INITIATION À LA RECHERCHE

PLANIFICATION D'ACTIONS DANS UN MONDE CONTINU

Léopold BELLEC, Pierre-Marie JUNGES

ENCADRANTS : *Olivier BUFFET, Vincent THOMAS*

MASTER 1 INFORMATIQUE 2017

Table des matières

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 En quoi ce sujet est-il intéressant ?

La principale raison est que les algorithmes que nous allons étudier permettent de résoudre des problèmes proches de l'environnement réel.

En effet, ces algorithmes peuvent résoudre des problèmes auxquels nous sommes régulièrement confrontés, par exemple, lorsque l'on suit quelqu'un en marchant, inconsciemment, nous faisons en sorte que notre vitesse ne soit pas plus rapide (resp. lente) que la personne que nous suivons, nous devons donc accélérer ou réduire notre vitesse.

Et bien, c'est exactement le genre de problème que ces algorithmes peuvent résoudre.

De plus, une variante de cet algorithme permettra de gérer des cas où un événement tiers perturbe les résultats, si nous faisons encore le parallèle avec le monde réel, cet événement peut être un coup de vent ou bien une glissade.

C'est pourquoi, nous avons souhaité prendre ce sujet et pousser un peu plus nos connaissances sur les algorithmes de décisions.

1.3 Notre approche

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

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

Afin de pouvoir mettre en place nos algorithmes, nous avions deux choix d'implémentations, utiliser le langage C ou bien le langage Java. Notre choix s'est finalement porté sur le langage Java, car nous sommes plus habitués à ce dernier et nous trouvons la gestion de mémoire également plus simple.

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

Dans cette partie nous allons introduire les notions importantes relatives à l'algorithme **M**onte **C**arlo **T**ree **S**earch [Cou06] appliqué à un problème discret, c'est-à-dire un problème où le nombre d'états ou actions est fini.

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 joués 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 :

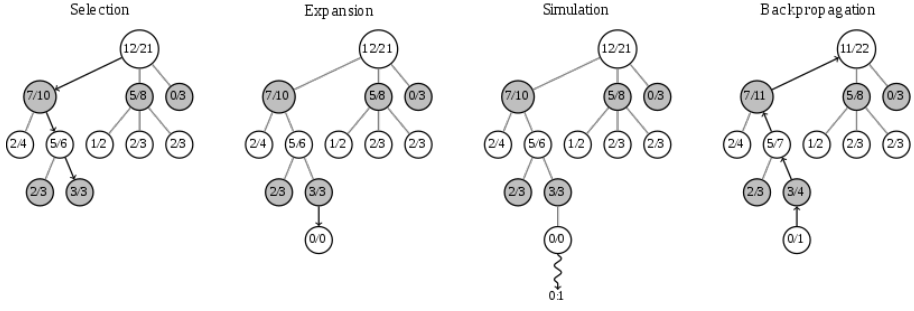


Figure 1 – Étapes de MCTS

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 dans la partie 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 parcourus.

