Adversarial Search

CSI 4106 Introduction to Artificial Intelligence

Marcel Turcotte
Version: Nov 26, 2025 10:46

Preamble

Message of the Day

https://www.youtube.com/watch?v=T-26Zv7pVW8

Could the AI bubble pop?, The Economist, 2025-11-05, 7**m** 50**s**

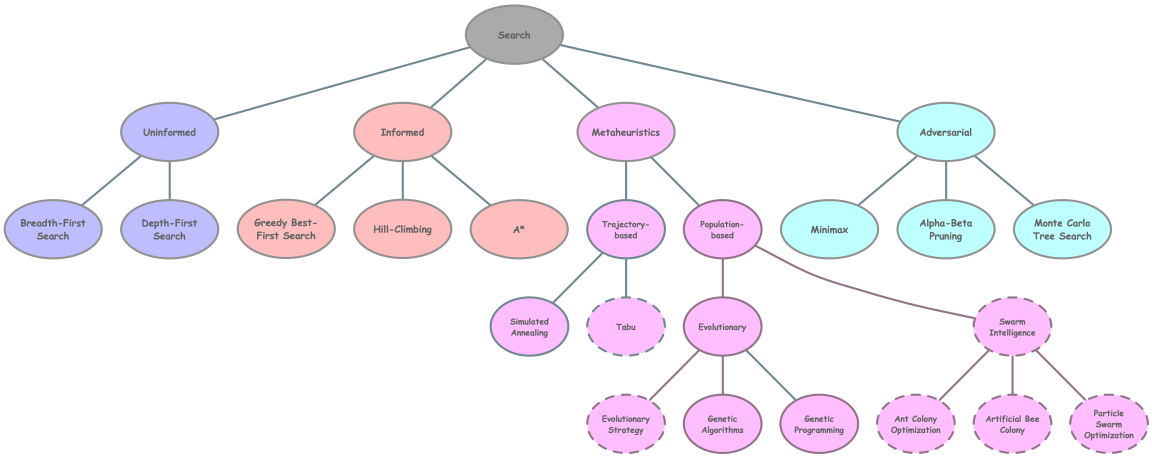
Adversarial Search

This lecture examines **competitive environments** where multiple agents have conflicting objectives, resulting in **adversarial search problems**.

Learning objectives

- **Explain** zero-sum game concepts
- **Formulate** never-lose strategies in Tic-Tac-Toe regardless of opponent moves
- **Utilize** the **minimax** algorithm to determine optimal moves in adversarial settings
- **Articulate** how **alpha-beta pruning** reduces the number of nodes evaluated without affecting outcomes

Search



Our examination of search algorithms has progressed through several stages, beginning with uninformed search techniques, advancing to informed search methods, and subsequently exploring metaheuristic approaches. We now turn our attention to the study of adversarial search algorithms.

Monte Carlo Tree Search (MCTS), while not intrinsically designed as an adversarial search algorithm, constructs a partial search tree by iteratively simulating actions from a given state. Nevertheless, when applied to two-player or multi-player games, MCTS can effectively address adversarial search problems.

Core Game Framework

Motivation

This lecture and the next will utilize **tic-tac-toe** to explore concepts in **adversarial search** and **reinforcement learning**.

We will implement and compare a range of solution strategies, including a random policy, three minimax variants, three alpha-beta pruning variants, and two Monte Carlo Tree Search algorithms.

All solvers will be developed within a general-purpose game framework designed to support a broad class of two-player games, such as Connect 4.

A broad category of classical board and combinatorial games precisely conforms to the Game interface, characterized by deterministic dynamics, alternating turns, two-player interaction, zero-sum outcomes, and perfect information. Such games are amenable to algorithmic approaches including Minimax, Alpha-Beta pruning, and Monte Carlo Tree Search (MCTS).

- Connect Four
- Gomoku

- Othello / Reversi
- Nim
- Take-the-Last-Coin
- Hex
- Chess
- Shogi
- Go

Chess, Shogi, and Go exhibit high branching factors, resulting in substantial computational demands. Consequently, their complexity renders them less appropriate as instructional examples within the scope of this course.

Game

In [2]:
```
class Game:

    """
    Abstract interface for a deterministic, 2-player, zero-sum,
    turn-taking game.

    Conventions (used by Tic-Tac-Toe and the solvers below):
    - Players are identified by strings "X" and "O".
    - evaluate(state) returns:
        > 0  if the position is good for "X"
        < 0  if the position is good for "O"
        == 0 for a draw or non-terminal equal position
    """

    def initial_state(self):

        """Return an object representing the starting position of the game."

        raise NotImplementedError

    def get_valid_moves(self, state):

        """
        Given a state, return an iterable of legal moves.
        The type of 'move' is game-dependent (e.g., (row, col) for Tic-Tac-T
        """

        raise NotImplementedError

    def make_move(self, state, move, player):

        """
        Return the successor state obtained by applying 'move' for 'player'
        to 'state'. The original state should not be modified in-place.
        """

        raise NotImplementedError
```

```
    def get_opponent(self, player):

        """Return the opponent of 'player'."""

        raise NotImplementedError

    def is_terminal(self, state):

        """
        Return True if 'state' is a terminal position (win, loss, or draw),
        False otherwise.
        """

        raise NotImplementedError

    def evaluate(self, state):

        """
        Return a scalar evaluation of 'state':
            +1 for X win, -1 for O win, 0 otherwise (for Tic-Tac-Toe).
        For other games this may be generalized, but here we keep it simple.
        """

        raise NotImplementedError

    def display(self, state):

        """Print a human-readable representation of 'state' (for debugging).

        raise NotImplementedError
```

Game

In [3]:
```
class Game:

    def initial_state(self):
        raise NotImplementedError

    def get_valid_moves(self, state):
        raise NotImplementedError

    def make_move(self, state, move, player):
        raise NotImplementedError

    def get_opponent(self, player):
        raise NotImplementedError

    def is_terminal(self, state):
        raise NotImplementedError

    def evaluate(self, state):
        raise NotImplementedError
```

```
    def display(self, state):
        raise NotImplementedError
```

Required Libraries

In [4]:
```
import math
import random
import numpy as np
import matplotlib.pyplot as plt
```

TicTacToe

In [5]:
```
class TicTacToe(Game):

    """
    Classic 3x3 Tic-Tac-Toe implementation using a NumPy array of strings.
    Empty squares are represented by " ".
    Player "X" is assumed to be the maximizing player.
    """

    def __init__(self):
        self.size = 3

    def initial_state(self):

        """Return an empty 3x3 board."""

        return np.full((self.size, self.size), " ")

    def get_valid_moves(self, state):

        """All (i, j) pairs where the board cell is empty."""

        return [
            (i, j)
            for i in range(self.size)
            for j in range(self.size)
            if state[i, j] == " "
        ]

    def make_move(self, state, move, player):

        """
        Return a new board with 'player' placed at 'move' (row, col).
        The original state is not modified.
        """

        new_state = state.copy()
        new_state[move] = player
        return new_state

    def get_opponent(self, player):
```

```
        """Swap player labels between 'X' and 'O'."""

        return "O" if player == "X" else "X"

    def is_terminal(self, state):

        """
        A state is terminal if:
        - Either player has a 3-in-a-row (evaluate != 0), or
        - There are no empty squares left (draw).
        """

        if self.evaluate(state) != 0:
            return True
        return " " not in state

    def evaluate(self, state):

        """
        Return +1 if X has three in a row, -1 if O has three in a row,
        and 0 otherwise (including non-terminal states and draws).

        This is a "game-theoretic" evaluation at terminal states; for
        non-terminal positions we simply return 0.
        """

        lines = []

        # Rows and columns
        for i in range(self.size):
            lines.append(state[i, :])   # row i
            lines.append(state[:, i])   # column i

        # Main diagonals
        lines.append(np.diag(state))
        lines.append(np.diag(np.fliplr(state)))

        # Check each line for a win
        for line in lines:
            if np.all(line == "X"):
                return 1
            if np.all(line == "O"):
                return -1
        return 0

    def display(self, state):

        """
        Visualize a Tic-Tac-Toe board using matplotlib.

        Parameters
        ----------
        state : np.ndarray of shape (size, size)
            Board containing ' ', 'X', or 'O'.
        """
```

```
        size = self.size

        fig, ax = plt.subplots()
        ax.set_aspect('equal')
        ax.set_xlim(0, size)
        ax.set_ylim(0, size)

        # Draw grid lines
        for i in range(1, size):
            ax.axhline(i, color='black')
            ax.axvline(i, color='black')

        # Hide axes completely
        ax.axis('off')

        # Draw X and O symbols
        for i in range(size):
            for j in range(size):
                cx = j + 0.5
                cy = size - i - 0.5     # invert y-axis for correct row orie

                symbol = state[i, j]

                if symbol == "X":
                    ax.plot(cx, cy, marker='x',
                            markersize=40 * (3/size),
                            color='blue',
                            markeredgewidth=3)
                elif symbol == "O":
                    circle = plt.Circle((cx, cy),
                                        radius=0.30 * (3/size),
                                        fill=False,
                                        color='red',
                                        linewidth=3)
                    ax.add_patch(circle)

        plt.show()
```

TicTacToe

In [6]:
```
class TicTacToe(Game):

    """
    Classic 3x3 Tic-Tac-Toe implementation using a NumPy array of strings.
    Empty squares are represented by " ".
    Player "X" is assumed to be the maximizing player.
    """

    def __init__(self):
        self.size = 3
```

`TicTacToe` is_a `Game`

initial_state

In [7]:
```
    def initial_state(self):

        """Return an empty 3x3 board."""

        return np.full((self.size, self.size), " ")
```

get_valid_moves

In [8]:
```
    def get_valid_moves(self, state):

        """All (i, j) pairs where the board cell is empty."""

        return [
            (i, j)
            for i in range(self.size)
            for j in range(self.size)
            if state[i, j] == " "
        ]
```

make_move

In [9]:
```
    def make_move(self, state, move, player):

        """
        Return a new board with 'player' placed at 'move' (row, col).
        The original state is not modified.
        """

        new_state = state.copy()
        new_state[move] = player

        return new_state
```

get_opponent

In [10]:
```
    def get_opponent(self, player):

        """Swap player labels between 'X' and 'O'."""

        return "O" if player == "X" else "X"
```

evaluate

In [11]:
```python
def evaluate(self, state):

    lines = []

    # Rows and columns
    for i in range(self.size):
        lines.append(state[i, :])   # row i
        lines.append(state[:, i])   # column i

    # Main diagonals
    lines.append(np.diag(state))
    lines.append(np.diag(np.fliplr(state)))

    # Check each line for a win
    for line in lines:
        if np.all(line == "X"):
            return 1
        if np.all(line == "O"):
            return -1
    return 0
```

is_terminal

In [12]:
```python
def is_terminal(self, state):

    """
    A state is terminal if:
    - Either player has a 3-in-a-row (evaluate != 0), or
    - There are no empty squares left (draw).
    """

    if self.evaluate(state) != 0:
        return True
    return " " not in state
```

display

In [13]:
```python
def display(self, state):

    """
    Visualize a Tic-Tac-Toe board using matplotlib.

    Parameters
    ----------
    state : np.ndarray of shape (size, size)
        Board containing ' ', 'X', or 'O'.
    """

    size = self.size

    fig, ax = plt.subplots()
```

```
        ax.set_aspect('equal')
        ax.set_xlim(0, size)
        ax.set_ylim(0, size)

        # Draw grid lines
        for i in range(1, size):
            ax.axhline(i, color='black')
            ax.axvline(i, color='black')

        # Hide axes completely
        ax.axis('off')

        # Draw X and O symbols
        for i in range(size):
            for j in range(size):
                cx = j + 0.5
                cy = size - i - 0.5     # invert y-axis for correct row orie

                symbol = state[i, j]

                if symbol == "X":
                    ax.plot(cx, cy, marker='x',
                            markersize=40 * (3/size),
                            color='blue',
                            markeredgewidth=3)
                elif symbol == "O":
                    circle = plt.Circle((cx, cy),
                                        radius=0.30 * (3/size),
                                        fill=False,
                                        color='red',
                                        linewidth=3)
                    ax.add_patch(circle)

        plt.show()
```

Solver

In [14]:
```
class Solver:

    """
    Base class for all solvers (Random, Minimax, AlphaBeta, MCTS, etc.).

    Solvers must implement:
        - select_move(game, state, player)

    Solvers may optionally implement:
        - reset()           : called at the start of each game
        - opponent_played() : used by persistent solvers (e.g., MCTS)

    Notes
    -----
    • Solvers may keep internal state that persists across moves.
    • GameRunner may call reset() automatically before every match.
    """
```

```
    def select_move(self, game, state, player):

        """
        Must be implemented by subclasses.
        Returns a legal move for the given player.
        """

        raise NotImplementedError

    def get_name(self):

        """
        Return the solver's name for reporting, logging, or tournament resul

        The default returns the class name, but solvers may override
        to include parameters (e.g., "MCTS(num_simulations=500)").
        """

        return self.__class__.__name__

    def opponent_played(self, move):
        """
        Optional. Called after the opponent moves.
        Useful for stateful solvers like MCTS.
        Stateless solvers can ignore it.
        """
        pass

    def reset(self):

        """
        Optional. Called once at the beginning of each game.
        Override only if the solver maintains internal state
        (e.g., MCTS tree, cached analysis, heuristic tables).
        """

        pass
```

Solver

In [15]:
```python
class Solver:

    def select_move(self, game, state, player):
        raise NotImplementedError

    def opponent_played(self, move):
        pass

    def reset(self):
        pass

    def get_name(self):
        return self.__class__.__name__
```

RandomSolver

In [16]:
```python
class RandomSolver(Solver):

    """
    A simple baseline solver:
    - At each move, chooses uniformly at random among all legal moves.
    - Does not maintain any internal state (no learning).
    """

    def __init__(self, seed=None):
        self.rng = random.Random(seed)

    def select_move(self, game, state, player):

        """Return a random legal move for the current player."""

        moves = game.get_valid_moves(state)

        return self.rng.choice(moves)

    def opponent_played(self, move):

        """Random solver has no internal state to update."""

        pass
```

In [17]:
```python
class RandomSolver(Solver):

    def __init__(self, seed=None):
        self.rng = random.Random(seed)

    def select_move(self, game, state, player):
        moves = game.get_valid_moves(state)
        return self.rng.choice(moves)
```

Executes **randomized** gameplay with **reproducible** outcomes.

General and **applicable** to a broad class of games beyond tic-tac-toe.

GameRunner

In [18]:
```
class GameRunner:

    """
    Utility to run a single game between two solvers on a given Game.

    This class is deliberately simple: it alternates moves between "X" and "
    until a terminal state is reached.
    """

    def __init__(self, game, verbose=False):
        self.game = game
        self.verbose = verbose

    def play_game(self, solver_X, solver_O):

        """
        Play one full game:
        - solver_X controls player "X"
        - solver_O controls player "O"

        Returns
        -------
        result : int
            +1 if X wins, -1 if O wins, 0 for a draw.
        """

        state = self.game.initial_state()
        player = "X"
        solvers = {"X": solver_X, "O": solver_O}

        # Play until terminal position
        while not self.game.is_terminal(state):
            # Current player selects a move
            move = solvers[player].select_move(self.game, state, player)

            # Apply the move
            state = self.game.make_move(state, move, player)

            if self.verbose:
                self.game.display(state)

            # Notify the opponent (for persistent solvers like MCTS)
            opp = self.game.get_opponent(player)
            solvers[opp].opponent_played(move)

            # Switch active player
            player = opp

        if self.verbose:
            print(self.game.evaluate(state), "\n")
```

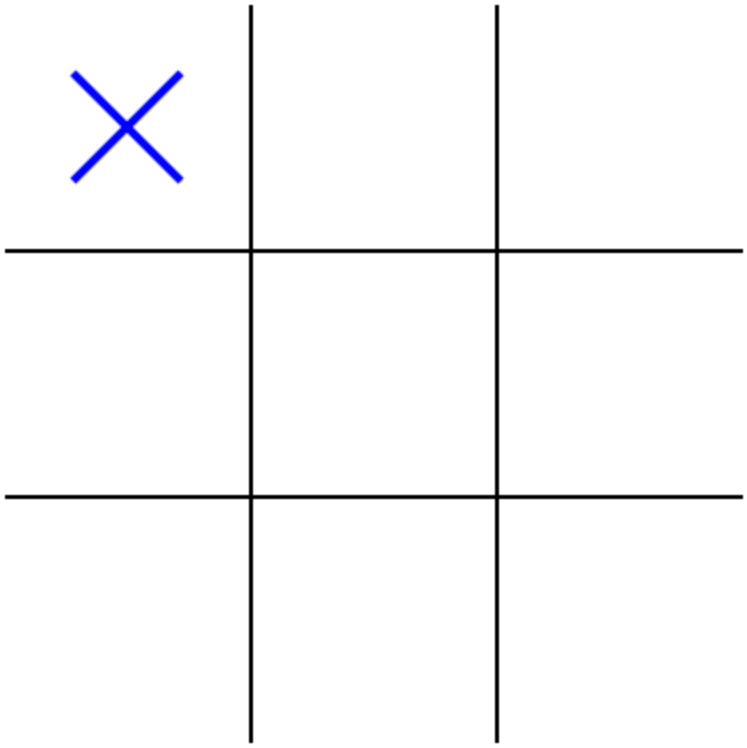
```
        # Final evaluation from X's perspective
        return self.game.evaluate(state)
```

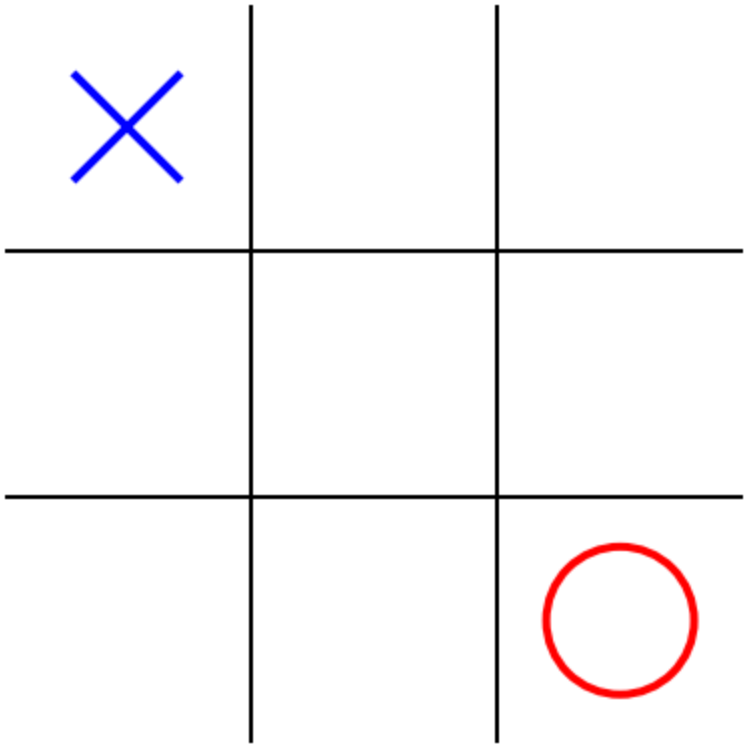
GameRunner

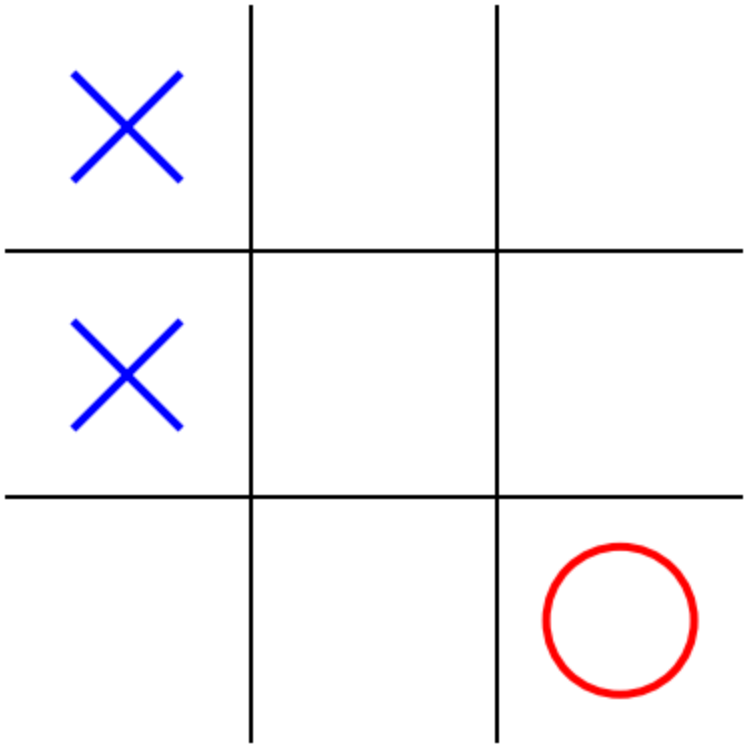
In [19]:
```
class GameRunner:

    def __init__(self, game):
        self.game = game

    def play_game(self, solver_X, solver_O):

        state = self.game.initial_state()
        player = "X"
        solvers = {"X": solver_X, "O": solver_O}

        while not self.game.is_terminal(state):
            move = solvers[player].select_move(self.game, state, player)
            state = self.game.make_move(state, move, player)
            opp = self.game.get_opponent(player)
            solvers[opp].opponent_played(move)
            player = opp

        return self.game.evaluate(state)
```

