Ecole Normale Supérieure de Cachan

Rapport de stage L3

Les algorithmes de type Monte Carlo Tree Search appliqués aux Processus de Décision Markoviens

Auteur : Xavier Erny
Encadrant : Olivier Buffet
Encadrant : Vincent Thomas
Equipe : AT-LOR
Laboratoire : INRIA Nancy

1er Juin 2015 – 24 Juillet 2015

Table des Matières

1 Préambule

1.1 Encadrement

J'ai fait ce stage au laboratoire INRIA Nancy Grand Est dans l'équipe AT-LOR sous la supervision d'Olivier Buffet, chargé de recherche dans AT-LOR et de Vincent Thomas, maitre de conférences à l'Université de Lorraine et membre de l'équipe LARSEN.

Le but principal de l'équipe LARSEN est d'introduire les robots dans la société, en dehors des laboratoires et des industries. Il y a plusieurs thèmes de recherche sur lesquels l'équipe travaille, en particulier, la planification de tâches dans l'incertain, en rapport direct avec mon stage. Ce thème est aussi un des deux champs de recherches principaux de l'équipe AT-LOR, avec les systèmes multi-agents.

Un problème de planification de tâches peut être modélisé par un sytème avec lequel un agent peut interagir en modifiant son état. Résoudre un tel problème consiste à trouver quelles sont les actions à faire et dans quel ordre, afin d'optimiser un certain critère de performance. Dans une planification dans l'incertain, les actions ne sont pas déterministes.

1.2 Sujet de recherche

Mon stage consistait, dans un premier temps, à me familiariser avec les Processus de Décision Markovien (MDP), ainsi que des algorithmes permettant de les résoudre.

Un MDP est un système dans lequel un agent fait des actions pour modifier l'état du système. Les actions peuvent rapporter des récompenses qui dépendent de l'état du système et les changements d'états dûs à ces actions ne sont pas déterministes. Un problème classique consiste à savoir quelles sont les actions à effectuer pour optimiser la somme cumulée des récompenses en un minimum de temps (l'expérience s'arrête quand le système atteint un état terminal). Les MDP sont donc appropriés pour modéliser les problèmes de planification de tâches.

J'ai vu plusieurs algorithmes permettant de résoudre les MDP, comme "Value Iteration" et "Policy Iteration" dans [Russell et Norvig, 1994] ou encore "RTDP" et "Sparse Sampling" dans [Sigaud et Buffet, 2010]. Je me suis particulièrement intéressé aux algorithmes de type Monte Carlo Tree Search (MCTS). Dans ces algorithmes, on construit un arbre de décision en répétant un processus qui peut être décrit en deux étapes. La première étape consiste à choisir une branche de l'arbre, à descendre dans cette branche et à y ajouter une feuille. La seconde consiste à remonter la branche parcourue en mettant à jour les informations contenues dans les noeuds rencontrés. Ce sont ces informations qui permettent de choisir une branche à l'itération suivante.

Dans un second temps, je me suis renseigné sur les Processus de Décision Markovien Partiellement Observable (POMDP), un modèle étendu des MDP, et aux algorithmes pour les résoudre. Un POMDP est un MDP dans lequel on ne connait pas l'état du système, mais où on a accès à des observations sur cet état (on peut par exemple imaginer un labyrinthe où l'état du système serait notre position et où les indices seraient la configuration des murs autour de notre position).

Dans un troisième temps, je me suis intéressé à l'objectif principal du stage qui consistait à proposer un nouvel algorithme de type MCTS pour résoudre un modèle étendu de POMDP, les ρPOMDP. Dans un ρPOMDP, les récompenses ne dépendent pas de l'état réel du sytème comme dans les POMDP, mais de l'information que l'agent possède sur cet état. En faisant en sorte que les récompenses sont d'autant plus grandes que l'on a d'information sur l'état du système, les ρPOMDP permettent de représenter un nouveau type de problème : les problèmes où un agent cherche à acquérir de l'information.

2 Processus de Décision Markovien

2.1 Définition

Comme cela a déjà été dit, un MDP est un système dans lequel on peut faire des actions qui modifient éventuellement l'état du système et qui rapportent éventuellement une récompense. Ces actions ne sont pas déterministes (c'est-à-dire que faire une certaine action dans un certain état ne conduit pas toujours au même état).

Formellement, un MDP est un quadruplet $\langle S, A, T, R \rangle$ où :

— S est l'ensemble des états possibles du système

— A est l'ensemble des actions que l'on peut faire

— $T : S \times A \times S \to [0, 1]$ est la fonction de transition ($T(s_1, a, s_2)$ donne la probabilité d'arriver en s_2 sachant que l'on est en s_1 et que l'on fait l'action a)

— $R : S \times A \to \mathbb{R}$ est la fonction de récompense ($R(s, a)$ renvoie la récompense que l'on obtient quand on fait l'action a dans l'état s)

Résoudre un problème de planification revient à trouver quelles sont les actions à faire et dans quel ordre les faire pour maximiser un critère de performance. Dans le cas d'un MDP, cela revient à trouver une politique $\pi : S \to A$ qui détermine pour chaque état l'action à faire. Le critère de performance dans le cas d'un MDP est habituellement la somme cumulée des récompenses.

Pour estimer la valeur d'un état du MDP en suivant une politique, on peut effectuer des parcours du MDP, guidés par cette politique, qui partent de cet état et prendre la valeur moyenne obtenue sur ces parcours. Par exemple, la valeur du parcour $s = s_0, a_0, s_1, a_1, ..., s_n, a_n$ est $V_s = \sum_{k=0}^{n} R(s_k, a_k)$

Cette technique marche si les parcours sont finis. Si un parcours est infini, il faut utiliser la notion de récompense amortie. Pour cela il faut ajouter un paramètre γ dans $]0, 1[$, et le critère de performance change, puisque la valeur du parcours $s = s_0, a_0, s_1, a_1, ..., s_n, a_n, ...$, vaut alors $V_{s,\gamma} = \sum_{k=0}^{+\infty} \gamma^k R(s_k, a_k)$.

En effet, avec la notion de récompense amortie, la valeur d'un parcours est bornée, même pour un parcours infini, puisque

$$|V_{s,\gamma}| \leq \sum_{k \in \mathbb{N}} \gamma^k R = \frac{\gamma R}{1 - \gamma}$$

où $R = max_{(s_0,a) \in S \times A} |R(s_0, a)|$

La notion de récompense amortie peut aussi être utilisée dans les problèmes où les parcours sont finis mais où on veut que le critère de performance tienne compte de la durée du parcours. C'est-à-dire, quand on veut que le parcours soit le plus court possible.