   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 ← selection(n)        ▷ L'étape d'expansion est incluse dans cet appel
5:         n ← simulation(n)
6:         n ← miseAjour(n)
7:     return meilleurEnfant(n)
```

2.2 Formule de sélection utilisée

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

Et, c'est dans cette partie que nous allons en expliquer une (la principale utilisée), puis de nouvelles seront introduites dans la partie 3.

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

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 l'exploration de l'arbre, il faut que la constante C soit différente de 0 sinon on voit bien qu'à l'état initial l'algorithme sélectionne un nœud n, alors ce dernier aura nécessaire le meilleur rapport $\frac{recompenses(n)}{simulations(n)}$ vu qu'il est le seul à avoir été développé et, il sera tout au long de l'algorithme choisit.

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, mettons le en pratique sur le jeu Puissance 4.

Pour rappel voici la règle du jeu selon wikipedia[1] :

> *"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 finale et voir si effectivement certaines sont plus efficaces que d'autres.

		Ordinateur 2		
		Robuste	Maxi	Robuste-Maxi
Ordinateur 1	Robuste	55-45-0	55-45-0	60-40-0
	Maxi	50-50-0	60-40-0	50-50-0
	Maxi-Robuste	45-55-0	55-45-0	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

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

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".

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 avions 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 possibles, 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 peut être préférable dans certains cas, car le fait qu'elle prenne le meilleur rapport $\frac{recompense}{simulation}$ permet dans le cas où le nombre de simulation (resp. récompense) de tous les nœuds est identique d'arriver à retourner un "meilleur" nœud, car son choix dépend de 2 critères différents.

3 Application dans un monde continu

Nous avons vu comment fonctionne le MCTS dans un monde discret, étudions maintenant son fonctionnement dans un monde continu à travers 3 variantes différentes.

La première variante (partie 3.3) consiste à réutiliser le MCTS discret et à l'adapter au monde continu. La seconde variante (partie 3.4) permet de travailler directement avec des entités continues cependant la gestion d'un bruit parasite est effectuée de manière sommaire. Enfin, la dernière variante (partie 3.5) gère les entités continues ainsi que la gestion des bruits parasites de manière plus réfléchie.

3.1 Définition

À la différence d'une variable discrète, qui a donc un ensemble de valeurs possibles fini; une variable continue est une variable dont l'ensemble des valeurs possibles est infini.

Par 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 vont travailler avec des variables de ce type.

3.2 Explications du problème

Afin de mettre nos algorithmes en pratique, nous avons dans un premier temps décidé de nous inspirer du *Trap problem* de [CHS+11], illustré par la figure 2.

Puis dans un second temps, nous modifierons le problème initial en lui ajoutant une difficulté supplémentaire, problème illustré par la figure 5.

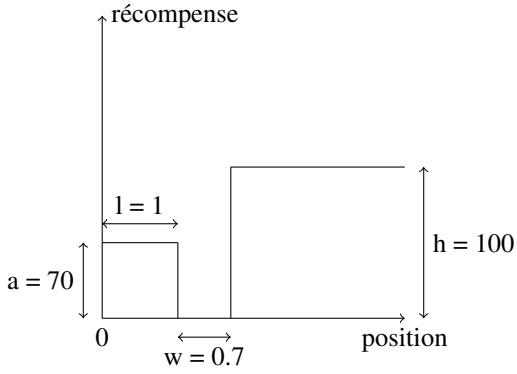


Figure 2 – Illustration du premier problème

Pour le premier problème, notre algorithme va partir de la position initiale 0 puis devra, en 2 pas, maximiser les récompenses cumulées. La fonction de récompense utilisée est illustrée par la figure 3 et un pas correspond à une action.

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

Figure 3 – Fonction de récompense pour une position x donnée.

Une action est un nombre réel $\in [0,1]$ et, la récompense cumulée optimale pour ce premier problème est de 170 (décomposée en $70 + 100$).

La principale difficulté réside dans le fait qu'il existe un "piège" illustré par la variable w située entre les positions 1 et 1.7. Il s'agit bien d'un "piège" car nous pouvons ajouter un bruit à chaque action effectuée.

Le bruit est exprimé de la manière suivante : $R \times Y$, avec $Y \in [0,1]$ une variable tirée aléatoirement suivant la loi uniforme (voir figure 6a) et $R > 0$ la valeur maximale que nous accordons au bruit.

Donc si un de nos algorithmes décide par exemple d'atteindre la position 0.99 et qu'à ce moment un bruit > 0.01 se produit alors notre position sera à > 1.0 et donc notre récompense, pour cette actions, passera de 70 à 0; l'algorithme sera donc tombé dans le piège.

Cependant pour atteindre la récompense optimale, il faut arriver à accéder à une position $>= 1.7$, or notre valeur d'action maximale est potentiellement de 1, donc il faut que nos algorithmes arrivent à atteindre une position après le premier pas d'au

moins 0.7. 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 et ainsi obtenir la récompense optimale.

Pour notre deuxième problème, voir figure 5, nous avons décidé de garder le même intervalle pour les actions, néanmoins cette fois-ci nous avons à disposition 3 pas et, notre nouvelle fonction de récompense correspond à la figure 4.

$$recompense(x) = \begin{cases} a & \text{si } x <= l \\ 0 & \text{si } l < x < l+w \text{ ou } l+w+b < x < l+w+b+k \\ r & \text{si } l+w <= x <= l+w+b \\ h & \text{si } x >= l+w+b+k \end{cases}$$

Figure 4 – Fonction de récompense pour une position x donnée.

Cette variante est d'autant plus intéressante car, cette fois-ci, deux pièges se trouvent entre la position initiale et la zone de récompense maximale. Donc il va être intéressant de voir si l'algorithme décide ou non d'atteindre la zone de récompense maximale, quitte à se risquer à être piégé à deux reprises. Ce problème a une récompense cumulée maximale de 270 (décomposée en $70+30+170$).

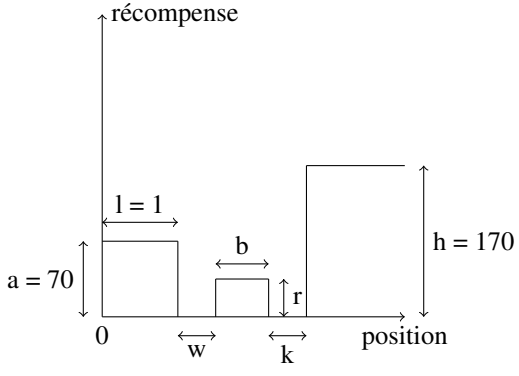


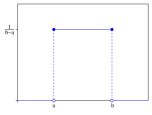
Figure 5 – Illustration du second problème, les valeurs que nous avons prises sont les suivantes: w = k = 0.5, r = 30, b = 0.7.

Avant de débuter sur les explications de nos algorithmes, nous avons jugé important d'expliquer notre choix pour la distribution des valeurs des actions. Pour rappel,

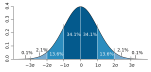
la valeur d'une action $\in [0, 1]$. À partir de cela, 2 choix s'offre à nous afin de générer cette valeur aléatoirement; utiliser un générateur suivant la loi uniforme (figure 6a), ou bien la loi normale (figure 6b).

Notre choix s'est finalement porté vers la loi normale, car nous avons jugé son comportement plus réaliste par rapport à notre problème.

En effet, si nous prenons l'exemple d'un humain qui marche, nos pas ont approximativement la même valeur, autrement dit, nous avons plus de chance de faire un pas proche d'une certaine moyenne que d'enjamber quelque chose ou bien de faire des "micros" pas. C'est donc pour ce souci de réalisme que nous avons préféré la loi normale.



(a) loi uniforme



(b) loi normale

Figure 6 – Exemple de distribution de valeurs

3.3 Première approche, modification du MCTS discret

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

Cependant, nous avons donc été confrontés 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 qui, en programmation logicielle, correspond à un design pattern *Adapter*.

Ci-dessous un diagramme de séquence simplifié (figure 7), pour mieux comprendre, ce que donne ce système lorsque le MCTS va demander à un problème ses actions possibles.

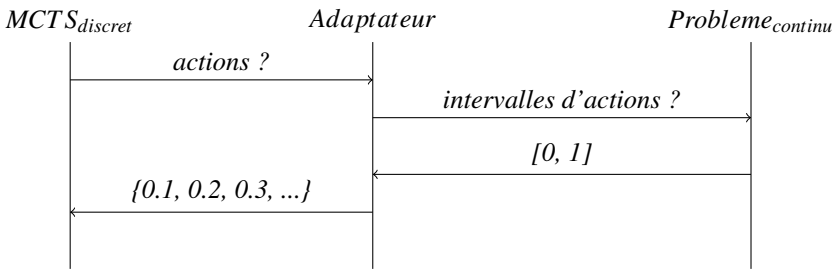


Figure 7 – Obtenir les actions possibles d'un problème continu dans le MCTS discret

À la différence des 2 prochaines variantes, cet algorithme ne nous permet pas de gérer les bruits parasites.

