Announcements

- Coding Homework 1 will be released
  - Due 1/25 at 11:59pm
- Written Homework 1 will be released
  - Due 1/25 at 11:59pm

Adversarial Search

Forest Agostinelli
University of South Carolina

Topics Covered in This Class

- **Part 1: Search**
  - Pathfinding
    - Uninformed search
    - Informed search
  - Adversarial search
  - Optimization
    - Local search
    - Constraint satisfaction
- **Part 2: Knowledge Representation and Reasoning**
  - Propositional logic
  - First-order logic
  - Prolog
- **Part 3: Knowledge Representation and Reasoning Under Uncertainty**
  - Probability
  - Bayesian networks
- **Part 4: Machine Learning**
  - Supervised learning
    - Inductive logic programming
    - Linear models
    - Deep neural networks
    - PyTorch
  - Reinforcement learning
    - Markov decision processes
    - Dynamic programming
    - Model-free RL
  - Unsupervised learning
    - Clustering
    - Autoencoders

Outline

- Background
- Minimax search
- Cutting off search
- Alpha-beta pruning
- Alpha-Go

Types of Games

- Number of players: One, two, more
- Information: Perfect/imperfect
- Cooperative/Adversarial

- We will focus on two-player, turn-taking, perfect information, zero-sum (constant-sum) games
  - Tic-tac-toe
  - Connect four
  - Checkers
  - Chess
  - Go

Zero-Sum Games

- **Checkers:** 1950: First computer player. 1994: First computer champion: Chinook ended 40-year-reign of human champion Marion Tinsley. 2007: Checkers solved! We know the game will always end in a draw if neither player makes a mistake.
- **Chess:** 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. We currently have the world's greatest chess player on our phones.
- **Go:** The best Go players have only recently been defeated by computers (2016). In Go, $b > 300$. Classic programs use pattern knowledge bases, but big recent advances use Monte Carlo (randomized) expansion methods and deep neural networks

Game Playing: Deterministic Case

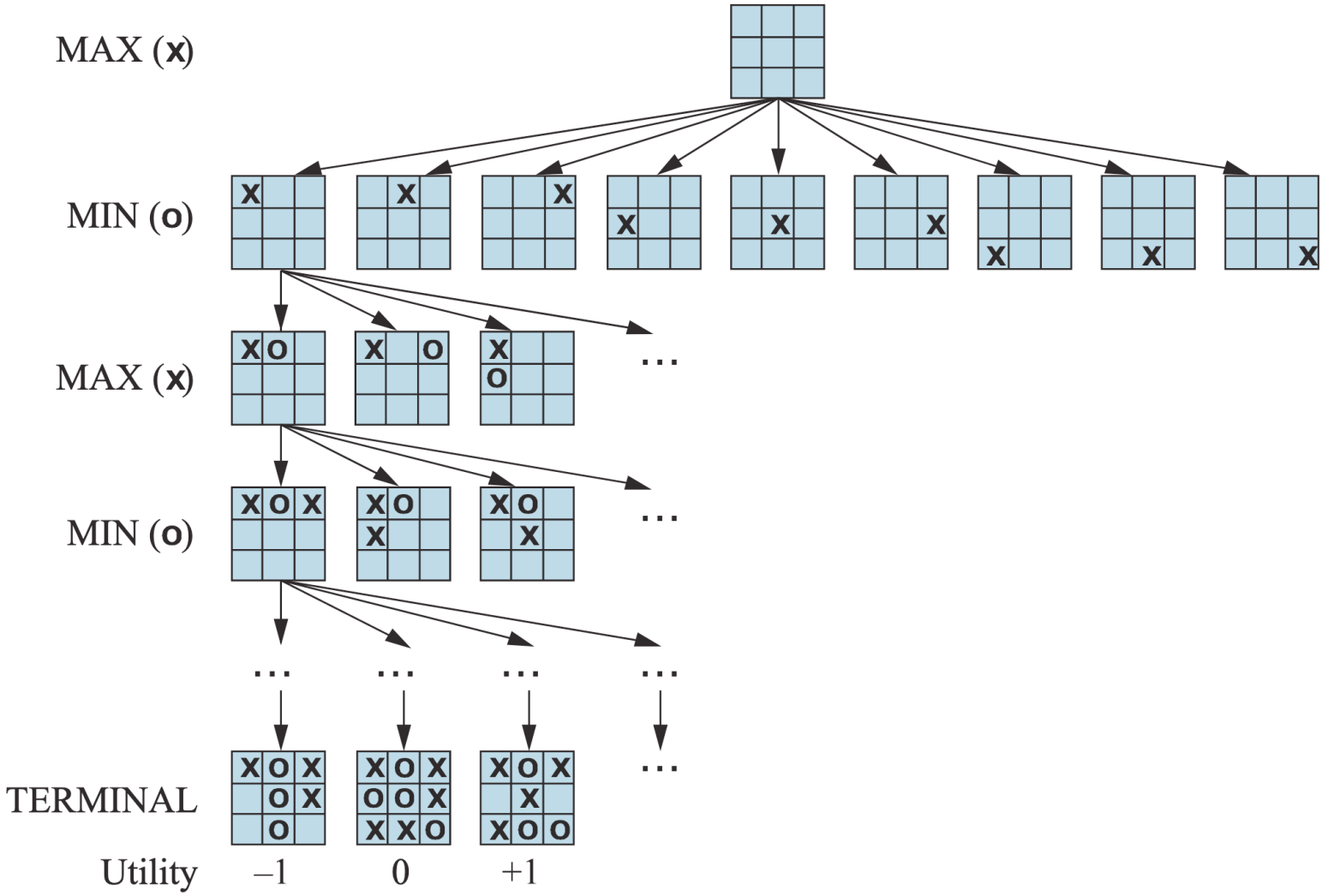
- Two-player, turn-taking, perfect information, zero-sum (constant-sum) games
  - Max (you): want to maximize your utility
  - Min (opponent): wants to minimize your utility
- TO-MOVE(s)
  - Whose turn it is
- ACTIONS(s)
  - The set of legal actions in state s
- RESULT(s, a)
  - The transition model which defines the state resulting from taking action a in state s
- IS-TERMINAL(s)
  - Returns true when the game is over and false otherwise
- UTILITY(s, p)
  - A utility function which defines the final numeric payoff to player p at the terminal state s
  - 1 (win), 0 (lose), or 0.5 (draw) -or-
  - 1 (win), -1 (lose), or 0 (draw)

Game Tree

- As in previous lectures, games form a state space tree where the nodes are states and the edges are actions
- When searching for the best move, this forms a type of search tree that we call a game tree
- A ply is a move by one player
  - Each level in the game tree is a ply

Game Tree

- Tic-tac-toe

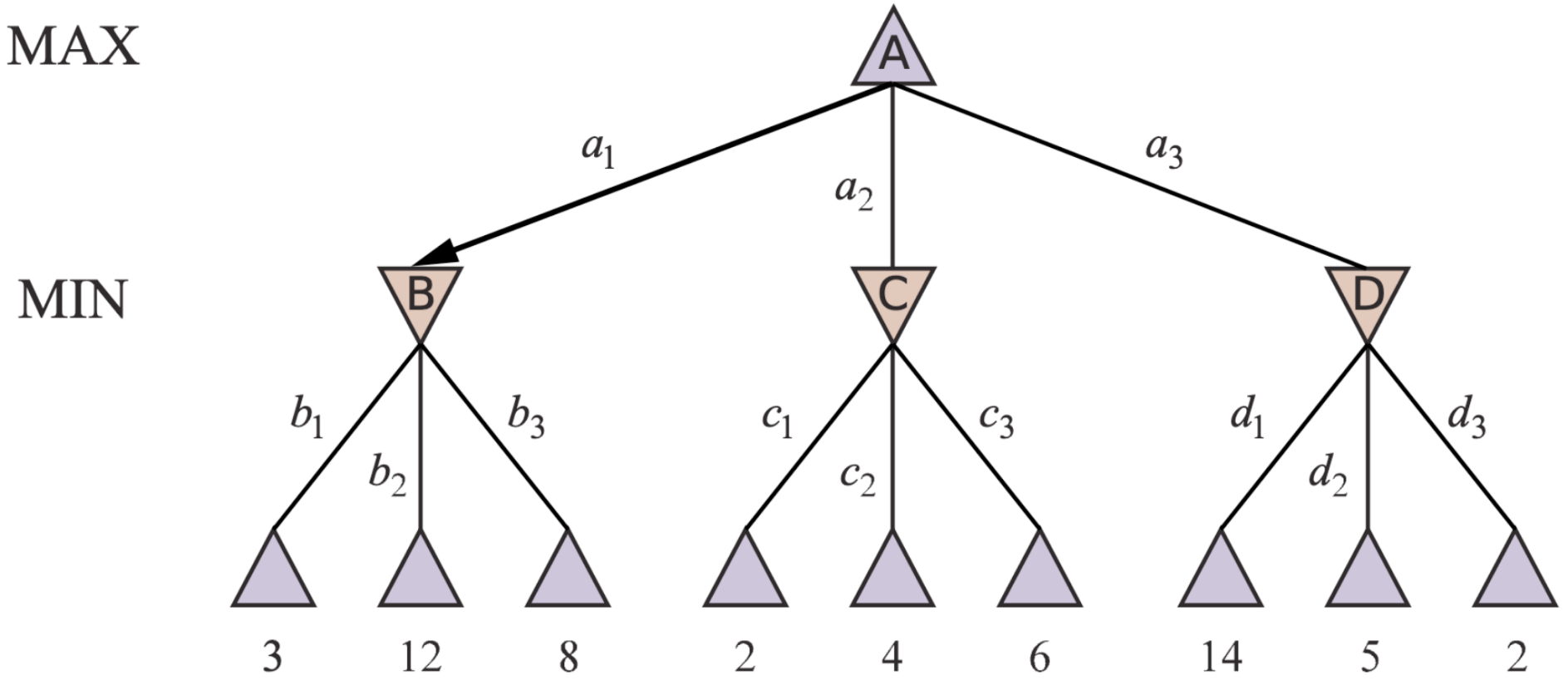


Outline

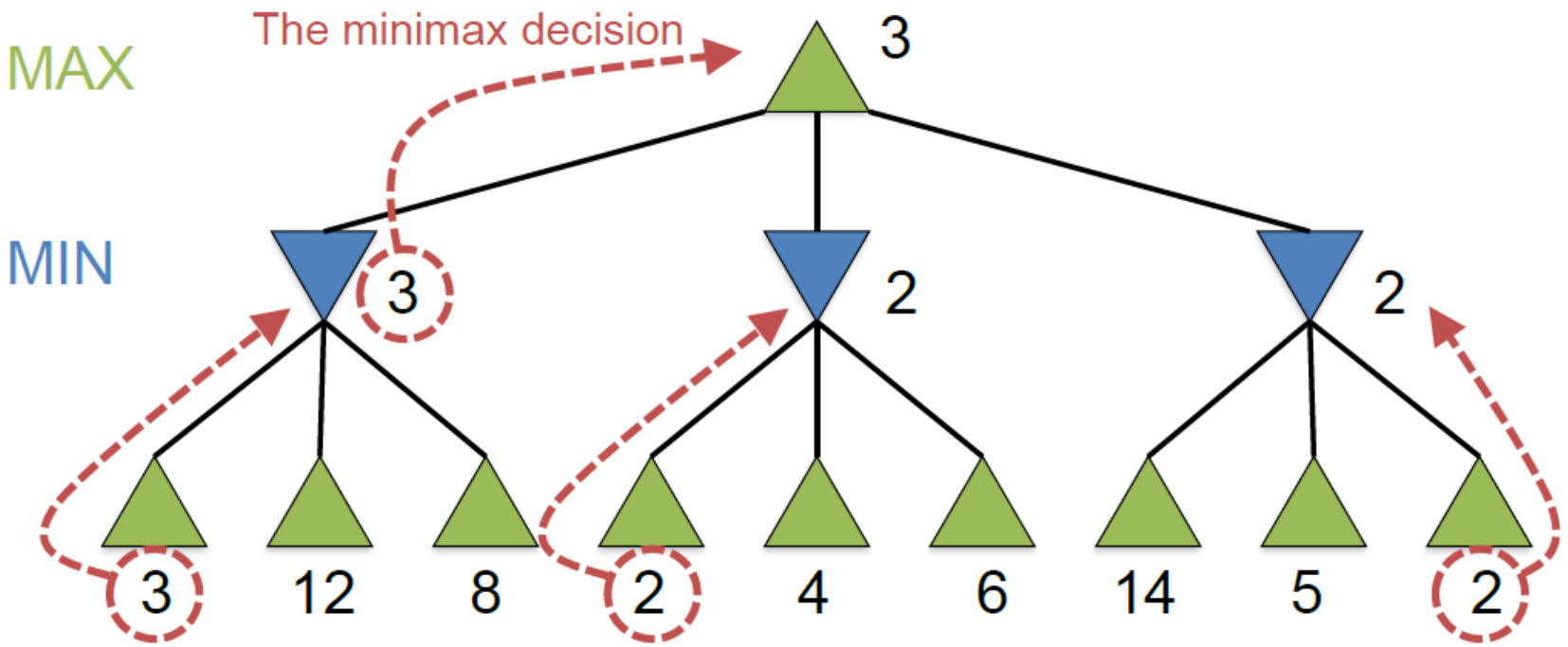
- Background
- Minimax search
- Cutting off search
- Alpha-beta pruning
- Alpha-Go

Which Action Should Max Choose?

- Two ply example



Minimax Search: Example



Minimax Search

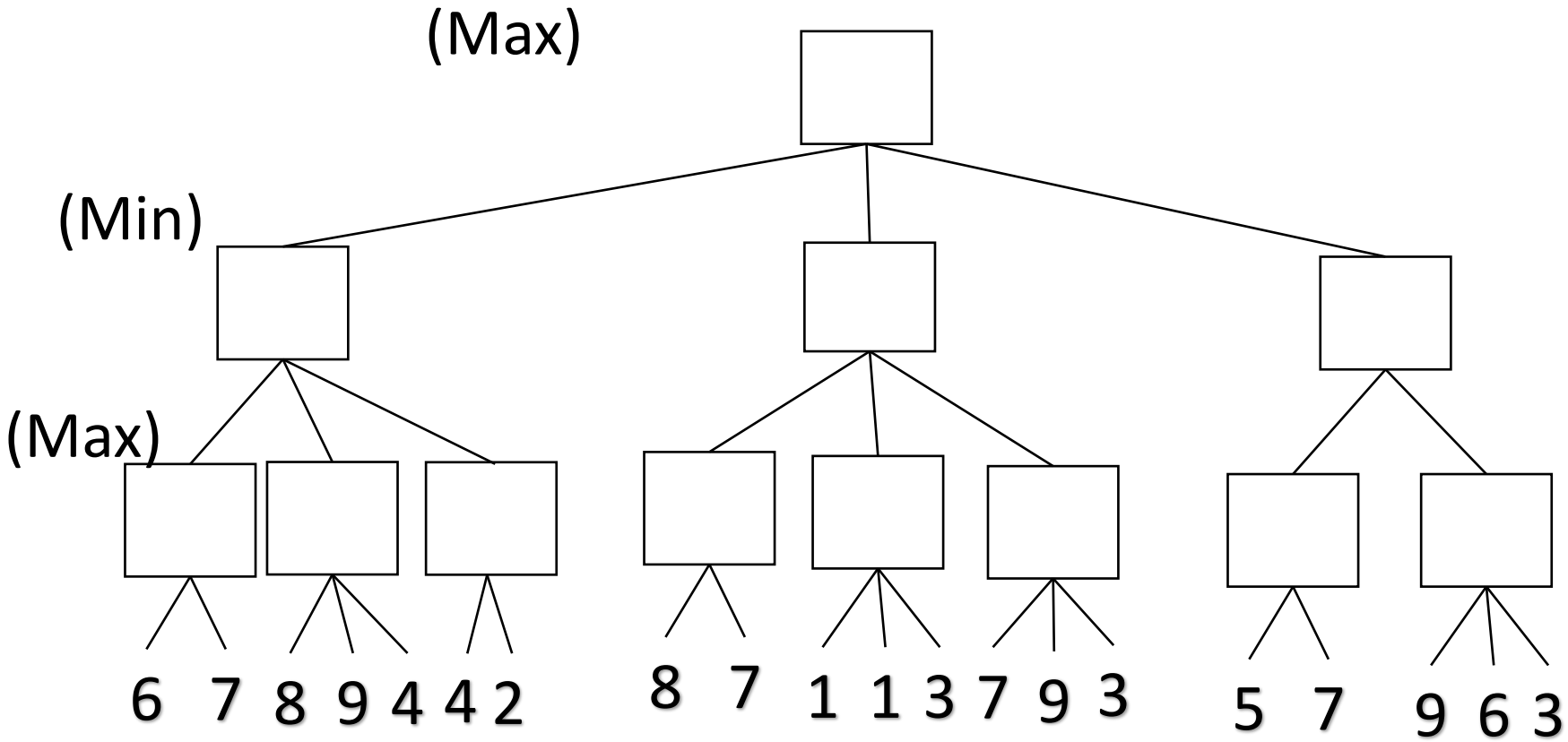
```
function MINIMAX-SEARCH(game, state) returns an action
    player ← game.TO-MOVE(state)
    value, move ← MAX-VALUE(game, state)
    return move

function MAX-VALUE(game, state) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← −∞
    for each a in game.ACTIONS(state) do
        v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
        if v2 > v then
            v, move ← v2, a
    return v, move

function MIN-VALUE(game, state) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← +∞
    for each a in game.ACTIONS(state) do
        v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
        if v2 < v then
            v, move ← v2, a
    return v, move
```

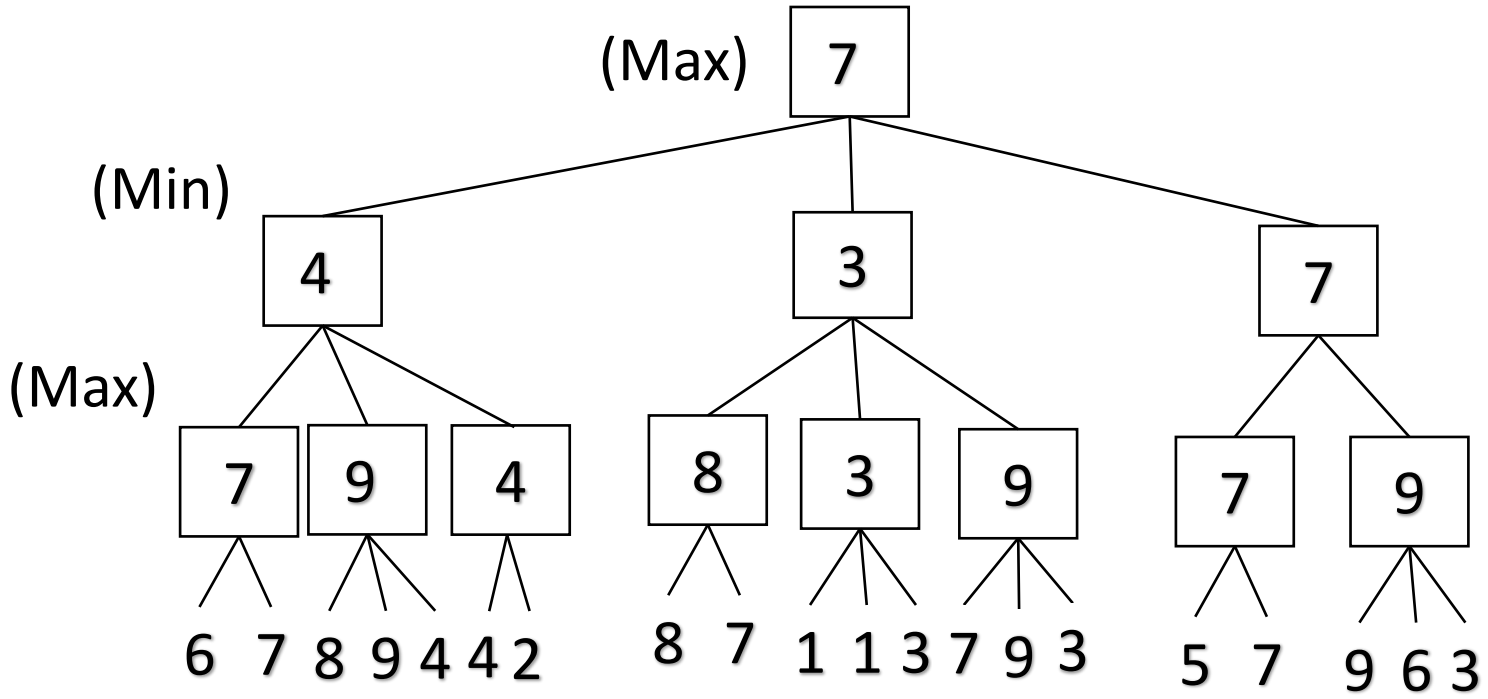
Quick Quiz

- Perform minimax search



Quick Quiz

- Perform minimax search

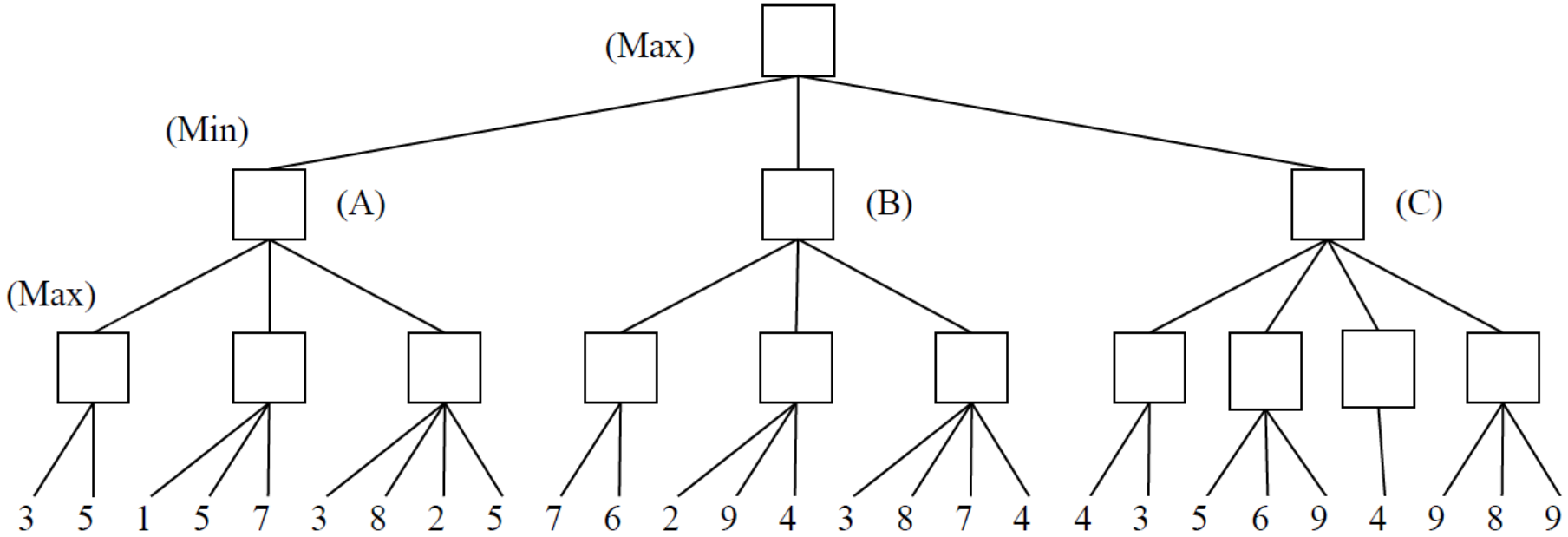


Quick Quiz

1.a. Fill in each blank square with the proper mini-max search value.

1.b. What is the best move for Max? (write A, B, or C) _____

1.c. What score does Max expect to achieve? _____



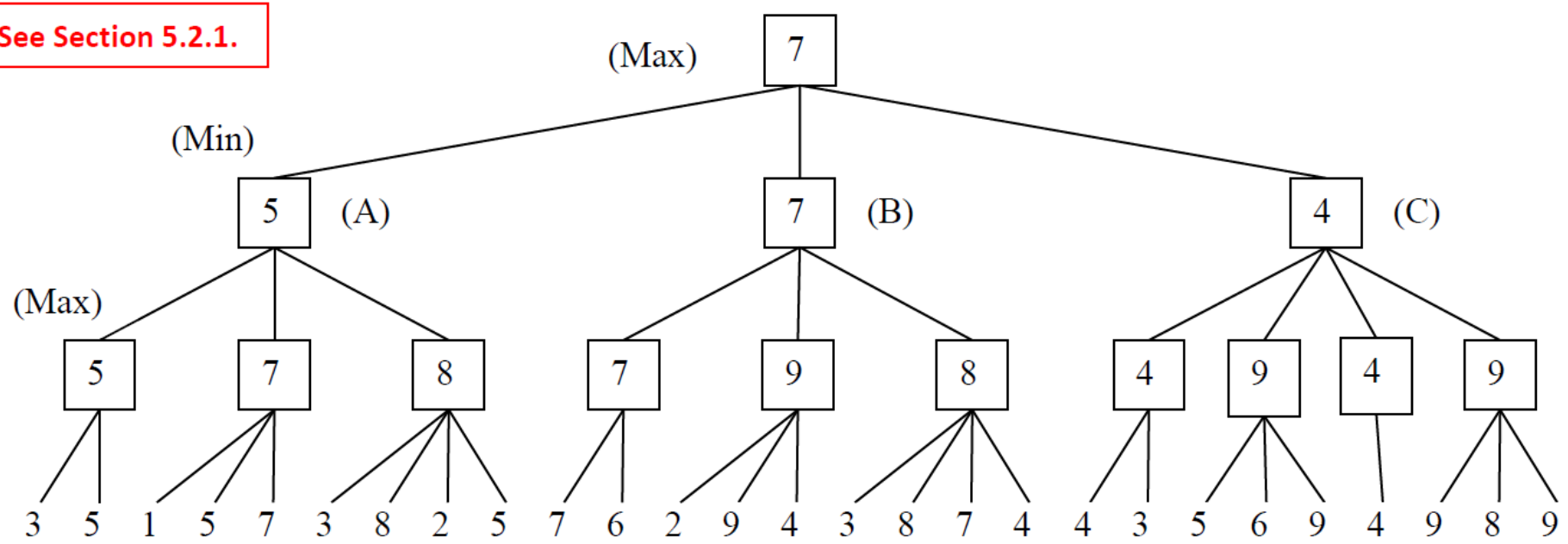
Quick Quiz

1.a. Fill in each blank square with the proper mini-max search value.

