Overview

- **Computer programs that play 2-player games**
  - game-playing as search
  - with the complication of an opponent

- **General principles of game-playing and search**
  - game tree
  - minimax principle; impractical, but theoretical basis for analysis
  - evaluation functions; cutting off search; replace terminal leaf utility fn with eval fn
  - alpha-beta-pruning
  - heuristic techniques
  - games with chance

- **Status of Game-Playing Systems**
  - in chess, checkers, backgammon, Othello, etc, computers routinely defeat leading world players.

- **Motivation: multiagent competitive environments**
  - think of “nature” as an opponent
  - economics, war-gaming, medical drug treatment

Types of games

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

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

Search versus Games

- **Search – no adversary**
  - Solution is a path from start to goal, or a series of actions from start to goal
  - Heuristics and search techniques can find *optimal* solution
  - Evaluation function: estimate of cost from start to goal through given node
  - Actions have cost
  - 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
  - Board configurations have utility
  - Examples: chess, checkers, Othello, backgammon

Solving 2-player Games

- **Two players, fully observable environments, deterministic, turn-taking, zero-sum games of perfect information**
- **Examples: e.g., chess, checkers, tic-tac-toe**
- **Configuration of the board = unique arrangement of "pieces"**
- **Statement of Game as a Search Problem:**
  - **States** = board configurations
  - **Operators** = legal moves. The transition model
  - **Initial State** = current configuration
  - **Goal** = winning configuration
  - **payoff function (utility)**= gives numerical value of outcome of the game
- **Two players, MIN and MAX taking turns. MIN/MAX will use search tree to find next move**
- **A working example: Grundy's game**
  - Given a set of coins, a player takes a set and divides it into two unequal sets. The player who cannot do uneven split, looses.
  - What is a state? Moves? Goal?

Game Trees: Tic-tac-toe

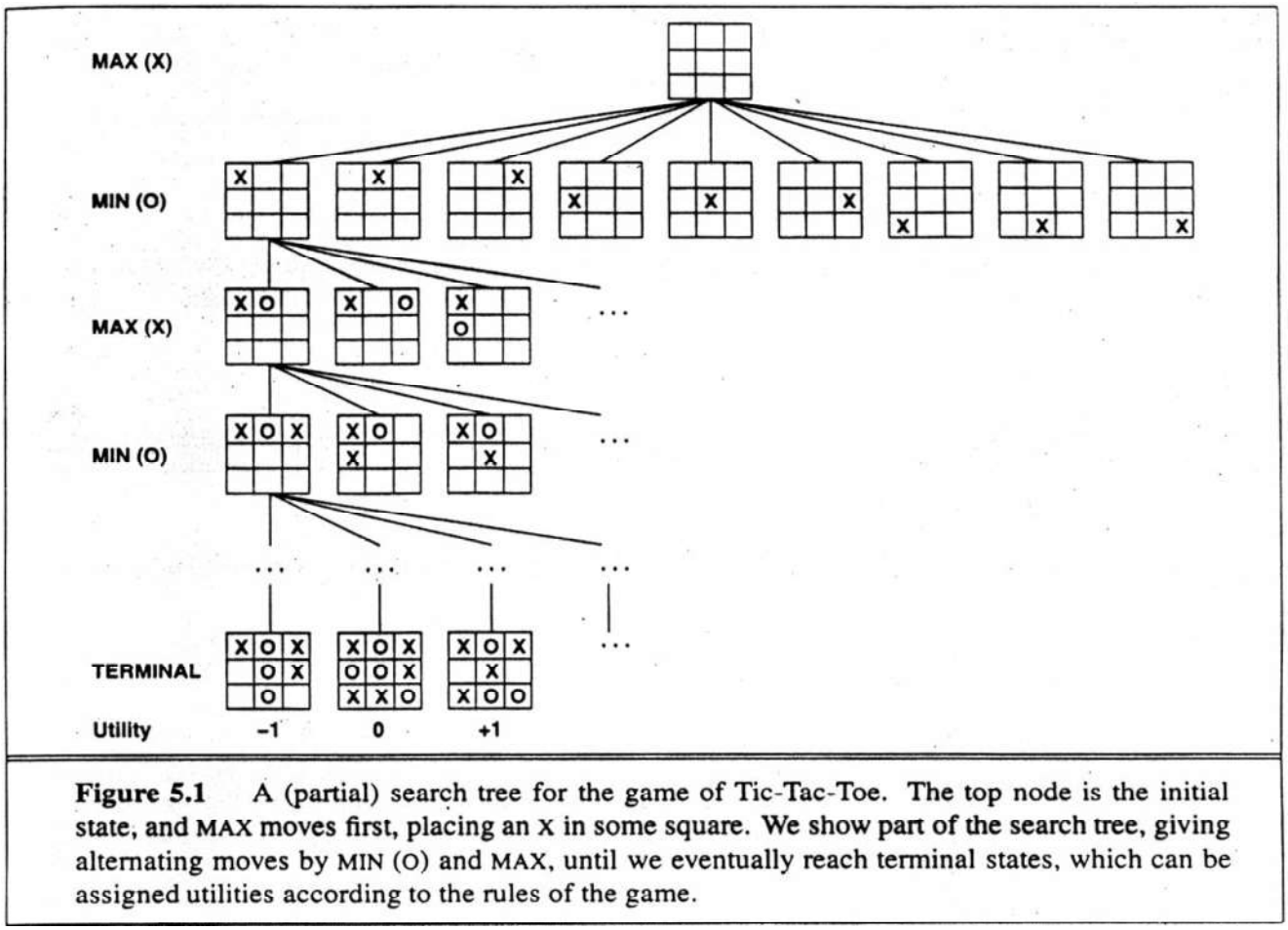


Figure 5.1 A (partial) search tree for the game of Tic-Tac-Toe. The top node is the initial state, and MAX moves first, placing an X in some square. We show part of the search tree, giving alternating moves by MIN (O) and MAX, until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

How do we search this tree to find the optimal move?

The Minimax Algorithm

- Designed to find the optimal strategy or just best first move for MAX
  - Optimal strategy is a solution tree

Brute-force:

- 1. Generate the whole game tree to leaves
- 2. Apply utility (payoff) function to leaves
- 3. Back-up values from leaves toward the root:
  - a Max node computes the max of its child values
  - a Min node computes the min of its child values
- 4. When value reaches the root: choose max value and the corresponding move.

Minimax:

Search the game-tree in a DFS manner to find the value of the root.

Game Trees

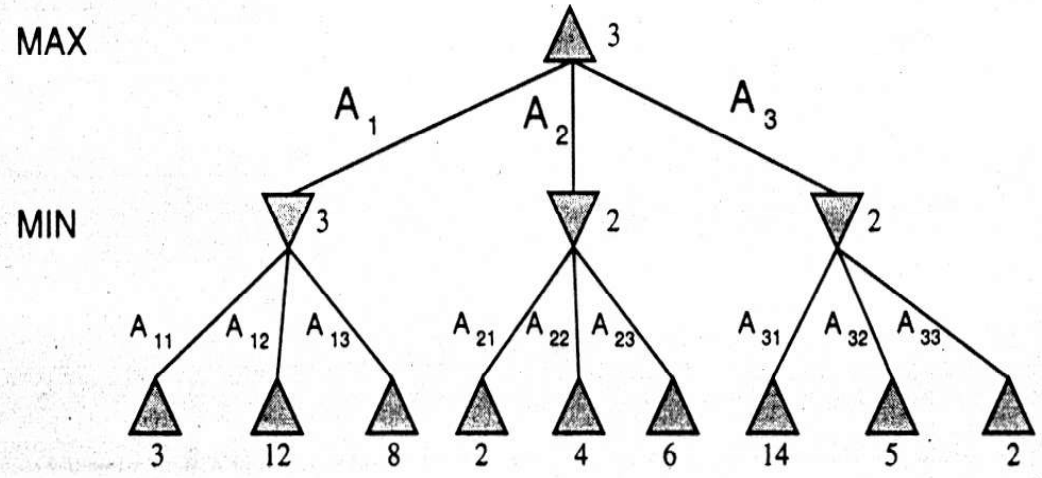
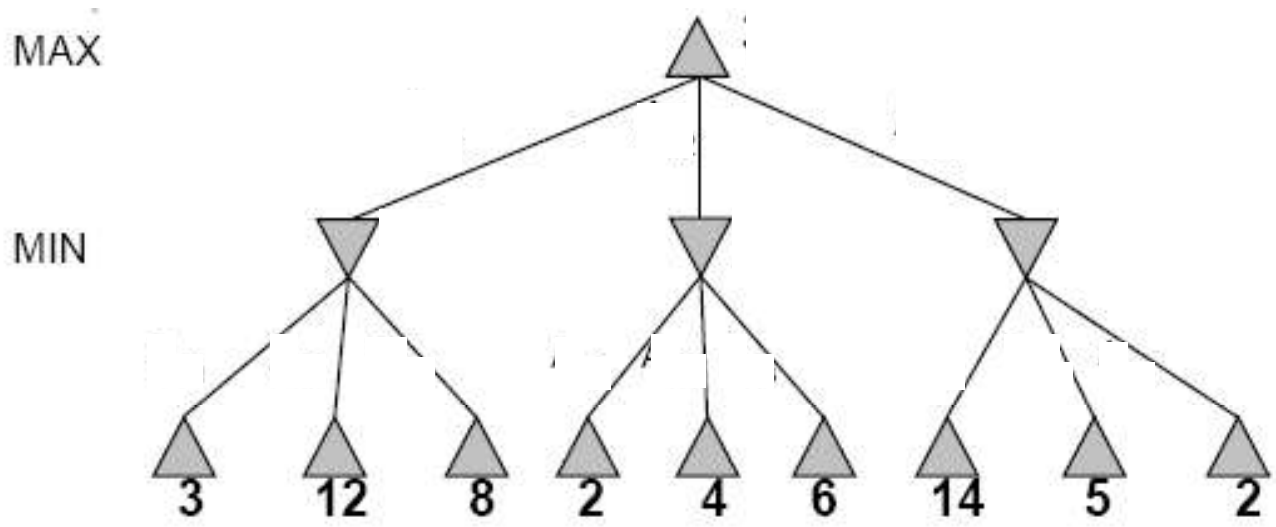
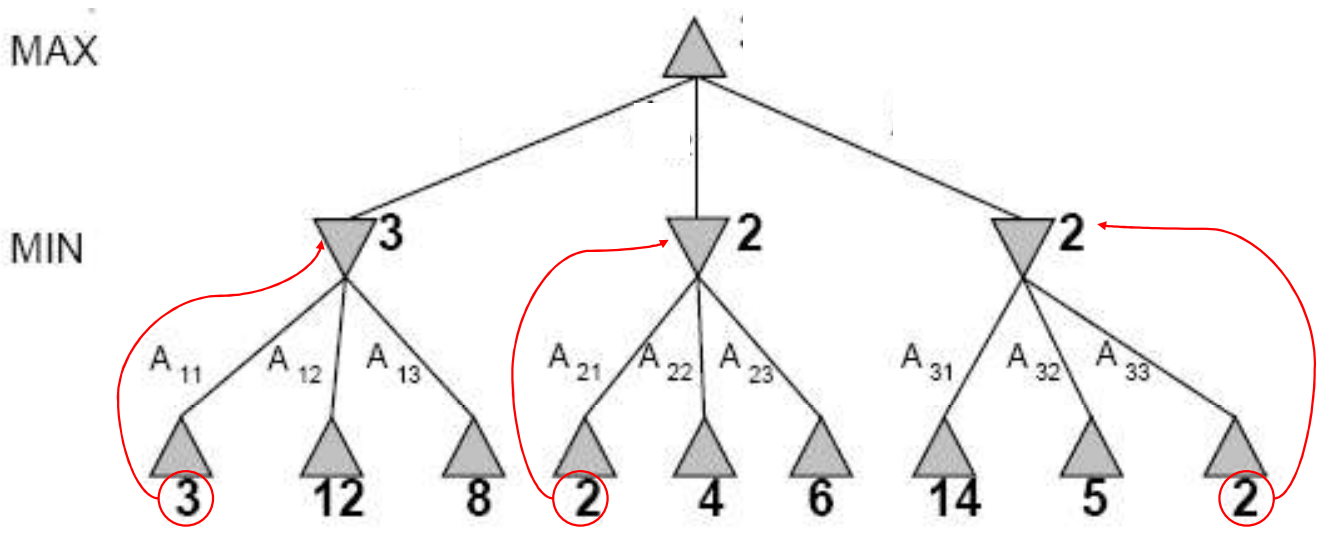


Figure 5.2 A two-ply game tree as generated by the minimax algorithm. The △ nodes are moves by MAX and the ▽ nodes are moves by MIN. The terminal nodes show the utility value for MAX computed by the utility function (i.e., by the rules of the game), whereas the utilities of the other nodes are computed 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

