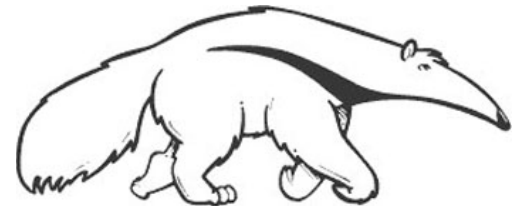


Games and Adversarial Search

CS271P, Winter 2018

Introduction to Artificial Intelligence

Prof. Richard Lathrop



Types of games

Deterministic:

Chance:

Perfect
Information:

chess, checkers, go,
othello

backgammon,
monopoly

Imperfect
Information:

battleship, Kriegspiel

Bridge, poker,
scrabble, ...

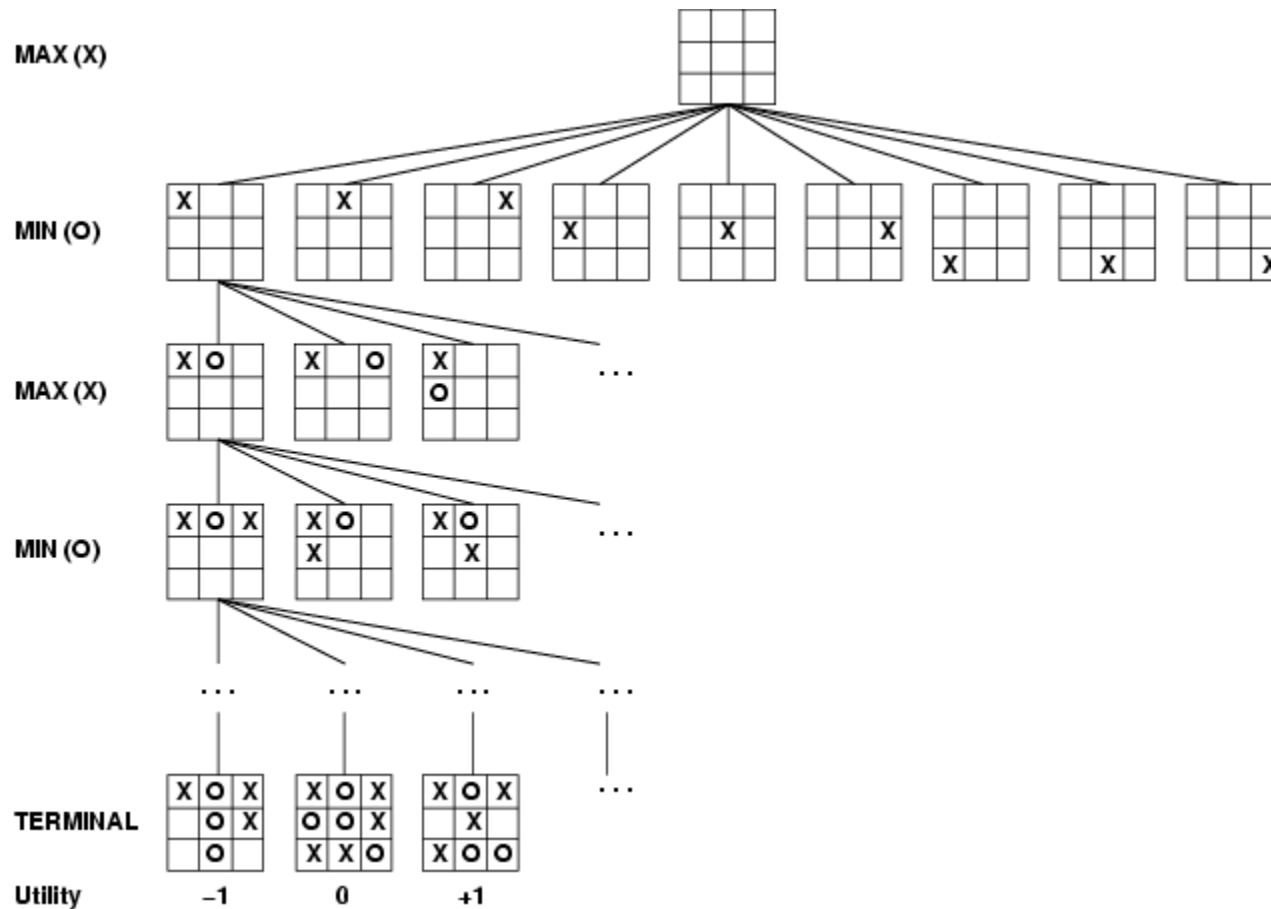
- Start with deterministic, perfect info games (easiest)
- Not considered:
 - Physical games like tennis, ice hockey, etc.
 - But, see “robot soccer”, <http://www.robocup.org/>

Typical assumptions

- Two agents, whose actions alternate
- Utility values for each agent are the opposite of the other
 - “Zero-sum” game; this creates adversarial situation
- Fully observable environments
- In game theory terms:
 - Deterministic, turn-taking, zero-sum, perfect information
- Generalizes: stochastic, multiplayer, non zero-sum, etc.
- Compare to e.g., Prisoner’s Dilemma” (R&N pp. 666-668)
 - Non-turn-taking, Non-zero-sum, Imperfect information

Game Tree (tic-tac-toe)

- All possible moves at each step



- 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 & CSP techniques can find optimal solution
 - Evaluation function: estimate cost from start to goal through a given node
 - Examples: path planning, scheduling activities, ...
- **Games:** adversary
 - Solution is a 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

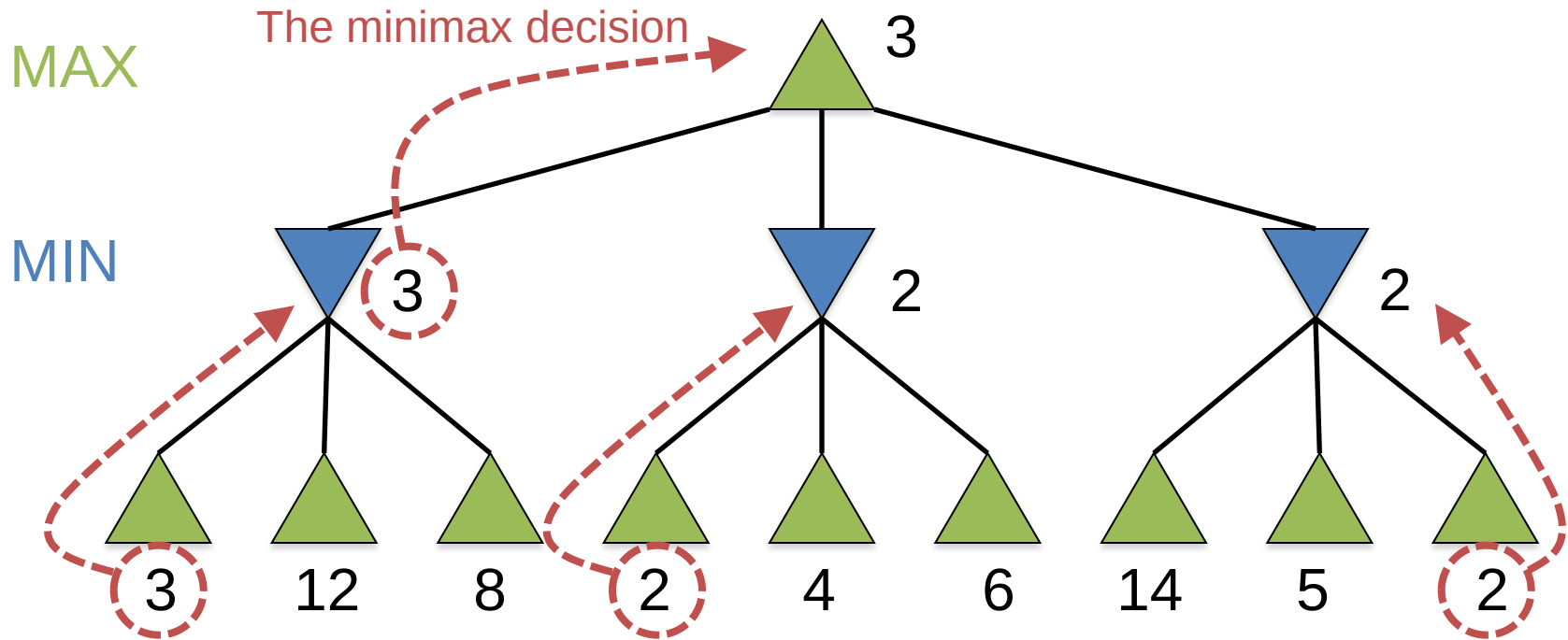
Games as search

- Two players, “MAX” and “MIN”
- MAX moves first, & take turns until game is over
 - Winner gets reward, loser gets penalty
 - “Zero sum”: sum of reward and penalty is constant
- Formal definition as a search problem:
 - **Initial state**: set-up defined by rules, e.g., initial board for chess
 - **Player(s)**: which player has the move in state s
 - **Actions(s)**: set of legal moves in a state
 - **Results(s,a)**: transition model defines result of a move
 - **Terminal-Test(s)**: true if the game is finished; false otherwise
 - **Utility(s,p)**: the 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 “best” next move

Min-Max: an optimal procedure

- Designed to find the optimal strategy & best move for MAX:
 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. At root: choose move leading to the child of highest value

Two-ply Game Tree



Minimax maximizes the utility of the worst-case outcome for MAX

Recursive min-max search

minMaxSearch(state)

return argmax([minValue(apply(state,a)) for each action a])

Simple stub to call recursion f'ns

maxValue(state)

if (terminal(state)) return utility(state);

v = -infty

for each action a:

v = max(v, minValue(apply(state,a)))

return v

If recursion limit reached, eval position

Otherwise, find our best child:

minValue(state)

if (terminal(state)) return utility(state);

v = infty

for each action a:

v = min(v, maxValue(apply(state,a)))

return v

If recursion limit reached, eval position

Otherwise, find the worst child:

Properties of minimax

- **Complete?** Yes (if tree is finite)
- **Optimal?**
 - Yes (against an optimal opponent)
 - Can it be beaten by a suboptimal opponent? (No – why?)
- **Time?** $O(b^m)$
- **Space?**
 - $O(bm)$ (depth-first search, generate all actions at once)
 - $O(m)$ (backtracking search, generate actions one at a time)

Game tree size

- Tic-tac-toe

- $B \approx 5$ legal actions per state on average; total 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 is quite reasonable

- Chess

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

It is usually impossible to develop the whole search tree.

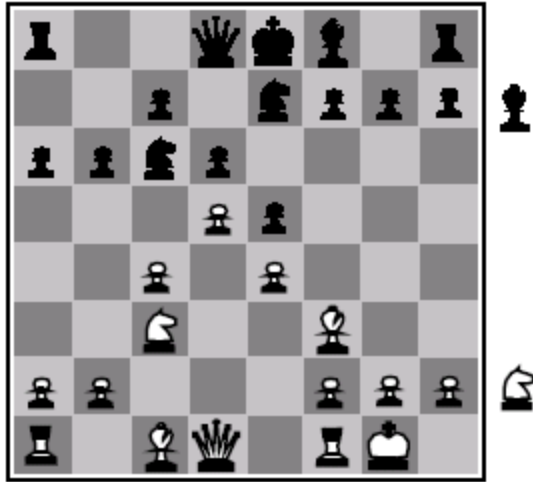
Cutting off search

- One solution: cut off tree before game ends
- Replace
 - Terminal(s) with Cutoff(s) – e.g., stop at some max depth
 - Utility(s,p) with Eval(s,p) – estimate position quality
- Does it work in practice?
 - $b^m \approx 10^6$, $b \approx 35 \rightarrow m \approx 4$
 - 4-ply look-ahead is a poor chess player
 - 4-ply $\approx$ human novice
 - 8-ply $\approx$ typical PC, human master
 - 12-ply $\approx$ Deep Blue, human grand champion Kasparov
 - $35^{12} \approx 10^{18}$ (Yikes! but possible, with other clever methods)

Static (Heuristic) Evaluation Functions

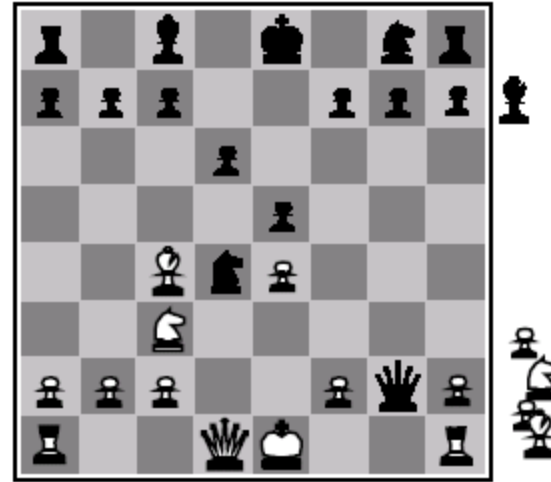
- An Evaluation Function:
 - Estimate how good the current board configuration is for a player.
 - Typically, evaluate how good it is for the player, and how good it is for the opponent, and subtract the opponent's score from the player's.
 - Often called “static” because it is called on a static board position
 - Ex: Othello: Number of white pieces - Number of black pieces
 - Ex: Chess: Value of all white pieces - Value of all black pieces
- Typical value ranges:
[-1, 1] (loss/win) or [-1 , +1] or [0 , 1]
- Board evaluation: X for one player => -X for opponent
 - Zero-sum game: scores sum to a constant

Evaluation functions



Black to move

White slightly better



White to move

Black winning

For chess, typically *linear* weighted sum of *features*

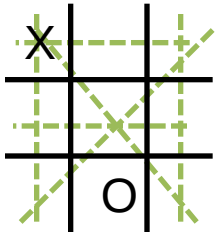
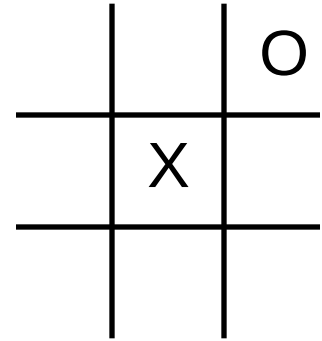
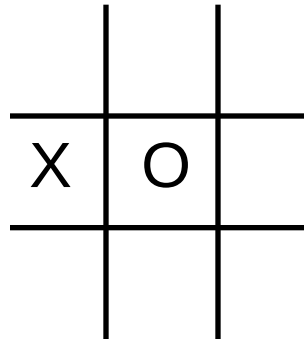
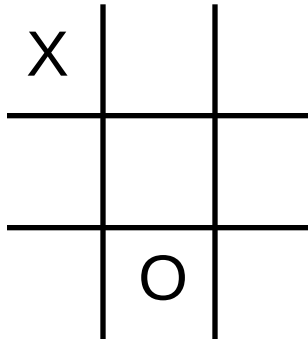
$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \dots + w_n f_n(s)$$

e.g., $w_1 = 9$ with

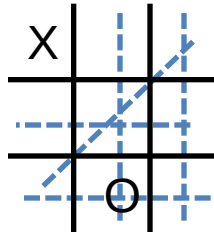
$f_1(s) = (\text{number of white queens}) - (\text{number of black queens}), \text{ etc.}$

Applying minimax to tic-tac-toe

- The static heuristic evaluation function:
 - Count the number of possible win lines



X has 6
possible win
paths



O has 5
possible win
paths

$$E(s) = 6 - 5 = 1$$

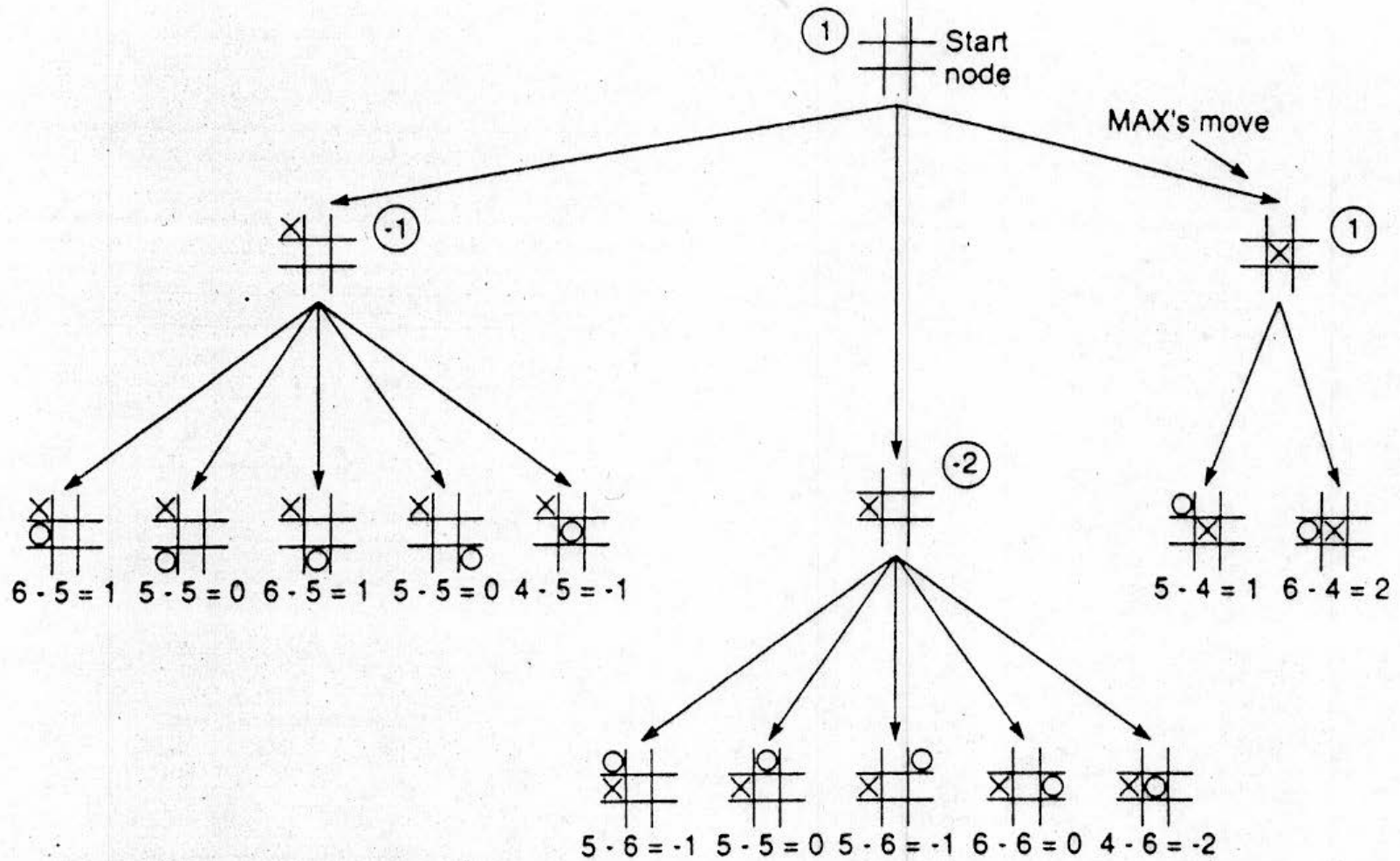
X has 4 possible wins
O has 6 possible wins

$$E(n) = 4 - 6 = -2$$

X has 5 possible wins
O has 4 possible wins

$$E(n) = 5 - 4 = 1$$

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



Minimax values (two ply)

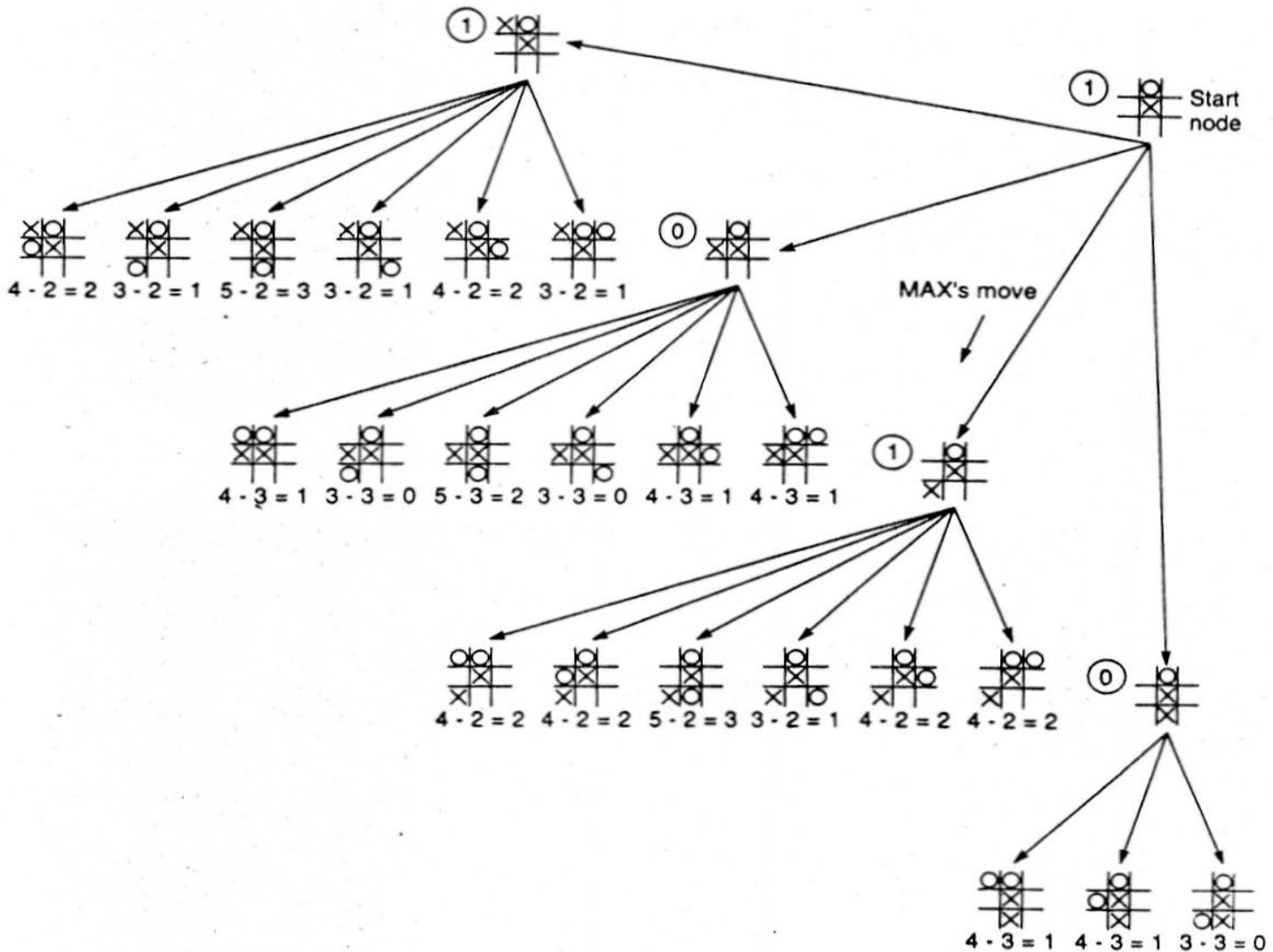


Figure 4.18 Two-ply minimax applied to X's second move of tic-tac-toe.

Minimax values (two ply)

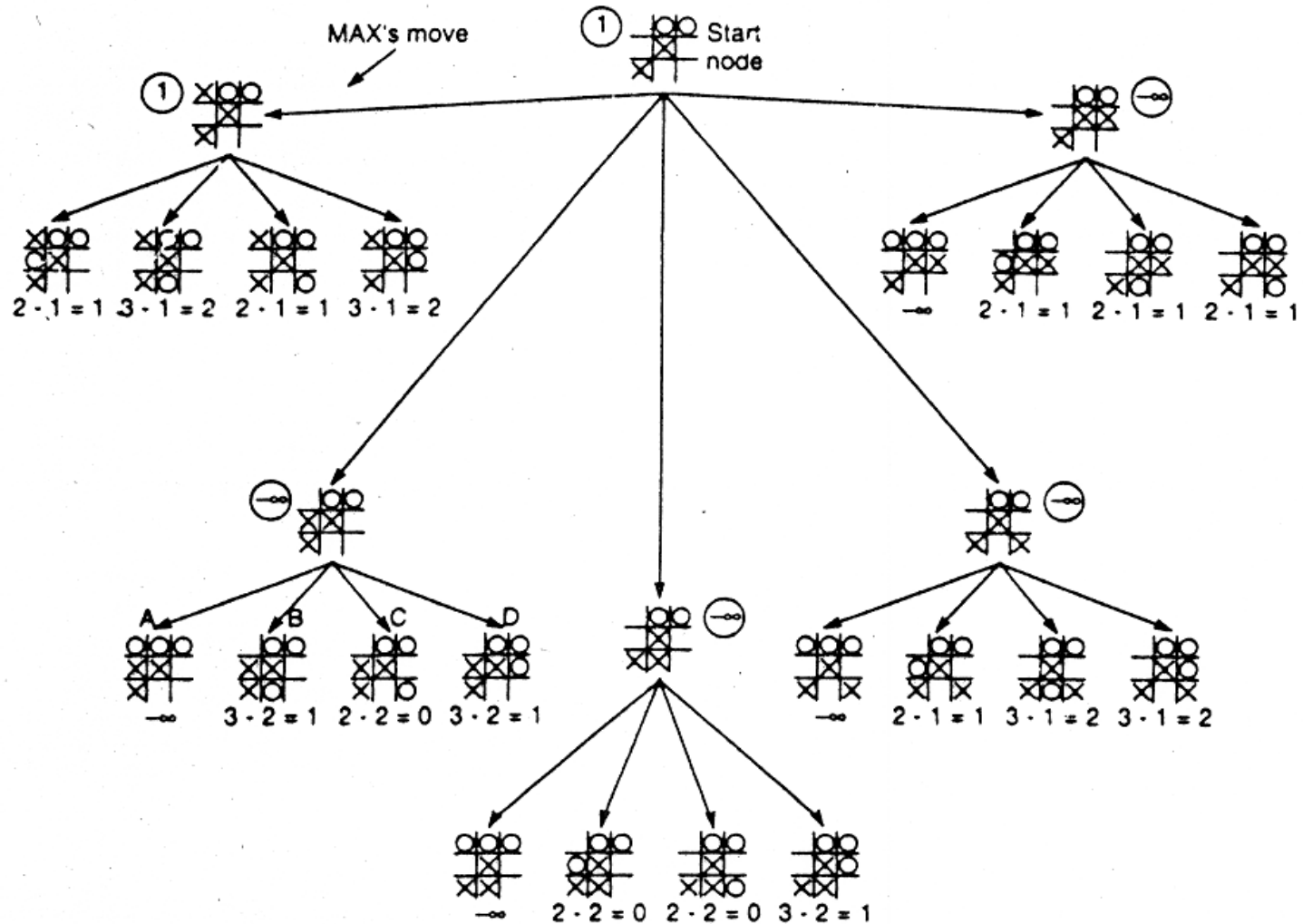
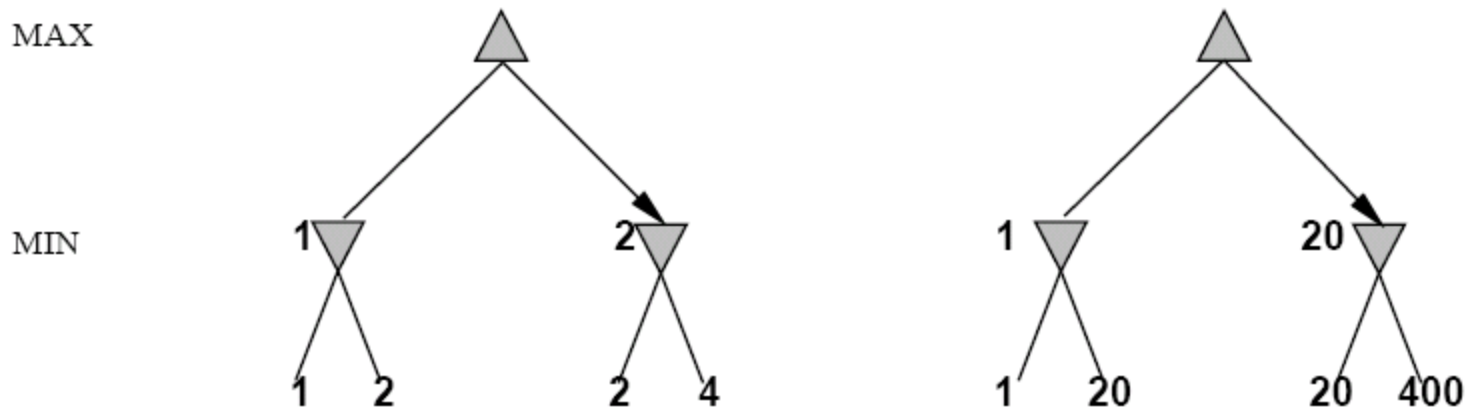


Figure 4.19 Two-ply minimax applied to X's move near end game.

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

Iterative deepening

- In real games, there is usually a time limit T to make a move
- How do we take this into account?
- Minimax cannot use “partial” results with any confidence, unless the full tree has been searched
 - Conservative: set small depth limit to guarantee finding a move in time $< T$
 - But, we may finish early – could do more search!
- In practice, iterative deepening search (IDS) is used
 - IDS: depth-first search with increasing depth limit
 - When time runs out, use the solution from previous depth
 - With alpha-beta pruning (next), we can sort the nodes based on values from the previous depth limit in order to maximize pruning during the next depth limit $\Rightarrow$ search deeper!