Nous verrons dans la partie 3.6 que cette modification du MCTS discret est quasiment efficace que le progressive widening sans bruitage néanmoins, pour obtenir des résultats convaincants, nous devons modifier à la main certains paramètres, chose que fait dynamiquement le progressive widening.

3.4 Progressive widening

3.4.1 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 (algorithme 1), la principale différence est que les étapes $selection(n)$ et $simulation(n)$ sont regroupées en une seule étape, voir algorithme 3. Ces deux étapes sont maintenant réalisées par l'algorithme progressive widening, algorithme 2

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) puis créera au fur et à mesure un nouveau nœud pour chaque actions non développées.

Afin de savoir si l'algorithme doit ajouter une ou plusieurs action(s), celui-ci va se baser sur la formule Ct^{α} avec t le nombre de simulation et C une constante d'exploration. Cette formule indiquera à l'algorithme s'il doit rajouter des actions possibles à sa liste d'actions ou non. Pour cela, le facteur d'accroissement de cette liste est donné par la variable α qui est comprise dans l'intervalle $]0,1[$.

D'après la figure 8, plus cette valeur sera proche de 0, moins l'algorithme créera de nouvelles actions possibles et donc favorisera la simulation des actions déjà existantes,

on dira que l'algorithme favorise l'exploitation, tandis que plus α sera proche de 1, plus celui-ci aura tendance à créer de nouvelles actions, on dira que l'algorithme favorise l'exploration, mais en contrepartie, il délaissera la simulation des actions déjà existantes. Dans le cas où $\alpha = 0.5$, l'algorithme aura tendance à ajouter des nouvelles actions vers le début et à diminuer cet ajout au fur et à mesure du temps. Il est donc intéressant de prendre un α proche de 0.5 pour trouver un bon équilibre entre l'ajout de nouvelles actions possibles et la simulation des actions déjà existantes.

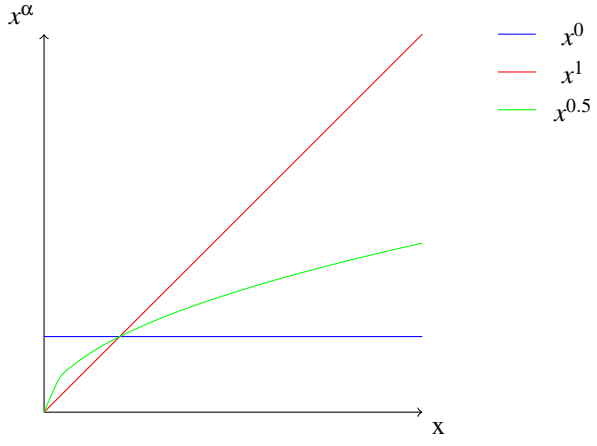


Figure 8 – Variations de x^α

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

Comme annoncé en introduction, cet algorithme permet de gérer également un bruit parasite, cette action peut s'effectuer à la ligne 19 de l'algorithme 2. Donc à chaque action choisit par l'algorithme, un bruit peut est ajouté.

Dans la partie 3.6, nous avons effectué des tests en bruitant l'action choisi afin de comparer ces performances avec le double progressive widening 3.5 puis nous avons ré-effectuer ces tests sans bruiter les actions.

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

function PW(État s)
  $nbSimulation(s) \leftarrow nbSimulation(s) + 1$
  $t \leftarrow nbSimulation(s)$
  $k \leftarrow \lceil C \times t^{\alpha} \rceil$
  // On va parcourir les actions possibles entre 0 et k
  for $i \in [0,k]$ **do**
    / On récupère la récompense de s plus la récompense si on applique $action_i$ à s */*
    $recompense \leftarrow recompense(s) + recompense(s, action_i)$
    // On récupère le nombre de simulation où $action_i$ a été appliqué à s
    $nb \leftarrow nbSimulation(s, action_i)$
    if $nb = 0$ **then**
      / Si $action_i$ fournit un État jamais exploré, alors on retourne ce dernier */*
      return $(s, action_i)$
    else
      // Sinon on calcul le score correspondant
      $score(i) \leftarrow \frac{recompense}{nb} + k_{ucb}\sqrt{\log(t)/(nb+1)}$
  // La boucle for est terminée, on recupère l'indice ayant le score maximal
  $i_{max} \leftarrow indiceMaximisant(score)$
  return $(s, action_{i_{max}})$ *// bruitage possible ici*

Algorithm 3 MCTS avec le progressive widening

function MCTS(État s)
  while $temps < tempsLimite$ **do** ▷ tant qu'il reste du temps
    while s n'est pas terminal **do**
      $s \leftarrow PW(s)$ ▷ On applique le progressive widening à s
    $s \leftarrow miseAjour(s)$
  / Lorsque le temps est écoulé, on applique la stratégie robuste pour déterminer le meilleur prochain État */*
  return Robuste(s)

3.5 Double progressive widening

Cette troisième variante, voir algorithme 4, utilise la même logique que le progressive widening (section 3.4).

Cependant, au lieu d'ajouter, à chaque appel, un bruit à une action comme on peut le faire dans le progressive widening, cette méthode utilise la formule présente à la ligne 22 de l'algorithme 4, or nous avons expliqué précédemment (figure 8) comment se comporte une formule de ce type.

Donc dans cet algorithme l'utilisation de cette formule nous permet de faire en sorte que, plus un nœud est simulé (donc s'il est souvent simulé, c'est que l'algorithme le trouve intéressant) plus il aura alors de chance d'être bruité.

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

Le premier nœud fils, noté s', correspondra à la meilleure action a appliqué à l'état s (même principe que le progressive widening); puis le second nœud noté s'', un nœud fils de s', 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.

Donc, cet algorithme va nous permettre de gérer des situations dans lesquelles un événement tiers perturbe les résultats, type d'événement que l'on retrouve très souvent dans la vie de tous les jours. Par exemple, en hiver avec les plaques de verglas, il est possible de glisser légèrement sur l'une d'elles et donc de faire un pas plus grand que prévu, le bruit est ici la légère glissade.

De plus, contrairement au progressive widening, cette variante développe de manière progressive le développement des actions à partir d'un nœud donné et aussi le développement des bruits parasites, d'où son nom "double élargissement progressif".

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

function DPW(État s)
  $nbSimulation(s) \leftarrow nbSimulation(s) + 1$
  $t \leftarrow nbSimulation(s)$
  $k \leftarrow \lceil C \times t^{\alpha} \rceil$
  // On va parcourir les actions possibles entre 0 et k
  for $i \in [0,k]$ **do**
    / On récupère la récompense de s plus la récompense si on applique $action_i$ à s */*
    $recompense \leftarrow recompense(s) + recompense(s, action_i)$
    // On récupère le nombre de simulation où $action_i$ a été appliqué à s
    $nb \leftarrow nbSimulation(s, action_i)$
    if $nb = 0$ **then**
      // Si $action_i$ renvoit un état jamais exploré, alors on retourne cet état
      return $(s, action_i)$
    else
      // Sinon on calcul le score correspondant
      $score(i) \leftarrow \frac{recompense}{nb} + k_{ucb}\sqrt{\log(t)/(nb+1)}$
  // La boucle for est terminée, on recupère l'indice ayant le score maximal
  $i_{max} \leftarrow indiceMaximisant(score)$
  $s' \leftarrow (s, action_{i_{max}})$ ▷ On récupère le meilleur enfant dans s'
  $nb \leftarrow nbSimulation(s')$
  $nbEnfant \leftarrow nbEnfant(s')$
  $k' \leftarrow \lceil C \times nb^{\alpha} \rceil$
  if $k' > nbEnfant$ **then**
    $s'' \leftarrow bruitage(s')$ ▷ On applique un bruit à s'
    if $s'' \notin enfants(s')$ **then**
      $enfants(s') \leftarrow enfants(s') \cup s''$ ▷ On ajoute s" comme enfant de s'
      return s'' ▷ On retourne l'enfant bruité
    else
      return s'' ▷ On retourne l'enfant bruité
  else
    // Sinon on retourne un enfant tiré aléatoirement dans s'
    return $enfantAleatoire(s')$

L'algorithme 5 correspond à l'algorithme MCTS où la stratégie double progressive widening est appliquée.

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 ← DPW(s)          ▷ On applique le double progressive widening à s
5:         s ← miseAjour(s)
6:     /* Lorsque le temps est écoulé, on applique la stratégie robuste pour déter-
       miner le meilleur prochain État */
7:     return Robuste(s)
```

