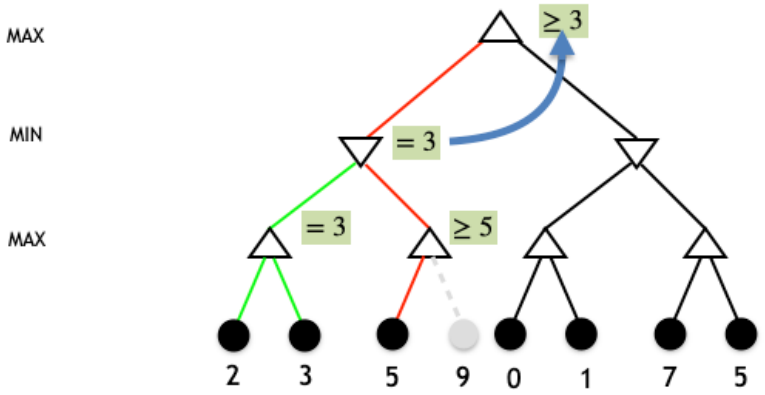
CS143: Artificial Intelligence



Minimax Algorithm

α-β Pruning

Drake UNIVERSITY

Review: Adversarial Search

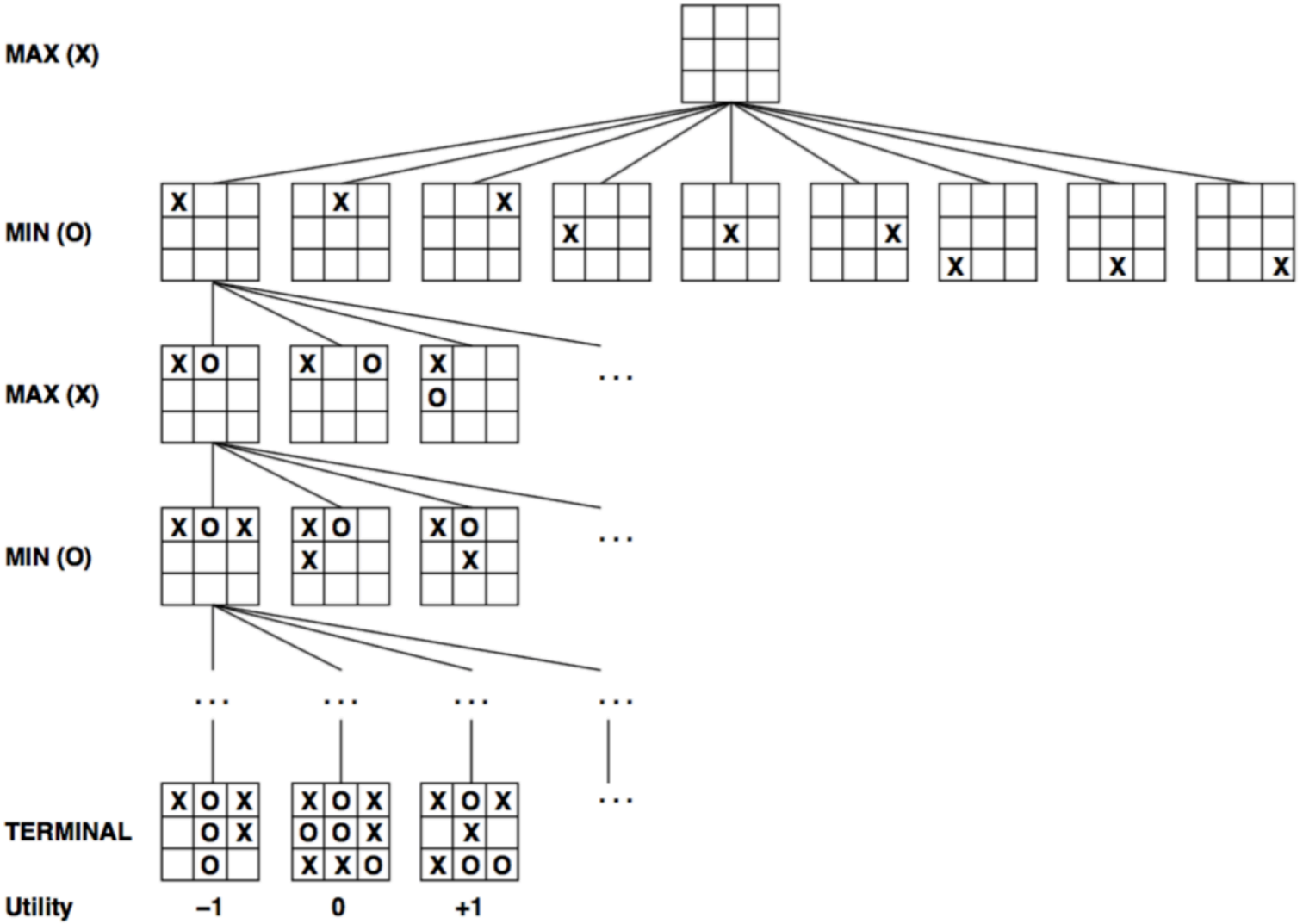
- "Adversarial Search" concern problems known as **games**:
  - eg, checkers, chess, go, backgammon, etc.

- We focus in particular on games that involve turn-taking and are **zero-sum**:
  - means the total payoff to players is the same for each game
  - eg, given player A vs. player B, the outcomes include:
  - **A wins (1), A loses (-1), or tie (0)**
  - A wins (1,0); B wins (0,1), or tie (0.5,0.5)

Review: Games

- Games are interesting because they are *too hard* to solve optimally
  - eg, the search tree for a chess game involves 10^{154} possible nodes
- Consider two players: MIN and MAX
  - MAX moves first
  - MIN moves next, etc
- The choice of possible moves for both players gives rise to a **game tree**
- A **utility function** determines a numeric value for a game that ends in a *terminal state*
  - MAX wins (1), MIN loses (-1), or tie (0)
  - MAX wins (1,0); MIN wins (0,1), or tie (0.5,0.5)

Review: Game Tree Example



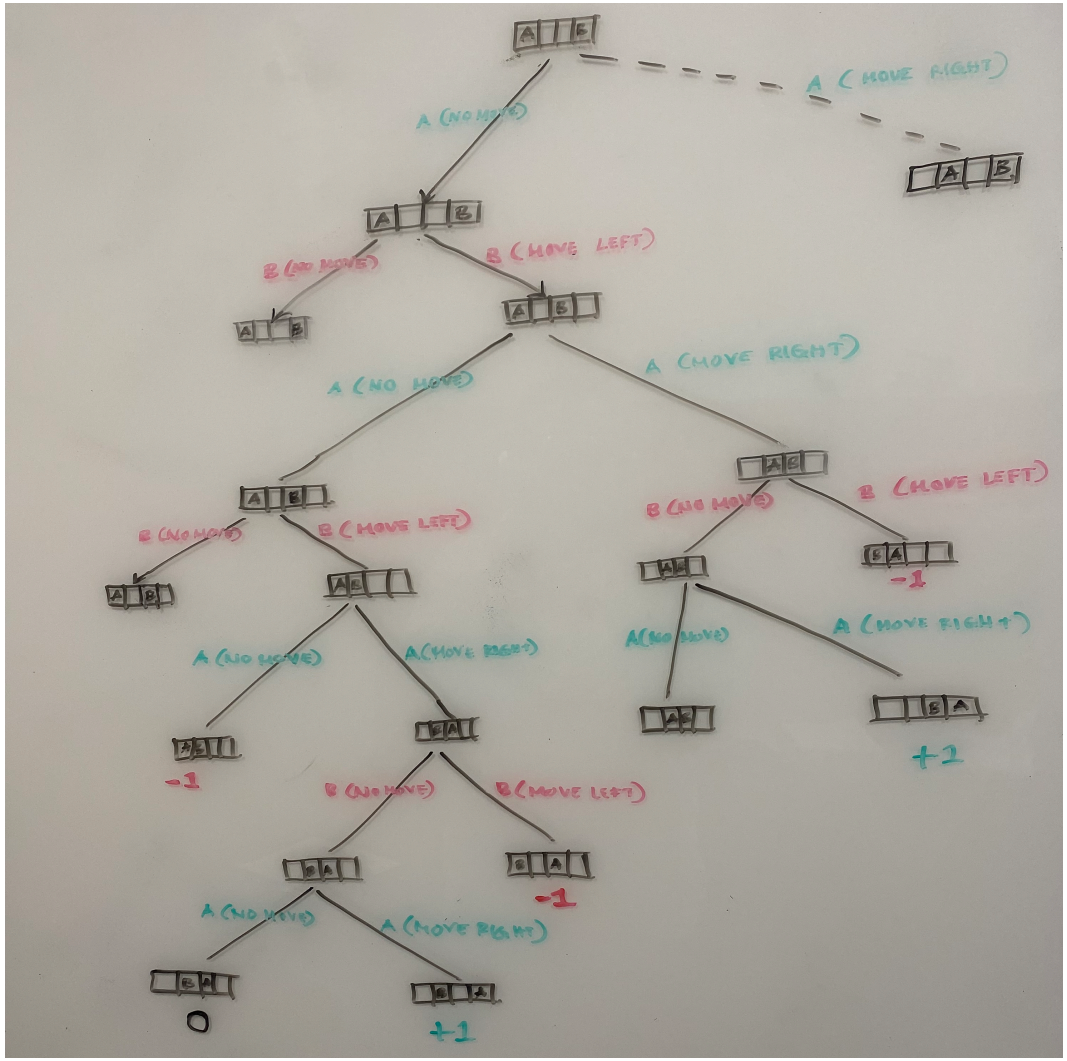
Review: Tile Sliding Game

- During A's turn, they can do one of the followings
  - stay put
  - move right one space

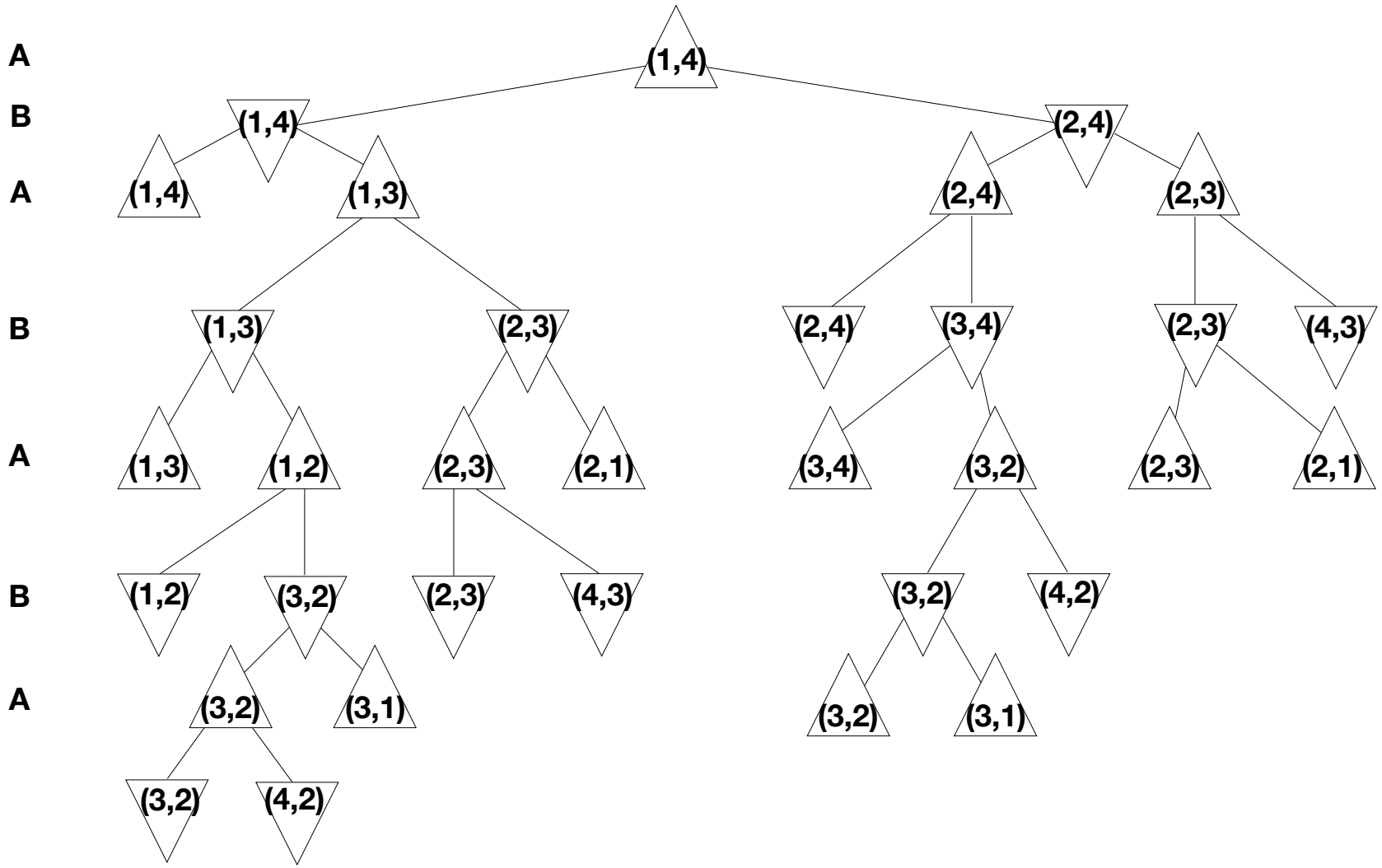
- During B's turn, they can do one of the followings
  - stay put
  - move left one space

- If the players occupy adjacent spaces, then the player may jump over the opponent to the next open space in the respective legal direction

Review: Tile Sliding Game Activity



Review: Tile Sliding Game Activity



Minimax Algorithm

Minimax Algorithm

- Idea: At each turn, choose to move to the position with best **minimax** value
  - aim for the best achievable payoff against the best play of the opponent

- Evaluate the game tree's utility function value for all possible moves
  - start from the bottom-up

- This results in perfect play for deterministic, perfect-information games

Minimax Function

$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \end{cases}$$

Minimax Function to Pseudocode

- The Minimax function is recursive one
  - One computes the maximum value from its children
  - Another computes the minimum value from its children

- You can create a two separate functions that mutually calls each other
  - Mutual recursion example

```
bool is_even(unsigned int n) {
    if (n == 0)
        return true;
    else
        return is_odd(n - 1);
}

bool is_odd(unsigned int n) {
    if (n == 0)
        return false;
    else
        return is_even(n - 1);
}
```