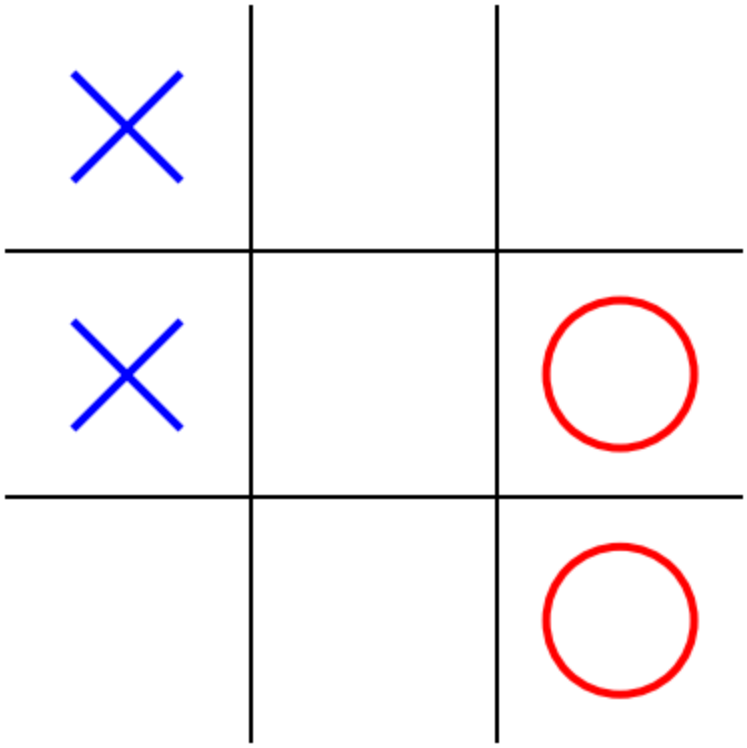
Example

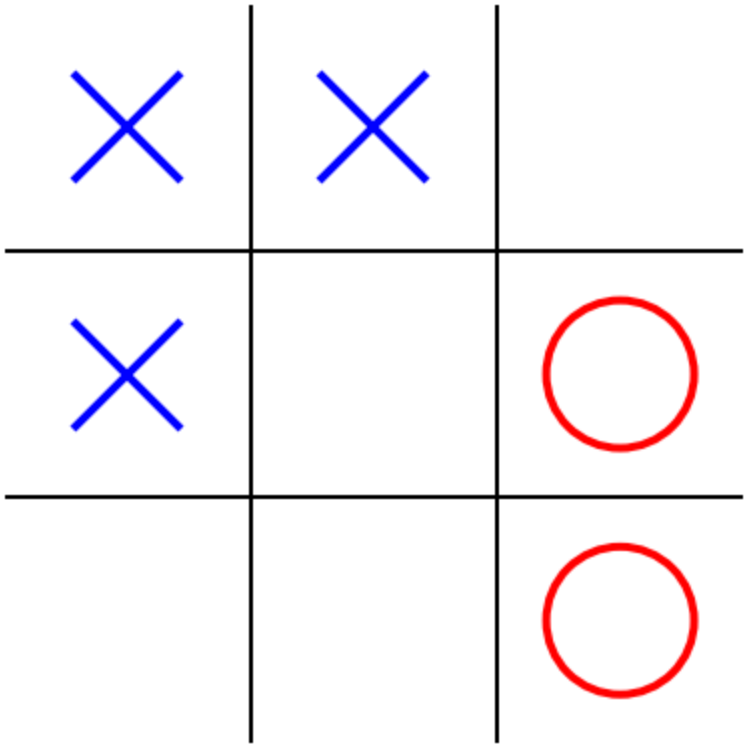
In [20]:
```
game = TicTacToe()

runner = GameRunner(game, verbose=True)

a = RandomSolver(123)
b = RandomSolver(456)

outcome = runner.play_game(a, b)
```

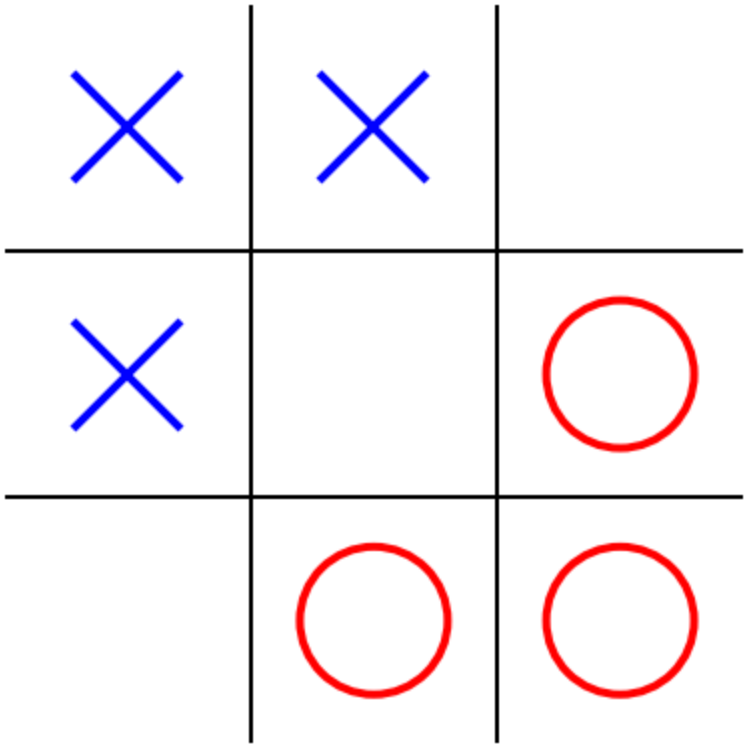


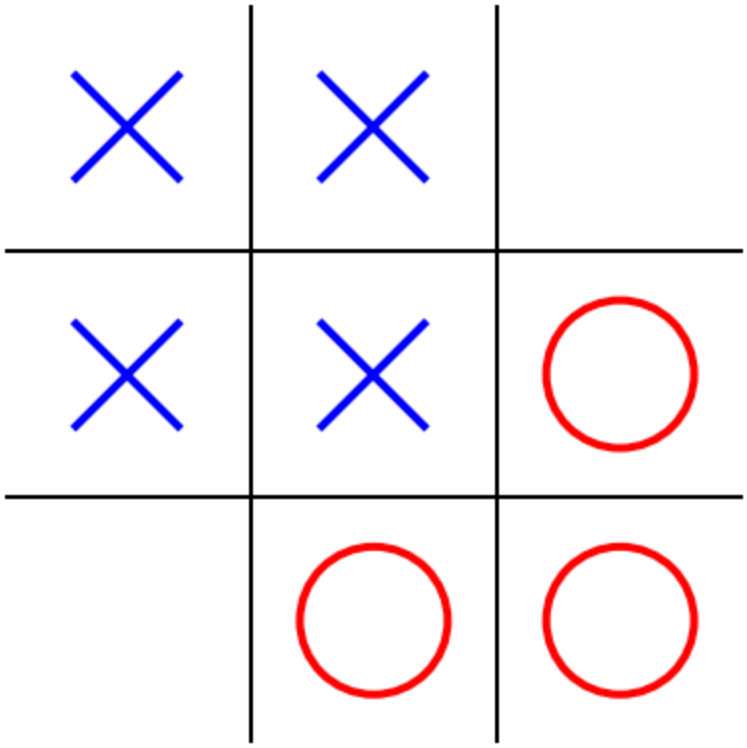


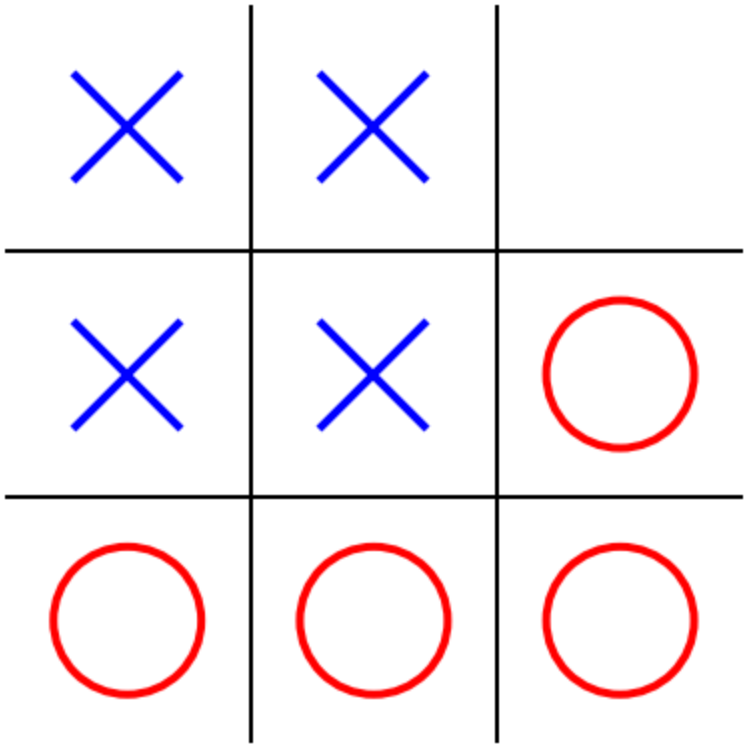












-1

evaluate_solvers

In [21]:
```
def evaluate_solvers(game, solver_X, solver_O, num_games, verbose=False):
    """
    Evaluate two solvers head-to-head on a given game.
```

```
    Parameters
    ----------
    game      : Game
        An instance of a Game (e.g., TicTacToe).
    solver_X  : Solver
        Solver controlling player "X" (the maximizing player).
    solver_O  : Solver
        Solver controlling player "O" (the minimizing player).
    num_games : int
        Number of games to play with these fixed roles.

    Notes
    -----
    - The same solver instances are reused across games.
      This allows *persistent* solvers (e.g., MCTS) to accumulate
      experience across games.
    - Outcomes are interpreted from X's perspective:
        +1 -> X wins
        -1 -> O wins
         0 -> draw
    """

    runner = GameRunner(game)

    # Aggregate statistics over all games
    results = {
        "X_wins": 0,
        "O_wins": 0,
        "draws": 0,
    }

    for i in range(num_games):
        # Play one game with solver_X as "X" and solver_O as "O"
        outcome = runner.play_game(solver_X, solver_O)

        # Update counters based on outcome (+1, -1, or 0)
        if outcome == 1:
            results["X_wins"] += 1
            if verbose:
                print(f"Game {i + 1}: X wins")
        elif outcome == -1:
            results["O_wins"] += 1
            if verbose:
                print(f"Game {i + 1}: O wins")
        else:
            results["draws"] += 1
            if verbose:
                print(f"Game {i + 1}: Draw")

    # Print final summary
    if verbose:
        print(f"\nAfter {num_games} games:")
        print(f"  X ({solver_X.get_name()}) wins: {results['X_wins']}")
        print(f"  O ({solver_O.get_name()}) wins: {results['O_wins']}")
        print(f"  Draws: {results['draws']}")
```

```
    return results
```

The method conducts a specified number of complete game runs, utilizing the designated solving algorithms for each run. It systematically records the results of these games and reports the aggregated performance metrics.

Can You Guess the Outcome?

In [22]:
```
game = TicTacToe()

a = RandomSolver(7)
b = RandomSolver(42)

results = evaluate_solvers(game, a, b, num_games=1000)

results
```

```
{'X_wins': 581, 'O_wins': 290, 'draws': 129}
```

X_wins: $\frac{1}{3}$, O_wins: $\frac{1}{3}$, draws: $\frac{1}{3}$ or X_wins: $\frac{2}{5}$, O_wins: $\frac{2}{5}$, draws: $\frac{1}{5}$ or something else?

Is this outcome consistent with your expectations? Please provide a rationale. We will revisit this result in detail shortly.

Introduction

Types of Games

- **Deterministic** or **stochastic**
- **One**, **two**, or **more** players
- **Zero-sum** or not
- **Perfect information** or not

Definition

Zero-sum games are competitive scenarios where one player's **gain** is exactly balanced by another player's **loss**, **resulting in a net change of zero** in total wealth or benefit.

Tic-tac-toe is a zero-sum game.

Have you previously studied zero-sum games? If so, please specify the context (e.g., coursework, research, or applied work) and the level of treatment (introductory, advanced, or applied).

Deterministic Games

- **States**: S (S_0 to S_k)
- **Players**: $P = 1, N$
- **Actions**: A (depends on P and S)
- **Transition function**: $S \times A \to S$
- **A final state**: S_{final}
- **Reward or utility**: S_{final}, p

Develop a policy $S_0 \to S_{\text{final}}$.

Unlike the earlier state-space search formulations, we introduce an explicit player parameter. Consequently, the transition function is conditioned on both the current state and the player identity (rather than on state alone).

What is a **policy**? How does it differ from the **transition** function?

The **transition function** describes the game's mechanics.

$$T : S \times A \to S$$

It is part of the *game definition*.

It tells you **what happens** when you take action a in state s:

- It is fixed by the rules of the game.
- It does **not** depend on how a particular player wants to play.
- Example (Tic-Tac-Toe): placing an X in an empty cell updates the board accordingly.

The **policy** describes the player's behaviour.

$$\pi : S \to A$$

It is part of the *solution*.

It tells the player **which action to choose** in each state they may encounter.

- It depends on strategy, preferences, optimality.
- It uses the transition function, but it is not defined by it.
- Example: "If the centre is free, play there; otherwise play a corner."

Concept	What it describes	Fixed or chosen?	Example
Transition function T	How states evolve when an action is taken	**Fixed by game rules**	Applying a move updates the board
Policy π	Which action the player selects in each state	**Chosen (e.g., optimal strategy)**	Always take center if possible

What do you think?

- Consider playing **tic-tac-toe**.
- Can you ensure **a never-lose strategy**, irrespective of your opponent's moves?

Does it matter if you play first or second?

Anyone who has played tic-tac-toe understands that a player can adopt a strategy that ensures they never lose.

In itself this statement is rather surprising. Why?

How many board configurations are there?

- Each square can be occupied by X or O or be empty. This leads to $3^9 = 19,683$ board configurations.
- Are all these configurations possible in a valid game?

Game progression.

- Not all of these configurations are valid game states because they may contain impossible numbers of Xs and Os or may not follow the rules of the game.
- A tic-tac-toe game can end in a win for either player or a draw. The longest game without a winner involves 9 moves (a full board).
- Valid games account for the rules that players alternate turns, starting with X.
- How many valid sequences of moves are there?

Are there more valid sequences of moves than board configurations?

`count_valid_sequences`

In [23]:
```
def count_valid_sequences(game, state, player):

    if game.is_terminal(state):
      return 1

    valid_moves = game.get_valid_moves(state)

    total = 0
    for move in valid_moves:
        new_state = game.make_move(state, move, player)
        total += count_valid_sequences(game, new_state, game.get_opponent(pl

    return total
```

In [24]:
```
game = TicTacToe()
state = game.initial_state()
player = "X"
```

```
total = count_valid_sequences(game, state, player)
print(f"The total number of valid sequences is: {total:,}")
```

```
The total number of valid sequences is: 255,168
```

Considering the number of valid sequences, it is unsurprising that the first player can adopt a strategy to avoid losing?

Symmetry (Digression)

. . .

Tic-tac-toe has **8 symmetrical transformations** (4 rotations and 4 reflections).

By considering these, many game sequences that are different in raw move order become equivalent.

The number of **unique sequences of moves** is **26,830**, whereas the number of unique board positions is **765**.

Exercise: write a Python program the confirm the above.

Search Tree

The **search tree size** for the tic-tac-toe game is **relatively small**, making it suitable for use as a running example in later discussions.

How does this compare to the search trees for chess and Go?

Search Tree

- **Chess**: $35^{80} \sim 10^{123}$
- **Go**: $361! \sim 10^{768}$

The estimated **number of atoms in the observable universe** is around 10^{78} to 10^{82}.

Chess:

Chess has a relatively small board (8×8) but a variety of pieces with different movement capabilities.

On average, each position has about 35 legal moves.

Considering an average game length of 80 ply (a ply is a half-move, so 40 moves per player), the total games are estimated using the formula: $35^{80} \sim 10^{123}$, adjusted down to account for illegal and redundant positions.

Go:

Go is played on a 19×19 board, providing 361 points where stones can be placed. Accordingly, the total number of games is $361! \sim 10^{768}$.

Definition

Optimal play involves executing the **best possible move at each step** to maximize winning chances, assuming the opponent also plays optimally.

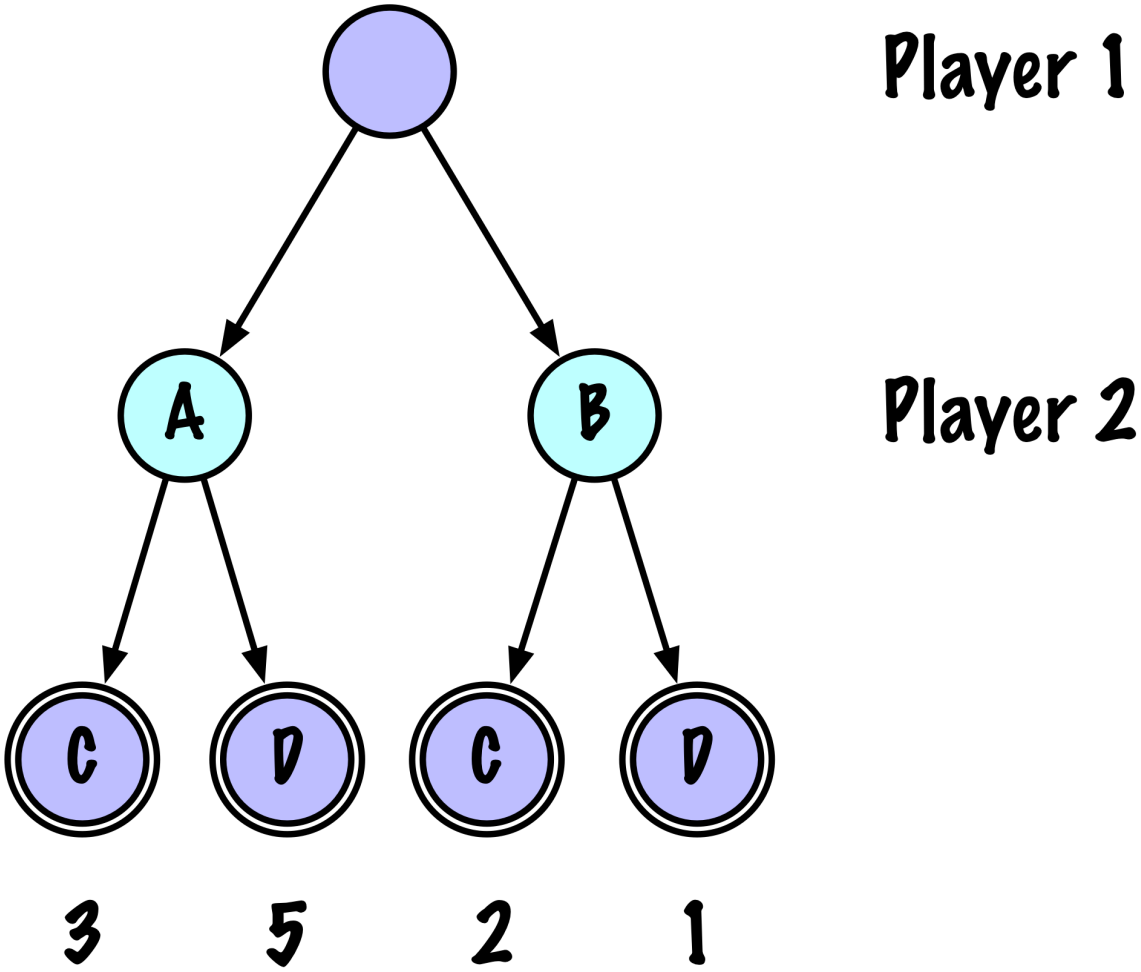
In **perfect information games** like tic-tac-toe or chess, it requires anticipating the opponent's moves and choosing actions that enhance one's position or minimize losses.

When **both players employ optimal strategies**, the outcome—win, loss, or draw—is dictated by the game's inherent mechanics and initial conditions.

