

INITIATION À LA RECHERCHE



**PLANIFICATION D' ACTIONS
DANS UN MONDE CONTINU**

Léopold BELLEC, Pierre-Marie JUNGES

ENCADRANTS : *Olivier BUFFET, Vincent THOMAS*

Table des matières

1	Introduction au problème	2
1.1	Problématique	2
1.2	Pourquoi ce sujet ?	2
1.3	Notre approche	2
2	Le B.A.-BA de l'algorithme MCTS	3
2.1	Les bases du Monte Carlo Tree Search	3
2.1.1	Définition	3
2.1.2	Déroulement de l'algorithme	3
2.1.3	Pseudo-code	4
2.2	Formule de sélection utilisée	4
2.3	Stratégies de sélection terminale	5
2.4	Application sur un exemple	5
3	Application dans un monde continu	8
3.1	Abstrait	8
3.1.1	Définition	8
3.1.2	Explications du problème	8
3.1.3	Première approche	9
3.1.3.a	Modification du MCTS discret	9
3.1.4	Progressive widening	10
3.1.4.a	Principe	10
3.1.4.b	Pseudo-code	11
3.1.5	Double progressive widening	12
3.1.5.a	Principe	12
3.1.5.b	Pseudo-code	14
3.2	Expérimentations	15
3.2.1	MCTS discret dans un monde continu	15
3.2.2	Progressive widening	15
3.2.2.a	Sans ajout de bruit	15
3.2.2.b	Avec ajout de bruit	15
3.2.3	Double progressive widening	15
3.3	Conclusion	15

1 Introduction au problème

1.1 Problématique

Une des problématiques importantes en intelligence artificielle est la planification automatique d'actions, que ce soit pour la robotique, pour les jeux (échecs, go, ...), pour la gestion de sources d'énergie, ou pour la protection d'espèces menacées. La plupart des algorithmes développés n'abordent que des situations dans lesquelles le nombre d'états (le nombre de situations) possibles est fini et relativement petit, tout comme le nombre d'actions disponibles. Ces limitations viennent de ce que ces algorithmes raisonnent sur l'arbre des évolutions possible du "système" à contrôler, arbre dont la taille subit une explosion combinatoire avec les nombres d'états et d'actions (et qui n'a plus de sens quand les facteurs de branchement sont infinis).

1.2 Pourquoi ce sujet ?

Nous avons étudié lors de l'UE *Modèles de perception, de raisonnement et d'interaction* l'algorithme **Monte Carlo Tree Search** [Cou06] appliqué au jeu Puissance 4.

Or cette application était dans un espace où les états et actions possibles étaient finis, et au vue des parties effectuées contre cet algorithme il était difficile voir impossible de le battre.

C'est pourquoi, nous avons souhaité pousser un peu plus nos connaissances sur cet algorithme et surtout étudier ses performances dans un espace qui serait maintenant continu (une infinité d'actions ou états possibles) soit un espace proche de la réalité, proche du comportement humain.

De plus, ce sujet est peu étudié ce qui le rend encore plus intéressant.

1.3 Notre approche

Étant donné le nombre imposant d'états ou d'actions possibles à chaque étape, nous avons donc tout simplement essayer à chaque étape de réduire ce nombre de choix possible jusqu'à avoir un espace plus restreint [CHS⁺11].

C'est donc cette solution que nous avons choisie pour répondre à cette problématique.

Afin de pouvoir mettre en place l'algorithme, nous avons deux choix d'implémentations, utiliser le langage C ou bien le langage Java. Notre choix s'est finalement porté sur le langage Java.

2 Le B.A.-BA de l'algorithme MCTS

2.1 Les bases du Monte Carlo Tree Search

2.1.1 Définition

Monte Carlo Tree Search ou bien MCTS est un algorithme de recherche, qui, à partir d'un nœud initial n_i indique la meilleure action a_i à utiliser afin d'atteindre un nœud intermédiaire n_{int} proche du nœud terminal recherché n_t .