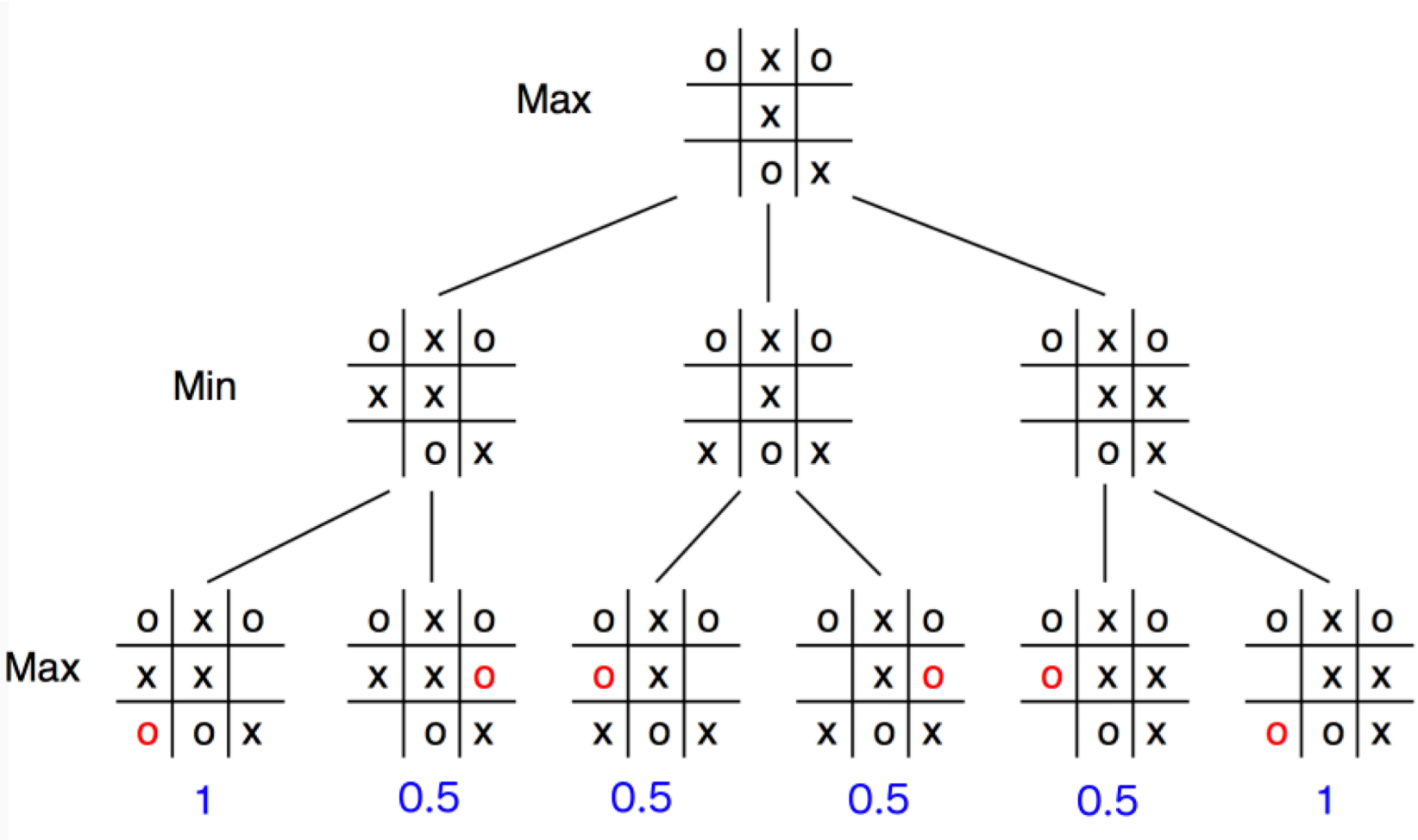
Minimax Algorithm Pseudocode

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))
```

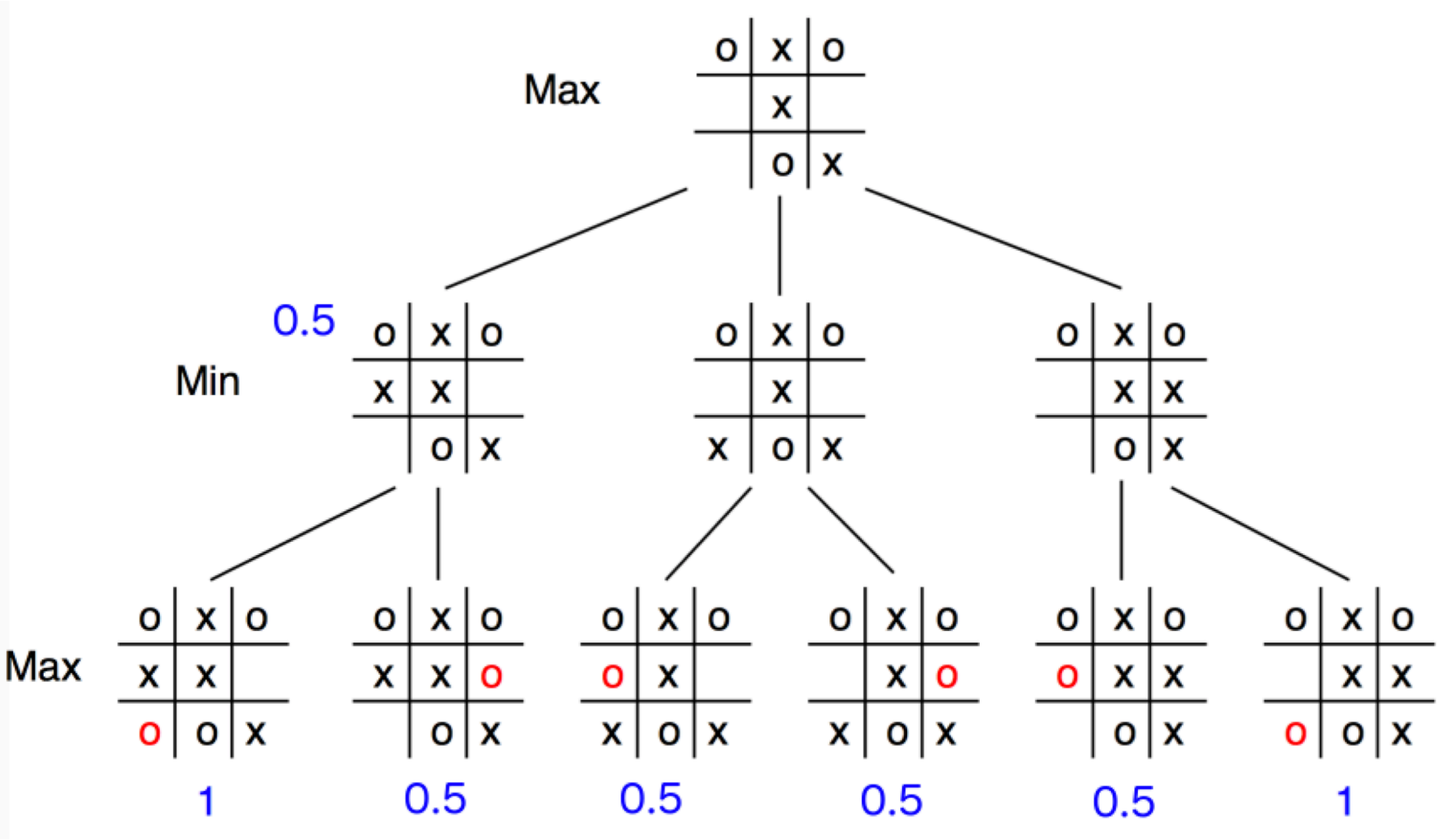
```
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v
```