1.b. What is the best move for Max? (write A, B, or C) B

1.c. What score does Max expect to achieve? 7

See Section 5.2.1.



Minimax Search Properties

- Performs a depth-first search of the game tree
- Say m is the maximum depth in the game tree
- Space complexity: $O(bm)$
- Time complexity: $O(b^m)$
- Tic-tac-toe
  - $b \approx 5$, 9 ply game
  - $5^9 = 1{,}953{,}125$ <- Easy!
- Checkers
  - $\approx 10^{20}$ positions
  - Dozens of computers have been working since 1989 to find a brute force solution
  - Checkers finally solved in 2007 (DOI: 10.1126/science.1144079)
- Chess
  - $b \approx 35$, Typical game is 100 ply
  - $35^{100} \approx 10^{154}$ <- Not feasible!

Minimax Search Properties

- Minimax search assumes the opponent will play optimally
  - That is, that your opponent will always pick the action that will minimize your utility
- If you suspect your opponent has a weakness, minimax will not exploit it

Outline

- Background
- Minimax search
- Cutting off search
- Alpha-beta pruning
- Alpha-Go

Cutting off Search

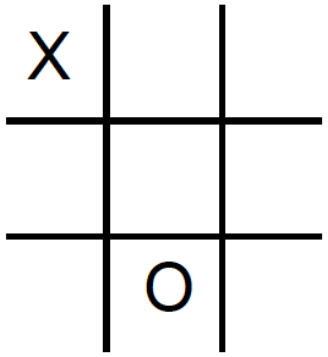
- At non-terminal nodes, we can use an evaluation/heuristic function to determine how good a state is
  - Terminal nodes: utility function
  - Non-terminal nodes: evaluation function
- Evaluation function should be fast to compute
- Does not have to be perfect
- Should be scaled correctly
  - If evaluation function is better than utility for winning or worse than the utility for losing, this can lead to undesirable behavior

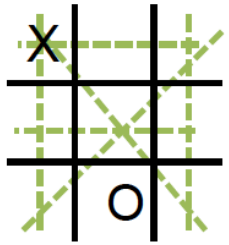
Evaluation Function for Tic-Tac-Toe

- Ideas for Tic-Tac-Toe?

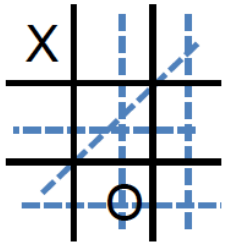
Evaluation Function for Tic-Tac-Toe

- Count the number of possible win lines



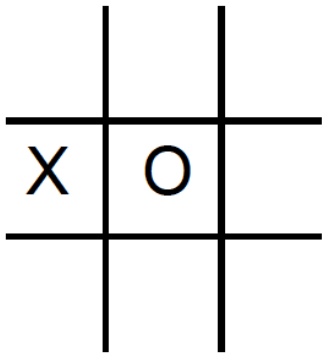


X has 6 possible win paths



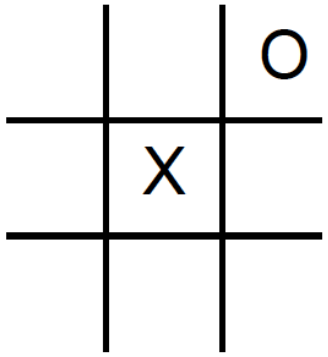
O has 5 possible win paths

E(s) = 6 – 5 = 1



X has 4 possible wins
O has 6 possible wins

E(n) = 4 – 6 = -2

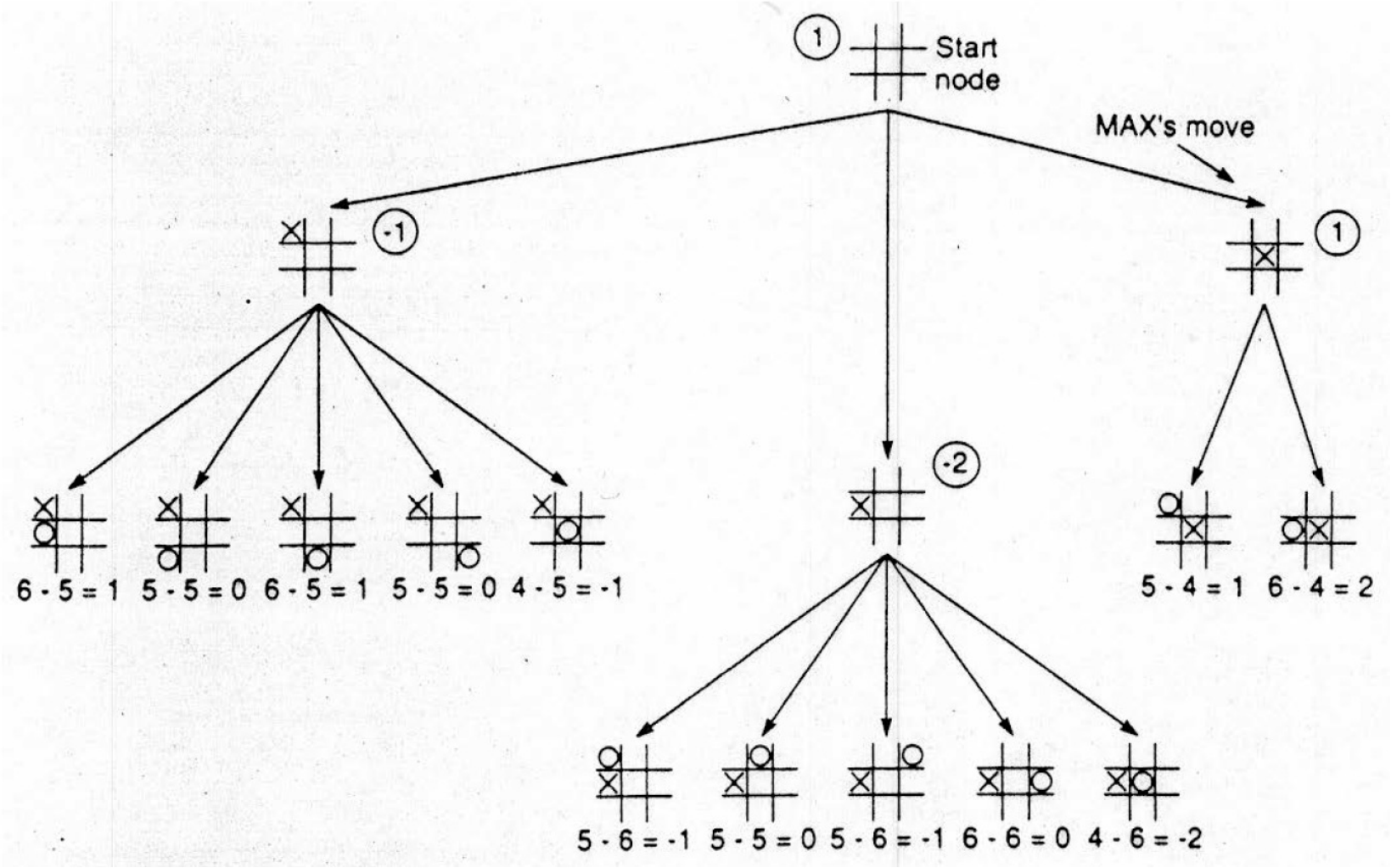


X has 5 possible wins
O has 4 possible wins

E(n) = 5 – 4 = 1

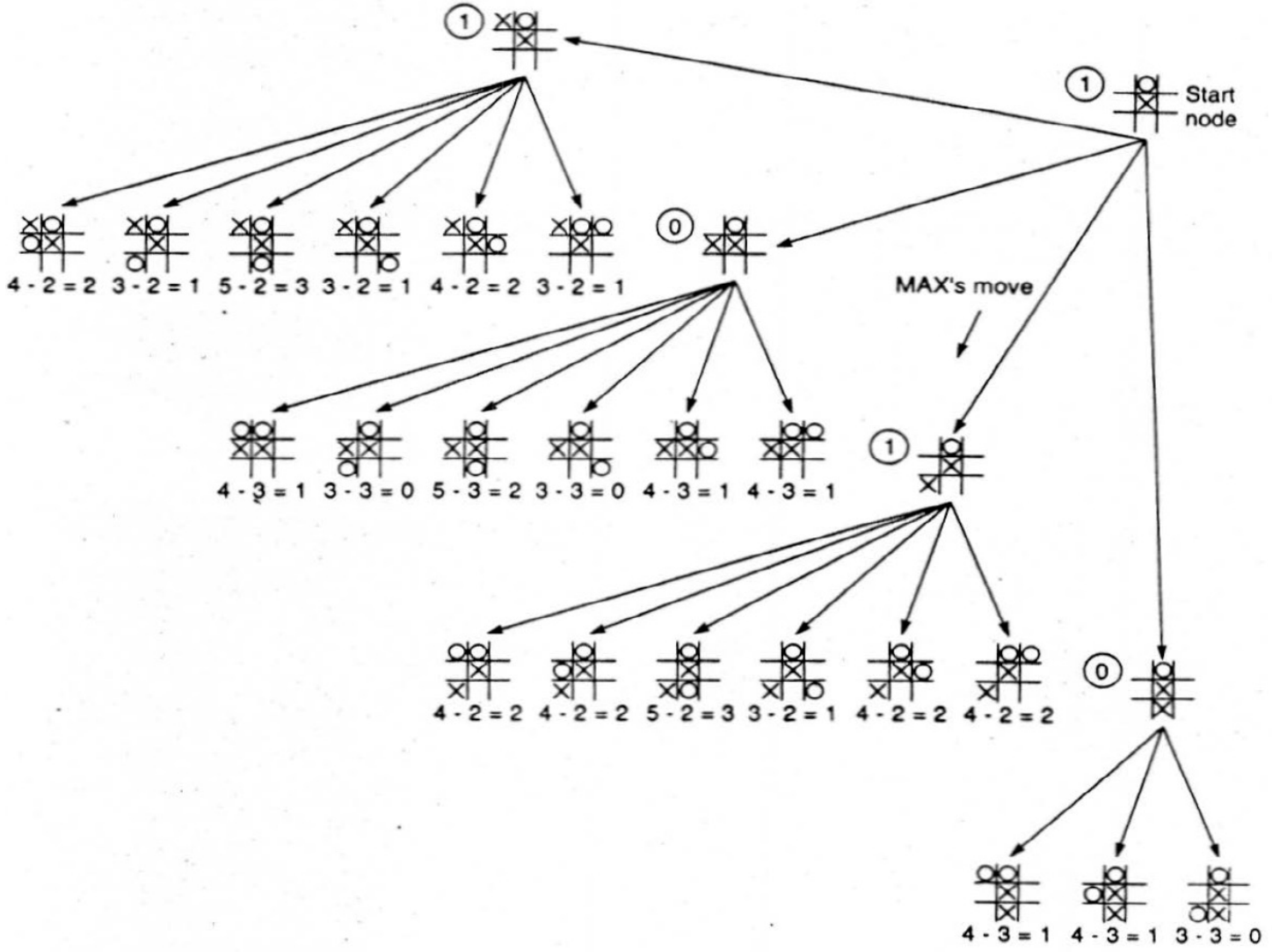
Two-ply Minimax

- We do not reach a terminal state, but we can still choose an action



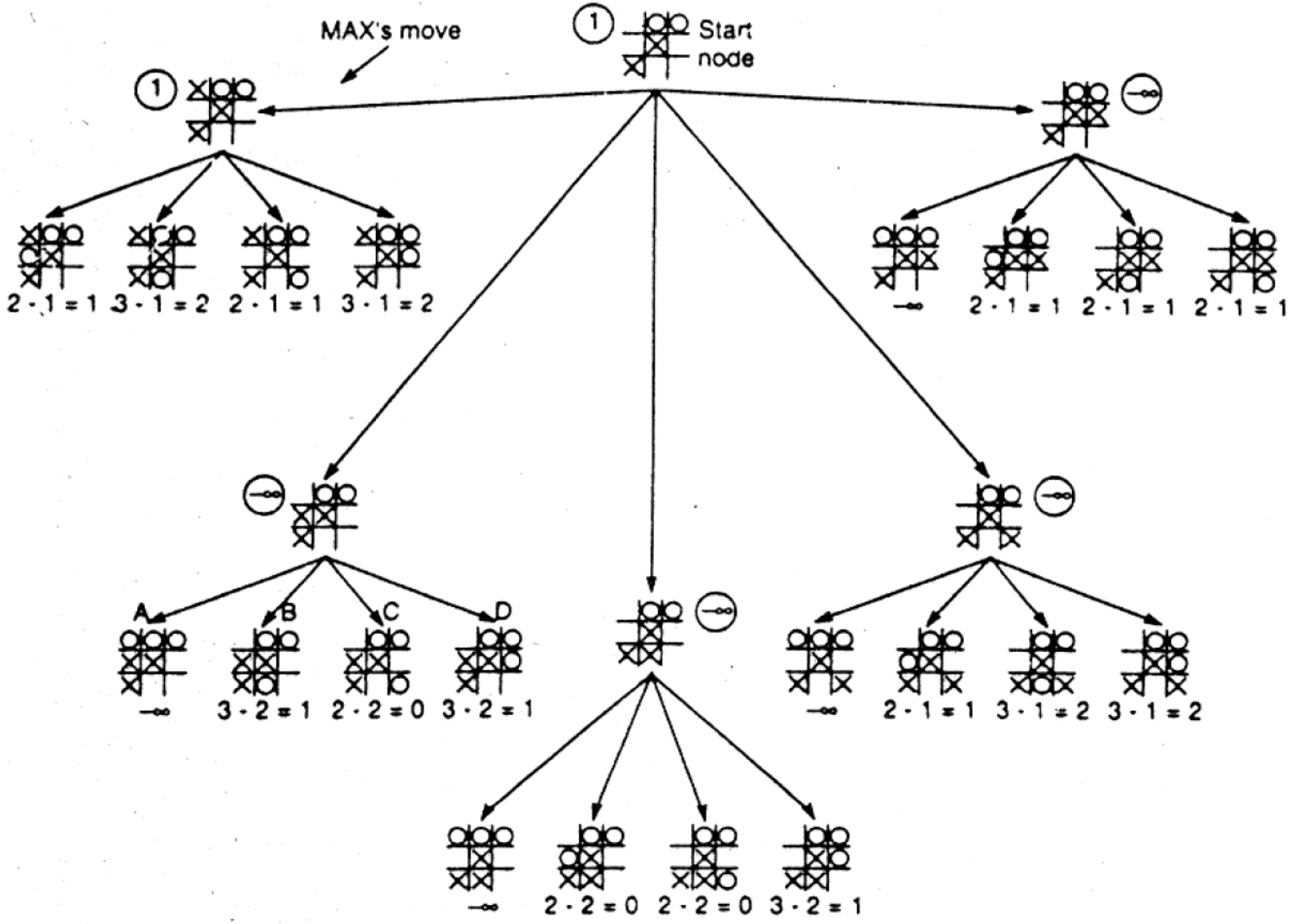
Two-ply Minimax

- Max’s second move

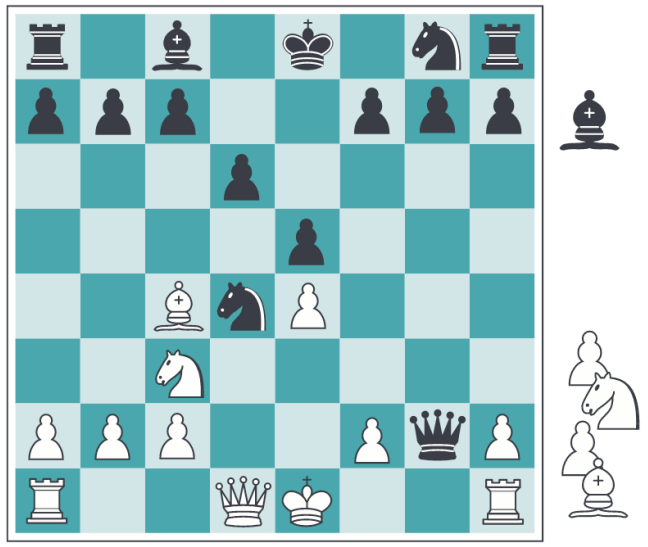


Two-ply Minimax

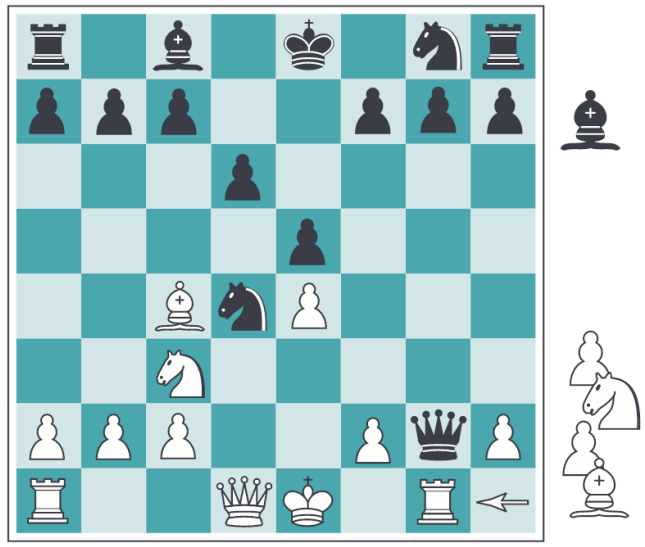
- Max's third move



Evaluation Functions for Chess



(a) White to move



(b) White to move

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

- One possibility is a linear sum of features

Deep Blue

- Deep Blue's evaluation function was initially written in a generalized form, with many to-be-determined parameters (e.g., how important is a safe king position compared to a space advantage in the center, etc.). Values for these parameters were determined by analyzing thousands of master games. The evaluation function was then split into 8,000 parts, many of them designed for special positions. The opening book encapsulated more than 4,000 positions and 700,000 grandmaster games, while the endgame database contained many six-piece endgames and all five and fewer piece endgames. An additional database named the “extended book” summarizes entire games played by Grandmasters. The system combines its searching ability of 200 million chess positions per second with summary information in the extended book to select opening moves.
- Before the second match, the program's rules were fine-tuned by grandmaster Joel Benjamin. The opening library was provided by grandmasters Miguel Illescas, John Fedorowicz, and Nick de Firmian. When Kasparov requested that he be allowed to study other games that Deep Blue had played so as to better understand his opponent, IBM refused, leading Kasparov to study many popular PC chess games to familiarize himself with computer gameplay.

https://en.wikipedia.org/wiki/Deep_Blue_(chess_computer)

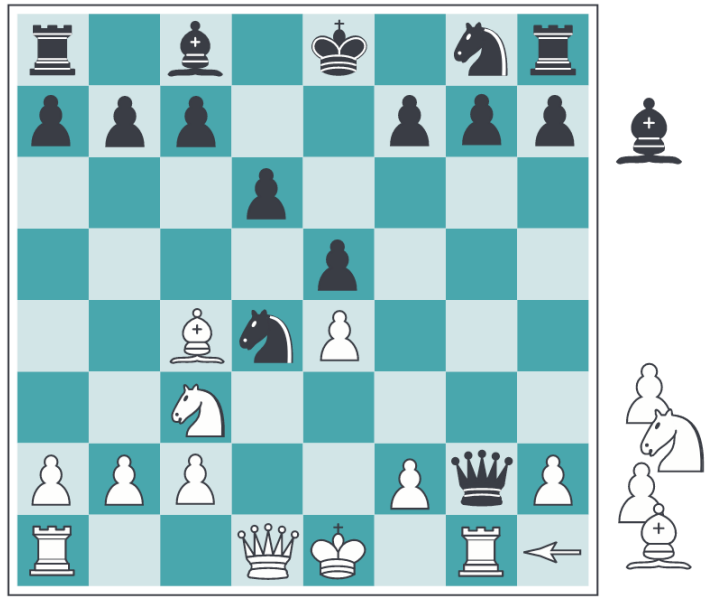
Connect Four Demonstration

When to Cut Off Search

- We can choose a particular maximum depth d
- What if you have to move before the search completes?
  - Iterative deepening search
  - When asked to move, return the action chosen at the depth of the deepest completed search

When to Cut Off Search

- Large fluctuations in the evaluation function can be a clue that search should continue down the current path
- Wait for quiescence
  - When there is no pending move that would wildly change the evaluation function
- If searching past the predefined cutoff, this extra quiescence search is sometimes restricted to only certain moves that could drastically change the outcome of the game



(b) White to move

A major change in the evaluation function could happen in the next move.

Horizon Effect

- Sometimes, there is a major even that exists just beyond the “horizon”
  - The evaluation function does not yet recognize that this will happen
- Allow for singular extensions: moves that look clearly better than all other moves
  - Moving the white rook to a1 to capture the bishop on a2

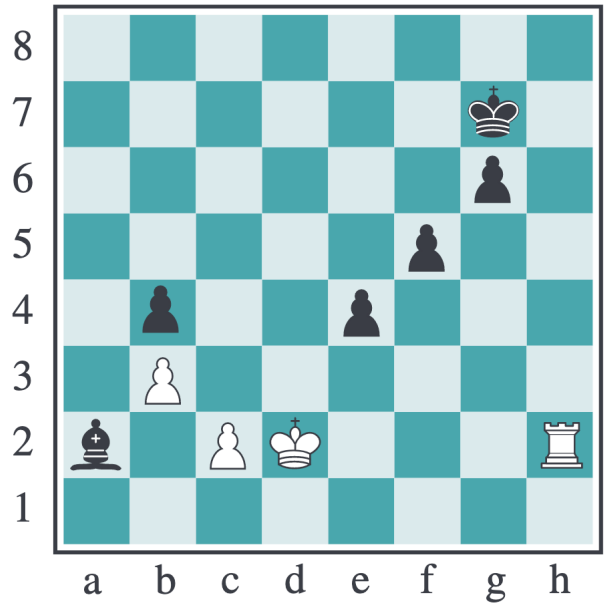


Figure 5.9 The horizon effect. With Black to move, the black bishop is surely doomed. But Black can forestall that event by checking the white king with its pawns, encouraging the king to capture the pawns. This pushes the inevitable loss of the bishop over the horizon, and thus the pawn sacrifices are seen by the search algorithm as good moves rather than bad ones.

