Artificial Intelligence

Lecture 4: Game Playing & Adversarial Search

Prof. Antonio Khalil Moretti

Week 5

SCIS 432
Spelman College

Today's Topics

So far: Single-agent search

- One player trying to reach a goal
- Environment is static

Today: Multi-agent adversarial search (games!)

1. Game trees and terminology
2. Minimax algorithm
3. Alpha-beta pruning
4. Monte Carlo Tree Search (MCTS)

What Makes Games Different?

Single-Agent Search:

- Find path to goal
- Environment is static
- Plan ahead to goal

Example: Maze, 8-puzzle

Adversarial Search:

- Opponent trying to win
- Environment changes
- Plan considering opponent

Example: Chess, checkers, Go

Key challenge: Opponent is unpredictable and adversarial!

Solution: Assume opponent plays optimally (worst case for us)

Game Terminology

Game Properties:

- **Two-player**
- **Zero-sum:** My win = your loss
- **Deterministic:** No chance
- **Perfect information:** Fully observable

Examples:

- Chess, checkers, Go
- Tic-tac-toe, Connect-4

Game Elements:

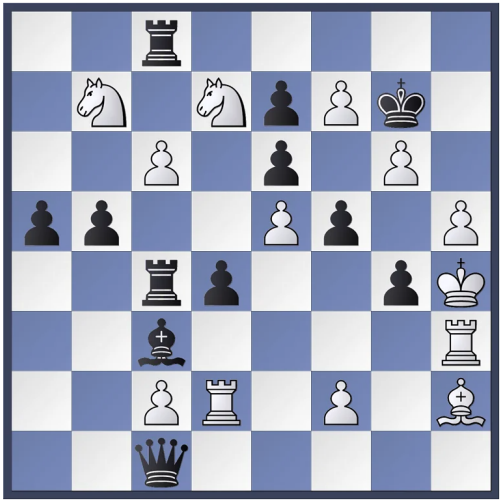
- **Initial state:** Starting position
- **Actions:** Legal moves
- **Transition:** Result of move
- **Terminal test:** Game over?
- **Utility:** Payoff at end

Players:

- **MAX:** Tries to maximize utility
- **MIN:** Tries to minimize utility

We use MAX/MIN instead of X/O because the concepts apply to any two-player game

Game Playing as Search



Single board position is one state in search space.

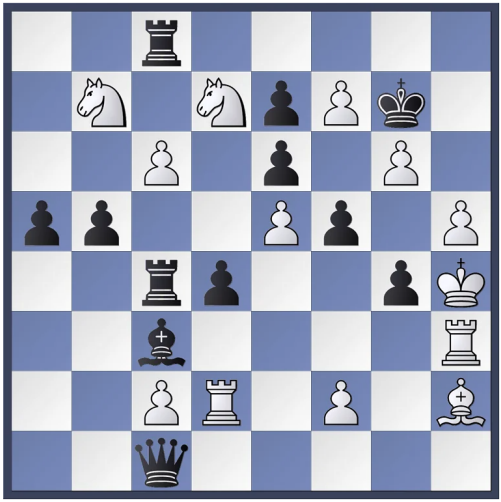
Try white to move Try black to move

Key idea:

A game-playing agent searches through a **state space**.

- Each board position is a **state**
- A move transforms one state into another
- The opponent also changes the state

Game Playing as Search



Single board position is one state in search space.

Try white to move Try black to move

Key idea:

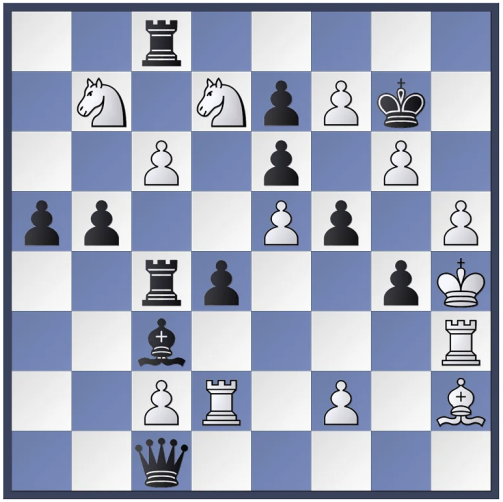
A game-playing agent searches through a **state space**.

- Each board position is a **state**
- A move transforms one state into another
- The opponent also changes the state

Challenge:

The future depends on *both* players' decisions.

Game Playing as Search



Single board position is one state in search space.

Try white to move Try black to move

Key idea:

A game-playing agent searches through a **state space**.

- Each board position is a **state**
- A move transforms one state into another
- The opponent also changes the state

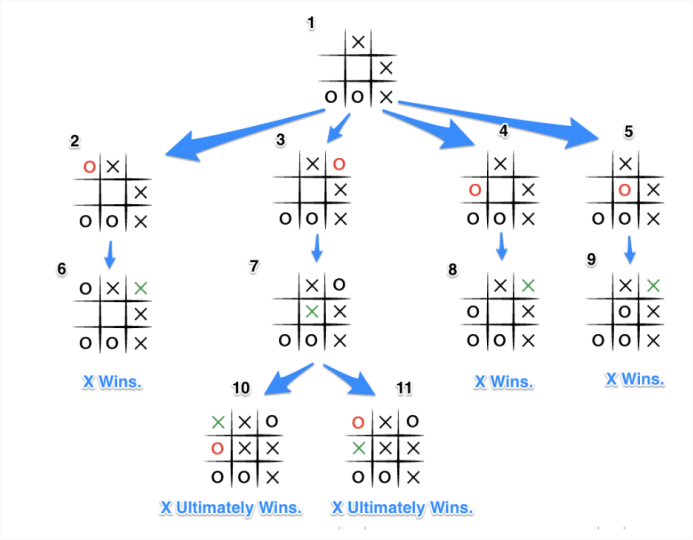
Challenge:

The future depends on *both* players' decisions.

Solution:

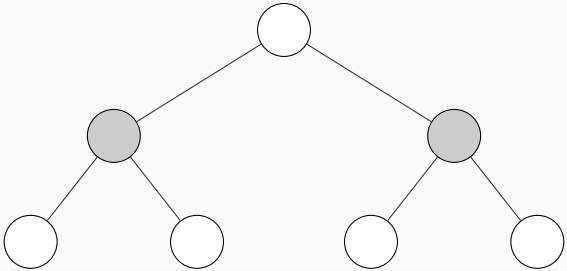
Explicitly model both players in a **game tree**.

