Recherche adversariale

CSI 4106 - Automne 2025

Marcel Turcotte
Version: nov. 26, 2025 10h49

Préambule

Message du jour

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

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

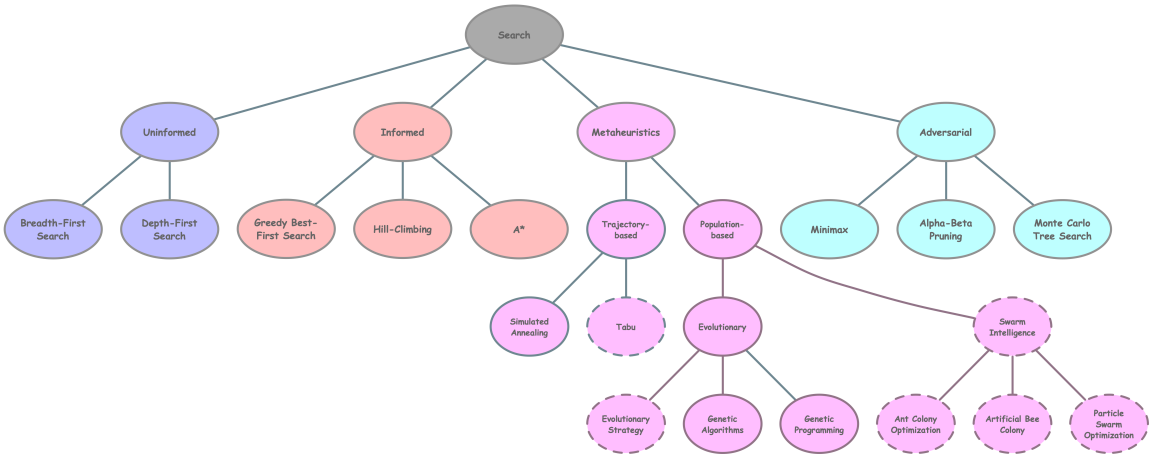
Recherche adversariale

Cette présentation examine les **environnements compétitifs** où plusieurs agents ont des objectifs conflictuels, ce qui entraîne des **problèmes de recherche adversariale**.

Objectifs d'apprentissage

- **Expliquer** le concept de jeu à somme nulle (*zero-sum game*)
- **Formuler** des stratégies pour ne jamais perdre au Tic-Tac-Toe, quel que soit le coup de l'adversaire
- **Utiliser** l'algorithme **minimax** pour déterminer les coups optimaux dans des contextes adversariaux
- **Articuler** comment l'**élagage alpha-bêta** réduit le nombre de nœuds évalués sans affecter les résultats

Recherche



Notre examen des algorithmes de recherche a progressé à travers plusieurs étapes, en commençant par les techniques de recherche non informée, en avançant vers les méthodes de recherche informée, et en explorant ensuite les approches métaheuristiques. Nous nous tournons maintenant vers l'étude des algorithmes de recherche adversariale.

La recherche arborescente de Monte Carlo (MCTS), bien qu'elle ne soit pas intrinsèquement conçue comme un algorithme de recherche adversarial, construit un arbre de recherche partiel en simulant de manière itérative des actions à partir d'un état donné. Néanmoins, lorsqu'elle est appliquée à des jeux à deux joueurs ou à plusieurs joueurs, la MCTS peut efficacement aborder les problèmes de recherche adversarial.

Architecture Commune de Jeu

Motivation

Cette présentation et la suivante utiliseront le **tic-tac-toe** pour explorer des concepts liés à la **recherche adversariale** et à **l'apprentissage par renforcement**.

Nous allons implémenter et comparer plusieurs stratégies de résolution, notamment une **politique aléatoire**, trois variantes du **minimax**, trois variantes de l'**élagage alpha-bêta**, ainsi que deux algorithmes de **recherche d'arbre de Monte-Carlo**.

Tous les solveurs seront développés dans un cadre **généraliste** conçu pour prendre en charge une large gamme de jeux à deux joueurs, tels que Puissance 4.

Une large catégorie de jeux de plateau classiques et de jeux combinatoires correspond précisément à l'interface Game, caractérisée par une dynamique déterministe, des tours alternés, une interaction à deux joueurs, des résultats à somme nulle et une information parfaite. De tels jeux se prêtent à des approches algorithmiques telles que Minimax, l'élagage Alpha-Beta et la recherche arborescente Monte Carlo (MCTS).

- Puissance 4 (Connect Four)
- Gomoku
- Othello / Reversi
- Nim
- Prendre-la-dernière-pièce (Take-the-Last-Coin)
- Hex
- Échecs
- Shogi
- Go

Les échecs, le shogi et le Go présentent des facteurs de branchement élevés, ce qui entraîne des exigences computationnelles importantes. Par conséquent, leur complexité les rend moins appropriés comme exemples pédagogiques dans le cadre de ce cours.

Game

In [1]:
```
class Game:

    """
    Interface abstraite pour un jeu déterministe, à deux joueurs, à somme nu
    avec alternance de tours.

    Conventions (utilisées par le Tic-Tac-Toe et les solveurs ci-dessous) :
    - Les joueurs sont identifiés par les chaînes de caractères "X" et "O".
    - evaluate(state) retourne :
        > 0  si la position est favorable à "X"
        < 0  si la position est favorable à "O"
        == 0 pour une égalité ou une position non terminale égale
    """

    def initial_state(self):

        """Retourne un objet représentant la position de départ du jeu."""

        raise NotImplementedError

    def get_valid_moves(self, state):

        """
        Étant donné un état, retourne un itérable des coups légaux.
        Le type de 'move' dépend du jeu (par exemple, (row, col) pour le Tic
        """

        raise NotImplementedError

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

        """
        Retourne l'état successeur obtenu en appliquant 'move' pour 'player'
        à 'state'. L'état original ne doit pas être modifié en place.
```

```
        """

        raise NotImplementedError

    def get_opponent(self, player):

        """Retourne l'adversaire de 'player'."""

        raise NotImplementedError

    def is_terminal(self, state):

        """
        Retourne True si 'state' est une position terminale (victoire, défai
        False sinon.
        """

        raise NotImplementedError

    def evaluate(self, state):

        """
        Retourne une évaluation scalaire de 'state' :
            +1 pour une victoire de X, -1 pour une victoire de O, 0 sinon (p
        Pour d'autres jeux, ceci peut être généralisé, mais ici nous gardons
        """

        raise NotImplementedError

    def display(self, state):

        """Affiche une représentation lisible par un humain de 'state' (pour

        raise NotImplementedError
```

Game

In [2]:

```
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
```

Bibliothèques requises

In [3]:
```
import math
import random
import numpy as np
import matplotlib.pyplot as plt
```

TicTacToe

In [4]:
```
class TicTacToe(Game):

    """
    Implémentation classique du Tic-Tac-Toe 3x3 utilisant un tableau NumPy c
    Les cases vides sont représentées par " ".
    Le joueur "X" est supposé être le joueur maximisant.
    """

    def __init__(self):
        self.size = 3

    def initial_state(self):

        """Retourne un plateau vide 3x3."""

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

    def get_valid_moves(self, state):

        """Tous les couples (i, j) où la case du plateau est vide."""

        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):

        """
        Retourne un nouveau plateau avec 'player' placé à 'move' (ligne, col
        L'état original n'est pas modifié.
        """

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

```
        return new_state

    def get_opponent(self, player):

        """Alterner entre les étiquettes de joueur 'X' et 'O'."""

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

    def is_terminal(self, state):

        """
        Un état est terminal si :
        - L'un des joueurs a une ligne de 3 (evaluate != 0), ou
        - Il n'y a plus de cases vides (égalité).
        """

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

    def evaluate(self, state):

        """
        Retourne +1 si X a trois en ligne, -1 si O a trois en ligne,
        et 0 sinon (y compris pour les états non terminaux et les égalités).

        Il s'agit d'une évaluation "théorique du jeu" pour les états termina
        pour les positions non terminales, on retourne simplement 0.
        """

        lines = []

        # Lignes et colonnes
        for i in range(self.size):
            lines.append(state[i, :])   # ligne i
            lines.append(state[:, i])   # colonne i

        # Diagonales principales
        lines.append(np.diag(state))
        lines.append(np.diag(np.fliplr(state)))

        # Vérifier chaque ligne pour une victoire
        for line in lines:
            if np.all(line == "X"):
                return 1
            if np.all(line == "O"):
                return -1
        return 0

    def display(self, state):

        """
        Visualiser un plateau de Tic-Tac-Toe avec matplotlib.

        Paramètres
        ----------
```

```
        state : np.ndarray de forme (size, size)
            Plateau contenant ' ', 'X' ou 'O'.
        """

        size = self.size

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

        # Tracer les lignes de la grille
        for i in range(1, size):
            ax.axhline(i, color='black')
            ax.axvline(i, color='black')

        # Masquer complètement les axes
        ax.axis('off')

        # Dessiner les symboles X et O
        for i in range(size):
            for j in range(size):
                cx = j + 0.5
                cy = size - i - 0.5     # inverser l'axe y pour la bonne ori

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

    """
    Implémentation classique du Tic-Tac-Toe 3x3 utilisant un tableau NumPy c
    Les cases vides sont représentées par " ".
    Le joueur "X" est supposé être le joueur maximisant.
    """

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

`TicTacToe` est un `Game`

initial_state

In [6]:
```python
    def initial_state(self):

        """Retourne un plateau vide de 3x3."""

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

get_valid_moves

In [7]:
```python
    def get_valid_moves(self, state):

        """Tous les couples (i, j) où la case du plateau est vide."""

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

make_move

In [8]:
```python
    def make_move(self, state, move, player):

        """
        Retourne un nouveau plateau avec le 'joueur' placé à 'move' (ligne,
        L'état original n'est pas modifié.
        """

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

        return new_state
```

get_opponent

In [9]:
```python
    def get_opponent(self, player):

        """Inverse les étiquettes des joueurs entre 'X' et 'O'."""

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

evaluate

In [10]:
```python
    def evaluate(self, state):

        lignes = []

        # Lignes et colonnes
        for i in range(self.size):
            lignes.append(state[i, :])   # ligne i
            lignes.append(state[:, i])   # colonne i

        # Diagonales principales
        lignes.append(np.diag(state))
        lignes.append(np.diag(np.fliplr(state)))

        # Vérifier chaque ligne pour une victoire
        for ligne in lignes:
            if np.all(ligne == "X"):
                return 1
            if np.all(ligne == "O"):
                return -1
        return 0
```

is_terminal

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

        """
        Un état est terminal si :
        - L'un des joueurs a aligné trois symboles (evaluate != 0), ou
        - Il n'y a plus de cases vides (match nul).
        """

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

display

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

        """
        Visualiser un plateau de Tic-Tac-Toe avec matplotlib.

        Paramètres
        ----------
        state : np.ndarray de forme (size, size)
            Plateau contenant ' ', 'X' ou 'O'.
        """

        size = self.size

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

        # Tracer les lignes de la grille
        for i in range(1, size):
            ax.axhline(i, color='black')
            ax.axvline(i, color='black')

        # Masquer complètement les axes
        ax.axis('off')

        # Dessiner les symboles X et O
        for i in range(size):
            for j in range(size):
                cx = j + 0.5
                cy = size - i - 0.5     # inverser l'axe y pour une bonne or

                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 [13]:
```
class Solver:
```

```
    """
    Classe de base pour tous les solveurs (Aléatoire, Minimax, AlphaBeta, MC

    Les solveurs doivent implémenter :
        - select_move(game, state, player)

    Les solveurs peuvent optionnellement implémenter :
        - reset()           : appelée au début de chaque partie
        - opponent_played() : utilisée par les solveurs persistants (ex. : M

    Remarques
    ---------
    • Les solveurs peuvent conserver un état interne persistant d'un coup à
    • GameRunner peut appeler automatiquement reset() avant chaque match.
    """

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

        """
        Doit être implémentée par les sous-classes.
        Retourne un coup légal pour le joueur donné.
        """

        raise NotImplementedError

    def get_name(self):

        """
        Retourne le nom du solveur pour les rapports, les journaux ou les ré

        Par défaut, retourne le nom de la classe, mais les solveurs peuvent
        cette méthode pour inclure des paramètres (par ex. : "MCTS(num_simul
        """

        return self.__class__.__name__

    def opponent_played(self, move):
        """
        Optionnel. Appelée après un coup de l'adversaire.
        Utile pour les solveurs à état comme MCTS.
        Les solveurs sans état peuvent l'ignorer.
        """
        pass

    def reset(self):

        """
        Optionnel. Appelée une fois au début de chaque partie.
        À surcharger uniquement si le solveur maintient un état interne
        (par ex. : arbre MCTS, analyse en cache, tables heuristiques).
        """

        pass
```

Solver

In [14]:
```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 [15]:
```python
class RandomSolver(Solver):

    """
    Un solveur de référence simple :
    - À chaque coup, choisit uniformément au hasard parmi tous les coups lég
    - Ne maintient aucun état interne (aucun apprentissage).
    """

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

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

        """Retourne un coup légal choisi au hasard pour le joueur courant."""

        moves = game.get_valid_moves(state)

        return self.rng.choice(moves)

    def opponent_played(self, move):

        """Le solveur aléatoire n'a aucun état interne à mettre à jour."""

        pass
```

In [16]:
```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)
```

Exécute un jeu **aléatoire** avec des résultats **reproductibles**.

Général et **applicable** à une large classe de jeux au-delà du tic-tac-toe.

GameRunner

In [17]:
```python
class GameRunner:

    """
    Utilitaire pour exécuter une seule partie entre deux solveurs sur un jeu

    Cette classe est volontairement simple : elle alterne les coups entre "X
    jusqu'à ce qu'un état terminal soit atteint.
    """

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

    def play_game(self, solver_X, solver_O):

        """
        Joue une partie complète :
        - solver_X contrôle le joueur "X"
        - solver_O contrôle le joueur "O"

        Retourne
        --------
        result : int
            +1 si X gagne, -1 si O gagne, 0 pour une égalité.
        """

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

        # Joue jusqu'à une position terminale
        while not self.game.is_terminal(state):
            # Le joueur actuel sélectionne un coup
            move = solvers[player].select_move(self.game, state, player)

            # Applique le coup
            state = self.game.make_move(state, move, player)
```

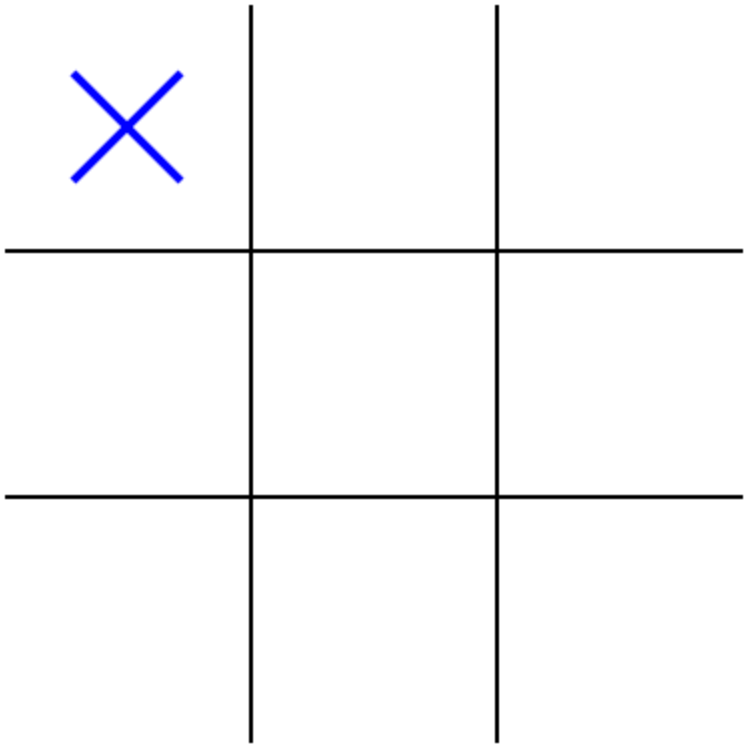
```
            if self.verbose:
                self.game.display(state)

            # Notifie l'adversaire (pour les solveurs persistants comme MCTS
            opp = self.game.get_opponent(player)
            solvers[opp].opponent_played(move)

            # Change de joueur actif
            player = opp

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

        # Évaluation finale du point de vue de X
        return self.game.evaluate(state)
```

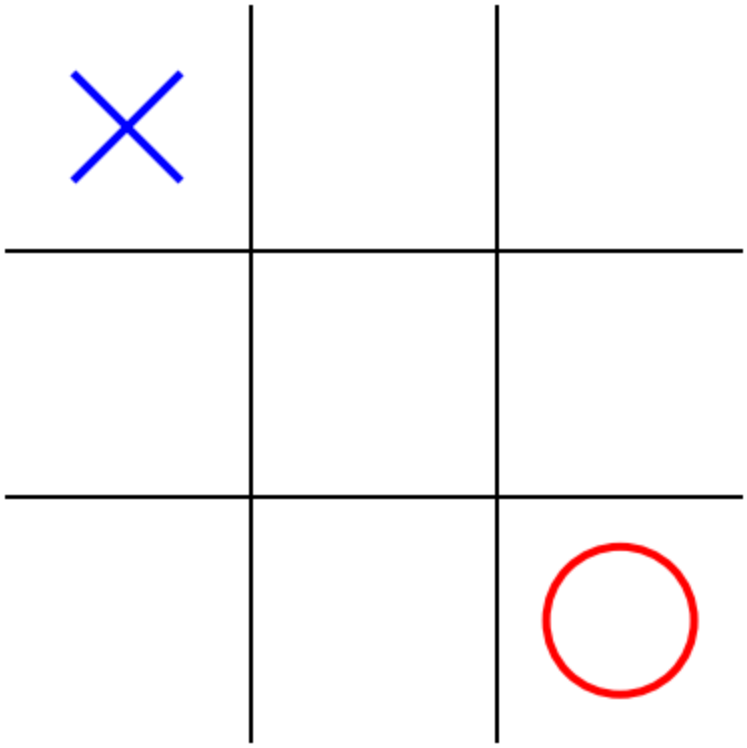
GameRunner

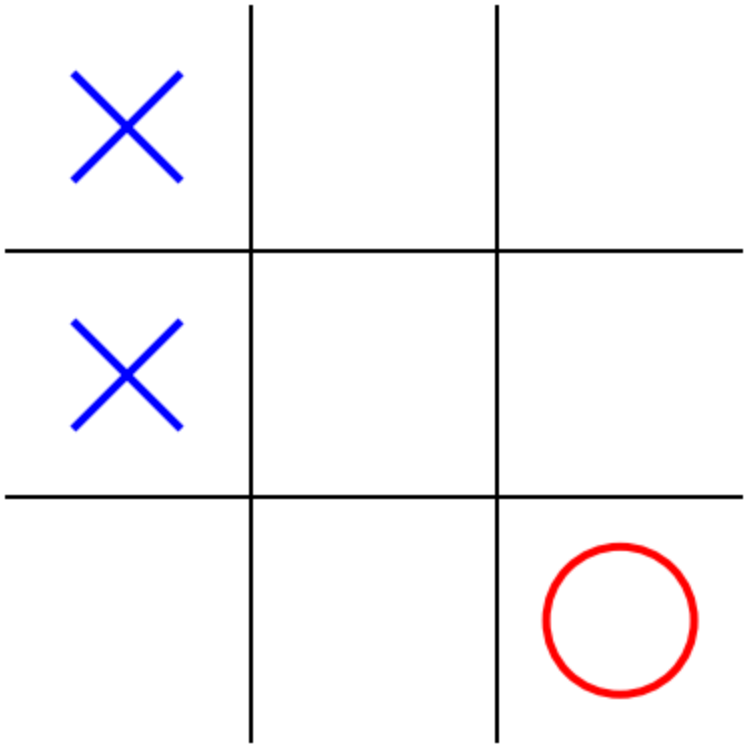
In [18]:
```
class GameRunner:

    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)
```