Chess vs Go

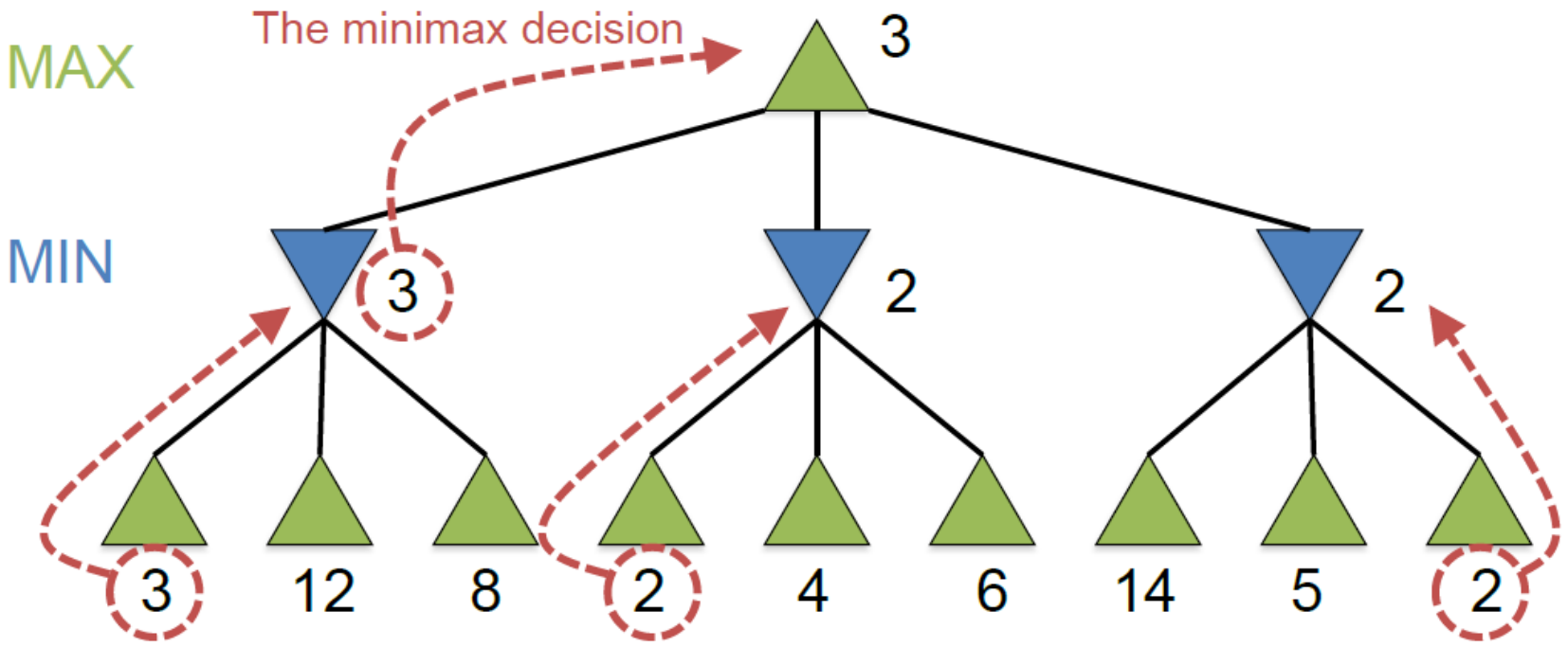
- Number of legal moves per position
  - Chess: ≈ 35
  - Go: ≈ 250
- Number of moves per game
  - Chess: ≈ 80
  - Go: ≈ 150
- Number of legal configurations
  - Chess: $\approx 10^{120}$
  - Go: $\approx 10^{170}$
- Empirically, it has been more difficult to evaluate a position in Go

Outline

- Background
- Minimax search
- Cutting off search
- Alpha-beta pruning
- Alpha-Go

Identifying Fruitless Paths

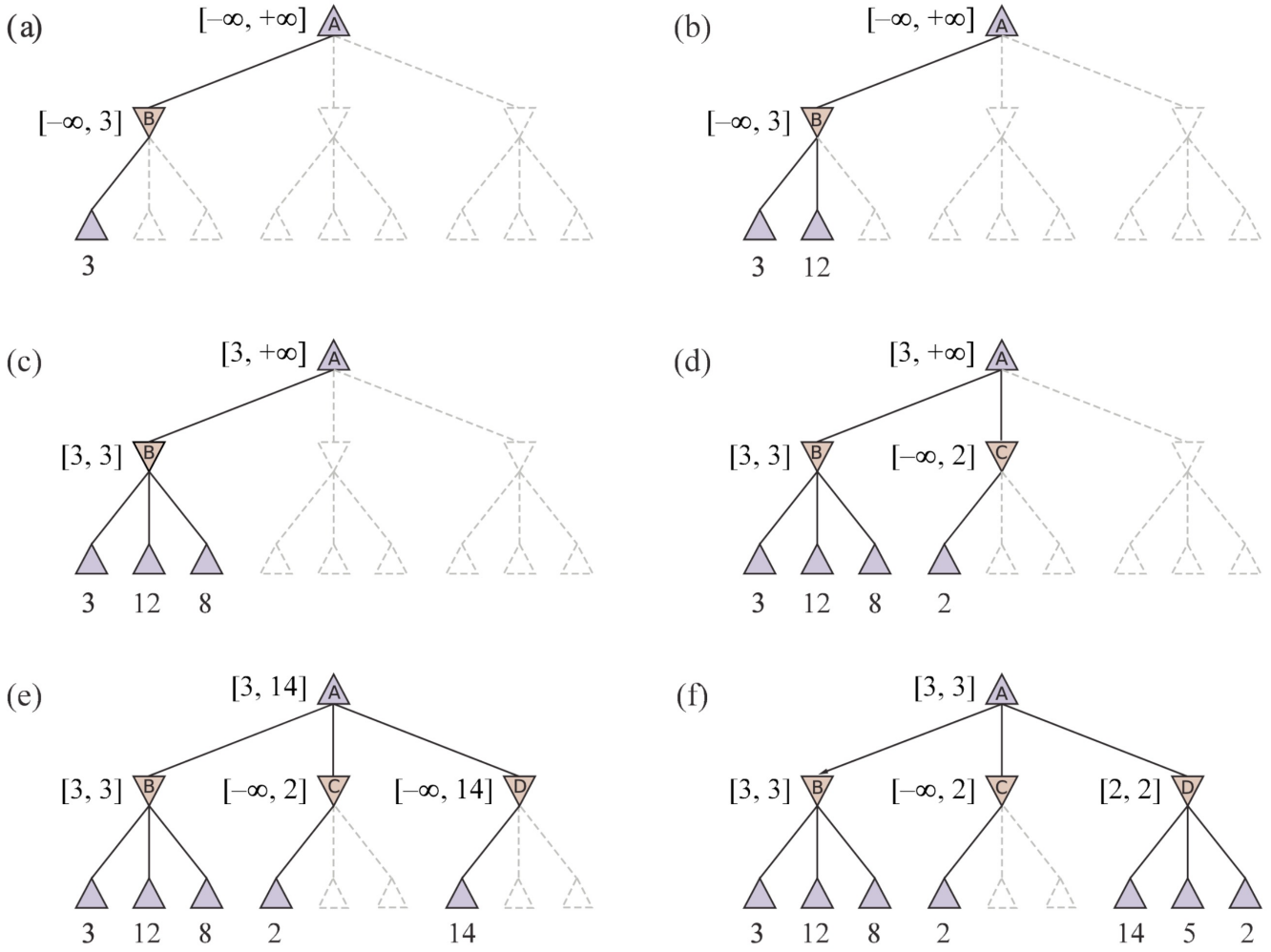
- Can we determine, before search has ended, what actions a player should never even consider?



Alpha-Beta Pruning: Implementation

- At any node in a tree, if a player has a better choice at some point previously in the game, then that node will not be reached
- We can prune nodes by keeping track of values α and β
- α Value of the highest utility choice that we have found so far at any choice point for MAX
  - We can get a utility of at least α
- β Value of the highest utility choice that we have found so far at any choice point for MIN
  - We can get a utility of at most β
- We prune when $\alpha \geq \beta$
  - When we can get, at least, α but the current path will give us, at most, β

Alpha-Beta Pruning: Example



- $\max(\min(3,12,8), \min(2, x, y), \min(14,5,2))$
- $= \max(3, \min(2, x, y), 2)$
- $= 3$
- It does not matter what x and y are because upper bound of middle min is 2.
- Move ordering matters
  - What if 2 was first on branch D?

Alpha-Beta Pruning

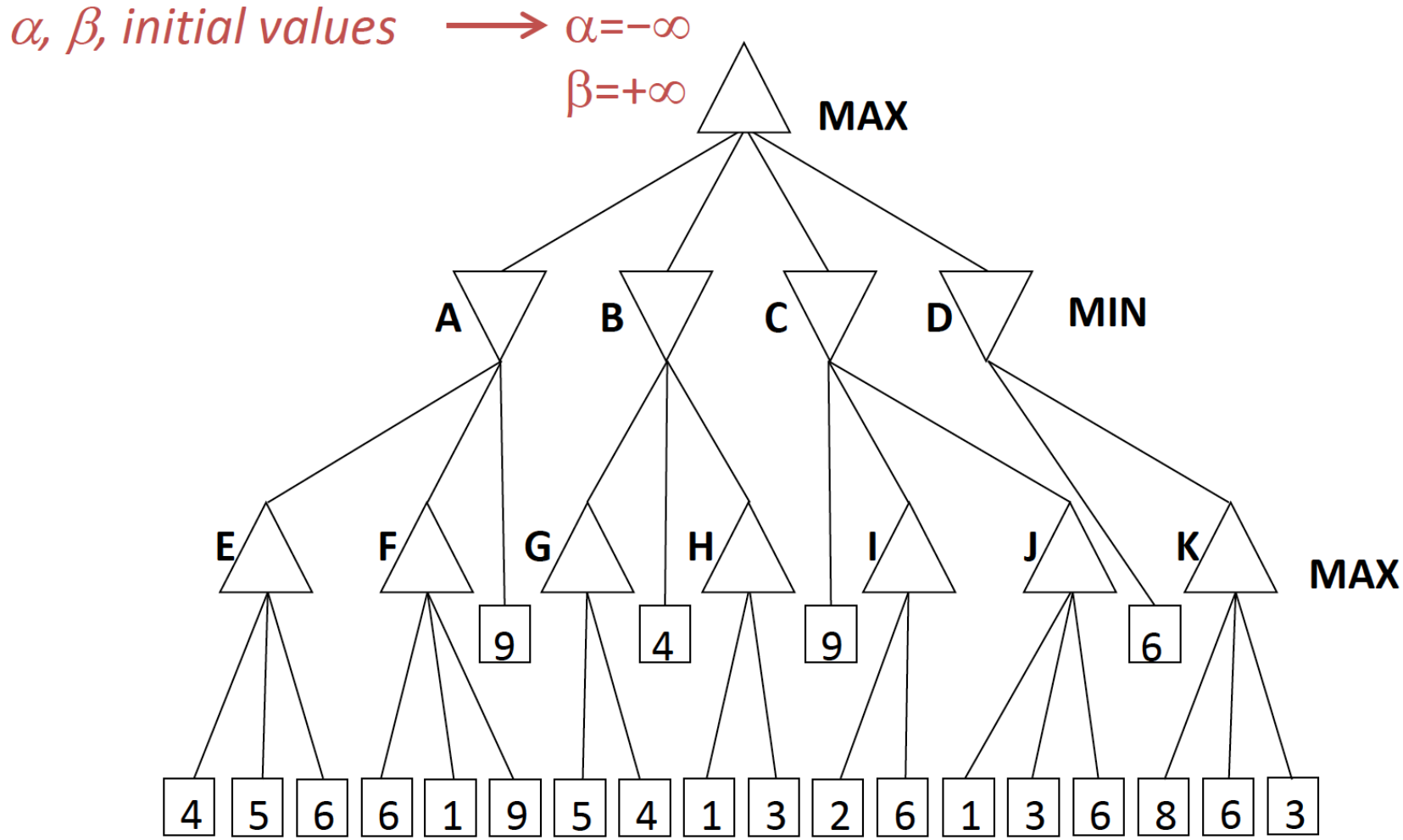
```
function ALPHA-BETA-SEARCH(game, state) returns an action
    player ← game.TO-MOVE(state)
    value, move ← MAX-VALUE(game, state, −∞, +∞)
    return move

function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← −∞
    for each a in game.ACTIONS(state) do
        v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
        if v2 > v then
            v, move ← v2, a
            α ← MAX(α, v)
        if v ≥ β then return v, move
    return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← +∞
    for each a in game.ACTIONS(state) do
        v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
        if v2 < v then
            v, move ← v2, a
            β ← MIN(β, v)
        if v ≤ α then return v, move
    return v, move
```

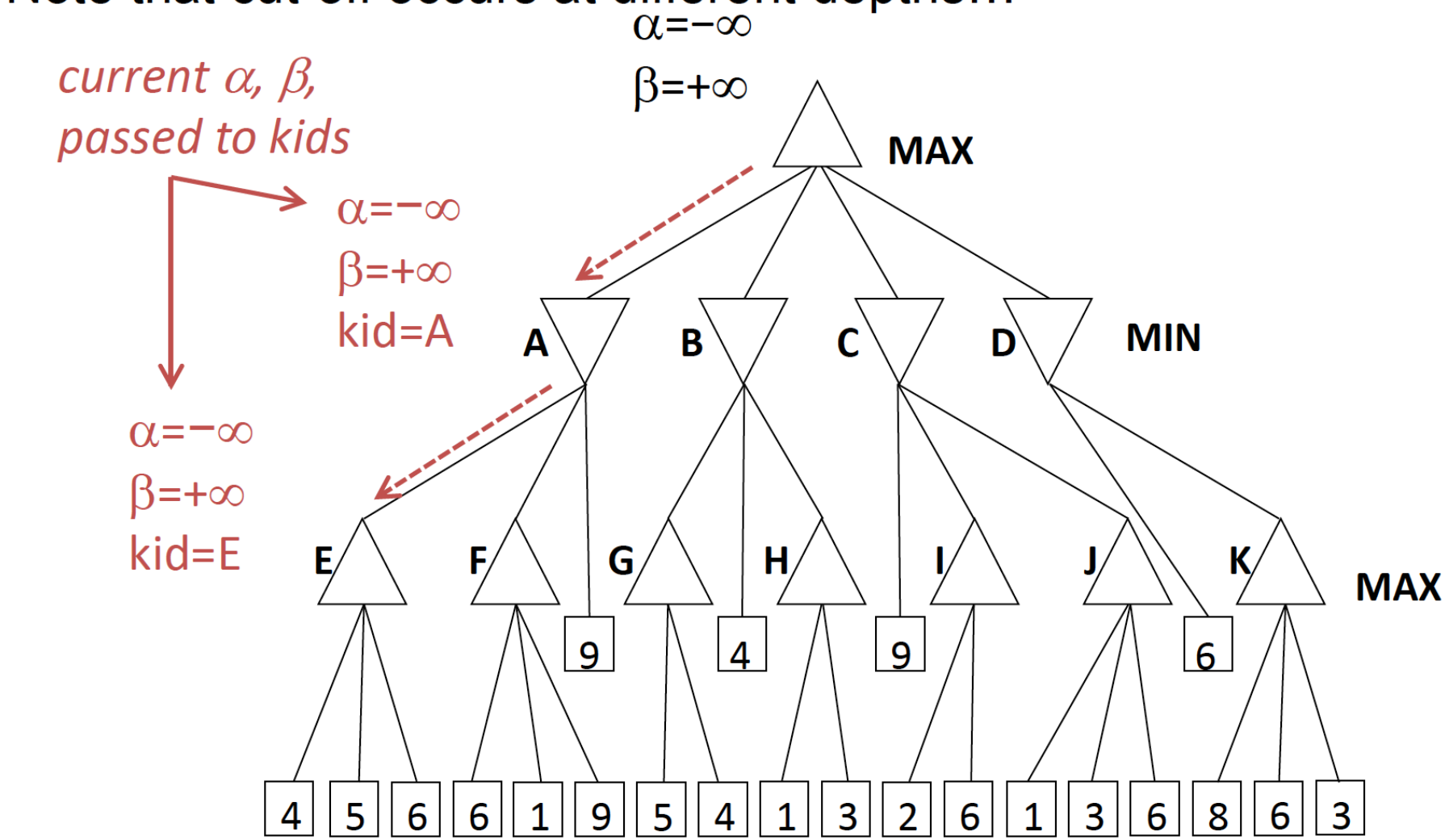
Longer Alpha-Beta Example

Branch nodes are labelel A..K for easy discussion

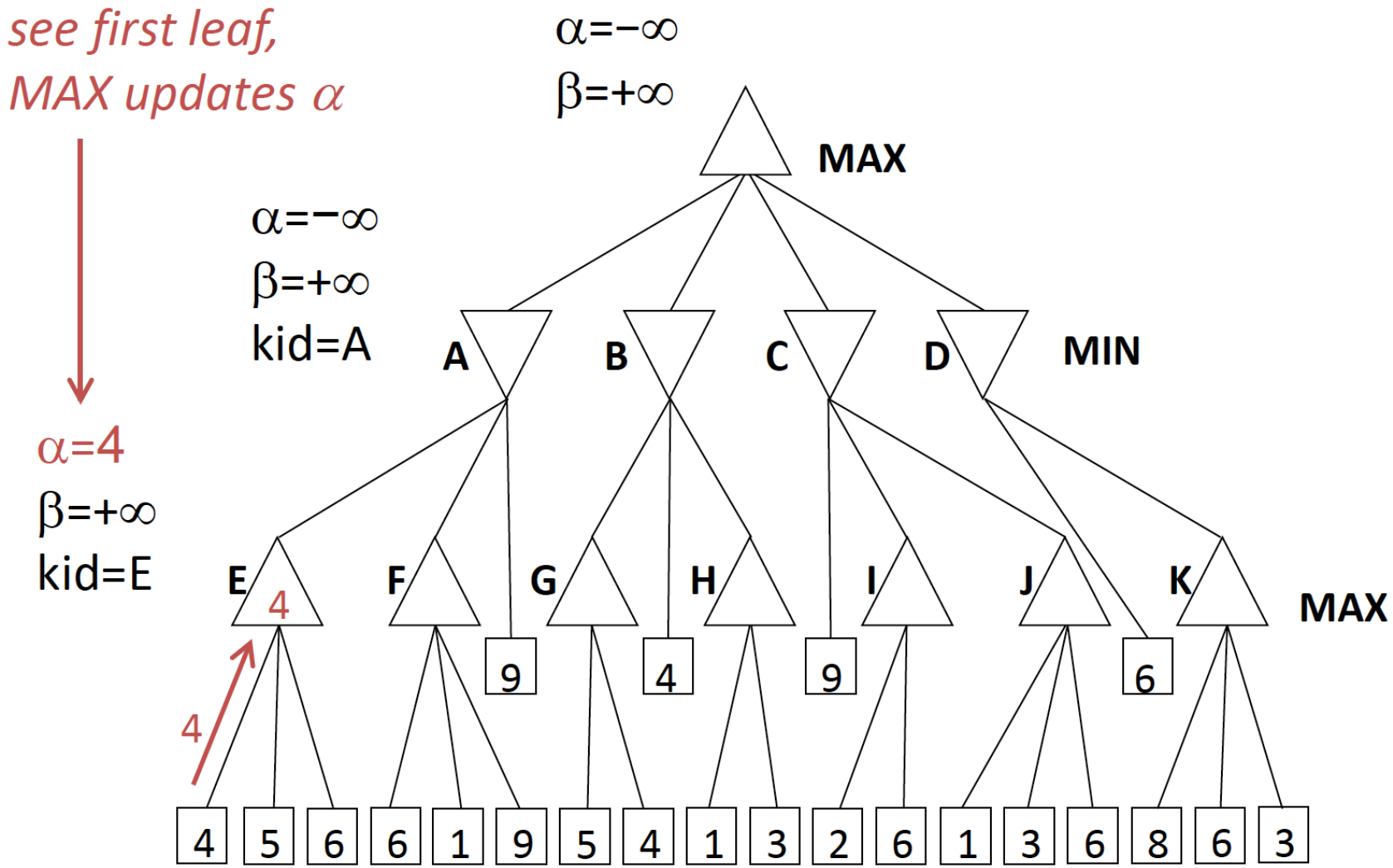


Longer Alpha-Beta Example

Note that cut-off occurs at different depths…

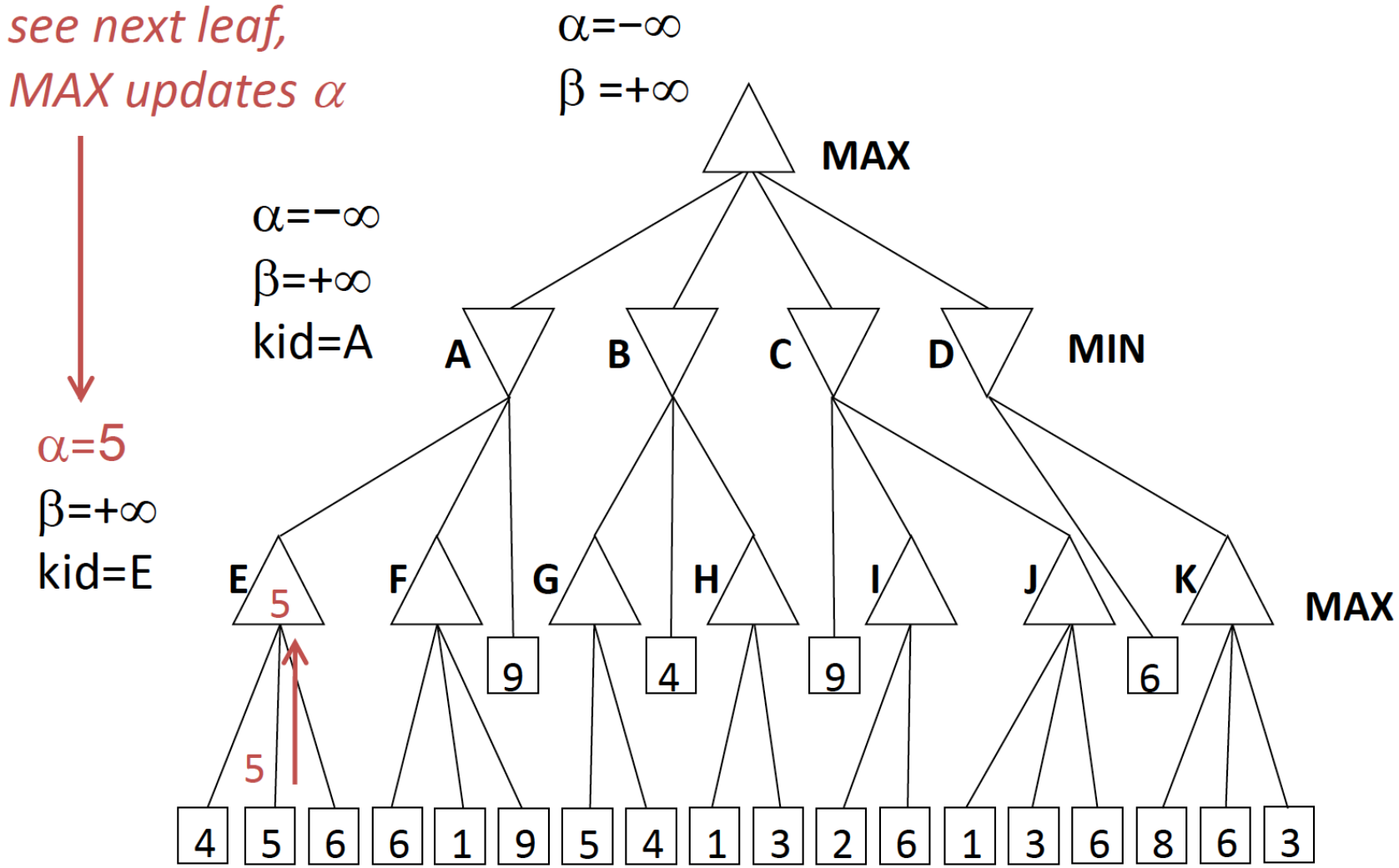


Longer Alpha-Beta Example

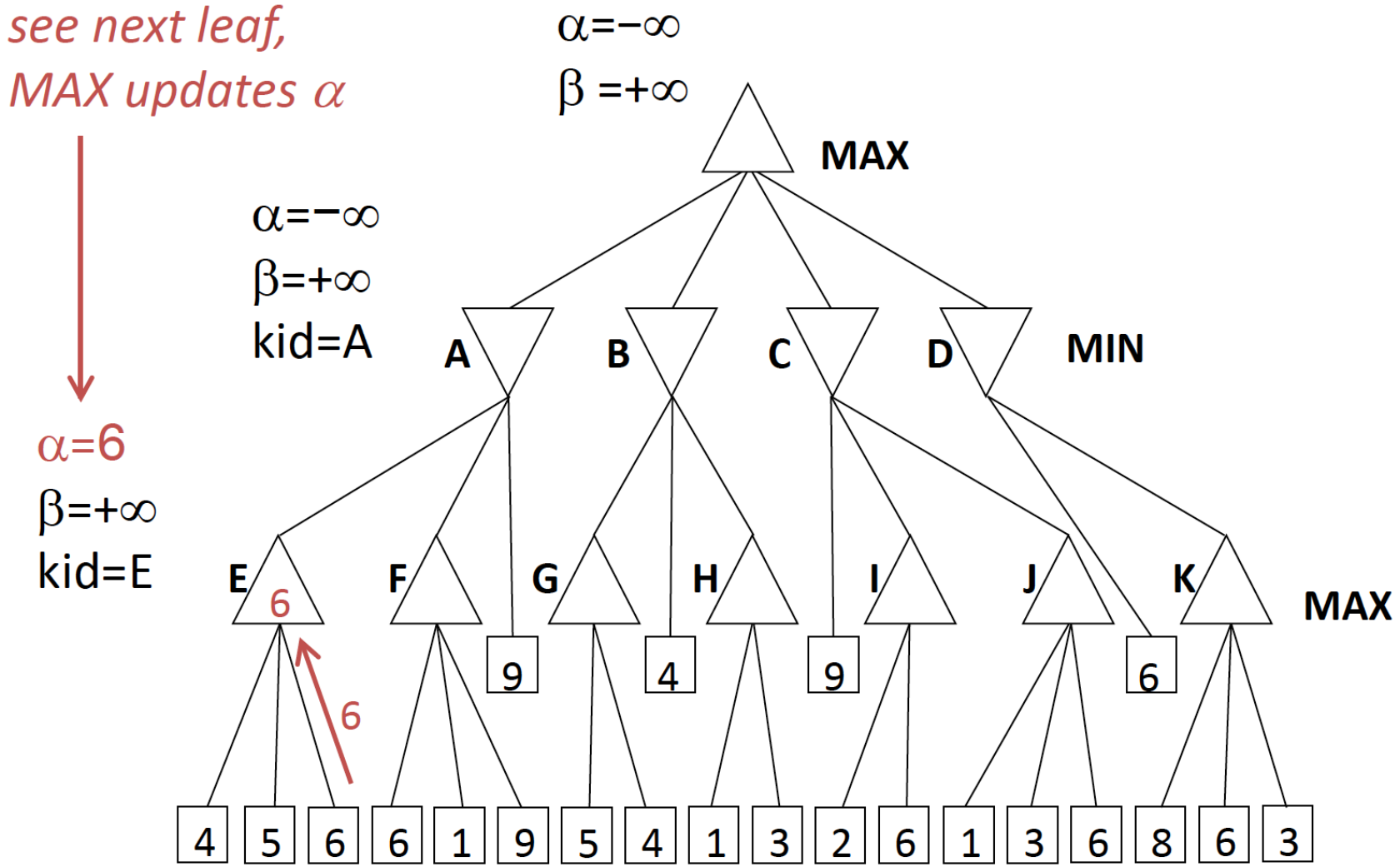


We also are running MiniMax search and recording node values within the triangles, without explicit comment.