3.6 Expérimentations

Dans cette partie, nous allons comparer les différents algorithmes décrit dans la partie 3 au problème (partie 3.2, figure 2), puis pour valider les performances de nos algorithmes, nous testerons uniquement le progressive widening ainsi que le double progressive widening sur le problème 2 (partie 3.2, figure 5) jugé plus complexe dans la partie 3.6.3.

Dans la partie 3.6.1, nous allons comparer les performances entre l'algorithme MCTS décrit à la partie 3.3 et le progressive widening sans bruitage (partie 3.4).

Puis nous comparerons dans la partie 3.6.2, les performances du double progressive widening (partie 3.5) et du progressive widening (partie 3.4) avec bruitage.

Dans tous les tests, la formule de sélection finale est la stratégie robuste (voir partie 2.3) et la valeur de R du bruit a été fixé à 0.05.

Ces tests ont été effectués sur un ordinateur avec un processeur i7 avec 16 Go de RAM.

3.6.1 Expérimentations sur le MCTS discret et le progressive widening sans bruitage

Notre première expérimentation va consister tout simplement à exécuter 500 fois ces 2 algorithmes avec la même valeur de temps, soit 10ms, sur le problème 1 (figure 2). D'après notre compréhension de ces 2 algorithmes, nous devrions obtenir des résultats quasiment identiques.

Étant donné que ces 2 algorithmes ne sont pas strictement similaires, nous sommes obligés de faire varier une variable afin d'essayer de rendre leurs 2 comportements plus proches :

- Pour l'algorithme MCTS, la variable à faire varier est la valeur de l'échantillonnage. En effet, cette valeur correspond au nombre d'actions maximales que nous accordons à un nœud donné, par exemple si nous fixons cette valeur à 10, cela signifie donc qu'un nœud pourra avoir au maximum 10 actions différentes, i.e 10 nœuds fils différents.

- Pour le progressive widening, la variable à faire fluctuer est la constante d'exploration C car c'est elle qui va indiquer à l'algorithme combien d'échantillonnage il s'autorise à prendre, voir notre explication dans la partie 3.4.

Suite à ces 500 simulations, nous obtenons les figures 9 et 10, et il est intéressant de remarquer que la moyenne des récompenses augmente légèrement voir stagne jusqu'à un seuil (différents pour les deux algorithmes) puis fait chuter de manière brutale la moyenne des récompenses, la faisant ainsi passer pour l'algorithme MCTS de 152 à 117, et de 170 à 115 pour le progressive widening.

Ce phénomène vient du fait que les algorithmes développent beaucoup de nouveaux nœuds, et passent donc peu de temps à les exploiter. On a donc trop favorisé l'exploration de l'arbre au dépit de l'exploitation. Donc dans ce cas, favoriser l'exploration entraîne la détérioration des résultats.

De plus, notre formule de sélection finale est la stratégie robuste, or cette stratégie est censée nous retourner le nœud enfant possédant le plus de simulation, donc si nos algorithmes ne font qu'explorer l'arbre, nous pouvons supposer que la valeur de simulation de la plupart des nœuds créés est proche de 1. Donc il est tout à fait possible que cette stratégie nous retourne dans ce cas un nœud ayant une très mauvaise valeur de récompense (resp. très bonne).

Nous pouvons également remarquer grâce à la table 2 que le progressive widening développe nettement moins de nœud par rapport à l'algorithme MCTS. Ce gain lui permet d'être en plus de performant, économe en ressources.

