# Adversarial Search

Curtis Larsen

Utah Tech University—Computing

September 4, 2025

# Today's Sections

# Motivation & Applications

- Why adversarial search and where it shows up.

# Why Adversarial Search?

- Single-agent search assumes a passive environment.
- In many domains, there is an **opponent** with conflicting goals.
- Need strategies that account for adversarial agents.

# Competitive Environments

- Zero-sum: one player's gain is the other's loss.
- Deterministic and perfect information: no hidden cards, no randomness.
- Classic setting for studying rational play.

# Applications of Adversarial Search

- Board games: chess, checkers, tic-tac-toe.
- Real-time games: strategy and video games.
- Economic or market competition models.
- Multi-agent simulations and robotics.

# Connection to AI Agents

- Rational play as a model of intelligent decision-making.
- Foundation for modern game-playing agents (e.g., AlphaGo).
- Links to Monte Carlo methods and LLM-based multi-agent systems.

# Games as Environments

- Deterministic, perfect-information, zero-sum setups.

# Games as AI Environments

- Games can be modeled with the agent–environment loop.
- Players alternate turns, producing sequential states.
- Actions are chosen strategically, anticipating opponent responses.

# Defining Characteristics

- Fully observable (perfect information)
- Deterministic (no chance events in classic board games)
- Sequential (moves matter in order)
- Dynamic (opponent can change the state while you “wait”)
- Discrete (finite states, actions, moves)
- Multi-agent (at least two players with competing goals)

**Classic adversarial games:** fully observable, deterministic, sequential, dynamic, discrete, multi-agent.

# Formal Components of a Game

- **States:** $s$, board positions or configurations.
- **Actions:** $A(s)$, legal moves available to a player.
- **Transition model:** $T(s, a) \to s'$, how actions lead to new states.
- **Players:** usually 2: MAX and MIN.
- **Initial state:** $s_0$, current conditions of the game.
- **Terminal states:** $Terminal(s)$, end conditions of the game.
- **Utility function:** $u(s)$, numerical payoff (win/loss/draw).

# Illustrative Examples

- **Deterministic, perfect information:** Chess, Checkers, Tic-Tac-Toe.
- **Hidden information:** Poker, Bridge.
- **Stochastic play:** Rock–Paper–Scissors with mixed strategies.

# Game Trees

- States as nodes, actions as edges; MAX/MIN alternating layers.

# Introduction to Game Trees

- States are represented as **nodes**, actions as **edges.**
- Players alternate turns: MAX tries to maximize, MIN tries to minimize.
- Game tree encodes all possible sequences of play.

# Structure of Game Trees

- Root node: the **initial state** (current game conditions).
- Internal nodes: non-terminal states with available actions.
- Leaf nodes: terminal states, labeled with utility values.

# MAX and MIN Nodes

- MAX nodes: the agent selects actions to **maximize utility.**
- MIN nodes: the opponent selects actions to **minimize utility.**
- Tree alternates layers of MAX and MIN.

# Example Game Tree

- A small example (tic-tac-toe fragment).
- Internal nodes alternate MAX and MIN.
- Leaves annotated with utility values (e.g., win, lose, draw).

# Minimax Algorithm

- Compute optimal play via recursive value propagation.

# Motivation for Minimax

- In adversarial games, MAX tries to maximize utility, MIN tries to minimize it.
- A game tree encodes all possible outcomes of play.
- The goal: compute the optimal strategy by reasoning about the opponent's moves.

# Definition of Minimax Value

- Terminal nodes have **utility values** (win, lose, draw).
- **MAX** nodes take the maximum of their children's values.
- **MIN** nodes take the minimum of their children's values.
- Values propagate upward from leaves to the root.

# Algorithmic Formulation

- Recursive algorithm:
  - If node is terminal: return its utility.
  - If node is MAX: return $\max$ of minimax values of children.
  - If node is MIN: return $\min$ of minimax values of children.
- Guarantees optimal play under perfect information.

