Games

Non-cooperative multi-agent systems

Games

Many problems can be modelled as games: multiple agents with (possibly competing) interests:

- Chess
- Go
- Multiple agents competing for limited resources

Games and agents acting in them can be described by:

- **Actions** available to each agent in various stages of a game
- **Utility** (payoff) functions, one for each agent. They assign a real number to every possible outcome of the game (typically terminal states). [If the utility functions are the same the agents become cooperative.]
- **Strategy** functions, one for each agent. They determine what action must be taken in each state. [Typically the goal is to find a strategy that maximises the utility for one agent or for a group of agents.]

Example: rock paper scissors

		Bob		
		rock	*paper*	*scissors*
	rock	$0, 0$	$-1, 1$	$1, -1$
Alice	*paper*	$1, -1$	$0, 0$	$-1, 1$
	scissors	$-1, 1$	$1, -1$	$0,0$

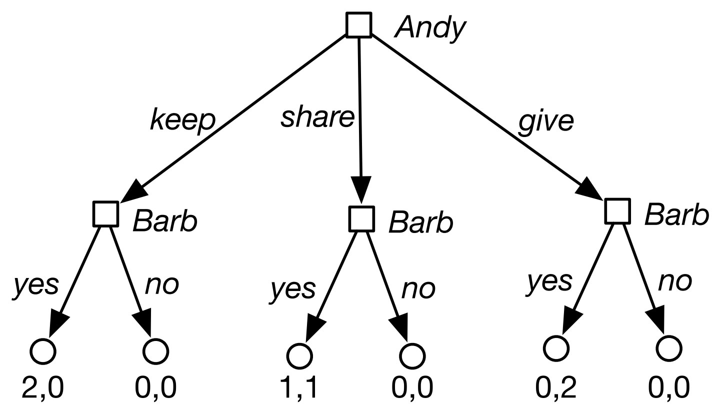
- The game is not turn-based; it is a *simultaneous action game*.
- The games has 9 possible outcomes.
- The table shows two utility functions (Alice's and Bob's).

Game Trees

In *perfect-information games* (where agents can see which states they are in) a **game tree** is a finite tree where the nodes are states and the arcs correspond to actions by the agents. In particular:

- Each internal node is labeled with an agent (or with nature). The agent is said to control the node.
- Each arc out of a node labeled with agent *i* corresponds to an action for agent *i*.
- Each internal node labeled with **nature** has a probability distribution over its children.
- The leaves represent final outcomes and are labeled with a utility foreach agent.

Example: game tree of a sharing game



There are two identical items to be divided between two agents. Andy first selects how they will be divided:

- Andy keeps both items,
- they share and each person gets one item,
- or he gives both items to Barb.

Then Barb gets to either

- reject the allocation and they both get nothing,
- or accept the allocation and they both get the allocated amount.

There are 3 possible strategies for Andy.

There are $2^3 = 8$ possible strategies for Barb.

Utility of agents at nodes

- The utility for each agent at a leaf is given as part of the leaf.
- The utility for agent of a node controlled by that agent is the utility for the agent of the child node that is selected by agent's strategy.
- The utility for agent j of a node controlled by another agent i is the utility for agent j of the child node that is selected by agent i's strategy.
- The utility for agent i for a node controlled by nature is the expected value of the utility for agent i of the children. That is, $u_i(n) = \sum_c P(c)u_i(c)$, where the sum is over the children c of node n, and $P(c)$ is the probability that nature will choose child c.

Example: In the sharing game, suppose we have the following strategy profile: Andy chooses *keep* and Barb chooses *no*, *yes*, *yes* for each of the nodes she gets to choose for.

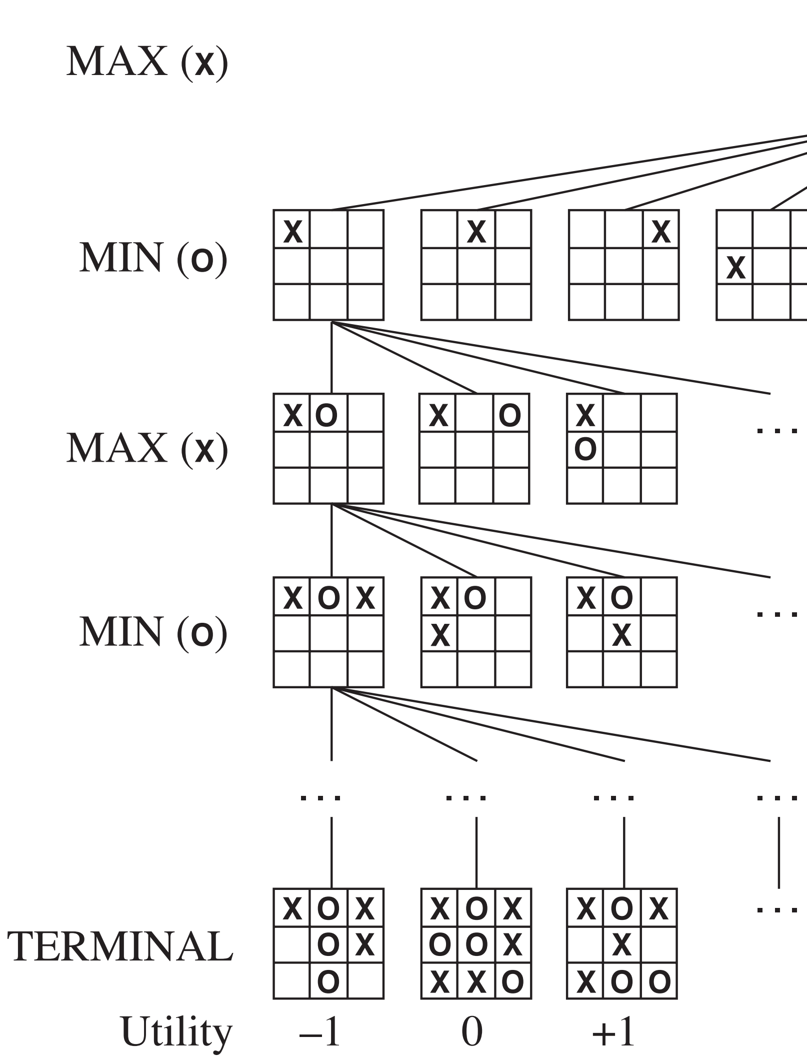
Under this strategy profile, the utility for Andy at the leftmost internal node is 0, the utility for Andy at the centre internal node is 1, and the utility for Andy at the rightmost internal node is 0. The utility for Andy at the root is 0.

Perfect-information zero-sum turn-based games

Properties:

– Typically two agents
– They take turn to play
– There is no chance involved.
– The state of the game is fully observable by all agents
– Utility (payoff) values for each agent are the opposite of those of the other. Also called **zero sum**; the sum of the reward and penalty is constant.
– Because they are opposite of each other, we use one value and assume that one player tries to <u>maximise</u> the utility, the other <u>minimise</u>.
– This creates adversary.

Example: noughts and crosses



- A game tree can be seen as a contingency tree containing all possibilities
- The root is the current state
- An actual play is a path from the root to one of the leaf nodes

An optimal strategy: Min-Max function

Designed to find the best move at each stage:

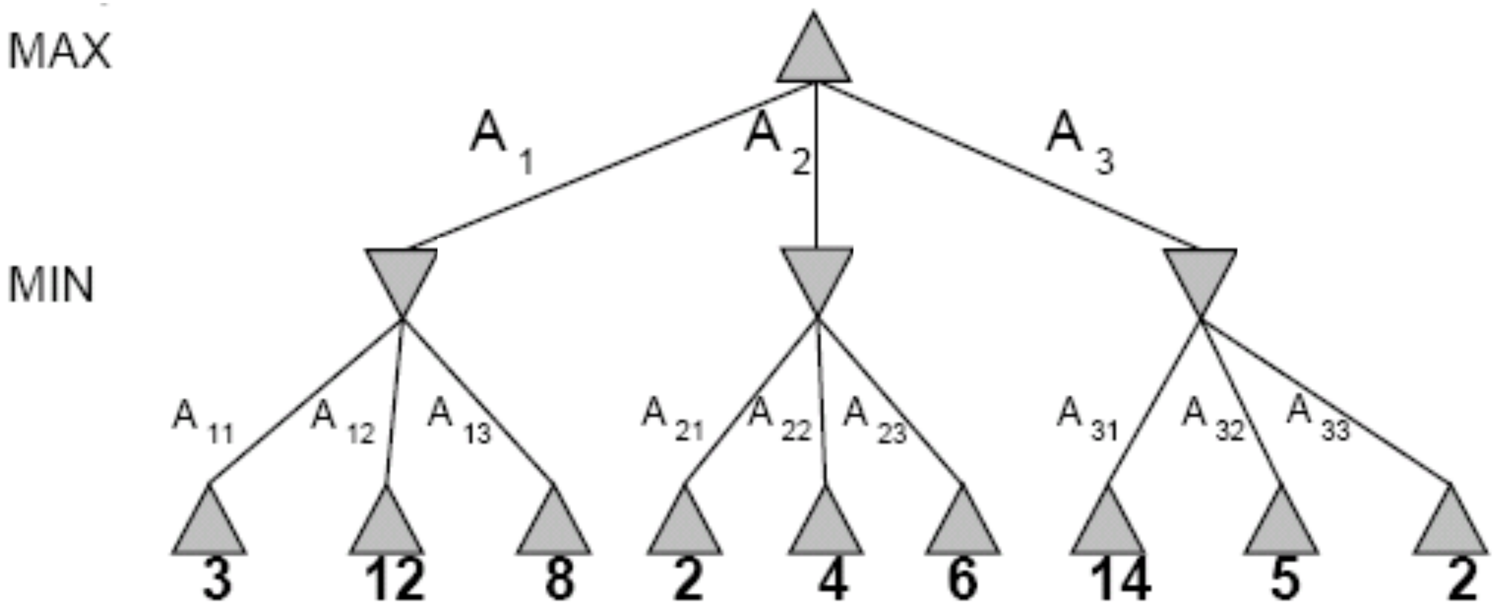
1. Generate the whole game tree, down to the leaves.

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:
   – If it's a **max node**, choose the move leading to the child with highest value.
   – If it's a **min node**, choose the move leading to the child with lowest value.