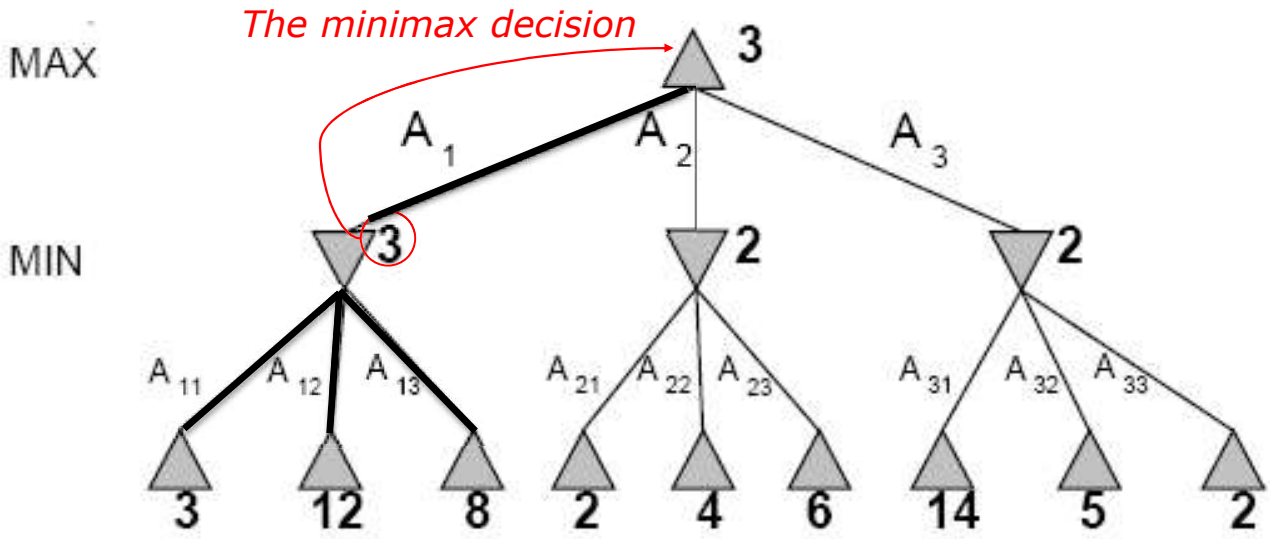


Two-Ply Game Tree



Two-Ply Game Tree

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



A solution tree is highlighted

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)
  - $b^d \approx 35^{100} \approx 10^{154}$ nodes!!
  - → **exact solution completely infeasible**

- **It is usually impossible to develop the whole search tree. Instead develop part of the tree up to some depth and evaluate leaves using an evaluation fn**
- **Optimal strategy (solution tree) too large to store.**

Static (Heuristic) Evaluation Functions

- **An Evaluation Function:**
  - Estimates how good the current board configuration is for a player
  - Typically, one figures how good it is for the player, and how good it is for the opponent, and subtracts the opponents score from the player
  - 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**
- **Example:**
  - Evaluating chess boards
  - Checkers
  - Tic-tac-toe

Applying MiniMax to tic-tac-toe

- **The static evaluation function heuristic**

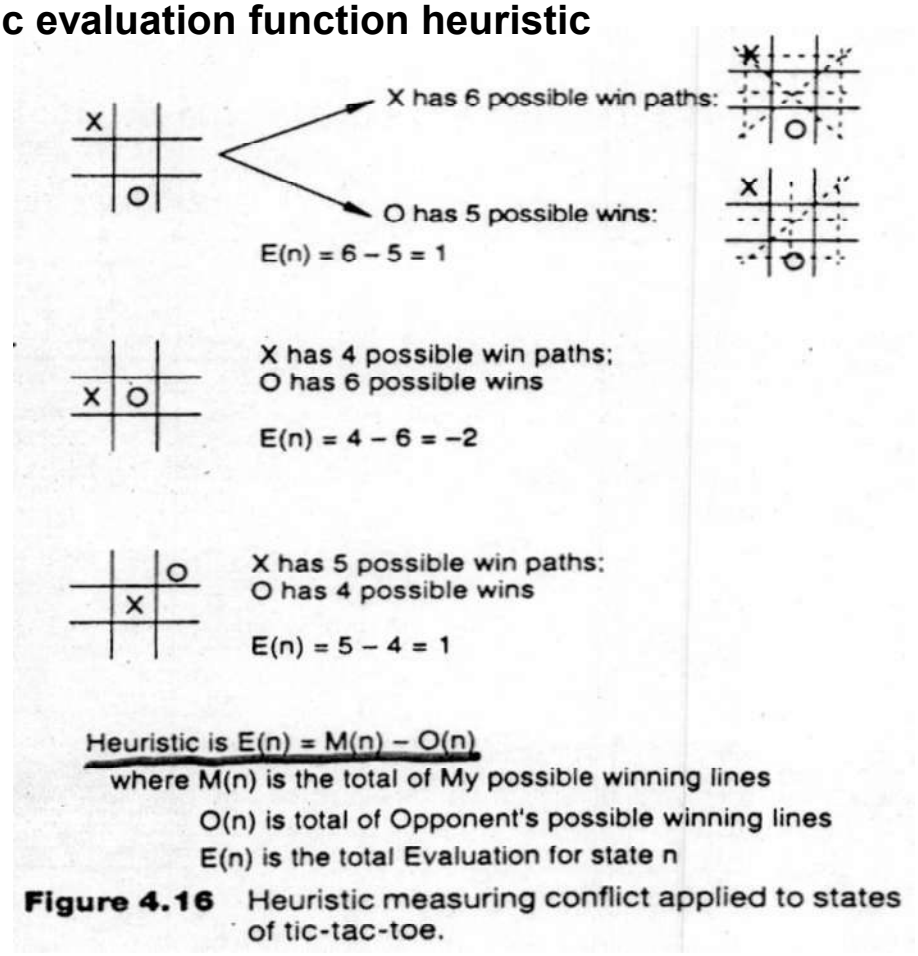


Figure 4.16 Heuristic measuring conflict applied to states of tic-tac-toe.

Backup Values

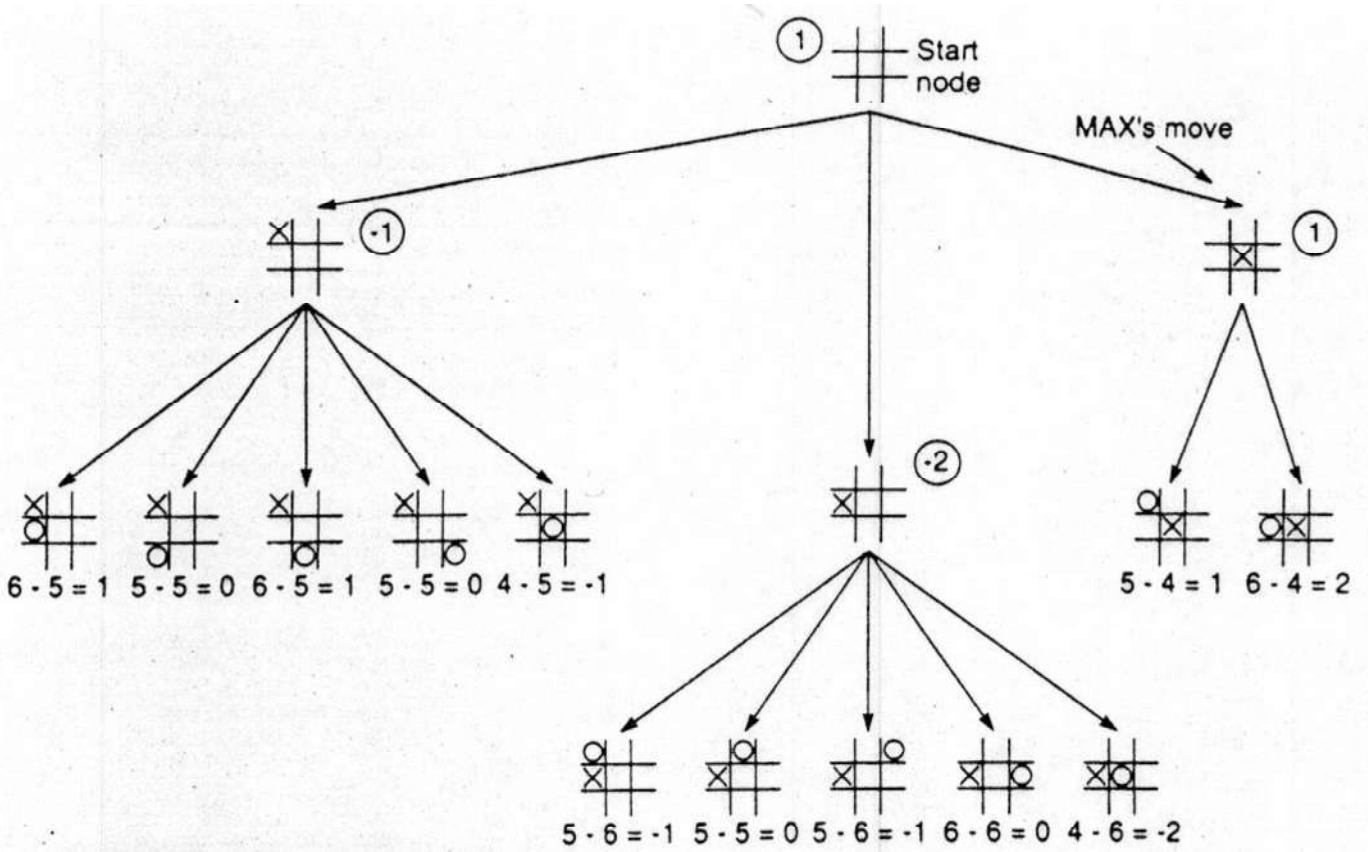


Figure 4.17 Two-ply minimax applied to the opening move of tic-tac-toe.

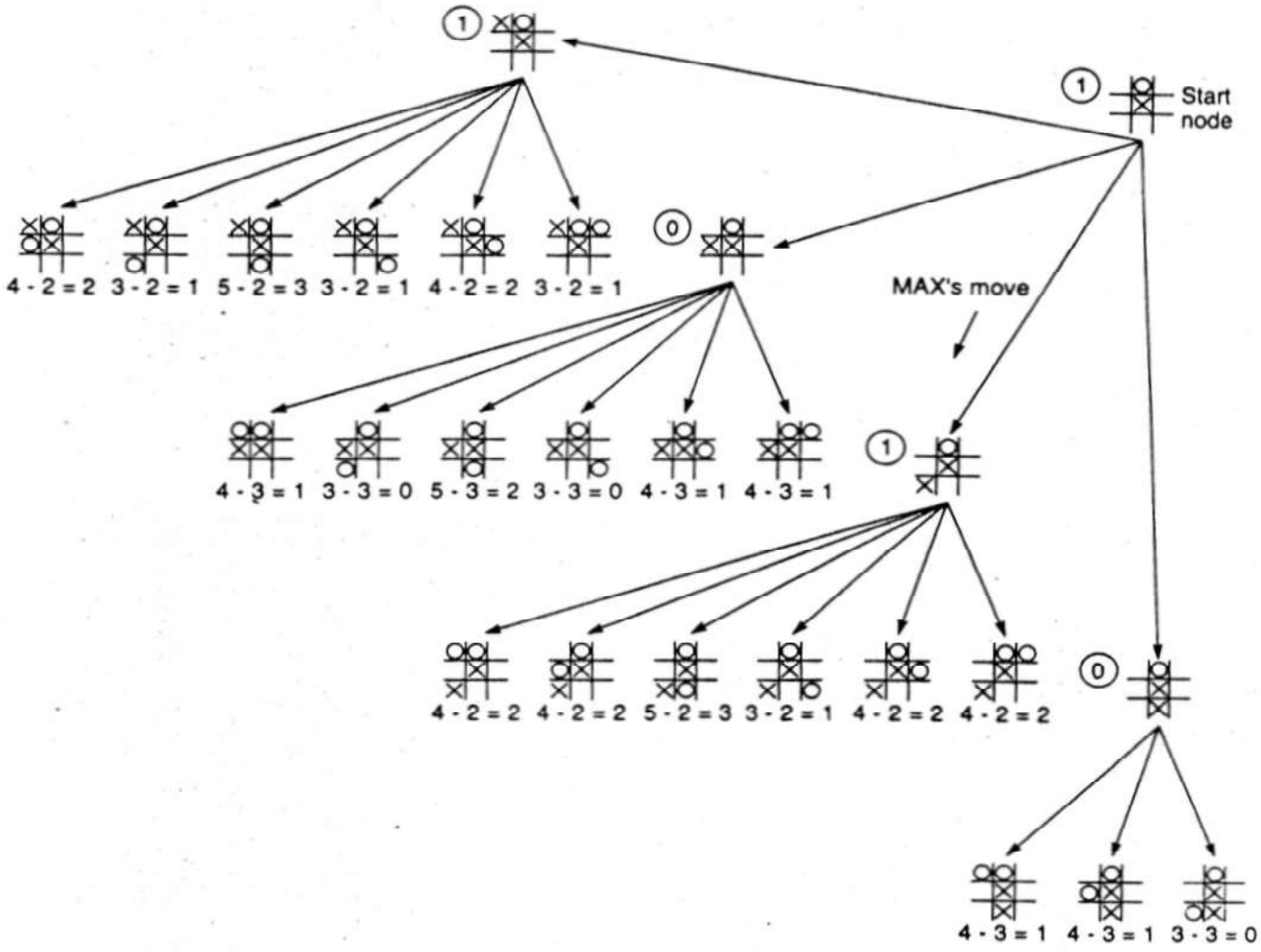


Figure 4.18 Two-ply minimax applied to **X**'s second move of tic-tac-toe.

