Chapter 9: Game Tree Search

Recap on previous chapter

- Stackelberg game: Player 1: leader, Player 2: follower
- Stackelberg vs strategy game
- Strategy game and stackelberg game in zero-sum
- Stackelberg for general 2-persons game (LP)
- In (>)3-player normal form games, an optimal Stackelberg strategy is an NP-Hard problem
- Stackelberg Competition
- Security Game

Recap on previous chapter

Follower

Leader

	L	M	R
U	1 1	3 0	2 2
M	0 0	2 1	1 1
D	3 4	2 3	3 0

Present a solution for the optimal Stackelberg strategy

Finite perfect-information zero-sum games

- Finite: finitely players, actions, states
- Perfect information: every player knows the current state, all of the strategies, and what they do
- No simultaneous actions – players move one-at-a-time
- Constant-sum: regardless of how the game ends, Σ players' payoff $= k$:
  - Zero-sum game when $k = 0$
  - Thus constant-sum games usually are called zero-sum games

Settings

- Two players whose actions alternate
- Utility values for each player are the opposite of the other
  - This creates the adversarial situation
- Fully observable environments
- Deterministic game: go, chess
- Generalizes to stochastic games: backgammon, monopoly, yahtzee, parcheesi, roulette, craps

A brief history

1846: machine to play tic-tac-toe

1928: minimax theorem (von Neumann)

1944: backward-induction algorithm to produce perfect play (von Neumann & Morgenstern)

1950: minimax algorithm -finite horizon, approximate evaluation (Shannon)

1951: program (on paper) for playing chess (Turing)

1956: pruning to allow deeper search

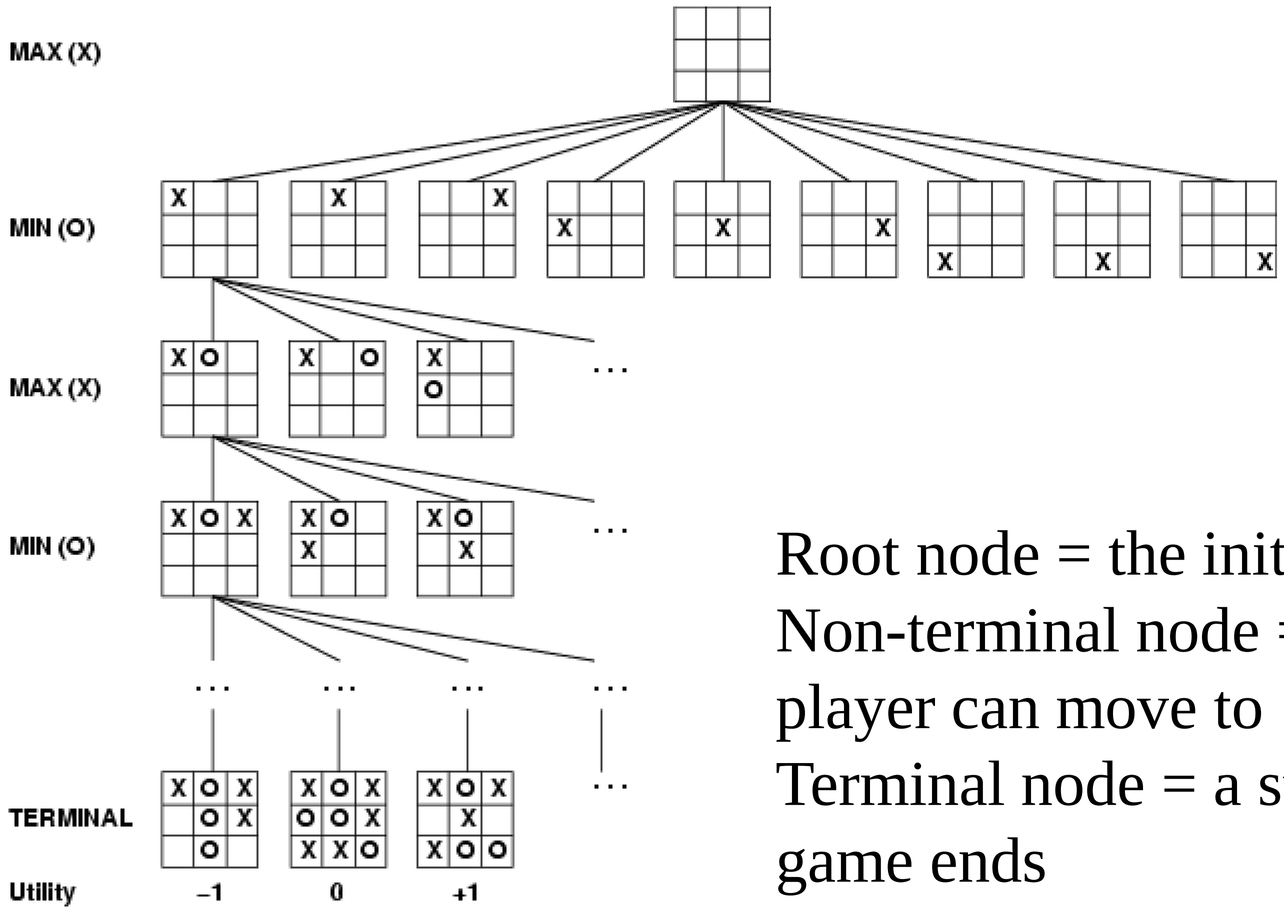
1957: first complete chess program

1967: first program to compete in human chess tournaments

1997 (IBM): Deep Blue

2017: (Deepmind): AlphaGo

Game tree (2-player, deterministic, turns)



Root node = the initial state
Non-terminal node = the states a player can move to
Terminal node = a state where the game ends

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 techniques can find *optimal* solution
  - Evaluation function: estimate of cost from start to goal through given node
- 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, go…

Two players: MAX and MIN

MAX moves first and 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 of chess
  - **Player:** Defines which player has the move in a state
  - **Actions:** the set of legal moves in a state
  - **Result:** Transition model defines the result state of a move
  - **Successor function:** list of (move,state) pairs specifying legal moves
  - **Terminal-Test:** Is the game finished? True if finished, false other.
  - **Utility function:** Gives numerical value of terminal state for player
    - 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

Find the optimal strategy for Max:

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

2. Apply utility (payoff) function to each leaf

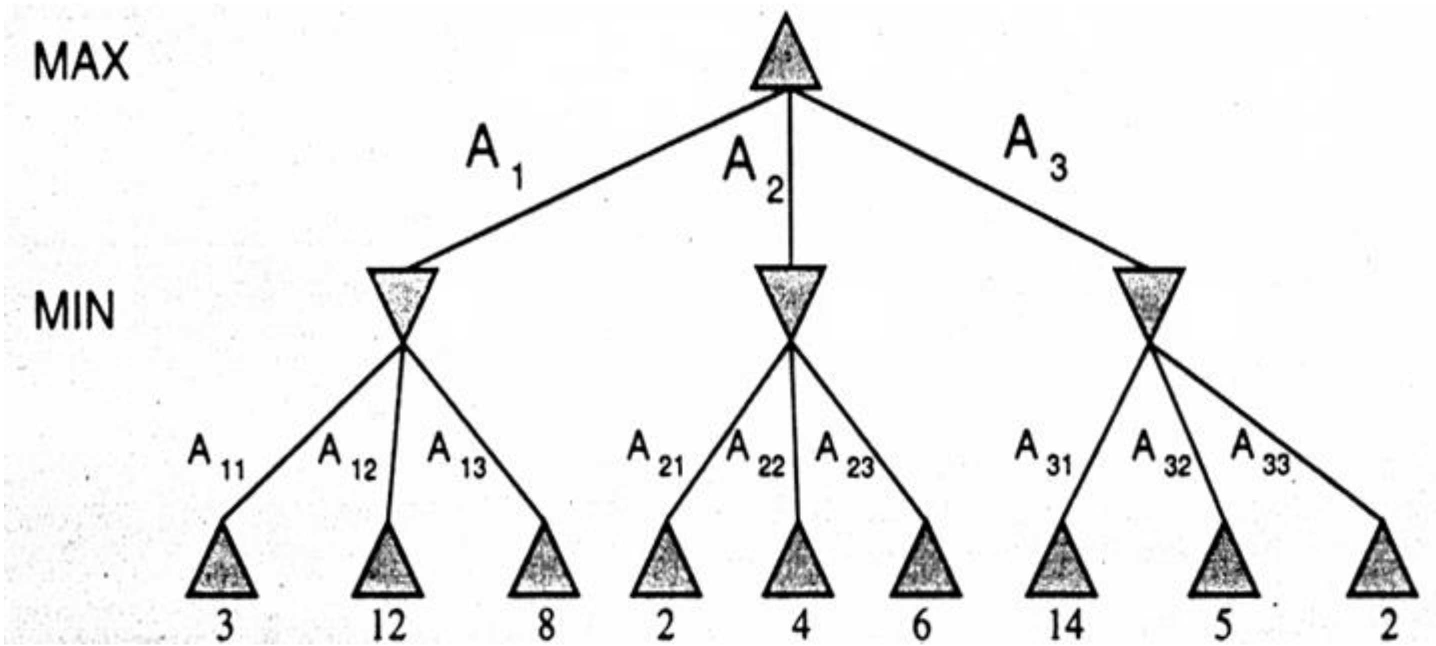
3. Back-up values from leaves through branch nodes:

    - a Max node computes the maximum of its child values

    - a Min node computes the minimum of its child values

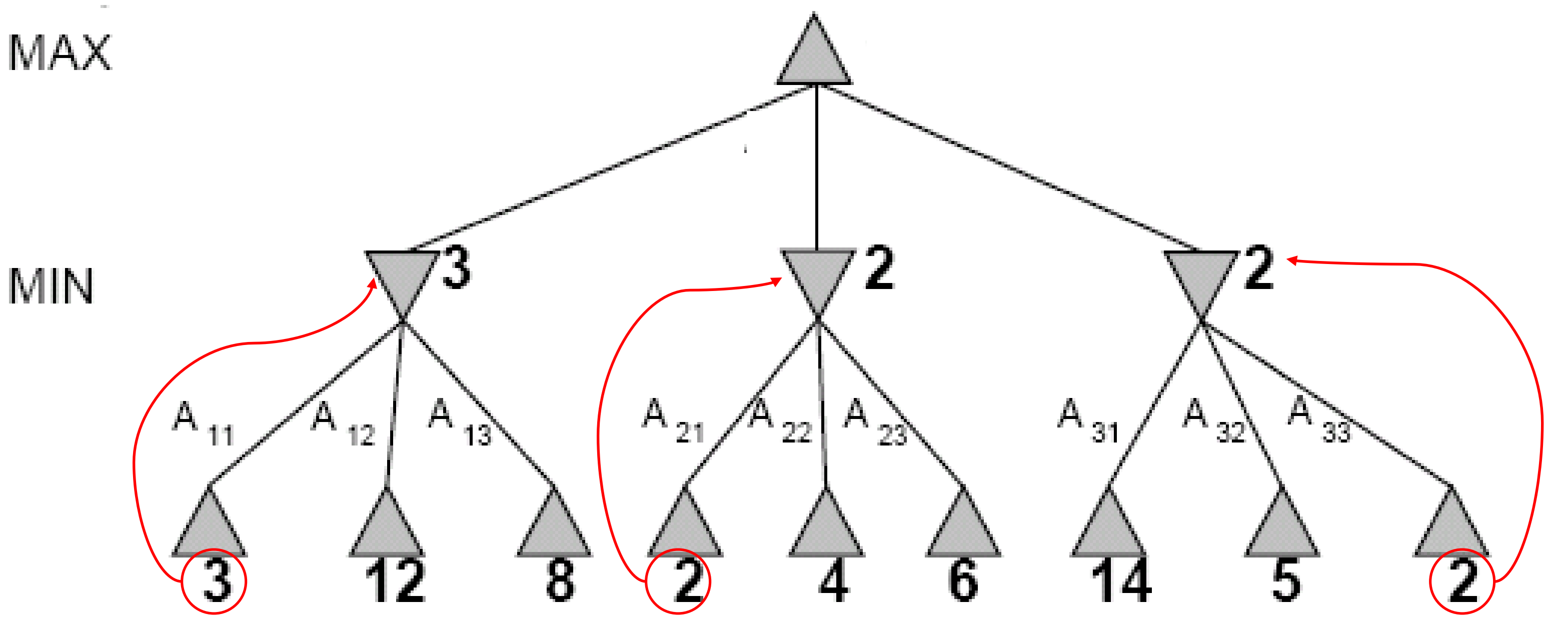
4. At root: choose the move leading to the child of highest value

Game Trees



A two-players game tree: the  and  nodes are moves by MAX and MIN, respectively. The terminial nodes show the payoff for MAX, whereas the payoffs of other nodes can be computed by the minimax algorithms

