Application des méthodes MonteCarlo sur le jeu de Puissance 4

Jihed BHAR, Yassine BEN ABDALLAH, Nadia BEN YOUSSEF

E-mails : jihed.bhar@dauphine.tn, yassine.benabdallah@dauphine.tn, nadia.benyoussef@dauphine.tn

Professor : Mr Tristan CAZENAVE

Dauphine | PSL TUNIS

Abstract

Ce rapport présente une étude comparative de quatre algorithmes d'apprentissage par renforcement basés sur les méthodes de Monte Carlo : l'algorithme standard Monte Carlo Tree Search (MCTS), sa variante améliorée UCB1-Tuned, l'algorithme Rapid Action Value Estimation (RAVE) et l'algorithme Nested Rollout Policy Adaptation (NRPA). Ces algorithmes sont implémentés et évalués dans le contexte du jeu de Puissance 4, un environnement de décision séquentiel parfaitement adapté aux méthodes d'exploration/exploitation. Cette étude met en évidence l'importance de l'ajustement de l'équilibre exploration/exploitation dans les algorithmes d'apprentissage par renforcement et démontre que des méthodes efficaces dans certains jeux peuvent s'avérer sous-optimales dans d'autres.

1. Introduction

Les méthodes de Monte Carlo ont révolutionné l'approche des problèmes complexes de prise de décision, particulièrement dans le domaine des jeux. Parmi ces méthodes, Monte Carlo Tree Search (MCTS) s'est imposé comme un algorithme puissant, capable de traiter des espaces d'états vastes sans nécessiter de connaissances expertes préalables. Son succès a été démontré dans divers jeux, notamment les échecs, le Go et d'autres jeux de plateau.

Le jeu de Puissance 4 constitue un cadre idéal pour cette étude comparative pour plusieurs raisons :

- Son espace d'états est suffisamment complexe pour mettre à l'épreuve les algorithmes
- Les règles sont simples et déterministes
- Un avantage théorique existe pour le premier joueur, ce qui permet d'évaluer si les algorithmes peuvent l'exploiter
- La durée limitée des parties permet de réaliser un grand nombre de simulations

Notre implémentation repose sur trois composantes principales : la modélisation du jeu de Puissance 4, la structure des nœuds pour l'arbre de recherche, et les quatres algorithmes : MCTS standard, UCB1-Tuned, RAVE et NRPA. L'objectif principal est d'évaluer les performances relatives à ces méthodes dans ce contexte, en mesurant leur taux de victoire, leur capacité à exploiter l'avantage du premier joueur, et leur robustesse face à différentes situations de jeu. Cette comparaison permet non seulement de déterminer quelle approche est la plus efficace pour Puissance 4, mais aussi d'approfondir notre compréhension des mécanismes d'ajustement de l'équilibre exploration/exploitation dans les algorithmes d'apprentissage par renforcement.

Dans ce rapport, nous présenterons d'abord les fondements théoriques des algorithmes étudiés, puis détaillerons notre implémentation et les expérimentations réalisées. Nous analyserons ensuite les résultats obtenus et discuterons des limitations et perspectives d'amélioration.

2. Fondements théoriques

2.1. Le jeu de Puissance 4

Le Puissance 4 est un jeu de stratégie combinatoire à deux joueurs, où chacun tente d'aligner quatre pions de sa couleur horizontalement, verticalement ou en diagonale sur une grille verticale de 6 lignes et 7 colonnes. À tour de rôle, les joueurs placent un pion dans l'une des colonnes, qui tombe alors jusqu'à la position la plus basse disponible. Le jeu se termine lorsqu'un joueur réussit à aligner quatre pions ou lorsque la grille est complètement remplie (match nul).

Du point de vue théorique, le Puissance 4 présente plusieurs caractéristiques intéressantes :

- C'est un jeu à **information parfaite** : les deux joueurs ont accès à toute l'information sur l'état du jeu.
- Il est **déterministe** : aucun élément aléatoire n'influence le déroulement du jeu.
- Son **espace d'états** est important mais fini (environ $4,5 \times 10^{12}$ positions possibles).

— Il a été **mathématiquement résolu** : avec un jeu parfait, le premier joueur peut toujours forcer une victoire en commençant au centre.

Ces propriétés en font un candidat idéal pour tester et comparer des algorithmes de recherche arborescente comme MCTS et ses variantes.

2.2. Monte Carlo Tree Search (MCTS)

MCTS est un algorithme de recherche heuristique qui construit progressivement un arbre de recherche asymétrique en fonction des résultats de simulations. Il se distingue des algorithmes traditionnels comme Minimax par sa capacité à fournir des décisions de qualité même sans connaissance experte du domaine, en s'appuyant uniquement sur des simulations aléatoires.