# Minimax Algorithm (Pseudocode)

**Algorithm 1** Minimax($s$)

```
if s is terminal then
    return u(s)                                  ▷ utility value of terminal state
else if player(s) = MAX then
    return max_{a ∈ A(s)} Minimax(T(s, a))
else
    return min_{a ∈ A(s)} Minimax(T(s, a))
end if
```

# Minimax Algorithm

**Algorithm 2** Minimax($s$)

```
return MAX-VALUE(s)
```

**Algorithm 3** Max-Value($s$)

```
if TERMINAL(s) then
    return u(s)
end if
v ← −∞
for a ∈ A(s) do
    v ← max(v, MIN-VALUE(T(s, a)))
end for
return v
```

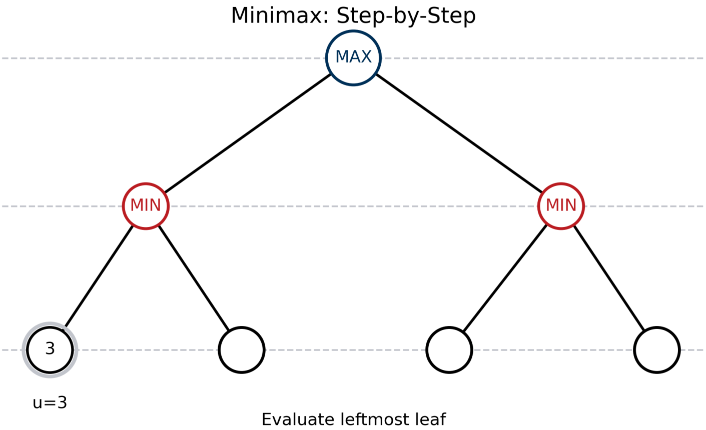
**Algorithm 4** Min-Value($s$)

```
if TERMINAL(s) then
    return u(s)
end if
v ← +∞
for a ∈ A(s) do
    v ← min(v, MAX-VALUE(T(s, a)))
end for
return v
```

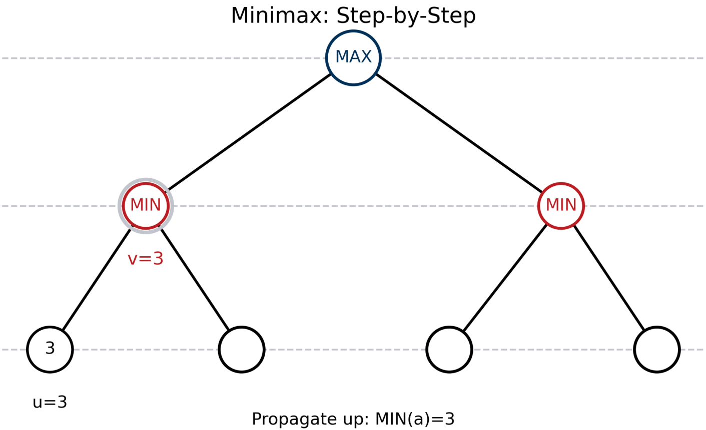
# Worked Example

- Consider a small game tree (e.g., tic–tac–toe fragment).
- Annotate terminal states with utilities.
- Show values being propagated upward.
- Root's minimax value determines the best move for MAX.

