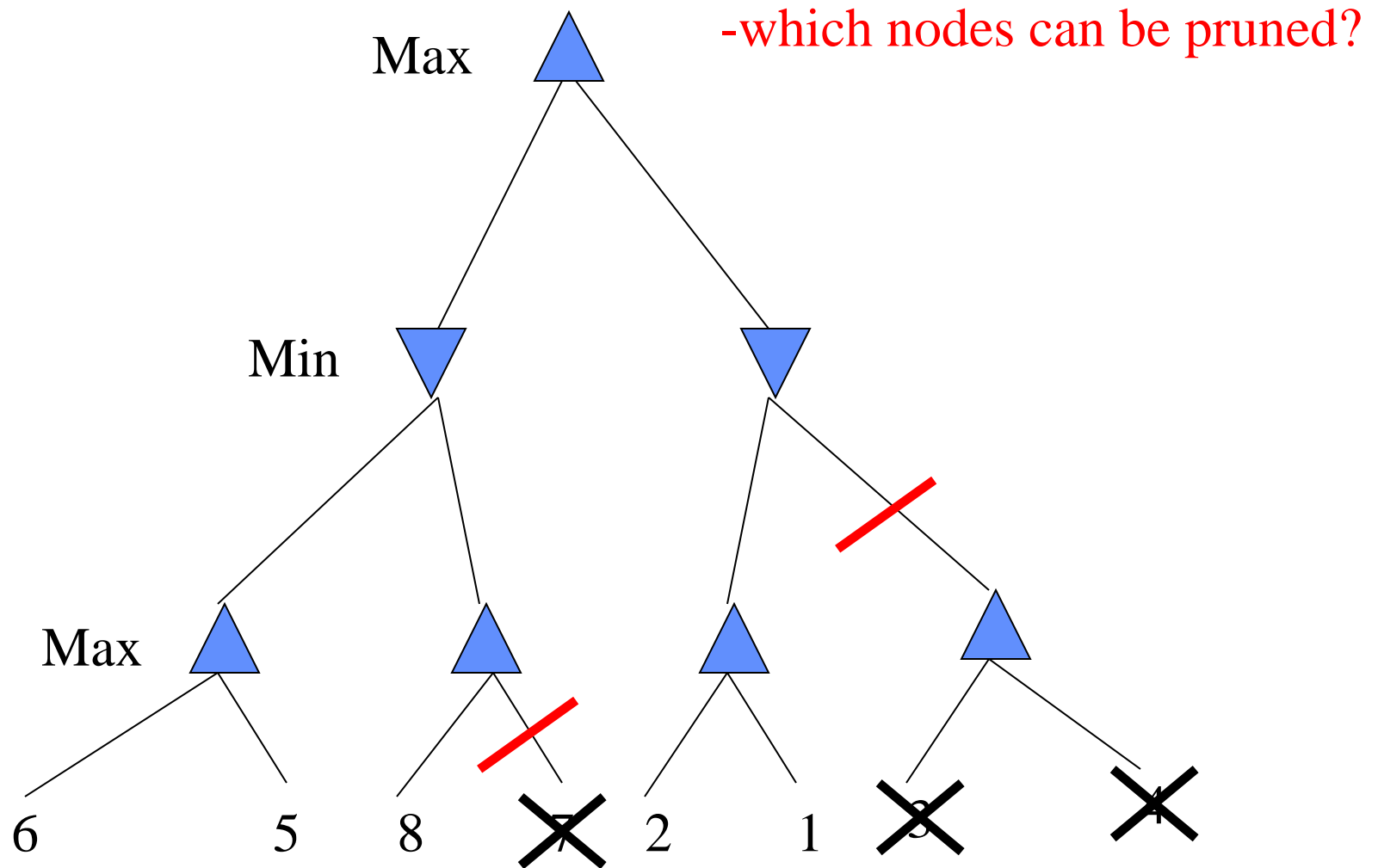
Answer: NONE! Because the most favorable nodes for both are explored last (i.e., in the diagram, are on the right-hand side).

Second Example
(the exact mirror image of the first example)

-which nodes can be pruned?

6   5   8   7   2   1   3   4

Answer to Second Example
(the exact mirror image of the first example)

Max

-which nodes can be pruned?