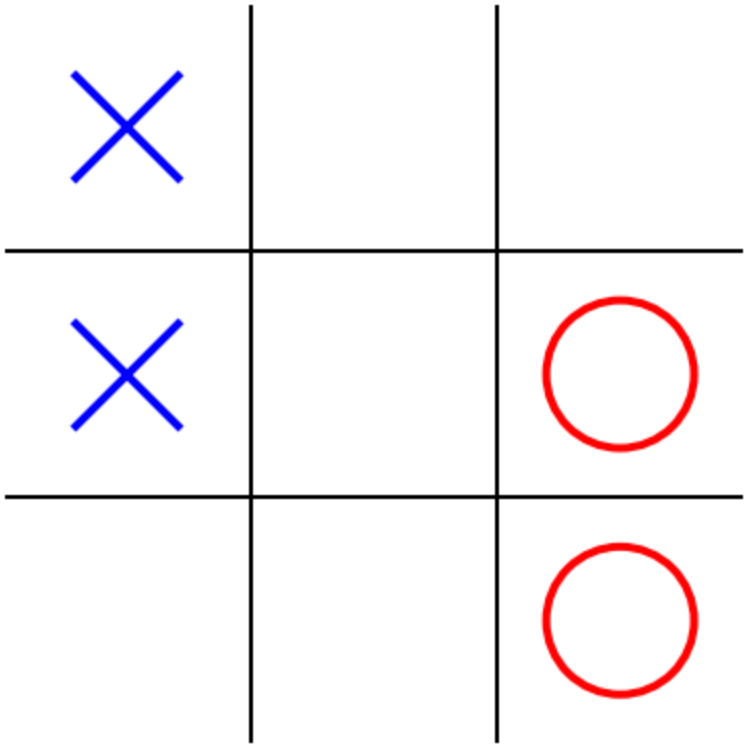
Exemple

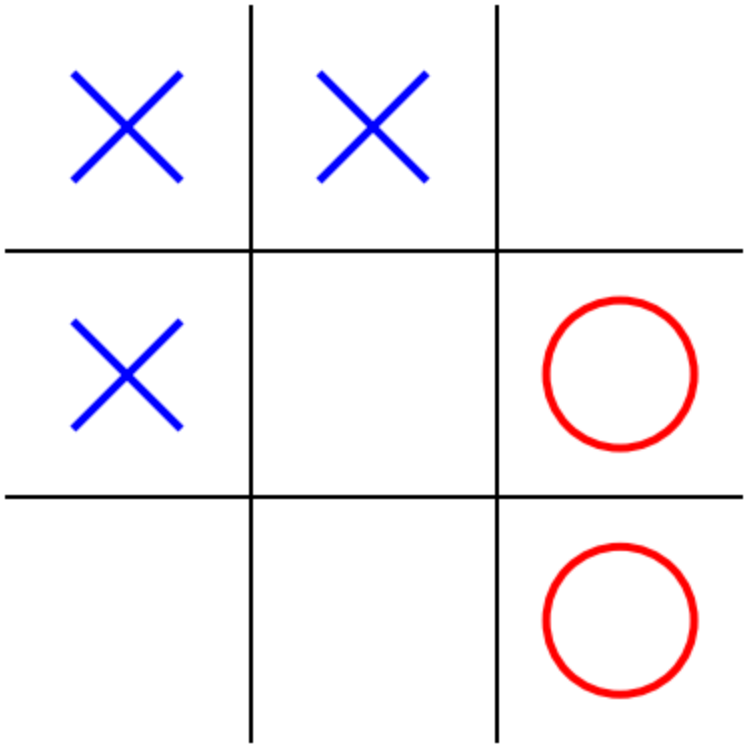
In [19]:
```
jeu = TicTacToe()

gestionnaire = GameRunner(jeu, verbose=True)

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

resultat = gestionnaire.play_game(a, b)
```

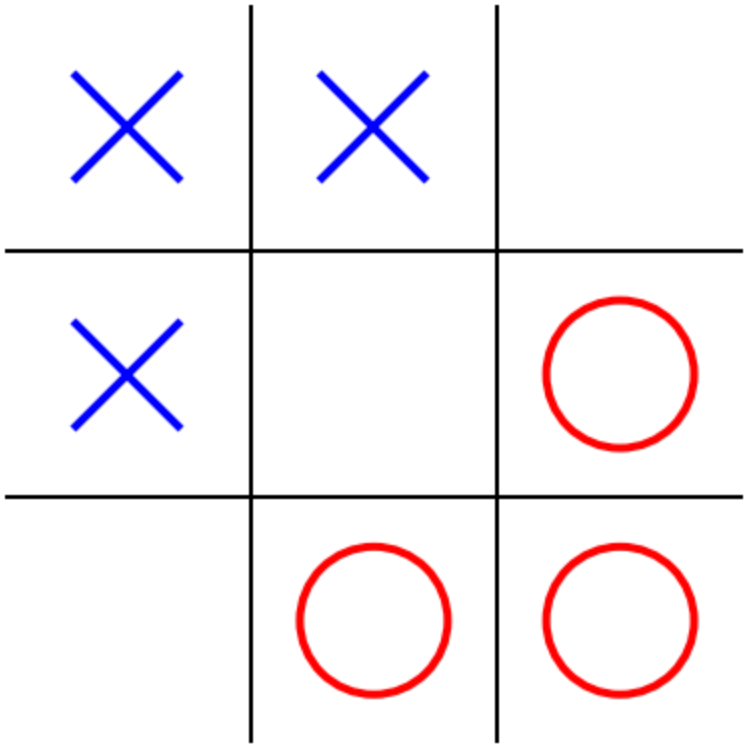


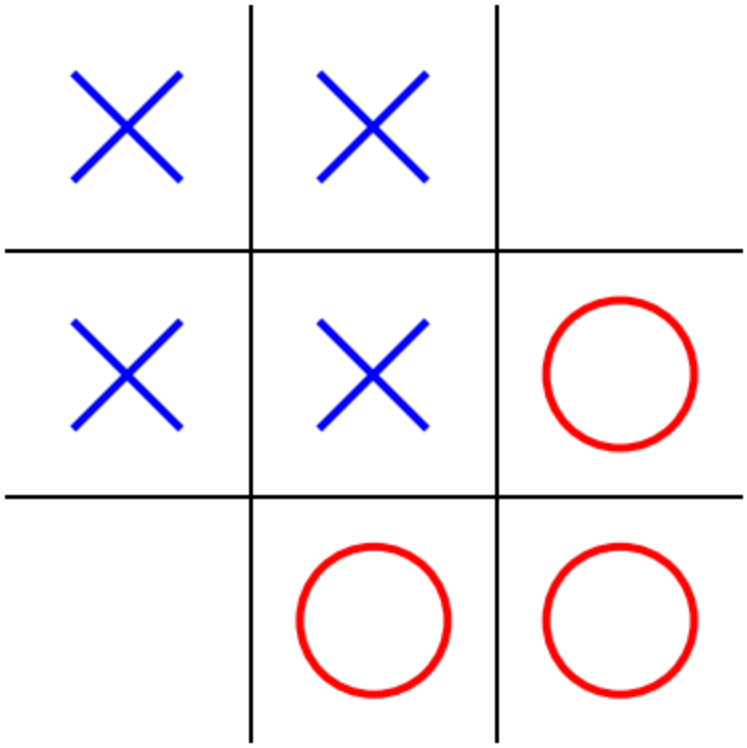


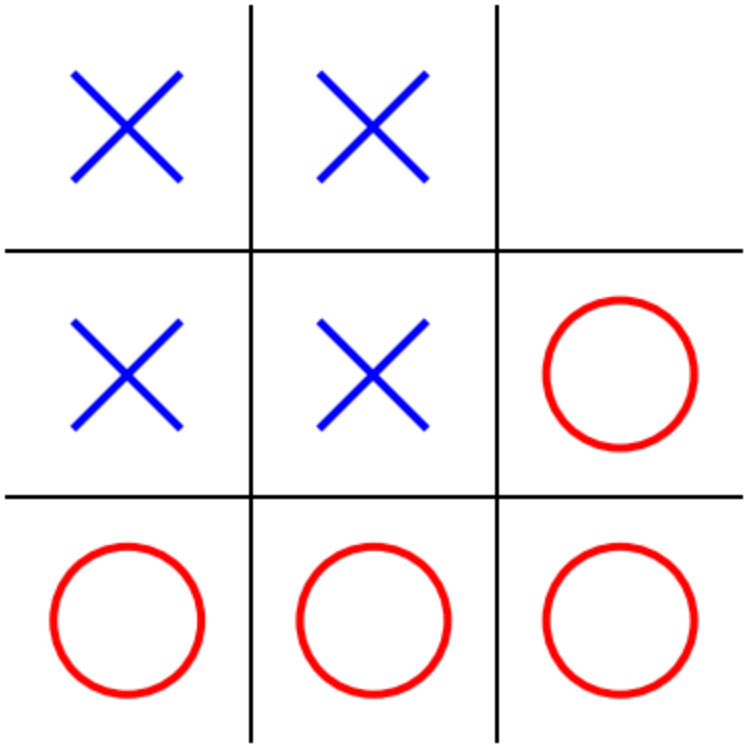












-1

evaluate_solvers

In [20]:
```
def evaluate_solvers(game, solver_X, solver_O, num_games, verbose=False):
    """
    Évalue deux solveurs l'un contre l'autre sur un jeu donné.
```

```
    Paramètres
    ----------
    game      : Game
        Une instance d'un jeu (par exemple, TicTacToe).
    solver_X  : Solver
        Solveur contrôlant le joueur "X" (le joueur maximisant).
    solver_O  : Solver
        Solveur contrôlant le joueur "O" (le joueur minimisant).
    num_games : int
        Nombre de parties à jouer avec ces rôles fixes.

    Remarques
    ---------
    - Les mêmes instances de solveurs sont réutilisées à travers les parties
      Cela permet aux solveurs *persistants* (par exemple, MCTS) d'accumuler
      de l'expérience d'une partie à l'autre.
    - Les résultats sont interprétés du point de vue de X :
        +1 -> X gagne
        -1 -> O gagne
         0 -> égalité
    """

    runner = GameRunner(game)

    # Agréger les statistiques sur toutes les parties
    results = {
        "X_wins": 0,
        "O_wins": 0,
        "draws": 0,
    }

    for i in range(num_games):
        # Joue une partie avec solver_X comme "X" et solver_O comme "O"
        outcome = runner.play_game(solver_X, solver_O)

        # Met à jour les compteurs selon le résultat (+1, -1 ou 0)
        if outcome == 1:
            results["X_wins"] += 1
            if verbose:
                print(f"Partie {i + 1} : X gagne")
        elif outcome == -1:
            results["O_wins"] += 1
            if verbose:
                print(f"Partie {i + 1} : O gagne")
        else:
            results["draws"] += 1
            if verbose:
                print(f"Partie {i + 1} : Égalité")

    # Affiche le résumé final
    if verbose:
        print(f"\nAprès {num_games} parties :")
        print(f"  X ({solver_X.get_name()}) victoires : {results['X_wins']}"
        print(f"  O ({solver_O.get_name()}) victoires : {results['O_wins']}"
        print(f"  Égalités : {results['draws']}")
```

```
    return results
```

La méthode exécute un nombre spécifié de parties complètes, en utilisant les algorithmes de résolution désignés pour chaque partie. Elle enregistre systématiquement les résultats de ces parties et rapporte les mesures de performance agrégées.

Pouvez-vous deviner le résultat ?

In [21]:
```
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_victoires : $\frac{1}{3}$, O_victoires : $\frac{1}{3}$, nulles : $\frac{1}{3}$ ou X_victoires : $\frac{2}{5}$, O_victoires : $\frac{2}{5}$, nulles : $\frac{1}{5}$ ou autre chose ?

Ce résultat est-il cohérent avec vos attentes ? Veuillez fournir une justification. Nous reviendrons sur ce résultat en détail sous peu.

Introduction

Types de jeux

- **Déterministes** ou **stochastiques**
- **Un**, **deux**, ou **plusieurs** joueurs
- **À somme nulle** ou non
- **À information parfaite** ou non

Définition

Les **jeux à somme nulle** sont des situations de compétition où le **gain** d'un joueur est exactement compensé par la **perte** d'un autre joueur, **entraînant ainsi un changement net nul** de la richesse ou du bénéfice total.

Le tic-tac-toe est un jeu à somme nulle.

Avez-vous déjà étudié les jeux à somme nulle ? Si oui, veuillez préciser le contexte (par exemple, cours, recherche ou travail appliqué) ainsi que le niveau de traitement

(introductif, avancé ou appliqué).

Jeux déterministes

- **États** : S (S_0 à S_k)
- **Joueurs** : $P = \{1, N\}$
- **Actions** : A (dépend de P et de S)
- **Fonction de transition** : $S \times A \to S$
- **Un état final** : S_{final}
- **Récompense ou utilité** : S_{final}, p

Développer une politique $S_0 \to S_{\text{final}}$.

Contrairement aux formulations précédentes de recherche dans l'espace d'états, nous introduisons explicitement un paramètre joueur. Par conséquent, la fonction de transition est conditionnée à la fois sur l'état courant et sur l'identité du joueur (plutôt que seulement sur l'état).

Qu'est-ce qu'une **politique** ? En quoi diffère-t-elle de la **fonction de transition** ?

La **fonction de transition** décrit la mécanique du jeu.

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

Elle fait partie de la *définition du jeu*.

Elle indique **ce qui se passe** lorsqu'on effectue l'action a dans l'état s :

- Elle est fixée par les règles du jeu.
- Elle ne dépend **pas** de la manière dont un joueur particulier souhaite jouer.
- Exemple (Tic-Tac-Toe) : placer un X dans une case vide met à jour le plateau en conséquence.

La **politique** décrit le comportement du joueur.

$$\pi : S \to A$$

Elle fait partie de la *solution*.

Elle indique au joueur **quelle action choisir** dans chaque état qu'il peut rencontrer.

- Elle dépend de la stratégie, des préférences, de l'optimalité.
- Elle utilise la fonction de transition, mais n'est pas définie par celle-ci.
- Exemple : « Si le centre est libre, jouer là ; sinon jouer un coin. »

Concept	Ce qu'il décrit	Fixé ou choisi ?	Exemple
Fonction de transition T	Comment les états évoluent lorsqu'une action est prise	**Fixée par les règles du jeu**	Appliquer un coup met à jour le plateau

Concept	Ce qu'il décrit	Fixé ou choisi ?	Exemple
Politique π	Quelle action le joueur choisit dans chaque état	**Choisie (ex. stratégie optimale)**	Prendre toujours le centre si possible

Qu'en pensez-vous ?

- Pensez à jouer au **tic-tac-toe**.
- Pouvez-vous garantir **une stratégie infaillible**, quel que soit le jeu de votre adversaire ?

Est-ce important de jouer en premier ou en second ?

Quiconque a déjà joué au tic-tac-toe comprend qu'un joueur peut adopter une stratégie qui lui assure de ne jamais perdre.

En soi, cette affirmation est plutôt surprenante. Pourquoi ?

Combien de configurations de plateau existe-t-il ?

- Chaque case peut être occupée par un X, un O ou rester vide. Cela conduit à $3^9 = 19\,683$ configurations possibles du plateau.
- Toutes ces configurations sont-elles réalisables dans une partie valide ?

Déroulement de la partie.

- Toutes ces configurations ne sont pas des états de jeu valides, car certaines peuvent contenir un nombre impossible de X et de O ou ne pas respecter les règles du jeu.
- Une partie de tic-tac-toe peut se terminer par une victoire de l'un ou l'autre joueur, ou par une égalité. La partie la plus longue sans gagnant comporte 9 coups (plateau rempli).
- Les parties valides tiennent compte des règles selon lesquelles les joueurs alternent les tours, en commençant par X.
- Combien de séquences valides de coups existe-t-il ?

Est-ce qu'il y a plus de séquences valides de coups que de configurations possibles du plateau?

`count_valid_sequences`

In [22]:
```
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 [23]:
```
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
```

Compte tenu du nombre de séquences valides, n'est-il pas surprenant que le premier joueur puisse adopter une stratégie pour éviter de perdre.

Symétrie (Digression)

. . .

Le tic-tac-toe possède **8 transformations symétriques** (4 rotations et 4 réflexions).

En tenant compte de celles-ci, de nombreuses séquences de jeu qui diffèrent par l'ordre brut des coups deviennent équivalentes.

Le nombre de **séquences uniques de coups** est de **26 830**, tandis que le nombre de positions de plateau uniques est de **765**.

Exercice : écrivez un programme Python pour confirmer ce qui précède.

Arbre de recherche

La **taille de l'arbre de recherche** pour le jeu de tic-tac-toe est **relativement petite**, ce qui le rend approprié comme exemple fil conducteur dans les discussions ultérieures.

Comment cela se compare-t-il aux arbres de recherche pour les **échecs** et le **Go** ?

Arbre de recherche

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

Le **nombre estimé d'atomes dans l'univers observable** est d'environ 10^{78} à 10^{82}.

Échecs :

Les échecs se jouent sur un plateau relativement petit (8×8) mais avec une variété de pièces ayant des capacités de déplacement différentes.

En moyenne, chaque position offre environ 35 coups légaux.

En considérant une partie typique d'une longueur de 80 demi-coups (un demi-coup correspond à un coup joué par un joueur, donc 40 coups par joueur), le nombre total de parties possibles est estimé par la formule $35^{80} \sim 10^{123}$, ajustée à la baisse pour tenir compte des positions illégales et redondantes.

Go :

Le go se joue sur un plateau de 19×19, offrant 361 points où placer des pierres. Ainsi, le nombre total de parties possibles est de $361! \sim 10^{768}$.

Définition

Le jeu optimal consiste à exécuter le **meilleur coup possible à chaque étape** afin de maximiser ses chances de gagner, en supposant que l'adversaire joue également de manière optimale.

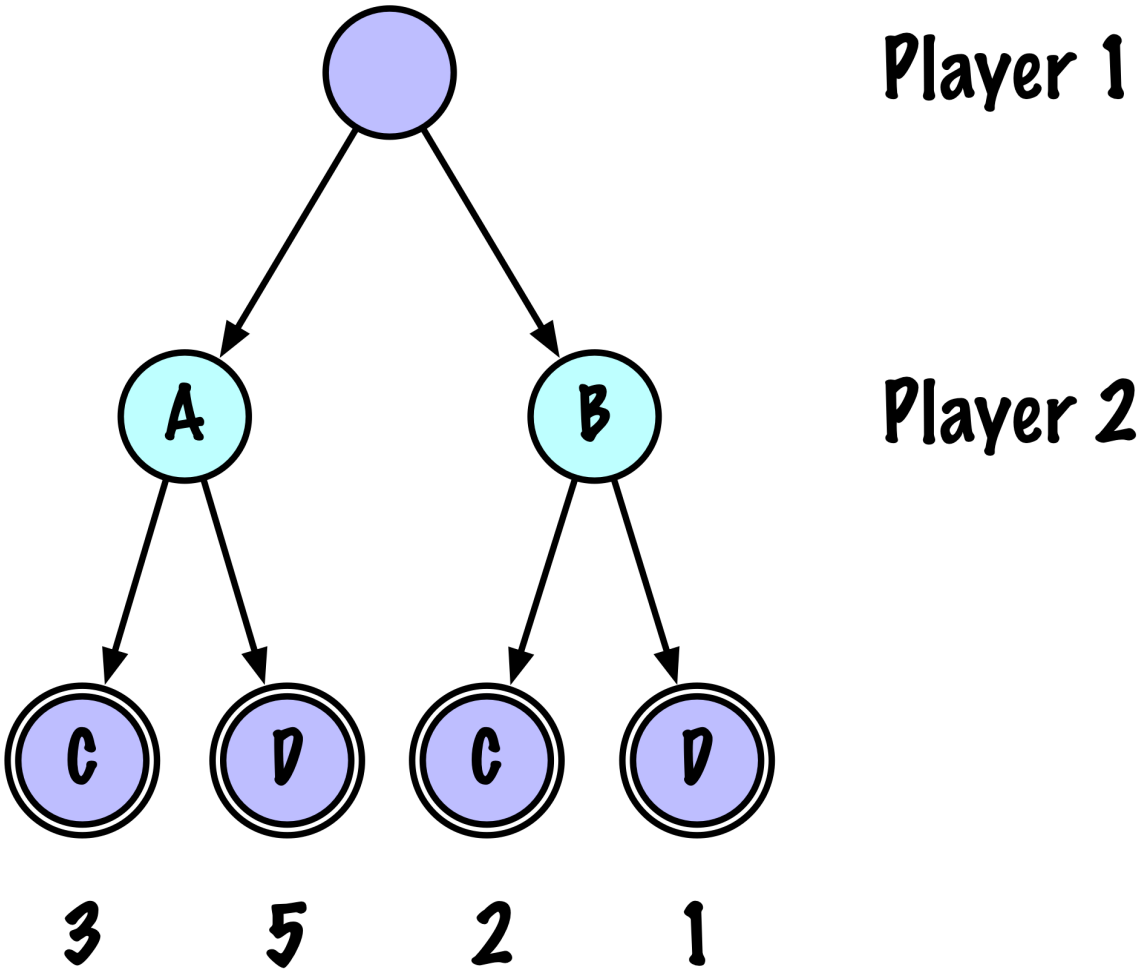
Dans les **jeux à information parfaite** comme le tic-tac-toe ou les échecs, cela nécessite d'anticiper les coups de l'adversaire et de choisir des actions qui améliorent sa propre position ou minimisent les pertes.