Intuition: Tic-Tac-Toe as a Search Tree



Credit: https://www.neverstopbuilding.com/blog/minimax

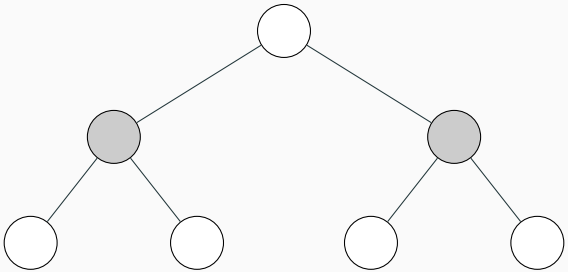
Looking Ahead in a Game Tree



Each node represents a **game position**. Each edge represents a **possible move**.

For visualization: Assume a simple two move game

Looking Ahead in a Game Tree

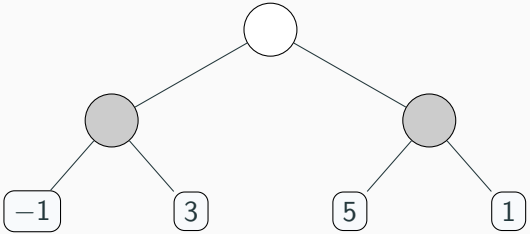


Each node represents a **game position**. Each edge represents a **possible move**.

For visualization: Assume a simple two move game

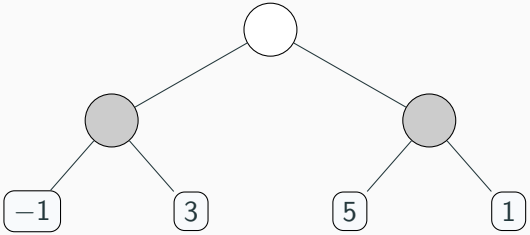
White (MAX) tries to **maximize** the score, Black (MIN) tries to **minimize** it.

Static Evaluation at the Leaves



At a fixed depth, we stop expanding the tree and apply a **static evaluation function**.

Static Evaluation at the Leaves

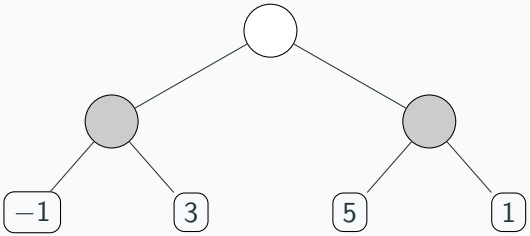


At a fixed depth, we stop expanding the tree and apply a **static evaluation function**.

Example: simple chess material values

♙ Pawn	1 point
♘, ♗ Knight, Bishop	3 points
♖ Rook	5 points
♕ Queen	9 points
♔ King	∞ (losing king ends game)

Static Evaluation at the Leaves



At a fixed depth, we stop expanding the tree and apply a **static evaluation function**.

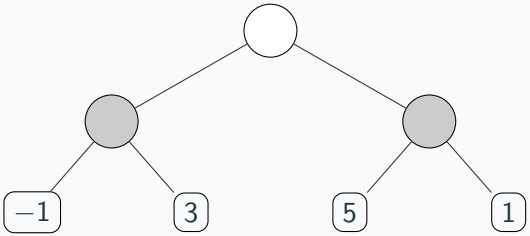
Example: simple chess material values

♙ Pawn	1 point
♘, ♗ Knight, Bishop	3 points
♖ Rook	5 points
♕ Queen	9 points
♔ King	∞ (losing king ends game)

Score computation:

$$\text{evaluation} = (\text{White material}) - (\text{Black material})$$

Static Evaluation at the Leaves



At a fixed depth, we stop expanding the tree and apply a **static evaluation function**.

Example: simple chess material values

♙ Pawn	1 point
♘, ♗ Knight, Bishop	3 points
♖ Rook	5 points
♕ Queen	9 points
♔ King	∞ (losing king ends game)

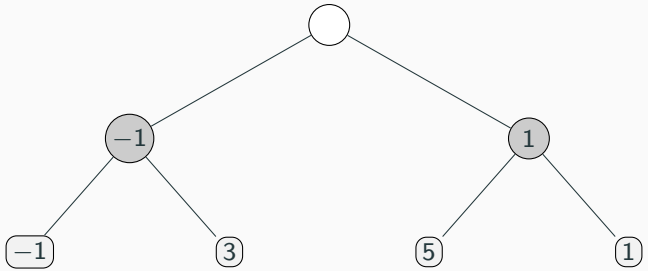
Score computation:

$$\text{evaluation} = (\text{White material}) - (\text{Black material})$$

Positive values favor White, negative values favor Black.

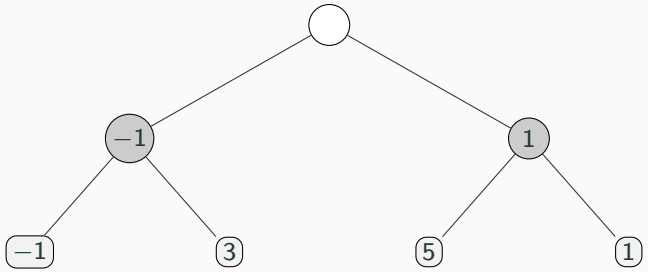
MIN Nodes: Choose the Minimum Value

Evaluating at MIN Nodes



MIN Nodes: Choose the Minimum Value

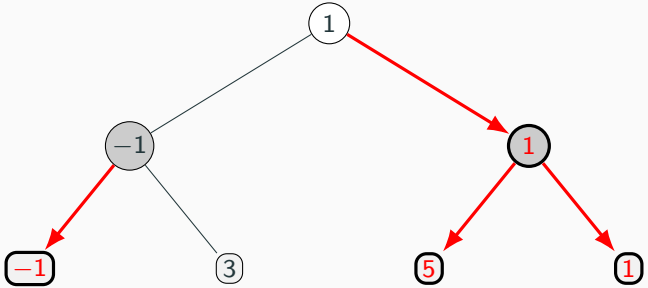
Evaluating at MIN Nodes



At a **MIN** node (Black's turn), the opponent is assumed to play optimally, so the node's value is the **minimum** of its children.