Min
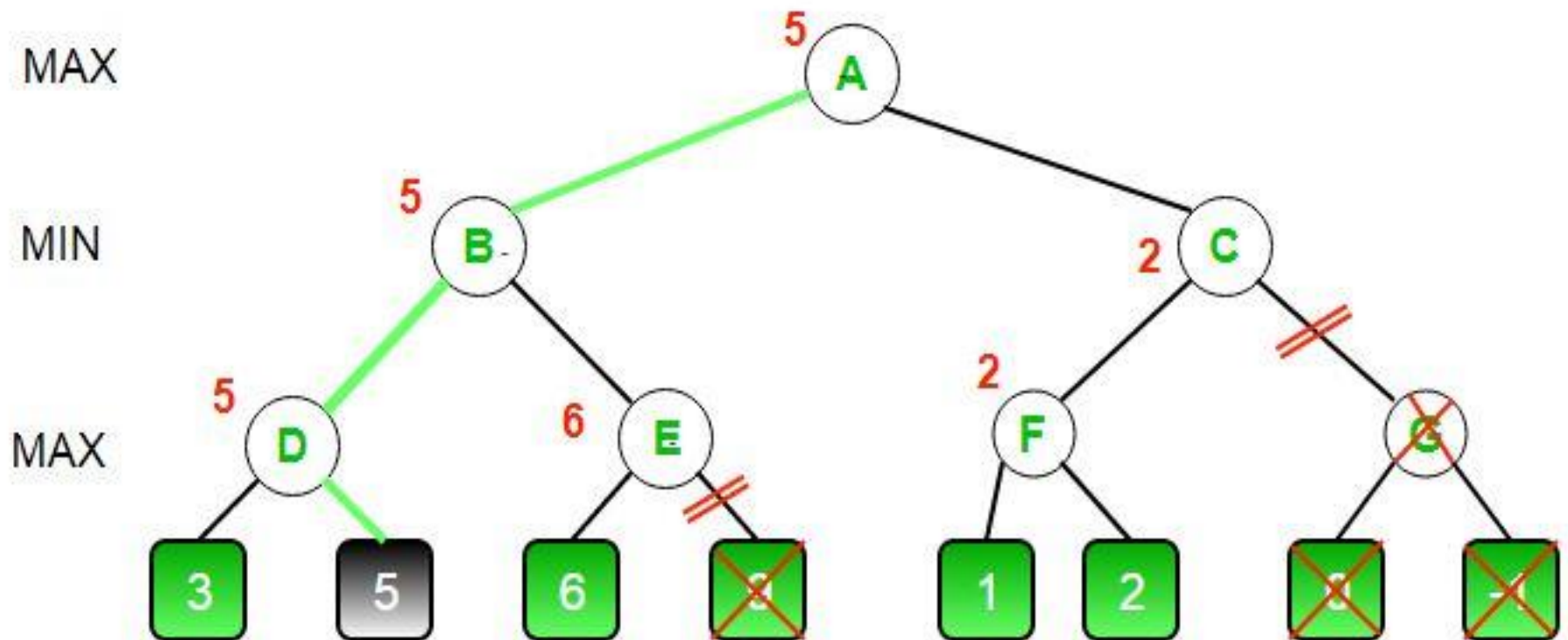
Max

6   5   8   7   2   1   8   4

Answer: LOTS! Because the most favorable nodes for both are explored first (i.e., in the diagram, are on the left-hand side).

