Dauphine | PSL
TUNIS

Université PSL

Université Paris Dauphine

Master IASD

Implémentation des algorithmes Monte Carlo pour le jeu Connect 4

Rapport de projet

Présenté par : Mariem Inoubli
Hiba Kabeda
Azza Chebbi
Ferdaws Ziadia

Encadré par : Pr. Tristan Cazenave

Année universitaire 2024-2025

15 mars 2025

Résumé

Ce rapport présente une implémentation détaillée de l'algorithme Monte Carlo Tree Search (MCTS) pour le jeu Connect 4, un jeu de stratégie à deux joueurs. Nous explorons plusieurs variantes de MCTS, notamment le MCTS standard, MCTS avec RAVE (Rapid Action Value Estimation), GRAVE (Graph-Based Relative Average Value Estimation), et AMAF (All Moves As First). Le rapport détaille l'approche générale, les algorithmes utilisés, leur implémentation et les résultats comparatifs obtenus.

Mots-clés : Monte Carlo Tree Search, Connect 4, Intelligence Artificielle, RAVE, GRAVE, AMAF, Algorithmes de jeux.

Table des matières

Chapitre 1

Introduction

Le jeu Connect 4 est un jeu de stratégie à deux joueurs où l'objectif est d'aligner quatre jetons de sa couleur horizontalement, verticalement ou en diagonale. Dans ce rapport, nous explorons en détail l'implémentation de l'algorithme Monte Carlo Tree Search (MCTS) pour résoudre ce jeu. MCTS est une méthode de recherche heuristique qui utilise des simulations aléatoires pour évaluer les mouvements possibles. Nous avons implémenté plusieurs variantes de MCTS, notamment MCTS standard, MCTS avec RAVE (Rapid Action Value Estimation), GRAVE (Graph-Based Relative Average Value Estimation), et AMAF (All Moves As First). Ce rapport détaille notre approche, les algorithmes utilisés, leur implémentation, et les résultats obtenus.

1.1 Contexte du projet

Le jeu Connect 4, également connu sous le nom de "Puissance 4" en français, est un jeu de stratégie combinatoire abstrait. Il consiste en un plateau vertical divisé en colonnes, où les joueurs insèrent alternativement des jetons de leur couleur. L'objectif est d'aligner quatre jetons de sa couleur horizontalement, verticalement ou en diagonale. Ce jeu présente un défi intéressant car il comporte :

— Un espace d'états relativement grand (environ 4.5×10^{12} positions possibles)
— Des règles simples mais une stratégie complexe
— La nécessité d'anticiper les mouvements de l'adversaire

1.2 Objectifs du projet

Les objectifs principaux de ce projet sont :

— Créer une interface graphique permettant à un joueur humain d'affronter l'IA
— Implémenter l'algorithme MCTS standard pour le jeu Connect 4
— Développer plusieurs variantes de MCTS (RAVE, GRAVE, AMAF)
— Comparer les performances des différentes variantes

1.3 Structure du rapport

Ce rapport est structuré comme suit :

— **Chapitre 1 : Introduction** - Présente le contexte et les objectifs du projet
— **Chapitre 2 : Approche générale** - Détaille l'approche globale utilisée pour développer l'IA

— **Chapitre 3 : Implémentation des algorithmes** - Explique en détail les algorithmes MCTS, RAVE, GRAVE et AMAF

— **Chapitre 4 : Comparaison des algorithmes** - Présente les résultats comparatifs des différentes variantes

Chapitre 2

Approche générale

Notre objectif était de développer un agent intelligent capable de jouer au Connect 4 en utilisant différentes variantes de l'algorithme MCTS. Pour cela, nous avons structuré notre projet en plusieurs composants :

2.1 Architecture du système

L'architecture de notre système est composée de plusieurs modules interconnectés :

— **GameBoard** : Une classe représentant le plateau de jeu, gérant les mouvements, la vérification des victoires, et l'affichage du plateau.

— **MCTS** : Une classe implémentant l'algorithme MCTS standard, avec des méthodes pour la sélection, l'expansion, la simulation, et la rétropropagation.

— **RAVE, GRAVE, AMAF** : Des variantes de MCTS qui améliorent l'efficacité de l'algorithme en utilisant des techniques d'estimation de valeur plus sophistiquées.

— **Interface graphique** : Une interface utilisateur développée avec Pygame pour permettre à un joueur humain d'interagir avec l'agent.

2.2 Modélisation du jeu

Nous avons modélisé le jeu Connect 4 comme suit :

— Un plateau de jeu de 6 lignes et 7 colonnes

— Deux joueurs (rouge et jaune) qui placent alternativement leurs jetons

— Un jeton tombe toujours dans la position la plus basse disponible d'une colonne

— La victoire est obtenue en alignant 4 jetons de même couleur horizontalement, verticalement ou en diagonale

— Le jeu se termine par une victoire d'un des joueurs ou par un match nul (plateau plein)

2.3 Choix technologiques

Pour implémenter notre système, nous avons fait les choix technologiques suivants :

— **Langage de programmation** : Python, pour sa facilité d'utilisation et ses nombreuses bibliothèques

— **Interface graphique** : Pygame, une bibliothèque Python dédiée au développement de jeux

— **Structures de données** : Utilisation de NumPy pour les opérations sur les tableaux, offrant de meilleures performances
— **Tests** : Pytest pour les tests unitaires et fonctionnels

2.4 Processus de développement

Le développement du projet a suivi les étapes suivantes :

1. Implémentation du jeu Connect 4 et de ses règles
2. Développement de l'algorithme MCTS standard
3. Extension de MCTS avec les variantes RAVE, GRAVE et AMAF
4. Création de l'interface graphique
5. Tests et comparaisons des performances

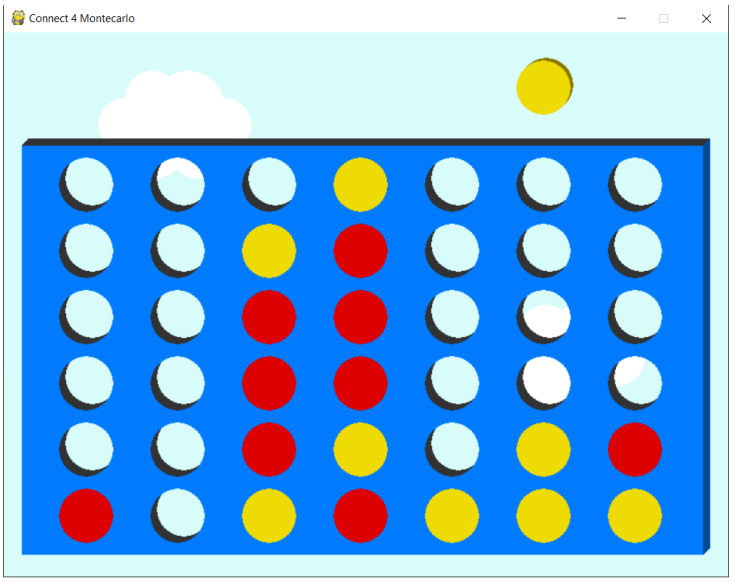

FIGURE 2.1 – Interface Grapique

FIGURE 2.2 – MC with GRAVE

Chapitre 3

Implémentation des algorithmes

3.1 MCTS standard

L'algorithme Monte Carlo Tree Search (MCTS) standard est une méthode de recherche heuristique particulièrement adaptée aux jeux avec un grand espace d'états comme Connect 4. Contrairement aux algorithmes traditionnels comme Minimax, MCTS ne nécessite pas d'évaluation statique de l'état du jeu, ce qui le rend idéal pour les jeux où il est difficile de concevoir une telle fonction d'évaluation.

3.1.1 Fondements théoriques et adaptation à Connect 4

MCTS repose sur le principe de l'exploitation et l'exploration équilibrée de l'arbre de jeu. Dans le contexte du jeu Connect 4, cette approche est particulièrement pertinente pour les raisons suivantes :

— **Espace d'états complexe** : Connect 4 possède environ 4.5×10^{12} positions possibles, rendant impossible l'exploration exhaustive.

— **Branchement limité** : Contrairement aux échecs ou au go, Connect 4 a un facteur de branchement relativement faible (maximum 7 coups possibles par tour), ce qui permet à MCTS d'atteindre une profondeur significative.

— **Terminaison claire** : Les parties de Connect 4 se terminent avec un vainqueur défini ou un match nul, ce qui facilite l'étape de simulation.

— **Structure incrémentale** : Dans Connect 4, les jetons s'accumulent du bas vers le haut, créant une contrainte naturelle qui réduit l'espace de recherche.

3.1.2 Étapes de l'algorithme et implémentation

L'algorithme MCTS standard que nous avons implémenté suit quatre étapes principales, chacune ayant été adaptée aux spécificités du jeu Connect 4 :

Sélection

La phase de sélection utilise la formule UCT (Upper Confidence Bound for Trees) pour équilibrer l'exploration de nouveaux nœuds et l'exploitation des nœuds prometteurs :

$$UCT = \frac{Q_i}{N_i} + c\sqrt{\frac{\ln N_p}{N_i}} \tag{3.1}$$

où Q_i représente le score total du nœud, N_i le nombre de visites, N_p le nombre de visites du parent, et c une constante d'exploration (fixée à 2 dans notre implémentation).

Pour Connect 4, notre méthode `select_uct` parcourt tous les enfants d'un nœud et choisit celui ayant la valeur UCT la plus élevée :

```python
def select_uct(self, node: "Node") -> "Node":
    """Select node with best UCT value."""
    best_uct = -np.inf
    best_node = None
    for child in node.children:
        uct = (child.q / child.n) + 2 * np.sqrt(np.log(node.n) / child.n
            )
        if uct > best_uct:
            best_uct = uct
            best_node = child
    if best_node is None:
        return node
    return best_node
```

Listing 3.1 – Méthode select_uct pour la sélection UCT

Cette implémentation est particulièrement efficace pour Connect 4 car elle permet de rapidement identifier les colonnes prometteuses tout en continuant d'explorer périodiquement les alternatives.

Expansion

Lors de l'expansion, un nouveau nœud enfant est ajouté à l'arbre pour chaque mouvement légal possible à partir du nœud sélectionné. Dans Connect 4, cela correspond à l'ajout d'un jeton dans chaque colonne non pleine.

Notre méthode `add_child` crée de nouveaux états de jeu en plaçant un jeton dans la position la plus basse disponible de chaque colonne :

```python
def add_child(self) -> None:
    """Add new child to node."""
    # V rification si d  j    tendu
    if self.expanded:
        return
    # R cup ration des plateaux enfants existants
    child_board = []
    for child in self.children:
        child_board.append(child.board)
    # Recherche de nouveaux enfants
    for i in range(7):  # Pour chaque colonne
        if self.board[5, i] == 0:  # Si la colonne n'est pas pleine
            for j in range(6):  # Trouver la position du jeton
                if self.board[j, i] == 0:
                    # Cr er le nouvel  tat
                    tmp = self.board.copy()
                    tmp[j, i] = 2 if self.turn == 1 else 1
                    # V rifier si cet  tat  n'existe pas d  j
                    if not child_board or not self.compare_children(tmp,
                        child_board):
                        self.children.append(Node(self, tmp, 1 if self.
                            turn == 2 else 2))
                        return
                    break
    # Aucun nouvel enfant possible
    self.expanded = True
```

Listing 3.2 – Méthode add_child pour l'expansion des nœuds

Cette implémentation est optimisée pour Connect 4 car elle tient compte de la physique du jeu (les jetons tombent au bas de la colonne) et permet d'éviter la création de nœuds en double.