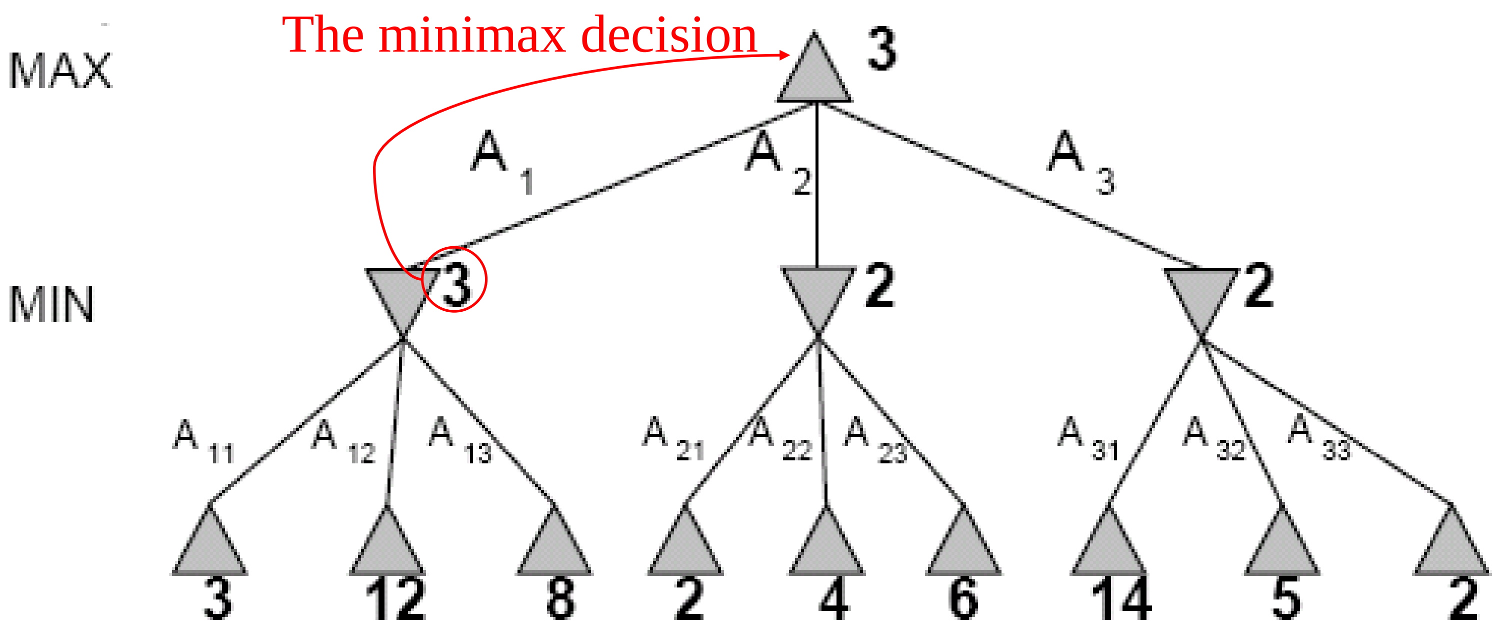
Two-Players Game Tree



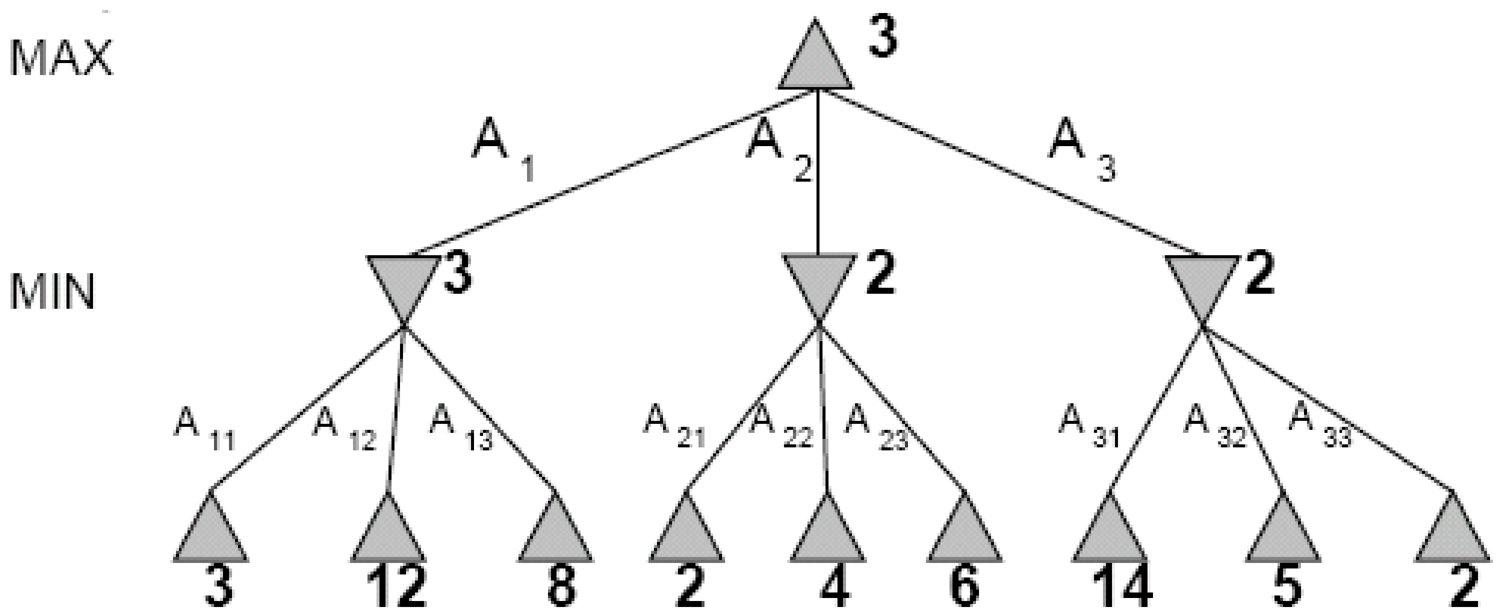
Backward induction

Two-Players Game Tree

Maximizes the payoffs for MAX



Two-Players Game Tree



Pseudocode for Minimax Algorithm

function MINIMAX-DECISION(*state*) **returns** *an action*
  inputs: current state in game
  $\boxed{\textbf{return} \ \arg\max_{a \in \text{ACTIONS}(state)} \text{MIN-VALUE}(\text{Result}(state, a))}$

---

function MAX-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for *a* in ACTIONS(*state*) **do**
    $\boxed{v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{Result}(state, a)))}$
  return *v*

---

function MIN-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for *a* in ACTIONS(*state*) **do**
    $\boxed{v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{Result}(state, a)))}$
  return *v*

Properties of Minimax Algorithm

- 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?
  - $O(b^m)$ [b =max. number of child., m =max. depth of any node]
- Space complexity?
  - $O(bm)$ (depth-first search, generate all actions at once)

Game Tree Size

- Tic-Tac-Toe
  - $b \approx 5$ legal actions per state on average, total of 9 depth in game
  - $5^9 = 1{,}953{,}125$
  - → **exact solution quite reasonable**

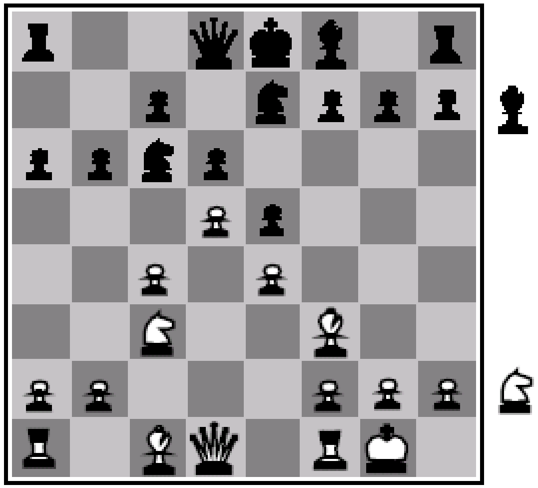
- Chess
  - $b \approx 35$ (approximate average branching factor)
  - $d \approx 100$ (depth of game tree for "typical" game)
  - $b^d \approx 35^{100} \approx 10^{154}$ nodes!!
  - → **exact solution completely infeasible**

It is usually impossible to search the whole tree

(Static) Heuristic Evaluation Functions

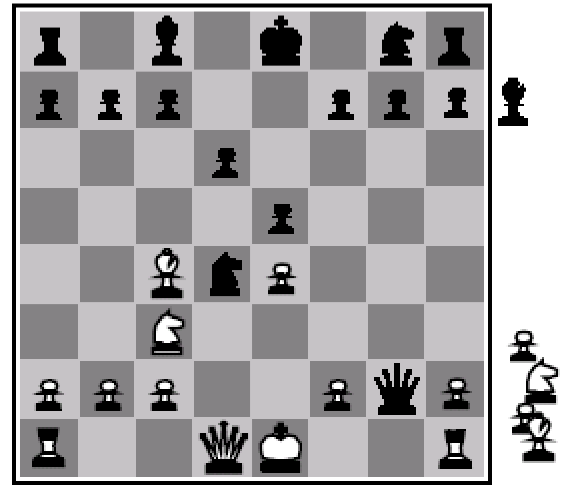
- An Evaluation Function
  - Estimates how good the current state is for a player
  - Evaluate how good it is for the player, how good it is for the opponent, then subtract the opponent's score from the player's
  - Called "static" because it is called on a static board position
  - Othello: Number of white pieces - Number of black pieces
  - Chess: Value of all white pieces - Value of all black pieces
- Typical values from $-\infty$ (loss) to $+\infty$ (win) or [-1, +1]
- If the board evaluation is X for a player, it's -X for the opponent e.g., zero-sum game

Evaluation Functions (example)



Black to move

White slightly better



White to move

Black winning

For chess, typical linear weighted sum of features:

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

e.g., $w_1 = 9$ with $f_1(s)$=number of white queens - number of black queens, etc.

Applying MiniMax to tic-tac-toe

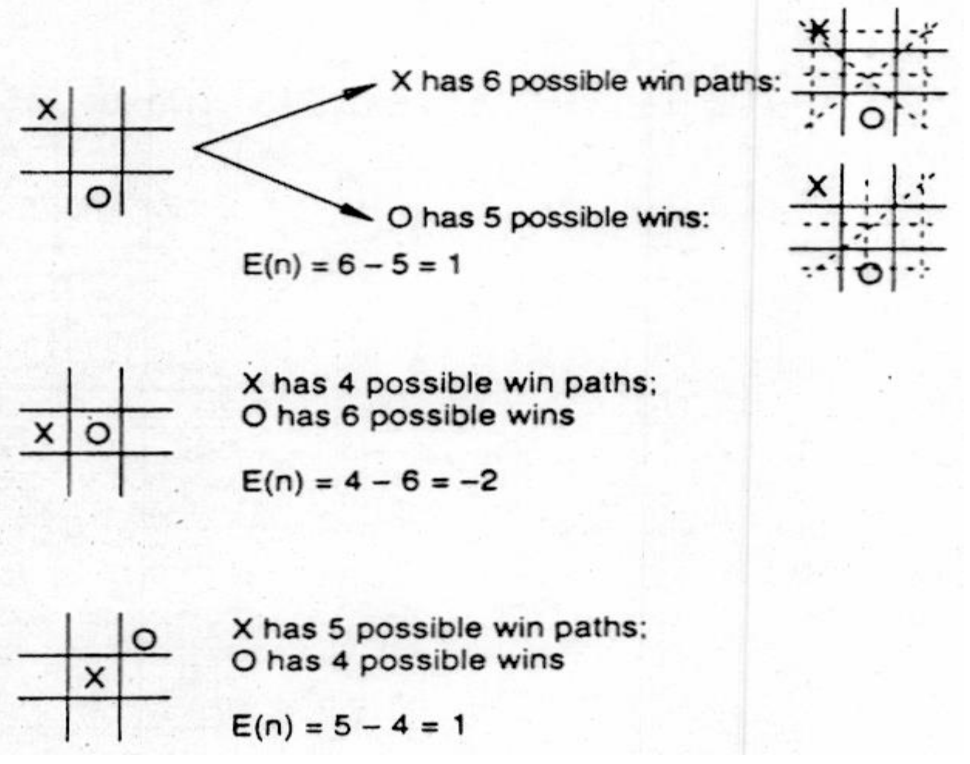
- The static heuristic evaluation function: current state n

  $M(n)$: the number of my possible winning lines

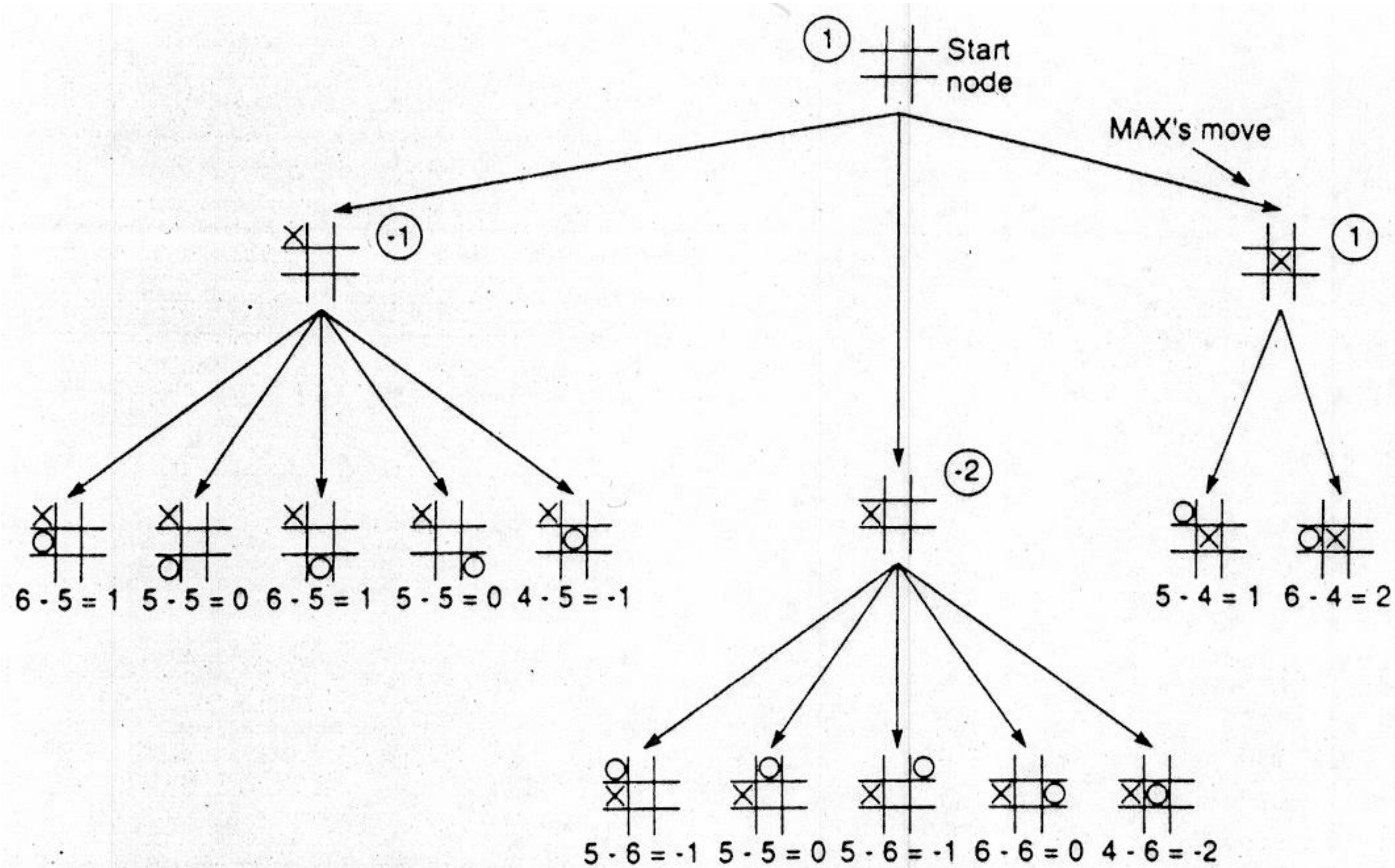
  $O(n)$: the number of the oppenent's possible winning lines