Perfect information refers to a feature of certain games or decision-making scenarios where all players have complete and accurate knowledge of the entire game state at all times. This includes full visibility of all actions taken previously and no hidden elements or randomness affecting the game's progression. In games with perfect information, such as chess or tic-tac-toe, players can make fully informed decisions based on the entire history and current status of the game, allowing for strategies that can be planned several moves ahead.

Make sure to fully understand how optimality is defined here. It might not correspond to your intuition. The algorithms to be developed next are based on this definition.

Two-Move Game



Two-Move is a hypothetical game involving two players, designed to facilitate discussions on the minimax algorithm.

Game Setup

- The **game starts** with a single decision point for Player 1, who has two possible moves: A and B.
- Each of these moves leads to a **decision point for Player 2**, who also has two possible responses: C and D.
- The **game ends** after Player 2's move, resulting in a terminal state with predefined scores.

Search Tree

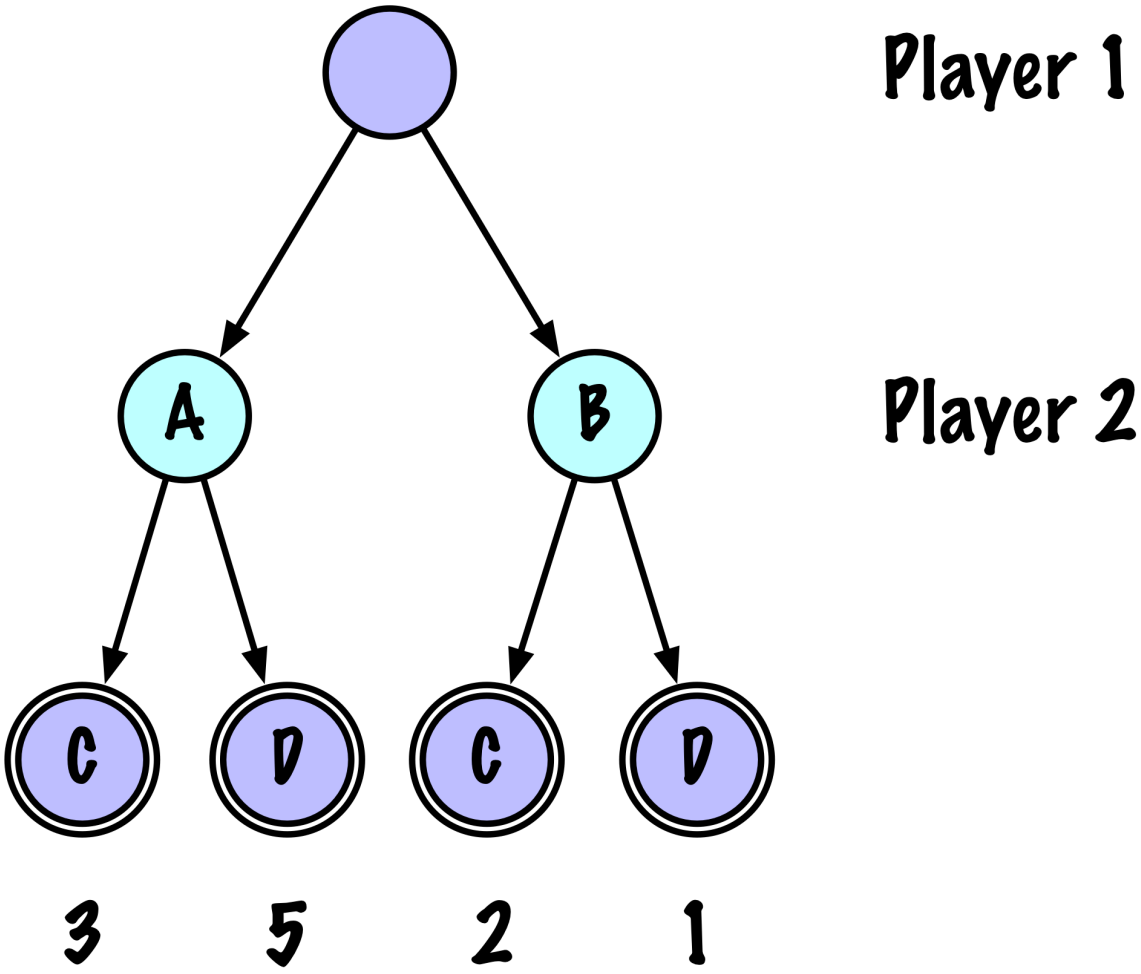
- **Root Node**: Represents the initial state before Player 1's move.
- **Ply 1**: Player 1 chooses between moves A and B.
- **Ply 2**: For each of Player 1's moves, Player 2 chooses between moves C and D.
- **Leaf Nodes**: Each branch's endpoint is a terminal state with an associated score.

Scores

- (A, C) results in a score of 3.
- (A, D) results in a score of 5.
- (B, C) results in a score of 2.
- (B, D) results in a score of 1.

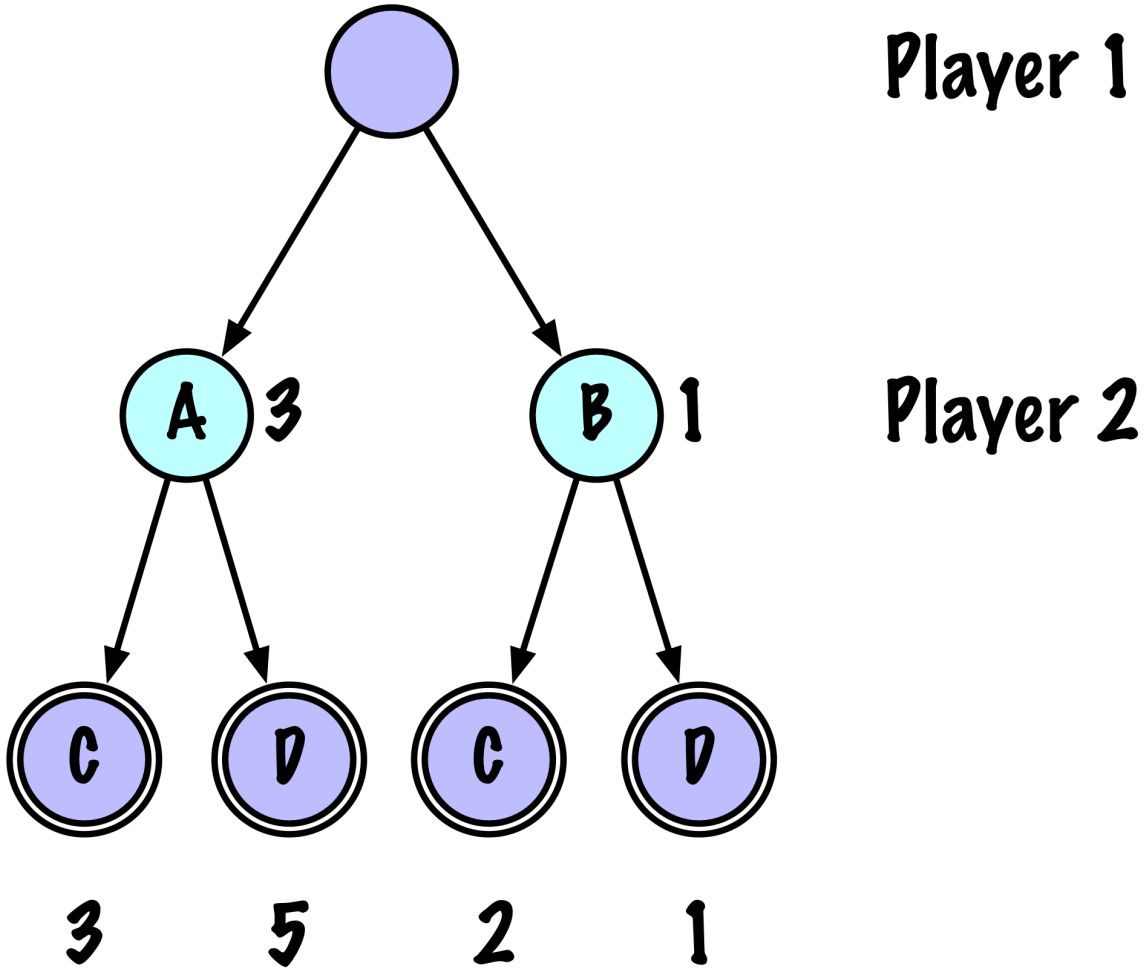
Player 1 wants to maximize its score.

Strategy



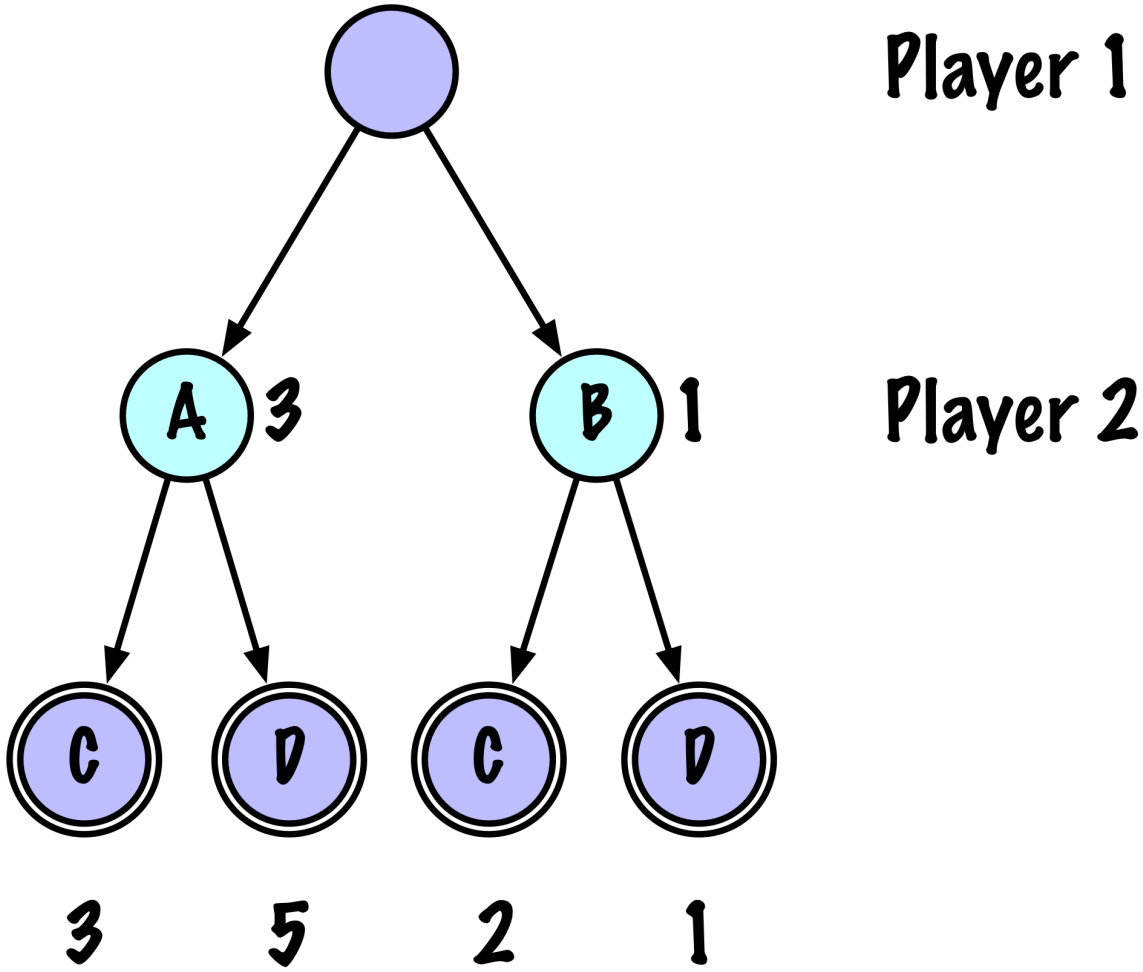
What should be **player 2's strategy** and **why**?

Strategy



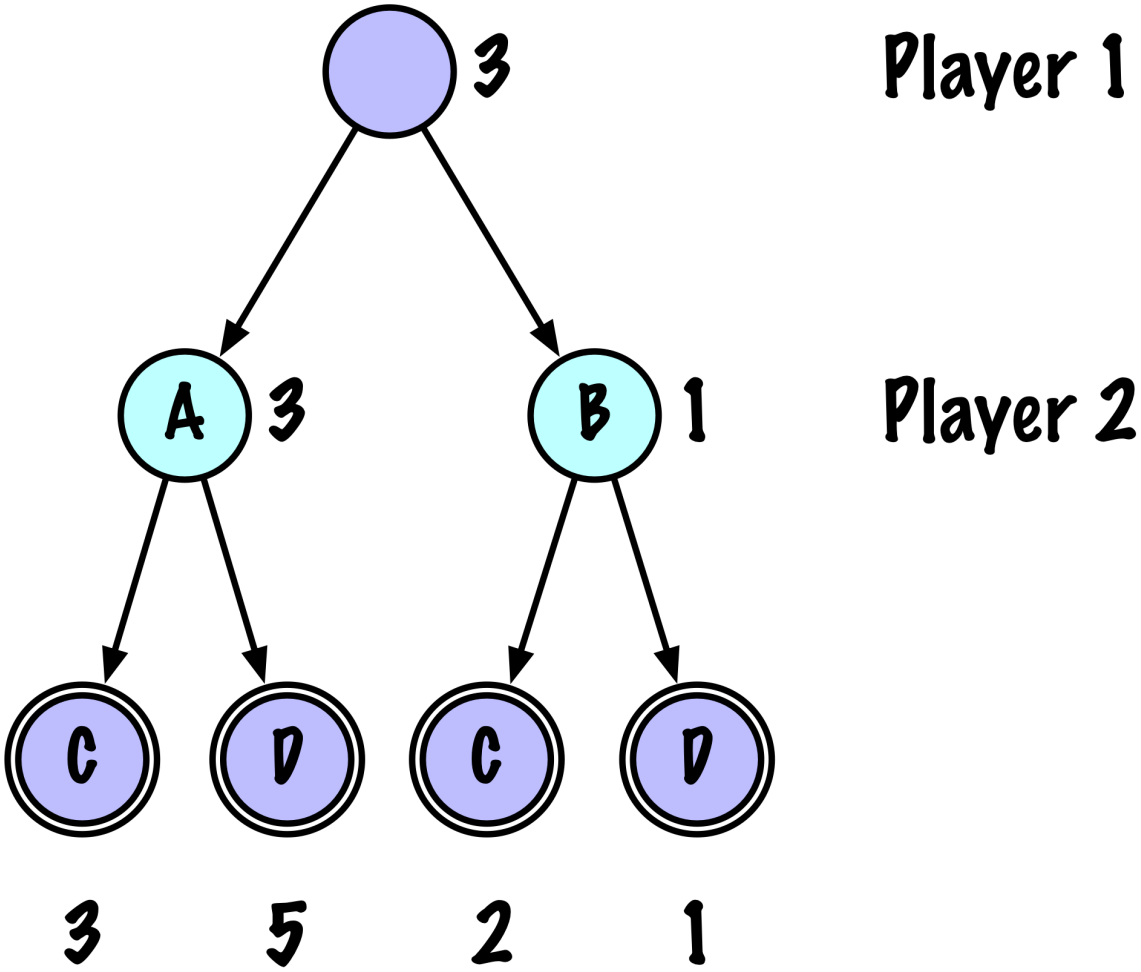
- **For move** A:
  - Player 2 can choose C (score = 3) or D (score = 5); they choose C (minimizing to 3).
- **For move** B:
  - Player 2 can choose C (score = 2) or D (score = 1); they choose D (minimizing to 1).

Strategy



What should now be the strategy for **Player 1**?

Strategy



Player 1, being the **maximizer**, will choose move A, as it leads to the higher score of 3 after Player 2 minimizes.

Minimax

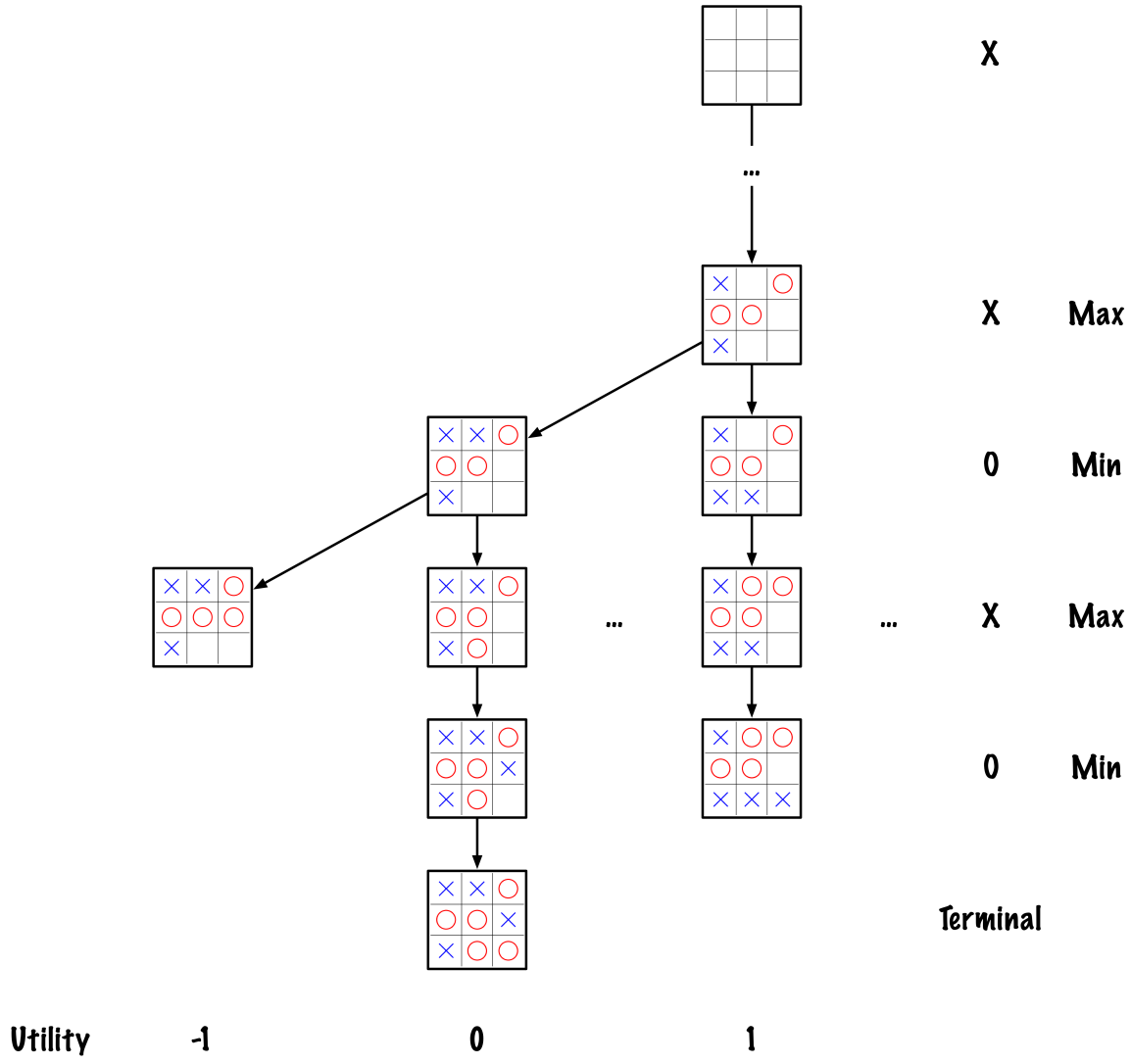
Minimax

- **Player 1** is the maximizing player, seeking the highest score.
- **Player 2** is the minimizing player, seeking the lowest score.

Evaluation:

- Player 2 evaluates the potential outcomes for each of their moves and chooses the **least favorable outcome** for Player 1.
- Player 1 then evaluates these outcomes, choosing the move that **maximizes their minimum guaranteed score**.

Minimax Search



Minimax Search

The **minimax algorithm** operates by exploring **all possible moves in a game tree**, evaluating the outcomes to **minimize the possible loss for a worst-case scenario**. At each node:

- **Maximizing Player's Turn:** Choose the move with the highest possible value.
- **Minimizing Player's Turn:** Choose the move with the lowest possible value.

By backtracking from the terminal nodes to the root, the algorithm selects the move that **maximizes the player's minimum gain**, effectively anticipating and **countering the opponent's best strategies**.