```
function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

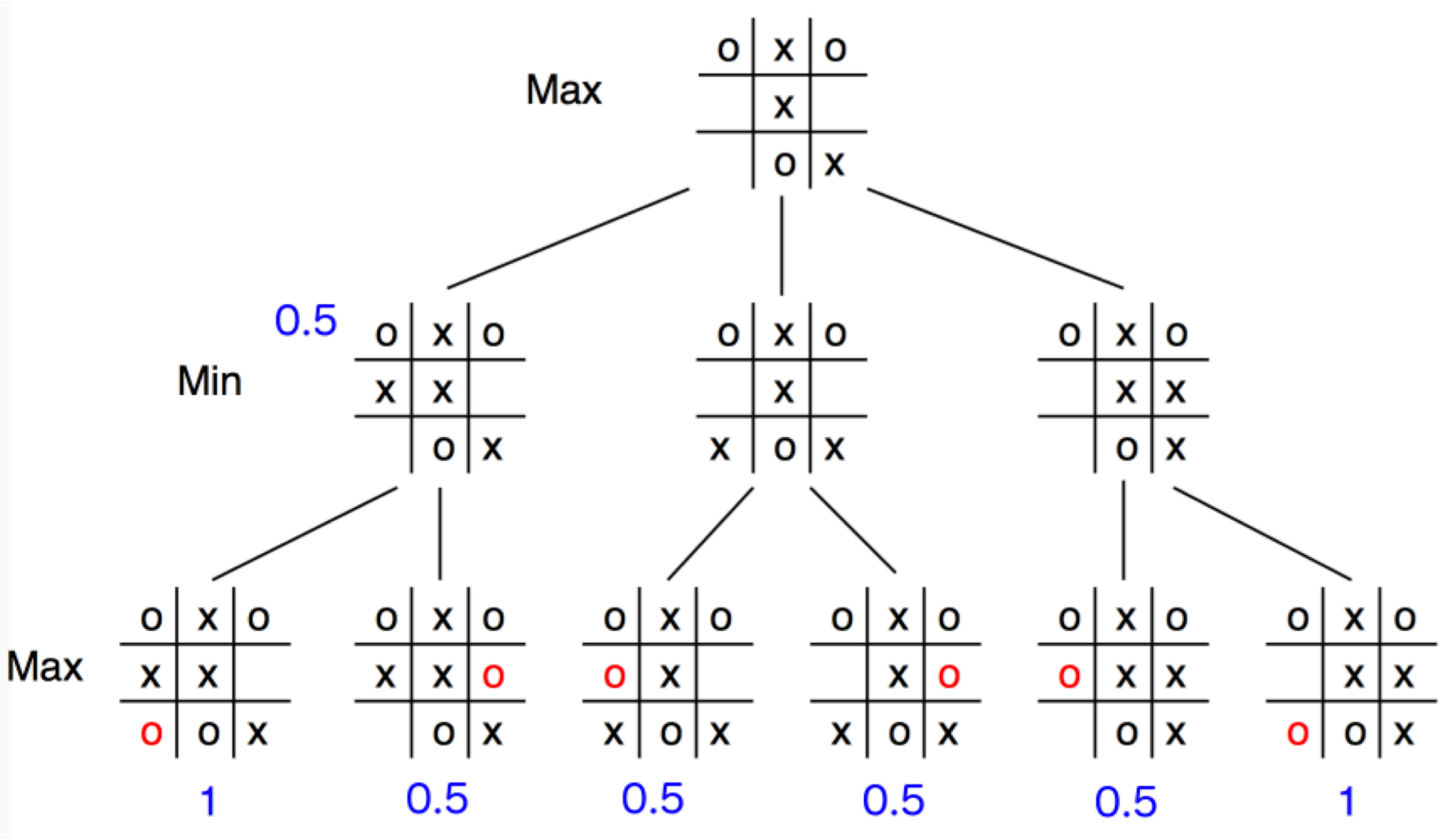
Let's analyze: Tic Tac Toe example



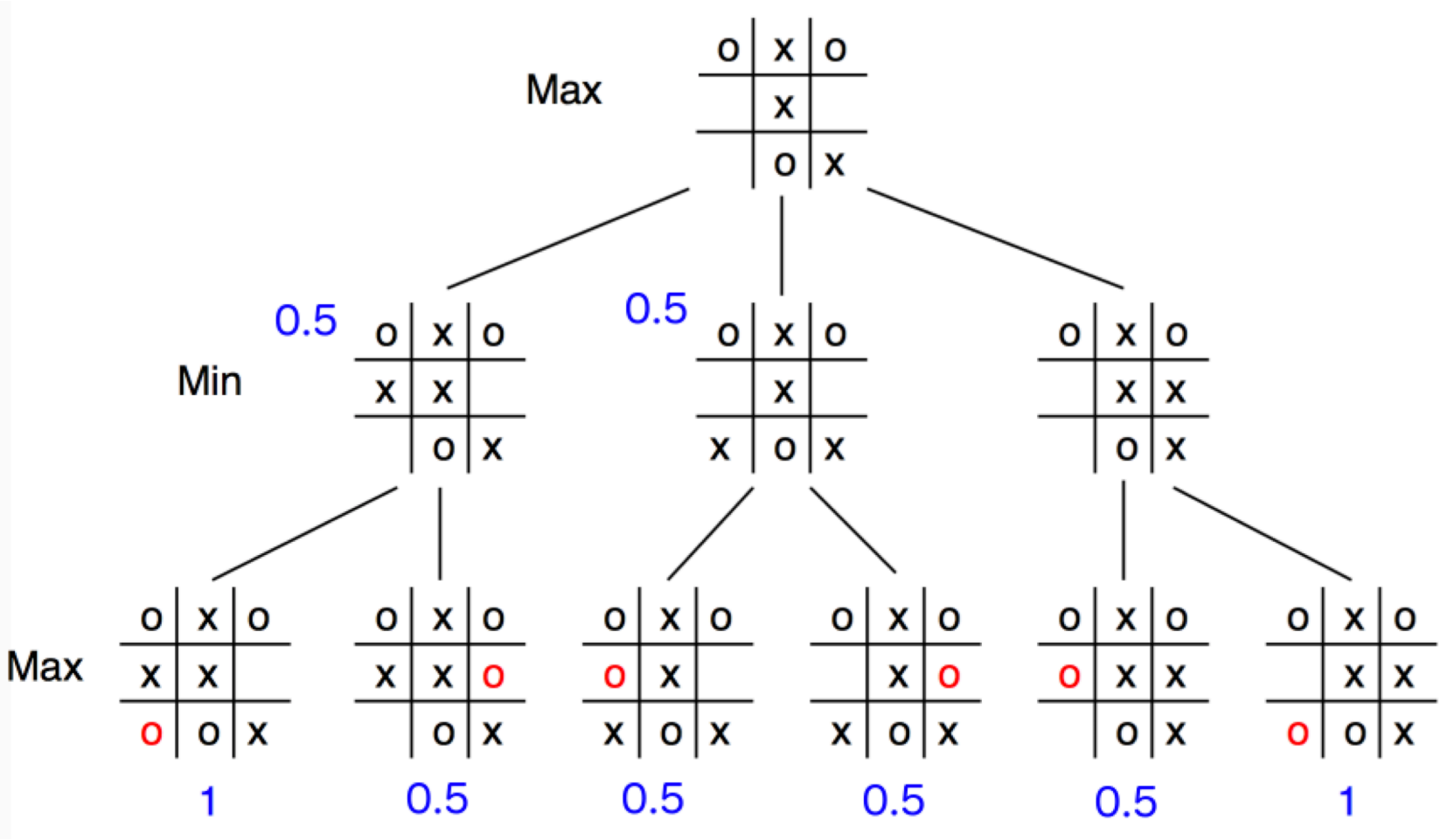
Let's analyze: Tic Tac Toe example



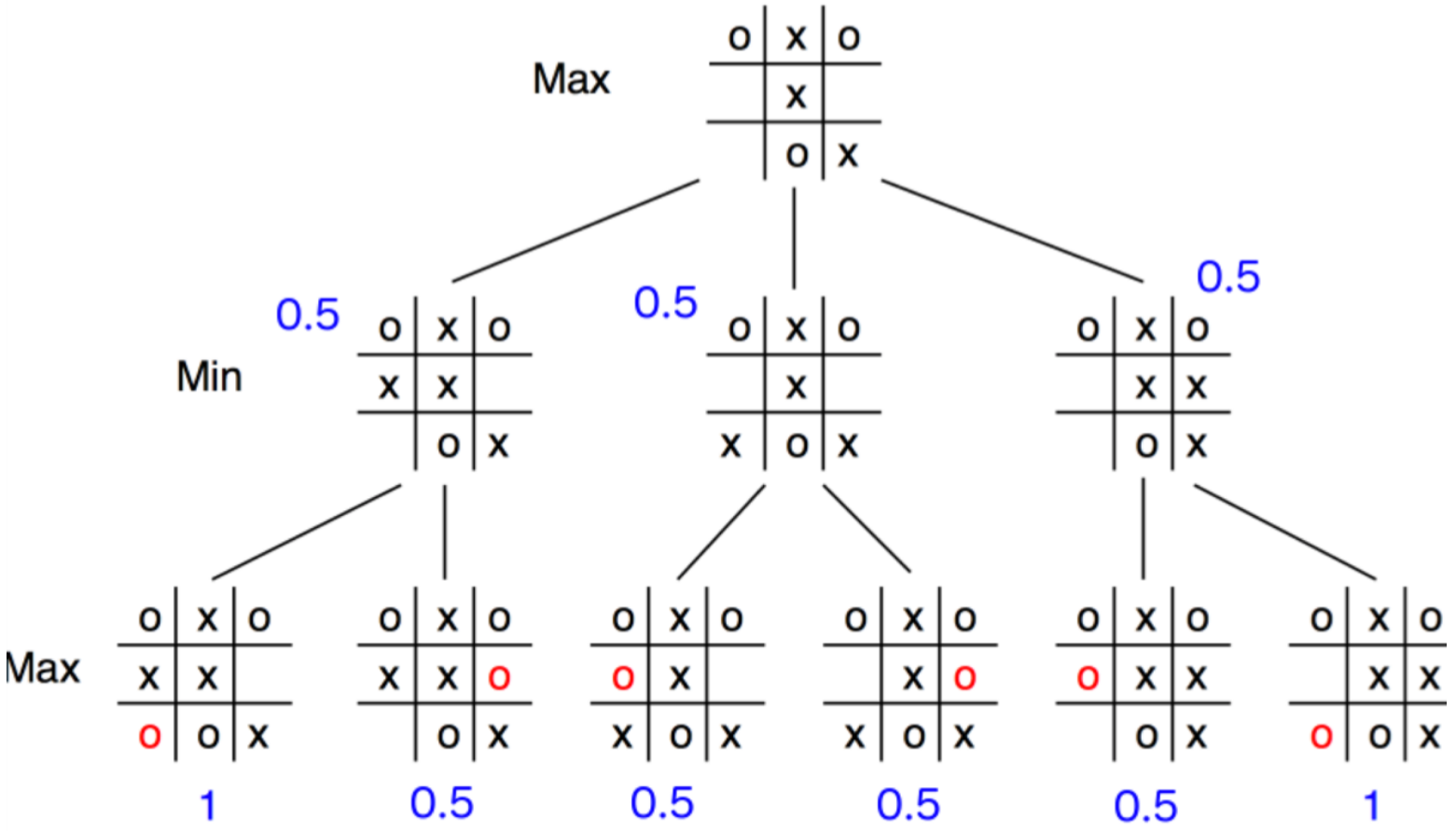
Let's analyze: Tic Tac Toe example



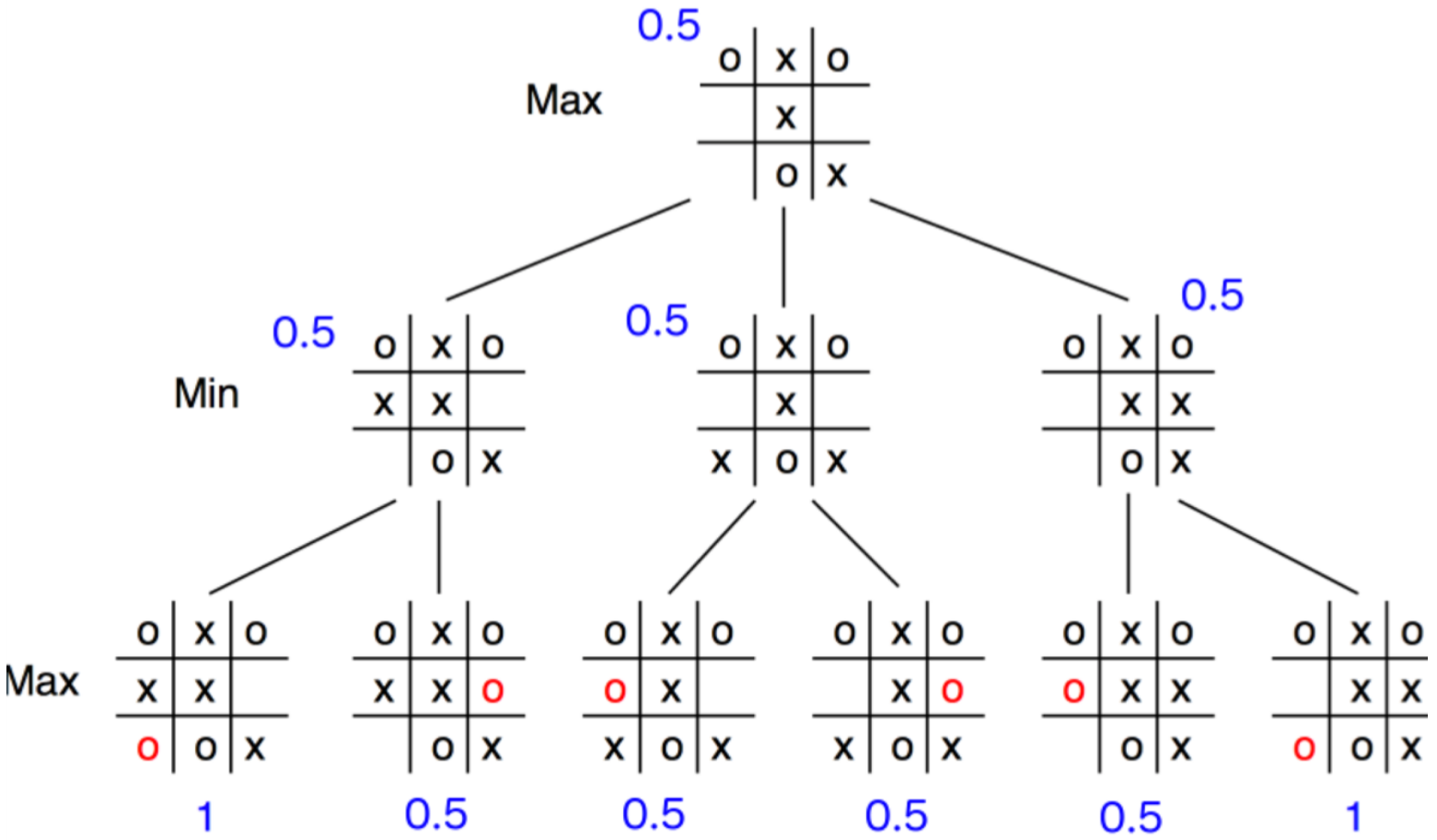
Let's analyze: Tic Tac Toe example



Let's analyze: Tic Tac Toe example



Let's analyze: Tic Tac Toe example



Recall that Properties of Depth-First Search

- b: branching factor
- l: depth of deepest leaf node

- Depth-first search
  - Not complete
  - Not optimal

- Number of nodes generated:

  $1 + b + b^2 + \ldots + b^l = O(b^l)$

- Time complexity $O(b^l)$ and space complexity is $O(b*l)$

Properties of MINIMAX Algorithm

- Minimax performs a full *Depth-First Search (DFS)* of the game tree
- Suppose the maximum depth of the search tree is l and the number of choices at each step is at most b
- Time complexity? $O(b^l)$
- Space complexity? $O(bl)$
- For chess, $b \approx 35$, $l \approx 100$ for "reasonable" games. Exact solution completely infeasible

Issues with Minimax

- The issue with minimax algorithm is the number of game states it has to examine is exponential in the depth of the tree $O(b^l)$
- Can we feasibly look at every node in the tree?
- How do we find shortcuts to save computation?

How can you improve Minimax?

- The key problem with minimax algorithm is that the number of game states it has to examine is exponential in the depth of the tree $O(b^l)$

- We can't eliminate the exponent, but we can effectively cut it in half

- Idea: don't visit nodes that won't make a difference

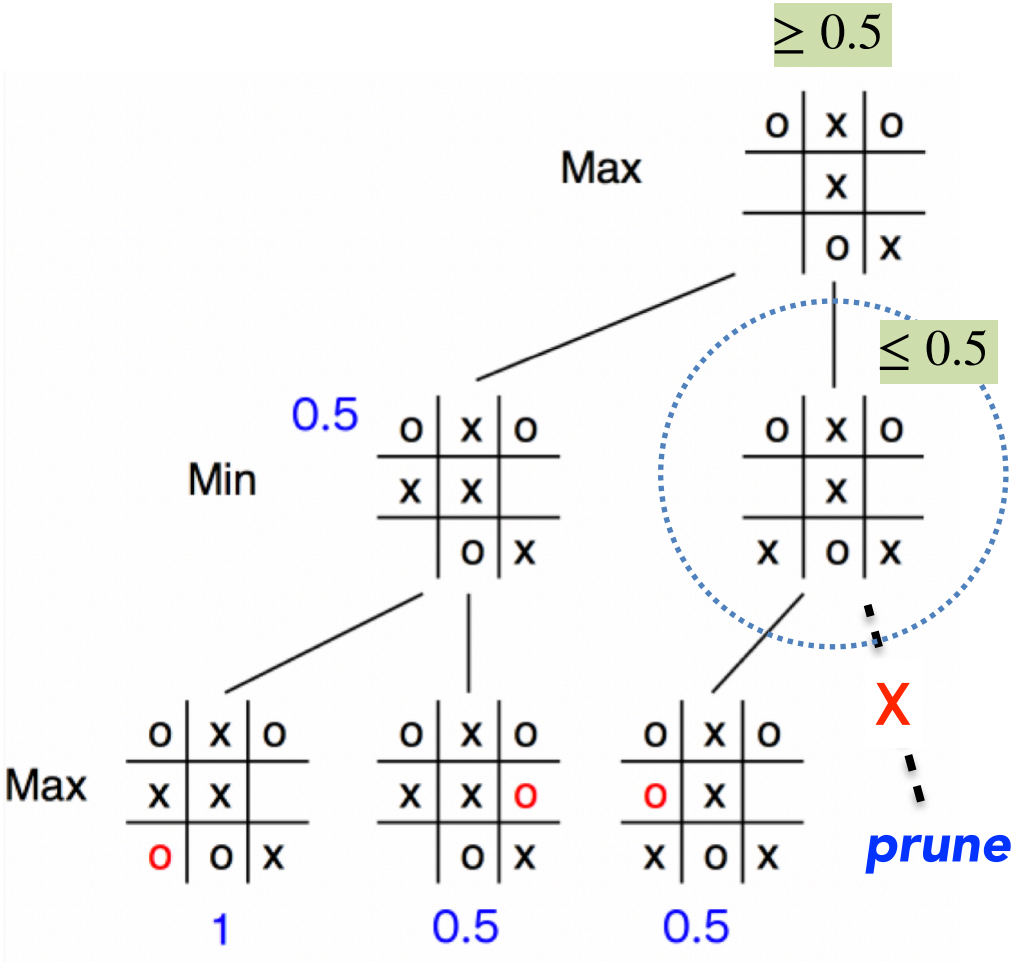
Key Observation 0.5

- When we get to this point, we don't have to look at MIN's next child

  Why not?

- When part of the tree is ignored, this called ***pruning***

- In the context of the minimax algorithm, this is called ***α-β pruning***

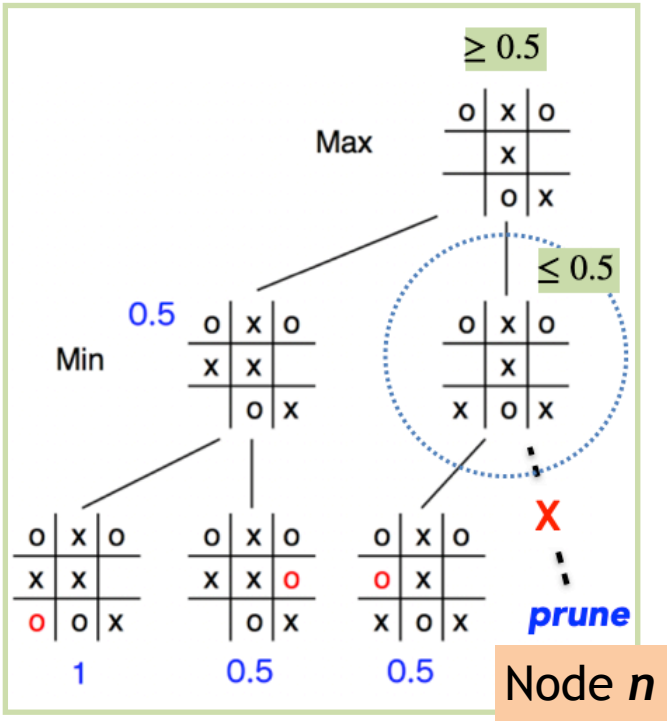


α-β pruning

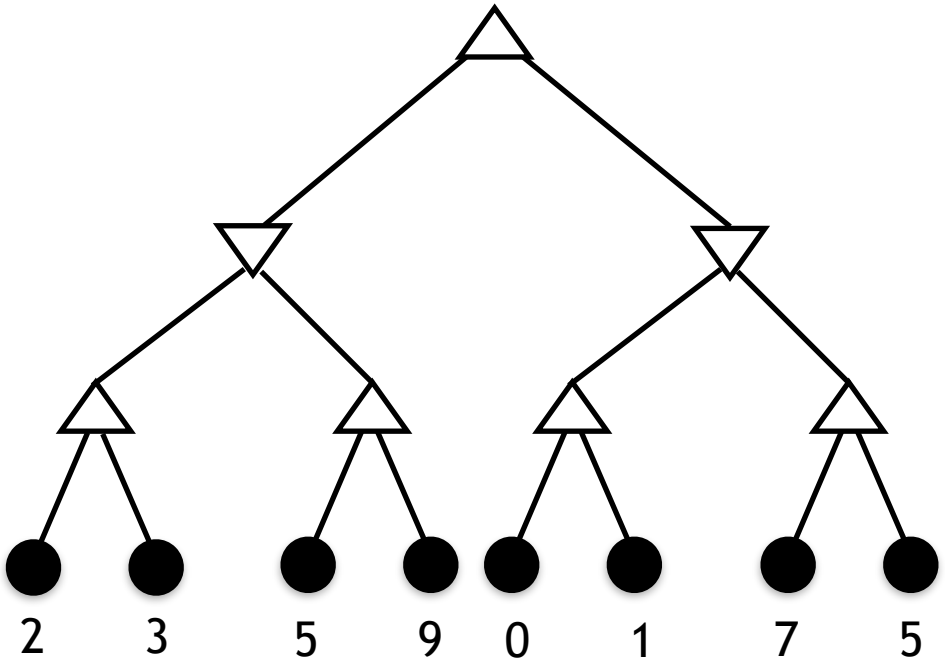
α-β Pruning: Main Idea

- Consider a node n that could be visited
- If a player has a better choice m at the parent of node n (or at any point farther up the tree), n will never be reached in actual play

