Adversarial search

EMI| Semestre 1 | Pr Mohamed RHAZZAF

A brief history

- **Checkers:**
  - 1950: First computer player
  - 1994: First computer world champion: Chinook defeats Tinsley

- **Chess:**
  - 1945-1960: Zuse, Wiener, Shannon, Turing, Newell & Simon, McCarthy
  - 1997: Deep Blue defeats human champion Gary Kasparov

- **Go:**
  - 1968: Zobrist’s program plays legal Go, barely (b>300!)
  - 1968-2005: various ad hoc approaches tried, novice level
  - 2005-2014: Monte Carlo tree search -> strong amateur
  - 2016-2017: AlphaGo defeats human world champions

Types of Games

- **Game = task environment with > 1 agent**

- **Axes:**
  - Deterministic or stochastic?
  - Perfect information (fully observable)?
  - One, two, or more players?
  - Turn-taking or simultaneous?
  - Zero sum?

- **Want algorithms for calculating a contingent plan (a.k.a. strategy or policy) which recommends a move for every possible eventuality**

"Standard" Games

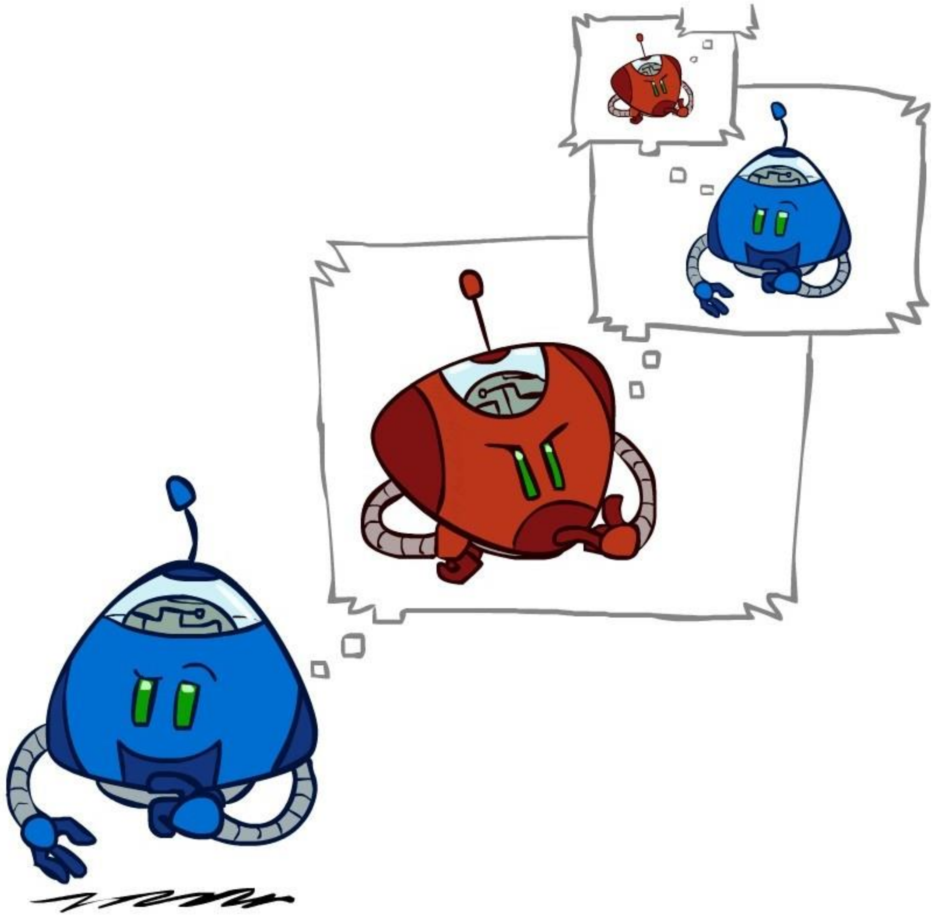
- **Standard games are deterministic, observable, two-player, turn-taking, zero-sum**
- **Game formulation:**
  - Initial state: s_0
  - Players: Player(s) indicates whose move it is
  - Actions: Actions(s) for player on move
  - Transition model: Result(s,a)
  - Terminal test: Terminal-Test(s)
  - Terminal values: Utility(s,p) for player p
    - Or just Utility(s) for player making the decision at root

Zero-Sum Games

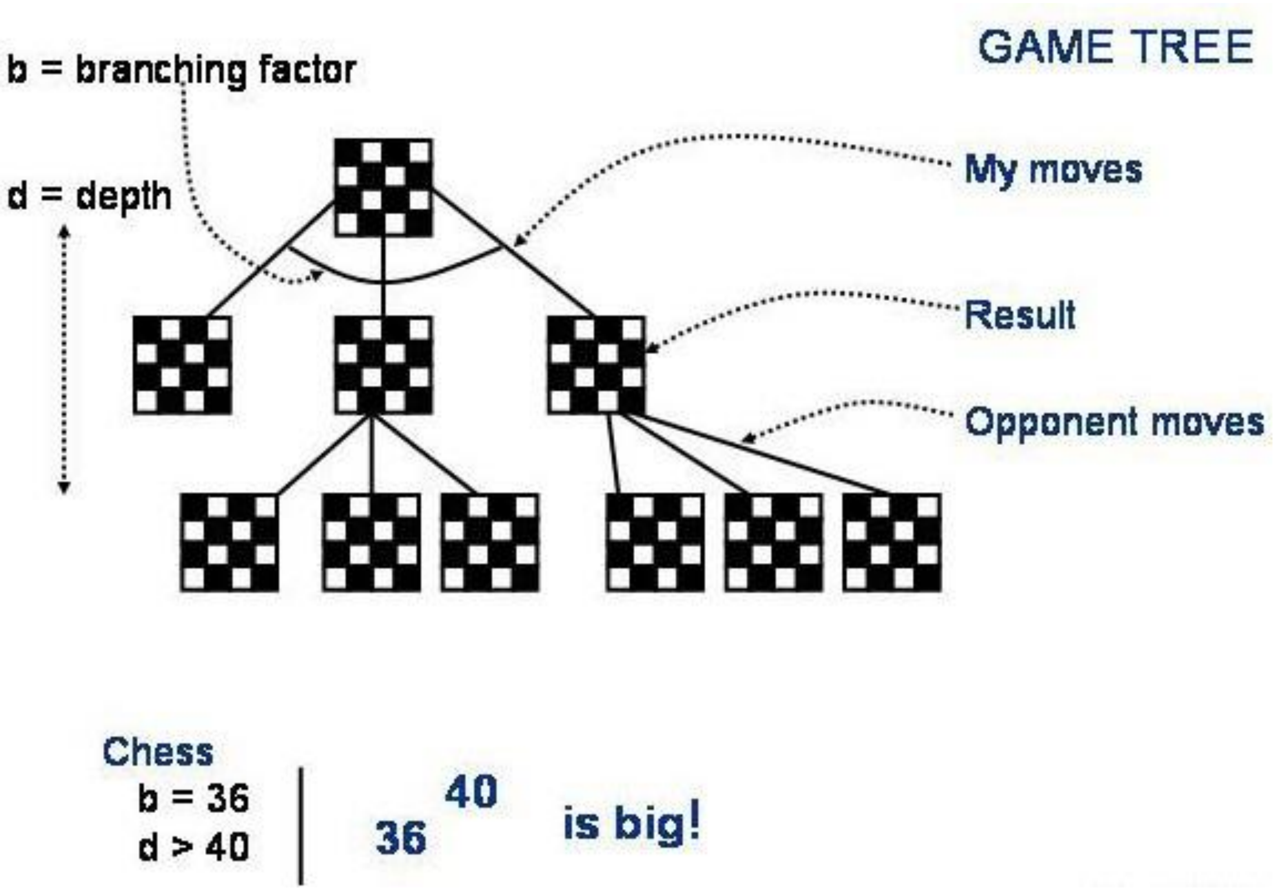
- **Zero-Sum Games**
  - Agents have opposite utilities
  - Pure competition:
    - One ***maximizes***, the other ***minimizes***

- **General Games**
  - Agents have independent utilities
  - Cooperation, indifference, competition, shifting alliances, and more are all possible

