CSE422: Artificial intelligence

Adversarial Search

Game playing using Minimax algorithm

Asif Shahriar
Lecturer, CSE, BRACU

WHY Adversarial Search

- So far we have seen TWO intelligent search techniques
- **Informed search (A-star search)**
  - Finds the optimal path to a **known** goal
- **Local search (Hill climbing, simulated annealing, genetic algorithm)**
  - Optimize a solution when the full search space is too large
  - No notion of path — only focuses on current state and neighbors
- Both search techniques assume:
  - The world **does not change** unless the agent acts
  - There is no **opponent** in the search space – agent is working alone
- In many real-world settings, the environment is **non-deterministic** and **competitive**
  - The outcome depends not only on your actions, but also on the opponent's actions
  - You are not just finding a path — you are **outsmarting an opponent**

Adversarial Search

- A search framework used in competitive, multi-agent environments
- Agents (players) take turns, and each tries to maximize their own utility (chances of winning the game or better their positions)
- Standard model for two-player **zero-sum games**
  - **Zero-sum:** one player's loss is the other's gain (chess, rock-paper, 1v1 poker)
  - **Non-zero-sum game:** players may both (all) gain or lose (monopoly, team games)
- Adversarial search assumes **perfect information:** both players have access to complete information about the state of the game
  - No chance (e.g., using dice) involved
- Adversarial search helps an agent make the best move **assuming the worst-case opponent response**

How to PLAY a Game

- A way to play such a game is to:
  - Consider **all legal moves** you can make
  - Compute the new position resulting from each move
  - **Evaluate** each resulting position and **determine** which is best
  - Make that move
  - Wait for your **opponent** to move
  - Repeat
- Key problems:
  - Representing the “game”
  - Generating all legal next boards
  - **Evaluating a position**

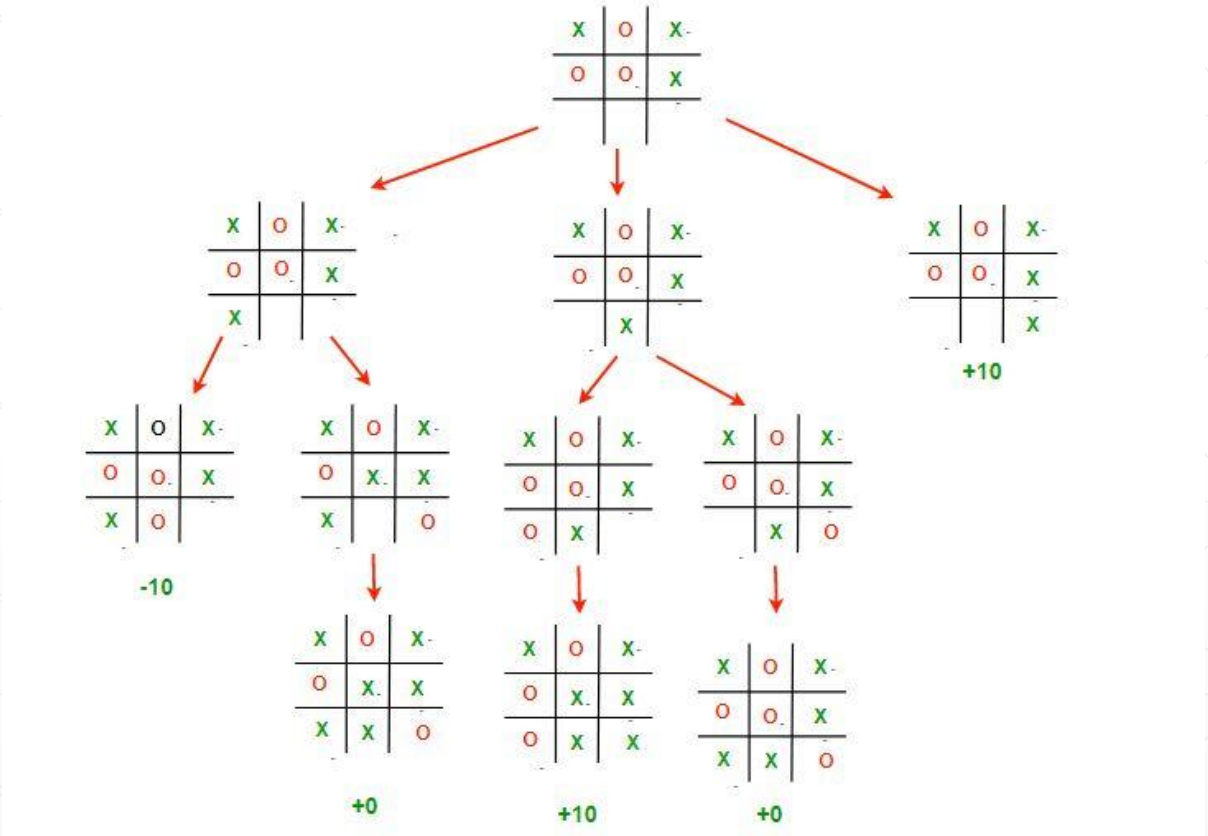
Let’s play a game - 1

- Suppose you and your friend both got caught doing something criminal and have been brought in for interrogation
- If both deny, both get 10 years sentence
- If both accept, both get 5 years sentence
- If one accepts and other deny, the one who denies goes free but the one who accepts get 10 years sentence
- What you gonna do?