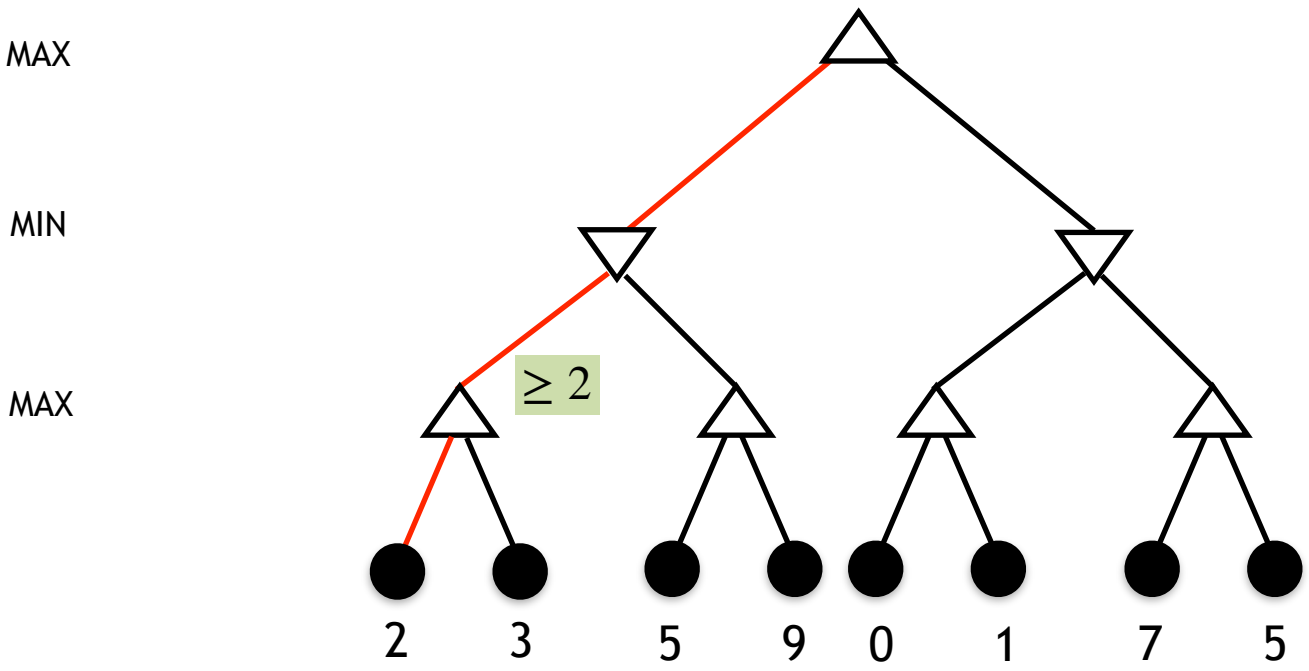
- If so, prune it



α-β Pruning Example

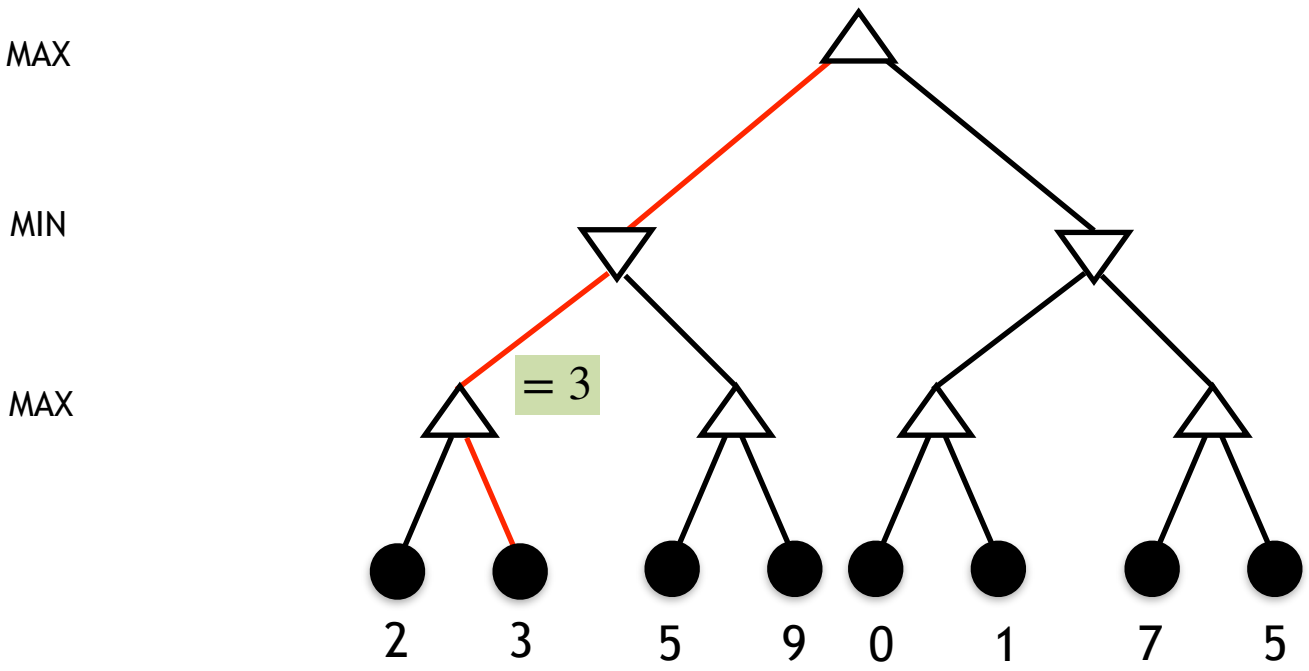


α-β Pruning Example



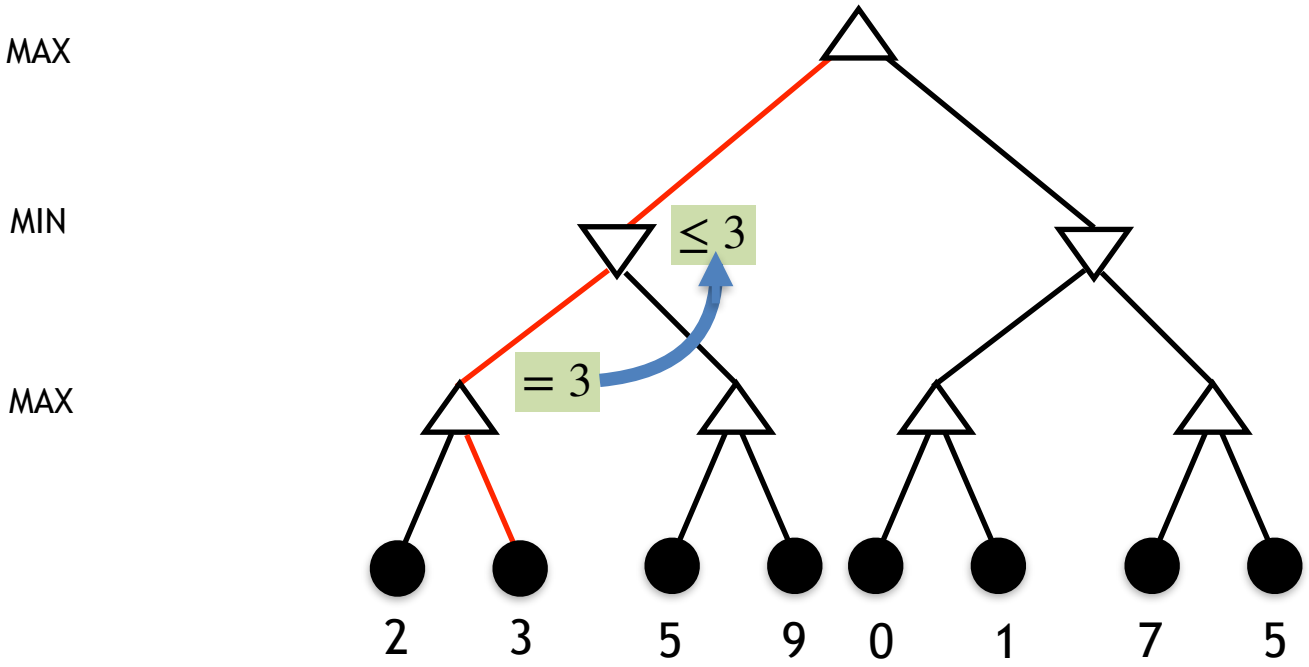
MAX node gets value at least 2
It expects more from the other unvisited branch

α-β Pruning Example



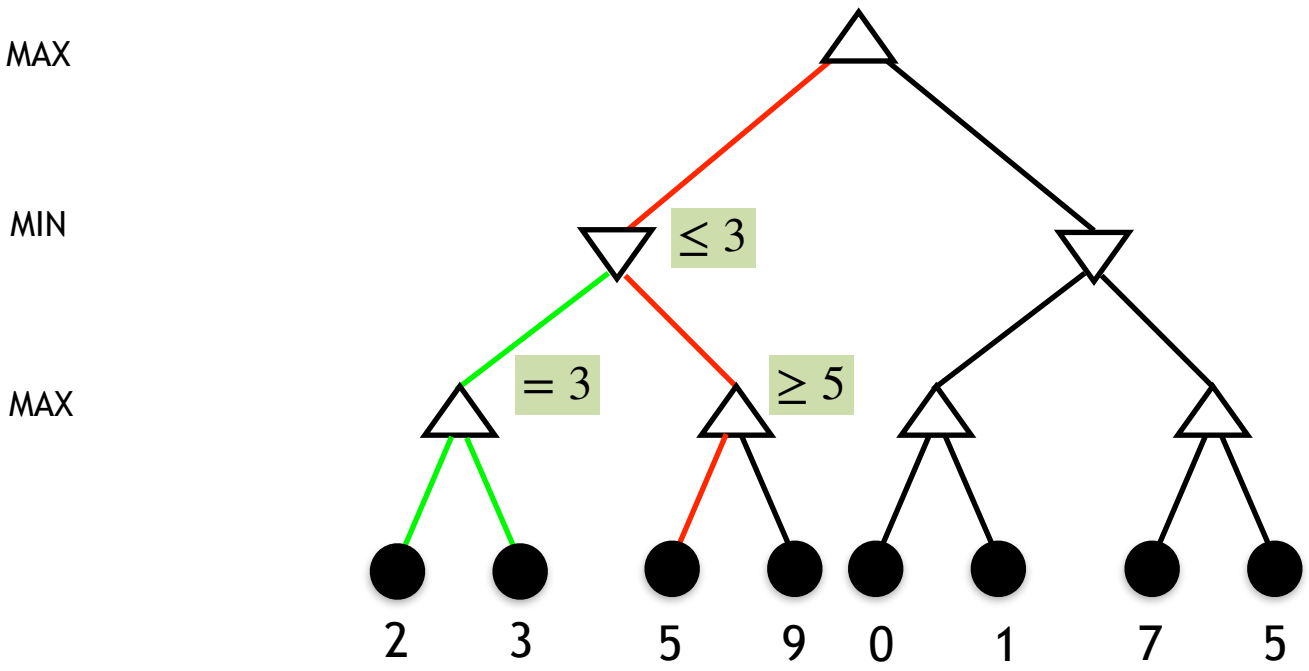
MAX node gets updated to exact value of 3

α-β Pruning Example



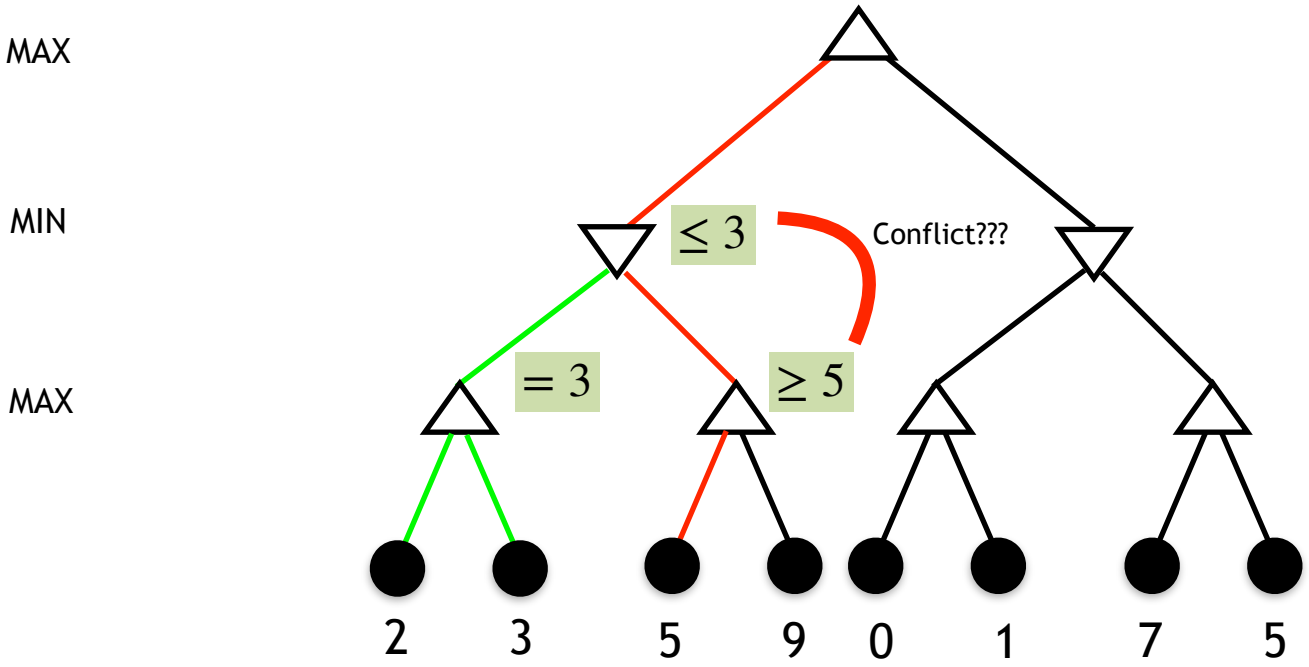
MAX node pushes the value upwards to MIN node
MIN gets a value of at least 3
It expects less than 3 from the other unvisited branch