Simulation (Rollout)

La phase de simulation consiste à jouer des parties aléatoires à partir du nœud nouvellement créé jusqu'à atteindre un état terminal. Notre méthode `rollout` effectue cette simulation en sélectionnant des mouvements aléatoires pour les deux joueurs.

Cette approche est particulièrement efficace pour Connect 4 car elle permet d'explorer rapidement l'arbre de jeu sans nécessiter de connaissances spécifiques ou d'heuristiques complexes.

Rétropropagation

La phase de rétropropagation met à jour les statistiques (nombre de visites et score) des nœuds visités pendant la sélection, de la feuille jusqu'à la racine. Notre méthode `backpropagate` implémente cette mise à jour récursive :

```
def backpropagate(self, node: "Node", winner: Optional[int]) -> None:
    """Update recursively node visits and scores from leaf to root."""
    # Incr mentation du score si victoire
    if node.turn == winner:
        node.q += 1
    # Incr mentation du nombre de visites
    node.n += 1
    # Arr t si n ud racine
    if node.parent is None:
        return
    # Appel r cursif sur le parent
    self.backpropagate(node.parent, winner)
```

Listing 3.3 – Méthode backpropagate pour la mise à jour des statistiques

3.1.3 Complexité et performances pour Connect 4

L'algorithme MCTS présente plusieurs avantages pour le jeu Connect 4 :

— **Complexité temporelle** : La complexité de chaque itération MCTS est $O(D)$ où D est la profondeur de l'arbre, ce qui est favorable pour Connect 4 où la profondeur maximale est de 42 (7 colonnes × 6 lignes).

— **Convergence** : MCTS converge asymptotiquement vers la stratégie optimale avec un nombre suffisant d'itérations, ce qui est crucial pour un jeu comme Connect 4 où il existe des stratégies gagnantes connues.

— **Anytime** : MCTS peut être interrompu à tout moment et fournir une estimation du meilleur coup, ce qui permet de contrôler le compromis entre temps de calcul et qualité de la décision.

Notre implémentation utilise une limite de temps fixe (`PROCESS_TIME`) pour déterminer le nombre d'itérations à effectuer, ce qui permet d'adapter la profondeur de recherche aux contraintes temps-réel du jeu.

3.1.4 Sélection du meilleur coup

Une fois que le temps alloué est écoulé, l'algorithme sélectionne le meilleur coup à jouer. Nous avons choisi d'utiliser le critère du nombre de visites (plutôt que le score moyen) pour cette sélection, car il est plus robuste aux erreurs d'évaluation :

```python
def best_child(self, node: "Node") -> Optional["Node"]:
    """Get child node with largest number of visits."""
    max_visit = 0
    best_node = None
    for child in node.children:
        if child.n > max_visit:
            max_visit = child.n
            best_node = child
    return best_node
```

Listing 3.4 – Méthode best_child pour la sélection du meilleur coup

Cette approche garantit que le coup sélectionné est celui qui a été le plus exploré, ce qui correspond généralement au coup le plus prometteur à long terme.

3.2 MCTS avec RAVE

3.2.1 Principe et intérêt de RAVE

RAVE (Rapid Action Value Estimation) est une amélioration significative de l'algorithme MCTS standard qui permet d'accélérer la convergence en exploitant davantage d'informations issues des simulations. Cette technique repose sur l'hypothèse *all-moves-as-first* (AMAF), qui suggère que la valeur d'une action reste relativement indépendante du moment où elle est jouée durant une partie.

Pour Connect4, RAVE est particulièrement pertinent pour plusieurs raisons :

— L'impact d'un jeton placé à une position donnée est souvent similaire indépendamment du moment exact du placement

— L'espace d'états est vaste mais le nombre d'actions possibles est limité (7 colonnes)

— La reconnaissance des motifs gagnants (alignements de 4 jetons) bénéficie d'une estimation rapide de la valeur des mouvements

3.2.2 Modification de l'algorithme

Structure des nœuds

Dans notre implémentation, chaque nœud de l'arbre MCTS maintient des statistiques supplémentaires pour RAVE :

```python
self.q = 0       # Somme standard des r sultats des simulations
self.n = 0       # Nombre standard de visites
self.rave_q = 0 # Somme des r sultats RAVE pour cette action
self.rave_n = 0 # Nombre de "visites RAVE" pour cette action
```

Listing 3.5 – Structure d'un nœud avec RAVE

Sélection avec RAVE

La formule UCT classique est étendue pour incorporer les statistiques RAVE selon l'équation :

$$Score(s,a) = (1 - \beta(s,a)) \cdot UCT(s,a) + \beta(s,a) \cdot RAVE(s,a) \tag{3.2}$$

Où :

— $UCT(s,a) = \frac{Q(s,a)}{N(s,a)} + c \cdot \sqrt{\frac{\ln N(s)}{N(s,a)}}$ représente la valeur UCT standard

— $RAVE(s,a) = \frac{Q_{RAVE}(s,a)}{N_{RAVE}(s,a)}$ représente la valeur RAVE

— $\beta(s,a)$ est un coefficient de pondération qui diminue progressivement avec le nombre de visites

L'implémentation de cette sélection est donnée par :

```python
def select_uct(self, node: "Node") -> "Node":
    """Select node with best UCT value, including RAVE."""
    best_uct = -np.inf
    best_node = None
    for child in node.children:
        # Classic UCT value
        uct = (child.q / child.n) + 2 * np.sqrt(np.log(node.n) / child.n
            )
        # RAVE value
        beta = np.sqrt(self.t / (3 * node.n + self.t))  # Weight for
            RAVE
        rave_value = (child.rave_q / child.rave_n) if child.rave_n > 0
            else 0
        # Combined value
        combined_value = (1 - beta) * uct + beta * rave_value
        if combined_value > best_uct:
            best_uct = combined_value
            best_node = child
    return best_node if best_node is not None else node
```

Listing 3.6 – Méthode select_uct avec RAVE

Le coefficient $\beta = \sqrt{\frac{t}{3 \cdot node.n + t}}$ joue un rôle crucial :

— Au début (peu de visites), β est proche de 1, privilégiant les statistiques RAVE

— À mesure que le nombre de visites augmente, β diminue, donnant plus d'importance aux statistiques UCT

— Le paramètre t (temps alloué) permet d'ajuster la vitesse de cette transition

Rétropropagation avec RAVE

La phase de rétropropagation est également modifiée pour mettre à jour les statistiques RAVE :

```python
def backpropagate(self, node: "Node", winner: Optional[int], actions:
    List[Tuple[int, int]]) -> None:
    """Update recursively node visits and scores from leaf to root,
        including RAVE."""
    if node.turn == winner:
        node.q += 1
    node.n += 1

    # Update RAVE statistics for all actions in the simulation
    for action in actions:
        for child in node.children:
            if (child.board[action[0], action[1]] != node.board[action
                [0], action[1]]):
                child.rave_q += (1 if node.turn == winner else 0)
                child.rave_n += 1

    if node.parent is None:
        return
```

```
16      self.backpropagate(node.parent, winner, actions)
```

Listing 3.7 – Rétropropagation avec mise à jour RAVE

Cette implémentation enregistre toutes les actions de la simulation et met à jour les statistiques RAVE pour tous les nœuds enfants correspondants, pas seulement ceux qui ont été visités pendant la phase de sélection.

3.2.3 Avantages pour Connect4

L'algorithme MCTS avec RAVE présente plusieurs avantages spécifiques pour le jeu Connect4 :

1. **Convergence accélérée** : RAVE permet d'identifier les mouvements prometteurs beaucoup plus rapidement que le MCTS standard, ce qui est crucial étant donné la contrainte de temps de calcul imposée dans notre implémentation ($t = 3.0$ secondes).
2. **Exploitation des motifs** : Connect4 étant un jeu basé sur la formation d'alignements, RAVE aide à identifier plus efficacement les positions qui contribuent régulièrement à des séquences gagnantes.
3. **Adaptation dynamique** : Le mécanisme de pondération β permet d'équilibrer automatiquement l'exploration et l'exploitation à mesure que l'arbre de recherche se développe.
4. **Meilleure utilisation des données** : Chaque simulation alimente non seulement les statistiques des nœuds visités mais aussi celles des actions similaires dans d'autres branches de l'arbre, maximisant ainsi l'information extraite de chaque simulation.

Ces caractéristiques font de RAVE une amélioration particulièrement pertinente pour MCTS dans le contexte du jeu Connect4, permettant à l'IA de jouer à un niveau élevé même avec un temps de calcul limité.

3.3 GRAVE

GRAVE (Graph-Based Relative Average Value Estimation) est une extension de RAVE qui utilise un graphe pour partager les statistiques entre les nœuds similaires. Cela permet une estimation plus précise de la valeur des mouvements, en particulier dans les jeux avec un grand nombre d'états similaires.

3.3.1 Structure et fonctionnement de la classe GRAVE

La classe GRAVE est le cœur de cette implémentation et représente l'intelligence artificielle qui joue au Connect4. Voici les aspects clés de cette classe :

Initialisation et structure de données

```
1  def __init__(self, symbol: int, t: float) -> None:
2      self.symbol = symbol
3      self.t = t
4      self.graph: Dict[Tuple[int, int], Tuple[int, int]] = {}
```

Listing 3.8 – Initialisation et structure de données

Le dictionnaire self.graph est la caractéristique distinctive de GRAVE. Il stocke pour chaque action (représentée par une position (row, column)) une paire de valeurs (q, n) où :

- q est la somme des récompenses obtenues lorsque cette action a été effectuée
- n est le nombre de fois que cette action a été jouée

Cette structure permet à GRAVE de maintenir des statistiques globales sur les actions, indépendamment de la position exacte dans l'arbre de recherche.

Sélection des nœuds avec GRAVE :

```python
def select_grave(self, node: "Node") -> "Node":
    """Select node with best GRAVE value."""
    best_value = -np.inf
    best_node = None
    for child in node.children:
        # Classic UCT value
        uct = (child.q / child.n) + 2 * np.sqrt(np.log(node.n) / child.n
            )
        # GRAVE value
        grave_value = self.get_grave_value(child)
        # Combined value
        combined_value = uct + grave_value
        if combined_value > best_value:
            best_value = combined_value
            best_node = child
    return best_node if best_node is not None else node
```

Listing 3.9 – select_grave

Cette méthode sélectionne le nœud enfant avec la meilleure valeur combinée composée de :

- La valeur UCT classique (exploitation/exploration)
- La valeur GRAVE spécifique à l'action

La valeur GRAVE est obtenue via la méthode get_grave_value :

```python
def get_grave_value(self, child: "Node") -> float:
    """Get GRAVE value for a child node."""
    action = self.get_action(child.parent.board, child.board)
    if action is None:
        return 0
    q, n = self.graph.get(action, (0, 0))
    return q / n if n > 0 else 0
```

Listing 3.10 – get_grave_value

Cette méthode extrait la valeur GRAVE pour un nœud enfant en :

- Identifiant l'action qui a conduit du nœud parent à ce nœud enfant
- Récupérant les statistiques globales (q, n) pour cette action à partir du graphe
- Calculant la valeur moyenne (q/n)

Mise à jour des statistiques GRAVE

```python
def backpropagate(self, node: "Node", winner: Optional[int], actions:
    List[Tuple[int, int]]) -> None:
    """Update recursively node visits and scores from leaf to root,
        including GRAVE."""
    if node.turn == winner:
        node.q += 1
    node.n += 1
    # Update GRAVE statistics for all actions in the simulation
    for action in actions:
        q, n = self.graph.get(action, (0, 0))
        q += 1 if node.turn == winner else 0
        n += 1
        self.graph[action] = (q, n)
```

```
12      if node.parent is not None:
13          self.backpropagate(node.parent, winner, actions)
```

Listing 3.11 – backpropagation

La méthode de rétropropagation met à jour :

- Les statistiques classiques MCTS (q, n) pour chaque nœud
- Les statistiques GRAVE pour chaque action jouée durant la simulation

L'aspect clé est la mise à jour du graphe GRAVE pour toutes les actions effectuées lors de la simulation, indépendamment de l'endroit où elles ont été jouées dans l'arbre.