Adversarial Search



Two players games



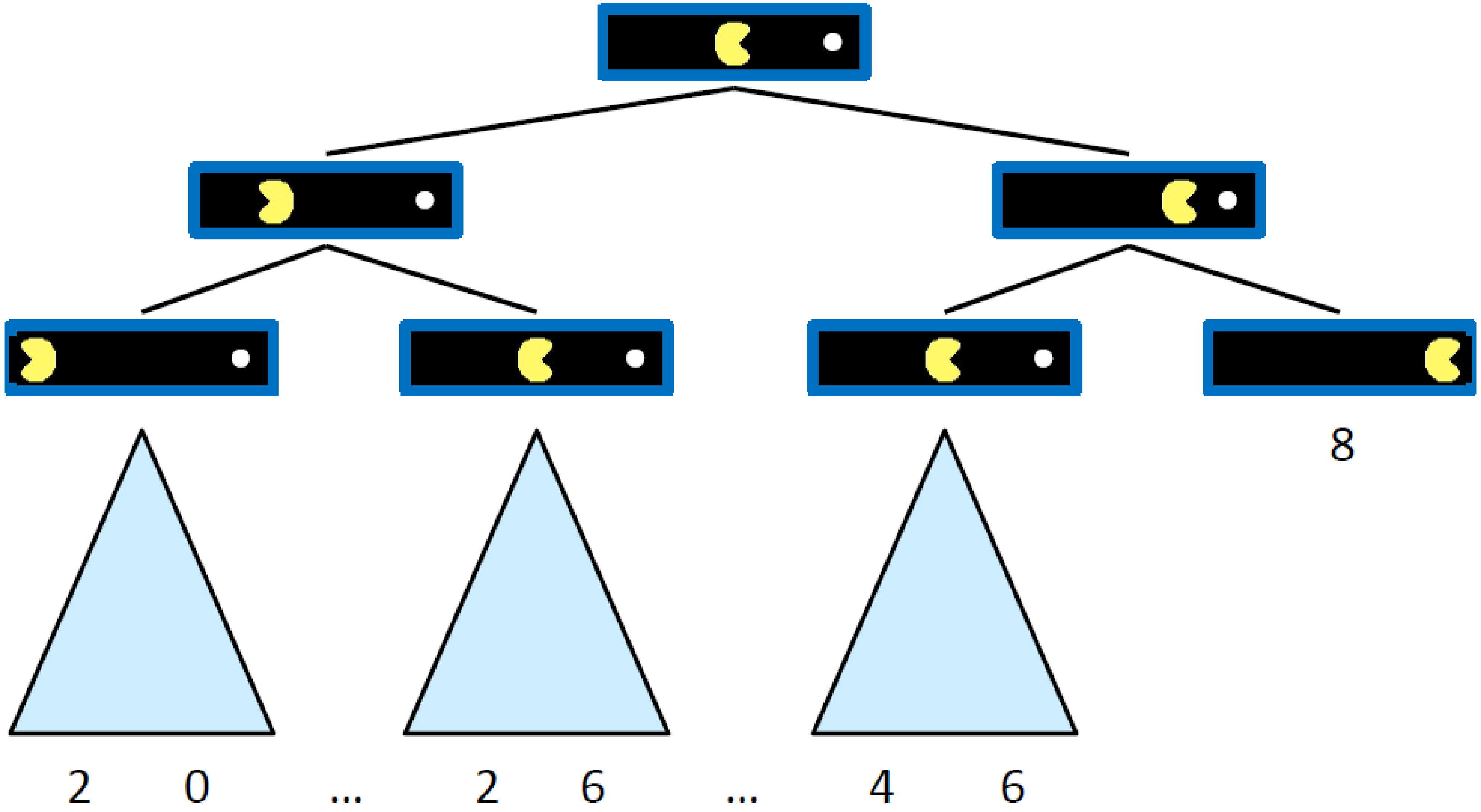
- Some board position represents the initial state

- We generate the children of this position by making all of the legal moves available to us

- Then, we consider the moves that our opponent can make to generate the descendants of each of these positions, etc.

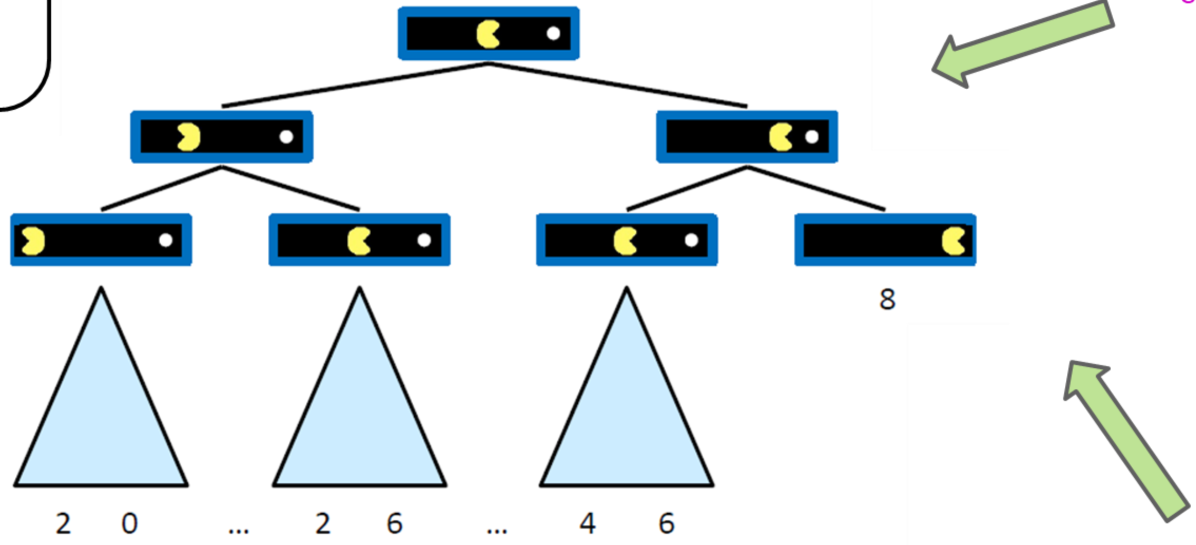
- Note that these trees are enormous and cannot be explicitly represented in their entirly for any complex game.

Single-Agent Trees



Value of a State

Value of a state:
The best achievable outcome (utility) from that state



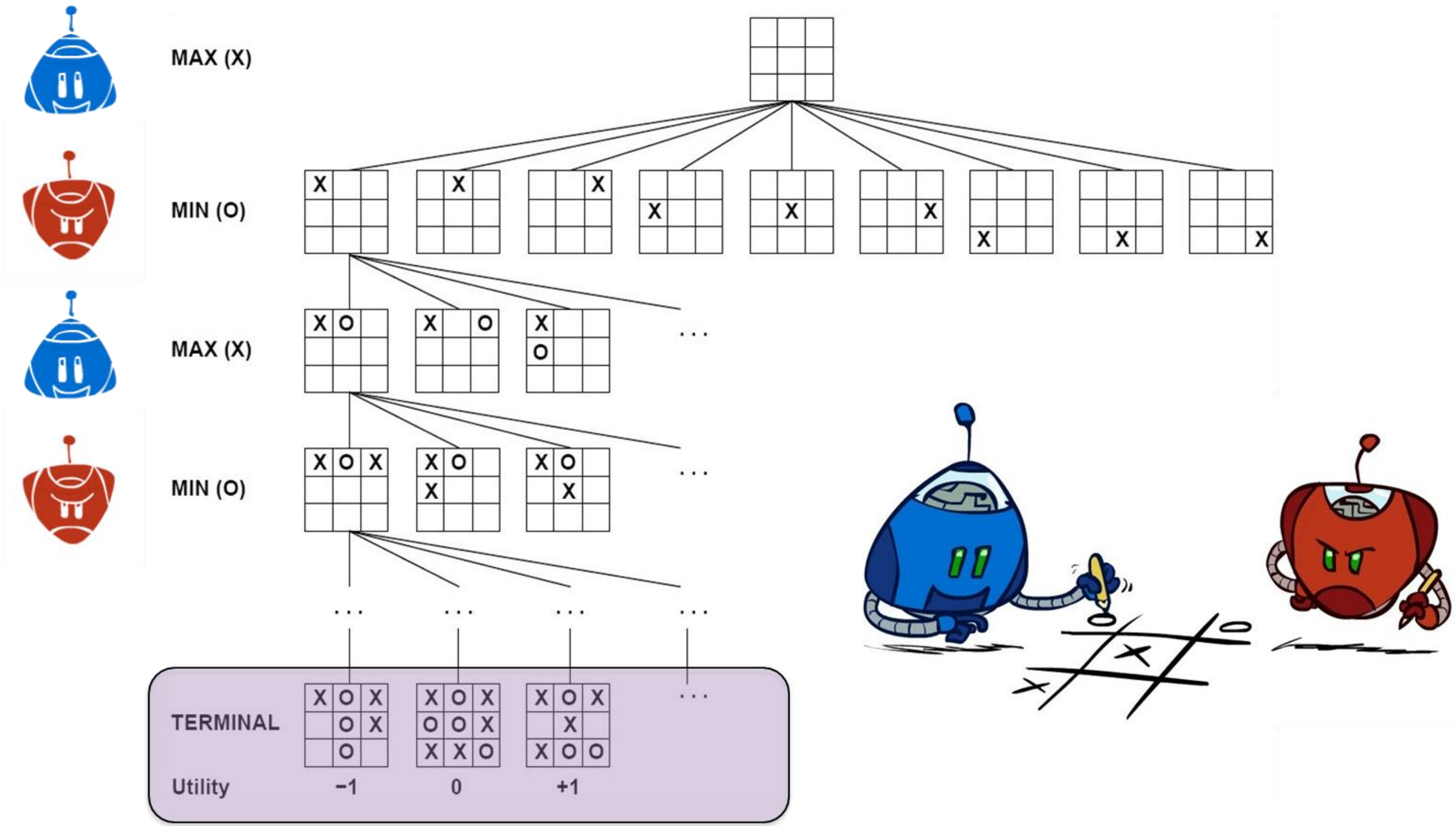
Non-Terminal States:

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

Terminal States:

$V(s)$ = known

Tic-Tac-Toe Game Tree



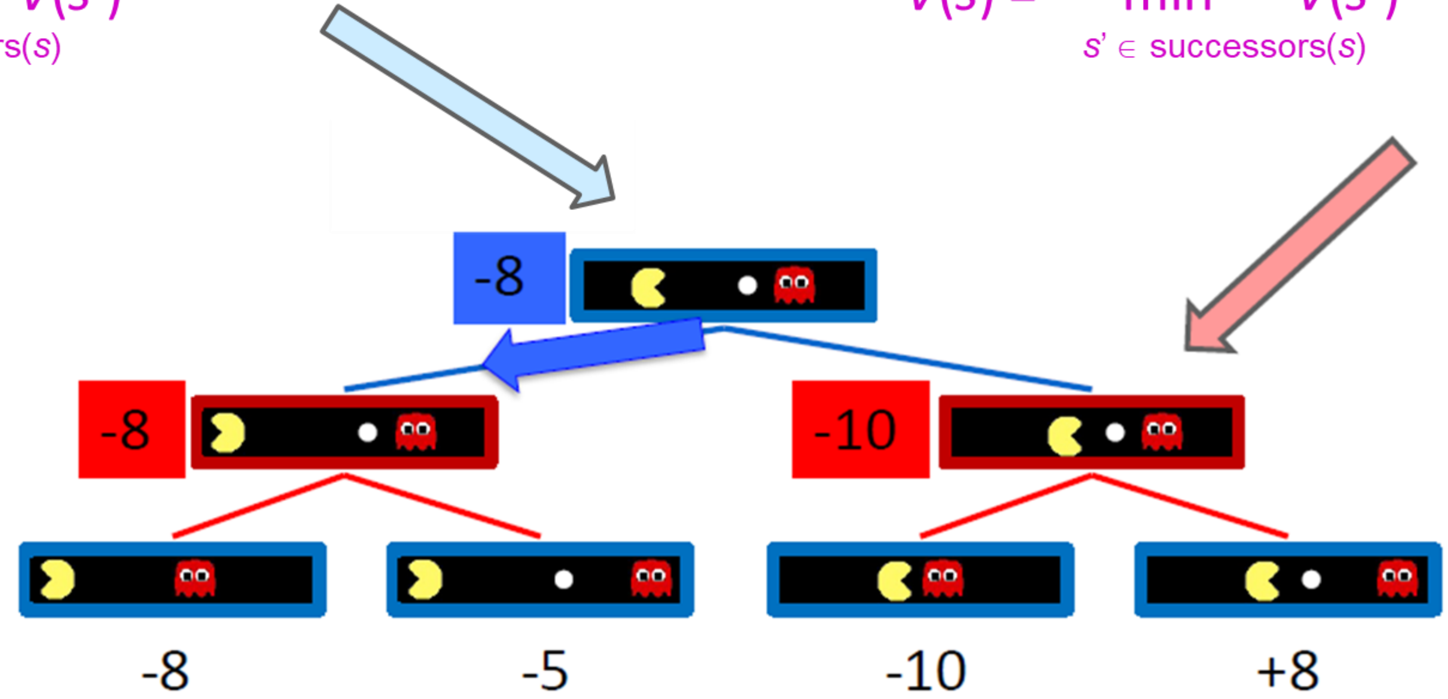
Minimax Values

MAX nodes: under Agent's control

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

MIN nodes: under Opponent's control

$$V(s) = \min_{s' \in \text{successors}(s)} V(s')$$



Terminal States:

$V(s)$ = known

Minimax Algorithm

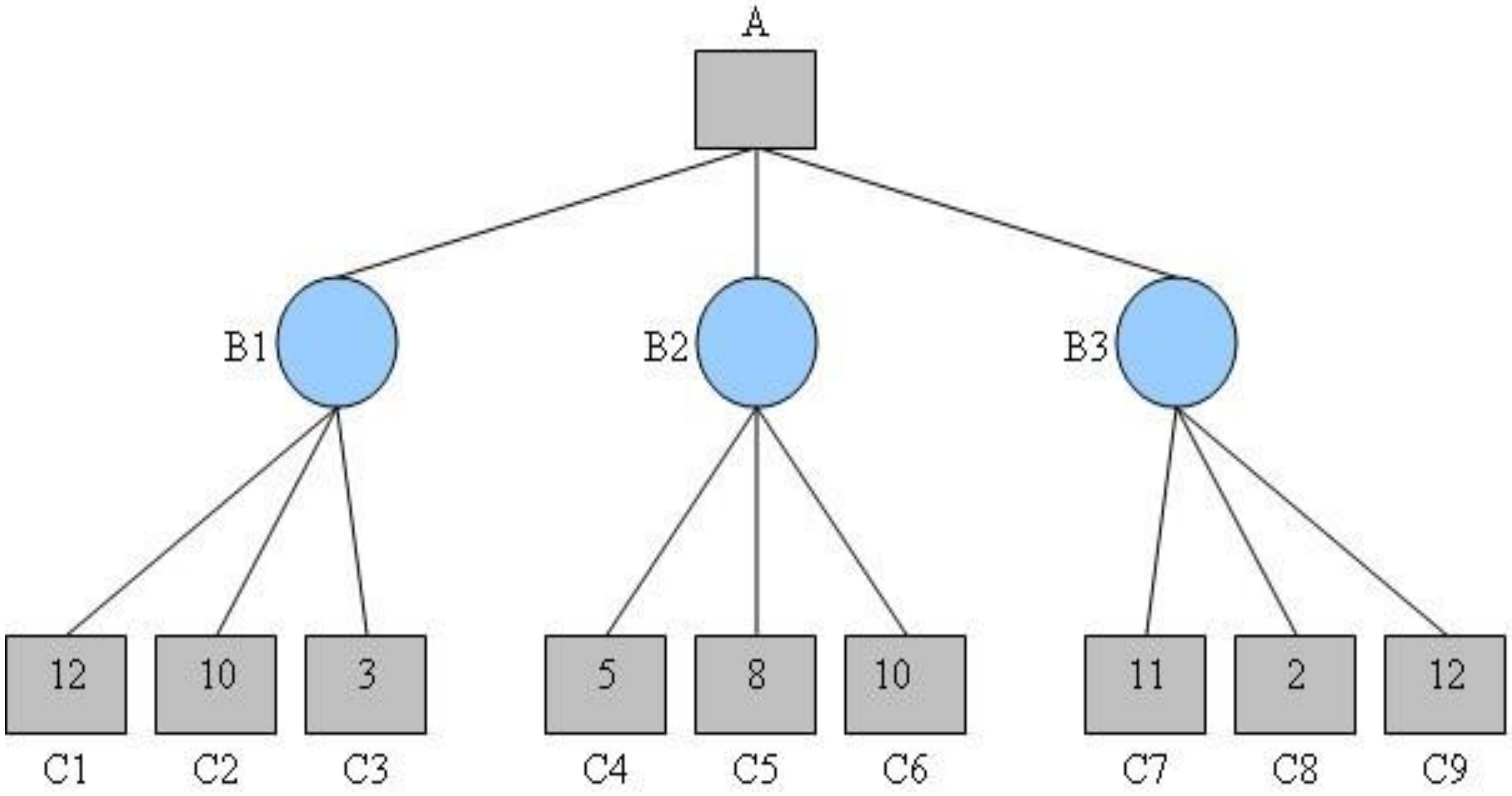
- **Problem setting**
  - Two players:
    - **MAX**: tries to maximize the utility
    - **MIN**: tries to minimize the utility (because in zero-sum, MIN’s gain = MAX’s loss)
  - Both players are assumed to play **optimally**.
  - A game is represented by a **game tree**:
    - Nodes = states
    - Edges = actions
    - Leaves = terminal states with utilities for MAX