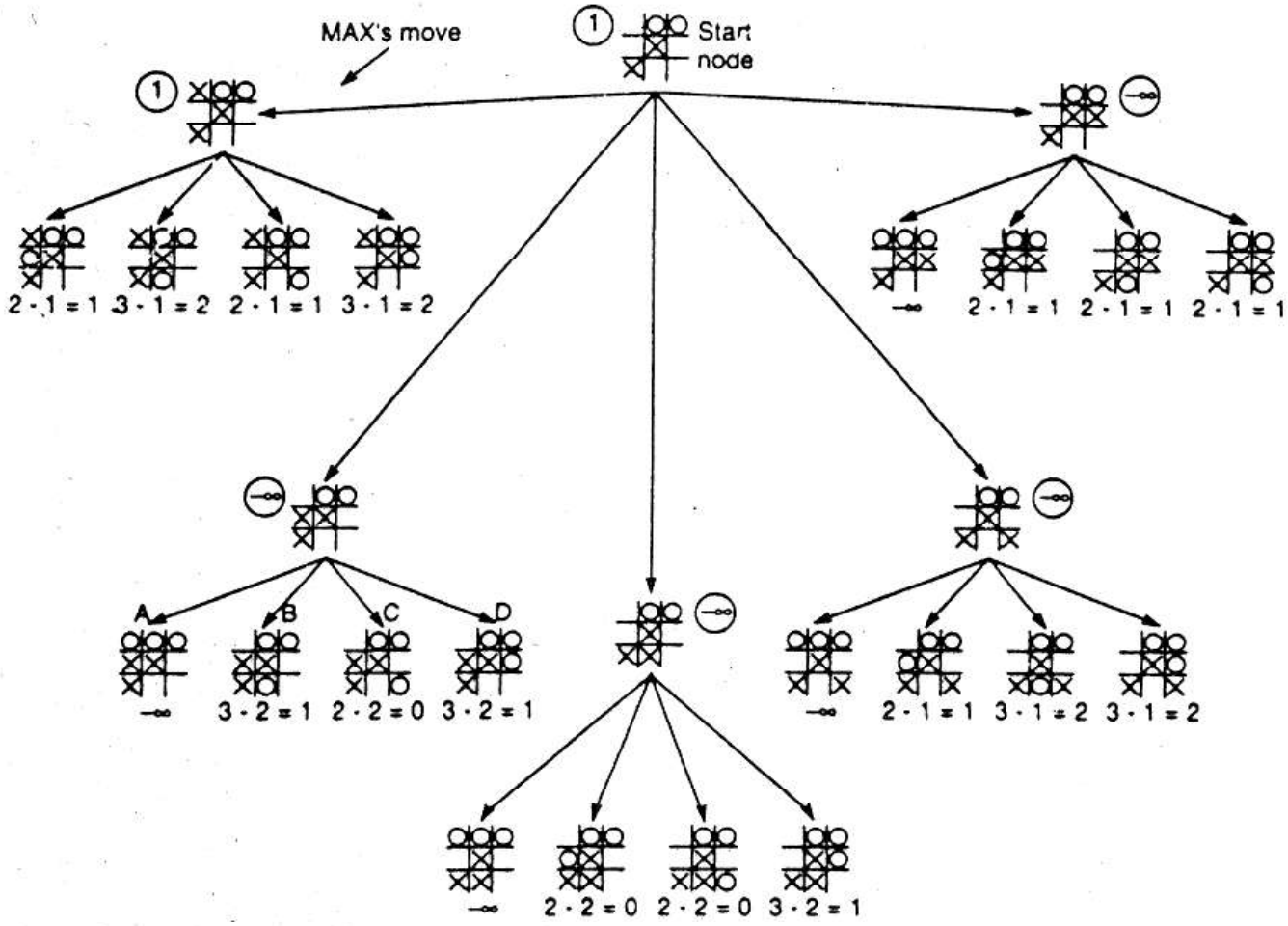
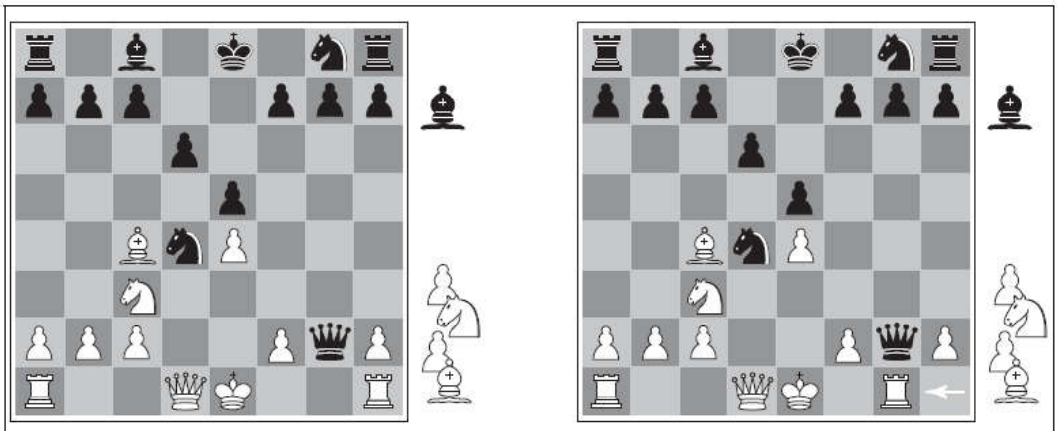


Figure 4.19 Two-ply minimax applied to **X**'s move near end game.

Feature-based evaluation functions

- Features of the state
- Features taken together define categories (equivalence) classes
- Expected value for each equivalence class
  - Too hard to compute
- Instead
  - Evaluation function = weighted linear combination of feature values



(a) White to move (b) White to move

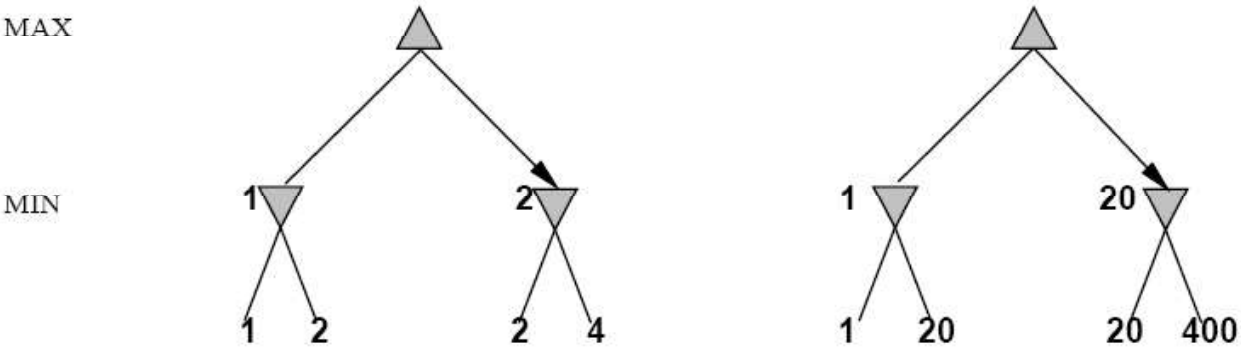
Figure 5.8 Two chess positions that differ only in the position of the rook at lower right. In (a), Black has an advantage of a knight and two pawns, which should be enough to win the game. In (b), White will capture the queen, giving it an advantage that should be strong enough to win.

For chess, typically *linear* weighted sum of features

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

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

Digression: Exact values don't matter



Behaviour is preserved under any *monotonic* transformation of EVAL

Only the order matters:
payoff in deterministic games acts as an *ordinal utility* function

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 you have a better alternative

- **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.**

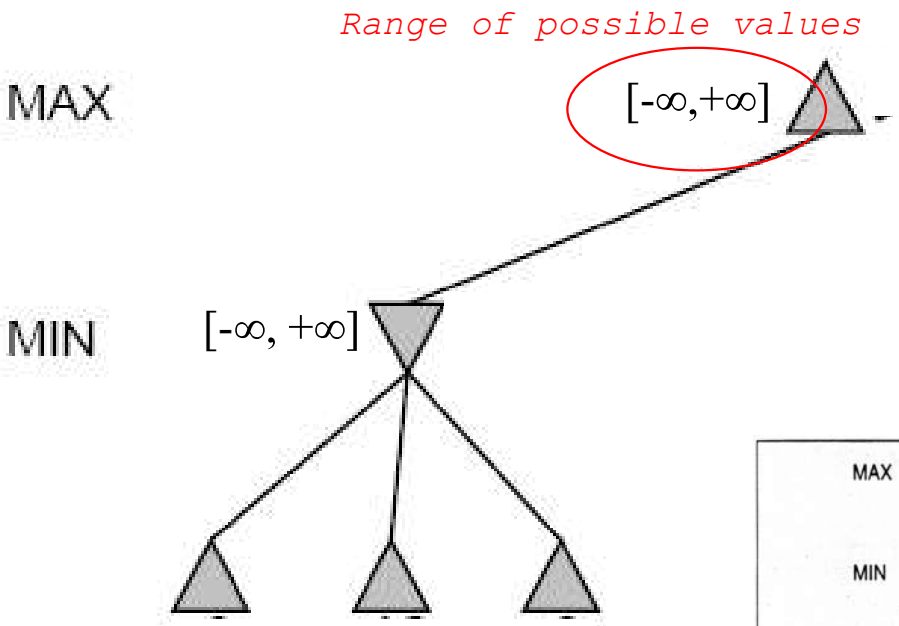
- **Contrast normal search:**
  - ANY node might be a winner.
  - ALL nodes must be considered.
  - (A* avoids this through knowledge, i.e., heuristics)

Alpha Beta Procedure

- **Idea:**
  - Do depth first search to generate partial game tree,
  - Give static evaluation function to leaves,
  - Compute bound on internal nodes.
- **α, β bounds:**
  - α value for max node means that max real value is at least α.
  - β for min node means that min can guarantee a value no more than β.
- **Computation:**
  - Pass current α/β down to children when expanding a node
  - Update α(Max)/β(Min) when node values are updated
    - α of MAX node is the max of children seen.
    - β of MIN node is the min of children seen.

