Amélioration de la performance de l'algorithme de Monte Carlo Tree Search en biaisant l'étape de simulation à l'aide d'une heuristique.

Projet Second Concours
École Normale Supérieure de Lyon

Gabriel BATHIE

3 juillet 2018

Résumé

L'algorithme de Monte Carlo Tree Search (MCTS) se distingue d'autres algorithmes de même classe par sa structure en quatre étapes successives répétées (Sélection, Expansion, Simulation, Rétropropagation) qui lui donne un avantage sur d'autre algorithmes (*e.g.* MiniMax) pour le traitement de certains problèmes, notamment pour les jeux à deux joueurs avec un grand espace de recherche (*e.g.* jeu de Go).
Des travaux sur chacune des étapes de MCTS ou bien sur sa parallélisation ont permis des améliorations des performances de l'algorithme. Dans la continuité de ceux-ci, nous avons exploré deux pistes d'utilisation d'une heuristique pour biaiser les choix effectués lors de la phase de simulation.
Tout d'abord, l'utilisation de l'heuristique pour pondérer le choix des coups dans la simulation n'a pas donné d'avantage clair à l'algorithme ainsi modifié.
En revanche, son utilisation pour sélectionner de façon gloutonne les coups dans la simulation a augmenté de façon notable les gains de l'algorithme modifié.

Table des matières

1 Algorithme de Monte Carlo Tree Search

L'algorithme de Monte Carlo Tree Search (MCTS) est un algorithme servant à trouver une solution optimale à certains problèmes de décision, comme par exemple les jeux à un ou plusieurs joueurs (Échecs, Go, etc.), ou encore à des problèmes d'optimisation comme le problème d'Allocation Dynamique de Ressources (Bertsimas et al., 2014), ou d'optimisation de l'implémentation d'une libraire (De Mesmay et al., 2009).
Dans ce projet, nous étudierons l'algorithme de MCTS uniquement appliqué au domaine des jeux dans lequel MCTS est le plus utilisé., et le comparerons avec d'autres algorithmes du même domaine.

1.1 Jeux, arbre de jeux et algorithmes de jeux

Ici, par jeux, nous entendrons principalement jeux d'opposition à un ou plusieurs joueurs, à **temps discret** (une partie du jeu se déroule par coups successifs, que les joueurs jouent en même temps ou chacun leur tour), à **somme nulle** (la somme des gains des joueurs est égale à zéro ; dans le cas des jeux à deux joueurs, cela veut dire que si un joueur gagne, l'autre perd) et **finis** (le jeu se finit en un nombre fini de coups).

Ces jeux ont généralement une seule situation initiale et la partie se déroule à partir de celle-ci. Si l'on représente l'état du jeu à un instant donné par un nœud et les coups des joueurs comme des arêtes orientées de la situation avant le coup vers la situation après le coup, en commençant à une situation s_0, on obtient alors une représentation du jeu et de toutes les parties possibles à partir de s_0 sous forme d'un arbre enraciné en s_0. Les feuilles correspondent alors aux fins de parties : c'est l'arbre de jeu. Dans cette construction, si l'on peut arriver à une même situation du jeu par plusieurs suites de coups différents, il y a alors deux nœuds différents dans l'arbre associées à cette situation.

Le problème de décision que l'on cherche à résoudre à l'aide d'un algorithme est alors le suivant :

> Étant donné une situation s du jeu et un ensemble A de coups (ou d'actions) que l'on peut effectuer, quelle est l'action $a \in A$ qui nous permettra d'obtenir le meilleur résultat (c'est-à-dire gagner la partie, ou si ce n'est pas possible, arriver à un match nul et, le cas échéant, maximiser son score).

Généralement, ce problème doit être résolu avec une contrainte de temps maximal d'exécution, soit par coup, soit pour l'ensemble des coups de la partie.
La majorité des algorithmes servant à répondre à cette problématique reposent sur un parcours de l'arbre de jeu : c'est par exemple le cas de l'algorithme MiniMax. Nous allons maintenant présenter l'algorithme MiniMax et le principe derrière son fonctionnement : il servira de base de comparaison avec l'algorithme de MCTS.

Principe du MiniMax et algorithme de MiniMax

Le principe du MiniMax donne une solution théorique au problème du meilleur coup à jouer dans le cadre des jeux à deux joueurs à somme nulle.

Considérons l'arbre de jeu d'un jeu à deux joueurs, à temps discret et à somme nulle, décrit par la **Figure**. 1.