Minimax Algorithm

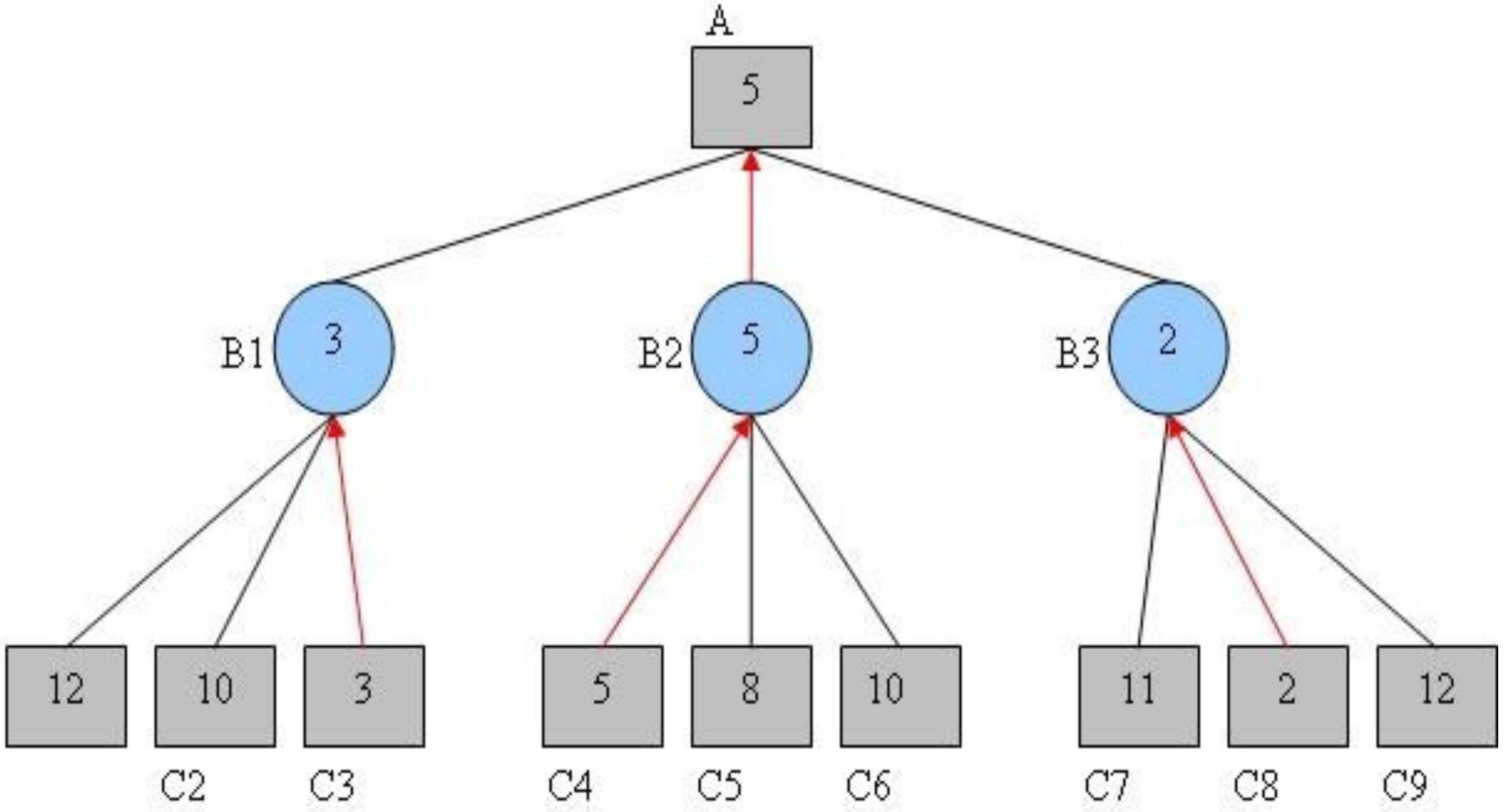
- **Key Idea of Minimax**

  - At each state:
    - **MAX node** chooses the action with the **maximum** value
    - **MIN node** chooses the action with the **minimum** value
  - Terminal nodes have fixed utility values (e.g., win = +1, loss = −1, draw = 0).
  - Values propagate **up the tree**.

Minimax Algorithm



Minimax Algorithm



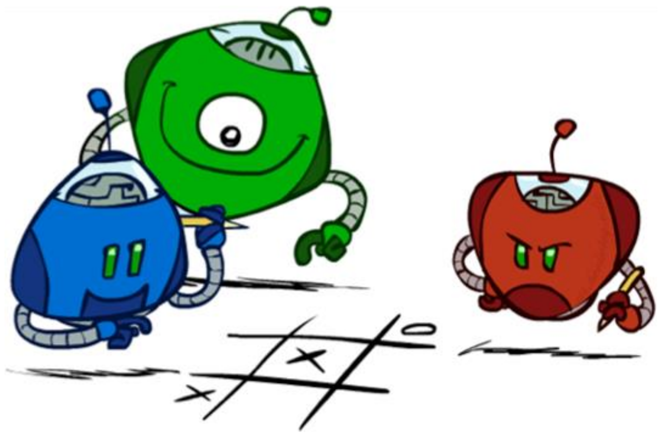
Minimax Algorithm

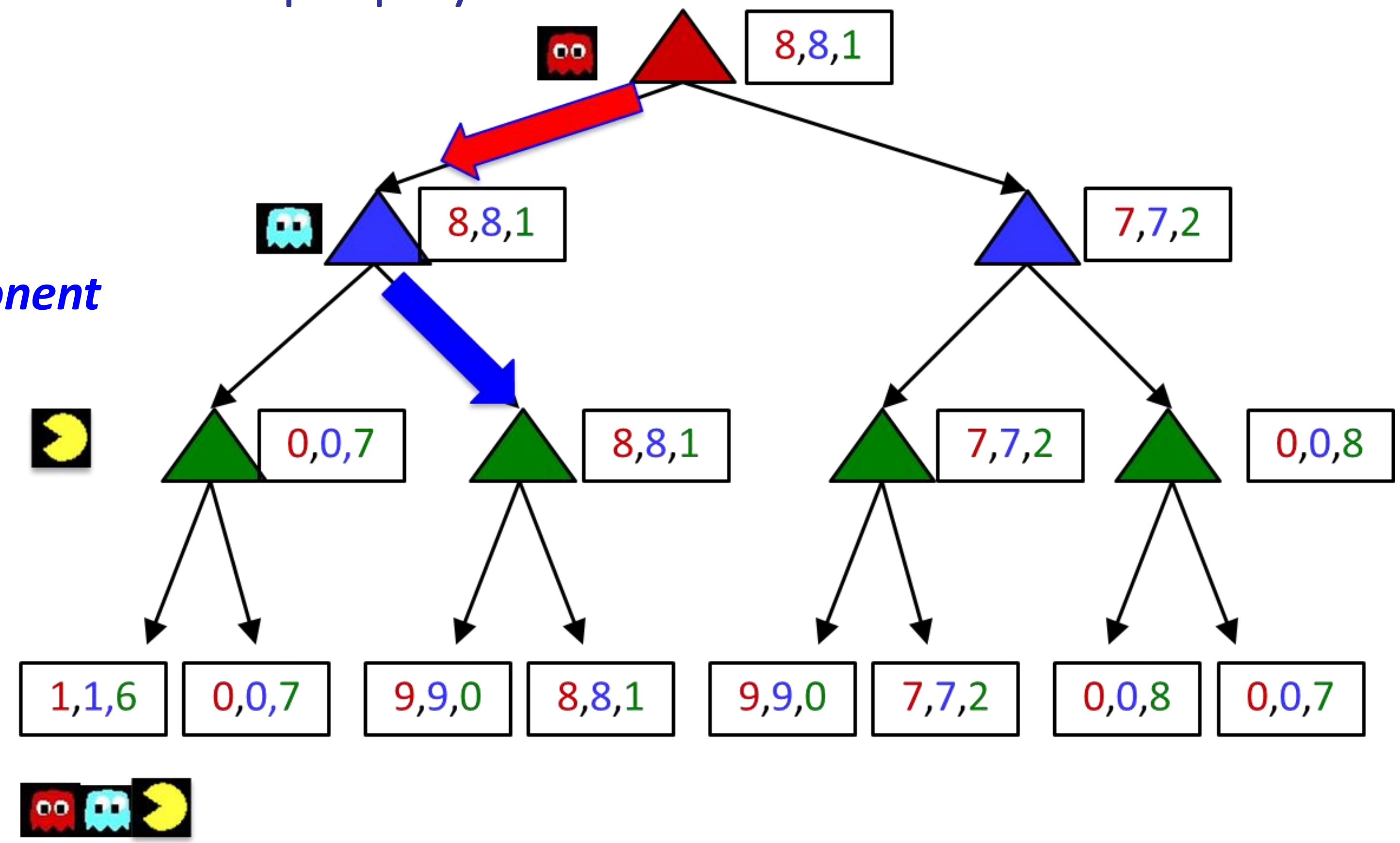
```python
def minimax_decision(state):
    """
    Returns the optimal action for MAX in the given state.
    """
    best_value = float("-inf")
    best_action = None

    for action in actions(state):
        value = minimax_value(result(state, action))
        if value > best_value:
            best_value = value
            best_action = action

    return best_action
```

```python
def minimax_value(state):
    """
    Returns the minimax value of a given state.
    """

    # If the game is over, return its utility
    if terminal_test(state):
        return utility(state)

    # If it's MAX's turn
    if player(state) == "MAX":
        value = float("-inf")
        for action in actions(state):
            value = max(value, minimax_value(result(state,
 action)))
        return value

    # If it's MIN's turn
    else:  # player(state) == "MIN"
        value = float("inf")
        for action in actions(state):
            value = min(value, minimax_value(result(state,
 action)))
        return value
```