Which nodes will be pruned?
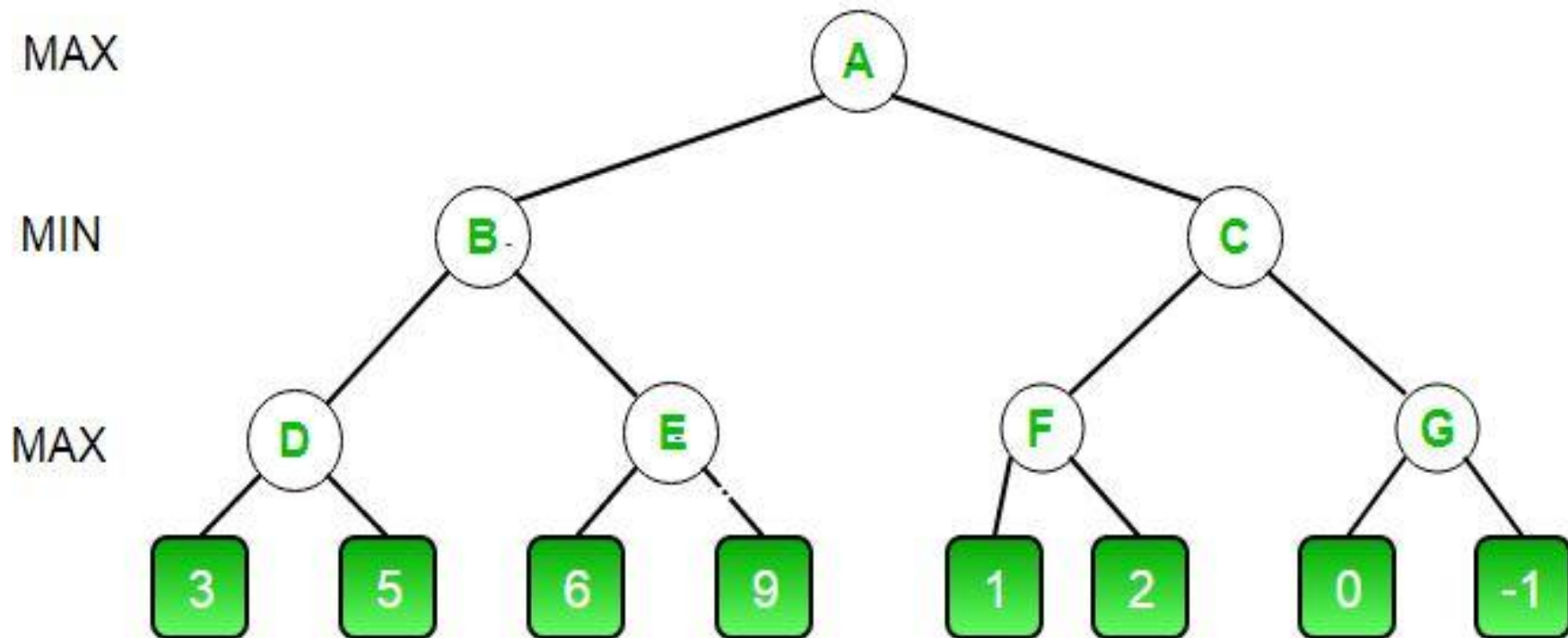
Answer

Iterative (Progressive) Deepening

- **In real games, there is usually a time limit T on making a move**

- **How do we take this into account?**
- **using alpha-beta we cannot use "partial" results with any confidence unless the full breadth of the tree has been searched**
  - So, we could be conservative and set a conservative depth-limit which guarantees that we will find a move in time < T
    - disadvantage is that we may finish early, could do more search

- **In practice, iterative deepening search (IDS) is used**
  - IDS runs depth-first search with an increasing depth-limit
  - when the clock runs out we use the solution found at the previous depth limit

Heuristics and Game Tree Search: limited horizon

- **The Horizon Effect**
  - sometimes there's a major "effect" (such as a piece being captured) which is just "below" the depth to which the tree has been expanded.
  - the computer cannot see that this major event could happen because it has a "limited horizon".
  - there are heuristics to try to follow certain branches more deeply to detect such important events
  - this helps to avoid catastrophic losses due to "short-sightedness"

- **Heuristics for Tree Exploration**
  - it may be better to explore some branches more deeply in the allotted time
 - various heuristics exist to identify "promising" branches

Eliminate Redundant Nodes

- **On average, each board position appears in the search tree approximately ~10^{150} / ~10^{40} ≈ 10^{100} times.**

  => Vastly redundant search effort.

- **Can't remember all nodes (too many).**

  => Can't eliminate all redundant nodes.