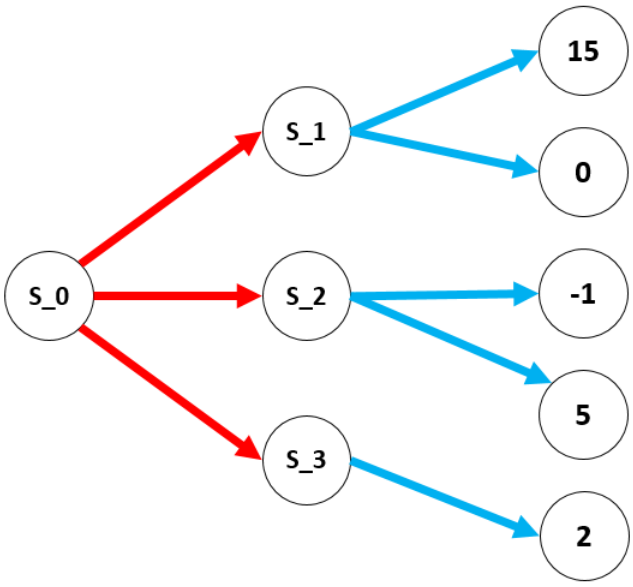

FIGURE 1 – Exemple d'arbre de jeu

Les nœuds représentent les situations, les arêtes rouges (resp. bleues) représentent les coups joués par le premier (resp. le deuxième) joueur j_1 (resp. j_2). Les feuilles de l'arbre représentent les situations où la partie est finie, et dans ce cas, le nombre sur le nœud représente le score du joueur 1. Le but des deux joueurs est de maximiser leur propre score. Comme le jeu est à somme nulle, $score_{j_2} = -score_{j_1}$, et maximiser son score est équivalent à minimiser celui de l'adversaire.

Intéressons nous alors à la situation s_1. C'est au joueur j_2 de jouer, et son but est de maximiser son score, ce qui revient à minimiser le score de j_1 (score affiché dans les feuilles de l'arbre en **Fig**. 1). Il va donc jouer le coup qui l'amène vers le score final de j_1 minimal si il se retrouve dans la situation s_1 (à savoir, 0). Ainsi, la situation s_1 aura comme valeur effective 0 pour j_1. En appliquant le raisonnement de manière similaire pour s_2 et s_3, on obtient alors l'arbre de la **Fig.** 2.

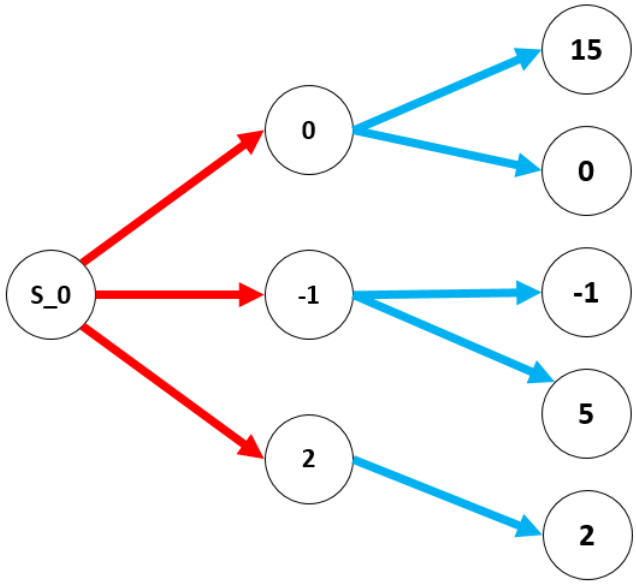

FIGURE 2 – Arbre de jeu avec prédiction du coup du joueur 2

On peut alors réitérer le raisonnement précédent pour le joueur 1 cherchant à maximiser son score : il a intérêt à sélectionner la situation s_3 pour maximiser son score, et le score effectif de la position s_0 est 2.

Cet algorithme peut se généraliser de manière récursive à un arbre de jeu de n'importe quelle hauteur finie, en partant des feuilles : lorsque c'est au joueur 2 de jouer, le nœud prend la valeur du minimum des valeurs de ses fils, et inversement lorsque c'est au joueur 1 de jouer (d'où le nom de MiniMax). On obtient alors l'algorithme suivant :

Algorithm 1 Algorithme MiniMax théorique

```
procedure MiniMax(s_0, MaxPlayer)
    if s_0 is terminal then
        return Score(s_0)
    else
        if MaxPlayer then
            return max {MiniMax(s_0 + m, false) for m in LegalMoves(s_0)}
        else
            return min {MiniMax(s_0 + m, true) for m in LegalMoves(s_0)}
        end if
    end if
end procedure
```

Ensuite, il faut appeler cette procédure sur chaque coup légal de la situation initiale du problème, et on choisit de jouer celui avec le score maximal.
Cet algorithme permet donc d'obtenir une réponse théorique au problème du meilleur coup, mais il demande une exploration complète de l'arbre de jeu, ce qui est en pratique impossible dans un court intervalle de temps. En effet, si on note f le facteur de branchement maximal de l'arbre de jeu (qui est égal au nombre maximal de coups possible pour un joueur dans la partie), le nombre de nœuds de l'arbre est $n = \mathcal{O}(d^f)$, ou d est la hauteur de l'arbre, un nombre de coups [1]. Par exemple, au jeu d'échecs, la taille totale de l'arbre de jeu (pour les parties de 40 coups par joueur) est estimée à 10^{120} (d'après Shannon, 1950).
En pratique, l'algorithme de MiniMax explore jusqu'à une profondeur d donnée, et utilise une fonction *heuristique* d'évaluation pour extrapoler la valeur théorique des situation à une profondeur d dans l'arbre. L'algorithme précédent devient alors :