Minimax Search

function MINIMAX-SEARCH(*game*, *state*) **returns** *an action*
  player ← *game*.TO-MOVE(*state*)
  value, *move* ← MAX-VALUE(*game*, *state*)
  return *move*

function MAX-VALUE(*game*, *state*) **returns** a (*utility*, *move*) pair
  if *game*.IS-TERMINAL(*state*) **then return** *game*.UTILITY(*state*, *player*), *null*
  $v \leftarrow -\infty$
  for each *a* **in** *game*.ACTIONS(*state*) **do**
    v2, *a2* ← MIN-VALUE(*game*, *game*.RESULT(*state*, *a*))
    if $v2 > v$ **then**
      v, *move* ← *v2*, *a*
  return v, *move*

function MIN-VALUE(*game*, *state*) **returns** a (*utility*, *move*) pair
  if *game*.IS-TERMINAL(*state*) **then return** *game*.UTILITY(*state*, *player*), *null*
  $v \leftarrow +\infty$
  for each *a* **in** *game*.ACTIONS(*state*) **do**
    v2, *a2* ← MAX-VALUE(*game*, *game*.RESULT(*state*, *a*))
    if $v2 < v$ **then**
      v, *move* ← *v2*, *a*
  return v, *move*

Attribution: (Russell and Norvig 2020, fig. 5.3)

Walkthrough (first 4 minutes)

https://youtu.be/l-hh51ncgDI

Attribution: Sebastian Lague

Watch the first 5m 20s of the video.

The video introduces a variant of the **minimax** algorithm that includes a **maximum depth parameter**, enabling the algorithm to terminate at a user-specified depth. This approach is essential for games with extensive search spaces, such as chess and Go. However, it necessitates a reliable method for evaluating the current state, referred to as static evaluation in the video.

`MinimaxSolverV1`

In [25]:
```python
class MinimaxSolverV1(Solver):

    """
    A classic, exact Minimax solver for Tic-Tac-Toe.
```

```
    Key properties:
    - Assumes that "X" is the maximizing player.
    - Performs a full game-tree search (Tic-Tac-Toe is small enough).
    """

    # ------------------------------------------------------------------
    # PUBLIC API — Solver interface
    # ------------------------------------------------------------------

    def select_move(self, game, state, player):

        """
        Return the optimal move for `player` using the full minimax search.

        Notes
        -----
        - Because Tic-Tac-Toe is small, a full search is instantaneous.
        - X is always maximizing; O is always minimizing.
        """

        maximizing = (player == "X")
        _, move = self._minimax(game, state, player, maximizing)
        return move

    # ------------------------------------------------------------------
    # CORE MINIMAX
    # ------------------------------------------------------------------

    def _minimax(self, game, state, player, maximizing):

        """
        The core recursive minimax procedure.

        Parameters
        ----------
        game       : the TicTacToe instance
        state      : NumPy array representing the current board
        player     : the player to move ("X" or "O")
        maximizing : True if this node corresponds to the maximizing player
                     (i.e., X must play optimally), False for minimizing (O)

        Returns
        -------
        (value, move)
            value : +1, 0, or -1 from the perspective of X
            move  : the best move found at this node
        """

        # ----------------------------------------------------------
        # TERMINAL TEST
        # ----------------------------------------------------------

        if game.is_terminal(state):
            # game.evaluate() returns 1 for X win, -1 for O win, 0 otherwise
            return game.evaluate(state), None
```

```
        # ------------------------------------------------------------
        # GENERATE ALL LEGAL MOVES
        # ------------------------------------------------------------

        moves = game.get_valid_moves(state)

        # ------------------------------------------------------------
        # MAX NODE: X to play
        # ------------------------------------------------------------

        if maximizing:
            best_value = -math.inf
            best_move = None

            for move in moves:
                next_state = game.make_move(state, move, player)
                # After X plays, it's O's turn (minimizing)
                value, _ = self._minimax(
                    game,
                    next_state,
                    game.get_opponent(player),
                    maximizing=False
                )
                if value > best_value:
                    best_value = value
                    best_move = move

            return best_value, best_move

        # ------------------------------------------------------------
        # MIN NODE: O to play
        # ------------------------------------------------------------

        else:
            best_value = math.inf
            best_move = None

            for move in moves:
                next_state = game.make_move(state, move, player)
                # After O plays, it's X's turn (maximizing)
                value, _ = self._minimax(
                    game,
                    next_state,
                    game.get_opponent(player),
                    maximizing=True
                )
                if value < best_value:
                    best_value = value
                    best_move = move

            return best_value, best_move
```

`MinimaxSolverV1`

In [26]:
```python
class MinimaxSolverV1(Solver):

    def select_move(self, game, state, player):

        maximizing = (player == "X")

        _, move = self._minimax(game, state, player, maximizing)

        return move
```

`MinimaxSolverV1` is a `Solver`.

It executes the minimax algorithm from `state` to determine the optimal move for `player`.

`_minimax`

In [27]:
```python
    def _minimax(self, game, state, player, maximizing):

        if game.is_terminal(state):
            return game.evaluate(state), None

        moves = game.get_valid_moves(state)
```

`_minimax`: if maximizing

In [28]:
```python
        if maximizing:
            best_value = -math.inf
            best_move = None

            for move in moves:
                next_state = game.make_move(state, move, player)
                value, _ = self._minimax(
                    game,
                    next_state,
                    game.get_opponent(player),
                    maximizing=False
                )
                if value > best_value:
                    best_value = value
                    best_move = move

            return best_value, best_move
```

`_minimax`: if minimizing

In [29]:
```
        else:
            best_value = math.inf
            best_move = None

            for move in moves:
                next_state = game.make_move(state, move, player)
                value, _ = self._minimax(
                    game,
                    next_state,
                    game.get_opponent(player),
                    maximizing=True
                )
                if value < best_value:
                    best_value = value
                    best_move = move

            return best_value, best_move
```

Execution (1/2)

In [30]:
```
import time

# Record the start time
start_time = time.perf_counter()
```

In [31]:
```
a = RandomSolver(13)
b = MinimaxSolverV1()

results = evaluate_solvers(game, a, b, num_games=1)
results
```

```
{'X_wins': 0, 'O_wins': 0, 'draws': 1}
```

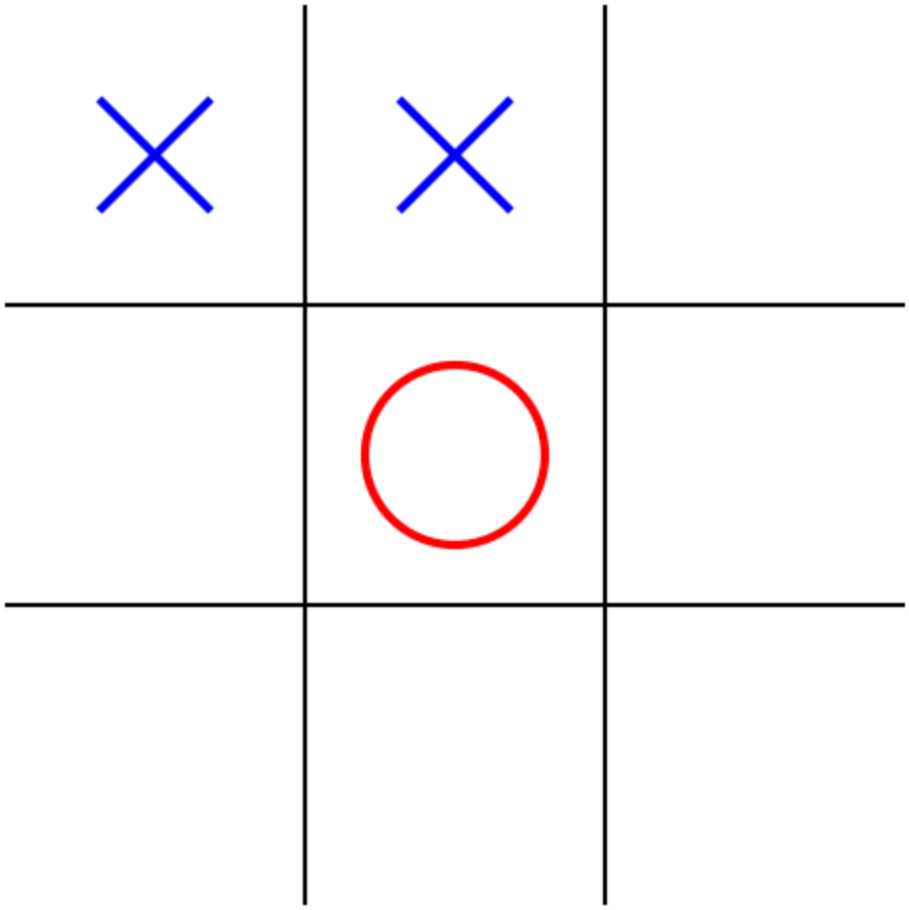
In [32]:
```
# Record the end time
end_time = time.perf_counter()

# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time in seconds
print(f"Elapsed time: {elapsed_time:.6f} seconds!")
```

```
Elapsed time: 2.257900 seconds!
```

Discussion (Digression)

- Is `test_tic_tac_toe` faster or slower than expected?
- Do you see an area for improvement?

Discussion (Digression)



Memoization (Caching)

In [33]:
```python
from functools import lru_cache

def canonical(state):

    """
    Convert a NumPy array board into a hashable, immutable representation
    (tuple of tuples). This allows us to use it as a key in dicts or
    as an argument to lru_cache. MCTS can also reuse this representation.
    """

    return tuple(map(tuple, state))
```

`MinimaxSolver`

In [34]:
```python
class MinimaxSolver(Solver):

    """
    A classic, exact Minimax solver for Tic-Tac-Toe.

    - Assumes "X" is the maximizing player.
```

```
    - Uses memoization (lru_cache) to avoid recomputing values for
      identical positions.
    """

    def select_move(self, game, state, player):

        """
        Public interface: choose the best move for 'player' using Minimax.
        For Tic-Tac-Toe we can safely search the full game tree.
        """

        # Store game on self so _minimax can use it
        self.game = game

        # From X's perspective: X is maximizing, O is minimizing
        maximizing = (player == "X")

        # For Tic-Tac-Toe, depth=9 is enough to cover all remaining moves.
        _, move = self._minimax(canonical(state), player, maximizing, 9)
        return move

    @lru_cache(maxsize=None)
    def _minimax(self, state_key, player, maximizing, depth):

        """
        Internal recursive minimax.

        Parameters
        ----------
        state_key : hashable representation of the board (tuple of tuples)
        player    : player to move at this node ("X" or "O")
        maximizing: True if this node is a 'max' node (X to move),
                    False if this is a 'min' node (O to move)
        depth     : remaining search depth (not used for cutoffs in this
                    full-search Tic-Tac-Toe implementation, but kept for
                    didactic purposes and easy extension).
        """

        # Recover the NumPy board from the canonical state_key
        state = np.array(state_key)

        # Terminal test: win, loss, or draw
        if self.game.is_terminal(state):
            # Evaluation is always from X's perspective: +1, -1, or 0
            return self.game.evaluate(state), None

        moves = self.game.get_valid_moves(state)
        best_move = None

        if maximizing:
            # X to move: maximize the evaluation
            best_val = -math.inf
            for move in moves:
                st2 = self.game.make_move(state, move, player)
                val, _ = self._minimax(
                    canonical(st2),
```

```
                    self.game.get_opponent(player),
                    False,
                    depth - 1
                )
                if val > best_val:
                    best_val = val
                    best_move = move
            return best_val, best_move

        else:
            # O to move: minimize the evaluation (since evaluation is for X)
            best_val = math.inf
            for move in moves:
                st2 = self.game.make_move(state, move, player)
                val, _ = self._minimax(
                    canonical(st2),
                    self.game.get_opponent(player),
                    True,
                    depth - 1
                )
                if val < best_val:
                    best_val = val
                    best_move = move
            return best_val, best_move
```

Execution (2/2)

In [35]:
```
import time

# Record the start time
start_time = time.perf_counter()
```

In [36]:
```
a = RandomSolver(13)
b = MinimaxSolver()

results = evaluate_solvers(game, a, b, num_games=1)
results
```

```
{'X_wins': 0, 'O_wins': 0, 'draws': 1}
```

In [37]:
```
# Record the end time
end_time = time.perf_counter()

# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time in seconds
print(f"Elapsed time: {elapsed_time:.6f} seconds!")
```

```
Elapsed time: 0.110848 seconds!
```

Yet Another Digression

In [38]:
```
    def get_valid_moves(self, state):

        moves = [
            (i, j)
            for i in range(self.size)
            for j in range(self.size)
            if state[i, j] == " "
        ]

        return random.shuffle(moves)
```

Games can become monotonous if you quickly discern patterns in your opponent's strategy, such as consistently choosing moves in a specific sequence.

We are randomizing the order of moves prior to returning them. However, the optimal move for a given configuration will remain fixed, if solutions are cached.

Exploration

- Compare the reduction in execution time achieved through **symmetry considerations** versus **caching techniques**. Evaluate the **combined effect of both approaches**.
- Develop a **Connect Four** game implementation employing a **minimax search algorithm**.
- Connect Four is **symmetric across its vertical axis**. Develop a new implementation that leverages this symmetry.

See also: Connect 4: Principles and Techniques

Can You Guess the Outcome?

In [39]:
```
game = TicTacToe()

a = MinimaxSolver()
b = RandomSolver(7)

results = evaluate_solvers(game, a, b, num_games=1000)

results
```

```
{'X_wins': 998, 'O_wins': 0, 'draws': 2}
```

Can You Guess the Outcome?

In [40]:
```
game = TicTacToe()
```

```
a = RandomSolver(7)
b = MinimaxSolver()

results = evaluate_solvers(game, a, b, num_games=1000)

results
```

```
{'X_wins': 0, 'O_wins': 793, 'draws': 207}
```

Can You Guess the Outcome?

In [41]:
```
game = TicTacToe()

a = MinimaxSolver()
b = MinimaxSolver()

results = evaluate_solvers(game, a, b, num_games=1000)

results
```

```
{'X_wins': 0, 'O_wins': 0, 'draws': 1000}
```

Summary

What have we obtained?

. . .

We derived a **policy** (a decision algorithm) that recommends actions leading to the game's **optimal solution**, under the assumption that the **opponent adopts the same policy**.

Remark

The **number of valid sequences of actions grows factorially**, with particularly large growth observed in games like chess and Go.

Alpha-beta pruning

Pruning

To enhance the efficiency of the minimax algorithm, one could possibly **prune** certain parts of the search tree, thereby **avoiding the exploration of descendant nodes**.

Pruning

How would you implement this modification? **What** factors would you take into account?

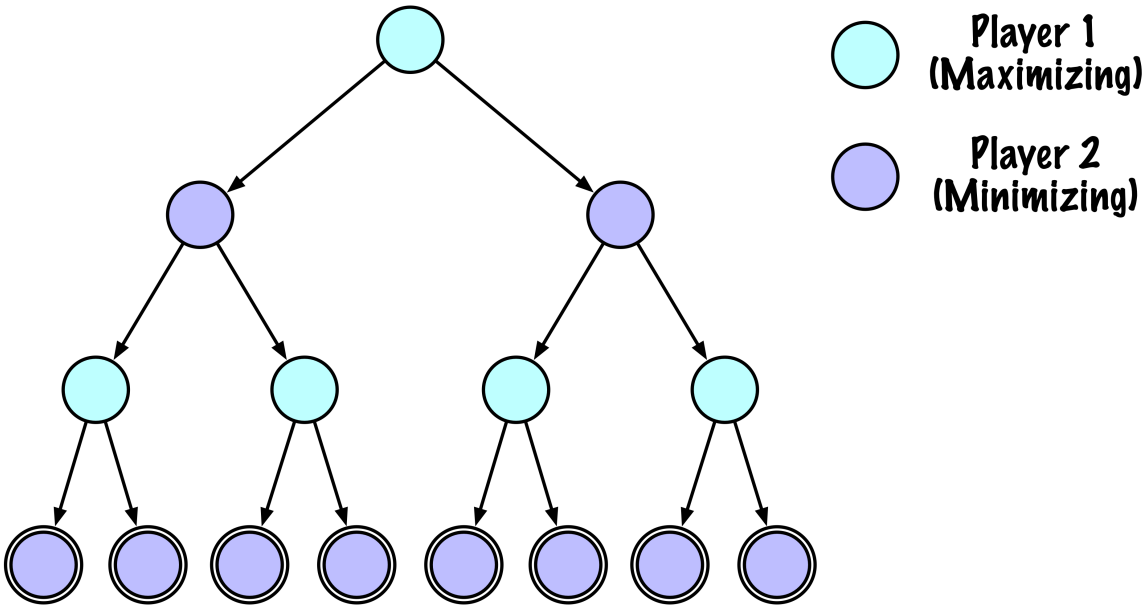
Pruning

Tree pruning should be performed **only when it can be demonstrated that those subtrees cannot yield better solutions**.

But **how**?

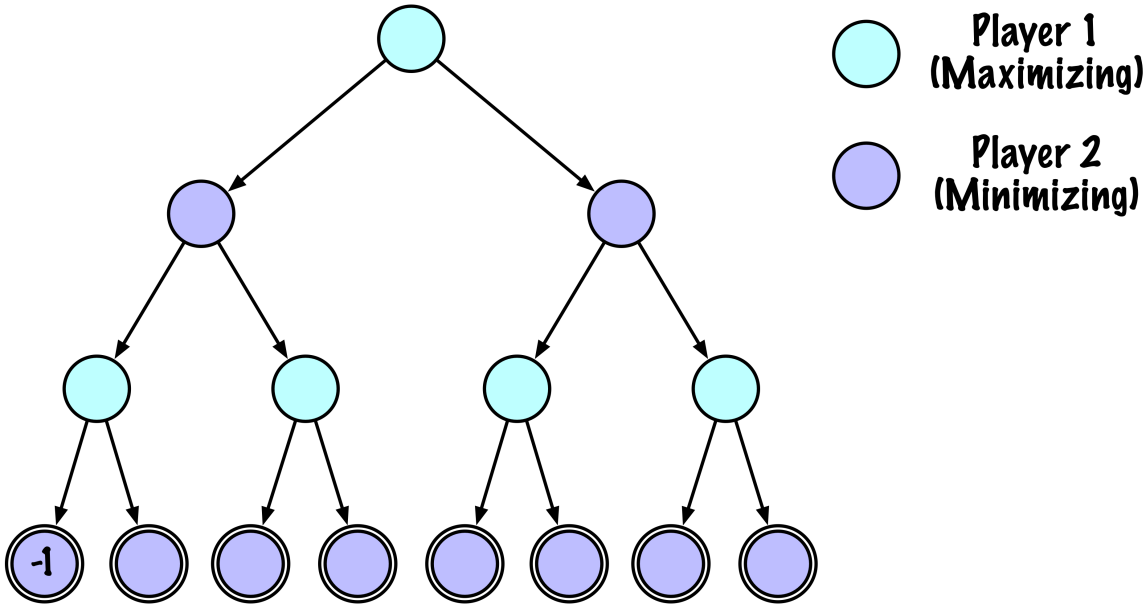
Because alpha-beta pruning discards only subtrees that cannot improve the current bound, it preserves the minimax value and the resulting optimal move relative to standard minimax, while exploring fewer nodes.

Criteria for Pruning

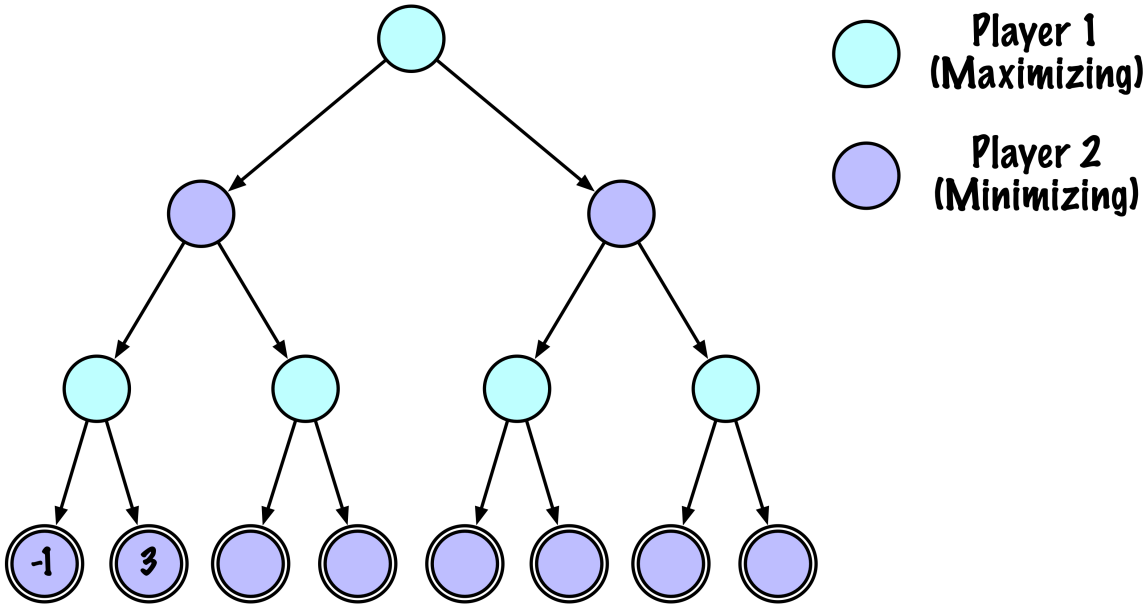


Based on Algorithms Explained – minimax and alpha-beta pruning by Sebastian Lague.