Alpha-Beta Example

Do DF-search until first leaf



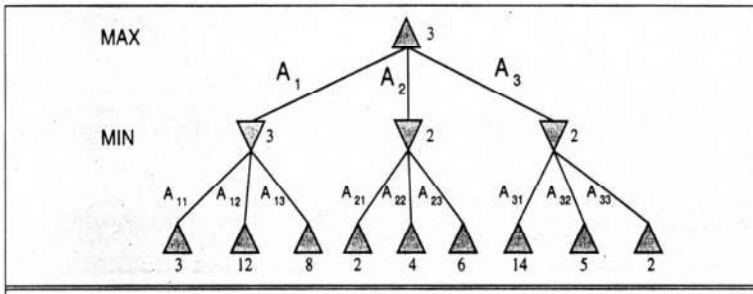
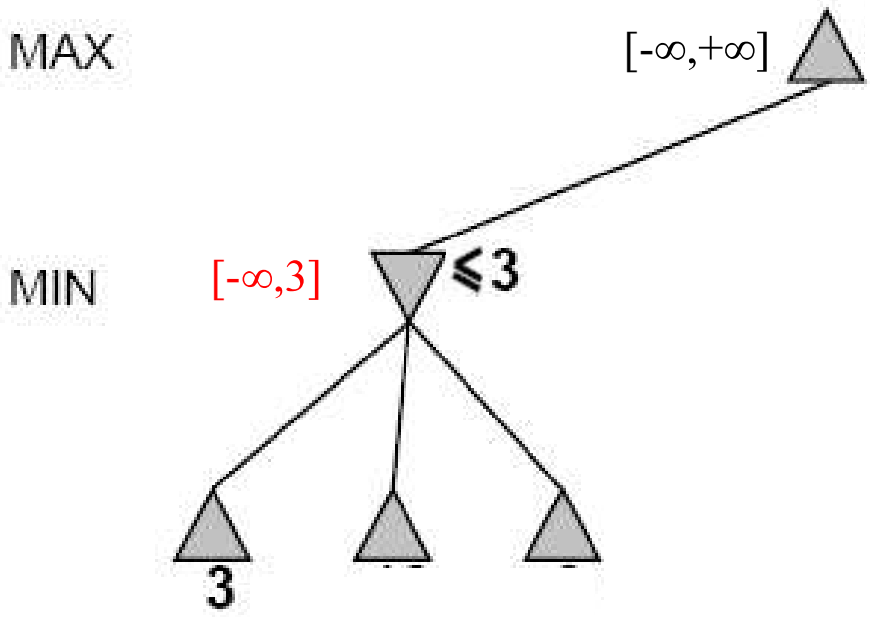
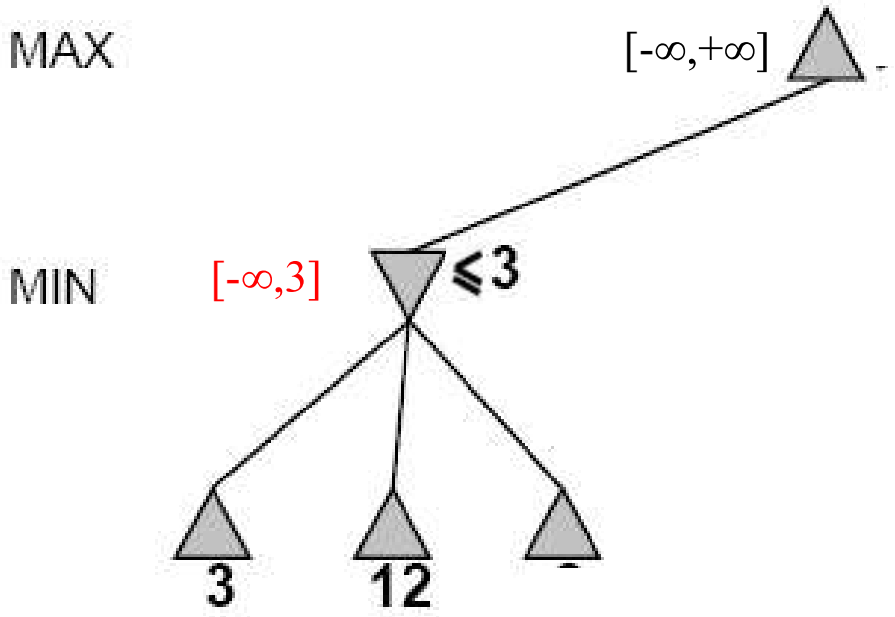


Figure 5.2 A two-ply game tree as generated by the minimax algorithm. The △ nodes are moves by MAX and the ▽ nodes are moves by MIN. The terminal nodes show the utility value for MAX computed by the utility function (i.e., by the rules of the game), whereas the utilities of the other nodes are computed 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}.

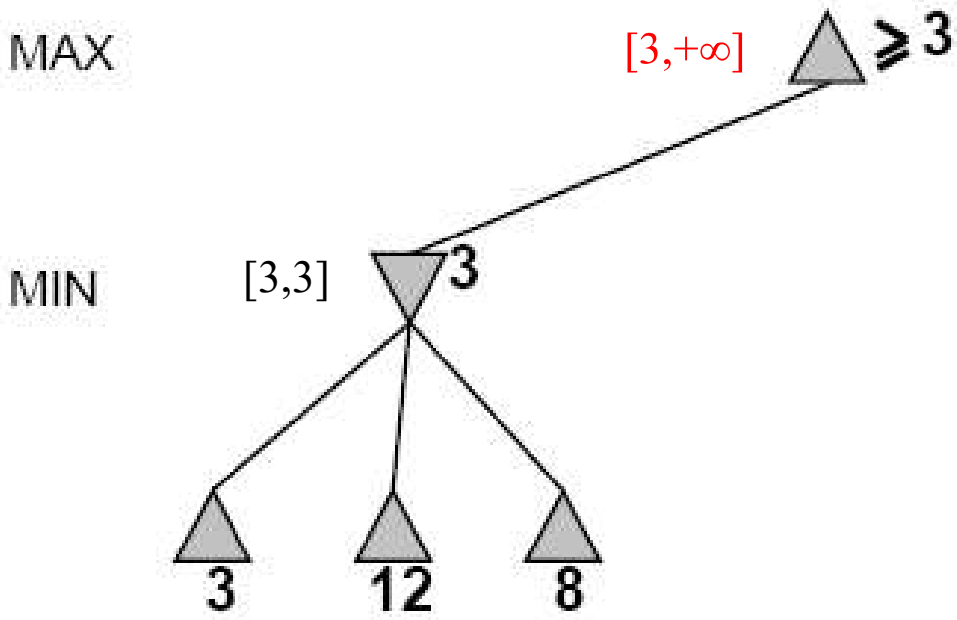
Alpha-Beta Example (continued)



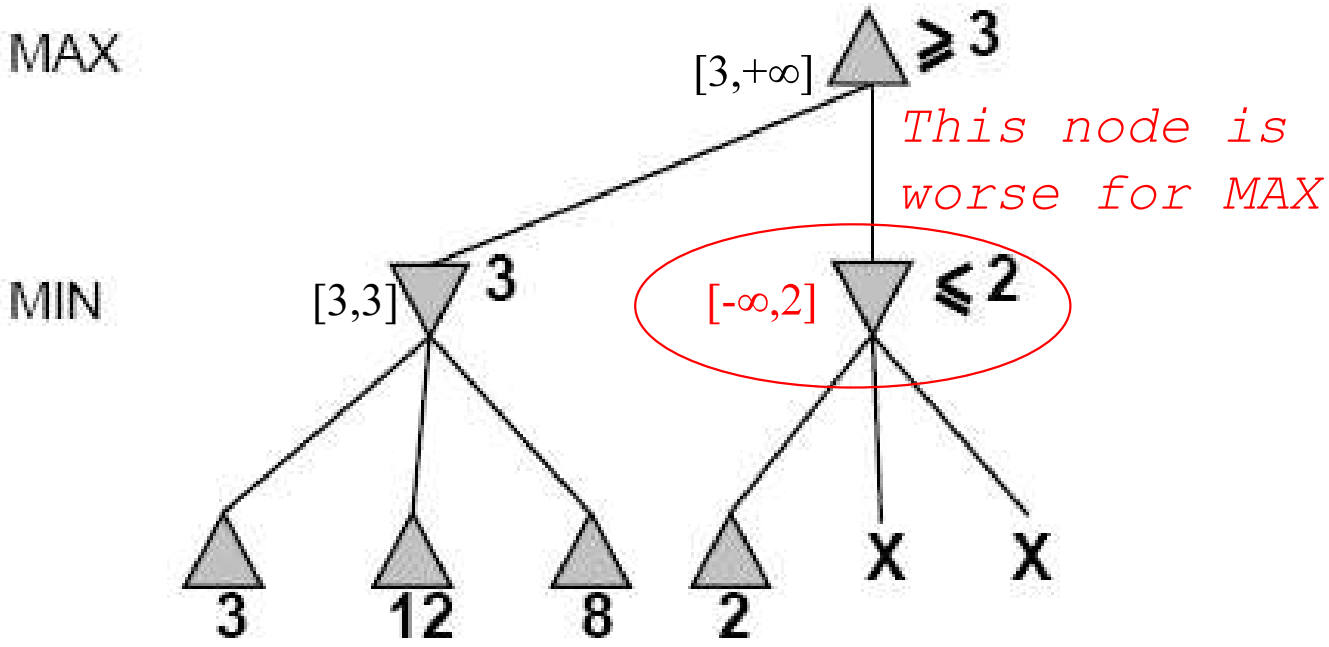
Alpha-Beta Example (continued)



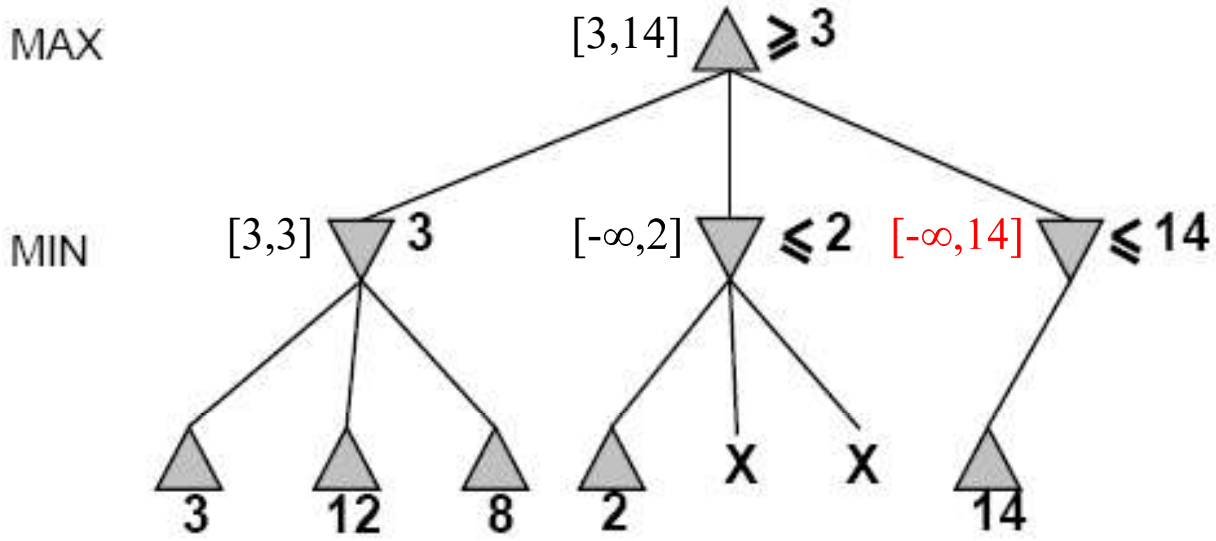
Alpha-Beta Example (continued)



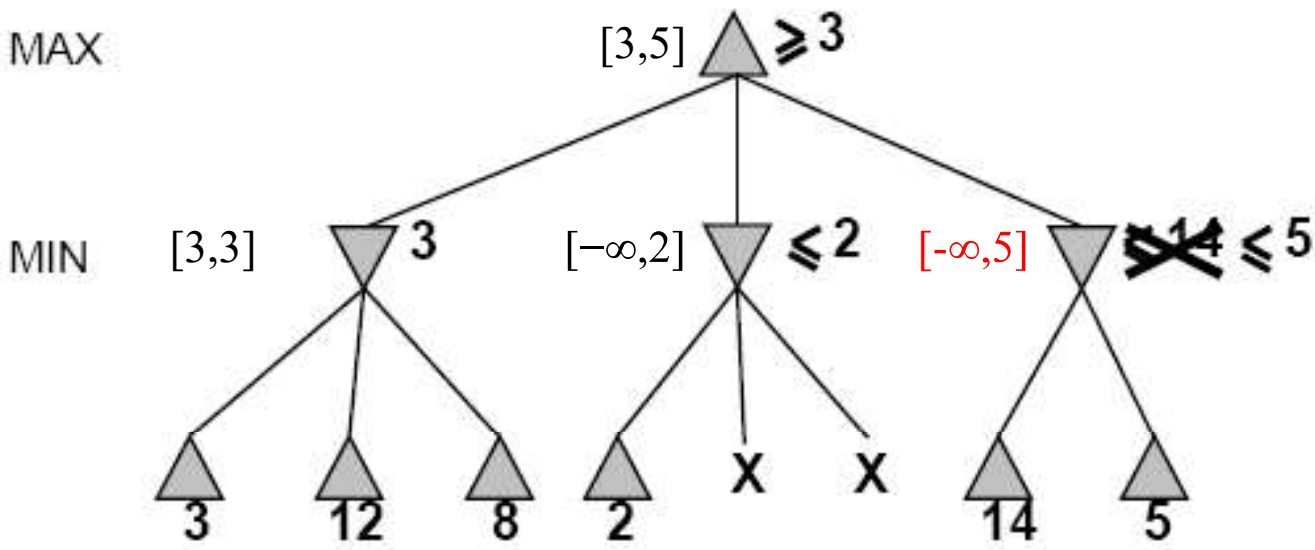
Alpha-Beta Example (continued)