Lorsque **les deux joueurs adoptent des stratégies optimales**, l'issue du jeu, victoire, défaite ou match nul, est dictée par la mécanique inhérente du jeu et les conditions initiales.

L'information parfaite désigne une caractéristique de certains jeux ou scénarios de prise de décision où tous les joueurs disposent à tout moment d'une connaissance complète et précise de l'état du jeu. Cela inclut la visibilité totale de toutes les actions précédemment entreprises, sans éléments cachés ni aléatoires affectant la progression du jeu. Dans les jeux à information parfaite, tels que les échecs ou le tic-tac-toe, les joueurs peuvent prendre des décisions pleinement éclairées sur la base de l'historique complet et de l'état actuel du jeu, ce qui permet d'élaborer des stratégies sur plusieurs coups à l'avance.

Assurez-vous de bien comprendre comment l'optimalité est définie ici. Elle peut ne pas correspondre à votre intuition. Les algorithmes qui seront développés par la suite sont fondés sur cette définition.

Jeu à deux coups



Le Jeu à deux coups est un jeu hypothétique impliquant deux joueurs, conçu pour faciliter les discussions sur l'algorithme minimax.

Configuration du jeu

- Le **jeu commence** par un point de décision unique pour le Joueur 1, qui dispose de deux coups possibles : A et B.
- Chacun de ces coups conduit à un **point de décision pour le Joueur 2**, qui a également deux réponses possibles : C et D.
- Le **jeu se termine** après le coup du Joueur 2, menant à un état terminal avec des scores prédéfinis.

Arbre de recherche

- **Nœud racine** : Représente l'état initial avant le coup du Joueur 1.
- **Premier demi-coup** (*Ply 1*) : Le Joueur 1 choisit entre les coups A et B.

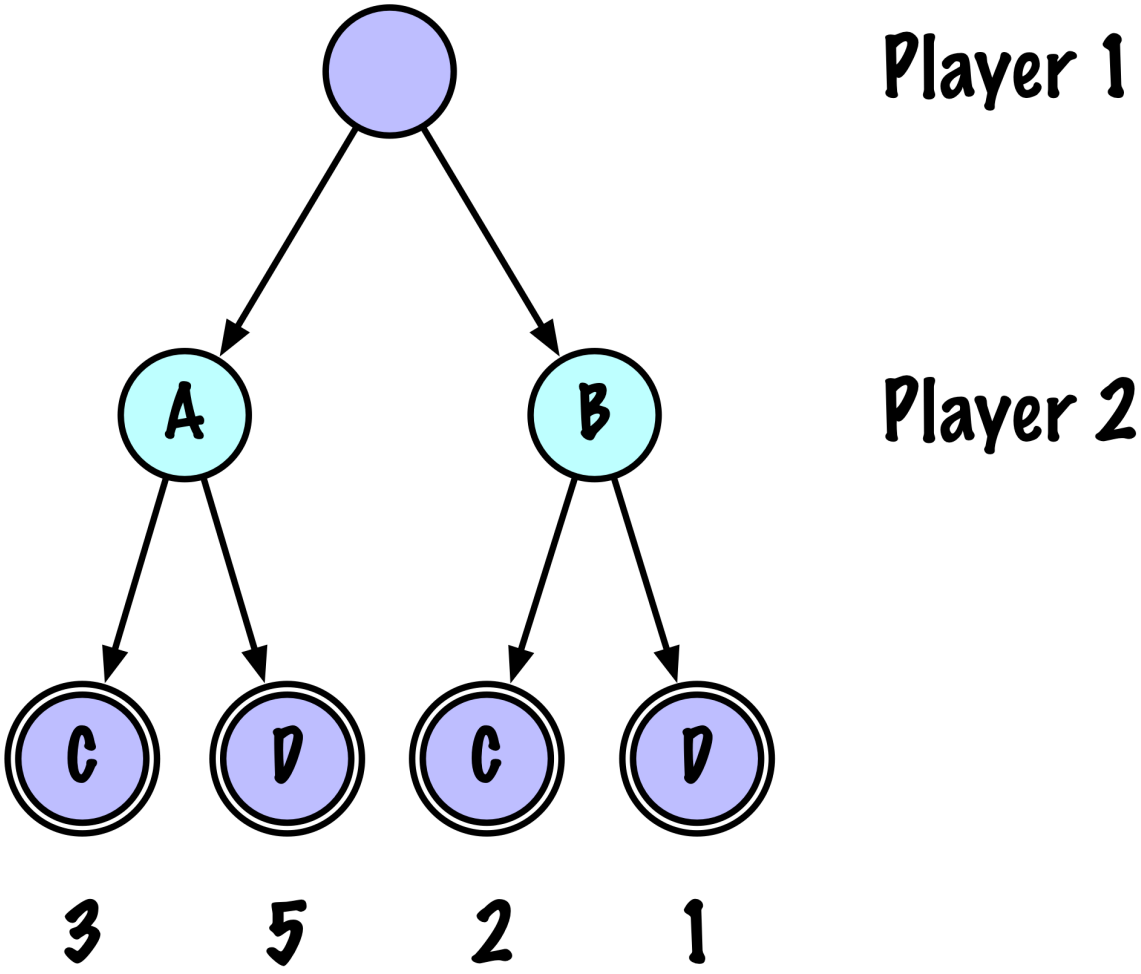
- **Deuxième demi-coup** (*Ply 2*) : Pour chacun des coups du Joueur 1, le Joueur 2 choisit entre les coups C et D.
- **Nœuds feuilles** : L'extrémité de chaque branche est un état terminal avec un score associé.

Scores

- (A, C) donne un score de 3.
- (A, D) donne un score de 5.
- (B, C) donne un score de 2.
- (B, D) donne un score de 1.

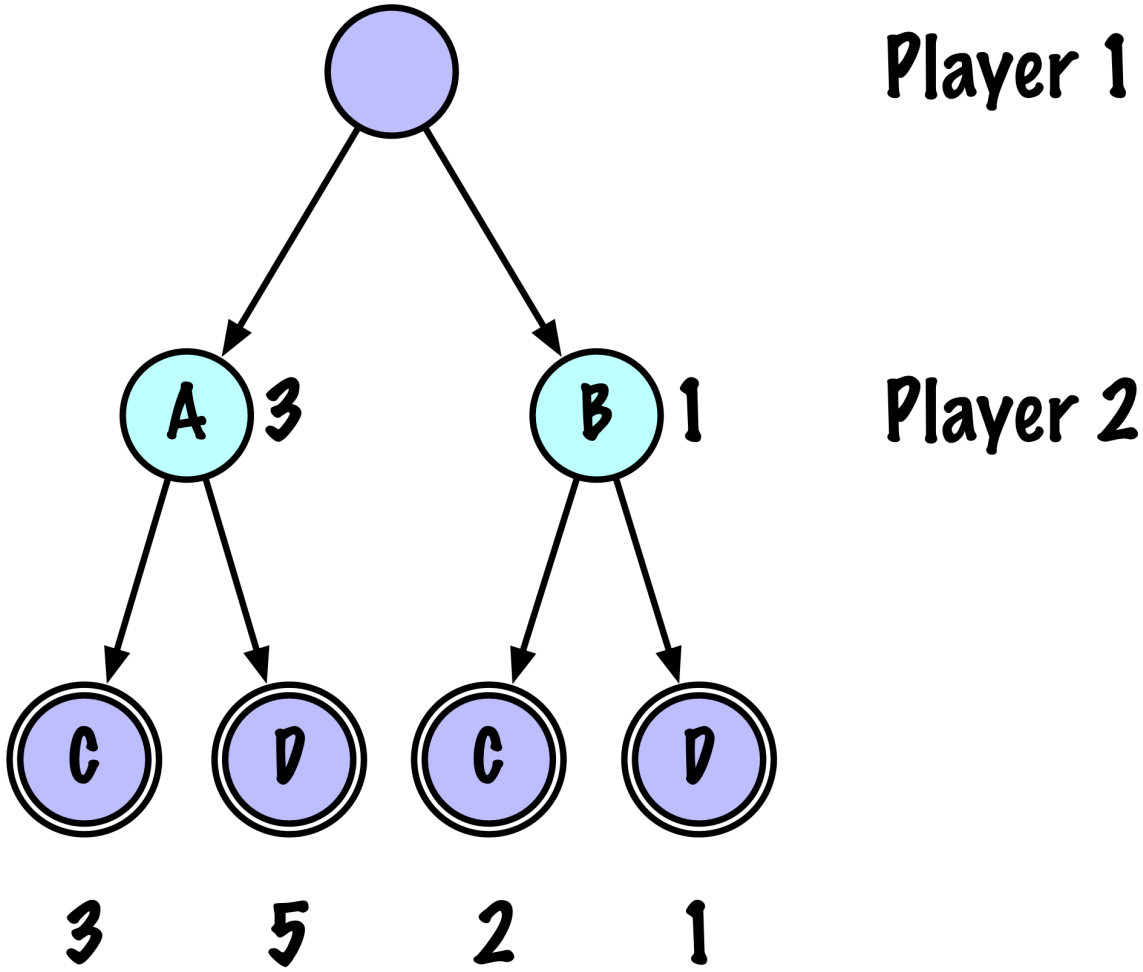
Le Joueur 1 cherche à maximiser son score.

Stratégie



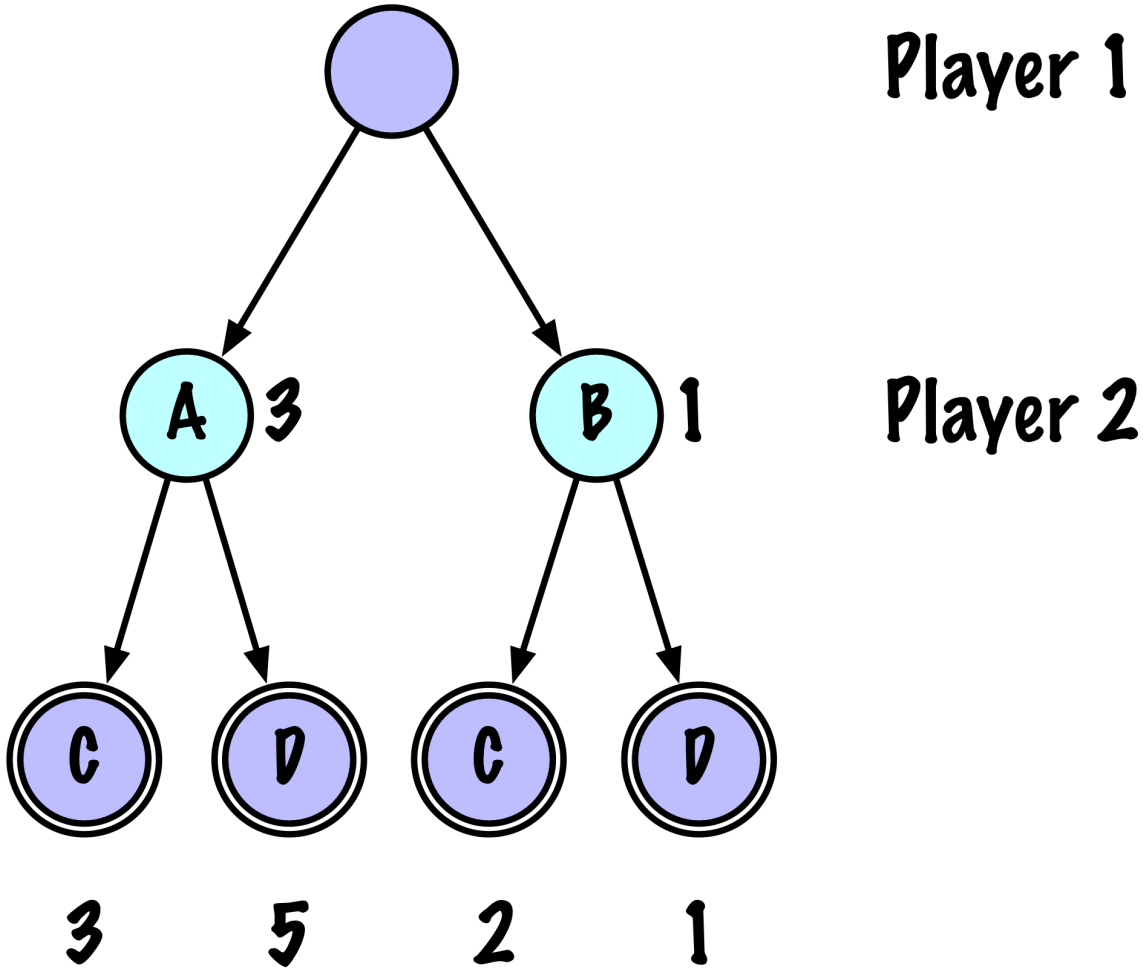
Quelle devrait être la **stratégie du Joueur 2** et **pourquoi** ?

Stratégie



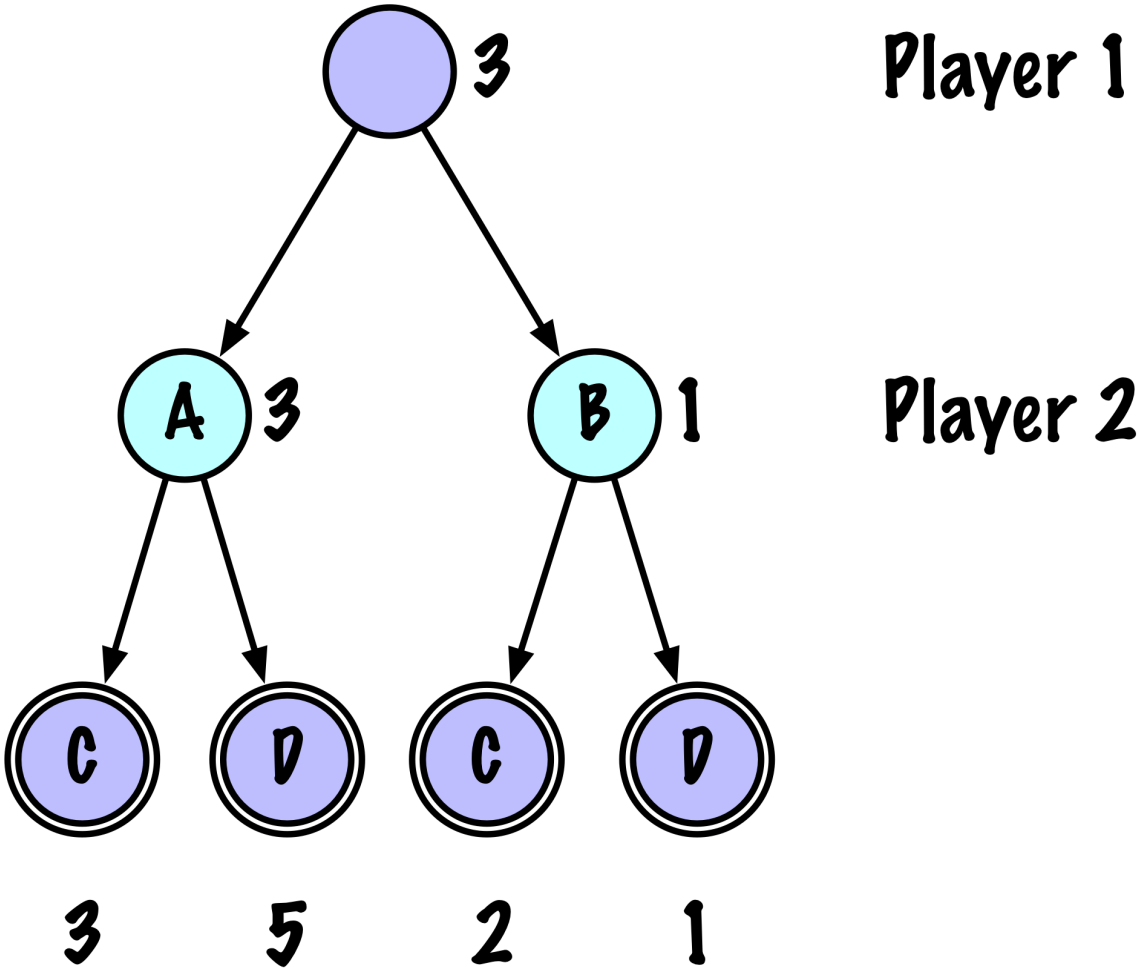
- **Pour le coup** A :
  - Le Joueur 2 peut choisir C (score = 3) ou D (score = 5) ; il choisit C (pour minimiser à 3).
- **Pour le coup** B :
  - Le Joueur 2 peut choisir C (score = 2) ou D (score = 1) ; il choisit D (pour minimiser à 1).

Stratégie



Quelle devrait maintenant être la stratégie pour **le Joueur 1** ?

Stratégie



Le Joueur 1, étant le **maximiseur**, choisira le coup A, puisqu'il mène au score le plus élevé de 3 après la minimisation par le Joueur 2.

Minimax

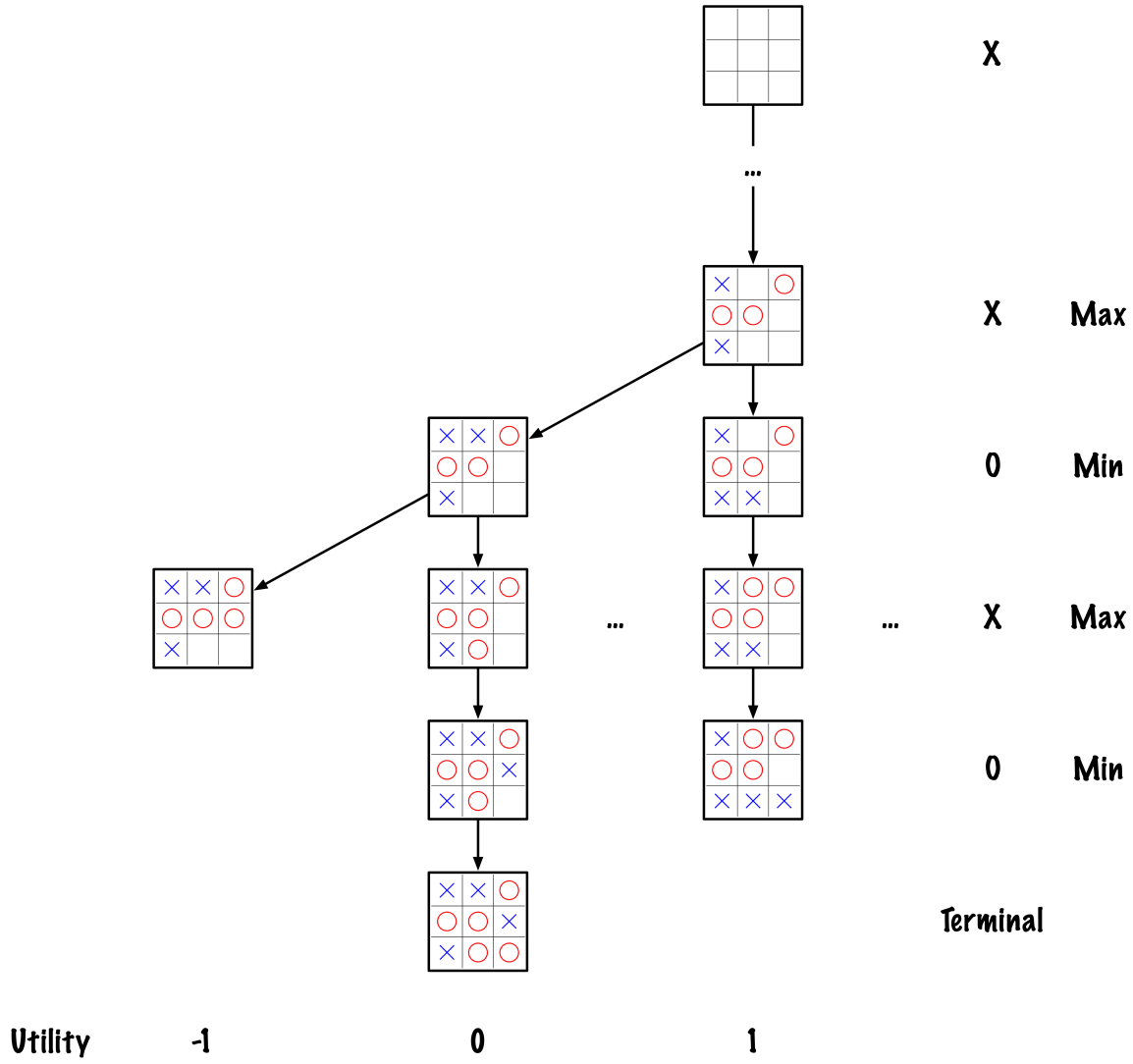
Minimax

- **Joueur 1** est le joueur maximisant, cherchant à obtenir le score le plus élevé.
- **Joueur 2** est le joueur minimisant, cherchant à obtenir le score le plus bas.