3.3.2 Avantages de GRAVE par rapport à MCTS classique et RAVE

Partage d'information amélioré

GRAVE permet un partage d'information plus efficace entre différentes parties de l'arbre de recherche. Contrairement à RAVE qui partage uniquement l'information entre des nœuds frères, GRAVE maintient des statistiques globales pour chaque action possible, indépendamment de sa position dans l'arbre.

Estimation plus précise pour les actions rares

Pour les actions qui sont jouées rarement dans certaines branches de l'arbre, GRAVE peut fournir des estimations plus fiables en exploitant les informations recueillies dans d'autres parties de l'arbre.

Meilleure exploitation de l'expérience de jeu

GRAVE accumule des connaissances sur les actions dans un graphe global, permettant à l'algorithme d'exploiter toute l'expérience acquise lors des simulations précédentes.

Algorithme plus simple et plus efficace

Par rapport à RAVE, GRAVE a une formulation plus simple et plus directe. Il n'y a pas besoin de calculer un coefficient β pour équilibrer les valeurs RAVE et UCT - on additionne simplement la valeur UCT et la valeur GRAVE.

Complexité de l'implémentation

GRAVE a une implémentation plus simple grâce à sa structure de graphe centralisée. RAVE nécessite de mettre à jour les statistiques séparément pour chaque nœud enfant.

L'implémentation de GRAVE pour Connect4 représente une amélioration significative par rapport aux algorithmes MCTS classiques et RAVE. En maintenant des statistiques globales sur les actions dans un graphe, GRAVE permet une meilleure exploitation de l'expérience acquise et une convergence plus rapide vers des stratégies optimales.
Cette approche est particulièrement adaptée aux jeux comme Connect4 où les motifs se répètent dans différentes configurations du plateau. La simplicité conceptuelle de GRAVE, combinée à son efficacité, en fait une excellente solution pour développer une IA performante pour Connect4 avec des ressources de calcul limitées.

3.4 AMAF

AMAF (All Moves As First) est une autre variante de MCTS qui utilise les statistiques de toutes les actions prises pendant les simulations, indépendamment de l'ordre dans lequel elles ont été prises. Cela permet une estimation plus rapide de la valeur des mouvements, en particulier dans les jeux où les actions ont un impact immédiat.

3.4.1 Fonctionnement principal pour AMAF

La structure de données clé

```python
def __init__(self, symbol: int, t: float) -> None:
    self.symbol = symbol
    self.t = t
    self.amaf_stats: Dict[Tuple[int, int], Tuple[int, int]] = {}
```

Listing 3.12 – Méthode init pour AMAF

L'élément central d'AMAF est le dictionnaire amaf_stats qui stocke pour chaque action possible (représentée par une position (row, column)) une paire de valeurs (q, n) où :

- q est la somme des récompenses obtenues lorsque cette action a été jouée
- n est le nombre de fois que cette action a été jouée

Ce dictionnaire permet à AMAF de maintenir des statistiques globales sur les actions, indépendamment du moment où elles ont été jouées dans l'arbre.

La sélection de nœuds avec AMAF

```python
def select_amaf(self, node: "Node") -> "Node":
    """Select node with best AMAF value."""
    best_value = -np.inf
    best_node = None
    for child in node.children:
        # Classic UCT value
        uct = (child.q / child.n) + 2 * np.sqrt(np.log(node.n) / child.n
            )
        # AMAF value
        amaf_value = self.get_amaf_value(child)
        # Combined value
        combined_value = uct + amaf_value
        if combined_value > best_value:
            best_value = combined_value
            best_node = child
    return best_node if best_node is not None else node
```

Listing 3.13 – Méthode pour la séléction des noeuds avec AMAF select_amaf

Cette méthode sélectionne le nœud avec la meilleure valeur combinée :

- La valeur UCT classique (compromis entre exploration et exploitation)
- La valeur AMAF spécifique à l'action

La valeur AMAF est calculée par la méthode get_amaf_value :

```python
def get_amaf_value(self, child: "Node") -> float:
    """Get AMAF value for a child node."""
    action = self.get_action(child.parent.board, child.board)
    if action is None:
```

```
        return 0
    q, n = self.amaf_stats.get(action, (0, 0))
    return q / n if n > 0 else 0
```

Listing 3.14 – Méthode pour calculer la valeur AMAF get_amaf_value

Cette méthode :

- Identifie l'action qui mène du nœud parent au nœud enfant
- Récupère les statistiques AMAF (q, n) pour cette action
- Calcule la valeur moyenne (q/n)

La simulation et collecte de données pour AMAF

```
def rollout(self, node: "Node") -> Tuple[Optional[int], List[Tuple[int, int]]]:
    """Perform a random game simulation and record actions for AMAF."""
    board, turn, actions = node.board, node.turn, []
    while not node.terminal:
        turn = 3 - turn
        moves = self.get_moves(board, turn)
        if not moves:
            break
        next_board = random.choice(moves)
        action = self.get_action(board, next_board)
        if action is not None:
            actions.append(action)
        board = next_board
        terminal = self.result(board)
        if terminal != 0:
            return terminal, actions
    return self.result(board), actions
```

Listing 3.15 – Méthode rollout

Pendant la phase de simulation, l'algorithme enregistre toutes les actions effectuées, quelle que soit leur position dans la séquence de jeu. C'est le principe fondamental d'AMAF : considérer que la valeur d'une action est la même, qu'elle soit jouée en premier ou plus tard dans la partie.

La mise à jour des statistiques AMAF

```
def backpropagate(self, node: "Node", winner: Optional[int], actions: List[Tuple[int, int]]) -> None:
    """Update recursively node visits and scores from leaf to root, including AMAF."""
    if node.turn == winner:
        node.q += 1
    node.n += 1
    # Update AMAF statistics for all actions in the simulation
    for action in actions:
        q, n = self.amaf_stats.get(action, (0, 0))
        q += 1 if node.turn == winner else 0
        n += 1
        self.amaf_stats[action] = (q, n)
    if node.parent is not None:
        self.backpropagate(node.parent, winner, actions)
```

Listing 3.16 – backpropagation avec AMAF

Lors de la rétropropagation, AMAF met à jour :

1. Les statistiques MCTS classiques pour chaque nœud
2. Les statistiques AMAF pour toutes les actions jouées durant la simulation

L'aspect crucial est que AMAF met à jour les statistiques pour chaque action indépendamment de l'ordre dans lequel elles ont été jouées. C'est ce qui donne à AMAF son nom : "All Moves As First" (tous les coups comme s'ils étaient joués en premier).

3.4.2 Avantages d'AMAF pour Connect4

1. **Apprentissage plus rapide**
   AMAF permet un apprentissage plus rapide en utilisant plus efficacement l'information des simulations. En considérant toutes les actions comme si elles étaient jouées en premier, l'algorithme accumule plus rapidement des statistiques sur les actions.
2. **Meilleure exploration de l'espace des états**
   Pour Connect4, où l'espace des états est très vaste, AMAF permet une exploration plus efficace en identifiant plus rapidement les actions prometteuses.
3. **Réduction de la variance**
   En agrégeant les informations sur les actions à travers différentes parties de l'arbre, AMAF réduit la variance des estimations de valeur, ce qui conduit à des décisions plus stables.
4. **Performance améliorée avec un temps limité**
   Avec un temps de calcul limité (ici 5 secondes par coup), AMAF peut prendre de meilleures décisions car il exploite plus efficacement l'information disponible dans les simulations.

Différences avec MCTS standard

1. **Partage d'information global**
   Dans MCTS standard, les informations sont strictement locales à chaque nœud. AMAF partage les informations sur les actions à travers tout l'arbre, permettant une meilleure utilisation des expériences passées.
2. **Traitement équivalent des actions**
   AMAF traite toutes les actions d'une simulation comme ayant la même importance, indépendamment de leur ordre. Cela part du principe que dans Connect4, la qualité d'une action est souvent indépendante du moment où elle est jouée.
3. **Calcul de la valeur des nœuds**
   AMAF modifie la fonction de sélection UCT traditionnelle en ajoutant un terme basé sur les statistiques globales des actions, permettant une meilleure évaluation des nœuds peu explorés.

Chapitre 4

Comparaison des algorithmes

L'évaluation se déroule en deux phases :

1. **Affrontement contre un joueur aléatoire**
   Chaque algorithme est testé contre un adversaire jouant de manière aléatoire afin de mesurer son taux de victoire, son temps de calcul et le nombre de simulations effectuées.
2. **Tournoi entre algorithmes**
   Les variantes de MCTS s'affrontent dans un tournoi à élimination directe. Les algorithmes sont appariés aléatoirement et s'affrontent en plusieurs parties, le plus performant avançant jusqu'à la finale.

Les variantes analysées incluent :

— MCTS-UCT (Upper Confidence bounds applied to Trees)
— MCTS-RAVE (Rapid Action Value Estimation)
— MCTS-GRAVE (Generalized RAVE)
— MCTS-AMAF (All Moves As First)

Grâce à cette évaluation, nous cherchons à identifier l'algorithme offrant le meilleur compromis entre efficacité, rapidité et coût computationnel dans le cadre du jeu du Puissance 4.

4.1 MCTS UCB

4.1.1 MCTS UCB contre un joueur aléatoire

Interprétation des Résultats

L'évaluation de l'algorithme MCTS-UCB contre un joueur aléatoire dans le jeu Connect 4 révèle des performances significativement supérieures de l'approche MCTS. Cette analyse examine les résultats obtenus sur un ensemble de 5 parties simulées.

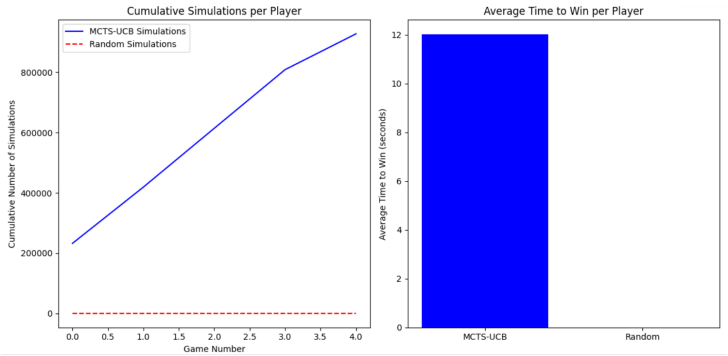

FIGURE 4.1 – MCTS UCB vs Random player

Taux de Victoire :

Les résultats montrent une domination complète de l'algorithme MCTS-UCB sur le joueur aléatoire :

— MCTS-UCB a remporté 100

— Le joueur aléatoire n'a gagné aucune partie (0%)

— Aucun match nul n'a été enregistré (0%)

Cette victoire totale était prévisible étant donné la nature du jeu Connect 4 et la différence fondamentale entre une stratégie éclairée (MCTS) et une approche aléatoire. Connect 4 est un jeu résolu où le premier joueur, avec une stratégie optimale, peut toujours gagner, ce que l'algorithme MCTS-UCB parvient à exploiter efficacement.

