Planification robuste à l'aide d'une montée de gradient

Olivier Buffet[1,2], Douglas Aberdeen[1,2]

[1] National ICT Australia

[2] Australian National University

prénom.nom@nicta.com.au

http://rsise.anu.edu.au/∼{buffet,daa}

Résumé : Les problèmes réels présentent de vrais défis pour la recherche sur la planification basée sur la théorie de la décision (Decision-Theoretic Planning (DTP)). Une approche classique est de modéliser de tels problèmes sous la forme de problèmes de décision markoviens (MDP), et d'utiliser des techniques de programmation dynamique. Toutefois, deux difficultés majeures apparaissent : 1- la programmation dynamique passe mal à l'échelle quand le nombre de tâches augmente, et 2- le modèle probabiliste peut être incertain, menant à des politiques non sûres. Nous nous basons ici sur les algorithmes de montée de gradient d'une politique pour répondre à la première difficulté, et sur la planification robuste pour répondre à la seconde en utilisant des algorithmes entrainant des agents apprenants adversaires. Le premier agent apprend un plan alors que le second apprend le modèle qui perturbera le plan le plus possible. Dans les travaux récents utilisant des montées de gradient en théorie des jeux, au moins un des deux joueurs peut ne pas converger vers une politique stationnaire. Nous nous concentrons donc sur la convergence du plan robuste seulement, en employant des algorithmes non symétriques.

Mots-clés : Planification robuste, montée de gradient d'une politique, théorie des jeux

1 Introduction

La planification robuste en théorie de la décision (DTP robuste) examine des problèmes dans lesquels le modèle stochastique est incertain, et cherche le meilleur plan face au pire modèle possible. Comme tout problème de DTP peut être transformé en un problème de décision markovien (MDP), les recherches sur la robustesse dans les MDPs (Bagnell *et al.*, 2001; Nilim & Ghaoui, 2004; Buffet & Aberdeen, 2005) est un point de départ logique pour la DTP robuste. Il est en effet possible dans le cadre MDP d'utiliser des algorithmes de programmation dynamique, i.e. de procéder à une optimisation locale (au niveau des états) et obtenir un optimum global.

Toutefois, cet emploi de la programmation dynamique requiert une hypothèse (hypothèse 1 présentée en section 2.3) qui, dans de nombreux problèmes de planifica-

tion, n'est pas vérifiée. Pour cette raison, plutôt que d'utiliser ainsi une traduction du problème sous forme de MDP, nous proposons de chercher directement le pire modèle (et le meilleur plan) avec la formulation DTP originale. Ceci amène à une approche complètement différente de la planification robuste, approche basée sur des algorithmes de montée de gradient pour politiques.

Sans l'hypothèse 1, ce problème de planification robuste est un jeu de Markov à deux joueurs, à somme nulle, et avec observabilité partielle (comme expliqué en section 2.3), dans lequel un joueur doit choisir un plan et l'autre un modèle. Notre première idée était d'apprendre simultanément plan et modèle, en utilisant des algorithmes en-ligne jouant l'un contre l'autre. Si de telles approches "self play" ont eu un certain succès pour des jeux séquentiels (tels que le backgammon (Tesauro, 1994)), il n'existe pas à ce jour d'algorithme garantissant la convergence des politiques (stratégies) vers un équilibre du jeu dans notre cadre partiellement observable. Voici quelques résultats importants dans ce domaine :

- Dans (Singh *et al.*, 2000), Singh et al. étudient la montée de gradient infinitésimale appliquée à des jeux à deux joueurs et deux actions. Pour des jeux à somme nulle, elle génère un comportement oscillant autour de l'équilibre de Nash.
- Dans (Bowling & Veloso, 2002), Bowling et Veloso présentent le principe "Gagne ou Apprend Vite", rendant possible la convergence dans ces mêmes jeux. Mais ces résultats ne se généralisent pas à plus de deux actions, comme par exemple pour le jeu pierre-papier-ciseaux (Roshambo).(Buffet & Aberdeen, 2006)
- Dans (Chang & Kaelbling, 2001), Chang et Kaelbling expliquent comment un apprentissage par montée de gradient peut être trompé par son adversaire si celui-ci utilise des changements soudain de sa politique.
- Dans (Zinkevich, 2003), Zinkevich propose d'utiliser des algorithmes minimisant le regret pour éviter d'être trompé de la sorte.