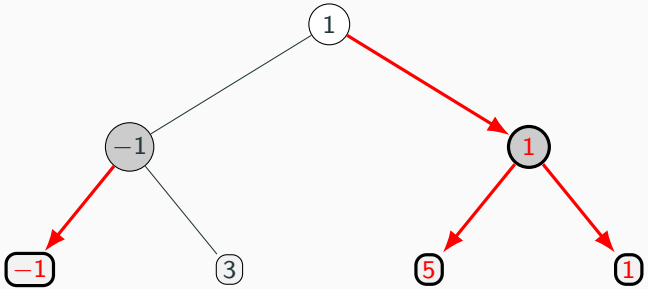
MAX Nodes: Choose the Maximum Value

Evaluating at MAX Nodes



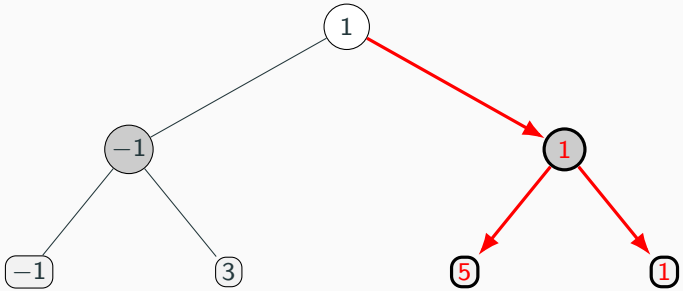
MAX Nodes: Choose the Maximum Value

Evaluating at MAX Nodes

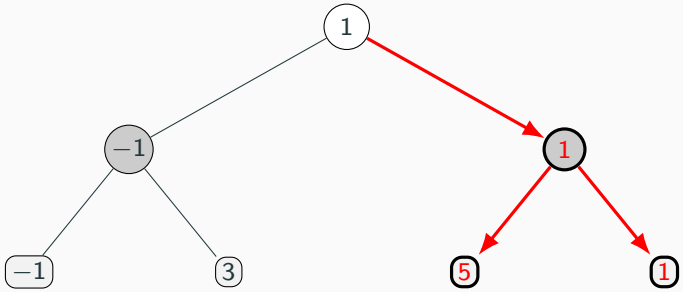


At a **MAX** node (White's turn), we select the move that leads to the **largest** value returned by its children.

Minimax Decision at the Root



Minimax Decision at the Root



By alternating MAX and MIN updates up the tree, the root value represents the best guaranteed outcome assuming optimal play from both sides.

Minimax: From Idea to Code

```
def minimax(position, depth,
    maximizing_player):
    if depth == 0 or position.
        is_terminal():
        return evaluate(position)
    # White (MAX)
    if maximizing_player:
        max_eval = float('-inf')
        for child in position.
            children():
            eval = minimax(child,
                depth - 1, False)
            max_eval = max(max_eval,
                eval)
        return max_eval
```

Minimax: From Idea to Code

```
def minimax(position, depth,
    maximizing_player):
    if depth == 0 or position.
        is_terminal():
        return evaluate(position)
    # White (MAX)
    if maximizing_player:
        max_eval = float('-inf')
        for child in position.
            children():
            eval = minimax(child,
                depth - 1, False)
            max_eval = max(max_eval,
                eval)
        return max_eval
```

```
    else:  # Black (MIN)
        min_eval = float('inf')
        for child in position.
            children():
            eval = minimax(child,
                depth - 1, True)
            min_eval = min(
                min_eval, eval)
        return min_eval
```

White (MAX) looks ahead and keeps the **best possible outcome**.

Minimax: From Idea to Code

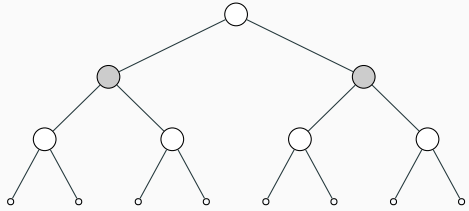
```
def minimax(position, depth,
    maximizing_player):
    if depth == 0 or position.
        is_terminal():
        return evaluate(position)
    # White (MAX)
    if maximizing_player:
        max_eval = float('-inf')
        for child in position.
            children():
            eval = minimax(child,
                depth - 1, False)
            max_eval = max(max_eval,
                eval)
        return max_eval
```

White (MAX) looks ahead and keeps the **best possible outcome**.

```
    else:  # Black (MIN)
        min_eval = float('inf')
        for child in position.
            children():
            eval = minimax(child,
                depth - 1, True)
            min_eval = min(
                min_eval, eval)
        return min_eval
```

Black (MIN) assumes the worst for White and keeps the **lowest value**.

Minimax Walk-Through: Tree + Code



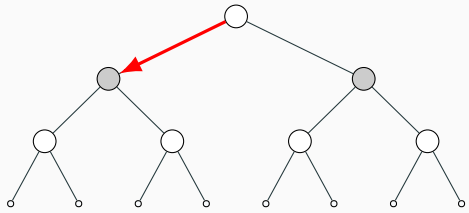
Start at the root (White = MAX). Evaluate the tree by recursion.

```
def minimax(pos, depth, is_max):
    if depth == 0 or pos.is_terminal():
        return evaluate(pos)

    if is_max:  # White (MAX)
        max_eval = -inf
        for child in pos.children():
            eval = minimax(child,
                           depth - 1,
                           False)
            max_eval = max(max_eval, eval)
        return max_eval

    else:  # Black (MIN)
        min_eval = +inf
        for child in pos.children():
            eval = minimax(child,
                           depth - 1,
                           True)
            min_eval = min(min_eval, eval)
        return min_eval
```

Minimax Walk-Through: Tree + Code



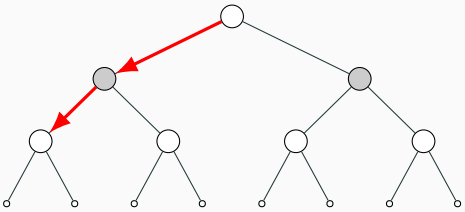
MAX wants the **max** of its two children ⇒ recurse on the left child.

```
def minimax(pos, depth, is_max):
    if depth == 0 or pos.is_terminal():
        return evaluate(pos)

    if is_max:  # White (MAX)
        max_eval = -inf
        for child in pos.children():
            eval = minimax(child,
                           depth - 1,
                           False)
            max_eval = max(max_eval, eval)
        return max_eval

    else:  # Black (MIN)
        min_eval = +inf
        for child in pos.children():
            eval = minimax(child,
                           depth - 1,
                           True)
            min_eval = min(min_eval, eval)
        return min_eval
```

Minimax Walk-Through: Tree + Code



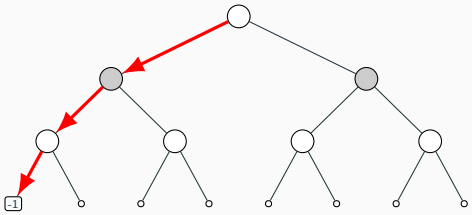
At a MIN node (Black), recurse on its left child (a MAX node).

```
def minimax(pos, depth, is_max):
    if depth == 0 or pos.is_terminal():
        return evaluate(pos)

    if is_max:  # White (MAX)
        max_eval = -inf
        for child in pos.children():
            eval = minimax(child,
                           depth - 1,
                           False)
            max_eval = max(max_eval, eval)
        return max_eval

    else:  # Black (MIN)
        min_eval = +inf
        for child in pos.children():
            eval = minimax(child,
                           depth - 1,
                           True)
            min_eval = min(min_eval, eval)
        return min_eval
```