  Heuristic evaluation function

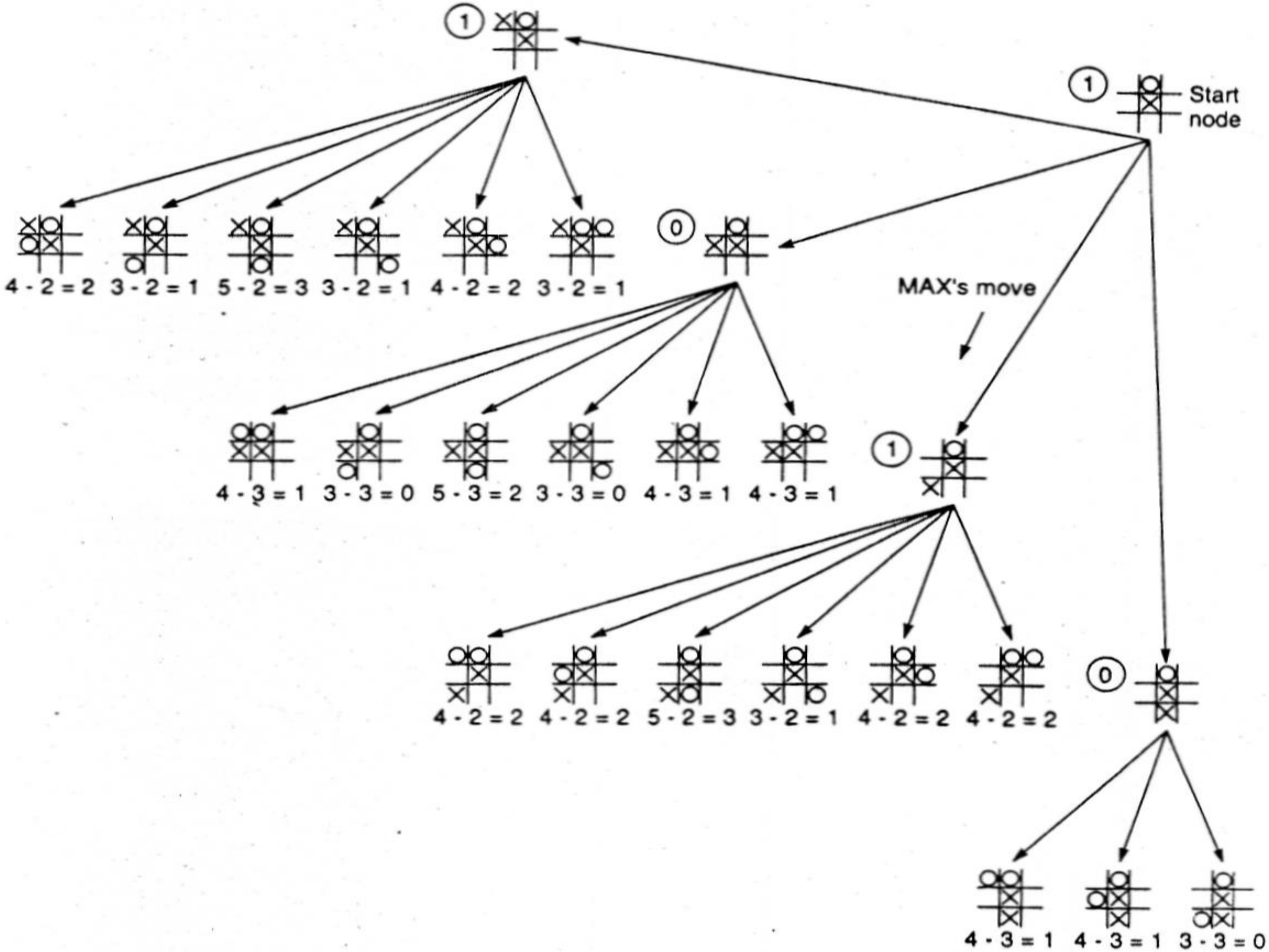
  $$E(n) = M(n) - O(n)$$



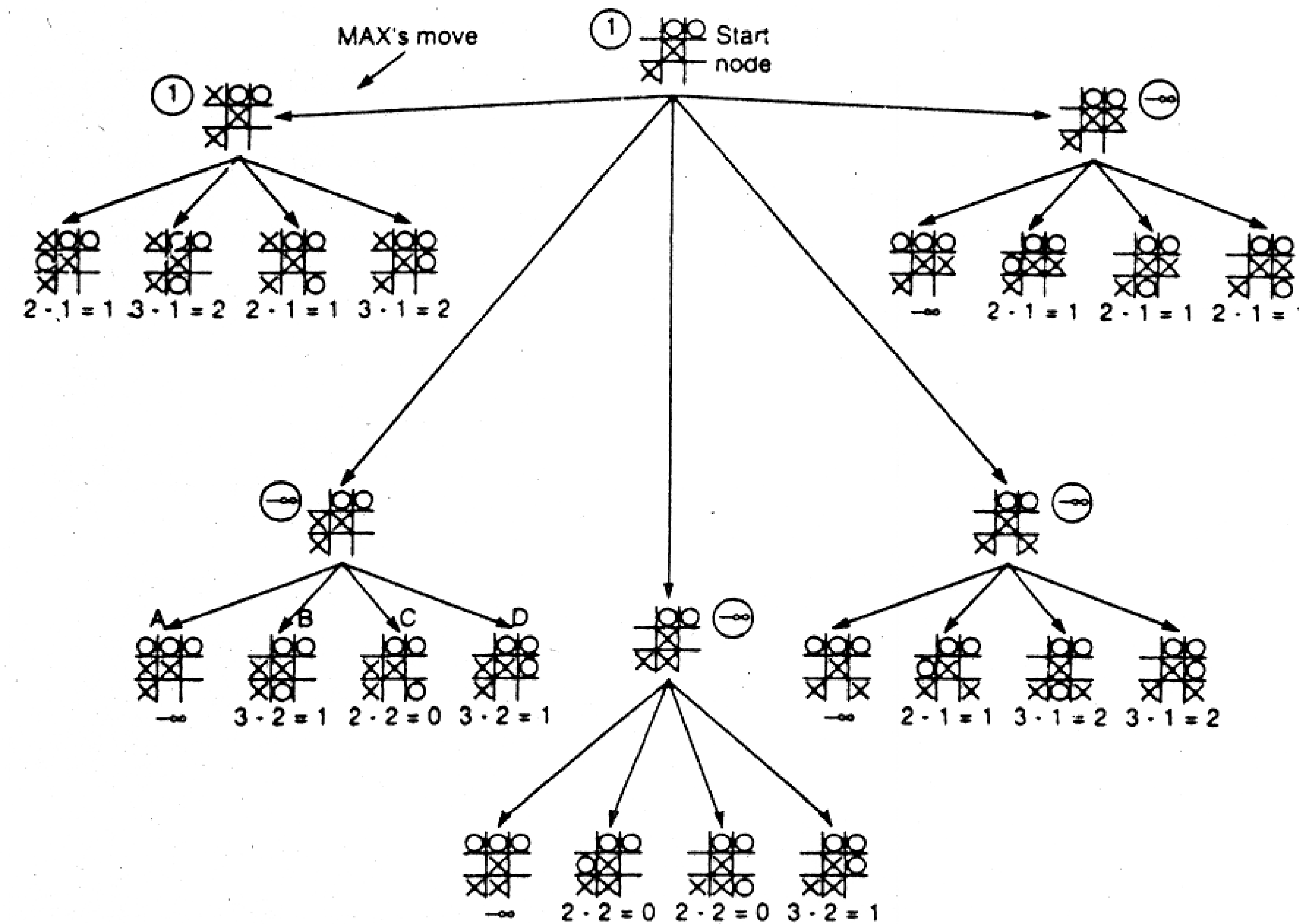
Backup Values



Backup Values



Near the end game



Chapter 9: Game Tree Search

Recap on last course

- Two players made adversary actions alternately
- Game problem -> search problem
- Minimax algorithm
- Evaluation function

Applying MiniMax to tic-tac-toe

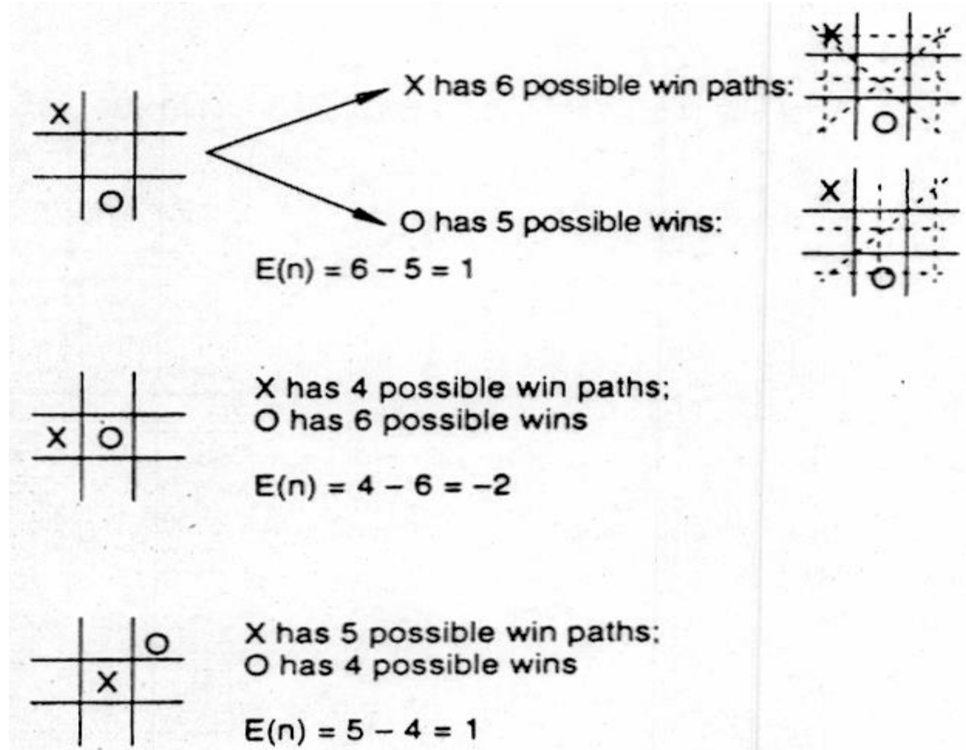
- The static heuristic evaluation function: current state n

  $M(n)$: the number of my possible winning lines

  $O(n)$: the number of the oppenent's possible winning lines

  Heuristic evaluation function

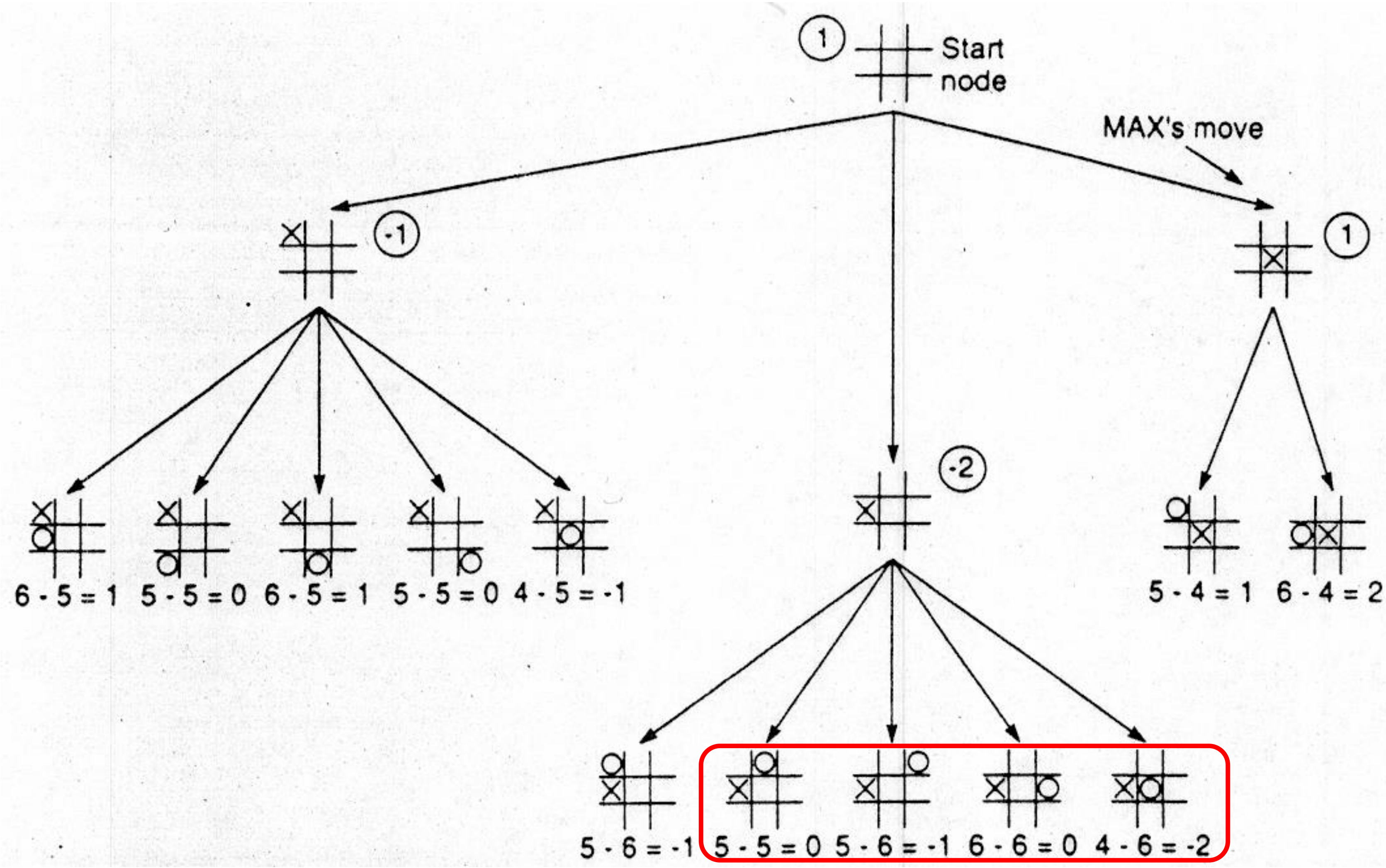
  $$E(n) = M(n) - O(n)$$



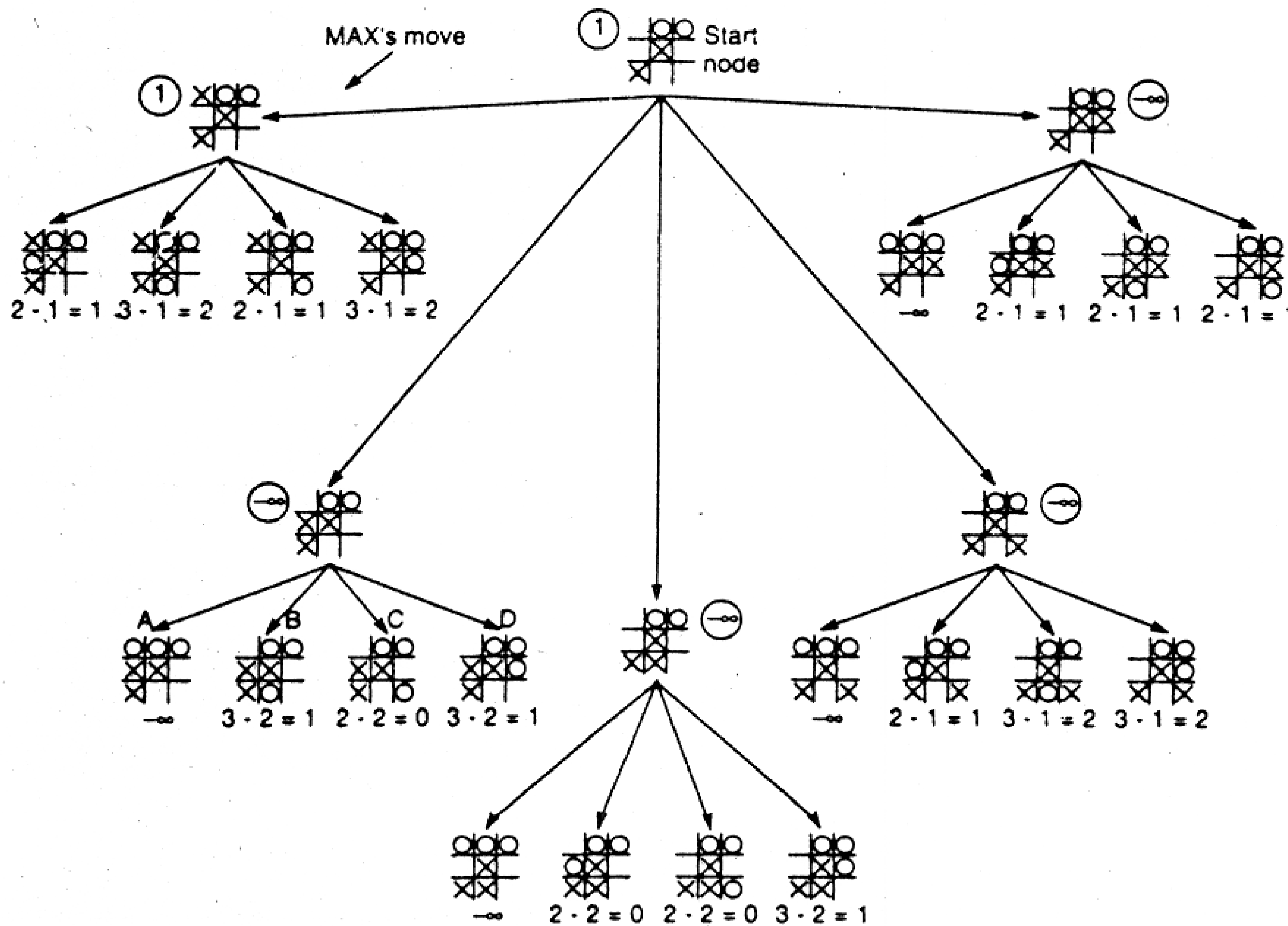
Alpha-Beta Pruning: Exploiting the fact of an Adversary

- If a position is provably bad:
  - It is NO USE expending search time to find out exactly how bad
- If the adversary can force a bad position:
  - It is NO USE expending 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.