L'application de méthodes de Monte Carlo fait appel à de nombreuses simulations, cela veut donc dire que lors du déroulement de l'algorithme, un certain nombre d'actions seront jouées de manière aléatoire.

2.1.2 Déroulement de l'algorithme

L'algorithme est exécuté tant que le temps limite t n'est pas écoulé, et se décompose en 4 étapes :

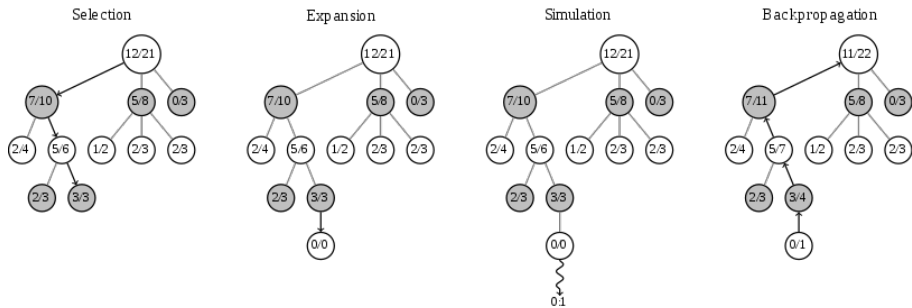


Figure 1 – Étapes de MCTS

Notation : Le nœud correspond à un état du jeu et (12/21) signifie 12 victoires sur 21 simulations.

1. Sélection

À cette étape, l'algorithme va sélectionner parmi ses nœuds suivants le nœud avec la plus grande valeur selon la formule de sélection utilisée 2.2.

Dans la figure 1, on s'aperçoit que l'algorithme sélectionne à tour de rôle le nœud (7/10), puis le nœud (5/6) et enfin le nœud (3/3) et s'arrête car ce nœud n'est pas développé.

2. Expansion

Une fois le meilleur nœud sélectionné, cette étape va lui rajouter un ou plusieurs nœud(s) enfant, accessible en faisant 1 seule action à partir du nœud courant.

Dans la figure 1, l'algorithme rajoute au nœud courant (3/3) le nœud enfant (0/0).

3. Simulation

Maintenant qu'un nouveau nœud a été ajouté, l'algorithme va jouer aléatoirement jusqu'à atteindre un nœud terminal. C'est dans cette partie qu'on utilise le principe Monte Carlo, c'est-à-dire l'utilisation de l'aléatoire.

4. Mise à jour

L'algorithme a atteint un nœud terminal, il faut donc récupérer sa récompense (par exemple +1 si c'est un nœud gagnant 0 sinon). Puis faire remonter cette récompense jusqu'au nœud racine et mettre à jour les statistiques de chacun des nœuds parcouru.

Dans la figure 1 ci-dessus le nœud terminal était donc perdant car (0/1), donc on va augmenter uniquement le nombre de simulation de chacun de ces nœuds, d'où la transformation du nœud (3/3) en (3/4) par exemple.

2.1.3 Pseudo-code

Algorithm 1 MCTS générique

```

1: function MCTS(État  $s$ )
2:   création d'un nœud  $n$  à partir de  $s$ 
3:   while  $temps < tempsLimite$  do                                ▷ tant qu'il reste du temps
4:      $n \leftarrow selection(n)$                                        ▷ L'étape d'expansion est incluse dans cet appel
5:      $n \leftarrow simulation(n)$ 
6:      $n \leftarrow miseAjour(n)$ 
7:   return  $meilleurEnfant(n)$ 

```

2.2 Formule de sélection utilisée

Précédemment, nous avons parlé lors de l'étape de sélection, de formule de sélection. En effet, afin de choisir le meilleur nœud possible des formules ont été mise en place.

C'est dans cette partie que nous allons en expliquer une, de nouvelles formules de sélection seront introduites dans la section suivante.

Notation : Soit n un nœud, N son nœud parent, et C une constante d'exploration.