Le processus de MCTS se déroule en quatre phases principales :

1. **Sélection** : Descente dans l'arbre depuis la racine jusqu'à une feuille en utilisant une stratégie d'équilibrage exploration/exploitation, généralement basée sur la formule UCB1.
2. **Expansion** : Si la feuille atteinte n'est pas un état terminal, ajout d'un ou plusieurs nœuds enfants.
3. **Simulation** : Exécution d'une partie à partir du nouvel état, généralement selon une politique aléatoire, jusqu'à atteindre un état terminal.
4. **Rétropropagation** : Mise à jour des statistiques (visites et victoires) pour tous les nœuds traversés, de la feuille à la racine.

La formule UCB1 standard utilisée dans la phase de sélection est :

$$UCB1(s,a) = \frac{Q(s,a)}{N(s,a)} + C \times \sqrt{\frac{2\ln N(s)}{N(s,a)}} \tag{1}$$

où $Q(s,a)$ est le total des récompenses obtenues après avoir joué l'action a dans l'état s, $N(s,a)$ est le nombre de fois que cette action a été jouée, $N(s)$ est le nombre total de visites de l'état s, et C est une constante d'exploration (généralement fixée à $\sqrt{2}$).

2.3. UCB1-Tuned

UCB1-Tuned est une amélioration de l'algorithme UCB1 qui prend en compte la variance des récompenses pour ajuster dynamiquement l'équilibre entre exploration et exploitation. Il remplace la partie exploration de la formule UCB1 par une expression qui tient compte de la variance empirique des récompenses :

$$UCB1\text{-}Tuned(s,a) = \frac{Q(s,a)}{N(s,a)} + Bonus_{exp}(s,a) \tag{2}$$

$$Bonus_{exp}(s,a) = \sqrt{\frac{\ln N(s)}{N(s,a)} \min\left(0.25, V(s,a) + \sqrt{\frac{2\ln N(s)}{N(s,a)}}\right)} \tag{3}$$

où $V(s,a)$ est la variance empirique des récompenses pour l'action a dans l'état s, calculée comme :

$$V(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{N(s,a)} X_i^2 - \left(\frac{Q(s,a)}{N(s,a)}\right)^2 \tag{4}$$

Cette formule ajuste l'incitation à l'exploration en fonction de la variance observée : les actions avec une variance élevée sont explorées plus prudemment, tandis que celles avec une variance faible peuvent être explorées plus agressivement. Cela peut être particulièrement avantageux dans des contextes comme le Puissance 4, où certaines positions peuvent avoir des valeurs très stables (clairement gagnantes ou perdantes), tandis que d'autres sont plus incertaines.

2.3.1. RAVE (Rapid Action Value Estimation)

Une autre variante notable de MCTS est RAVE (Rapid Action Value Estimation), qui vise à accélérer la convergence de l'algorithme en combinant les statistiques traditionnelles de MCTS avec une heuristique appelée AMAF (All Moves As First). Dans RAVE, la valeur d'une action est estimée non seulement à partir des simulations où elle est jouée dans une branche spécifique, mais aussi à partir de toutes les simulations où elle apparaît, quel que soit l'ordre des coups. Cette approche est particulièrement efficace dans des jeux comme le Go, où les coups ont des corrélations globales fortes.

La formule de sélection dans RAVE combine le score MCTS standard et le score AMAF via un coefficient β, qui dépend d'une constante k et du nombre de visites :

$$\text{Score}_{\text{RAVE}}(s,a) = (1-\beta)\cdot\frac{Q(s,a)}{N(s,a)} + \beta\cdot\frac{Q_{\text{AMAF}}(s,a)}{N_{\text{AMAF}}(s,a)} + C\sqrt{\frac{\ln N(s)}{N(s,a)}} \tag{5}$$

où $\beta = \sqrt{\frac{k}{3N(s,a)+k}}$, Q_{AMAF} et N_{AMAF} sont les récompenses et visites cumulées pour l'action a dans toutes les simulations, et C est une constante d'exploration. Le paramètre k contrôle l'influence de l'heuristique AMAF : une valeur élevée favorise AMAF, tandis qu'une valeur faible rapproche RAVE de MCTS standard. Nous avons testé RAVE dans le contexte du Puissance 4 pour évaluer son efficacité par rapport à MCTS.