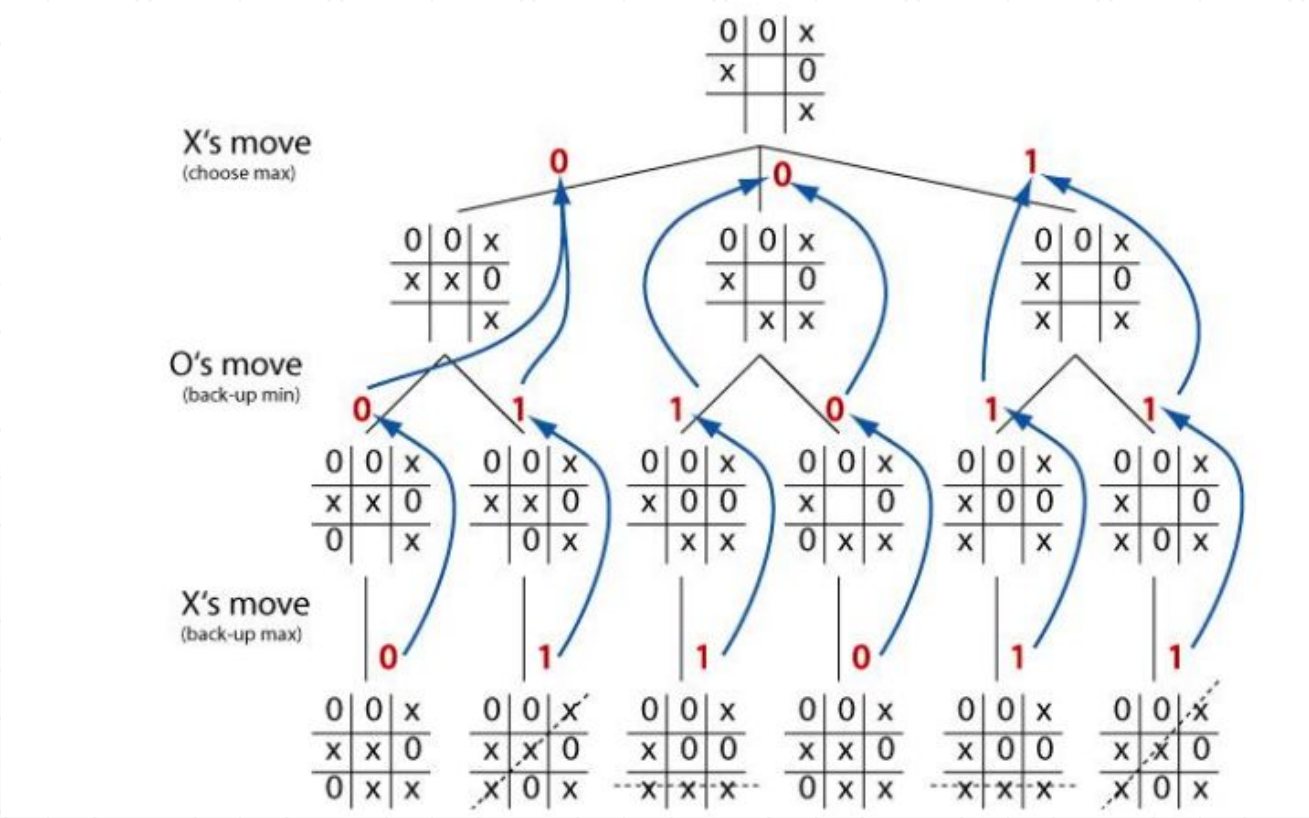
Let’s play a game - 2

- There are four coins and 2 players
- Each player can take ONE or TWO coins
- The one who takes the last coin, loses
- You can choose to go first or go second
- What you gonna do?

Let's Play TIC-TAC-TOE 1



Let's Play TIC-TAC-TOE 2



Evaluating a Position

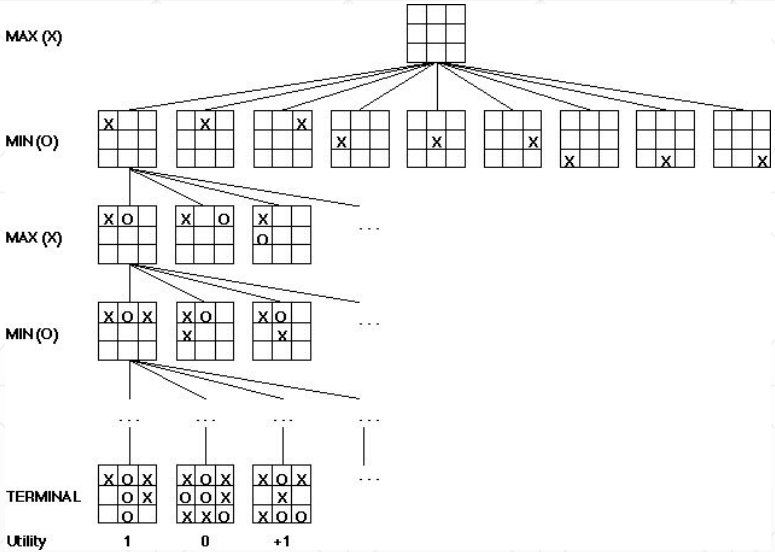
- **Evaluation function** or static evaluator is used to evaluate the “goodness” of a game position
  - Contrast with heuristic search where the evaluation function was a non-negative estimate of the cost from the start node to a goal and passing through the given node
- The zero-sum assumption allows us to use a **single evaluation function** to describe the goodness of a board with respect to both players
  - f(n) >> 0: position n good for me and bad for opponent
  - f(n) << 0: position n bad for me and good for opponent
  - f(n) = 0: position n is a neutral position
  - f(n) = +infinity: win for me
  - f(n) = -infinity: win for opponent