Simulations Cumulées (Figure de gauche) :

Le graphique des simulations cumulatives illustre l'écart considérable entre les deux approches :

— La courbe bleue (MCTS-UCB) montre une croissance rapide et constante, atteignant approximativement 950 000 simulations cumulées après 5 parties

— La ligne rouge pointillée (joueur aléatoire) reste proche de zéro tout au long des parties

Cette différence massive s'explique par la nature même des deux approches : alors que MCTS-UCB effectue des milliers de simulations pour chaque décision, explorant systématiquement l'arbre de jeu, le joueur aléatoire ne fait qu'un seul choix aléatoire par tour. Le paramètre UCB (avec t=3.0 dans l'implémentation) guide efficacement l'exploration de l'algorithme en équilibrant l'exploitation des stratégies prometteuses et l'exploration de nouvelles possibilités.

Temps Moyen pour Gagner (Figure de droite) :

Le graphique du temps moyen pour gagner met en évidence l'investissement computationnel nécessaire pour l'algorithme MCTS-UCB :

— MCTS-UCB prend en moyenne environ 12 secondes pour remporter une partie complète

— Aucune donnée n'est disponible pour le joueur aléatoire puisqu'il n'a gagné aucune partie

Ce temps reflète la complexité calculatoire de l'algorithme MCTS qui doit simuler de nombreuses parties virtuelles pour évaluer la qualité des coups potentiels. Il est important de noter que ce temps reste raisonnable pour une application pratique dans un jeu comme Connect 4.

Conclusion

L'analyse des résultats démontre la supériorité incontestable de l'algorithme MCTS-UCB face à un joueur aléatoire dans le contexte du jeu Connect 4. Cette performance peut être attribuée à plusieurs facteurs clés :

1. L'exploration intelligente de l'arbre de jeu par MCTS, qui permet d'identifier et d'exploiter des stratégies gagnantes
2. L'utilisation efficace du paramètre UCB pour équilibrer l'exploration et l'exploitation
3. La nature déterministe du jeu Connect 4, qui favorise une approche systématique plutôt qu'aléatoire

Ces résultats soulignent l'efficacité des algorithmes basés sur MCTS pour les jeux à information parfaite comme Connect 4. Ils confirment également que le coût computationnel plus élevé de cette approche est justifié par les performances supérieures obtenues.

4.1.2 MCTS-UCB contre MCTS-RAVE

Interprétation des Résultats

Cette analyse compare les performances des algorithmes RAVE (Rapid Action Value Estimation) et MCTS-UCB (Monte Carlo Tree Search avec Upper Confidence Bound). Dans cette analyse, nous évaluons la supériorité de l'algorithme MCTS-UCB par rapport à RAVE dans le jeu de Connect 4 en nous appuyant sur plusieurs métriques clés. Les résultats obtenus montrent une domination de MCTS-UCB, qui se révèle plus performant en termes d'exploration et d'exploitation de l'espace de recherche.

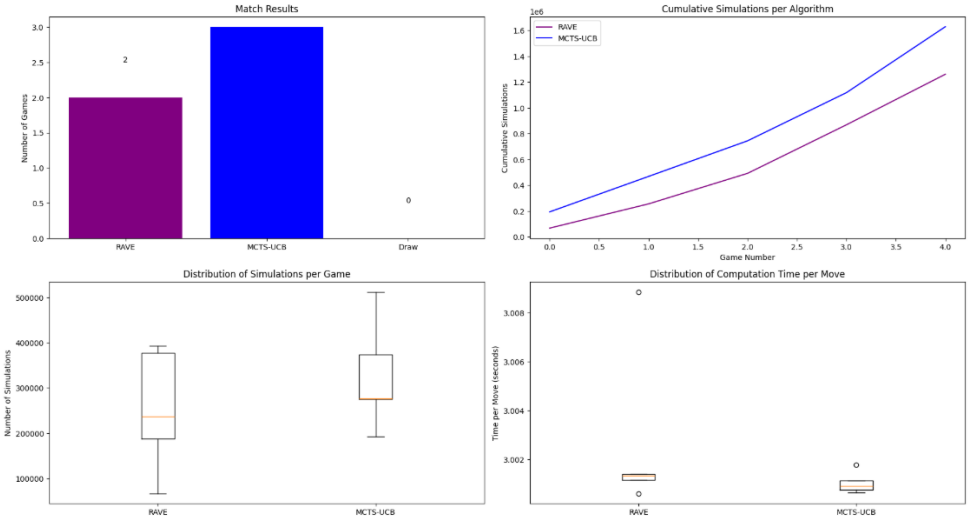

FIGURE 4.2 – MCTS UCB vs Rave

Taux de Victoire :

L'analyse des résultats montre une performance légèrement supérieure de l'algorithme MCTS-UCB par rapport à RAVE :

— MCTS-UCB a remporté 60% des parties (3 sur 5)

— RAVE a remporté 40% des parties (2 sur 5)

— Aucun match nul n'a été enregistré

Ces résultats suggèrent une légère dominance de l'algorithme MCTS-UCB. Cependant, le fait que RAVE ait remporté 2 parties indique qu'il reste compétitif face à MCTS-UCB, contrairement au joueur aléatoire qui n'avait aucune chance contre ces algorithmes avancés.

Simulations Cumulées (Figure de droite) :

Le graphique des simulations cumulatives révèle des différences intéressantes dans le comportement des deux algorithmes :

— La courbe bleue (MCTS-UCB) montre une croissance plus rapide, atteignant environ 1,6 millions de simulations après 5 parties

— La courbe violette (RAVE) progresse plus lentement, atteignant environ 1,2 millions de simulations sur la même période

Cette différence de volume de simulations indique que MCTS-UCB est capable d'explorer un plus grand nombre de possibilités dans le même intervalle de temps (3 secondes par coup). L'écart entre les deux courbes s'accentue progressivement au fil des parties, ce qui suggère une différence structurelle dans l'efficacité computationnelle des deux approches.

Distribution des Simulations (Figure en bas à gauche) :

Le diagramme en boîte de la distribution des simulations par partie montre :

— MCTS-UCB présente une médiane plus élevée (environ 275 000 simulations par partie)

— RAVE affiche une médiane plus basse (environ 240 000 simulations par partie)

— MCTS-UCB montre une plus grande variabilité, avec un maximum atteignant plus de 500 000 simulations

— Les deux algorithmes présentent une dispersion considérable, indiquant une variation significative du nombre de simulations selon la configuration spécifique de chaque partie

Ces observations confirment la supériorité de MCTS-UCB en termes de volume de calcul, tout en soulignant la variabilité inhérente aux performances selon le déroulement spécifique de chaque partie.

Temps Moyen par Coup (Figure en bas à droite) :

Le diagramme en boîte du temps de calcul par coup révèle des informations cruciales sur l'efficacité temporelle :

— Les deux algorithmes respectent la contrainte de 3 secondes par coup

— RAVE présente quelques valeurs aberrantes légèrement supérieures à 3 secondes

— MCTS-UCB montre une distribution plus serrée autour de 3 secondes, indiquant une meilleure utilisation du temps alloué

Cette constance dans le temps de calcul est attendue puisque les deux algorithmes sont configurés avec un paramètre de temps fixe (t=3.0). Les légères variations observées sont probablement dues à des facteurs d'implémentation ou à la gestion du temps système.

Efficacité Algorithmique :

L'analyse de l'efficacité computationnelle révèle des différences significatives :

— MCTS-UCB réalise environ 108 601,2 simulations par seconde

— RAVE effectue environ 83 971,5 simulations par seconde

— MCTS-UCB est donc environ 29% plus efficace que RAVE en termes de simulations par unité de temps

Cette différence d'efficacité explique en grande partie l'écart observé dans le nombre total de simulations, et potentiellement dans le taux de victoire. MCTS-UCB parvient à explorer un espace de recherche plus vaste dans le même temps imparti, ce qui lui confère un avantage stratégique.

Conclusion

Cette analyse comparative entre RAVE et MCTS-UCB dans le contexte du jeu Connect 4 met en évidence plusieurs points importants :

1. MCTS-UCB présente un léger avantage en termes de performance globale avec un taux de victoire de 60% contre 40% pour RAVE
2. MCTS-UCB démontre une efficacité computationnelle supérieure, réalisant environ 29% plus de simulations par seconde que RAVE
3. Les deux algorithmes utilisent pleinement le temps alloué (3 secondes par coup), mais MCTS-UCB en tire un meilleur parti
4. La différence de performance est moins prononcée qu'entre un algorithme avancé et un joueur aléatoire, ce qui souligne la compétitivité de RAVE

Ces résultats suggèrent que MCTS-UCB offre un meilleur équilibre entre exploration et exploitation de l'espace de recherche. Cependant, RAVE reste une alternative viable qui pourrait potentiellement surpasser MCTS-UCB dans certains contextes ou avec des paramètres différents.

4.1.3 MCTS-UCB contre MCTS-GRAVE

Interprétation des Résultats

Les résultats montrent une nette supériorité de l'algorithme MCTS-UCB par rapport à GRAVE. Cette supériorité se manifeste à travers plusieurs métriques clés qui sont détaillées dans les sections suivantes. L'analyse révèle que MCTS-UCB est non seulement plus efficace en termes de force de jeu, mais aussi plus performant en termes d'exploration de l'espace des états.