α-β Pruning Example



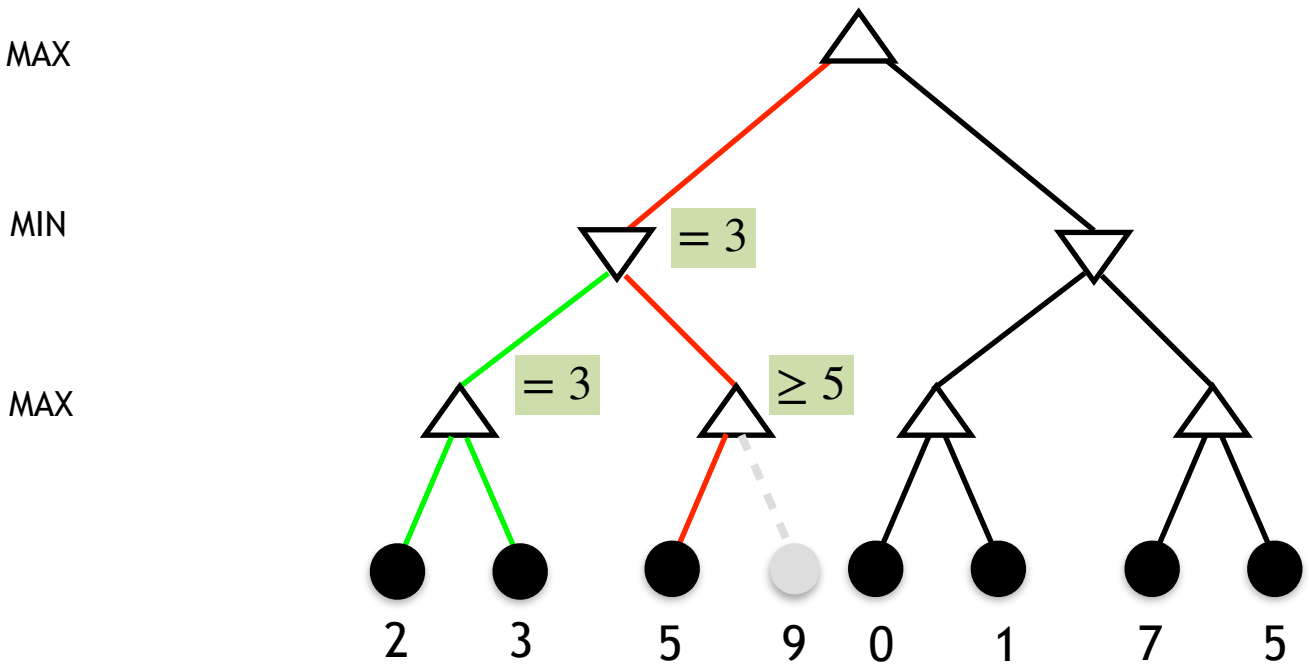
MAX node gets value at least 5
It expects more from the other unvisited branch

α-β Pruning Example



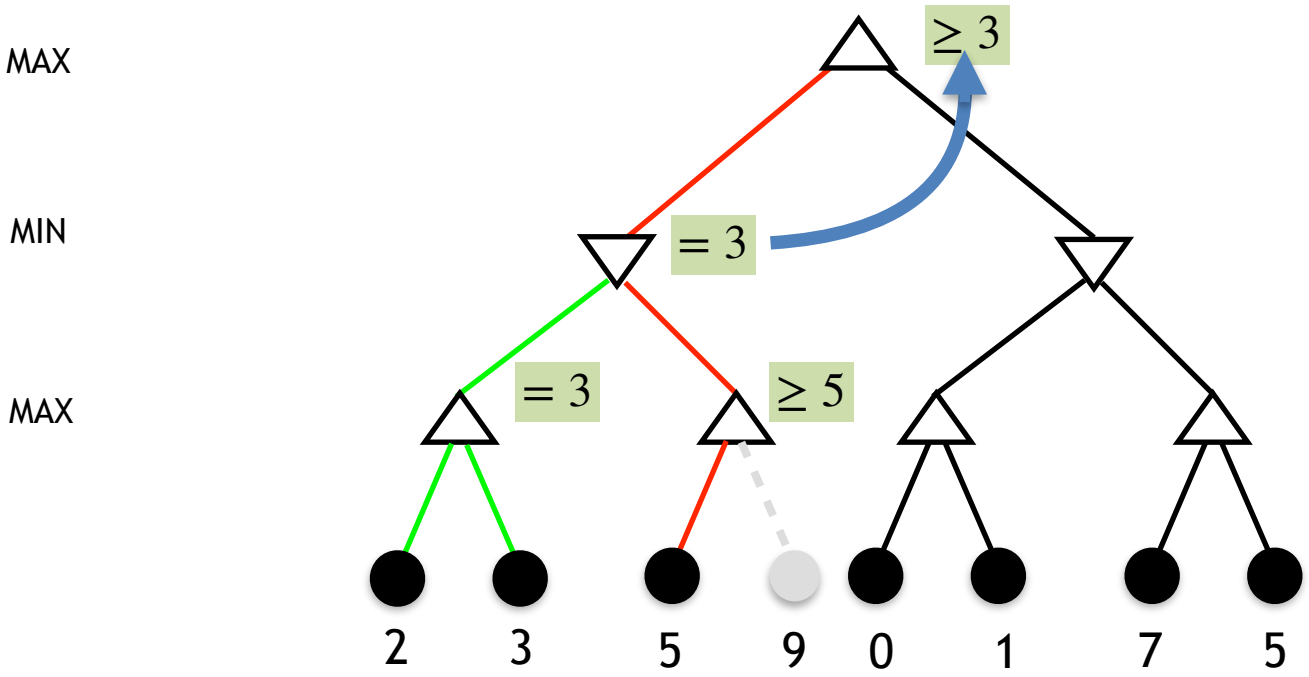
However, MAX's parent (which is a MIN) can never get more than 3
hence prune other branches from the MAX node

α-β Pruning Example



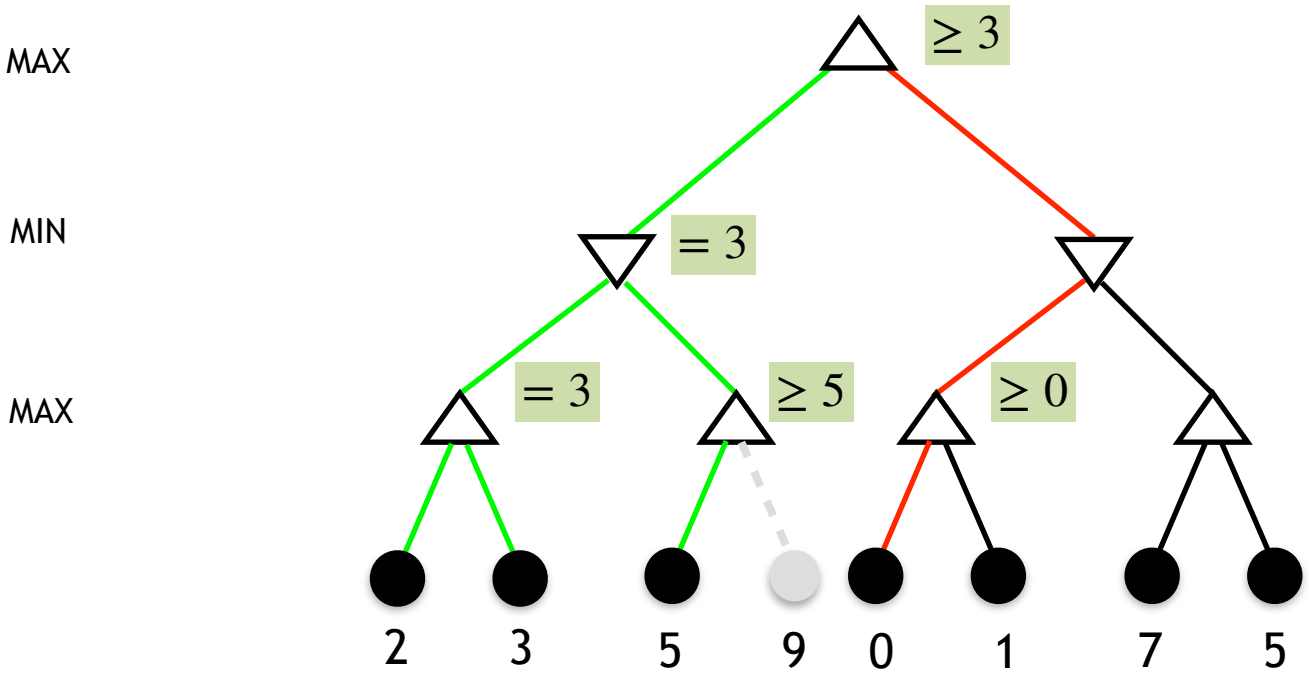
Prune other branches from the MAX node (highlighted in gray with 9)
MAX's parent (which is MIN) gets updated to exact value of 3

α-β Pruning Example



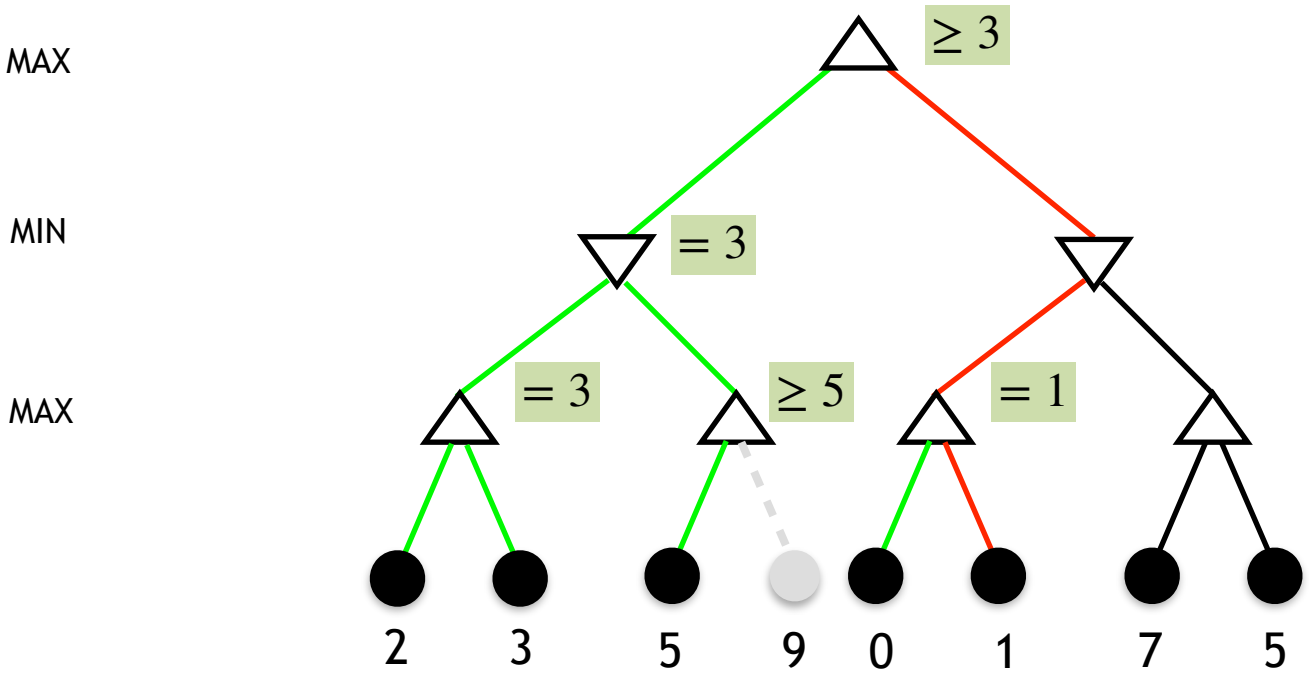
MAX (MIN's parent) gets value at least 3
It expects more from the other unvisited branch

α-β Pruning Example



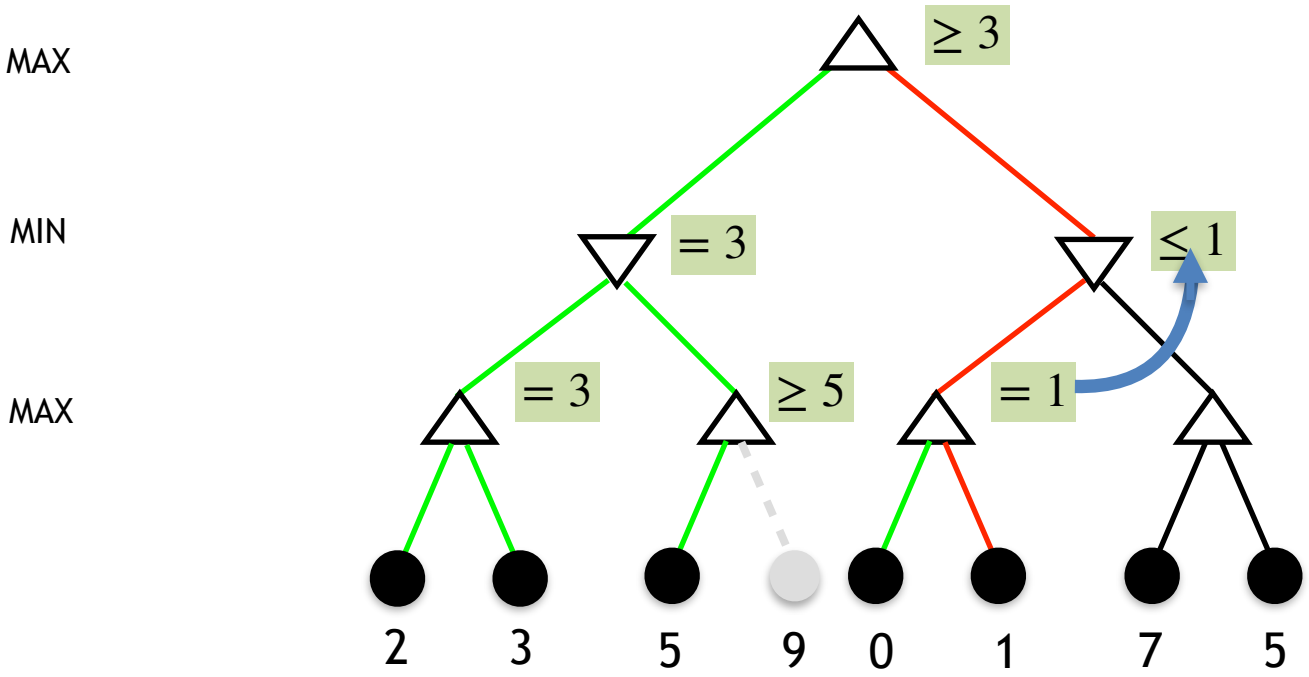
MAX gets value at least 0
It expects more from the other unvisited branch