Example Evaluation Functions

- Example of an evaluation function for Tic-Tac-Toe:
  - f(n) = [# of 3-lengths open for me] - [# of 3-lengths open for you]
  - where a 3-length is a complete row, column, or diagonal
- Alan Turing's **function for chess**
  - **f(n) = w(n)/b(n)** where w(n) = sum of the point value of white's pieces and b(n) = sum of black's
  - Piece value: Queen = 9, rook = 5, bishop / knight = 3 (but bishop pair = 7 instead of 6), pawn = 1
- Most evaluation functions are specified as a weighted sum of position features:
- f(n) = w1 * f1(n) + w2 * f2(n) + … + wn * fn(n)
- Features for chess are piece count, piece placement, squares controlled, etc

Game Trees

- Problem spaces for typical games are represented as **trees**
- Root node represents the current board configuration; player must decide the best single move to make next
- Static evaluation function rates a board position. f(board) = real number with f>0 (me better), f<0 (opponent better), f=0 (equal)
- Arcs represent the possible legal moves for a player
- If it is my turn to move, then the root is labeled a **"MAX"** node; otherwise it is labeled a **"MIN"** node, indicating my opponent's turn
- Each level of the tree has nodes that are all MAX or all MIN; nodes at level i are of the opposite kind from those at level i+1



Game Trees

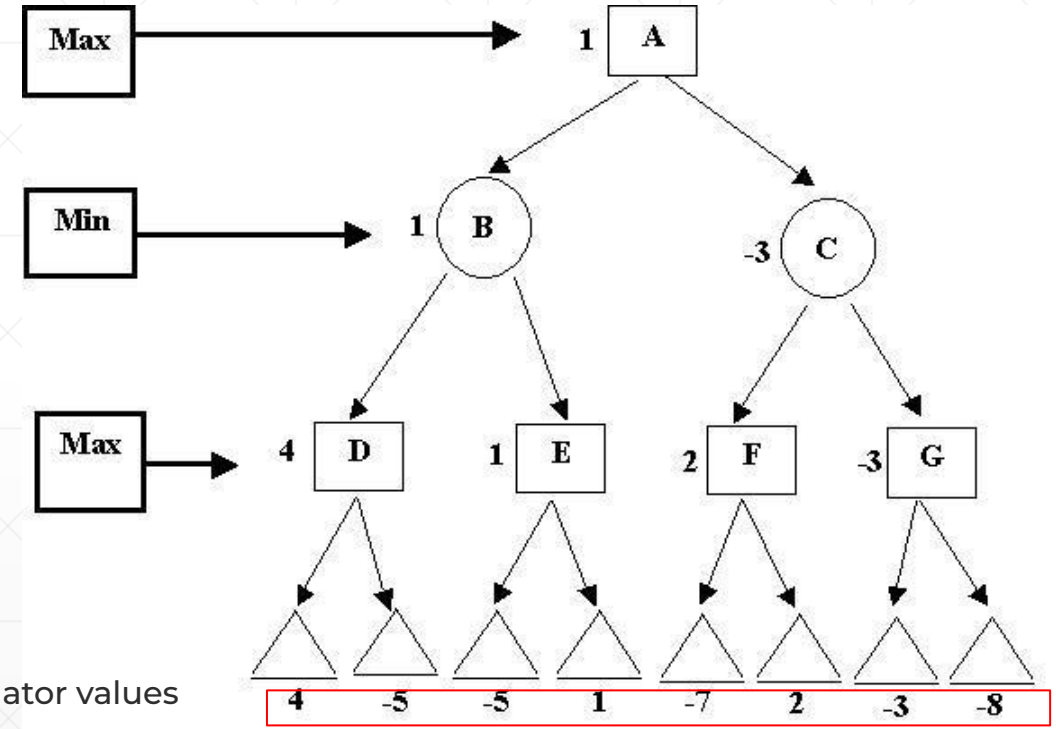
- Create **start** node as a **MAX** node with current board configuration
- Expand nodes down to some **depth** (a.k.a. ply) of lookahead in the game
- Apply the **evaluation function** at each of the **leaf** nodes
- **"Back up"** values for each of the **non-leaf** nodes until a value is computed for the root node
- At **MIN** nodes, the backed-up value is the **minimum** of the values associated with its children.
- At **MAX** nodes, the backed-up value is the **maximum** of the values associated with its children.
- Pick the operator associated with the child node whose backed-up value determined the value at the root