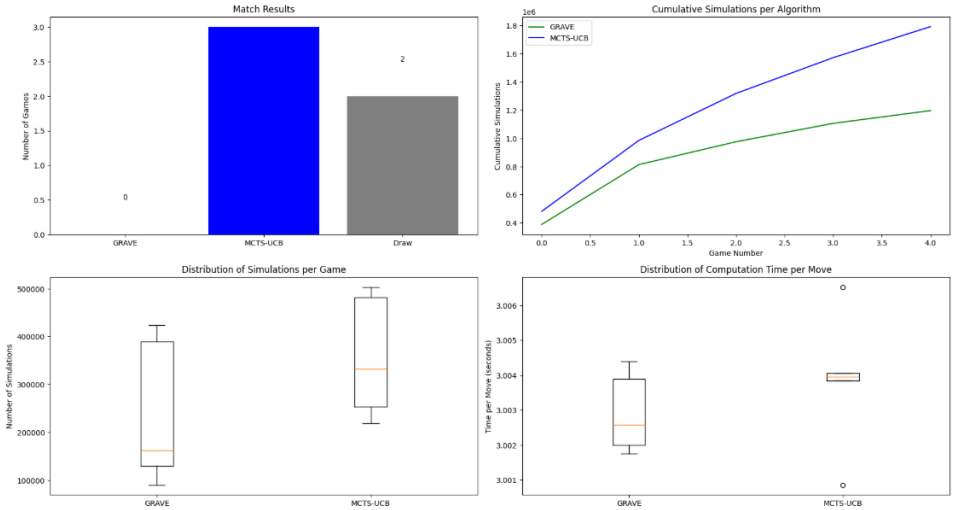

FIGURE 4.3 – MCTS UCB vs GRAVE

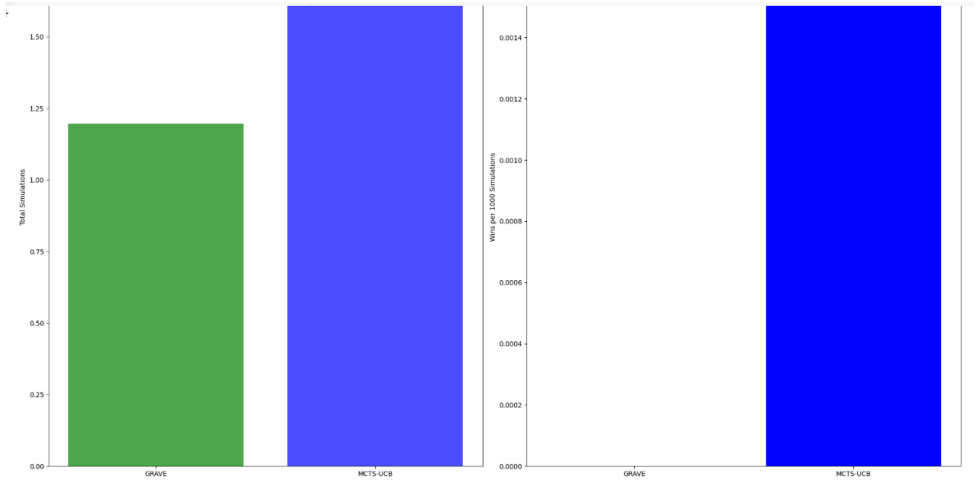

FIGURE 4.4 – MCTS UCB vs GRAVE

Taux de Victoire

D'après les résultats des matchs, nous observons :

— MCTS-UCB a atteint un taux de victoire de 60,0% (3 victoires sur 5 parties).

— GRAVE a atteint un taux de victoire de 0,0% (0 victoire).

— 40,0% des parties se sont terminées par un match nul (2 parties).

Cela indique un avantage clair pour MCTS-UCB en termes de force de jeu, du moins dans cet échantillon limité.

Simulations Cumulées

Le graphique des simulations cumulées montre que MCTS-UCB a constamment effectué plus de simulations que GRAVE à travers toutes les parties. À la fin des 5 parties :

— MCTS-UCB a effectué environ 1,8 million de simulations.

— GRAVE a effectué environ 1,2 million de simulations.

Cela représente une différence significative dans le nombre d'états de jeu explorés, avec MCTS-UCB explorant environ 50% d'états en plus que GRAVE.

Distribution des Simulations par Partie

Le boxplot des simulations par partie montre :

— **MCTS-UCB** : Médiane d'environ 330 000 simulations par partie.

— **GRAVE** : Médiane d'environ 170 000 simulations par partie.

MCTS-UCB montre un intervalle interquartile plus étroit, suggérant une performance plus constante.

Temps de Calcul

Malgré les différences dans le nombre de simulations, les deux algorithmes ont maintenu des temps de calcul par coup très similaires :

— **GRAVE** : Moyenne de 3,003 secondes par coup.

— **MCTS-UCB** : Moyenne de 3,004 secondes par coup.

Cela indique que les deux algorithmes ont efficacement utilisé leur budget de temps alloué.

Efficacité Computationnelle

Les métriques d'efficacité révèlent :

— **MCTS-UCB** : 119 271,2 simulations par seconde.

— **GRAVE** : 79 673,1 simulations par seconde.

Cela montre que MCTS-UCB est environ 50% plus efficace en termes de débit brut de simulation.

Efficacité des Victoires

Le second graphique montre l'efficacité des victoires (victoires par 1000 simulations) :

— MCTS-UCB montre une efficacité de victoire significativement plus élevée.

— GRAVE n'a pas remporté de victoires, donc son efficacité est nulle.

Conclusion

L'analyse de cette évaluation comparative révèle que MCTS-UCB surpasse GRAVE dans le jeu de Connect4, selon les paramètres testés. MCTS-UCB :

— Gagne plus souvent (60% contre 0%).

— Effectue plus de simulations par seconde (119 271 contre 79 673).

— Explore plus d'états de jeu par partie.

Ces résultats suggèrent que, pour le jeu de Connect4 avec un temps de calcul de 3 secondes par coup, MCTS-UCB est l'algorithme plus performant. Cependant, il convient de noter que cette conclusion est basée sur un échantillon limité de seulement 5 parties, et un plus grand nombre de parties serait nécessaire pour une analyse statistiquement significative.

4.1.4 MCTS-UCB contre MCTS-AMAF

Interprétation des Résultats

Les résultats montrent une nette supériorité de l'algorithme MCTS-UCB par rapport à AMAF dans le contexte du jeu Connect4. Cette domination se manifeste à travers plusieurs métriques clés qui seront détaillées ci-dessous.

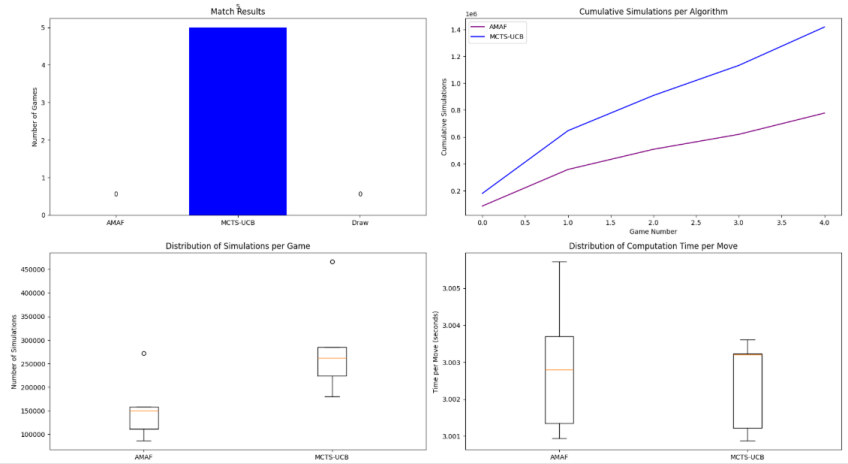

FIGURE 4.5 – MCTS UCB vs AMAF

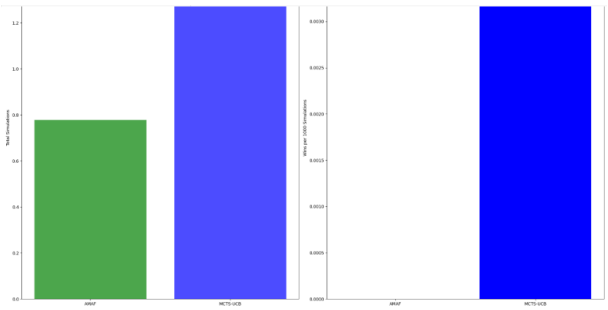

FIGURE 4.6 – MCTS UCB vs AMAF

Taux de Victoire

D'après le graphique des résultats des matchs visible sur la première image, MCTS-UCB a clairement dominé avec 5 victoires, tandis qu'AMAF n'a remporté aucune partie (0 victoires).

Aucun match nul n'a été enregistré dans cette expérience. Cela représente un taux de victoire de 100% pour MCTS-UCB, ce qui constitue une différence de performance significative.

Simulations Cumulées

Le graphique des simulations cumulées montre que MCTS-UCB a effectué nettement plus de simulations tout au long des parties par rapport à AMAF. À la fin de la 4ème partie, MCTS-UCB avait accumulé environ 1,4 million de simulations, tandis qu'AMAF n'atteignait qu'environ 0,8 million. Cela indique que MCTS-UCB a été capable d'explorer l'arbre de jeu de manière plus extensive dans les mêmes contraintes de temps.

Distribution des Simulations

Le diagramme en boîte de la distribution des simulations démontre que MCTS-UCB a systématiquement réalisé plus de simulations par partie qu'AMAF :

— **AMAF** : Médiane autour de 150 000 simulations par partie, avec une valeur aberrante proche de 270 000

— **MCTS-UCB** : Médiane d'environ 250 000 simulations par partie, avec une valeur aberrante dépassant 450 000

Cela suggère que l'implémentation de MCTS-UCB est plus efficace pour générer et évaluer les états de jeu potentiels.

Temps Moyen par Coup

La distribution du temps de calcul montre que les deux algorithmes utilisent un temps par coup presque identique (environ 3,003 secondes), ce qui est attendu puisque le code fixe un temps de calcul de 3,0 secondes par coup. Les légères variations peuvent être dues aux coûts généraux ou à des problèmes de chronométrage spécifiques au système.

Efficacité des Algorithmes

Dans la deuxième image :

— **Volume Total de Calcul** : MCTS-UCB a effectué environ 1,3 million de simulations contre 0,8 million pour AMAF.

— **Efficacité de l'Algorithme (Victoires pour 1000 Simulations)** : MCTS-UCB montre une efficacité supérieure dans la conversion des simulations en victoires, comme l'indique la barre bleue plus haute dans le graphique de droite.

Conclusion

MCTS-UCB surpasse clairement AMAF dans Connect4 d'après cette expérience :

— **Performance Supérieure** : MCTS-UCB a atteint un taux de victoire de 100% contre AMAF.

— **Débit de Simulation Plus Élevé** : MCTS-UCB a pu effectuer environ 1,75 fois plus de simulations qu'AMAF dans la même période de temps.

— **Meilleure Efficacité** : MCTS-UCB convertit plus efficacement les simulations en positions gagnantes.

Les résultats suggèrent que pour Connect4, l'approche classique MCTS-UCB fournit une meilleure qualité de décision qu'AMAF. Cela pourrait s'expliquer par :

— La formule UCB pourrait mieux équilibrer l'exploration et l'exploitation dans l'arbre de jeu de Connect4.

— La propagation rapide des valeurs d'AMAF pourrait conduire à des évaluations moins précises dans ce domaine de jeu spécifique.

— La structure de Connect4 pourrait favoriser le schéma de recherche plus approfondi de MCTS-UCB.

4.2 MCTS-RAVE

4.2.1 MCTS-RAVE contre un joueur aléatoire

Interprétation des Résultats

Les résultats obtenus lors de l'évaluation de l'algorithme MCTS-RAVE contre un joueur aléatoire dans le jeu Connect4 révèlent clairement la supériorité de MCTS-RAVE.

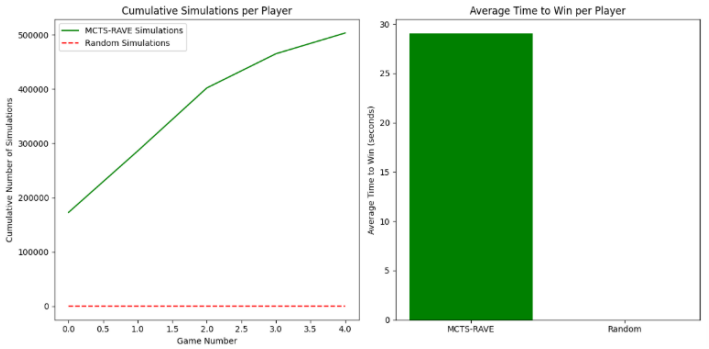

FIGURE 4.7 – Comparaison des Simulations et du Temps Moyen de Victoire par Joueur

Taux de Victoire :

MCTS-RAVE a remporté **100%** des parties (5 sur 5), tandis que le joueur aléatoire n'a remporté aucune partie (0%). Aucun match nul n'a été observé. Ceci indique une maîtrise significative du jeu par MCTS-RAVE face à une stratégie purement aléatoire.