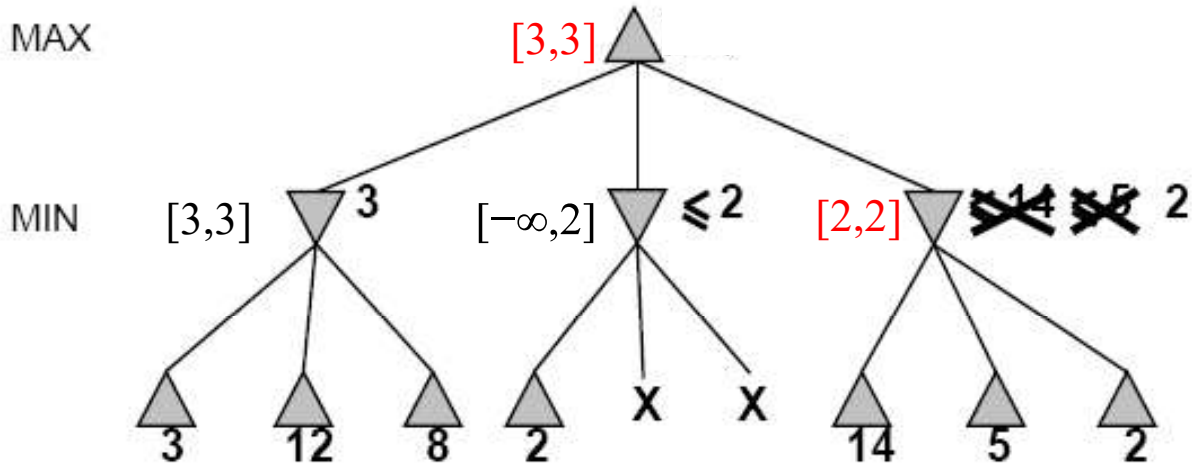
Alpha-Beta Example (continued)



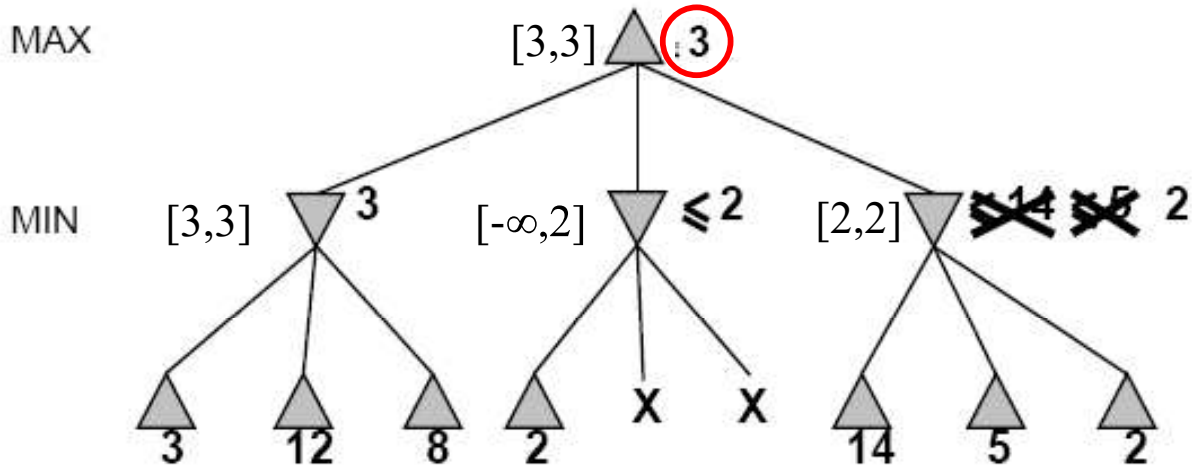
Alpha-Beta Example (continued)



Alpha-Beta Example (continued)



Alpha-Beta Example (continued)



Tic-Tac-Toe Example with Alpha-Beta Pruning

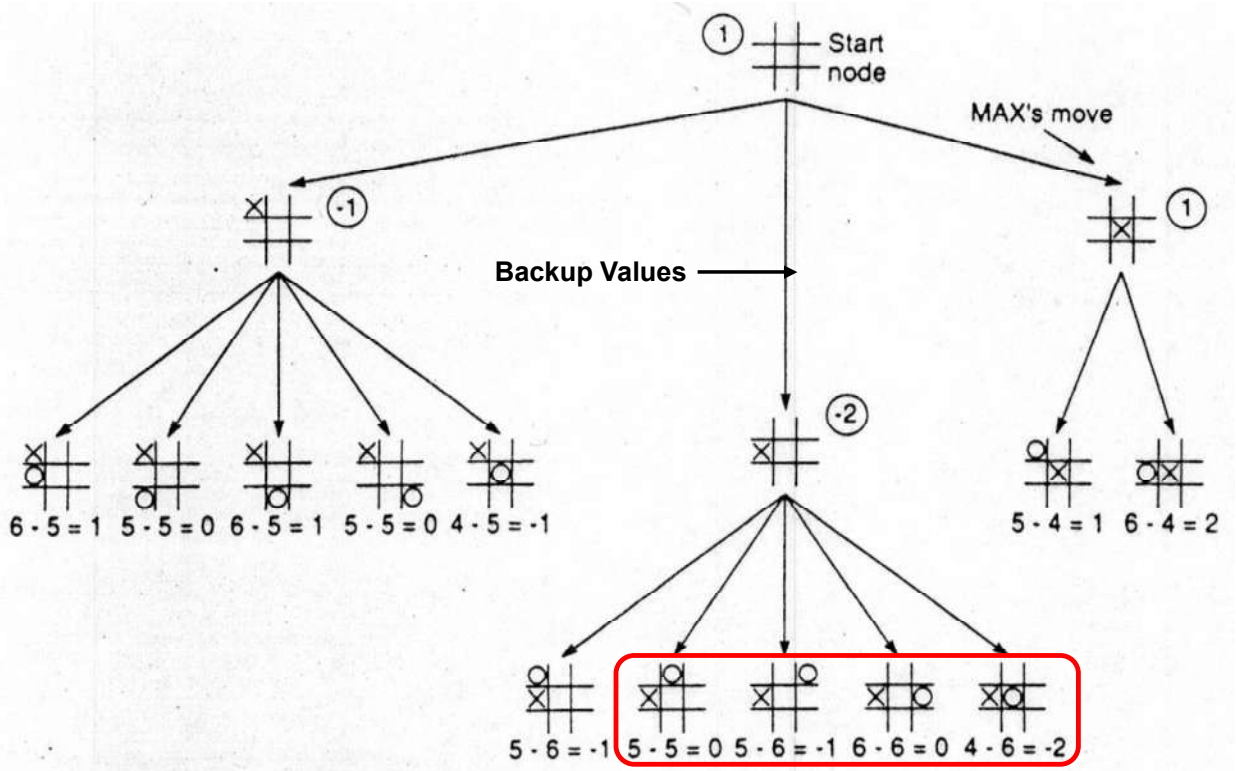


Figure 4.17 Two-ply minimax applied to the opening move of tic-tac-toe.

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)

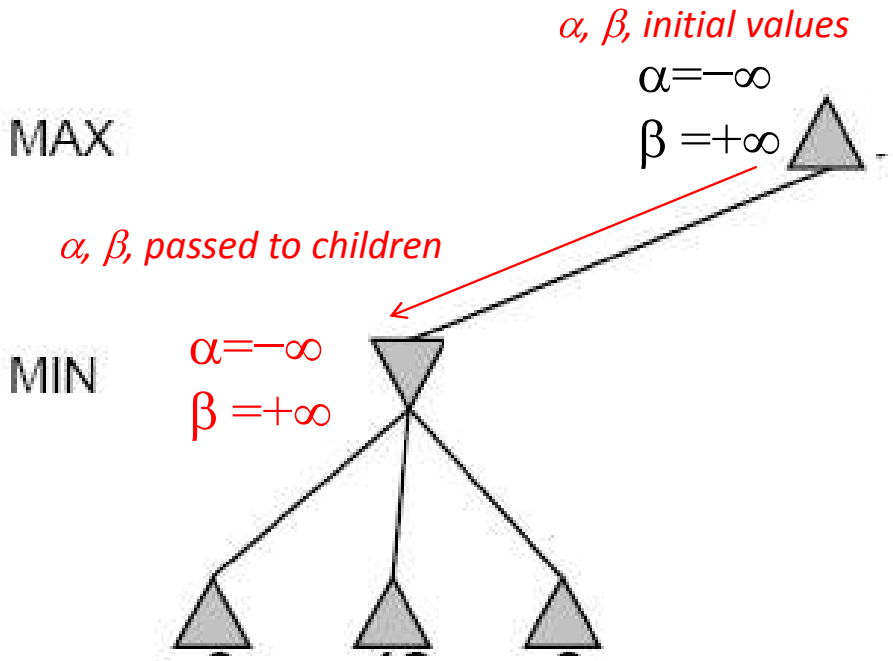
- **Pass current values of α and β down to child nodes during search.**
- **Update values of α and β during search:**
  - MAX updates α at MAX nodes
  - MIN updates β at MIN nodes

When to Prune

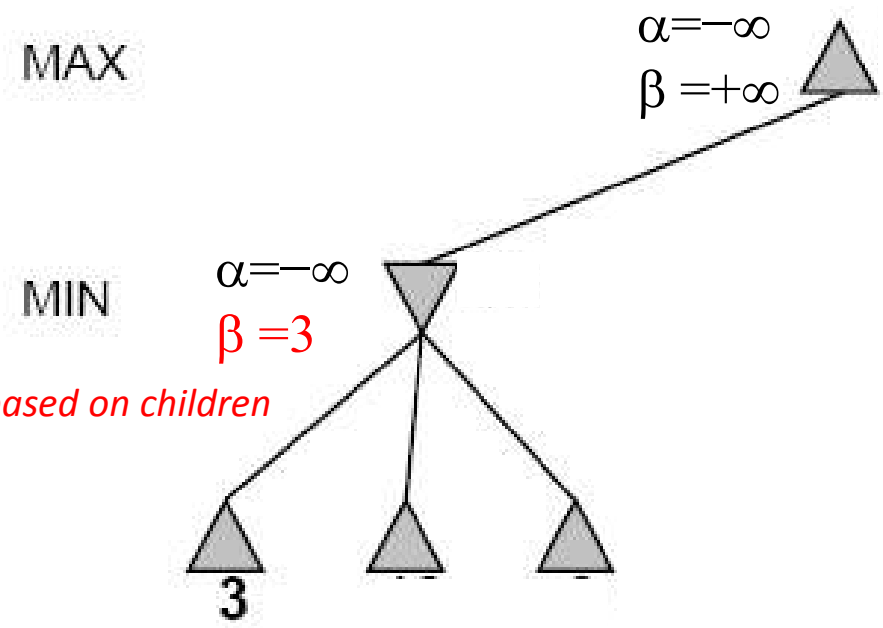
- **Prune whenever $\alpha \geq \beta$.**
  - Prune below a Max node whose alpha value becomes greater than or equal to the beta value of its ancestors.
    - **Max nodes update alpha** based on children's returned values.
  - Prune below a Min node whose beta value becomes less than or equal to the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

Alpha-Beta Example Revisited

Do DF-search until first leaf

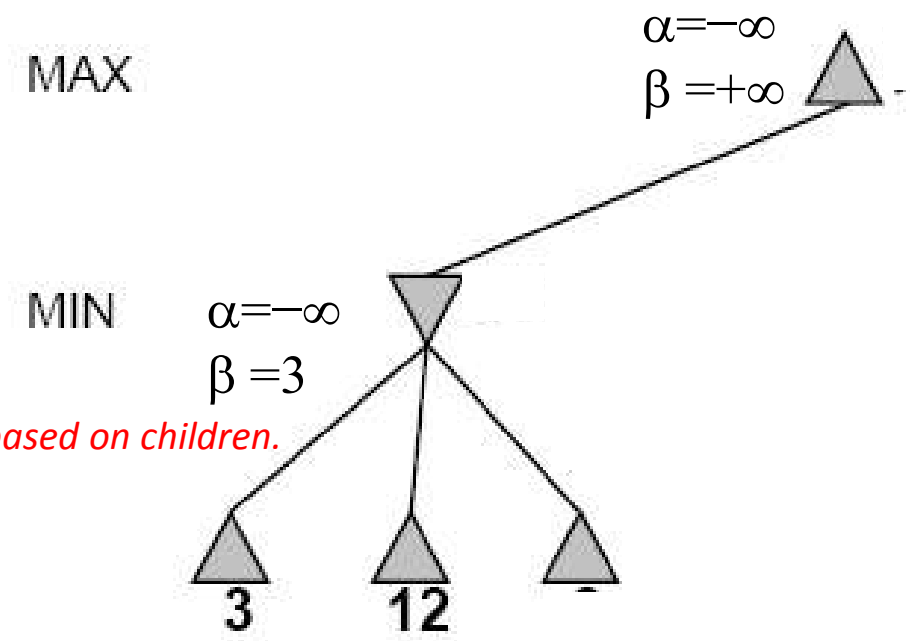


Alpha-Beta Example (continued)