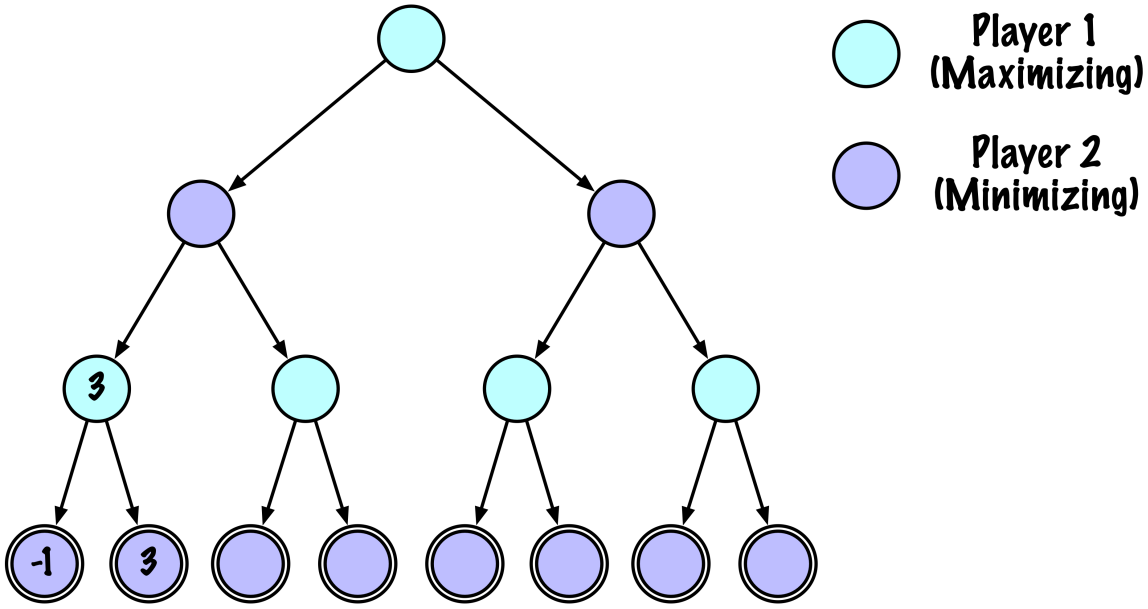
Criteria for Pruning



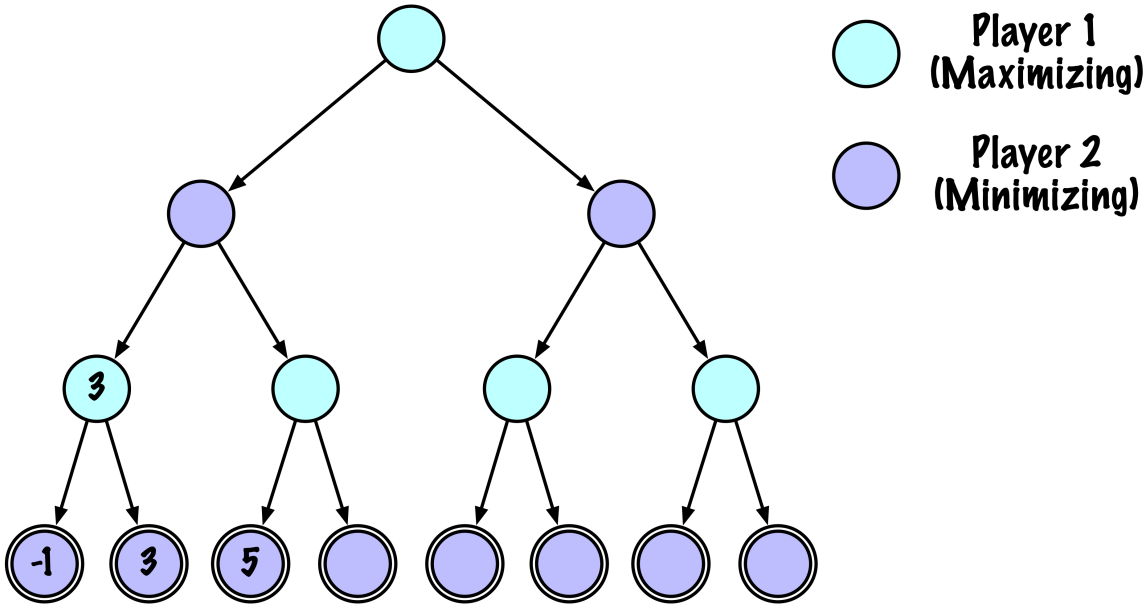
Criteria for Pruning



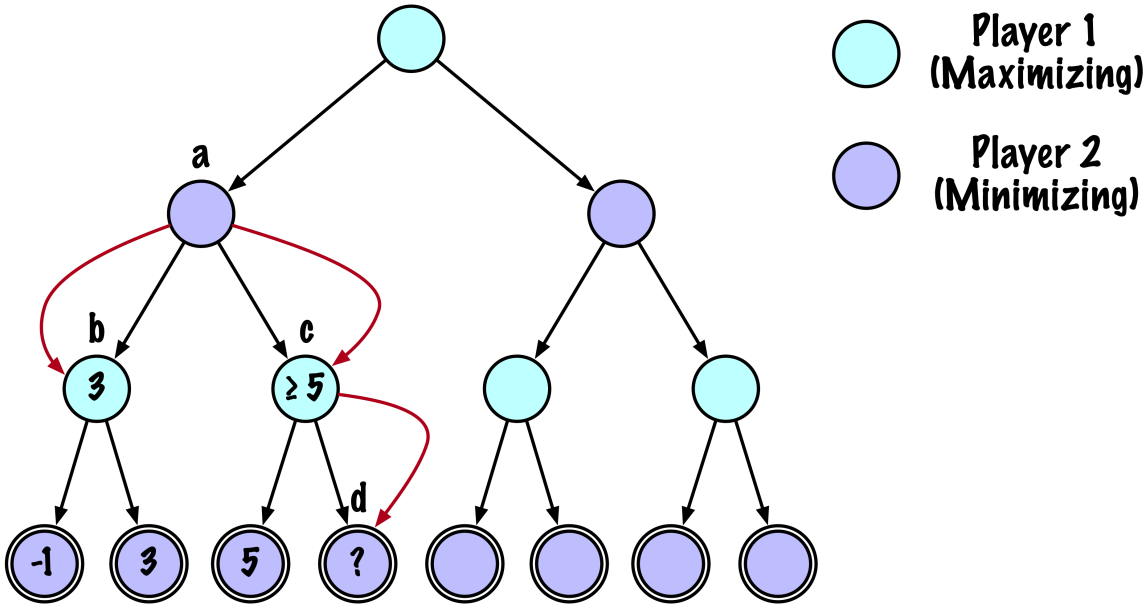
Criteria for Pruning



Criteria for Pruning



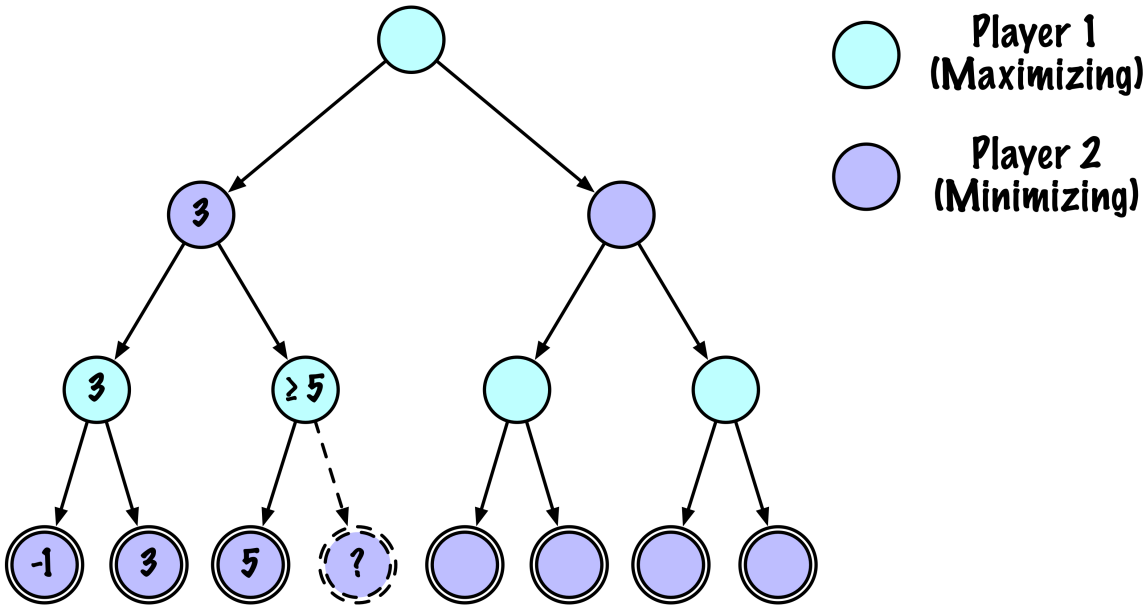
Criteria for Pruning



We know that the value of node **c** is at least 5, since **c** is a maximizing node.

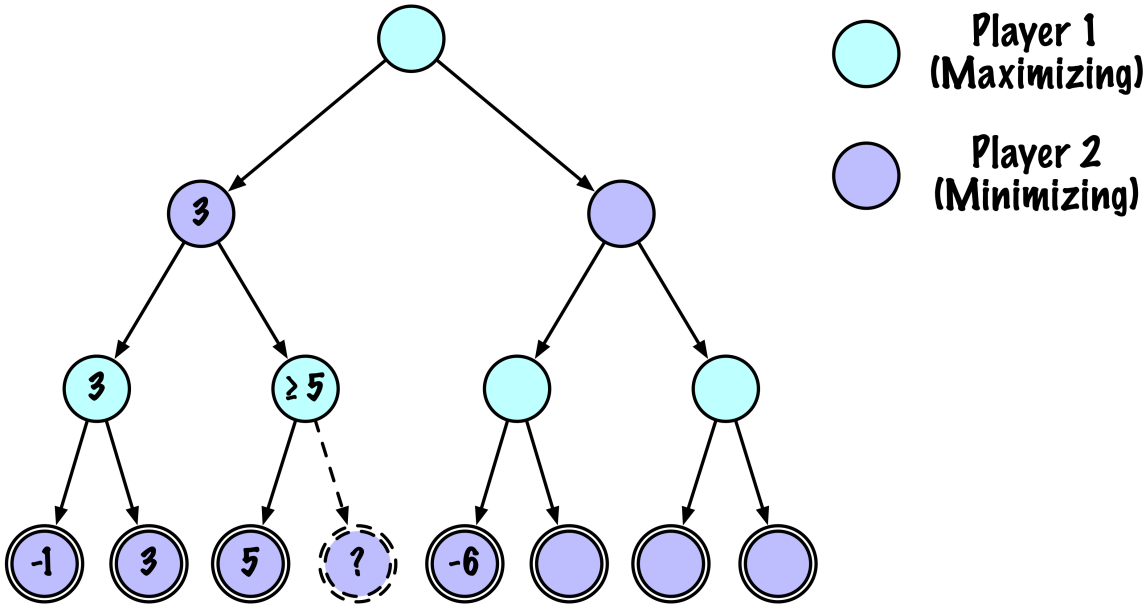
Node **a** is minimizing node and has to choose between 3 (node **b**) and at least 5 (node **c**). No matter what we find in node **d** does not matter, node **a** will choose 3.

Criteria for Pruning



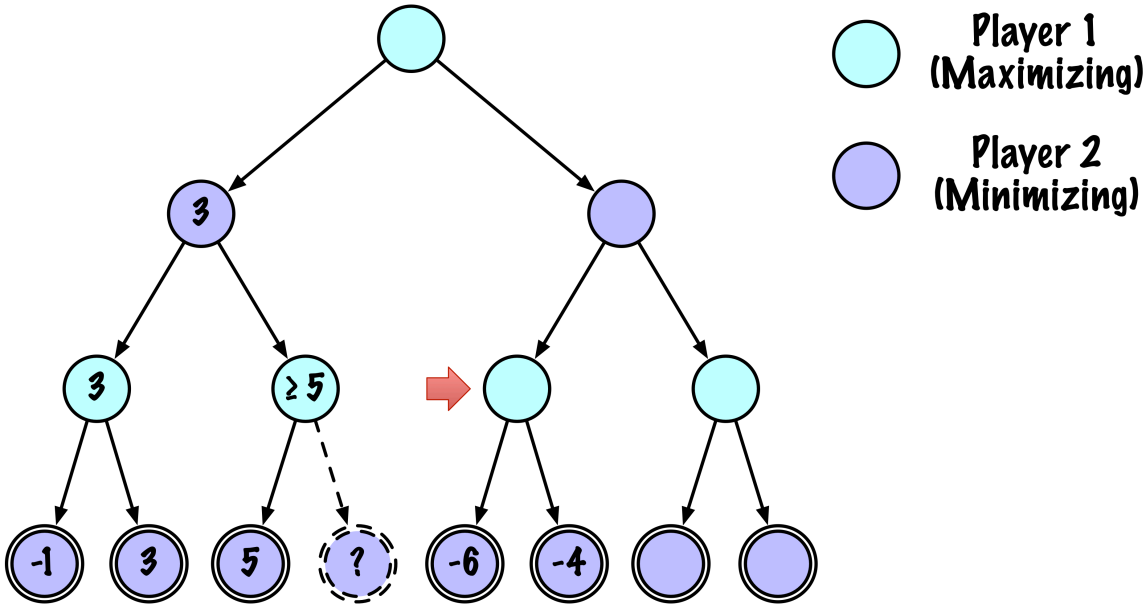
Let us continue briefly.

Criteria for Pruning



Let us continue briefly.

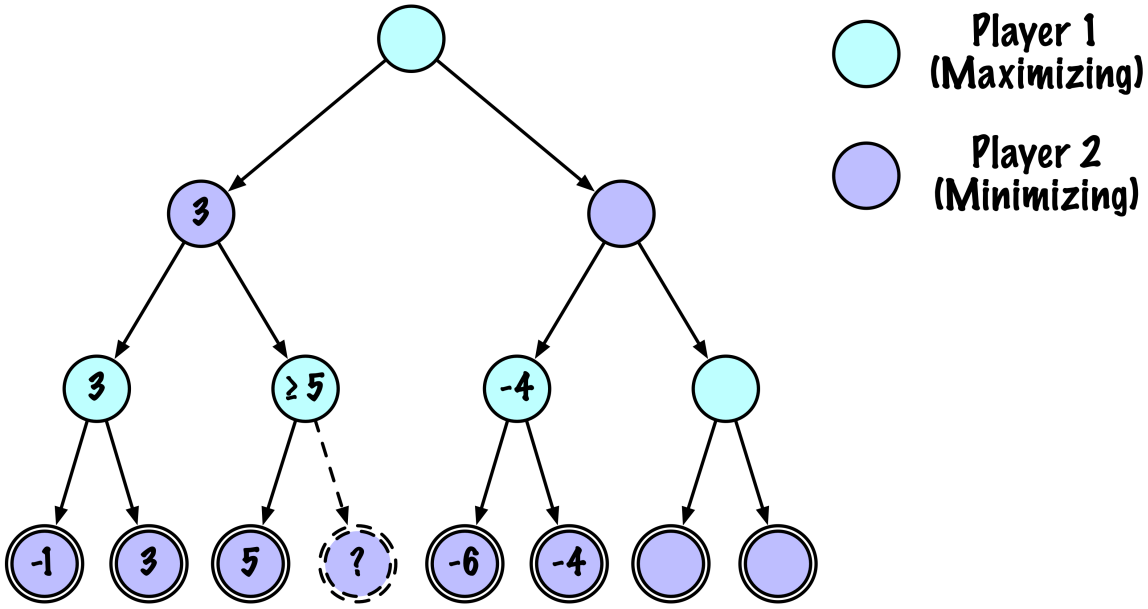
Criteria for Pruning



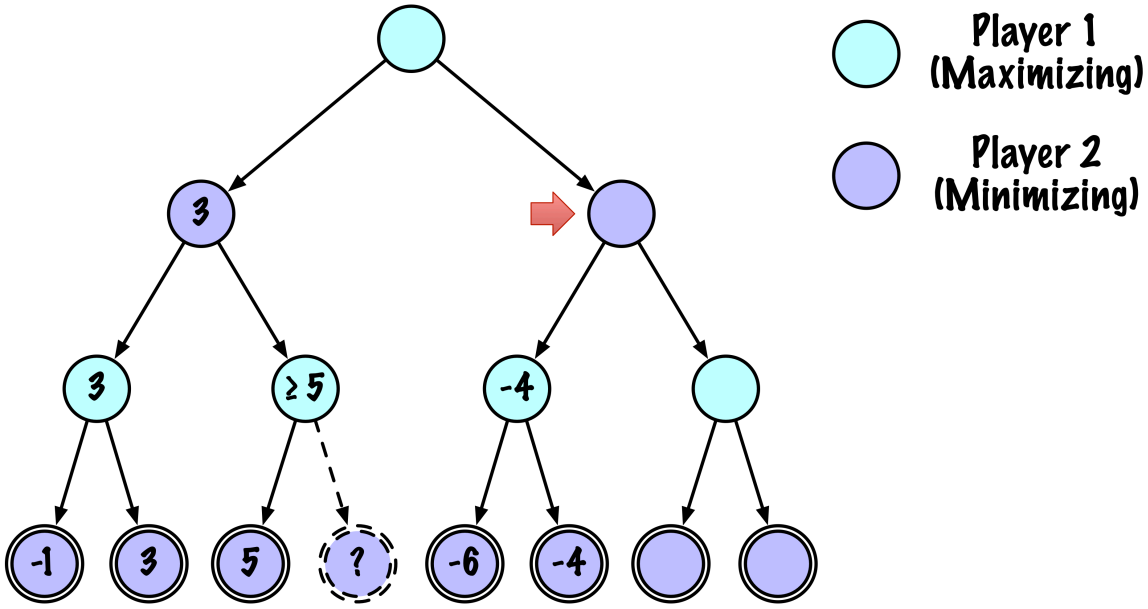
What value should be assigned to the node indicated by the arrow?

This is a maximizing node; therefore, we will assign it the value of -4.

Criteria for Pruning



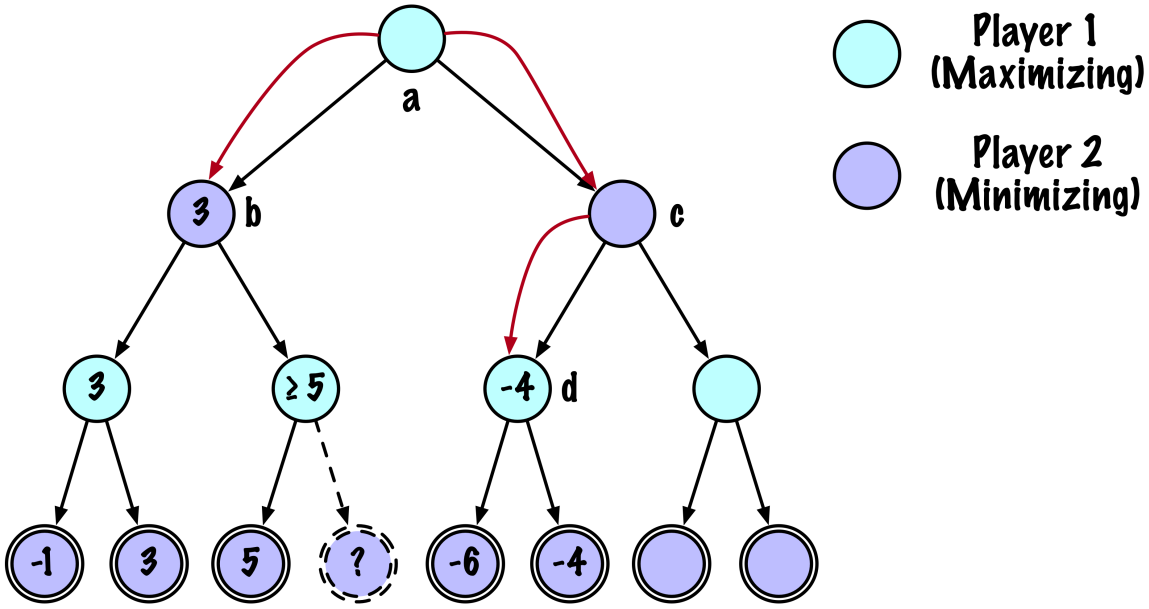
Criteria for Pruning



The node marked by an arrow presents an interesting scenario.