Algorithm 2 Algorithme MiniMax pratique

```
procedure MiniMax(s_0, MaxPlayer, d)
    if s_0 is terminal or d = 0 then
        return Evaluation(s_0)
    else
        if MaxPlayer then
            return max {MiniMax(s_0 + m, false, d − 1) for m in LegalMoves(s_0)}
        else
            return min {MiniMax(s_0 + m, true, d − 1) for m in LegalMoves(s_0)}
        end if
    end if
end procedure
```

1. Il existe des améliorations de l'algorithme MiniMax qui nécessitent seulement d'étudier une partie de l'arbre. Nous ne les aborderons pas ici, mais la taille de l'espace de recherche augmentera quand même de façon exponentielle dans le cas général.

Cette approche (et ses variantes) est très utilisée depuis le milieu du XIX^e siècle, et a notamment permis à un ordinateur (Deep Blue) de battre le numéro un mondial du jeu d'échecs en 1997, Garry Kasparov.

Toutefois, cet algorithme nécessite de connaître une bonne fonction d'évaluation pour le jeu étudié. C'est le cas du jeu d'échecs, mais ce n'est pas le cas de d'autres jeux à priori plus complexe comme le jeu de Go. C'est pourquoi la recherche s'est orientée ces dernières années sur d'autres algorithme, notamment celui de Monte Carlo Tree Search.

1.2 Structure de l'algorithme de Monte Carlo Tree Search

L'algorithme de MCTS (cf. **Algo.** 3) construit un arbre de jeu en répétant 4 étapes successives à partir de la racine (le noeud de l'arbre comprenant la situation initiale, celle où l'on cherche le meilleur coup à jouer) : Sélection, Expansion, Simulation et Rétropropagation (*Back-propagation)* tant qu'il est en dessous du temps de calcul maximal t_{max}, puis il retourne le nœud fils de la racine considéré le plus intéressant (cette sélection se fait différemment en fonction de la stratégie). Les noeuds de l'arbre construit par l'algorithme portent plusieurs informations :

Algorithm 3 Algorithme de Monte Carlo Tree Search

```
procedure MCTS(s_0, t_max)
    root ← node(s_0)
    while time ≤ t_max do
        v_s ← Select(root)
        Expand(root, v_s)
        r ← Simulate(v_s)
        Backprop(v_s, r)
    end while
    return BestChild(root)
end procedure
```

le nombre n de fois qu'ils on été visités, un score w et les données associées à la situation du jeu qu'ils représentent.

L'étape de Sélection doit choisir un nouveau coup v_s à explorer à partir de la racine, en fonction des scores w_i et des nombre de visites n_i des nœuds de l'arbre. L'étape d'expansion consiste ensuite à ajouter le nœud v_s à l'arbre de jeu. L'étape de Simulation consiste à donner un score r au nœud v_s, et l'étape de rétropropagation fait remonter ce score en l'ajoutant à celui de tous les nœuds ancêtres de v_s dans l'arbre, jusqu'à la racine, en incrémentant également leur nombre de visites n.

En pratique, le comportement de chacune des 4 étapes de l'algorithme varie en fonction des cas d'étude et de tests empiriques. Toutefois, il existe une version de l'algorithme qui sert généralement de base aux travaux basés sur MCTS et qui démontre nativement des bons résultats : il s'agit d'une variante de MCTS nommée UCT (*Upper Confidence bounds applied to Trees*), introduite par Kocsis et Szepesvári (2006). La popularité de cette variante fait que l'on désigne souvent l'UCT par l'appellation MCTS. La figure 3 illustre le fonctionnement de l'UCT, décrit ci-dessous.