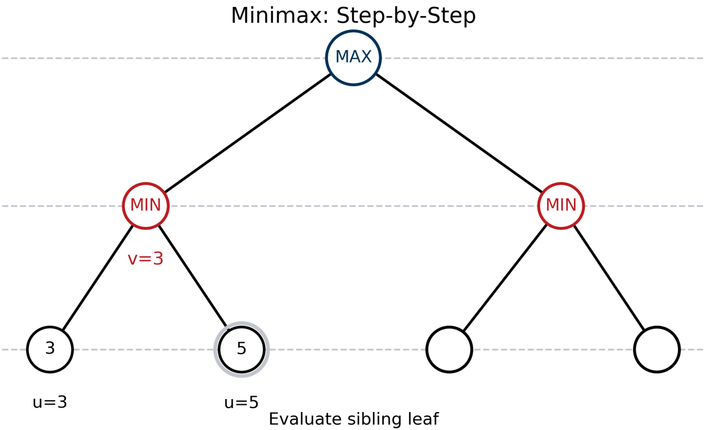
# Minimax: Step 0



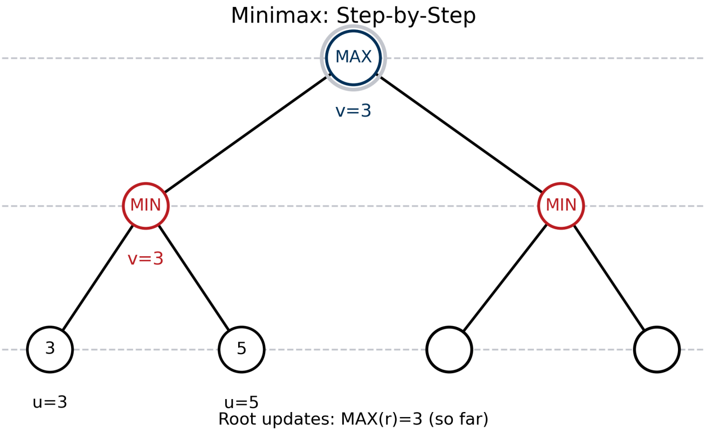
# Minimax: Step 1



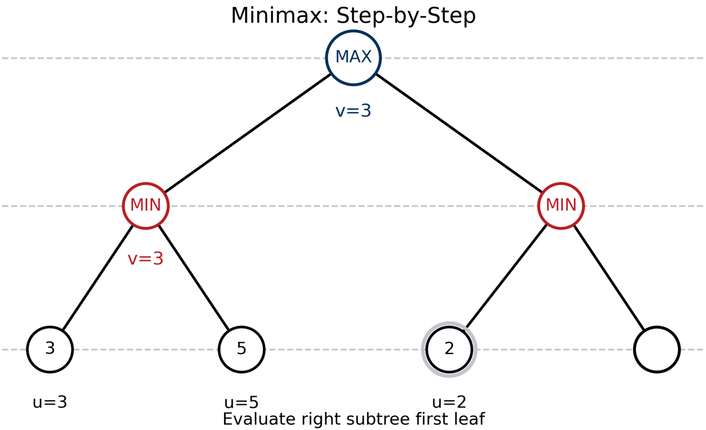
# Minimax: Step 2



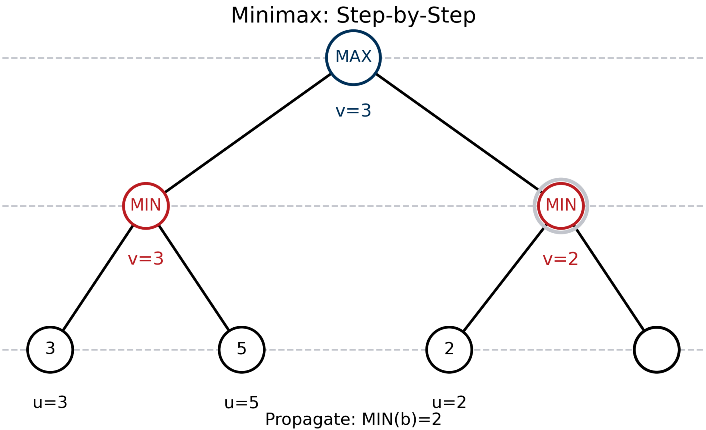
# Minimax: Step 3



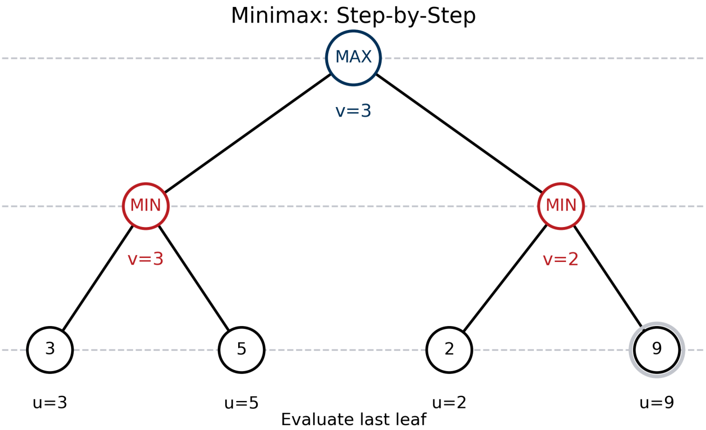
# Minimax: Step 4



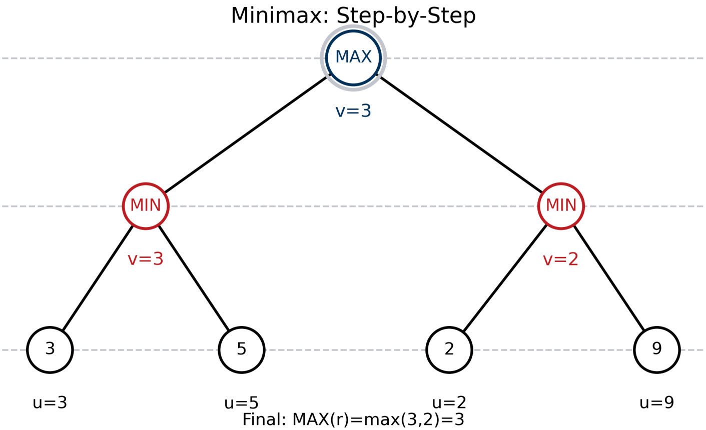
# Minimax: Step 5



# Minimax: Step 6



# Minimax: Step 7



# Alpha–Beta Pruning

- Prune branches while preserving the minimax result.

# Why Alpha–Beta Pruning?

- Minimax is **complete** and **optimal**, but explores the entire game tree.
- Number of nodes grows exponentially: $O(b^d)$ for branching factor $b$ and depth $d$.
- Alpha–Beta pruning reduces the effective branching factor.

# Key Idea of Alpha–Beta

- Maintain two bounds while searching:
  - $\alpha$: best value MAX can guarantee so far.
  - $\beta$: best value MIN can guarantee so far.
- If a node's value is outside these bounds, further exploration can be **pruned**.

# Pruning in Action

- As we traverse the tree, some branches cannot influence the final decision.
- These branches are cut off (“pruned”) without full evaluation.