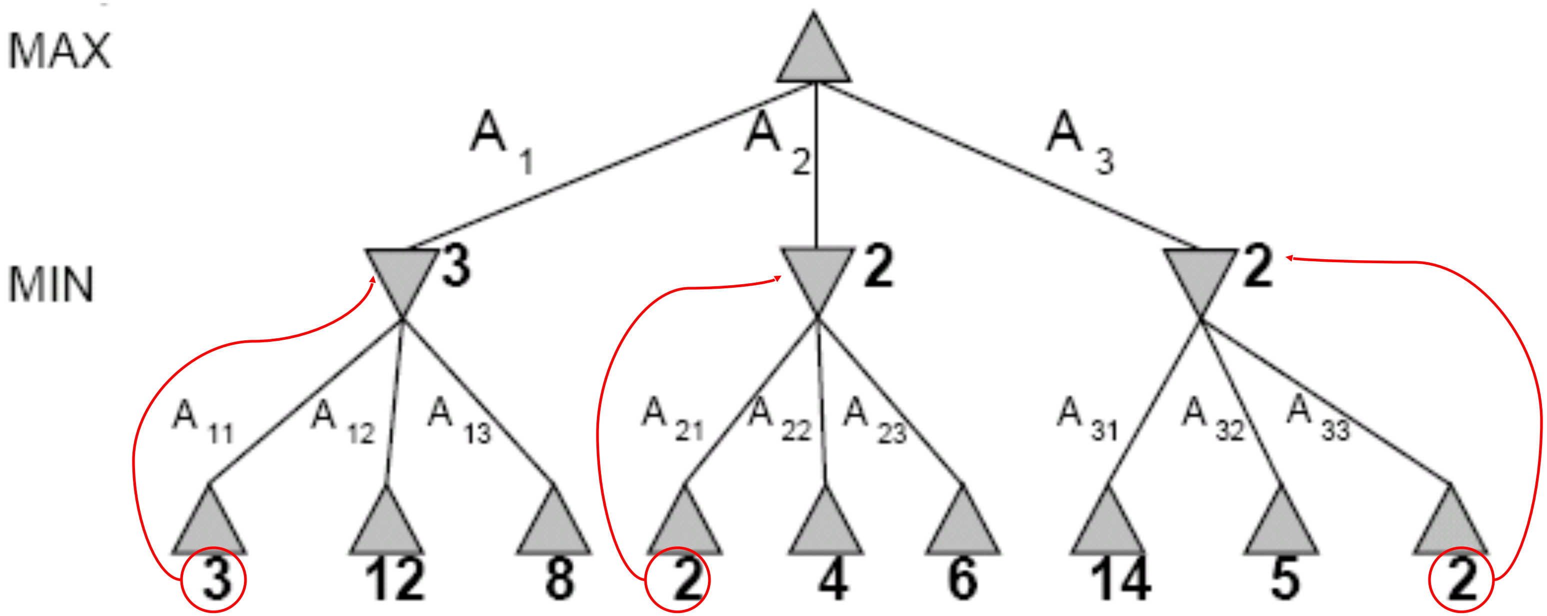
Example: a two-ply game tree



- Two symbols (triangles):
    - Max nodes pointing up
    - Min nodes pointing down

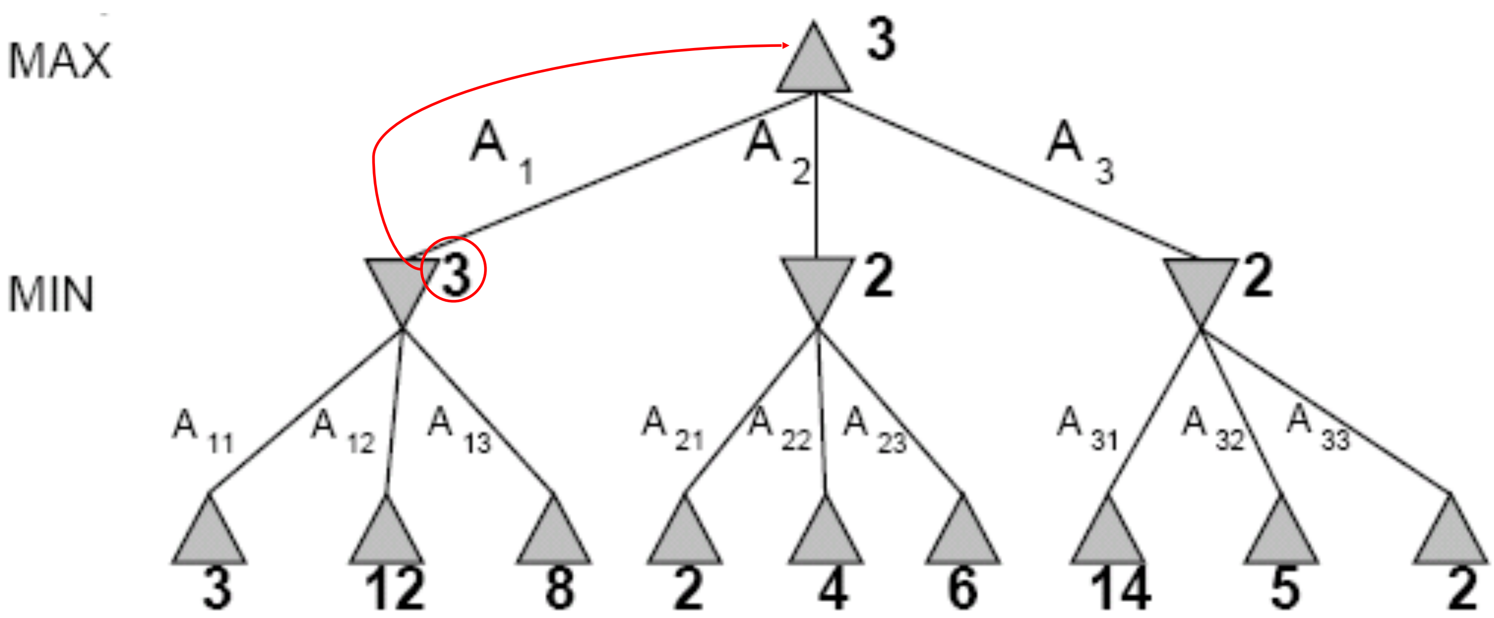
- Leaf (terminal) nodes have a utility (payoff) value.

Example: A Two-Ply Game Tree



- The utility of other nodes are determined by applying MinMax algorithm.

Example: A Two-Ply Game Tree



- The best action for Max is A1
- The best response from Min is A11.

MinMax Algorithm

Assuming the root node is Max

```
function MINIMAX-DECISION(state) returns an action
    inputs: state, current state in game

    return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))
```

```
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
```

Functions used in the algorithm

Actions(s)

Returns the set of legal moves in the given state *s*.

Result(a, s)

Returns a state that is the result of applying action *a* on state *s*.

Successors(s)

Returns a list of pairs of actions/states that can be reached from state *s*.

Terminal-Test(state)

Is the game finished? True if finished, false otherwise.

Utility(s)

Gives numerical value of the terminal state *s*.

Game Tree Size

- Tic-Tac-Toe
  - $b \approx 5$ legal actions per state on average
  - total of 9 moves, $d = 9$
  - $b^d \approx 5^9 = 1{,}953{,}125$

  → **Searching the entire tree quite reasonable**

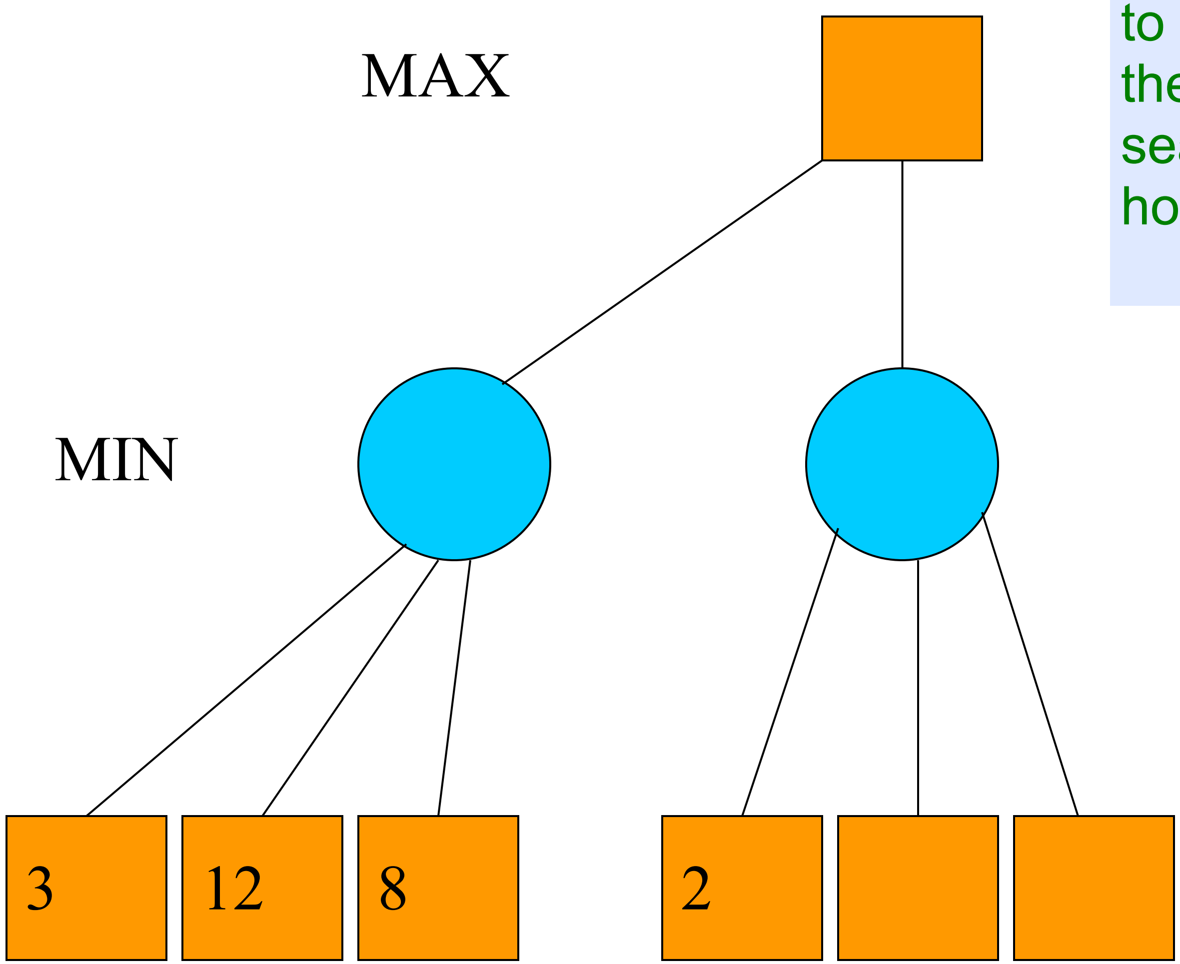
- Chess
  - $b \approx 35$ (approximate average branching factor)
  - $d \approx 100$ (depth of game tree for a “typical” game)
  - $b^d \approx 35^{100} \approx 10^{154}$ nodes!!