Tic-Tac-Toe Example with Alpha-Beta Pruning

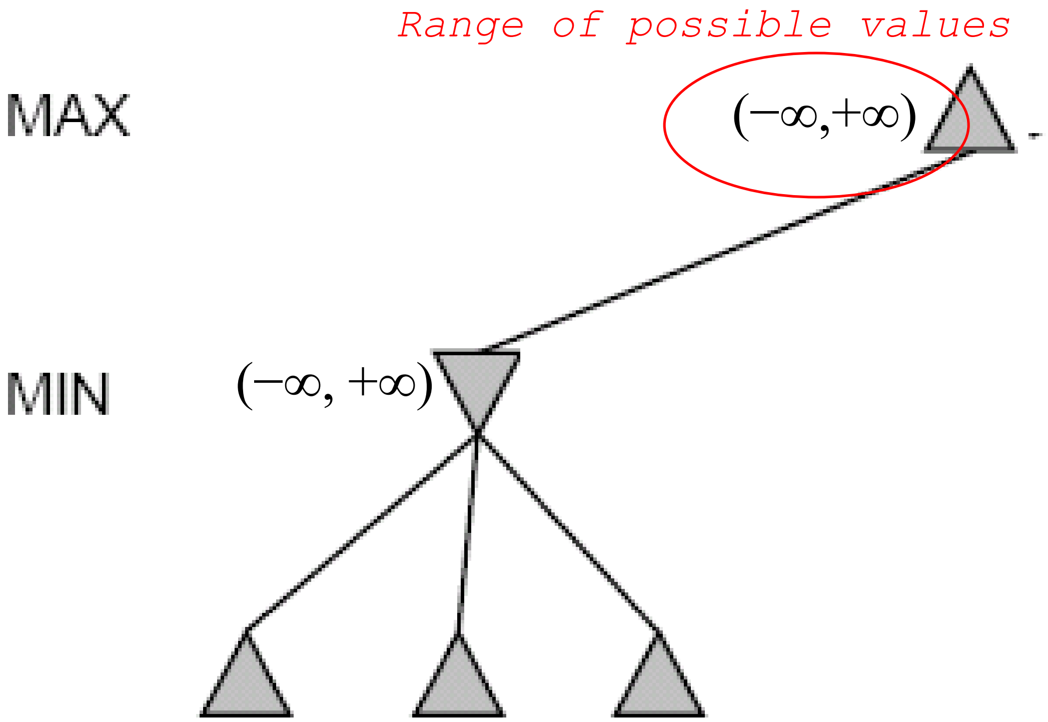


Tic-Tac-Toe Example with Alpha-Beta Pruning

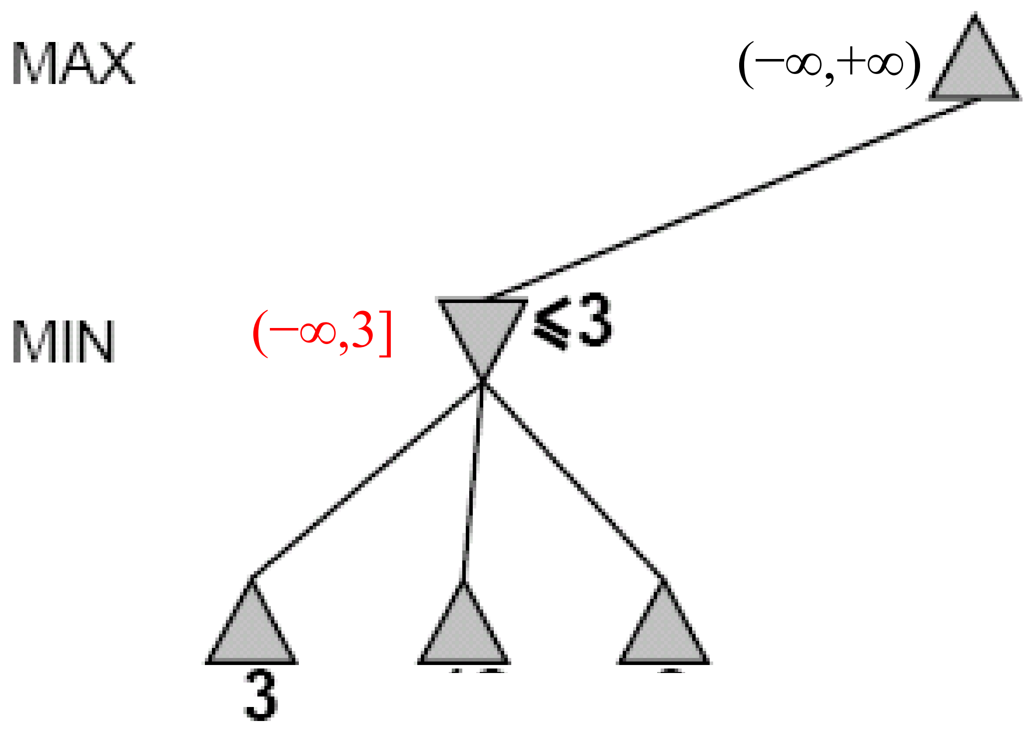


Another Alpha-Beta Example

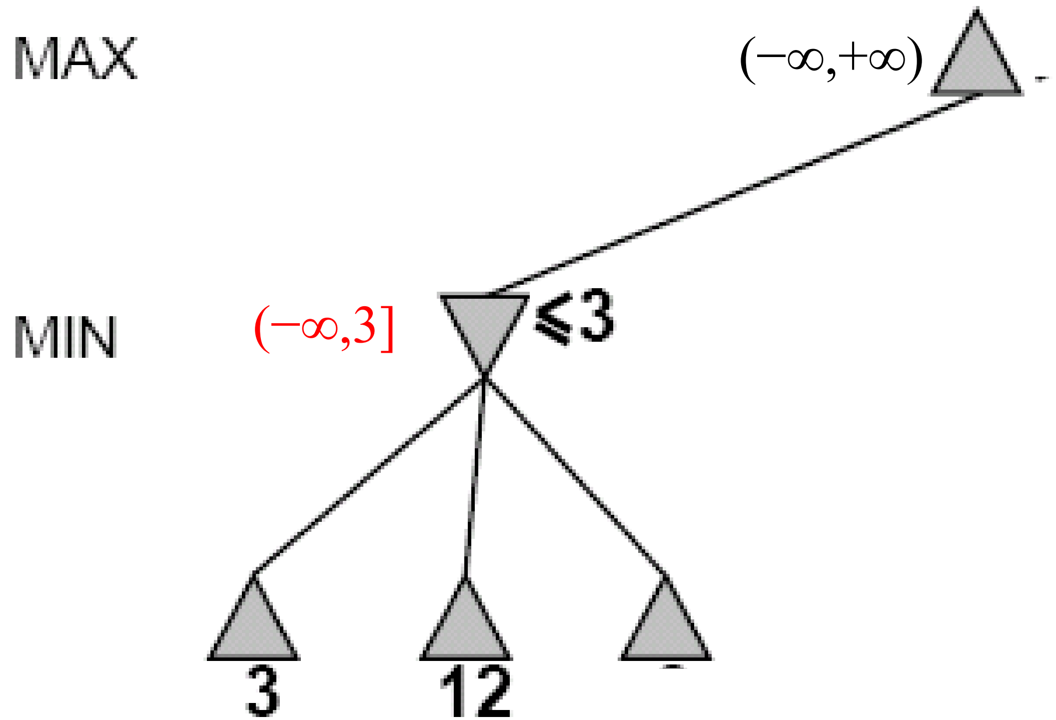
Do DF-search until first leaf



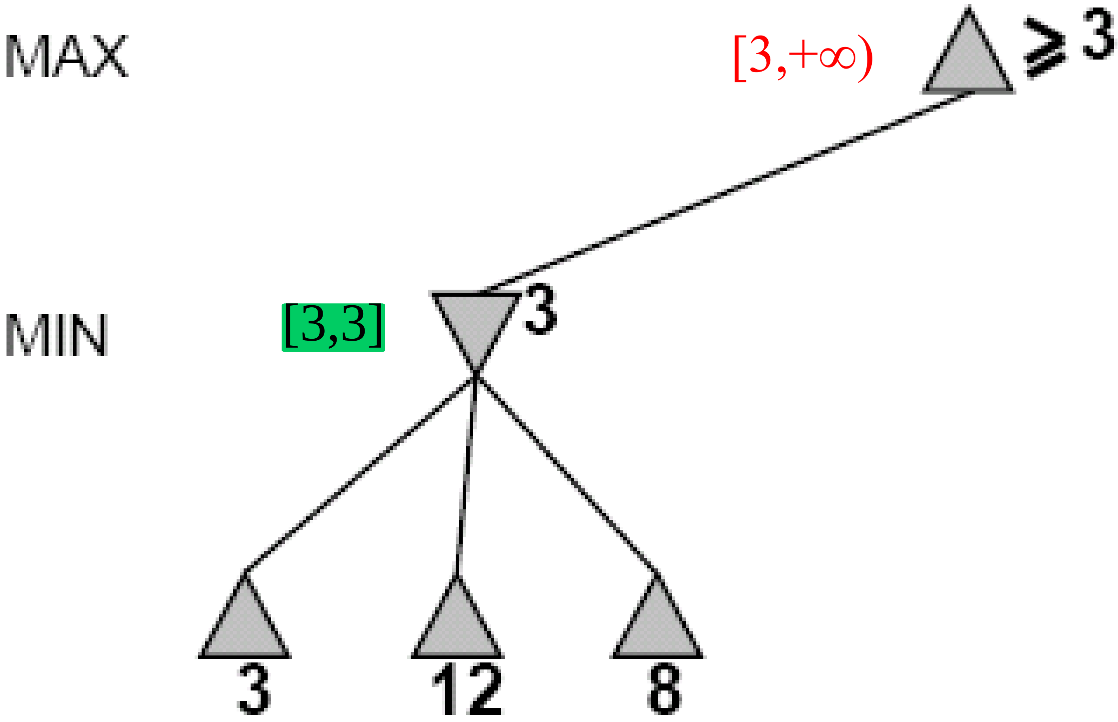
Alpha-Beta Example (continued)



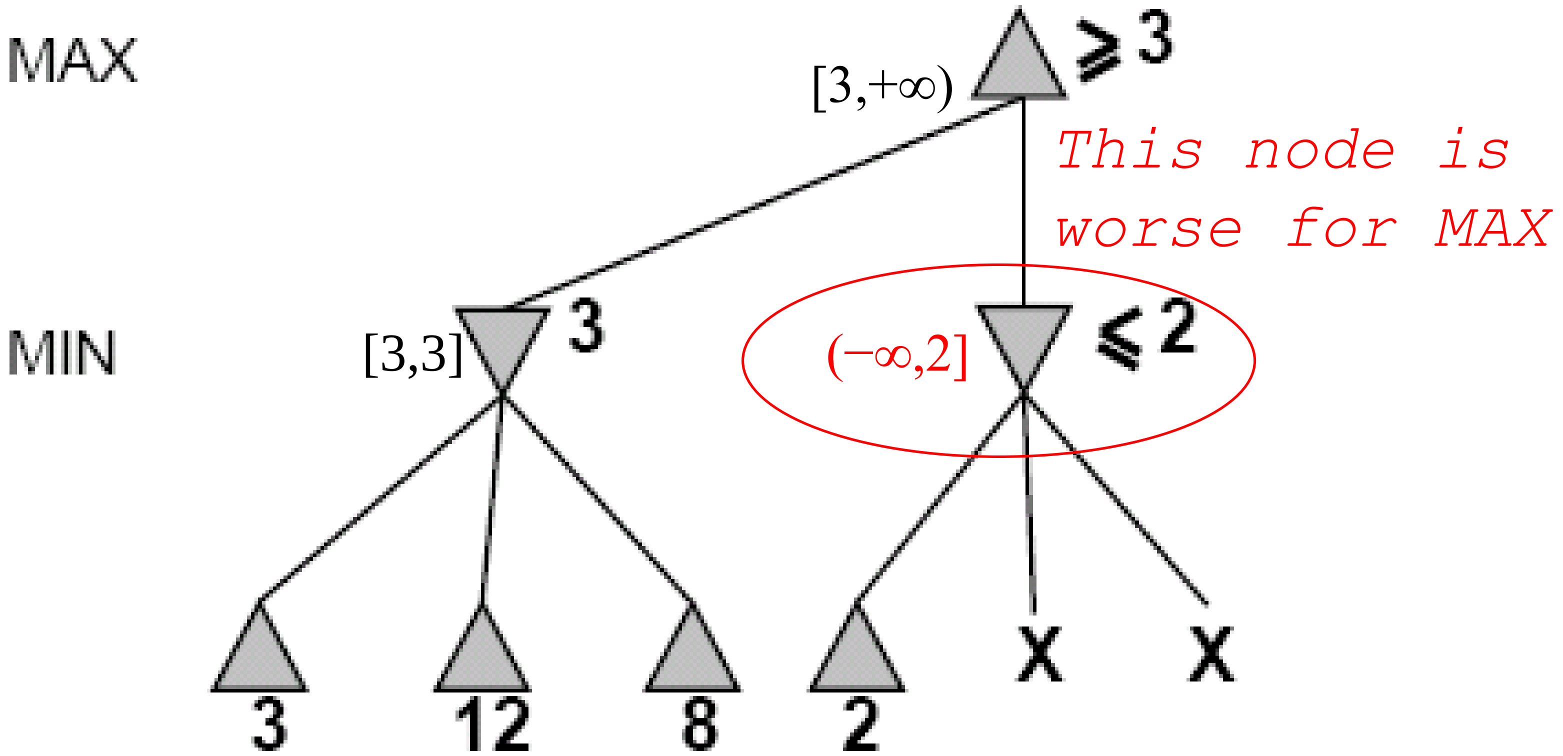
Alpha-Beta Example (continued)