Limited horizon effects

- 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
 - Often better to explore some branches more deeply in the allotted time
 - Various heuristics exist to identify “promising” branches
 - Stop at “quiescent” positions – all battles are over, things are quiet
 - Continue when things are in violent flux – the middle of a battle

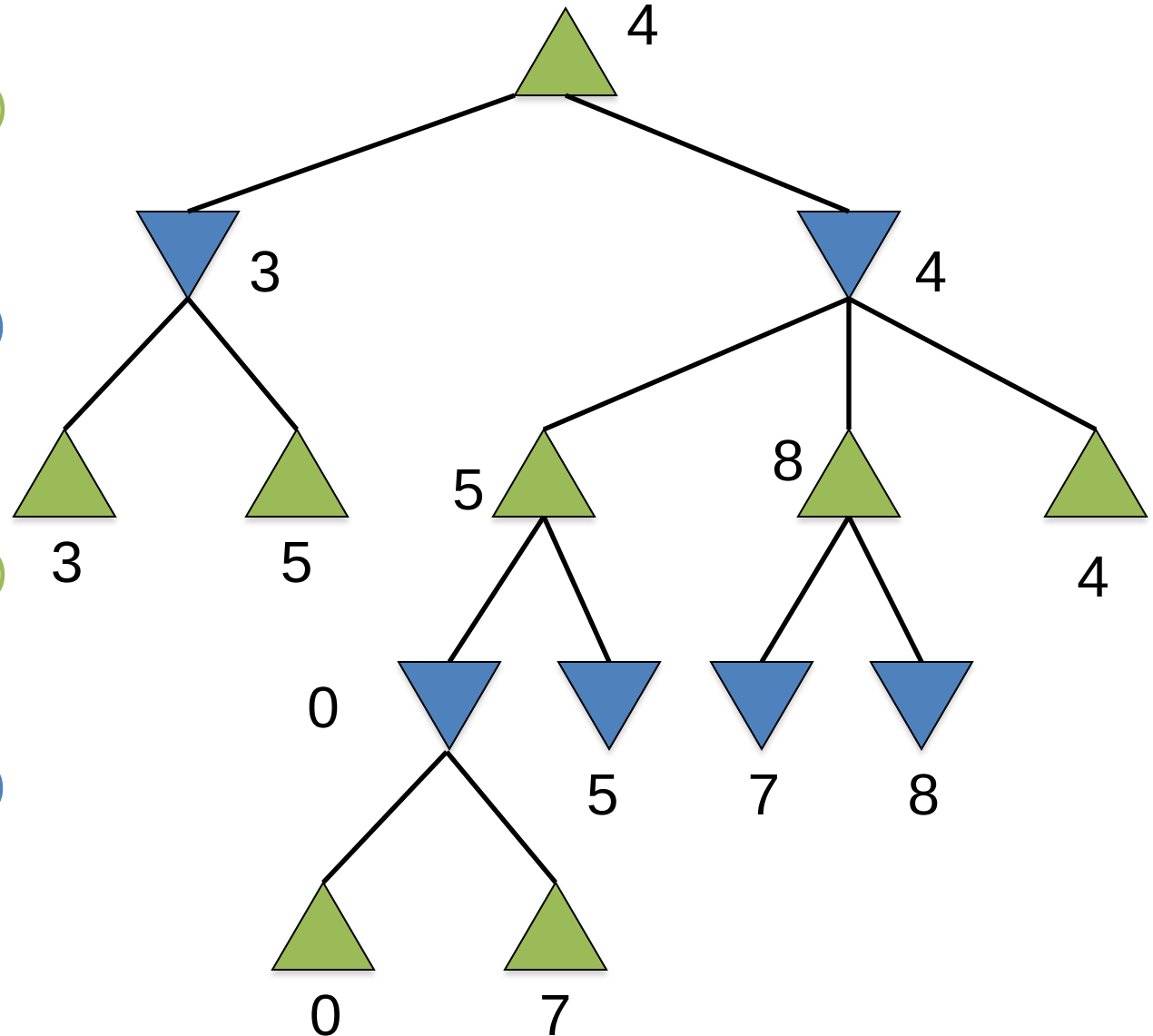
Selectively deeper game trees

MAX
(Computer's move)

MIN
(Opponent's move)

MAX
(Computer's move)

MIN
(Opponent's move)



Eliminate redundant nodes

- On average, each board position appears in the search tree approximately $10^{150} / 10^{40} \approx 10^{100}$ times
 - Vastly redundant search effort
- Can't remember all nodes (too many)
 - Can't eliminate all redundant nodes
- 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

Summary

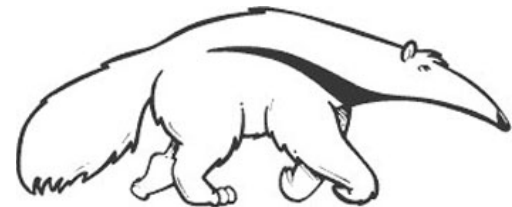
- Game playing as a search problem
- Game trees represent alternate computer / opponent moves
- Minimax: choose moves by assuming the opponent will always choose the move that is best for them
 - Avoids all worst-case outcomes for Max, to find the best
 - If opponent makes an error, Minimax will take optimal advantage (after) & make the best possible play that exploits the error
- Cutting off search
 - In general, it's infeasible to search the entire game tree
 - In practice, Cutoff-Test decides when to stop searching
 - Prefer to stop at quiescent positions
 - Prefer to keep searching in positions that are still in flux
- Static heuristic evaluation function
 - Estimate the quality of a given board configuration for MAX player
 - Called when search is cut off, to determine value of position found

Games & Adversarial Search: Alpha-Beta Pruning

CS171, Fall 2017

Introduction to Artificial Intelligence

Prof. Richard Lathrop



Alpha-Beta pruning

- Exploit the “fact” of an adversary
- If a position is **provably bad**
 - It’s no use searching to find out just how bad
- If the adversary **can force a bad position**
 - It’s no use searching to find the good positions the adversary won’t let you achieve
- Bad = **not better** than we can get elsewhere

Pruning with Alpha/Beta

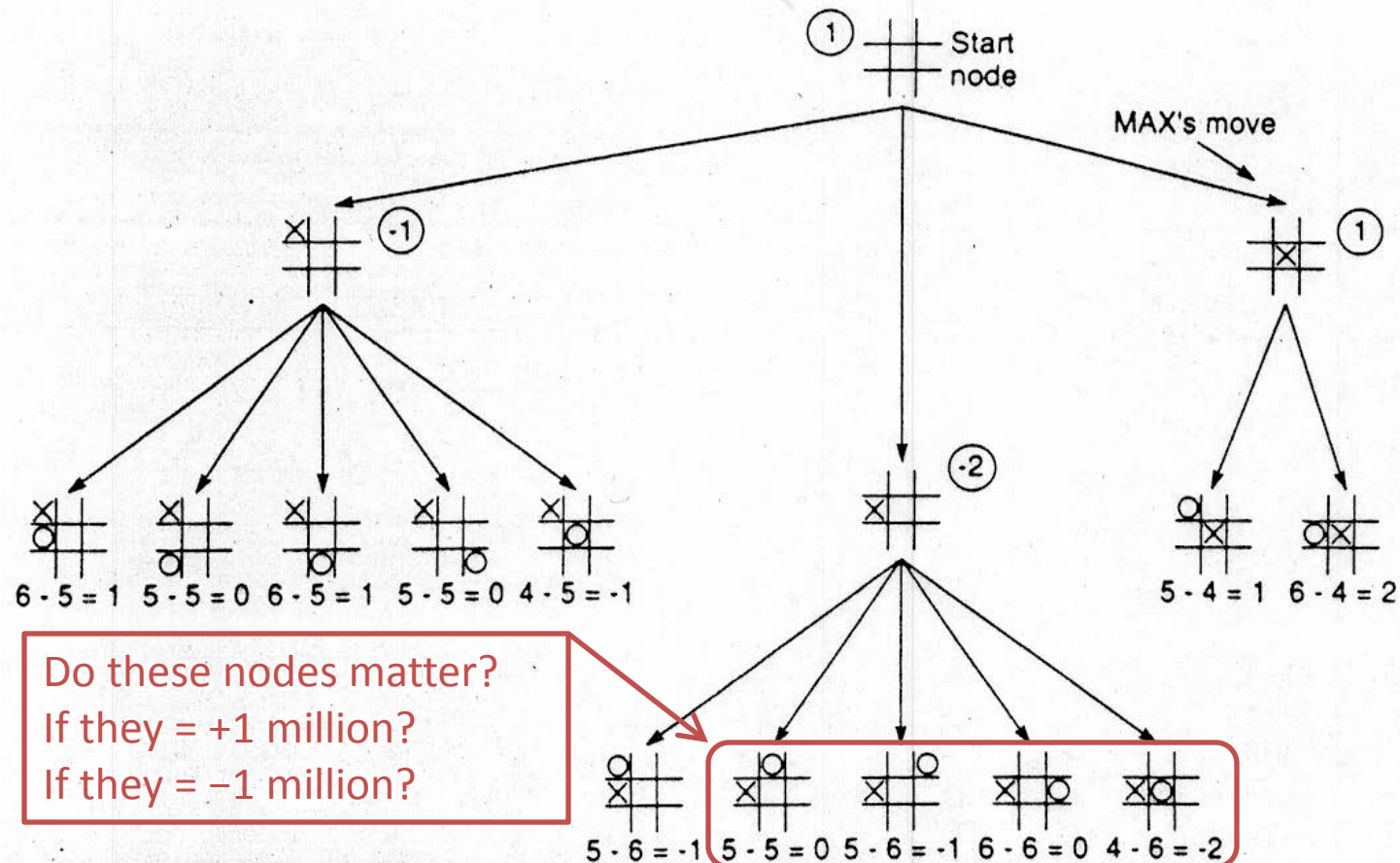
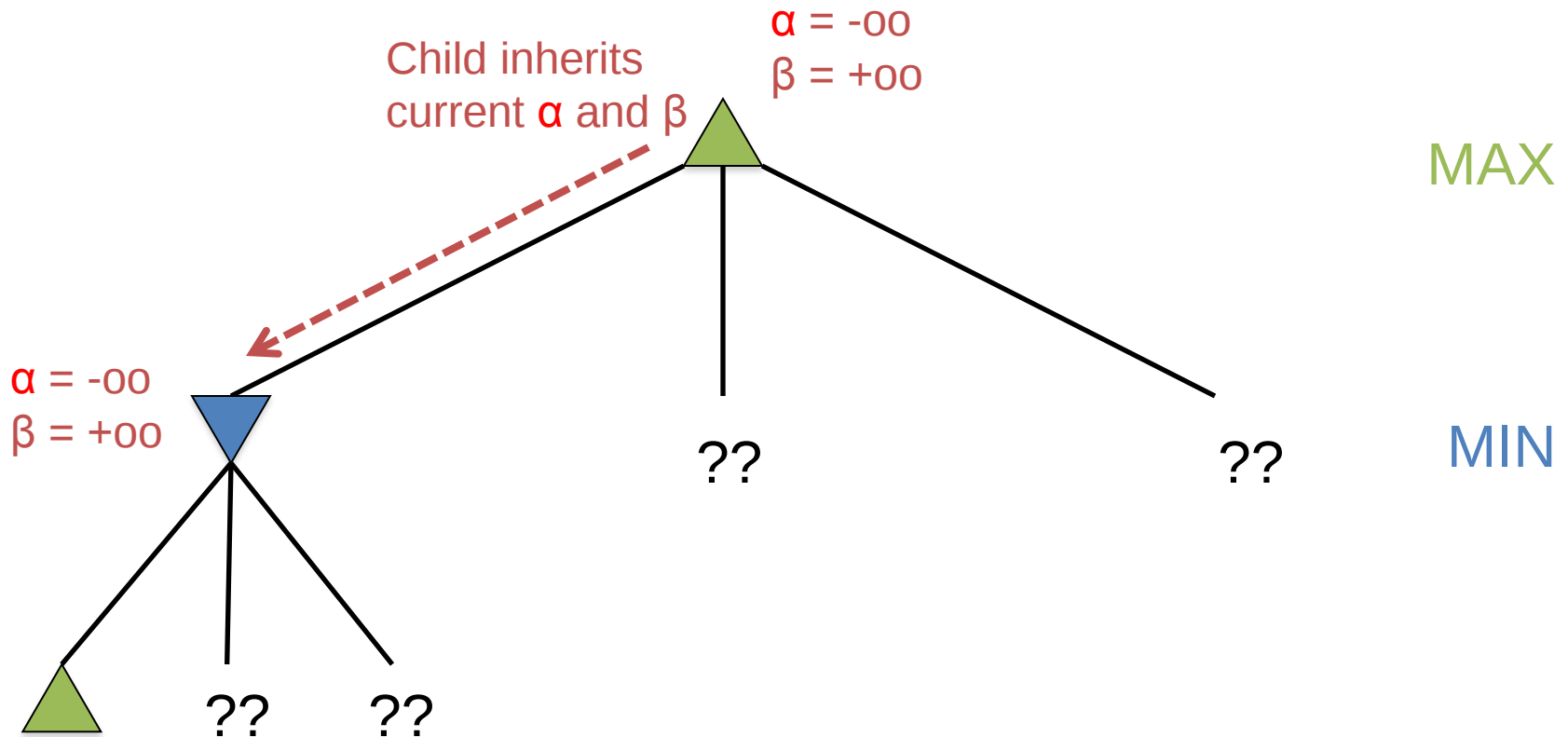


Figure 4.17 Two-ply minimax applied to the opening move of tic-tac-toe.

Alpha-Beta Example

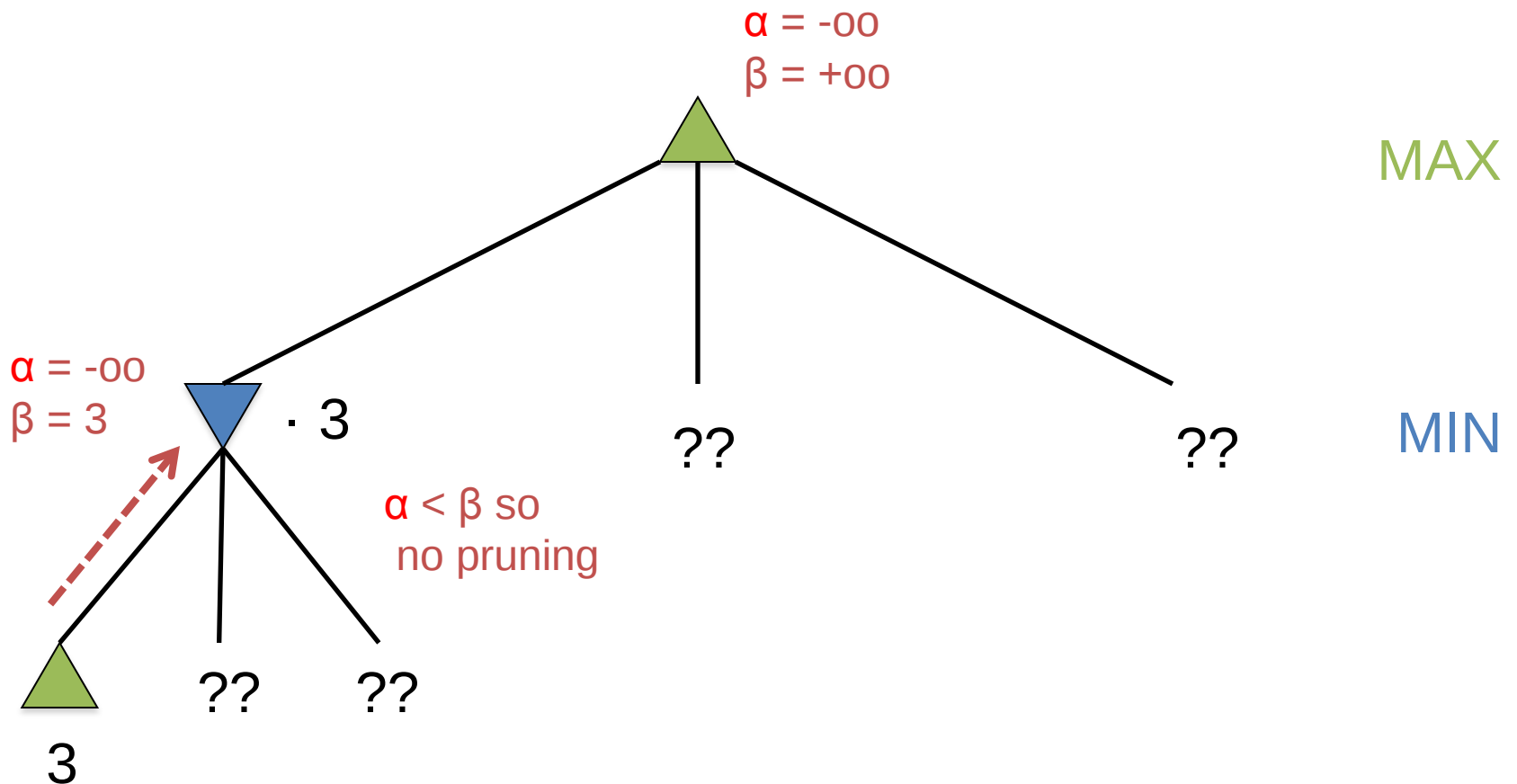
Initially, possibilities are unknown: range ($\alpha = -\infty$, $\beta = +\infty$)

Do a depth-first search to the first leaf.



Alpha-Beta Example

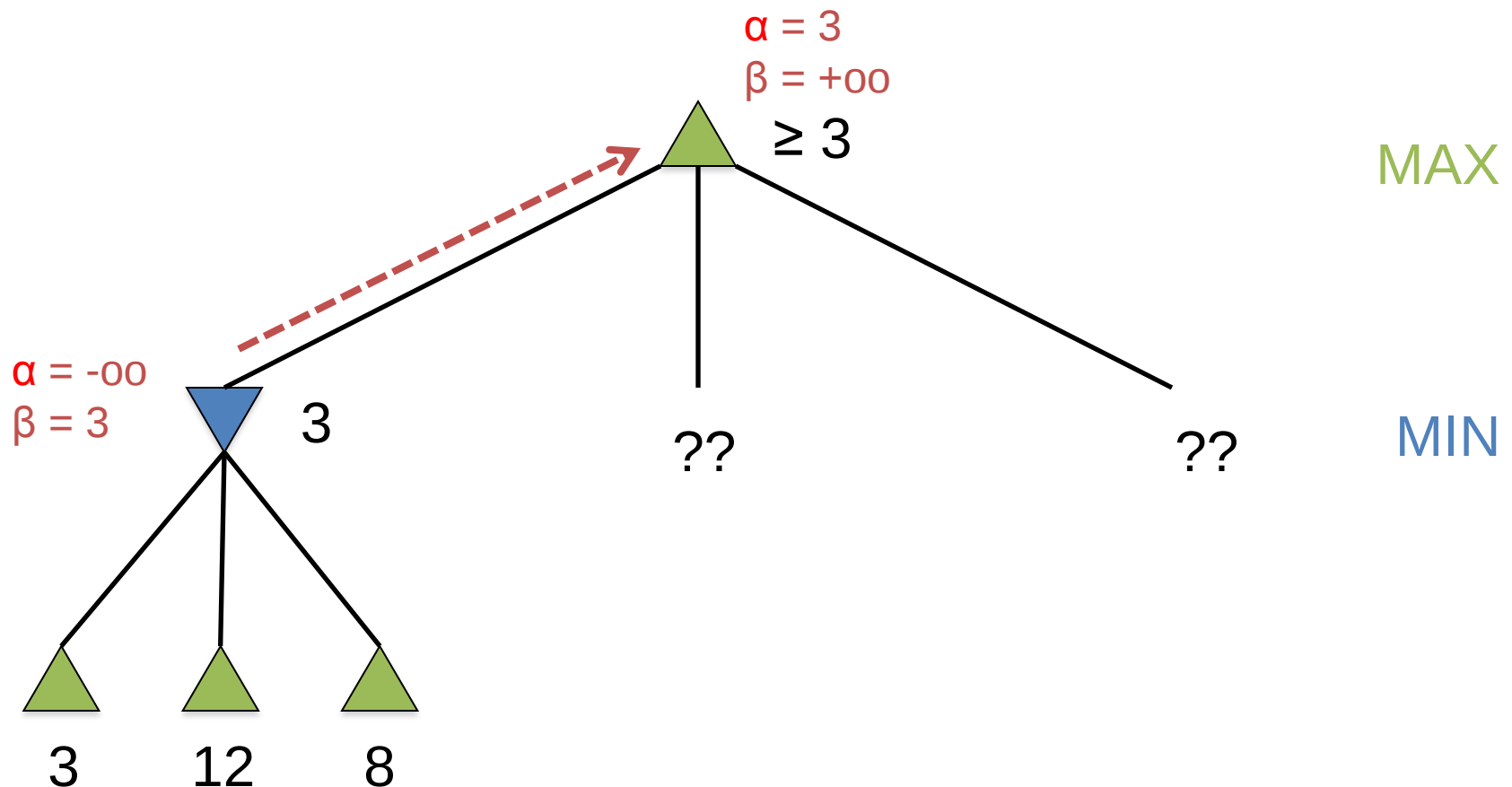
See the first leaf, after MIN's move: MIN updates β



Alpha-Beta Example

See remaining leaves; value is known

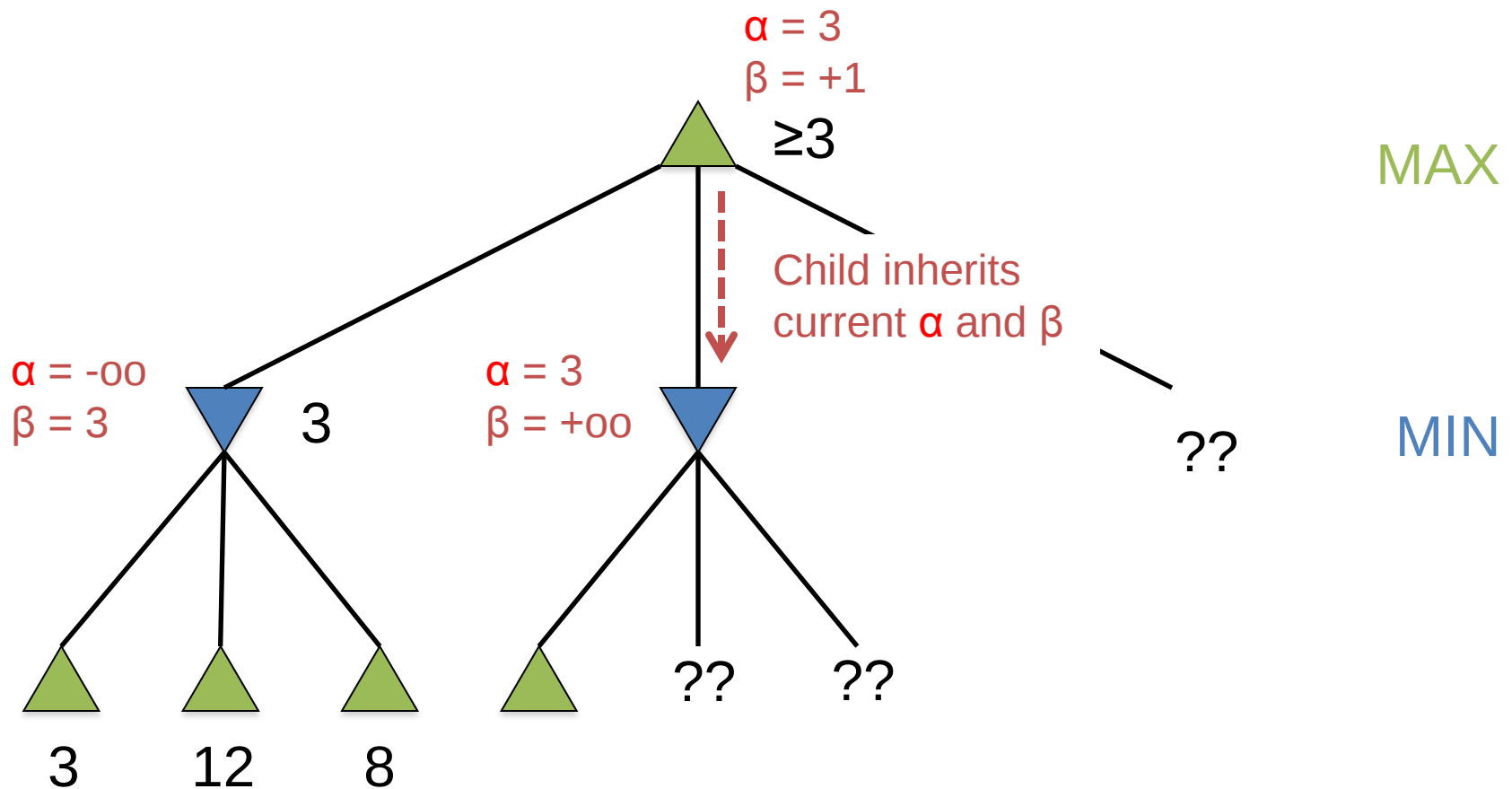
Pass outcome to caller; MAX updates α



Alpha-Beta Example

Continue depth-first search to next leaf.

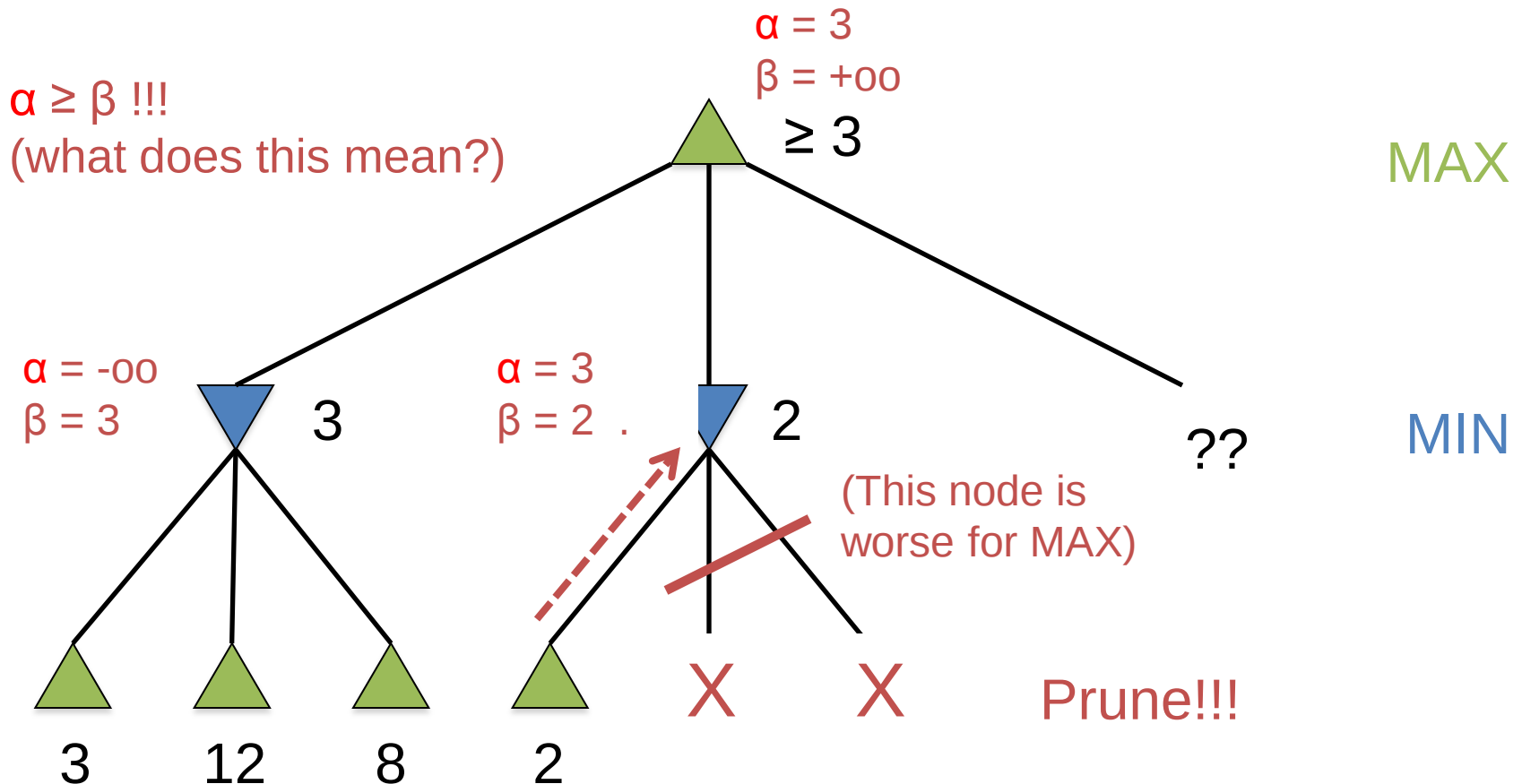
Pass α , β to descendants



Alpha-Beta Example

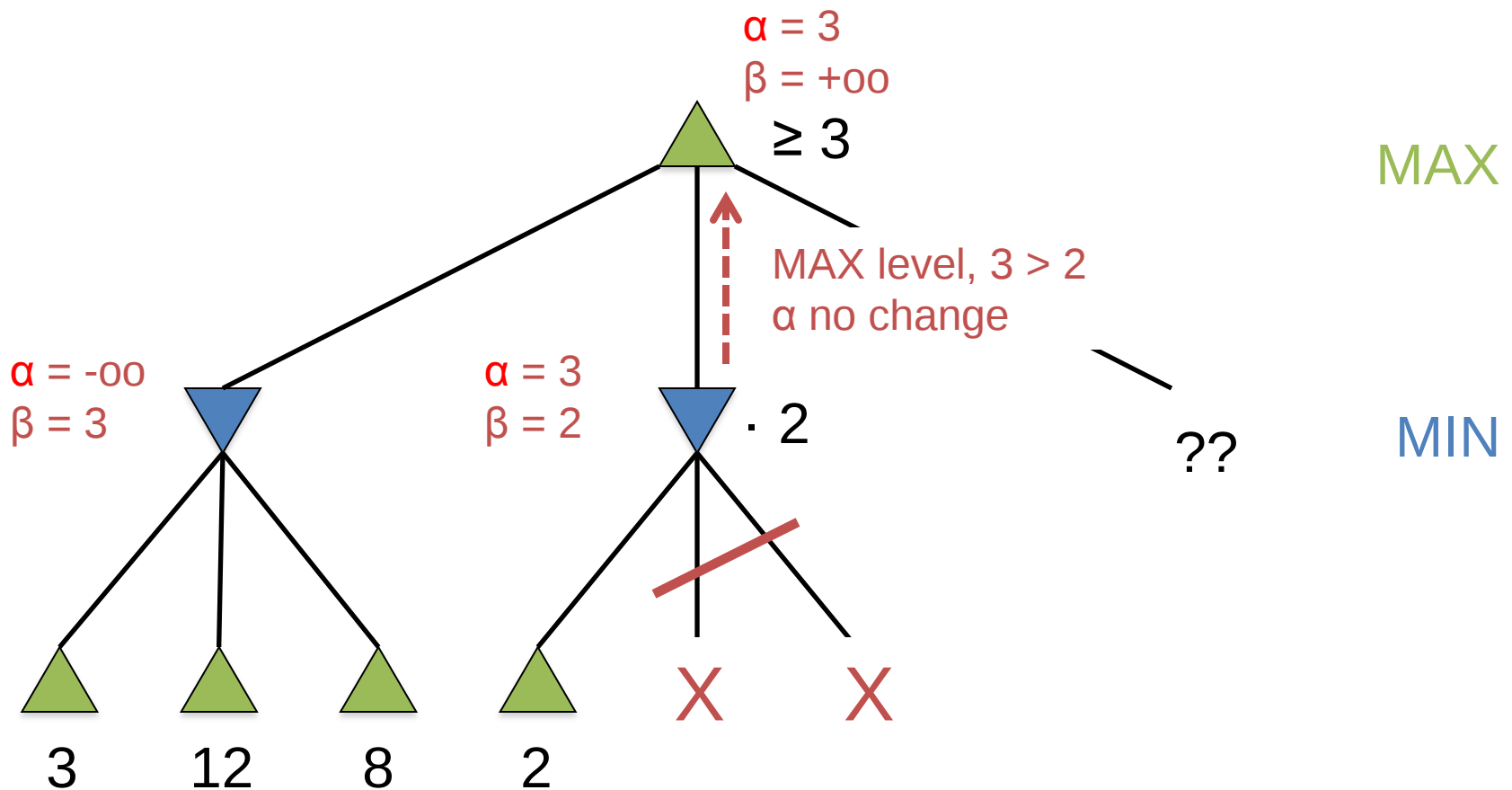
Observe leaf value; MIN's level; MIN updates β

Prune – play will never reach the other nodes!



Alpha-Beta Example

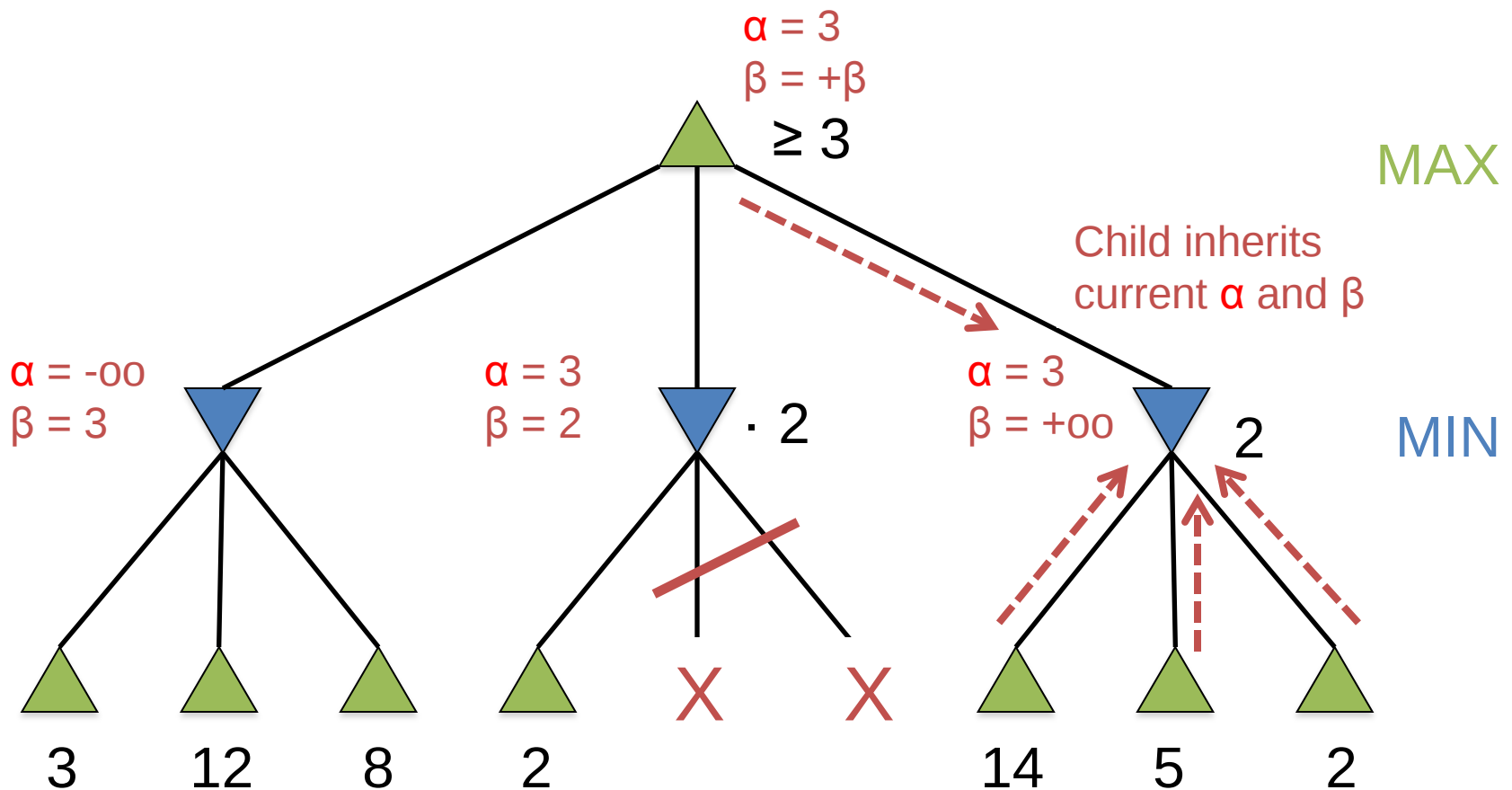
Pass outcome to caller & update caller:



Alpha-Beta Example

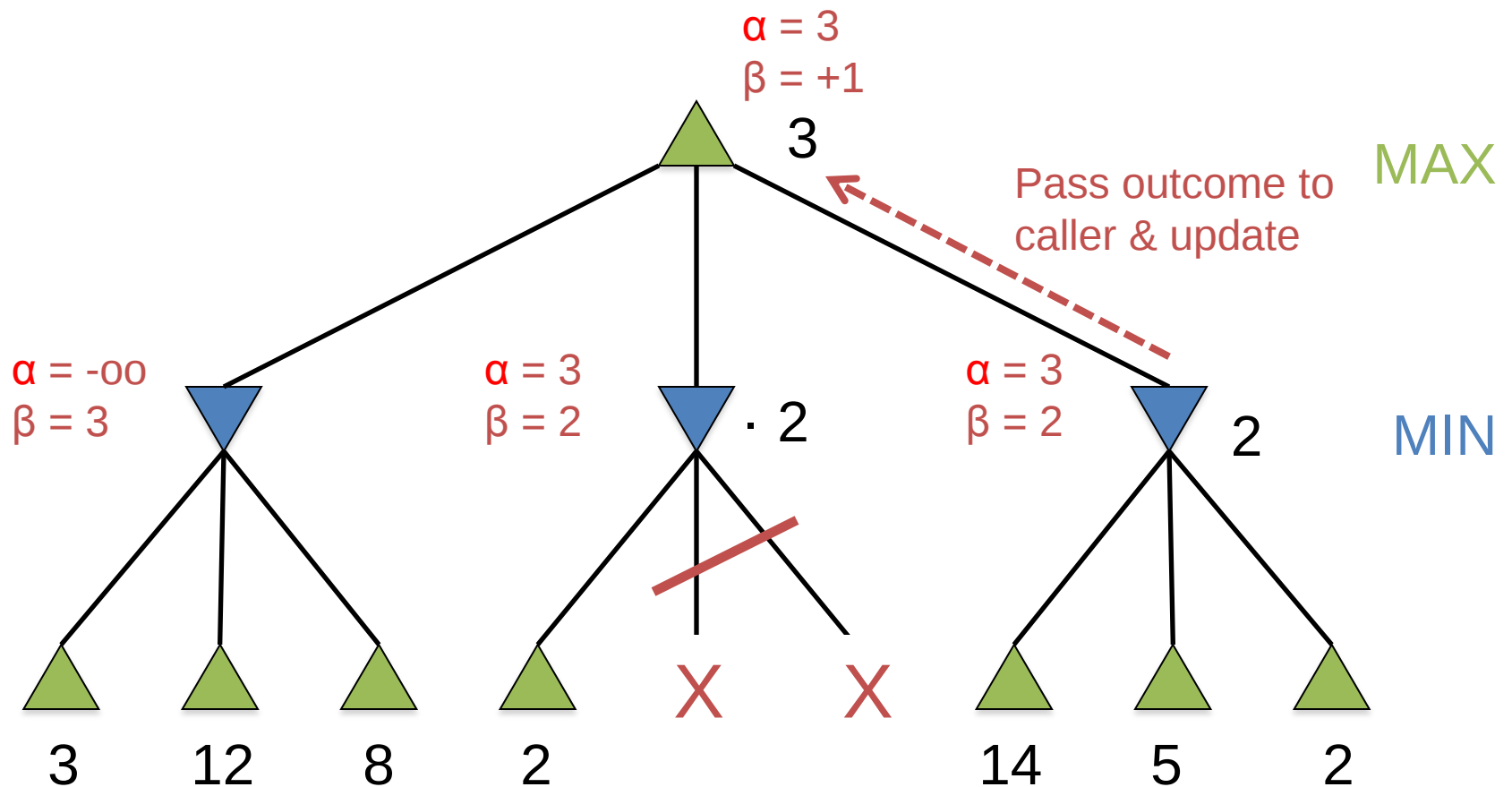
Continue depth-first exploration...