Minimax Walk-Through: Tree + Code



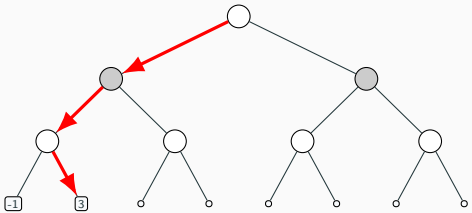
Depth $= 0$ at the leaf $\Rightarrow$ return the **static evaluation** (-1).

```
def minimax(pos, depth, is_max):
    if depth == 0 or pos.is_terminal():
        return evaluate(pos)

    if is_max:  # White (MAX)
        max_eval = -inf
        for child in pos.children():
            eval = minimax(child,
                           depth - 1,
                           False)
            max_eval = max(max_eval, eval)
        return max_eval

    else:  # Black (MIN)
        min_eval = +inf
        for child in pos.children():
            eval = minimax(child,
                           depth - 1,
                           True)
            min_eval = min(min_eval, eval)
        return min_eval
```

Minimax Walk-Through: Tree + Code



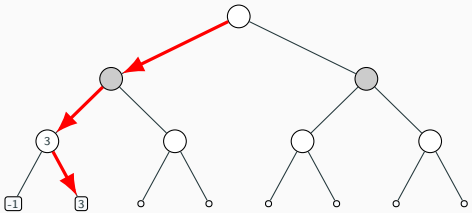
Evaluate the sibling leaf (3). MAX at this node returns $\max(-1, 3) = 3$.

```
def minimax(pos, depth, is_max):
    if depth == 0 or pos.is_terminal():
        return evaluate(pos)

    if is_max:  # White (MAX)
        max_eval = -inf
        for child in pos.children():
            eval = minimax(child,
                           depth - 1,
                           False)
            max_eval = max(max_eval, eval)
        return max_eval

    else:  # Black (MIN)
        min_eval = +inf
        for child in pos.children():
            eval = minimax(child,
                           depth - 1,
                           True)
            min_eval = min(min_eval, eval)
        return min_eval
```

Minimax Walk-Through: Tree + Code



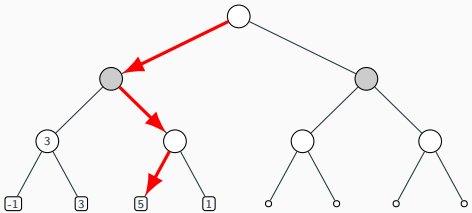
That value passes up. MIN now evaluates its other child.

```
def minimax(pos, depth, is_max):
    if depth == 0 or pos.is_terminal():
        return evaluate(pos)

    if is_max:  # White (MAX)
        max_eval = -inf
        for child in pos.children():
            eval = minimax(child,
                           depth - 1,
                           False)
            max_eval = max(max_eval, eval)
        return max_eval

    else:  # Black (MIN)
        min_eval = +inf
        for child in pos.children():
            eval = minimax(child,
                           depth - 1,
                           True)
            min_eval = min(min_eval, eval)
        return min_eval
```

Minimax Walk-Through: Tree + Code



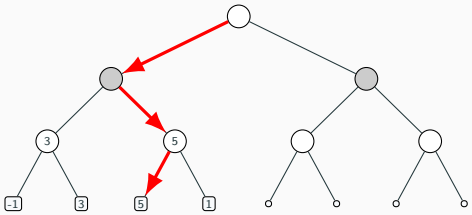
Evaluate the two leaves under the second MAX node (here: 5 and 1).

```
def minimax(pos, depth, is_max):
    if depth == 0 or pos.is_terminal():
        return evaluate(pos)

    if is_max:  # White (MAX)
        max_eval = -inf
        for child in pos.children():
            eval = minimax(child,
                           depth - 1,
                           False)
            max_eval = max(max_eval, eval)
        return max_eval

    else:  # Black (MIN)
        min_eval = +inf
        for child in pos.children():
            eval = minimax(child,
                           depth - 1,
                           True)
            min_eval = min(min_eval, eval)
        return min_eval
```

Minimax Walk-Through: Tree + Code



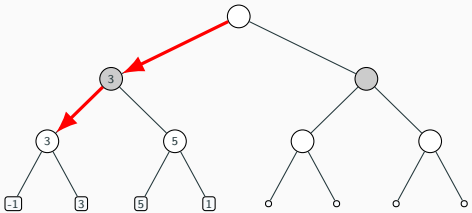
MAX returns $\max(5, 1) = 5$ to its parent.

```
def minimax(pos, depth, is_max):
    if depth == 0 or pos.is_terminal():
        return evaluate(pos)

    if is_max:  # White (MAX)
        max_eval = -inf
        for child in pos.children():
            eval = minimax(child,
                           depth - 1,
                           False)
            max_eval = max(max_eval, eval)
        return max_eval

    else:  # Black (MIN)
        min_eval = +inf
        for child in pos.children():
            eval = minimax(child,
                           depth - 1,
                           True)
            min_eval = min(min_eval, eval)
        return min_eval
```

Minimax Walk-Through: Tree + Code



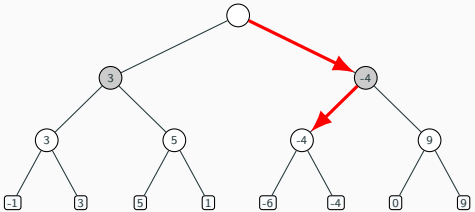
MIN returns $\min(3, 5) = 3$ upward (assumes opponent plays optimally).

```
def minimax(pos, depth, is_max):
    if depth == 0 or pos.is_terminal():
        return evaluate(pos)

    if is_max:  # White (MAX)
        max_eval = -inf
        for child in pos.children():
            eval = minimax(child,
                           depth - 1,
                           False)
            max_eval = max(max_eval, eval)
        return max_eval

    else:  # Black (MIN)
        min_eval = +inf
        for child in pos.children():
            eval = minimax(child,
                           depth - 1,
                           True)
            min_eval = min(min_eval, eval)
        return min_eval
```

Minimax Walk-Through: Tree + Code



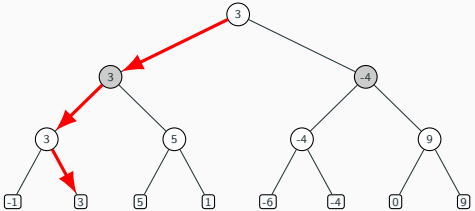
Repeat on the right subtree (values appear). MIN on the right returns -4.

```
def minimax(pos, depth, is_max):
    if depth == 0 or pos.is_terminal():
        return evaluate(pos)

    if is_max:  # White (MAX)
        max_eval = -inf
        for child in pos.children():
            eval = minimax(child,
                           depth - 1,
                           False)
            max_eval = max(max_eval, eval)
        return max_eval

    else:  # Black (MIN)
        min_eval = +inf
        for child in pos.children():
            eval = minimax(child,
                           depth - 1,
                           True)
            min_eval = min(min_eval, eval)
        return min_eval
```

Minimax Walk-Through: Tree + Code



Root MAX picks $\max(3, -4) = 3 \Rightarrow$ choose the **left move** (red path).

```
def minimax(pos, depth, is_max):
    if depth == 0 or pos.is_terminal():
        return evaluate(pos)

    if is_max:  # White (MAX)
        max_eval = -inf
        for child in pos.children():
            eval = minimax(child,
                           depth - 1,
                           False)
            max_eval = max(max_eval, eval)
        return max_eval

    else:  # Black (MIN)
        min_eval = +inf
        for child in pos.children():
            eval = minimax(child,
                           depth - 1,
                           True)
            min_eval = min(min_eval, eval)
        return min_eval
```

Example: Tic-Tac-Toe

Question: What is the *state space* (set of all possible configurations)?