- **However, some short move sequences provably lead to a redundant position.**

  – These can be deleted dynamically with no memory cost

- **Example:**

  1. P-QR4 P-QR4; 2. P-KR4 P-KR4

  leads to the same position as

  1. P-QR4 P-KR4; 2. P-KR4 P-QR4

Nondeterministic games in general

In nondeterministic games, chance introduced by dice, card-shuffling

Simplified example with coin-flipping:
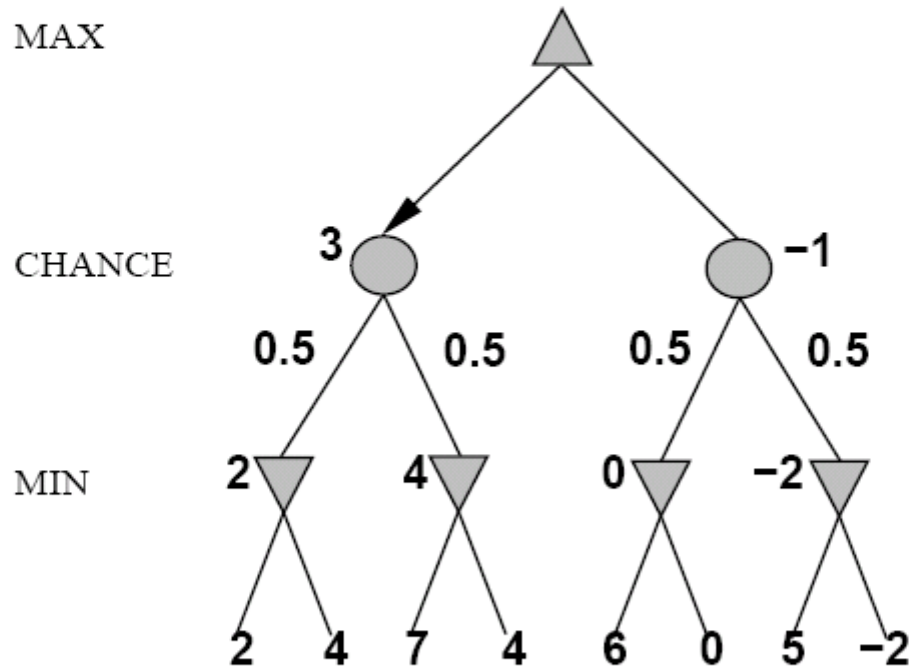
Algorithm for nondeterministic games

EXPECTIMINIMAX gives perfect play

Just like MINIMAX, except we must also handle chance nodes:

. . .

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*)

. . .

The State of Play

- **Checkers:**
  - Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994.

- **Chess:**
  - Deep Blue defeated human world champion Garry Kasparov in a six-game match in 1997.

- **Othello:**
  - human champions refuse to compete against computers: they are too good.

- **Go:**
  - human champions refuse to compete against computers: they are too bad
  - b > 300 (!)

- **See (e.g.) http://www.cs.ualberta.ca/~games/ for more information**

PHILADELPHIA (Reuter) -

IBM chess computer Deep Blue made chess history Saturday when it defeated world champion Garry Kasparov, the first time a computer program has beaten a grandmaster under strict tournament conditions.

IBM Deep Blue - Kasparov,G [B22]
Philadelphia (1), 1996

1.e4 c5 2.c3 d5 3.exd5 Qxd5 4.d4 Nf6 5.Nf3 Bg4 6.Be2 e6 7.h3

Bh5 8.0-0 Nc6 9.Be3 cxd4 10.cxd4 Bb4 11.a3 Ba5 12.Nc3 Qd6

13.Nb5 Qe7 14.Ne5 Bxe2 15.Qxe2 0-0 16.Rac1 Rac8 17.Bg5 Bb6

18.Bxf6 gxf6 19.Nc4 Rfd8 20.Nxb6 axb6 21.Rfd1 f5 22.Qe3 Qf6

23.d5 Rxd5 24.Rxd5 exd5 25.b3 Kh8 26.Qxb6 Rg8 27.Qc5 d4

28.Nd6 f4 29.Nxb7 Ne5 30.Qd5 f3 31.g3 Nd3 32.Rc7 Re8 33.Nd6

Re1+ 34.Kh2 Nxf2 35.Nxf7+ Kg7 36.Ng5+ Kh6 37.Rxh7+ 1-0

Deep Blue

- **1957: Herbert Simon**
  - "within 10 years a computer will beat the world chess champion"

- **1997: Deep Blue beats Kasparov**

- **Parallel machine with 30 processors for "software" and 480 VLSI processors for "hardware search"**

- **Searched 126 million nodes per second on average**
  - Generated up to 30 billion positions per move
  - Reached depth 14 routinely

- **Uses iterative-deepening alpha-beta search with transpositioning**
  - Can explore beyond depth-limit for interesting moves

Moore's Law in Action?