No pruning here; value is not resolved until final leaf.



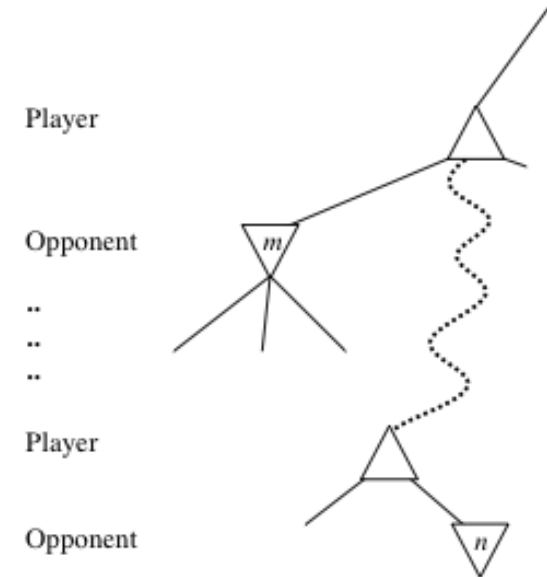
Alpha-Beta Example

Value at the root is resolved.



General alpha-beta pruning

- Consider a node n in the tree:
- If player has a better choice at
 - Parent node of n
 - Or, any choice further up!
- Then n is never reached in play



- So:
 - When that much is known about n , it can be pruned

Recursive α - β pruning

abSearch(state)

alpha, beta, a = -infty, +infty, None

for each action a:

alpha, a = max((alpha,a) , (minValue(apply(state,a), alpha, beta), a))

return a

Simple stub to call recursion f'ns

Initialize alpha, beta; no move found

Score each action; update alpha & best action

maxValue(state, al, be)

if (cutoff(state)) return eval(state);

for each action a:

al = max(al, minValue(apply(state,a), al, be)

if (al , be) return +infty

return al

If recursion limit reached, eval heuristic

Otherwise, find our best child:

If our options are too good, our min
ancestor will never let us come this way

Otherwise return the best we can find

minValue(state, al, be)

if (cutoff(state)) return eval(state);

for each action a:

be = min(be, maxValue(apply(state,a), al, be)

if (al , be) return -infty

return be

If recursion limit reached, eval heuristic

Otherwise, find the worst child:

If our options are too bad, our max
ancestor will never let us come this way

Otherwise return the worst we can find

Effectiveness of α - β 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
- In practice often get $O(b^{(d/2)})$ rather than $O(b^d)$
 - This is the same as having a branching factor of $\sqrt{b}$,
 - since $(\sqrt{b})^d = b^{(d/2)}$ (i.e., we have effectively gone from b to square root of b)
 - In chess go from $b \sim 35$ to $b \sim 6$
 - permitting much deeper search in the same amount of time

Iterative deepening

- In real games, there is usually a time limit T to make a move
- How do we take this into account?
- Minimax cannot use “partial” results with any confidence, unless the full tree has been searched
 - Conservative: set small depth limit to guarantee finding a move in time $< T$
 - But, we may finish early – could do more search!
- Added benefit with Alpha-Beta Pruning:
 - Remember node values found at the previous depth limit
 - Sort current nodes so that each player’s best move is left-most child
 - Likely to yield good Alpha-Beta Pruning $\Rightarrow$ better, faster search
 - Only a heuristic: node values will change with the deeper search
 - Usually works well in practice

Comments on alpha-beta pruning

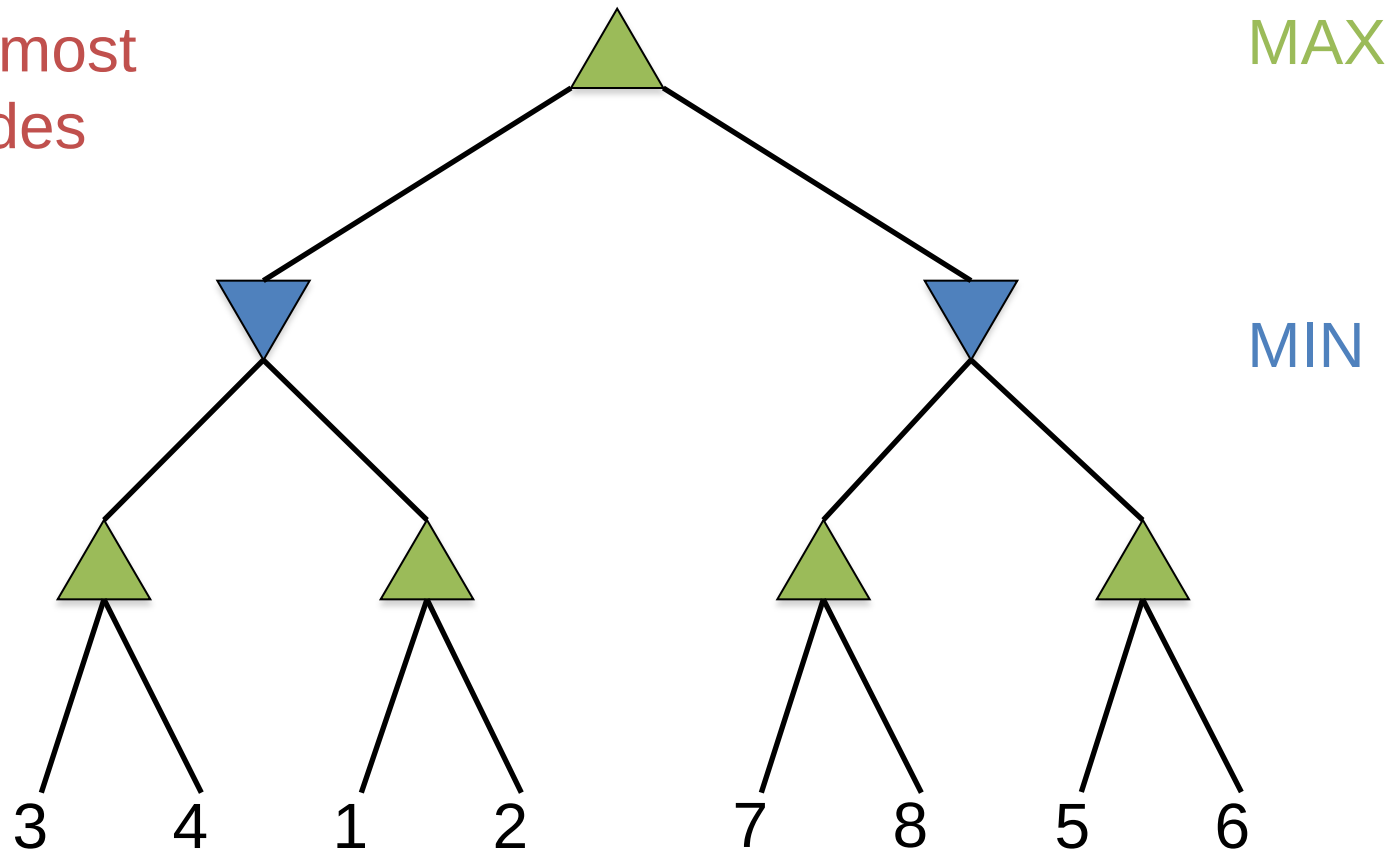
- Pruning does not affect final results
- Entire subtrees can be pruned
- Good move ordering improves pruning
 - Order nodes so player's best moves are checked first
- Repeated states are still possible
 - Store them in memory = transposition table

Iterative deepening reordering

Which leaves can be pruned?

None!

because the most
favorable nodes
are explored
last...

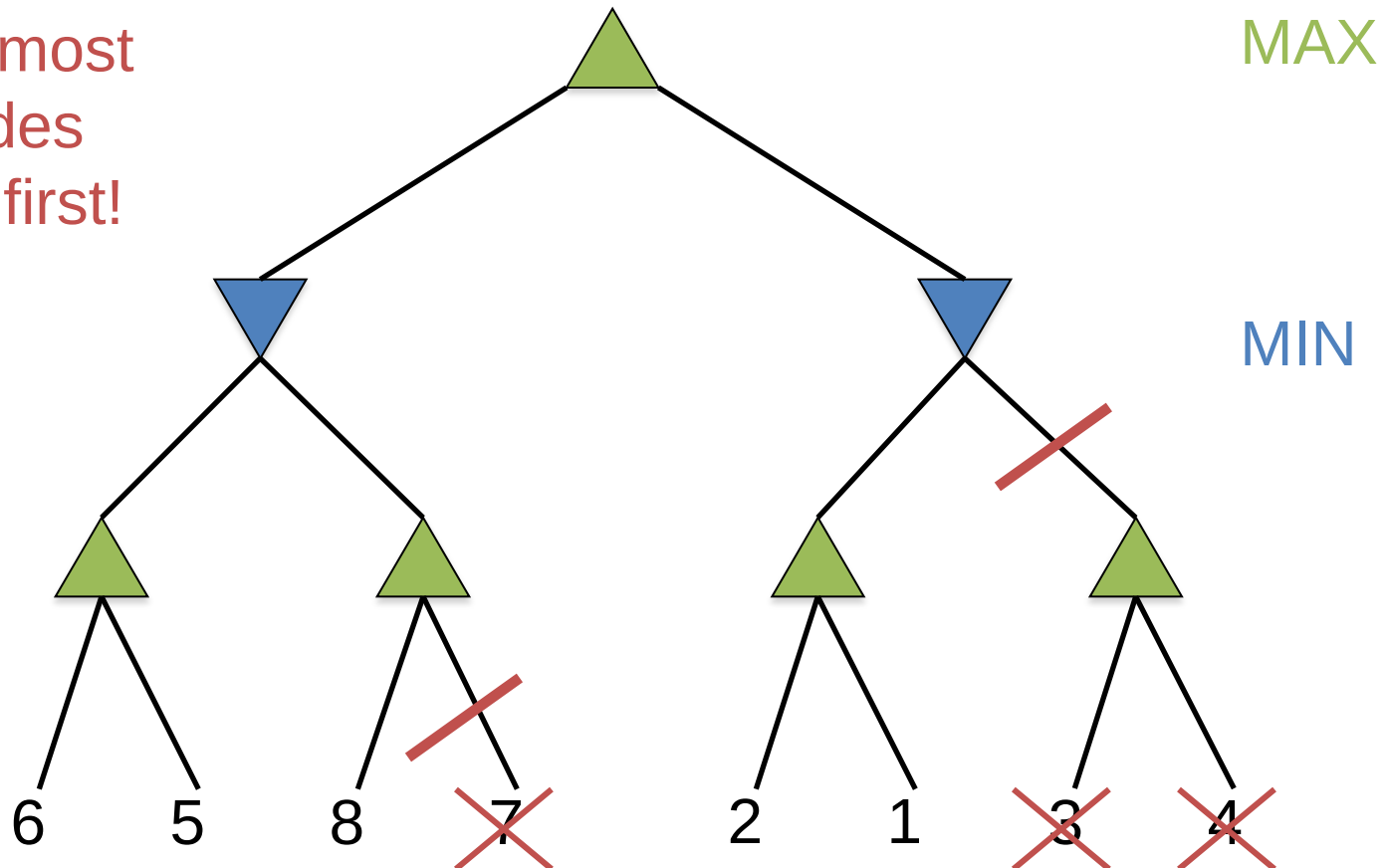


Iterative deepening reordering

Different exploration order: now which leaves can be pruned?

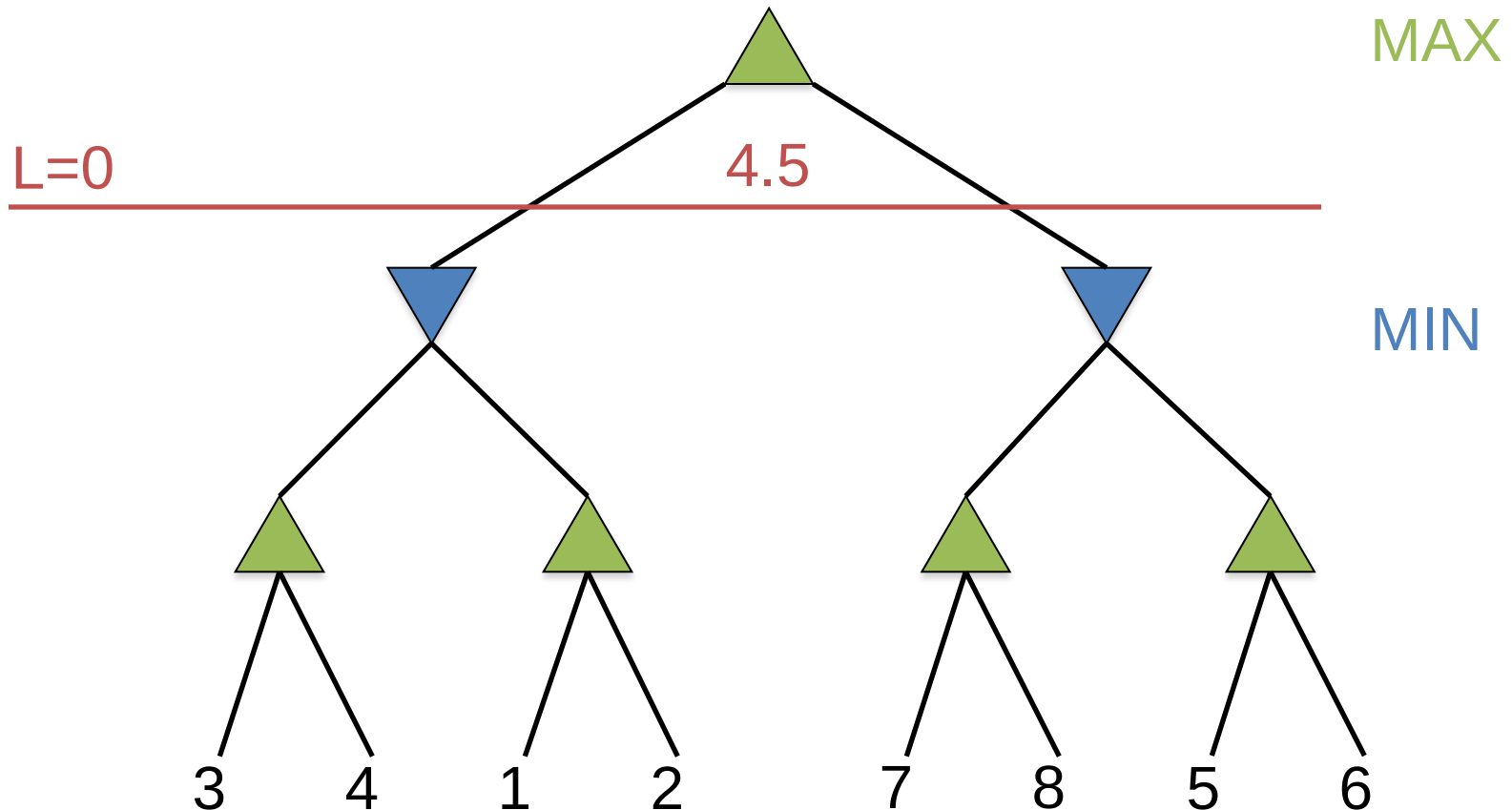
Lots!

because the most
favorable nodes
are explored first!



Iterative deepening reordering

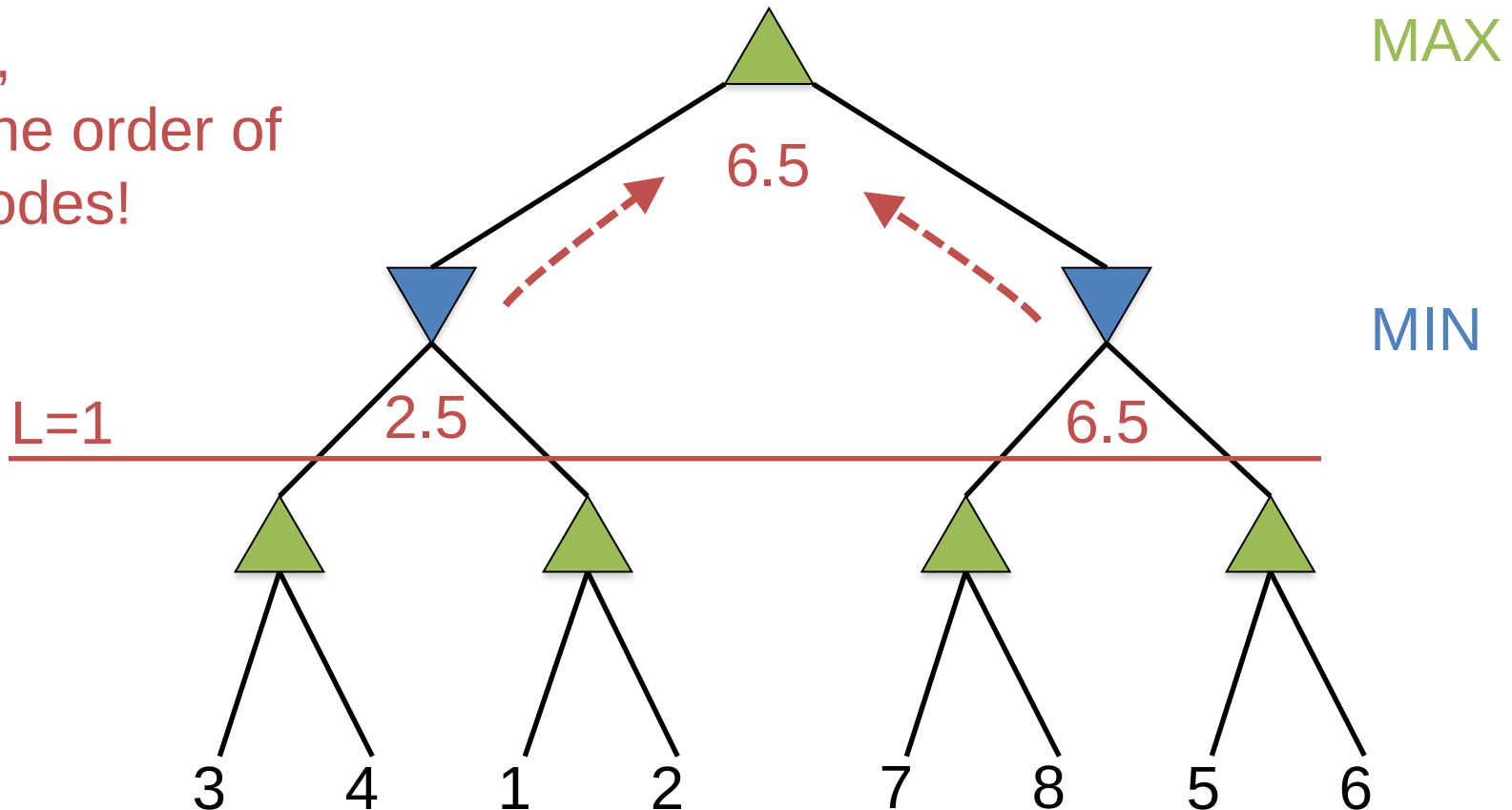
Order with no pruning; use iterative deepening approach.
Assume node score is the average of leaf values below.



Iterative deepening reordering

Order with no pruning; use iterative deepening approach.
Assume node score is the average of leaf values below.

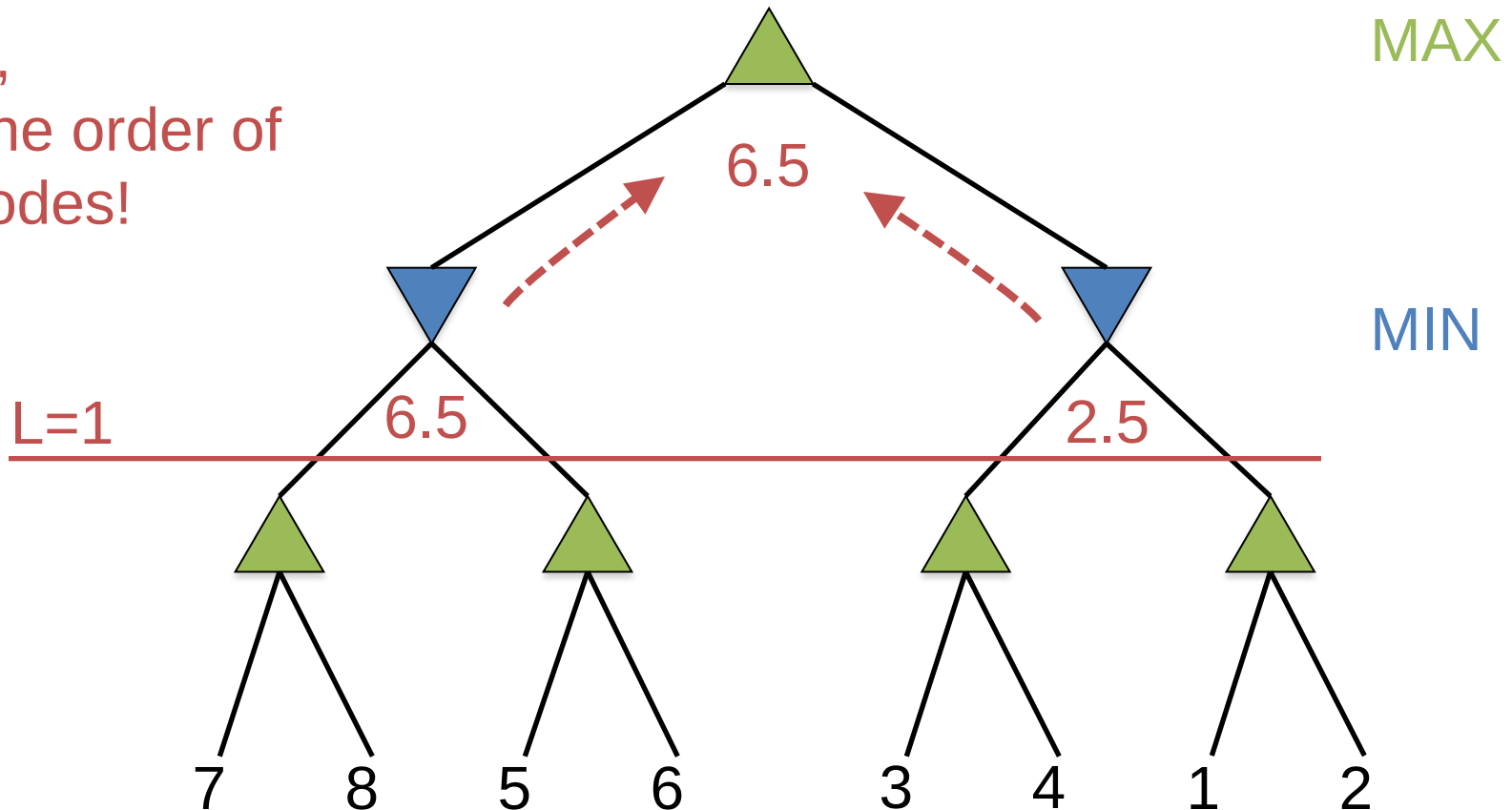
For $L=2$,
switch the order of
these nodes!



Iterative deepening reordering

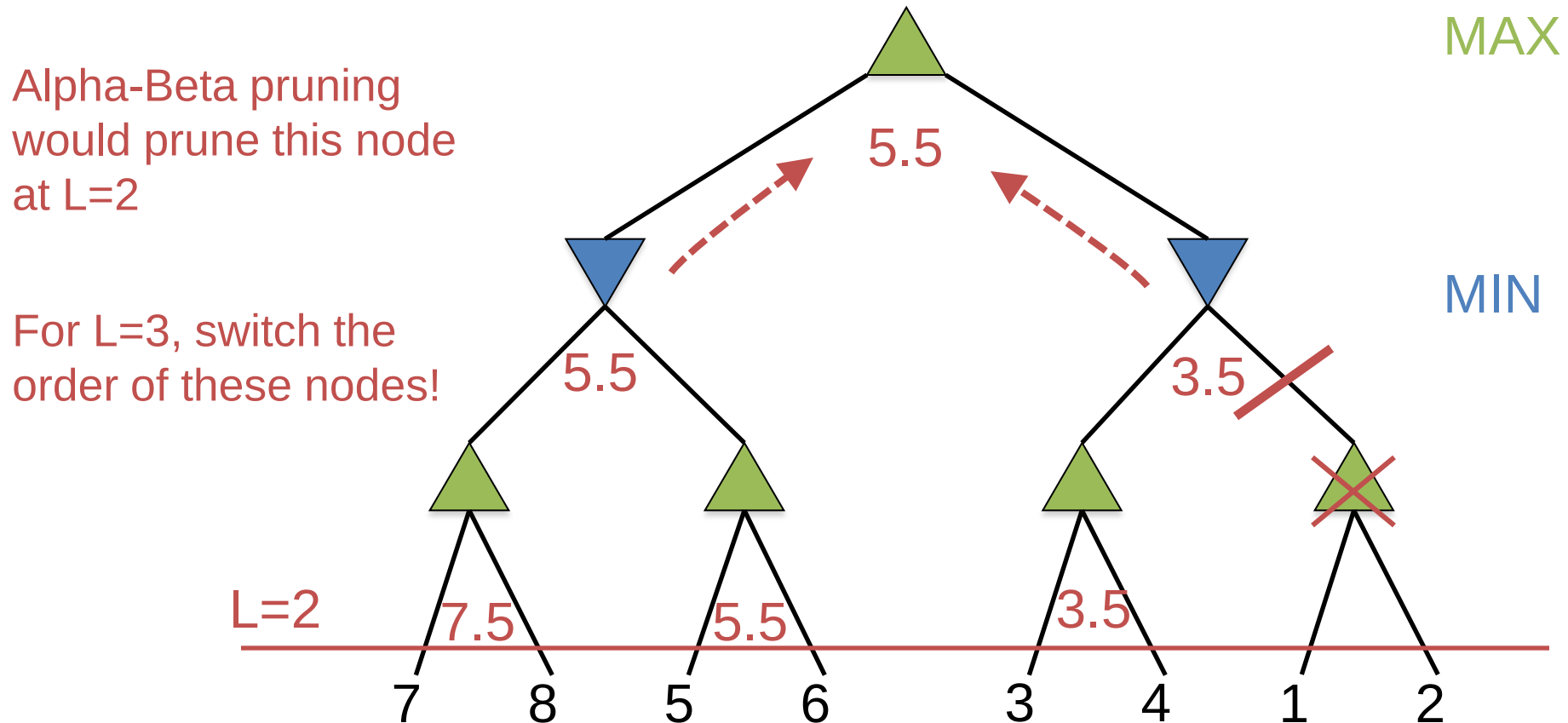
Order with no pruning; use iterative deepening approach.
Assume node score is the average of leaf values below.

For $L=2$,
switch the order of
these nodes!

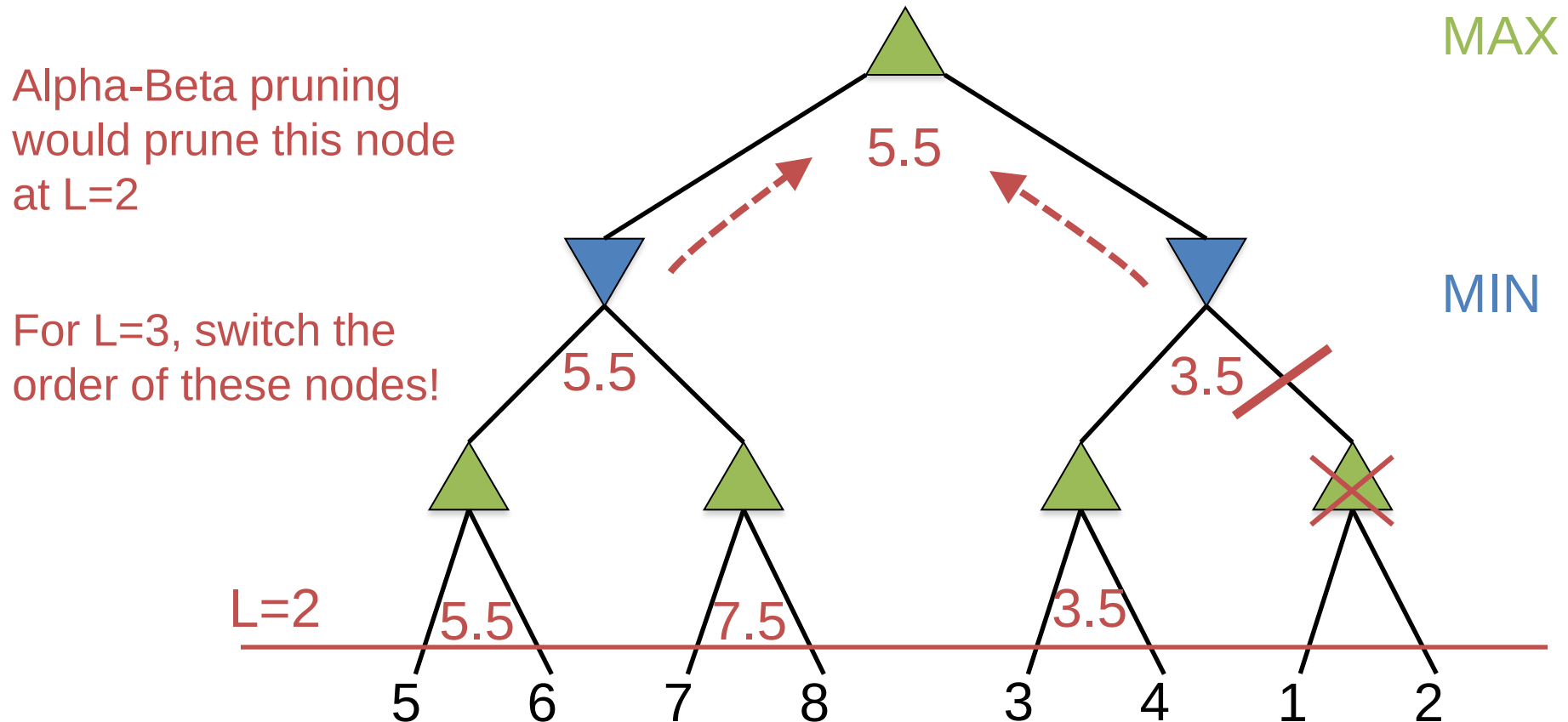


Iterative deepening reordering

Order with no pruning; use iterative deepening approach.
Assume node score is the average of leaf values below.