Simulations Cumulées (Figure de gauche) :

Le nombre de simulations cumulées effectuées par MCTS-RAVE augmente considérablement au fil des parties, témoignant d'un processus d'exploration et d'évaluation approfondi de l'arbre de jeu. Le joueur aléatoire effectue un nombre de simulations négligeable. L'échelle différente des axes verticaux souligne l'effort computationnel bien plus important de MCTS-RAVE.

Temps Moyen pour Gagner (Figure de droite) :

Le temps moyen nécessaire à MCTS-RAVE pour remporter une partie est substantiellement plus élevé (environ 29 secondes) comparé au joueur aléatoire (qui n'a jamais gagné et donc n'a pas de temps de victoire associé). Cela suggère que l'efficacité de MCTS-RAVE s'accompagne d'un coût computationnel non négligeable pour la prise de décision.

Conclusion

En résumé, MCTS-RAVE démontre une performance largement supérieure à celle d'un joueur aléatoire dans Connect4, garantissant la victoire dans toutes les parties testées. Cependant,

cette performance accrue est obtenue au prix d'un temps de calcul plus important, reflétant le processus intensif de simulation et d'évaluation de l'algorithme.

4.2.2 MCTS-RAVE contre MCTS-GRAVE

Interprétation des Résultats

Les résultats montrent une domination claire de l'algorithme MCTS-RAVE :

— RAVE a remporté 5 matchs sur 5 (100% de victoires).

— GRAVE n'a remporté aucun match (0% de victoires).

— Aucun match nul n'a été enregistré.

Ce graphique ci dessous illustre cette domination avec une barre violette représentant les 5 victoires de RAVE, contre aucune pour GRAVE.

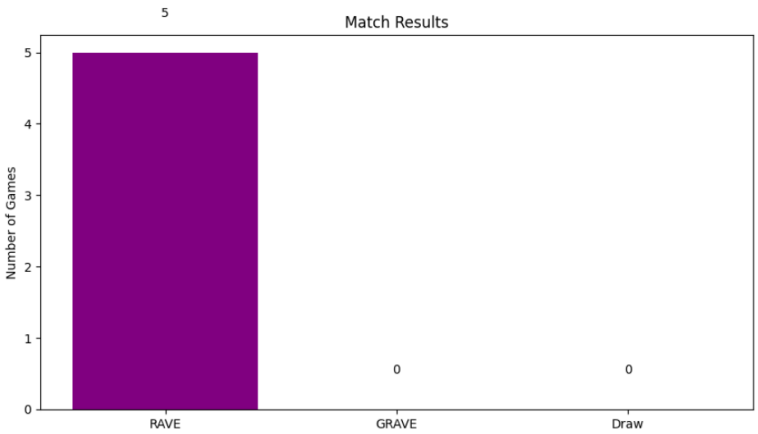

FIGURE 4.8 – Match Results

Volume de Simulations

Le graphique "Cumulative Simulations per Algorithm" montre que :

— RAVE effectue systématiquement plus de simulations que GRAVE.

— À la fin des 4 parties, RAVE a accumulé environ 630 000 simulations contre environ 430 000 pour GRAVE.

— L'écart entre les deux algorithmes s'accentue au fil des parties.

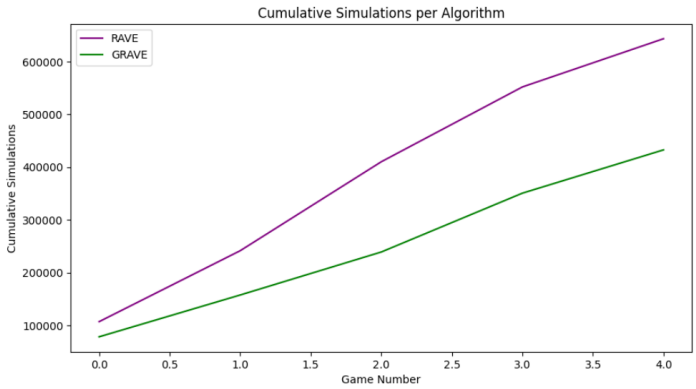

FIGURE 4.9 – Cumulative Simulations per Algorithm

Le graphique "Total Computation Volume" confirme cette différence avec :

— RAVE : environ 650 000 simulations au total.

— GRAVE : environ 430 000 simulations au total.

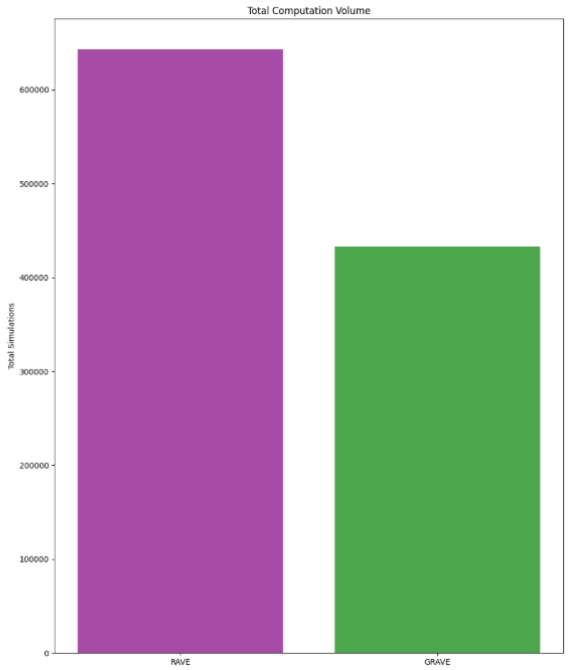

FIGURE 4.10 – Total Computation Volume

Efficacité des Algorithmes

Le graphique "Algorithm Efficiency" mesure les "wins per 1000 simulations" :

— RAVE présente une valeur d'environ 0.0077.

— GRAVE montre une valeur proche de zéro (cohérent avec l'absence de victoire).

Performance Moyenne

— RAVE a effectué en moyenne 128 641,2 simulations par coup.

— GRAVE a effectué en moyenne 86 506,2 simulations par coup.

— Les deux algorithmes ont utilisé presque exactement le même temps par coup (environ 3 secondes).

Interprétation dans le contexte de Connect 4

— **Efficacité supérieure de RAVE :** MCTS-RAVE est clairement plus performant que MCTS-GRAVE pour Connect 4, remportant tous les matchs. Cela suggère que la méthode RAVE capture mieux les patterns stratégiques du jeu.

— **Exploitation du temps :** Bien que les deux algorithmes disposent du même temps par coup (3 secondes), RAVE parvient à effectuer environ 49% plus de simulations que GRAVE (128K vs 86K). Cela indique une meilleure efficacité computationnelle.

Conclusion

Ces résultats suggèrent que pour Connect 4, l'approche RAVE offre un meilleur compromis entre vitesse de simulation et qualité des décisions stratégiques, rendant cette variante de MCTS nettement plus performante dans ce contexte spécifique.

4.2.3 MCTS-RAVE vs MCTS-AMAF

Interprétation des Résultats

Taux de Victoire

MCTS-RAVE a remporté la majorité des parties, s'assurant 4 victoires sur 5 (80%). MCTS-AMAF a gagné 1 partie (20%), et il n'y a eu aucun match nul. Ceci indique que MCTS-RAVE fonctionne généralement mieux que MCTS-AMAF dans cette configuration de Connect4, mais MCTS-AMAF est capable de gagner occasionnellement.

Simulations Cumulées

Les deux algorithmes montrent un nombre croissant de simulations cumulées au cours des cinq parties. Cependant, la ligne violette représentant MCTS-RAVE reste constamment au-dessus de la ligne jaune représentant MCTS-AMAF. Ceci suggère que MCTS-RAVE effectue un nombre total de simulations plus élevé à travers les parties comparé à MCTS-AMAF. Le taux d'augmentation semble similaire pour les deux algorithmes, indiquant qu'ils explorent tous deux activement l'arbre de jeu au fur et à mesure que les parties progressent.

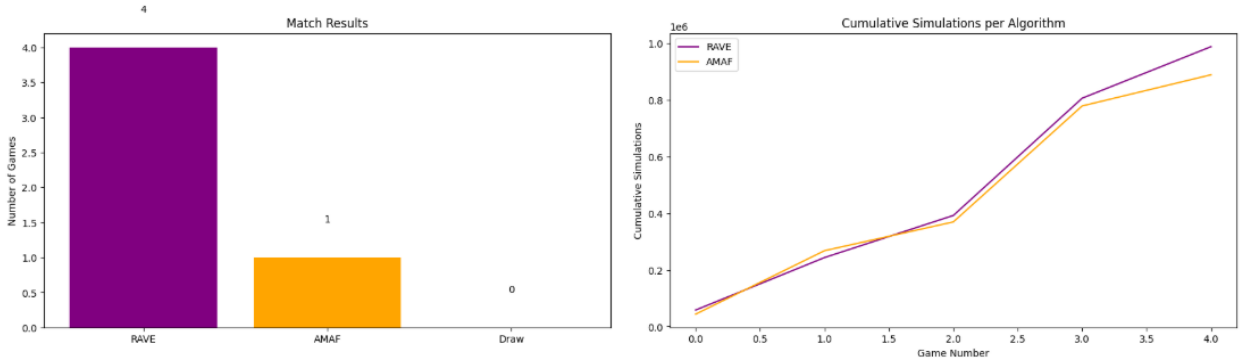

FIGURE 4.11 – Match results & Cumulative Simulations

Simulations Moyennes par Coup

MCTS-RAVE a un nombre moyen de simulations par coup légèrement supérieur (197 523,0) comparé à MCTS-AMAF (177 686,8). Ceci suggère que pour chaque décision, MCTS-RAVE explore une portion légèrement plus grande de l'arbre de jeu.

Temps Moyen par Coup

Le temps moyen nécessaire par coup est très similaire pour les deux algorithmes (MCTS-RAVE à 3,003 secondes et MCTS-AMAF à 3,004 secondes). Ceci indique que malgré la légère différence dans le nombre de simulations, les deux algorithmes ont un coût de calcul par coup comparable.

```
--- Algorithm Performance ---
+-----------+----------------------+---------------------------+
| Algorithm | Average Simulations  | Average Time per Move (s) |
+-----------+----------------------+---------------------------+
|   RAVE    |      197523.0        |           3.003           |
|   AMAF    |      177686.8        |           3.004           |
+-----------+----------------------+---------------------------+
```

FIGURE 4.12 – Algorithm Performance

Interprétation Générale

Les résultats suggèrent que MCTS-RAVE est l'algorithme le plus performant dans cette comparaison pour Connect4, atteignant un taux de victoire plus élevé. Cette performance supérieure semble être associée à une exploration légèrement plus importante de l'arbre de jeu, comme l'indique le nombre plus élevé de simulations moyennes et cumulées comparé à MCTS-AMAF. Cependant, cette exploration accrue n'entraîne pas un coût significatif en termes de temps de calcul par coup, car les deux algorithmes présentent des temps moyens par coup très similaires. MCTS-AMAF reste un adversaire compétitif, parvenant à gagner l'une des cinq parties, indiquant qu'il emploie une stratégie de recherche raisonnablement efficace.