Mais garantir un regret nul n'implique pas nécessairement une convergence des stratégies en self-play. Dans Roshambo, il est courant de voir les deux joueurs alterner entre pierre-contre-pierre $\rightarrow$ papier-contre-papier $\rightarrow$ ciseaux-contre-ciseaux. Le gain moyen est toujours la valeur de l'équilibre de Nash (0), et le regret est nul puisque chaque joueur ne fait pas pire qu'une stratégie aléatoire.

Dans cet article, nous proposons deux algorithmes pour chercher le meilleur plan face au pire modèle, mais sans chercher le pire modèle en même temps. Après avoir introduit le problème en détails en section 2, la section 3 présente ces deux algorithmes. Ensuite, des expérimentations illustrent leurs comportements respectifs avant une discussion et une conclusion.[1]

2 Contexte

2.1 Problèmes de décision markoviens partiellement observables

Un problème de décision markovien partiellement observable (POMDP) (Cassandra, 1998) est défini ici par un tuple $\langle S, A, T, r, \Omega, O \rangle$. Il décrit un problème de contrôle

[1] De plus amples détails sont présentés dans la version étendue (Buffet & Aberdeen, 2006).

où S est l'ensemble fini des **états** du système considéré et A est l'ensemble fini des **actions** a possibles. Les actions contrôlent les transitions d'un état s à un autre s' selon la dynamique stochastique du système, décrite par la **fonction de transition** T définie par $T(s, a, s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)$. L'objectif est d'optimiser une mesure de performance basée sur la **fonction de récompense** $r : S \times A \times S \rightarrow \mathbb{R}$.[2] De l'état courant, seule une observation partielle o ($\in \Omega$: ensemble fini des observations possibles) est accessible, échantillonnée d'après la distribution de probabilité $O(o, s) = Pr(o|s)$.

Nous préférons ici que nos agents n'utilisent pas de mémoire à court terme dans leur prise décision. Ainsi, notre algorithme d'optimisation doit trouver une **politique** associant à chaque observation une *distribution* de probabilité sur les actions $\pi : \Omega \rightarrow \Pi(A)$ de manière à optimiser la mesure de performance choisie, ici la *récompense* moyenne R gagnée pendant une transition.

Recherche de politique

Du fait de l'observabilité partielle, les approches de programmation dynamique classique s'avèrent inappropriées. Nous leur préférons les algorithmes de recherche directe de politique, qui reviennent souvent à optimiser une fonction (la mesure de performance) dans l'espace des paramètres dont dépend la politique : $R = R(\pi(\vec{\theta}))$. *REINFORCE* de William (Williams, 1992) est le premier algorithme réglant les paramètres d'un contrôlleur connexioniste à l'aide de signaux de renforcement. De tels algorithmes ne dépendent pas nécessairement de la connaissance de l'état exact du système : ils peuvent trouver une politique (localement) optimale utilisant des observations partielles. En outre, les procédures de recherche de politique peuvent trouver la meilleure politique en un temps indépendant de la taille de l'espace d'état.

Nous utilisons principalement des algorithmes de montée de gradient à base de simulations tels que décrits par Baxter et al. (Baxter & Bartlett, 2001; Baxter *et al.*, 2001). A chaque pas de simulation, ils mettent à jour une trace d'éligibilité $\mathbf{e}$ gardant trace des directions de gradient suivies dans le passé :

$$\mathbf{e}_{t+1} = \beta \mathbf{e}_t + \frac{\nabla \pi_{a_t}(\vec{\theta}_t, o_t)}{\pi_{a_t}(\vec{\theta}_t, o_t)}. \tag{1}$$

Ici, nous considérons principalement deux algorithmes :

– `GPOMDP`, lequel estime le gradient comme suit :

$$\nabla R_{t+1} = \nabla R_t + \frac{1}{t+1}[r_t \mathbf{e}_{t+1} - \nabla R_t], \tag{2}$$

et doit être alterné avec un pas de suivi du gradient pour modifier les paramètres ; et

– `OL-POMDP`, lequel est continuellement en train de ré-estimer la trace d'éligibilité tout en renforçant simultanément les paramètres par :

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha_t r_t \mathbf{e}_{t+1}, \tag{3}$$

[2]Comme le modéle est insuffisament connu, nous ne faisons pas l'hypothèse usuelle $r(s, a) = \mathbb{E}_{s'}[r(s, a, s')]$. Cette espérance dépend du vrai modèle (inconnu).

où α_t est taux d'apprentissage.