3. Implémentation : MCTS - UCB

3.1. Modélisation du jeu de Puissance 4

La classe `Connect4Game` encapsule toutes les règles et l'état du jeu :

```python
class Connect4Game:
    def __init__(self):
        self.rows = 6
        self.columns = 7
        self.board = np.zeros((self.rows, self.columns), dtype=int)
        self.current_player = 1  # 1 pour Rouge, 2 pour Jaune
        self.game_over = False
        self.winner = None
```

Listing 1: Classe Connect4Game

Les méthodes principales de cette classe sont :

- `get_valid_moves()` : Retourne la liste des colonnes où un pion peut être joué (non pleines).
- `make_move(column)` : Place un pion dans la colonne spécifiée et met à jour l'état du jeu.
- `check_win(player)` : Vérifie si un joueur a aligné 4 pions dans n'importe quelle direction.
- `print_board()` : Affiche l'état actuel du plateau de jeu.

La méthode `make_move` est particulièrement importante car elle gère la logique de jeu, vérifie les conditions de victoire et alterne les joueurs :

```python
def make_move(self, column):
    if column not in self.get_valid_moves():
        return False
    for row in range(self.rows-1, -1, -1):
        if self.board[row][column] == 0:
            self.board[row][column] = self.current_player
            break
    if self.check_win(self.current_player):
        self.game_over = True
        self.winner = self.current_player
    elif not self.get_valid_moves():
        self.game_over = True
    self.current_player = 3 - self.current_player  # Alternance entre 1 et 2
    return True
```

Listing 2: Méthode make_move de Connect4Game

Cette implémentation permet de simuler efficacement le déroulement d'une partie et de vérifier rapidement si un joueur a gagné ou si la partie s'est terminée par un match nul.

3.2. Structure des nœuds pour l'arbre de recherche

La classe `Node` est au cœur des algorithmes MCTS et UCB1-Tuned. Elle représente un nœud dans l'arbre de recherche, correspondant à un état du jeu spécifique :

```python
class Node:
    def __init__(self, game_state, parent=None, move=None):
        self.game_state = deepcopy(game_state)
        self.parent = parent
        self.move = move
        self.children = []
        self.wins = 0
        self.visits = 0
        self.sum_of_squares = 0  # Pour le calcul de la variance
        self.untried_moves = game_state.get_valid_moves()
```

Listing 3: Classe Node pour l'arbre de recherche

Chaque nœud maintient :

- Une copie de l'état du jeu
- Des références à son parent et aux nœuds enfants
- Le coup qui a mené à cet état
- Des statistiques sur les visites et les résultats (victoires)
- La somme des carrés des résultats pour calculer la variance (utilisée par UCB1-Tuned)
- Une liste des coups non encore essayés à partir de cet état

Les méthodes principales de cette classe sont :

```python
def select_child(self):
    # S lectionne un enfant selon la formule UCB1 standard
    c = 1.41  # Constante d'exploration
    best_score = -float('inf')
    best_child = None
    for child in self.children:
        exploitation = child.wins / child.visits if child.visits > 0 else 0
        exploration = math.sqrt(2 * math.log(self.visits) / child.visits) if child.visits > 0 else float('inf')
        score = exploitation + c * exploration
        if score > best_score:
            best_score = score
            best_child = child
    return best_child

def add_child(self, move, game_state):
    # Ajoute un nouveau n ud enfant
    child = Node(game_state, self, move)
    self.untried_moves.remove(move)
    self.children.append(child)
    return child

def update(self, result):
    # Met    jour les statistiques du n ud
    self.visits += 1
    self.wins += result
    self.sum_of_squares += result * result
```

Listing 4: Méthodes principales de la classe Node

La méthode `update` est particulièrement importante car elle maintient à jour les statistiques nécessaires pour les deux algorithmes, notamment la somme des carrés qui permet de calculer la variance empirique pour UCB1-Tuned.

3.3. Implémentation des algorithmes MCTS et UCB1-Tuned

Les deux algorithmes partagent une structure similaire, leur principale différence étant dans la méthode de sélection des nœuds.