Generalized Minimax

- What if the game is not zero-sum, or has multiple players?
- Generalization of minimax:
  - Terminals have ***utility tuples***
  - Node values are also utility tuples
  - ***Each player maximizes its own component***
  - Can give rise to cooperation and competition dynamically…

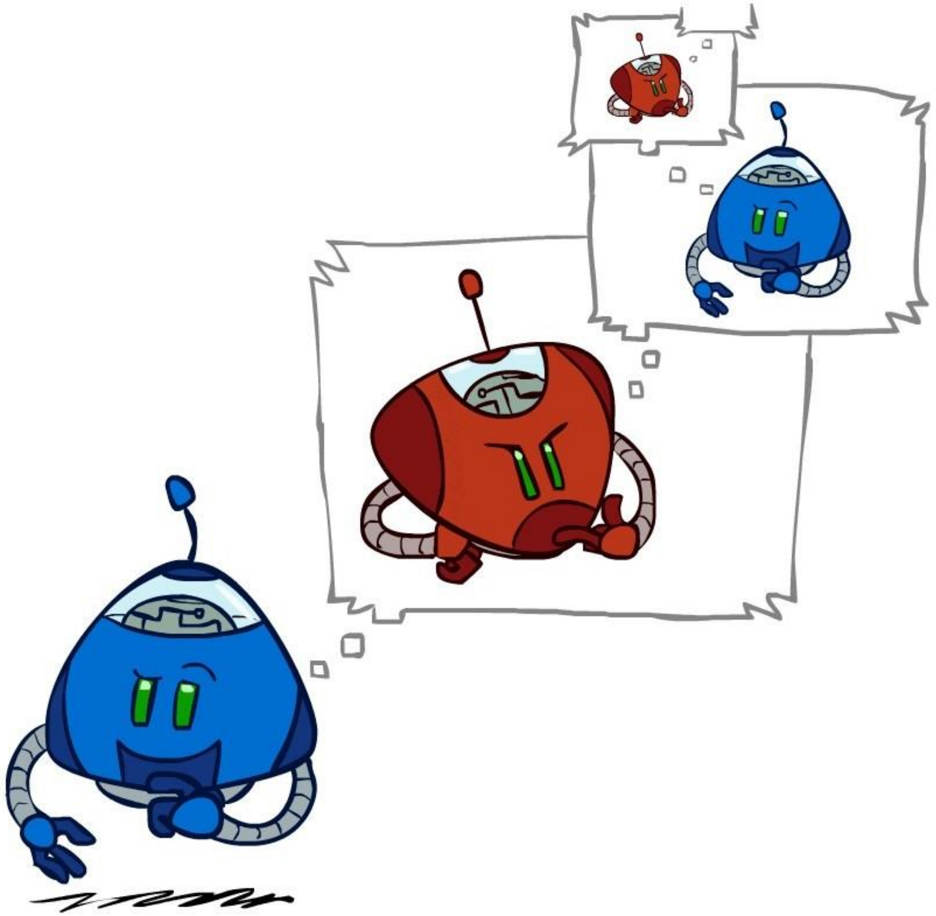




Minimax Efficiency

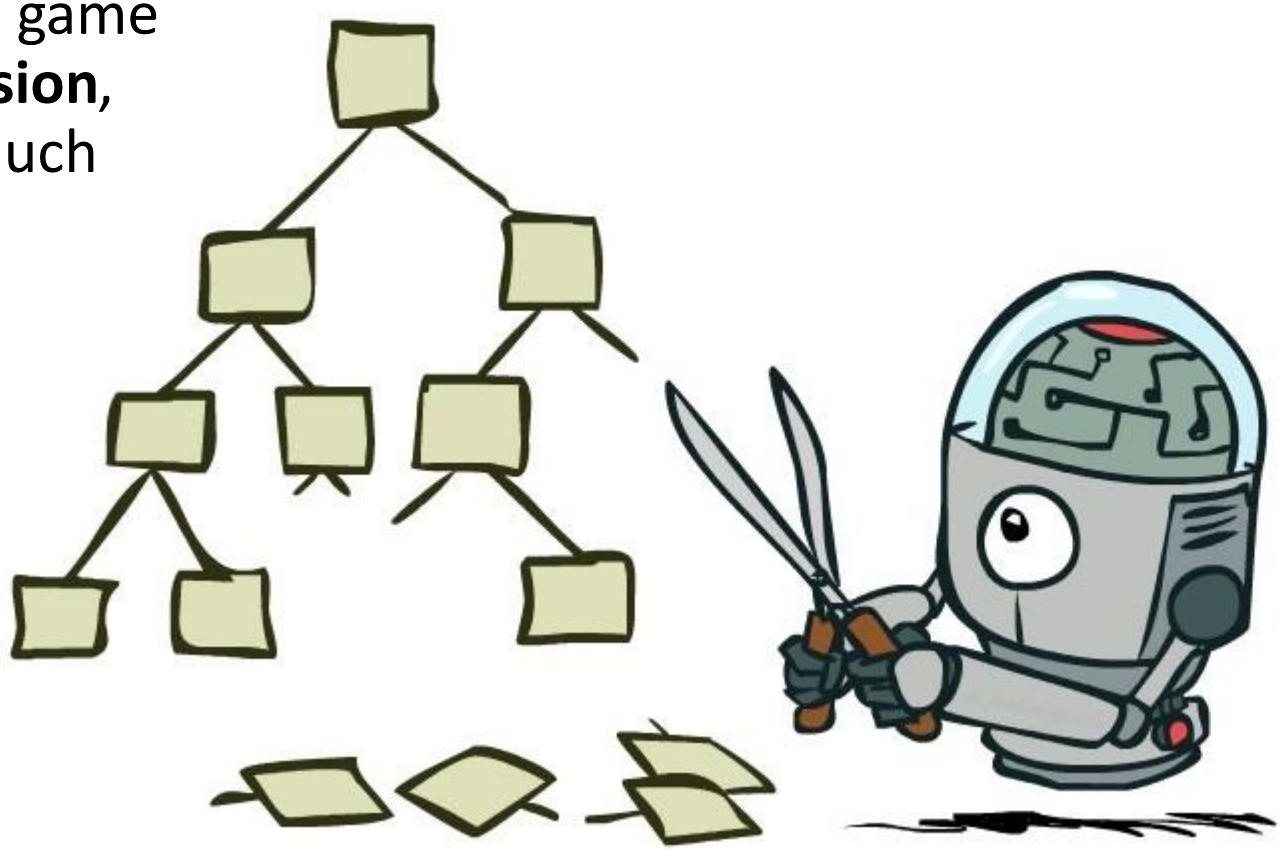
- **How efficient is minimax?**
  - Just like (exhaustive) DFS
  - Time: $O(b^m)$
  - Space: $O(bm)$

- **Example: For chess, b =35, m = 100**
  - Exact solution is completely infeasible
  - Humans can’t do this either, so how do we play chess?

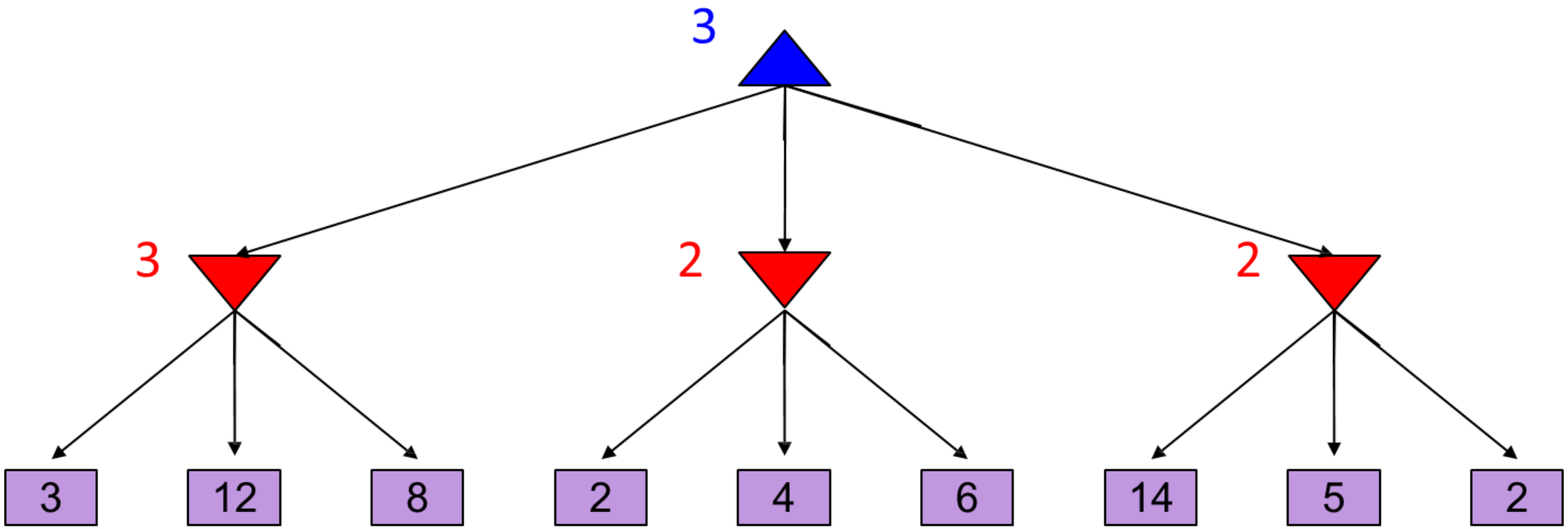


Game Tree Pruning

- **Pruning** refers to eliminating parts of the game tree that **cannot influence the final decision**, allowing the minimax algorithm to run much faster without changing the result