Game Trees

Positive root value suggests that MAX is in better position

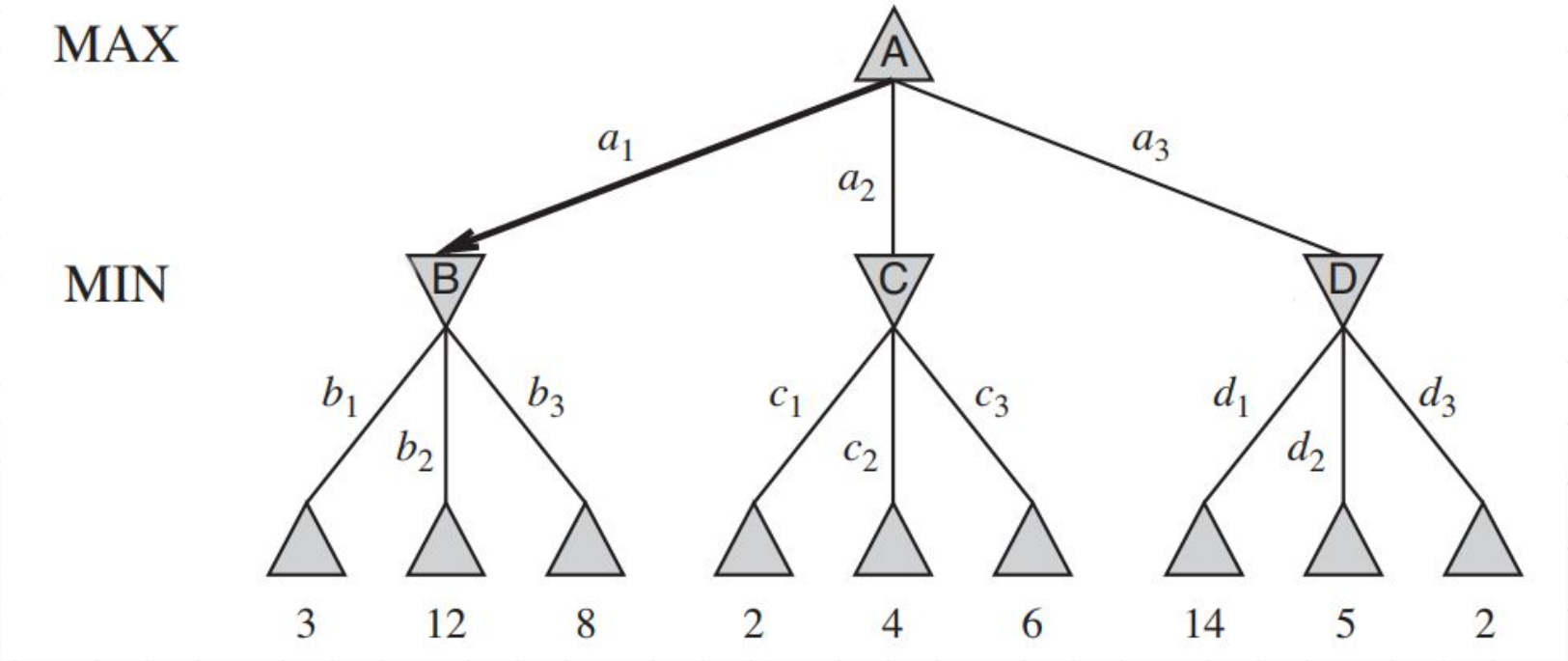
MIN chooses 1 as 1 < 4

MAX chooses 4 as 4 > -5



Static evaluator values

Game Trees



Minimax Algorithm

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

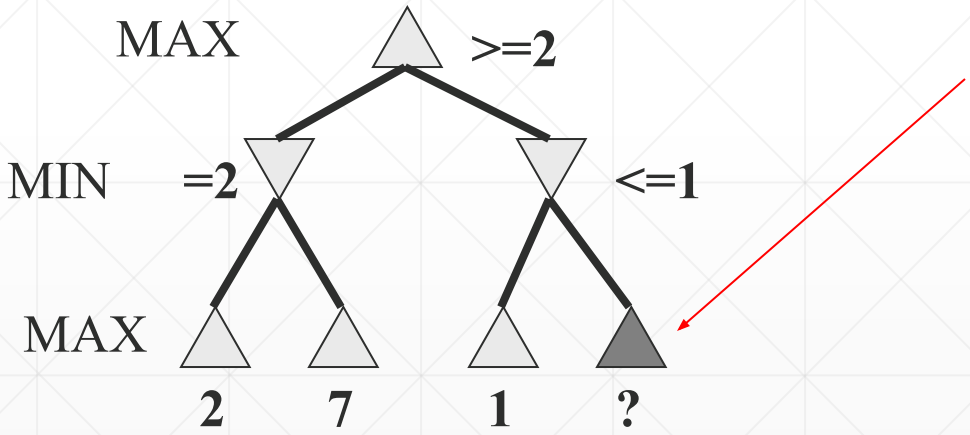
- Complete depth-first exploration
- Let depth = m, average moves available = b (i.e. branching factor)
- Time complexity: $O(b^m)$, space complexity: O(bm)
- $O(10^{40})$ in chess – making minimax impractical
- Solution: **Alpha-beta pruning**