Évaluation :

- Le joueur 2 évalue les résultats potentiels de chacun de ses coups et choisit l'issue **la moins favorable** pour le joueur 1.
- Le joueur 1 évalue ensuite ces issues, choisissant le coup qui **maximise son score minimal garanti**.

Recherche Minimax



Recherche Minimax

L'**algorithme minimax** fonctionne en explorant **tous les coups possibles dans un arbre de jeu**, en évaluant les issues afin de **minimiser la perte possible dans le pire des cas**. À chaque nœud :

- **Tour du joueur maximisant :** choisir le coup avec la valeur la plus élevée possible.
- **Tour du joueur minimisant :** choisir le coup avec la valeur la plus faible possible.

En revenant des nœuds terminaux à la racine, l'algorithme sélectionne le coup qui **maximise le gain minimal du joueur**, anticipant ainsi et **contrant les meilleures stratégies de l'adversaire**.

Minimax

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 et Norvig 2020, fig. 5.3)

Démonstration (premières 4 minutes)

https://youtu.be/l-hh51ncgDI

Attribution : Sebastian Lague

Regardez les 5 premières minutes et 20 secondes de la vidéo.

La vidéo présente une variante de l'algorithme **minimax** qui inclut un **paramètre de profondeur maximale**, permettant à l'algorithme de s'arrêter à une profondeur spécifiée par l'utilisateur. Cette approche est essentielle pour les jeux ayant de grands espaces de recherche, tels que les échecs et le Go. Cependant, elle exige une méthode fiable pour évaluer l'état courant, appelée évaluation statique dans la vidéo.

`MinimaxSolverV1`

In [24]:
```python
class MinimaxSolverV1(Solver):

    """
    Un solveur Minimax classique et exact pour le Tic-Tac-Toe.
```

```
    Propriétés principales :
    - Supposons que "X" est le joueur maximisant.
    - Effectue une recherche complète de l'arbre de jeu (le Tic-Tac-Toe est
    """

    # ------------------------------------------------------------------------
    # API PUBLIQUE — Interface Solver
    # ------------------------------------------------------------------------

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

        """
        Retourne le coup optimal pour `player` en utilisant la recherche mir

        Remarques
        ----------
        - Comme le Tic-Tac-Toe est de petite taille, une recherche complète
        - X maximise toujours ; O minimise toujours.
        """

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

    # ------------------------------------------------------------------------
    # CŒUR DE MINIMAX
    # ------------------------------------------------------------------------

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

        """
        Procédure récursive centrale du minimax.

        Paramètres
        ----------
        game       : l'instance TicTacToe
        state      : tableau NumPy représentant le plateau courant
        player     : le joueur qui doit jouer ("X" ou "O")
        maximizing : Vrai si ce nœud correspond au joueur maximisant
                     (c.-à-d., X doit jouer de façon optimale), Faux pour mi

        Retourne
        -------
        (valeur, coup)
            valeur : +1, 0 ou -1 du point de vue de X
            coup   : le meilleur coup trouvé à ce nœud
        """

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

        if game.is_terminal(state):
            # game.evaluate() retourne 1 pour une victoire de X, -1 pour O,
            return game.evaluate(state), None
```

```
        # ------------------------------------------------------------
        # GÉNÉRER TOUS LES COUPS LÉGAUX
        # ------------------------------------------------------------

        moves = game.get_valid_moves(state)

        # ------------------------------------------------------------
        # NŒUD MAX : à X de jouer
        # ------------------------------------------------------------

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

            for move in moves:
                next_state = game.make_move(state, move, player)
                # Après le coup de X, c'est au tour de O (minimisant)
                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

        # ------------------------------------------------------------
        # NŒUD MIN : à O de jouer
        # ------------------------------------------------------------

        else:
            best_value = math.inf
            best_move = None

            for move in moves:
                next_state = game.make_move(state, move, player)
                # Après le coup de O, c'est au tour de X (maximisant)
                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 [25]:
```python
class MinimaxSolverV1(Solver):

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

        maximizing = (player == "X")

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

        return move
```

`MinimaxSolverV1` est un `Solver`.

Il exécute l'algorithme minimax à partir de `state` afin de déterminer le coup optimal pour `player`.

`_minimax`

In [26]:
```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 [27]:
```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 [28]:
```
        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
```

Exécution (1/2)

In [29]:
```
import time

# Enregistrer le temps de début
start_time = time.perf_counter()
```

In [30]:
```
a = RandomSolver(13)
b = MinimaxSolverV1()

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

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

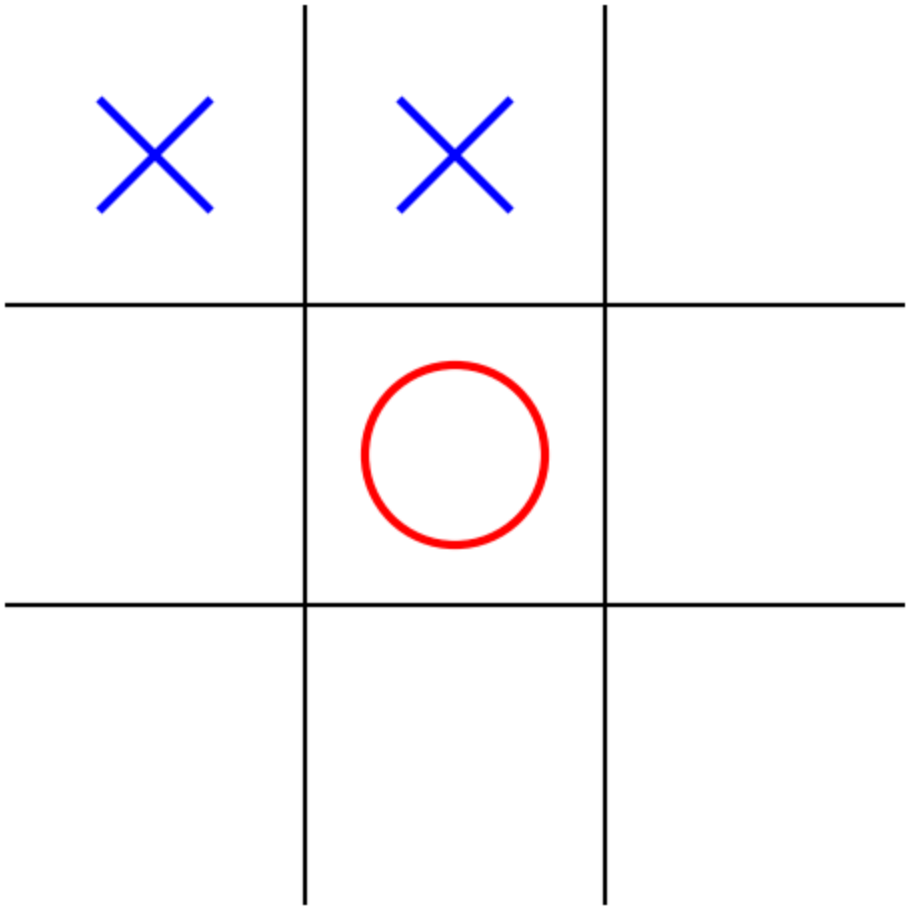
In [31]:
```
# Enregistrer le temps de fin
end_time = time.perf_counter()

# Calculer le temps écoulé
elapsed_time = end_time - start_time

# Afficher le temps écoulé en secondes
print(f"Temps écoulé : {elapsed_time:.6f} secondes !")
```

```
Temps écoulé : 2.125289 secondes !
```

Discussion (Digression)

- Le test `test_tic_tac_toe` est-il plus rapide ou plus lent que prévu ?
- Voyez-vous un domaine à améliorer ?

Discussion (Digression)



Mémorisation (Mise en cache)

In [32]:
```python
from functools import lru_cache

def canonical(state):

    """
    Convertit un plateau sous forme de tableau NumPy en une représentation
    hachable et immuable (tuple de tuples). Cela nous permet de l'utiliser
    comme clé dans des dictionnaires ou comme argument pour lru_cache.
    MCTS peut également réutiliser cette représentation.
    """

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

`MinimaxSolver`

In [33]:
```python
class MinimaxSolver(Solver):

    """
    Un solveur Minimax classique et exact pour le Tic-Tac-Toe.
```

```
    - Suppose que "X" est le joueur maximisant.
    - Utilise la mémoïsation (lru_cache) pour éviter de recalculer
      les valeurs pour des positions identiques.
    """

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

        """
        Interface publique : choisir le meilleur coup pour « player » en uti
        Pour le Tic-Tac-Toe, il est possible d'explorer tout l'arbre du jeu
        """

        # Stocke le jeu dans self pour que _minimax puisse l'utiliser
        self.game = game

        # Du point de vue de X : X maximise, O minimise
        maximizing = (player == "X")

        # Pour le Tic-Tac-Toe, depth=9 suffit à couvrir tous les coups resta
        _, move = self._minimax(canonical(state), player, maximizing, 9)
        return move

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

        """
        Minimax récursif interne.

        Paramètres
        ----------
        state_key : représentation hachable du plateau (tuple de tuples)
        player    : joueur devant jouer à ce nœud (« X » ou « O »)
        maximizing: Vrai si ce nœud est un nœud « max » (X doit jouer),
                    Faux si c'est un nœud « min » (O doit jouer)
        depth     : profondeur de recherche restante (non utilisée pour les
                    coupures dans cette implémentation exhaustive du Tic-Tac
                    mais conservée à des fins didactiques et pour faciliter
        """

        # Reconstitue le plateau NumPy à partir du state_key canonique
        state = np.array(state_key)

        # Test terminal : victoire, défaite ou égalité
        if self.game.is_terminal(state):
            # L'évaluation se fait toujours du point de vue de X : +1, -1 ou
            return self.game.evaluate(state), None

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

        if maximizing:
            # X doit jouer : maximiser l'évaluation
            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 doit jouer : minimiser l'évaluation (puisque l'évaluation es
            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
```

Exécution (2/2)

In [34]:
```
import time

# Enregistrer le temps de début
start_time = time.perf_counter()
```

In [35]:
```
a = RandomSolver(13)
b = MinimaxSolver()

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

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

In [36]:
```
# Enregistrer le temps de fin
end_time = time.perf_counter()

# Calculer le temps écoulé
elapsed_time = end_time - start_time

# Afficher le temps écoulé en secondes
print(f"Temps écoulé : {elapsed_time:.6f} secondes !")
```

```
Temps écoulé : 0.106057 secondes !
```

Encore une digression

In [37]:
```
    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)
```

Les jeux peuvent devenir monotones si vous discernez rapidement des schémas dans la stratégie de votre adversaire, comme le fait de toujours choisir les coups dans une séquence spécifique.

Nous randomisons l'ordre des coups avant de les retourner. Cependant, le coup optimal pour une configuration donnée restera fixe, si les solutions sont mises en cache.

Exploration

- Comparez la réduction du temps d'exécution obtenue grâce aux **considérations de symétrie** par rapport aux **techniques de mise en cache**. Évaluez **l'effet combiné des deux approches**.
- Développez une implémentation du jeu **Puissance 4** utilisant un **algorithme de recherche minimax**.
- Puissance 4 est **symétrique par rapport à son axe vertical**. Développez une nouvelle implémentation qui exploite cette symétrie.

Voir aussi : Connect 4: Principles and Techniques

Pouvez-vous deviner le résultat ?

In [38]:
```
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}
```

Pouvez-vous deviner le résultat ?

In [39]:
```
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}
```

Pouvez-vous deviner le résultat ?

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

a = MinimaxSolver()
b = MinimaxSolver()