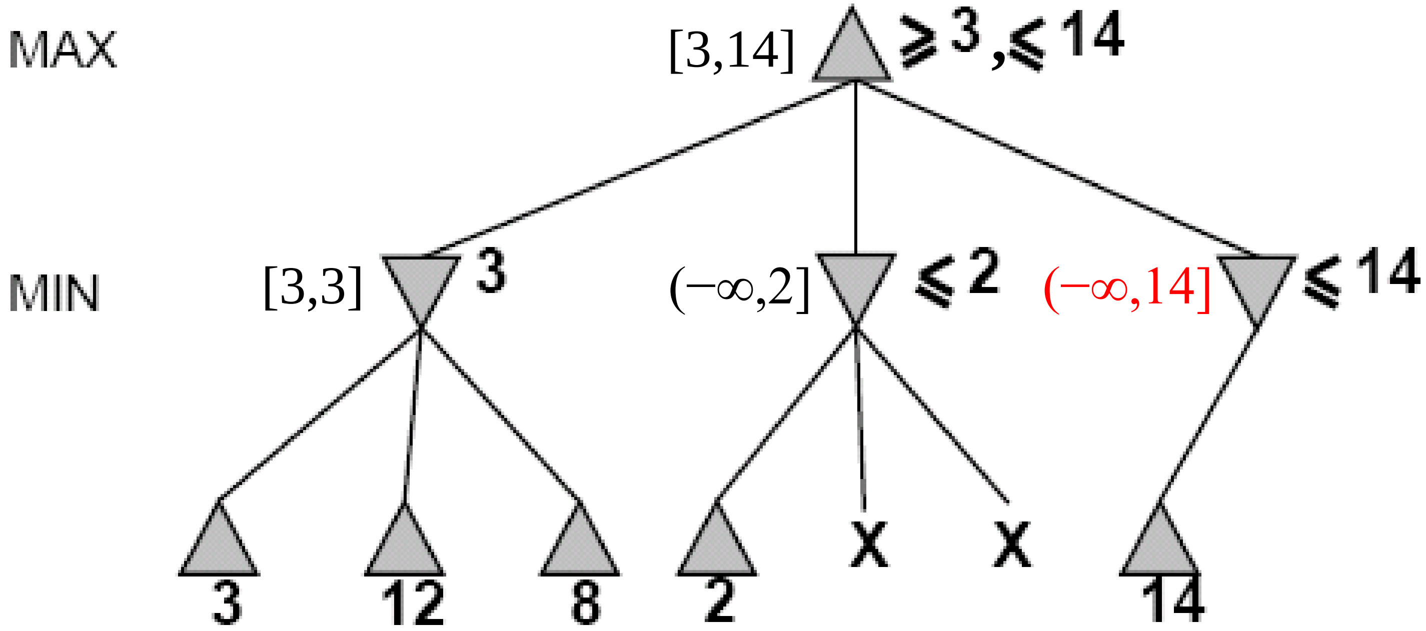
Alpha-Beta Example (continued)



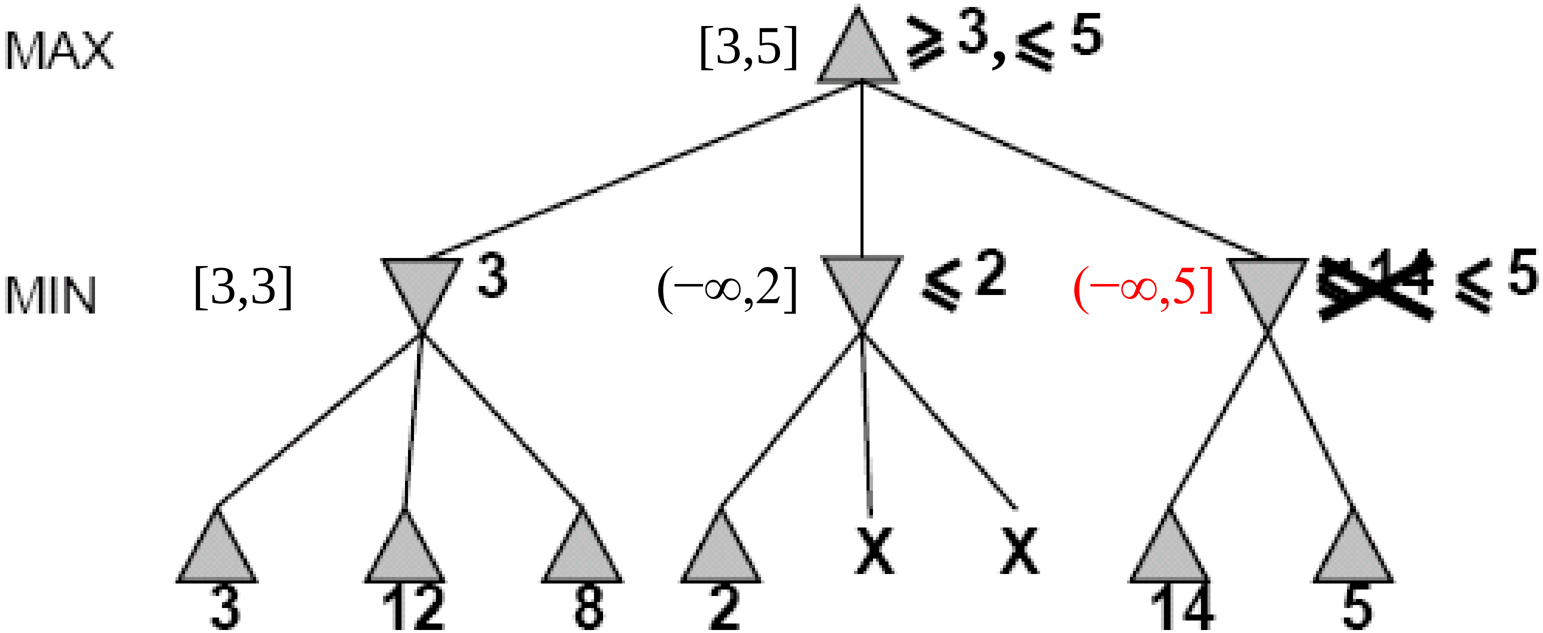
Alpha-Beta Example (continued)



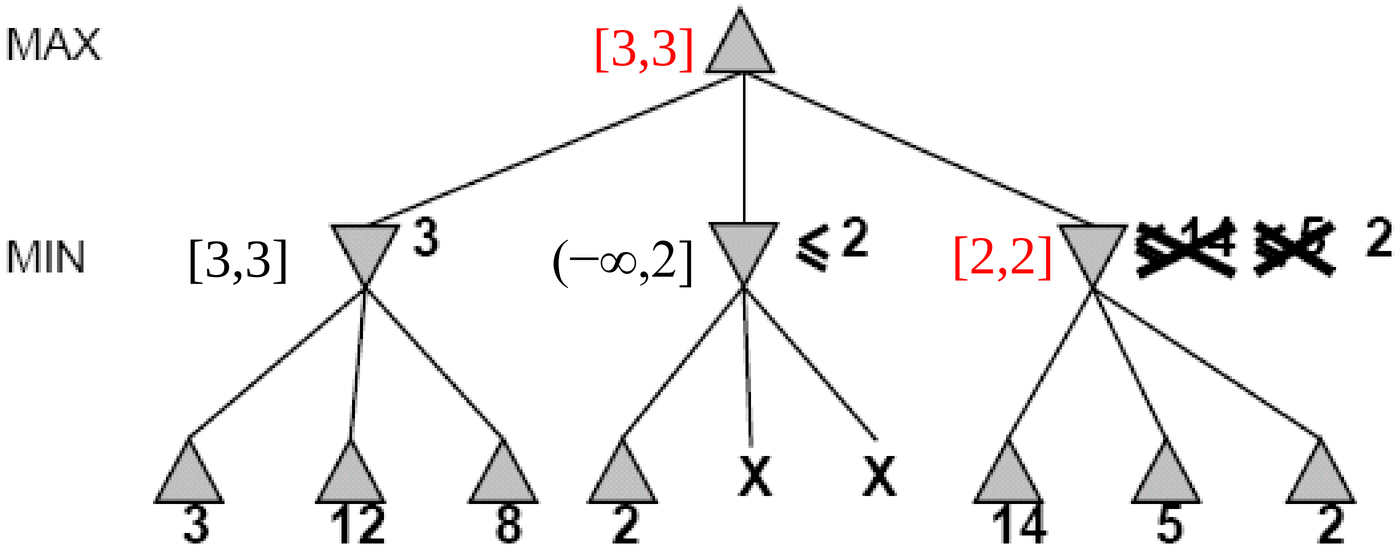
Alpha-Beta Example (continued)



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
  - $\alpha =$ highest-value choice found at any choice point of path for MAX (initially, $\alpha = -\infty$)
  - β = lowest-value choice found at any choice point of path for MIN (initially, $\beta = +\infty$)
- 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 α value becomes greater than or equal to the β value of its ancestors
    - **Max nodes update** α based on children's returned values.
  - Prune below a Min node whose β value becomes less than or equal to the α value of its ancestors
    - **Min nodes update** β 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 ACTIONS(*state*) with value v

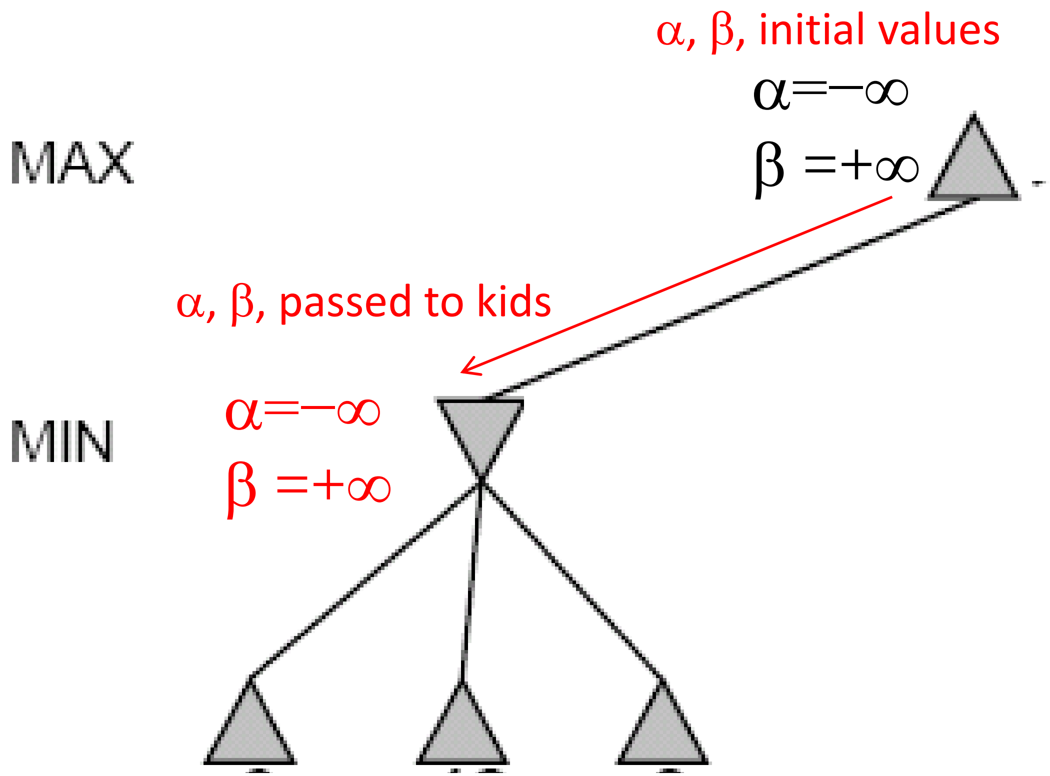
---

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

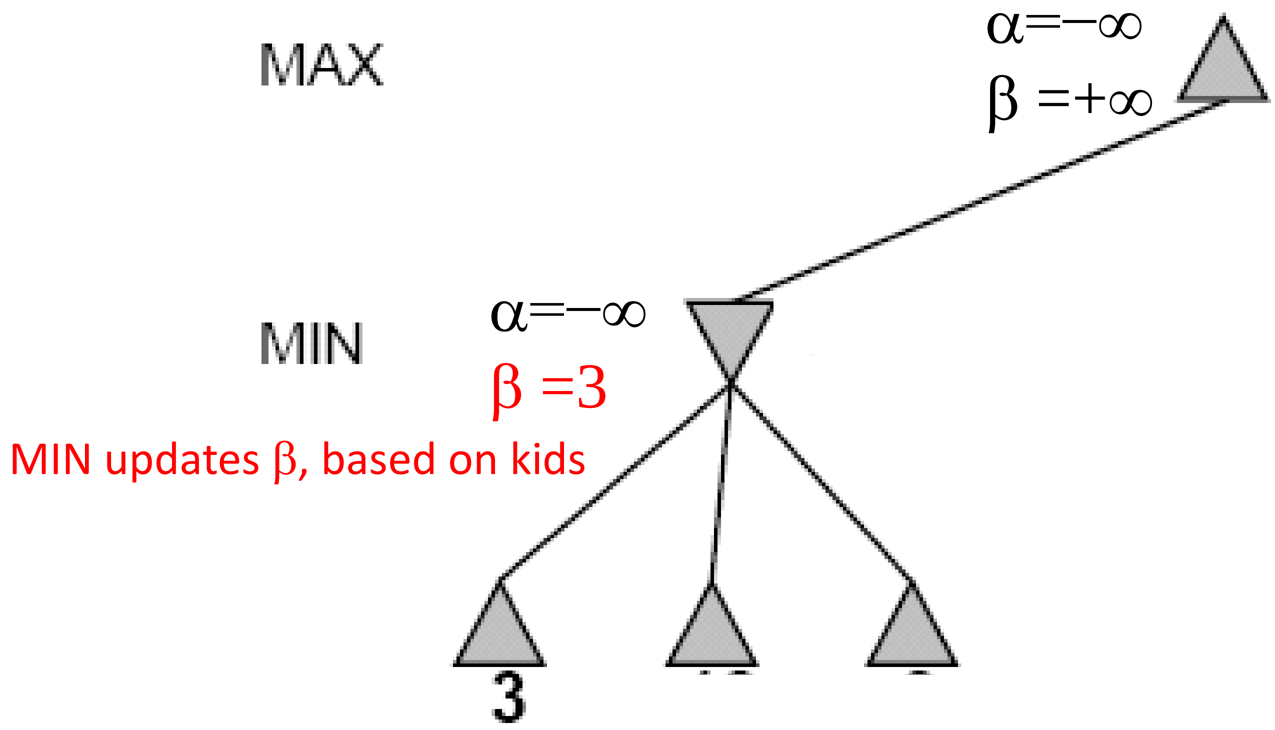
(MIN-VALUE is defined analogously)

Alpha-Beta Example Revisited

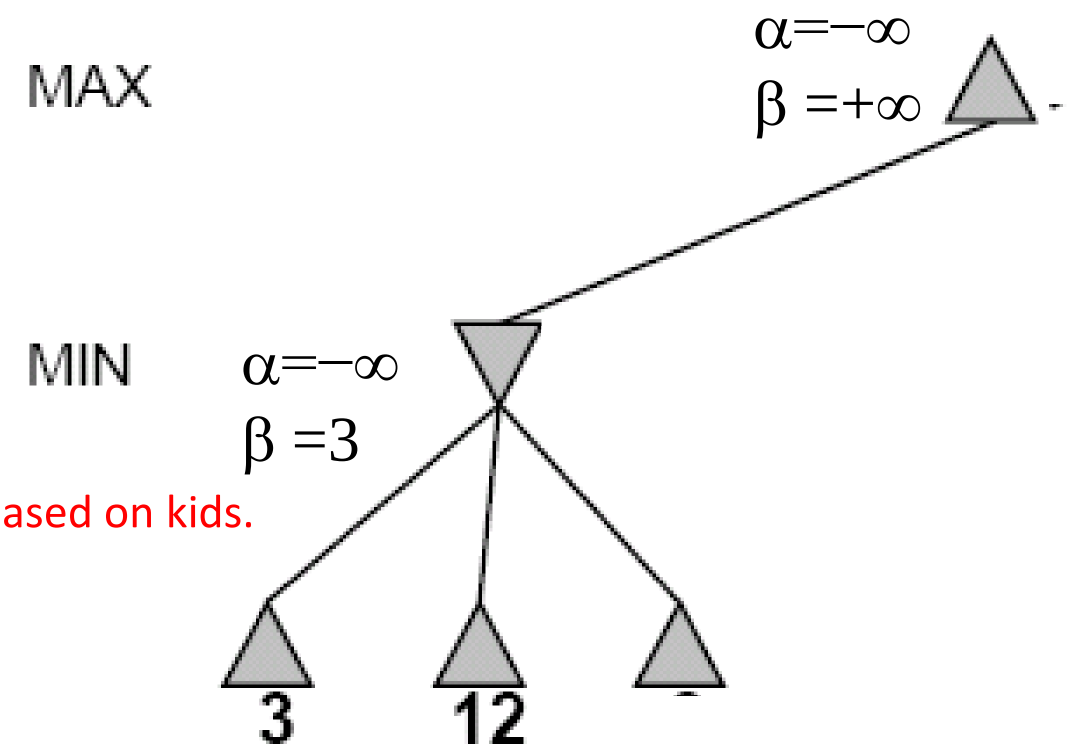
Do DF-search until first leaf



Alpha-Beta Example (continued)