  → **Searching the entire tree completely infeasible**

How to reduce the search space

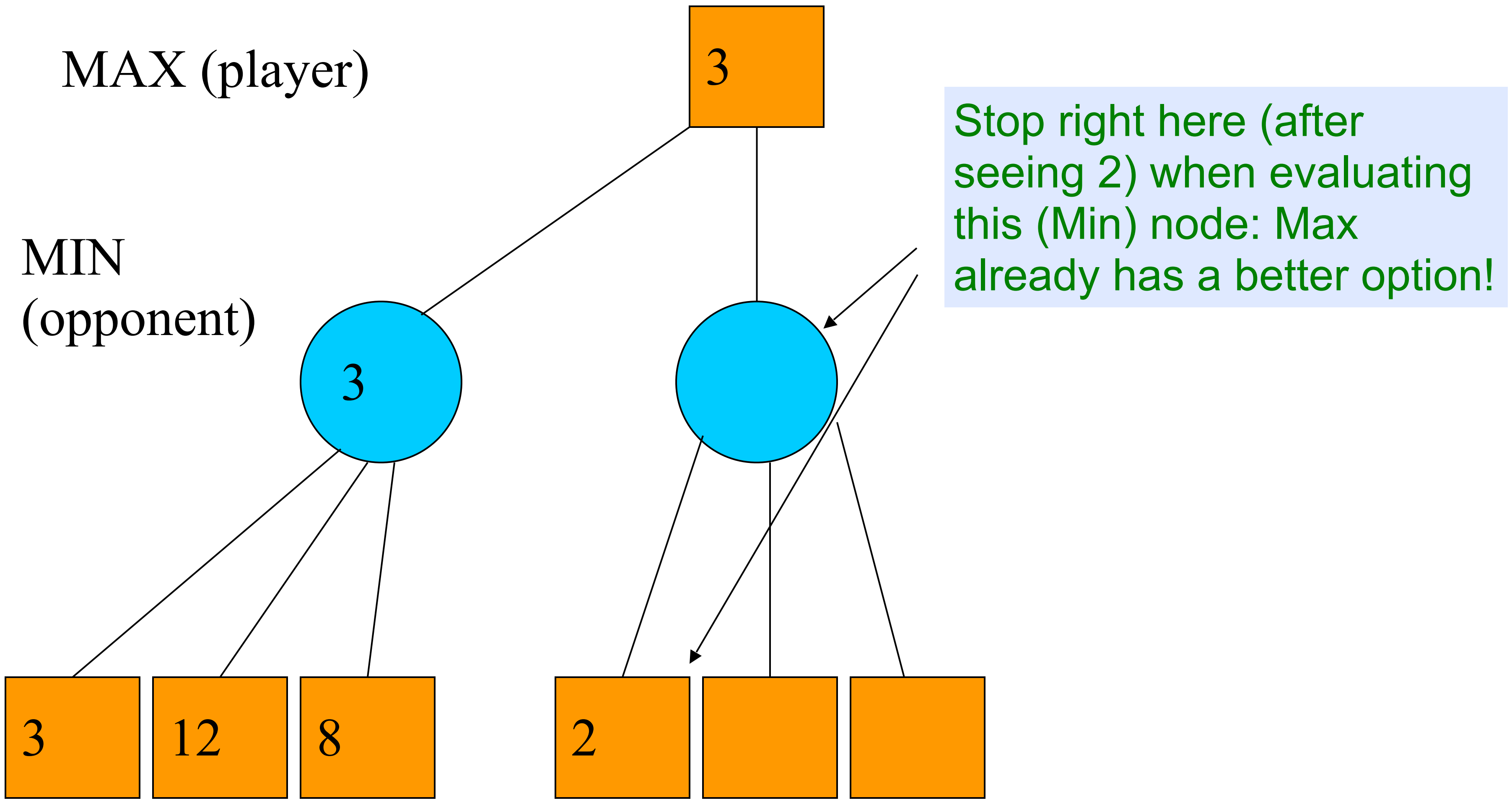
- Pruning the game tree
- Heuristic evaluations of states
- Table lookup instead of search (for opening and closing situations for example)

Pruning: The Idea

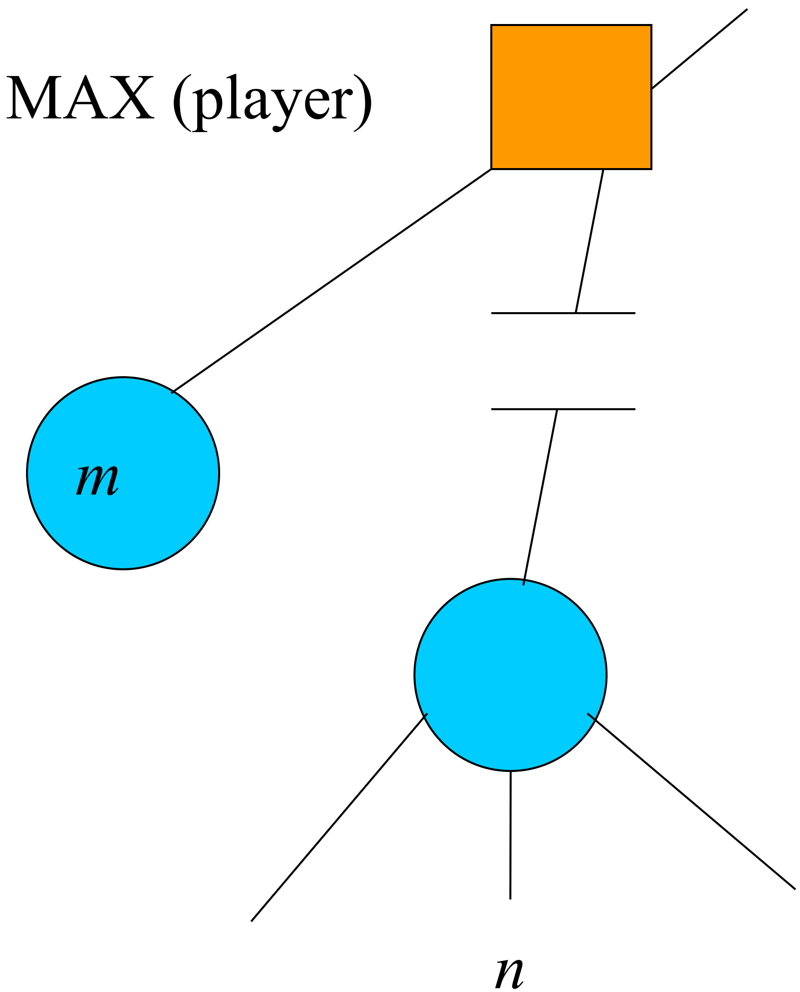


If an option is bad compared to other available options, there is no point in expending search time to find out exactly how bad.

Pruning: The Idea



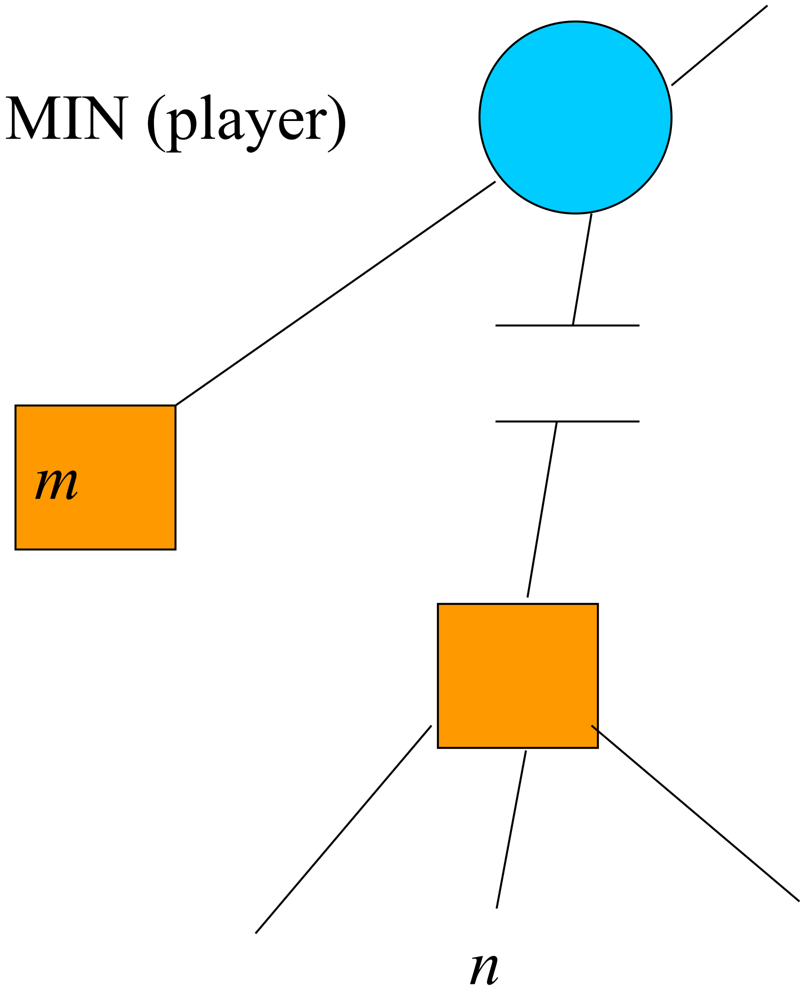
Alpha-Beta Pruning: The Concept



If *m* > *n*, **Max** would choose the *m*-node to get a guaranteed utility of at least *m.*

The Min node with utility *n* (or less) would never be reached; stop further evaluation.

Alpha-Beta Pruning: The Concept



If *m* < *n*, **Min** would choose the *m*-node to get a guaranteed utility of at most *m.*

The Max node with utility *n* (or more) would never be reached; stop further evaluation.