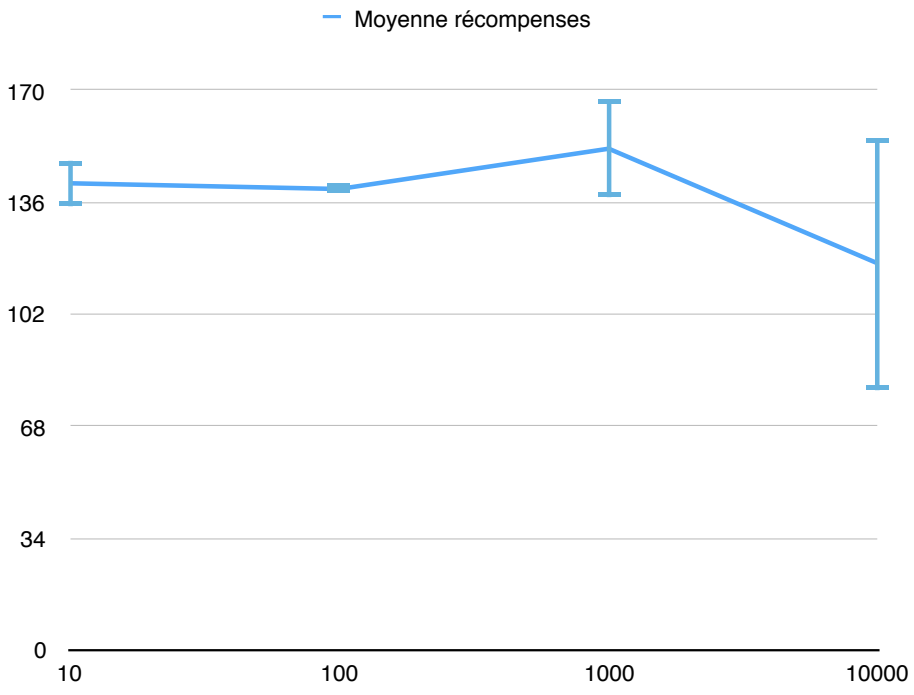


Figure 9 – Variations de l'adaptateur MCTS discret en fonction du nombre d'expérimentations, $ErreurStandard_{MCTSdiscret} = [0.32; 0.06; 0.66; 1.72]$

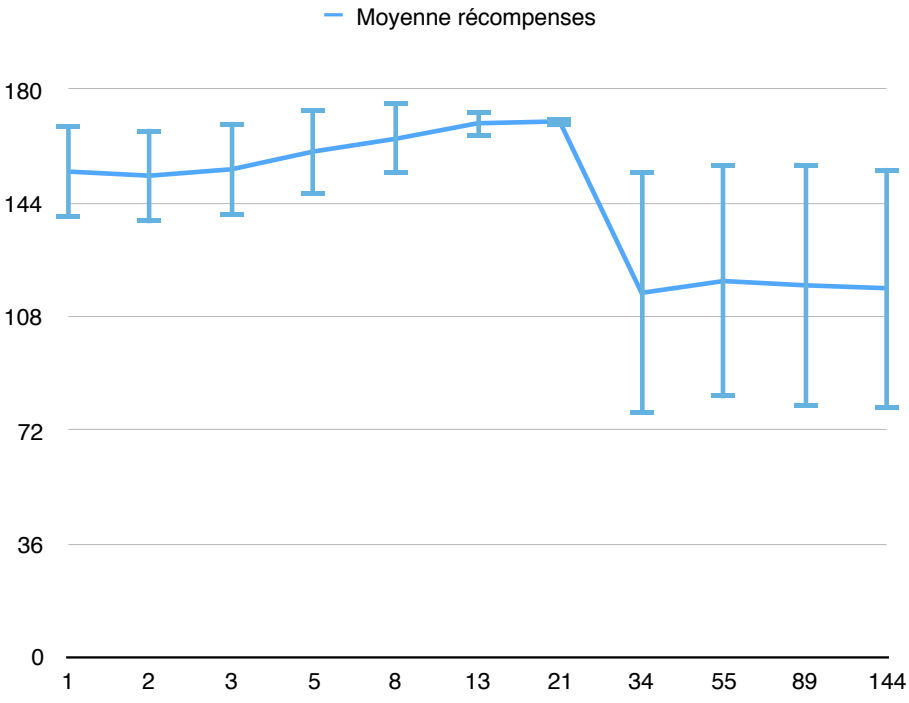


Figure 10 – Variations de PW non bruité en fonction de C, $ErreurStandard_{PW} = [0.67; 0.66; 0.67; 0.63; 0.52; 0.20; 0.06; 1.72; 1.66; 1.72; 1.71]$

C	1	2	3	5	8	10	21	34	55	89	144
Échantillonnage min	31	59	71	95	117	151	195	209	149	133	105
Échantillonnage max	41	61	76	100	126	160	207	255	259	263	259

Table 2 – Intervalles des échantillons obtenus en faisant variant C pour le PW non bruité

3.6.2 Expérimentations sur le progressive widening avec bruitage et le double progressive widening

Cette seconde expérimentation vise donc à comparer deux algorithmes permettant la gestion de bruits parasites. Il est donc inutile ici de les comparer à notre précédent algorithme MCTS et, comme dit dans la partie précédente (3.6.1), le progressive widening est bien plus efficace.

Lorsque nous faisons varier la constante d'exploration C, les résultats obtenus (Figure 11) montre globalement que l'algorithme double progressive widening a toujours une valeur moyenne de récompense supérieure à celle de l'algorithme progressive widening.

Il est aussi intéressant de remarquer que, comme dans la partie 3.6.1, à partir de la valeur $C = 21$ les performances de l'algorithme double progressive widening chute, passant ainsi de 161 à 120 de moyenne. Alors que l'algorithme progressive widening reste stable sur ce domaine.

De plus, nous pouvons remarquer que les écart-types des récompenses moyennes sont beaucoup plus faibles (pour $C <= 21$) pour le double progressive widening par rapport à ceux du progressive widening. Cela indique une plus grande précision dans les résultats.

Néanmoins, lorsque $C > 21$, les valeurs d'écart-types du double progressive widening deviennent plus importantes, nous pouvons expliquer cela par le fait que l'algorithme tombe plus souvent dans le piège du problème.

En effet, pour rappel si notre algorithme tombe dans le piège sa récompense sera de 70; alors que s'il arrive à accéder à la plate-forme finale sa récompense sera de 170. De ce fait si nos résultats finaux alternent énormément entre 70 et 170, notre moyenne va tendre vers 120 et notre écart-type sera également grand.