3.3.1. Algorithme MCTS standard

```python
class MCTS:
    def __init__(self, iterations=1000):
        self.iterations = iterations

    def get_best_move(self, game_state):
        root = Node(game_state)
        for _ in range(self.iterations):
            node = root
            state = deepcopy(game_state)

```

```
            # Phase de s lection
            while node.untried_moves == [] and node.children != []:
                node = node.select_child()
                state.make_move(node.move)

            # Phase d'expansion
            if node.untried_moves:
                move = random.choice(node.untried_moves)
                state.make_move(move)
                node = node.add_child(move, state)

            # Phase de simulation
            while not state.game_over and state.get_valid_moves():
                state.make_move(random.choice(state.get_valid_moves()))

            # Phase de r tropropagation
            while node:
                result = 0.5 if state.winner is None else 1.0 if state.winner == game_state.current_player else 0.0
                node.update(result)
                node = node.parent

        # S lection du meilleur coup selon le nombre de visites
        best_child = max(root.children, key=lambda c: c.visits, default=None)
        return best_child.move if best_child else random.choice(game_state.get_valid_moves())
```

Listing 5: Classe MCTS

3.3.2. Algorithme UCB1-Tuned

La classe `UCB1TunedSearch` est similaire à MCTS, mais utilise une méthode de sélection différente qui prend en compte la variance empirique des récompenses :

```
def _select_child_ucb_tuned(self, node):
    log_visits = math.log(node.visits) if node.visits > 0 else 0
    best_score = -float('inf')
    best_child = None
    for child in node.children:
        if child.visits == 0:
            return child

        mean = child.wins / child.visits
        variance = (child.sum_of_squares / child.visits) - (mean * mean) if child.visits > 1 else 0
        variance = min(0.25, max(0, variance))  # Borne la variance    0.25

        two_log_N_over_n = 2 * log_visits / child.visits
        sqrt_two_log_N_over_n = math.sqrt(two_log_N_over_n)

        exploration = math.sqrt(two_log_N_over_n * (variance + sqrt_two_log_N_over_n))
        score = mean + 1.41 * exploration

        if score > best_score:
            best_score = score
            best_child = child

    return best_child
```

Listing 6: Méthode de sélection UCB1-Tuned

Cette méthode implémente la formule UCB1-Tuned présentée dans la section précédente. L'ajustement clé est le calcul de la variance empirique et son utilisation pour moduler le terme d'exploration.

3.4. Fonction de simulation

Pour comparer les performances des deux algorithmes, nous avons développé une fonction `run_simulation` qui orchestre une série de parties entre MCTS et UCB1-Tuned :

```
def run_simulation(num_games=100):
    wins_mcts = 0
    wins_ucb_tuned = 0
    draws = 0
    wins_starting = 0
    wins_second = 0

    mcts = MCTS(iterations=1000)
    ucb_tuned = UCB1TunedSearch(iterations=1000)

    for game_num in range(num_games):
        game = Connect4Game()
        # Alterne quel algorithme joue en premier
        AI_red = ucb_tuned if game_num % 2 == 1 else mcts
        AI_yellow = mcts if game_num % 2 == 1 else ucb_tuned
        red_is_mcts = game_num % 2 == 0

        while not game.game_over:
            ai_move = AI_red.get_best_move(game) if game.current_player == 1 else AI_yellow.get_best_move(game)
            game.make_move(ai_move)

        # Comptabilisation des r sultats
        if game.winner is None:
            draws += 1
        elif game.winner == 1:  # Rouge gagne
            wins_starting += 1
            if red_is_mcts:
                wins_mcts += 1
            else:
                wins_ucb_tuned += 1
        else:  # Jaune gagne
            wins_second += 1
            if red_is_mcts:
                wins_ucb_tuned += 1
            else:
                wins_mcts += 1
```

Listing 7: Fonction de simulation des parties

Cette fonction alterne systématiquement quel algorithme joue en premier pour garantir l'équité de la comparaison, et maintient des statistiques détaillées sur les performances de chaque algorithme et sur l'avantage du premier joueur.

3.5. Considérations d'implémentation

Plusieurs choix d'implémentation méritent d'être soulignés :

— **Nombre d'itérations** : Nous avons fixé le nombre d'itérations à 1000 pour les deux algorithmes, ce qui représente un compromis entre la qualité des décisions et le temps d'exécution.

— **Politique de simulation** : Nous utilisons une politique purement aléatoire pour la phase de simulation, ce qui est simple mais peut ne pas refléter un jeu optimal.
— **Sélection du meilleur coup** : À la fin de l'algorithme, nous sélectionnons le coup correspondant au nœud enfant le plus visité, plutôt que celui avec le meilleur ratio de victoires, ce qui est une pratique courante en MCTS.
— **Gestion de la mémoire** : Nous utilisons `deepcopy` pour créer des copies indépendantes des états de jeu, ce qui garantit l'intégrité des simulations mais peut avoir un impact sur les performances.

Notre code est disponible sur GitHub : **lien**

4. Résultats et analyse

4.1. Protocole expérimental

Nous avons mené une campagne de 100 parties entre MCTS et UCB1-Tuned, en alternant quel algorithme joue en premier (Rouge) pour assurer l'équité. Chaque algorithme a effectué 1000 itérations par coup, ce qui représente un compromis entre la qualité des décisions et le temps d'exécution.