Example: Tic-Tac-Toe

Question: What is the *state space* (set of all possible configurations)?

Answer: All possible board configurations (with 0-9 X's and O's)

Example: Tic-Tac-Toe

Question: What is the *state space* (set of all possible configurations)?

Answer: All possible board configurations (with 0-9 X's and O's)

Question: What is the maximum branching factor (how many possible first moves)?

Example: Tic-Tac-Toe

Question: What is the *state space* (set of all possible configurations)?

Answer: All possible board configurations (with 0-9 X's and O's)

Question: What is the maximum branching factor (how many possible first moves)?

Answer: 9 (first move), then 8, 7, ... decreasing

Tic-Tac-Toe Complexity

State space size:

- Upper bound: $3^9 = 19{,}683$ positions
- Legal positions: $\sim$5,478
- After symmetries: $\sim$765

Game tree:

- Max depth: 9 moves
- First move: 9 choices
- Total nodes: $\sim$294,778

Complexity:

- Small enough to solve completely!
- Can explore entire tree
- Optimal play always draws

Contrast with chess:

- $\sim 10^{120}$ possible games
- Cannot explore everything
- Need smart search

Combinatorics Reminder

Counting game states often reduces to **counting ways to choose discrete choices**.

Binomial coefficient ("n choose k"):

$$\binom{n}{k} \quad = \quad \text{number of ways to choose } k \text{ objects from } n$$

Factorial definition:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Intuition:

- $n!$: all possible orderings
- divide by $k!$: order of chosen items doesn't matter
- divide by $(n-k)!$: order of unchosen items doesn't matter

We'll use these terms repeatedly when counting reachable game states.

Why Tic-Tac-Toe Has 5,478 Legal Positions

Step 1: Naive upper bound

Each square can be {X, O, blank}:

$$3^9 = 19{,}683 \quad \text{possible boards}$$

Step 2: Enforce turn-taking (X moves first)

In any reachable position: $\#X = \#O$ or $\#X = \#O + 1$

Counting boards that satisfy this:

$$\sum_{o=0}^{4} \left[\binom{9}{o} \binom{9-o}{o} + \binom{9}{o+1} \binom{8-o}{o} \right] = 6{,}046$$

Step 3: Remove unreachable boards

Some of these boards violate rules (both players winning, or moves played after a win).

$$\boxed{6{,}046 - 568 = 5{,}478}$$

State Space Sizes Across Games

Game	State Space	Branching	Solved?
Tic-Tac-Toe	$\approx 5 \times 10^3$	≤ 9	Yes
Connect Four	$\approx 4 \times 10^{13}$	≤ 7	Yes
Checkers	$\approx 5 \times 10^{20}$	≈ 8	Yes
Chess	$\approx 10^{43}$	≈ 35	No
Go	$\approx 10^{170}$	≈ 250	No

The number of possible **chess games** ($\sim 10^{120}$) vastly exceeds the number of atoms in the *observable universe* ($\sim 10^{80}$).

Key idea: Exhaustive search becomes impossible very quickly.

Minimax Algorithm

Optimal play assuming perfect opponent

Key idea: Assume opponent plays optimally

MAX's strategy:

- Choose move that maximizes the minimum achievable payoff
- "What's the best I can guarantee against optimal play?"

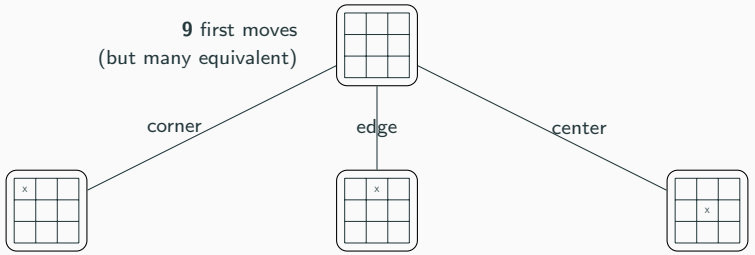
MIN's strategy:

- Choose move that minimizes the maximum achievable payoff
- "How can I minimize MAX's best outcome?"

Implementation:

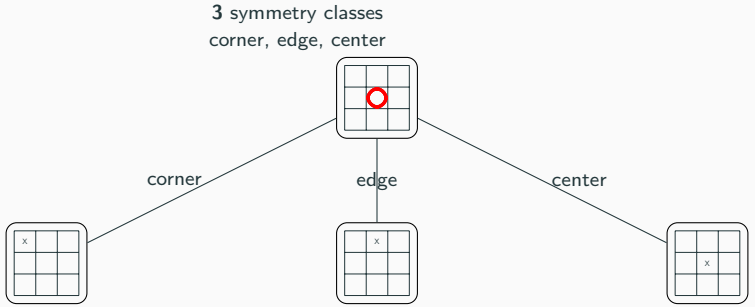
- Recursive depth-first search
- Values propagate up from terminal states
- MAX takes maximum, MIN takes minimum

Top of the Tic-Tac-Toe Minimax Tree (Symmetry Classes)



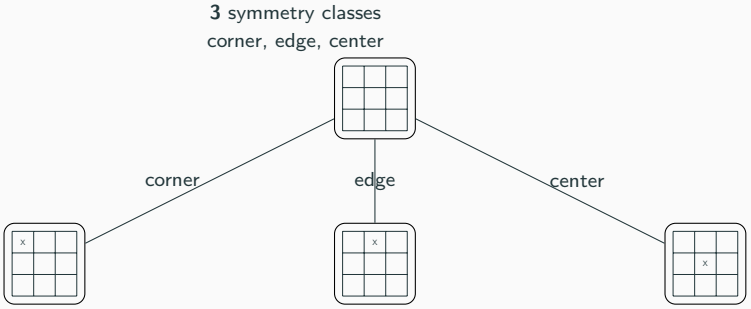
Minimax starts from the empty board: **9** possible moves for X.

Top of the Tic-Tac-Toe Minimax Tree (Symmetry Classes)



We don't need all 9: rotations/reflections make many moves equivalent.