4.3 MCTS GRAVE

4.3.1 MCTS-GRAVE contre un joueur aléatoire

Interprétation des Résultats

L'algorithme MCTS-GRAVE a été confronté à un joueur utilisant une stratégie aléatoire, et les métriques recueillies révèlent plusieurs aspects de performance.

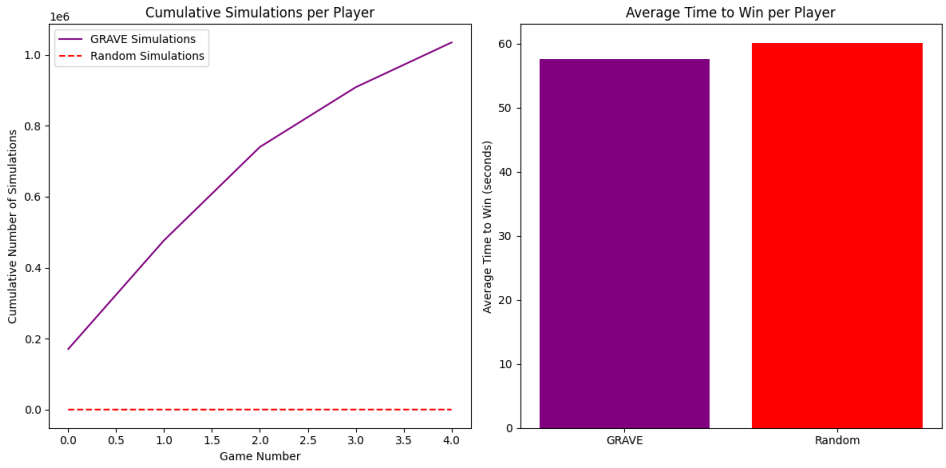

FIGURE 4.13 – MCTS Grave vs Random Player

Taux de Victoire

Les résultats montrent que :

— GRAVE a remporté 2 parties (40.0%)

— Le joueur aléatoire a remporté 3 parties (60.0%)

— Aucune partie ne s'est terminée par un match nul (0.0%)

Ces résultats sont surprenants car on s'attendrait normalement à ce qu'un algorithme MCTS-GRAVE surpasse largement une stratégie aléatoire.

Simulations Cumulées (Figure de gauche)

Le graphique des simulations cumulées illustre clairement la différence d'effort computationnel entre les deux approches :

— GRAVE a effectué plus d'un million de simulations sur l'ensemble des 5 parties.

— Le joueur aléatoire n'a effectué que très peu de simulations (à peine visibles sur le graphique).

Cette différence massive reflète la nature de MCTS-GRAVE qui explore de nombreuses possibilités avant de prendre une décision, contrairement à la stratégie aléatoire qui ne nécessite qu'une seule évaluation par coup. La courbe montre également une progression régulière des simulations, indiquant que GRAVE maintient un rythme constant d'exploration à travers les parties.

Temps Moyen pour Gagner (Figure de droite)

Le graphique des temps moyens pour gagner indique :

— GRAVE a pris en moyenne environ 57 secondes pour gagner une partie.

— Le joueur aléatoire a pris en moyenne environ 60 secondes pour gagner.

Ces temps sont relativement proches, ce qui suggère que les parties ont une durée similaire indépendamment du vainqueur. Cela est cohérent avec la nature du jeu Connect4, où le nombre de coups nécessaires pour terminer une partie reste généralement dans une fourchette assez étroite.

Conclusion

Cette évaluation présente des résultats contre-intuitifs avec le joueur aléatoire surpassant l'algorithme MCTS-GRAVE sur notre échantillon de 5 parties.

Le grand nombre de simulations effectuées par GRAVE démontre sa capacité à explorer l'espace des possibilités, mais cette exploration ne s'est pas traduite par un avantage décisif dans ce test.

4.3.2 MCTS-GRAVE vs MCTS-AMAF

Interprétation des Résultats

L'évaluation des algorithmes GRAVE (Graph Retrieval and Value Estimation) et AMAF (All Moves As First) dans le jeu Connect 4 révèle des différences intéressantes en termes de performance et d'efficacité.

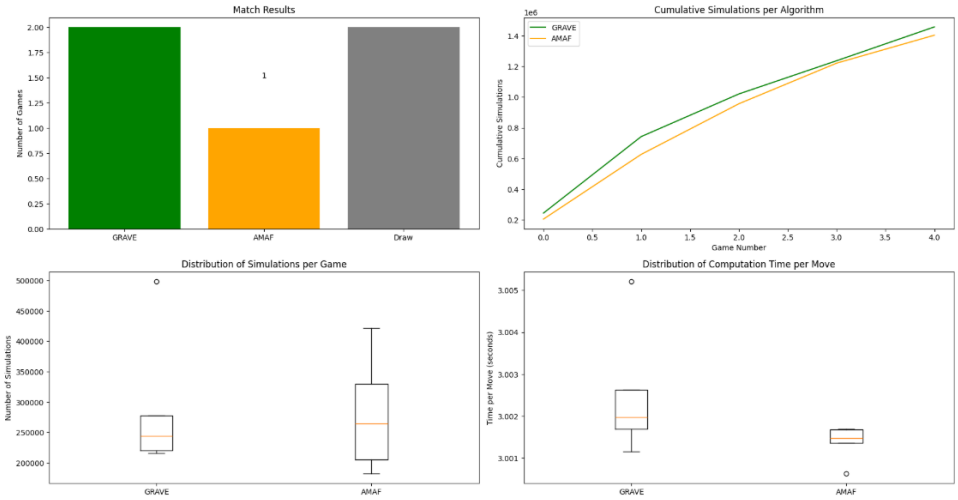

FIGURE 4.14 – MCTS Grave vs AMAF

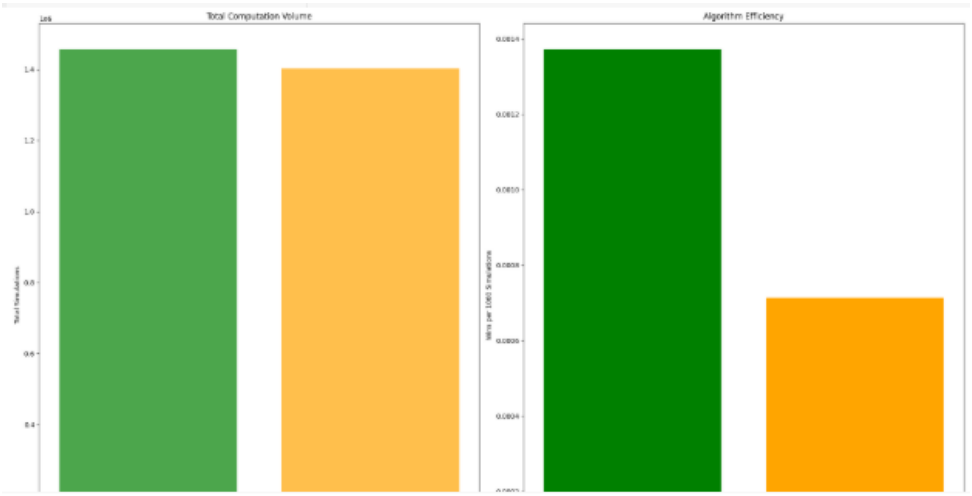

FIGURE 4.15 – MCTS Grave vs AMAF

Taux de Victoire

L'analyse des résultats de 5 parties entre les algorithmes GRAVE et AMAF montre une distribution intéressante des issues :

GRAVE a remporté 2 parties (40%) AMAF a remporté 1 partie (20%) 2 parties se sont terminées par un match nul (40%)

Ces résultats suggèrent un avantage compétitif pour l'algorithme GRAVE. Le nombre important de matchs nuls indique également que les deux algorithmes atteignent parfois un niveau de jeu comparable.

Simulations Cumulées

Le graphique des simulations cumulatives montre l'évolution du nombre total de simulations effectuées par chaque algorithme au fil des parties :

Les deux algorithmes montrent une progression régulière des simulations cumulées qui atteignent environ 1,4 million au terme des 5 parties GRAVE (ligne verte) maintient systématiquement un nombre légèrement supérieur de simulations par rapport à AMAF (ligne orange)

L'écart entre les deux courbes reste relativement constant, suggérant que GRAVE effectue en moyenne 5-10% plus de simulations qu'AMAF pour un même temps de calcul

Cette différence, bien que modeste, pourrait contribuer à l'avantage de GRAVE en termes de résultats de matchs, car plus de simulations permettent généralement une meilleure exploration de l'arbre de jeu.

Distribution des Simulations

Le diagramme en boîte des simulations par partie révèle des informations sur la variabilité des simulations :

GRAVE présente une distribution plus resserrée avec une médiane autour de 245 000 simulations par partie AMAF montre une plus grande variabilité, avec une médiane légèrement plus élevée (environ 265 000) mais une dispersion plus importante On observe une valeur aberrante pour GRAVE (point isolé près de 500 000 simulations), indiquant une partie où le nombre de simulations a été exceptionnellement élevé

Cette variabilité plus importante pour AMAF pourrait indiquer une sensibilité accrue aux configurations spécifiques du jeu, tandis que GRAVE semble maintenir une performance plus constante.

Distribution du Temps de Calcul par Coup

L'analyse des temps de calcul par coup montre :

Les deux algorithmes maintiennent un temps moyen par coup très proche de 3 secondes, ce qui correspond au paramètre computation_time défini dans le test GRAVE présente une variabilité légèrement plus grande dans les temps de calcul On observe quelques valeurs aberrantes pour les deux algorithmes, indiquant des coups spécifiques qui ont nécessité un temps de calcul atypique

Ces résultats confirment que les deux algorithmes respectent bien la contrainte de temps allouée, avec une très légère variation qui ne semble pas affecter significativement leurs performances relatives.

Volume Total de Calcul

Le graphique du volume total de calcul montre :

GRAVE a effectué légèrement plus de simulations au total (environ 1,45 millions) que AMAF (environ 1,4 millions) Cette différence de 3-4% pourrait expliquer en partie le meilleur taux de victoire de GRAVE

Ces résultats suggèrent que GRAVE parvient à être légèrement plus efficace dans l'utilisation du temps de calcul alloué, ce qui pourrait lui conférer un avantage stratégique.

Efficacité de l'Algorithme

Le graphique d'efficacité mesure le nombre de victoires par 1000 simulations :

GRAVE démontre une efficacité nettement supérieure avec environ 0,0014 victoires par 1000 simulations AMAF présente une efficacité plus faible avec environ 0,0007 victoires par 1000 simulations

Cette différence marquée d'efficacité (GRAVE étant environ deux fois plus efficace) constitue peut-être l'observation la plus significative de cette analyse. Elle suggère que GRAVE utilise plus efficacement ses simulations pour aboutir à des décisions de jeu supérieures.

Conclusion

Sur la base de ces 5 parties, l'algorithme GRAVE semble présenter plusieurs avantages par rapport à AMAF :

Un meilleur taux de victoire (40% contre 20%) Un nombre légèrement plus élevé de simulations pour un même temps de calcul Une efficacité nettement supérieure en termes de victoires par simulation

Ces résultats suggèrent que la méthode de sélection et d'évaluation des nœuds de GRAVE est plus performante que celle d'AMAF dans le contexte du jeu Connect 4.