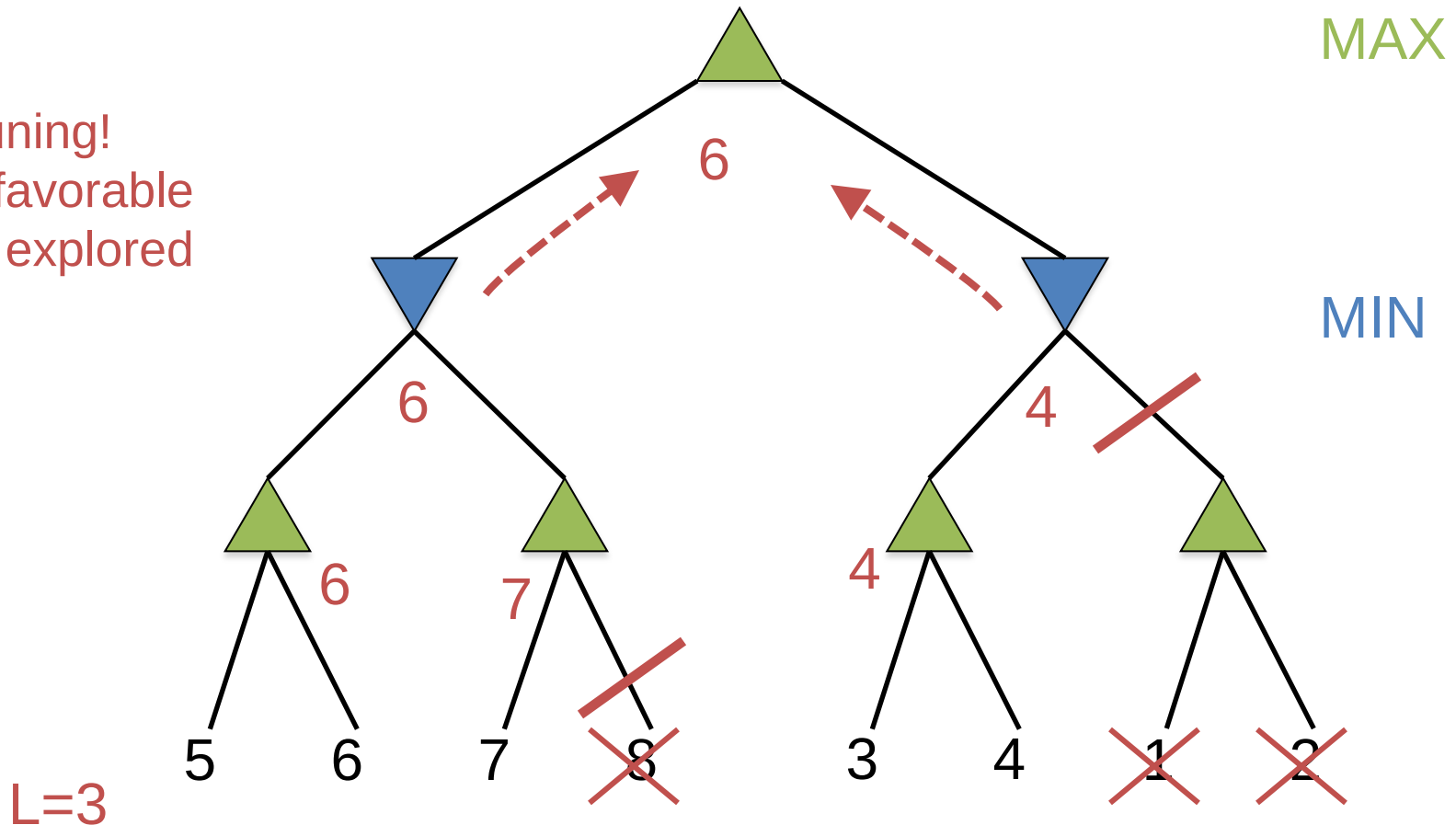
Order with no pruning; use iterative deepening approach.
Assume node score is the average of leaf values below.



Iterative deepening reordering

Order with no pruning; use iterative deepening approach.
Assume node score is the average of leaf values below.

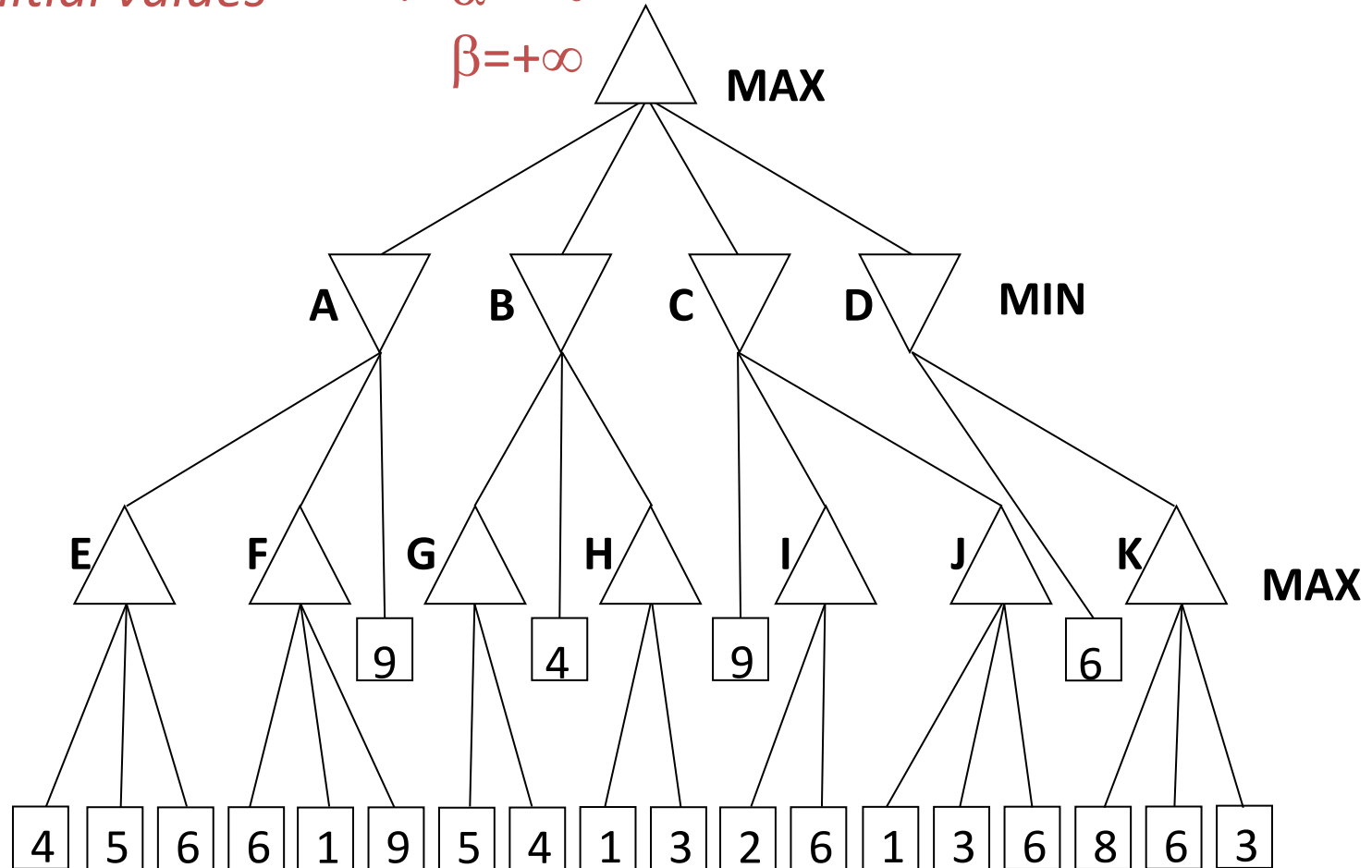
Lots of pruning!
The most favorable
nodes are explored
earlier.



Longer Alpha-Beta Example

Branch nodes are labeled A..K for easy discussion

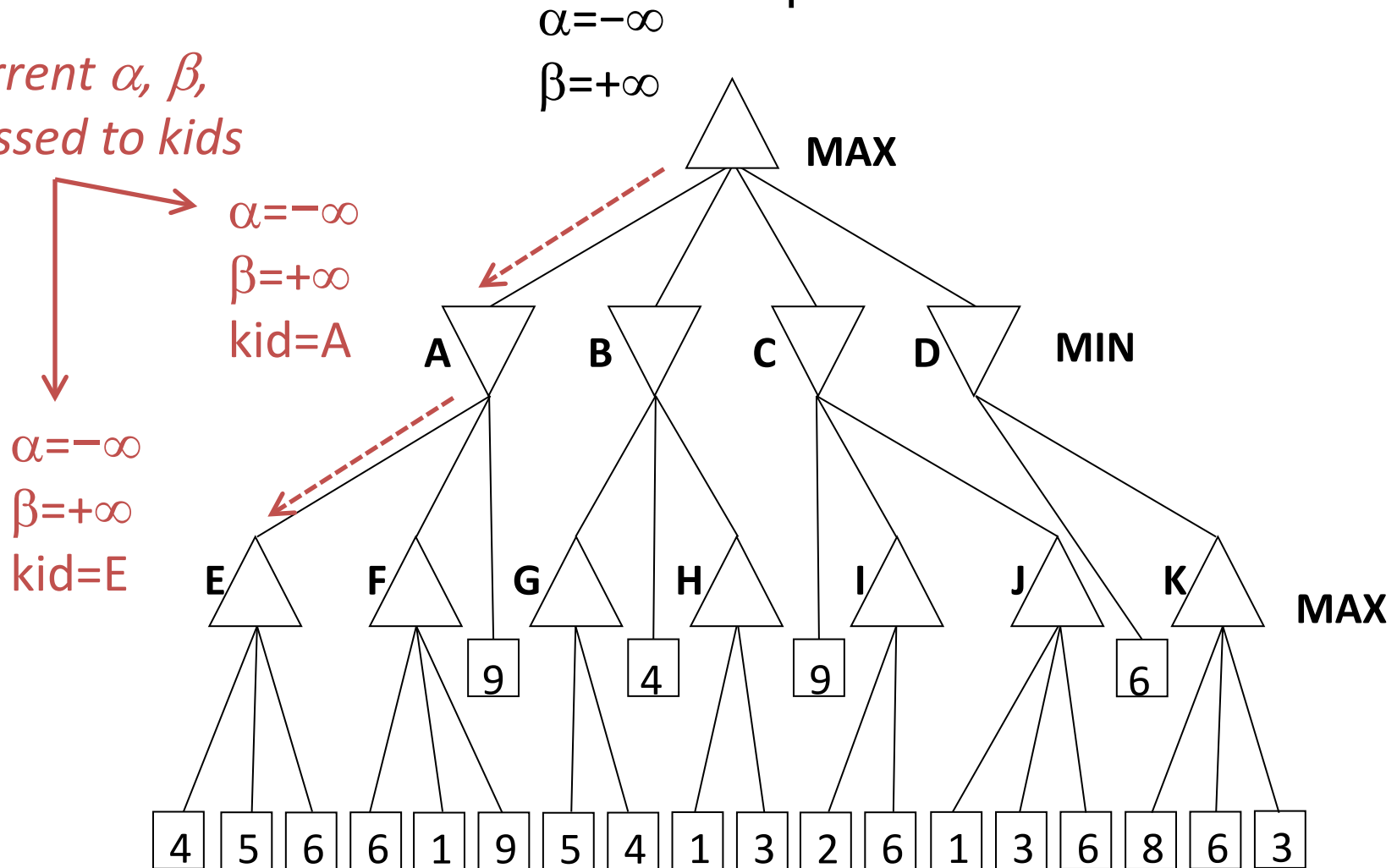
α, β , initial values $\longrightarrow \alpha = -\infty$
 $\beta = +\infty$



Longer Alpha-Beta Example

Note that cut-off occurs at different depths...

*current α , β ,
passed to kids*

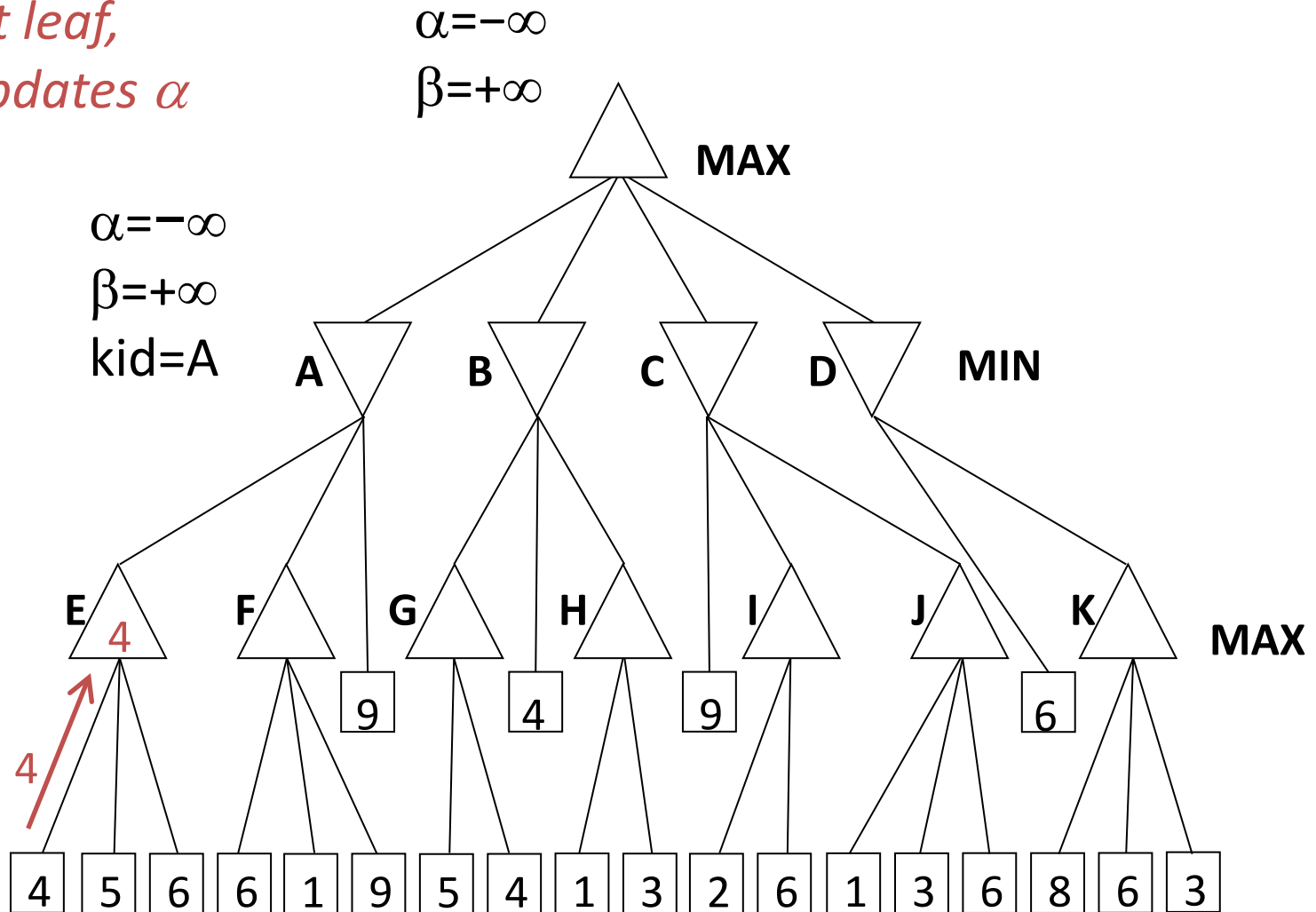


Longer Alpha-Beta Example

*see first leaf,
MAX updates α*



$\alpha=4$
 $\beta=+\infty$
kid=E

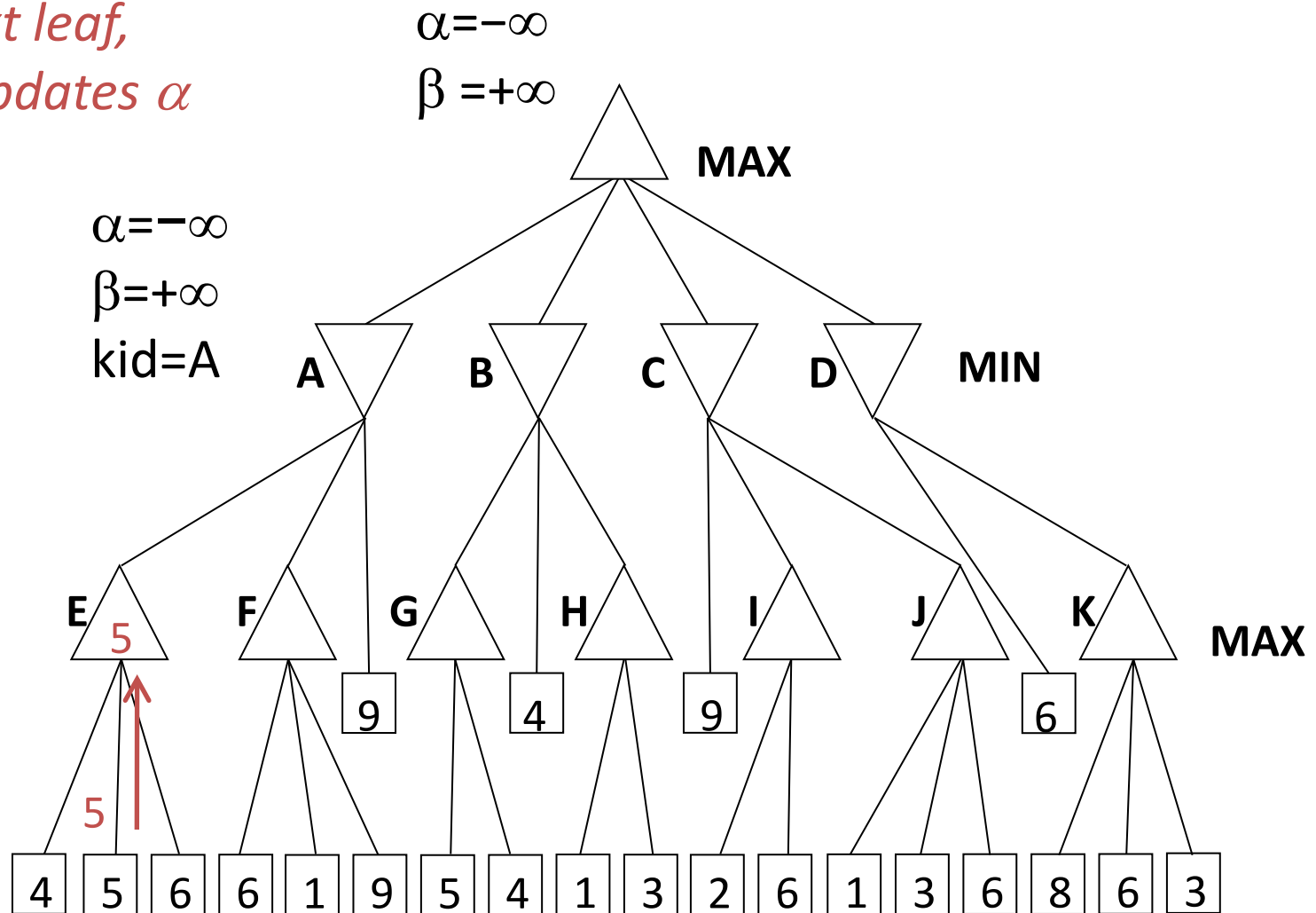


We also are running MiniMax search and recording node values within the triangles, without explicit comment.