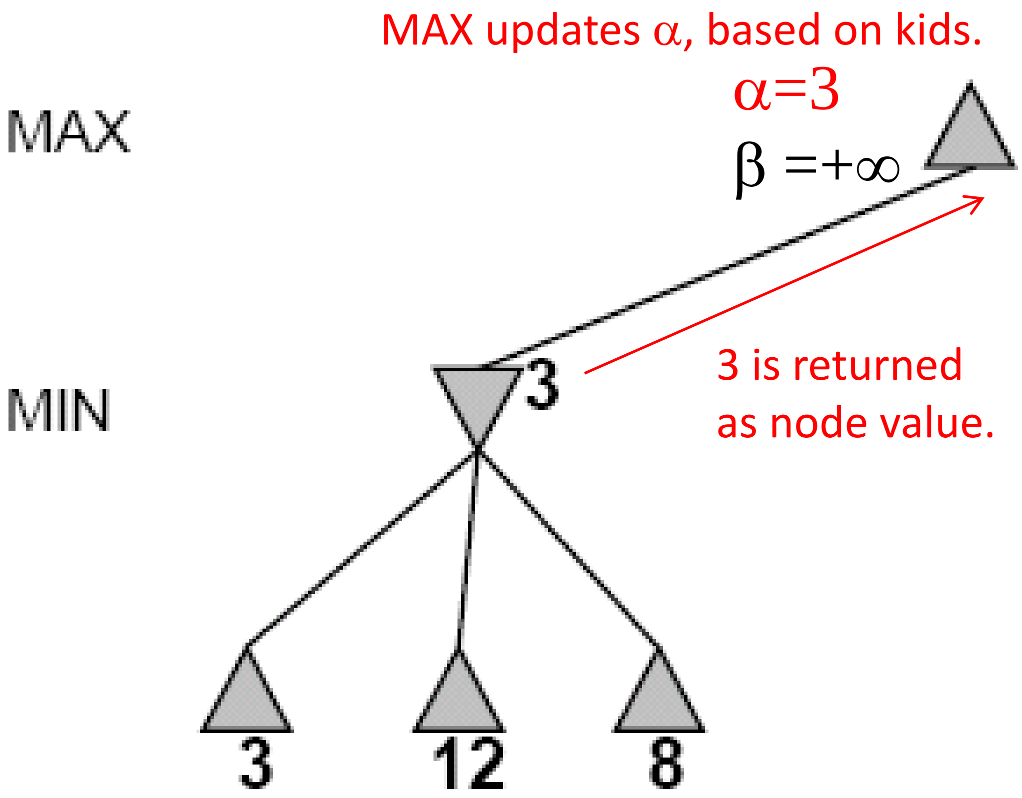


Alpha-Beta Example (continued)

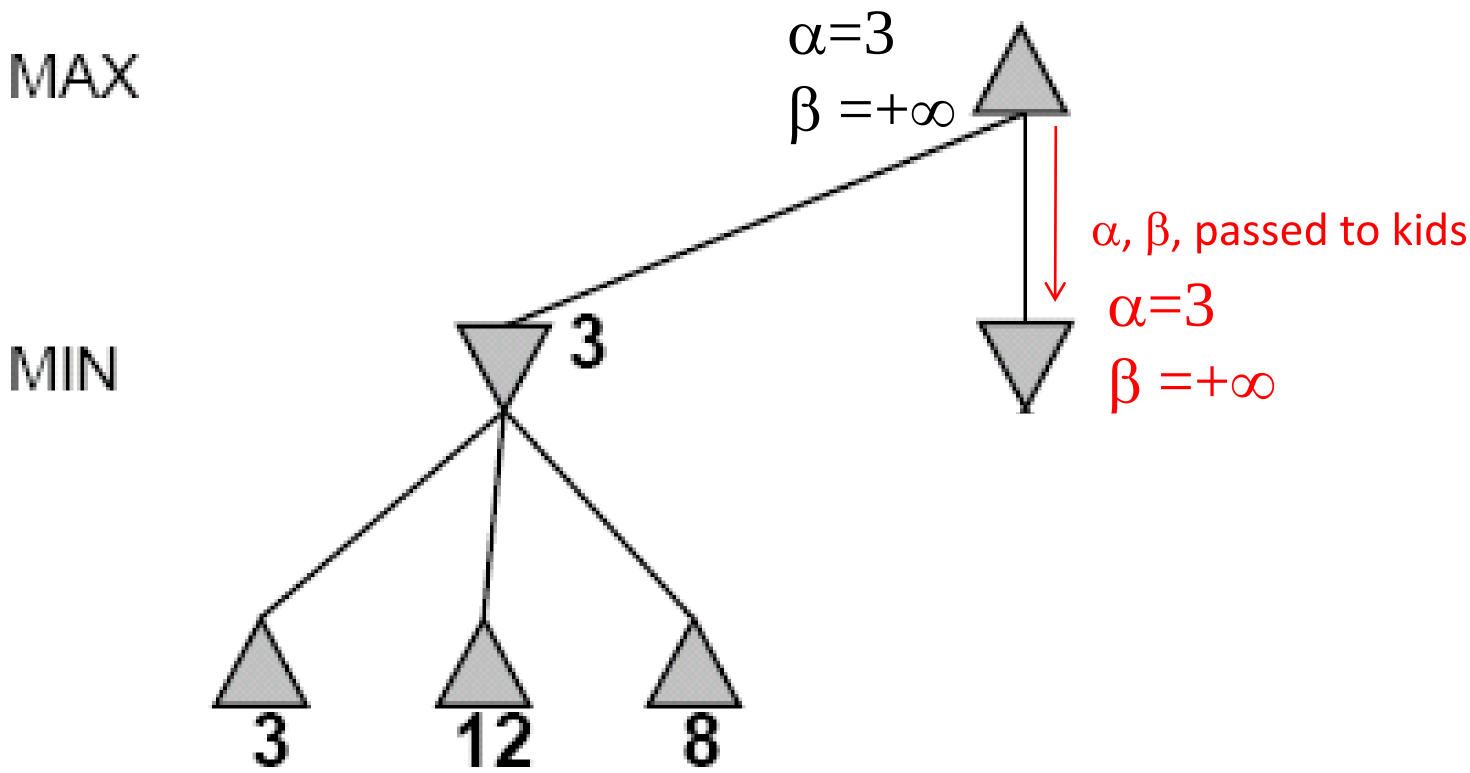


MIN updates β, based on kids.
No change.

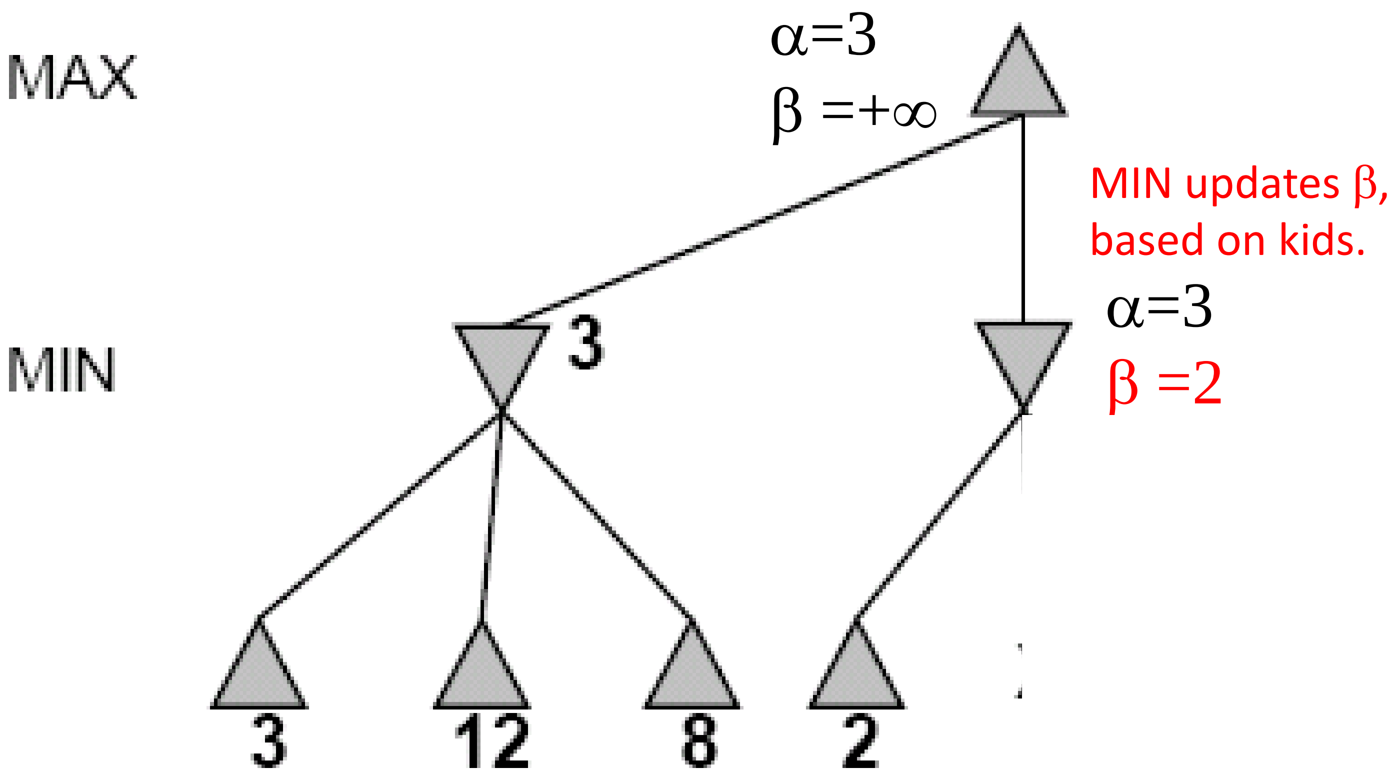
Alpha-Beta Example (continued)



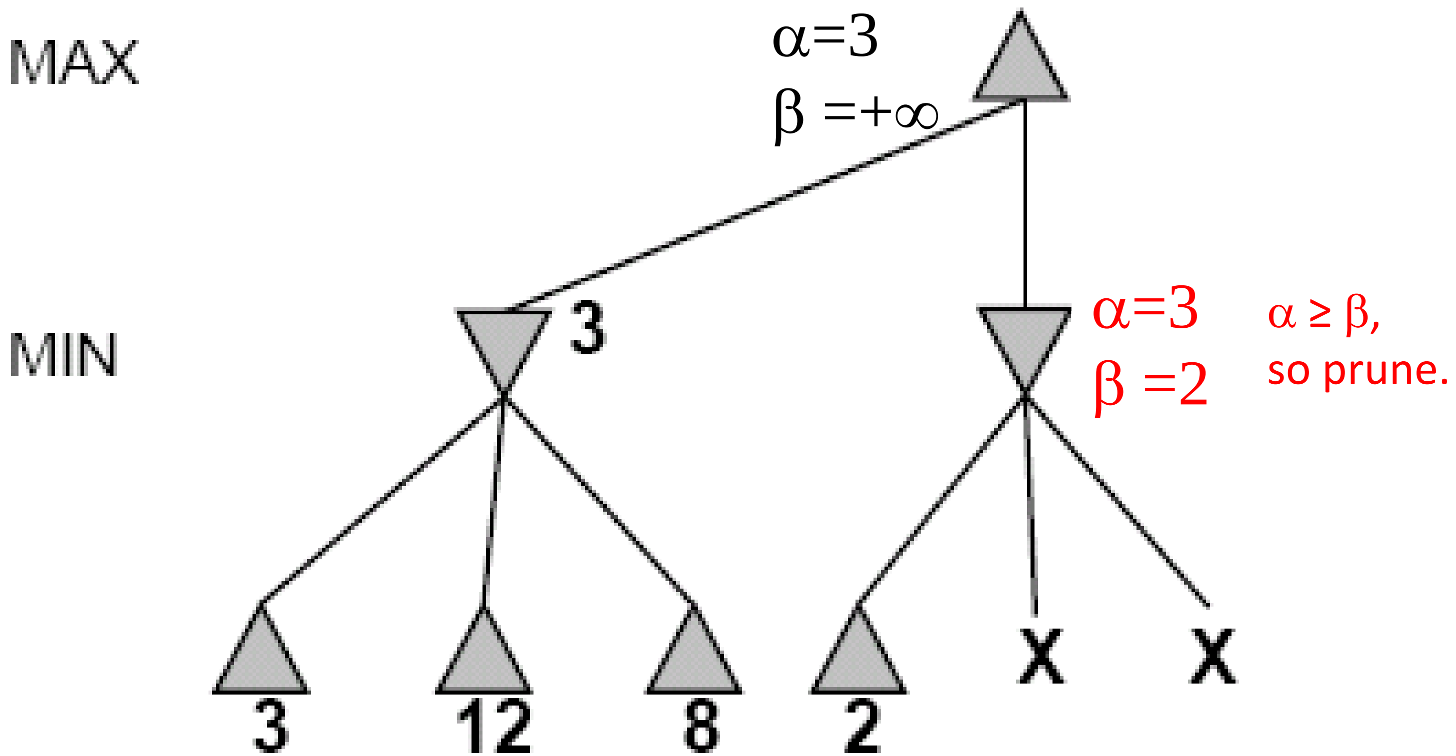
Alpha-Beta Example (continued)