- Example: once MAX has a choice better than what MIN allows elsewhere, skip the rest.
- **Important:** pruning never changes the final minimax value.

# Alpha–Beta Pruning (Wrapper)

**Algorithm 5** AlphaBeta($s$)

1: **return** MAX-VALUE-AB($s,\ \alpha = -\infty,\ \beta = +\infty$)

# Alpha–Beta: MAX-VALUE-AB

**Algorithm 6** Max-Value-AB($s,\ \alpha,\ \beta$)

**if** TERMINAL($s$) **then**<br>
    **return** $u(s)$<br>
**end if**<br>
$v \leftarrow -\infty$<br>
**for** $a \in A(s)$ **do**<br>
    $v \leftarrow \max\left(v,\ \text{MIN-VALUE-AB}(T(s,a),\ \alpha,\ \beta)\right)$<br>
    $\alpha \leftarrow \max(\alpha,\ v)$<br>
    **if** $\alpha \geq \beta$ **then**<br>
        **break** ▷ prune: MIN has a better option elsewhere<br>
    **end if**<br>
**end for**<br>
**return** $v$

# Alpha–Beta: MIN-VALUE-AB

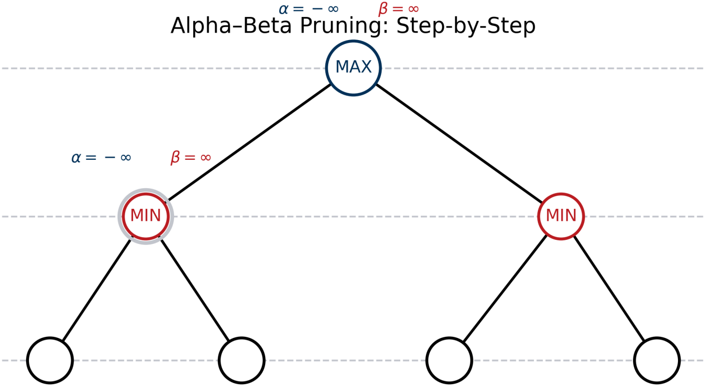
**Algorithm 7** Min-Value-AB($s,\ \alpha,\ \beta$)

```
if TERMINAL(s) then
    return u(s)
end if
v ← +∞
for a ∈ A(s) do
    v ← min(v, MAX-VALUE-AB(T(s, a), α, β))
    β ← min(β, v)
    if α ≥ β then
        break                ▷ prune: MAX has a better option elsewhere
    end if
end for
return v
```