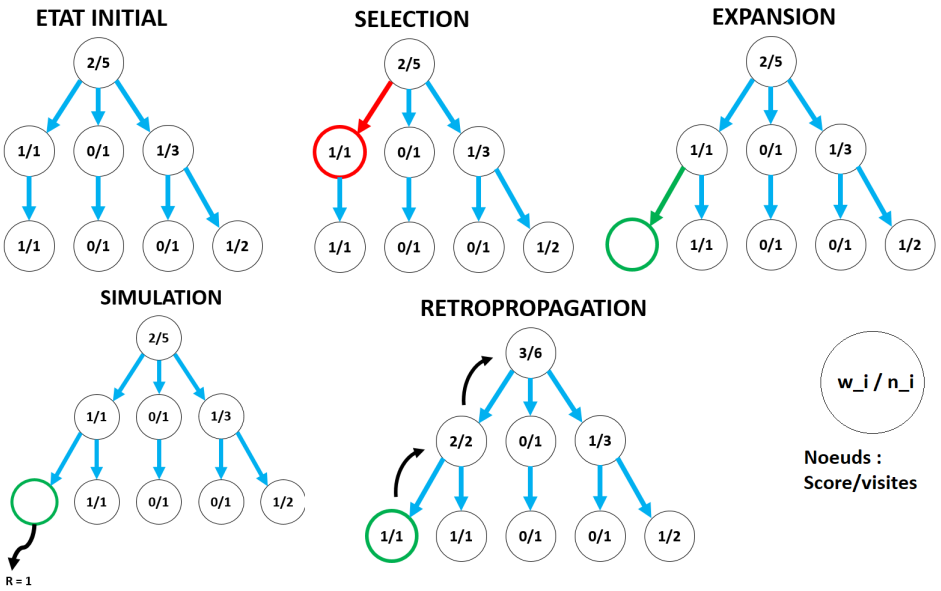

FIGURE 3 – Illustration d'une itération du fonctionnement de l'UCT

Sélection :

Dans UCT, la sélection est considérée comme une répétition d'instance du problème du bandit à k bras, où dans chaque situation, les bras correspondent aux coups, et la récompense associée est le score à l'issue de la partie vers laquelle mène chaque coup.
La sélection se fait en appliquant la formule de l'UCB (*Upper Confidence Bounds)*, introduite par Auer et al. (2002), qui permet de choisir entre explorer en profondeur les coups prometteurs (ceux avec un haut score w/n) et raffiner les connaissances des coups incertains (ceux avec peu de visites). On choisit, dans une situation, un coup parmi ceux qui maximisent la quantité

$$\frac{w_i}{n_i} + C\sqrt{\frac{\log(n_p)}{n_i}}$$

avec w_i le score du nœud après le coup, n_i son nombre de visites, n_p nombre de visites du nœud avant le coup et C une constante en théorie égale [2] à $\sqrt{2}$.
On obtient alors une politique d'exploration optimale : le regret [3] croit en $\mathcal{O}(\log(n))$. Si un nœud n'a jamais été visité, on considère son score comme $+\infty$, et il est donc sélectionné avant tous les autres.
On répète ensuite cette opération récursivement jusqu'à sélectionner un nœud ayant au moins un fils n_s non exploré (n'étant pas encore dans l'arbre) et on lui applique l'étape suivante d'expansion, ou jusqu'à avoir sélectionné un coup terminal : on passe alors directement à la rétropropagation. L'algorithme de la sélection peut alors s'exprimer comme suit :

2. Cette constante est en pratique souvent choisie empiriquement
3. Regret : nombre de fois que l'on choisit d'explorer un coup qui n'est pas le meilleur coup. Il peut pas croître moins vite que $\mathcal{O}(\log(n))$ dans ce problème.

Algorithm 4 Algorithme de Sélection dans l'UCT

```
procedure SELECT(s_0)
    if s_0 is terminal then
        r ← score(s_0)
        Backprop(r, s_0)
    else
        if s_0 has unexplored children then
            Expand(s_0)
        else
            s_n ← argmax_nodes{w_i/n_i + C√(log(n)/n_i)}
            Select(s_n)
        end if
    end if
end procedure
```

Expansion :

On ajoute le nœud n_s à l'arbre de jeu, et on passe à l'étape de simulation à partir de sa situation associée s_e.

Simulation :

On tire des coups aléatoires (légaux) successifs à partir de s_e jusqu'à la fin de la partie, et on récupère le résultat r que l'on passe dans la rétropropagation.

Rétropropagation (*Backpropagation*) :

On ajoute r au score de tous les nœuds ancêtres de s_e, et on augmente de 1 leur nombre de visites.

On répète ensuite de nouveau ces quatre étapes si la condition de temps n'est pas dépassée.

Sélection du meilleur coup :

L'algorithme UCT retourne alors le nœud fils de la racine maximisant la quantité $\frac{w_i}{n_i}$.

Caractéristiques et intérêt de MCTS et UCT

Une des premières caractéristiques intéressante de MCTS est le fait qu'il peut s'arrêter à (presque) tout moment : un cycle des 4 étapes prends généralement un temps très court, et on obtient un résultat qui s'affine au cours du temps, plus l'on a d'itération, alors que la recherche MiniMax ne peut pas retourner de résultat si toutes les feuilles n'ont pas été explorées.
D'autre part, UCT ne nécessite pas d'heuristique pour fonctionner, contrairement à MiniMax : c'est notamment cela qui l'a rendu très utilisé pour le jeu de Go, pour lequel on ne connaît pas d'heuristique d'évaluation efficace.
Enfin, Kocsis et al.(2006) ont prouvé que la probabilité que le résultat donné par UCT soit le résultat optimal de la recherche MiniMax converge à l'infini vers 1.

Ces caractéristiques ont suscité l'intérêt de nombreux chercheurs qui ont travaillé à essayer d'améliorer plusieurs aspects de l'algorithme.

2 Travaux principaux sur MCTS jusqu'à aujourd'hui

Les travaux d'optimisation de l'algorithme de MCTS, initiés notamment par Kocsis et Szepesvári (2006), se rangent en deux grandes catégories : les recherches visant à améliorer les performances de l'algorithme de manière générale et celles visant à améliorer les performances de MCTS sur un domaine ou un jeu précis..