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

results
```

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

Résumé

Qu'avons-nous obtenu ?

...

Nous avons dérivé une **politique** (un algorithme de décision) qui recommande des actions menant à la **solution optimale** du jeu, sous l'hypothèse que **l'adversaire adopte la même politique**.

Remarque

Le **nombre de séquences valides d'actions croît de façon factorielle**, avec une croissance particulièrement importante dans des jeux comme les échecs et le Go.

Élagage alpha-bêta

Élagage

Pour améliorer l'efficacité de l'algorithme minimax, on peut potentiellement **élaguer** certaines parties de l'arbre de recherche, ce qui permet **d'éviter l'exploration des nœuds descendants**.

Élagage

Comment mettriez-vous en œuvre cette modification ? **Quels** facteurs prendriez-vous en compte ?

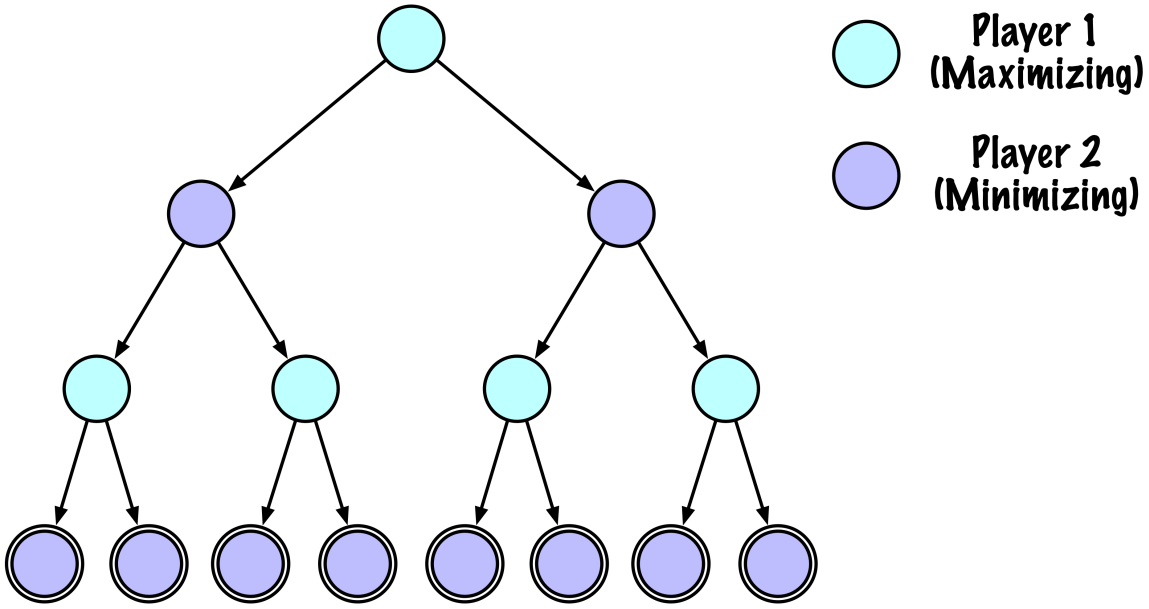
Élagage

L'élagage de l'arbre doit être effectué **uniquement lorsqu'il peut être démontré que ces sous-arbres ne peuvent pas mener à de meilleures solutions**.

Mais **comment** ?

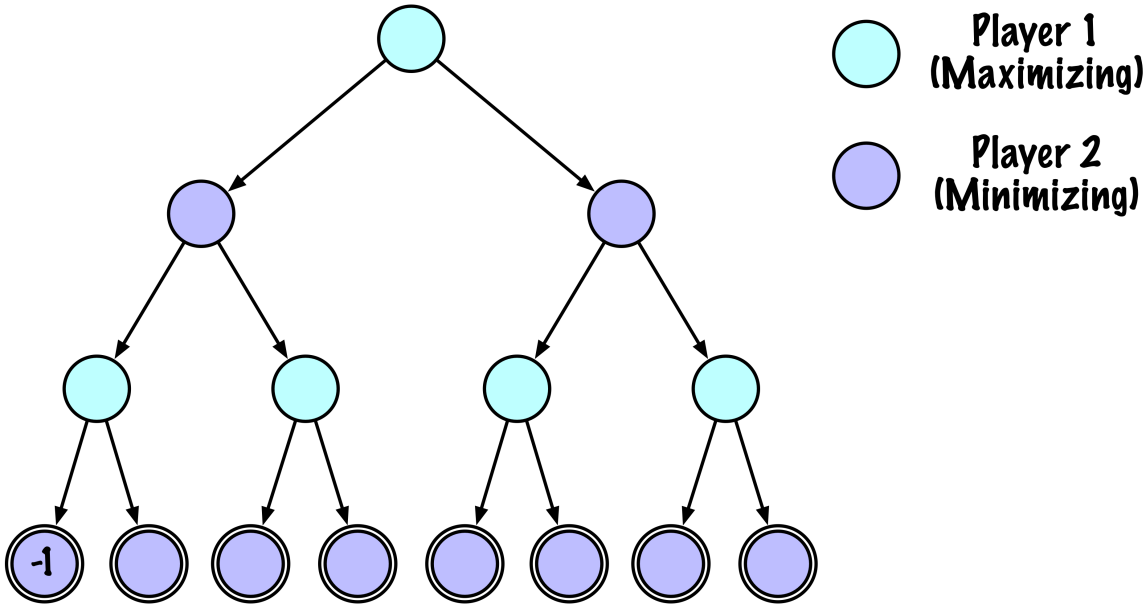
Puisque l'élagage alpha-bêta ne rejette que les sous-arbres qui ne peuvent pas améliorer la borne courante, il préserve la valeur minimax et le coup optimal résultant par rapport au minimax standard, tout en explorant moins de nœuds.

Critères d'élagage

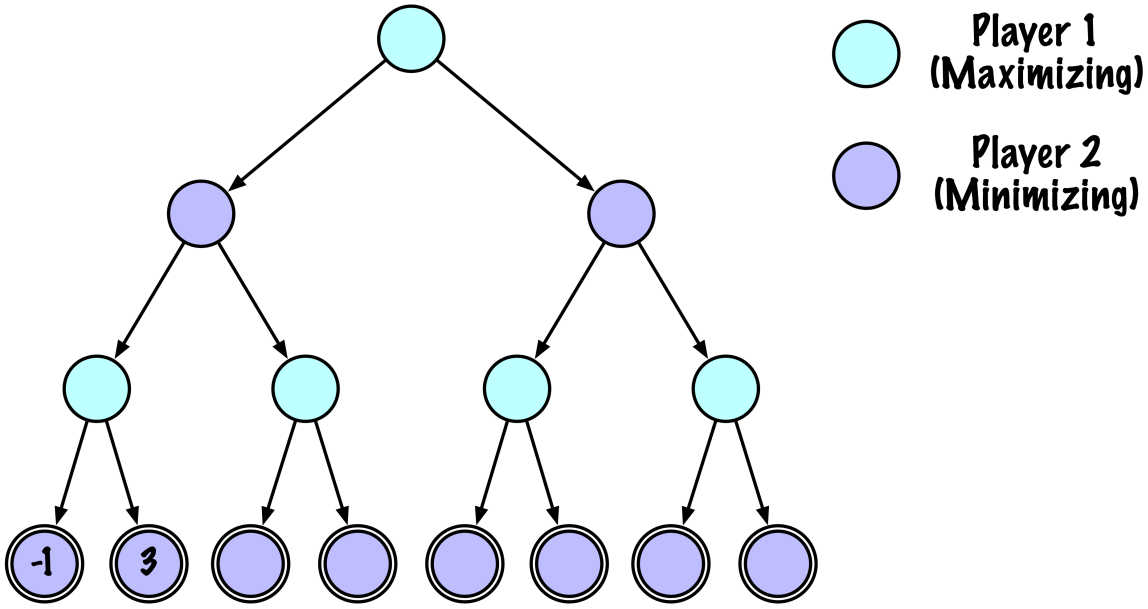


D'après Algorithms Explained – minimax and alpha-beta pruning par Sebastian Lague.

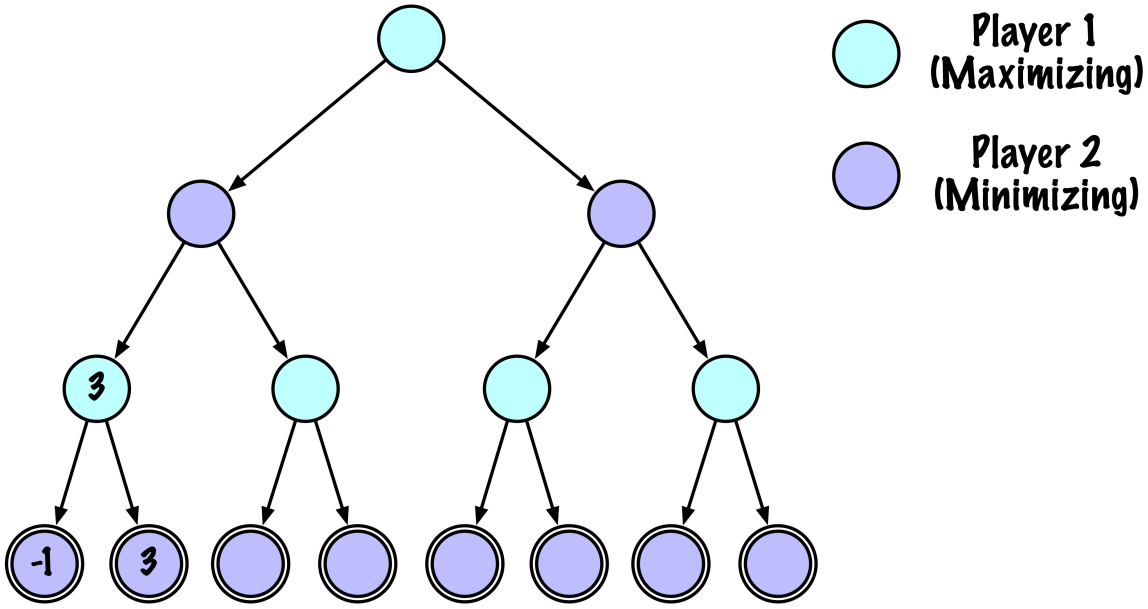
Critères d'élagage



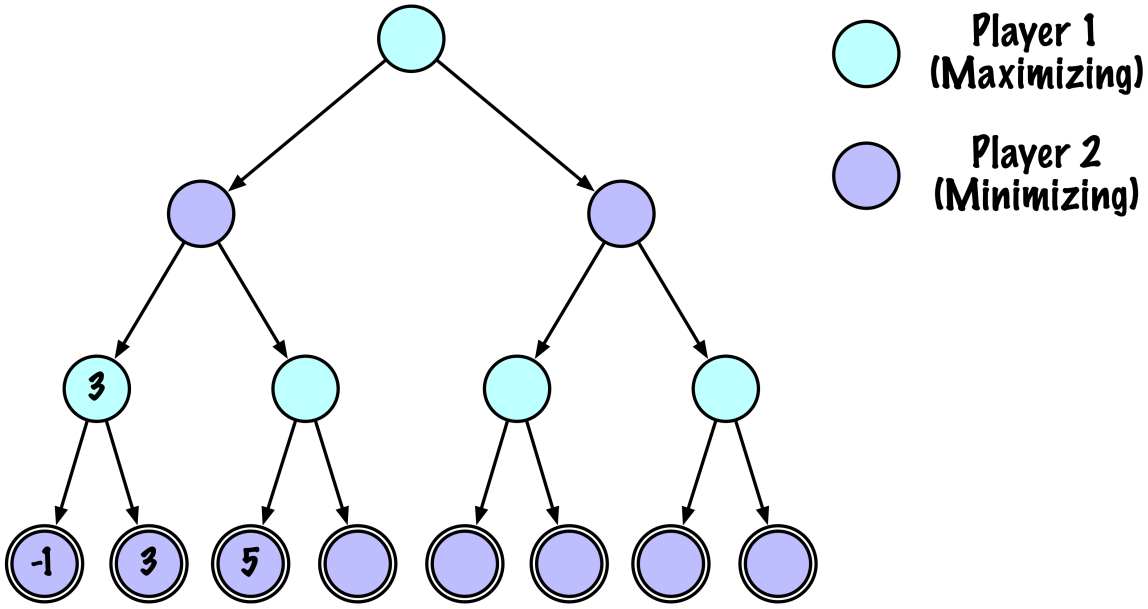
Critères d'élagage



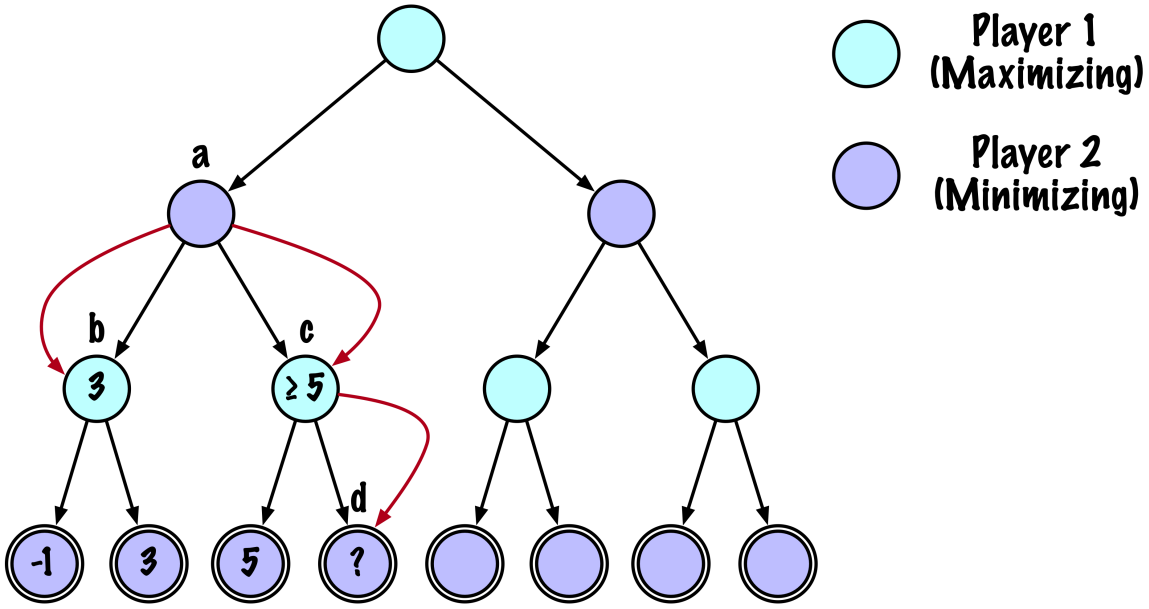
Critères d'élagage



Critères d'élagage



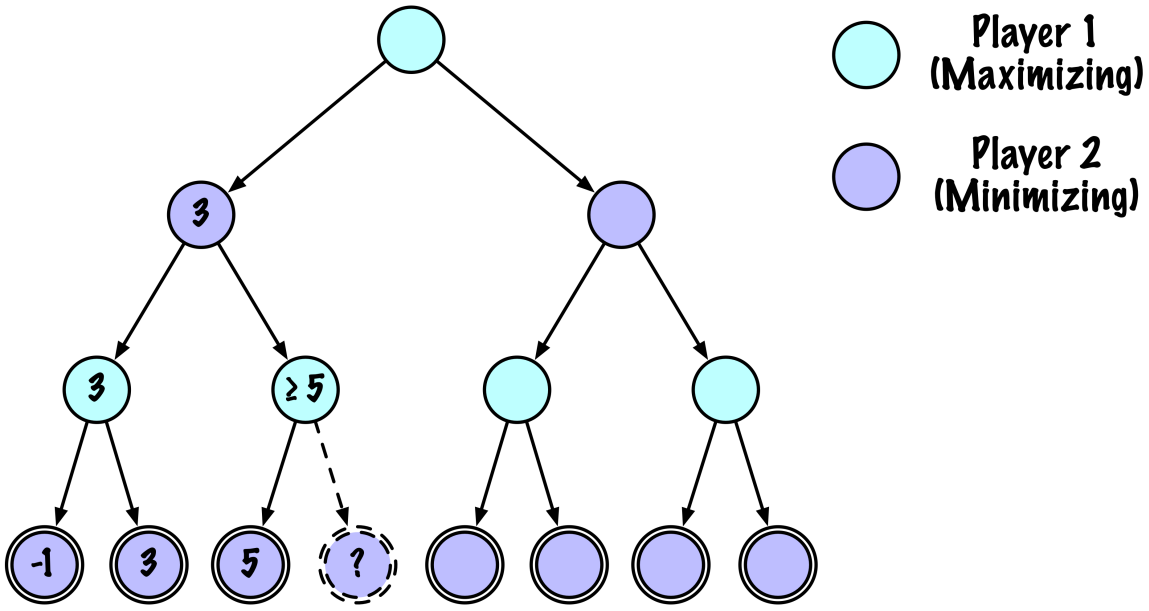
Critères d'élagage



Nous savons que la valeur du nœud **c** est d'au moins 5, puisque **c** est un nœud maximisant.

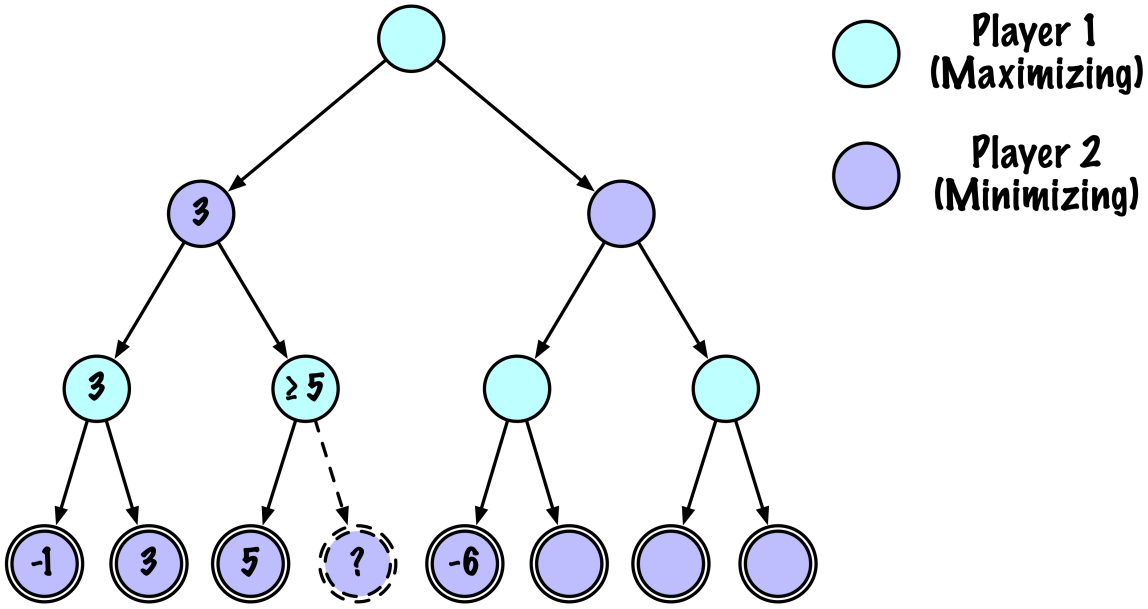
Le nœud **a** est un nœud minimisant et doit choisir entre 3 (nœud **b**) et au moins 5 (nœud **c**). Peu importe ce que nous trouvons au nœud **d**, cela n'a pas d'importance : le nœud **a** choisira 3.

Critères d'élagage



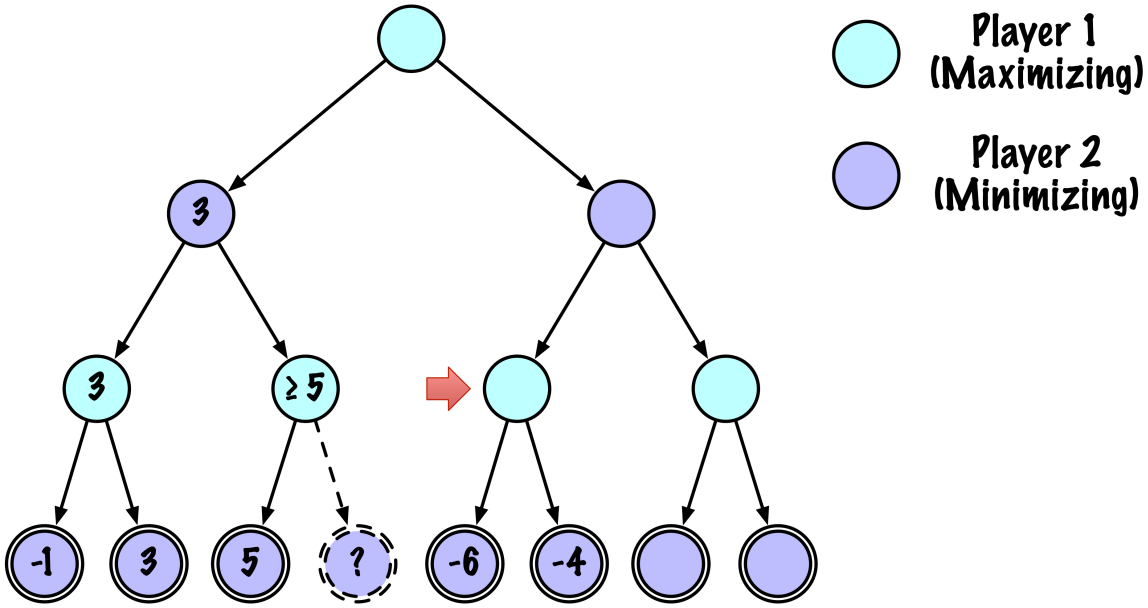
Poursuivons brièvement.

Critères pour l'élagage



Poursuivons brièvement.

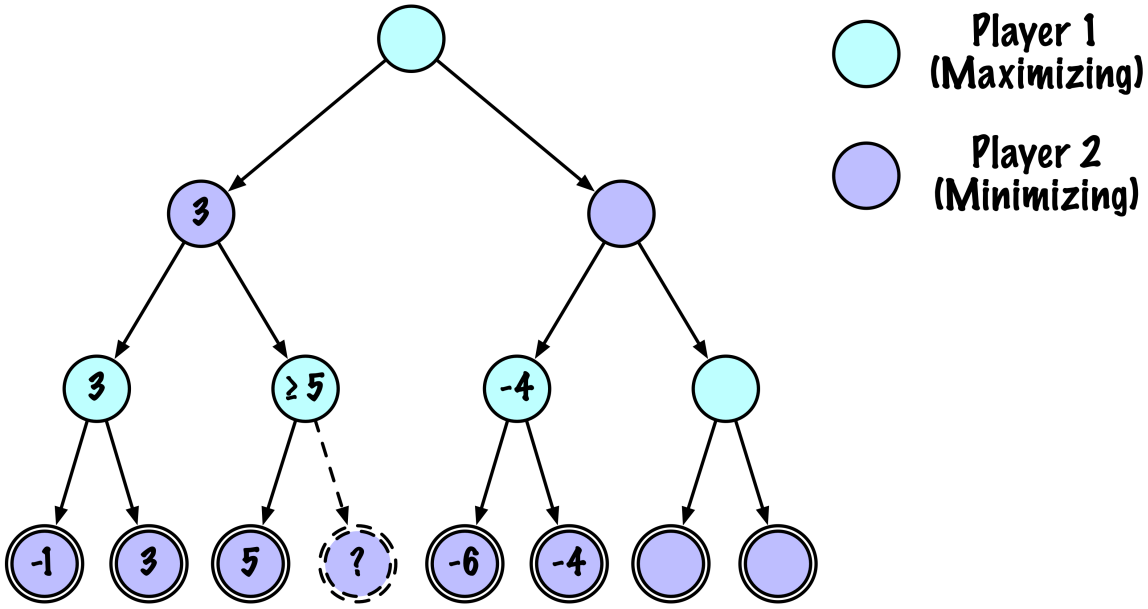
Critères d'élagage



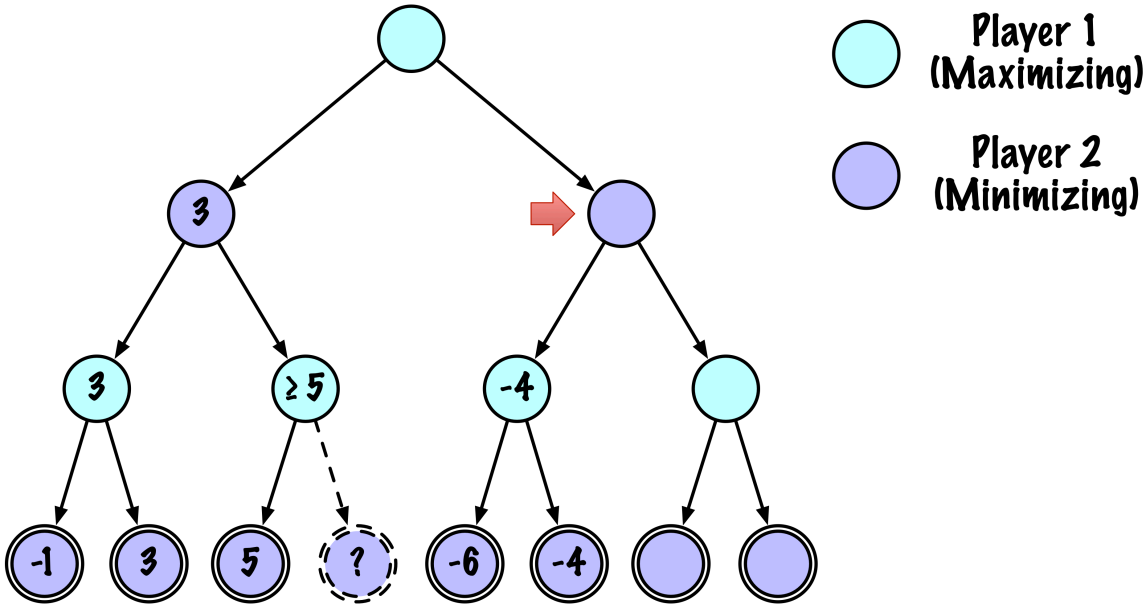
Quelle valeur doit être attribuée au nœud indiqué par la flèche ?

Il s'agit d'un nœud de maximisation ; par conséquent, nous lui attribuerons la valeur -4.

Critères d'élagage



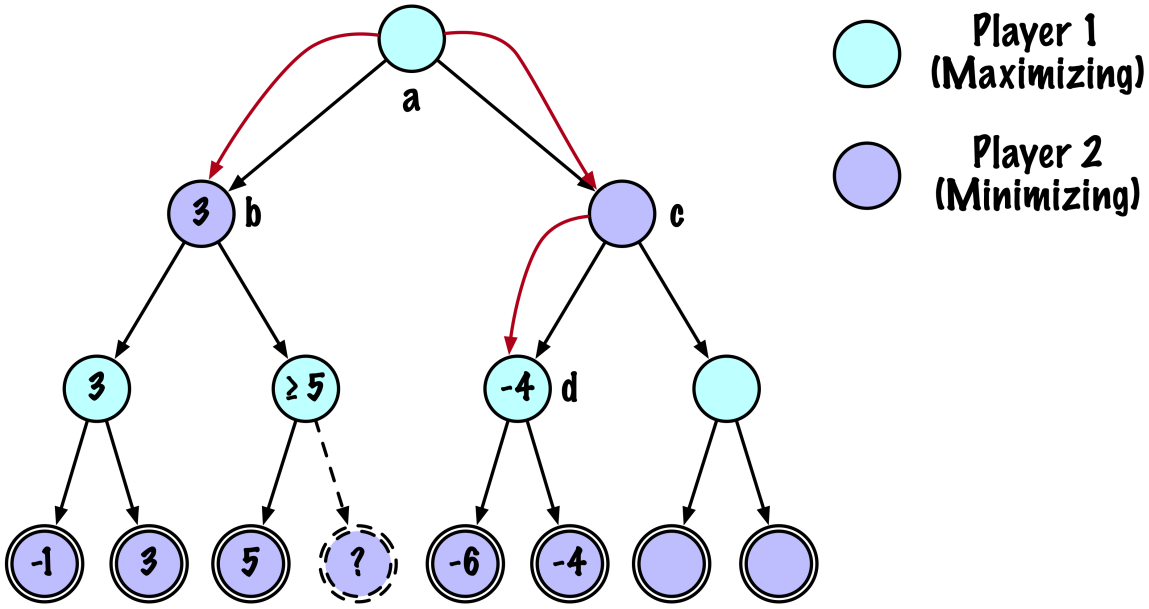
Critères pour l'élagage



Le nœud indiqué par une flèche présente un scénario intéressant.

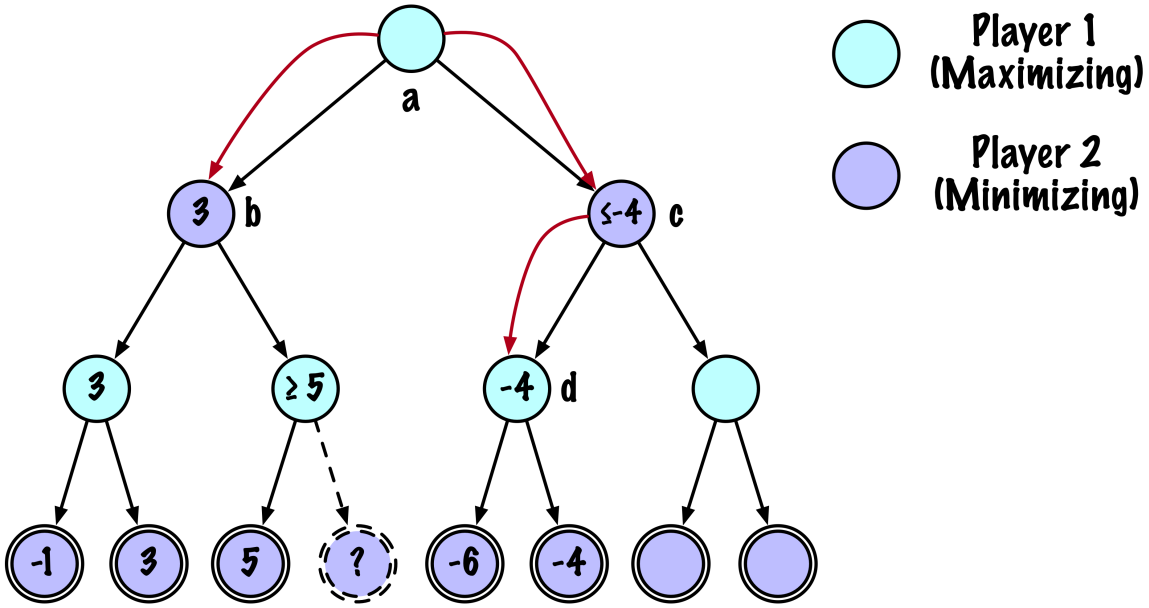
Pouvez-vous en déterminer la valeur ?

Critères d'élagage



Que savons-nous ?

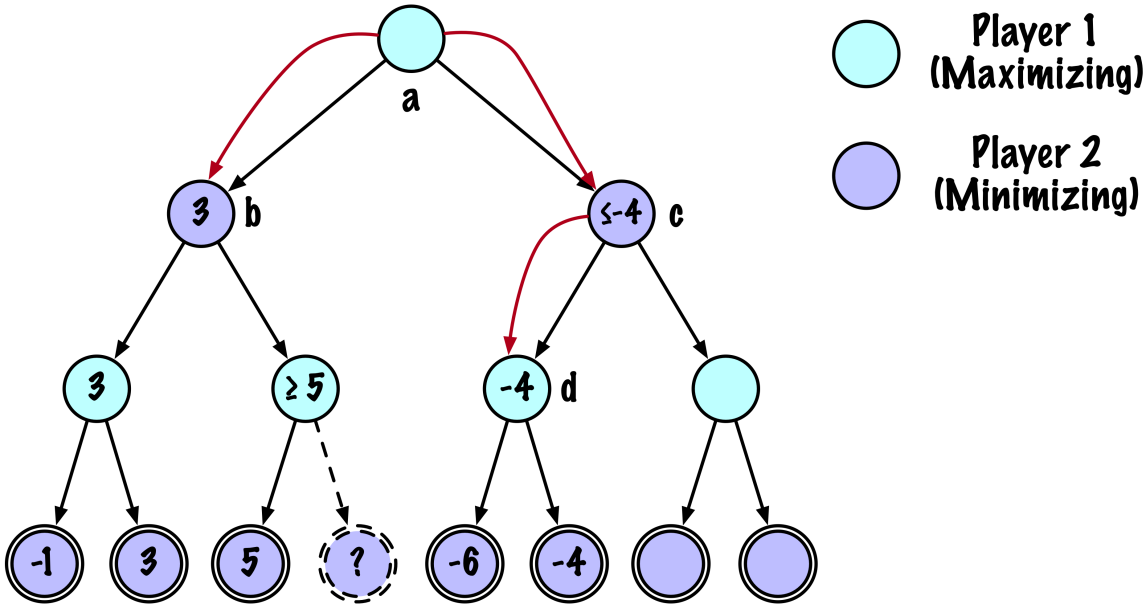
Critères d'élagage



Puis-je dire que la valeur maximale pour le nœud **c** est -4 ?

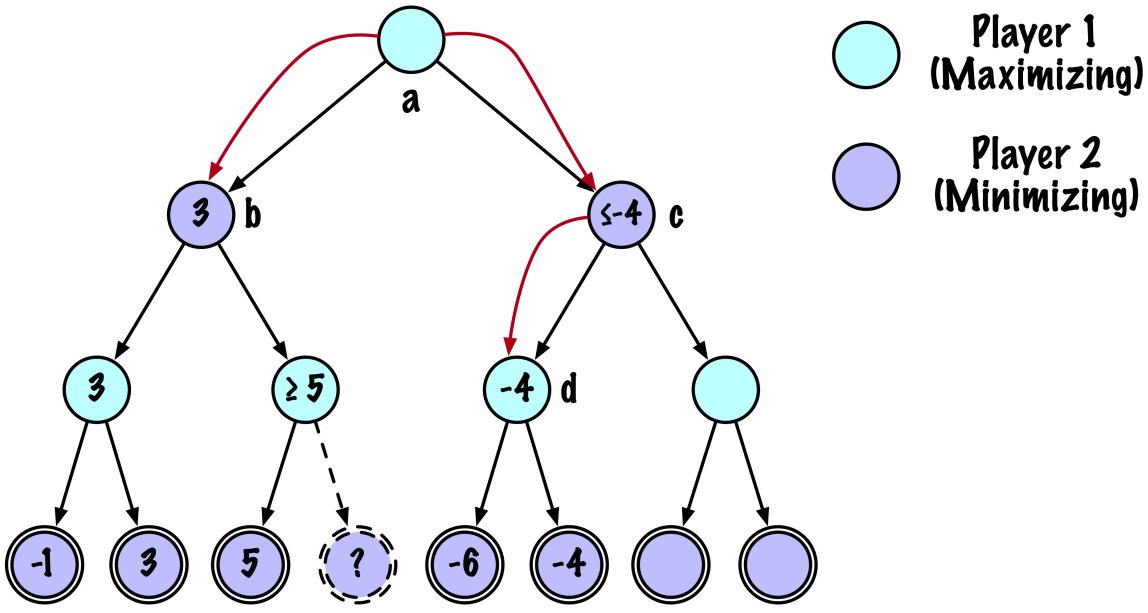
Supposons que le fils droit du nœud **c** contienne la valeur 5. Attribueriez-vous 5 au nœud **c** ? Certainement pas. En tant que nœud de minimisation, le nœud **c** aura une valeur qui ne dépassera pas -4.

Critères d'élagage



Explorer l'enfant droit du nœud **c** pourrait révéler une valeur inférieure à -4. **Est-ce important ?**

Critères pour l'élagage



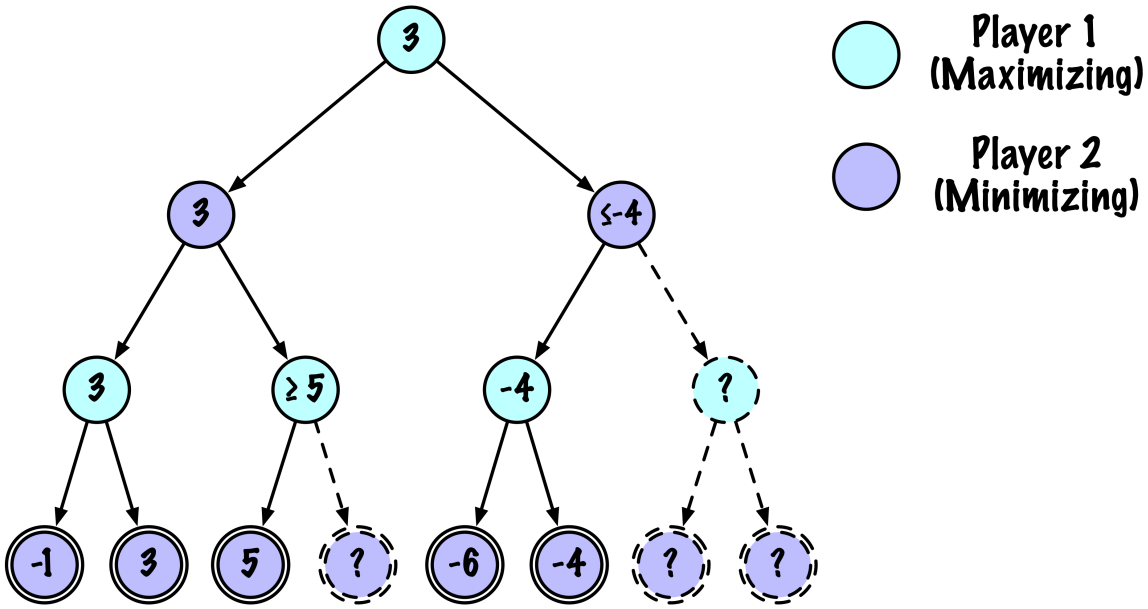
Le nœud **a** est un nœud maximisant. À partir du nœud **b**, il détermine qu'une valeur minimale atteignable est 3.

Le nœud **c** est un nœud minimisant. À partir du nœud **d**, il constate que la valeur maximale qu'il peut retourner est -4.

Par conséquent, le nœud **a** choisira toujours le nœud **b**, indépendamment de la valeur stockée dans le nœud **c**, puisque la valeur de **c** n'excédera pas -4.

Même si le nœud **c**, en tant que nœud minimisant, découvre une valeur extrêmement faible (par exemple, -100) en explorant son enfant droit, le nœud **a**, étant un nœud maximisant, optera tout de même pour le nœud **b**.

Critères d'élagage



Les décisions prises par les joueurs 1 et 2 demeurent inchangées avec ou sans élagage. Cependant, l'élagage réduit le nombre de nœuds visités.

Élagage Alpha-Bêta

L'**élagage alpha-bêta** est une technique d'optimisation de l'algorithme minimax qui **réduit le nombre de nœuds évalués** dans l'arbre de recherche.

Élagage Alpha-Bêta

Il y parvient en **éliminant les branches qui ne peuvent pas influencer la décision finale**, à l'aide de deux paramètres :

- **alpha**, le score maximal que le joueur maximisant peut garantir, et
- **bêta**, le score minimal que le joueur minimisant peut garantir.

Maximiser la perspective du joueur

À un **nœud de maximisation** :

- Le **maximiseur** cherche à **maximiser** le score.

- **Alpha (α)** est mise à jour avec la **valeur la plus élevée** trouvée parmi les nœuds enfants.
- **Processus** :
  - Initialiser $\alpha = -\infty$.
  - Pour chaque nœud enfant :
    - Calculer le score d'évaluation.
    - Mettre à jour $\alpha = \max(\alpha, \text{child\_score})$.

Perspective du joueur minimisant

À un **nœud de minimisation** :

- Le **minimiseur** cherche à **minimiser** le score.
- **Bêta (β)** est mis à jour avec la **plus faible** valeur trouvée parmi les nœuds enfants.
- **Processus** :
  - Initialiser $\beta = \infty$.
  - Pour chaque nœud enfant :
    - Calculer le score d'évaluation.
    - Mettre à jour $\beta = \min(\beta, \text{child\_score})$.

Élagage Alpha-Bêta

Lorsqu'une évaluation de nœud montre qu'elle **ne peut pas améliorer** la valeur actuelle d'alpha ou de bêta, toute **exploration supplémentaire de cette branche est arrêtée**, ce qui améliore ainsi l'efficacité computationnelle sans affecter le résultat.

Rôle d'Alpha et Bêta dans l'élagage

Condition d'élagage :

- **Si** $\beta \leq \alpha$, il n'est plus nécessaire d'explorer les autres enfants du nœud courant.
- **Justification** :
  - Le maximiseur a un score garanti d'au moins α.

- Le minimiseaur peut s'assurer que le maximiseur n'obtienne pas un score supérieur à β.
- Si $\beta \leq \alpha$, le maximiseur ne trouvera pas de meilleure option dans cette branche.

Recherche Alpha-Bêta