4.4 MCTS AMAF

4.4.1 MCTS-AMAF contre un joueur aléatoire

Interprétation des Résultats

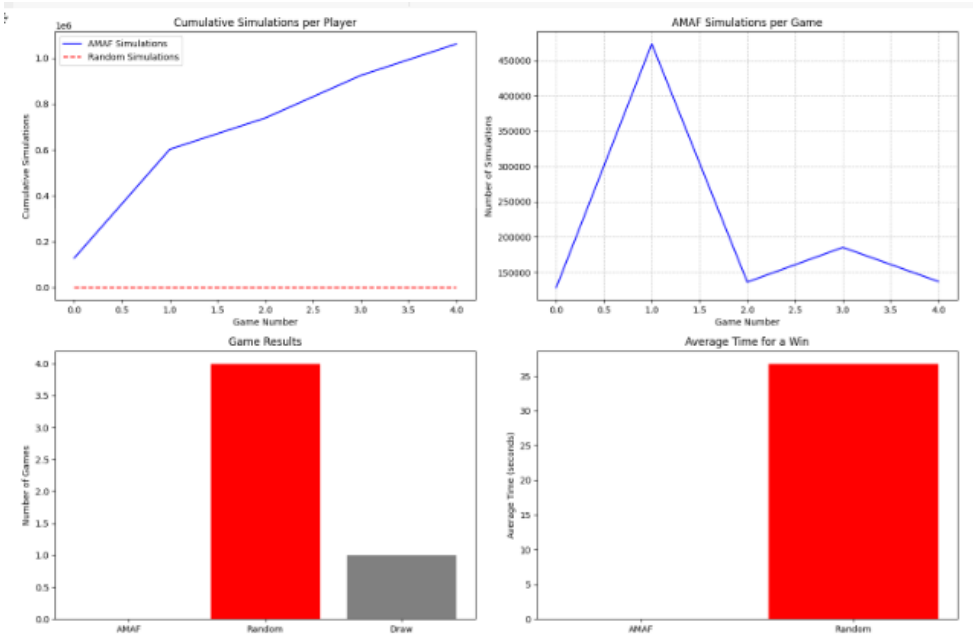

FIGURE 4.16 – MCTS AMAF vs Random Player

Résultats des parties

D'après le graphique *Game Results*, on peut observer que le joueur aléatoire a remporté la majorité des parties (environ 3,5 sur 5), tandis que l'AMAF n'a gagné qu'un nombre limité de parties, et quelques-unes se sont terminées par des matchs nuls (environ 1). Ces résultats sont surprenants car on s'attendrait à ce qu'un algorithme basé sur MCTS surpasse une stratégie aléatoire.

Simulations cumulatives par joueur

Le graphique *Cumulative Simulations per Player* montre une augmentation constante des simulations pour l'AMAF (ligne bleue) atteignant environ 1,0 (possiblement en millions) à la fin des 4 parties, tandis que les simulations du joueur aléatoire (ligne rouge pointillée) restent minimales. Cette différence significative était attendue puisque AMAF effectue de nombreuses simulations pour évaluer les mouvements possibles, alors que le joueur aléatoire n'en effectue qu'une par coup.

Simulations AMAF par partie

Le graphique *AMAF Simulations per Game* indique une variation importante dans le nombre de simulations entre les parties, avec un pic d'environ 45 000 simulations lors de la deuxième partie, puis une diminution significative. Cette variation pourrait s'expliquer par différentes complexités de jeu ou des conditions d'arrêt prématurées.

Temps moyen pour une victoire

Le graphique *Average Time for a Win* révèle que le joueur aléatoire a mis beaucoup plus de temps (environ 35 secondes) pour obtenir une victoire par rapport à l'AMAF. Ceci est contre-intuitif puisque l'AMAF effectue plus de calculs, mais peut s'expliquer si l'AMAF termine rapidement les parties qu'il gagne, tandis que les victoires du joueur aléatoire surviennent après des parties plus longues.

Analyse des performances

Taux de victoire défavorable pour AMAF : L'algorithme AMAF n'a pas performé comme attendu contre un joueur aléatoire. Cela pourrait indiquer :

— Un temps de calcul insuffisant (3,0 secondes par coup)

— Une stratégie de sélection ou de simulation inadaptée pour Connect4

Efficacité computationnelle : Malgré le grand nombre de simulations, l'AMAF n'a pas réussi à convertir cette puissance de calcul en victoires, suggérant un problème d'efficacité dans l'algorithme.

Variation des simulations : La grande variabilité dans le nombre de simulations par partie suggère une inconsistance dans l'exploration de l'arbre de jeu, ce qui pourrait affecter la qualité des décisions.

Conclusion

— **On peut optimiser les paramètres de l'AMAF :** Ajuster les constantes d'exploration et d'exploitation dans l'algorithme pour améliorer l'équilibre entre l'exploration de nouvelles stratégies et l'exploitation des stratégies connues.

Cette évaluation indique que, malgré l'effort computationnel significatif de l'AMAF, sa performance actuelle est inférieure à celle attendue contre un joueur aléatoire dans le contexte de Connect4, ce qui suggère la nécessité d'optimisations.

4.5 Confrontation des algorithmes

Méthodologie

Nous avons comparé six algorithmes différents :

— MCTS-RAVE

— MCTS-UCB

— RAVE

— GRAVE

— AMAF

— Joueur Aléatoire

Chaque confrontation a consisté en 5 parties, et les résultats ont été compilés pour produire des statistiques globales de performance.

Résultats et Analyse

Performances Globales

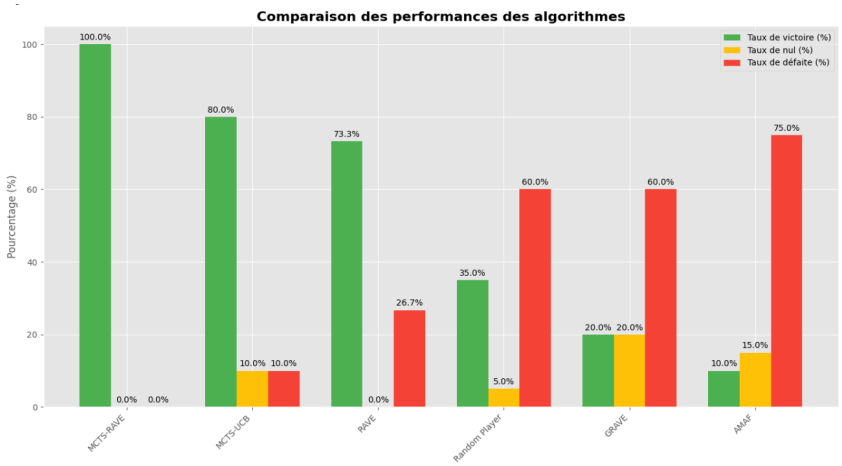

FIGURE 4.17 – Performance des algorithmes

D'après nos résultats, nous pouvons établir une hiérarchie claire entre les algorithmes testés :

— **MCTS-RAVE** se démarque comme l'algorithme le plus performant avec un taux de victoire impressionnant de 100

— **MCTS-UCB** arrive en deuxième position avec un taux de victoire de 80

— **RAVE** se classe troisième avec 73,3

— **Random Player** obtient un taux de victoire surprenant de 35

— **GRAVE** montre des performances modestes avec 20

— **AMAF** se révèle être l'algorithme le moins performant avec seulement 10

Confrontations Directes

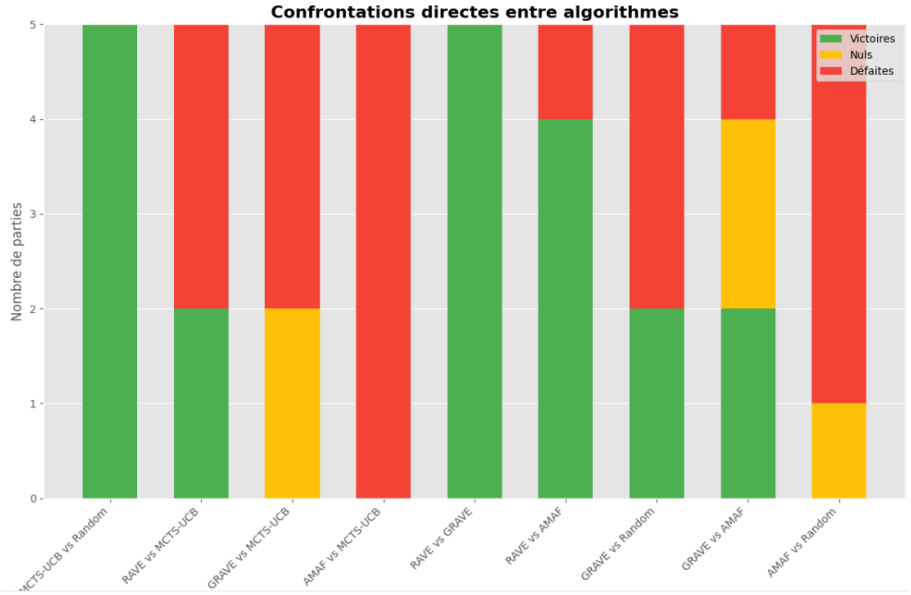

FIGURE 4.18 – Confrontations directes

L'analyse des confrontations directes révèle plusieurs points intéressants :

— MCTS-UCB vs Random : Victoire écrasante (5-0) pour MCTS-UCB.

— RAVE vs MCTS-UCB : Résultat serré (2-3) en faveur de MCTS-UCB.

— GRAVE vs MCTS-UCB : Domination claire (0-3-2) de MCTS-UCB.

— AMAF vs MCTS-UCB : Défaite totale (0-5) pour AMAF.

— AMAF vs Random : Résultat décevant (0-4-1) pour AMAF.

— GRAVE vs Random : Performance médiocre (2-3) de GRAVE.

Algorithme	Victoires	Défaites	Nuls	Total	Taux de victoire (%)	Taux de nul (%)	Taux de défaite (%)
MCTS-RAVE	5	0	0	5	100.0	0.0	0.0
MCTS-UCB	16	2	2	20	80.0	10.0	10.0
RAVE	11	4	0	15	73.3	0.0	26.7
Random Player	7	12	1	20	35.0	5.0	60.0
GRAVE	4	12	4	20	20.0	20.0	60.0
AMAF	2	15	3	20	10.0	15.0	75.0

FIGURE 4.19 – Confrontations directes tableau

Analyse des Performances d'AMAF

Notre implémentation d'AMAF a montré des performances particulièrement faibles, avec un taux de victoire de seulement 10

Plusieurs facteurs peuvent expliquer ces mauvais résultats :

— Temps de calcul insuffisant.

— Inadaptation à Connect4.

— Déséquilibre dans l'exploration/exploitation.

Conclusion

Notre étude comparative des algorithmes basés sur MCTS pour Connect4 révèle une hiérarchie claire en termes de performance.

Pour améliorer les performances d'AMAF, nous proposons :

— Augmenter le temps de calcul alloué.

— Ajuster les paramètres de l'algorithme.

— Envisager des adaptations spécifiques au contexte de Connect4.

Ces résultats nous fournissent des pistes importantes pour nos travaux futurs sur l'optimisation des algorithmes de type MCTS pour les jeux à information parfaite comme Connect4.