Pour améliorer les performances de l'algorithme, les travaux se concentrent généralement sur une des quatre étapes ou sur la manière de choisir le meilleur coup ou sur les possibilités de parallélisation de l'algorithme.

Les améliorations de la sélection visent généralement à trouver des méthodes qui donnent de meilleurs résultat que l'UCB, que ce soit pour un problème particulier ou dans le cas général. Par exemple, Auer et al. (2002) présentent une autre formule, nommée UCB-Tuned, qui, dans leur expériences, donne de meilleurs résultats que la formule UCB simple présentée en partie 1. Les améliorations de la simulation sont en général spécifiques à des domaines particuliers. On en trouve notamment plusieurs pour le jeu de Go, visant à accélérer la simulation aléatoire. La durée d'une partie de Go pouvant excéder 300 coups sur un plateau 19×19, Chaslot et al. (2010) proposent d'accélérer la simulation en remplissant aléatoirement N intersections du plateau dont tous les voisins sont vides, avant de commencer la simulation. Il reste alors beaucoup moins de mouvement à simuler pour arriver à la fin de la partie (plateau rempli).
Browne et al. (2012) présente un panorama de la recherche autour l'algorithme jusqu'en 2012.

Une avancée particulièrement notable depuis 2012 a été accomplie par les travaux de Silver et al. (2016, 2017) qui ont utilisé des réseaux de neurones pour créer une fonction de sélection et pour accélérer l'étape de simulation de MCTS. Leur travaux ont permis à un ordinateur de battre à plusieurs reprise les meilleur joueurs de jeu de Go au monde (Silver et al., 2016). Ils ont également étendu par la suite leur algorithme à d'autre jeux, comme le jeu d'échecs (Silver et al., 2017).

3 Utilisation d'une fonction heuristique pour biaiser la sélection grâce à des connaissances externes

L'algorithme UCT utilise des simulations aléatoires pour estimer l'intérêt d'un coup. Toutefois, en jouant des coups aléatoires, l'algorithme explore des pistes improbables, qui n'influent donc pas sur le score réel du coup. Pour cela, nous allons donc explorer différentes pistes d'utilisation d'une heuristique donnant un score aux coups pour biaiser la simulation : le but est de sélectionner le plus souvent les meilleurs coups. Cela permettrait d'obtenir plus rapidement une bonne approximation du score d'un coup, et donc d'accélérer la vitesse convergence de l'algorithme, en terme de nombre d'itérations.

Pour nos expériences, nous utiliserons ici le jeu du Puissance 4. La fonction heuristique utilisée est décrite en annexe A. L'intégralité du code utilisé pour les tests est disponible sur github, à l'adresse `https://github.com/GBathie/MCTS`

3.1 Utilisation de l'heuristique pour générer des probabilités de sélection

Tout d'abord, nous avons utilisé l'heuristique pour pondérer le choix des coups dans la simulation, en affectant une probabilité de sélection plus élevée aux coups dont le score heuristique est plus élevé. Pour cela, on utilise la fonction softmax, qui associe à un vecteur $(x_i) \in \mathbb{R}^n$ en un

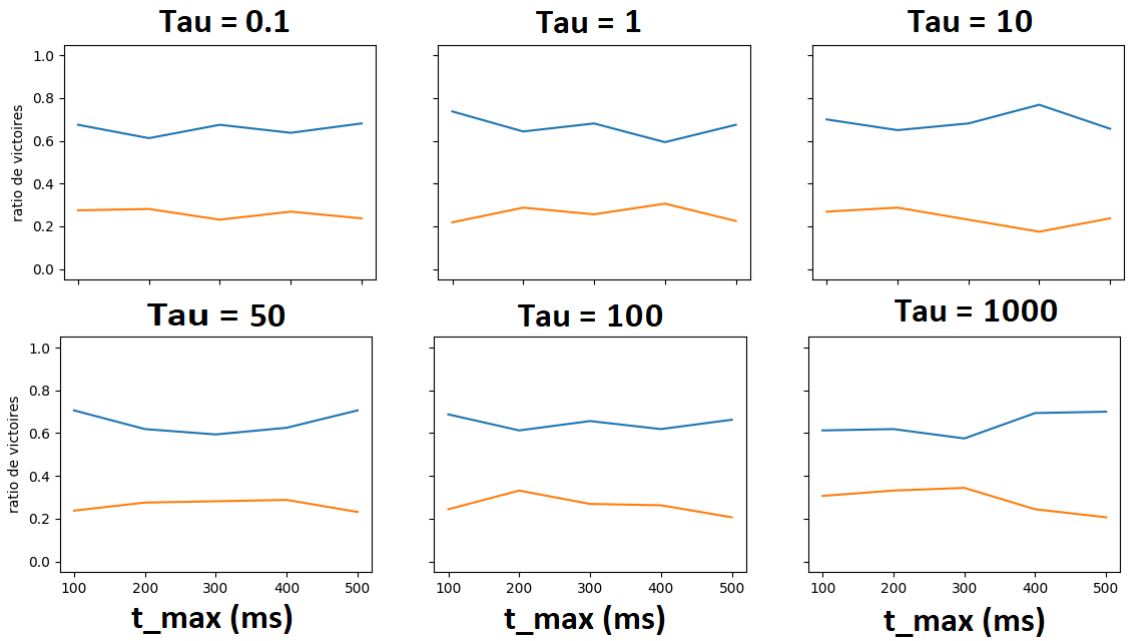

FIGURE 4 – Résultats des parties de S-MCTS (en orange) contre UCT (en bleu), pour différentes valeurs de τ