α-β Pruning Example



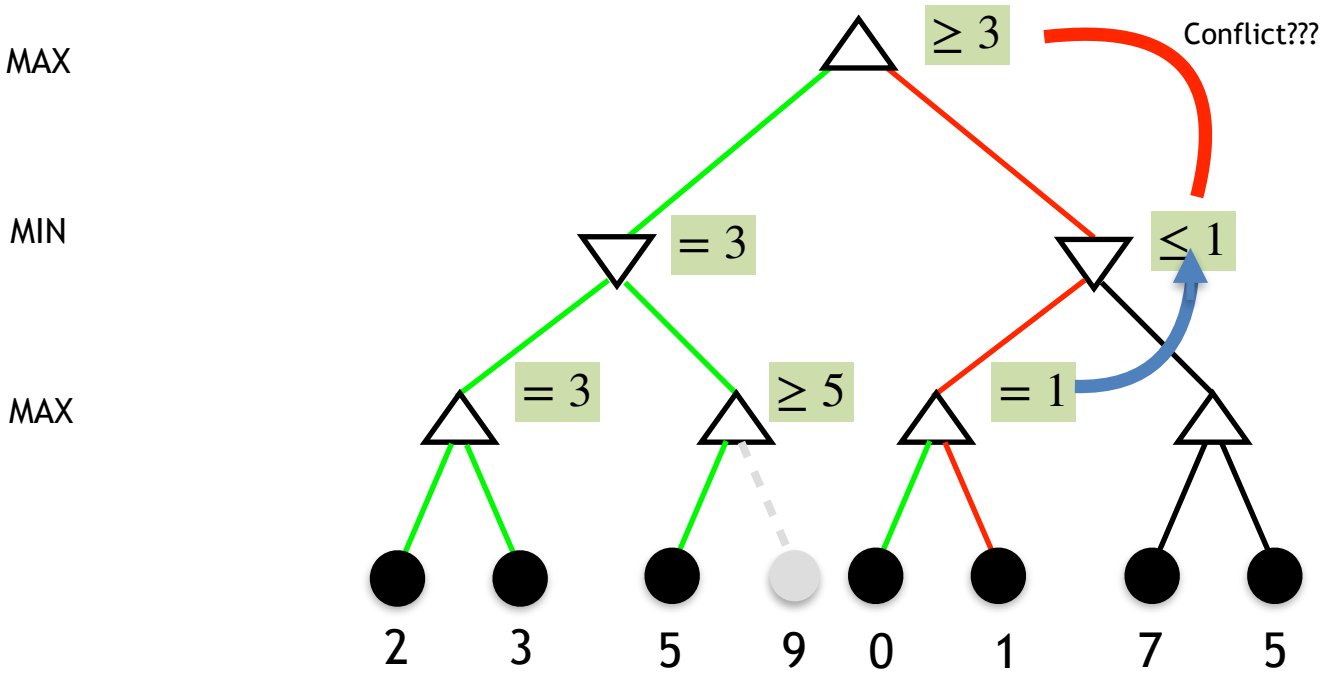
MAX node gets updated to exact value of 1

α-β Pruning Example



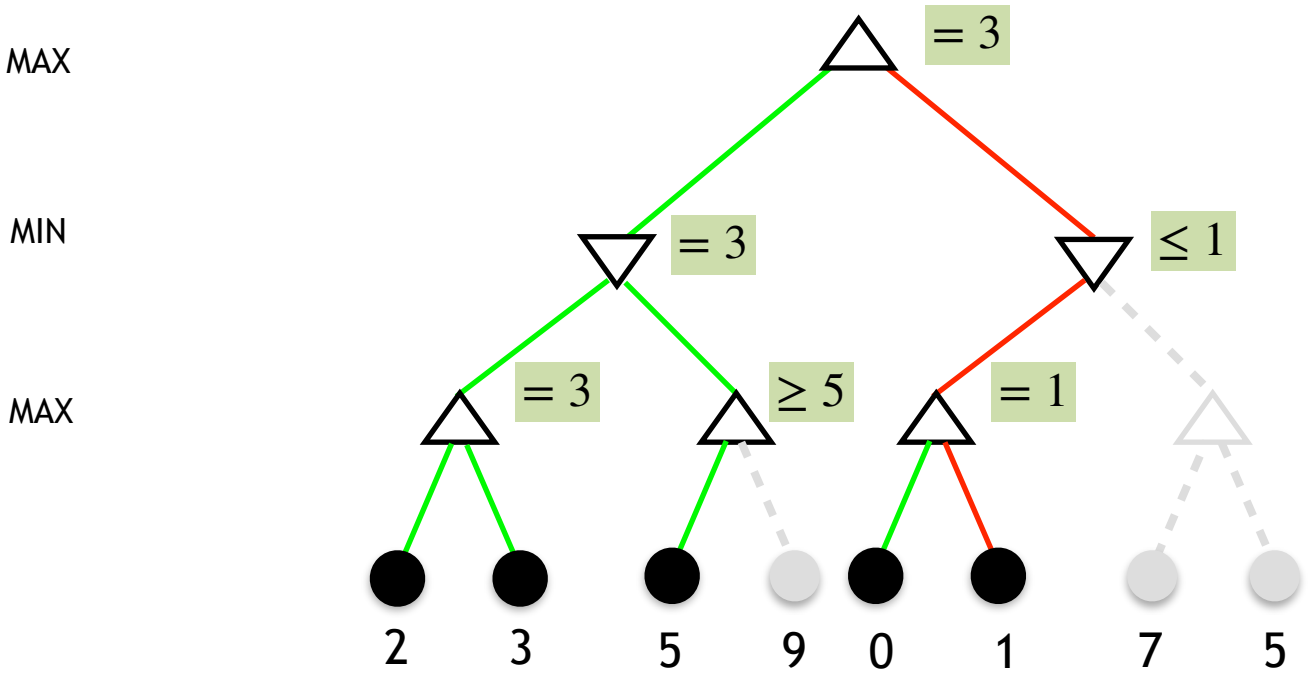
MAX node pushes the value upwards to MIN node
MIN gets a value of at least 1
It expects less than 1 from the other unvisited branch

α-β Pruning Example



However, MIN's parent (which is a MAX) can never get less than 3
hence prune other branches from the MIN node

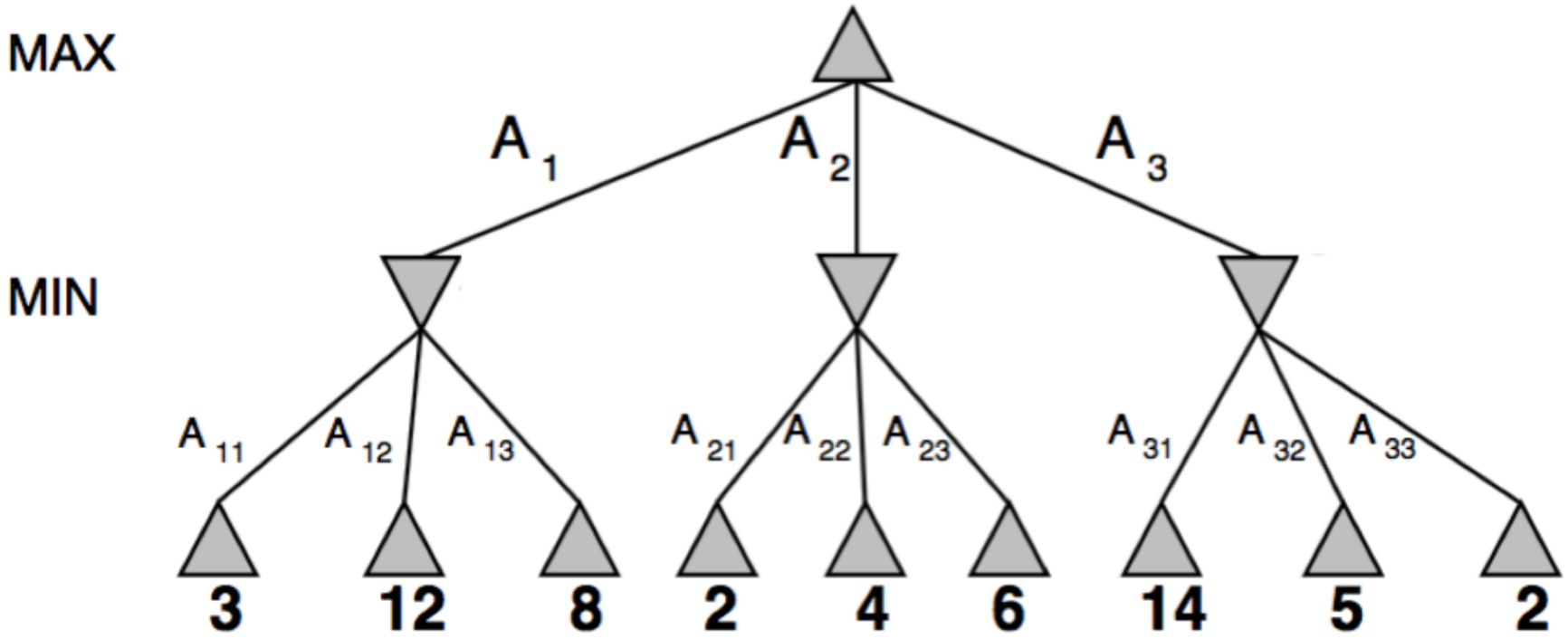
α-β Pruning Example



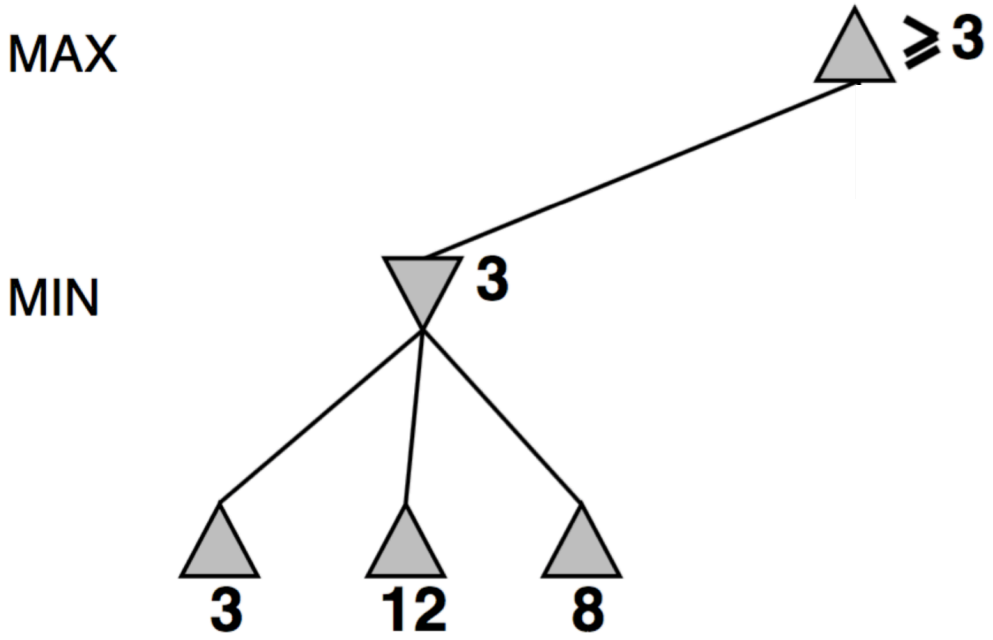
Prune other branches from the MIN node (highlighted in gray)
MIN's parent (which is MAX) gets updated to exact value of 3