Longer Alpha-Beta Example

*see next leaf,
MAX updates α*

$\alpha=5$
 $\beta=+\infty$
kid=E

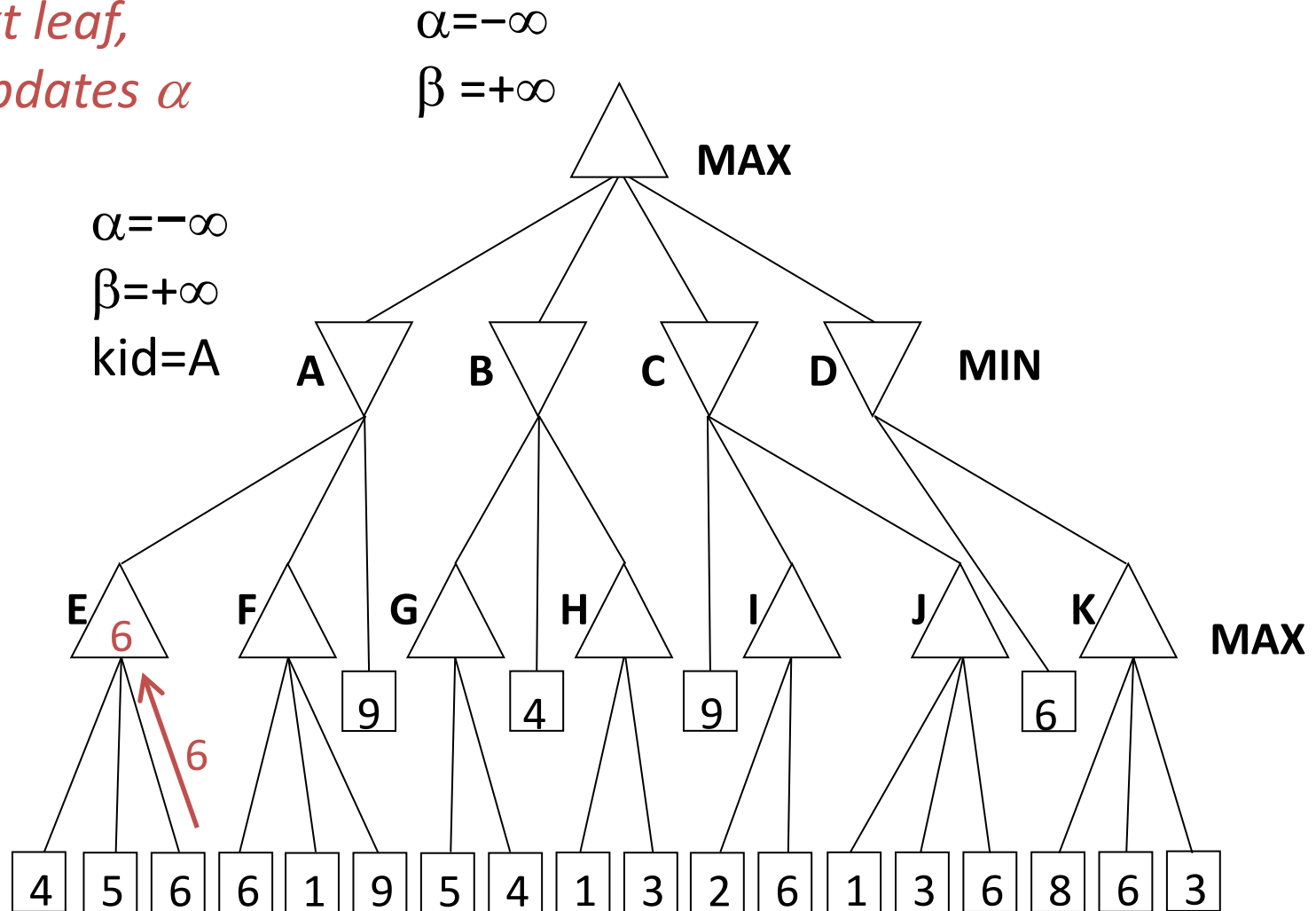


Longer Alpha-Beta Example

*see next leaf,
MAX updates α*

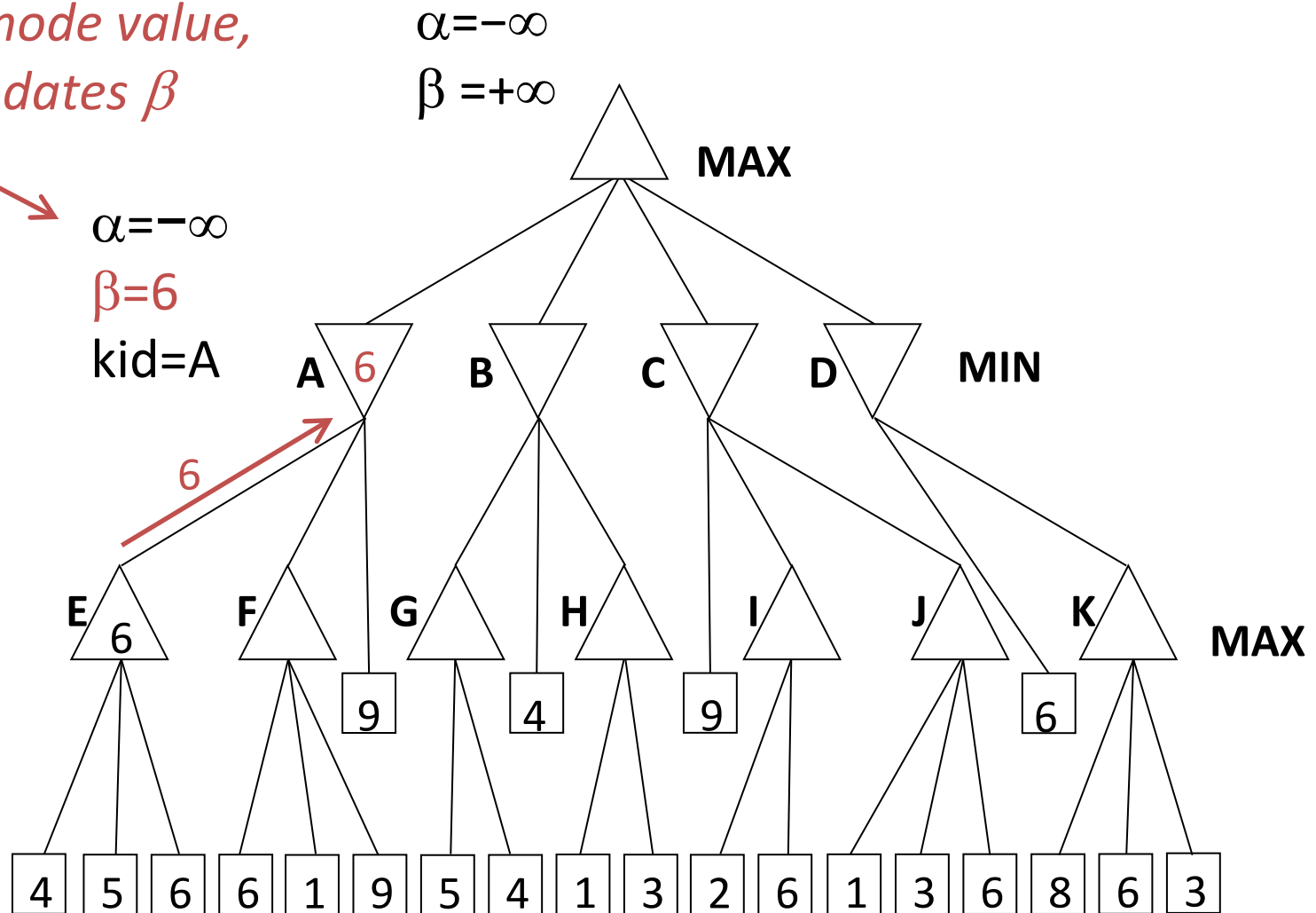


$\alpha=6$
 $\beta=+\infty$
kid=E



Longer Alpha-Beta Example

*return node value,
MIN updates β*

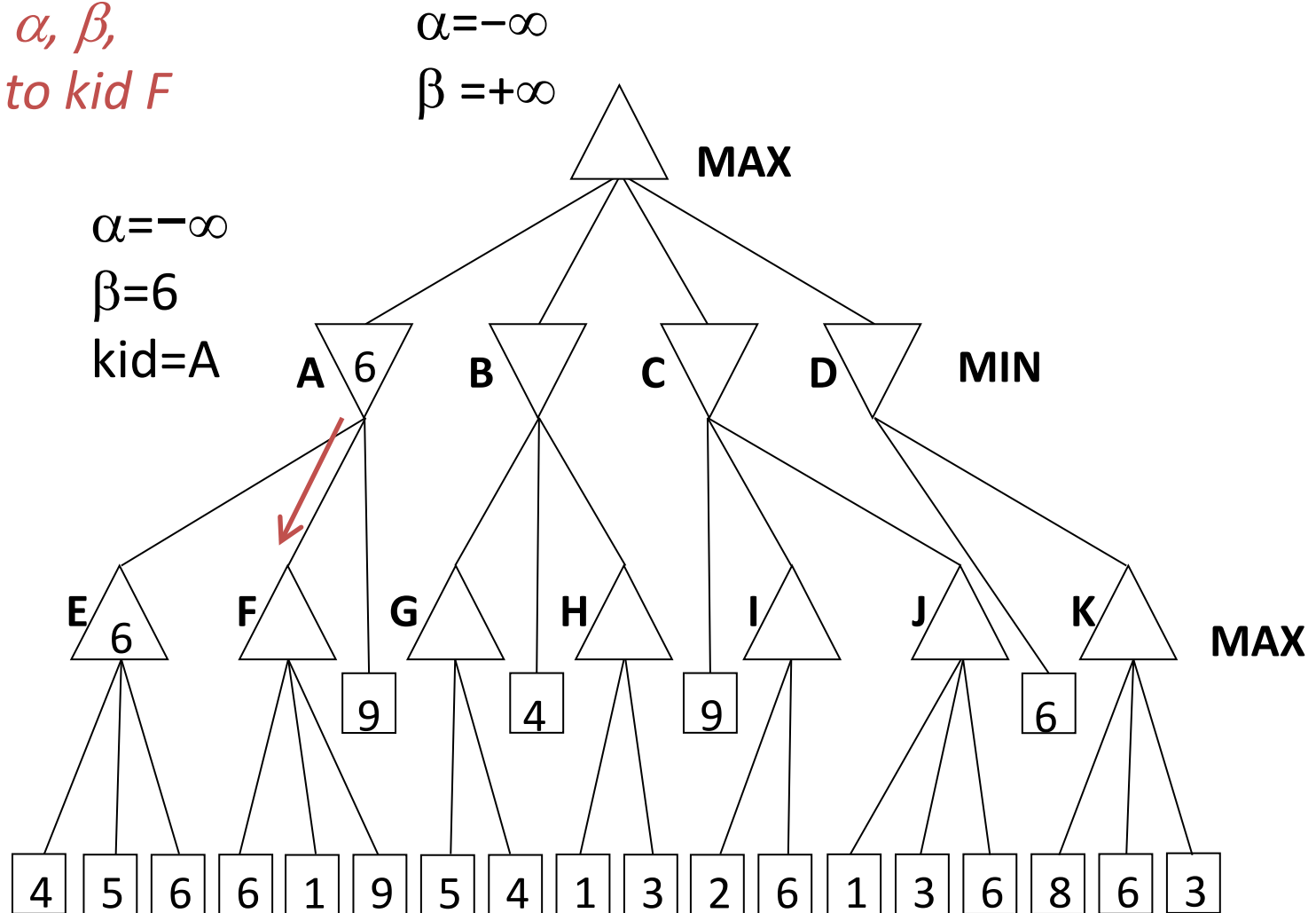


Longer Alpha-Beta Example

*current α , β ,
passed to kid F*



*$\alpha = -\infty$
 $\beta = 6$
kid = F*

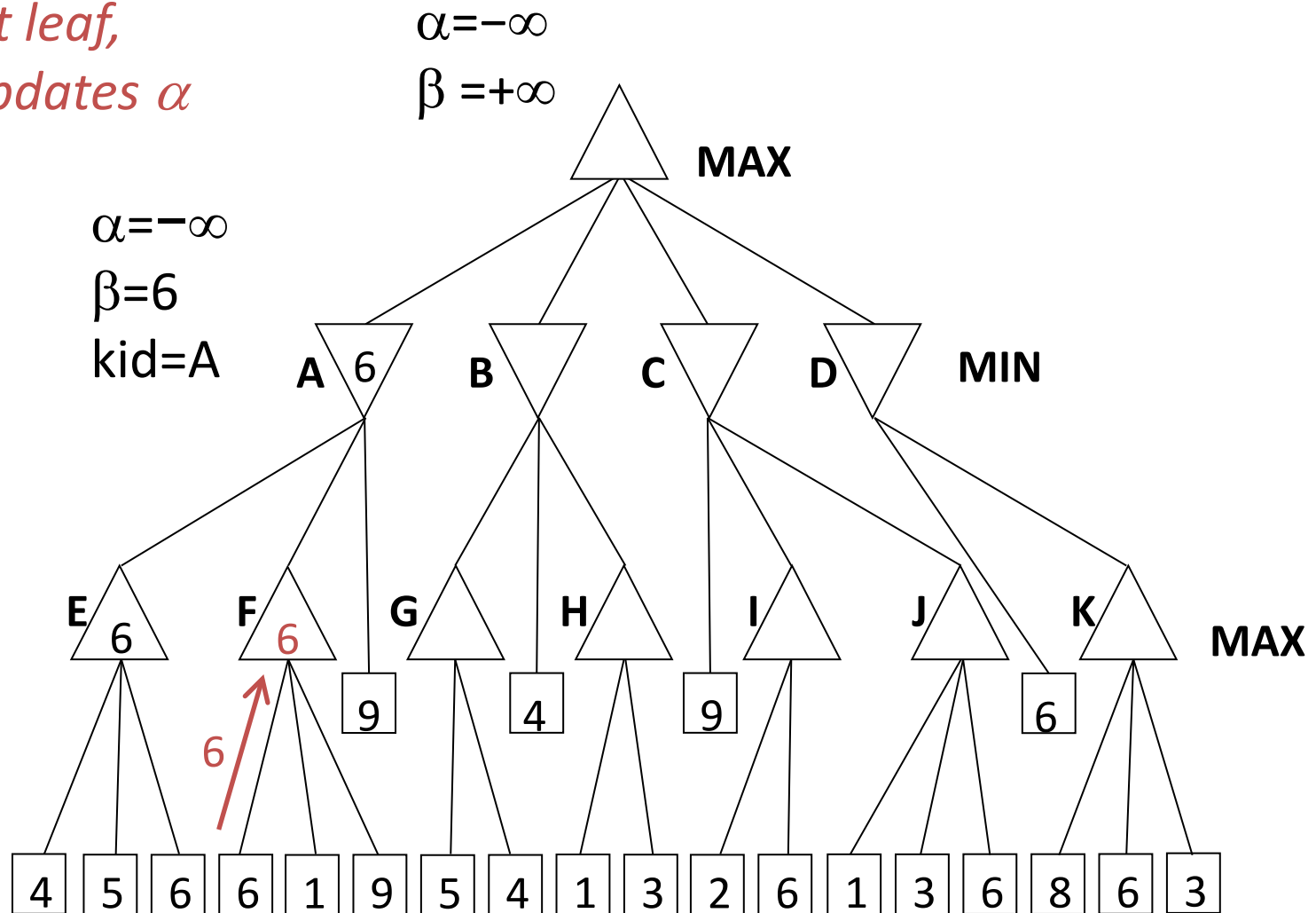


Longer Alpha-Beta Example

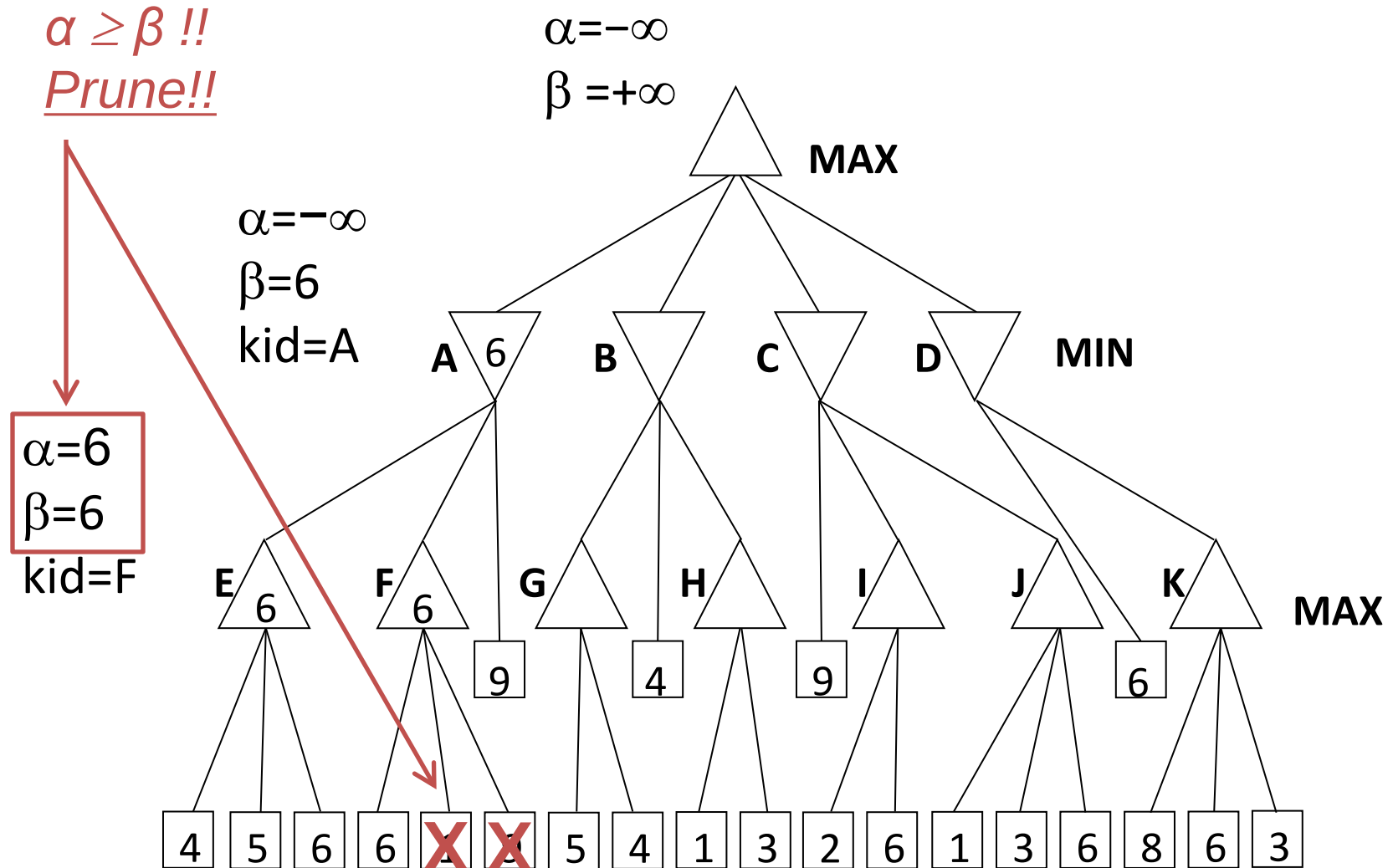
*see first leaf,
MAX updates α*



$\alpha=6$
 $\beta=6$
kid=F

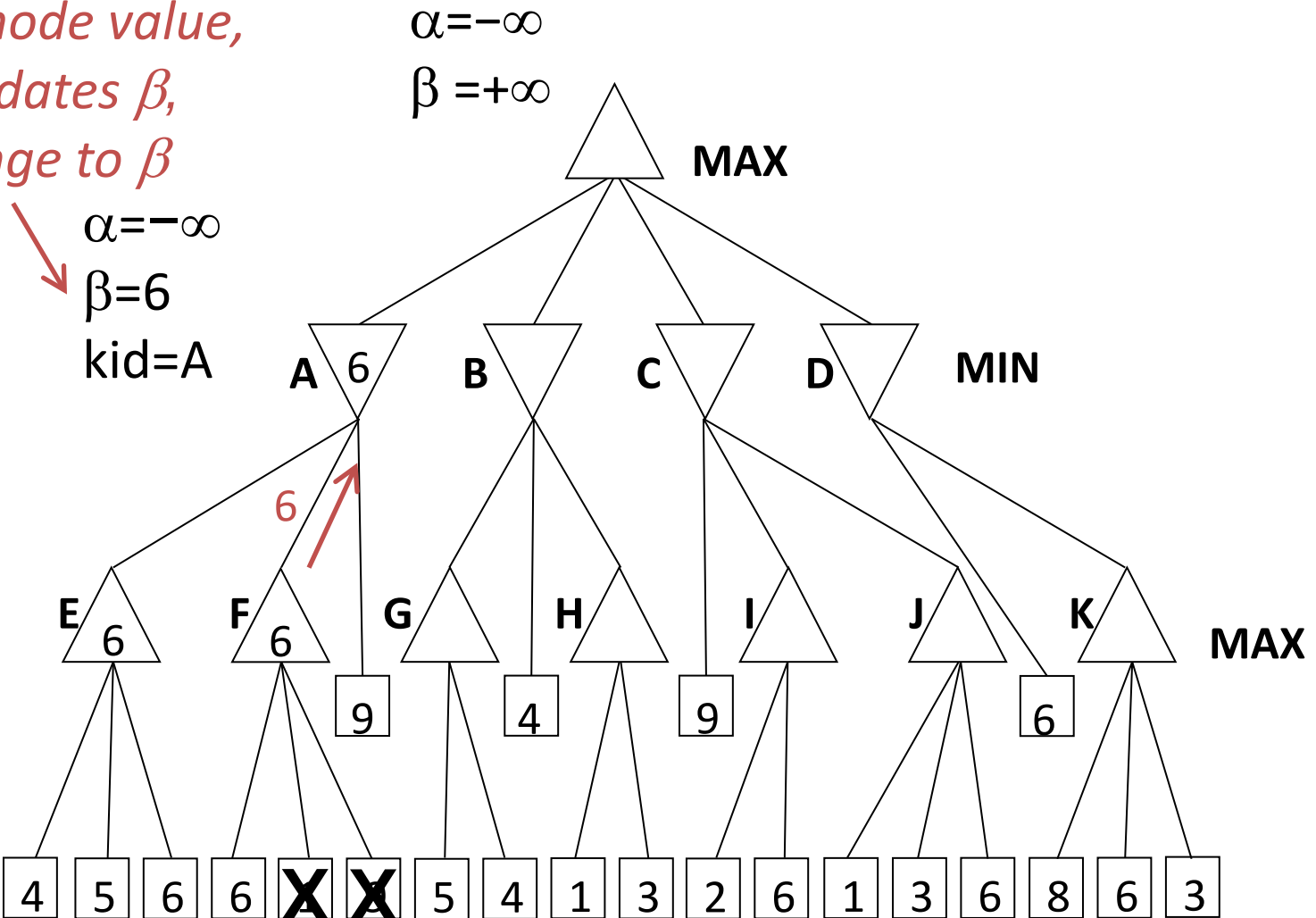


Longer Alpha-Beta Example



Longer Alpha-Beta Example

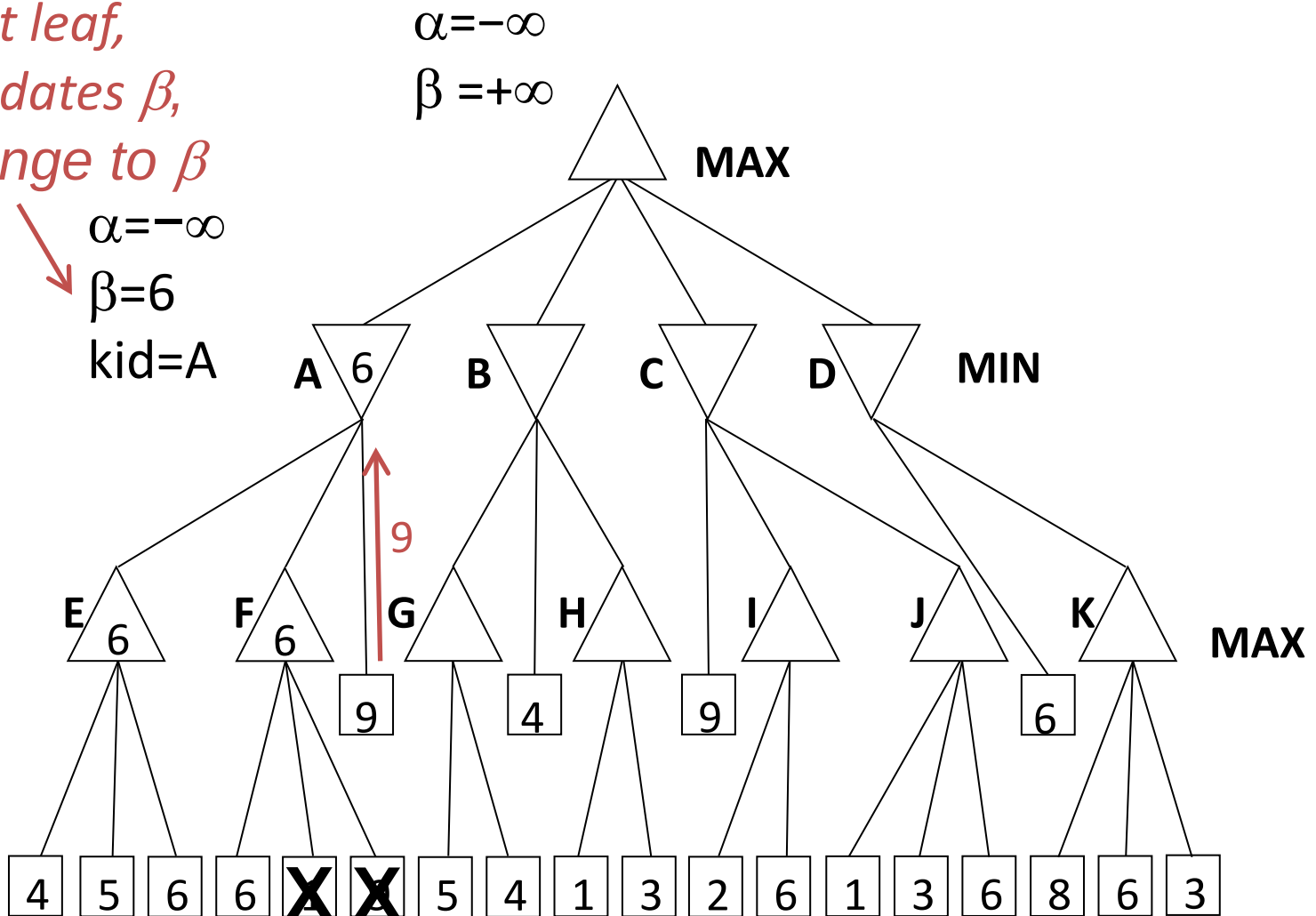
*return node value,
MIN updates β ,
no change to β*



If we had continued searching at node F, we would see the 9 from its third leaf. Our returned value would be 9 instead of 6. But at A, MIN would choose E(=6) instead of F(=9). Internal values may change; root values do not.

Longer Alpha-Beta Example

*see next leaf,
MIN updates β ,
no change to β*

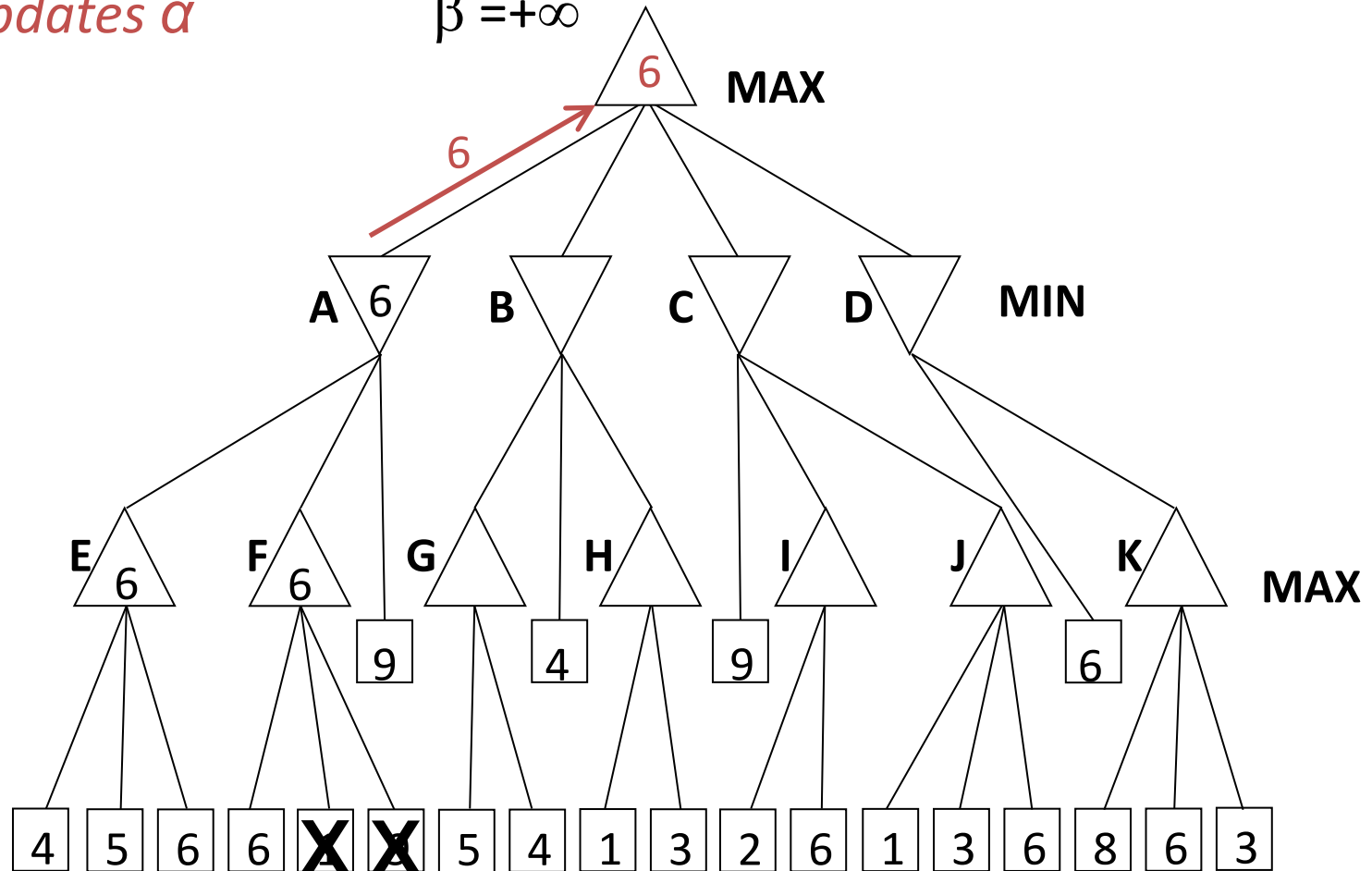


Longer Alpha-Beta Example

return node value, $\rightarrow \alpha=6$

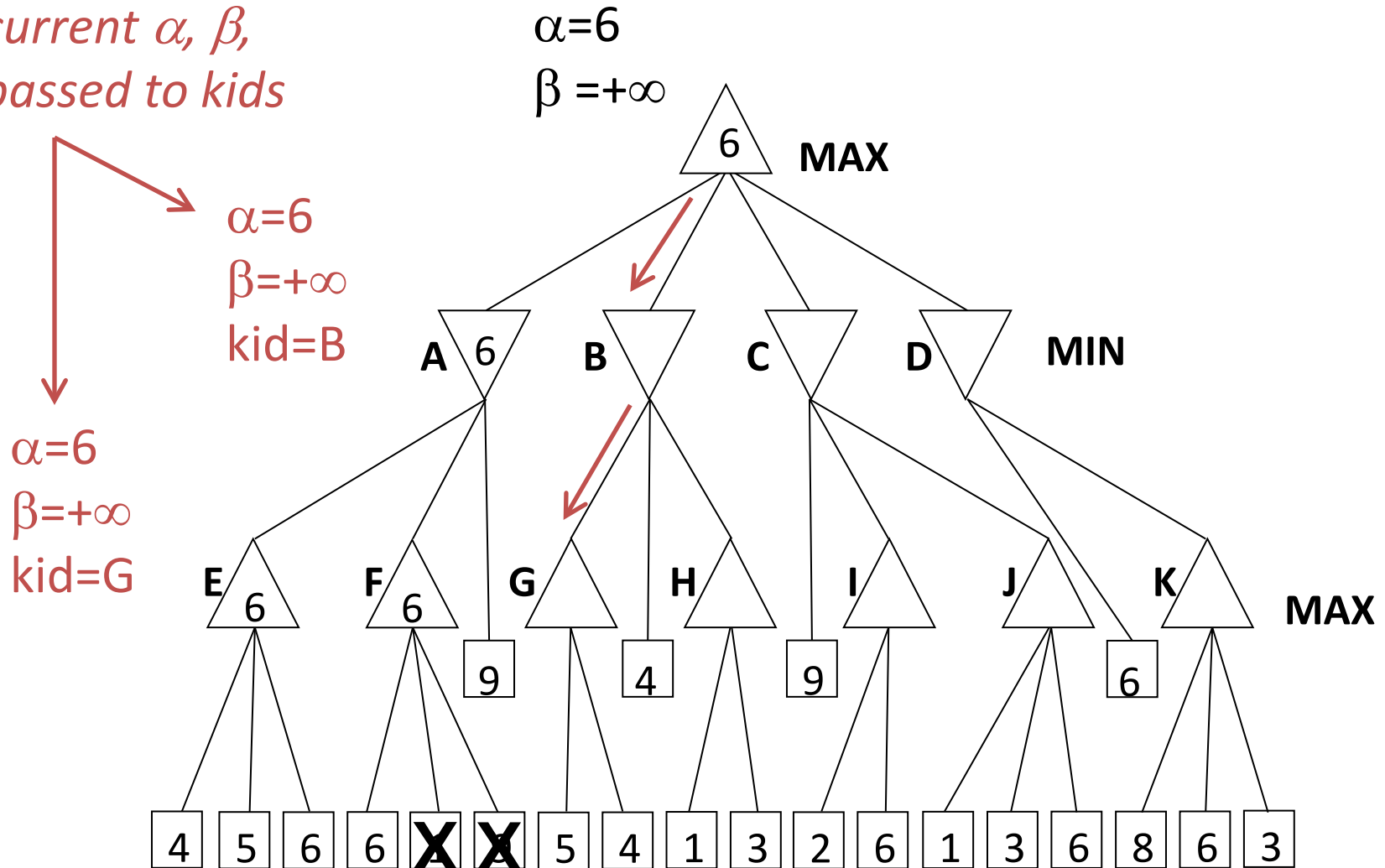
MAX updates α

$\beta = +\infty$

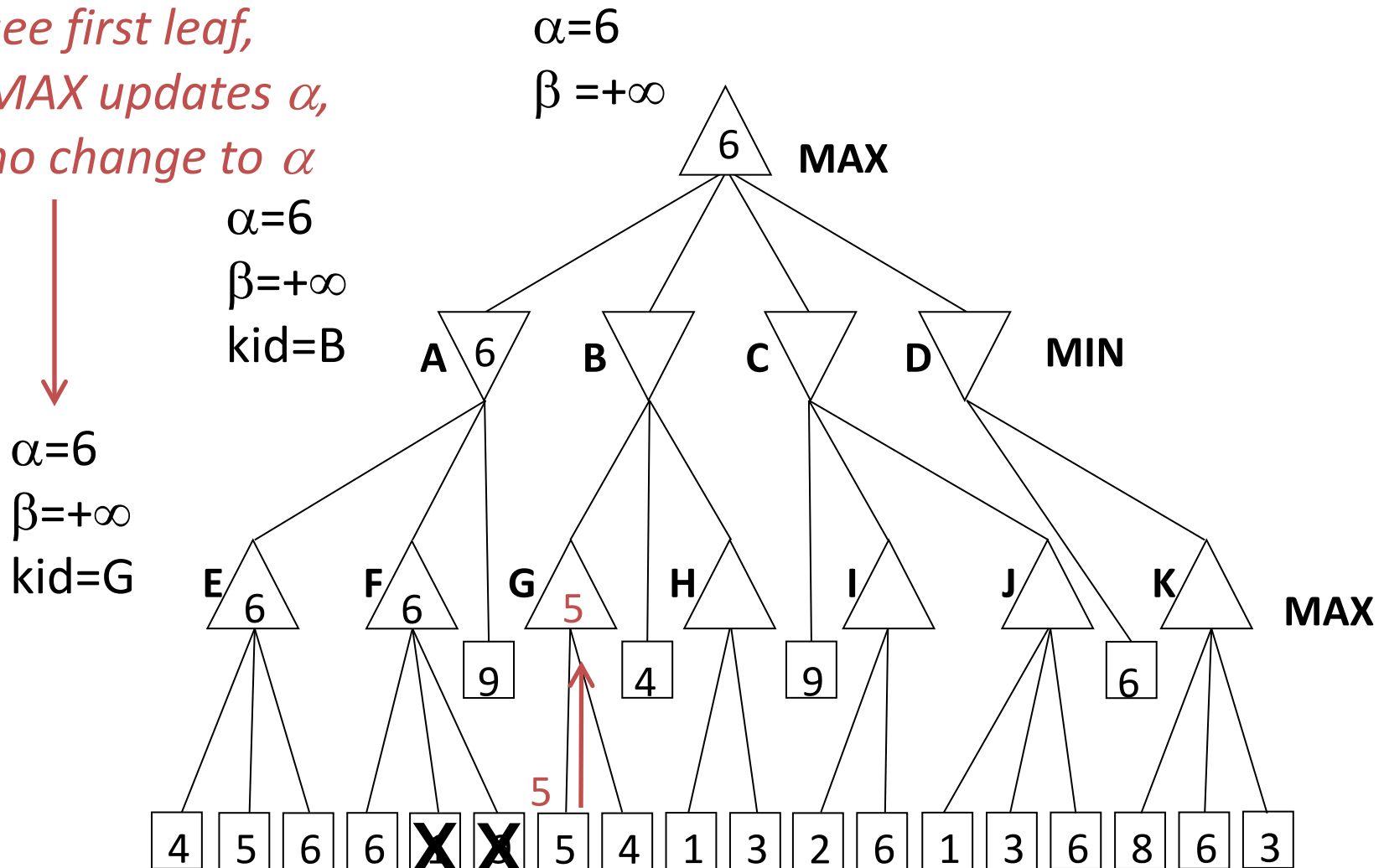


Longer Alpha-Beta Example

*current α , β ,
passed to kids*



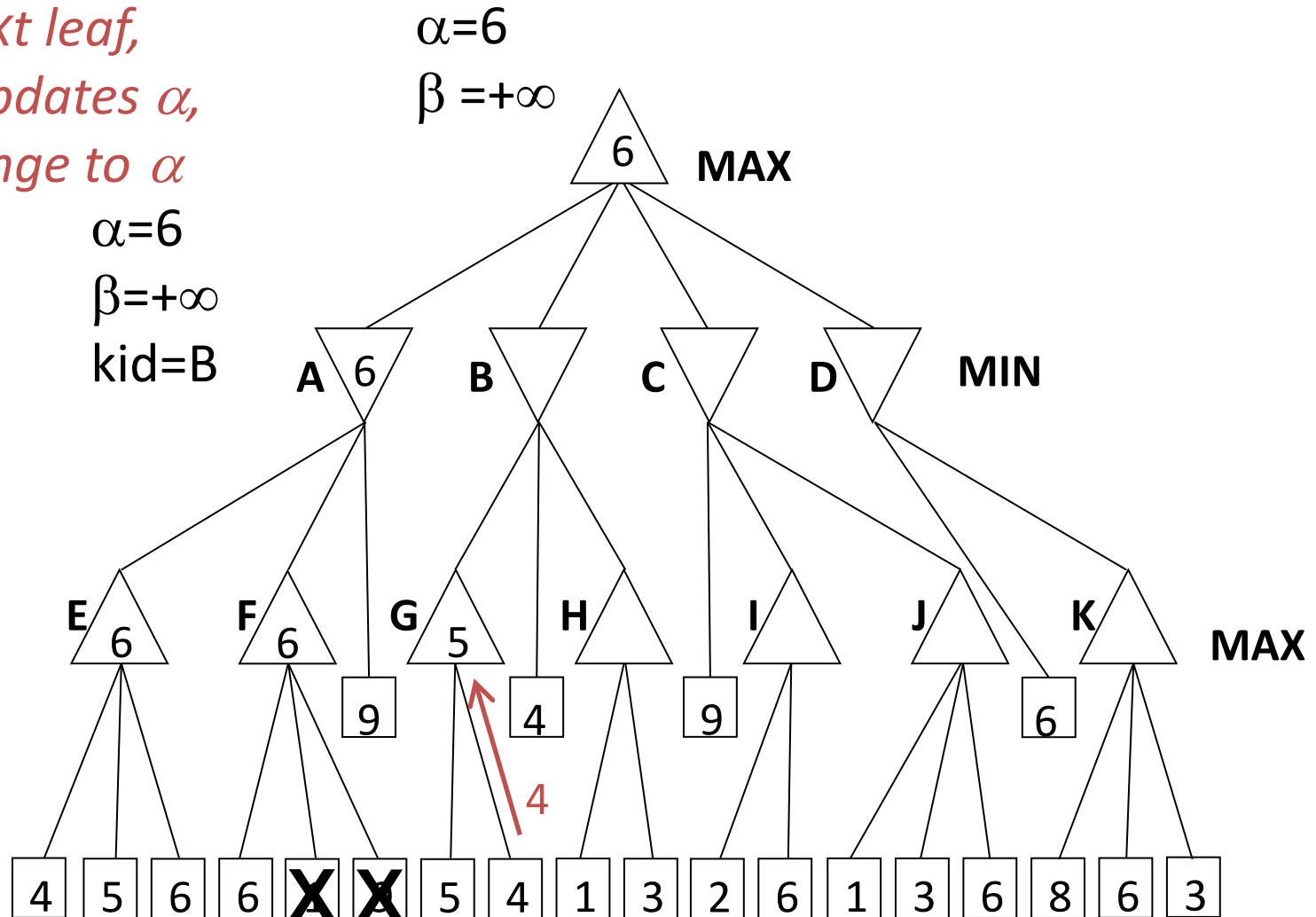
see first leaf,
MAX updates α ,
no change to α



Longer Alpha-Beta Example

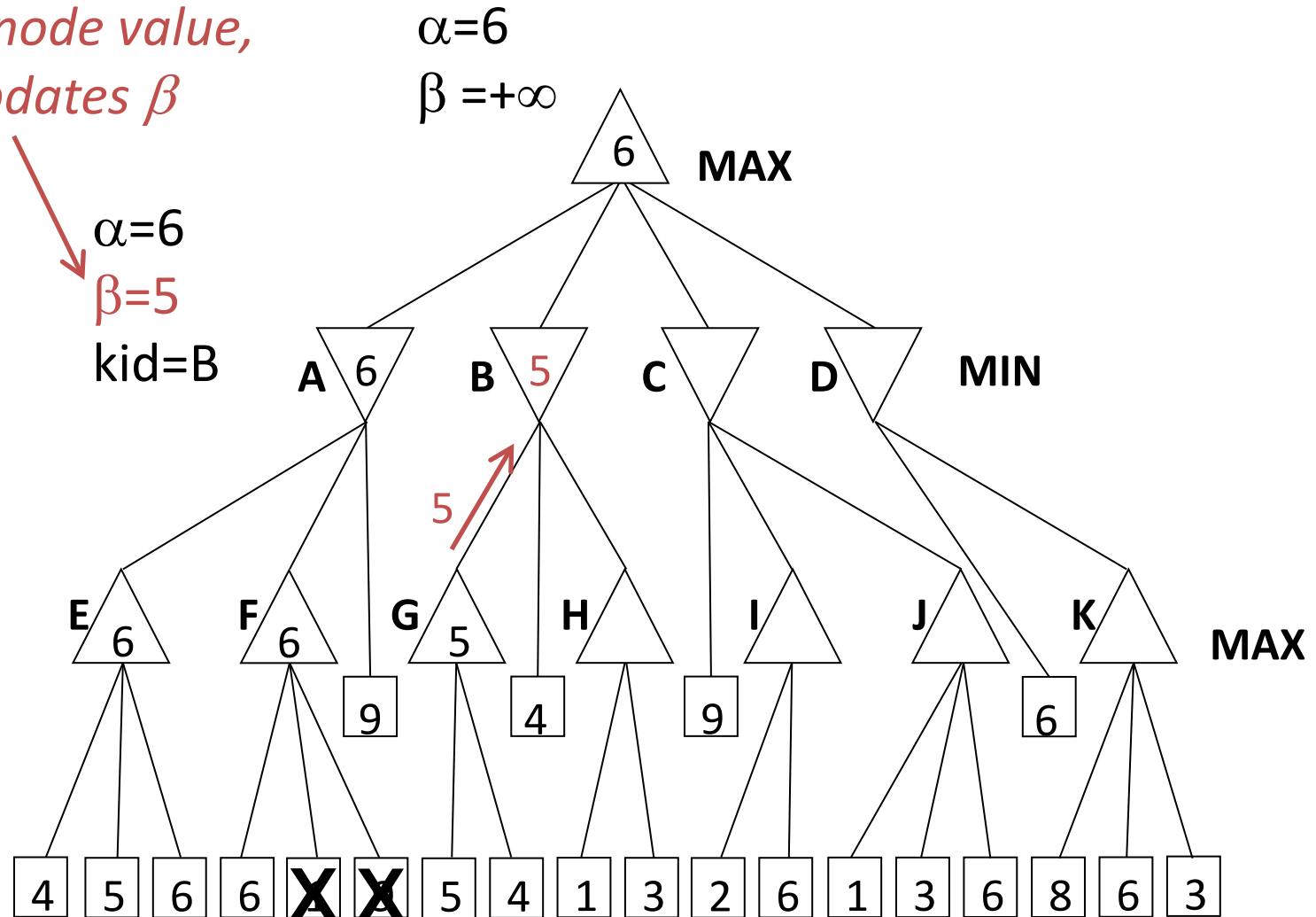
*see next leaf,
MAX updates α ,
no change to α*

$\alpha=6$
 $\beta=+\infty$
kid=G



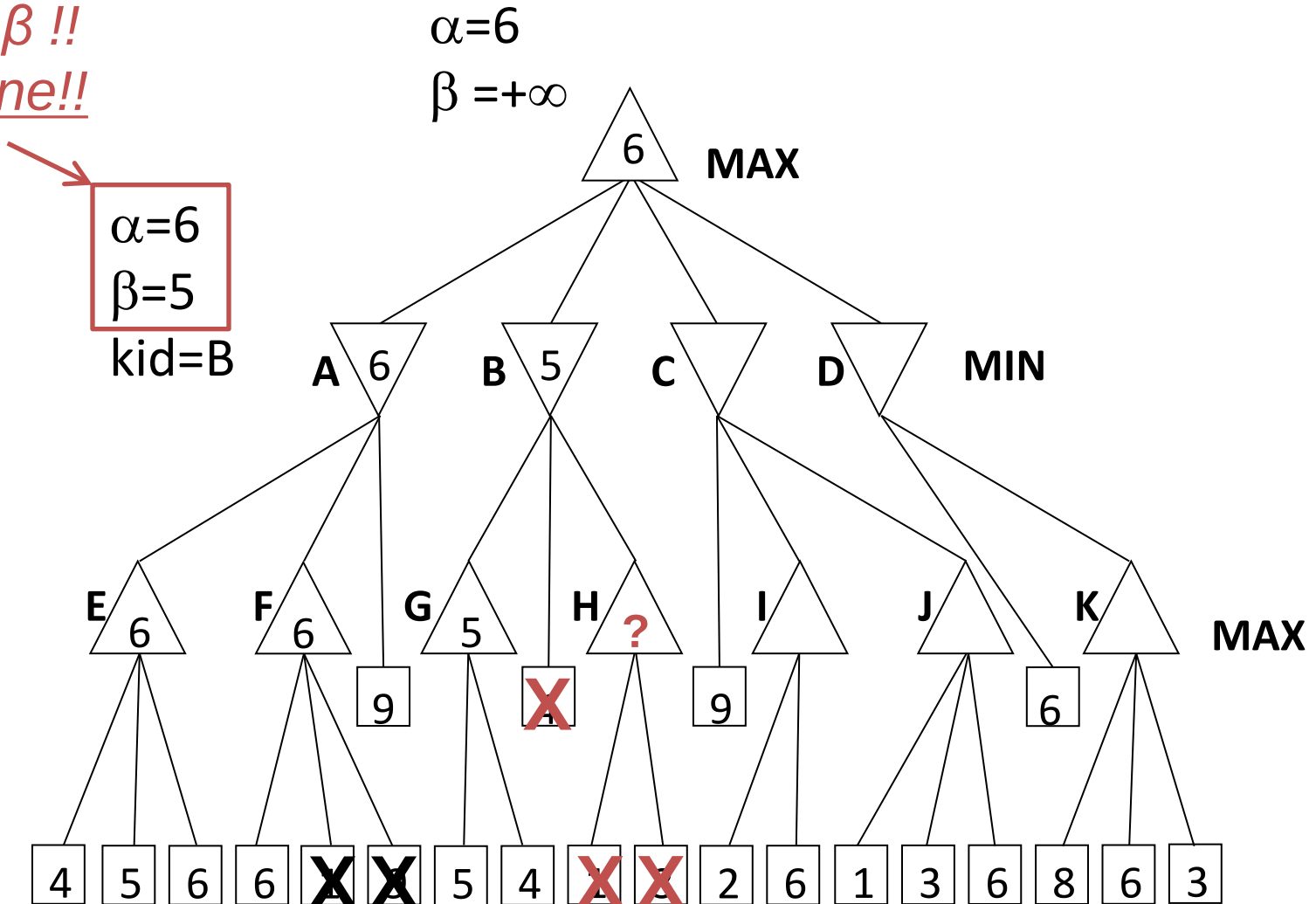
Longer Alpha-Beta Example

*return node value,
MIN updates β*



Longer Alpha-Beta Example

$\alpha \geq \beta$!!
Prune!!