Donc le comportement observé pour le double progressive widening à partir de $C > 21$ est explicable.

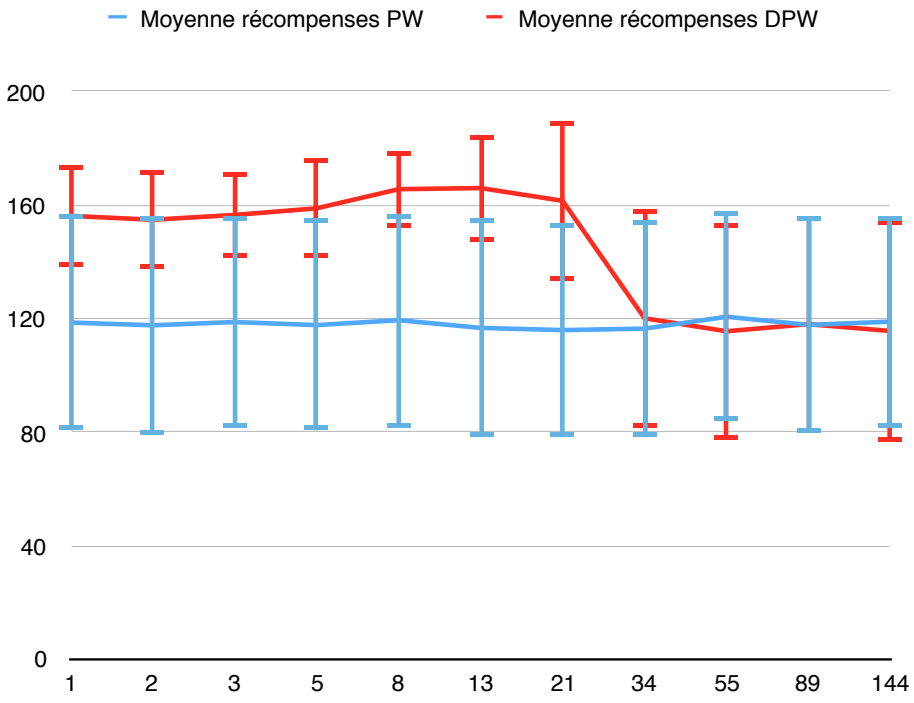


Figure 11 – Variations des moyennes de récompenses en fonction de C pour PW et DPW avec $\alpha = 0.4$ et le $temps = 10$, $ErreurStandard_{PWbruit} = [1.71; 1.74; 1.66; 1.67; 1.69; 1.73; 1.69; 1.70; 1.64; 1.71; 1.66]$, $ErreurStandard_{DPW} = [0.80; 0.79; 0.67; 0.77; 0.60; 0.84; 1.24; 1.72; 1.69; 1.71; 1.74]$

Posons $C = 8$ et faisons maintenant varier la variable α, nous pouvons remarquer sur la figure 12 que le même phénomène qu'énoncé précédemment se produit, c'est-à-dire qu'à partir d'une certaine valeur (liée à α), l'efficacité du double progressive widening va grandement chuter tandis que le progressive widening va stagner au niveau des récompenses mais globalement, les performances du double progressive widening sont toujours supérieures à celle du progressive widening simple.

Expliquons maintenant pourquoi la valeur d'α peut causer des problèmes de performances. Pour rappel, dans le double progressive widening nous utilisons à 2 reprises la variable α (voir algorithme 4), la première occurrence d'$alpha$ est : $C \times t^{\alpha}$ avec t le nombre de simulation d'un nœud n.

Donc si $alpha$ est égal à 0 (cas qui n'est pas admis par l'algorithme, mais intéressant à observer), alors le nœud n pourra avoir uniquement C actions possibles, car peu importe la valeur de t, celle-ci à la puissance 0 donnera tout le temps 1. De même, le

double progressive widening ne pourra ajouter qu'au maximum C bruits à chacun de ces nœuds enfants. Alors, comment expliquer que l'algorithme arrive tout de même à avoir une récompense respectable ? La réponse est simple, nos valeurs de pas suivent la loi normale.

Intéressons-nous maintenant au cas où $\alpha = 1$ (cas également non admis), alors l'algorithme à chaque nouvel appel pour un noeud n va lui indiquer qu'il a de nouvelles actions disponibles, car la formule étant : $C \times t^{\alpha}$ avec t le nombre de simulation d'un nœud, si $\alpha = 1$ on se retrouve alors avec $C \times t$ et sachant que t croît de manière linéaire, on ajoute donc C actions possibles à chaque appel. Et donc, notre algorithme passe beaucoup de temps à explorer l'arbre au lieu de l'exploiter.

D'où l'important pour la suite, de choisir une valeur intermédiaire de α, proche de 0.5.