α-β pruning: Another Example

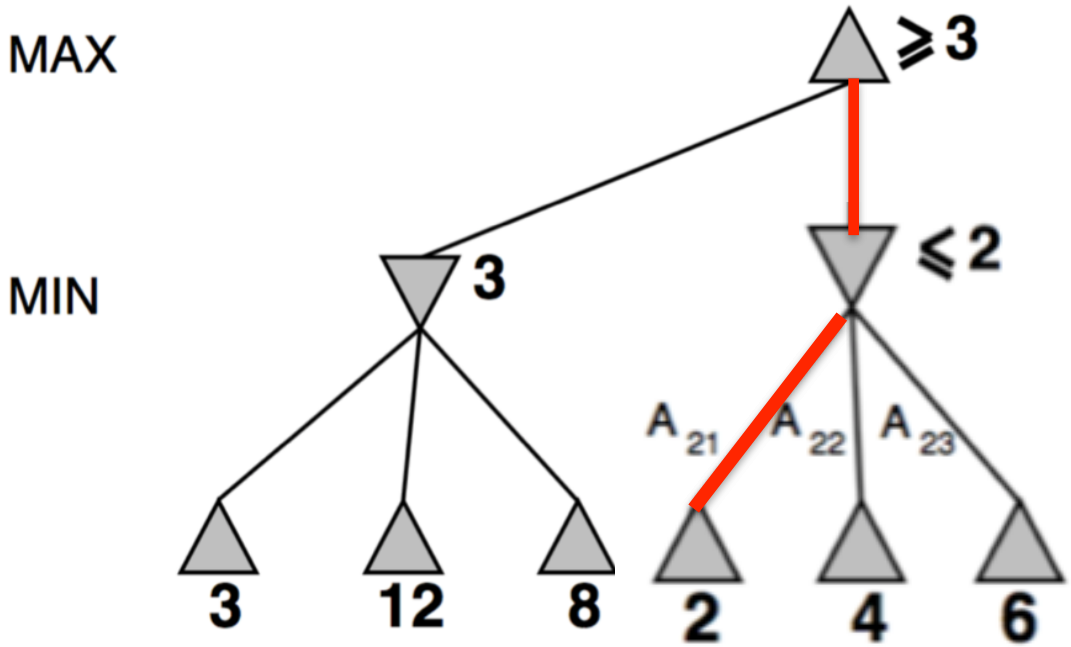
α-β Pruning Example



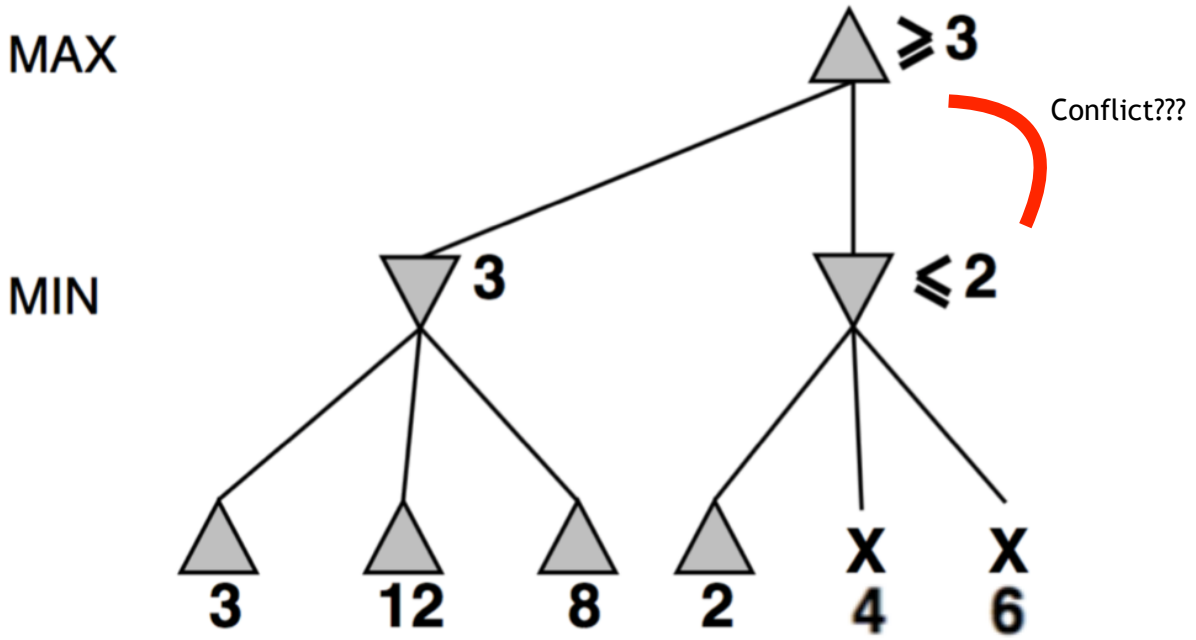
α-β Pruning Example



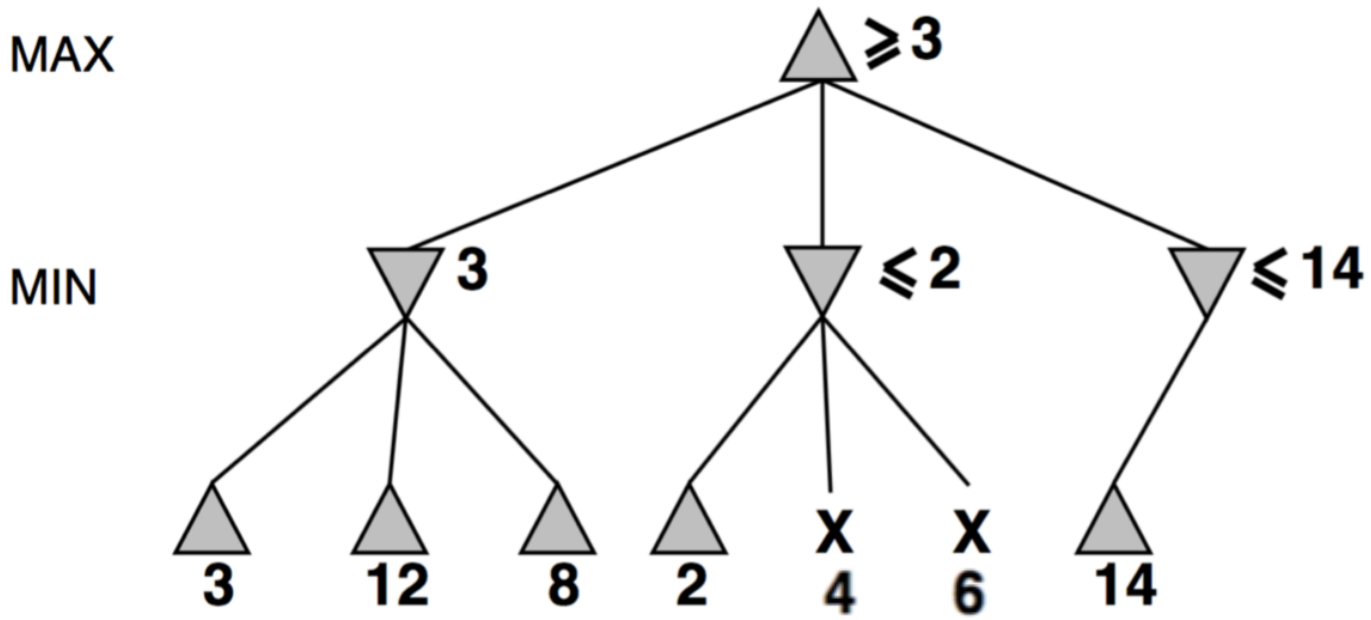
α-β Pruning Example



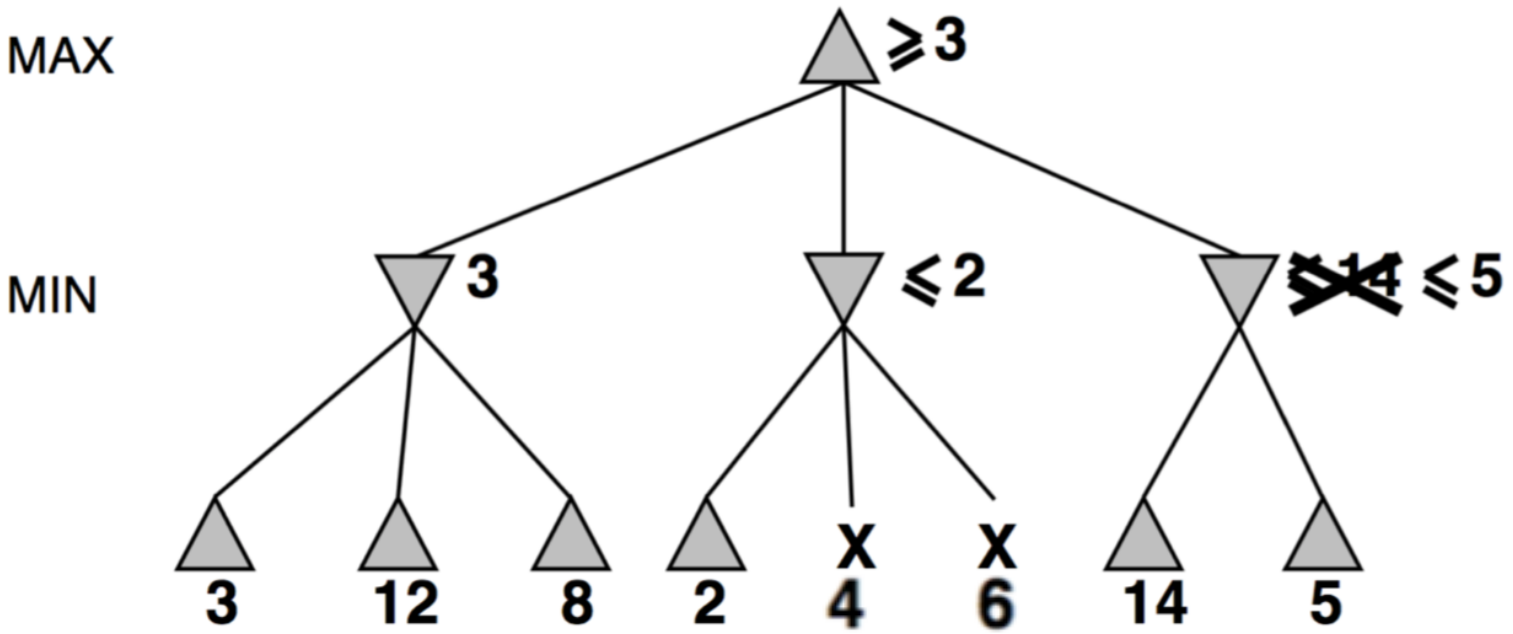
α-β Pruning Example



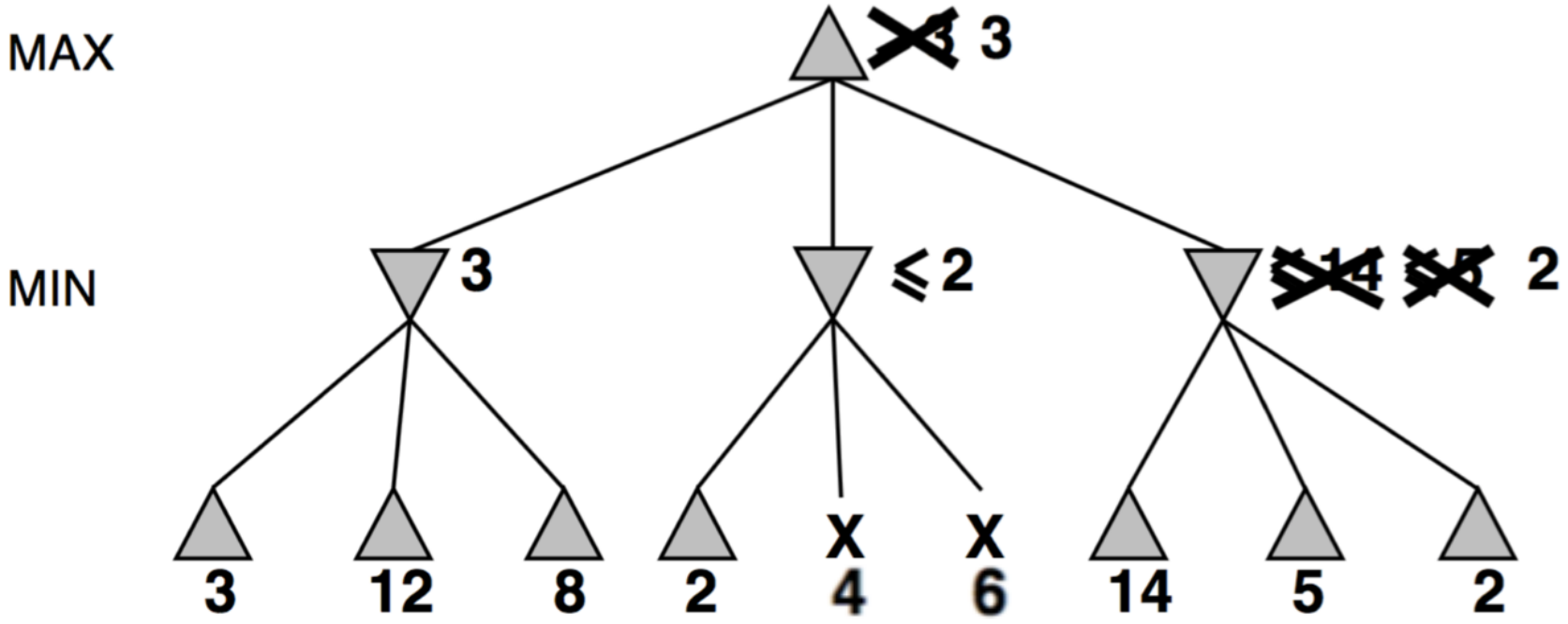
α-β Pruning Example



α-β Pruning Example

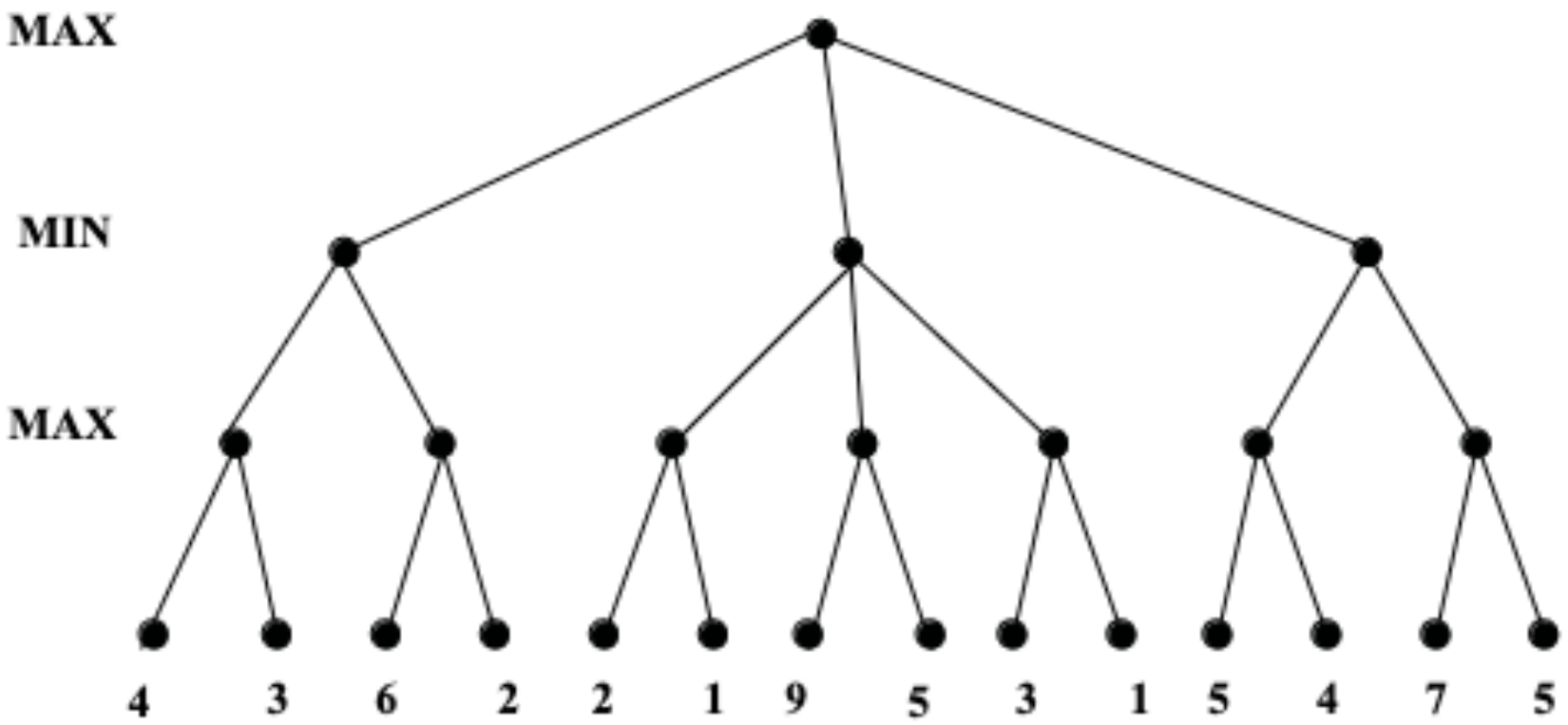