Minimax Algorithm

```
function minimax(position, depth, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, ____)
            maxEval = max(maxEval, eval)
        return maxEval

    else
        minEval = +infinity
        for each child of position
            eval = minimax(child, depth - 1, ____)
            minEval = min(minEval, eval)
        return minEval


// initial call
minimax(currentPosition, 3, true)
```

Alpha-beta Pruning

- Basic idea: “If you have an idea that is surely bad, don't take the time to see how truly awful it is.” -- Pat Winston
- In other words: We DO NOT need to check ALL moves

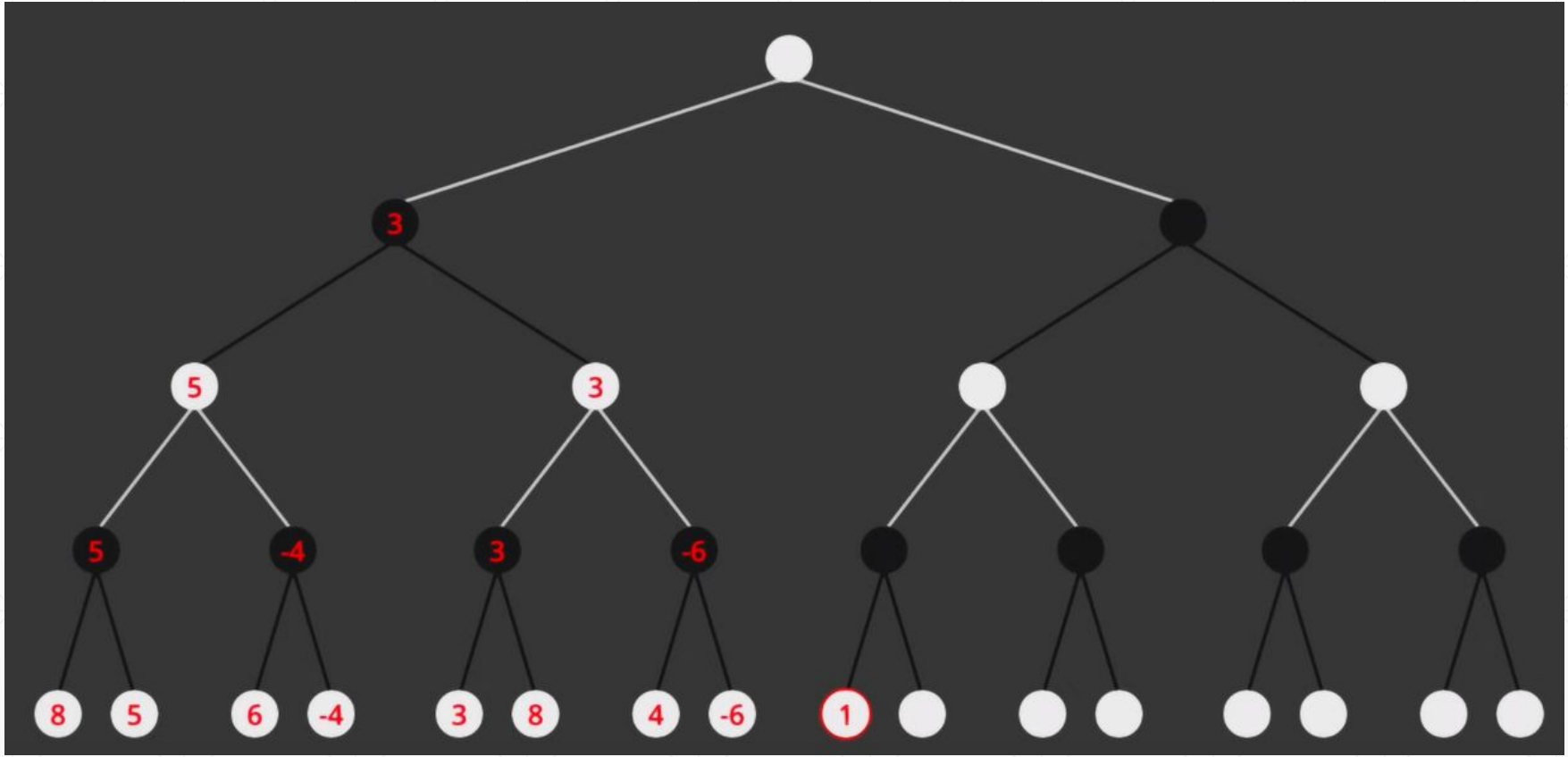


- We don’t need to compute the value at this node.
- No matter what it is, it can’t affect the value of the root node.