Top of the Tic-Tac-Toe Minimax Tree (Symmetry Classes)



We evaluate only these **3 representatives**; the other first moves reuse the same result via `canonical(pos)`.

Minimax Properties

Guarantees:

- Optimal against optimal opponent
- Complete (finite games)
- Finds best move

Complexity:

- Time: $O(b^m)$
- Space: $O(bm)$ (DFS)
- b = branching factor
- m = maximum depth

Examples:

- Tic-tac-toe: tractable
- Chess: $b \approx 35$, $m \approx 100$
- Go: $b \approx 250$, $m \approx 150$

Problem: Exponential!

Solutions:

- Alpha-beta pruning
- Depth-limited search
- Evaluation functions

Alpha–Beta Pruning: The Idea

Problem: Plain minimax explores *every* node in the game tree even when some branches cannot possibly affect the final decision.

Key insight: We can keep track of bounds on the possible values and stop exploring branches that are already worse than what we've seen.

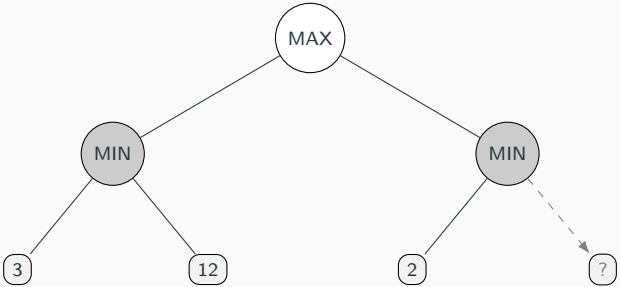
- α **(alpha):** Best value MAX can guarantee so far (lower bound for MAX)
- β **(beta):** Best value MIN can guarantee so far (upper bound for MIN)
- **Pruning rule:** If at any point $\alpha \geq \beta$, **this branch cannot affect the final choice** and can be ignored.

Why this is safe

If MIN can already force the value to be $\leq \beta$, and MAX already has a move guaranteeing $\geq \alpha$, then when $\alpha \geq \beta$, neither player would ever choose this branch.

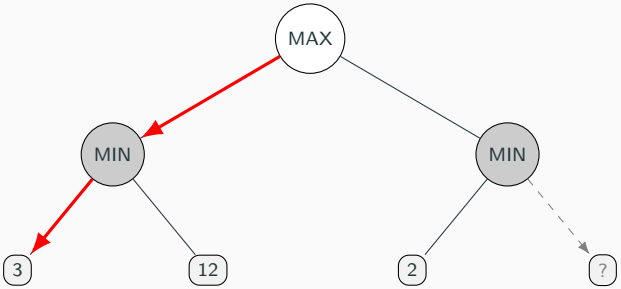
Result: Same answer as minimax but **far fewer nodes explored**.

Alpha–Beta Pruning: Worked Example



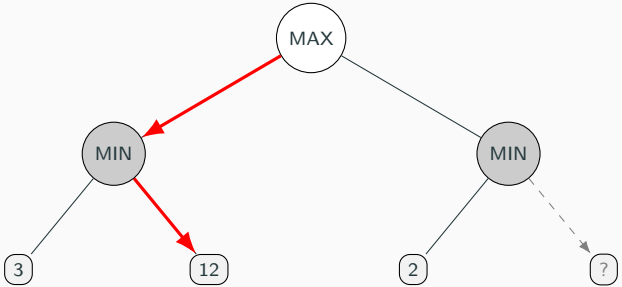
Start at the root (MAX). Initially: $\alpha = -\infty, \beta = +\infty$.

Alpha–Beta Pruning: Worked Example



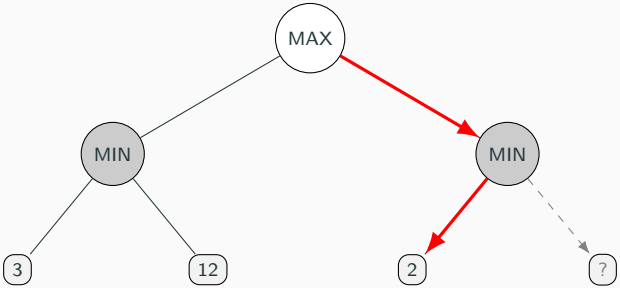
Explore left subtree: MIN sees value 3 so far $\Rightarrow \beta = 3$.

Alpha–Beta Pruning: Worked Example



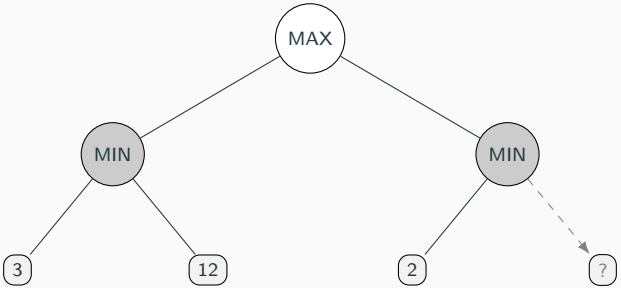
Second leaf is 12, but MIN keeps $\min(3, 12) = 3$. Return 3 to root $\Rightarrow \alpha = 3$.

Alpha–Beta Pruning: Worked Example



Explore right subtree: first leaf gives 2. Now at MIN: $\beta = 2$, but $\beta \leq \alpha$. **Prune the remaining branch**.

Alpha–Beta Pruning: Worked Example



Final result: MAX chooses value **3**, same as minimax — with less work.

Alpha-Beta Pruning

Minimax without searching irrelevant branches

Key insight: Don't explore branches that can't affect final decision

Maintain two values:

- α = best value for MAX so far (lower bound for MAX)
- β = best value for MIN so far (upper bound for MIN)

Pruning rule:

- At a MIN node: if $v \leq \alpha$, prune (MAX won't choose this branch)
- At a MAX node: if $v \geq \beta$, prune (MIN won't choose this branch)

Alpha-Beta Pruning: From Minimax to Faster Search

```
def alphabeta(pos, depth, alpha, beta, is_max):
    if depth == 0 or pos.is_terminal():
        return evaluate(pos)
    if is_max:  # White (MAX)
        value = float('-inf')
        for child in pos.children():
            value = max(value, alphabeta(child, depth - 1, alpha, beta, False))
            alpha = max(alpha, value)
            if alpha >= beta:
                break  # PRUNE: MIN won't allow this branch
        return value
    else:  # Black (MIN)
        value = float('inf')
        for child in pos.children():
            value = min(value, alphabeta(child, depth - 1, alpha, beta, True))
            beta = min(beta, value)
            if beta <= alpha:
                break  # PRUNE: MAX won't allow this branch
        return value
```