# Properties of Alpha–Beta

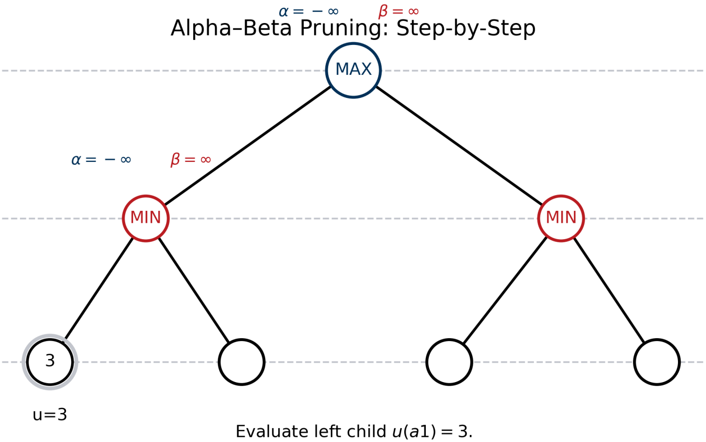
- Returns the same optimal move as minimax.
- Reduces number of nodes expanded.
- With perfect move ordering: $O(b^{d/2})$ instead of $O(b^d)$.
- In practice: huge efficiency gain, especially in deeper trees.

# Alpha–Beta: Step 0

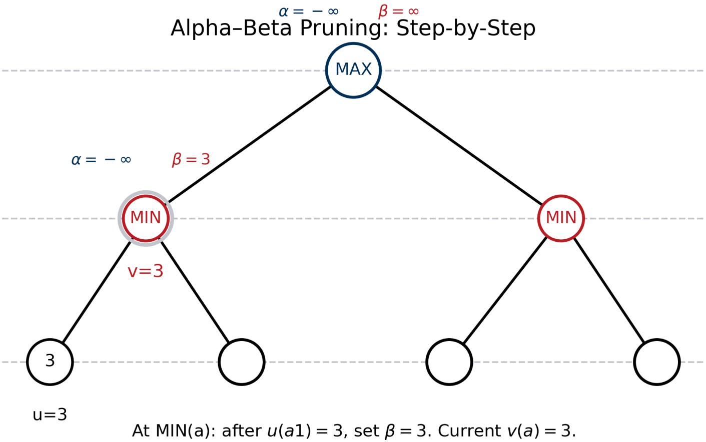



Enter left MIN(a) with initial bounds: $\alpha = -\infty,\ \beta = +\infty$.