```
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 et Norvig 2020, fig. 5.7)

Démonstration (6:21 à 8:10)

https://youtu.be/l-hh51ncgDI

Attribution : Sebastian Lague. Commencez à regarder à 6**m** 21**s**.

Ordre des nœuds

- L'**efficacité de l'élagage** dépend de l'**ordre** dans lequel les nœuds sont évalués.
- Un élagage plus important est obtenu si les nœuds sont ordonnés du **plus prometteur au moins prometteur**.

Voir : Shannon (1959) pour une discussion dans le contexte des échecs.

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.

À mesure que le facteur de branchement augmente, le potentiel d'élagage efficace s'accroît également.

Si l'ordre des coups était parfait, les procédures de recherche seraient inutiles. En pratique, cependant, des heuristiques efficaces, telles que les coups tueurs, l'heuristique d'historique et les priorités issues de réseaux neuronaux, améliorent considérablement l'ordre des coups. Ces approches augmentent de façon significative l'efficacité de l'élagage dans les algorithmes de recherche.

`MinimaxAlphaBetaSolverV1`

In [41]:
```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 [42]:
```python
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` est un `Solver`.

`_alphabeta`

In [43]:
```python
    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` : si maximisant

In [44]:
```python
        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,            # le prochain minimise
                    depth - 1,
                    alpha,
                    beta
                )

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

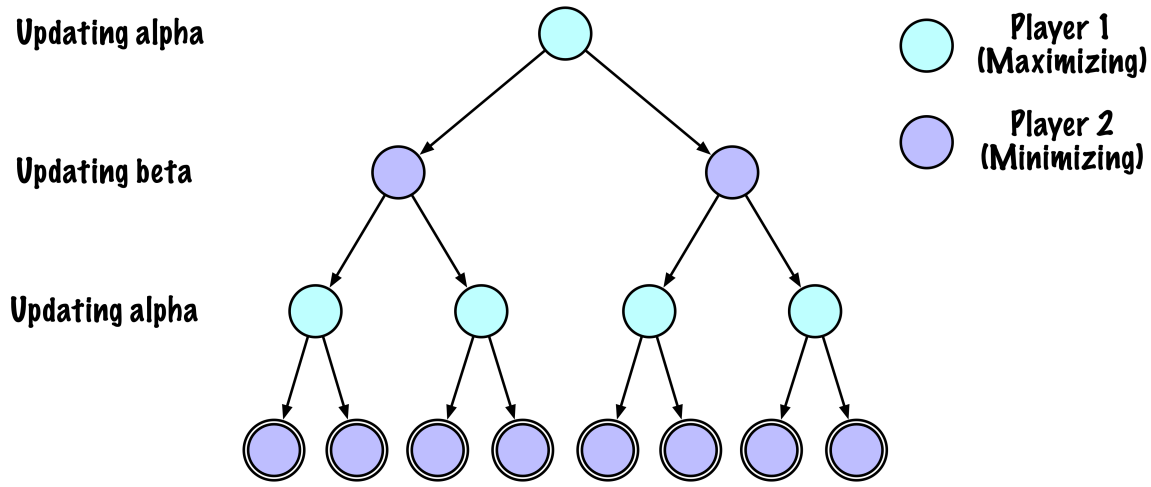
```
                # Mettre à jour alpha
                alpha = max(alpha, value)

                # Élagage
                if beta <= alpha:
                    break

            return value, best_move
```

`_alphabeta` : si on minimise

In [45]:

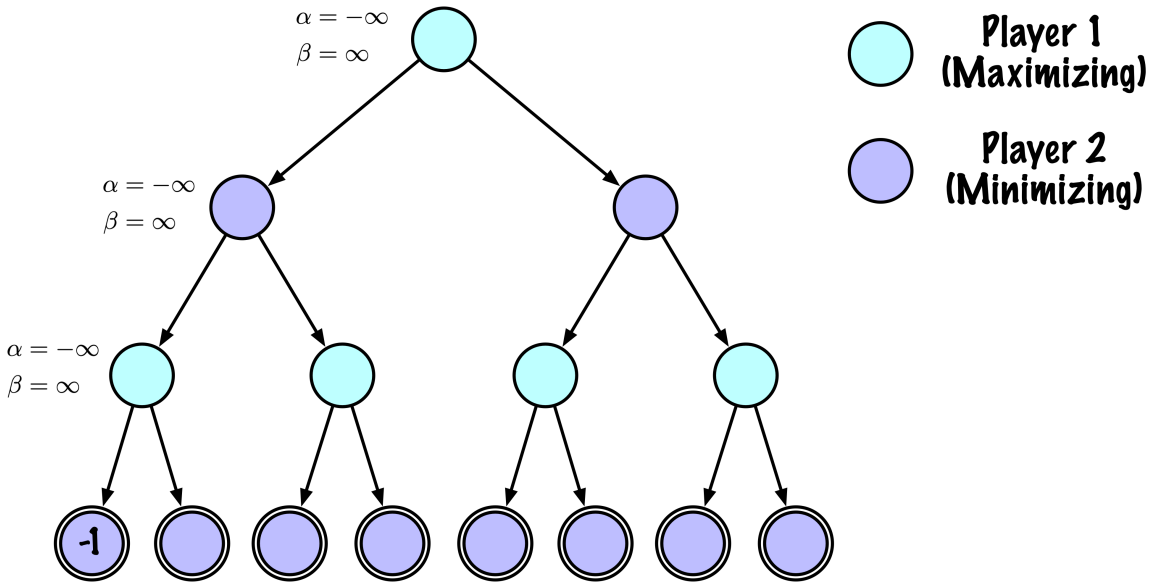
```
        else:
            value = math.inf

            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,              # le prochain est maximisant
                    depth - 1,
                    alpha,
                    beta
                )

                if child_val < value:
                    value = child_val
                    best_move = move

                # Mettre à jour beta
                beta = min(beta, value)

                # Élaguer
                if beta <= alpha:
                    break

            return value, best_move
```

Démonstration



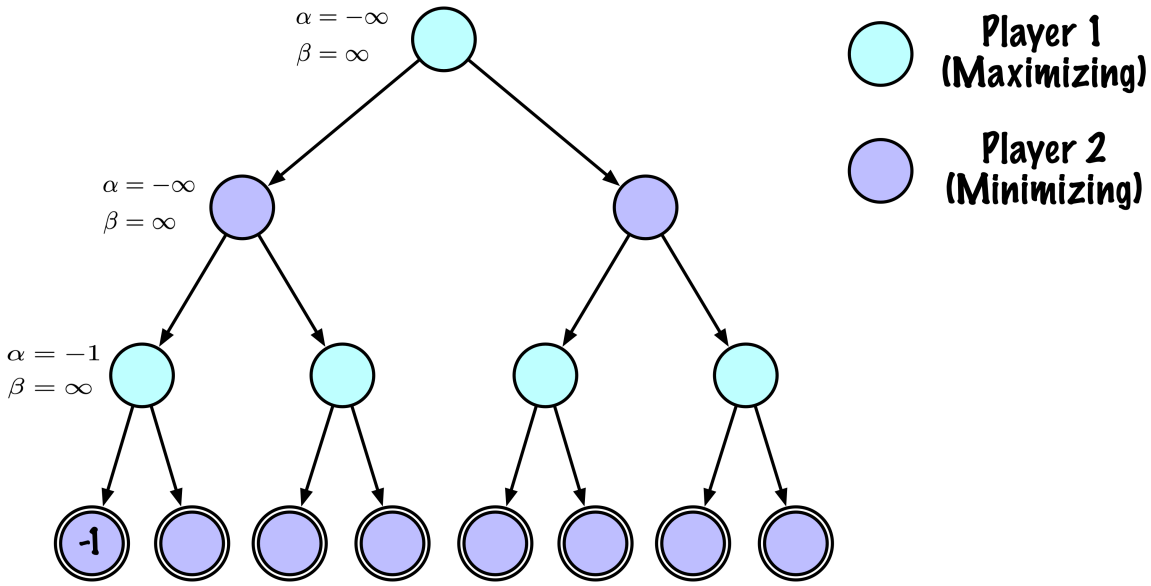
Les nœuds de **maximisation** mettent à jour les valeurs **alpha**, tandis que les nœuds de **minimisation** mettent à jour les valeurs **bêta**.