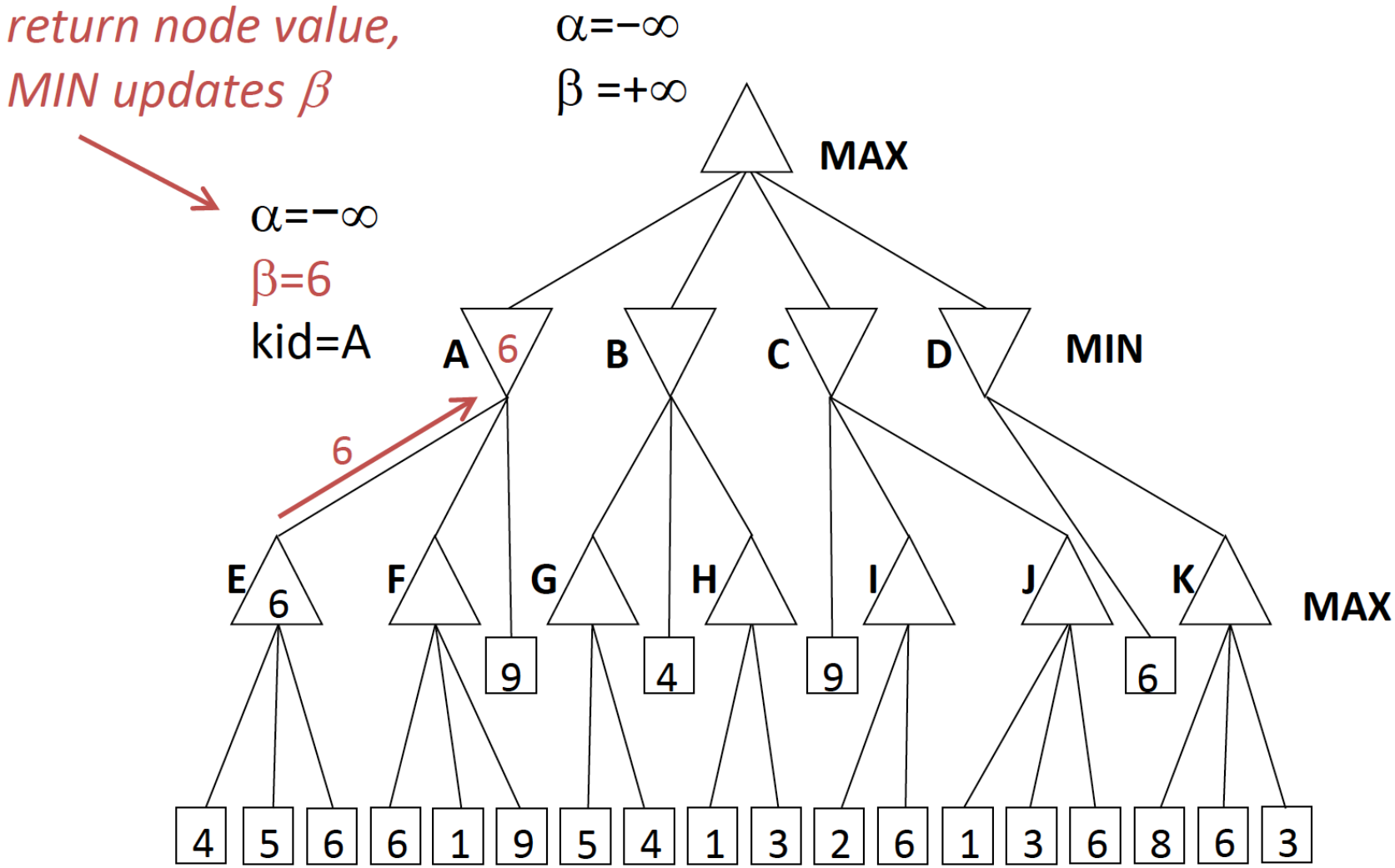
Longer Alpha-Beta Example



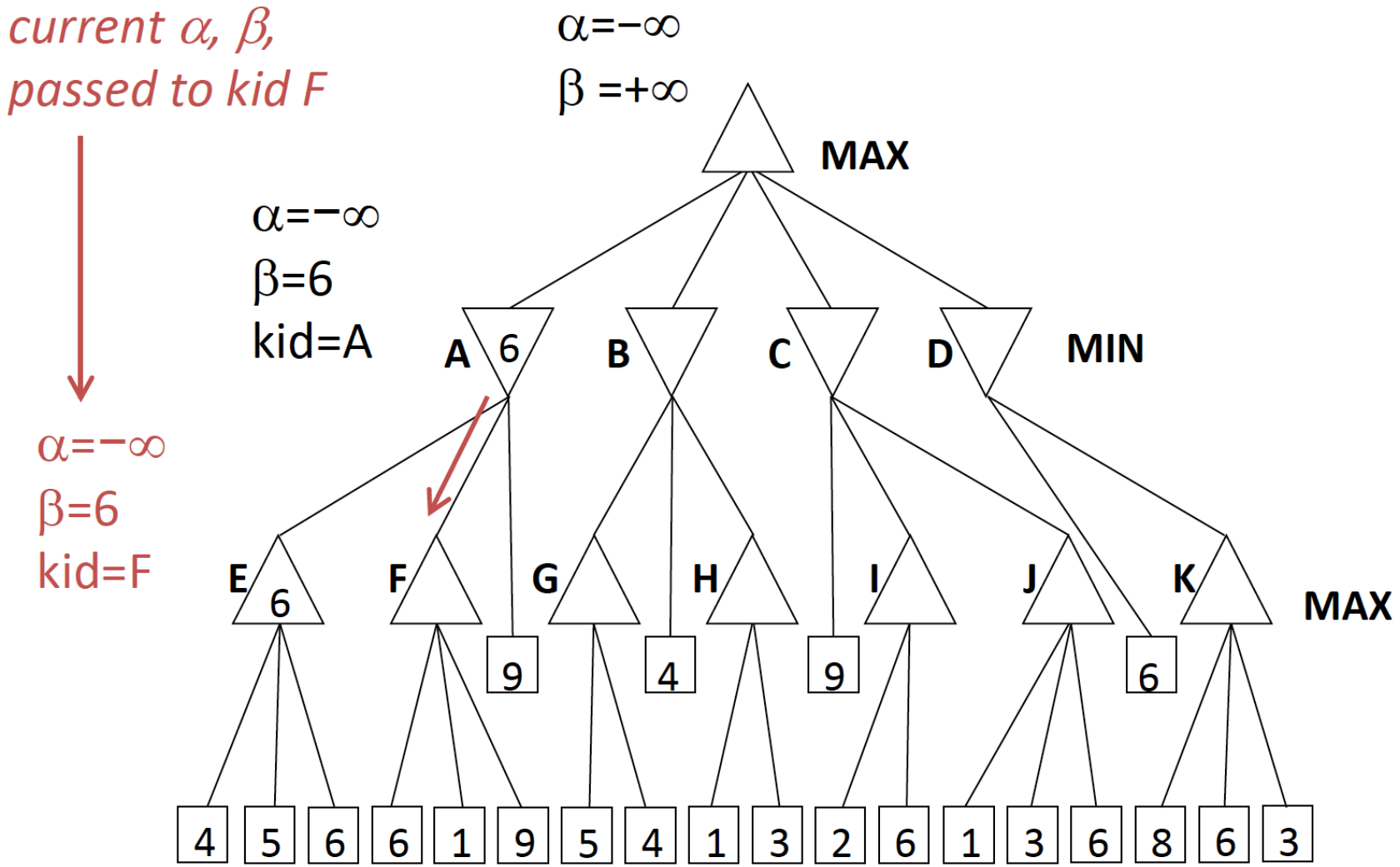
Longer Alpha-Beta Example



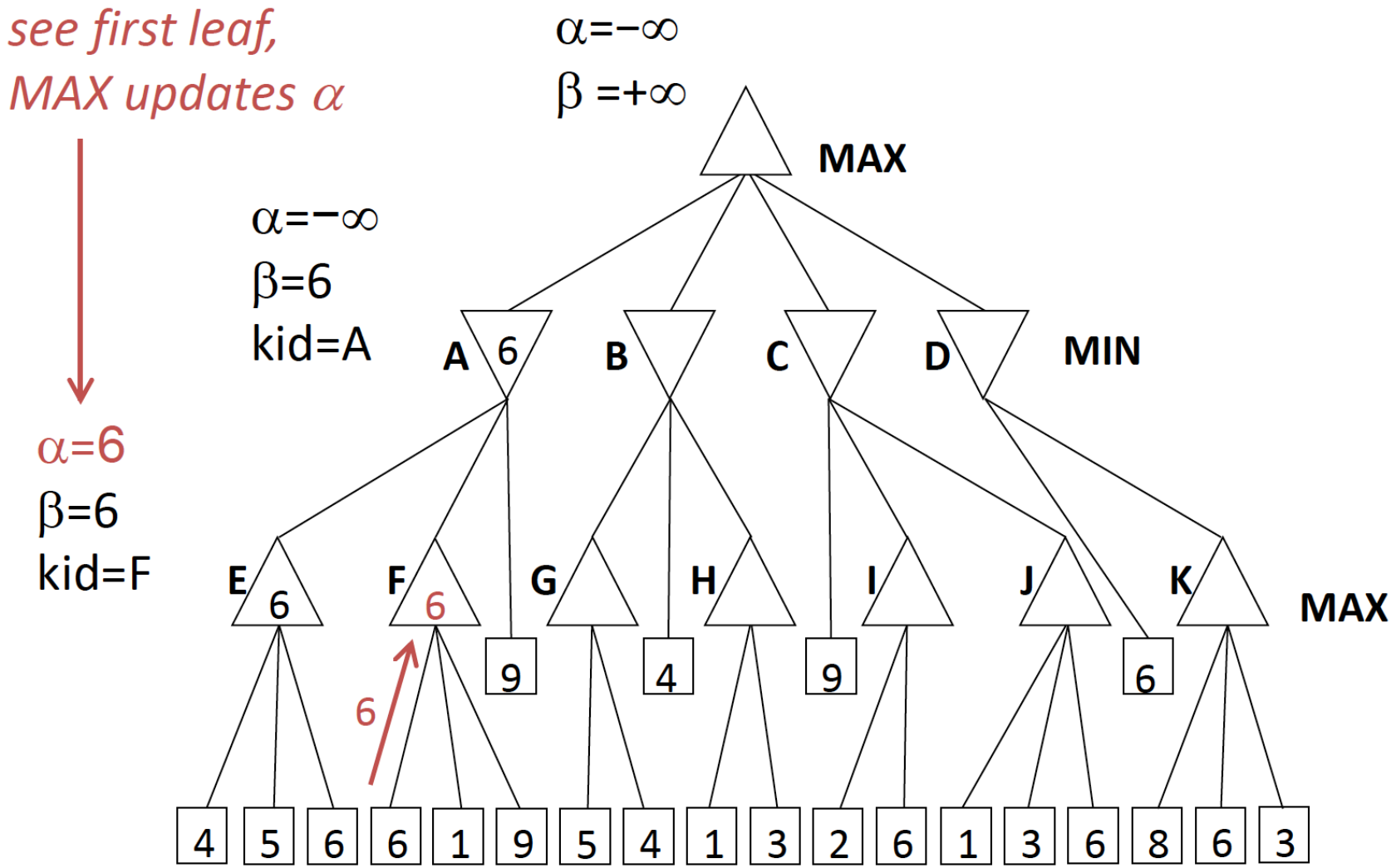
Longer Alpha-Beta Example



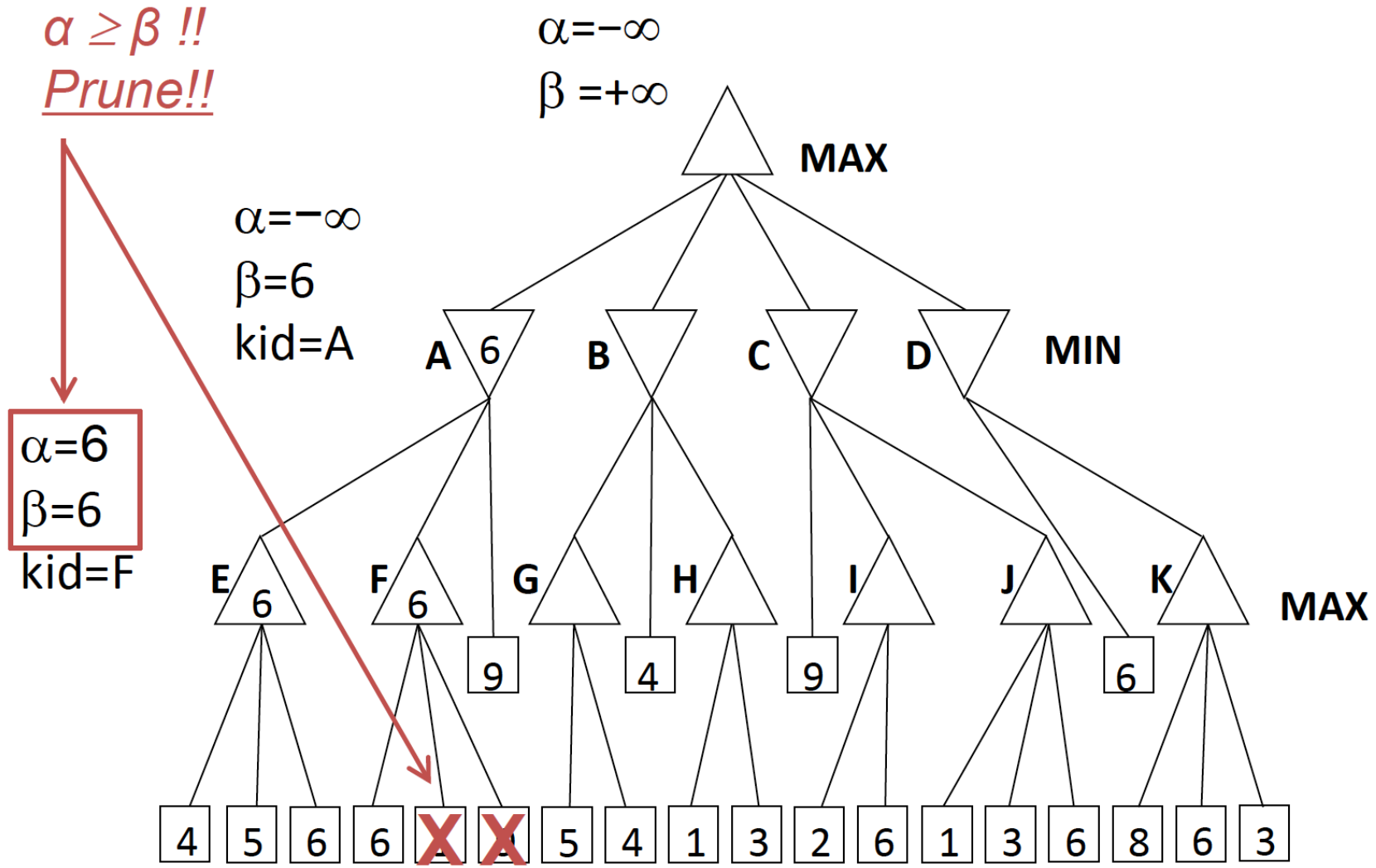
Longer Alpha-Beta Example



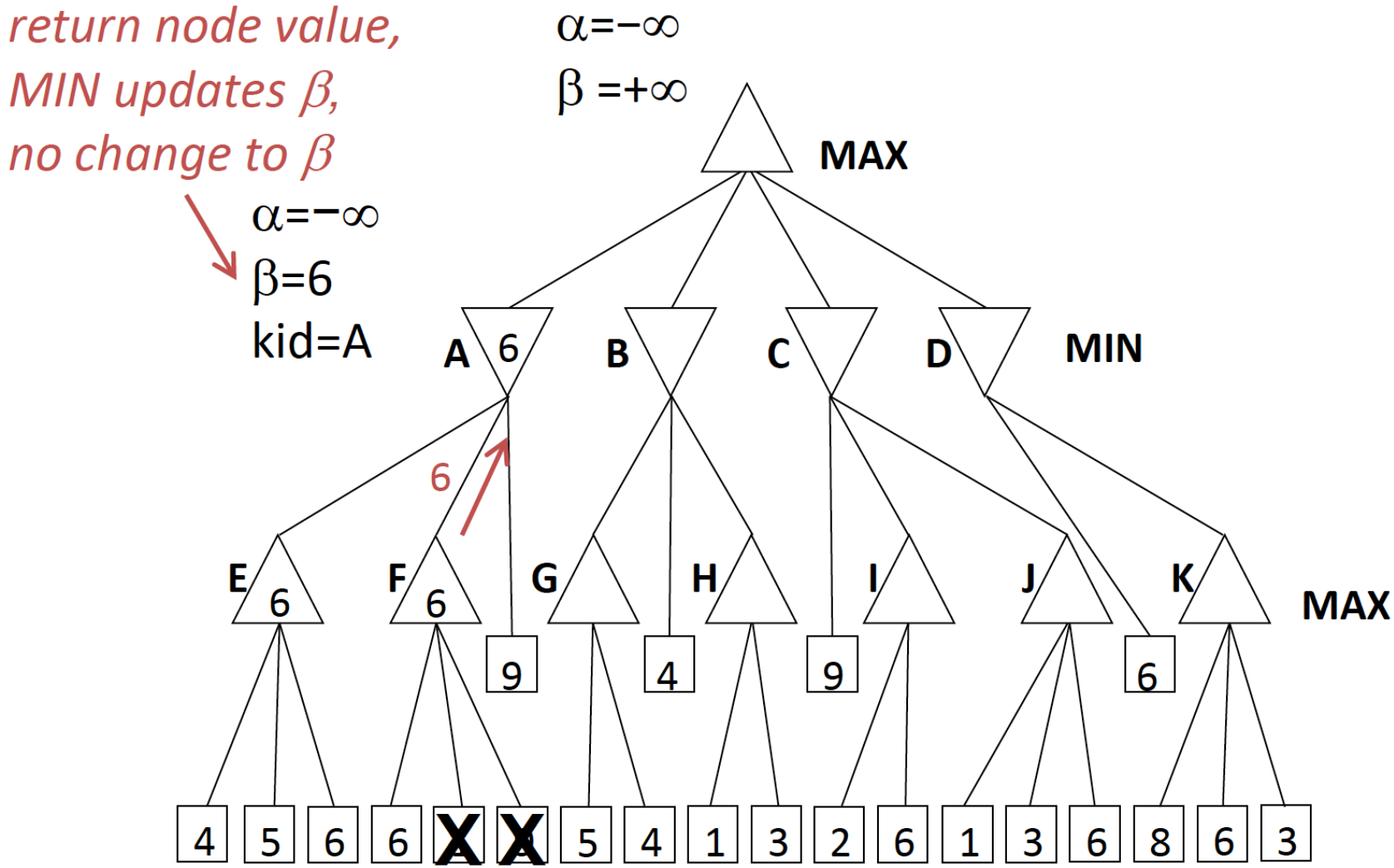
Longer Alpha-Beta Example



Longer Alpha-Beta Example

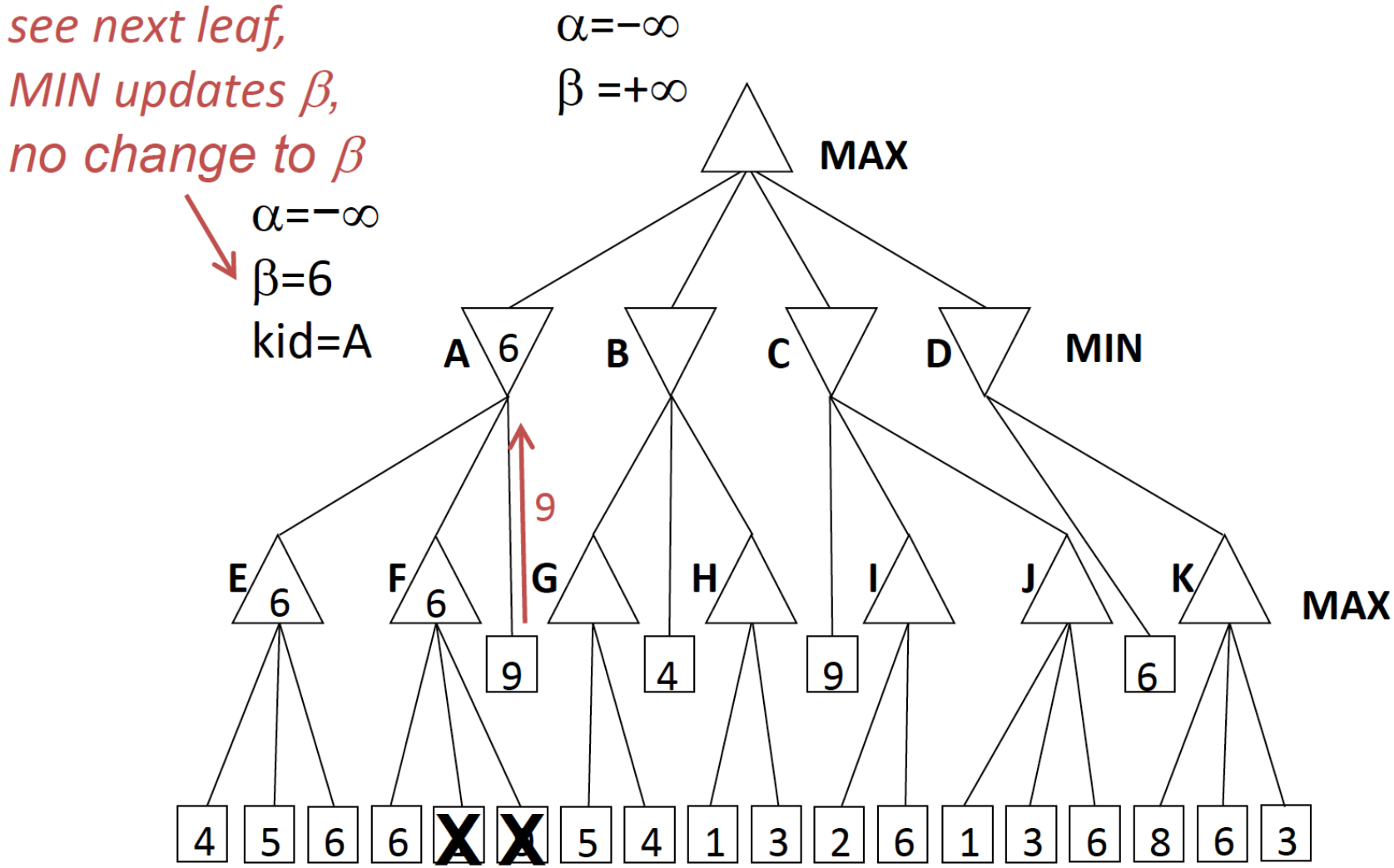


Longer Alpha-Beta Example



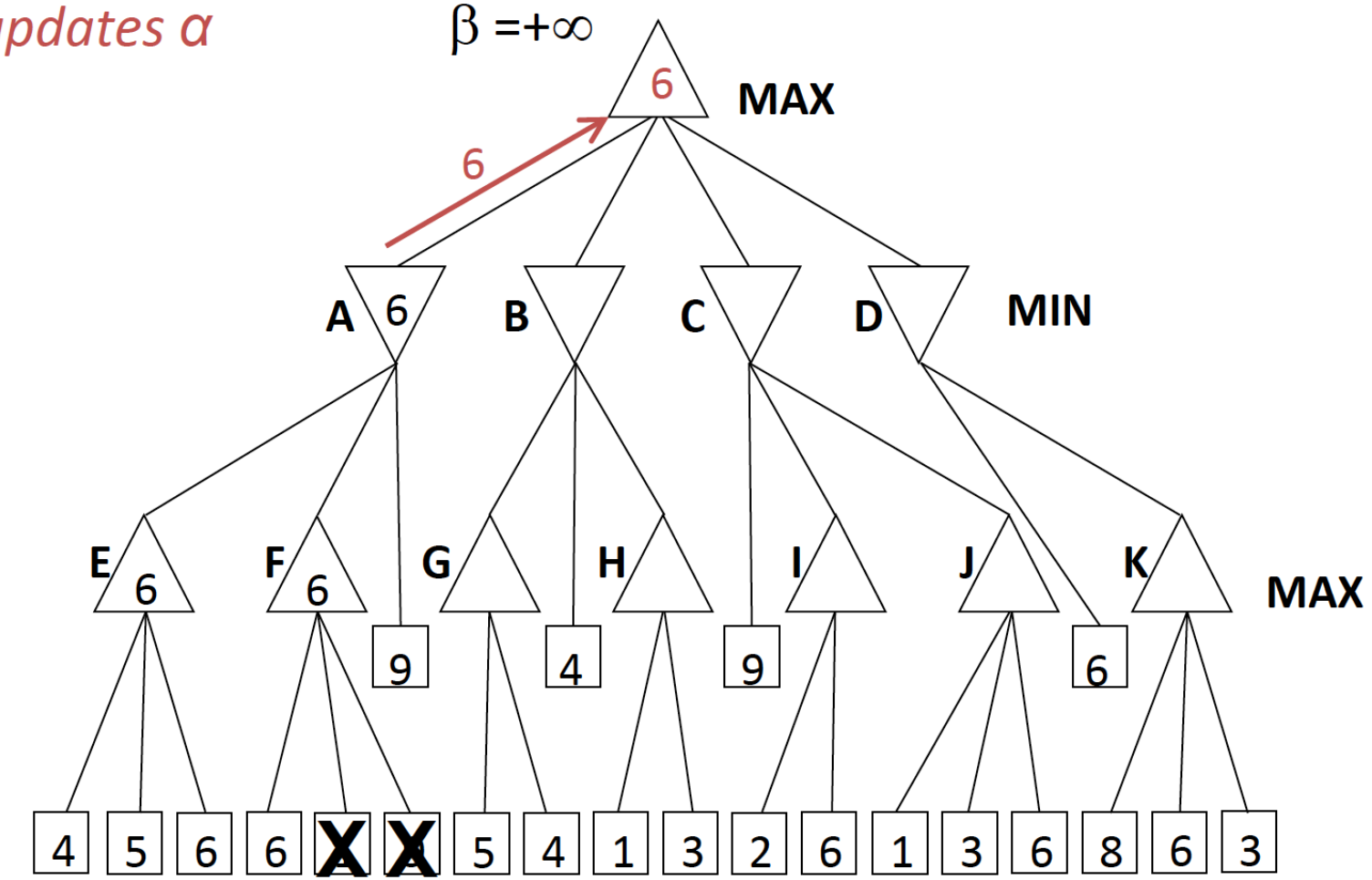
If we had continued searching at node F, we would see the 9 from its third leaf. Our returned value would be 9 instead of 6. But at A, MIN would choose E(=6) instead of F(=9). Internal values may change; root values do not.