2.2 Algorithme Monte Carlo Tree Search (MDP)

Un algorithme MCTS construit un arbre de décision d'une forme particulière, puisque cet arbre va alterner noeuds états et noeuds actions. Plus exactement, la racine est un noeud état (qui correspond à l'état initial s_0). Les fils de la racine sont des noeuds actions. Les fils de chacun des noeuds actions sont des noeuds états qui correspondent tous à un état qui a été atteint au moins une fois en partant de s_0 après avoir fait l'action correspondant au noeud père, et ainsi de suite (cf annexe A).

2.2.1 Stratégie UCB (Upper Confidence Bound)

L'algoritme UCT (UCB applied to Trees) est de type MCTS. Il repose sur un parcours de l'arbre, notamment sur une stratégie qui permet, quand on est sur un noeud état, de choisir une action (et donc de descendre dans un noeud action). La stratégie de sélection d'action utilisée ici est la stratégie UCB, introduite dans [Kocsis et Szepesvári, 2006].

La stratégie UCB permet de répondre au compromis exploration-exploitation, qui est le suivant : si on a plusieurs choix d'action qui conduisent à des gains différents, vaut-il mieux exploiter l'action qui nous a donné les gains les plus élevés ou alors explorer les autres actions au cas où l'une d'elles rapporterait un gain élevé que l'on a pas encore eu.

Introduisons quelques notations. Supposons qu'il existe n actions possibles. Pour $1 \leq j \leq n$, soient $\overline{X_j}$ le gain moyen obtenu en faisant l'action j, n_j le nombre de fois que l'on a fait l'action j et $N = \sum_{j=1}^{n} n_j$.

La stratégie UCB consiste à choisir l'action j qui maximise la valeur

$$\overline{X_j} + K\sqrt{\frac{\ln N}{n_j}}$$

où K est la constante d'exploration

Remarque : Si $n_j = 0$, la valeur ci-dessus vaut $+\infty$, ce qui signifie qu'on préférera toujours une action qui n'a jamais été choisie à celles déjà testées.

Intuitivement, on voit bien que le terme $\overline{X_j}$ favorise l'exploitation et que le terme $\sqrt{\frac{\ln N}{n_j}}$ favorise l'exploration. Donc on peut s'attendre à ce que l'exploration soit favorisée quand K grandit.

Dans l'arbre de décision qui est construit dans l'algorithme MCTS, les noeuds actions doivent contenir deux nombres. Le premier est la valeur estimée du noeud, qui correspond à $\overline{X_j}$, c'est-à-dire la valeur moyenne que l'on a obtenue à partir de ce noeud. Le second est un compteur qui correspond à n_j, c'est-à-dire le nombre de fois où on est passé par ce noeud.

2.2.2 L'algorithme UCT

L'algorithme MCTS décrit ici s'appelle UCT et a été introduit dans [Kocsis et Szepesvári, 2006].

UCT repose sur un processus que l'on répète, qui consiste à ajouter un noeud à l'arbre et à mettre à jour les valeurs et compteurs des noeuds rencontrés.

Ce processus peut être décrit en quatre étapes :

- **la sélection de la branche** : On part de la racine (l'état de la racine est notre état courant), on descend dans un nœud action selon la stratégie UCB, puis on simule une transition du MDP (à partir de l'état courant et de l'action choisie) pour obtenir un nouvel état courant. Si le nœud existe déjà parmi les fils du nœud action (et que l'état n'est pas un état terminal), alors on descend dans ce nœud et on recommence l'étape de sélection, sinon le nœud est à créer et on passe à l'étape suivante.
- **la création du nœud** : Si l'état courant est terminal, le nœud n'a pas besoin d'être développé, et on passe directement à l'étape suivante. Sinon, l'algorithme se trouve dans un nœud action qui ne possède pas l'état courant comme fils, on ajoute alors l'état courant comme nouveau fils du nœud action et on passe à l'étape suivante.
- **la simulation de la politique par défaut** : Dans cette étape, on cherche à estimer une valeur correspondant à la récompense que l'on peut espérer à partir de l'état du nœud que

l'on vient de créer. Il faut disposer d'une politique par défaut, et simuler cette politique depuis l'état courant jusqu'à un état terminal et retenir les récompenses générées dans une variable v.

— **la mise à jour de l'arbre** : On remonte enfin les valeurs dans l'arbre de la façon suivante : quand on rencontre un nœud action, on incrémente son compteur, on met à jour la variable v (ie on lui ajoute la récompense obtenue à ce noeud via la fonction de récompense) et on met à jour la valeur du nœud grâce à la variable v (la valeur du nœud correspond à la moyenne de toutes les valeurs des variables qui ont été remontées dans ce nœud).

Ces quatres étapes, ainsi que l'algorithme qui fait la synthèse de ces étapes, sont illustrés en annexe A. Pour simplifier, appellons ces quatres étapes, les étapes d'expansion. L'algorithme UCT consiste à répéter un certain nombre de fois les étapes d'expansion, en partant d'un arbre initialement réduit à un nœud correspondant à l'état initial.

3 Processus de Décision Markovien Partiellement Observable

3.1 Définition

La différence avec un MDP, est que dans les POMDP on ne connait pas l'état du système, mais à chaque pas de temps, l'agent reçoit une observation qui dépend de l'état courant du sytème et de l'action effectuée. Cette observation fournit à l'agent une information sur l'état du système.

Formellement, un POMDP est sixtuplet $\langle S, A, T, R, \Omega, O \rangle$:

— S est l'ensemble des états
— A est l'ensemble des actions
— $T : S \times A \times S \to [0, 1]$ est la fonction de transition
— $R : S \times A \to \mathbb{R}$ est la fonction de récompense
— Ω est l'ensemble des observations
— $O : S \times \Omega \to [0, 1]$ est la fonction d'observation ($O(s, \omega)$ donne la probabilité d'observer ω sachant que l'on est dans l'état s)

La notion de politique n'est plus la même que pour les MDP. Dans un MDP, l'agent choisit une politique qui dépend de l'état courant. Cet état n'étant pas accessible dans un POMDP, l'agent doit travailler sur l'historique des actions/observations pour se "repérer". Les politiques dans les POMDP dépendent de cet historique et l'arbre de décision des algorithmes MCTS vont construire ces politiques.