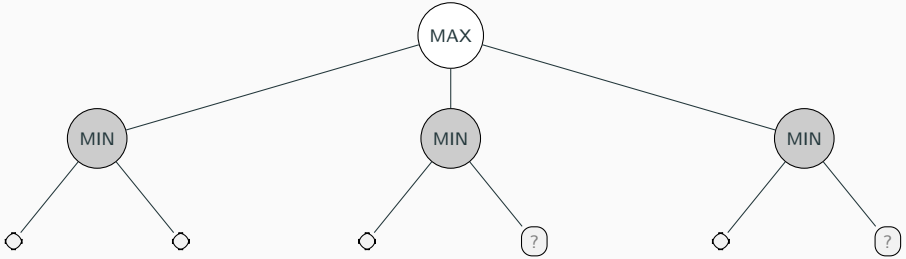
Alpha-Beta Pruning: From Minimax to Faster Search

```
def alphabeta(pos, depth, alpha, beta, is_max):
    if depth == 0 or pos.is_terminal():
        return evaluate(pos)
    if is_max:  # White (MAX)
        value = float('-inf')
        for child in pos.children():
            value = max(value, alphabeta(child, depth - 1, alpha, beta, False))
            alpha = max(alpha, value)
            if alpha >= beta:
                break  # PRUNE: MIN won't allow this branch
        return value
    else:  # Black (MIN)
        value = float('inf')
        for child in pos.children():
            value = min(value, alphabeta(child, depth - 1, alpha, beta, True))
            beta = min(beta, value)
            if beta <= alpha:
                break  # PRUNE: MAX won't allow this branch
        return value
```

- Same result as minimax but **fewer nodes explored**
- α: best value MAX can guarantee so far
- β: best value MIN can guarantee so far
- Stop searching a branch once it **can't affect the final decision**

Alpha-Beta Pruning: Interactive Walkthrough

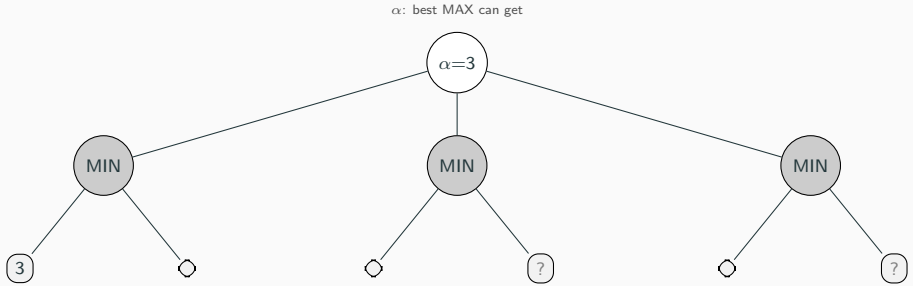
Can you spot when to prune?



Start: $\alpha = -\infty$, $\beta = +\infty$. Explore left subtree first.

Alpha-Beta Pruning: Interactive Walkthrough

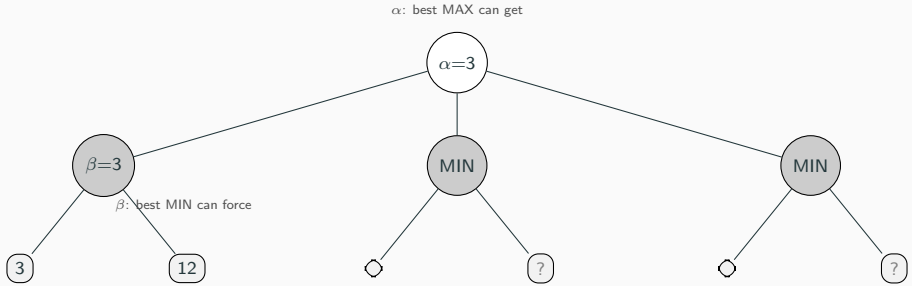
Can you spot when to prune?



Left MIN sees value **3**. Set β=3 (MIN's best so far).

Alpha-Beta Pruning: Interactive Walkthrough

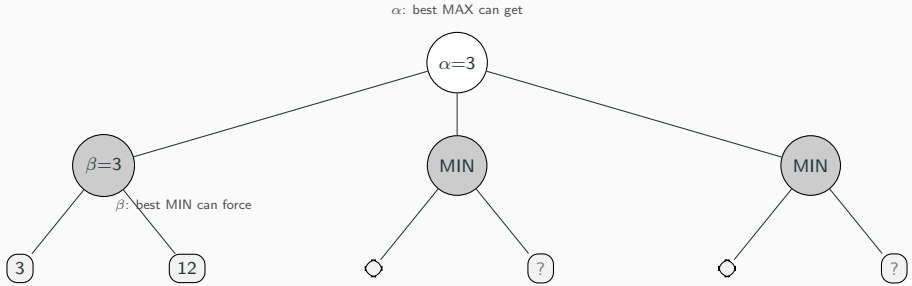
Can you spot when to prune?



Next leaf is 12, but MIN chooses $\min(3, 12)$=3. Return 3 to MAX.

Alpha-Beta Pruning: Interactive Walkthrough

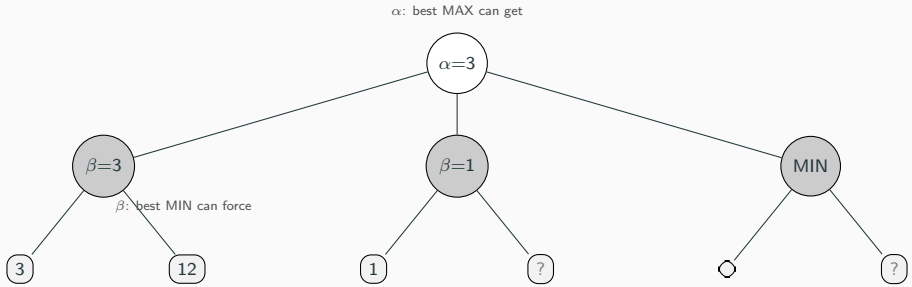
Can you spot when to prune?



MAX updates α=3 (best MAX can guarantee). Now explore middle branch.

Alpha-Beta Pruning: Interactive Walkthrough

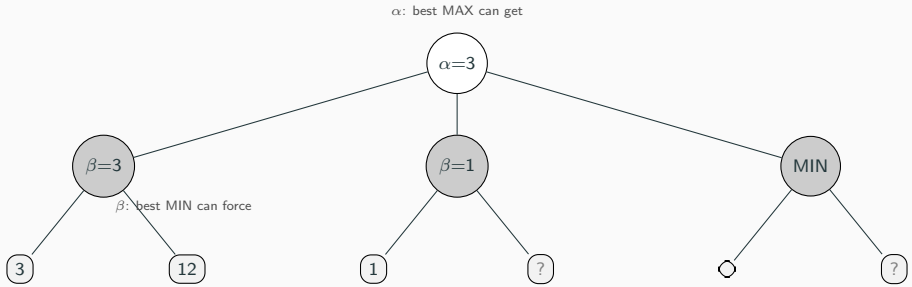
Can you spot when to prune?