Note that we never find out, what is the node value of H? But we have proven it doesn't matter, so we don't care.

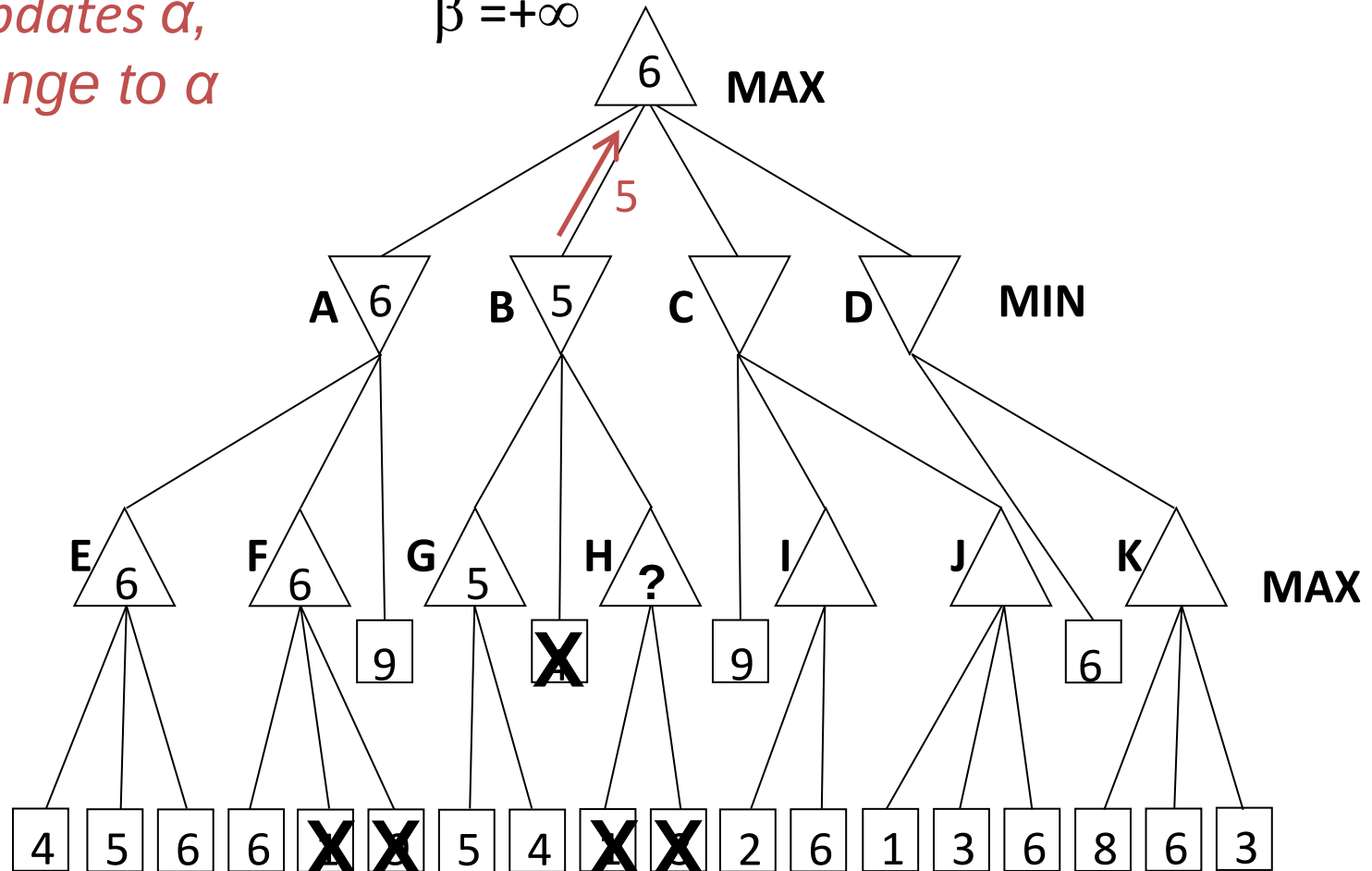
Longer Alpha-Beta Example

return node value, $\longrightarrow \alpha=6$

MAX updates α ,

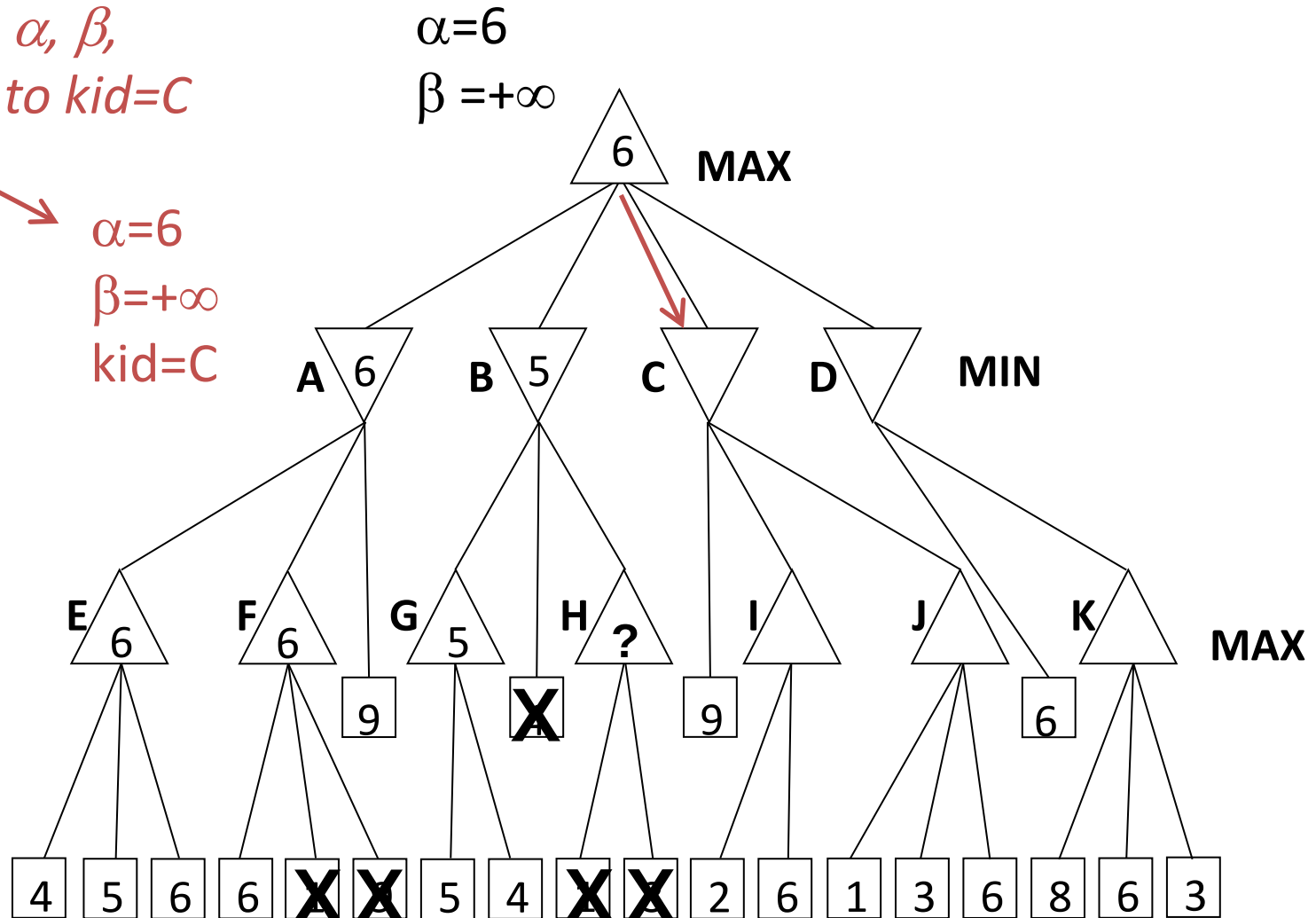
no change to α

$\beta = +\infty$



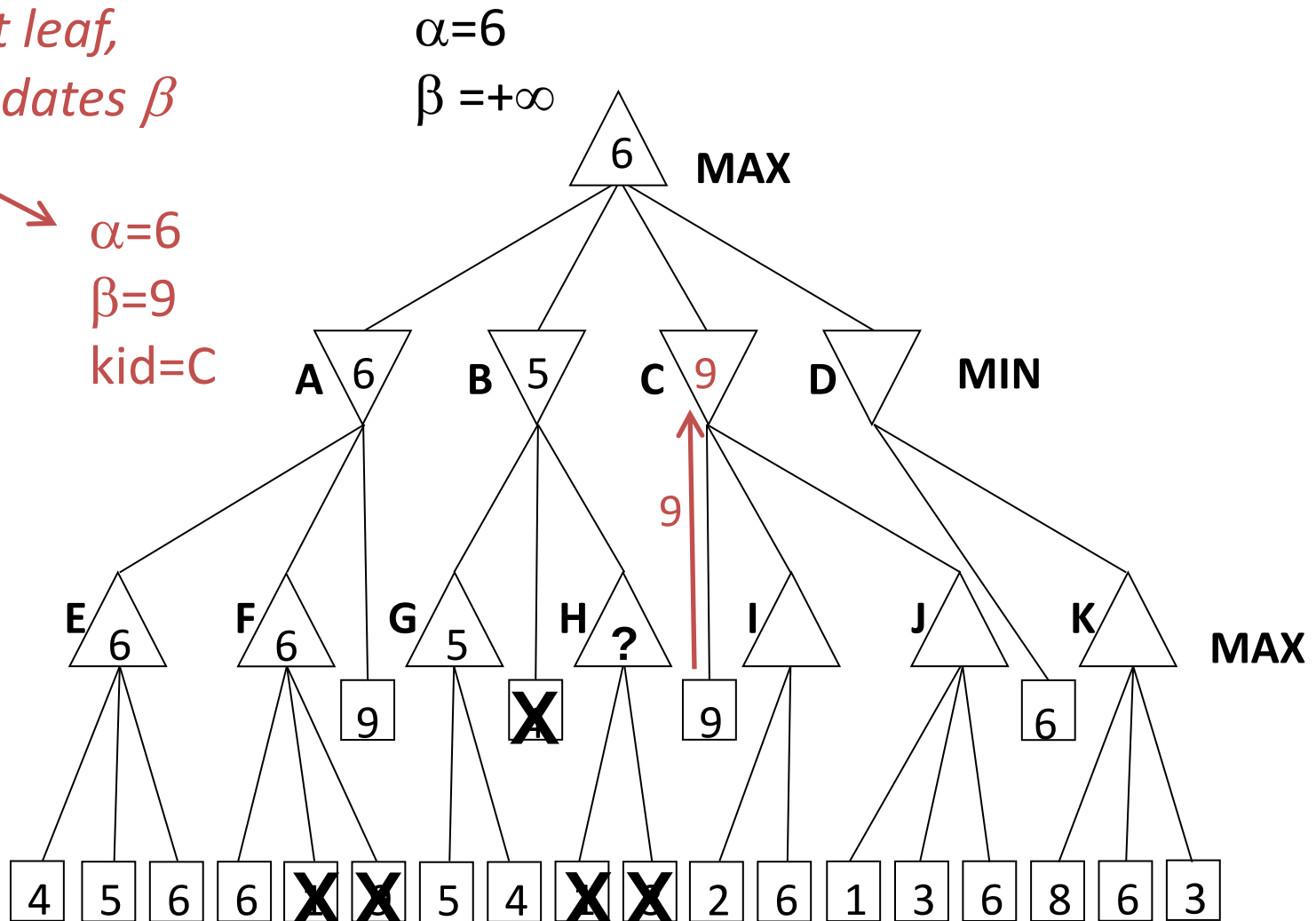
Longer Alpha-Beta Example

*current α , β ,
passed to kid=C*



Longer Alpha-Beta Example

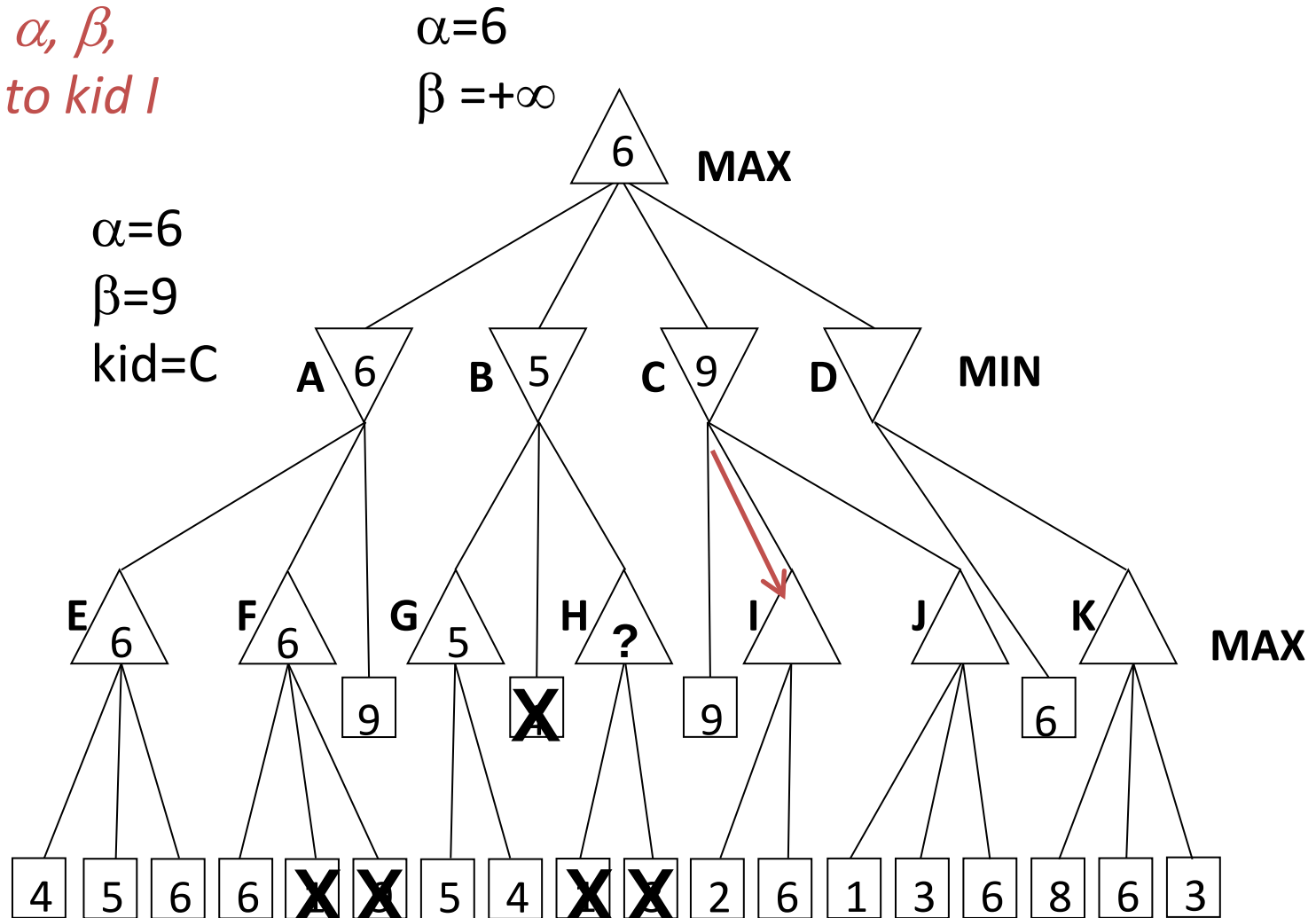
*see first leaf,
MIN updates β*

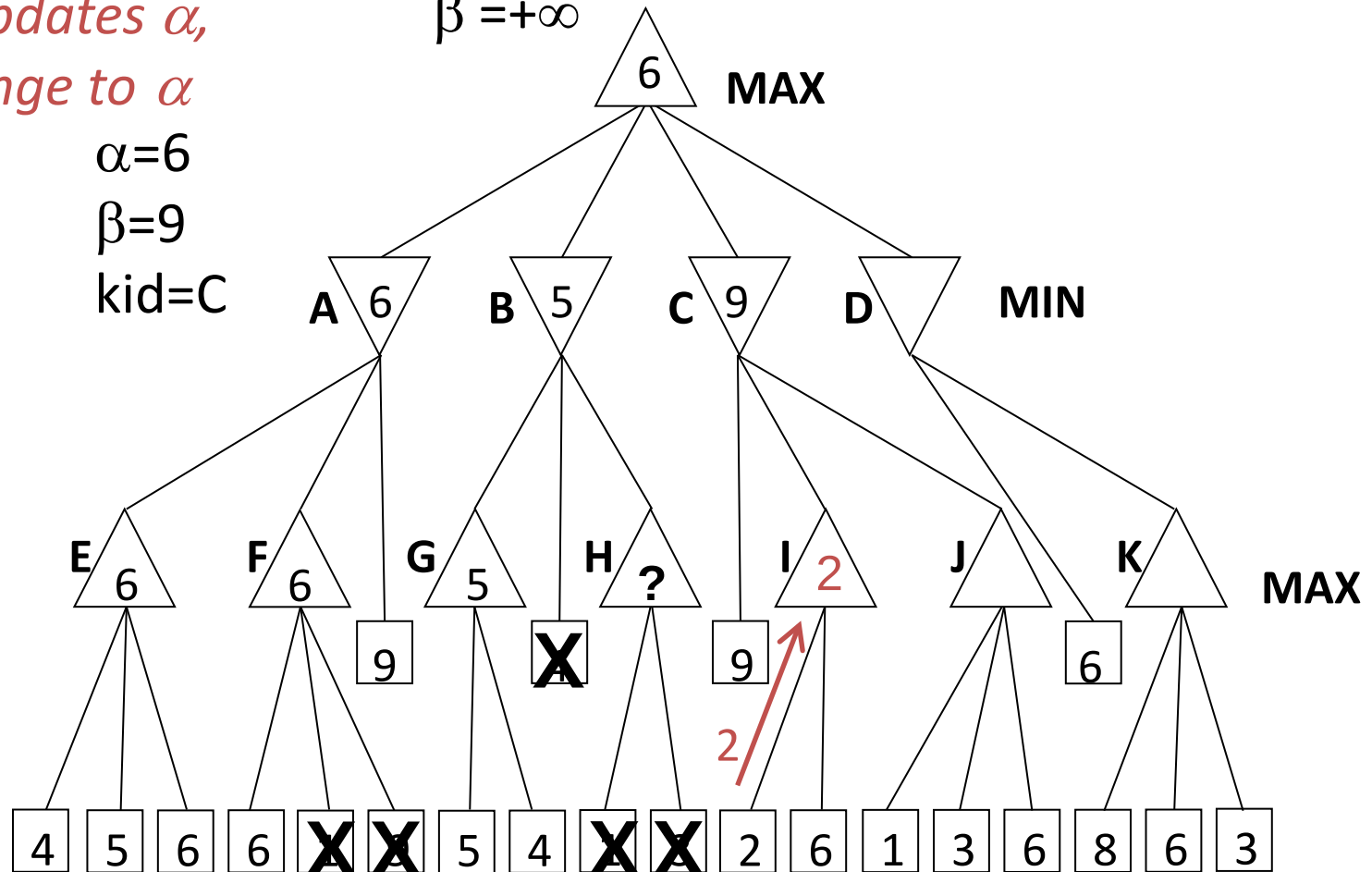


Longer Alpha-Beta Example

*current α , β ,
passed to kid I*

$\alpha=6$
 $\beta=9$
kid=I

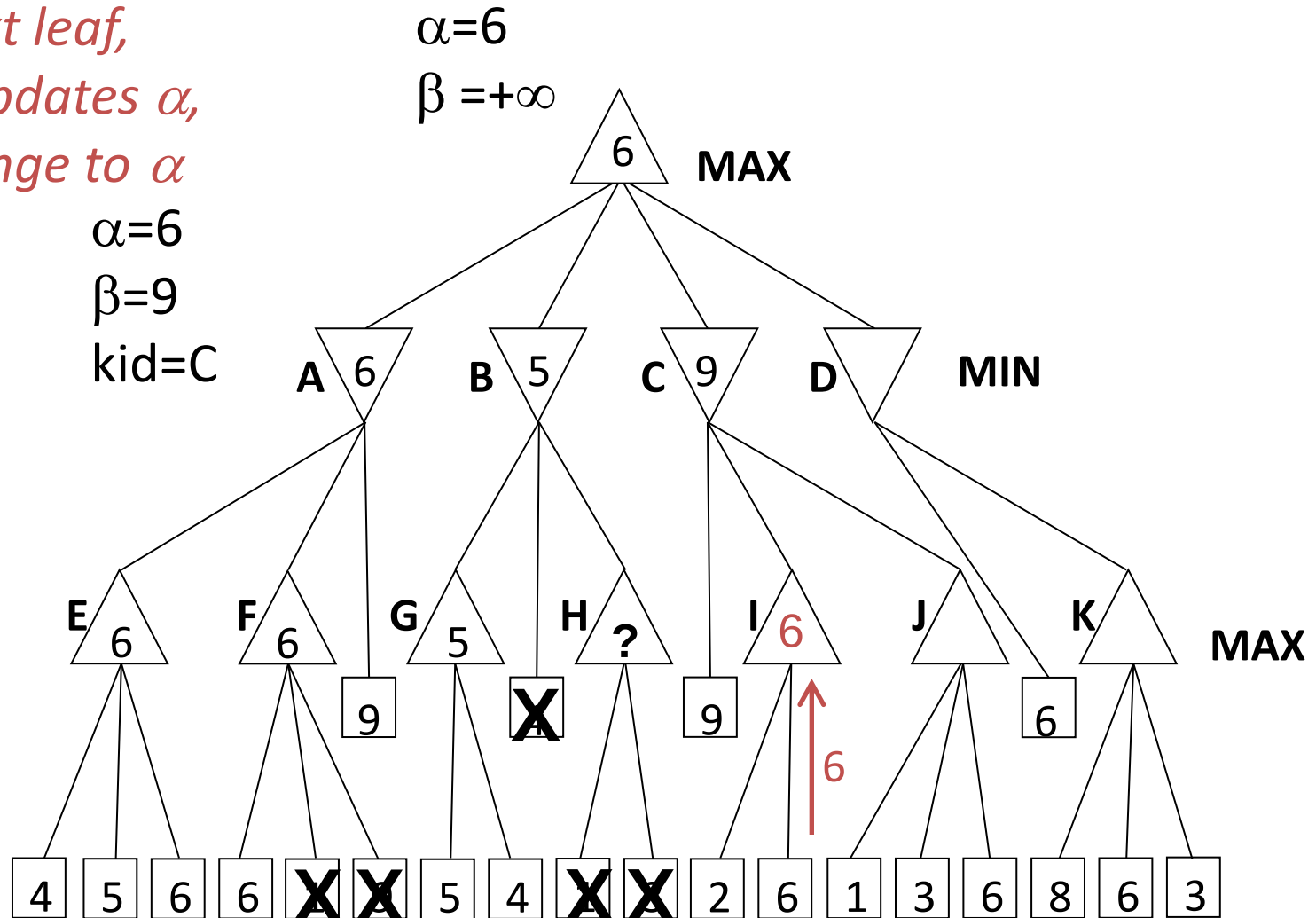




Longer Alpha-Beta Example

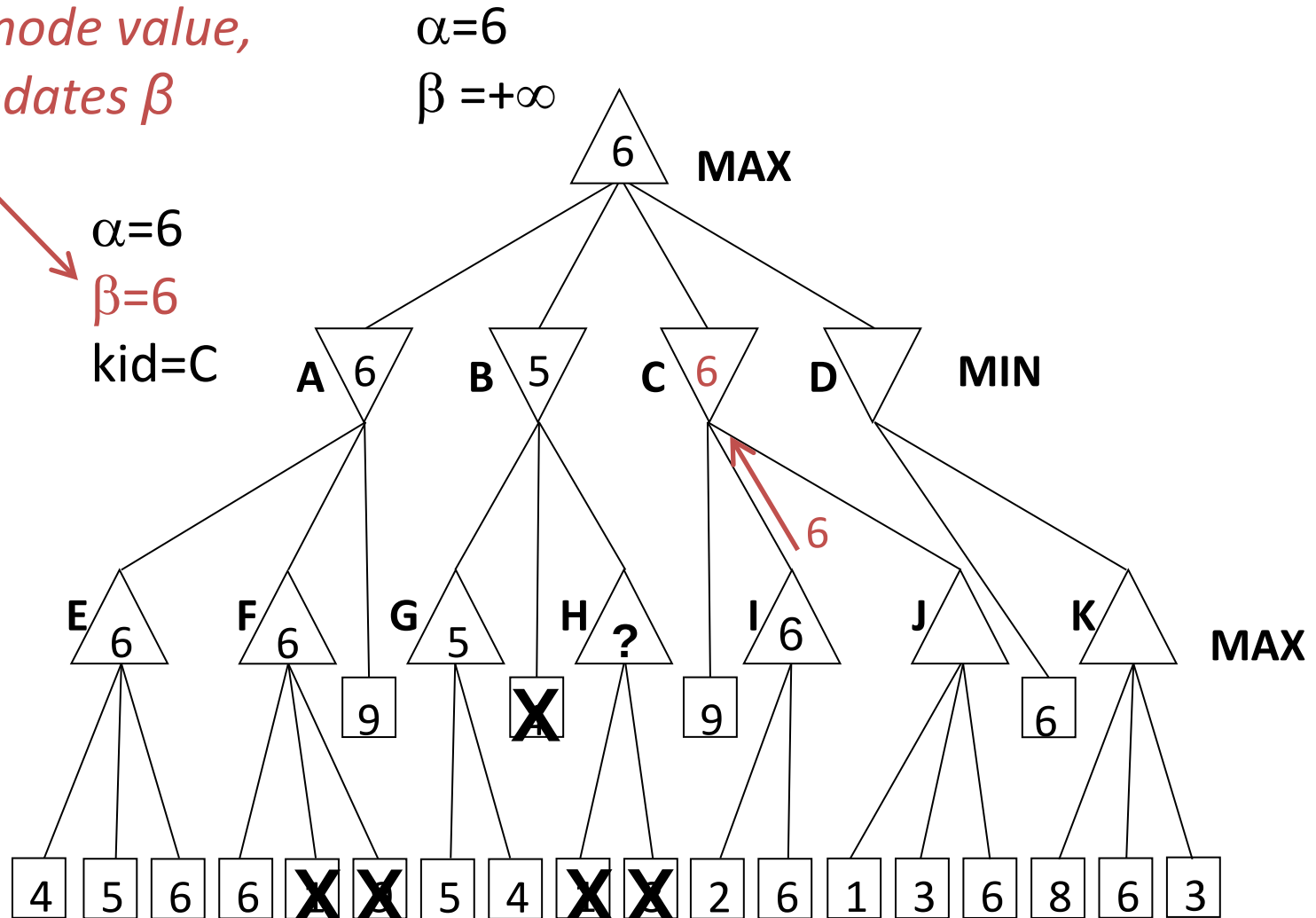
*see next leaf,
MAX updates α ,
no change to α*

$\alpha=6$
 $\beta=9$
kid=I



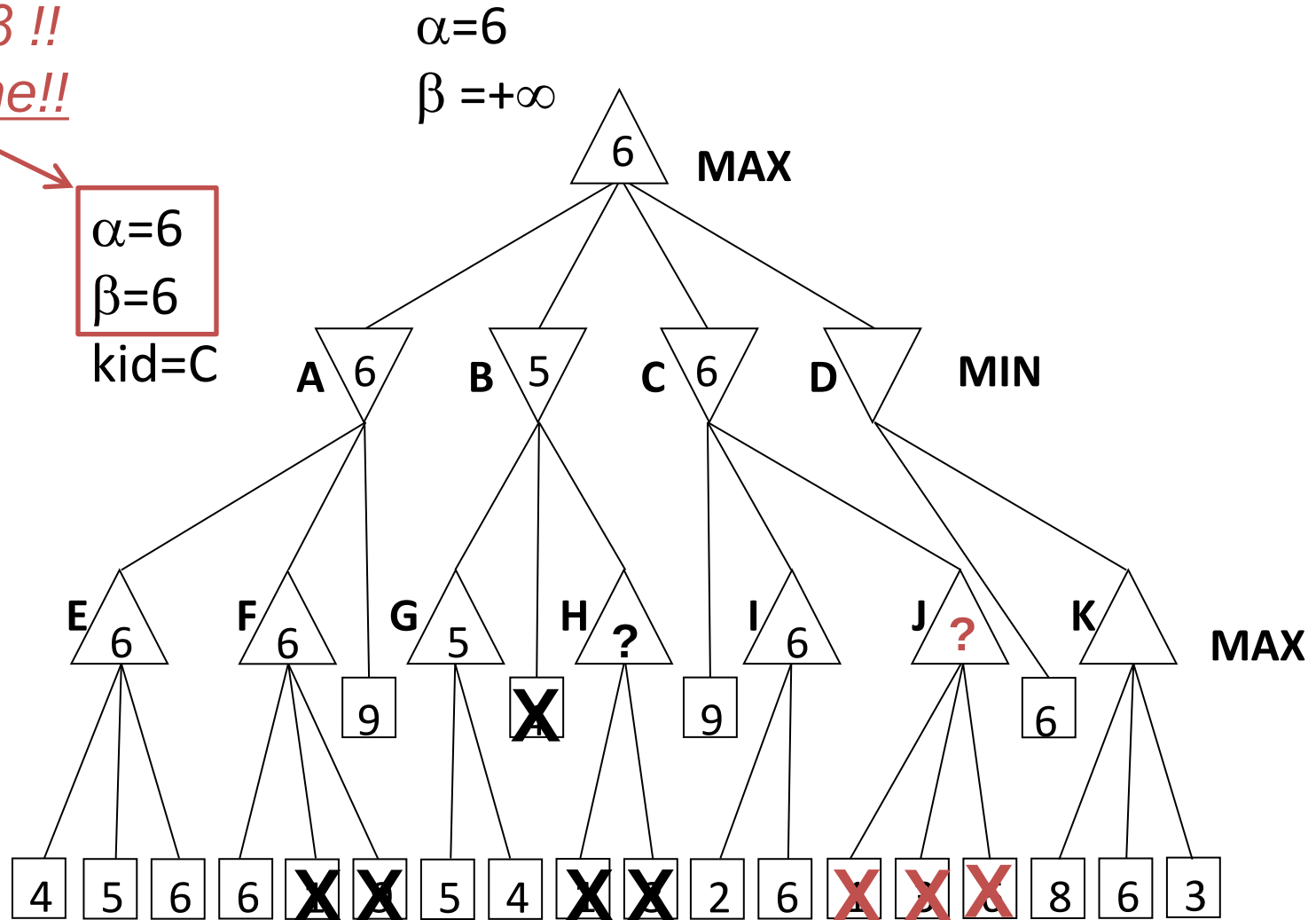
Longer Alpha-Beta Example

*return node value,
MIN updates β*



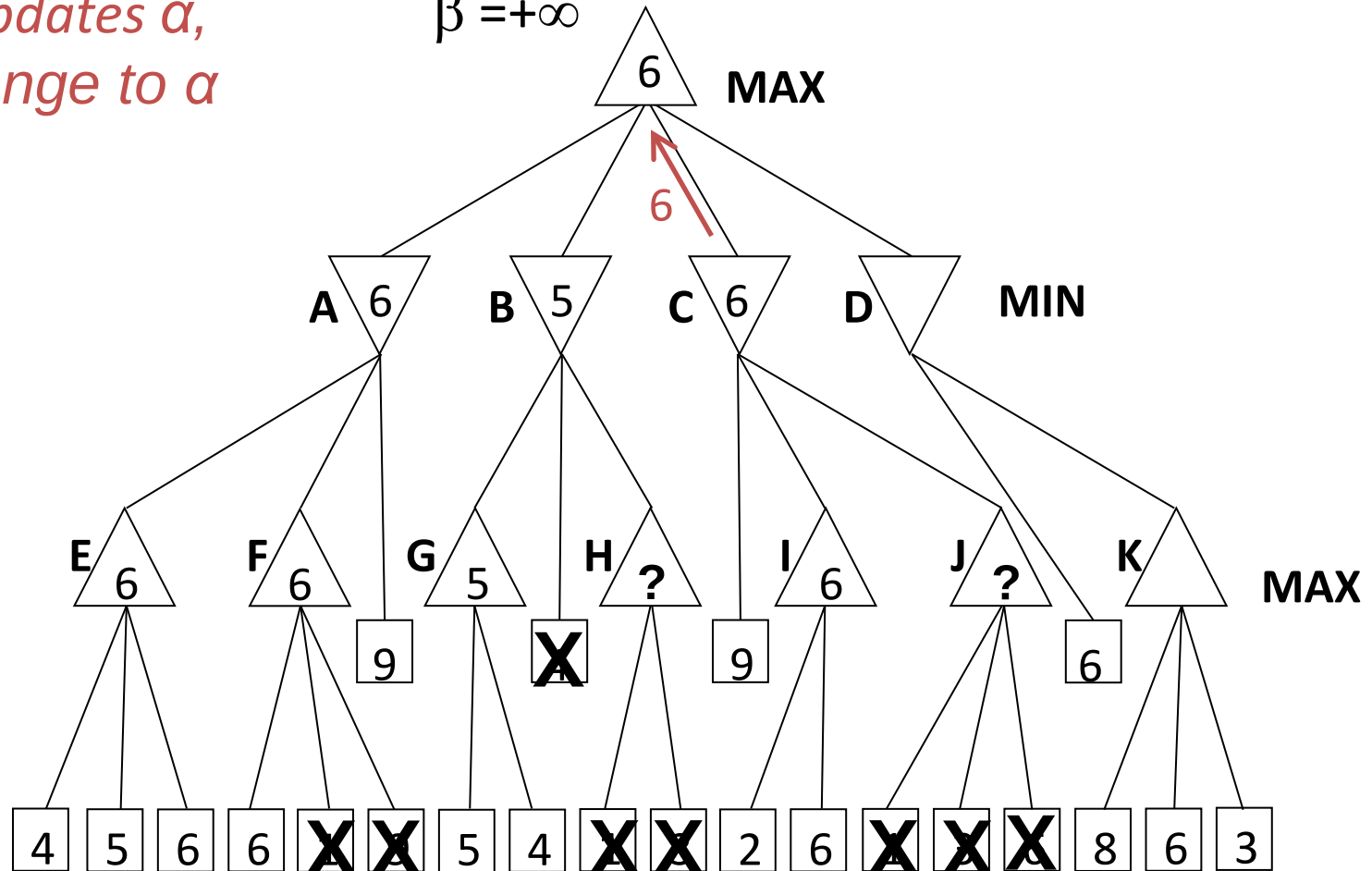
Longer Alpha-Beta Example

$\alpha \geq \beta$!!
Prune!!