Alpha-Beta Pruning Algorithm

Depth first search

- only considers nodes along a single path from root at any time

α = highest-value choice found for MAX higher up in the tree

(initially, α = −infinity)

β = lowest-value choice found for MIN higher up in the tree

(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 $\alpha \geq \beta$

Alpha-Beta Pruning Algorithm

function ALPHA-BETA-SEARCH($state$) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the $action$ in ACTIONS($state$) with value v

Assuming the root node is Max

---

function MAX-VALUE($state, \alpha, \beta$) **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), α, β))
    $\alpha \leftarrow$ MAX(α, v)
    if $\alpha \geq \beta$ **then return** v
  return v

---

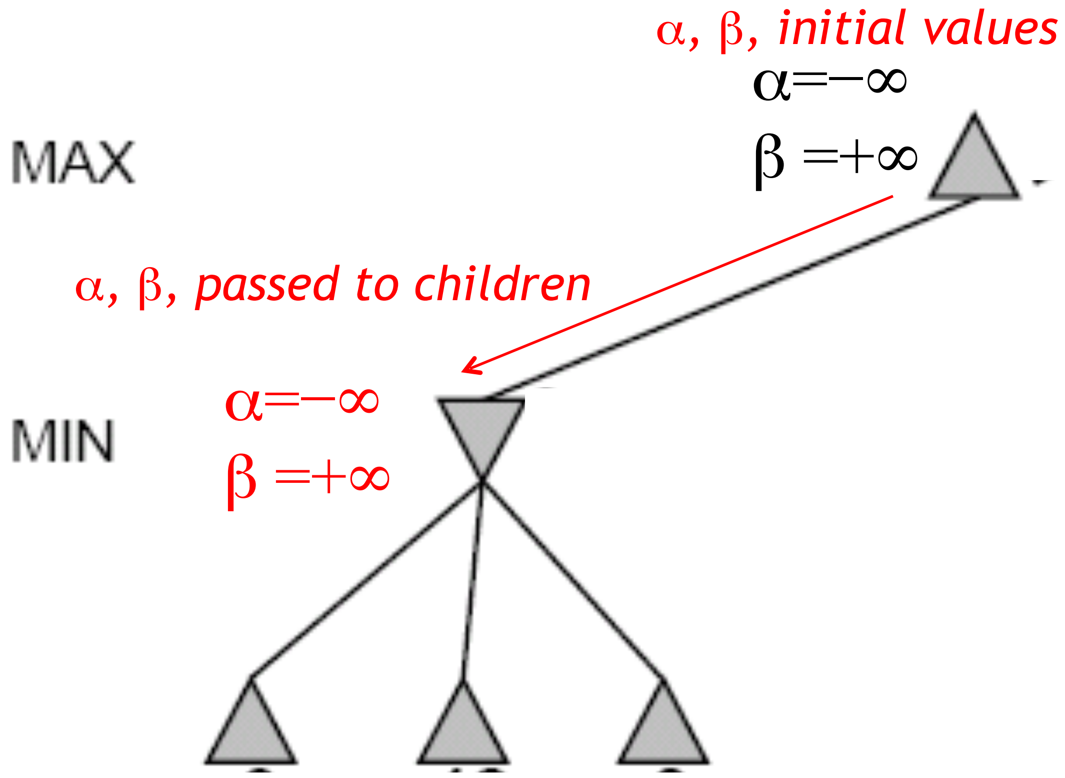
function MIN-VALUE($state, \alpha, \beta$) **returns** a $utility$ $value$
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    $\beta \leftarrow$ MIN(β, v)
    if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

When to Prune

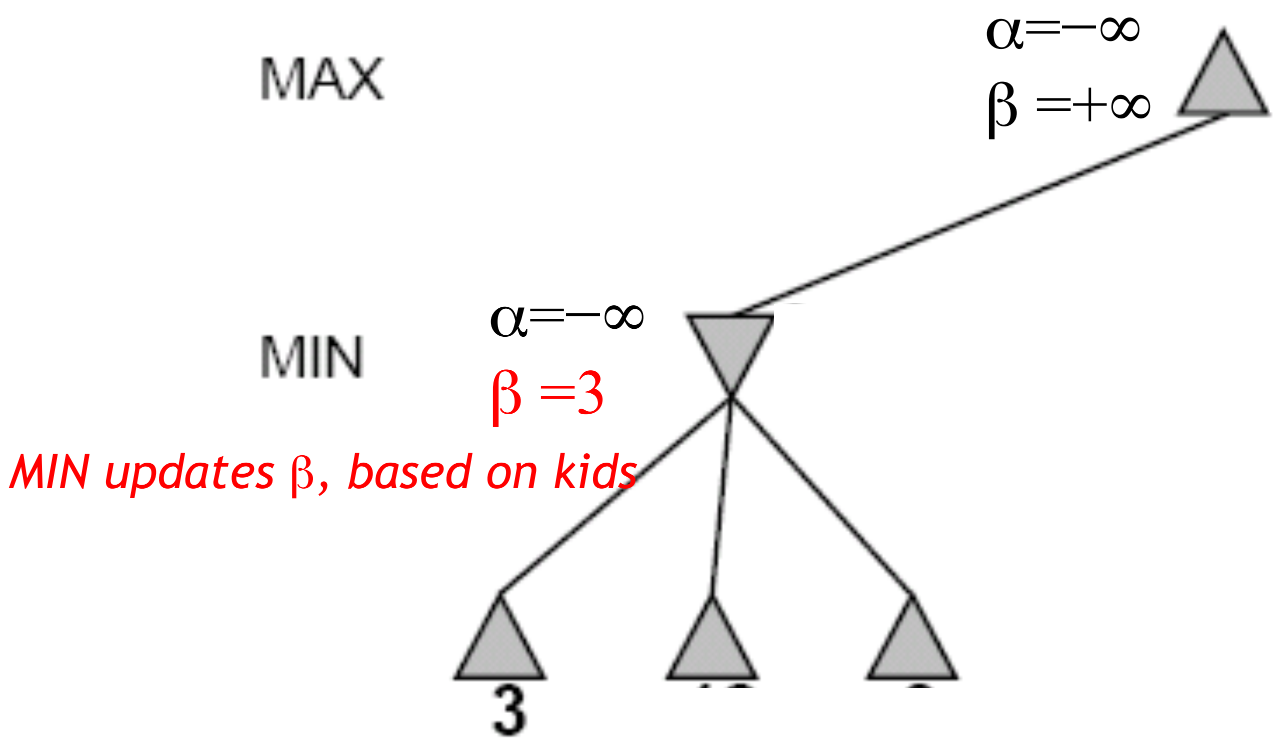
- **Prune whenever $\alpha \geq \beta$**
  - Prune below a Max node whose alpha value becomes greater than or equal to the beta value of its ancestors.
    - **Max nodes update alpha** based on children's returned values.
  - Prune below a Min node whose beta value becomes less than or equal to the alpha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

Alpha-Beta Example

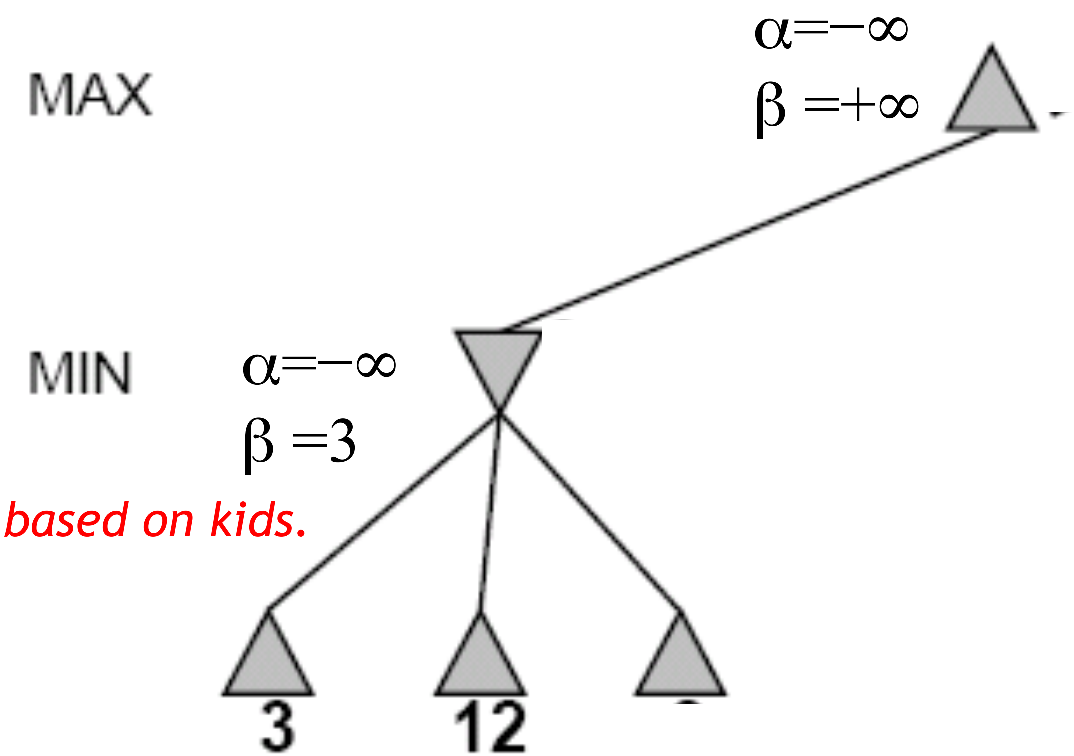
Do DF-search until first leaf



Alpha-Beta Example (continued)

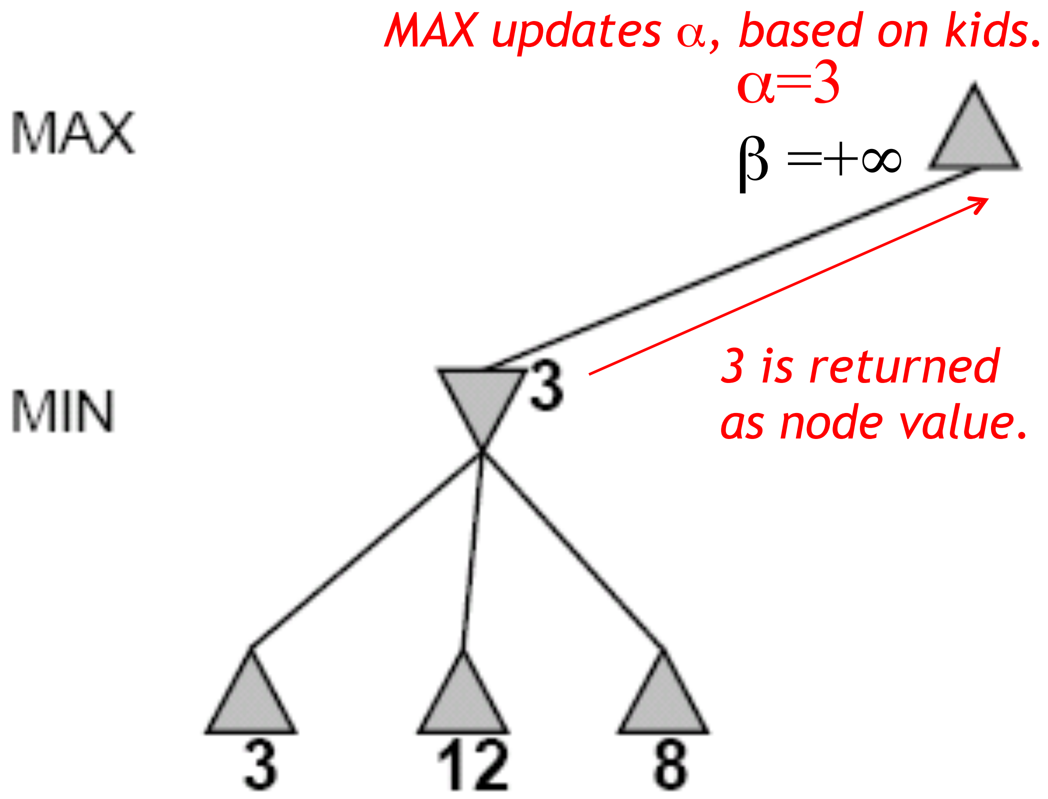


Alpha-Beta Example (continued)