Longer Alpha-Beta Example

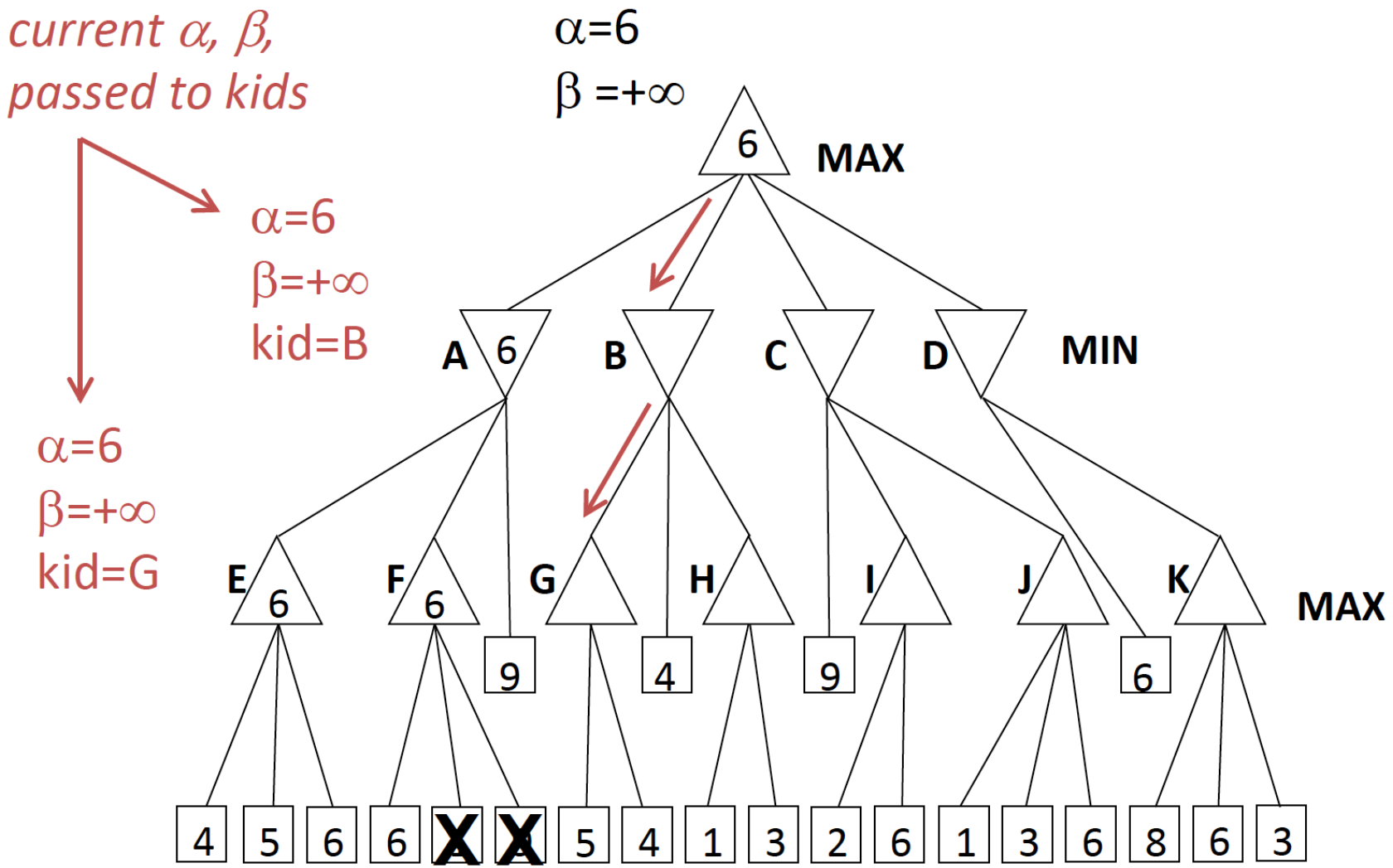


Longer Alpha-Beta Example

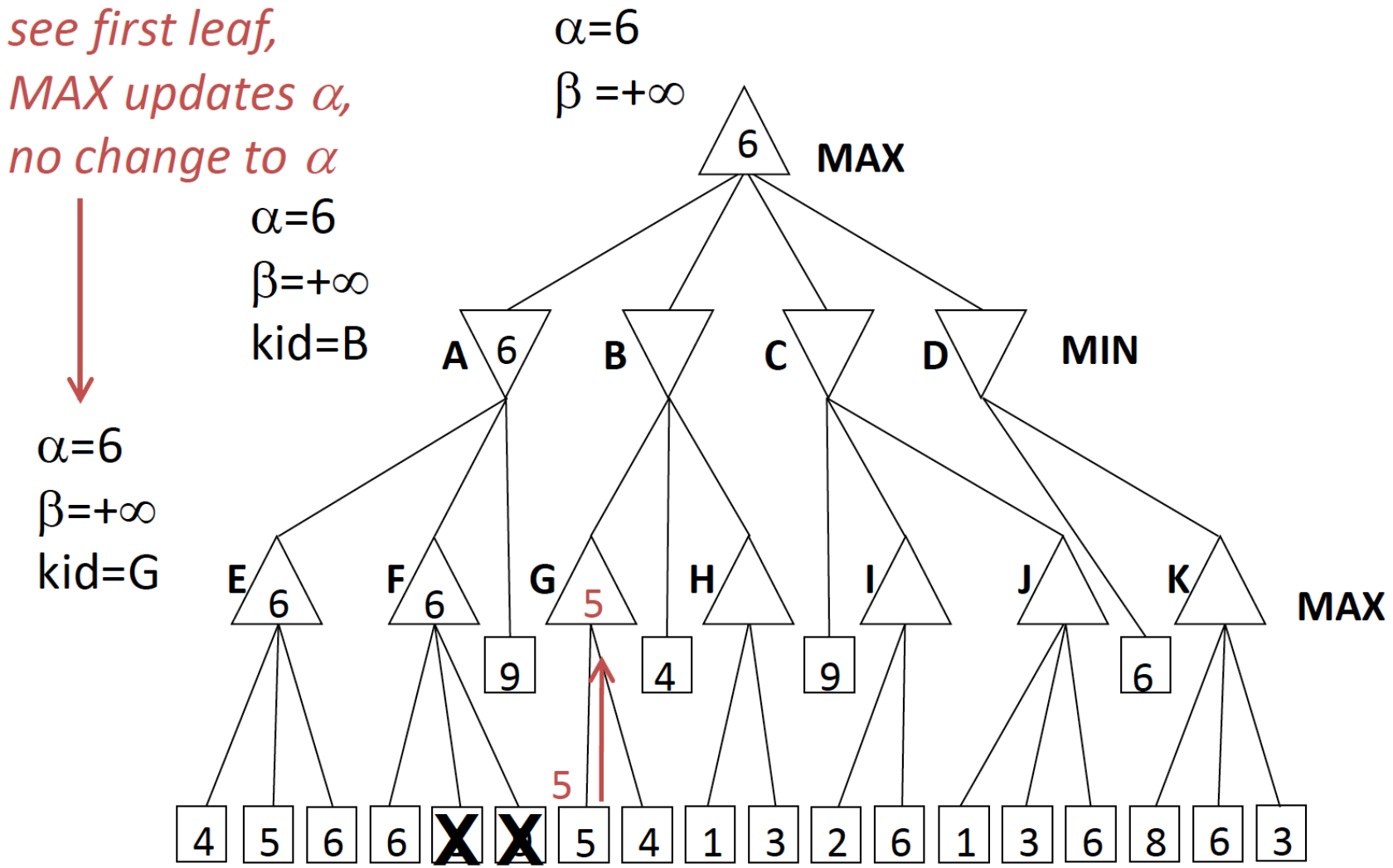
return node value, → α=6
MAX updates α



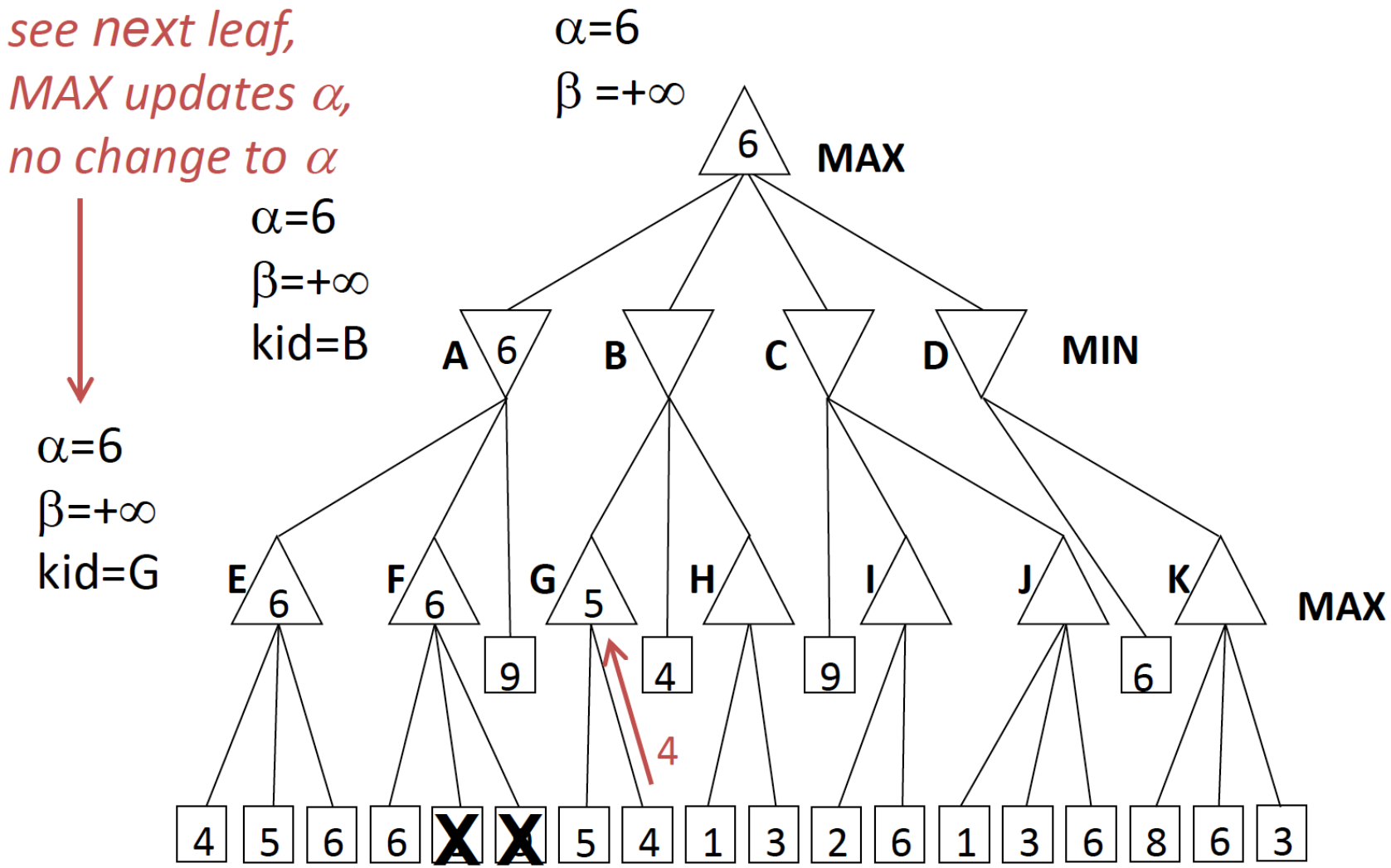
Longer Alpha-Beta Example



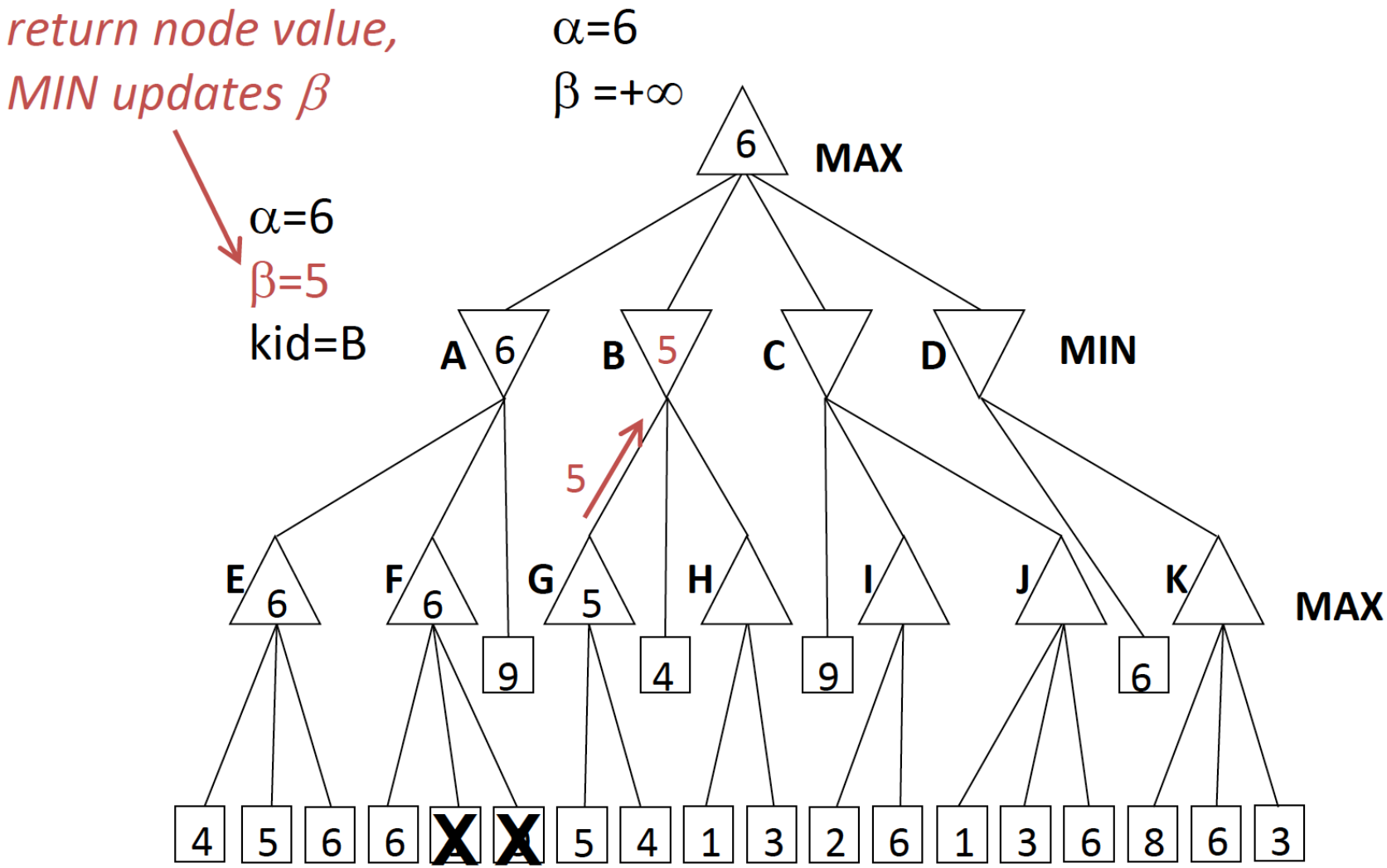
Longer Alpha-Beta Example



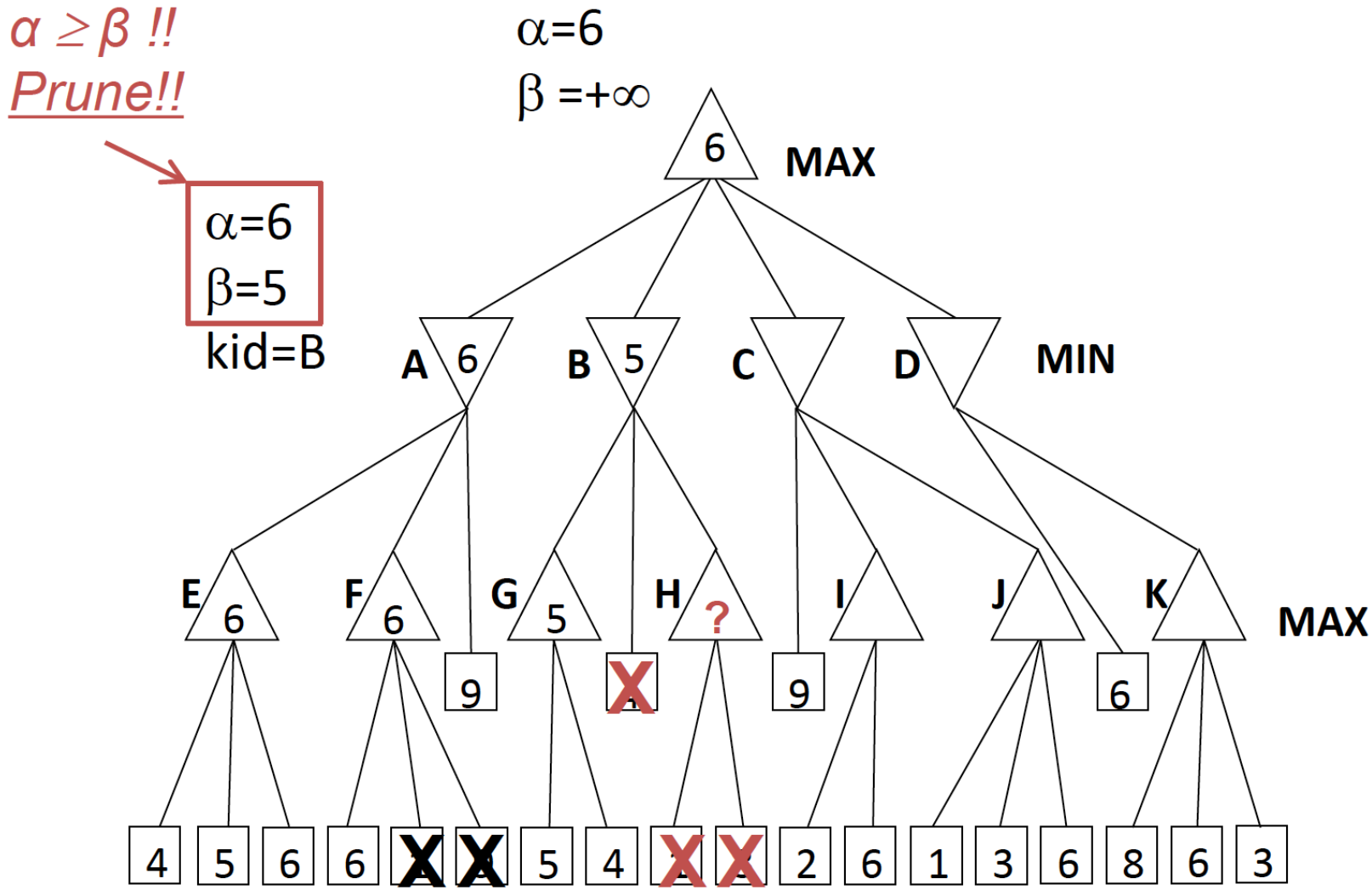
Longer Alpha-Beta Example



Longer Alpha-Beta Example



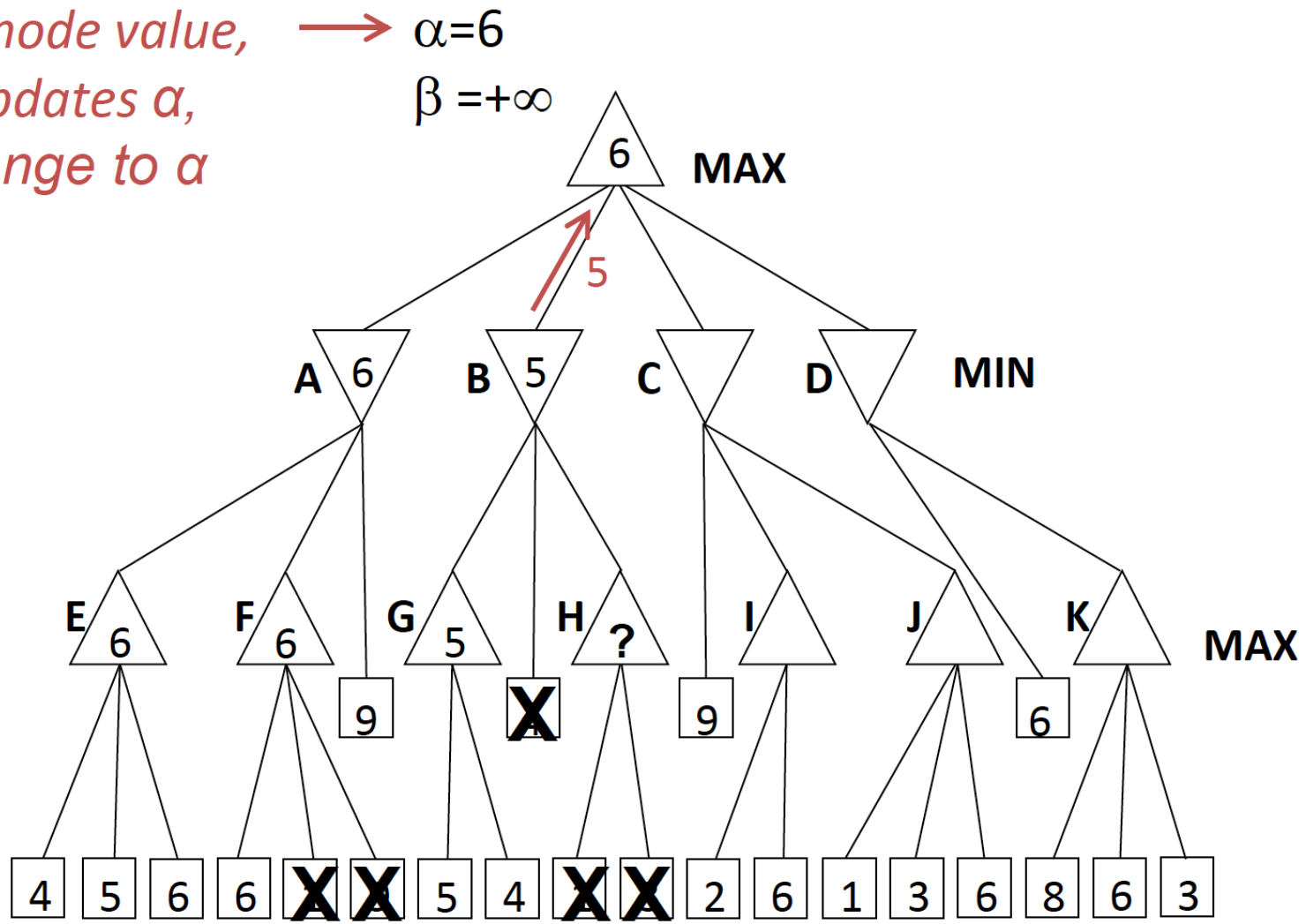
Longer Alpha-Beta Example



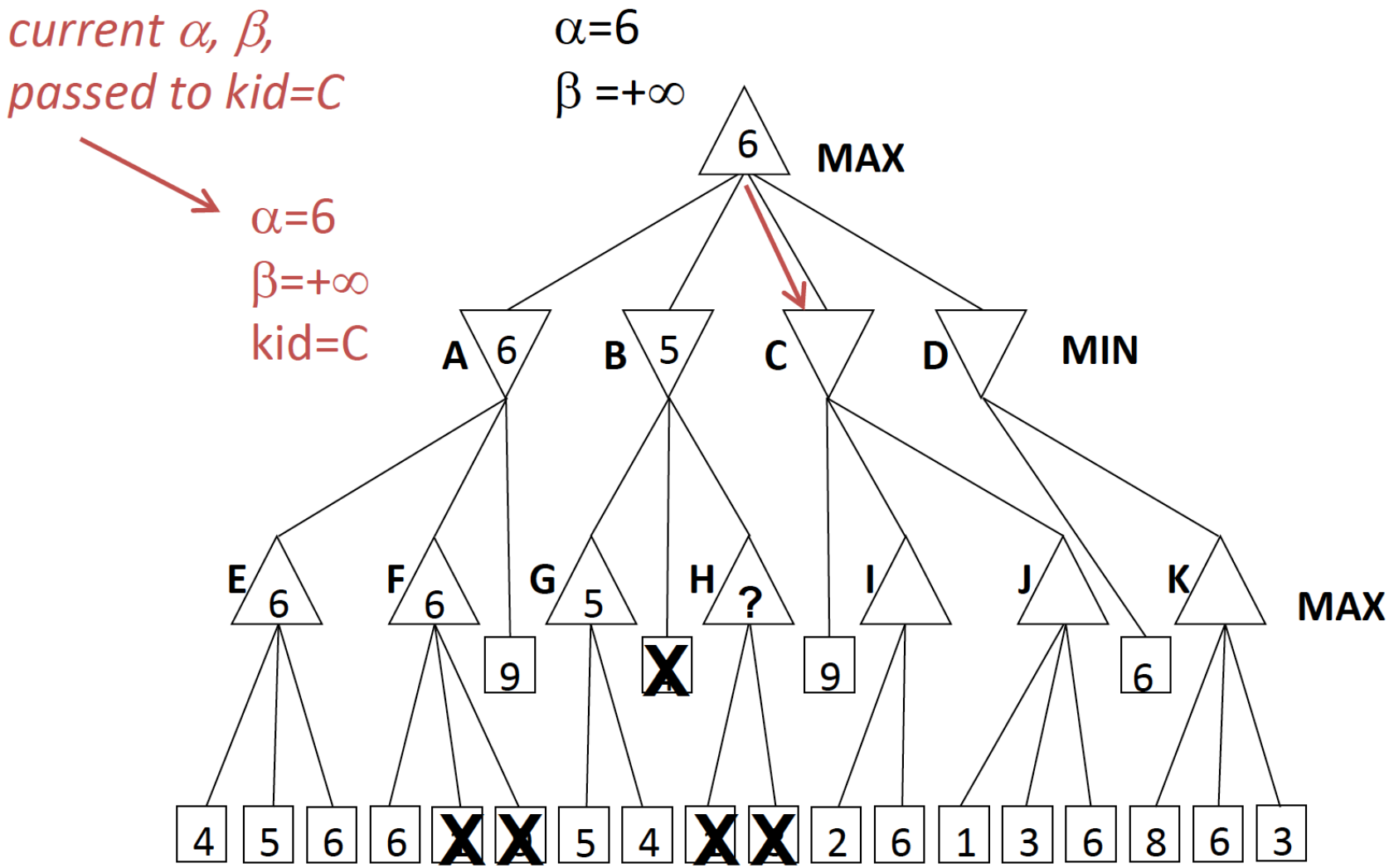
Note that we never find out, what is the node value of H? But we have proven it doesn't matter, so we don't care.