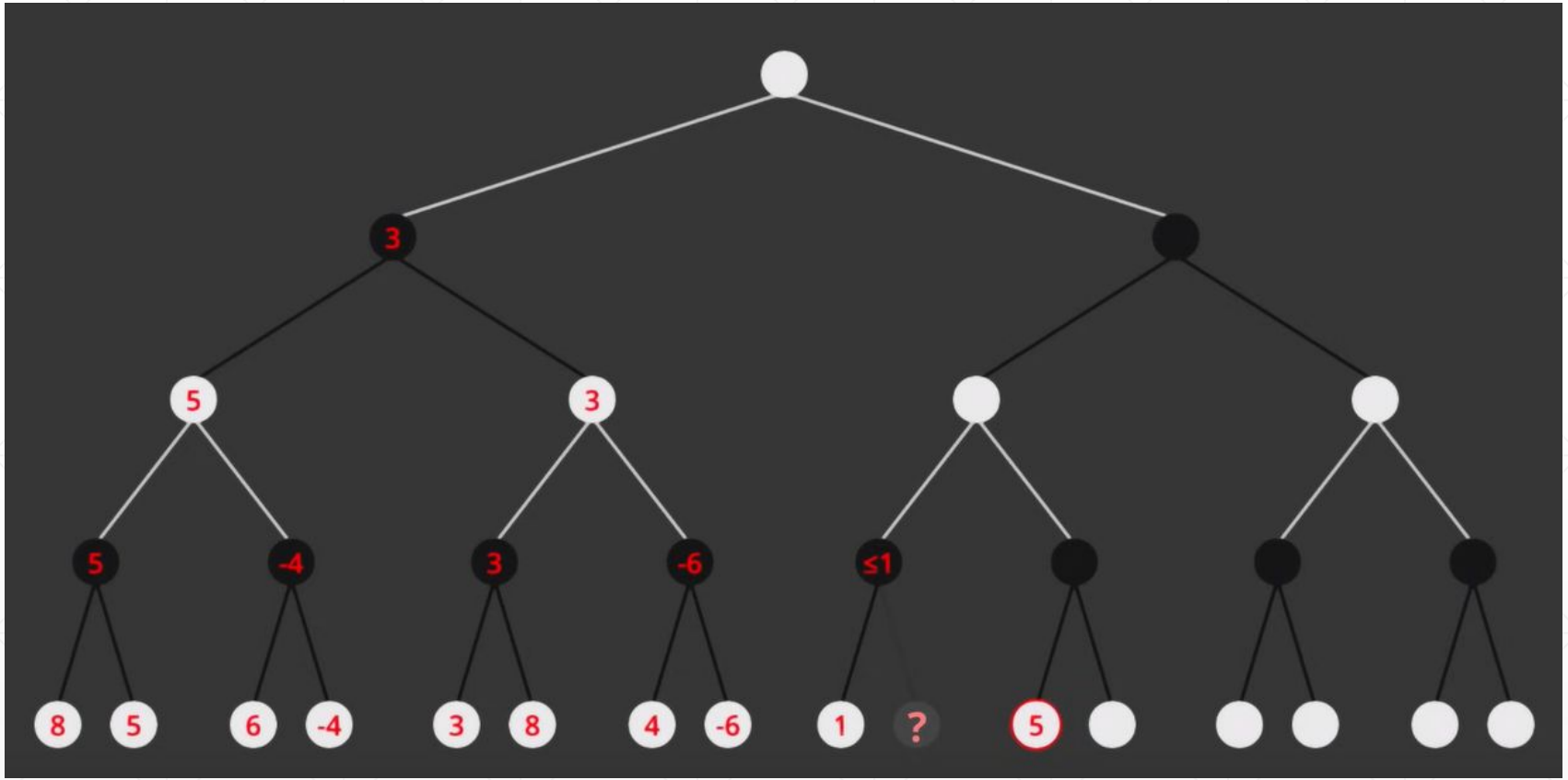
Alpha-beta Pruning

- α (alpha) = value of best choice so far for **MAX** (highest value)
- β (beta) = value of best choice so far for **MIN** (lowest value)
- Each node keeps track of its [α, β] values