vecteur de probabilités $(p_i)_{i=1..n} \in [0;1]^n$, tout en conservant l'ordre : si $x_i \geq x_j$, alors $p_i \geq p_j$. La formule de la fonction softmax sur le vecteur de scores $(x_i)_{i=1..n}$ est :

$$\forall i \in \{1,..,n\}, p_i = softmax(x_i) = \frac{e^{\tau x_i}}{\sum_{k=1}^{n} e^{\tau x_k}}$$

On peut vérifier que $\sum_{i=1}^{n} p_i = 1$ et $\forall i, p_i \geq 0$. τ est un paramètre positif qui permet de lisser ou non les p_i : si $\tau = 0$, $\forall i, p_i = 1/n$, ce qui donne une sélection uniforme des coups. Si $\tau = +\infty$ alors les p_i sont tous très petits sauf p_j tel que $x_j = \max x_i$. La fonction softmax avec $\tau = +\infty$ réalise alors une sélection du maximum.

Nous appellerons **S-MCTS** (pour *Softmax*-MCTS) l'algorithme UCT dont la phase de sélection se fait en choisissant les coups avec une probabilité résultant de la fonction softmax appliquée au vecteur des scores donnés par la fonction heuristique.

Test de S-MCTS :

Nous avons ensuite testé S-MCTS en lui faisant jouer des parties de Puissance 4 contre l'algorithme UCT, avec le même temps de calcul t_{max}. Les parties sont toujours jouées par deux, S-MCTS jouant en tant que premier joueur, et second dans l'autre, pour enlever le biais de résultat dû à l'ordre de jeu : le premier joueur est avantagé au puissance 4, si sa stratégie est parfaite, il gagne la partie, quelles que soient les réponses de l'adversaire.
Nous avons ensuite regardé les pourcentages de victoire sur 4000 parties, pour chaque temps, pour différentes valeurs de τ.
Valeurs de t_{max} étudiées : 100, 200, 300, 400, 500 ms. Valeurs de τ étudiées : 0.1, 1, 10, 50, 100, 1000.

Résultats et Analyse :

La figure 4 présente la part de parties gagnées par S-MCTS et UCT en fonction des valeurs de t_{max} et de *tau* On remarque tout d'abord que les performances de S-MCTS sont inférieures à celles de MCTS dans toutes les situations. Cela est dû à l'implémentation de la fonction

softmax : son calcul est très couteux en temps, mais la modification de la sélection vers des meilleurs coups (d'après l'heuristique) ne suffit pas a contrebalancer cette utilisation de temps supplémentaire. Cela peut être confirmé en comptant le nombre d'itérations effectuées par S-MCTS : il est généralement 3 fois inférieur au nombres d'itérations de UCT, pour un même temps de calcul.

Ensuite, on peut également remarquer que les performances de S-MCTS augmentent, en moyenne, avec τ. En effet, pour $\tau = 0.1$, la moyenne de victoires de S-MCTS est de 24%, contre 29% pour $\tau = 1000$. Il semblerait donc que sélectionner le meilleur coup de manière gloutonne apporte les meilleurs résultats. C'est la piste que nous avons ensuite choisi d'explorer.

3.2 Utilisation de l'heuristique pour réaliser la sélection de manière gloutonne

Nous avons donc testé une nouvelle version de l'algorithme, dans lequel la sélection se fait en choisissant le coup auquel l'heuristique attribue le meilleur score. Cette version est nommée **G-MCTS**, pour *Greedy*-MCTS.

Premier test de G-MCTS :

Nous avons donc testé G-MCTS contre UCT dans les mêmes condition que précédemment : on s'intéresse à la proportion de victoires de chacun au cours d'un match de 8000 parties, et ce, pour plusieurs valeurs de t_{max}. Valeurs de t_{max} étudiées : $100, 200, 300, 400, 500$ ms. Les résultats sont présentés en figure 5