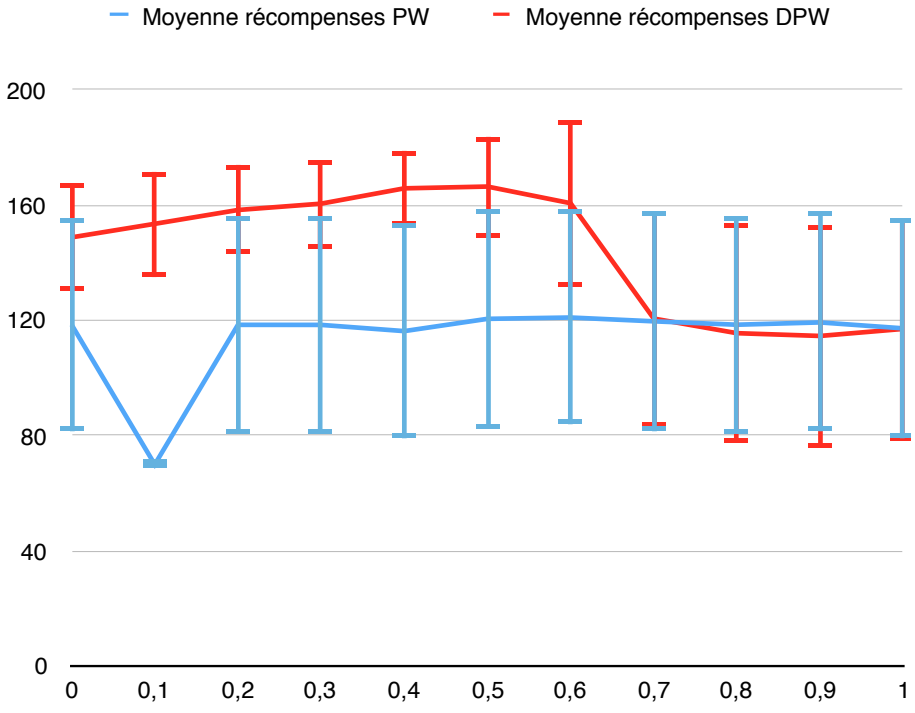


Figure 12 – Variations des moyennes de récompenses en fonction d'alpha pour PW et DPW avec $C = 8$ et le $temps = 10$, $ErreurStandard_{PWbruit} = [1.66; 0; 1.69; 1.67; 1.68; 1.71; 1.66; 1.71; 1.70; 1.70; 1.72]$, $ErreurStandard_{DPW} = [0.84; 0.82; 0.70; 0.69; 0.57; 0.78; 1.29; 1.67; 1.69; 1.72; 1.72]$

Maintenant que nous connaissons les conséquences liées aux variables C et α, faisons varier le temps de réflexion de l'ordinateur en fixant cette fois-ci C à 8 et α à 0.6. Nous remarquons grâce à la figure 13, que si le temps de réflexion n'est pas assez grand, le double progressive widening aura les mêmes résultats en moyenne que son homologue progressive widening mais plus le temps croît plus le double progressive widening aura de meilleurs récompenses en moyenne que le progressive widening mais, une fois la valeur de temps $T = 8$ms les moyennes des deux algorithmes commencent à stagner.

Donc la figure 13 nous permet de savoir qu'il n'est pas utile de laisser réfléchir trop longtemps ces deux algorithmes.

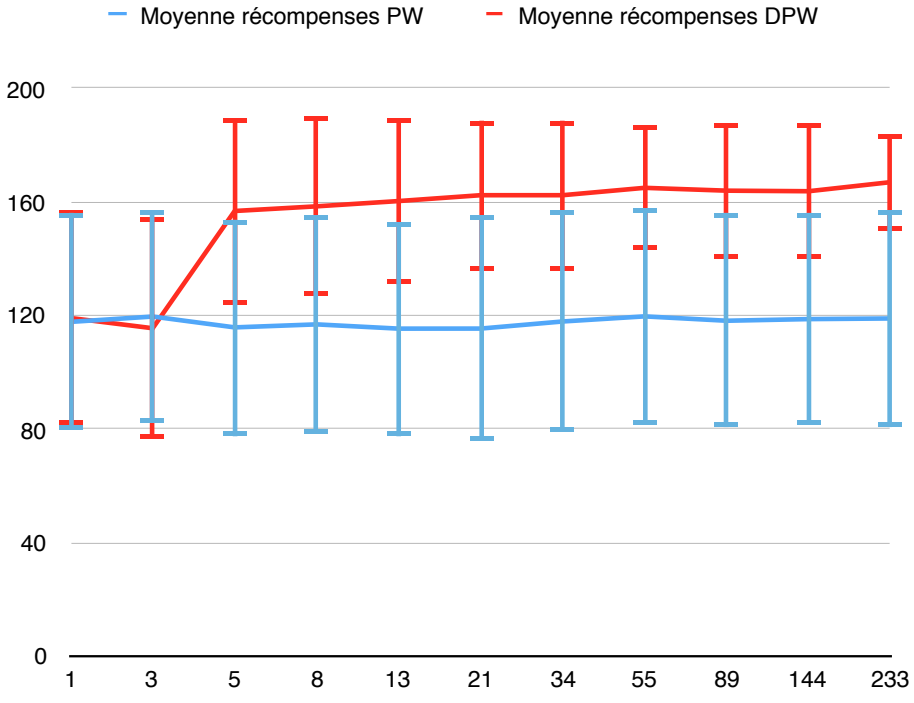


Figure 13 – Variations des moyennes de récompenses en fonction du temps (en ms) pour PW et DPW avec $C = 8$ et $\alpha = 0.6$, , $ErreurStandard_{PWbruit} = [1.71; 1.68; 1.70; 1.74; 1.69; 1.78; 1.73; 1.71; 1.69; 1.67; 1.71]$, $ErreurStandard_{DPW} = [1.68; 1.74; 1.47; 1.42; 1.32; 1.19; 1.19; 0.98; 1.06; 1.08; 0.76]$

Suite à tous ces tests, nous en avons déduit qu'une bonne combinaison de valeurs était $C = 8$, $\alpha = 0.6$ et $T = 34$ms.

La figure 14 montre la répartition des récompenses en pourcentage.

Nous pouvons clairement observer que le double progressive widening est bien meilleur que le progressive widening. En effet, celui-ci atteint à plus de 90% la récompense optimale contre seulement 15% pour le simple progressive widening.

Malgré les 10% de score à 70 i.e le double progressive widening tombe 1 fois sur 10 dans le piège. Il est tout de même préférable de choisir cet algorithme dans le cas où nous avons un problème dans un monde continu où un bruit parasite se produit.