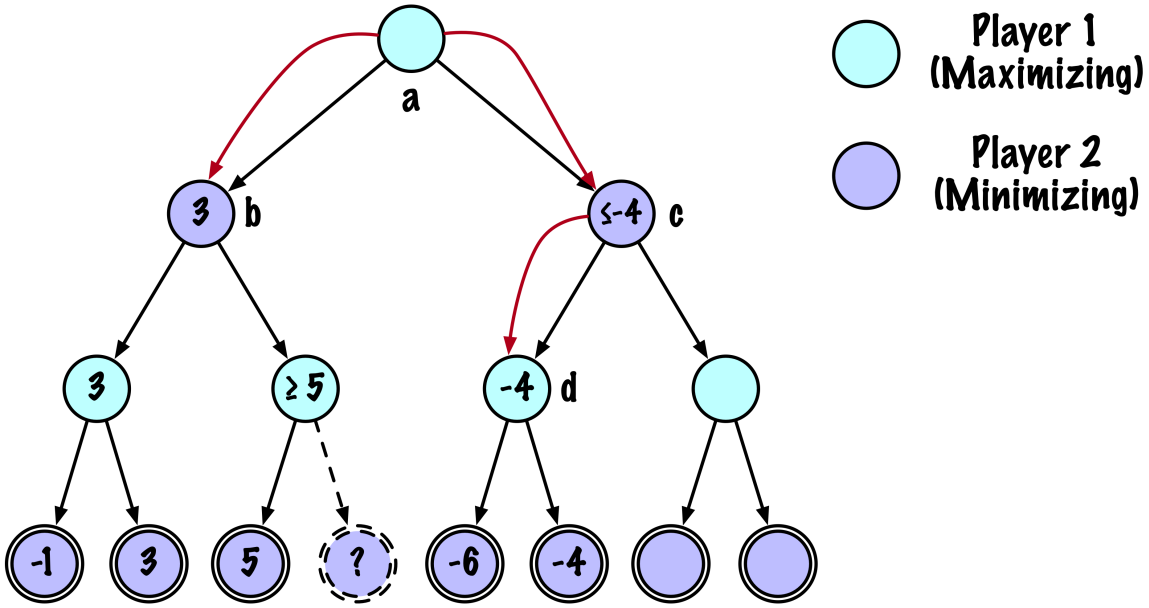
Can you work out its value?

Criteria for Pruning



What do we know?

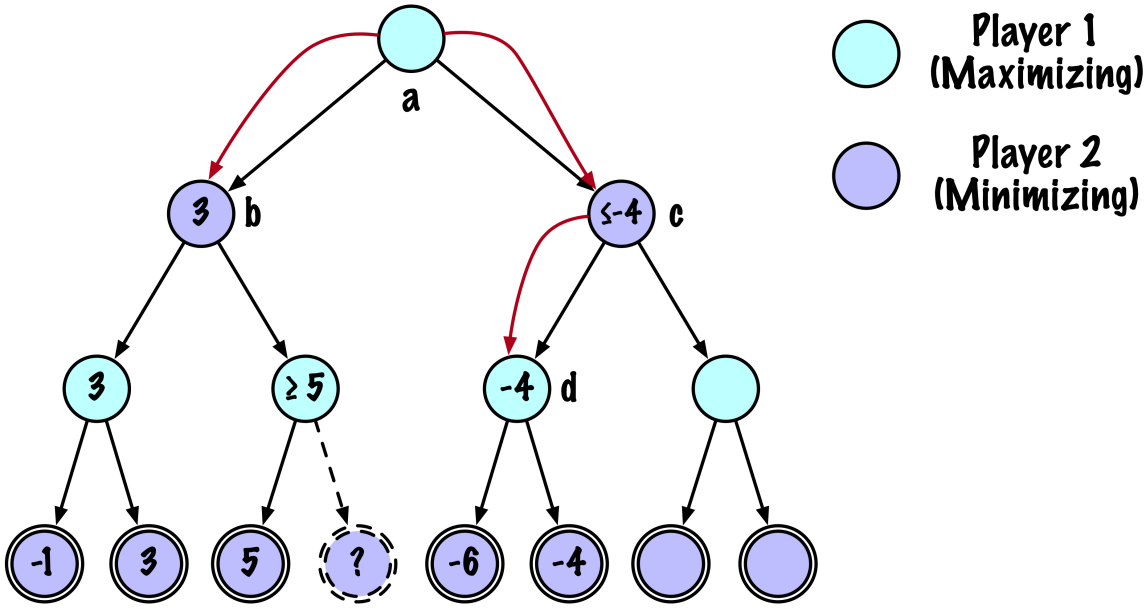
Criteria for Pruning



Can I say that the maximum value for node **c** is -4?

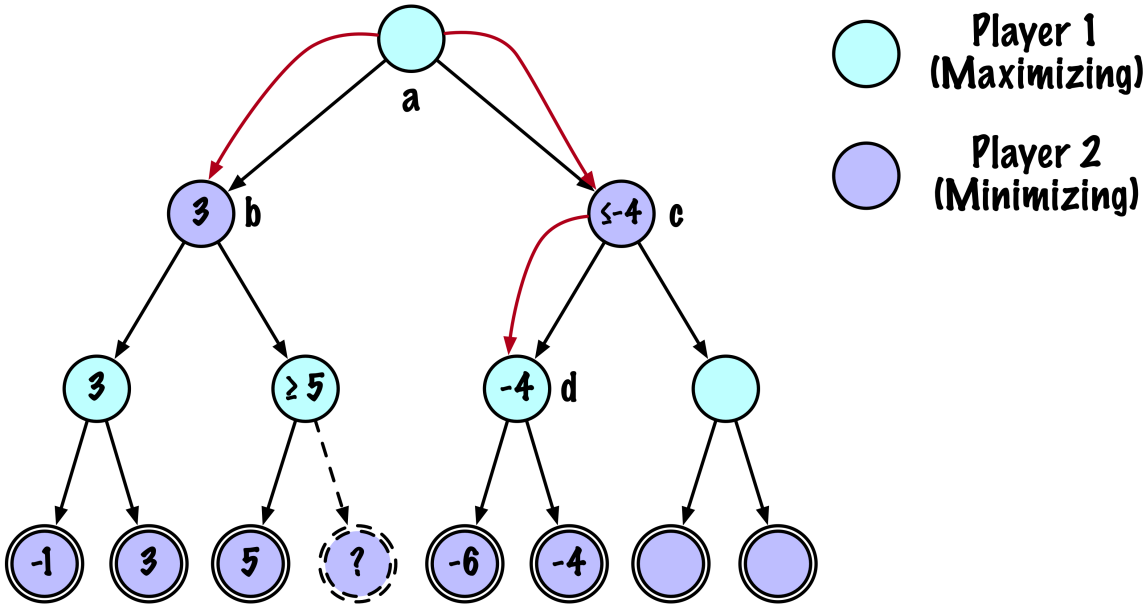
Suppose the right child of node **c** contains the value 5. Would you assign 5 to node **c**? Certainly not. As a minimizing node, node **c** will have a value no greater than -4.

Criteria for Pruning



Exploring the right child of node **c** might reveal a value lower than -4. **Does this matter?**

Criteria for Pruning



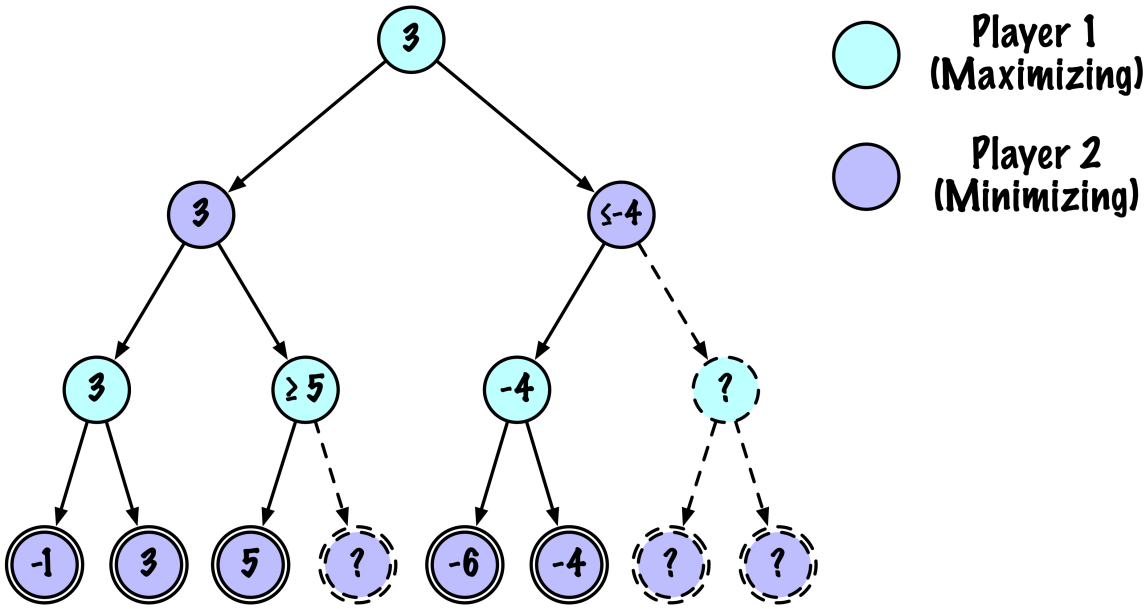
Node **a** is a maximizing node. From node **b**, it ascertains a minimum achievable value of 3.

Node **c** is a minimizing node. From node **d**, it determines that the maximum value it can return is -4.

Consequently, node **a** will always select node **b**, regardless of the value stored in node **c**, since the value of **c** will not exceed -4.

Even if node **c**, as a minimizing node, discovers an extremely low value (e.g., -100) by exploring its right child, node **a**, being a maximizing node, will still opt for node **b**.

Criteria for Pruning



The decisions made by players 1 and 2 remain unchanged with or without pruning. However, pruning reduces the number of nodes visited.

Alph-Beta Pruning

Alpha-beta pruning is an optimization technique for the minimax algorithm that **reduces the number of nodes evaluated** in the search tree.

Alph-Beta Pruning

It achieves this by **eliminating branches that cannot possibly influence the final decision**, using two parameters:

- **alpha**, the maximum score that the maximizing player is assured, and
- **beta**, the minimum score that the minimizing player is assured.

Maximizing Player's Perspective

At a **maximizing node**:

- The **maximizer** aims to **maximize** the score.
- **Alpha (α)** is updated to the **highest** value found so far among child nodes.
- **Process**:
  - Initialize $\alpha = -\infty$.

    - For each child node:
        - Compute the evaluation score.
        - Update $\alpha = \max(\alpha, \text{child\_score})$.

Minimizing Player's Perspective

At a **minimizing node**:

- The **minimizer** aims to **minimize** the score.
- **Beta (β)** is updated to the **lowest** value found so far among child nodes.
- **Process**:
    - Initialize $\beta = \infty$.
    - For each child node:
        - Compute the evaluation score.
        - Update $\beta = \min(\beta, \text{child\_score})$.

Alph-Beta Pruning

When a node's evaluation proves it **cannot improve** on the current alpha or beta, further **exploration of that branch is halted**, thereby enhancing computational efficiency without affecting the outcome.

Role of Alpha and Beta in Pruning

Pruning Condition:

- **If** $\beta \leq \alpha$, further exploration of the current node's siblings is unnecessary.
- **Rationale**:
    - The maximizer has a guaranteed score of at least α.
    - The minimizer can ensure that the maximizer cannot get a better score than β.
    - If $\beta \leq \alpha$, the maximizer won't find a better option in this branch.

Alpha-Beta Search

```
function ALPHA-BETA-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state, −∞, +∞)
  return move

function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
    if v2 > v then
      v, move ← v2, a
      α ← MAX(α, v)
    if v ≥ β then return v, move
  return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
    if v2 < v then
      v, move ← v2, a
      β ← MIN(β, v)
    if v ≤ α then return v, move
  return v, move
```

Attribution: (Russell and Norvig 2020, fig. 5.7)

Walkthrough (6:21 to 8:10)

https://youtu.be/l-hh51ncgDI

Attribution: Sebastian Lague. Start watching at 6**m** 21**s**.

Node Order

- The **effectiveness of pruning** is influenced by the **order** in which nodes are evaluated.
- Greater pruning is achieved if nodes are ordered from **most to least promising**.

Refer to: Shannon (1959) for a discussion within the context of chess.

If X already has a branch that ensures at least a draw, and an ancestor minimizing node knows it can force X into a loss, then exploring positions where X could hypothetically do even better is pointless. O will never allow those lines to occur.

If O already has a move that guarantees X cannot win, then evaluating further children that would give X an even worse outcome (for X) makes no difference. X will avoid those branches anyway.

As the branching factor increases, the potential for effective pruning correspondingly rises.

If move ordering were perfect, search procedures would be unnecessary. In practice, however, effective heuristics, such as killer moves, the history heuristic, and neural network-derived policy priors, significantly enhance move ordering. These approaches substantially improve pruning efficiency within search algorithms.

`MinimaxAlphaBetaSolverV1`

In [42]:
```python
class MinimaxAlphaBetaSolverV1(Solver):

    """
    A classical Minimax solver enhanced with Alpha–Beta pruning.

    - Assumes "X" is the maximizing player.
    - Performs a full search of the Tic–Tac–Toe game tree.
    """

    # ------------------------------------------------------------------
    # Solver interface
    # ------------------------------------------------------------------

    def select_move(self, game, state, player):

        """
        Choose the best move for `player` using Minimax with
        Alpha–Beta pruning.

        For Tic–Tac–Toe, depth=9 suffices to search the entire game.
        """

        self.game = game
        maximizing = (player == "X")

        value, move = self._alphabeta(
            state=state,
            player=player,
            maximizing=maximizing,
            depth=9,
            alpha=-math.inf,
            beta=math.inf
        )
```

```
        return move

    # ------------------------------------------------------------
    # CORE MINIMAX WITH ALPHA-BETA PRUNING
    # ------------------------------------------------------------

    def _alphabeta(self, state, player, maximizing, depth, alpha, beta):

        """
        Internal recursive minimax search with alpha-beta pruning.

        Parameters
        ----------
        state      : NumPy array, current board
        player     : "X" or "O", the player to move
        maximizing : True if this is a maximizing node (X to move)
        depth      : remaining search depth
        alpha      : best value found so far for the maximizer
        beta       : best value found so far for the minimizer

        Returns
        -------
        (value, move)
            value : evaluation of the state from X's perspective (+1/-1/0)
            move  : the best move found at this node
        """

        # Terminal test: win/loss/draw
        if self.game.is_terminal(state) or depth == 0:
            return self.game.evaluate(state), None

        moves = self.game.get_valid_moves(state)
        best_move = None

        # ------------------------------------------------------------
        # Maximizing node (X)
        # ------------------------------------------------------------

        if maximizing:
            value = -math.inf

            for move in moves:
                next_state = self.game.make_move(state, move, player)

                child_val, _ = self._alphabeta(
                    next_state,
                    self.game.get_opponent(player),
                    False,            # next is minimizing
                    depth - 1,
                    alpha,
                    beta
                )

                if child_val > value:
                    value = child_val
```

```
                    best_move = move

                # Update alpha
                alpha = max(alpha, value)

                # Prune
                if beta <= alpha:
                    break

            return value, best_move

        # ------------------------------------------------------------
        # Minimizing node (O)
        # ------------------------------------------------------------

        else:
            value = math.inf

            for move in moves:
                next_state = self.game.make_move(state, move, player)

                child_val, _ = self._alphabeta(
                    next_state,
                    self.game.get_opponent(player),
                    True,              # next is maximizing
                    depth - 1,
                    alpha,
                    beta
                )

                if child_val < value:
                    value = child_val
                    best_move = move

                # Update beta
                beta = min(beta, value)

                # Prune
                if beta <= alpha:
                    break

            return value, best_move
```

MinimaxAlphaBetaSolverV1

In [43]:
```
class MinimaxAlphaBetaSolverV1(Solver):

    def select_move(self, game, state, player):

        self.game = game
        maximizing = (player == "X")

        value, move = self._alphabeta(
            state=state,
```

```
            player=player,
            maximizing=maximizing,
            depth=9,
            alpha=-math.inf,
            beta=math.inf
        )

        return move
```

`MinimaxAlphaBetaSolverV1` is_a `Solver`.

`_alphabeta`

In [44]:
```
    def _alphabeta(self, state, player, maximizing, depth, alpha, beta):

        if self.game.is_terminal(state) or depth == 0:
            return self.game.evaluate(state), None

        moves = self.game.get_valid_moves(state)
        best_move = None
```

`_alphabeta`: if maximizing

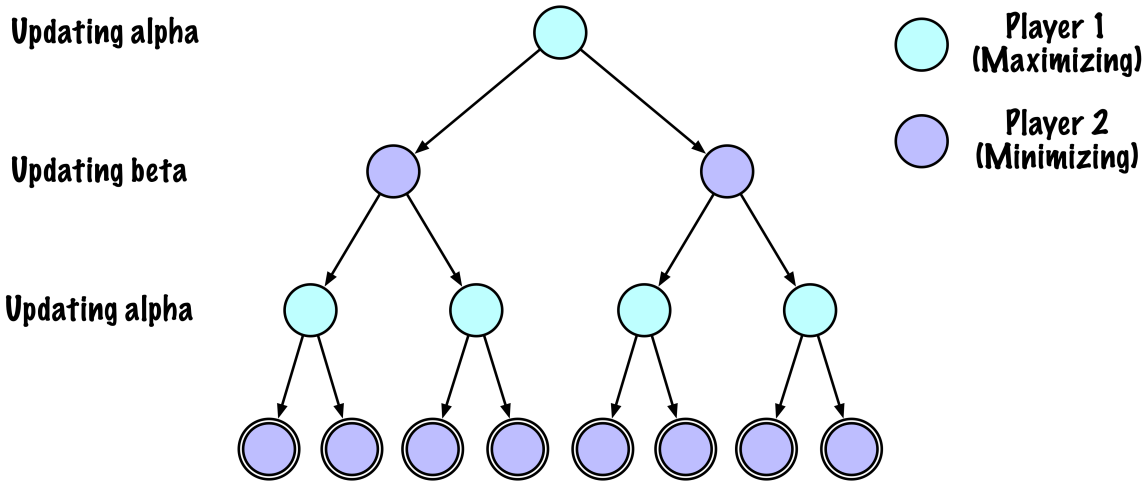
In [45]:
```
        if maximizing:
            value = -math.inf

            for move in moves:
                next_state = self.game.make_move(state, move, player)

                child_val, _ = self._alphabeta(
                    next_state,
                    self.game.get_opponent(player),
                    False,            # next is minimizing
                    depth - 1,
                    alpha,
                    beta
                )

                if child_val > value:
                    value = child_val
                    best_move = move

                # Update alpha
                alpha = max(alpha, value)

                # Prune
                if beta <= alpha:
                    break

            return value, best_move
```

`_alphabeta`: if minimizing

In [46]:

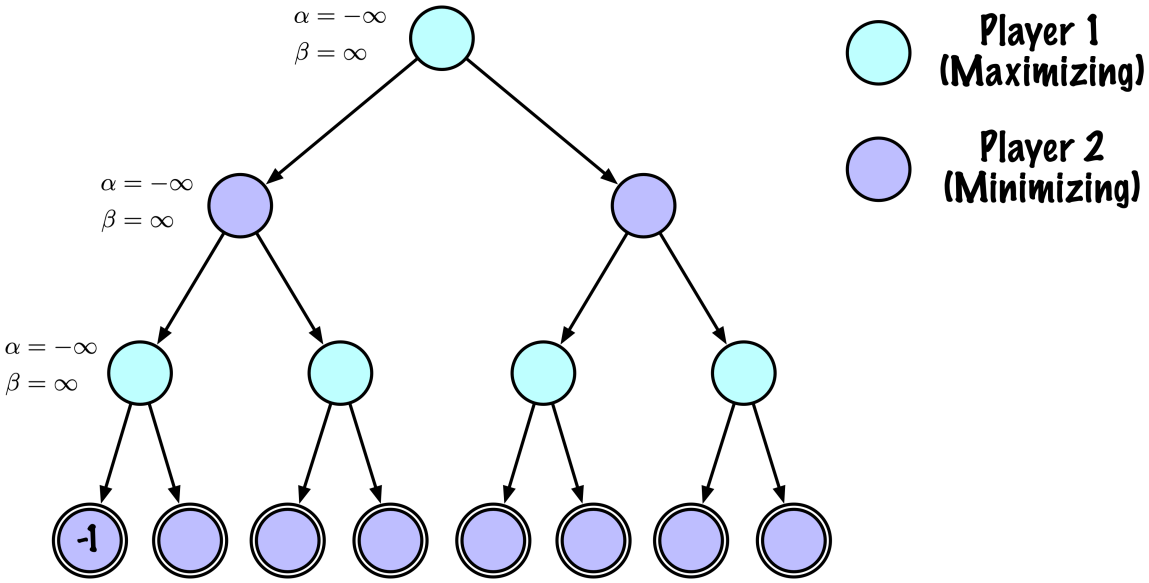
```
        else:
            value = math.inf

            for move in moves:
                next_state = self.game.make_move(state, move, player)

                child_val, _ = self._alphabeta(
                    next_state,
                    self.game.get_opponent(player),
                    True,              # next is maximizing
                    depth - 1,
                    alpha,
                    beta
                )

                if child_val < value:
                    value = child_val
                    best_move = move

                # Update beta
                beta = min(beta, value)

                # Prune
                if beta <= alpha:
                    break

            return value, best_move
```

Walkthrough



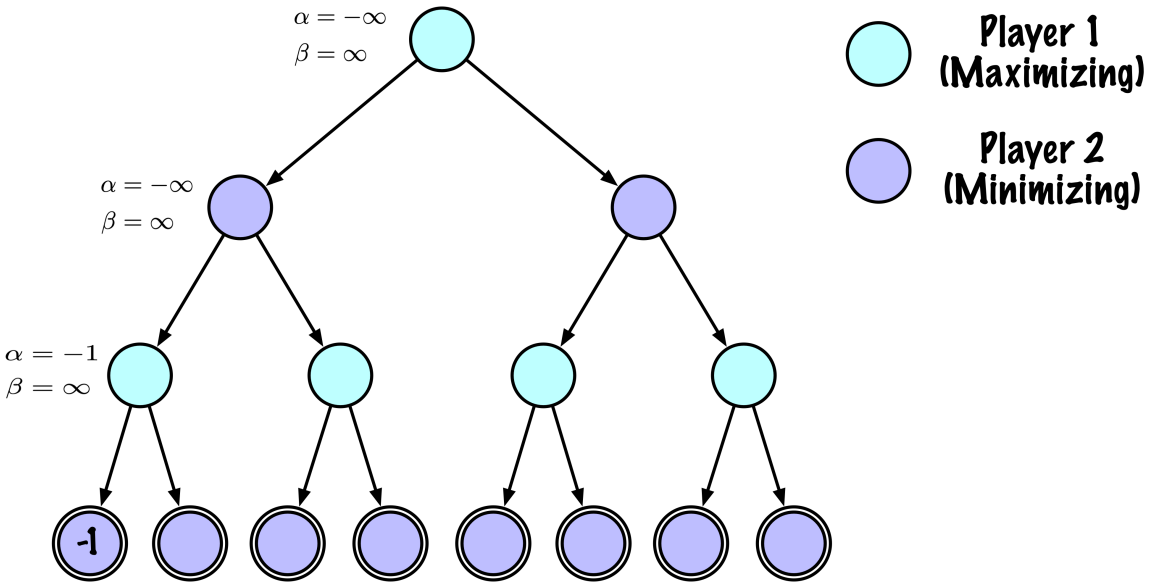
Maximizing nodes update the **alpha** values, while **minimizing** nodes update the **beta** values.