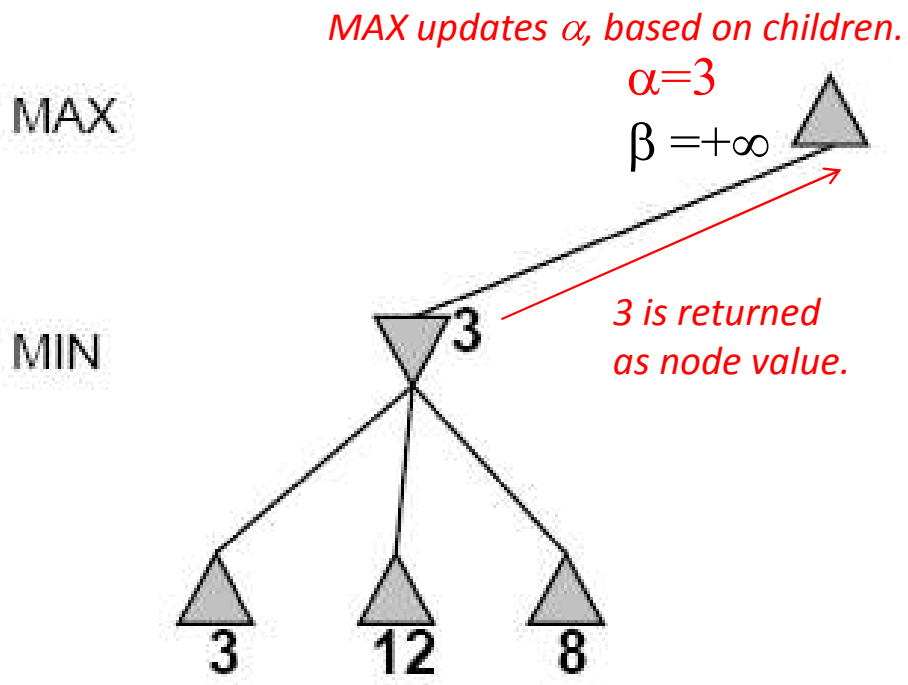
MIN updates β, based on children

Alpha-Beta Example (continued)

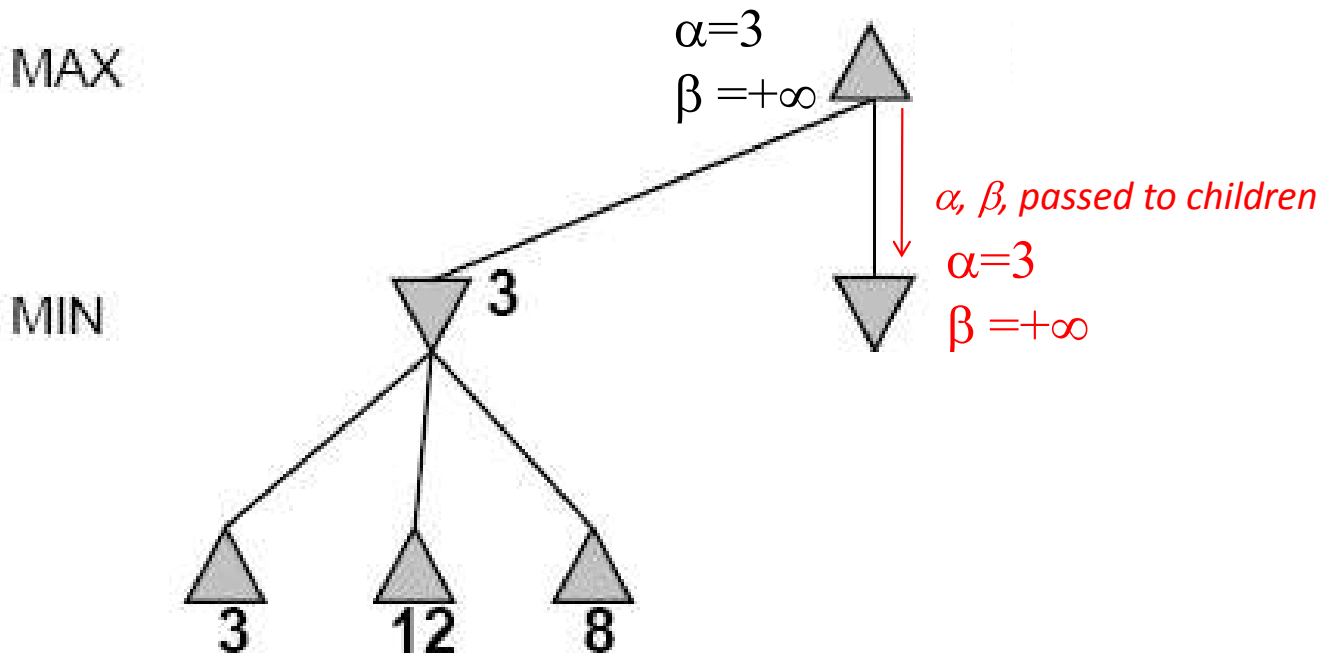


MIN updates β, based on children.
No change.

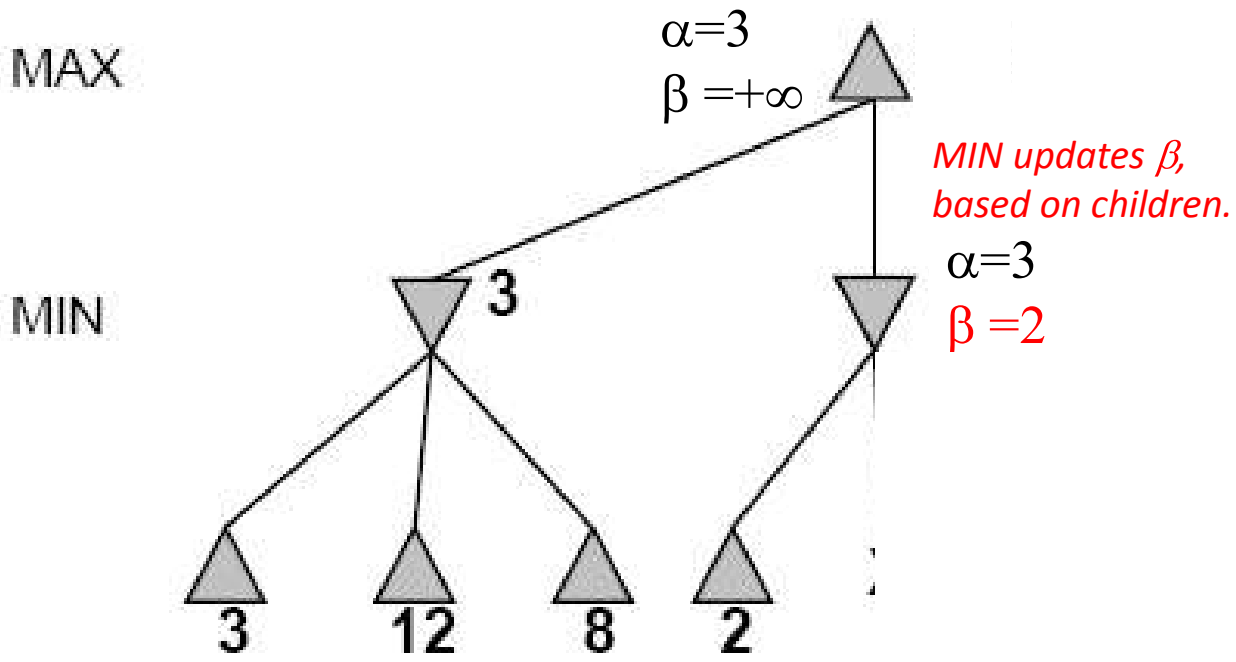
Alpha-Beta Example (continued)



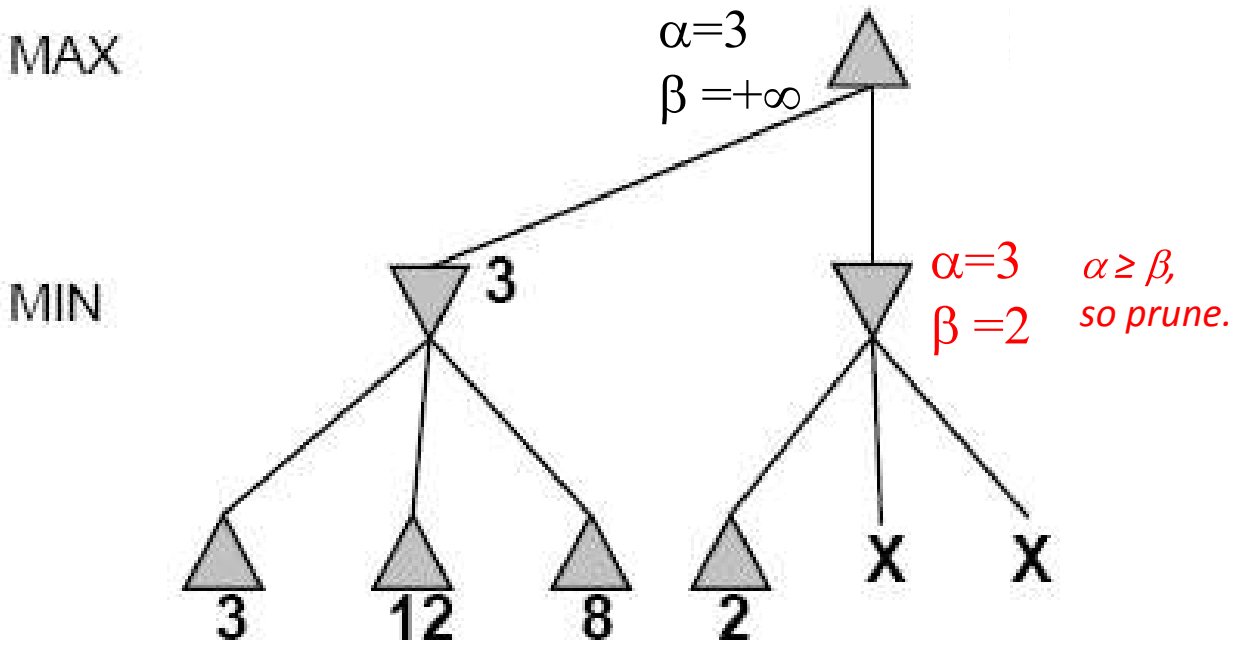
Alpha-Beta Example (continued)



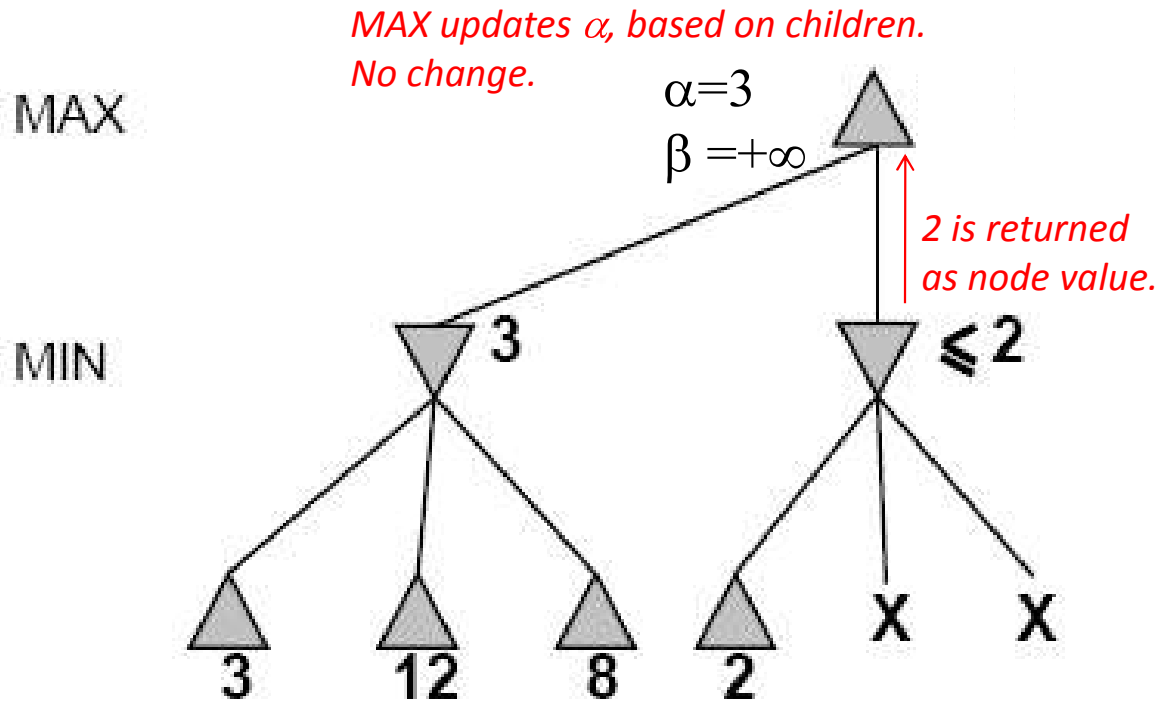
Alpha-Beta Example (continued)