Figure 5.12 Ratings of human and machine chess champions.
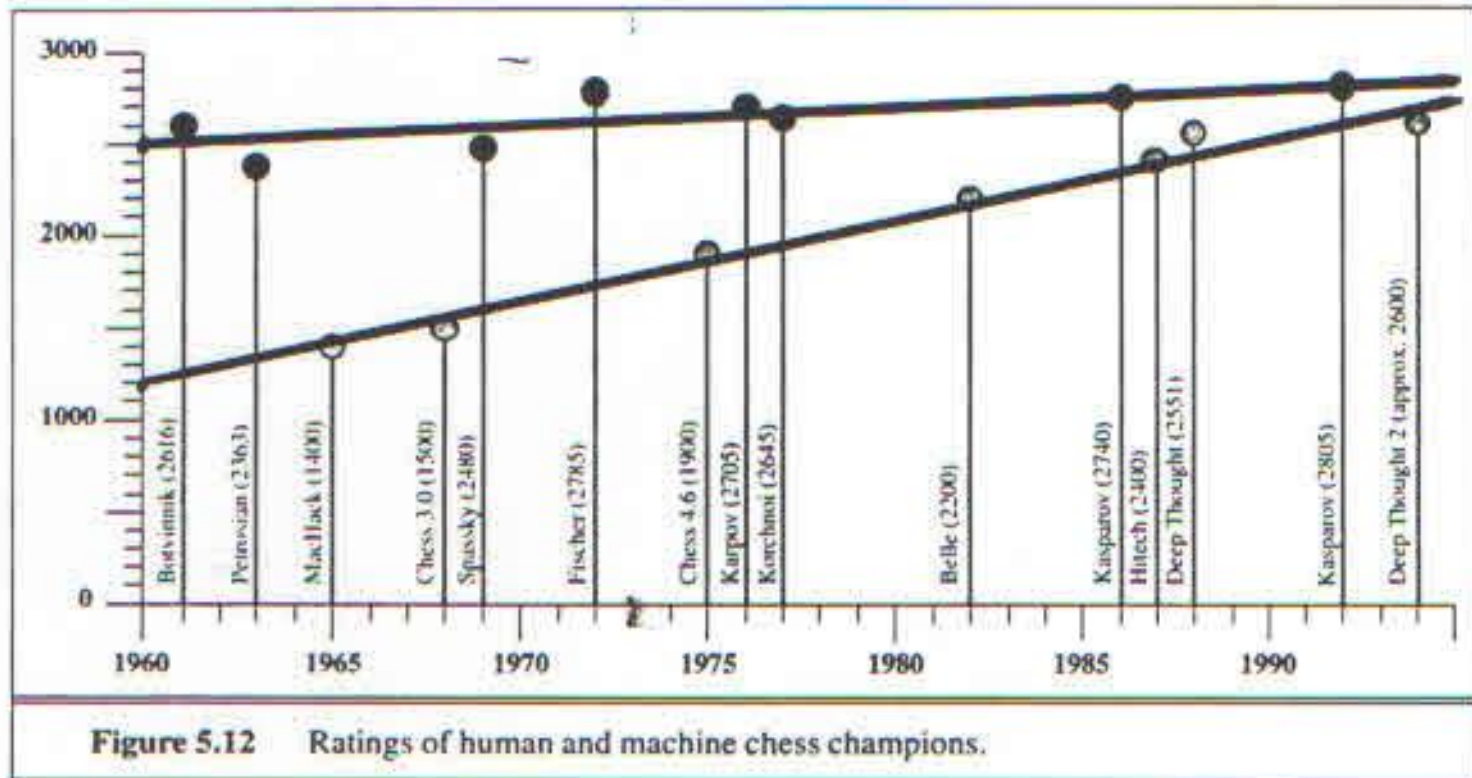
Summary

- Game playing is best modeled as a search problem

- Game trees represent alternate computer/opponent moves

- Evaluation functions estimate the quality of a given board configuration for the Max player.

- Minimax is a procedure which chooses moves by assuming that the opponent will always choose the move which is best for them

- Alpha-Beta is a procedure which can prune large parts of the search tree and allow search to go deeper

- For many well-known games, computer algorithms based on heuristic search match or out-perform human world experts.