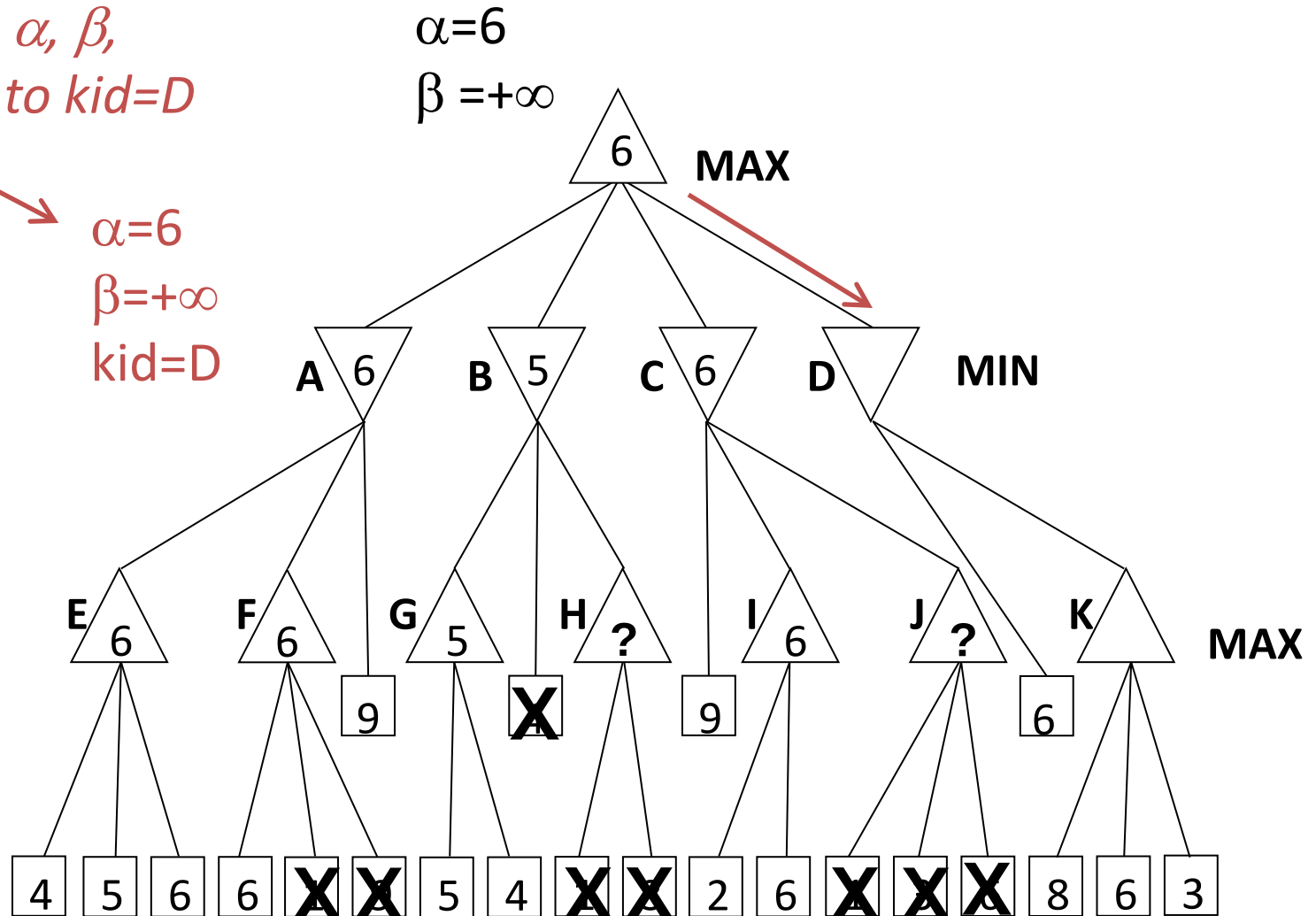
Longer Alpha-Beta Example

return node value, $\rightarrow \alpha=6$
MAX updates α ,
no change to α
 $\beta = +\infty$



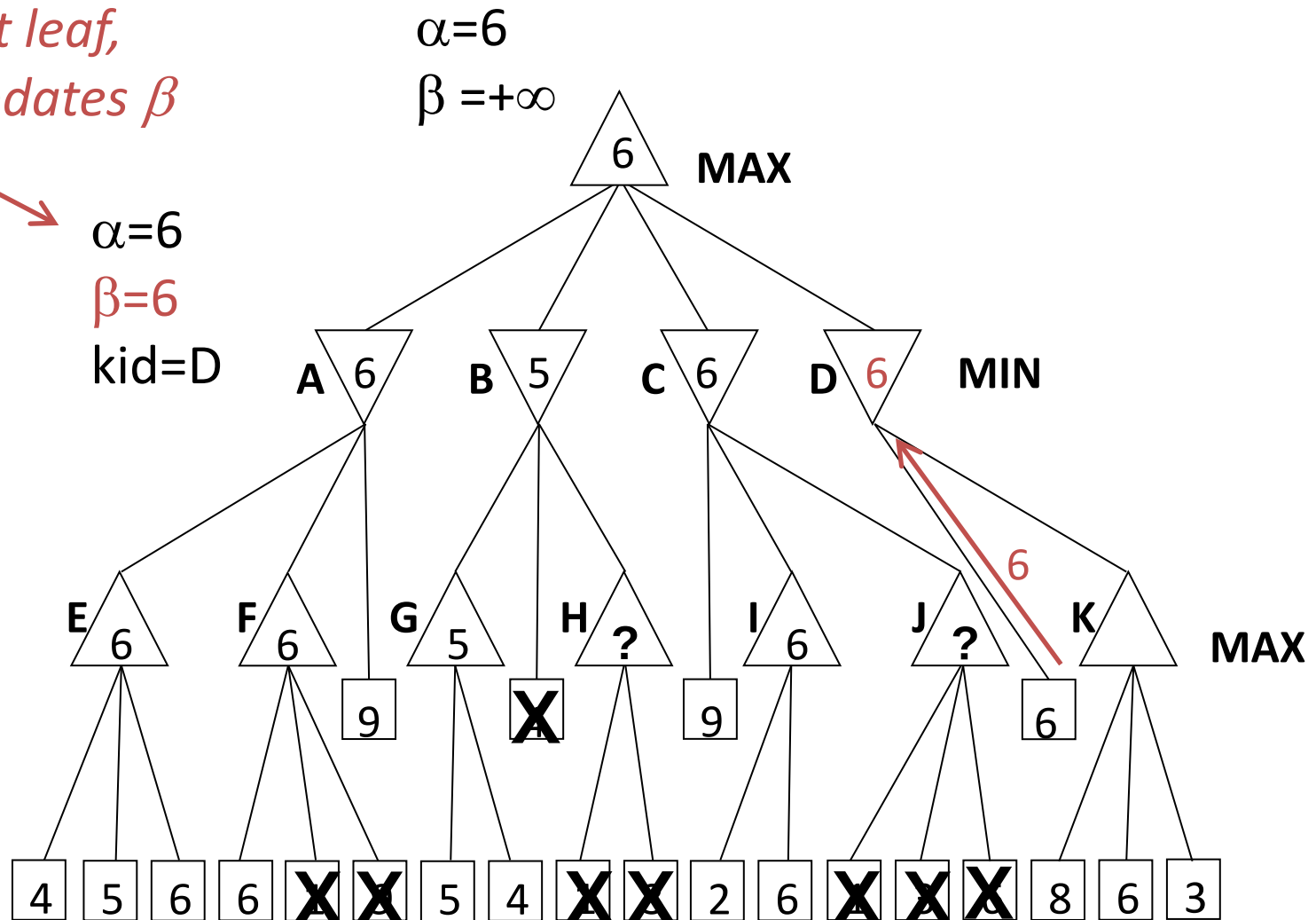
Longer Alpha-Beta Example

*current α , β ,
passed to kid=D*



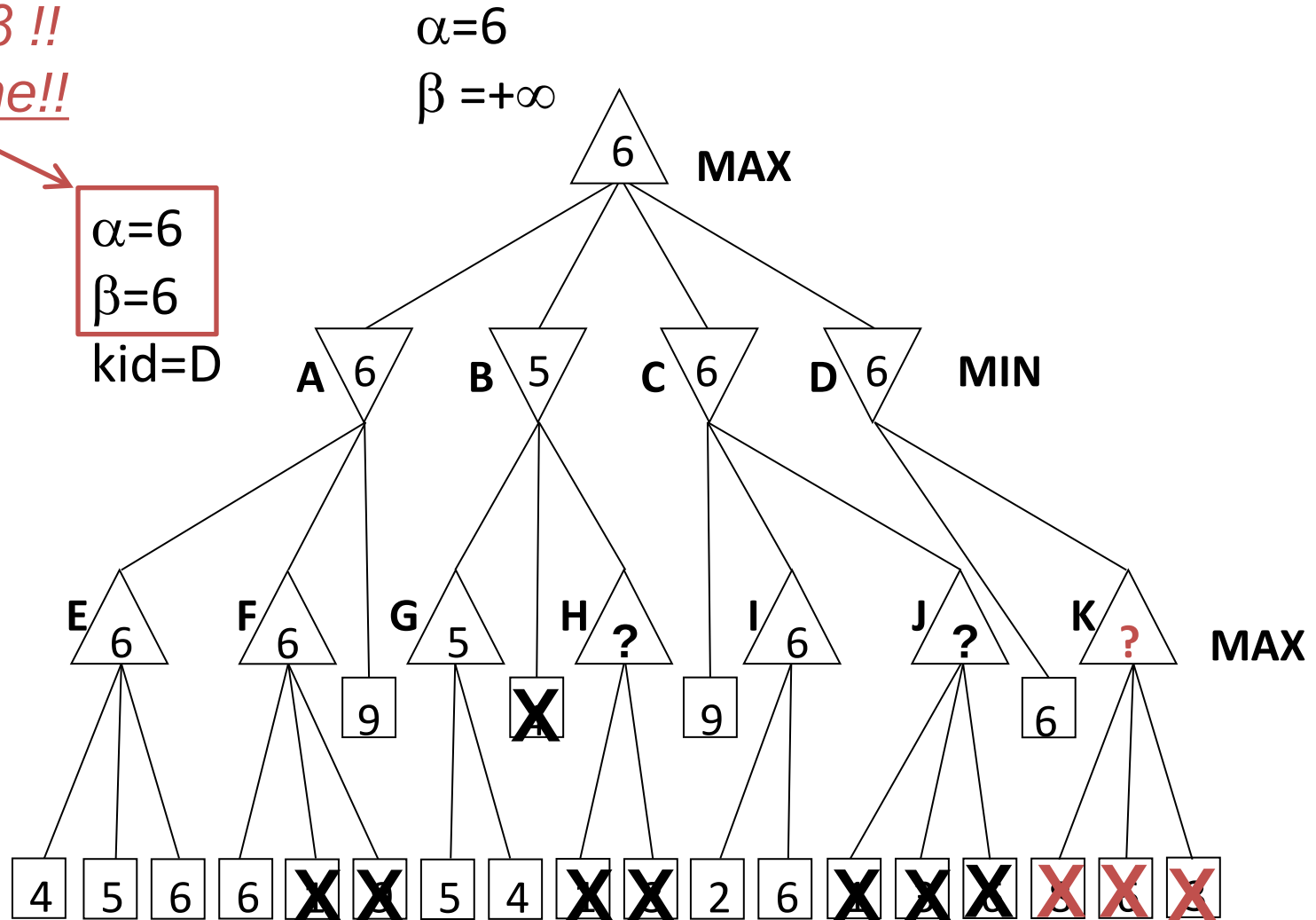
Longer Alpha-Beta Example

*see first leaf,
MIN updates β*



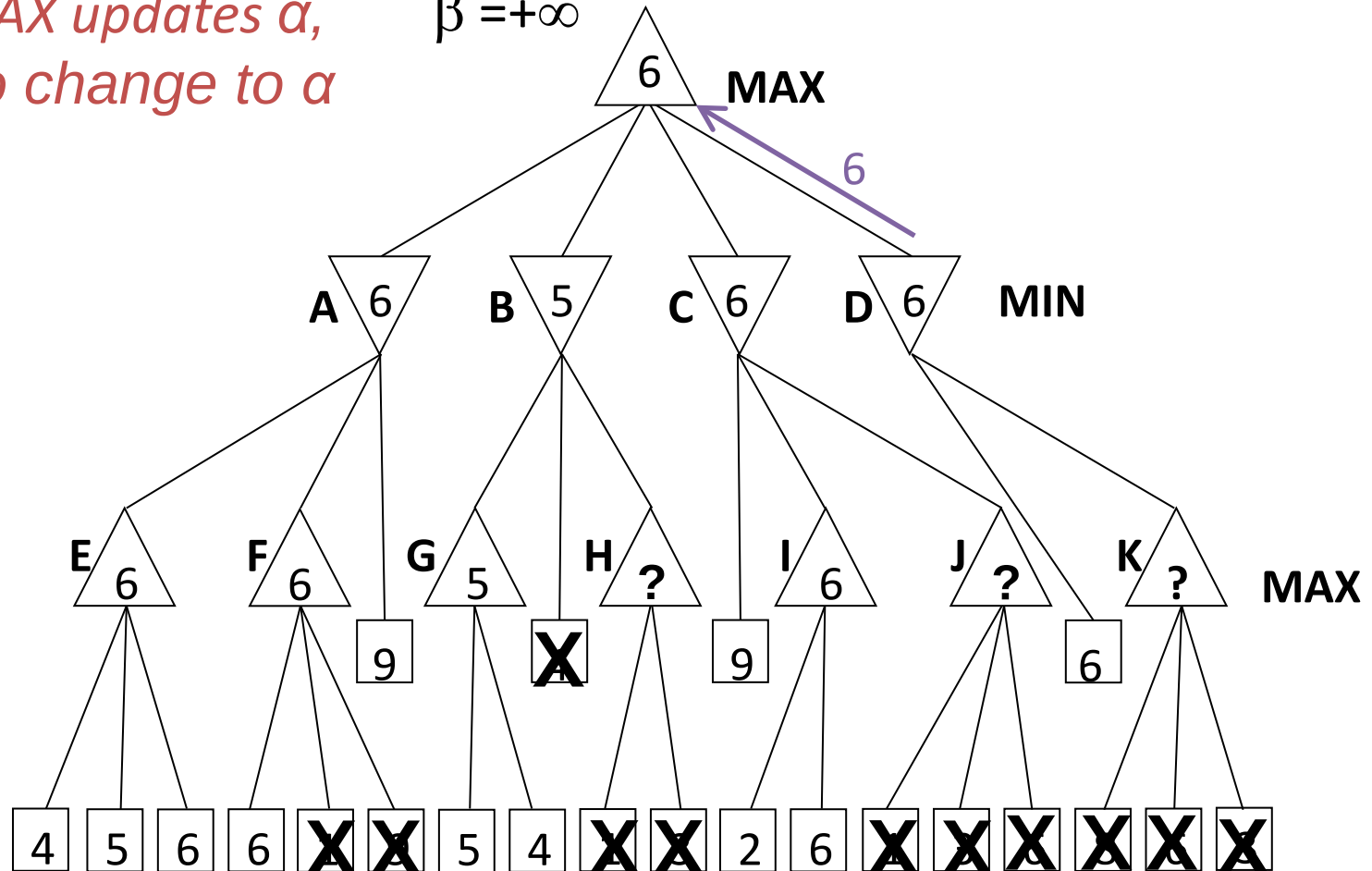
Longer Alpha-Beta Example

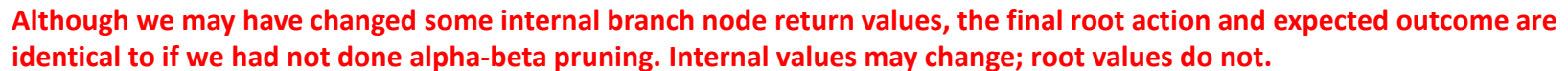
$\alpha \geq \beta$!!
Prune!!



Alpha-Beta Example #2

return node value, $\alpha=6$
MAX updates α , $\beta = +\infty$
no change to α





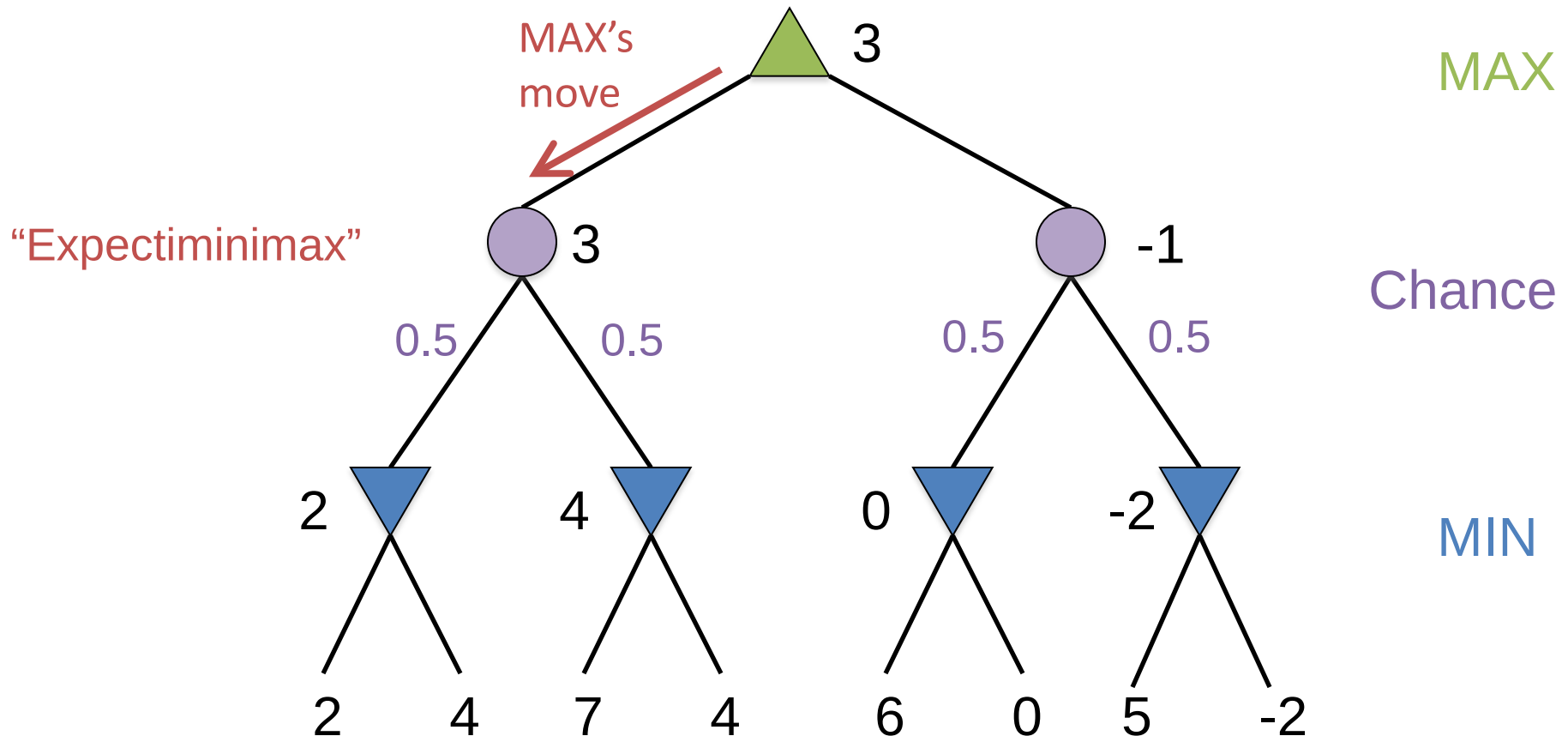
Nondeterministic games

- Ex: Backgammon
 - Roll dice to determine how far to move (random)
 - Player selects which checkers to move (strategy)



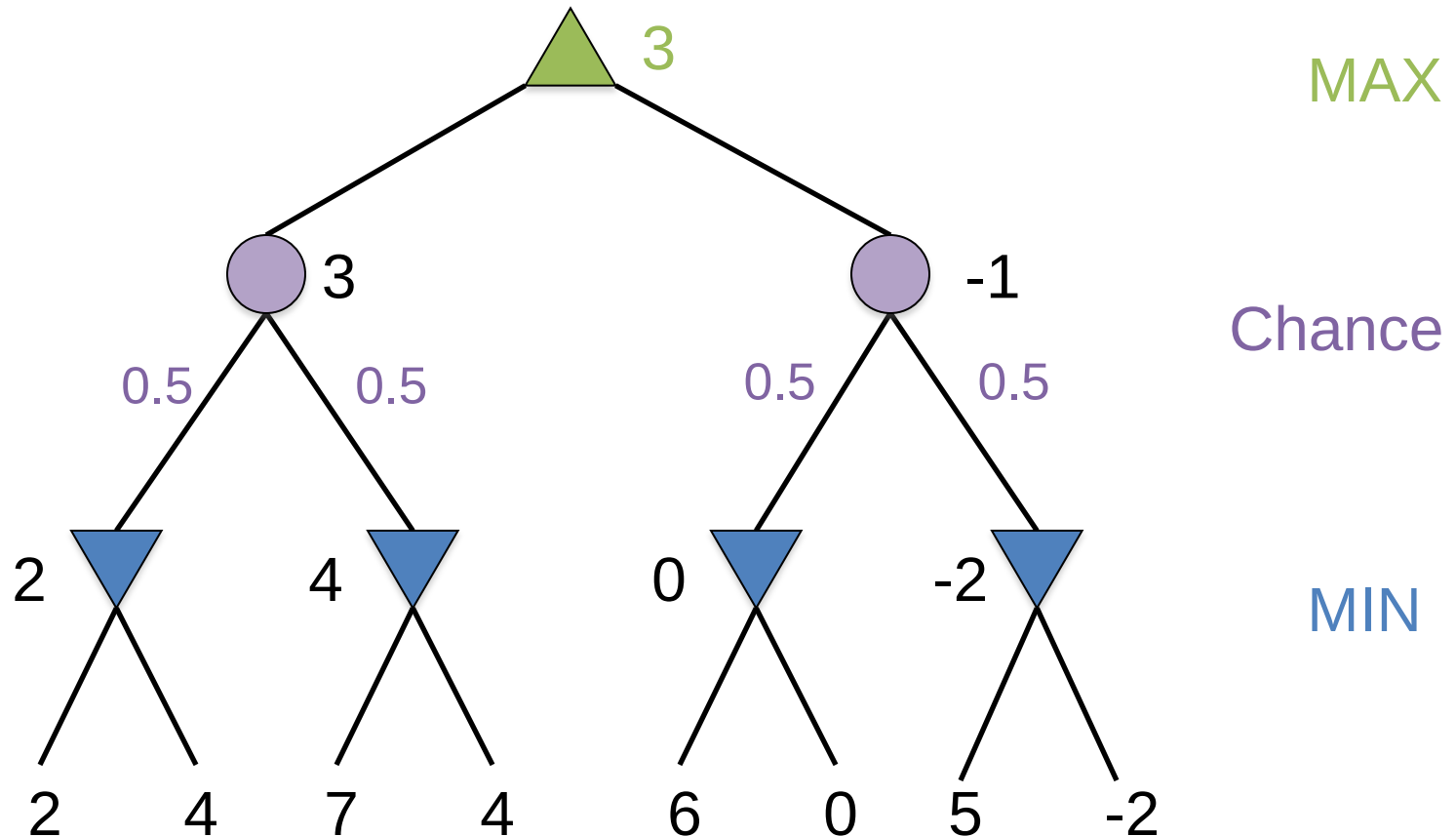
Nondeterministic games

- Chance (random effects) due to dice, card shuffle, ...
- Chance nodes: expectation (weighted average) of successors
- Simplified example: coin flips