Les métriques d'évaluation que nous avons utilisées sont :
— Le taux de victoire de chaque algorithme
— L'avantage du premier joueur (Rouge)
— Le nombre de matchs nuls

4.2. Résultats globaux

Après 100 parties, les statistiques suivantes ont été enregistrées :

Métrique	**Nombre**	**Pourcentage**
Victoires MCTS	45	45.0%
Victoires UCB1-Tuned	54	54.0%
Matchs nuls	1	1.0%
Victoires Rouge (premier joueur)	68	68.0%
Victoires Jaune (second joueur)	31	31.0%

Table 1: Résultats globaux après 100 parties

4.3. Analyse des résultats

Les résultats montrent que UCB1-Tuned surpasse légèrement MCTS avec 54% de victoires contre 45%. Cette différence, bien que modeste, suggère que la prise en compte de la variance dans UCB1-Tuned offre un avantage dans le contexte du Puissance 4.

Plusieurs facteurs peuvent expliquer cette supériorité :
— **Gestion adaptative de l'exploration** : UCB1-Tuned ajuste dynamiquement l'équilibre exploration/exploitation en fonction de la variance observée, ce qui peut être particulièrement utile dans le Puissance 4 où certaines positions sont clairement favorables ou défavorables.
— **Robustesse face aux positions incertaines** : En tenant compte de la variance, UCB1-Tuned peut mieux gérer les positions où le résultat est incertain, en évitant de surexplorer les positions à haute variance.
— **Efficacité des simulations** : Avec le même nombre d'itérations (1000), UCB1-Tuned semble extraire plus d'information utile des simulations grâce à sa formule de sélection plus sophistiquée.

Un autre résultat notable est le fort avantage du premier joueur (Rouge), avec 68% de victoires. Ceci est cohérent avec la théorie du jeu de Puissance 4, où le premier joueur a un avantage théorique. Il est intéressant de noter que les deux algorithmes semblent capables d'exploiter cet avantage lorsqu'ils jouent en premier.

Le faible nombre de matchs nuls (seulement 1%) indique que les algorithmes, même avec un nombre limité d'itérations, sont capables de trouver des stratégies gagnantes dans la plupart des cas. Cela suggère que les 1000 itérations par coup sont suffisantes pour explorer efficacement l'espace de recherche du Puissance 4, du moins pour le niveau de jeu présenté ici.

5. Implémentation RAVE

L'implémentation complète de la méthode RAVE est disponible sur GitHub : **lien**.

5.0.1. Considérations d'implémentation pour RAVE

L'implémentation de RAVE présente également des choix spécifiques qui méritent d'être soulignés :
— **Nombre d'itérations** : Nous avons fixé le nombre d'itérations à 2000 pour RAVE (contre 1000 pour MCTS et UCB1-Tuned), afin de donner à l'heuristique AMAF suffisamment de simulations pour potentiellement converger, tout en restant dans des limites computationnelles raisonnables.
— **Politique de simulation** : Comme pour MCTS, nous utilisons une politique semi-aléatoire qui vérifie d'abord les victoires immédiates avant de choisir un coup aléatoire parmi les options restantes, ce qui améliore légèrement la qualité des simulations par rapport à une approche purement aléatoire.
— **Sélection du meilleur coup** : À l'instar de MCTS, le coup final est choisi en fonction du nœud enfant le plus visité, une pratique standard qui privilégie la robustesse des estimations.
— **Gestion des statistiques AMAF** : RAVE maintient des dictionnaires supplémentaires pour les victoires et visites AMAF par action, ce qui augmente la complexité mémoire et le coût computationnel par rapport à MCTS standard.
— **Gestion de la mémoire** : Comme pour les autres algorithmes, nous utilisons `deepcopy` pour garantir l'intégrité des états de jeu lors des simulations, avec

un impact potentiel sur les performances, particulièrement marqué pour RAVE en raison des calculs AMAF.