Démonstration



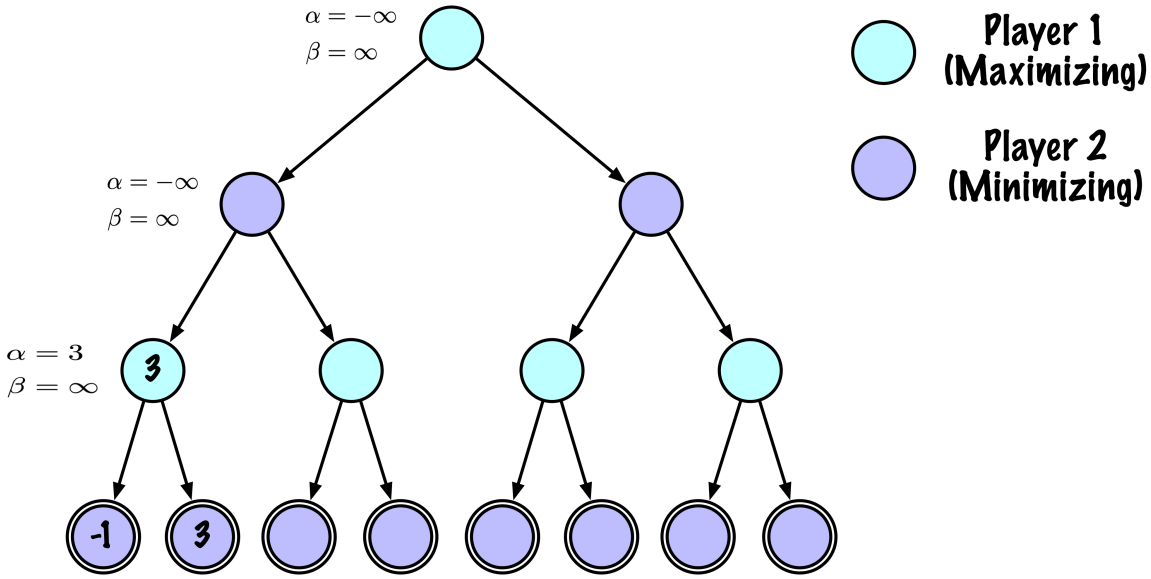
Les valeurs de α et β sont initialement fixées à $-\infty$ et ∞, respectivement. Les appels récursifs continuent de parcourir l'arbre jusqu'à ce que le nœud le plus à gauche soit atteint.

Démonstration



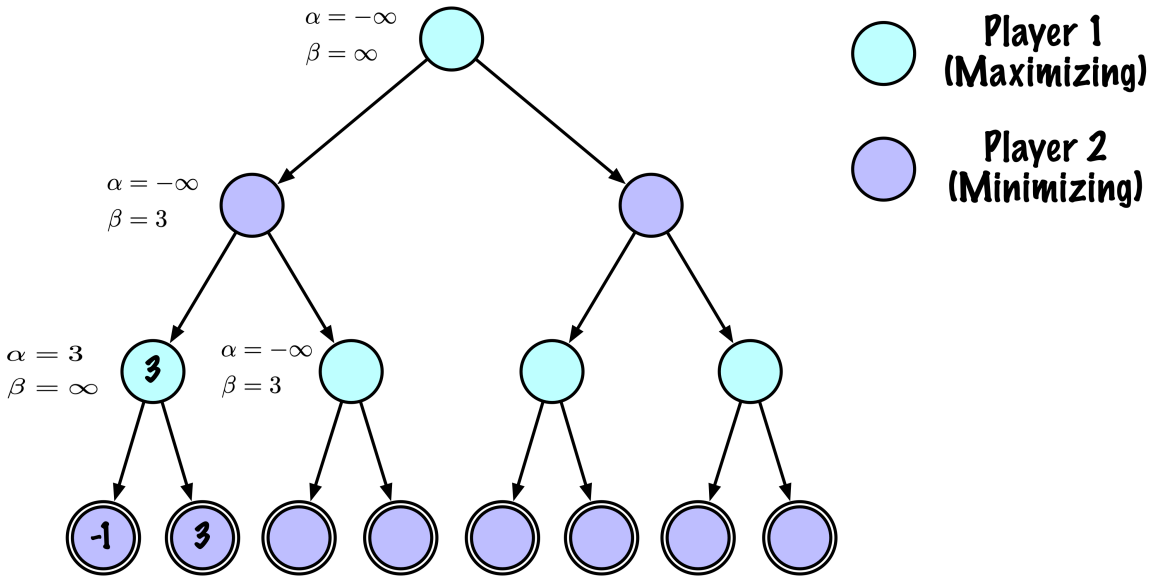
La valeur de α est mise à jour à -1.

Démonstration



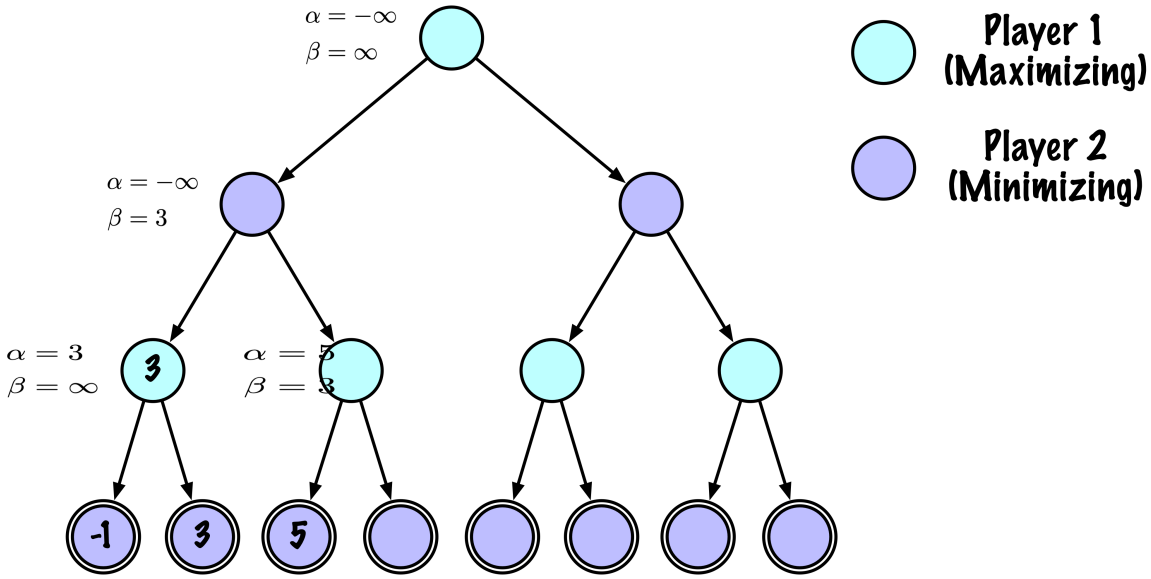
Mise à jour de α à 3.

Démonstration



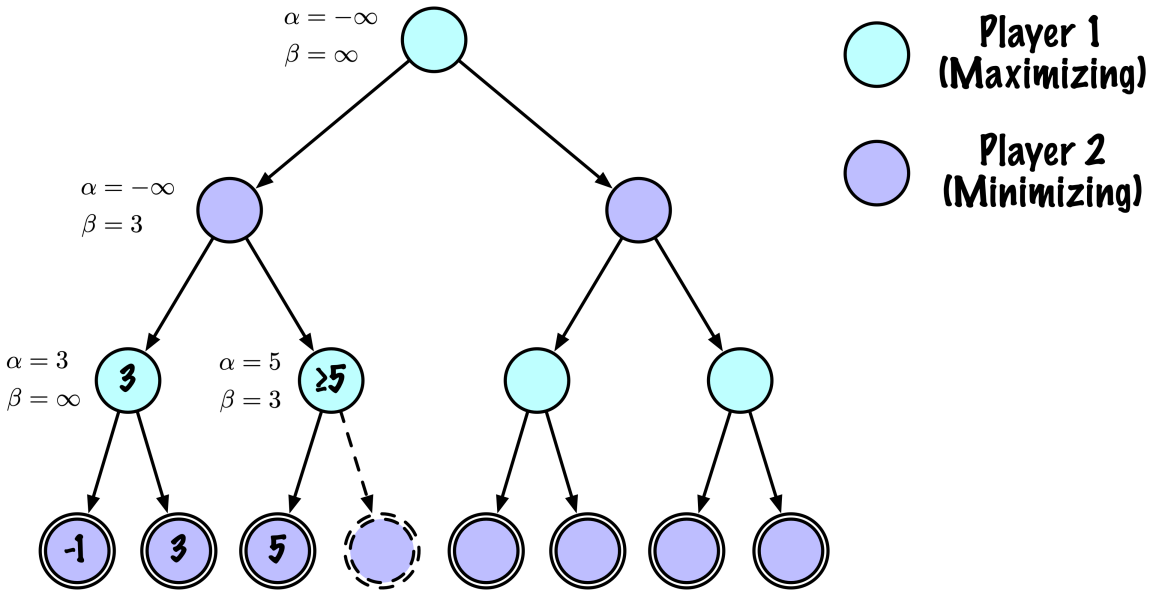
Mise à jour de β de ∞ à 3. Transmission de ces valeurs à l'enfant droit.

Démonstration



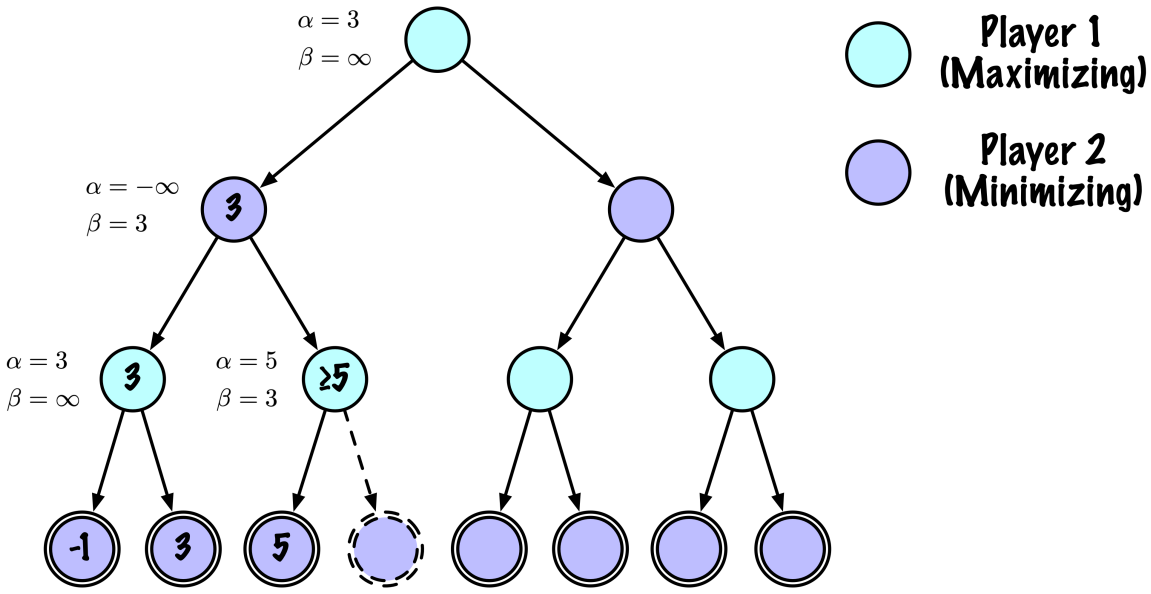
Mise à jour de α de $-\infty$ à 5. Nous avons maintenant $\beta \leq \alpha$.

Démonstration



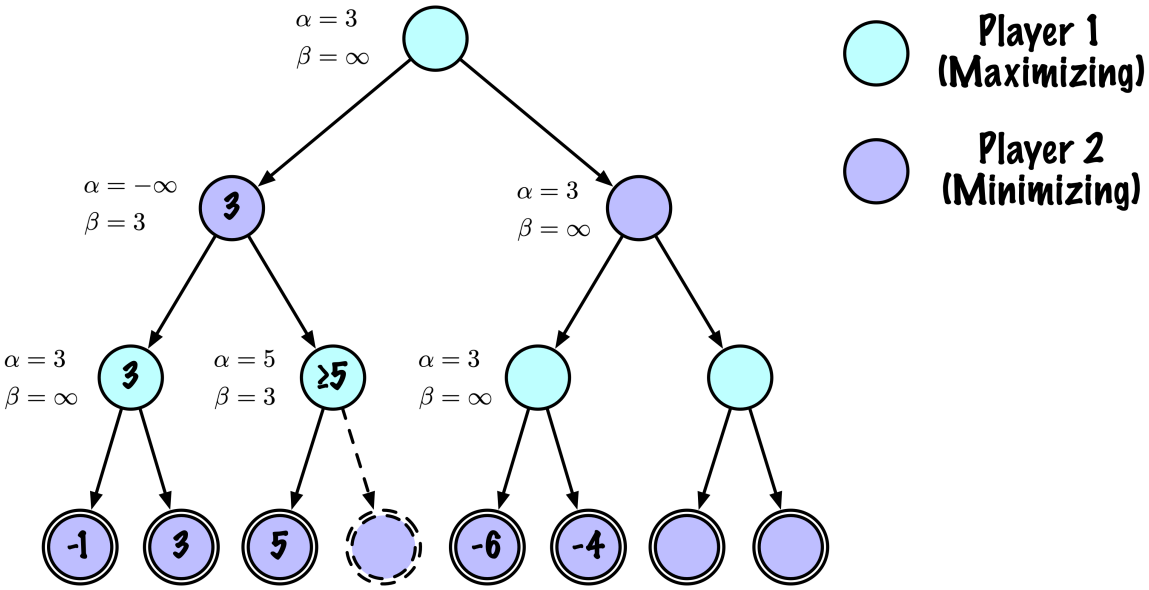
L'enfant droit n'est pas visité. Le parent (un minimiseur) a une meilleure option (β) que α .

Démonstration



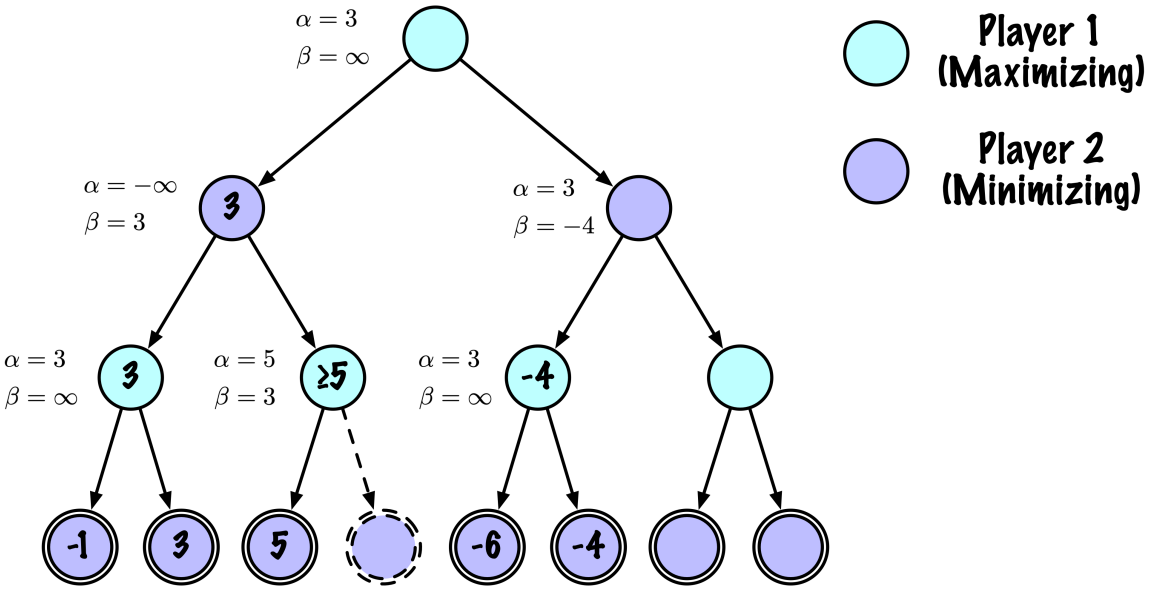
En revenant des appels récursifs, le nœud racine reçoit une valeur de 3 de son enfant gauche et met à jour son α à 3, ce qui dépasse sa valeur initiale de $-\infty$.

Démonstration



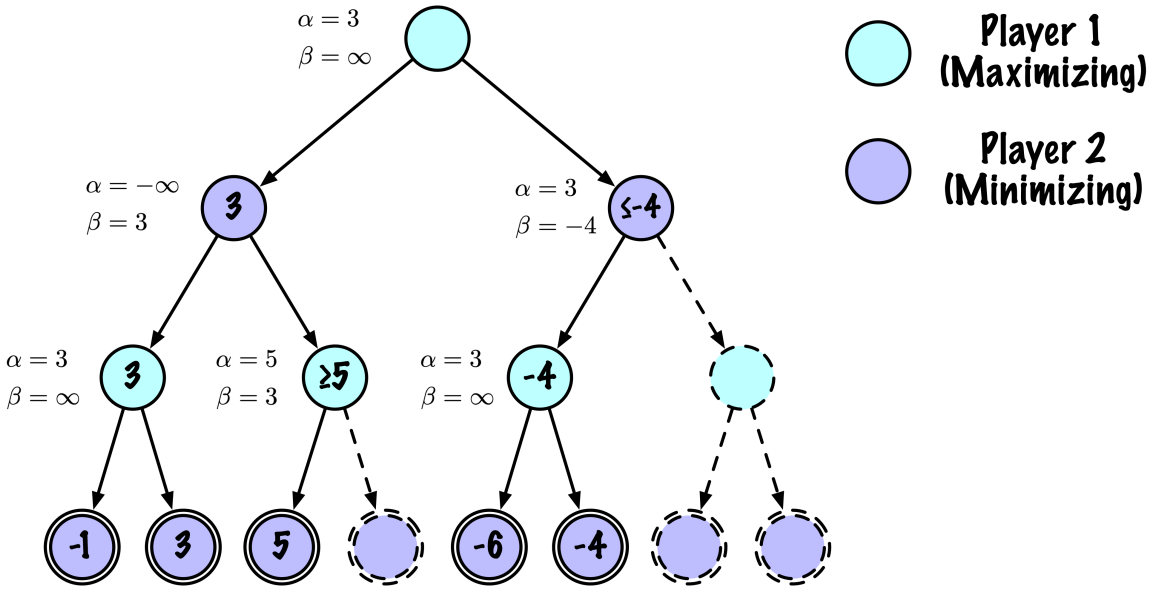
Nous parcourons maintenant le sous-arbre droit à partir de la racine. En atteignant le nœud maximisant le plus à gauche du sous-arbre droit, α n'est pas mis à jour puisque -6 et -4 sont inférieurs à 3.