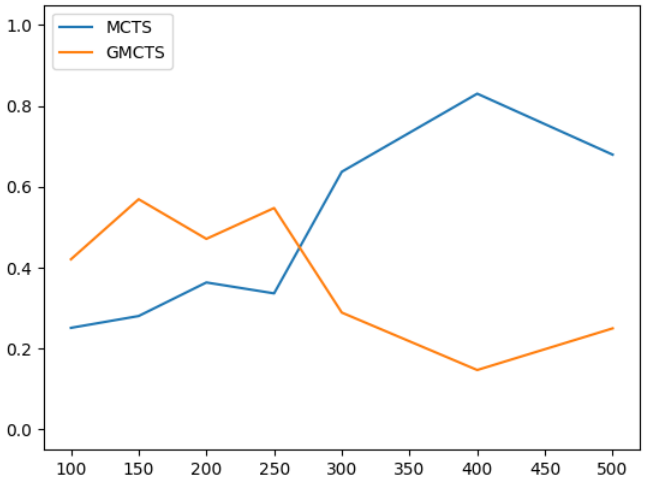

FIGURE 5 – Résultats des parties de G-MCTS (en orange) contre UCT (en bleu).

On remarque que G-MCTS donne des meilleures performances que UCT pour les temps de calculs faibles. Cela peut se comprendre car les deux algorithmes calculent peu d'itérations, mais celles de G-MCTS donnent le score des coups de manière plus précise grâce à l'heuristique : le choix du meilleur coup est donc plus précis.

Toutefois, on observe également que les performances de G-MCTS sont en dessous de celles de UCT pour des temps de calculs plus grands.

Second test de G-MCTS : variation de la constante de sélection

Nous avons alors essayé une autre piste d'amélioration : comme les simulations de G-MCTS apportent plus d'informations en moyenne que celles de UCT, il ne faut peut-être pas réaliser la même quantité d'exploration, c'est-à-dire ne pas utiliser la même constante C dans la formule de la sélection (voir chap. 1.3), qui valait précédemment $\sqrt{2}$.
Nous avons renouvelé le test précédent, en variant les valeurs de C. Valeurs étudiées : $0.5, 1, 1.6$.
Les résultats sont présentés en figure 6

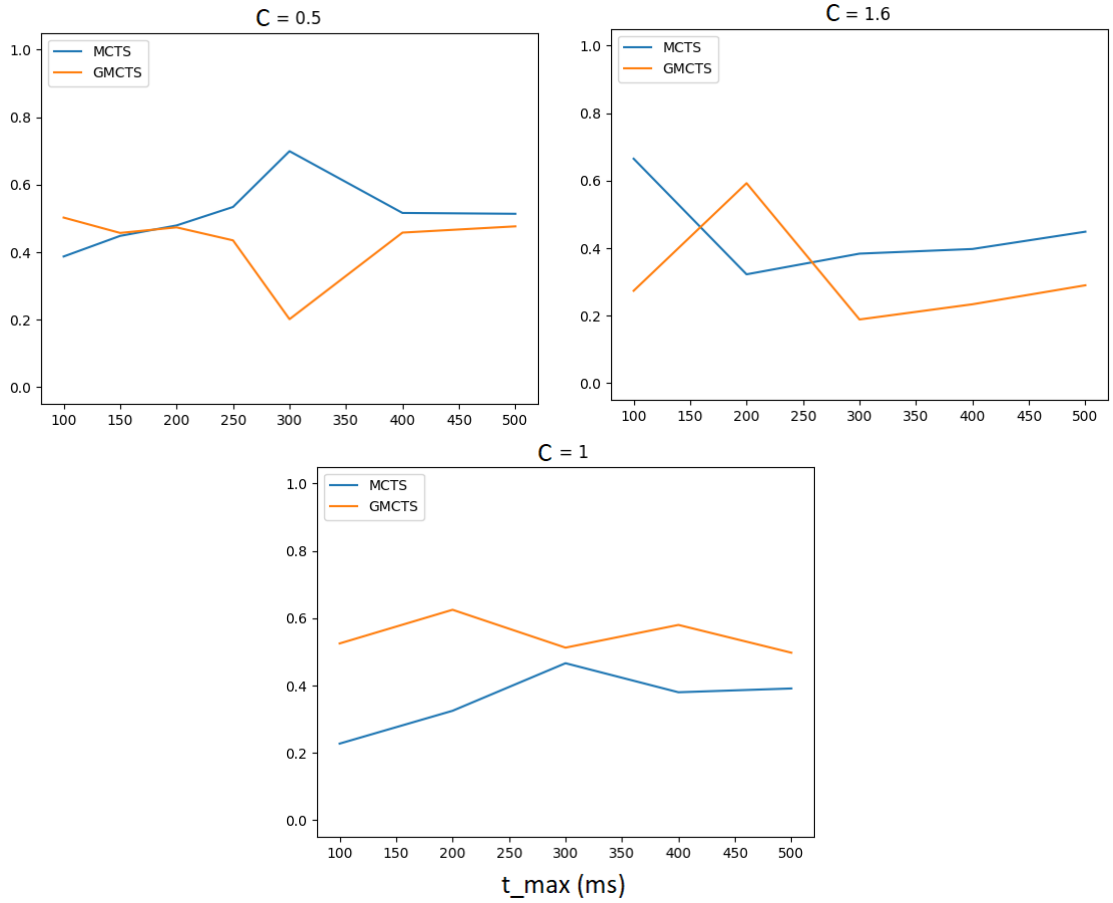

FIGURE 6 – Résultats des parties de G-MCTS (en orange) contre UCT (en bleu), pour différentes valeurs de C.