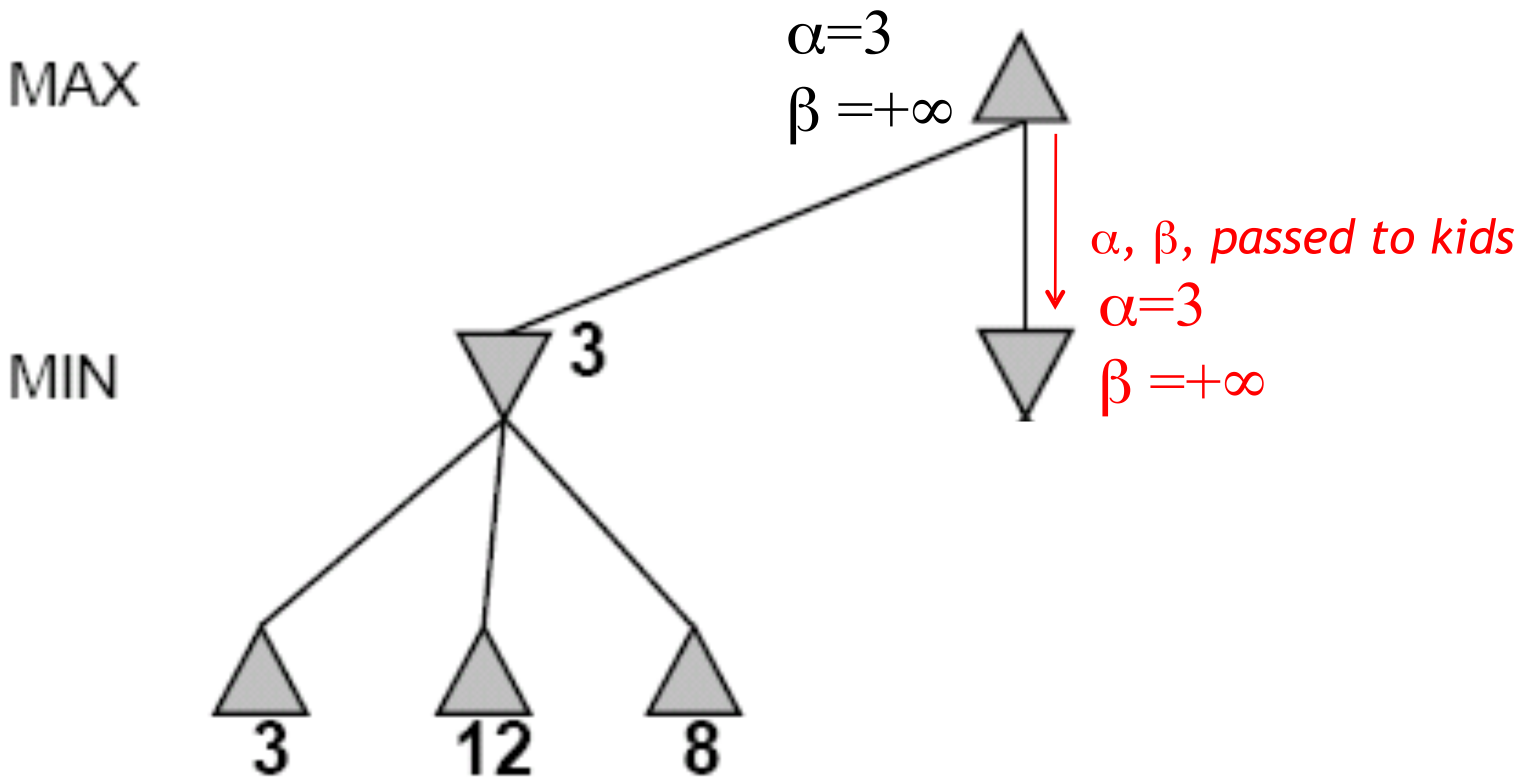


MIN updates β, based on kids.
No change.

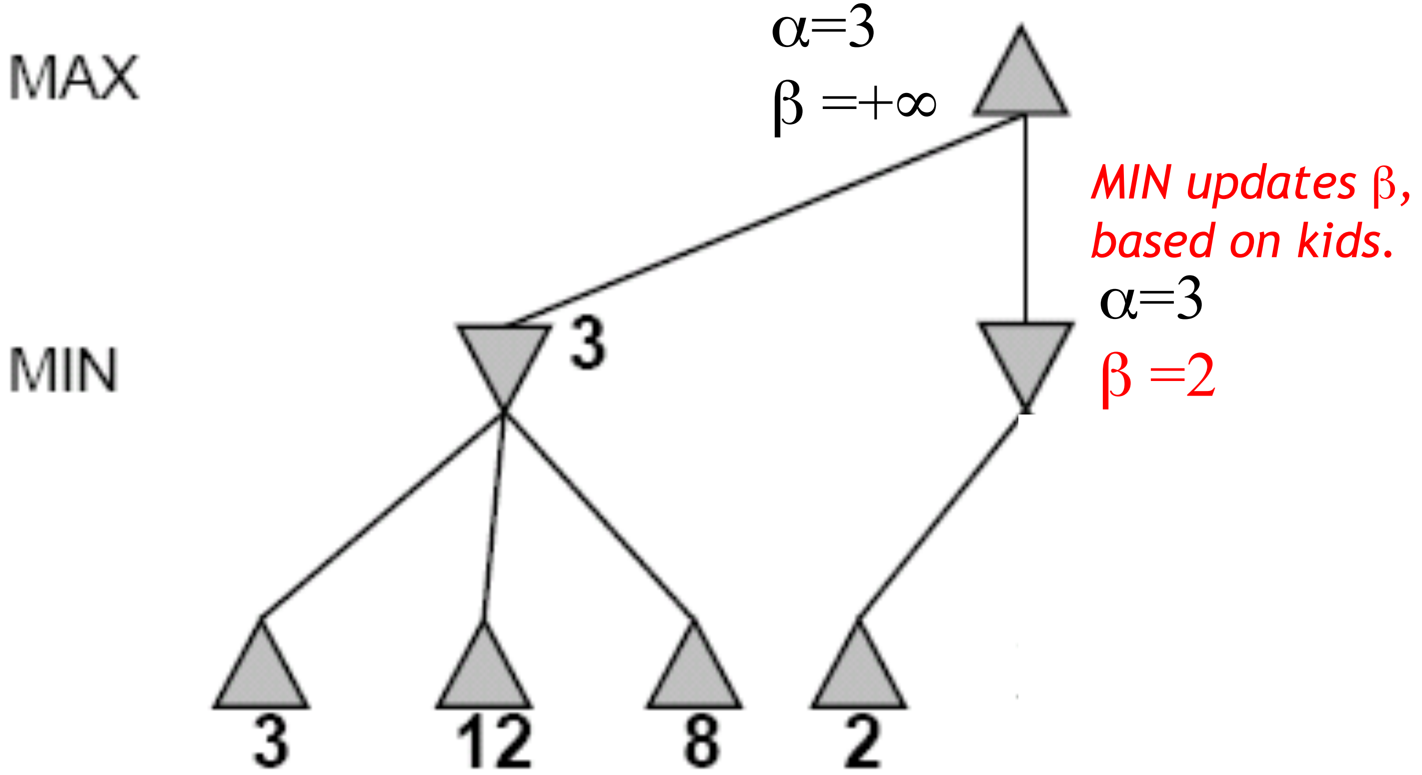
Alpha-Beta Example (continued)



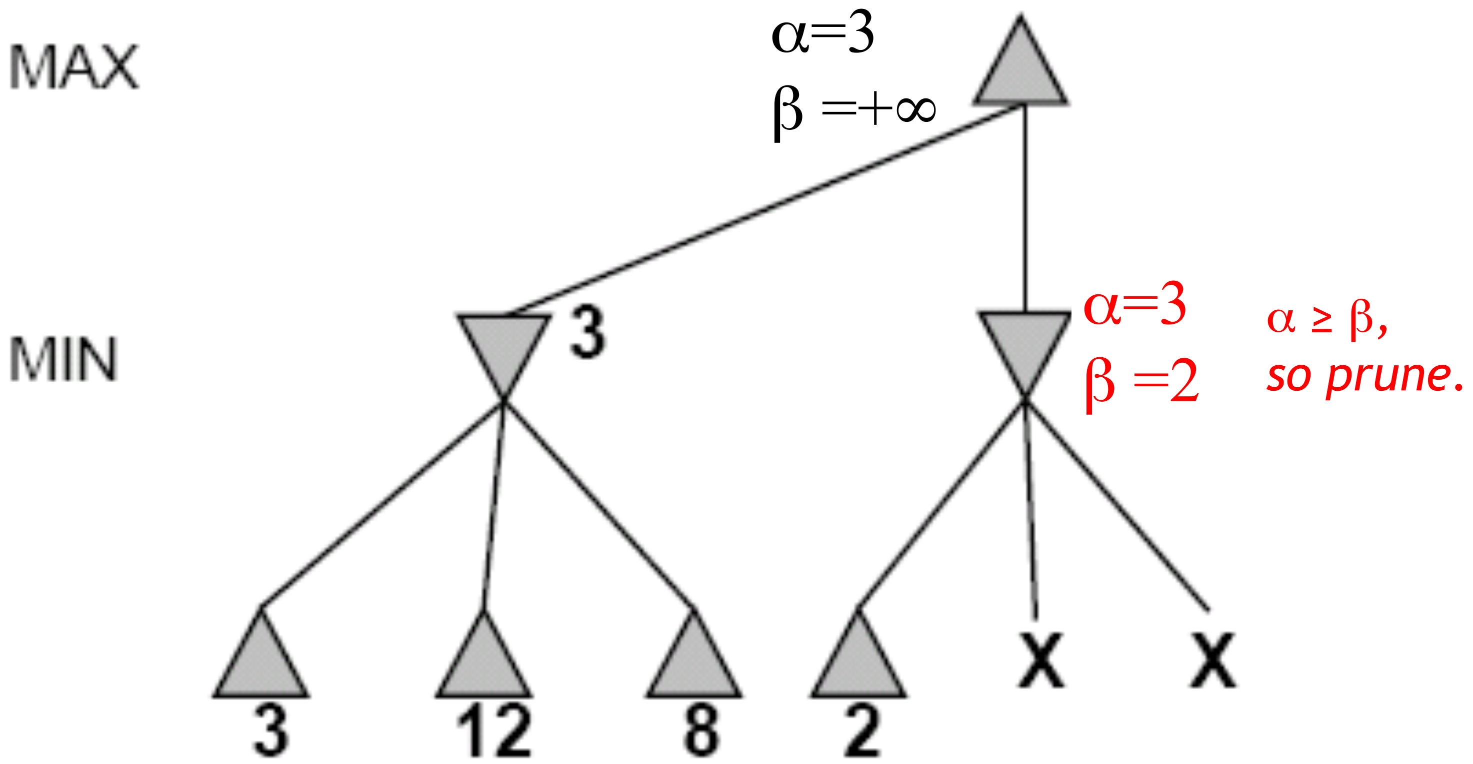
Alpha-Beta Example (continued)