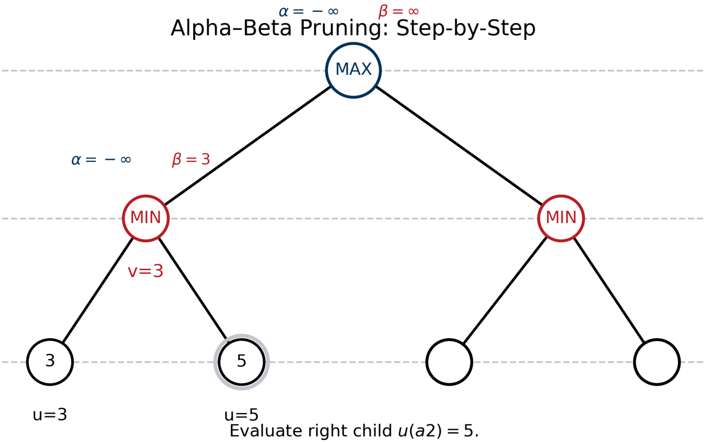
# Alpha–Beta: Step 1



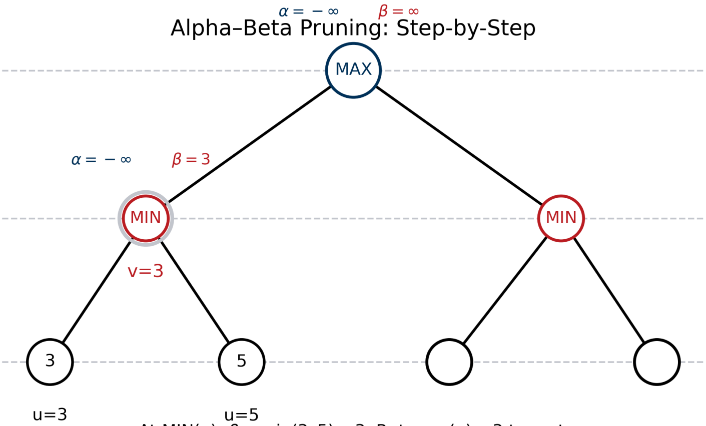
# Alpha–Beta: Step 2



# Alpha–Beta: Step 3

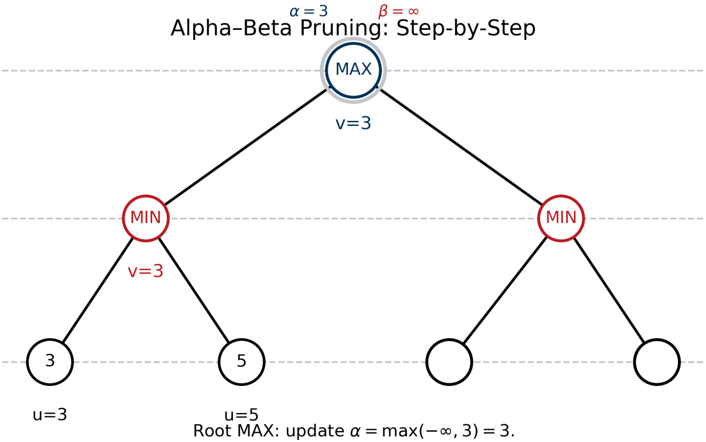


# Alpha–Beta: Step 4

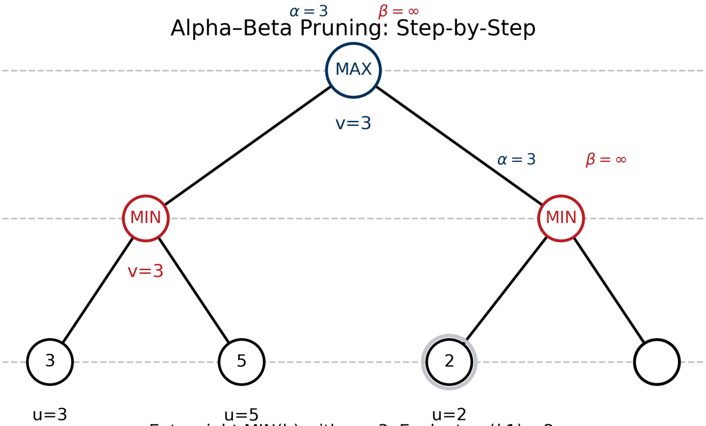



At MIN(a): $\beta = \min(3, 5) = 3$. Return $v(a) = 3$ to root.