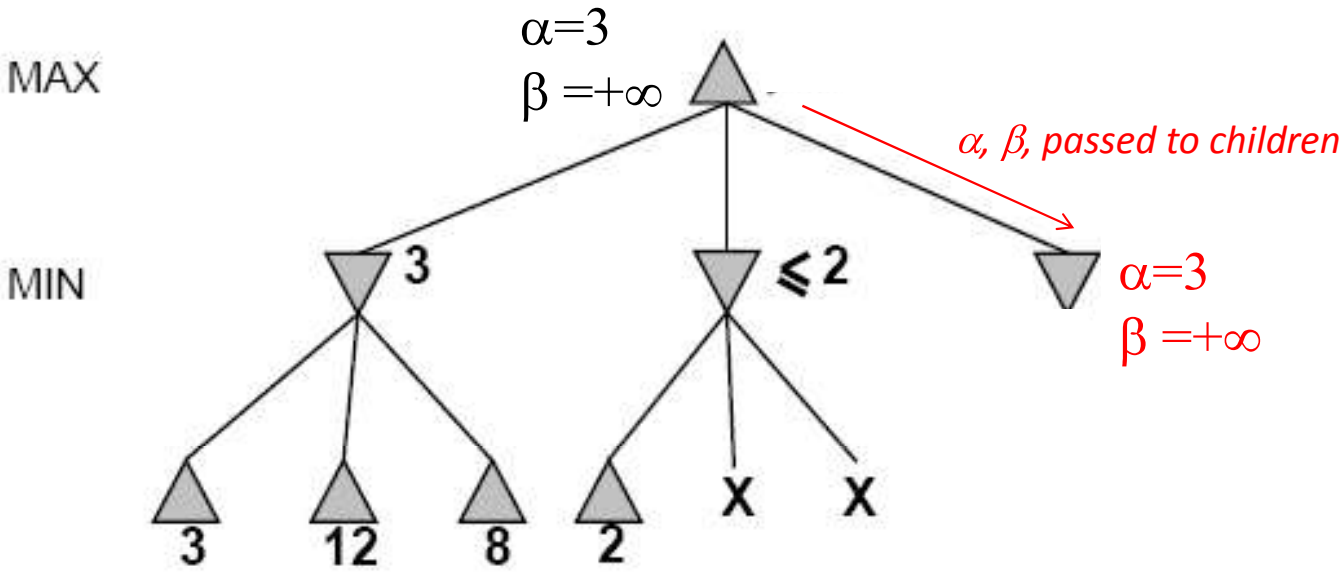
Alpha-Beta Example (continued)



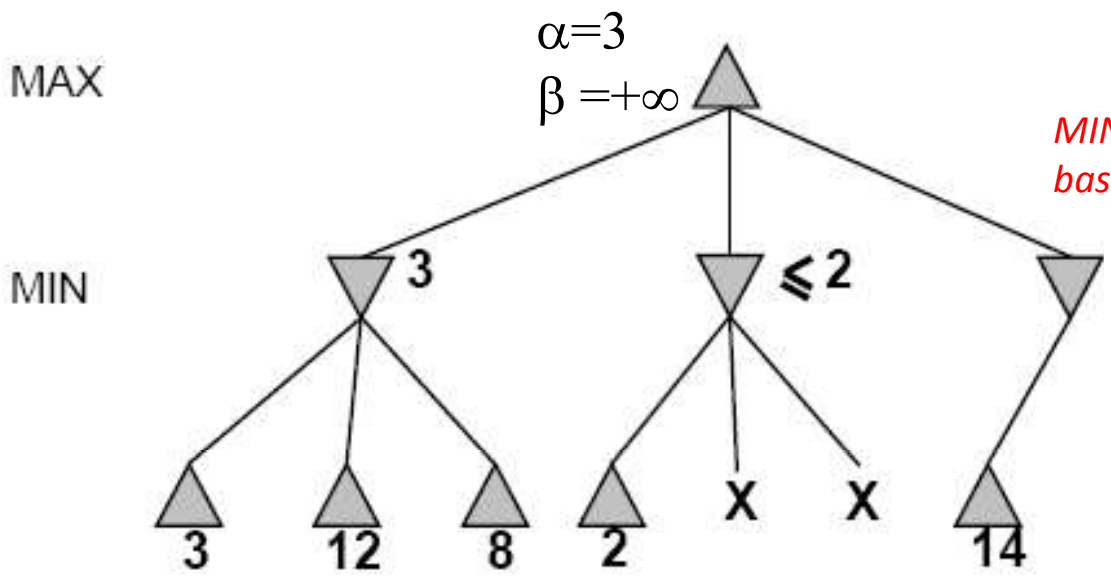
Alpha-Beta Example (continued)



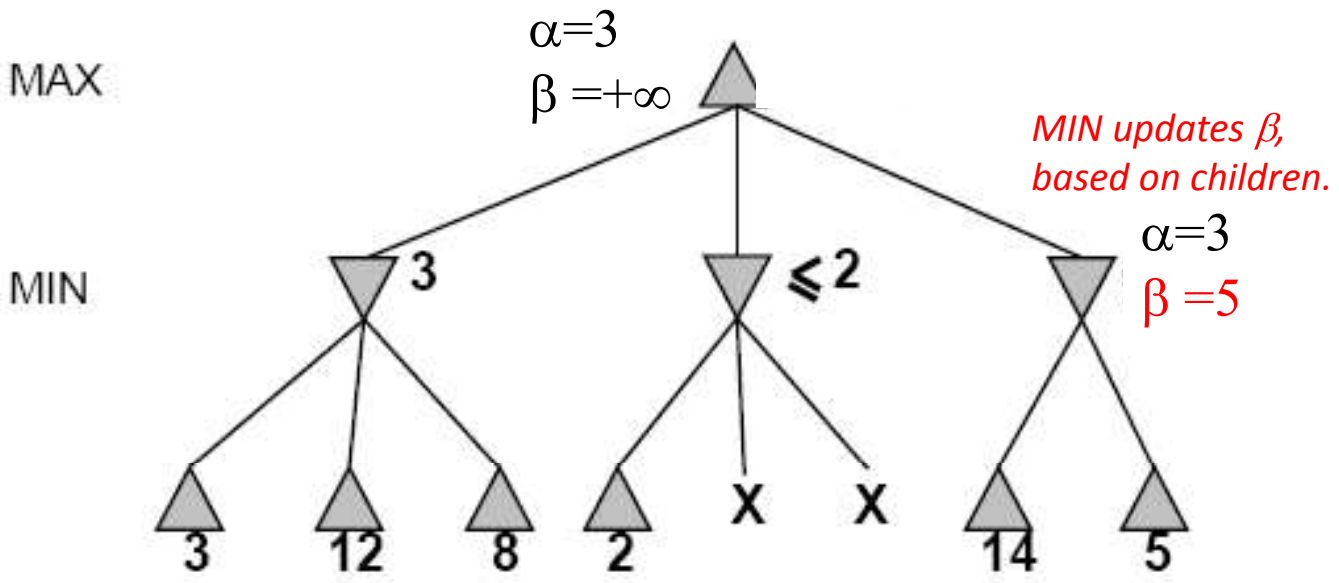
Alpha-Beta Example (continued)



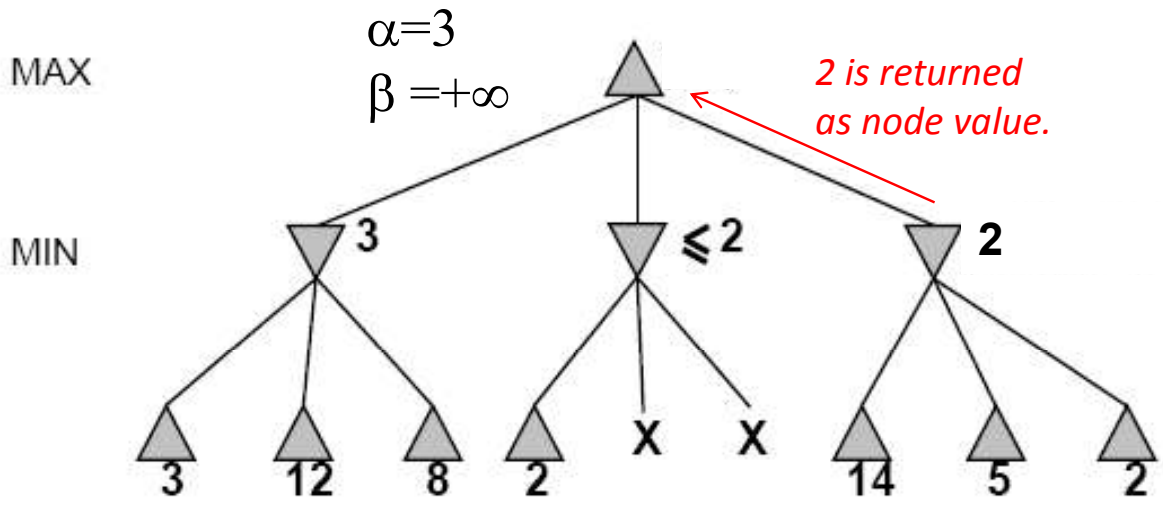
Alpha-Beta Example (continued)



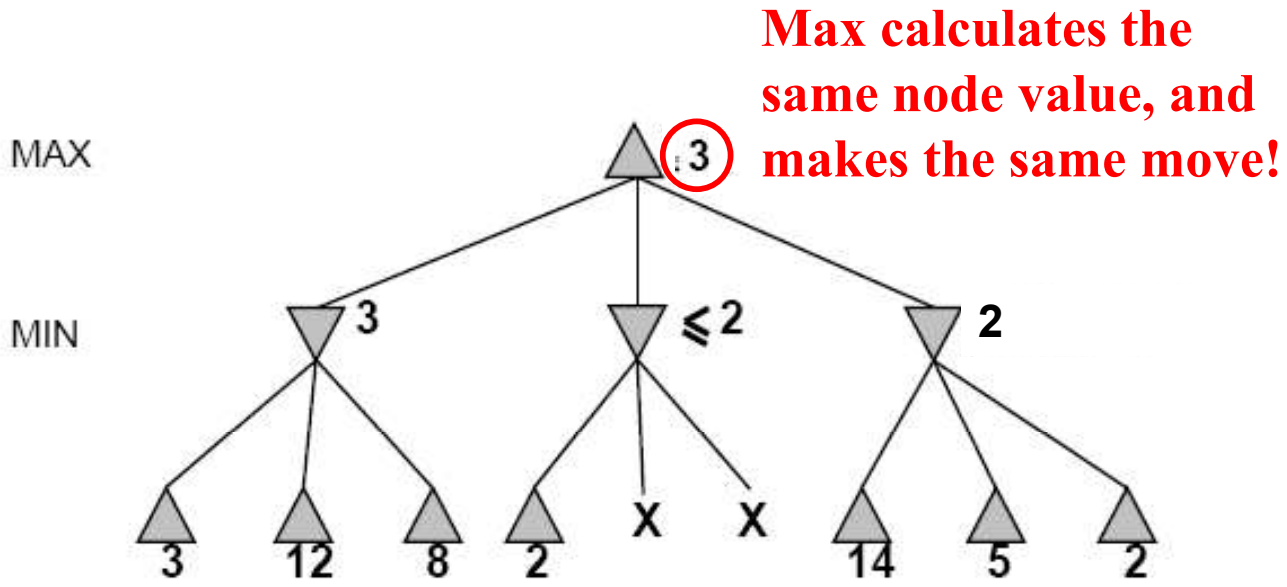
Alpha-Beta Example (continued)



Alpha-Beta Example (continued)



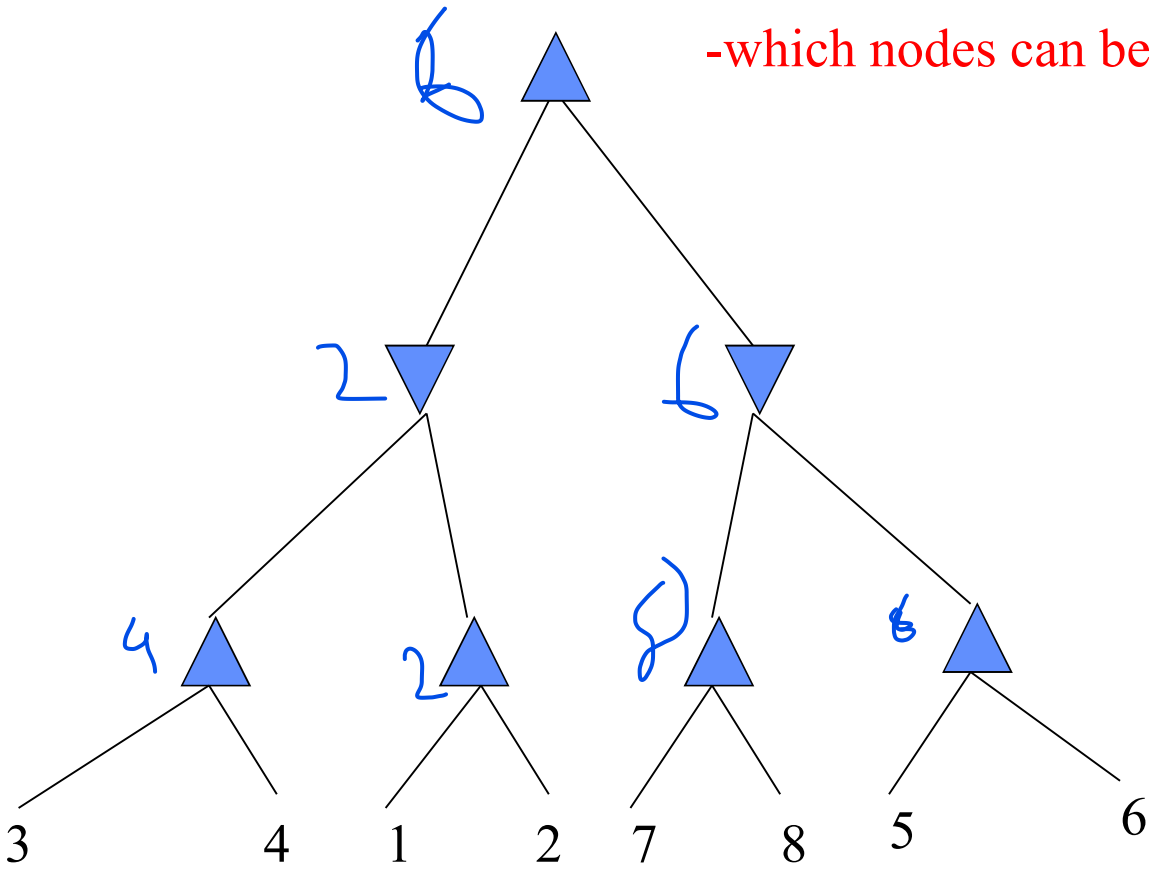
Alpha-Beta Example (continued)