5.0.2. Comparaison entre MCTS et RAVE

Pour enrichir notre analyse, nous avons comparé MCTS standard à RAVE dans une série supplémentaire de 100 parties, avec 2000 itérations par coup pour chaque algorithme. Nous avons testé deux valeurs de la constante k dans RAVE ($k = 500$ et $k = 100$) afin d'évaluer l'impact de l'heuristique AMAF. Les résultats sont résumés dans le tableau suivant :

Métrique	$k = 500$		$k = 100$	
	Nombre	%	Nombre	%
Victoires MCTS	73	73.0%	81	81.0%
Victoires RAVE	27	27.0%	19	19.0%
Matchs nuls	0	0.0%	0	0.0%
Victoires Rouge (1)	77	77.0%	69	69.0%
Victoires Jaune (2)	23	23.0%	31	31.0%
Temps moyen MCTS	252.41		275.90	
Temps moyen RAVE	311.38		412.41	

Table 2: Résultats de MCTS contre RAVE pour deux valeurs de k

Analyse : MCTS surpasse largement RAVE dans les deux configurations, avec 73% de victoires pour $k = 500$ et 81% pour $k = 100$. Réduire k de 500 à 100, ce qui diminue l'influence de l'heuristique AMAF, a dégradé les performances de RAVE (de 27% à 19% de victoires) tout en augmentant son temps moyen par coup (de 311 ms à 412 ms). Cela suggère que, dans le Puissance 4, l'approche AMAF de RAVE est moins efficace que les simulations pures de MCTS. Contrairement à des jeux comme le Go, où AMAF exploite des corrélations globales entre les coups, le Puissance 4 présente des dépendances locales et un arbre de recherche peu profond, favorisant la simplicité de MCTS. De plus, l'avantage du premier joueur (Rouge) reste marqué (77% avec $k = 500$, 69% avec $k = 100$), mais RAVE semble moins capable de l'exploiter efficacement. Le temps d'exécution plus élevé de RAVE (jusqu'à 49% supérieur à MCTS avec $k = 100$) reflète le coût computationnel de la gestion des statistiques AMAF, sans bénéfice notable dans ce contexte.

5.1. Analyse complémentaire des simulations MCTS vs RAVE

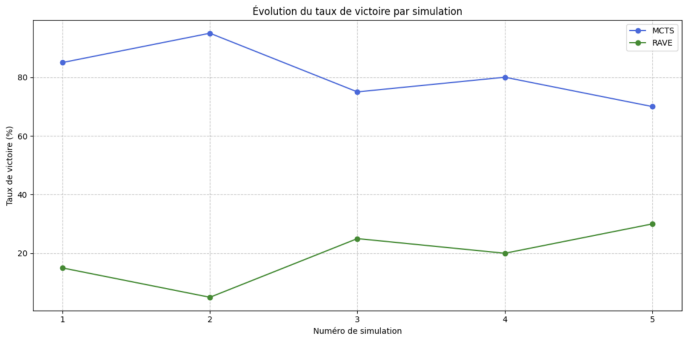

Figure 1: Évolution du taux de victoire par simulation pour MCTS et RAVE

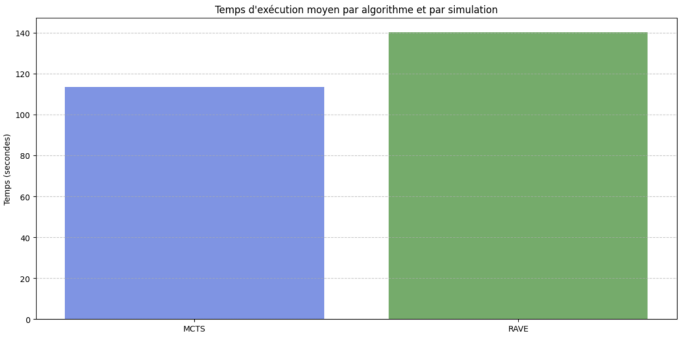

Figure 2: Temps d'exécution moyen par algorithme et par simulation

5.1.1. Interprétation des résultats

L'analyse des simulations multiples entre MCTS et RAVE pour le jeu de Puissance 4 révèle :

- **Performance** : MCTS maintient un avantage constant avec des taux de victoire entre 70% et 95%, tandis que RAVE oscille entre 5% et 30%
- **Évolution** : Les performances de RAVE montrent une tendance à l'amélioration dans les dernières simulations, passant d'environ 5% à 30% de victoires
- **Temps d'exécution** : RAVE nécessite environ 140 secondes par simulation contre 115 secondes pour MCTS, soit 22% de temps supplémentaire

Ces résultats confirment que l'heuristique AMAF utilisée par RAVE, bien qu'efficace pour des jeux comme le Go, ne semble pas bien adaptée à la structure du Puissance 4. Le surcoût computationnel ne se traduit pas par une amélioration significative des performances, MCTS restant supérieur tant en efficacité qu'en temps d'exécution.

6. Implémentation NRPA

L'implémentation complète de la méthode RAVE est disponible sur GitHub : **lien**.