Une tactique qui peut être utilisée pour raisonner sur les POMDP est d'utiliser la notion de Belief State. Le Belief State est une distribution sur tous les états, qui donne la probabilité pour chaque état d'être l'état réel du sytème. Nous reviendrons sur le Belief State plus tard, puisqu'il s'agit d'une notion clef pour les ρPOMDP.

3.2 Exemple : *Rock Sample*

Un exemple de POMDP que l'on peut trouver dans la littérature est le problème *Rock Sample*, qui a été proposé par Trey Smith et Reid Simmons dans [Smith et Simmons, 2004].

Le problème est caractérisé par un robot capable d'échantillonner des rochers pour gagner des récompenses. Le robot et les rochers se trouvent dans une grille et deux types de rochers sont présents dans l'environnement : les bons et les mauvais. Pour que le robot puisse échantillonner

un rocher, il doit être sur la même case que le rocher. On connait la position du robot ainsi que celles des rochers, mais on ne connait pas leur type. Plus formellement, un état de ce POMDP est la donnée d'une grille, une position du robot, les positions des rochers présents, ainsi que leur type. Seuls les types des rochers ne sont pas directement observables.

Si le robot échantillonne un bon rocher, il reçoit une récompense de 10, s'il en échantillonne un mauvais, la récompense est -10. Le robot dispose aussi d'un scanner qui lui permet d'estimer à distance le type d'un rocher, mais le scanner est plus fiable quand le robot est proche du rocher scanné. Quand le robot échantillonne un bon rocher, ce rocher change de type, car cela n'a aucun intérêt d'échantillonner deux fois le même rocher. Les états terminaux sont ceux où le robot sort de la grille à droite, et quand cela arrive, le récompense reçue est 10. Il y a donc $5+r$ actions possibles (où r désigne le nombre de rochers) : quatre actions de déplacement, une action d'échantillonnage et r actions permettant de scanner chacun des rochers. Toutes les actions autres que l'échantillonage ont une récompense nulle.

L'objectif de ce problème est de voir comment utiliser le scanner à distance, pour générer les récompenses le plus vite possible. On veut donc utiliser un critère de performance qui prenne la durée de l'expérience en compte, ce qui peut être représenté avec la notion de récompense amortie γ, comme cela a déjà été présenté à la section 2.1. Dans le problème *Rock Sample*, on prend $\gamma = 0.95$.

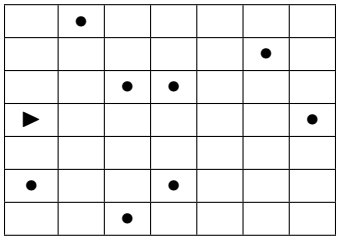

FIGURE 1 – Exemple de positions initiales pour *Rock Sample* avec huit rochers

Sur la figure 1, les rochers sont représentés par • et le robot par ▶. Dans le Belief State initial, les états qui ne correspondent pas à la grille de la figure 1, ont une probabilité nulle, et tous les états correspondant à cette grille ont une probabilité de $\frac{1}{2^8}$ (les 2^8 états possibles correspondent à toutes les configurations possibles des types des rochers)

3.3 Algorithme Monte Carlo Tree Search (POMDP) : POMCP

L'algorithme MCTS décrit ici s'appelle POMCP (Partially Observable Monte Carlo Planning) et a été introduit dans [Silver et Veness, 2010].

La différence principale entre POMCP et UCT, et que comme on n'a pas accès à l'état du sytème, il faut remplacer les noeuds états par des noeuds observations. Les branches correspondent alors à des historiques actions/observations.

Pour pouvoir générer les observations et les récompenses, il faut tout de même manipuler un état. Pour ce faire, à chaque fois que l'on parcourt l'abre, on génère un état initial à partir du Belief State initial au niveau de la racine, puis lors du parcours de l'arbre, on simule une transition du POMDP sur l'état courant à chaque fois que l'on passe par un noeud action.

Il faut aussi faire attention à changer l'état initial à chaque nouveau parcours, pour rendre compte des incertitudes sur l'état initial. Sinon, l'algorithme résolverait le problème pour un seul

état initial, et il aurait le même comportement que s'il traitait un MDP. Par exemple, si on s'intéresse à la grille de la figure 1, à chaque nouveau parcours, il faut générer un état initial dont les positions des rochers et du robot sont toujours les mêmes, mais où les types des rochers sont tirés aléatoirement à partir du Belief State initial. Ainsi, selon la branche de l'arbre où on se trouve, l'historique des actions/observations n'est pas le même, et les actions à faire dépendent de ces observations. Et donc, cet algorithme va construire une politique de POMDP.

3.4 Expériences

On peut déjà étudier le rôle qu'ont certains paramètres de l'algorithme, comme la constante d'exploration K ou γ. Notamment, on peut vérifier l'intuition sur le rôle de la constante d'exploration K, à savoir qu'elle favorise bien l'exploration.

Pour cela, j'ai tracé les histogrammes de la figure 3. Chaque histogramme correspond à une constante d'exploration différente et chaque barre correspond à un rocher. Plus exactement une barre représente le nombre de fois que le robot a utilisé le scanner sur le rocher correspondant divisé par le nombre de fois qu'on a utilisé les étapes d'expansion de l'arbre. Ainsi, plus une barre est grande, plus le robot a tendance à s'intéresser au rocher correspondant.

Pour plus de lisibilité, j'ai associé à chaque barre une couleur différente. La figure 2 montre quelles couleurs sont associées à quels rochers.