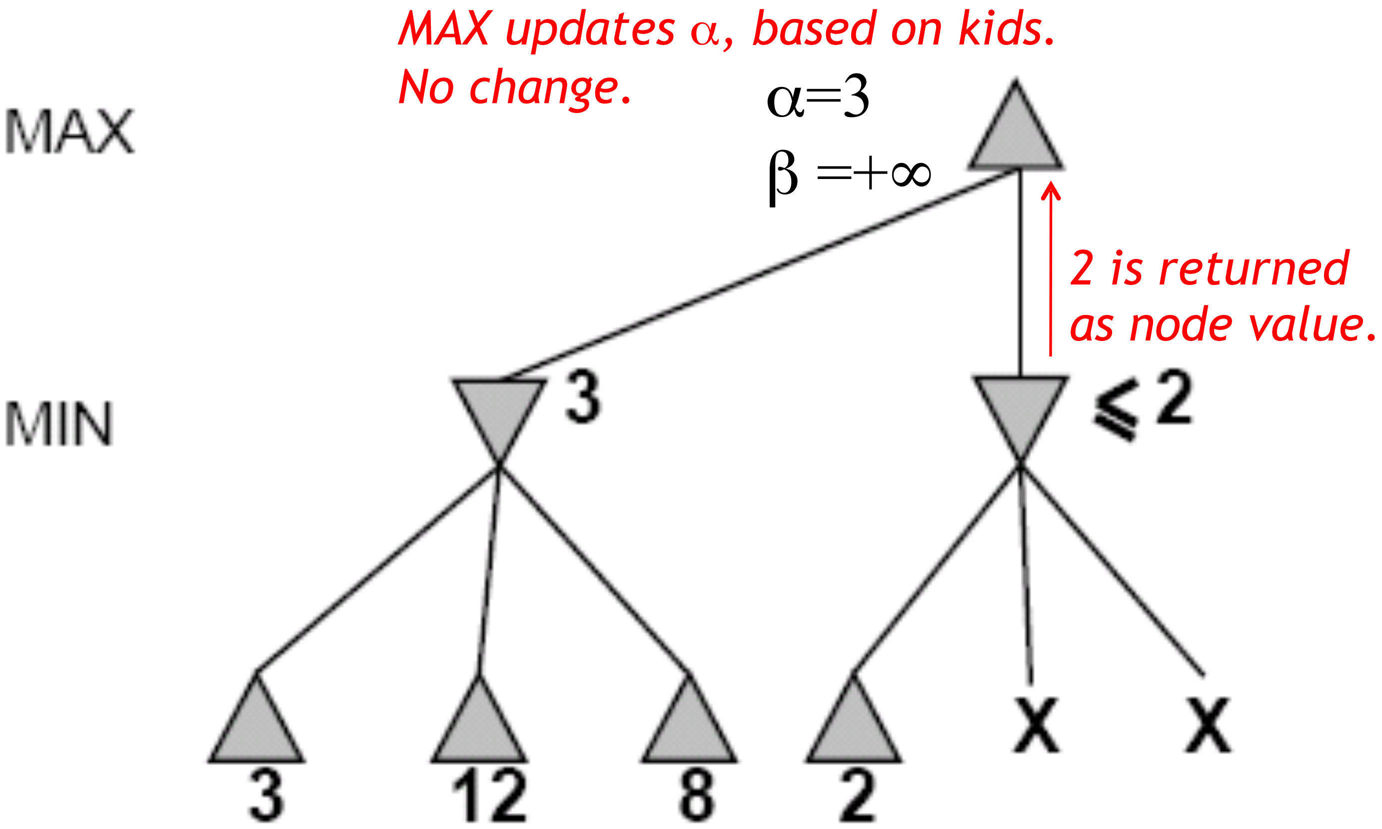
Alpha-Beta Example (continued)



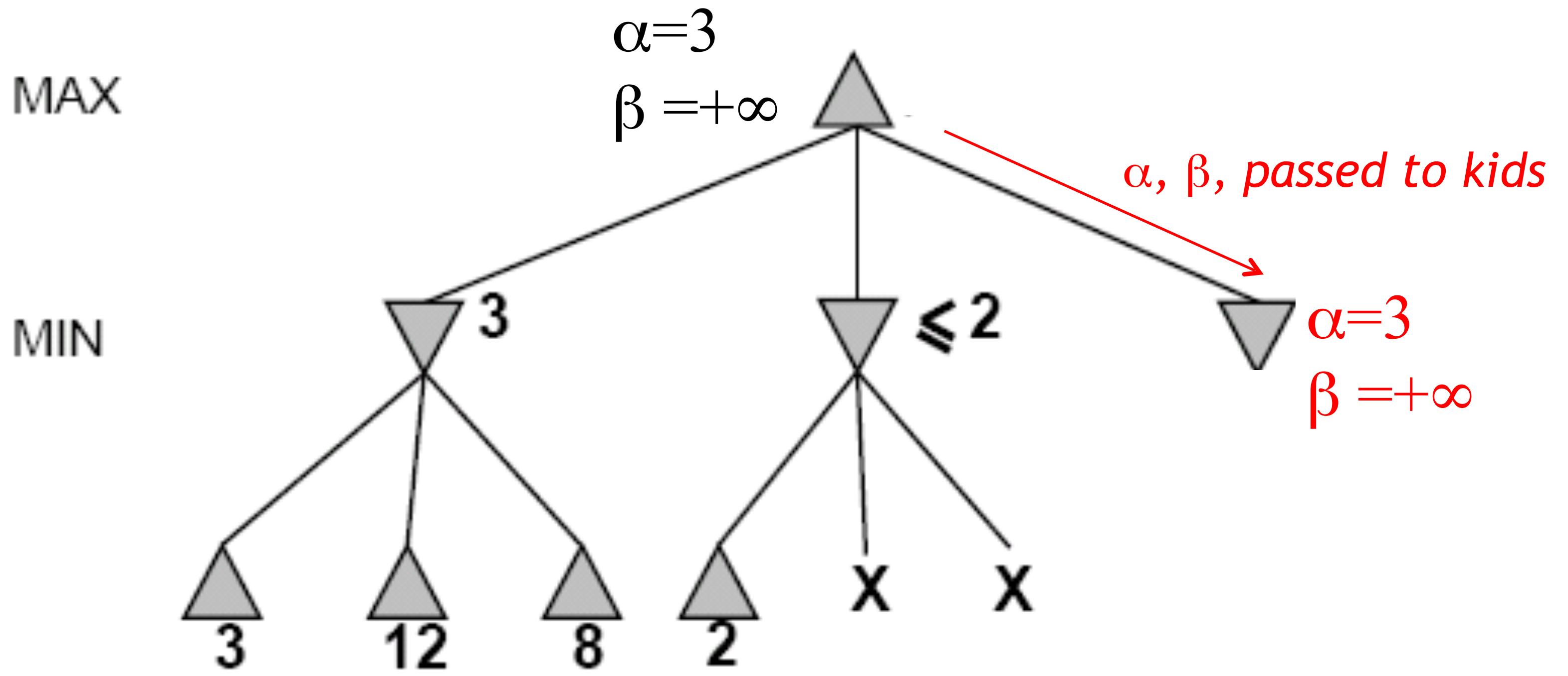
Alpha-Beta Example (continued)