Longer Alpha-Beta Example

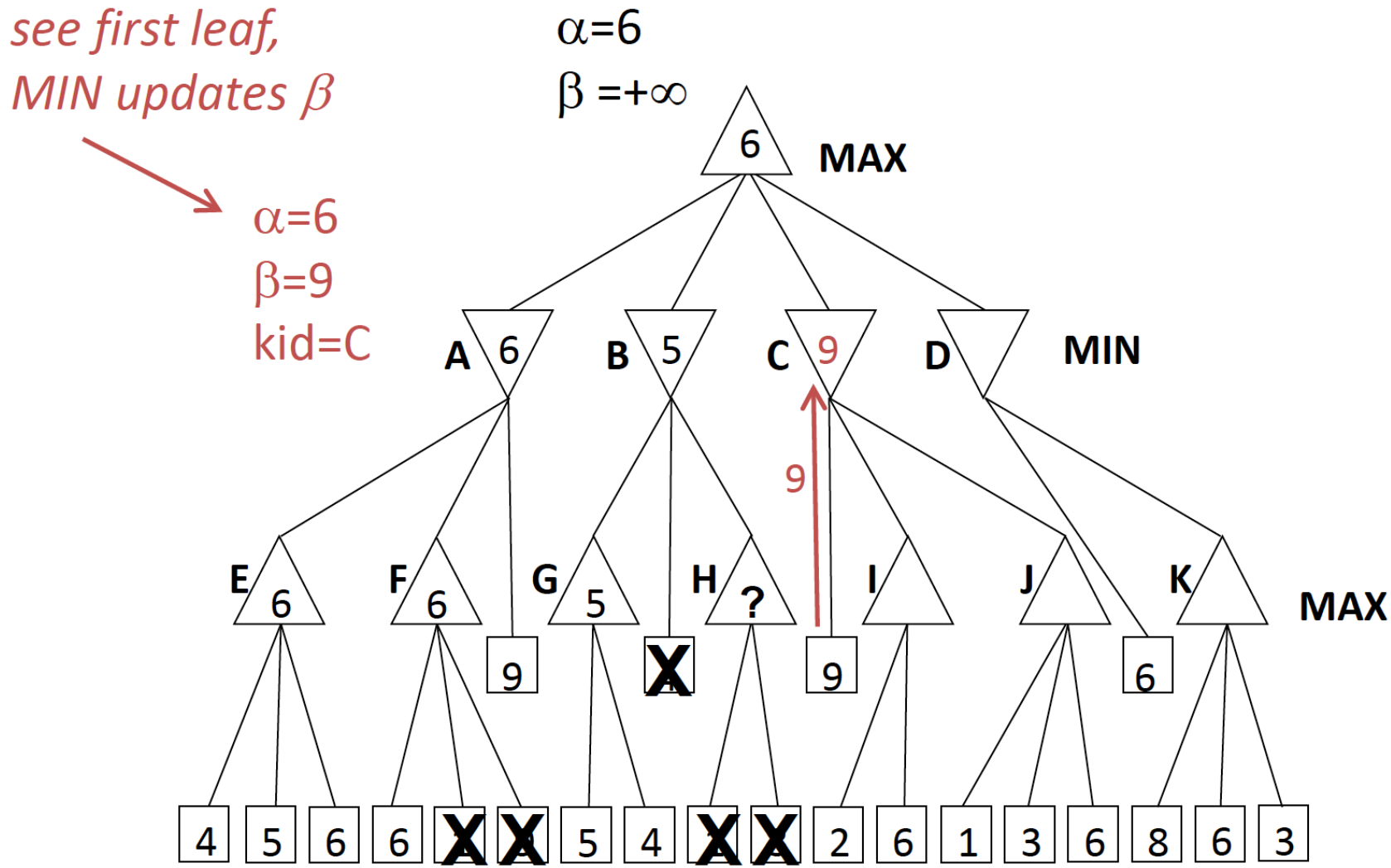
return node value,
MAX updates α,
no change to α