1. Upper Confidence Bounds (UCB)

$$\frac{recompenses(n)}{simulations(n)} + C \times \sqrt{\frac{simulations(N)}{simulations(n)}}$$

On se rend compte ici que si notre constante d'exploration C est égale à 0 alors, l'algorithme va choisir le nœud n avec le meilleur rapport $\frac{recompenses(n)}{simulations(n)}$ possible.

Donc si l'on souhaite permettre une exploration de l'arbre, il faut que C soit différent de 0 sinon on voit bien que si à l'état initial l'algorithme sélectionne un nœud n , alors ce dernier aura nécessairement le meilleur rapport $\frac{recompenses(n)}{simulations(n)}$ vu qu'il est le seul à avoir été développé, et donc il sera tout au long de l'algorithme choisi.

Lexique : On utilise le terme UCT (*Upper Confidence Bound for Trees*) pour désigner l'algorithme MCTS utilisant une formule de sélection de type UCB.

2.3 Stratégies de sélection terminale

Les formules explicitées ci-dessus ont pour rôle de choisir le meilleur enfant lorsque l'algorithme se trouve à l'étape de sélection.

Dans cette partie, l'algorithme est terminé et l'arbre a été développé. Nous ne pouvons donc plus réutiliser les formules précédentes car la constante d'exploration C n'a plus de sens ici.

Il faut donc trouver d'autres moyens pour choisir le meilleur nœud enfant. Cette étape correspond à l'appel *meilleurEnfant(n)* dans l'algorithme 1. Or, nous savons qu'un nœud n possède un nombre de simulation, et de récompense. À partir de ces deux informations nous avons utilisé 3 stratégies de sélection :

1. Robuste :

On choisit le nœud avec le plus grand nombre de simulation.

2. Maxi :

On choisit le nœud qui possède la valeur de récompense la plus élevée.

3. Maxi-Robuste :

On choisit le nœud ayant le meilleur rapport $\frac{recompense}{simulation}$.

2.4 Application sur un exemple

Maintenant que nous avons vu comment l'algorithme fonctionnait de manière générale, nous l'avons mis en pratique sur le jeu du type Puissance 4.

Pour rappel voici la règle du jeu selon wikipedia¹ :

“Le but du jeu est d’aligner 4 pions sur une grille comptant 6 rangées et 7 colonnes. Chaque joueur dispose de 21 pions d’une couleur (par convention, en général jaune ou rouge). Tour à tour les deux joueurs placent un pion dans la colonne de leur choix, le pion coulisse alors jusqu’à la position la plus basse possible dans la dite colonne à la suite de quoi c’est à l’adversaire de jouer. Le vainqueur est le joueur qui réalise le premier un alignement (horizontal, vertical ou diagonal) d’au moins quatre pions de sa couleur. Si, alors que toutes les cases de la grille de jeu sont remplies, aucun des deux joueurs n’a réalisé un tel alignement, la partie est déclarée nulle.”

Après de nombreux essais, humain contre ordinateur, nous avons remarqué qu’il était très compliqué voir impossible de battre l’algorithme à partir d’un certain temps t , et sur un ordinateur avec un processeur type $i7$ ce temps est d’environ 2 secondes.

Afin de poursuivre un peu plus notre expérience, nous avons décidé de faire jouer 2 ordinateurs l’un contre l’autre, et, de cette façon, comparer les différentes stratégies de sélection finales et voir si effectivement certaines sont plus efficaces que d’autres.

		Ordinateur 2		
		Robuste	Maxi	Robuste-Maxi
Ordinateur 1	Robuste	70-30-0	60-40-0	60-40-0
	Maxi	80-20-0	80-20-0	50-50-0
	Maxi-Robuste	90-10-0	60-40-0	70-30-0

Table 1 – Comparatif des différentes stratégies

Méthodes expérimentales et explications de la table 1 :

L’ordinateur 1 commence **toujours** les parties, la notation $x - y - z$ signifie $x\%$ de victoire(s) pour l’ordinateur 1, $y\%$ de victoire(s) pour l’ordinateur 2, $z\%$ de match(s) nul(s).