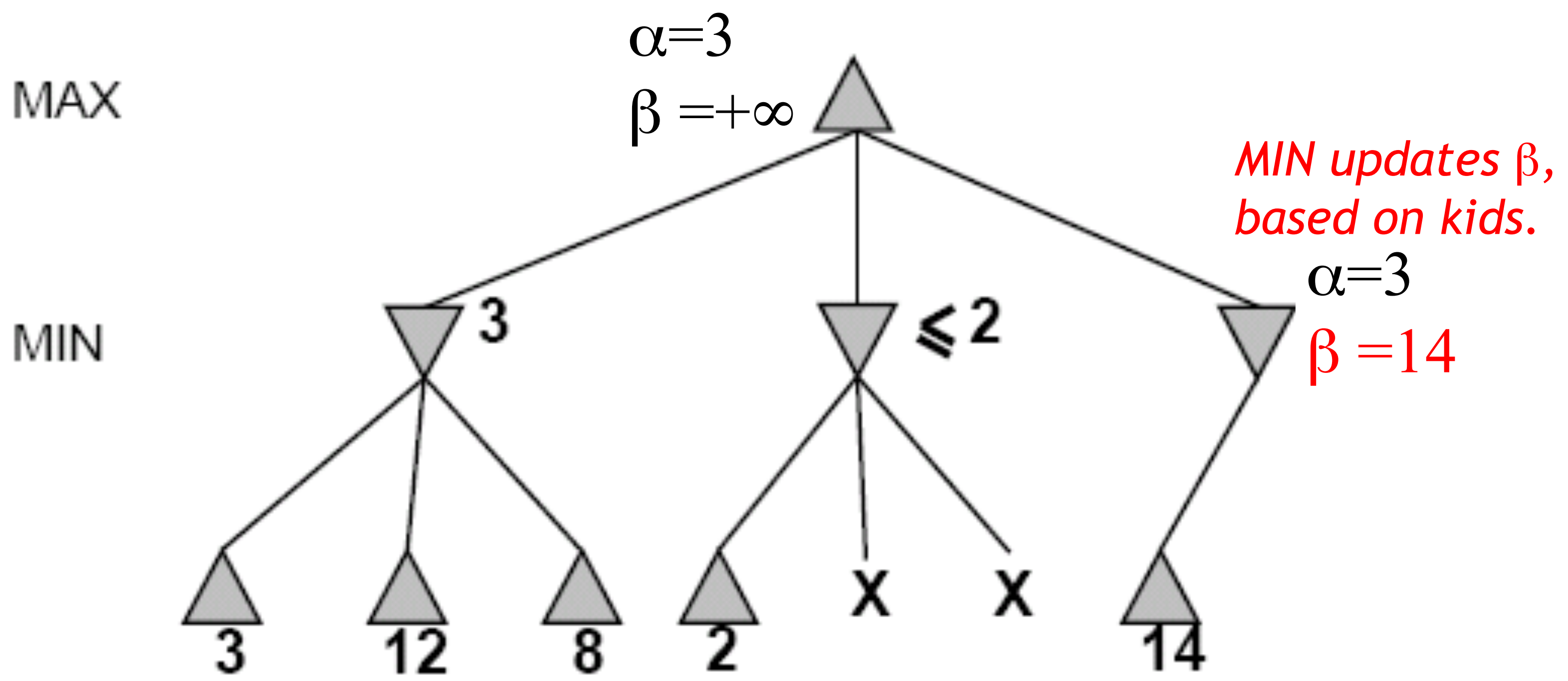
Alpha-Beta Example (continued)



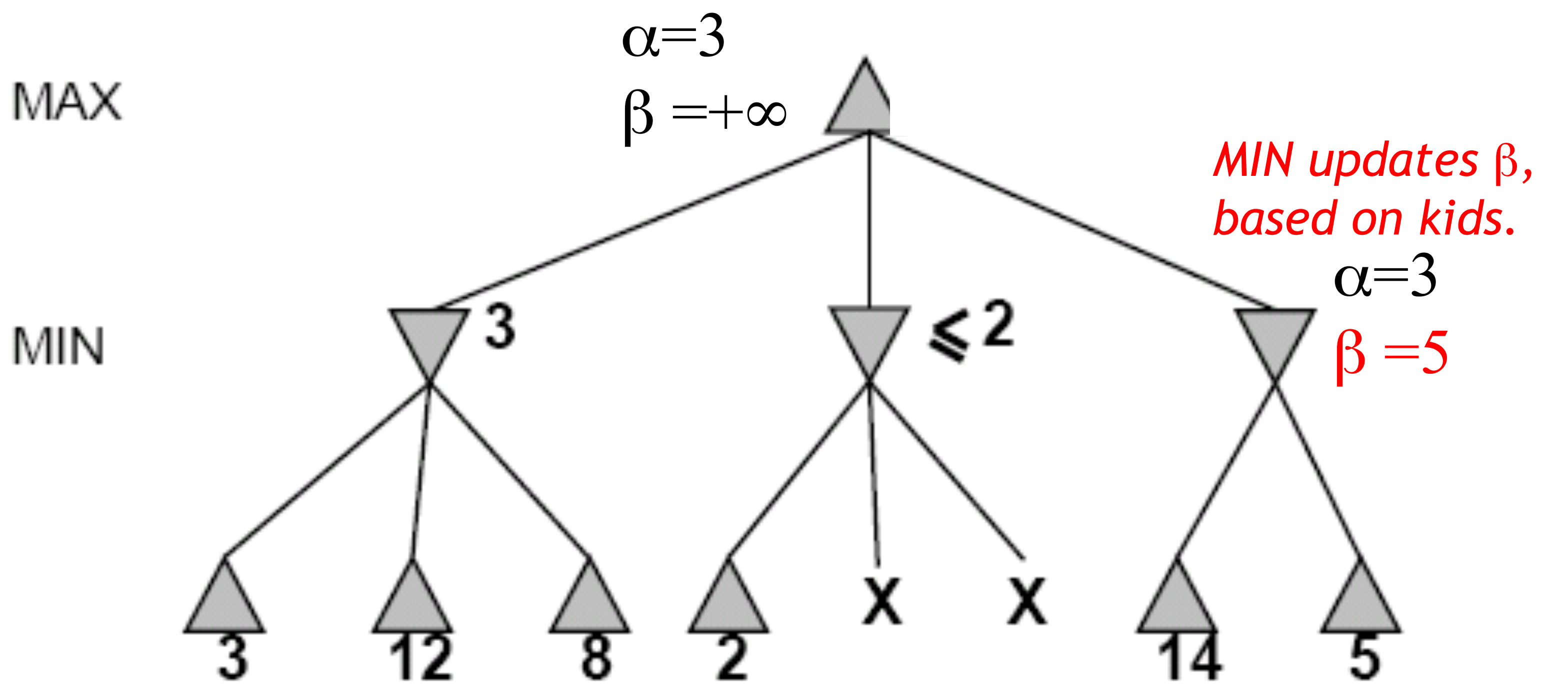
Alpha-Beta Example (continued)



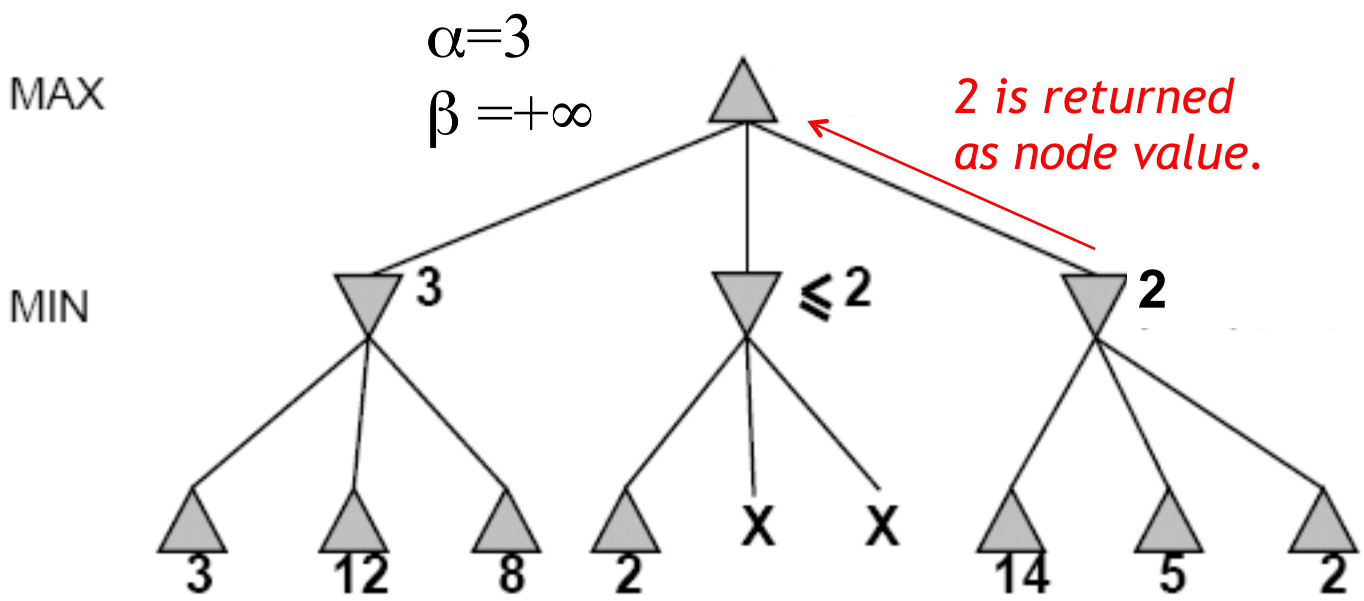
Alpha-Beta Example (continued)



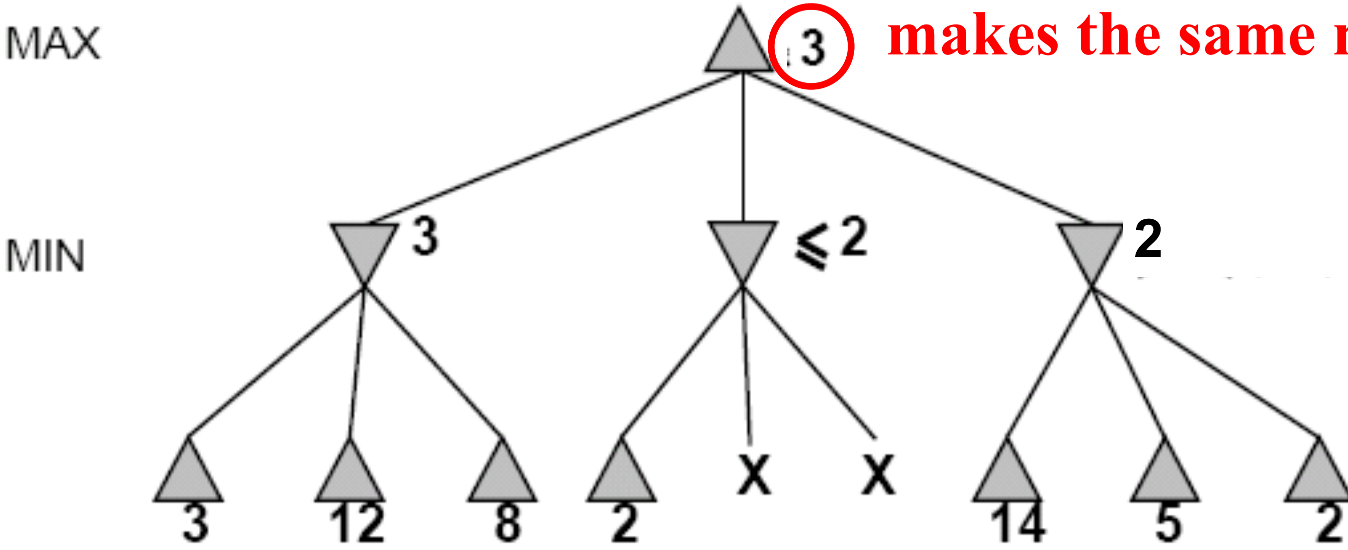
Alpha-Beta Example (continued)



Alpha-Beta Example (continued)