6.0.1. *Considérations d'implémentation pour NRPA*

L'implémentation de NRPA (Nested Rollout Policy Adaptation) présente plusieurs particularités techniques qui influencent son comportement :

- **Niveaux de récursion** : Nous avons testé NRPA avec différents niveaux de récursion (1, 2), chaque niveau augmentant la profondeur de recherche mais aussi considérablement le temps de calcul.
- **Nombre d'itérations** : Le nombre d'itérations par niveau a été paramétré (10 pour niveau 1, 20 pour niveau 2), ce qui détermine combien de simulations sont effectuées avant l'adaptation de la politique.
- **Adaptation de politique** : Un taux d'apprentissage fixe (alpha = 0.5) a été utilisé pour la mise à jour de la politique, ce qui influence la vitesse de convergence de l'algorithme.
- **Représentation des états** : Les états du jeu sont convertis en chaînes de caractères pour servir de clés dans la table de politique, une approche simple mais potentiellement coûteuse en mémoire.
- **Politique initiale** : La politique est initialisée avec des poids uniformes, puis ajustée au fur et à mesure des simulations, permettant un apprentissage progressif des stratégies efficaces.
- **Gestion de la mémoire** : L'utilisation extensive de `deepcopy` pour les états et les politiques garantit l'intégrité des données mais augmente significativement la charge computationnelle, particulièrement pour les niveaux de récursion élevés.

6.0.2. *Comparaison entre MCTS et NRPA*

Pour évaluer l'efficacité relative de NRPA par rapport à MCTS, nous avons effectué deux séries de tests sur 20 parties chacune, avec différentes configurations. Les résultats sont résumés dans le tableau suivant :

Métrique	Nombre	%
Config 1 (MCTS 500 iter vs NRPA niveau 1)		
Victoires MCTS	20	100.0%
Victoires NRPA	0	0.0%
Matchs nuls	0	0.0%
Temps moyen MCTS (ms)	269	
Temps moyen NRPA (ms)	24	
Nombre moyen de coups	11.6	
Config 2 (MCTS 1000 iter vs NRPA niveau 2)		
Victoires MCTS	19	95.0%
Victoires NRPA	1	5.0%
Matchs nuls	0	0.0%
Temps moyen MCTS (ms)	550	
Temps moyen NRPA (ms)	35585	
Nombre moyen de coups	10.8	

TABLE 3: Résultats de MCTS contre NRPA pour deux configurations

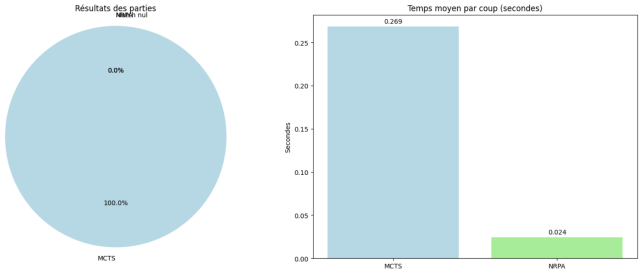

FIGURE 3: Résultats de la première configuration : MCTS 500 itérations vs NRPA niveau 1

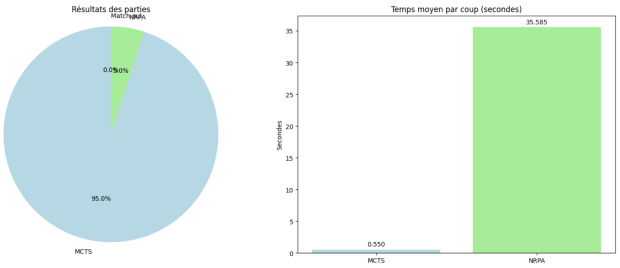

FIGURE 4: Résultats de la deuxième configuration : MCTS 1000 itérations vs NRPA niveau 2

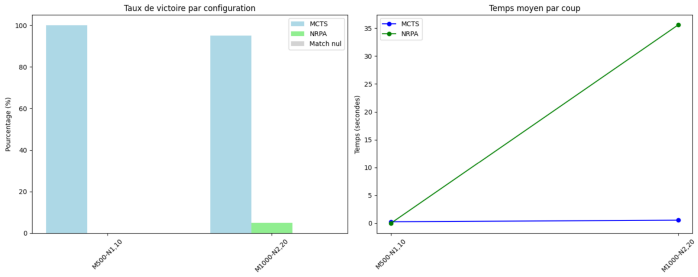

FIGURE 5: Comparaison des taux de victoire et temps moyen par coup pour MCTS et NRPA

6.0.3. *Analyse des résultats*

Notre étude comparative révèle une domination claire de MCTS sur NRPA dans le contexte du jeu Puissance 4. Dans la première configuration (MCTS 500 itérations vs NRPA niveau 1), MCTS a remporté 100% des parties. La seconde configuration (MCTS 1000 itérations vs NRPA niveau 2) montre une légère amélioration pour NRPA qui parvient à gagner une partie (5% de victoires), mais au prix d'une explosion du temps de calcul.