Comme nous pouvons le remarquer, l’ordinateur commençant la partie à globalement plus de chance de gagner, peu importe la stratégie utilisée.

Réfléchissons sur la stratégie Maxi, dans le jeu Puissance 4 nous avons fixé la récompense en cas de victoire à 1 peu importe si la victoire était "serrée" ou bien "écrasante".

¹https://fr.wikipedia.org/wiki/Puissance_4

Or plaçons nous maintenant dans le cas où nous aurions fait des récompenses différentes selon le type de victoire, posons par exemple 1 par une victoire "serrée" et 10 pour une victoire "écrasante".

De ce fait, si nous avons un nœud avec 19 de récompense décomposé en 19 victoires dites "serrées" et 1 défaite, et un autre nœud avec 20 de récompense mais composée seulement de 2 victoires "écrasantes" et 18 défaites. Alors dans ce cas, la stratégie Maxi nous retournera le nœud avec 20 de récompense alors que nous aurions pu prendre le nœud qui a 19 victoires sur 20 simulations et qui est donc plus intéressant car son pourcentage de victoire est plus élevé.

Donc, pour pouvoir utiliser la stratégie Maxi de manière efficace, il est préférable de l'utiliser sur un jeu où l'on a une unique récompense en cas de victoire (ce qui est le cas ici).

Utilisons maintenant ce même raisonnement ainsi que ce même cas de figure pour la stratégie Robuste, nous avons donc 2 nœuds avec 20 simulations chacun.

Or on est censé choisir le nœud ayant le maximum de simulations possible, donc notre stratégie Robuste va tout simplement retourner le premier nœud lu car le deuxième nœud lu n'aura pas un nombre de simulation strictement supérieur au premier nœud. Alors, dans notre cas de figure, la stratégie Robuste aura une probabilité de $\frac{1}{2}$ de retourner le meilleur nœud (celui avec les 19 victoires).

Donc, de la même façon que la stratégie Maxi, la stratégie Robuste n'est pas nécessairement bonne pour choisir le meilleur nœud possible.

À partir des raisonnements ci-dessus il est donc facile de se convaincre que choisir la stratégie Maxi-Robuste est préférable, car le fait qu'elle prenne le meilleur rapport $\frac{\text{recompense}}{\text{simulation}}$ l'empêche donc d'être biaisé, car son choix dépend de 2 critères.

Maintenant que nous savons comment MCTS fonctionne dans un monde avec un nombre d'états et d'actions discrets, essayons de le mettre en pratique dans un monde à actions ou états continues.

3 Application dans un monde continu

3.1 Abstrait

3.1.1 Définition

Précédemment, nous avons vu quel était l'algorithme MCTS appliqué aux variables discrètes, c'est-à-dire des variables dont l'ensemble des valeurs possibles est fini, dénombrable. Maintenant comme le titre le laisse supposer, nous allons nous intéresser au monde continu.

De ce fait, nous allons rappeler ce qu'est une variable continue, et pour ce faire nous allons utiliser un exemple :

Supposons que nous avons une voiture, sa vitesse correspond donc à une variable continue car elle peut être de 40,00 km/h tout comme de 16,24575 km/h, il existe donc une infinité de valeurs réelles entre les intervalles 0 km/h et $Vitesse_{max}$ km/h.

Donc, dans cette partie, nos différentes variantes de MCTS vous travailler avec des variables de ce type.

3.1.2 Explications du problème

Voici une illustration du problème [CHS⁺11] que nous allons essayer de résoudre :

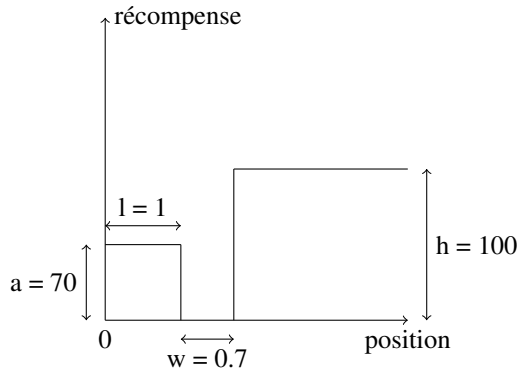


Figure 2 – Illustration du problème

Notre algorithme va partir de la position initiale 0, puis devra en 2 actions, maximiser les récompenses cumulées.