α-β Pruning Example

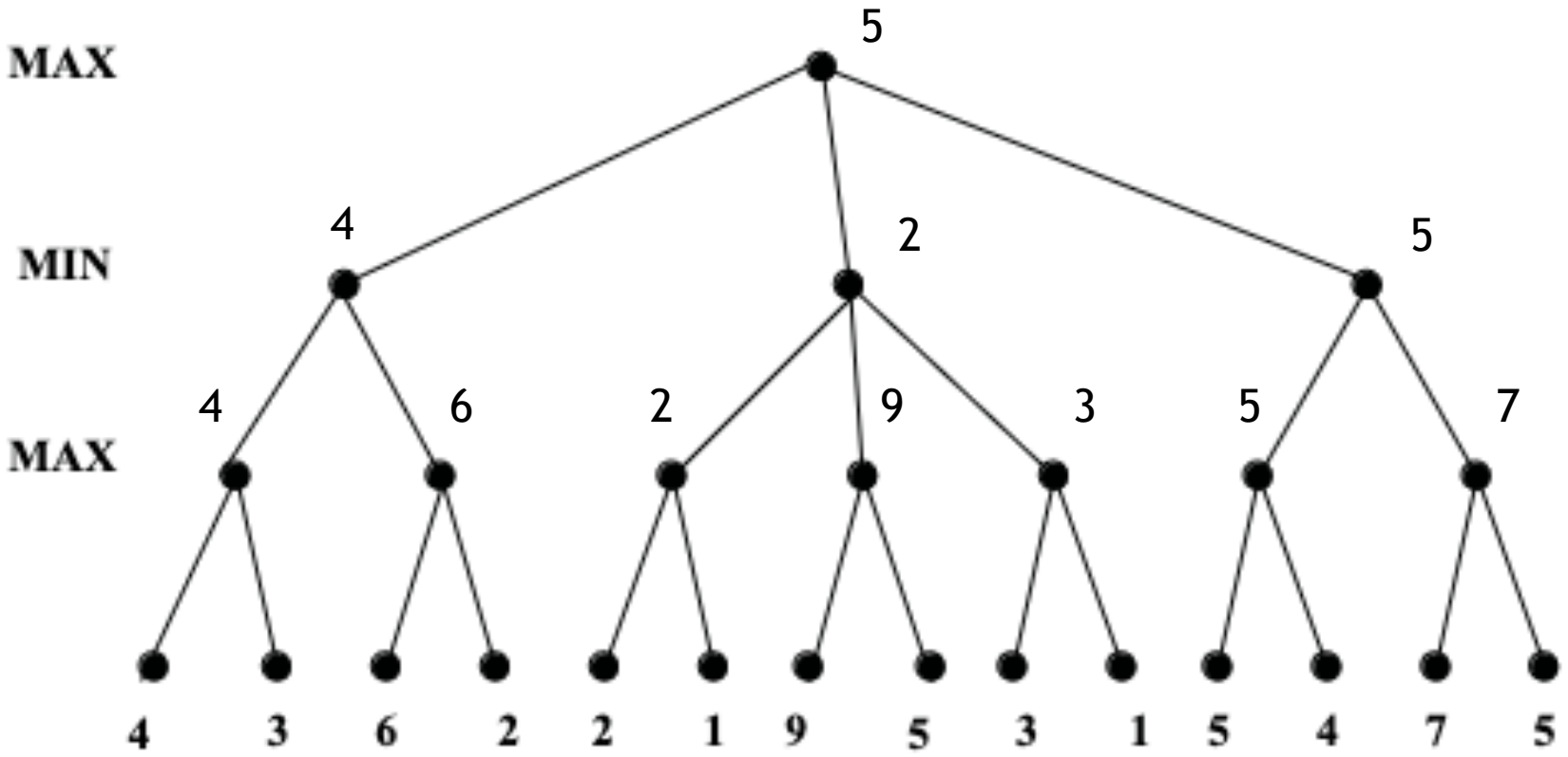


α-β pruning Activities

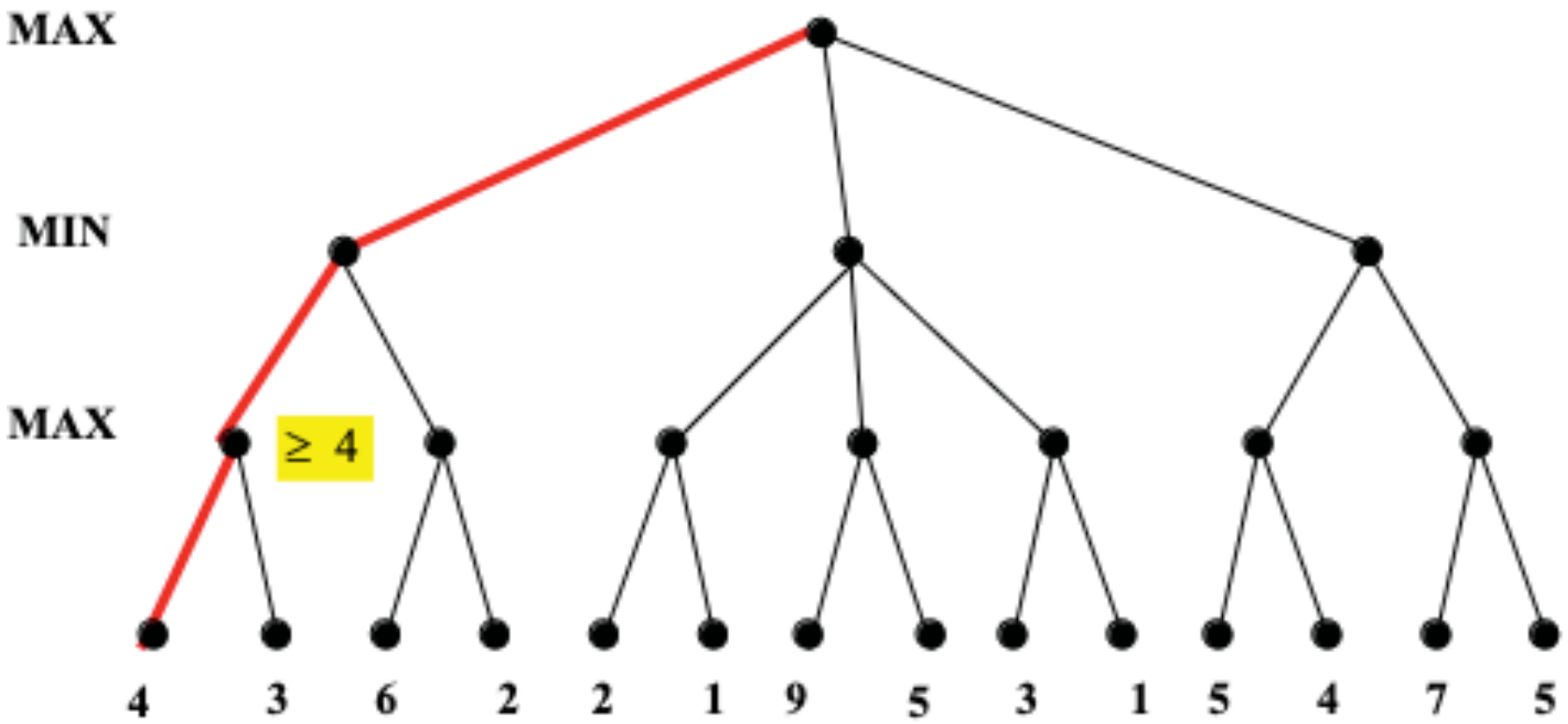
Minimax Algorithm (no pruning)



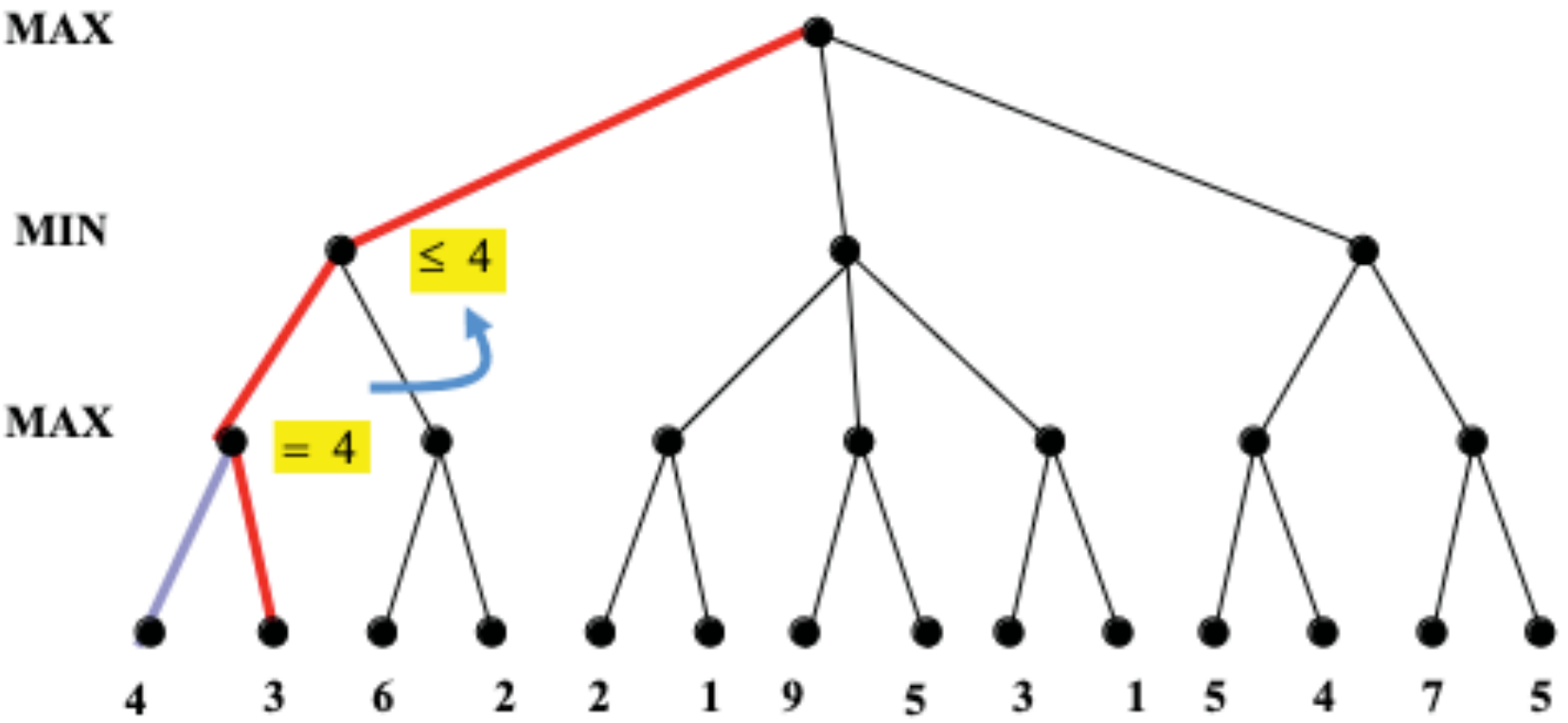
Minimax Algorithm (no pruning)



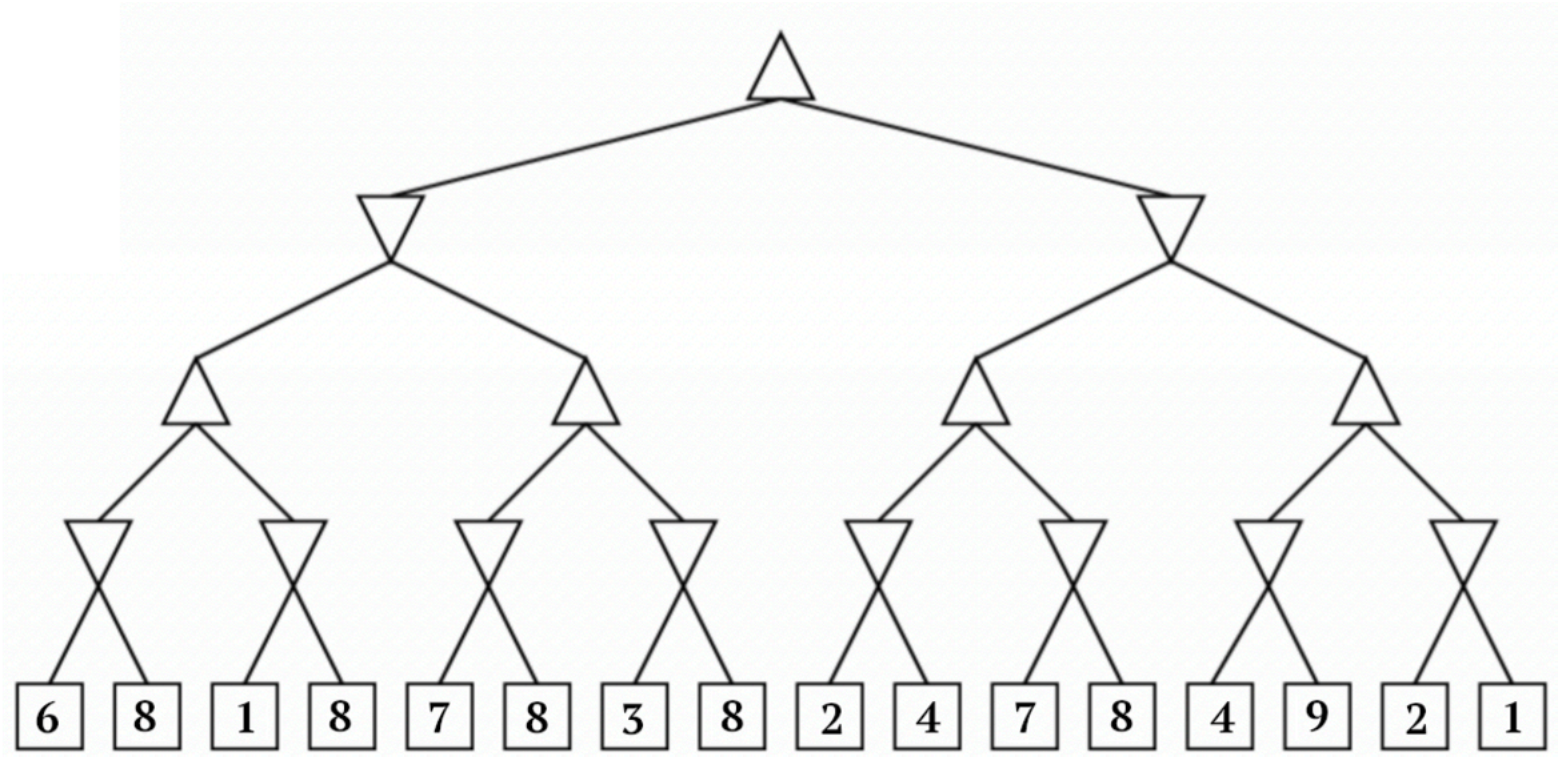
α-β Pruning Activity



α-β Pruning Activity



Additional α-β Pruning example for exam



α-β Pruning Pseudocode: Step-by-Step Execution