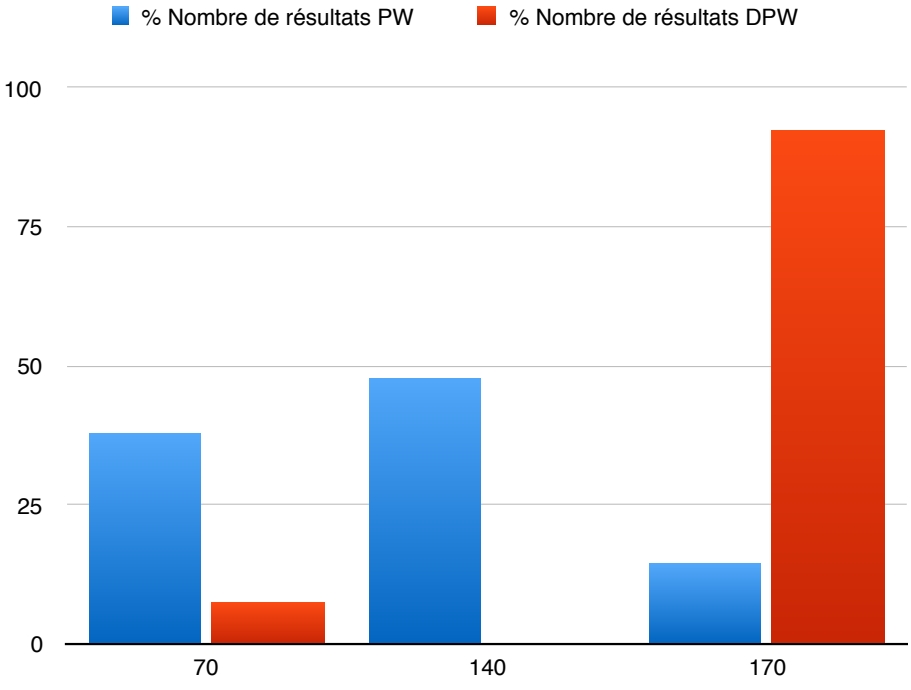


Figure 14 – Pourcentage de score obtenu pour PW et DPW avec $C = 8$, $\alpha = 0.6$ et $T = 34$

3.6.3 Expérimentation sur le problème 2

Dans cette partie nous avons juste voulu vérifier si le problème (figure 5) contenant cette fois-ci 2 pièges était encore réalisable par les algorithmes progressive widening et double progressive widening.

Les résultats obtenus sur la figure 15 confirme la supériorité du double progressive widening face au progressive widening dans un monde continu stochastique. En effet, dans ce problème la récompense optimale est de 270 et, comme nous pouvons

l'observer, l'algorithme double progressive widening arrive à atteindre cette récompense plus de 95% des fois, soit environ 20% de plus que le progressive widening.

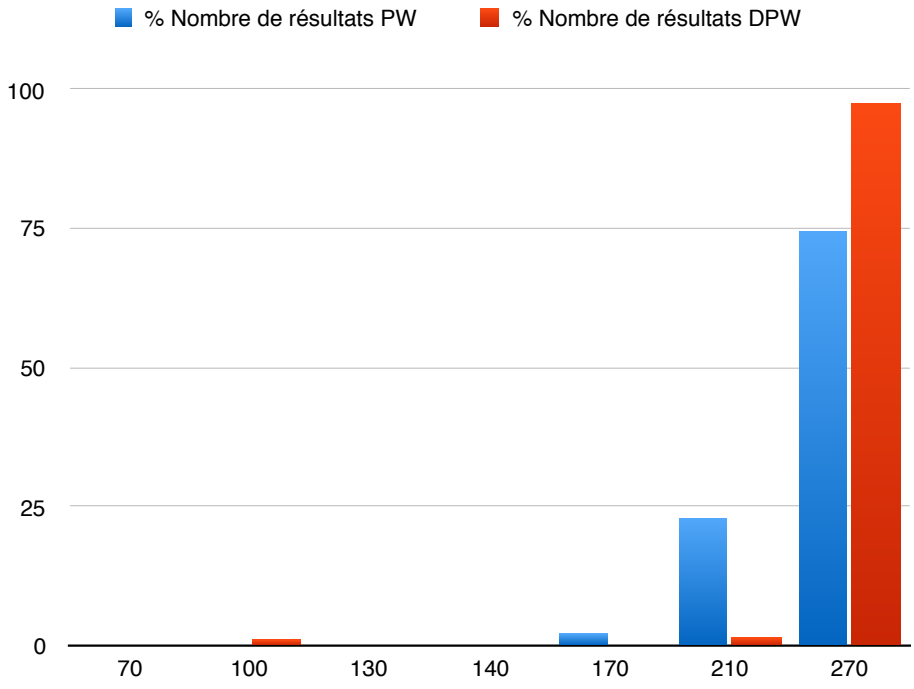


Figure 15 – Pourcentage de score obtenu pour PW et DPW avec $C = 8$, $\alpha = 0.6$ et $temps = 34$

Donc même en modifiant le monde et en rajoutant une difficulté supplémentaire, l'algorithme double progressive widening arrive tout de même à obtenir la récompense maximale de manière plus fiable que son homologue progressive widening.

3.7 Conclusion

L'algorithme de décision **M**onte **C**arlo **T**ree **S**earch [Cou06] est très facilement adaptable à toutes sortes de mondes, tantôt il peut être utilisé dans un monde discret (partie 2.4) tantôt il peut être adapté à un monde continue (partie 3).

Cependant, cette dernière adaptation se fait avec plus ou moins de réussite, en effet si nous voulons réutiliser le MCTS discret (partie 3.3) nous devons manuellement indiquer combien de valeurs d'actions nous souhaitons échantillonner, puis ajuster

ce paramètre par rapport aux précédents résultats, ceci est très fastidieux et peu dynamique.

Or, le progressive widening (partie 3.4) permet d'effectuer cet échantillonnage de façon dynamique, et est également plus performant que le MCTS discret (voir figures 9 et 10) sur notre premier problème. De plus, nous avons remarqué que lorsque le progressive widening ne gérait pas les bruits (figure 10) sa moyenne de récompense tend vers 170 pour $C <= 21$ soit la valeur optimale.

Donc, nous pouvons supposer que si nous avons un problème dans un monde continu où aucun bruit parasite risque de se produire alors l'utilisation du MCTS avec le progressive widening peut être une bonne initiative.

Maintenant, si nous nous retrouvons dans une situation où le monde est continu et où des bruits aléatoires risquent de se produire alors nous pensons qu'il est préférable d'utiliser le double progressive widening, car ces performances sont globalement meilleures que celle du progressive widening bruité (figure 13).

Donc nous avons pu déterminer qu'il était possible de planifier des actions dans un monde continu en utilisant un algorithme tel que le **M**onte **C**arlo **T**ree **S**earch, néanmoins il serait intéressant de voir s'il est également possible de gérer la planification d'actions dans un monde continu **et** dynamique.

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.