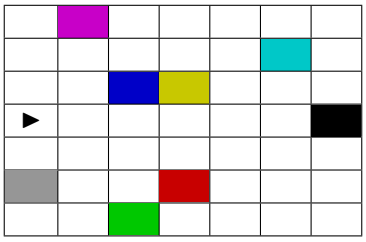

FIGURE 2 – les couleurs des rochers

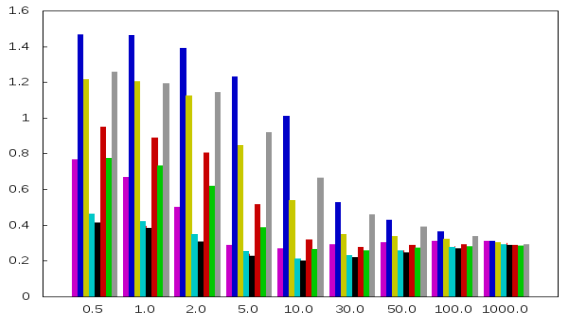

FIGURE 3 – Histogrammes du nombre de fois que l'on scanne chaque rocher en fonction de la constante d'exploration K de la stratégie UCB (chaque histogramme est la moyenne de 1000 résutats de l'algorithme MCTS, où l'algorithme a appelé 150000 fois les étapes d'expansion)

La figure 3 montre que quand K augmente, le robot utilise autant son scanner sur chaque rocher, ce qui montre bien que l'algorithme MCTS va plutôt explorer les choix possibles qu'exploiter les politiques correspondant aux branches des parcours précédents.

On peut faire le même type de figure en faisant varier γ, et en fixant la constante d'exploration $K = 5$. On obtient la figure 4. On peut voir que plus γ diminue, moins le robot utilise le scanner, sauf sur le rocher gris. En fait, quand γ diminue, le robot va essayer de gagner des récompenses le plus vite possible, et il va donc s'intéresser au rocher le plus proche du point de départ, le rocher gris, même s'il est plus isolé que d'autres qui sont un peu plus loin, comme les rochers bleu et jaune. Ensuite le robot trouve plus rentable d'aller directement à la sortie que de perdre du temps à utiliser le scanner sur le rocher rouge qui est pourtant sur le chemin. En effet, le rocher rouge peut être de type mauvais, et donc l'espérance de ce que rapporte ce rocher est inférieure à la récompense de la sortie.

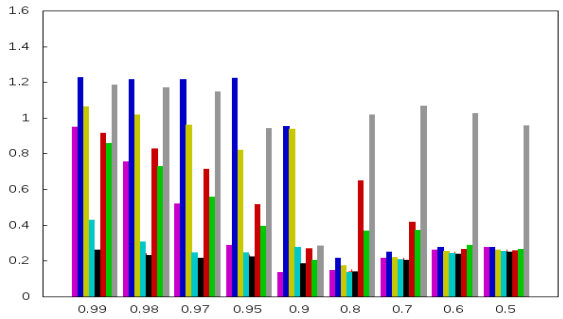

FIGURE 4 – Histogrammes du nombre de fois que l'on scanne chaque rocher en fonction de γ (chaque histogramme est la moyenne de 1000 résutats de l'algorithme MCTS, où l'algorithme a appelé 150000 fois les étapes d'expansion)

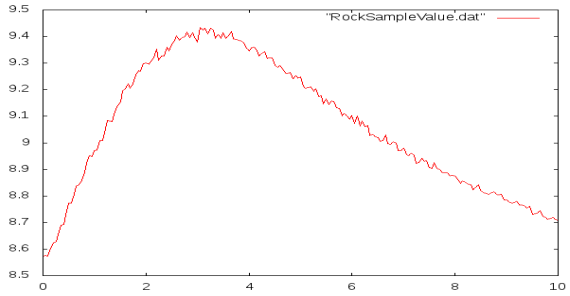

FIGURE 5 – La valeur de la racine de l'arbre de l'algorithme UCT en fonction de la constante d'exploration K (pour chaque constante d'exploration, la valeur est la moyenne de 1000 résutats de l'algorithme MCTS, où l'algorithme a appelé 150000 fois la processus de création de noeud)

Pour revenir sur le compromis exploitation-exploration, on peut se demander comment choisir la constante d'exploration pour optimiser l'efficacité de l'algorithme. J'ai tracé la figure 5 pour répondre à cette question. Sur la courbe, on voit que, sur cet exemple, il y a une constante d'explo-

ration optimale qui vaut à peu près 3. Dans la suite, on va revenir sur l'étude du rôle de la constante d'exploration K dans le compromis exploitation-exploration.

4 ρ-Processus de Décision Markovien Partiellement Observable

4.1 Définition

Pour passer de la définition d'un POMDP à celle d'un ρPOMDP, il suffit de changer la forme de la fonction de récompense. En effet, dans un POMDP, la fonction de récompense R est du type $R : S \times A \to \mathbb{R}$, alors que dans un ρPOMDP, la fonction de récompense ρ dépend du Belief State, et est donc de la forme $\rho : [0,1]^S \to \mathbb{R}$. On peut aussi définir ρ de la forme $\rho : [0,1]^S \times A \to \mathbb{R}$ pour rester le plus général possible.

Cette nouvelle fonction de récompense permet de définir un autre type de problème : les problèmes où il faut acquérir de l'information. Pour cela, il suffit de faire en sorte que la fonction ρ soit d'autant plus grande qu'il y a de certitude concernant les états possibles du sytème. On peut par exemple, se servir de la notion d'entropie pour choisir ρ, plus excatement l'opposé de l'entropie.

On peut donc choisir la fonction ρ comme l'opposé de l'entropie de Shannon :

$$\rho(b) = \sum_{s \in S} b(s) \ln b(s)$$

En effet, on remarque que pour tout nombre x dans $[0,1]$, la quantité $x \ln x$ est négative, et elle est nulle en $x = 0$ et $x = 1$. Ce qui signifie donc que si on a un état s, plus on a de certitude sur l'état (ie plus $b(s)$ est proche de 1 ou de 0) plus la quantité $b(s) \ln b(s)$ est grande. Cela correspond bien à ce que l'on veut pour la fonction ρ.

En pratique, on préfère avoir des valeurs positives, donc on définit plutôt ρ de la façon suivante :

$$\rho(b) = \ln |S| + \sum_{s \in S} b(s) \ln b(s)$$

4.2 Exemple : *Rock Diagnosis*

Pour illustrer cette définition, on peut prendre l'exemple d'une variante de *Rock Sample*, qui est *Rock Diagnosis*. *Rock Diagnosis* a été introduit pour la première fois par Mauricio Araya-López, Olivier Buffet et Vincent Thomas dans l'article [Araya-López *et al.*, 2013].

Dans cette variante, on ne cherche plus à échantillonner les bon rochers, mais seulement à connaître le type des rochers, c'est-à-dire avoir le plus d'information possible sur l'état réel du sytème. Il n'y a donc plus l'action d'échantillonnage, et les "bons" rochers ne sont pas plus intéressants à trouver que les "mauvais".