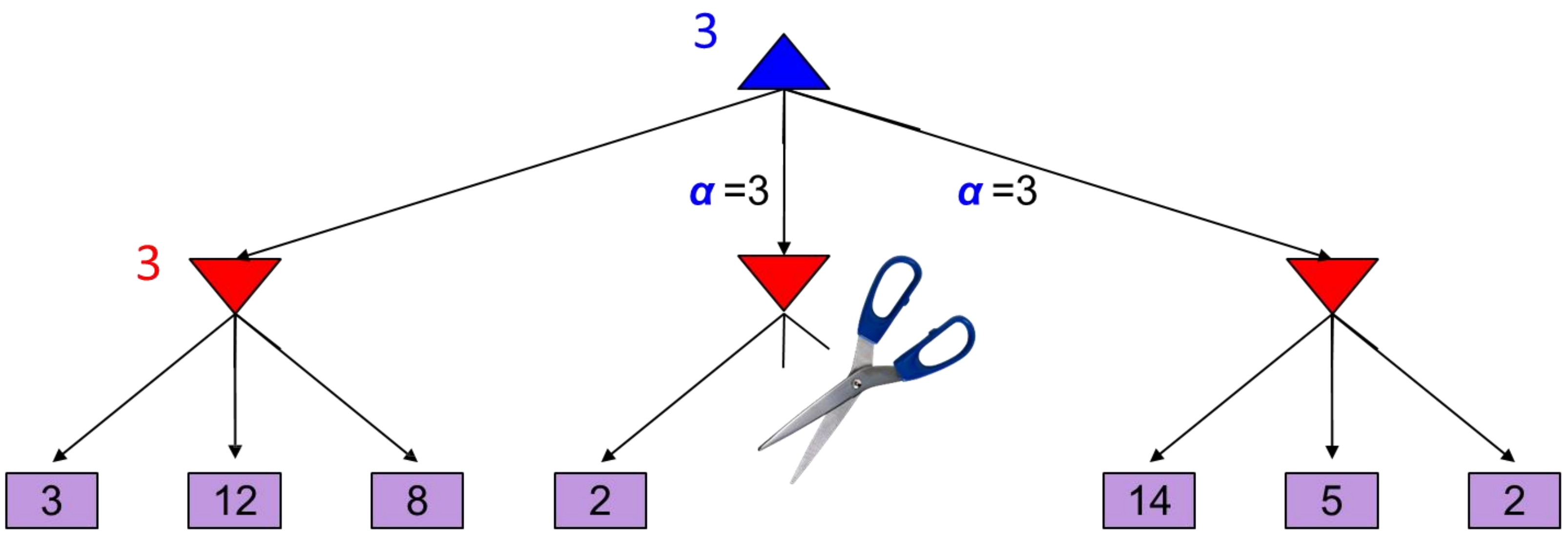


Minimax Example



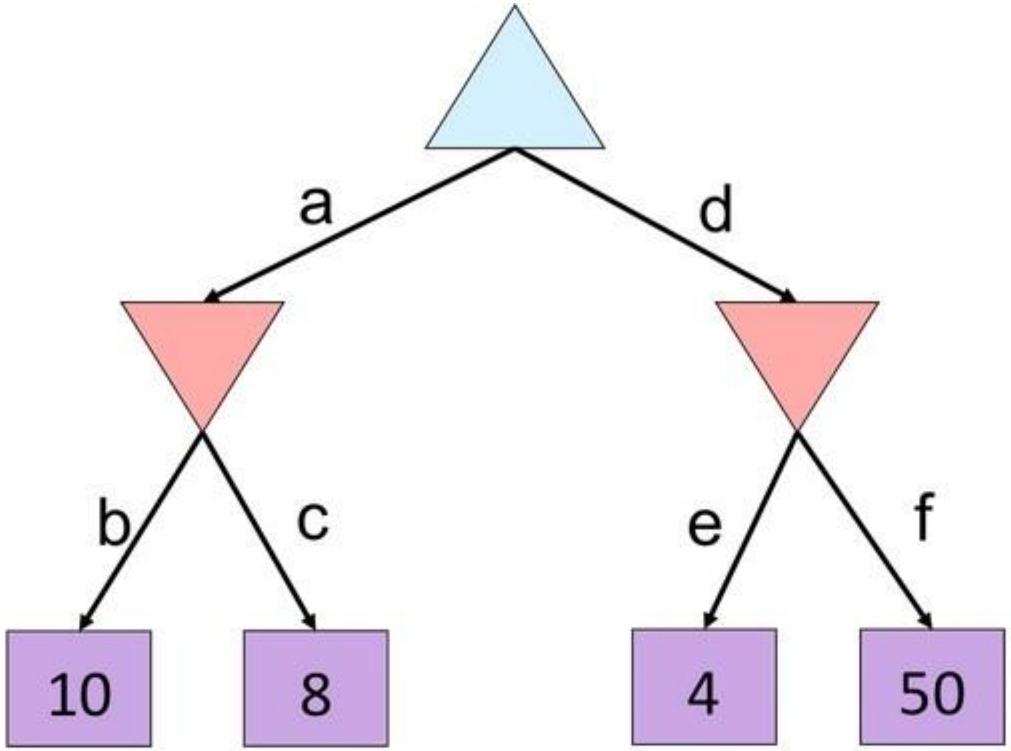
Alpha-Beta Example

α = best option so far from any MAX node on this path

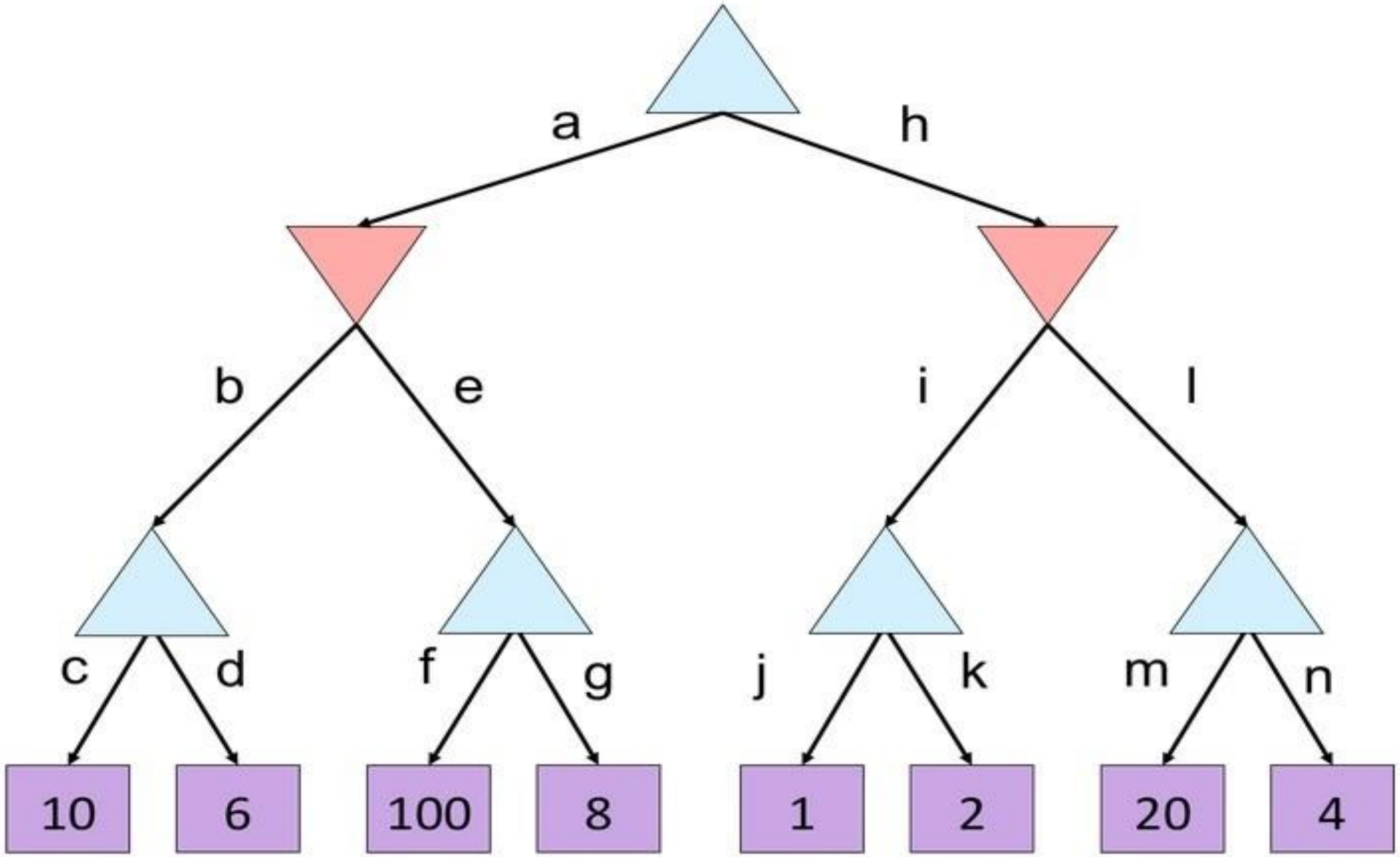


The order of generation matters: more pruning is possible if good moves come first