Walkthrough



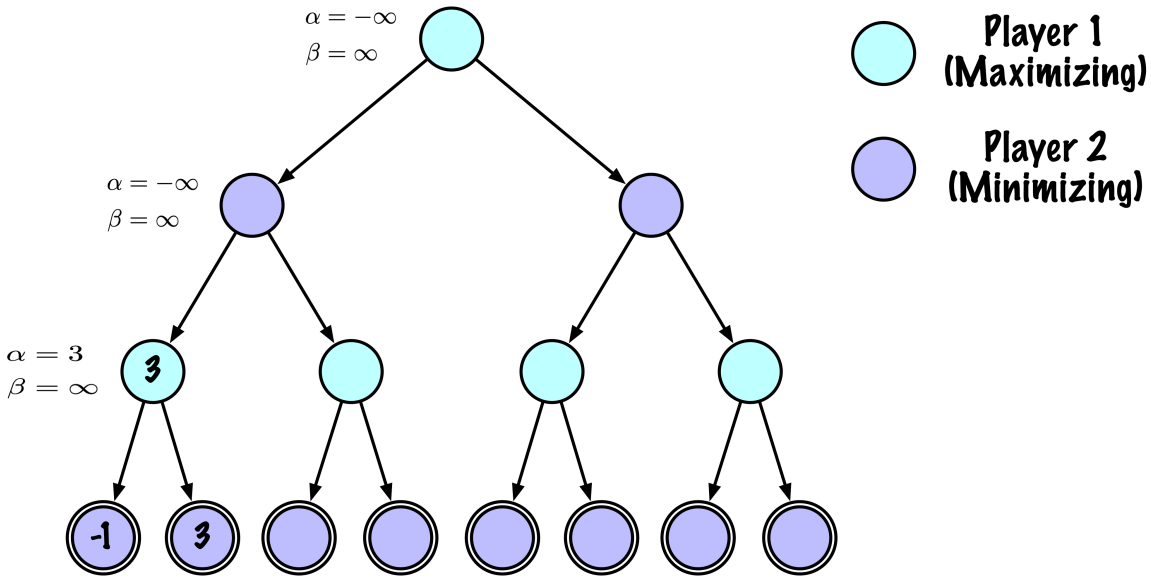
The values of α and β are initially set to $-\infty$ and ∞, respectively. The recursive calls continue traversing the tree until the leftmost node is reached.

Walkthrough



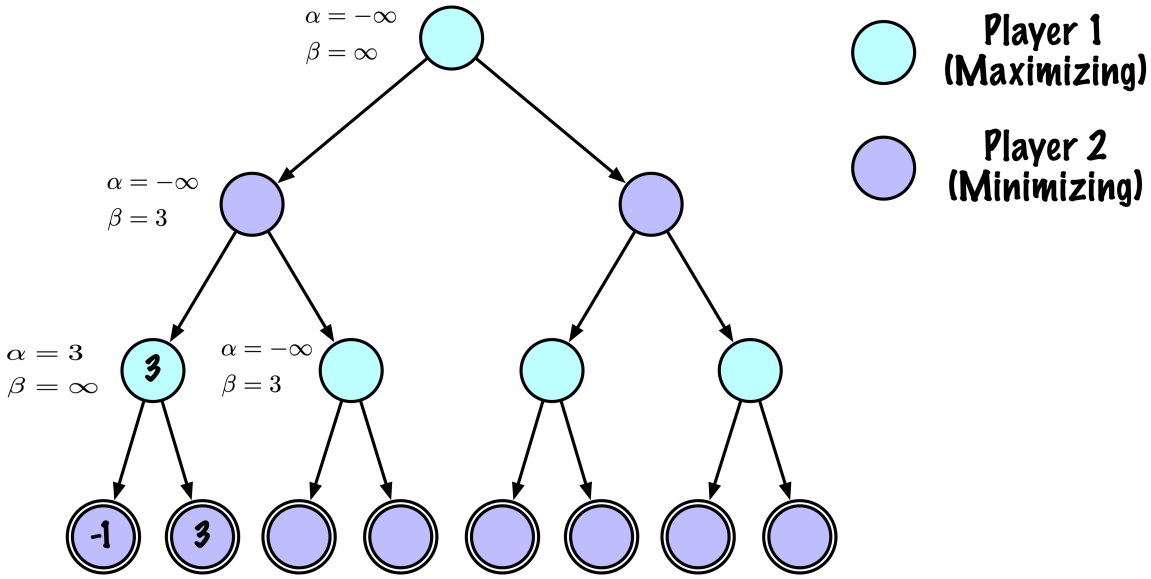
The value of α is updated to -1.

Walkthrough



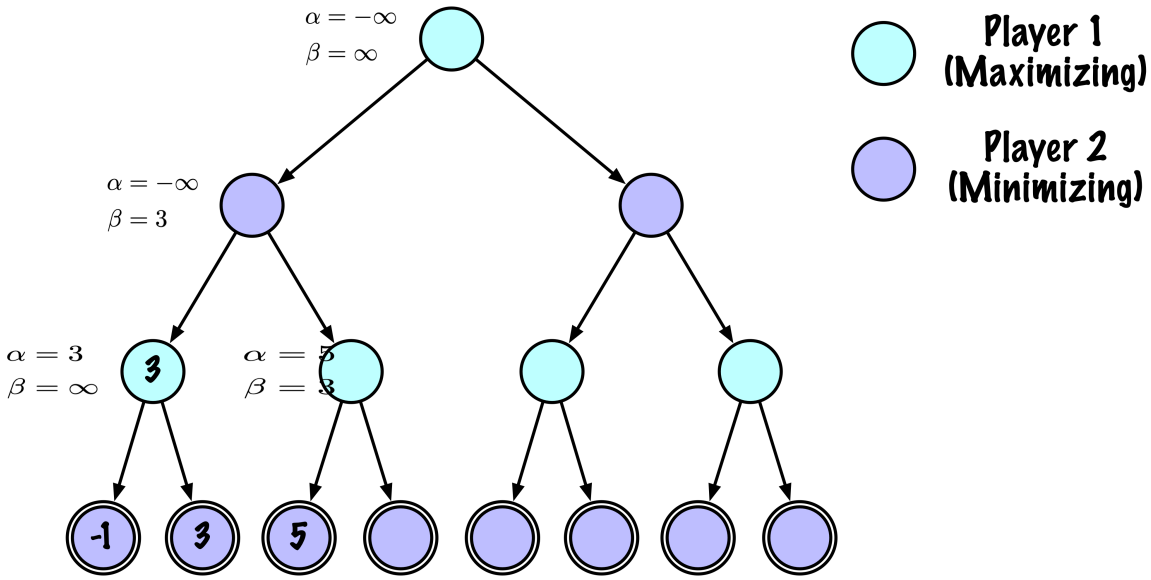
Updating α to 3.

Walkthrough



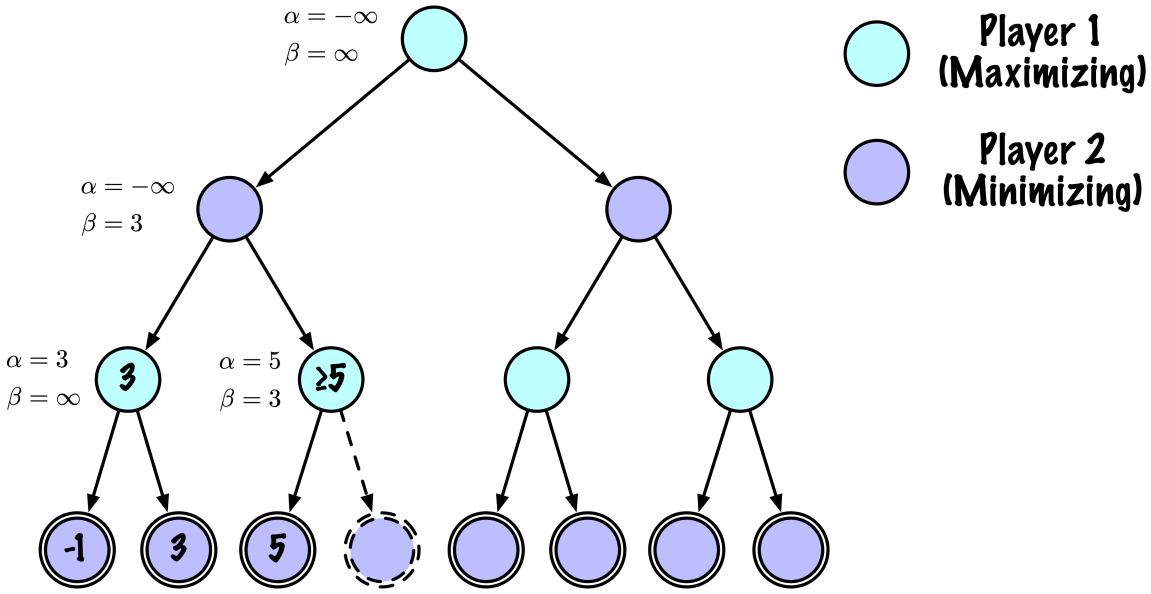
Updating β from ∞ to 3. Passing those values to the right child.

Walkthrough



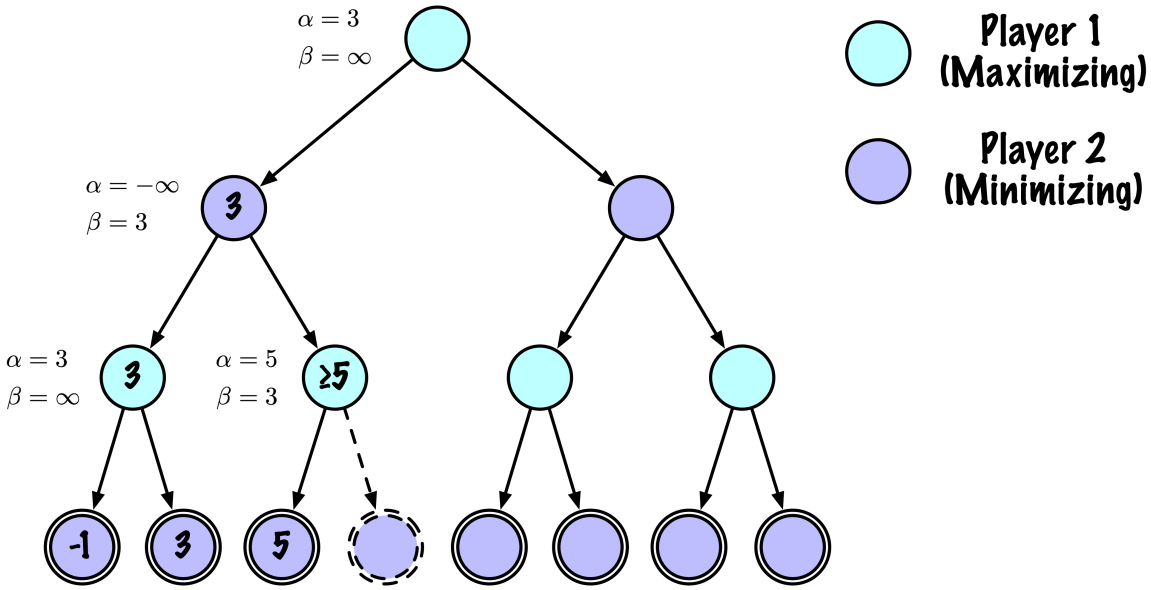
Updating α from $-\infty$ to 5. We now have $\beta \leq \alpha$.

Walkthrough



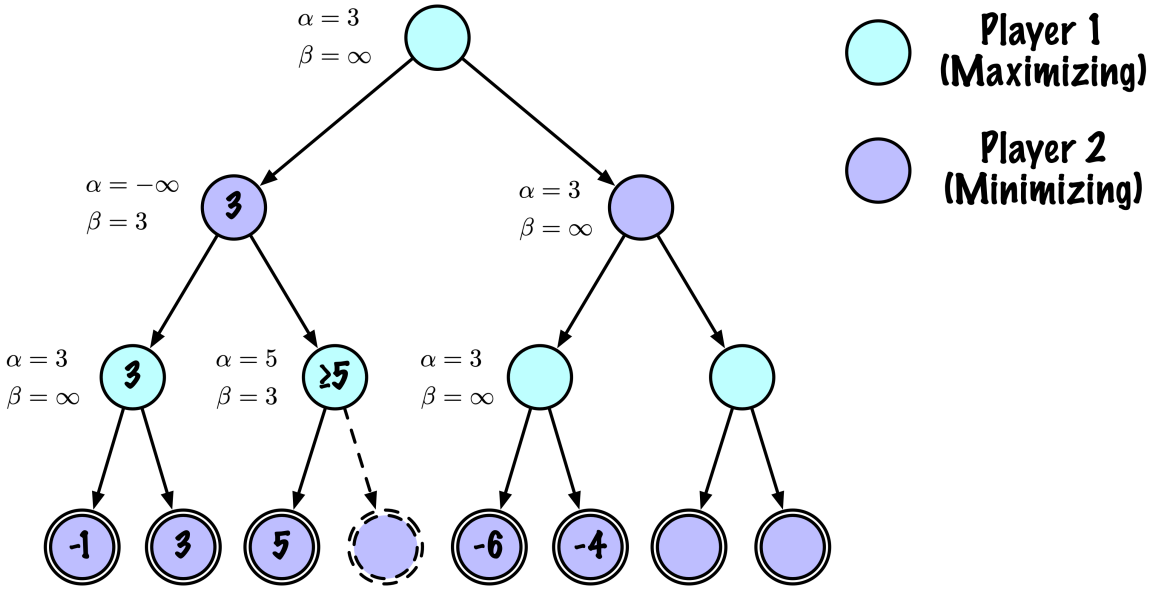
The right child is not visited. The parent (a minimizer) has a better option (β) than α.

Walkthrough



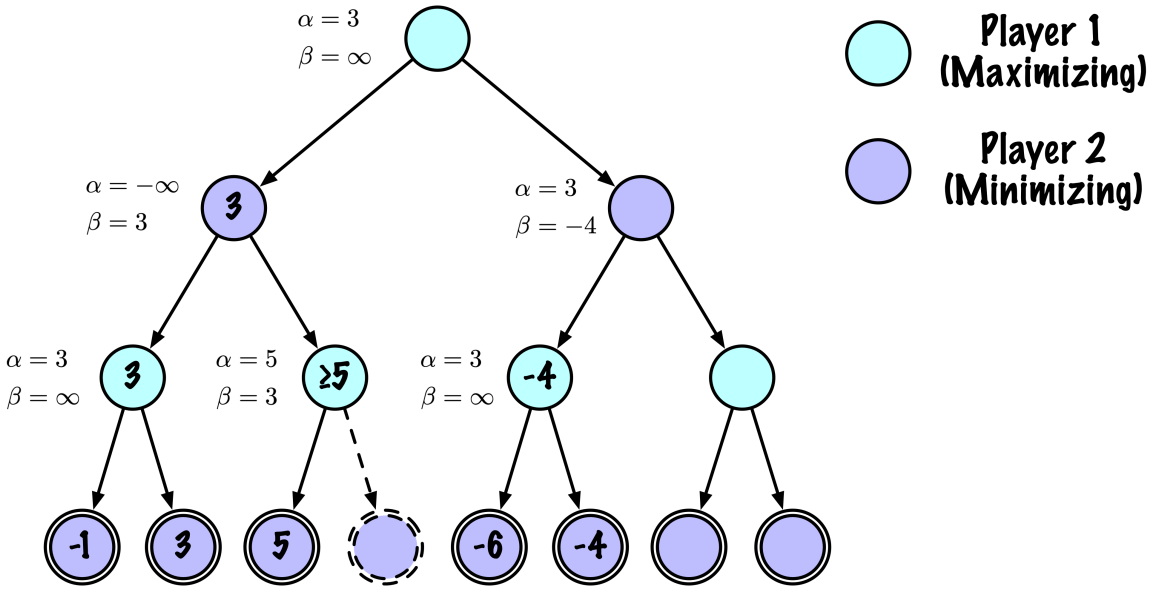
Returning from the recursive calls, the root node receives a value of 3 from its left child and updates its α to 3, which exceeds its initial value of $-\infty$.

Walkthrough



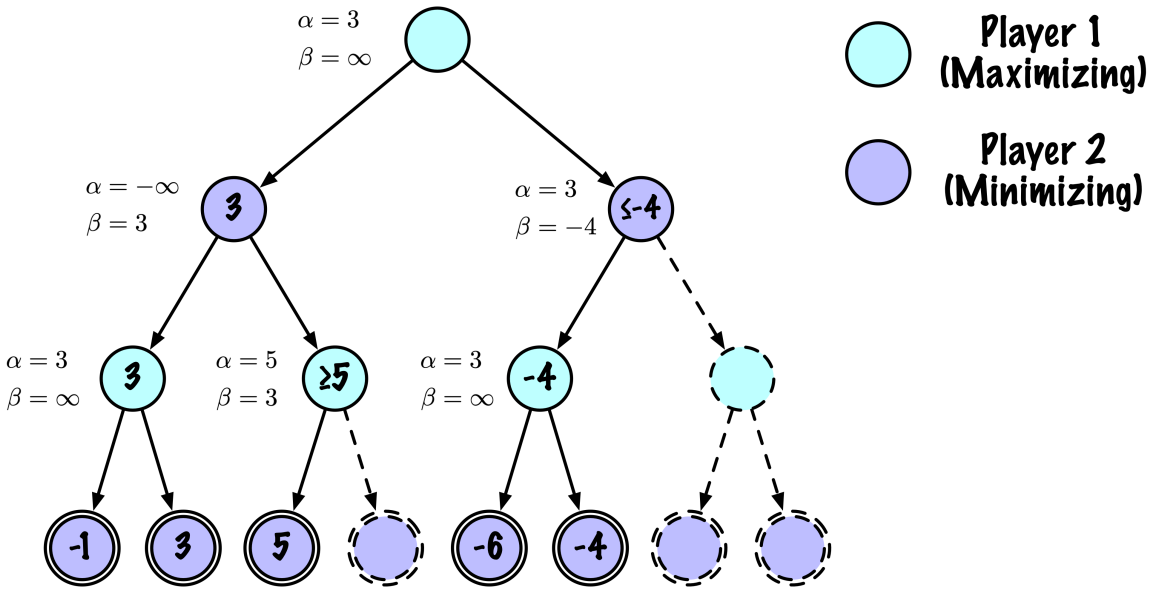
Now traversing the right subtree from the root. Upon reaching left-most maximizing node of the right subtree, α is not updated since -6 and -4 are less than 3.