Il y a une autre différence avec *Rock Sample* qui concerne les récompenses. Quand on fait un parcours dans une grille *Rock Sample*, on génère des récompenses à chaque étape. Dans *Rock Diagnosis*, on cherche à maximiser la connaissance à la sortie de la grille, on ne reçoit donc une récompense sur l'état de croyance qui a été construit qu'à la sortie.

Comme dans Rock Sample, on veut aussi que le robot cherche à optimiser son temps de parcours, sinon il lui suffirait de scanner chaque rocher un à un en allant sur chaque case. Pour cela, on réutilise la notion de récompense amortie en prenant un $\gamma = 0.95$.

4.3 Algorithme Monte Carlo Tree Search (ρPOMDP)

En plus de faire évoluer un état courant, qui doit générer les observations permettant de construire l'arbre, il faut faire évoluer un Belief State pour générer les récompenses. Il faut donc un moyen pour le faire évoluer, Olivier Buffet a proposé d'utiliser un filtrage particulaire, qui est approprié à la forme de (ρ)POMDP.

4.3.1 Filtrage particulaire

Le filtrage particulaire est un procédé pour approcher une distribution via un ensemble fini de couples $\langle point, masse \rangle$.

On part d'un certain nombre de points qui ont chacun une certaine masse, qui dépend de ce que l'on connait sur l'état du sytème. Après avoir effectué une action et obtenu une observation, on met à jour les points et les masses.

Pour mettre à jour les points, il suffit de simuler une transition du ρPOMDP sur chaque point.

Pour mettre à jour les masses, j'ai utilisé les formules de [Fil,].

Introduisons quelques notations pour expliquer la mise à jour des masses. Soient x_0^k $(1 \leq k \leq n)$ les n points de départ, w_0^k leurs poids respectifs, x_1^k les n points obtenus après la simulation de l'action a et ω l'observation obtenu.

D'abord on calcule les masses non normalisées $w_1^{k\prime}$

$$w_1^{k\prime} = w_0^k P(\omega | x_1^k, a)$$

où $P(\omega|x_1^k, a)$ est la probabilité d'oberver ω sachant que l'on a fait l'action a et que l'on est arrivé dans l'état x_1^k

Puis on normalise les $w_1^{k\prime}$ pour obtenir les masses w_1^k des points x_1^k

$$w_1^k = \frac{w_1^{k\prime}}{\sum_{l=1}^{n} w_1^{l\prime}}$$

En pratique, si l'ensemble des états possibles est infini (ou très grand), il faut tirer uniformément des états à partir du Belief State et leur donner à tous la même masse de départ.

Dans les exemples que j'ai traités, le nombre d'états possibles était raisonnable (toujours inférieur à 128), donc j'ai directement pris les états possibles et leur ai attribué leur masse selon le Belief State. Cela permet de faire moins d'approximations, et donc d'être plus précis, mais cela me semblait aussi plus simple et plus rapide à coder.

4.3.2 L'algorithme : ρPOMCP

La différence avec POMCP, est que c'est le Belief State et non l'état réel du sytème qui génère les récompenses. Pour mettre à jour l'état réel, on ne fait que simuler des transitions. Autrement dit, il y a une fonction du type $S \times A \to S$ qui met à jour l'état du système.

L'idée qui m'a semblé la plus naturelle pour adapter l'algorithme aux ρPOMDP est donc de définir une fonction du type $[0,1]^S \times \Omega \times A \to [0,1]^S$ pour mettre à jour le Belief State. Et la forme de cette fonction est parfaitement adaptée à un filtrage particulaire.

Avec une telle fonction, on peut facilement définir un algorithme MCTS pour les ρPOMDP. En effet, il suffit de faire comme dans l'algorithme MCTS pour POMDP, sauf qu'au moment où on

simule une transition pour faire avancer l'état du sytème, il faut aussi utiliser la fonction ci-dessus pour faire évoluer le Belief State.

Cet algorithme ρPOMCP est écrit en pseudo-code en annexe B.

4.4 Expériences

Afin de tester la performance de ce nouvel algorithme, j'ai repris trois exemples de grille *Rock Diagnosis* dans la thèse de Mauricio Araya-López [López, 2013]. Ce sont les exemples de la figure 6.

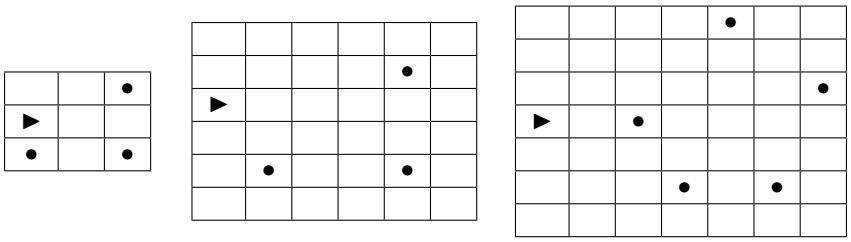

FIGURE 6 – positions initiales pour *Rock Diagnosis*

Sur chacun des exemples des figures 7, 8 et 9, j'ai tracé plusieurs courbes, montrant, la valeur que trouve l'algorithme ρPOMCP en fonction du nombre de fois que l'on a utilisé les étapes d'expansion de l'arbre.

Chaque courbe correspond à une constante d'exploration différente. Cela permet de voir le rôle de la constante d'exploration dans ρPOMCP. Sur chaque figure les courbes correspondent aux constantes d'exploration : $0.08, 0.1, 0.5, 1, 2$ et 5.

La première chose que l'on remarque sur ces courbes, c'est que plus la constante d'exploration est grande, plus la courbe met de temps pour croître. C'est normal, puisque quand la constante d'exploration augmente, alors l'exploration est favorisée, et donc l'algorithme met plus de temps pour exploiter une stratégie intéressante.

Sur la figure 7, on voit que l'on peut basculer très rapidement d'une constante d'exploration presqu'optimale, pour $K = 0.1$, à une constante d'exploration très mauvaise, pour $K = 0.08$.

Il y a une chose intéressante à noter, ce sont les variations de croissance des courbes. Ces variations se voient particulièrement bien sur la courbe bleue clair de la figure 9. En effet, on voit une croissance exponentielle inverse jusqu'au point d'abscisse 20000, puis la courbe recroît en exponentielle inverse alors qu'elle s'était aplatie.