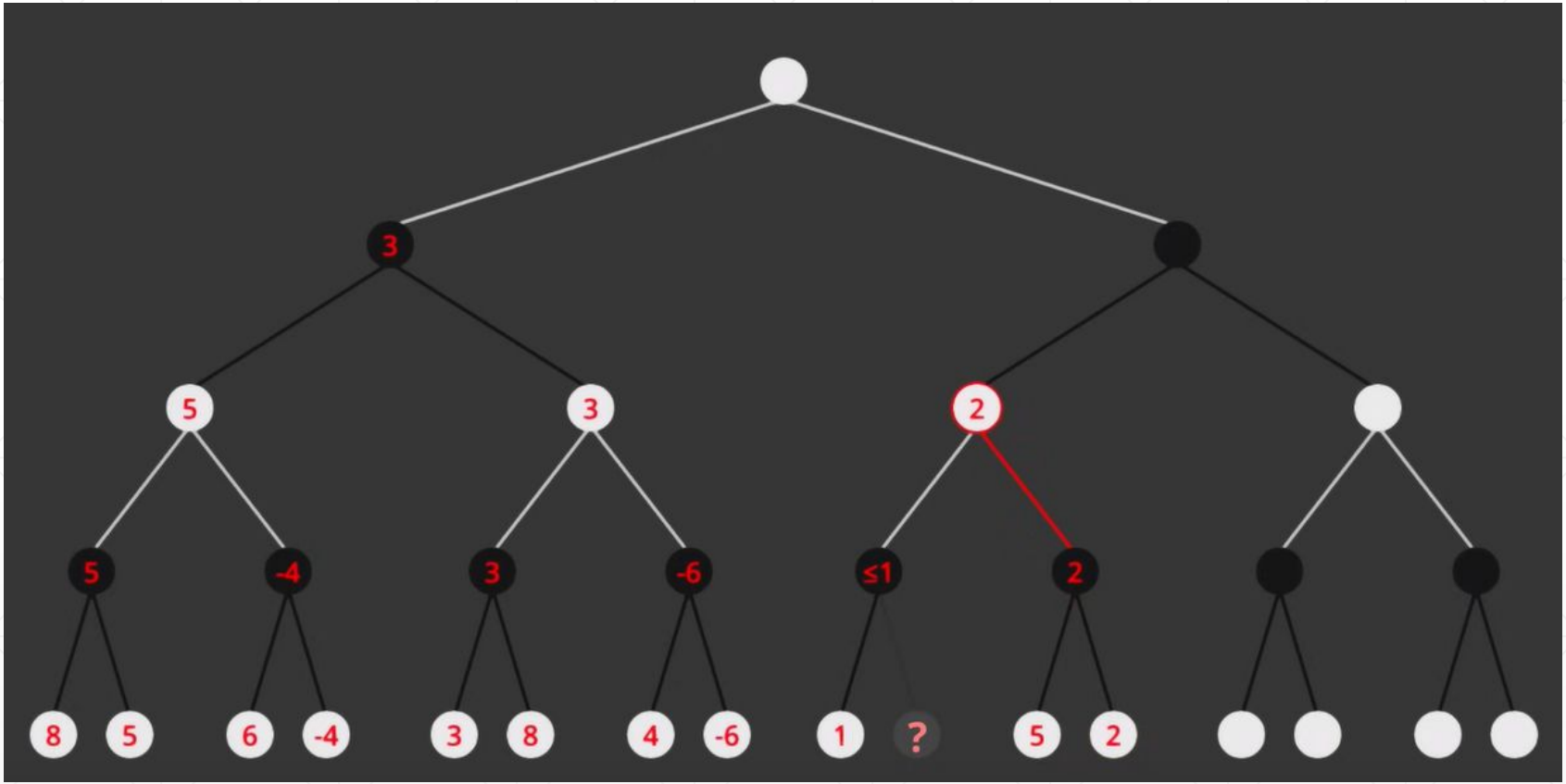
Alpha-beta Pruning



Alpha-beta Pruning



Alpha-beta Pruning

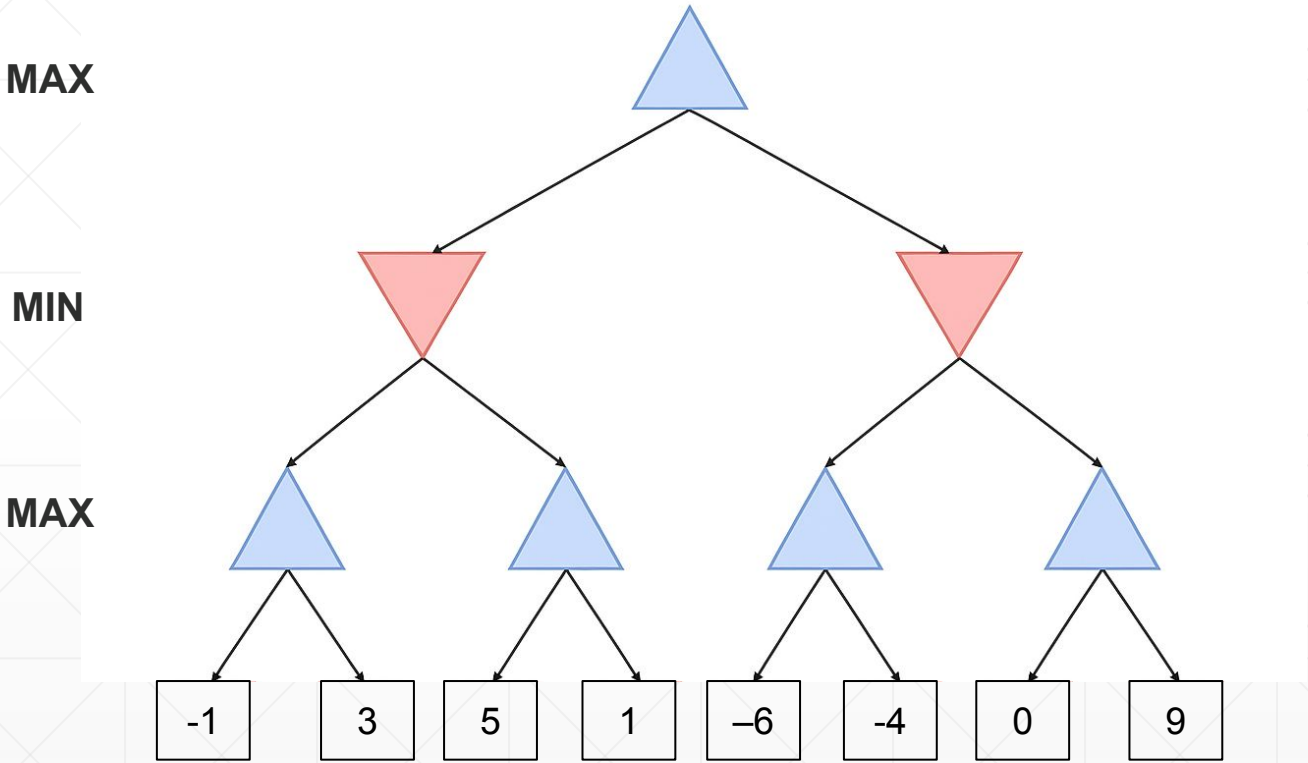


Alpha-beta Algorithm

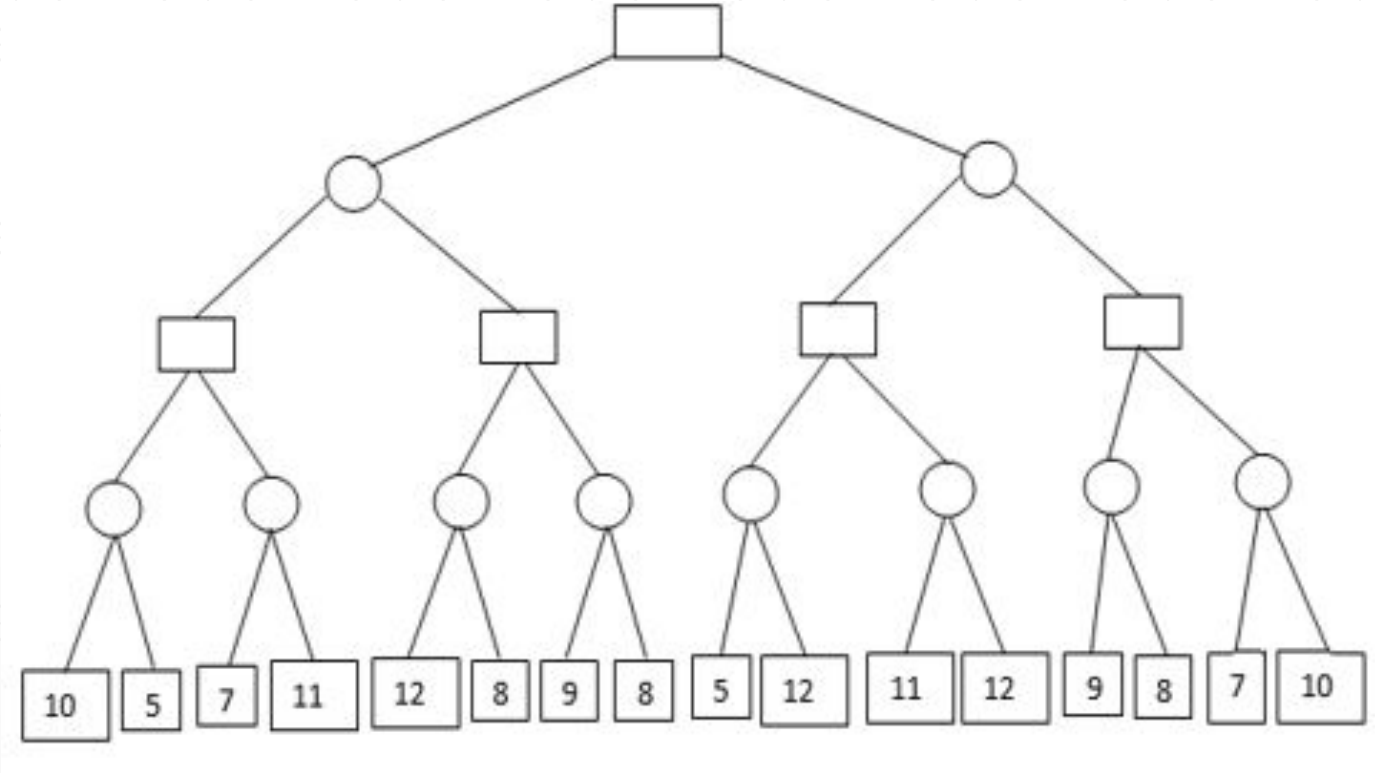
```
function minimax(position, depth, alpha, beta, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, alpha, beta, false)
            maxEval = max(maxEval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha
                break
        return maxEval

    else
        minEval = +infinity
        for each child of position
            eval = minimax(child, depth - 1, alpha, beta, true)
            minEval = min(minEval, eval)
            beta = min(beta, eval)
            if beta <= alpha
                break
        return minEval
```

```
// initial call
minimax(currentPosition,3,-∞, +∞,true)
```

Alpha-beta Pruning Simulation - 1



Alpha-beta Pruning Simulation - 2



Alpha-beta Pruning Simulation - 3