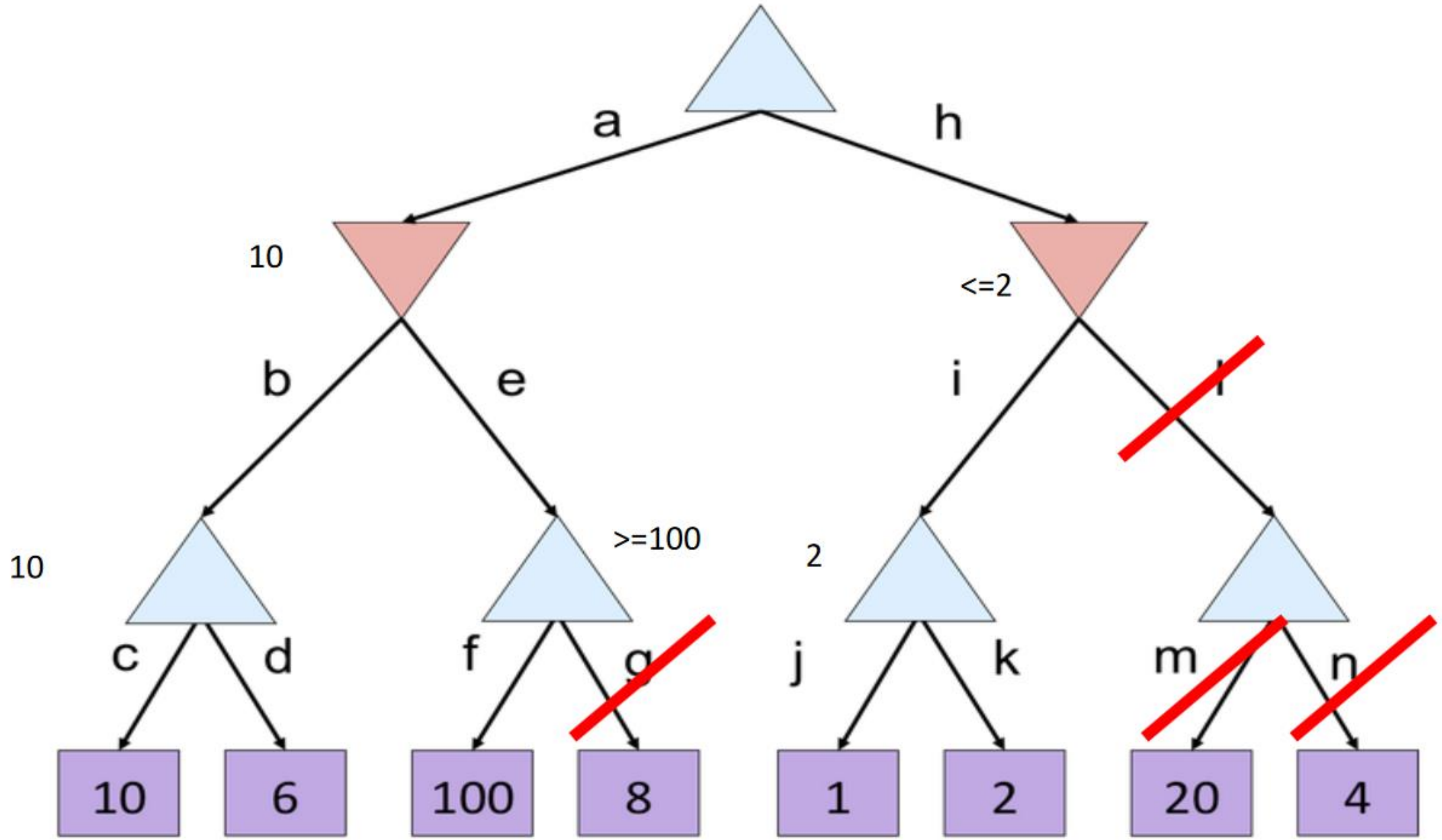
Alpha-Beta Quiz



Alpha-Beta Quiz 2



Alpha-Beta Quiz 2



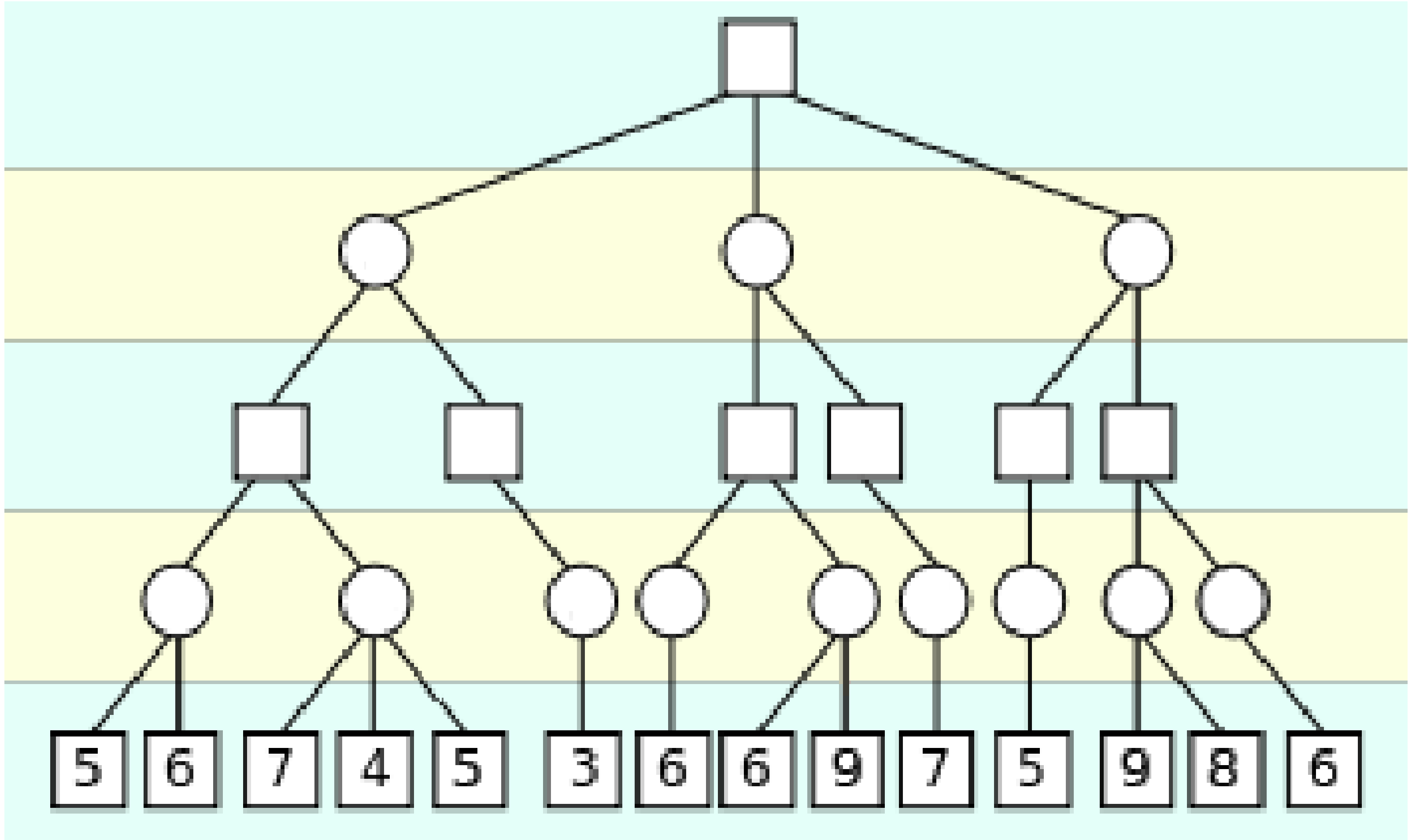
Alpha-Beta pruning algorithm

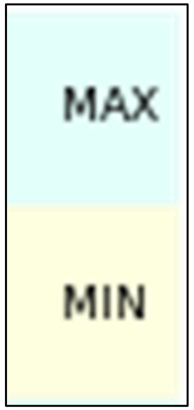
```python
def minimax_decision(state):
    """
    Returns the optimal action for MAX, using alpha–beta pruning.
    """
    best_value = float("-inf")
    best_action = None
    alpha = float("-inf")
    beta = float("inf")

    for action in actions(state):
        value = minimax_value(result(state, action), alpha, beta)
        if value > best_value:
            best_value = value
            best_action = action

        alpha = max(alpha, best_value)  # update alpha

    return best_action
```