Pour avoir ces variations, il ne faut pas que la constante d'exploration soit trop faible. Sinon l'algorithme ne va pas explorer, il ne changera donc pas la stratégie qu'il exploite, or ces variations sont justement dûes à des changements de stratégies. Toutefois si la constante d'exploration est trop élevée, il n'y aura pas non plus ces variations, puisque pour changer de stratégie, il faut l'exploiter.

Ces variations sont intéressantes, car elles représentent vraiment comment l'algorithme travaille à travers le compromis exploration-exploitation. Ces exemples montrent notamment qu'il est difficile de trouver une bonne solution au compromis, puisque cette solution doit dépendre, entre autre, du temps dont on dispose. En effet, si on s'intéresse la figure 8, on voit que dans le domaine entre les abscisses 0 et 25000, la meilleur constante d'exploration est 0.1, tandis que dans le domaine supérieur à l'abscisse 25000, la constante 0.5 est meilleure.

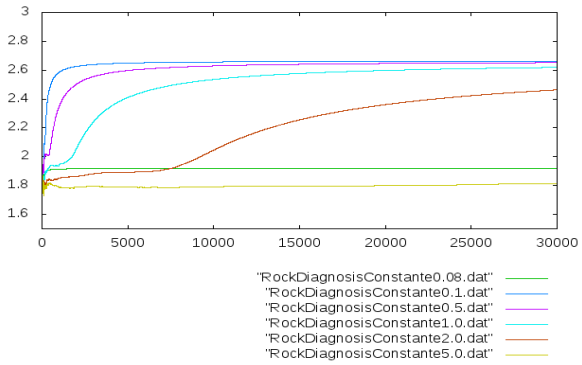

FIGURE 7 – Rock Diagnosis : grille 3×3

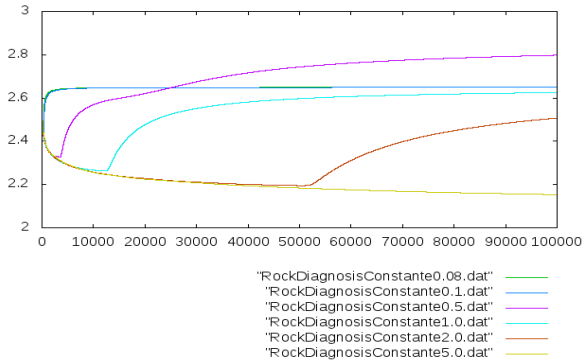

FIGURE 8 – Rock Diagnosis : grille 6×6

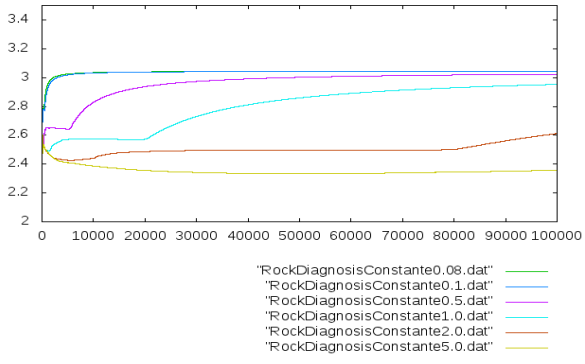

FIGURE 9 – Rock Diagnosis : grille 7×7

Bilan et perspectives

Ce stage a été intéressant et bénéfique sur plusieurs points. En effet, il m'a permis d'apprendre la programmation orientée objet avec java, ainsi que de me familiariser avec les processus stochastiques et les algorithmes pour les résoudre. Certains algorithmes sont d'autant plus intéressants qu'ils peuvent être appliqués dans d'autres domaines, comme la théorie des jeux.

Au cours de ce stage, j'ai pu développer un nouvel algorithme (ρPOMCP) permettant de résoudre les ρPOMDP. Pour cela, j'ai adapté l'agorithme POMCP en lui ajoutant une fonction de mise à jour du Belief State. Cette fonction permet de faire évoluer le Belief State en même temps que l'état du ρPOMDP, et cela permet à l'algorithme ρPOMCP de résoudre les ρPOMDP comme POMCP avec les POMDP. On a pu constater que l'algorithme ρPOMCP que j'ai proposé a l'air de bien faire ce qu'on attend de lui.

Si j'avais eu plus de temps, j'aurais testé l'algorithme ρPOMCP sur d'autres exemples de ρPOMDP, pour voir, par exemple, s'il existe des constantes d'exploration qui donnent souvent de bons résultats.

Toutefois, il y a beaucoup d'algorithmes efficaces pour résoudre les POMDP, on pourrait donc essayer d'en adapter d'autres aux ρPOMDP, comme par exemple, les algorithmes HSVI ou PERSEUS, introduits respectivement dans les articles [Smith et Simmons, 2004] et [Spaan et Vlassis, 2005]. Cela permettrait de les comparer avec mon algorithme, en s'intéressant aux points positifs et négatifs de chacun.

Références

[Fil,] Filtre particulaire. `https://fr.wikipedia.org/wiki/Filtre_particulaire`. Échantillonnage avec rééchantillonnage par importance (SIR).

[Araya-López *et al.*, 2013] Araya-López, M., Buffet, O. et Thomas, V. (2013). Active Diagnosis Through Information-Lookahead Planning. Rapport technique.

[Kocsis et Szepesvári, 2006] Kocsis, L. et Szepesvári, C. (2006). Bandit based Monte-Carlo Planning. *In Proceedings of the Seventeenth European Conference on Machine Learning (ECML'06)*.

[López, 2013] López, M. A. A. (2013). *Des algorithmes presques optimaux pour les problèmes de décision séquentielle à des fins de collecte d'information*. Thèse de doctorat, Université de Lorraine.

[Russell et Norvig, 1994] Russell, S. et Norvig, P. (1994). *Artificial Intelligence : A Modern Approach*. Prentice Hall, Englewood Cliffs.

[Sigaud et Buffet, 2010] Sigaud, O. et Buffet, O. (2010). *Markov Decision Processes in Artificial Intelligence*. ISTE Ltd and John Wileys & sons.

[Silver et Veness, 2010] Silver, D. et Veness, J. (2010). Monte-Carlo Planning in Large POMDPs.

[Smith et Simmons, 2004] Smith, T. et Simmons, R. (2004). Heuristic Search Value Iteration for POMDPs. *Robotics Institute, Carnegie Mellon University*, page 524.