Walkthrough



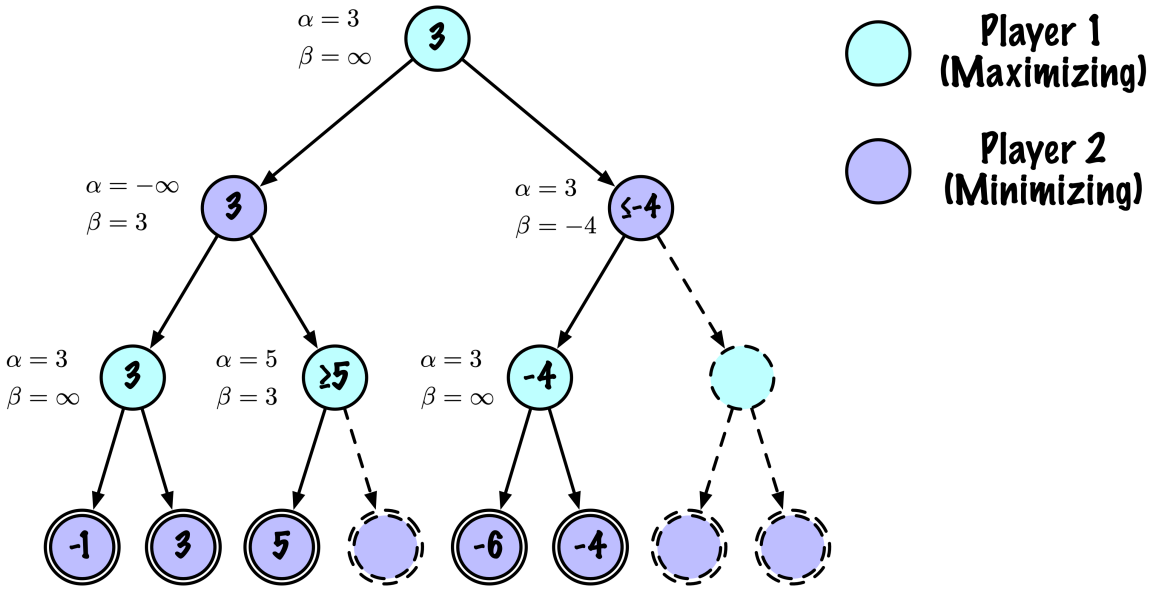
β is now updated to -4. We now have that $\beta \leq \alpha$.

Walkthrough



The right child is pruned.

Walkthrough



The maximizing player knows that its best move has a value of 3 (going left).

Sanity Check

In [47]:
```
game = TicTacToe()

a = MinimaxAlphaBetaSolverV1()
b = RandomSolver(7)

results = evaluate_solvers(game, a, b, num_games=100)

results
```

```
{'X_wins': 100, 'O_wins': 0, 'draws': 0}
```

Sanity Check

In [48]:
```
game = TicTacToe()

a = RandomSolver(7)
b = MinimaxAlphaBetaSolverV1()

results = evaluate_solvers(game, a, b, num_games=100)

results
```

```
{'X_wins': 0, 'O_wins': 82, 'draws': 18}
```

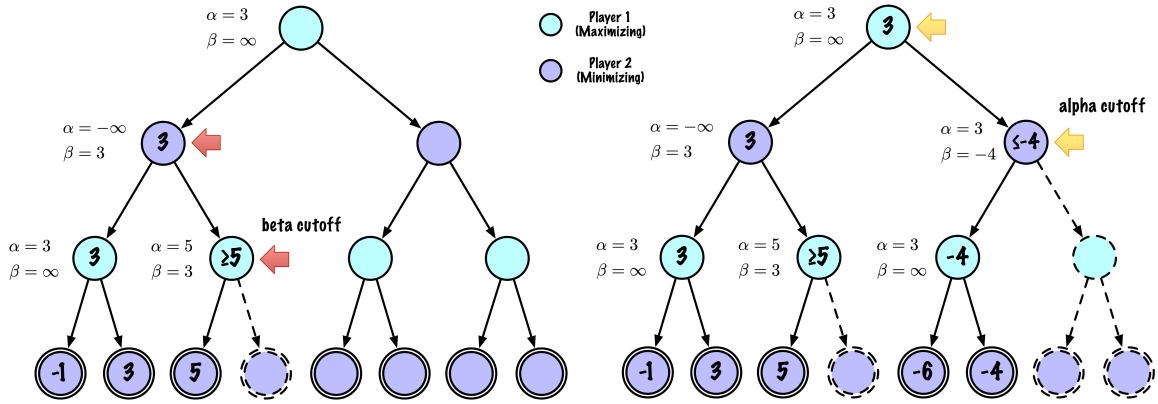
Sanity Check

In [49]:
```
game = TicTacToe()

a = MinimaxAlphaBetaSolverV1()
b = MinimaxAlphaBetaSolverV1()
```

```
results = evaluate_solvers(game, a, b, num_games=100)

results
```

```
{'X_wins': 0, 'O_wins': 0, 'draws': 100}
```

Summmary



- **Alpha Cutoff:** Occurs at **minimizer nodes** when $\beta \leq \alpha$.
- **Beta Cutoff:** Occurs at **maximizer nodes** when $\alpha \geq \beta$.

1. **Pruning at Both Node Types:**
   - Pruning can occur during both the minimization and maximization phases. This means that both **minimizer** and **maximizer** nodes can be pruned if certain conditions are met.
2. **Updating Alpha and Beta:**
   - At **maximizer nodes**, the algorithm updates the **alpha** value to the maximum of its current value and the value of the child nodes evaluated so far.
   - At **minimizer nodes**, the algorithm updates the **beta** value to the minimum of its current value and the value of the child nodes evaluated so far.
3. **Pruning Conditions:**
   - **At Maximizer Nodes:**
     - If **alpha** becomes **greater than or equal to beta** ($\alpha \geq \beta$), further exploration of the current node's descendants can be stopped. This is because the minimizer (opponent) can force the outcome to be no better than beta, so the maximizer cannot improve the result beyond this point.
     - This is often referred to as a **beta cutoff** because the value of beta is causing the pruning at a maximizer node.
   - **At Minimizer Nodes:**
     - If **beta** becomes **less than or equal to alpha** ($\beta \leq \alpha$), the algorithm can prune the remaining child nodes of the minimizer node. This is because the maximizer can force a result of at least alpha, so the minimizer cannot find a better (lower) outcome.

    - This is known as an **alpha cutoff** because the value of alpha is causing the pruning at a minimizer node.
4. **Process of Pruning:**
    - The pruning occurs **not when alpha or beta are updated**, but when the pruning condition ($\alpha \geq \beta$ at maximizer nodes or $\beta \leq \alpha$ at minimizer nodes) is met.
    - Once these conditions are satisfied, the algorithm knows that further exploration will not yield a better outcome, and thus it can safely prune those branches.

Summary:

- **Alpha Cutoff:** Occurs at **minimizer nodes** when $\beta \leq \alpha$.
- **Beta Cutoff:** Occurs at **maximizer nodes** when $\alpha \geq \beta$.
- **Why Pruning Occurs:**
    - In both cases, the pruning occurs because further exploration cannot influence the final decision. The opponent can force the game into a situation that's no better than the current evaluation.
- **Impact on Algorithm Efficiency:**
    - By implementing these cutoffs, the alpha-beta pruning algorithm reduces the number of nodes that need to be evaluated compared to the standard minimax algorithm, thus improving efficiency without affecting the outcome.

Discussion

- Grasping **why alpha-beta pruning boosts minimax efficiency** without altering outcomes requires **careful thought**.
- The algorithm **changes are minimal**.
- Is the enhancement **justified**?

`MinimaxSolverV2` (INSTRUMENTED)

In [50]:
```
class MinimaxSolverV2(Solver):

    """
    A classic, exact Minimax solver for Tic-Tac-Toe - *instrumented version*

    Differences from V1
    -------------------
    - Adds a counter `self.nodes_visited` that counts each recursive call.
```

```
    - Uses `reset()` so the GameRunner or TournamentRunner can correctly
      prepare the solver before each game.
    - Still performs a full game-tree search without memoization.

    Characteristics
    ---------------
    - Assumes "X" is the maximizing player.
    - Performs a complete minimax search (Tic-Tac-Toe is small enough).
    """

    def __init__(self):

        # Count how many nodes have been visited in the current run
        self.nodes_visited = 0

    # ------------------------------------------------------------
    # Solver interface
    # ------------------------------------------------------------

    def select_move(self, game, state, player):

        """
        Choose the optimal move for `player` using a full minimax search.
        """

        maximizing = (player == "X")
        value, move = self._minimax(game, state, player, maximizing)
        return move

    def reset(self):

        """
        Reset instrumentation counters at the start of a game (or tournament

        GameRunner / TournamentRunner should call solver.reset() before
        starting a new game.
        """

        self.nodes_visited = 0

    def get_name(self):

        """
        Name is inherited from Solver but solvers may override get_name()
        to display additional instrumentation info.
        """

        return f"{self.__class__.__name__} (nodes={self.nodes_visited})"

    # ------------------------------------------------------------
    # Core recursive minimax
    # ------------------------------------------------------------

    def _minimax(self, game, state, player, maximizing):

        """
```

```
        The core recursive minimax computation.

        Parameters
        ----------
        game       : TicTacToe instance
        state      : NumPy array of current board position
        player     : "X" or "O" — the player whose turn it is at this node
        maximizing : True if this node corresponds to X; False if O

        Returns
        -------
        (value, move)
            value : +1 for X win, -1 for O win, 0 otherwise
            move  : best move selected at this node
        """

        # Instrumentation
        self.nodes_visited += 1

        # ------------------------
        # TERMINAL NODE?
        # ------------------------
        if game.is_terminal(state):
            return game.evaluate(state), None

        # ------------------------
        # GENERATE ALL MOVES
        # ------------------------
        moves = game.get_valid_moves(state)

        # ------------------------
        # MAX NODE (X plays)
        # ------------------------
        if maximizing:
            best_value = -math.inf
            best_move = None

            for move in moves:
                next_state = game.make_move(state, move, player)
                val, _ = self._minimax(
                    game,
                    next_state,
                    game.get_opponent(player),
                    maximizing=False
                )
                if val > best_value:
                    best_value = val
                    best_move = move

            return best_value, best_move

        # ------------------------
        # MIN NODE (O plays)
        # ------------------------

        else:
```

```
            best_value = math.inf
            best_move = None

            for move in moves:
                next_state = game.make_move(state, move, player)
                val, _ = self._minimax(
                    game,
                    next_state,
                    game.get_opponent(player),
                    maximizing=True
                )
                if val < best_value:
                    best_value = val
                    best_move = move

            return best_value, best_move
```

`MinimaxAlphaBetaSolverV2` (INSTRUMENTED)

In [51]:
```
class MinimaxAlphaBetaSolverV2(Solver):

    """
    A classical Minimax solver enhanced with Alpha–Beta pruning,
    instrumented to count how many nodes are visited.

    - Assumes "X" is the maximizing player.
    - Performs a full search of the Tic–Tac–Toe game tree.
    - Alpha–Beta pruning reduces the number of explored states
      without changing the final result.

    Instrumentation
    ---------------
    - self.nodes_visited counts how many times _alphabeta() is called.
    """

    def __init__(self):

        # Count how many nodes have been visited in the current run
        self.nodes_visited = 0

    # ------------------------------------------------------------
    # Solver interface
    # ------------------------------------------------------------

    def select_move(self, game, state, player):

        """
        Choose the best move for `player` using Minimax with
        Alpha–Beta pruning.

        For Tic–Tac–Toe, depth=9 suffices to search the entire game.
        """

        self.game = game
```

```
        maximizing = (player == "X")

        value, move = self._alphabeta(
            state=state,
            player=player,
            maximizing=maximizing,
            depth=9,
            alpha=-math.inf,
            beta=math.inf
        )

        return move

    def reset(self):

        """
        Reset any per-game state.

        Called by GameRunner (or similar) at the start of a new game.
        """

        self.nodes_visited = 0

    # ------------------------------------------------------------
    # Private
    # ------------------------------------------------------------

    def _alphabeta(self, state, player, maximizing, depth, alpha, beta):

        """
        Internal recursive minimax search with alpha-beta pruning.

        Parameters
        ----------
        state      : NumPy array, current board
        player     : "X" or "O", the player to move
        maximizing : True if this is a maximizing node (X to move)
        depth      : remaining search depth
        alpha      : best value found so far for the maximizer
        beta       : best value found so far for the minimizer

        Returns
        -------
        (value, move)
            value : evaluation of the state from X's perspective (+1/-1/0)
            move  : the best move found at this node
        """

        # Instrumentation: count this node
        self.nodes_visited += 1

        # Terminal test: win/loss/draw or depth cutoff
        if self.game.is_terminal(state) or depth == 0:
            return self.game.evaluate(state), None

        moves = self.game.get_valid_moves(state)
```

```
        best_move = None

        # ------------------------------------------------------------
        # Maximizing node (X)
        # ------------------------------------------------------------
        if maximizing:
            value = -math.inf

            for move in moves:
                next_state = self.game.make_move(state, move, player)

                child_val, _ = self._alphabeta(
                    next_state,
                    self.game.get_opponent(player),
                    False,            # next is minimizing
                    depth - 1,
                    alpha,
                    beta
                )

                if child_val > value:
                    value = child_val
                    best_move = move

                # Update alpha
                alpha = max(alpha, value)

                # Prune
                if beta <= alpha:
                    break

            return value, best_move

        # ------------------------------------------------------------
        # Minimizing node (O)
        # ------------------------------------------------------------

        else:
            value = math.inf

            for move in moves:
                next_state = self.game.make_move(state, move, player)

                child_val, _ = self._alphabeta(
                    next_state,
                    self.game.get_opponent(player),
                    True,             # next is maximizing
                    depth - 1,
                    alpha,
                    beta
                )

                if child_val < value:
                    value = child_val
                    best_move = move
```

```
                # Update beta
                beta = min(beta, value)

                # Prune
                if beta <= alpha:
                    break

            return value, best_move
```

Can You Guess the Outcome?

In [52]:
```
runner = GameRunner(game)

mm = MinimaxSolverV2()
ab = MinimaxAlphaBetaSolverV2()

outcome = runner.play_game(mm, ab)

print("Nodes visited for MinimaxSolverV2:", mm.nodes_visited)
print("Nodes visited for MinimaxAlphaBetaSolverV2:", ab.nodes_visited)
print(f"Pruning efficiency: {(mm.nodes_visited - ab.nodes_visited) / mm.node
```

```
Nodes visited for MinimaxSolverV2: 557492
Nodes visited for MinimaxAlphaBetaSolverV2: 2435
Pruning efficiency: 0.996
```

In tic-tac-toe, the total number of valid action sequences equates to 255,168, corresponding to the number of leaf nodes in the game tree.

Here, we present the total count of nodes visited, encompassing both internal nodes and leaf nodes, thereby accounting for the higher aggregate number.

Memoization

In [53]:
```
class AlphaBetaSolver(Solver):

    """
    A classical Minimax solver enhanced with Alpha-Beta pruning.

    - Assumes "X" is the maximizing player.
    - Uses memoization (lru_cache) to avoid recomputing states.
    - Performs a *full* search of Tic-Tac-Toe (depth=9).
    - Returns the optimal move for the current player.
    """

    # ------------------------------------------------------------
    # Solver interface
    # ------------------------------------------------------------

    def select_move(self, game, state, player):

        """
```

```
        Public interface required by Solver.
        Runs Alpha-Beta search from the current state.
        """

        self.game = game

        maximizing = (player == "X")   # X maximizes, O minimizes

        # Reset cache between games to avoid storing millions of keys
        self._alphabeta.cache_clear()

        value, move = self._alphabeta(
            canonical(state),
            player,
            maximizing,
            9,               # full-depth search
            -math.inf,       # alpha
            math.inf         # beta
        )
        return move

    # ------------------------------------------------------------
    # Internal alpha-beta with memoization
    # ------------------------------------------------------------

    @lru_cache(maxsize=None)
    def _alphabeta(self, state_key, player, maximizing, depth, alpha, beta):

        """
        Parameters
        ----------
        state_key : tuple-of-tuples board
        player    : player whose turn it is ('X' or 'O')
        maximizing: True if this node is a maximizing node for X
        depth     : remaining depth
        alpha     : best guaranteed value for maximizer so far
        beta      : best guaranteed value for minimizer so far
        """

        state = np.array(state_key)

        # Terminal or horizon case
        if self.game.is_terminal(state) or depth == 0:
            return self.game.evaluate(state), None

        moves = self.game.get_valid_moves(state)
        best_move = None

        # ------------------------------------------------------------
        # MAX (X)
        # ------------------------------------------------------------

        if maximizing:
            value = -math.inf

            for move in moves:
```

```
                st2 = self.game.make_move(state, move, player)

                child_val, _ = self._alphabeta(
                    canonical(st2),
                    self.game.get_opponent(player),
                    False,              # now minimizing
                    depth - 1,
                    alpha,
                    beta
                )

                if child_val > value:
                    value = child_val
                    best_move = move

                alpha = max(alpha, value)
                if beta <= alpha:
                    break  # β-cutoff

            return value, best_move

        # ------------------------------------------------------
        # MIN (O)
        # ------------------------------------------------------

        else:
            value = math.inf

            for move in moves:
                st2 = self.game.make_move(state, move, player)

                child_val, _ = self._alphabeta(
                    canonical(st2),
                    self.game.get_opponent(player),
                    True,               # now maximizing
                    depth - 1,
                    alpha,
                    beta
                )

                if child_val < value:
                    value = child_val
                    best_move = move

                beta = min(beta, value)
                if beta <= alpha:
                    break  # α-cutoff

            return value, best_move
```

For our notes only.

Exploration

Implement a **Connect Four** game using the **Alpha-Beta Search algorithm**. Conduct a comparative analysis between the Minimax and Alpha-Beta Search implementations.

Prologue

Further exploration

- **Expetimax** search: handling players that are not perfect;
- **Expectiminimax**: handling chance in games such as backgammon.

Summary

- Introduction to adversarial search
- Zero-sum games
- Introduction to the minimax search method
- Role of alpha and beta pruning in minimax search
- **Introduction to Adversarial Search:**
  - Exploration of competitive environments with conflicting objectives.
- **Classification of Games:**
  - Types based on determinism (deterministic vs. stochastic).
  - Number of players (one, two, or more).
  - Nature of competition (zero-sum vs. non-zero-sum).
  - Availability of information (perfect vs. imperfect information).
- **Zero-Sum Games:**
  - Definition and characteristics.
  - Example: Tic-Tac-Toe as a zero-sum game.
- **Deterministic Games Framework:**
  - Components: states, players, actions, transition functions, final states, rewards.
  - Development of policies from initial to final states.
- **Game Strategies and Complexity:**
  - Analysis of never-lose strategies in Tic-Tac-Toe.
  - Discussion on the impact of move order (first or second player).
  - Exploration of game complexity in Tic-Tac-Toe, Chess, and Go.
- **Optimal Play and Perfect Information:**
  - Concepts of optimal strategies and their implications.

    - Importance of perfect information in game theory.
- **Minimax Algorithm:**
    - Introduction to the minimax search method.
    - Application in determining optimal moves in adversarial games.
    - Implementation details with a Python example for Tic-Tac-Toe.
- **Efficiency Improvements:**
    - Use of caching (memoization) to enhance algorithm performance.
    - Reduction of computational overhead in game search trees.
- **Pruning Techniques:**
    - Introduction to pruning in search trees to avoid unnecessary computations.
    - Detailed explanation of Alpha-Beta pruning.
    - Criteria for pruning and walkthrough examples illustrating the process.
- **Alpha-Beta Pruning:**
    - Integration with the minimax algorithm.
    - Role of alpha and beta parameters in optimizing search.
    - Impact on the number of nodes evaluated.
- **Performance Comparison:**
    - Analysis of node exploration between minimax and alpha-beta pruning.
    - Quantitative demonstration of efficiency gains.
- **Node Ordering and Pruning Effectiveness:**
    - Discussion on how node evaluation order affects pruning success.
    - Strategies for ordering nodes to maximize pruning potential.

Next lecture

- We will look at the Monte Carlo Tree Search (MCTS) algorithm

References

Russell, Stuart, and Peter Norvig. 2020. *Artificial Intelligence: A Modern Approach*. 4th ed. Pearson. http://aima.cs.berkeley.edu/.

Shannon, Claude E. 1959. “Programming a Computer Playing Chess.” *Philosophical Magazine* Ser.7, 41 (312).

Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

School of Electrical Engineering and **Computer Science** (EE**CS**)

University of Ottawa