Alpha-Beta pruning algorithm

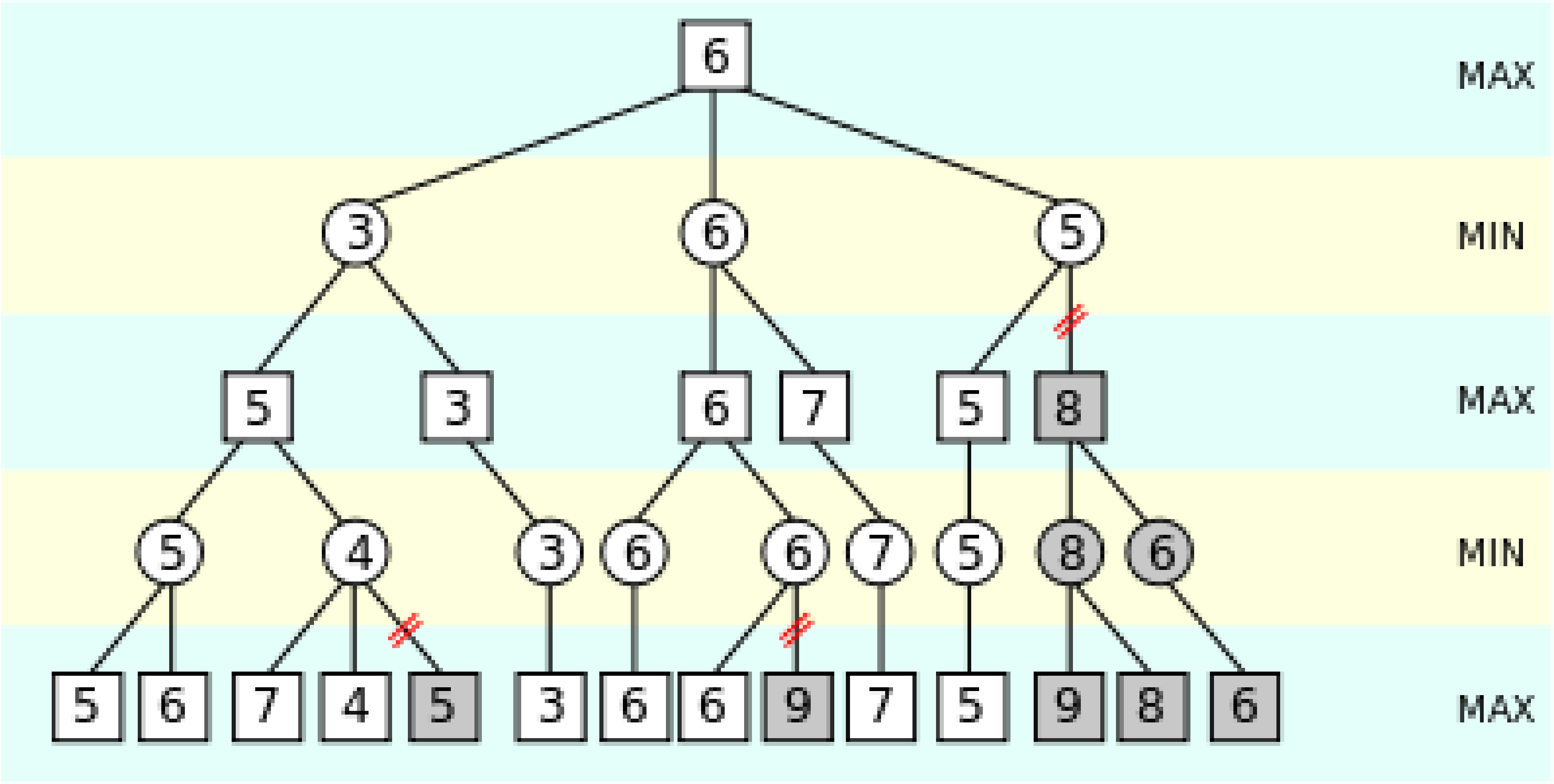
```
def minimax_value(state, alpha, beta):
    """
    Returns the minimax value of a state using alpha–beta
  pruning.
    """

    # Terminal state → return utility
    if terminal_test(state):
        return utility(state)

    # MAX player
    if player(state) == "MAX":
        value = float("-inf")
        for action in actions(state):
            value = max(value, minimax_value(result(state,
  action), alpha, beta))
            alpha = max(alpha, value)

            # PRUNING
            if value >= beta:
                return value    # prune branch

        return value

    # MIN player
    else:  # player(state) == "MIN"
        value = float("inf")
        for action in actions(state):
            value = min(value, minimax_value(result(state, action),
    alpha, beta))
            beta = min(beta, value)

            # PRUNING
            if value <= alpha:
                return value    # prune branch

        return value
```

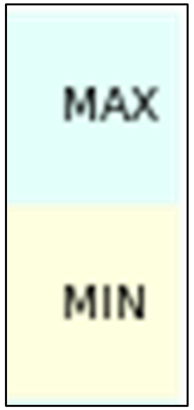
Alpha-Beta Pruning





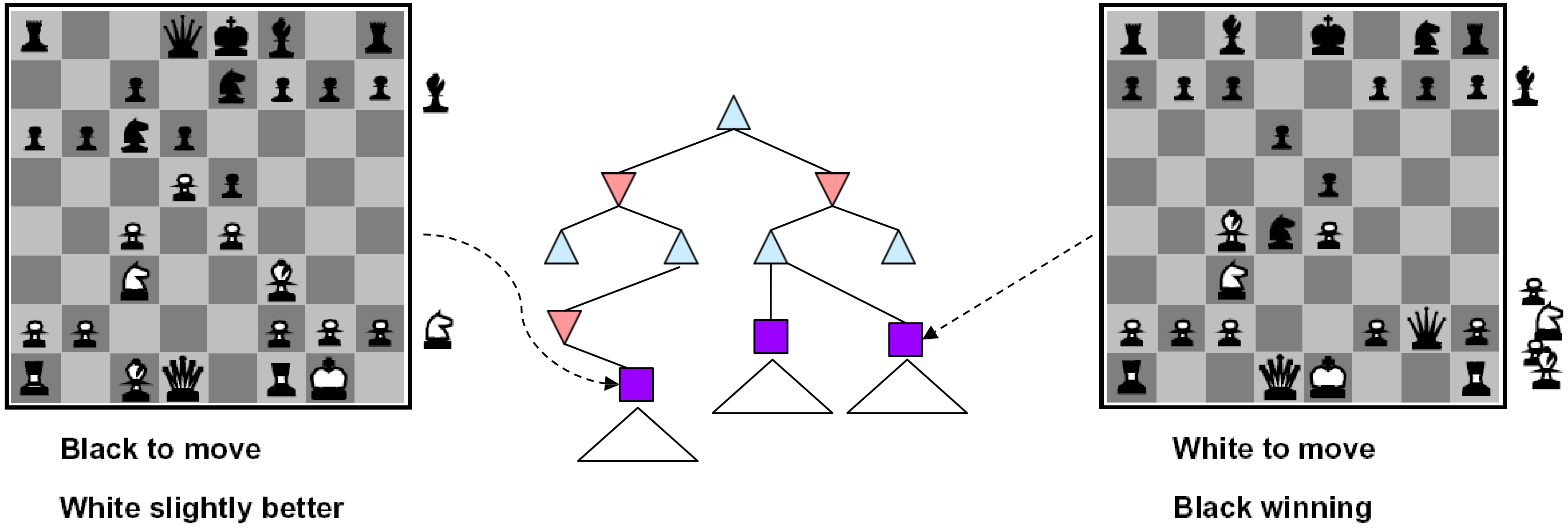
Alpha-Beta Pruning





Evaluation Functions

- Evaluation functions score non-terminals in depth-limited search



- Same idea as heuristics or using mathematical formula

Monte Carlo Tree Search

- Methods based on alpha-beta search assume a fixed horizon
  - Pretty hopeless for Go, with b > 300
- MCTS combines two important ideas:
  - ***Evaluation by rollouts*** – play multiple games to termination from a state s (using a simple, fast rollout policy) and count wins and losses
  - ***Selective search*** – explore parts of the tree that will help improve the decision at the root, regardless of depth

MCTS Version 0

- Do *N* rollouts from each child of the root, record fraction of wins
- Pick the move that gives the best outcome by this metric