Pour ce faire la fonction récompense est définie ainsi :

$$recompense(x) = \begin{cases} a & \text{si } x \leq l \\ 0 & \text{si } l < x < l + w \\ h & \text{si } x \geq l + w \end{cases}$$

Sachant qu'une action $\in [0, 1]$, et que la récompense cumulée optimale est de 170 (décomposée en $70 + 100$). La principale difficulté ici réside dans le fait qu'il existe un "piège" illustré par la variable w située entre les positions 1 et 1.7.

Il va donc être intéressant de voir si l'algorithme préfère effectuer 2 actions dans l'intervalle $[0, 1]$ et donc obtenir en récompenses cumulées 140 ou bien s'il va essayer de franchir ce fameux piège est ainsi obtenir la récompense optimale.

3.1.3 Première approche

3.1.3.a Modification du MCTS discret

Étant donné que nous avons passé quelques temps sur le MCTS discret, nous avons essayé de ré-utiliser ce dernier.

Cependant, nous avons donc été confronté à une première problématique : "Comment faire fonctionner un problème ayant des variables continues dans un algorithme fait pour des problèmes discrets ?"

Et la réponse est simple, nous avons tout bonnement créé une entité permettant de faire la jonction entre le problème continu et notre MCTS discret.

Et ce type d'entité, en programmation logicielle, correspond à un design pattern *Adapter*.

Ci-dessous un diagramme de séquence simplifié 3, pour mieux comprendre, ce que donne ce système lorsque le MCTS va demander à un problème ses actions possibles.

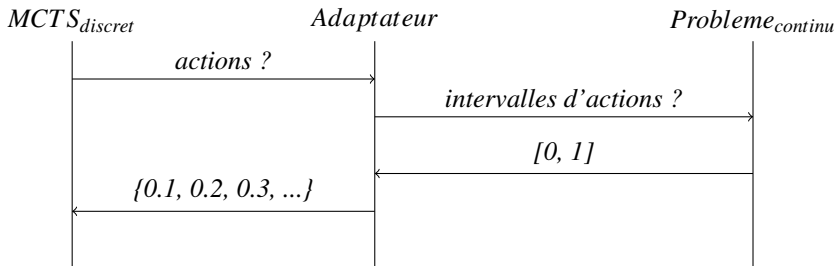


Figure 3 – Obtenir les actions possibles d'un problème continu dans le MCTS discret

Néanmoins, nous verrons dans la partie 3.2.1 que cette modification n'est pas aussi efficace que les deux prochains algorithmes.

3.1.4 Progressive widening

3.1.4.a Principe

Contrairement à la version précédente, cette amélioration permet de résoudre des problèmes dont le nombre d'actions est très grand voire infini sans utiliser d'adaptateurs ou autres artifices.

De plus, cet algorithme garde à peu près la même structure que l'algorithme générique MCTS 1, la principale différence est que les étapes *selection*(n) et *simulation*(n) seront regroupées en une seule étape.

Étant donné le fait que nous avons pour tout nœud n un nombre extrêmement grand d'actions possibles, cet algorithme va donc à chaque itération ajouter (ou non) à la liste des actions possibles d'un nœud courant, une ou plusieurs action(s) supplémentaire(s).

Afin de savoir si l'algorithme doit ajouter une ou plusieurs action(s), on va principalement utiliser le nombre de simulation, noté t , du nœud courant. Puis le principe sera le suivant : initialement, un nœud aura une liste d'actions possibles de taille égale à la valeur de la constante d'exploration C 2.2 utilisée, puis plus le nœud sera visité plus sa liste d'actions possibles augmentera.

Ce principe nous permet de développer de manière efficace les nœuds les plus visités, donc les nœuds les plus intéressants.