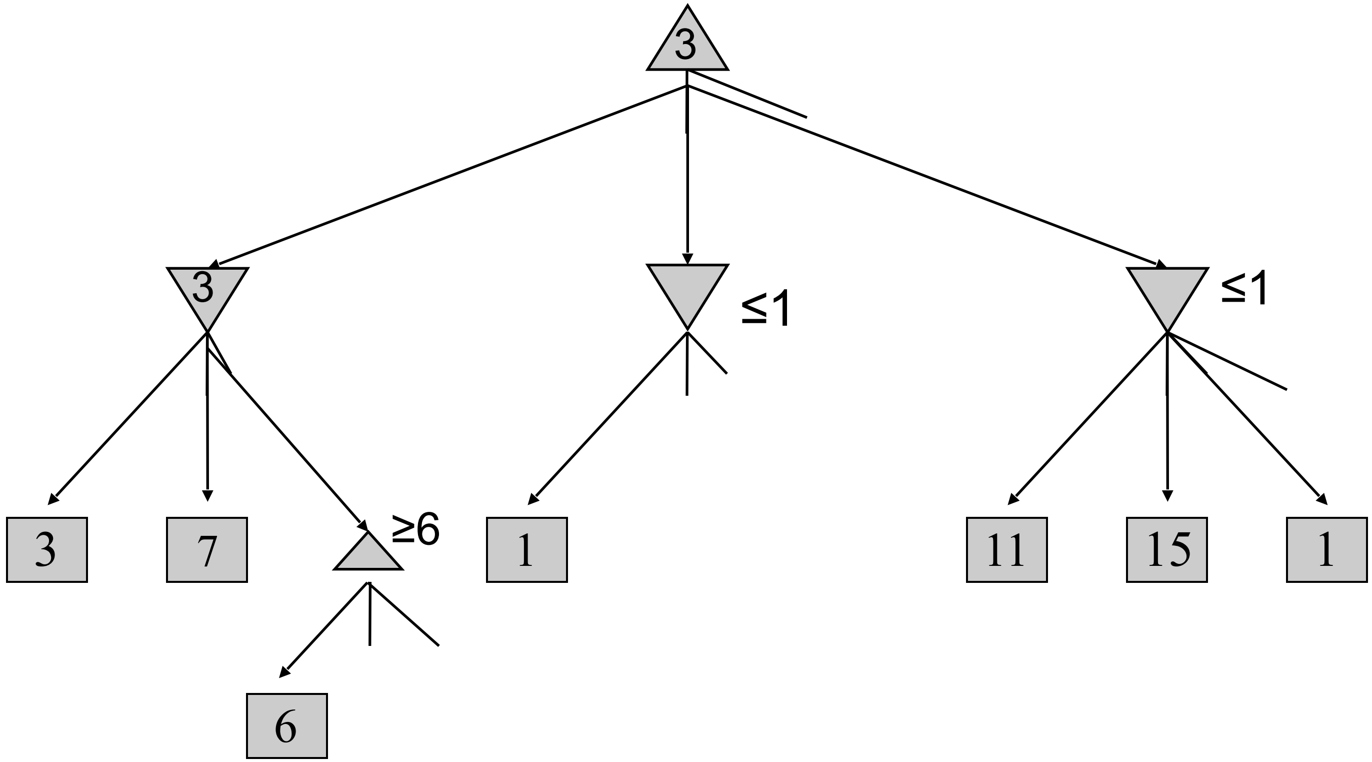


Alpha-Beta Example (continued)

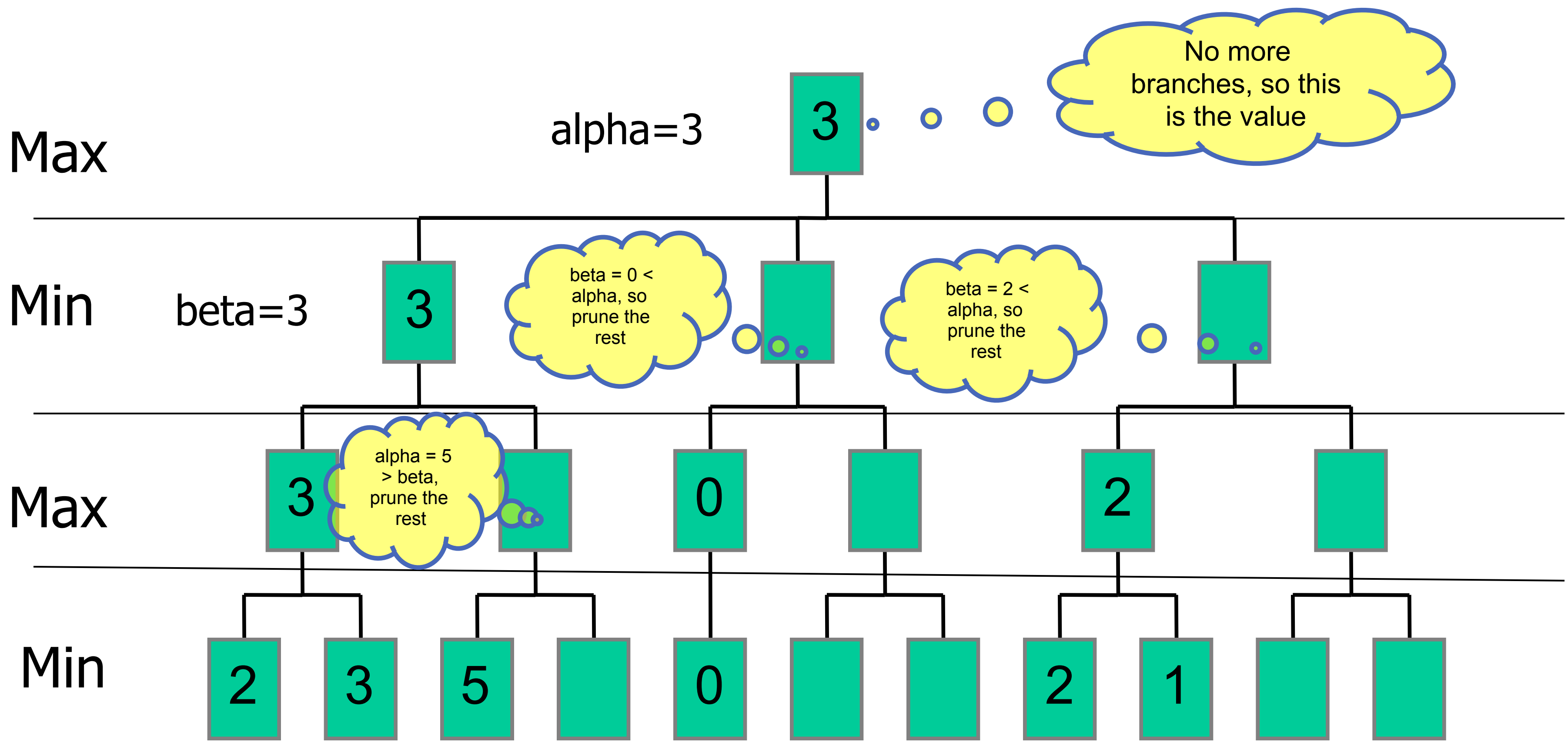


Max calculates the same node value, and makes the same move!

Alpha-Beta Pruning Example 2



Alpha-Beta Pruning Example 3



Effectiveness of Alpha-Beta Search

- Pruning does not affect the final result (optimal move).

- An entire sub-tree can be pruned.

- 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)
- Good move *ordering* improves effectiveness of pruning
  - 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),
    - $(\text{sqrt}(b))^d = b^{(d/2)}$,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

Static (Heuristic) Evaluation Functions

- Estimates how good the current board configuration is for a player.

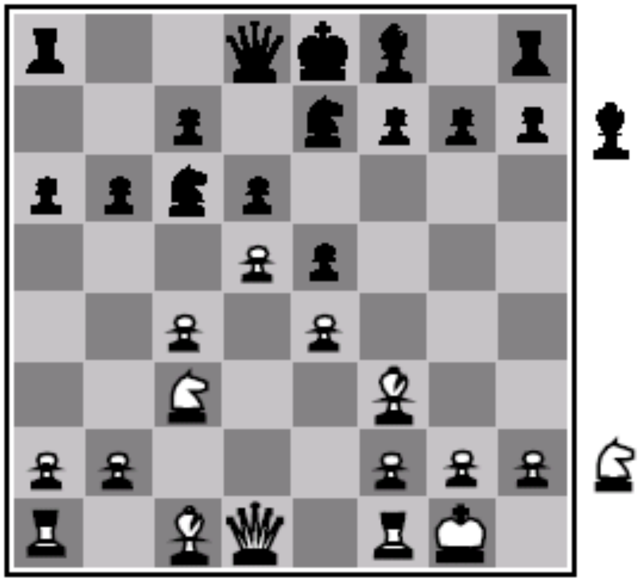
- Typically, evaluate how good it is for the player, how good it is for the opponent, then subtract the opponent's score from the player's.
  - Othello: Number of white pieces - Number of black pieces
  - Chess: Value of all white pieces - Value of all black pieces

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

- If the board evaluation is X for one player, it's -X for the other.

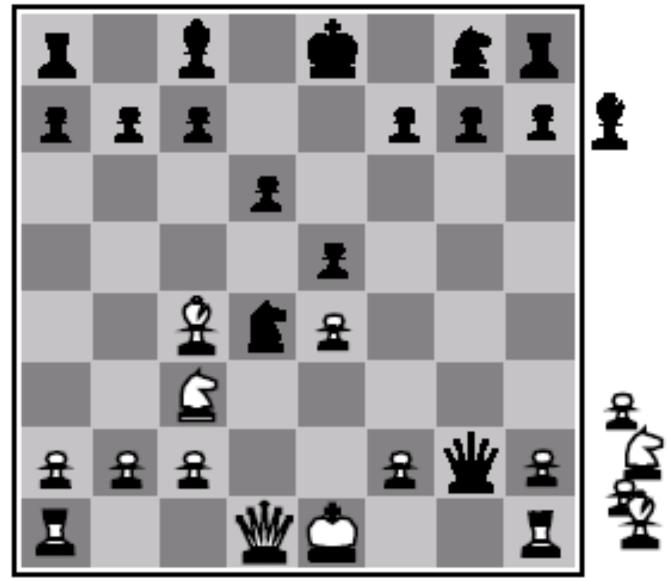
- This allows to perform **cut-off search**: after a maximum depth is reached, use a heuristic evaluation function instead of actual utility.

Evaluation functions



Black to move

White slightly better



White to move

Black winning

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.