Cette augmentation du niveau de récursion de NRPA a entraîné une hausse spectaculaire du temps de traitement, passant de 24 ms à plus de 35 secondes par coup (soit un facteur multiplicatif de 1482), rendant l'algorithme impraticable pour des utilisations en temps réel ou pour des niveaux de récursion supérieurs.

Plusieurs facteurs expliquent cette différence de performance :

- **Structure du jeu** : Le Puissance 4 présente des dé-

pendances locales entre les coups et un arbre de jeu relativement peu profond, ce qui favorise l'approche équilibrée d'exploration/exploitation de MCTS.
- **Complexité calculatoire** : La croissance exponentielle du temps de calcul avec le niveau de récursion limite considérablement le potentiel d'amélioration de NRPA pour ce jeu.
- **Comparaison avec RAVE** : RAVE, qui obtenait entre 19% et 27% de victoires contre MCTS, s'avère nettement plus compétitif que NRPA. Cela suggère que la méthode AMAF (All Moves As First) de RAVE parvient à capturer certaines corrélations utiles entre les coups de Puissance 4, contrairement à l'adaptation de politique récursive de NRPA.

En conclusion, notre étude établit une hiérarchie claire entre les trois algorithmes évalués pour le jeu de Puissance 4 :

1. **MCTS** offre le meilleur équilibre entre qualité des décisions et efficacité computationnelle
2. **RAVE** propose une alternative intéressante mais sous-optimale
3. **NRPA** s'avère inadapté à ce contexte particulier, malgré ses succès documentés dans d'autres domaines comme les puzzles combinatoires ou certains jeux solitaires

7. Conclusion

Cette étude comparative des algorithmes MCTS, UCB1-Tuned, RAVE et NRPA dans le contexte du jeu de Puissance 4 a permis de mettre en évidence plusieurs aspects importants concernant l'efficacité de ces méthodes basées sur Monte Carlo.

Nos expérimentations ont clairement établi une hiérarchie entre ces quatre algorithmes. MCTS et UCB1-Tuned présentent les meilleures performances, UCB1-Tuned surpassant légèrement MCTS standard (54

Ces résultats suggèrent que la nature du jeu de Puissance 4, caractérisée par des dépendances locales entre les coups et un arbre de recherche relativement peu profond, favorise les approches équilibrées d'exploration/exploitation comme MCTS et UCB1-Tuned. L'heuristique AMAF utilisée par RAVE, bien que conçue pour accélérer la convergence, n'apporte pas de bénéfice significatif dans ce contexte, contrairement à son efficacité documentée pour des jeux comme le Go. Quant à NRPA, son mécanisme d'adaptation de politique récursive semble mal adapté à la structure du Puissance 4, et sa complexité calculatoire devient rapidement prohibitive.

Un résultat secondaire mais cohérent avec la théorie du jeu est l'avantage significatif du premier joueur, observé dans toutes nos expérimentations. Cet avantage confirme que les algorithmes, même avec un nombre limité d'itérations, parviennent à exploiter les caractéristiques fondamentales du jeu.

Pour des travaux futurs, plusieurs pistes d'amélioration pourraient être explorées :

- L'optimisation des paramètres de MCTS et UCB1-Tuned (constante d'exploration, nombre d'itérations) pour maximiser leur efficacité
- L'implémentation de politiques de simulation plus sophistiquées, potentiellement guidées par des heuristiques spécifiques au Puissance 4
- L'exploration d'autres variantes de MCTS comme Rapid Action Value Estimation with Exploring Moves (RAVE-EM) ou Last Good Reply (LGR)
- Le développement d'approches hybrides combinant les forces de différents algorithmes, comme l'intégration de mécanismes adaptatifs inspirés d'UCB1-Tuned dans RAVE
- L'application de techniques d'apprentissage profond pour guider les simulations ou évaluer les positions, comme dans AlphaZero

En conclusion, cette étude démontre l'efficacité des méthodes Monte Carlo pour le jeu de Puissance 4, tout en soulignant l'importance de choisir l'algorithme approprié en fonction des caractéristiques spécifiques du problème. Elle confirme également que des ajustements relativement simples, comme la prise en compte de la variance dans UCB1-Tuned, peuvent apporter des améliorations significatives par rapport aux approches standard.