3.1.4.b Pseudo-code

Algorithm 2 Progressive widening (PW) appliqué à un État s avec la constante d'exploration $C > 0$ and $\alpha \in]0, 1[$.

```

1: function PW(État  $s$ )
2:    $nbSimulation(s) \leftarrow nbSimulation(s) + 1$ 
3:    $t \leftarrow nbSimulation(s)$ 
4:    $k \leftarrow \lceil C \times t^\alpha \rceil$ 
5:   // On va parcourir les actions possibles entre 0 et  $k$ 
6:   for  $i \in [0, k]$  do
7:     /* On récupère la récompense de  $s$  plus la récompense si on applique
       action $i$  à  $s$  */
8:      $recompense \leftarrow recompense(s) + recompense(s, action_i)$ 
9:     // On récupère le nombre de simulation où action $i$  a été appliqué à  $s$ 
10:     $nb \leftarrow nbSimulation(s, action_i)$ 
11:    if  $nb = 0$  then
12:      /* Si action $i$  fournit un État jamais exploré, alors on retourne ce
       dernier */
13:      return  $(s, action_i)$ 
14:    else
15:      // Sinon on calcul le score correspondant
16:       $score(i) \leftarrow \frac{recompense}{nb} + k_{ucb} \sqrt{\log(t)/(nb + 1)}$ 
17:      // La boucle for est terminée, on récupère l'indice ayant le score maximal
18:       $i_{max} \leftarrow indiceMaximisant(score)$ 
19:      return  $(s, action_{i_{max}})$ 

```

Même si nous avons explicité dans la partie 3.1.4.a quelle était la différence entre le MCTS générique 1 et le MCTS avec le progressive widening, voici tout de même le pseudo code de ce dernier 3.

Algorithm 3 MCTS avec le progressive widening

```
1: function MCTS(État  $s$ )
2:   while  $\text{temps} < \text{tempsLimite}$  do                                ▷ tant qu'il reste du temps
3:     while  $s$  n'est pas terminal do
4:        $s \leftarrow PW(s)$                                           ▷ On applique le progressive widening à  $s$ 
5:        $s \leftarrow \text{miseAJour}(s)$ 
6:     /* Lorsque le temps est écoulé, on applique la stratégie robuste pour déterminer le meilleur prochain État */
7:   return Robuste( $s$ )
```

3.1.5 Double progressive widening

3.1.5.a Principe

Cette seconde amélioration prend la même base que le progressive widening 3.1.4 mais permet, en plus de gérer un nombre d'action très grand, gérer le cas où, lorsque l'on a sélectionné la meilleure action possible, un bruit parasite se produit.

De ce fait, en un seul appel, la fonction double progressive widening va produire à partir d'un état s (potentiellement) 2 nœuds (pas de 2 fils).

Le premier nœud fils, noté s' correspondra à la meilleure action a appliqué à l'état s (même principe que PW); puis le second nœud noté s'' , un nœud fils du premier nœud, correspondra à un bruit parasite appliqué à s' . Dans notre cas, le bruit est toujours additif, néanmoins l'algorithme est censé marcher peu importe si le bruit est additif ou soustractif.

Afin de savoir si l'on doit ajouter un bruit, le double progressive widening va principalement utiliser le nombre de simulation ainsi que le nombre d'enfants de l'état s' .

Puis le principe sera le suivant: plus un état s' sera simulé, plus il aura de chance d'être bruité, donc d'avoir un nouvel enfant s'' . Néanmoins, à partir d'un certain seuil aucun enfant ne sera ajouté, et l'algorithme retournera alors un enfant tiré aléatoirement parmi les enfants de s' .

Donc, cet algorithme va nous permettre de gérer des situations dans lesquelles un événement tiers perturbe les résultats, type d'évènement que l'on retrouve très souvent dans la vie de tous les jours. Par exemple, en hiver avec les plaques de verglas, pour calculer les pas optimaux à faire afin de ne pas tomber où glisser hors du trottoir.