Middle MIN sees value **1**. Set β=1.

Alpha-Beta Pruning: Interactive Walkthrough

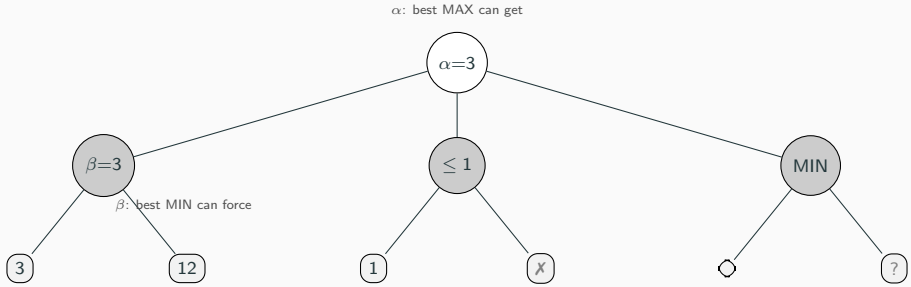
Can you spot when to prune?



QUESTION: Do we need to evaluate the ? position, or can we prune?

Alpha-Beta Pruning: Interactive Walkthrough

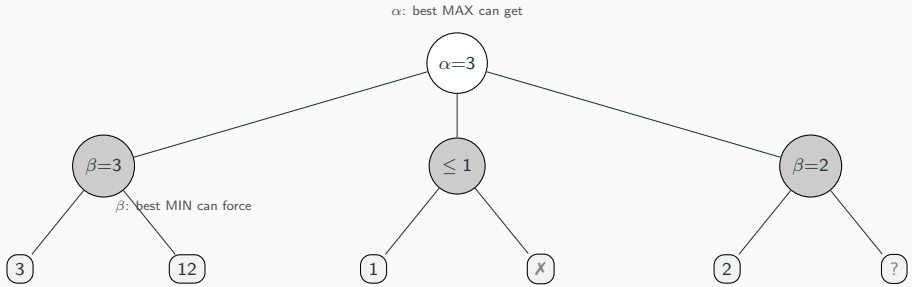
Can you spot when to prune?



ANSWER: Prune! MIN guarantees ≤ 1, but MAX already has $\alpha=3$. Since $\beta=1 < \alpha=3$, MAX will never choose this branch. **Prune!**

Alpha-Beta Pruning: Interactive Walkthrough

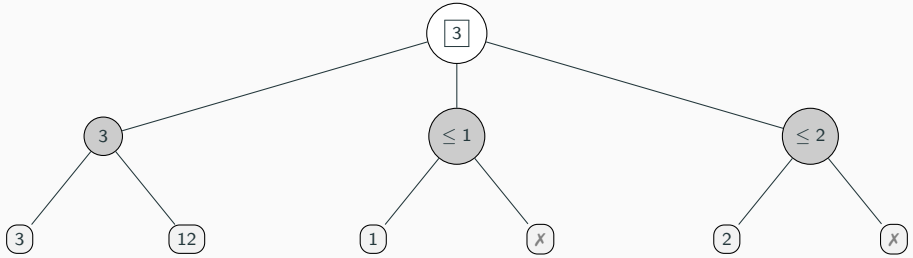
Can you spot when to prune?



Explore right branch: MIN sees **2**, so β=2.

Alpha-Beta Pruning: Interactive Walkthrough

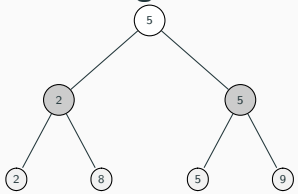
Can you spot when to prune?



Again: β=2 $<$ α=3 $\Rightarrow$ **Prune remaining nodes**. Final answer: MAX chooses **3** (left branch).

Move Ordering Matters for Pruning!

Bad Ordering: Worst First

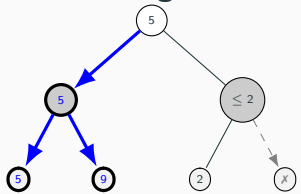


✗ Explores left (worse) branch first

✗ Must check all 4 leaves

✗ **No pruning occurs!**

Good Ordering: Best First



✓ Explores right (better) branch first

✓ Prunes after 3 leaves

✓ **25% fewer nodes!**

Move Ordering Heuristics (Chess Example)

Good moves to try first:

- Captures (especially ♙ takes ♕)
- Checks (forcing moves)
- Threats against high-value pieces
- Previously good moves (iterative deepening)

Key insight: Ordering from *best to worst for current player* maximizes pruning!

Alpha-Beta Properties

Pruning effectiveness:

- Worst case: No pruning, $O(b^m)$
- Best case: Perfect ordering, $O(b^{m/2})$
- Doubles solvable depth!

Move ordering matters:

- Search best moves first
- More pruning occurs
- Use heuristics for ordering

Chess example:

- $b = 35$, depth 6: $35^6 \approx 1.8$ billion
- With alpha-beta: $35^3 \approx 43{,}000$

Same result as minimax!

- Just faster
- Finds identical move
- Guaranteed optimal

Still exponential, but much better in practice!

Algorithm Comparison

Property	Minimax	Alpha-Beta	MCTS
Optimal?	Yes	Yes	Converges
Evaluation needed?	Yes	Yes	No
Time complexity	$O(b^m)$	$O(b^{m/2})$	Anytime
Best for	Small trees	Medium trees	Large trees
Chess	Good	Better	Okay
Go	Poor	Poor	Excellent

Modern game AI: Hybrid approaches

- Neural networks for evaluation
- MCTS for search
- Learned from self-play (AlphaZero)

Real-World Game AI

Checkers (1994):

- Chinook: World champion
- Alpha-beta search
- Solved in 2007!

Chess (1997):

- Deep Blue beats Kasparov
- Alpha-beta + evaluation
- Now: Stockfish, AlphaZero

Go (2016):

- AlphaGo beats Lee Sedol
- MCTS + deep learning
- Self-play learning

StarCraft II (2019):

- AlphaStar: Grandmaster level
- Partial observability
- Real-time decisions

Beyond two-player: Poker, Diplomacy (imperfect information, multi-agent)

Summary

Minimax:

- Optimal against perfect opponent
- MAX maximizes, MIN minimizes
- Exponential complexity

Alpha-Beta:

- Prunes irrelevant branches
- Same result, faster
- Best case: $O(b^{m/2})$

MCTS:

- Learn from random playouts
- No evaluation needed
- Good for large state spaces

Practical AI:

- Depth limits
- Evaluation functions
- Deep learning integration

Next week: Games with uncertainty or chance elements (like dice rolls, random tile spawns, or hidden information). Expectimax and Monte Carlo Tree Search.