[Spaan et Vlassis, 2005] Spaan, M. T. J. et Vlassis, N. (2005). Perseus : Randomized Point-based Value Iteration for POMDPs. *Journal of Artificial Intelligence Research*, pages 195–220.

A Annexe : illustration de UCT

Dans les arbres de la page suivante, les nœuds états sont carrés et les nœuds actions ronds. Dans les noeuds états, on écrit seulement l'état associé au nœud, il est possible qu'un état apparaisse dans deux nœuds différents. Dans les nœuds actions, en plus de l'action associée, on écrit deux nombres : celui du haut représente la valeur du nœud, celui du bas le nombre de fois que l'on est passé par ce nœud.

Sur la figure 10, l'algorithme UCT sélectionne une branche. Pour cela, il part de la racine, puis choisit entre les deux actions possibles a_0 et a_1. Si on prend une constante d'exploration $K = 5$ par exemple, on a : $8 + 5 \times \sqrt{\frac{\ln 6}{2}} > 9 + 5 \times \sqrt{\frac{\ln 6}{4}}$. UCT va donc choisir l'action a_0. Puis il simule une transition partant du MDP depuis l'état s_0 avec l'action a_0. Sur cet exemple, le MDP va être dans l'état s_1. Et enfin, UCT va choisir une action parmi les deux possibles, ici ça sera a_1.

Sur la figure 11, l'algorithme simule une transition du MDP depuis l'état s_1 avec l'action a_1, ce qui va conduire à l'état s_7. Puis, UCT créé un nouveau nœud état.

Sur la figure 12, l'algorithme fait un parcours du MDP depuis l'état s_7 avec une politique par défaut. Ce parcours va rapporter une certaine récompense, qui vaut dans cet exemple 20.

Sur la figure 13, l'algorithme part du nouveau nœud (ici le nœud s_7), et initialise une variable à $v = 20$. Il remonte l'arbre d'un nœud pour mettre à jour les nombres du nœud action sur lequel il vient d'arriver. Pour cela, il incrémente le nombre du bas (ie celui qui compte le nombre de fois que l'on ait passé dans ce nœud). Et pour connaitre la nouvelle valeur du nombre du haut, il regarde l'état du nœud père et génère la récompense $R(s_1, a_1)$, puis il actualise v via la formule $v = v + R(s_1, a_1)$. Comme c'est la première fois que l'algorithme passe par ce nœud, on met simplement la valeur v à la place du nombre du haut (qui valait ∞). Puis sur le nœud action suivant, on refait la même chose, sauf que pour mettre à jour le nombre du haut, il faut faire la moyenne entre v et toutes les valeurs précédentes de ce nœud : dans cet exemple, l'ancienne valeur était la moyenne de deux nombres et valait 8. La nouvelle valeur doit donc être $\frac{8 \times 2 + 26}{3} = 14$

Remarque : Les nœuds actions sont toujours présents sous les nœuds états, ils sont éventuellements implicites si l'algorithme ne les a pas encore parcourus. Ils ne sont pas construits par la phase de création de nœud, ils sont construits automatiquement quand on créé un nouveau nœud état.