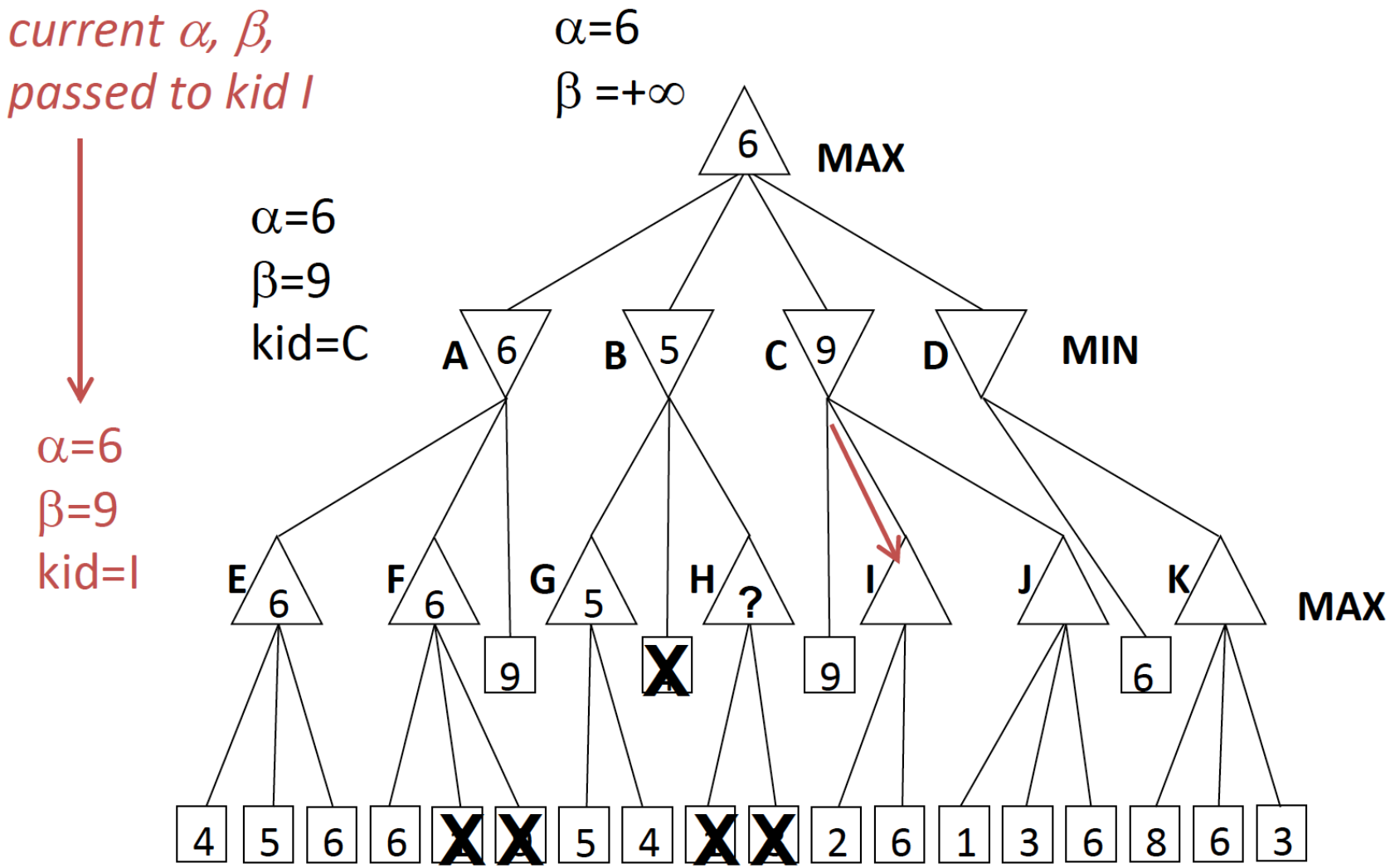
Longer Alpha-Beta Example



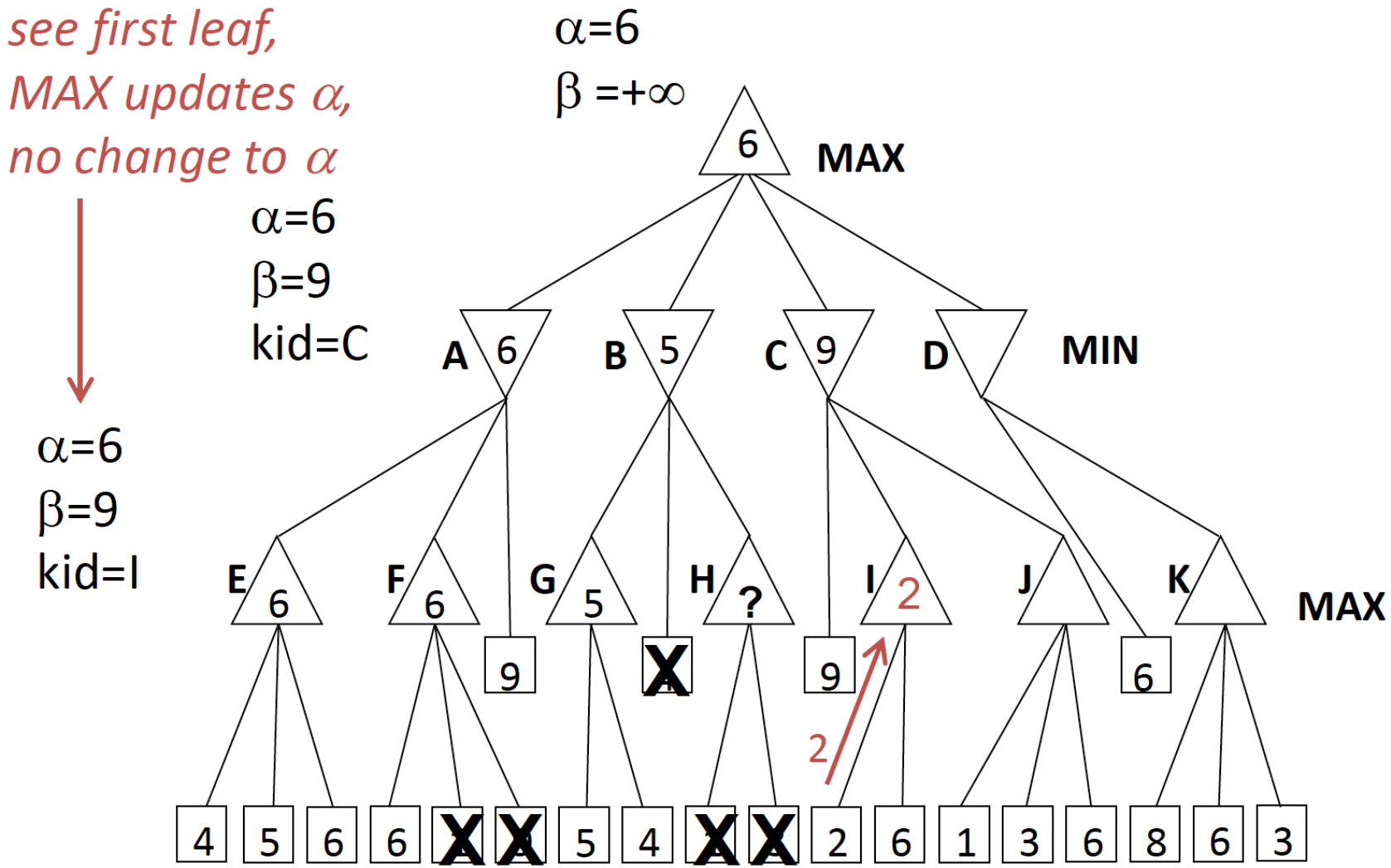
Longer Alpha-Beta Example



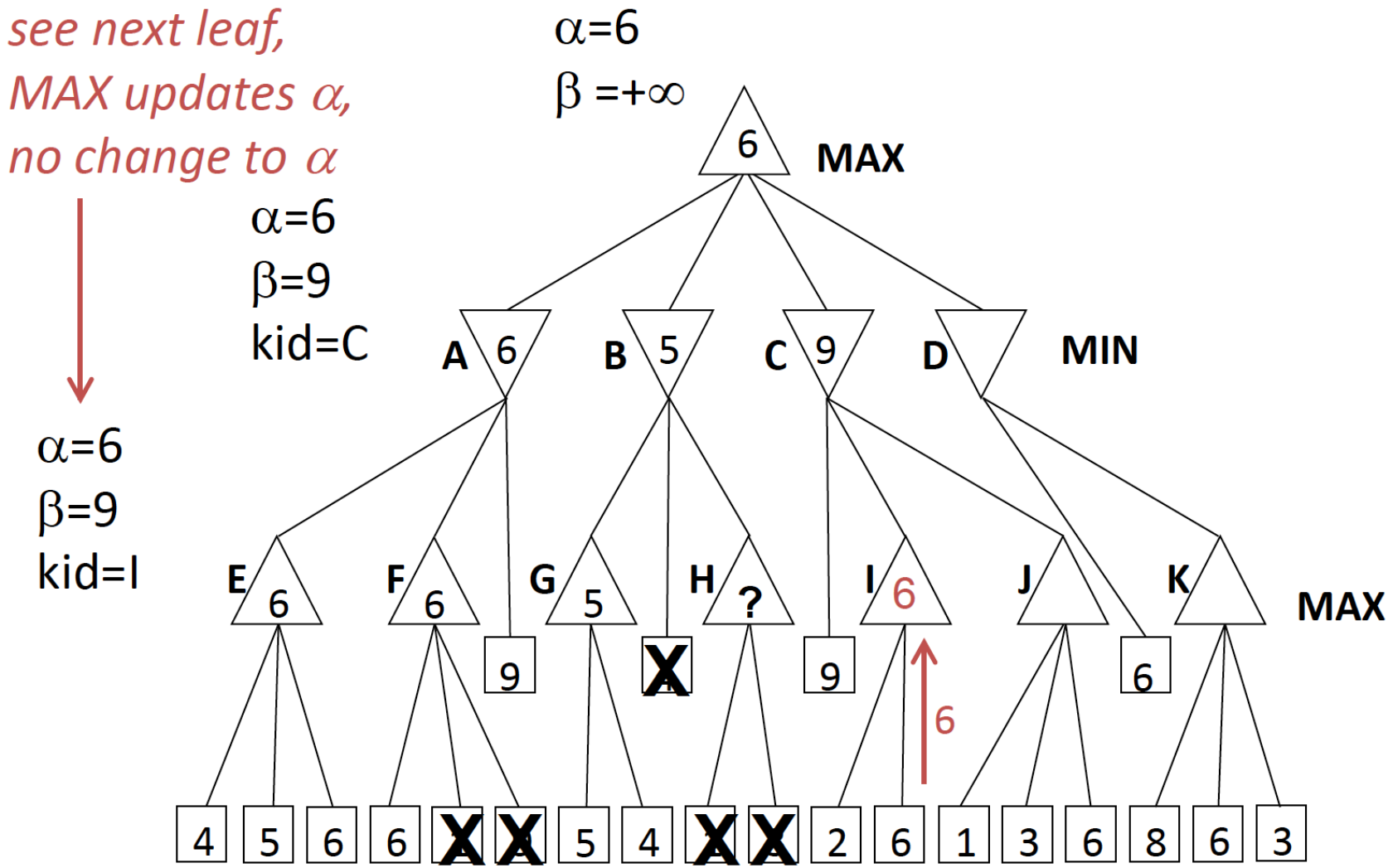
Longer Alpha-Beta Example



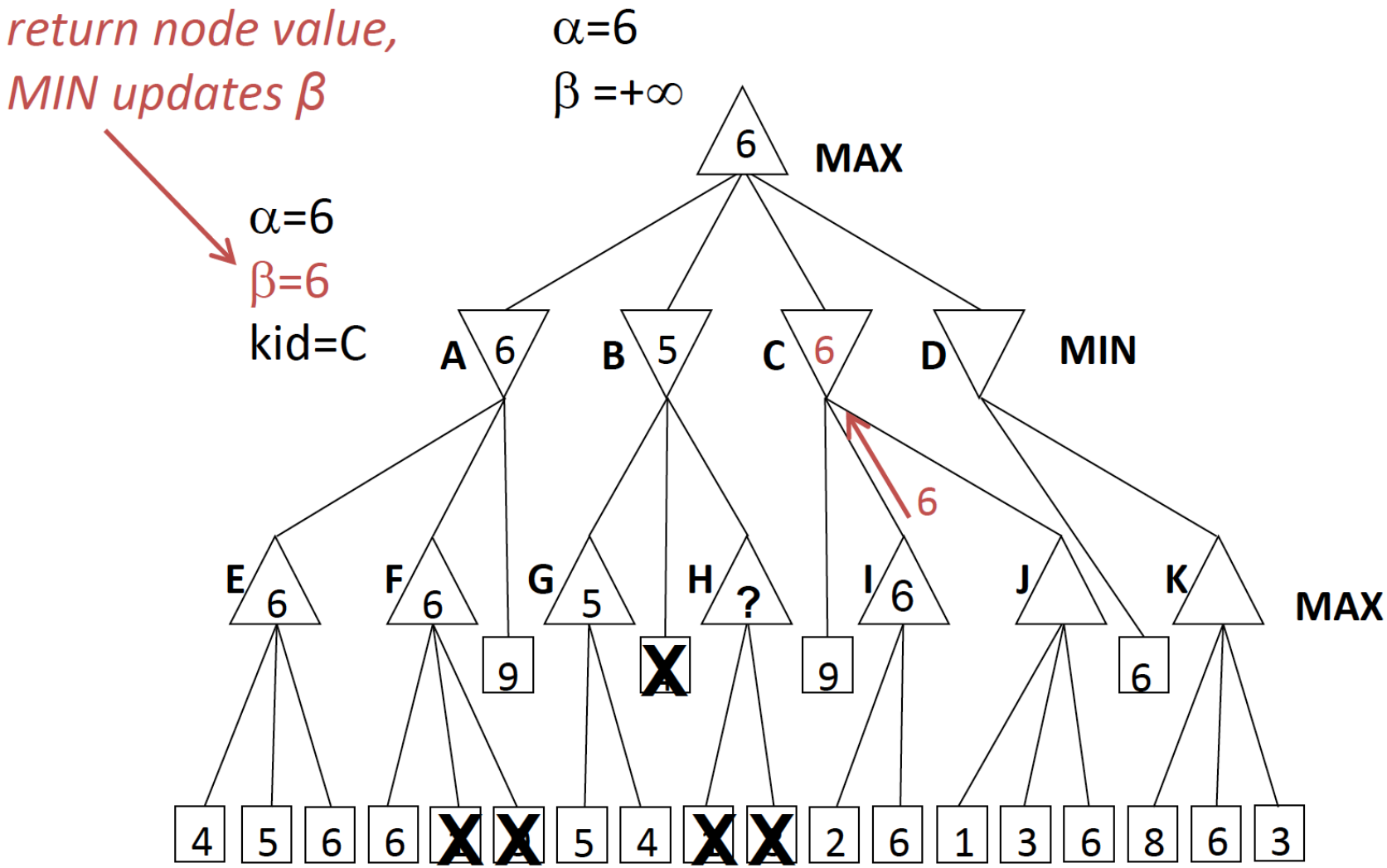
Longer Alpha-Beta Example



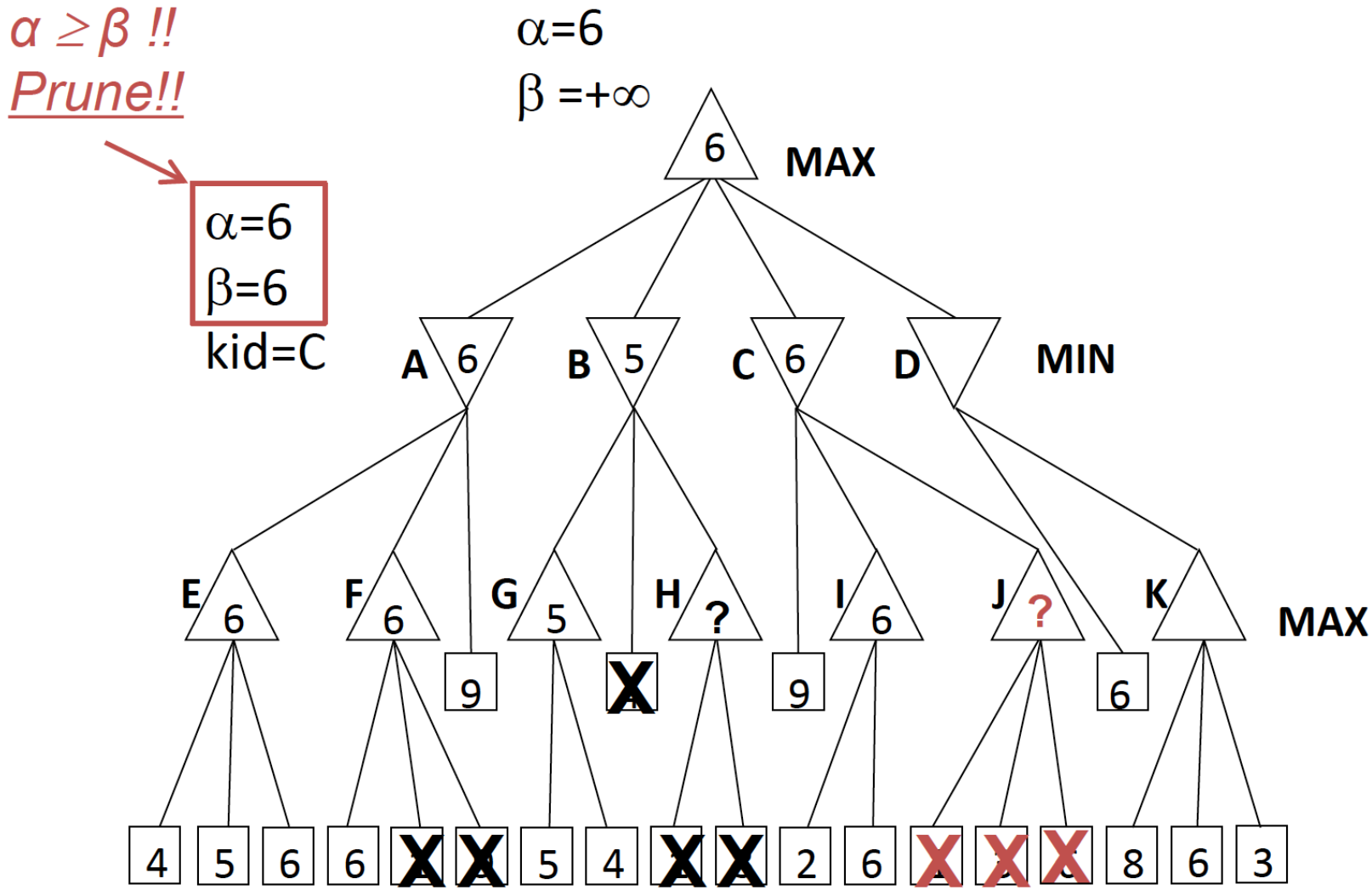
Longer Alpha-Beta Example



Longer Alpha-Beta Example

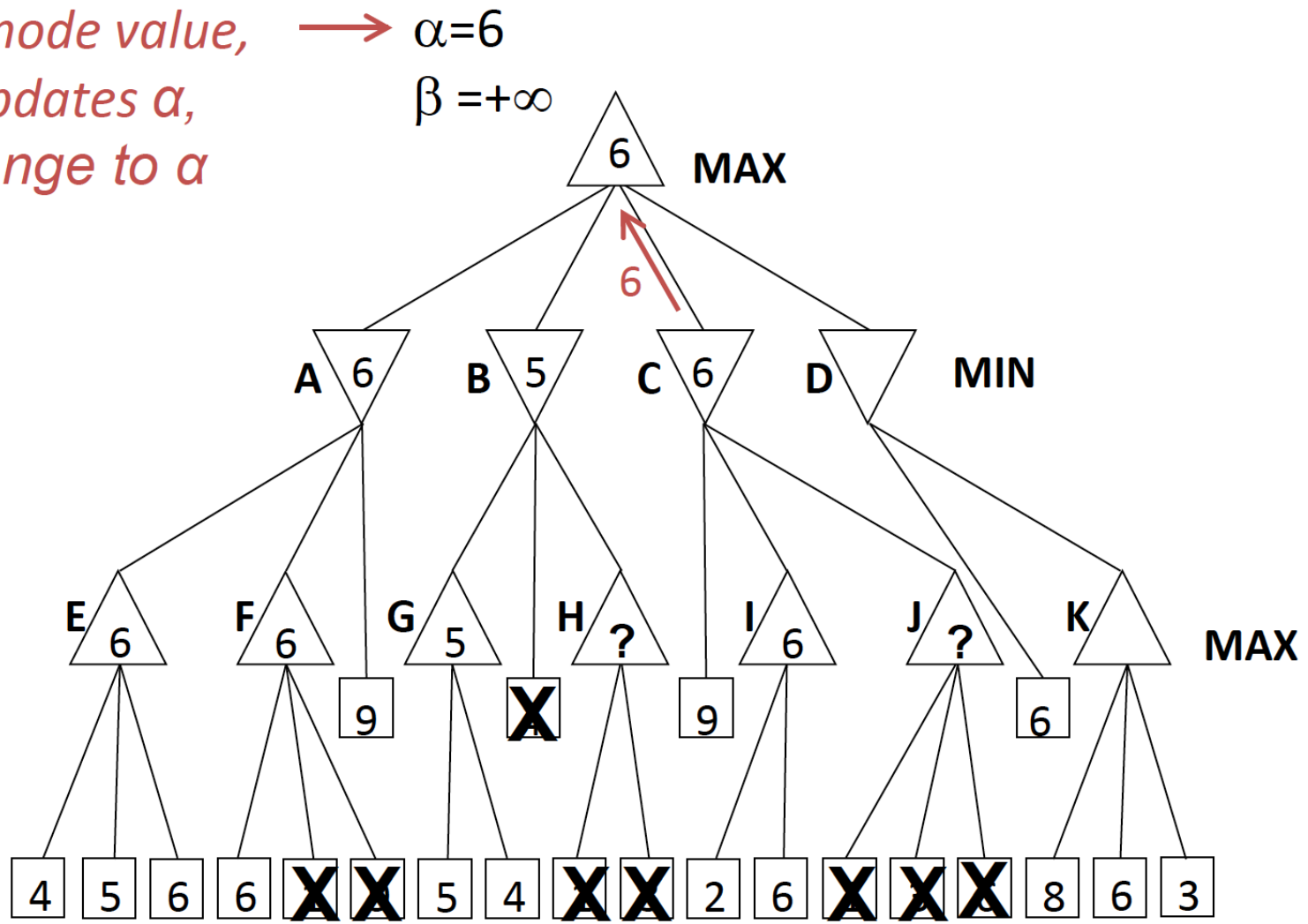


Longer Alpha-Beta Example

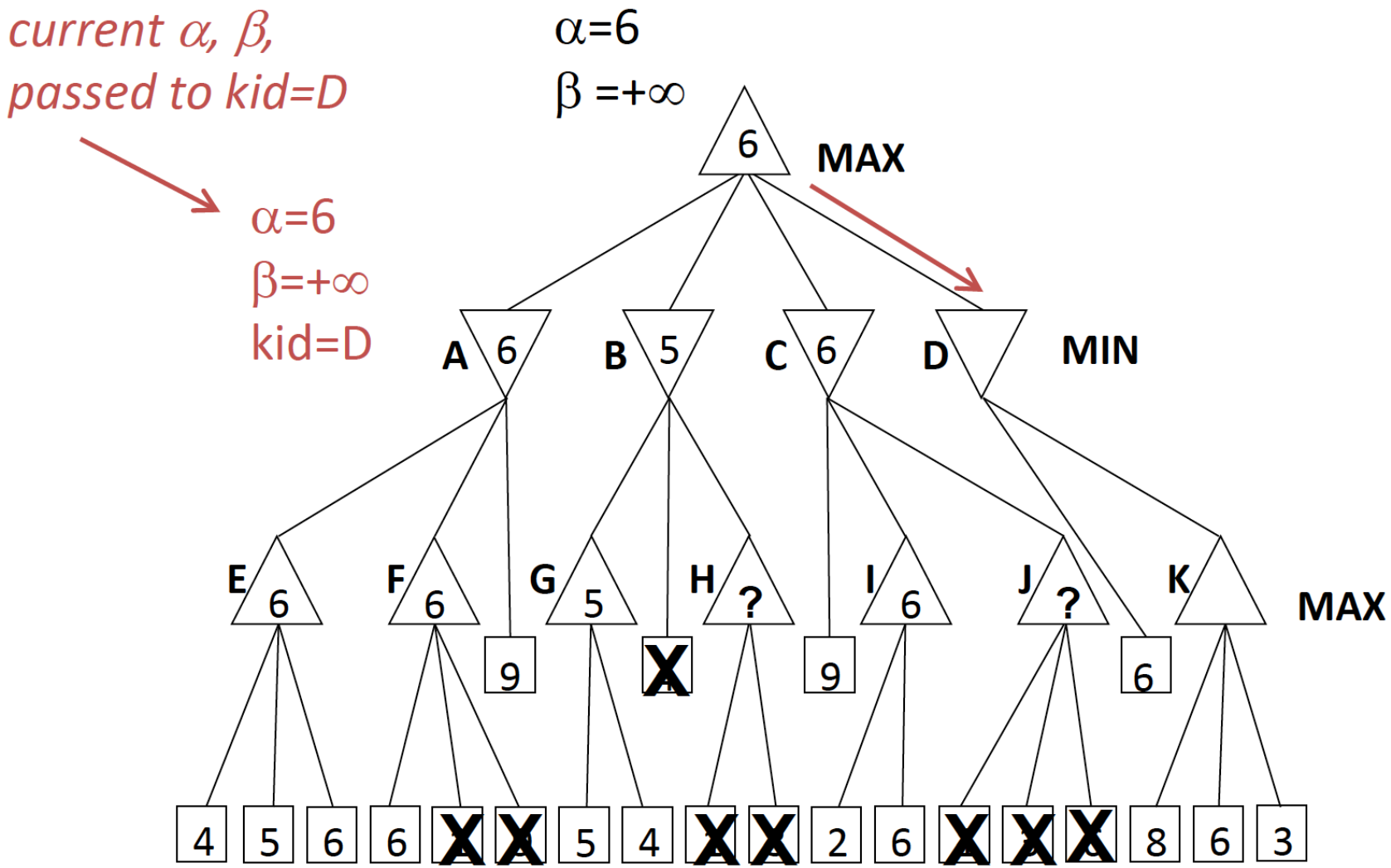


Longer Alpha-Beta Example

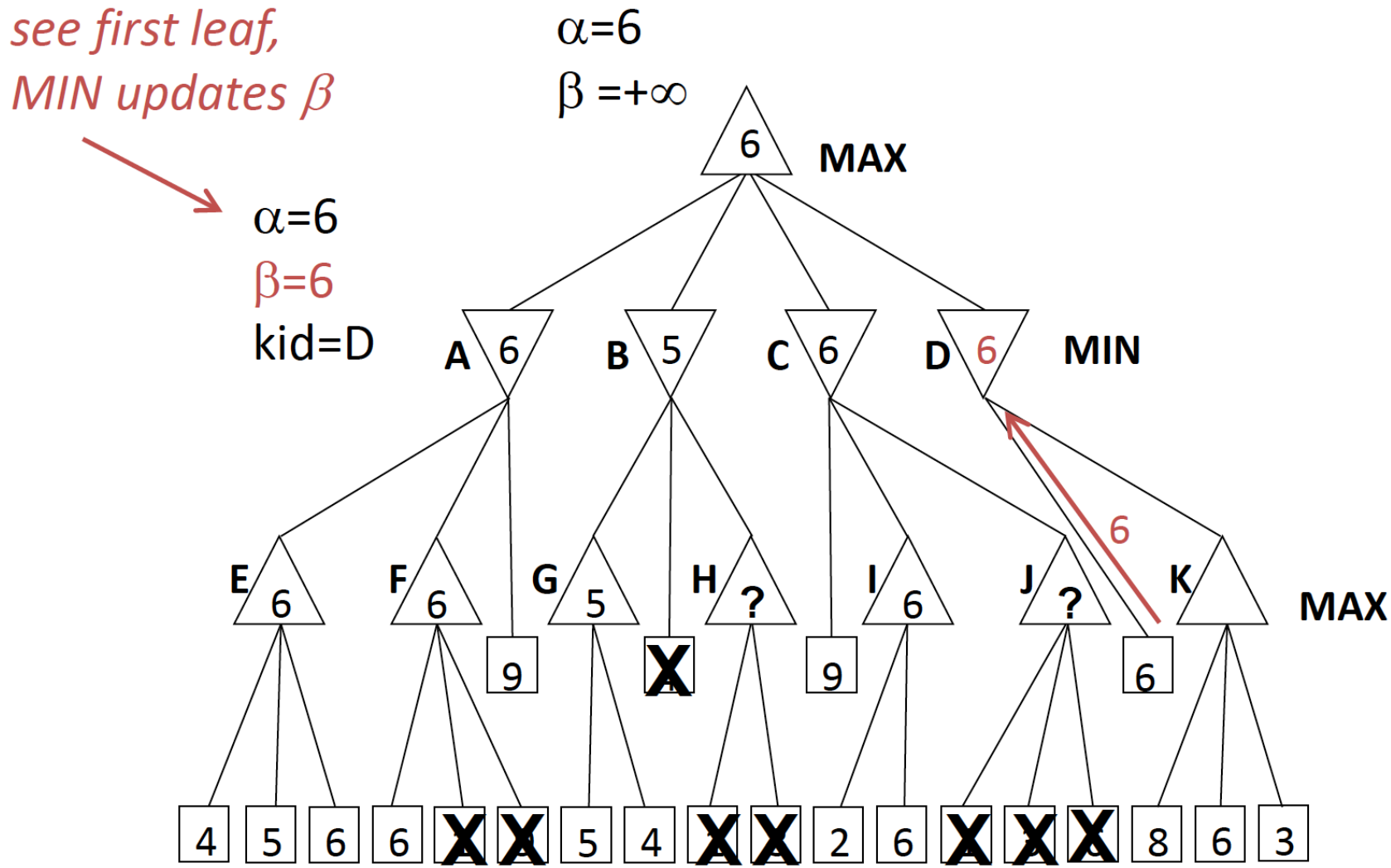
return node value,
MAX updates α,
no change to α



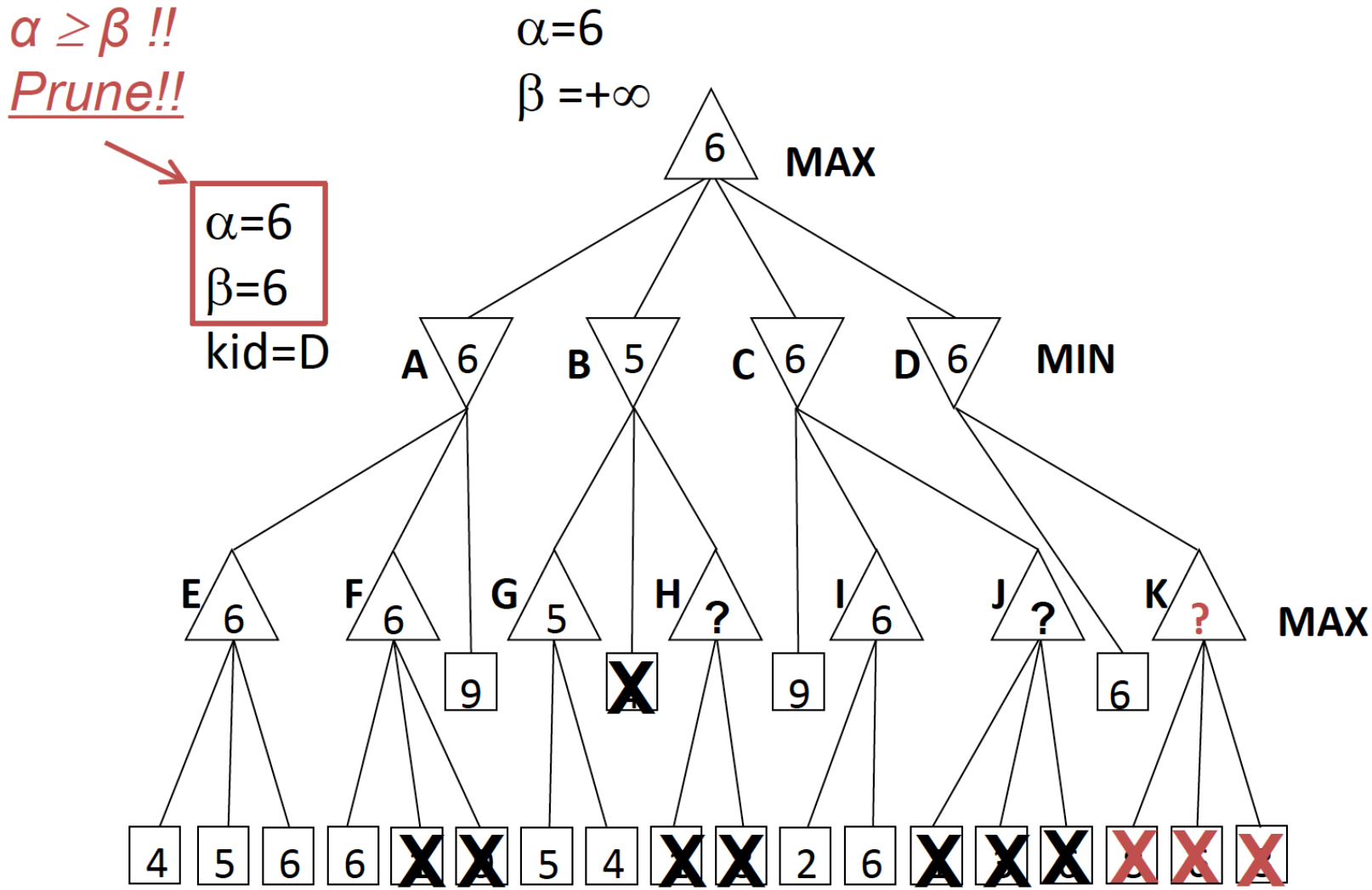
Longer Alpha-Beta Example



Longer Alpha-Beta Example



Longer Alpha-Beta Example

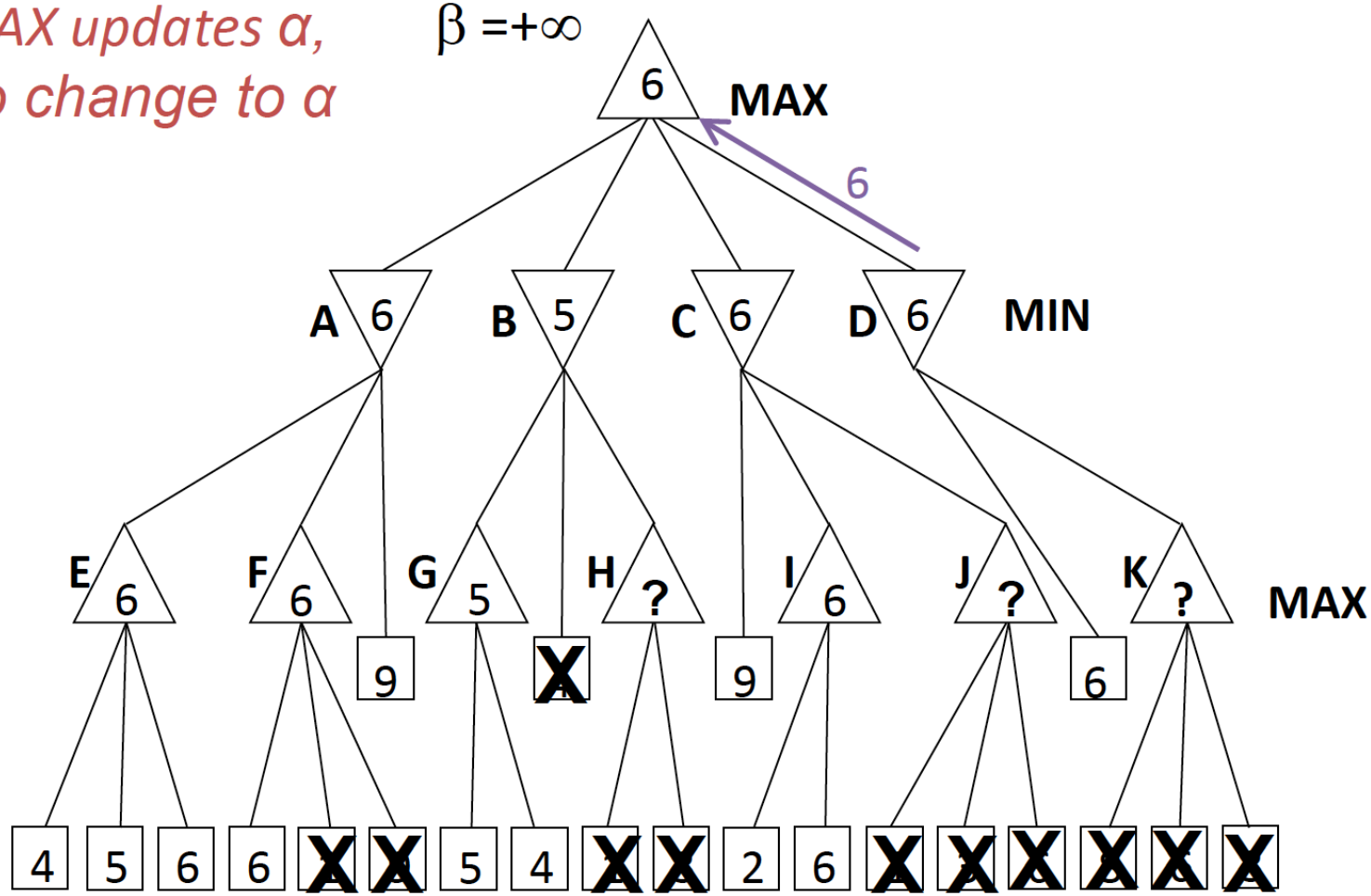


Alpha-Beta Example #2

return node value,
MAX updates α,
no change to α

α=6

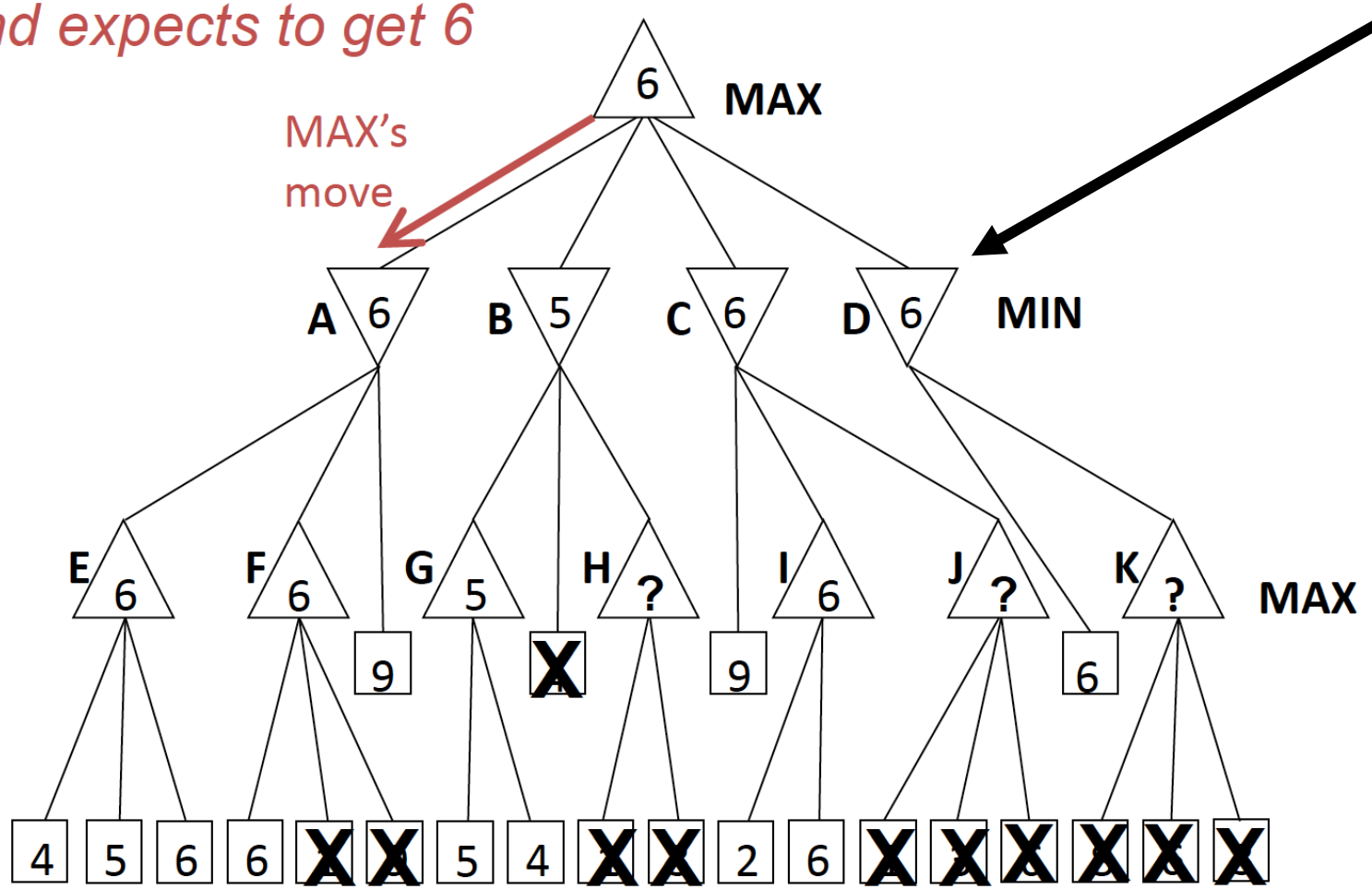
β =+∞



Alpha-Beta Example #2

MAX moves to A,
and expects to get 6

Note: this node could be lower than 6!



Although we may have changed some internal branch node return values, the final root action and expected outcome are identical to if we had not done alpha-beta pruning. Internal values may change; root values do not.

Alpha-Beta Pruning Calculator

- https://pascscha.ch/info2/abTreePractice/

Alpha-Beta Pruning Properties

- Has no effect on minimax value computed for the root
- Ordering improves effectiveness
  - Try promising moves first (will result in higher α)
- With "perfect ordering"
  - Time requirements: $O(b^{\frac{m}{2}})$
  - Effectively makes the branching size $\sqrt{b}$
  - Chess is still not feasible
    - $35^{100} \approx 10^{154}$ without alpha-beta pruning
    - $6^{100} \approx 10^{77}$
  - However, modern day chess solvers on your phone use a combination of alpha-beta pruning and heuristics
    - See Stockfish
    - Can play better than any human player

https://en.wikipedia.org/wiki/Stockfish_(chess)

Outline

- Background
- Minimax search
- Cutting off search
- Alpha-beta pruning
- Alpha-Go

Search Strategies

- Type A
  - Wide but shallow
  - Consider all possible moves up to a certain depth, then use the evaluation function to estimate the utility of the states at that depth
  - Minimax search
  - Alpha-beta pruning
- Type B
  - Deep but narrow
  - Ignore moves that look bad and explore promising moves
  - Quiescence search
  - Singular extensions
  - Monte Carlo tree search

Monte Carlo Tree Search

- Selection
  - Select a leaf node
- Expansion
  - Generate a new node child node from the selected node
- Simulation
  - Play a game from the generated child node where moves thought to be good are selected with higher probability than bad moves
- Backup
  - Update all ancestor nodes based on the result of the game

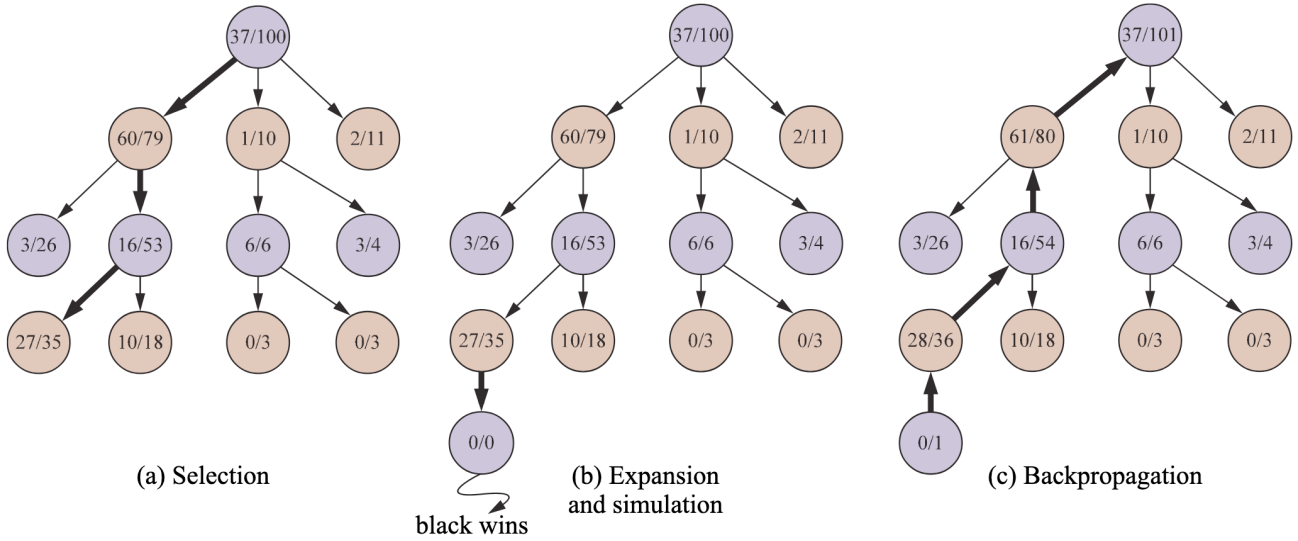
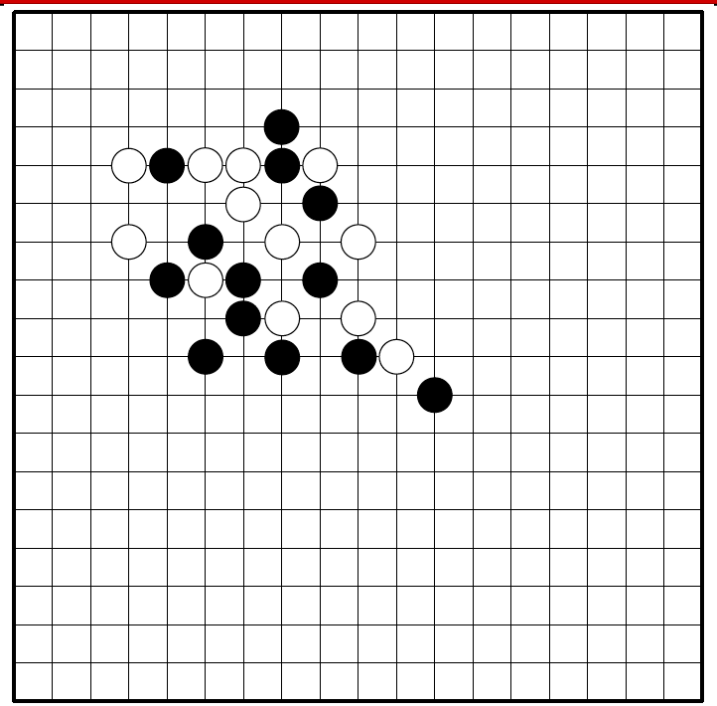
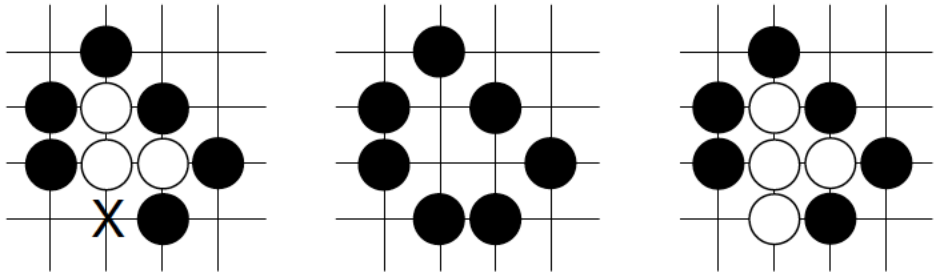


Figure 5.10 One iteration of the process of choosing a move with Monte Carlo tree search (MCTS) using the upper confidence bounds applied to trees (UCT) selection metric, shown after 100 iterations have already been done. In (a) we select moves, all the way down the tree, ending at the leaf node marked 27/35 (for 27 wins for black out of 35 playouts). In (b) we expand the selected node and do a simulation (playout), which ends in a win for black. In (c), the results of the simulation are back-propagated up the tree.