3.1.5.b Pseudo-code

Algorithm 4 Double progressive widening (PW) appliqué à un État s avec la constante d'exploration $C > 0$ and $\alpha \in]0, 1[$.

```

1: function DPW(État  $s$ )
2:    $nbSimulation(s) \leftarrow nbSimulation(s) + 1$ 
3:    $t \leftarrow nbSimulation(s)$ 
4:    $k \leftarrow \lceil C \times t^\alpha \rceil$ 
5:   // On va parcourir les actions possibles entre 0 et  $k$ 
6:   for  $i \in [0, k]$  do
7:     /* On récupère la récompense de  $s$  plus la récompense si on applique
       action $i$  à  $s$  */
8:      $recompense \leftarrow recompense(s) + recompense(s, action_i)$ 
9:     // On récupère le nombre de simulation où action $i$  a été appliqué à  $s$ 
10:     $nb \leftarrow nbSimulation(s, action_i)$ 
11:    if  $nb = 0$  then
12:      /* Si action $i$  fournit un État jamais exploré, alors on retourne ce
       dernier */
13:      return  $(s, action_i)$ 
14:    else
15:      // Sinon on calcul le score correspondant
16:       $score(i) \leftarrow \frac{recompense}{nb} + k_{ucb} \sqrt{\log(t)/(nb + 1)}$ 
17:      // La boucle for est terminée, on récupère l'indice ayant le score maximal
18:       $i_{max} \leftarrow indiceMaximisant(score)$ 
19:      // On récupère le meilleur enfant dans  $s'$ 
20:       $s' \leftarrow (s, action_{i_{max}})$ 
21:      // On récupère le nombre de simulation de  $s'$ 
22:       $nb \leftarrow nbSimulation(s')$ 
23:      // On récupère le nombre d'enfant de  $s'$ 
24:       $nbEnfant \leftarrow nbEnfant(s')$ 
25:       $k' \leftarrow \lceil C \times nb^\alpha \rceil$ 
26:      if  $k' > nbEnfant$  then
27:        // On bruite le nœud  $s'$ 
28:         $s'' \leftarrow bruitage(s')$ 
29:        if  $s'' \notin enfants(s')$  then
30:          // On ajoute  $s''$  comme enfant de  $s'$ 
31:           $enfants(s') \leftarrow enfants(s') \cup s''$ 
32:          return  $s''$ 
33:        else
34:          // On retourne directement  $s''$ 
35:          return  $s''$ 
36:      else
37:        // Sinon on retourne un enfant tiré aléatoirement dans  $s'$ 
38:        return  $enfantAleatoire(s')$ 

```

Et voici, l'algorithme MCTS avec la stratégie double progressive widening 5, cet algorithme reste structurellement très proche de l'algorithme MCTS avec progressive widening 3

Algorithm 5 MCTS avec le double progressive widening

```

1: function MCTS(État  $s$ )
2:   while  $temps < tempsLimite$  do                                ▷ tant qu'il reste du temps
3:     while  $s$  n'est pas terminal do
4:        $s \leftarrow DPW(s)$                                 ▷ On applique le double progressive widening à  $s$ 
5:        $s \leftarrow miseAJour(s)$ 
6:     /* Lorsque le temps est écoulé, on applique la stratégie robuste pour déterminer le meilleur prochain État */
7:   return Robuste( $s$ )

```

3.2 Expérimentations

3.2.1 MCTS discret dans un monde continu

3.2.2 Progressive widening

3.2.2.a Sans ajout de bruit

3.2.2.b Avec ajout de bruit

3.2.3 Double progressive widening

3.3 Conclusion

Références

- [CHS⁺11] A. Couetoux, J.-B. Hoock, N. Sokolovska, O. Teytaud, and N. Bonnard. Continuous upper confidence trees. In *Proc. of the 5th Int. Conf. on Learning and Intelligent Optimization (LION'11)*, 2011.
- [Cou06] R. Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *P. Ciancarini and H. J. van den Herik, editors, Proceedings of the 5th International Conference on Computers and Games*, 2006.