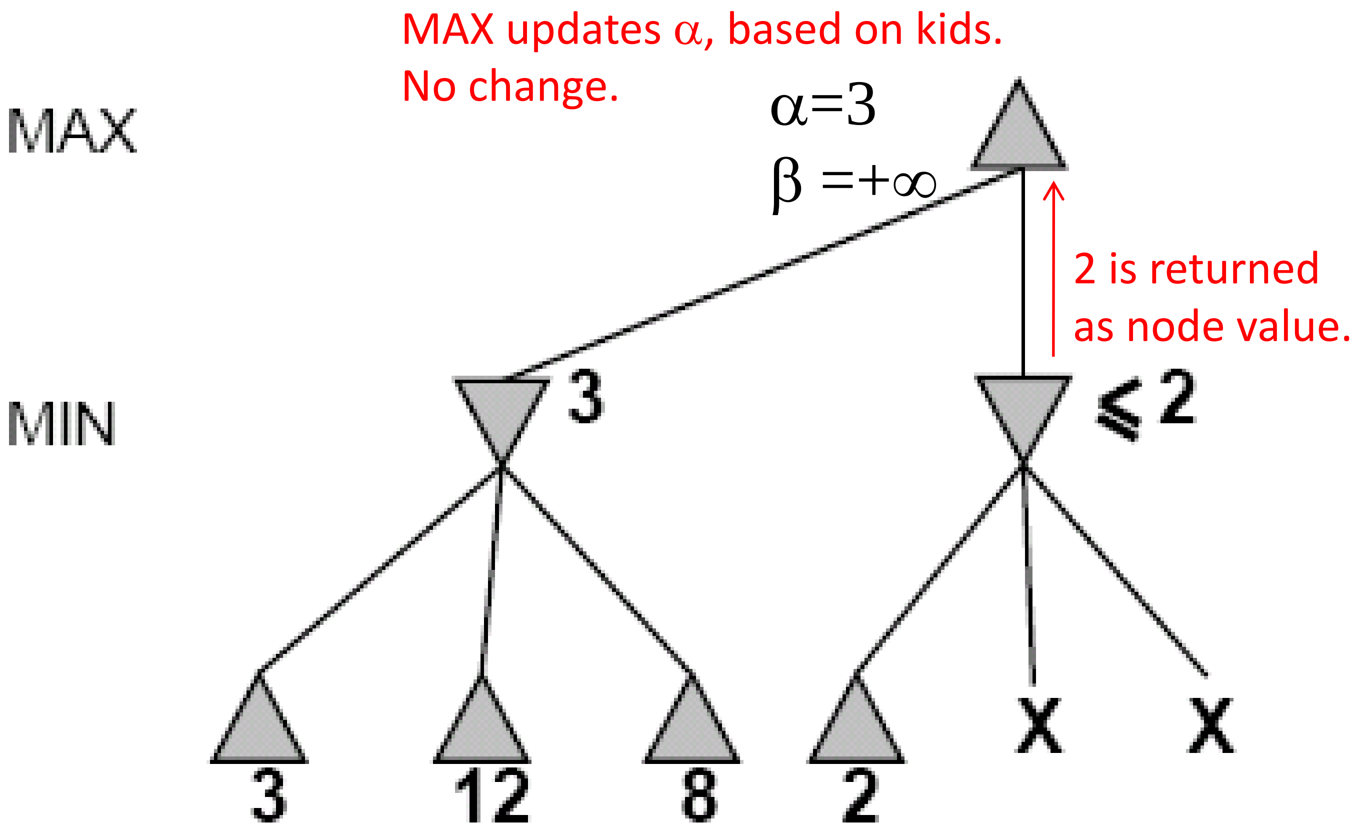
Alpha-Beta Example (continued)



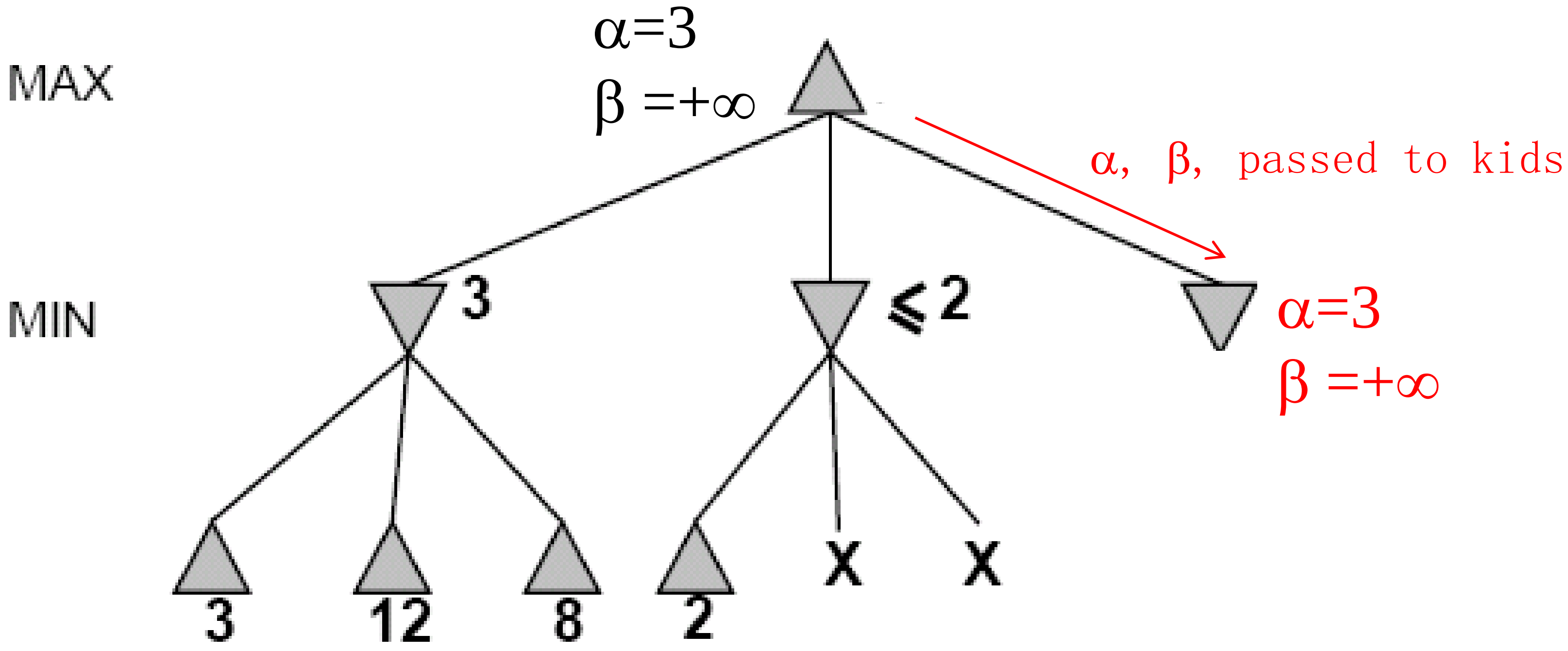
Alpha-Beta Example (continued)



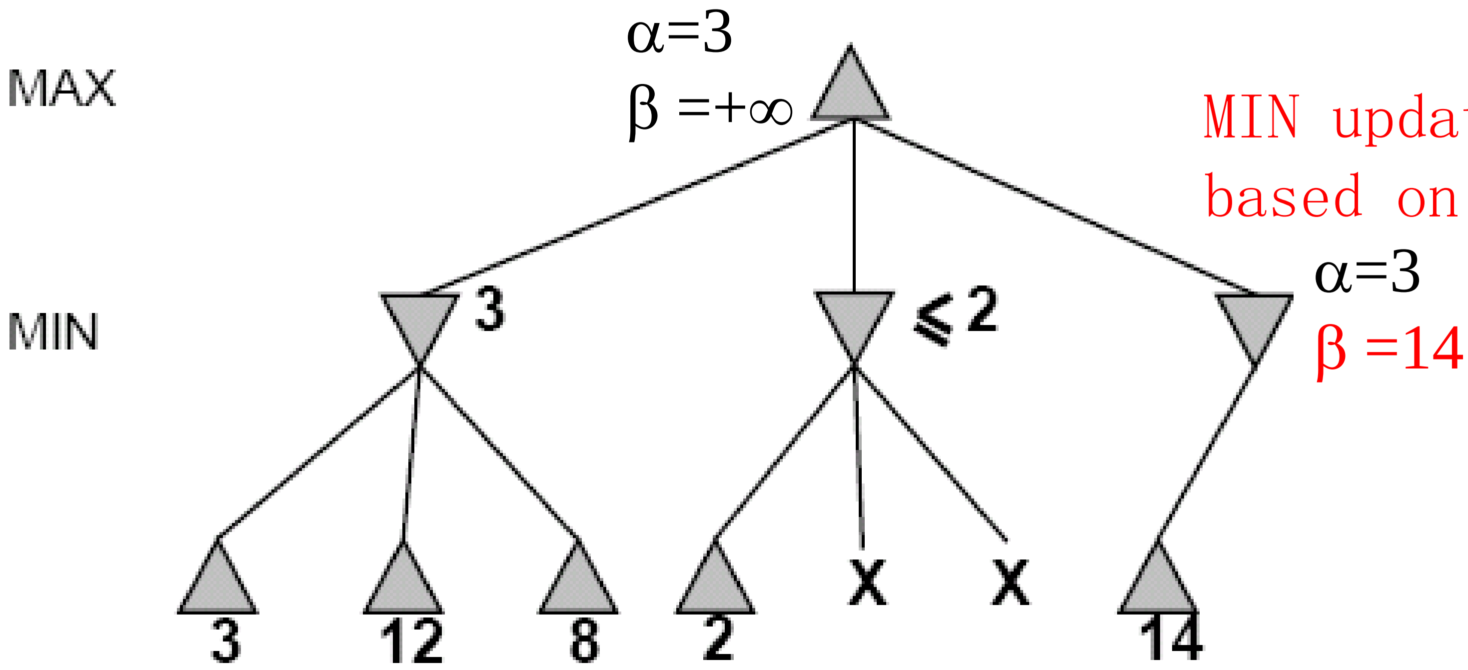
Alpha-Beta Example (continued)



Alpha-Beta Example (continued)

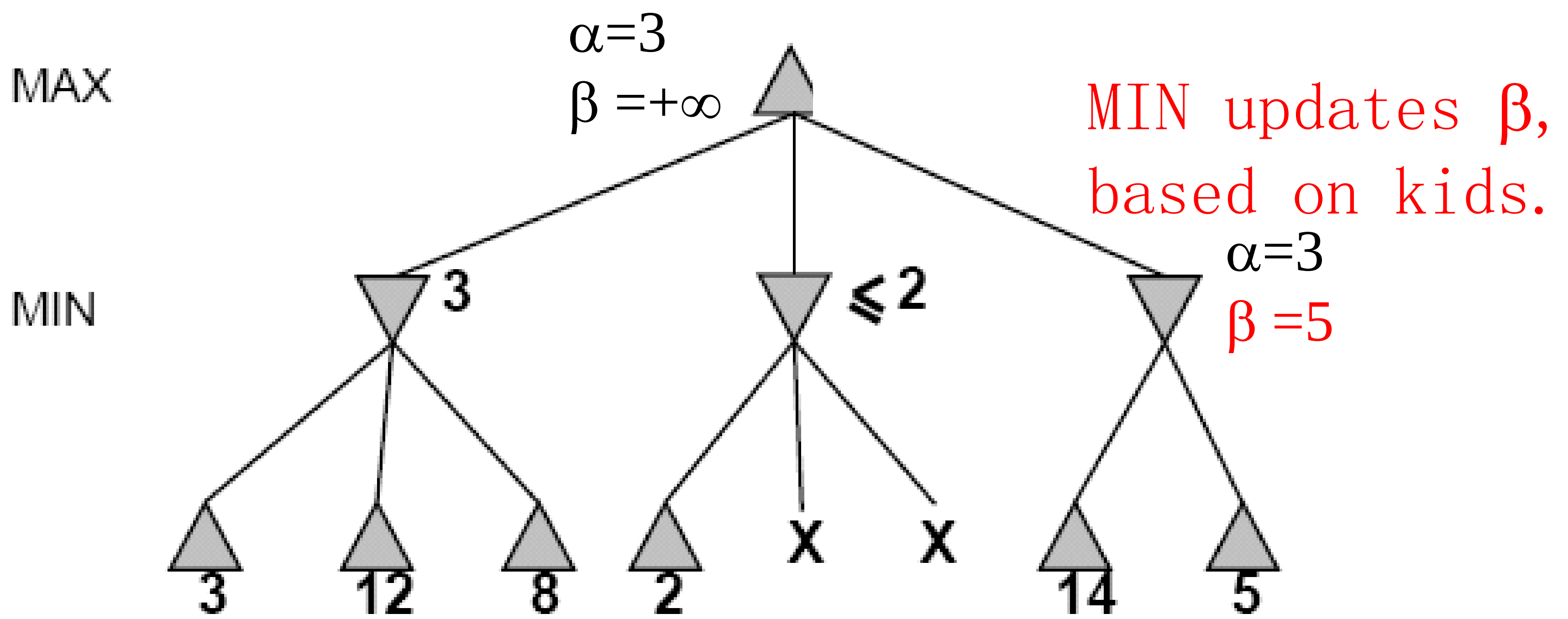


Alpha-Beta Example (continued)

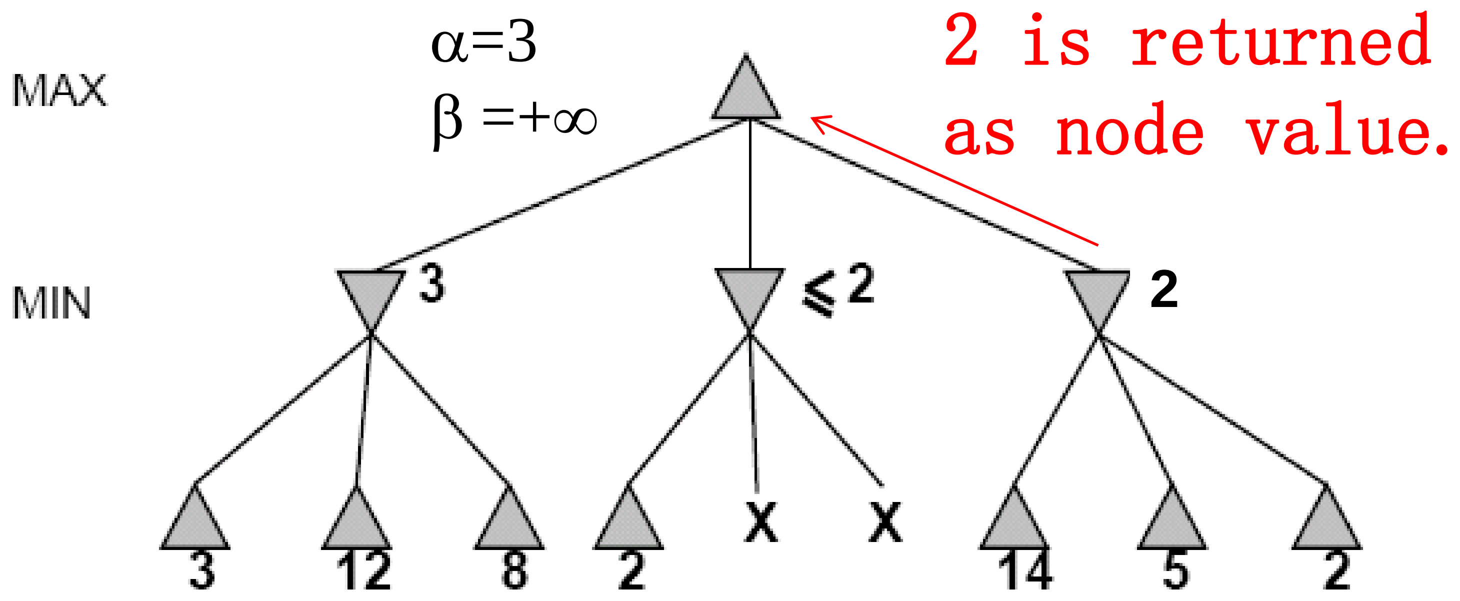


MIN updates β, based on kids.