# Alpha–Beta: Step 5

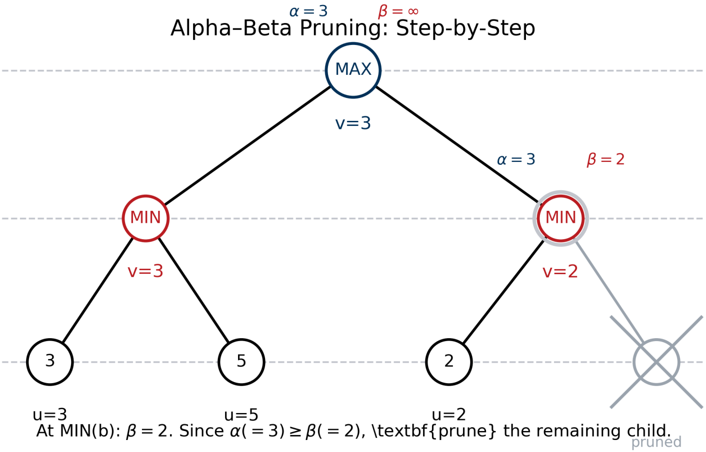


# Alpha–Beta: Step 6

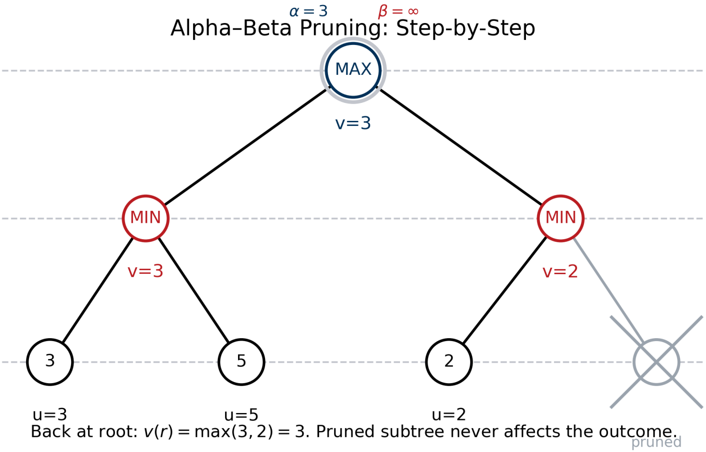



Enter right MIN(b) with $\alpha = 3$. Evaluate $u(b1) = 2$.

# Alpha–Beta: Step 7



# Alpha–Beta: Step 8



# Practical Considerations

- Depth limits, evaluation functions, and the horizon effect.

# Depth Limits

- In real games, the search tree is too large to fully expand.
- A common approach: limit search to a fixed depth $d$.
- Tradeoff: shallower depth is faster but less accurate.

# Evaluation Functions

- Used at cutoff nodes to approximate utility values.
- Must be quick to compute.
- Should correlate well with the true chance of winning.
- Example: material balance in chess.

# Alpha–Beta with Cutoff

- Use Alpha–Beta pruning, but stop at depth limit $d$.
- At cutoff nodes, apply an evaluation function instead of $u(s)$.
- This is the standard approach in practice (e.g., chess programs).

# Alpha–Beta with Cutoff (Wrapper)

**Algorithm 8** AlphaBetaCutoff($s$, $d$)

1: **return** MAX-VALUE-AB($s,\ \alpha = -\infty,\ \beta = +\infty,\ d$)

# Alpha–Beta with Cutoff: MAX-VALUE-AB

**Algorithm 9** Max-Value-AB($s,\ \alpha,\ \beta,\ d$)

**if** $s$ is terminal **then**
    **return** $u(s)$ ▷ true utility at terminal
**else if** $d = 0$ **then**
    **return** $\mathrm{Eval}(s)$ ▷ approximate value at cutoff
**end if**
$v \leftarrow -\infty$
**for** $a \in A(s)$ **do**
    $v \leftarrow \max\big(v,\ \text{MIN-VALUE-AB}(T(s,a),\ \alpha,\ \beta,\ d-1)\big)$
    $\alpha \leftarrow \max(\alpha,\ v)$
    **if** $\alpha \geq \beta$ **then**
        **break** ▷ beta cutoff (prune)
    **end if**
**end for**
**return** $v$

# Alpha–Beta with Cutoff: MIN-VALUE-AB

**Algorithm 10** Min-Value-AB($s,\ \alpha,\ \beta,\ d$)

**if** $s$ is terminal **then**
    **return** $u(s)$ ▷ true utility at terminal
**else if** $d = 0$ **then**
    **return** $\text{Eval}(s)$ ▷ approximate value at cutoff
**end if**
$v \leftarrow +\infty$
**for** $a \in A(s)$ **do**
    $v \leftarrow \min\big(v,\ $ MAX-VALUE-AB$(T(s,a),\ \alpha,\ \beta,\ d-1)\big)$
    $\beta \leftarrow \min(\beta,\ v)$
    **if** $\alpha \geq \beta$ **then**
        **break** ▷ alpha cutoff (prune)
    **end if**
**end for**
**return** $v$

# Horizon Effect

- Artificial cutoff can miss important consequences beyond $d$.
- Example: a forced loss that occurs just past the horizon.
- Agents may overvalue moves that only delay bad outcomes.

# Tradeoffs

- Deeper search $\Rightarrow$ more accurate but slower.
- Shallower search $\Rightarrow$ faster but less accurate.
- Move ordering heuristics improve efficiency of Alpha–Beta pruning.

# Modern Connections

- Classical evaluation functions vs. learned neural networks (e.g., AlphaZero).
- Monte Carlo methods (e.g., Monte Carlo Tree Search in Go).
- Balance of search and approximation is still central today.