Démonstration



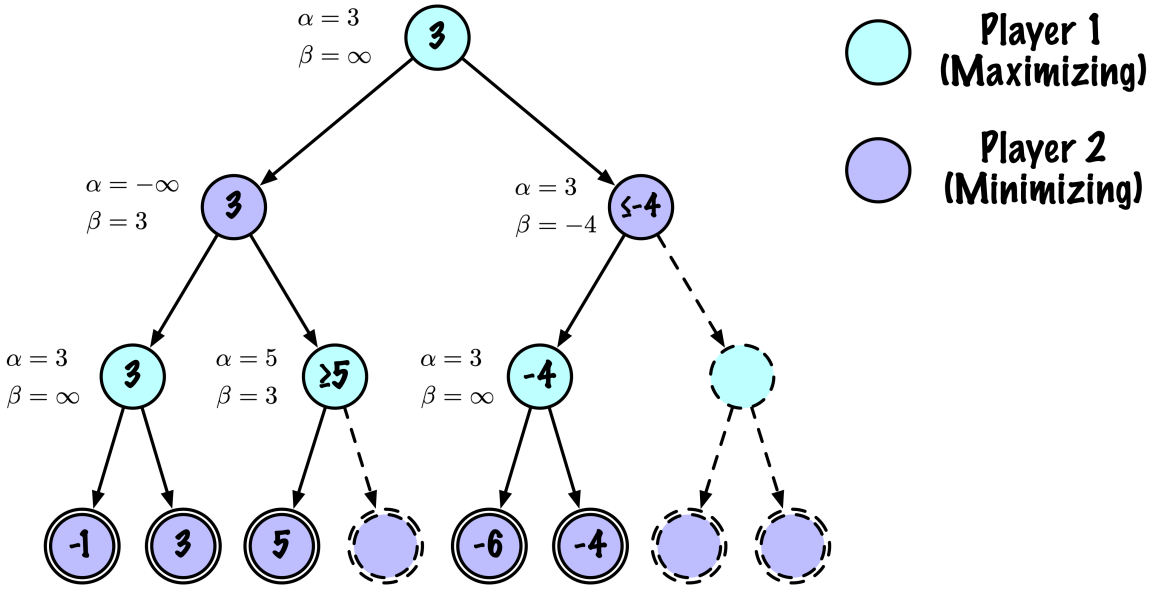
β est maintenant mis à jour à -4. Nous avons maintenant $\beta \leq \alpha$.

Démonstration



L'enfant droit est élagué.

Démonstration



Le joueur maximisant sait que son meilleur coup a une valeur de 3 (aller à gauche).

Vérification de validité

In [46]:
```
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}
```

Vérification de validité

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

a = RandomSolver(7)
b = MinimaxAlphaBetaSolverV1()

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

results
```

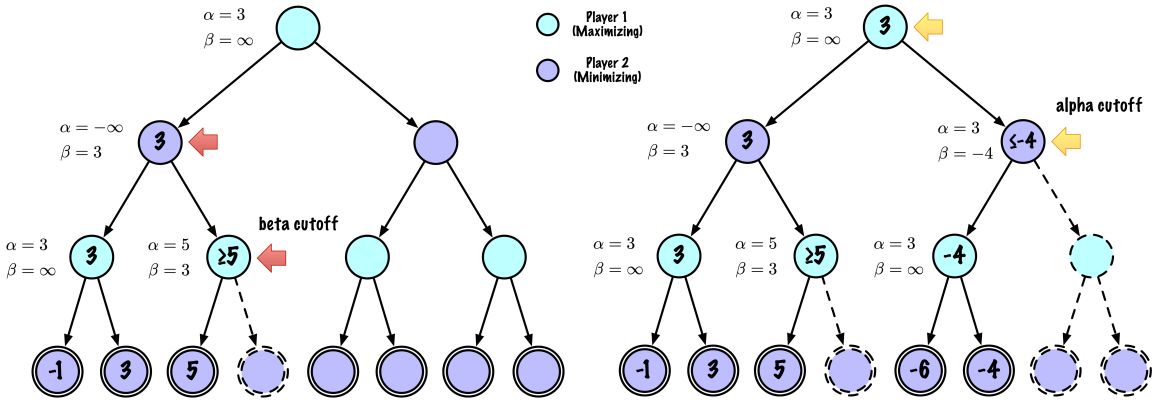
```
{'X_wins': 0, 'O_wins': 82, 'draws': 18}
```

Vérification de validité

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

a = MinimaxAlphaBetaSolverV1()
b = MinimaxAlphaBetaSolverV1()

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

results
```

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

Résumé



- **Coupure alpha :** Se produit aux **nœuds minimiseurs** lorsque $\beta \leq \alpha$.
- **Coupure bêta :** Se produit aux **nœuds maximiseurs** lorsque $\alpha \geq \beta$.

1. **Élagage aux deux types de nœuds :**
   - L'élagage peut survenir aussi bien lors des phases de minimisation que de maximisation. Cela signifie que les nœuds **minimiseurs** et **maximiseurs** peuvent être élagués si certaines conditions sont remplies.
2. **Mise à jour de alpha et bêta :**

- Aux **nœuds maximiseurs**, l'algorithme met à jour la valeur de **alpha** en prenant le maximum entre sa valeur actuelle et la valeur des nœuds enfants évalués jusqu'à présent.
- Aux **nœuds minimiseurs**, l'algorithme met à jour la valeur de **bêta** en prenant le minimum entre sa valeur actuelle et la valeur des nœuds enfants évalués jusqu'à présent.

3. **Conditions d'élagage :**
   - **Aux nœuds maximiseurs :**
     - Si **alpha** devient **supérieur ou égal à bêta** ($\alpha \geq \beta$), l'exploration des descendants du nœud courant peut être arrêtée. Cela s'explique par le fait que le minimiseur (l'adversaire) peut forcer le résultat à ne pas être meilleur que bêta, donc le maximiseur ne peut pas améliorer le résultat au-delà de ce point.
     - Cela est souvent appelé une **coupure bêta**, car c'est la valeur de bêta qui provoque l'élagage au niveau d'un nœud maximiseur.
   - **Aux nœuds minimiseurs :**
     - Si **bêta** devient **inférieur ou égal à alpha** ($\beta \leq \alpha$), l'algorithme peut élaguer les nœuds enfants restants du nœud minimiseur. Cela s'explique par le fait que le maximiseur peut forcer un résultat d'au moins alpha, donc le minimiseur ne peut pas trouver un meilleur (plus bas) résultat.
     - Cela est connu sous le nom de **coupure alpha**, car c'est la valeur de alpha qui provoque l'élagage au niveau d'un nœud minimiseur.
4. **Processus d'élagage :**
   - L'élagage ne se produit **pas lors de la mise à jour de alpha ou bêta**, mais lorsque la condition d'élagage ($\alpha \geq \beta$ aux nœuds maximiseurs ou $\beta \leq \alpha$ aux nœuds minimiseurs) est satisfaite.
   - Une fois ces conditions remplies, l'algorithme sait que poursuivre l'exploration n'apportera pas de meilleur résultat, et il peut donc élaguer ces branches en toute sécurité.

Résumé :

- **Coupure alpha :** Se produit aux **nœuds minimiseurs** lorsque $\beta \leq \alpha$.
- **Coupure bêta :** Se produit aux **nœuds maximiseurs** lorsque $\alpha \geq \beta$.
- **Pourquoi l'élagage a-t-il lieu :**
  - Dans les deux cas, l'élagage se produit car une exploration supplémentaire ne peut plus influencer la décision finale. L'adversaire peut forcer le jeu dans une situation qui ne sera pas meilleure que l'évaluation actuelle.
- **Impact sur l'efficacité de l'algorithme :**
  - En appliquant ces coupures, l'algorithme d'élagage alpha-bêta réduit le nombre de nœuds à évaluer par rapport à l'algorithme minimax classique, améliorant

ainsi l'efficacité sans affecter le résultat.

Discussion

- Comprendre **pourquoi l'élagage alpha-bêta améliore l'efficacité de l'algorithme minimax** sans modifier les résultats demande **une réflexion attentive**.
- Les **modifications de l'algorithme sont minimes**.
- Cette amélioration est-elle **justifiée**?

`MinimaxSolverV2` (INSTRUMENTÉ)

In [49]:
```python
class MinimaxSolverV2(Solver):

    """
    Un solveur Minimax classique et exact pour le Tic-Tac-Toe — *version ins

    Différences par rapport à V1
    ----------------------------
    - Ajoute un compteur `self.nodes_visited` qui compte chaque appel récurs
    - Utilise `reset()` afin que le GameRunner ou le TournamentRunner puisse
      préparer le solveur avant chaque partie.
    - Effectue toujours une recherche complète de l'arbre de jeu sans mémoïs

    Caractéristiques
    ----------------
    - Suppose que "X" est le joueur maximisant.
    - Effectue une recherche minimax complète (le Tic-Tac-Toe est suffisamme
    """

    def __init__(self):

        # Compte combien de nœuds ont été visités lors de l'exécution courar
        self.nodes_visited = 0

    # ------------------------------------------------------------------
    # Interface du solveur
    # ------------------------------------------------------------------

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

        """
        Choisir le coup optimal pour `player` à l'aide d'une recherche minim
        """

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

    def reset(self):
```

```
        """
        Réinitialise les compteurs d'instrumentation au début d'une partie (

        GameRunner / TournamentRunner doit appeler solver.reset() avant
        de commencer une nouvelle partie.
        """

        self.nodes_visited = 0

    def get_name(self):

        """
        Le nom est hérité de Solver mais les solveurs peuvent redéfinir get_
        pour afficher des informations d'instrumentation supplémentaires.
        """

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

    # ------------------------------------------------------------
    # Minimax récursif principal
    # ------------------------------------------------------------

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

        """
        Le cœur du calcul récursif minimax.

        Paramètres
        ----------
        game       : instance de TicTacToe
        state      : tableau NumPy représentant la position actuelle du plat
        player     : "X" ou "O" — le joueur dont c'est le tour à ce nœud
        maximizing : True si ce nœud correspond à X ; False si O

        Retourne
        -------
        (value, move)
            value : +1 si X gagne, -1 si O gagne, 0 sinon
            move  : meilleur coup sélectionné à ce nœud
        """

        # Instrumentation
        self.nodes_visited += 1

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

        # ----------------------
        # GÉNÉRER TOUS LES COUPS
        # ----------------------
        moves = game.get_valid_moves(state)
```

```
        # ------------------------
        # NŒUD MAX (X joue)
        # ------------------------
        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

        # ------------------------
        # NŒUD MIN (O joue)
        # ------------------------

        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` (INSTRUMENTÉ)

In [50]:

```
class MinimaxAlphaBetaSolverV2(Solver):

    """
    Un solveur Minimax classique amélioré avec l'élagage Alpha-Bêta,
    instrumenté pour compter le nombre de nœuds visités.

    - Suppose que "X" est le joueur maximisant.
    - Effectue une exploration complète de l'arbre du jeu de Tic-Tac-Toe.
    - L'élagage Alpha-Bêta réduit le nombre d'états explorés
```

```
    sans modifier le résultat final.

Instrumentation
---------------
- self.nodes_visited compte le nombre d'appels à _alphabeta().
"""

def __init__(self):

    # Compte le nombre de nœuds visités lors de l'exécution courante
    self.nodes_visited = 0

# ------------------------------------------------------------
# Interface du solveur
# ------------------------------------------------------------

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

    """
    Choisit le meilleur coup pour `player` en utilisant Minimax avec
    l'élagage Alpha-Bêta.

    Pour Tic-Tac-Toe, depth=9 suffit pour explorer tout le jeu.
    """

    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):

    """
    Réinitialise tout état interne spécifique à la partie.

    Appelé par GameRunner (ou équivalent) au début d'une nouvelle partie
    """

    self.nodes_visited = 0

# ------------------------------------------------------------
# Privé
# ------------------------------------------------------------

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

    """
```

```
        Recherche récursive interne minimax avec élagage alpha–bêta.

        Paramètres
        ----------
        state      : tableau NumPy, plateau courant
        player     : "X" ou "O", le joueur qui doit jouer
        maximizing : Vrai si ce nœud est maximisant (X doit jouer)
        depth      : profondeur de recherche restante
        alpha      : meilleure valeur trouvée jusqu'à présent pour le maximi
        beta       : meilleure valeur trouvée jusqu'à présent pour le minimi

        Retourne
        -------
        (value, move)
            value : évaluation de l'état du point de vue de X (+1/-1/0)
            move  : le meilleur coup trouvé à ce nœud
        """

        # Instrumentation : compter ce nœud
        self.nodes_visited += 1

        # Test terminal : victoire/défaite/nul ou profondeur atteinte
        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

        # ------------------------------------------------------------
        # Nœud maximisant (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,            # prochain nœud est minimisant
                    depth - 1,
                    alpha,
                    beta
                )

                if child_val > value:
                    value = child_val
                    best_move = move

                # Mettre à jour alpha
                alpha = max(alpha, value)

                # Élaguer
                if beta <= alpha:
                    break
```

```
            return value, best_move

        # ------------------------------------------------------------
        # Nœud minimisant (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,              # prochain nœud est maximisant
                    depth - 1,
                    alpha,
                    beta
                )

                if child_val < value:
                    value = child_val
                    best_move = move

                # Mettre à jour beta
                beta = min(beta, value)

                # Élaguer
                if beta <= alpha:
                    break

            return value, best_move
```

Pouvez-vous deviner le résultat ?

In [51]:
```
runner = GameRunner(game)

mm = MinimaxSolverV2()
ab = MinimaxAlphaBetaSolverV2()

outcome = runner.play_game(mm, ab)

print("Nœuds visités pour MinimaxSolverV2 :", mm.nodes_visited)
print("Nœuds visités pour MinimaxAlphaBetaSolverV2 :", ab.nodes_visited)
print(f"Efficacité de l'élagage : {(mm.nodes_visited - ab.nodes_visited) / m
```

```
Nœuds visités pour MinimaxSolverV2 : 557492
Nœuds visités pour MinimaxAlphaBetaSolverV2 : 2435
Efficacité de l'élagage : 0.996
```

Dans le jeu de tic-tac-toe, le nombre total de séquences d'actions valides s'élève à 255 168, ce qui correspond au nombre de nœuds feuilles dans l'arbre de jeu.

Ici, nous présentons le nombre total de nœuds visités, englobant à la fois les nœuds internes et les nœuds feuilles, ce qui explique le nombre total plus élevé.

Mémorisation

In [52]:
```python
class AlphaBetaSolver(Solver):

    """
    Un solveur classique Minimax amélioré avec l'élagage alpha-bêta.

    - Suppose que "X" est le joueur maximisant.
    - Utilise la mémorisation (lru_cache) pour éviter de recalculer les état
    - Effectue une recherche *complète* du Tic-Tac-Toe (profondeur=9).
    - Retourne le coup optimal pour le joueur actuel.
    """

    # ------------------------------------------------------------
    # Interface du solveur
    # ------------------------------------------------------------

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

        """
        Interface publique requise par Solver.
        Lance la recherche alpha-bêta à partir de l'état courant.
        """

        self.game = game

        maximizing = (player == "X")   # X maximise, O minimise

        # Réinitialise le cache entre les parties pour éviter de stocker des
        self._alphabeta.cache_clear()

        value, move = self._alphabeta(
            canonical(state),
            player,
            maximizing,
            9,               # recherche en profondeur complète
            -math.inf,       # alpha
            math.inf         # bêta
        )
        return move

    # ------------------------------------------------------------
    # Alpha-bêta interne avec mémorisation
    # ------------------------------------------------------------

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

        """
        Paramètres
        ----------
```

```
        state_key : plateau sous forme de tuple-de-tuples
        player    : joueur dont c'est le tour ('X' ou 'O')
        maximizing: True si ce nœud est un nœud maximisant pour X
        depth     : profondeur restante
        alpha     : meilleure valeur garantie pour le maximiseur jusqu'à pré
        beta      : meilleure valeur garantie pour le minimiseur jusqu'à pré
        """

        state = np.array(state_key)

        # Cas terminal ou d'horizon
        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,              # maintenant on minimise
                    depth - 1,
                    alpha,
                    beta
                )

                if child_val > value:
                    value = child_val
                    best_move = move

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

            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,               # maintenant on maximise
                    depth - 1,
                    alpha,
                    beta
                )

                if child_val < value:
                    value = child_val
                    best_move = move

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

            return value, best_move
```

Pour nos notes seulement.

Exploration

Implémentez un jeu de **Puissance 4 (Connect 4)** en utilisant l'**algorithme de recherche Alpha-Bêta**. Réalisez une analyse comparative entre les implémentations de Minimax et de recherche Alpha-Bêta.

Prologue

Exploration supplémentaire

- Recherche **Expetimax** : gérer les joueurs qui ne sont pas parfaits ;
- **Expectiminimax** : gérer le hasard dans des jeux tels que le backgammon.

Résumé

- Introduction à la recherche en environnement adversarial
- Jeux à somme nulle
- Introduction à la méthode de recherche minimax
- Rôle de l'élagage alpha et bêta dans la recherche minimax
- **Introduction à la recherche en environnement adversarial :**
    - Exploration des environnements compétitifs avec des objectifs conflictuels.

- **Classification des jeux :**
  - Types basés sur le déterminisme (déterministe vs stochastique).
  - Nombre de joueurs (un, deux ou plus).
  - Nature de la compétition (somme nulle vs non somme nulle).
  - Disponibilité de l'information (information parfaite vs imparfaite).
- **Jeux à somme nulle :**
  - Définition et caractéristiques.
  - Exemple : Tic-Tac-Toe comme jeu à somme nulle.
- **Cadre des jeux déterministes :**
  - Composants : états, joueurs, actions, fonctions de transition, états finaux, récompenses.
  - Développement de politiques des états initiaux aux états finaux.
- **Stratégies de jeu et complexité :**
  - Analyse des stratégies sans défaite dans le Tic-Tac-Toe.
  - Discussion sur l'impact de l'ordre des coups (premier ou deuxième joueur).
  - Exploration de la complexité des jeux dans le Tic-Tac-Toe, les échecs et le Go.
- **Jeu optimal et information parfaite :**
  - Concepts de stratégies optimales et leurs implications.
  - Importance de l'information parfaite en théorie des jeux.
- **Algorithme Minimax :**
  - Introduction à la méthode de recherche minimax.
  - Application pour déterminer les coups optimaux dans les jeux adversariaux.
  - Détails de l'implémentation avec un exemple en Python pour Tic-Tac-Toe.
- **Améliorations de l'efficacité :**
  - Utilisation de la mise en cache (mémorisation) pour améliorer les performances de l'algorithme.
  - Réduction de la surcharge computationnelle dans les arbres de recherche de jeu.
- **Techniques d'élagage :**
  - Introduction à l'élagage dans les arbres de recherche pour éviter des calculs inutiles.
  - Explication détaillée de l'élagage Alpha-Bêta.
  - Critères d'élagage et exemples illustrant le processus.
- **Élagage Alpha-Bêta :**
  - Intégration avec l'algorithme minimax.
  - Rôle des paramètres alpha et bêta dans l'optimisation de la recherche.
  - Impact sur le nombre de nœuds évalués.

- **Comparaison des performances :**
  - Analyse de l'exploration des nœuds entre minimax et élagage alpha-bêta.
  - Démonstration quantitative des gains d'efficacité.
- **Ordonnancement des nœuds et efficacité de l'élagage :**
  - Discussion sur l'effet de l'ordre d'évaluation des nœuds sur le succès de l'élagage.
  - Stratégies pour ordonner les nœuds afin de maximiser le potentiel d'élagage.

Prochain cours

- Nous aborderons l'algorithme de recherche arborescente Monte Carlo (MCTS)

Références

Russell, Stuart, et Peter Norvig. 2020. *Artificial Intelligence: A Modern Approach*. 4e éd. 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

École de **science informatique** et de génie électrique (**SI**GE)

Université d'Ottawa