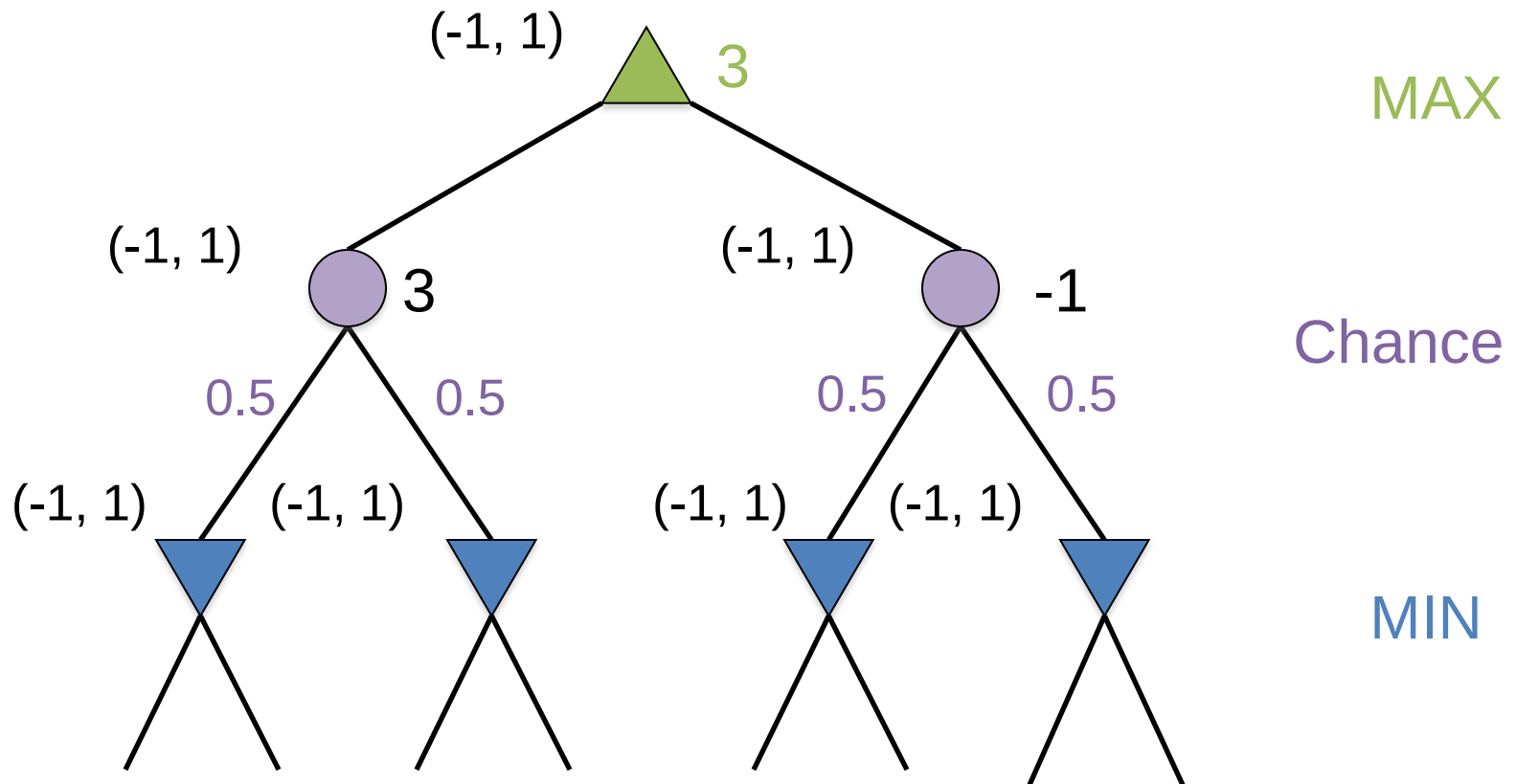
Pruning in nondeterministic games

- Can still apply a form of alpha-beta pruning



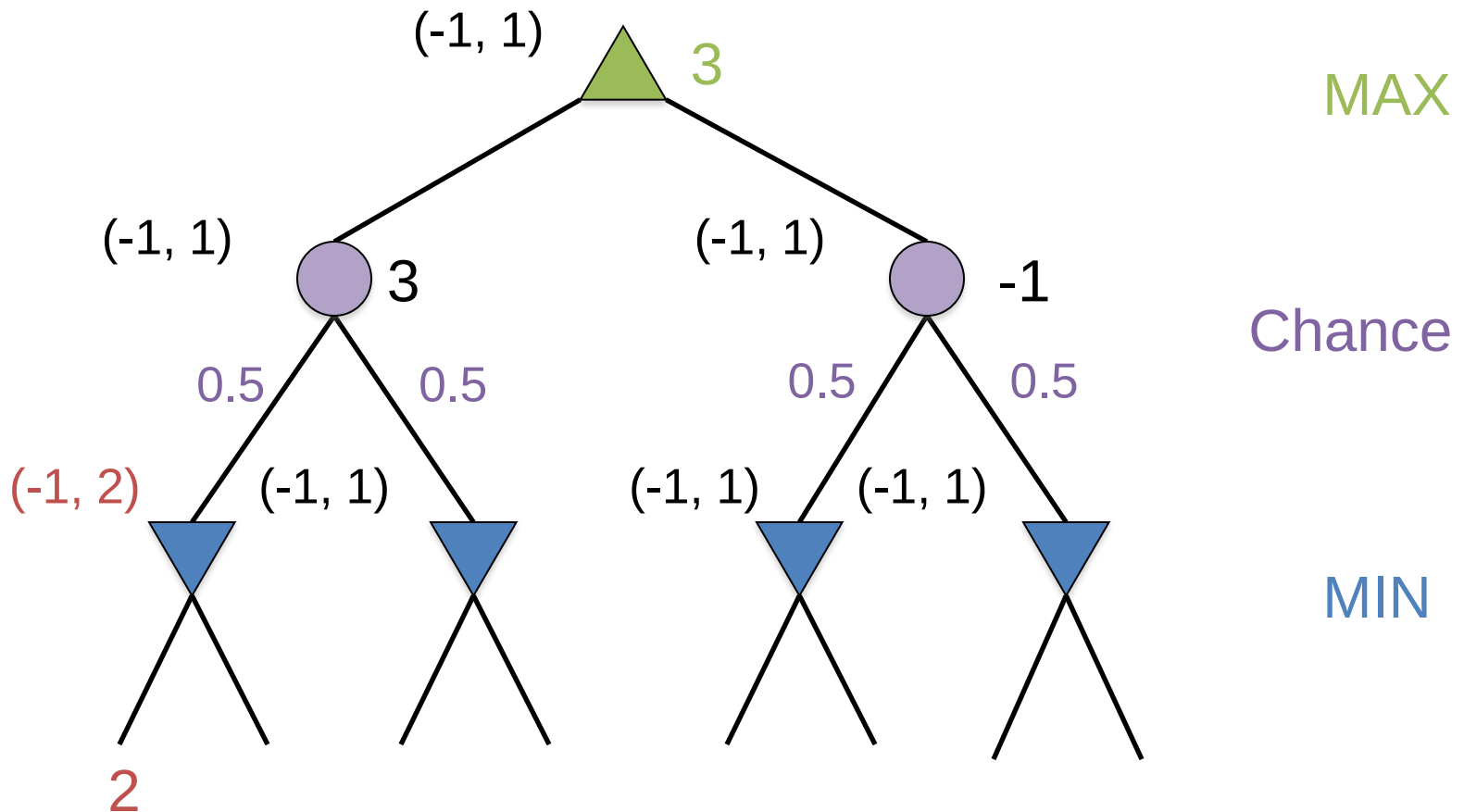
Pruning in nondeterministic games

- Can still apply a form of alpha-beta pruning



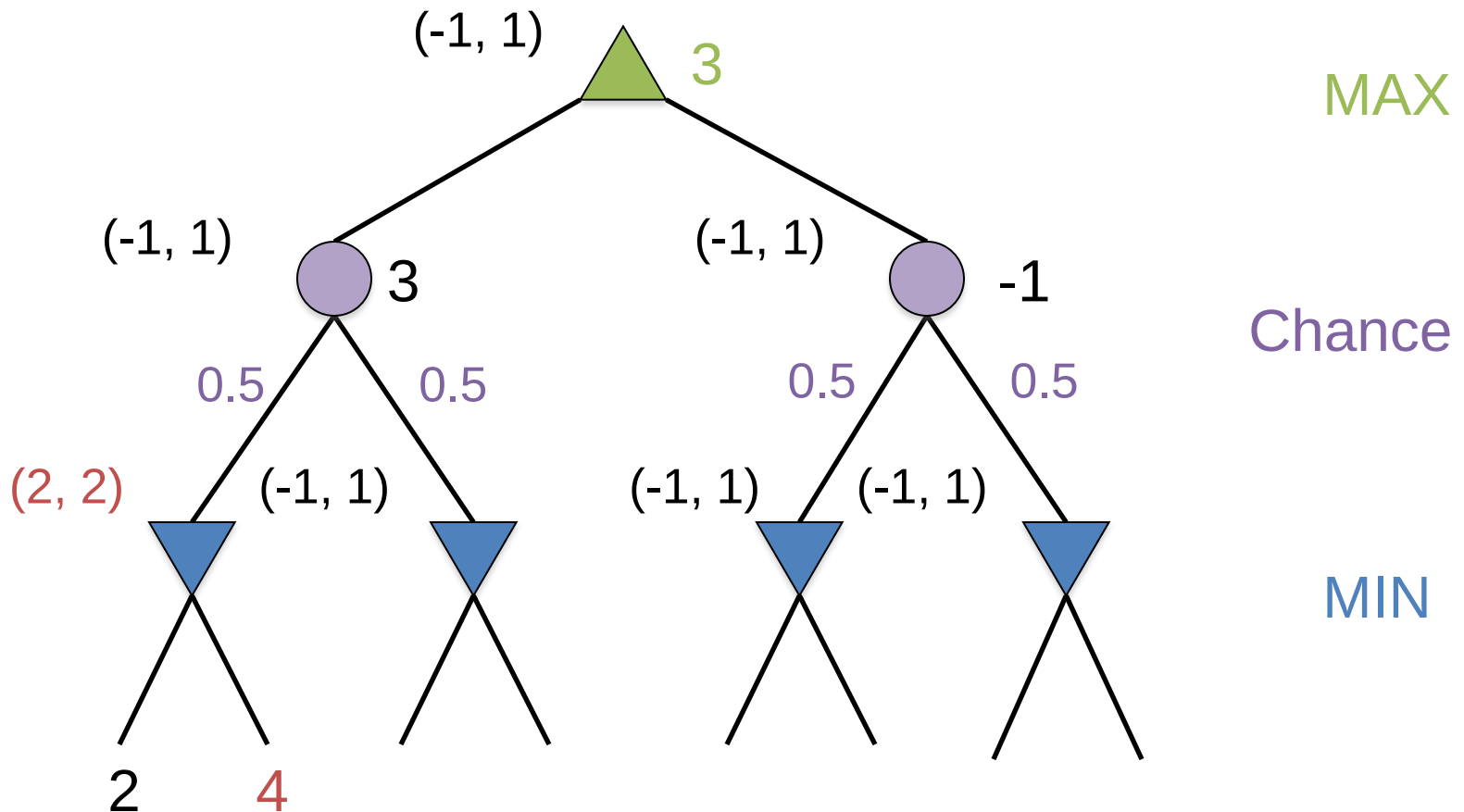
Pruning in nondeterministic games

- Can still apply a form of alpha-beta pruning



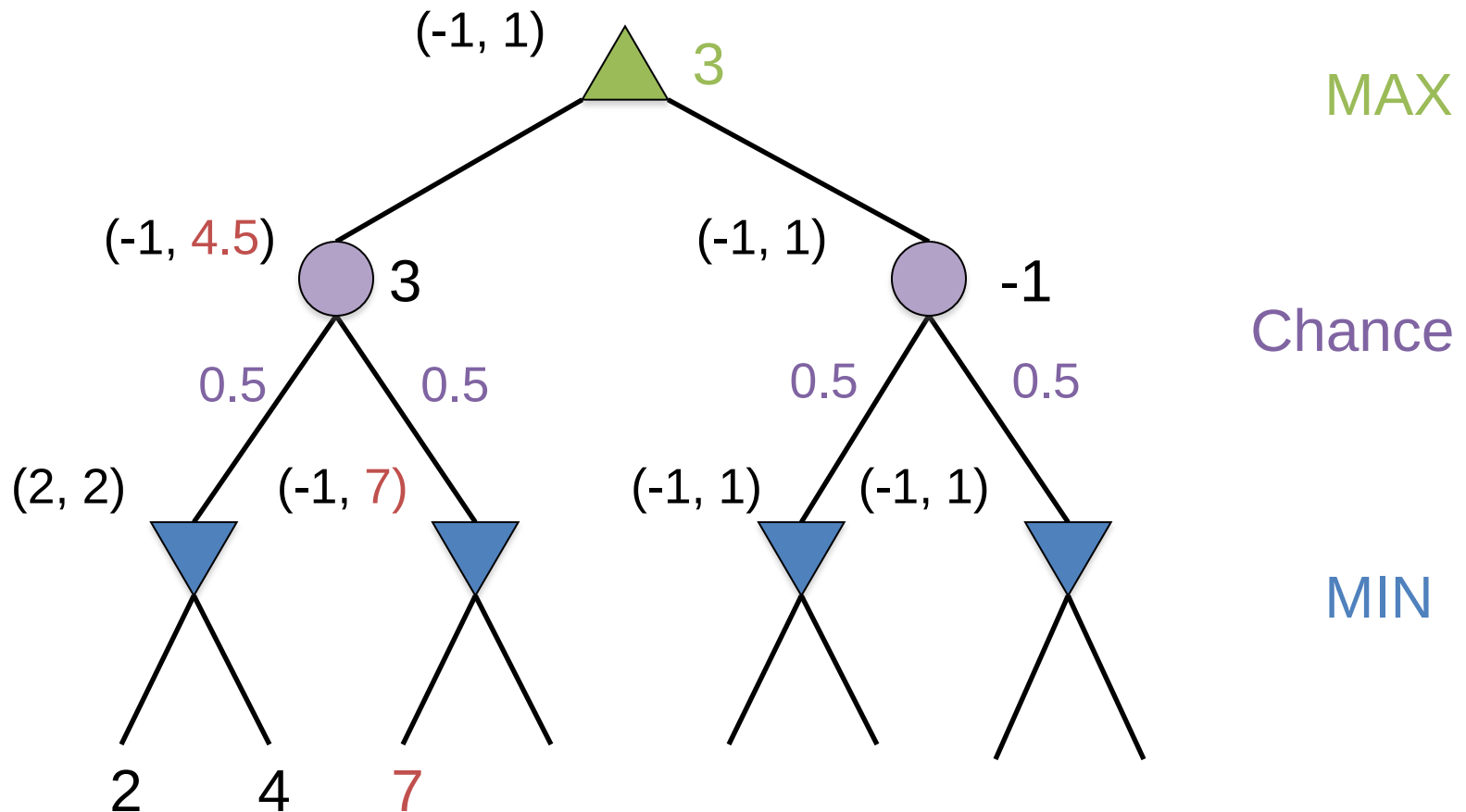
Pruning in nondeterministic games

- Can still apply a form of alpha-beta pruning



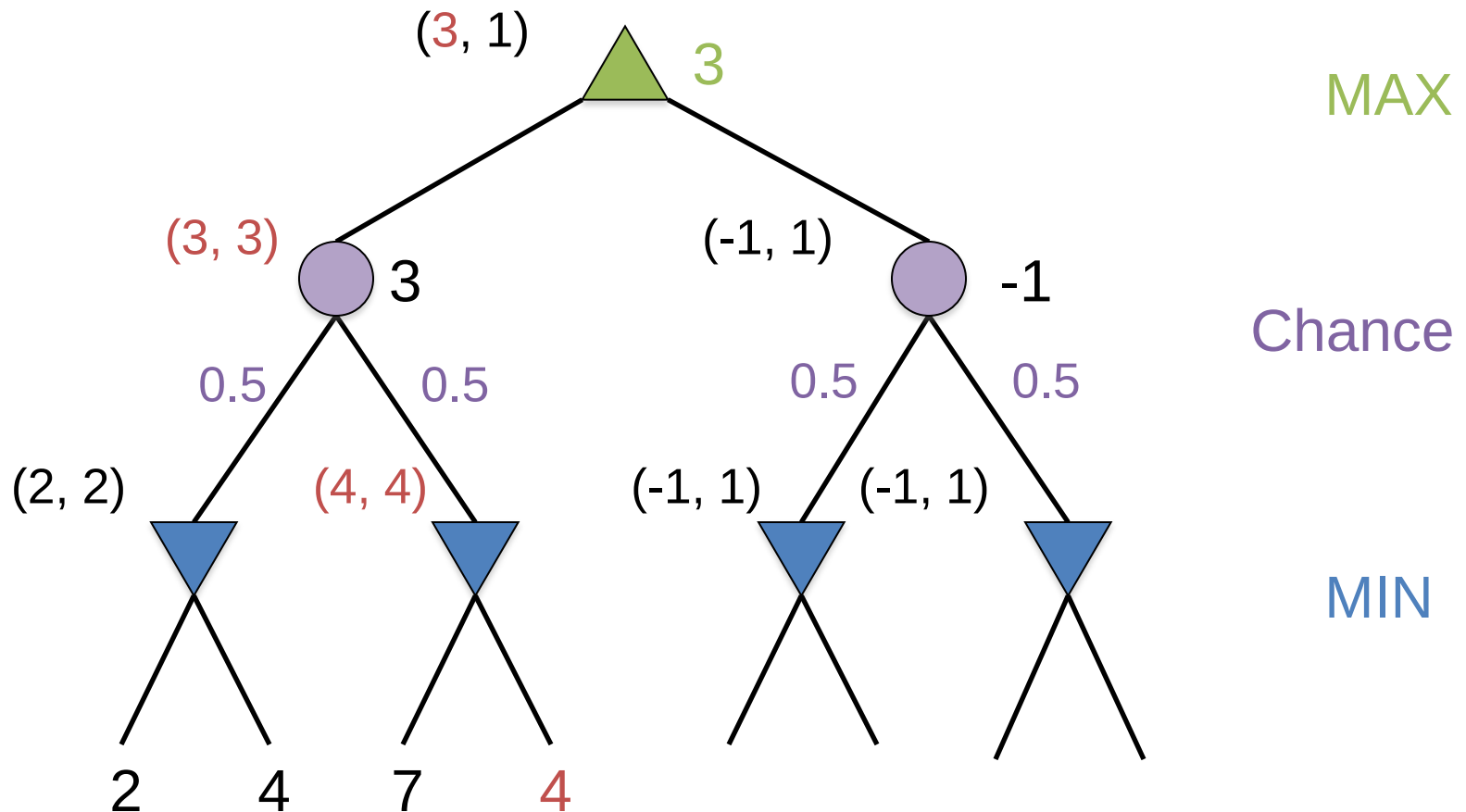
Pruning in nondeterministic games

- Can still apply a form of alpha-beta pruning



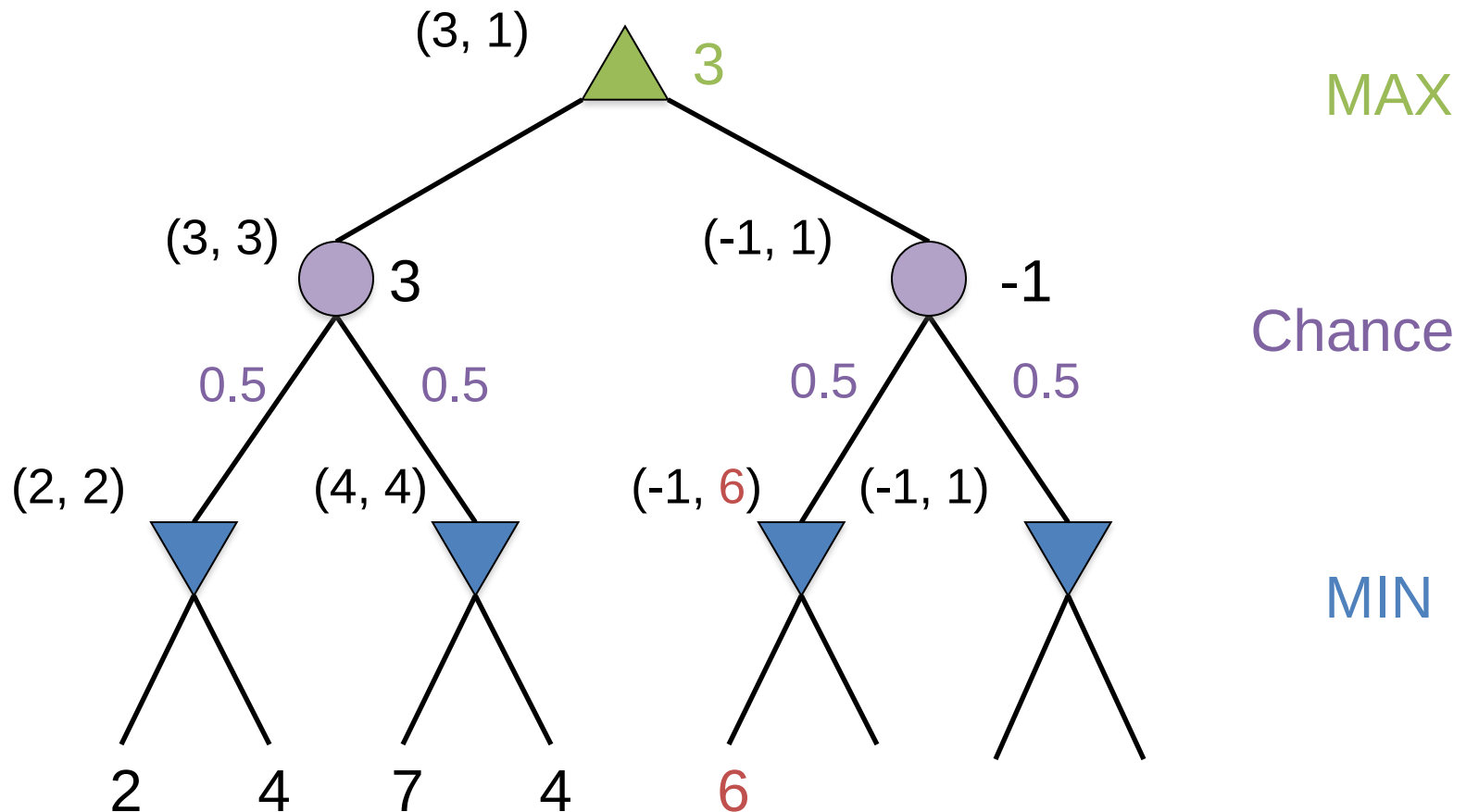
Pruning in nondeterministic games

- Can still apply a form of alpha-beta pruning



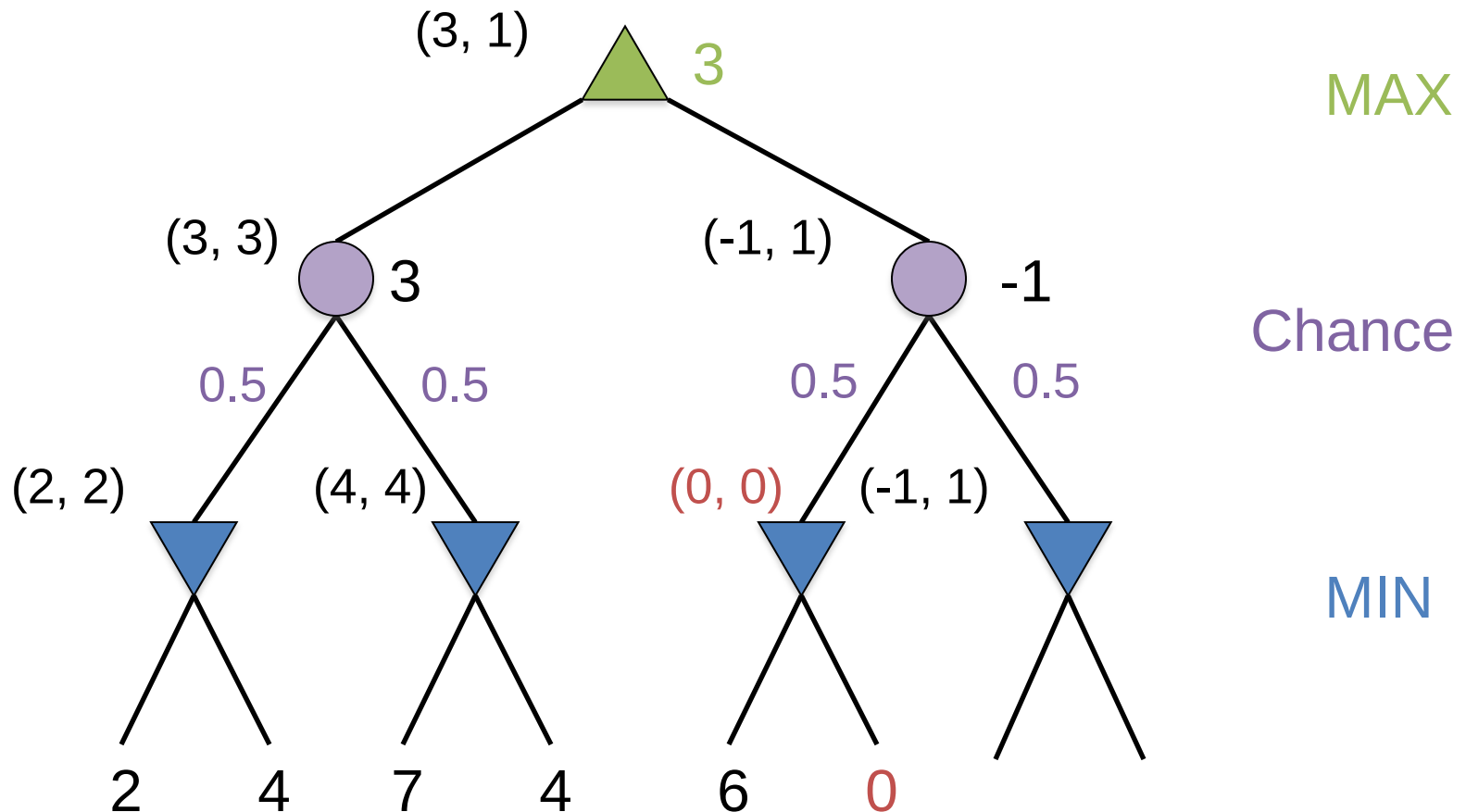
Pruning in nondeterministic games

- Can still apply a form of alpha-beta pruning



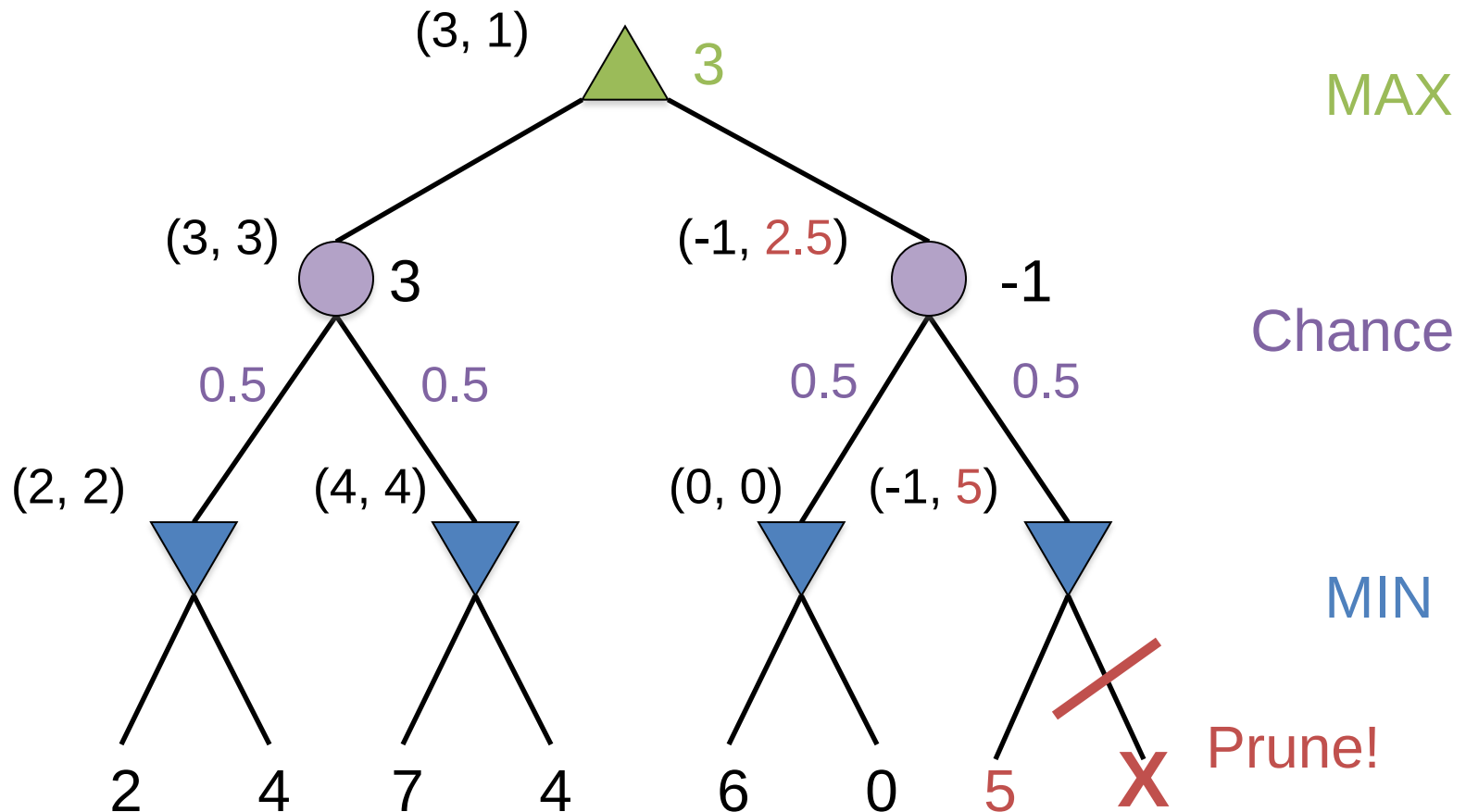
Pruning in nondeterministic games

- Can still apply a form of alpha-beta pruning



Pruning in nondeterministic games

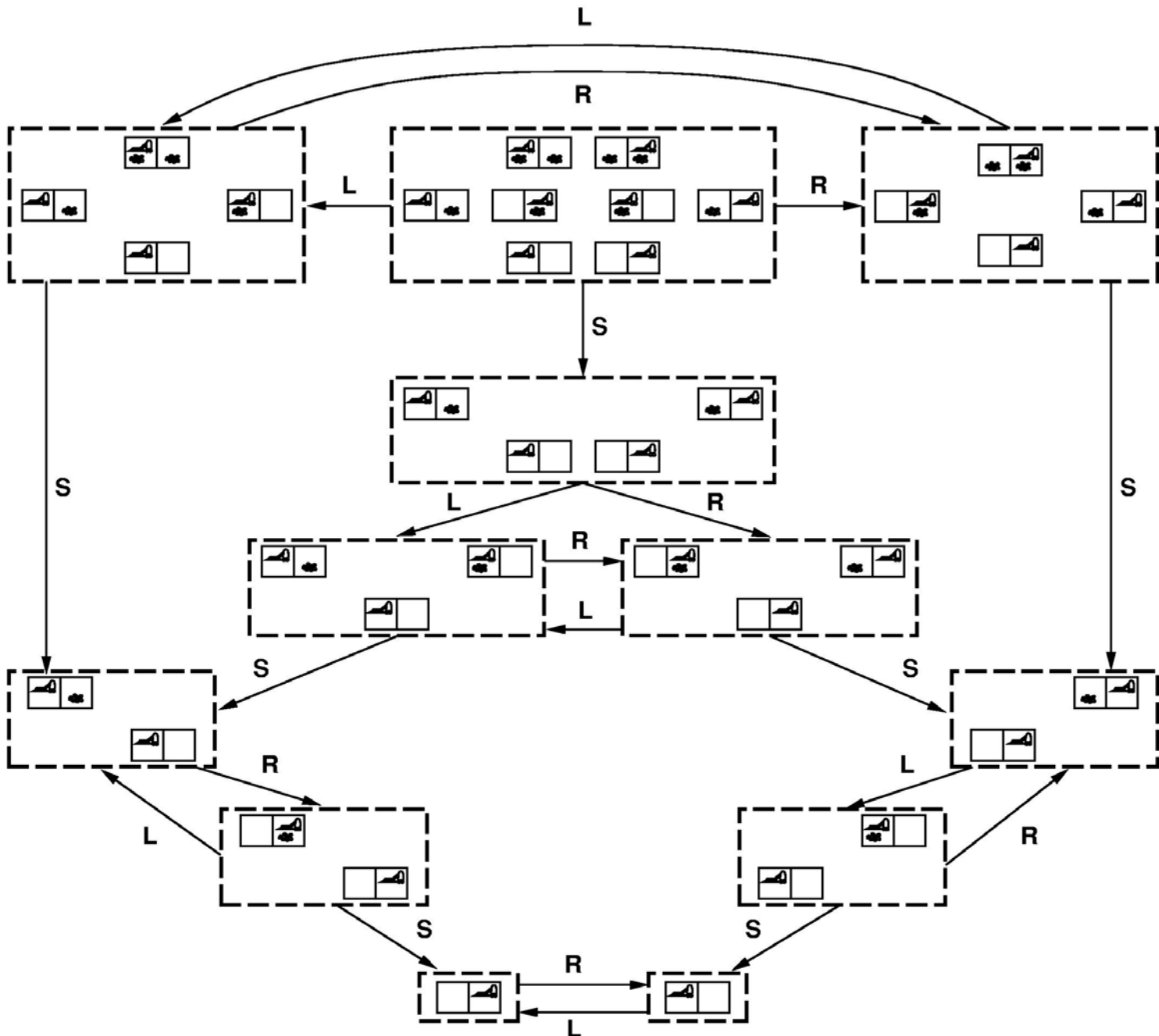
- Can still apply a form of alpha-beta pruning



Partially observable games

- R&N Chapter 5.6 – “The fog of war”
- Background: R&N, Chapter 4.3-4
 - Searching with Nondeterministic Actions/Partial Observations
- Search through Belief States (see Fig. 4.14)
 - Agent’s current belief about which states it might be in, given the sequence of actions & percepts to that point
- $\text{Actions}(b) = ??$ Union? Intersection?
 - Tricky: an action legal in one state may be illegal in another
 - Is an illegal action a NO-OP? or the end of the world?
- Transition Model:
 - $\text{Result}(b,a) = \{ s' : s' = \text{Result}(s, a) \text{ and } s \text{ is a state in } b \}$
- $\text{Goaltest}(b) = \text{every state in } b \text{ is a goal state}$

Belief States for Unobservable Vacuum World



Partially observable games

- R&N Chapter 5.6
- Player's current node is a belief state
- Player's move (action) generates child belief state
- Opponent's move is replaced by Percepts(s)
 - Each possible percept leads to the belief state that is consistent with that percept
- Strategy = a move for every possible percept sequence
- Minimax returns the worst state in the belief state
- Many more complications and possibilities!!
 - Opponent may select a move that is not optimal, but instead minimizes the information transmitted, or confuses the opponent
 - May not be reasonable to consider ALL moves; open P-QR3??
- **See R&N, Chapter 5.6, for more info**

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:
 - AlphaGo recently (3/2016) beat 9th dan Lee Sedol
 - $b > 300$ (!); full game tree has $> 10^{1760}$ leaf nodes (!!)
- See (e.g.) <http://www.cs.ualberta.ca/~games/> for more info

High branching factors

- What can we do when the search tree is too large?
 - Ex: Go ($b = 50\text{-}200\text{+}$ moves per state)
 - Heuristic state evaluation (score a partial game)
- Where does this heuristic come from?
 - Hand designed
 - Machine learning on historical game patterns
 - Monte Carlo methods – play random games



Monte Carlo heuristic scoring

- Idea: play out the game randomly, and use the results as a score
 - Easy to generate & score lots of random games
 - May use 1000s of games for a node
- The basis of Monte Carlo tree search algorithms...

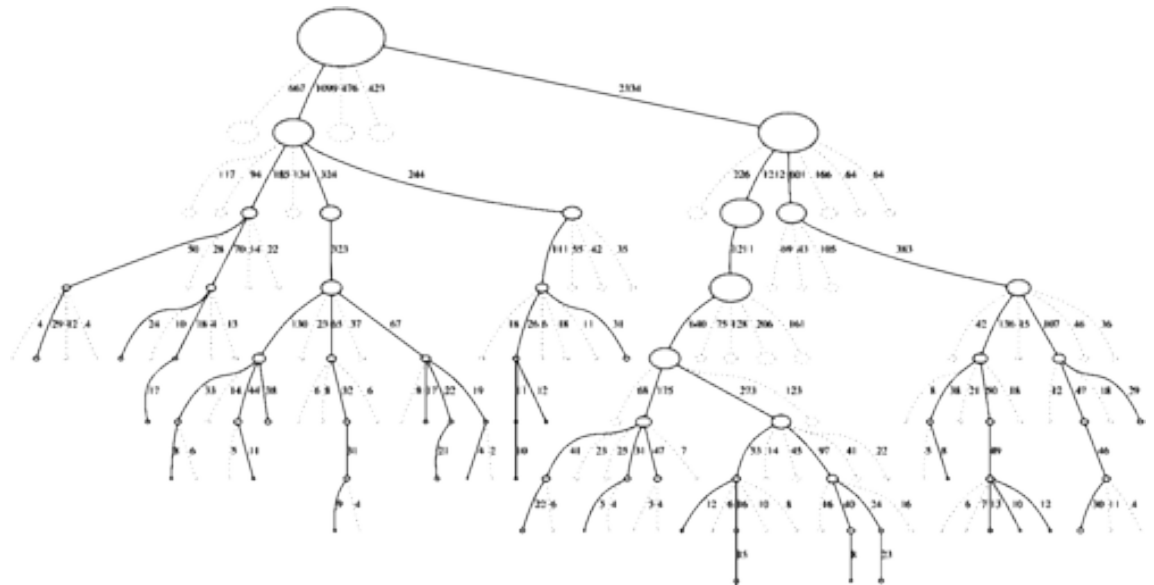


Image from www.mcts.ai

Monte Carlo Tree Search

- Should we explore the whole (top of) the tree?
 - Some moves are obviously not good...
 - Should spend time exploring / scoring promising ones
- This is a *multi-armed bandit* problem:
- Want to spend our time on good moves
- Which moves have high payout?
 - Hard to tell – random...
- *Explore vs. exploit* tradeoff

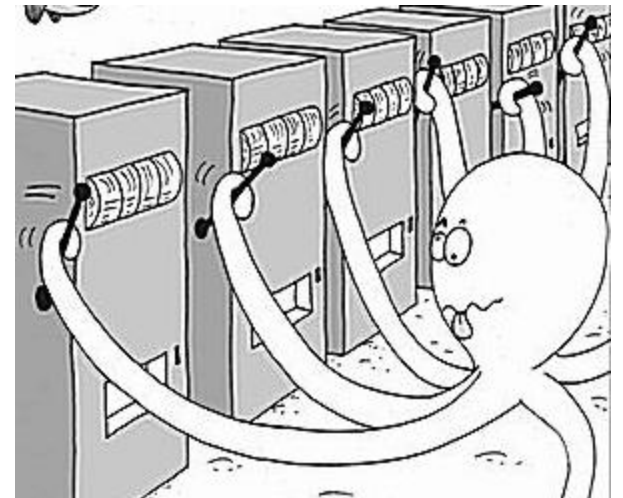
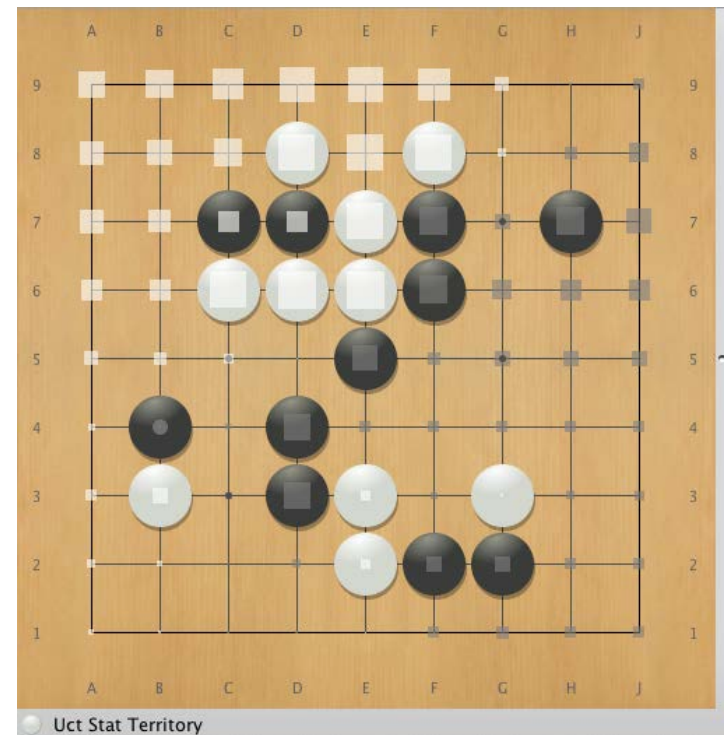
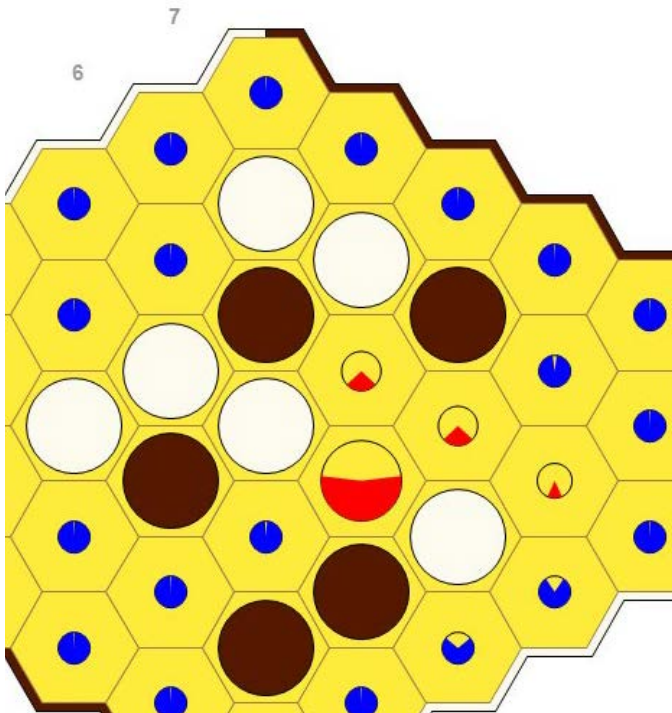


Image from Microsoft Research

Visualizing MCTS

- At each level of the tree, keep track of
 - Number of times we've explored a path
 - Number of times we won
- Follow winning (from max/min perspective) strategies more often, but also explore 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.