On remarque alors que pour $C = 1 < \sqrt{2}$, les performances de G-MCTS dépassent celles de UCT pour toutes valeurs de $t_{max} in \{100, 200, 300, 400, 500\}$. Une diminution de C signifie que l'algorithme explore moins les coups peut visités qui n'ont pas le meilleur score w_i/n_i. Une augmentation des performances de G-MCTS pour un C plus faible traduit le fait que les simulations étant plus précises que celle de UCT, il est moins nécessaire de visiter les nœuds pour avoir une bonne approximation de leur score. Toutefois, diminuer encore cette constante (par exemple, $C = 0.5$) diminue les performances de l'algorithme : dans ce cas, les nœuds dont les premières simulations sont mauvaises ne seront que rarement ré-explorés, et donc l'algorithme peut ne pas considérer un coup qui serait pourtant le meilleur.

4 Conclusion

L'algorithme de MCTS est aujourd'hui au cœur des travaux de recherche, notamment appliqué aux jeux, où il donne de meilleurs résultats que les algorithmes classiques pour certains

problèmes et a permis de nombreuses avancées.

Dans ce projet, nous avons exploré différentes manières d'utiliser une heuristique permettant d'évaluer les coups pour améliorer les performances de l'algorithme de MCTS. Tout d'abord, une première approche, S-MCTS, basée sur la fonction softmax pour générer des probabilités de sélection n'a pas donné de résultats positifs, notamment à cause du temps de calcul demandé pour calculé le résultat de softmax. Toutefois, les versions de S-MCTS sélectionnant plus souvent les coups désignés les meilleurs par la fonction d'évaluation obtenaient un score meilleur que les autres.

Nous avons ensuite crée une deuxième approche, G-MCTS, qui sélectionne les coups de manière gloutonne, en choisissant à chaque fois celui qui présente le meilleur score pour la fonction d'évaluation. Cette nouvelle version a obtenu de meilleurs résultats que MCTS simple pour les temps de calcul faible. De plus, après recherches autour de la constante définissant le taux d'exploration de l'algorithme a permis d'obtenir des résultats meilleurs que MCTS sur toute la gamme de temps de calcul étudiés.

Enfin, une piste d'amélioration possible dans la continuité de ce travail serait d'explorer les performances de G-MCTS dans d'autres situations, notamment avec des heuristiques d'évaluations moins fortes que celle utilisée dans ce projet (par exemple, ne permettant pas d'obtenir un algorithme de minimax performant), par exemple en l'appliquant au jeu de Go.

Références

[Auer et al., 2002] Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3) :235–256.

[Bertsimas et al., 2014] Bertsimas, D., Griffith, J. D., Gupta, V., Kochenderfer, M. J., Mišić, V. V., and Moss, R. (2014). A comparison of monte carlo tree search and mathematical optimization for large scale dynamic resource allocation. *arXiv preprint arXiv :1405.5498*.

[Browne et al., 2012] Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1) :1–43.

[Chaslot et al., 2010] Chaslot, G., Fiter, C., Hoock, J.-B., Rimmel, A., and Teytaud, O. (2010). Adding expert knowledge and exploration in monte-carlo tree search. In van den Herik, H. J. and Spronck, P., editors, *Advances in Computer Games*, pages 1–13, Berlin, Heidelberg. Springer Berlin Heidelberg.

[De Mesmay et al., 2009] De Mesmay, F., Rimmel, A., Voronenko, Y., and Püschel, M. (2009). Bandit-based optimization on graphs with application to library performance tuning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 729–736. ACM.

[Kocsis and Szepesvári, 2006] Kocsis, L. and Szepesvári, C. (2006). Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer.

[Shannon, 1950] Shannon, C. E. (1950). Programming a computer for playing chess. In *Computer chess compendium*, pages 2–13. Springer.

[Silver et al., 2016] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587) :484–489.

[Silver et al., 2017a] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T. P., Simonyan, K., and Hassabis, D. (2017a). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815.

[Silver et al., 2017b] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. (2017b). Mastering the game of go without human knowledge. *Nature*, 550(7676) :354.

A Heuristique d'évaluation du puissance 4 utilisée pour S-MCTS et G-MCTS

Son fonctionnement est le suivant :

- Si la partie est finie, le score du joueur ayant gagné est $+\infty$.
- Sinon, pour chaque groupement possible de quatre cellules, en ligne, en colonne ou en diagonale, on regarde :
  - Si des pions des deux joueurs sont présent, on passe.
  - Sinon, le score du joueur dont les pions sont présent est 10^n, où n est le nombre de pions du joueur sur les quatre cellules.

La valeur de l'heuristique, du point de vue du joueur j_1, est $\text{score}(j_1) - \text{score}(j_2)$, et *vice versa*.