Go

- Typically played on a 19x19 board
  - Smaller board sizes are also used
- Can put stones on cross sections of the grid
- Goal is to surround more territory than your opponent





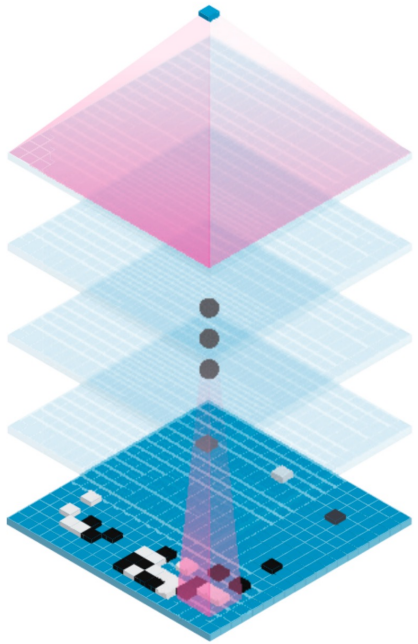
AlphaGo

- Uses neural networks to represent a simulation policy and a heuristic (value) function
- These neural networks are trained from expert games and then through self-play

AlphaGo: Value and Policy Networks

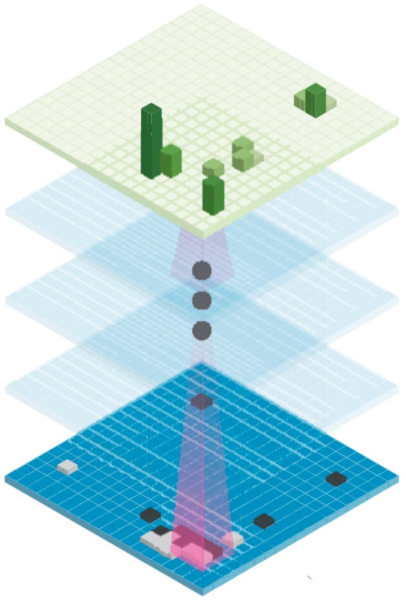
- Deep convolutional neural network
- 13 convolutional layers

$\mathrm{V}(s, \boldsymbol{w})$



s

$p(a|s, \boldsymbol{\theta})$



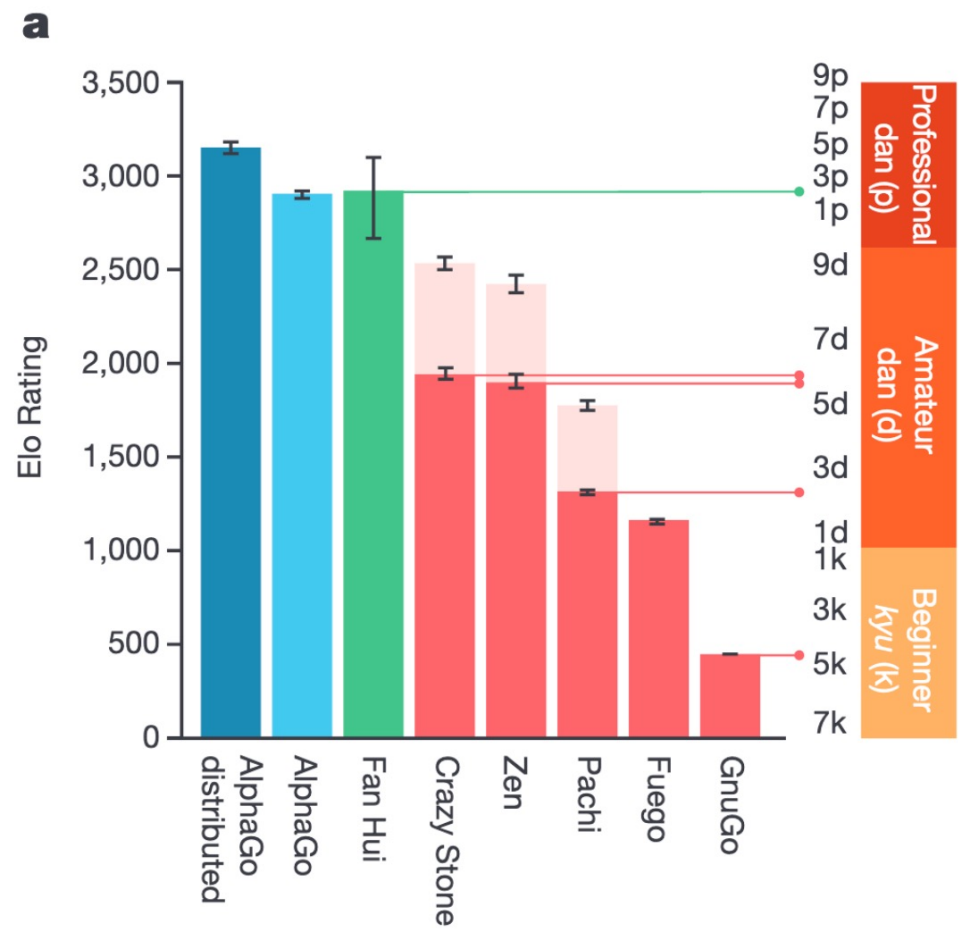
s

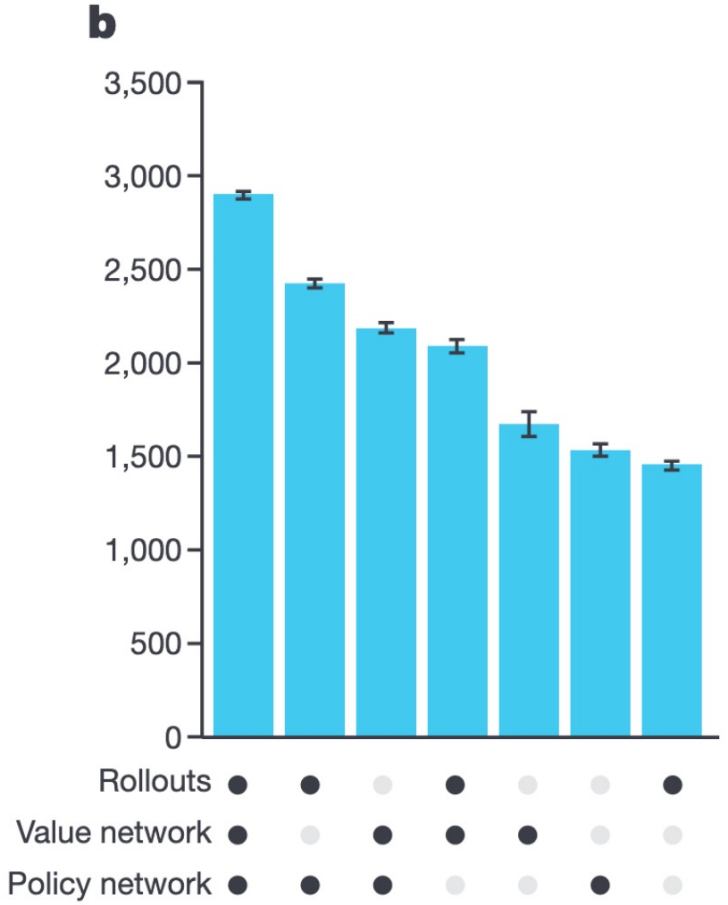
AlphaGo: Learning

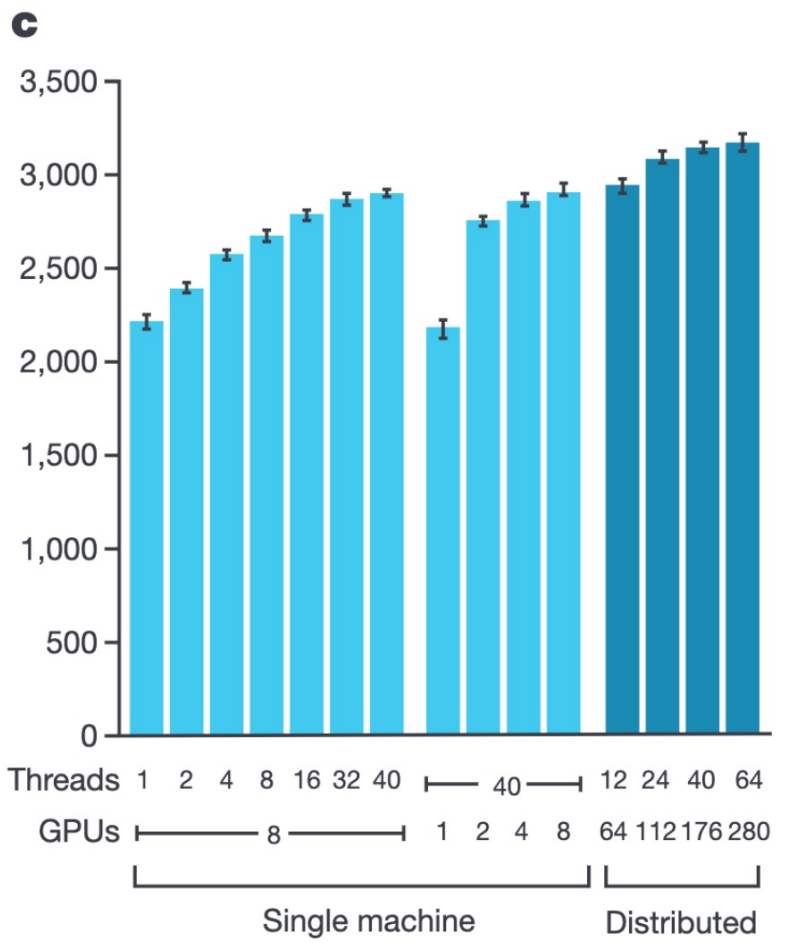
- Learns a policy function through supervised learning of 30 million expert games
  - Achieved 55.7% accuracy
  - The current state-of-the-art at the time achieved 44.4% accuracy
  - The large DNN architecture significantly contributes to this success
- Refines the policy function through self-play
  - Plays games against itself
  - Trains the policy function to increase the probability of actions of wining games and decrease the probability of actions of losing games
  - Trains the value function to predict the outcome of the games

AlphaGo: Results

- Uses a distributed version of MCTS

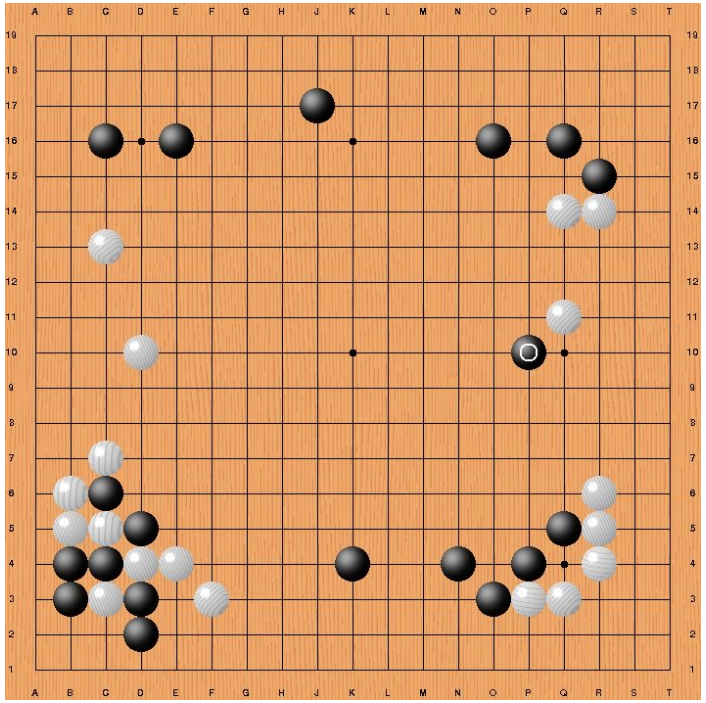






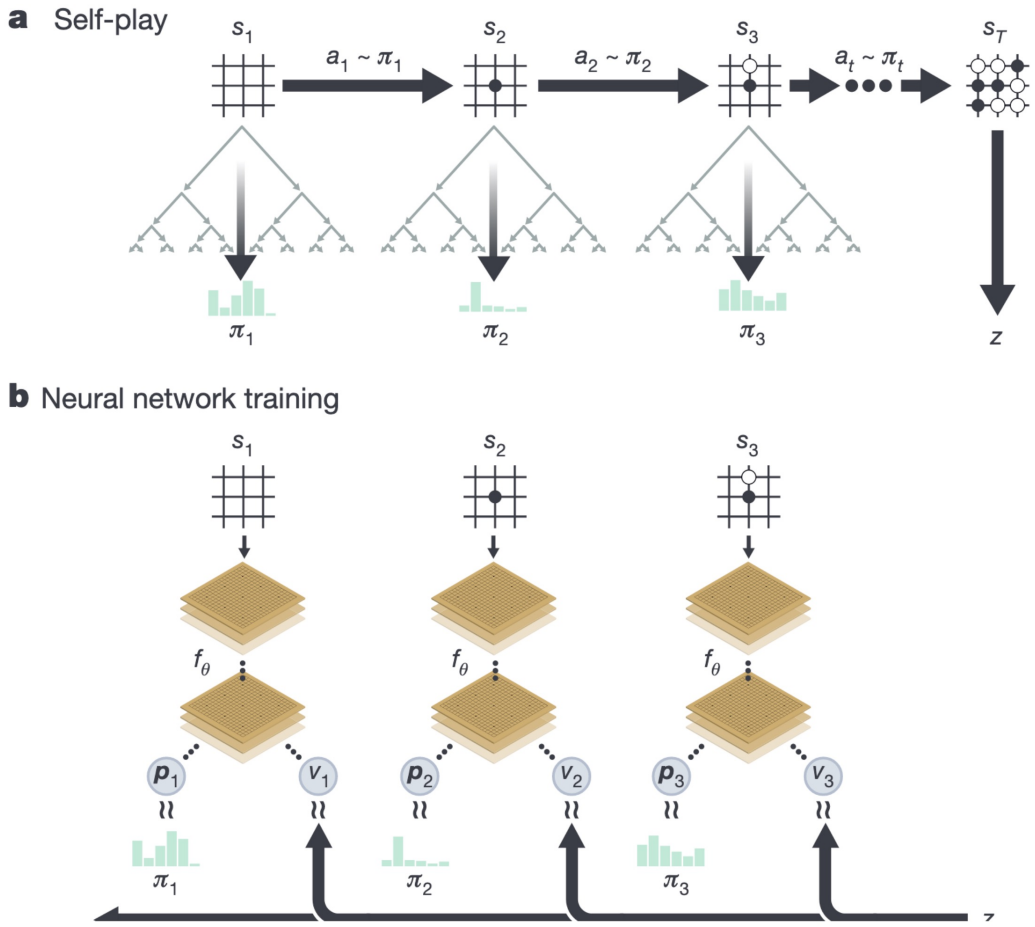
AlphaGo: Results

- Defeated European Go champion Fan Hui 5-0
- Defeated 18-time world champion Lee Sedol 4-1
- Move 37
  - Policy network assigned low probability to move 37
  - Using MCTS, AlphaGo decided that move 37 was actually a good move



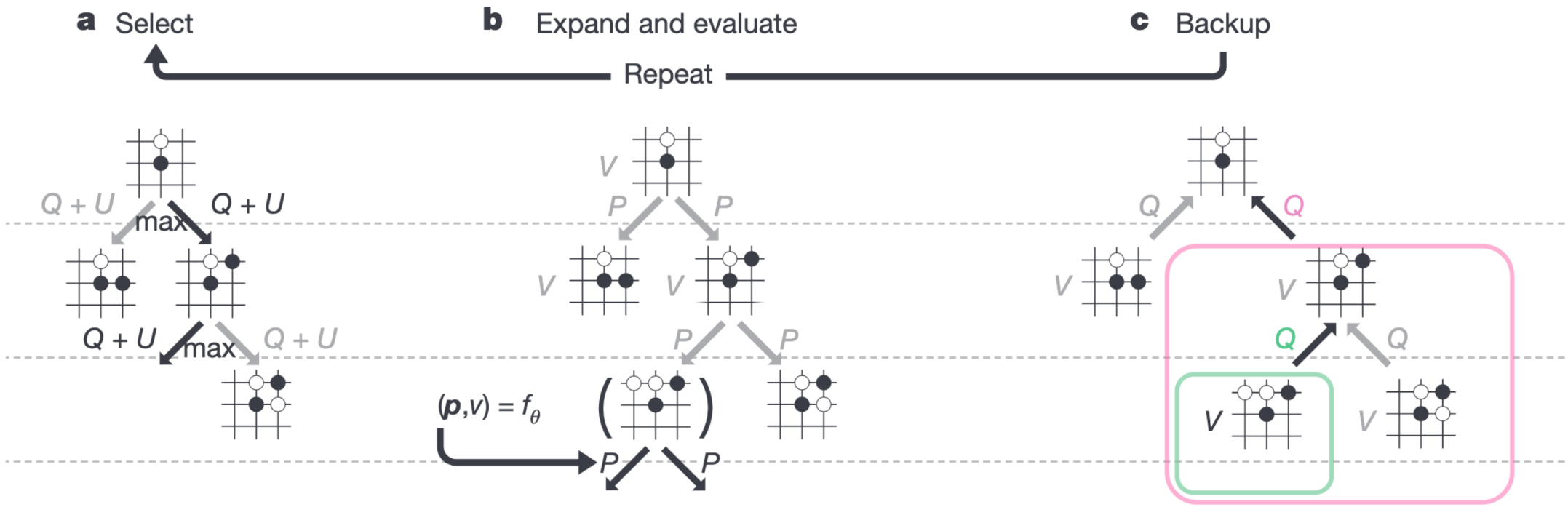
AlphaGo Zero: Learning

- The network being trained is played against the current best network
  - Start with a randomly initialized network as the best network
- When the network being trained has a win rate of over 55%, it is then set to be the best network
- Learns the heuristic function from scratch!



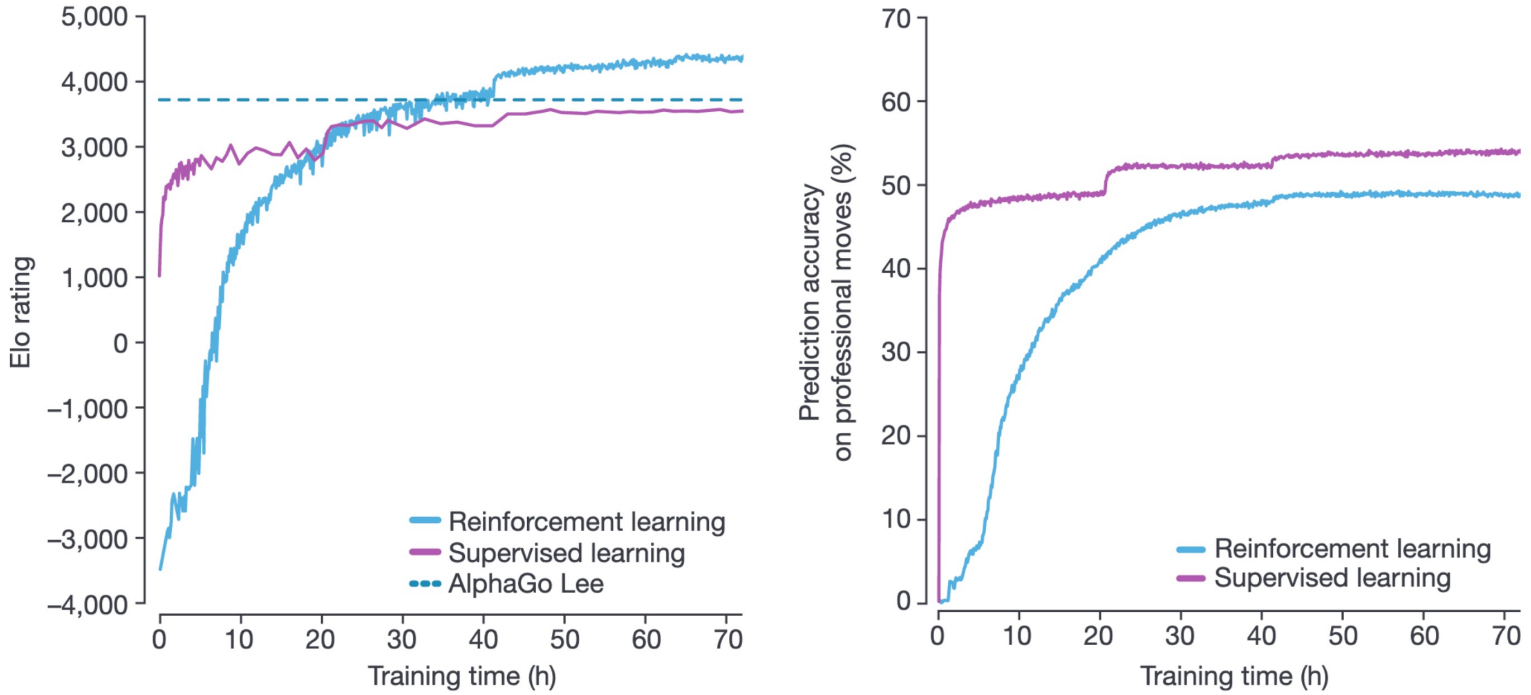
AlphaGo Zero: MCTS

- Only the value network is used to estimate the return
  - No need to do a rollout, just back up what the value from the value function



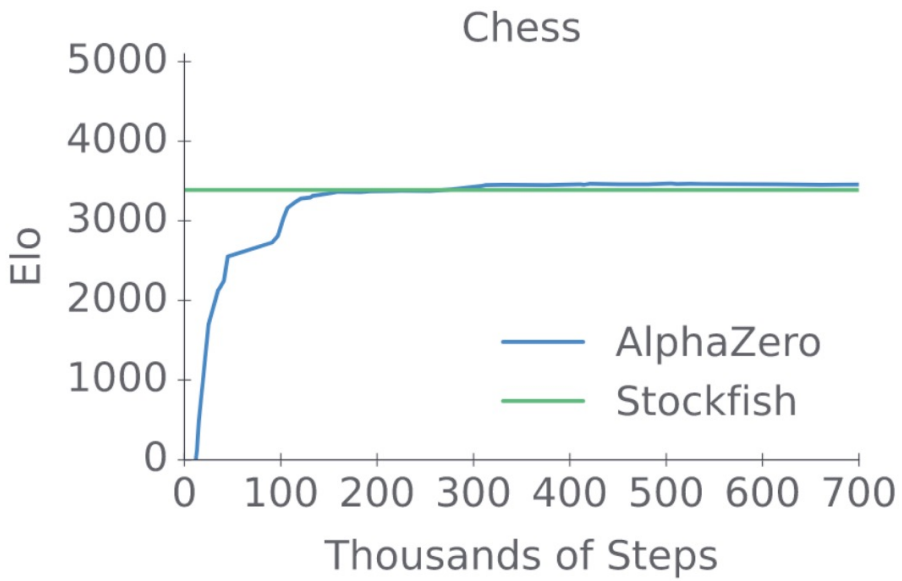
AlphaGo Zero: Results

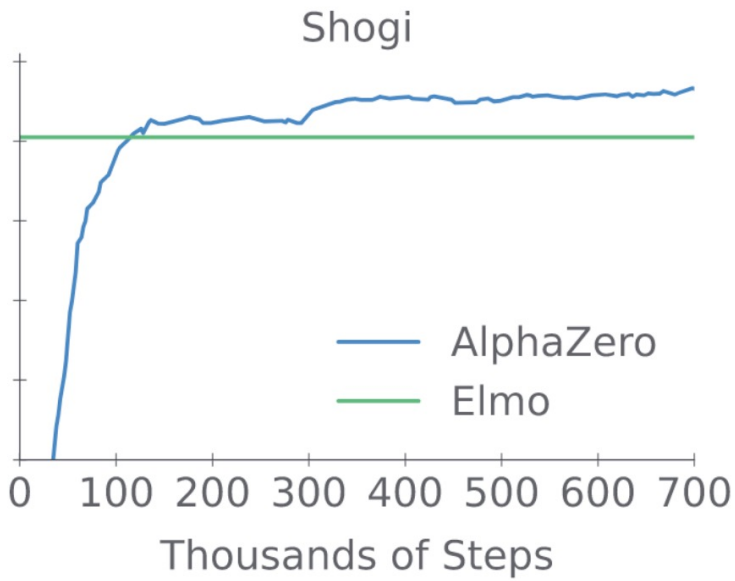
- Plays better than any of the previous computer Go players
- Though it plays better than the network learned through supervised learning, it has worse prediction accuracy on the supervised learning dataset
  - Indicates that it has learned a different strategy than that of human experts

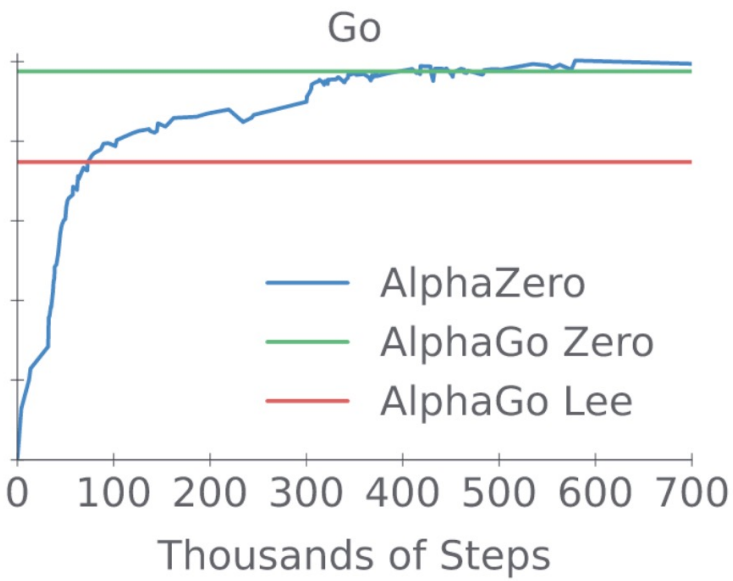


AlphaZero

- Generalized AlphaGo Zero to play Go, Chess, and Shogi
- Plays better than the best computer programs for these games







Summary

- Minimax search
  - Plays optimally against adversary who is also playing optimally
  - Time complexity is exponential in the depth of the game tree
- Cutting off search
  - We can estimate the utility of a position using a heuristic function
  - We can then use this heuristic function to cut off search early
- Alpha-beta pruning
  - We can get a utility of at least α
  - We can get a utility of at most β
  - Prune when $\alpha \geq \beta$

Next Time

- Optimization