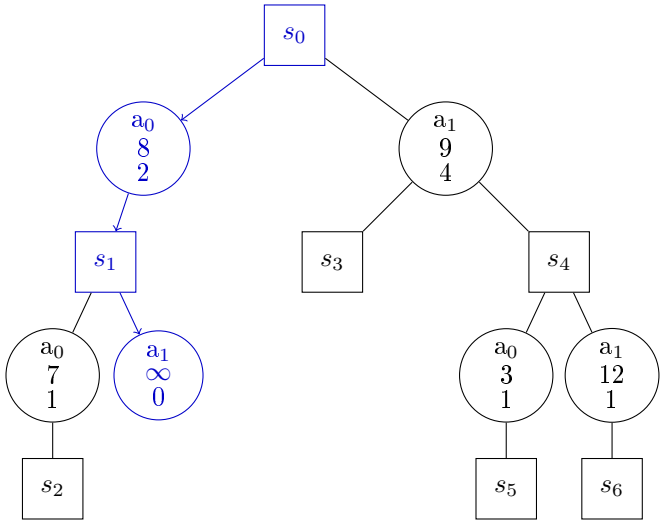

FIGURE 10 – la sélection d'une branche

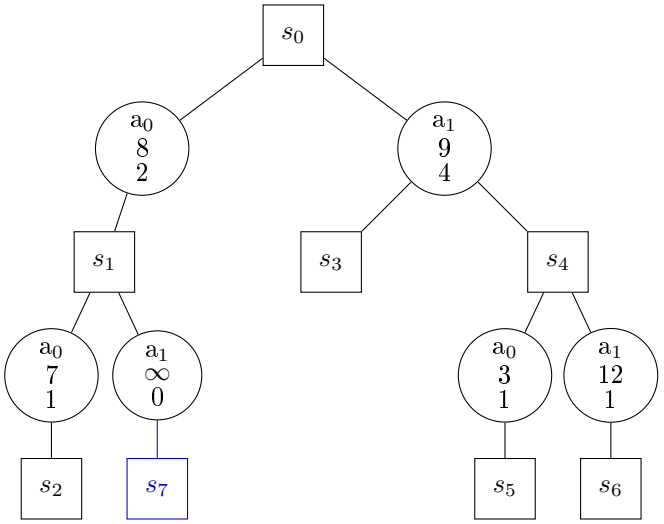

FIGURE 11 – la création d'un noeud

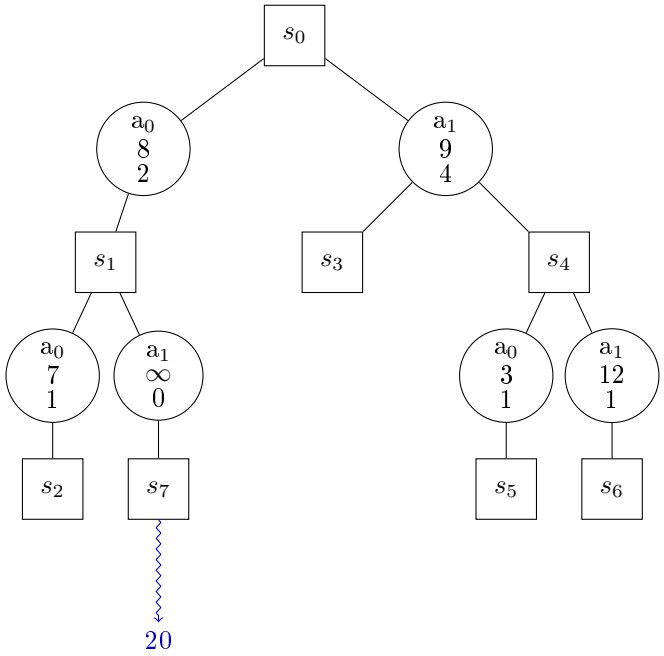

FIGURE 12 – la simulation de la politique par défaut

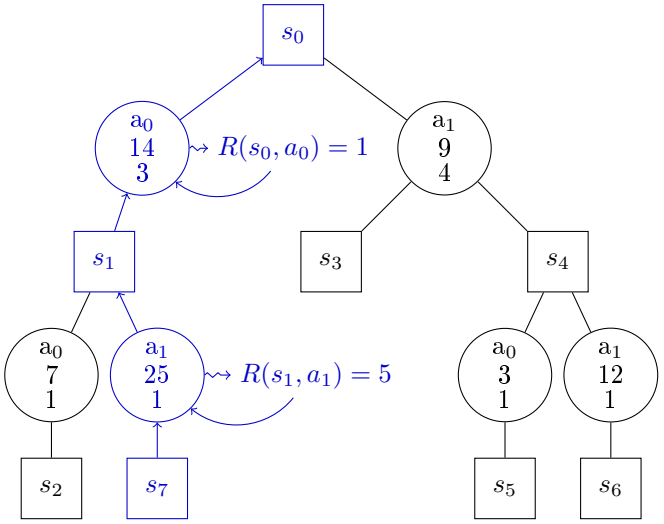

FIGURE 13 – la mise à jour de l'arbre

B Annexe : ρPOMCP

Pour écrire les algorithmes, on va supposer que l'on dispose d'un ρPOMDP implicite et des fonctions suivantes :

- observation : une fonction qui étant donné un état et une action, renvoie une observation générée par le ρPOMDP
- terminal : une fonction qui étant donné une observation renvoie $true$ si elle a été obtenu à partir d'un état terminal et $false$ sinon
- UCB : une fonction qui étant donnés un nœud état, renvoie l'action à faire selon la stratégie UCB
- ρPOMDP : une fonction qui étant donnés un état et une action, renvoie un état (cette fonction simule une transition du ρPOMDP)
- fils : une fonction qui étant donné un nœud action, renvoie l'ensemble des observations de ses fils
- fils_observation : une fonction qui étant donnés un nœud action et une observation renvoie le fils du nœud correspondant à cette observation (s'il existe)
- fils_action : une fonction qui étant donnés un nœud observation et une action renvoie le fils du nœud correspondant à cette action (s'il existe)
- : une fonction qui étant donnés un Belief State et une action renvoie un nombre réel (c'est la fonction de récompense)
- racine : une fonction qui étant donné un nœud, renvoie $true$ si le nœud est racine et $false$ sinon
- ajout_fils : une fonction qui étant donnés un nœud action et une observation, ajoute un fils à ce nœud, le nouveau nœud est un nœud observation.

On va aussi supposer que les nœuds actions sont des objets ayant six attributs :

- un attribut action
- un attribut père de type noeud
- un attribut fils de type ensemble de noeud
- un attribut compteur de type nombre entier
- un attribut valeur de type nombre réel
- un attribut récompense de type nombre réel

Les nœuds observations n'ont qu'un seul attribut : observation.

L'attribut récompense des nœuds actions était inutile dans UCT, puisqu'on pouvait générer les récompenses du MDP en remontant l'abre. Or, dans ρPOMCP, pour faire cela, il faudrait pouvoir remonter les mises à jour du Belief State. L'attribut récompense permet de générer une récompense lors de la descente dans l'arbre et de la retenir dans les nœuds, pour pouvoir l'utiliser lors de la montée.

Si b est un booléen, on notera $!b$ son opposé.

La politique pour un ρPOMDP ne peut pas dépendre de l'état courant du sytème comme pour un MDP, puisqu'il n'est pas connu par l'agent. La politique doit plutôt dépendre du Belief State.

Données : la racine et le belief state initial
Résultat : une feuille et la mise à jour du Belief State
Entrées : r,b
s = etat(b);
n = r;
existe = true;
répéter
  a = UCB(n);
  n = fils_action(n,a);
  o = observation(s,a);
  b = G(b,a,o);
  n.recompense = R(b,a);
  s = ρPOMDP(s,a);
  si *o∈ fils(n)* **alors**
    n = fils_observation(n,o);
  sinon
    existe = false;
  fin
jusqu'à *terminal(o) OU !existe*;
retourner *n*

Algorithme 1 : la sélection de la branche

Données : un noeud, un état et une observation
Résultat : la nouvelle feuille
Entrées : n,s
ajout_fils(n,o);
n = fils_observation(n,o);
retourner *n*

Algorithme 2 : la création du nœud

Données : un état,un Belief State, une observation et une politique
Résultat : la valeur de l'état s
Entrées : s,b,o,π
v = 0;
tant que *!terminal(s)* **faire**
  v = R(b,π(b));
  o = observation(s,π(b));
  b = G(b,π(b),o);
  s = ρPOMDP(s,π(b));
fin
v = R(b,π(b));
retourner *v*

Algorithme 3 : la simulation de la politique par défaut

Données : une feuille, un nombre
Résultat : une mise à jour de l'arbre
Entrées : n,v
tant que *!racine(n)* **faire**
  n = n.pere;
  n.compteur = n.compteur + 1;
  v = v + n.récompense;
  n.valeur = $\frac{(\text{n.compteur-1}) \times \text{n.val} + \text{v}}{\text{n.compteur}}$;
  n = n.pere;
fin

Algorithme 4 : la mise à jour de l'arbre

Données : la racine et une politique
Résultat : une mise à jour de l'arbre
Entrées : r,π
b = belief_state_initial;
n = selection_branche(r,b) `// b est mis à jour`
o = n.observation;
si *!terminal(o)* **alors**
  n = création_nœud(n,o);
fin
v = simulation_politique(b,π);
mise_à_jour(n,v);

Algorithme 5 : les étapes d'expansion de l'arbre pour ρPOMCP