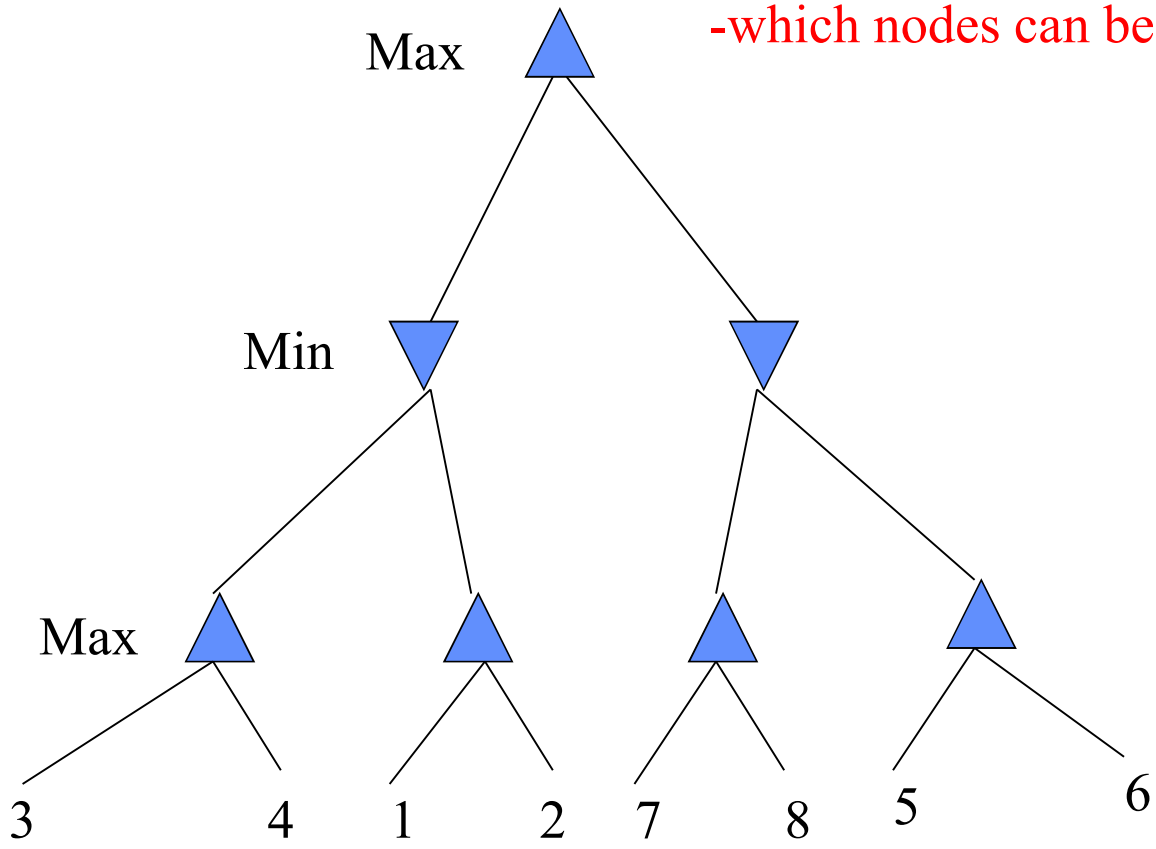
Alpha Beta Practical Implementation

- **Idea:**
  - Do depth first search to generate partial game tree
  - Cutoff test :
    - Depth limit
    - Iterative deepening
    - Cutoff when no big changes (quiescent search)
  - When cutoff, apply static evaluation function to leaves
  - Compute bound on internal nodes
  - Run α-β pruning using estimated values
  - IMPORTANT : use node values of previous iteration to order children during next iteration

Example

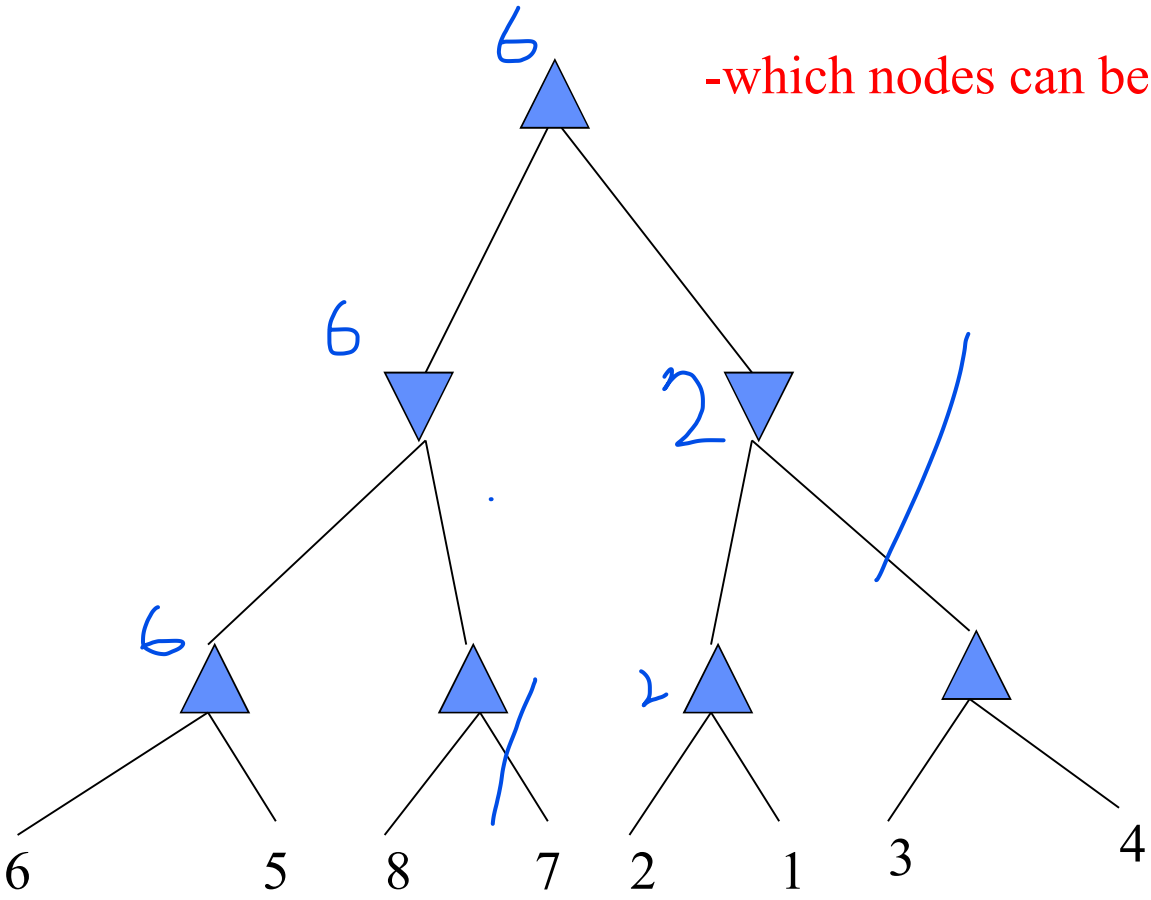


Answer to Example



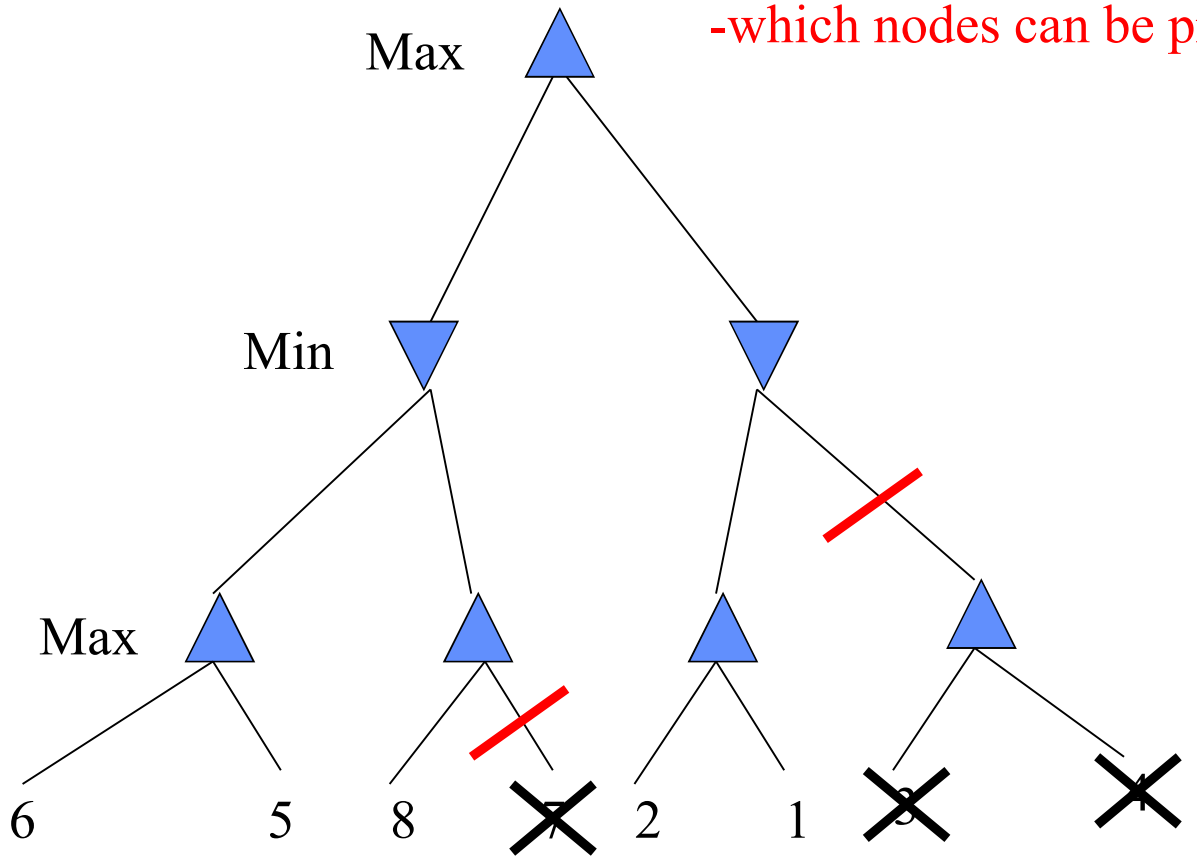
Answer: NONE! Because the most favorable nodes for both are explored last (i.e., in the diagram, are on the right-hand side).

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



-which nodes can be pruned?

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



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

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 alternative (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.

- **Alpha/beta best case is $O(b^{(d/2)})$ rather than $O(b^d)$**
  - This is the same as having a branching factor of sqrt(b),
    - $(\text{sqrt}(b))^d = b^{(d/2)}$ (i.e., we have effectively gone from b to square root of b)
  - In chess go from b ~ 35 to b ~ 6
    - permitting much deeper search in the same amount of time
  - In practice it is often $b^{(2d/3)}$

Final Comments about Alpha-Beta Pruning

- **Pruning does not affect final results!!! Alpha-beta pruning returns the MiniMax value!!!**
- **Entire subtrees can be pruned.**
- **Good move *ordering* improves effectiveness of pruning**
- **Repeated states are again possible.**
  - Store them in memory = transposition table
  - Even in depth-first search we can store the result of an evaluation in a hash table of previously seen positions. Like the notion of "explored" list in graph-search