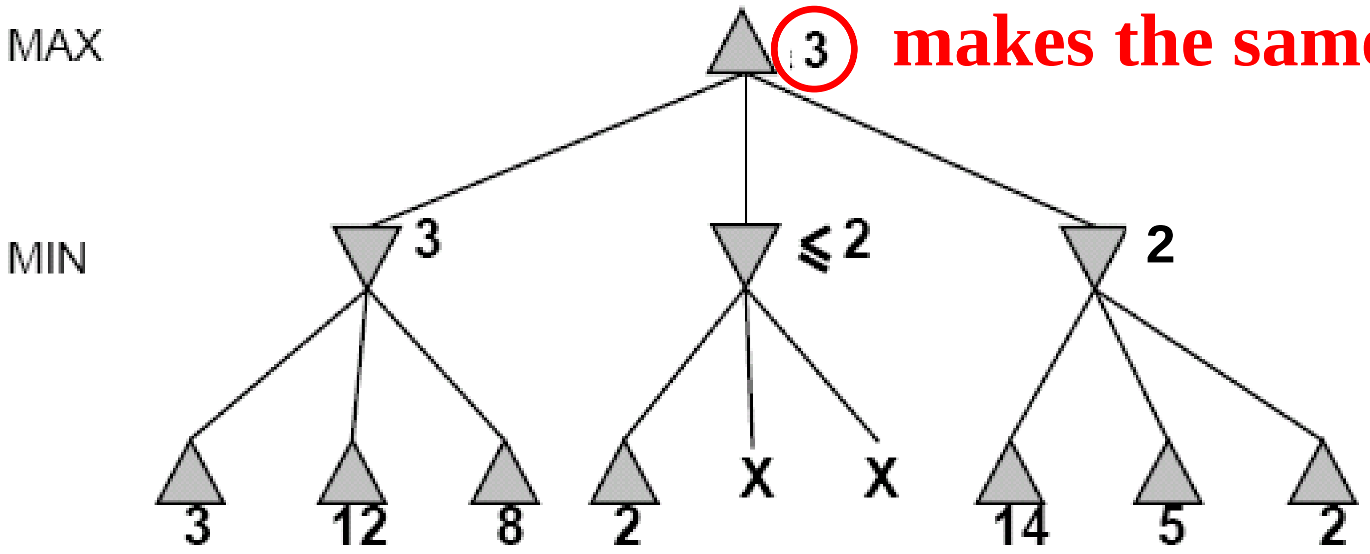
Alpha-Beta Example (continued)



Alpha-Beta Example (continued)



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

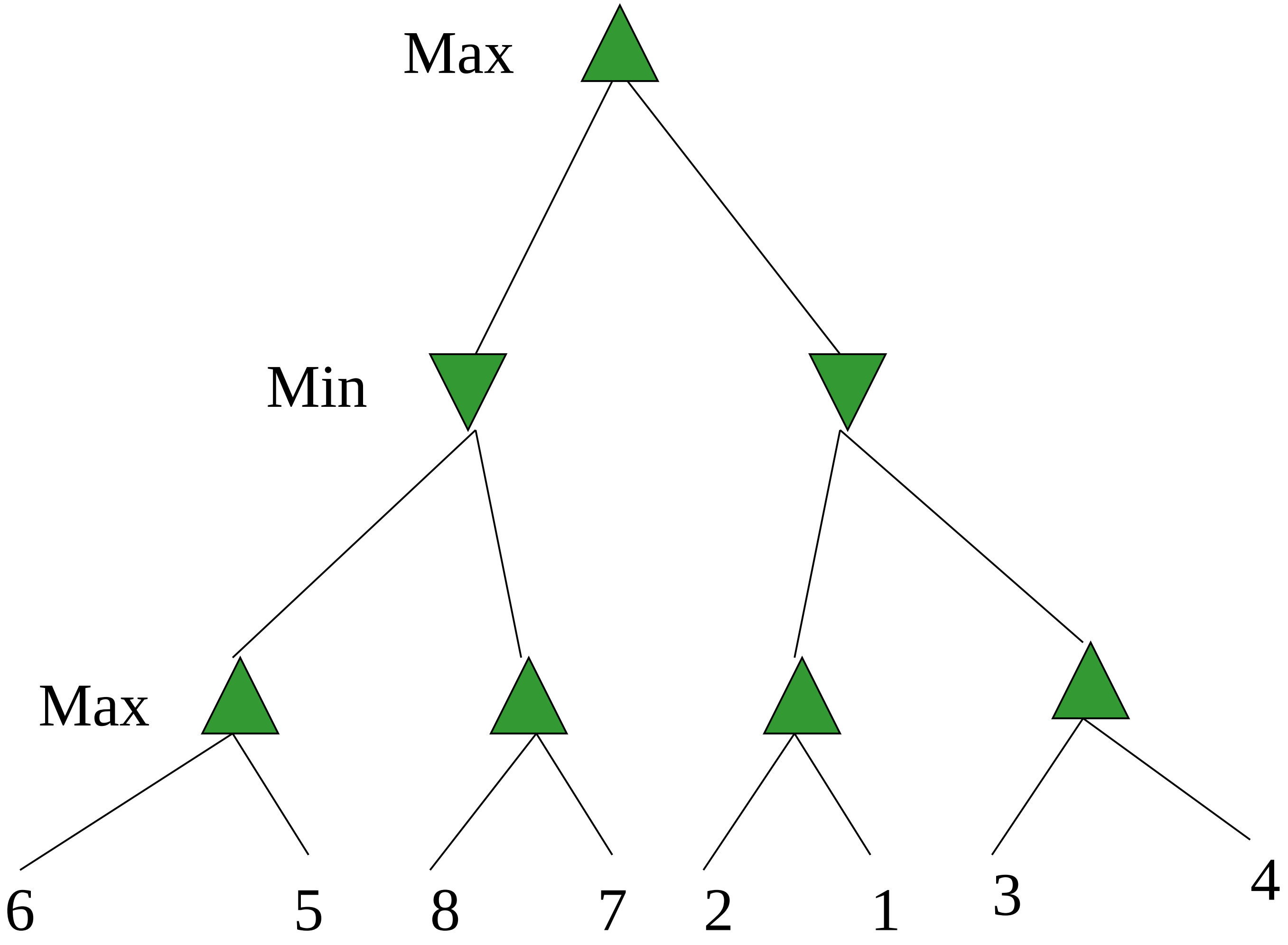
- Best Case
  - each player's best move is the left-most child (i.e., evaluated first)
  - in practice, performance is closer to best rather than worst-case

- In practice often get $O(b^{m/2})$ rather than $O(b^m)$ [b =max. number of child., m =max. depth of any node]
  - e.g., in chess go from $b \sim 35$ to $m \sim 6$
    - 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
- Repeated states are again possible
  - Store them in memory

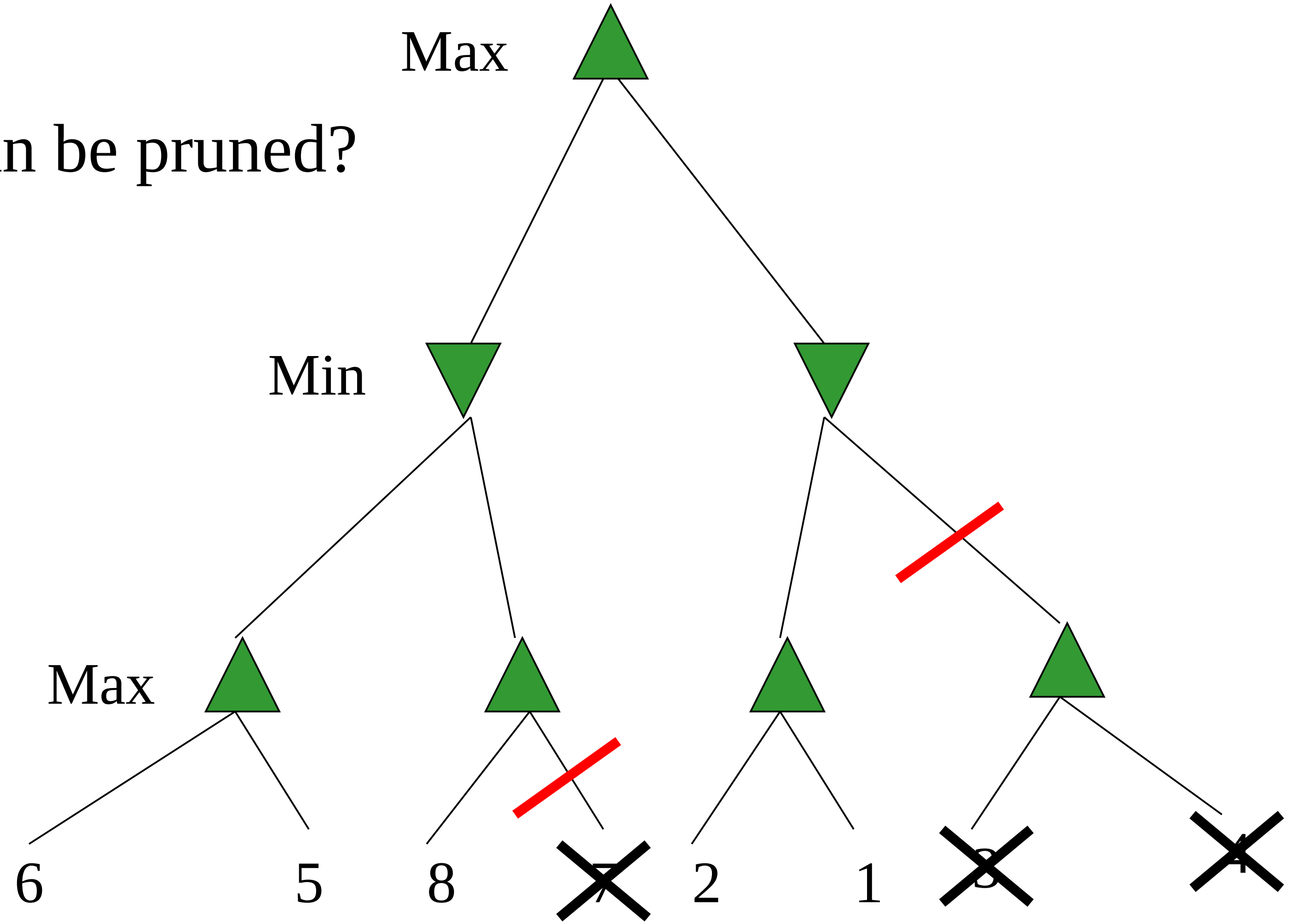
Example: the exact mirror image of the first example



which nodes can be pruned?

Example: the exact mirror image of the first example

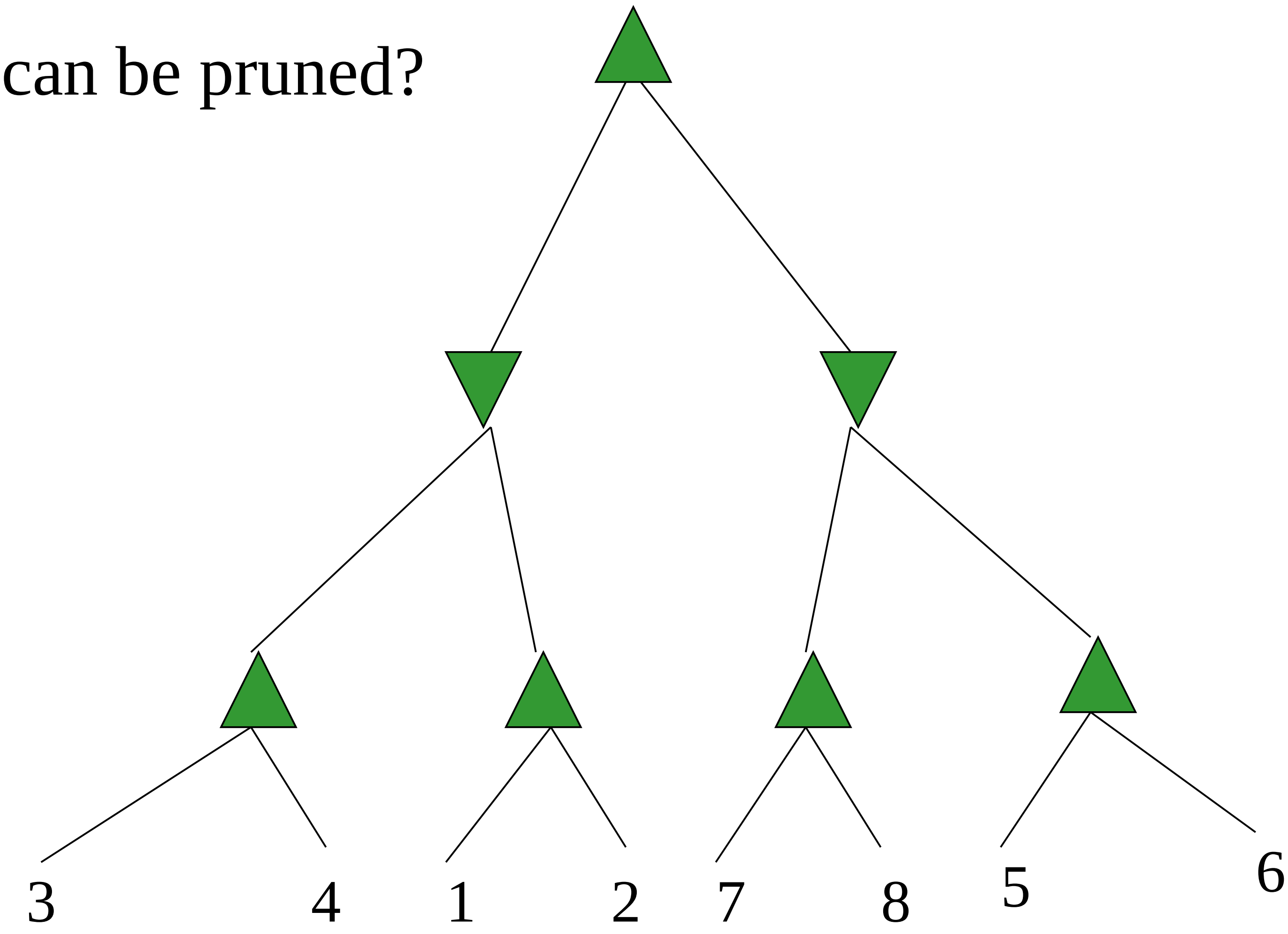
which nodes can be pruned?



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

Example

which nodes can be pruned?



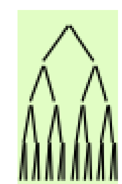
Answer: NONE! Because the most favorable nodes for both are explored last (i.e., in the diagram, are on the right-hand side).

Game-tree search in the game of go

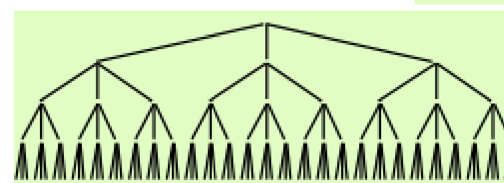
A game tree's size grows exponentially with both its depth an branching factor

- Go is huge: branching factor ≈ 200
- Game length ≈250 to 300 moves
- Number of paths in the game tree ≈ 10^{525} to 10^{620}
- Much too big for a normal game trees earch
- Comparison:
  - Number of atoms in universe: about 10^{80}
  - Number of particles in universe: about 10^{87}

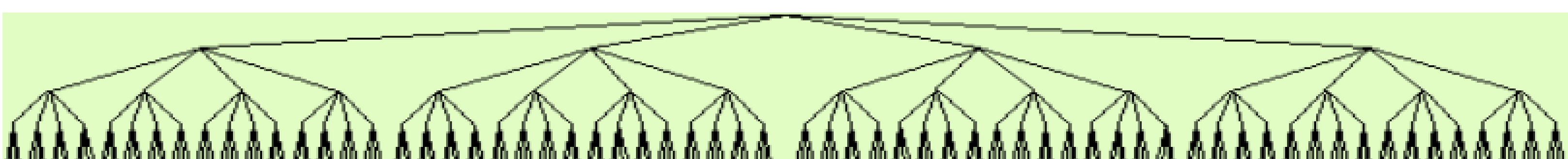
b =2



b =3



b =4



Game-tree search in the game of go

- During the past couple years, go programs have gotten much better
- Main reason: Monte Carlo roll-outs

- Basic idea: do a minimax search of a randomly selected subtree
- At each node that the algorithm visits,
  - It randomly selects some of the children
    There are heuristics for deciding how many
  - Calls itself recursively on these, ignores the others

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.

Homework

Design an algorithm for tic-tac-toe

- Including minimax algorithm
- Evaluation function
- Alpha-beta pruning