2.2 Planification basée sur la théorie de la décision

Nos domaines de planification sont décrits par : un ensemble de variables d'état $\mathcal{V} = \{v_1 \in V_1, \ldots, v_{|\mathcal{V}|} \in V_{|\mathcal{V}|}\}$, et un ensemble de tâches $\mathcal{T} = \{T_1, \ldots, T_{|\mathcal{T}|}\}$. L'état courant du système est décrit par les valeurs prises par les variables, par le pas de temps courant et par les tâches actuellement actives.

Dans un état donné, une tâche T peut être déclenchée (est éligible) si certaines conditions relatives aux variables d'état sont vérifiées (si certaines propositions sont vraies ou certaines ressources sont présentes en quantités suffisantes). Une fois la durée de la tâche écoulée, l'une des conséquences possibles $out_T(1), out_T(2) \ldots$ (out pour "outcome") survient, en fonction de la distribution $Pr(Out_T)$. Cette conséquence modifie les valeurs de certaines variables d'état.

Dans notre cadre, les tâches peuvent être actives simultanément, et une récompence est associée à chaque conséquence d'une tâche.

Un problème de planification est spécifié par un état initial s_0, la fonction de récompense et un ensemble d'états buts. Ces derniers peuvent être décrits par des conditions sur les valeurs des variables d'états ou avec une durée limite du plan (de sorte qu'un état but est nécessairement atteint). L'objectif est alors d'atteindre un état but en maximisant la récompense moyenne, ce qui nous place dans un cadre de théorie de la décision. Des problèmes similaires ont été abordés par des planificateurs tels que Tempastic (Younes & Simmons, 2004), un planificateur d'opérations militaires (Aberdeen *et al.*, 2004), CPTP (Mausam & Weld, 2005), Prottle (Little *et al.*, 2005), et FPG (Aberdeen, 2005).

Note : pour des raisons de lisibilité, la description qui précède est une version simplifiée du modèle employé.

DTP avec des MDPs

Une façon simple de traduire un tel problème de planification par un MDP est, comme dans (Hoffmann & Nebel, 2001), de définir :

- Les états comme les instants où certaines tâches se terminent, créant une nouvelle situation dans laquelle une décision peut être prise. Un état reste donc défini par : les valeurs des variables d'état, le pas de temps et les tâches en cours d'exécution.
- Les actions comme les décisions de déclencher un sous-ensemble des tâches éligibles. Ne déclencher aucune tâche est valide, puisque cela revient à attendre le prochain point de décision.
- Les probabilités de transition dépendent des probabilités des conséquences de chaque tâche.
- Les récompenses dépendent des récompenses associées aux conséquences et aux états buts.

La figure 1 montre un problème de planification en cours d'exécution et un MDP équivalent dans lequel s_1 correspond à l'état au temps t, l'action a_2 déclenche les tâches T_2 et T_3, et amène à s_5 quand les conséquences $out_{T_1}(2)$ et $out_{T_3}(1)$ surviennent à $t + 18$.

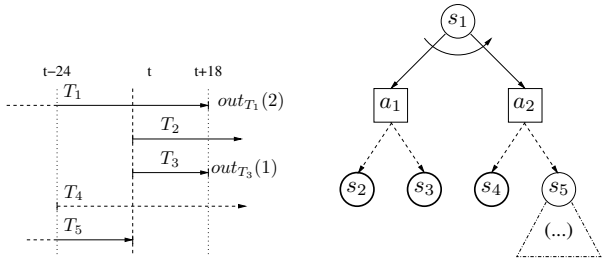

FIG. 1 – Un problème de planification et un simple MDP

2.3 Robustesse

Parce qu'ils sont souvent obtenus en utilisant une connaissance experte ou des statistiques, les modèles sont susceptibles d'être incertains. Cette section explique comment modéliser cette incertitude et comment la prendre en compte dans le problème de planification.

Modéliser l'incertitude

Notre modélisation de l'incertitude ne dit que si un modèle est possible ou pas, sans dire la probabilité de ce modèle. Pour notre problème de planification de départ, nous choisissons de modéliser l'incertitude sur les distributions de probabilité à l'aide d'intervalles :

$$Pr(out_T(k)) \in [Pr^{\min}(out_T(k)), Pr^{\max}(out_T(k))],$$

sachant que la somme des probabilités doit être un : $\sum_k Pr(out_T(k)) = 1$. Ce choix est motivé par la simplicité de cette représentation, et parce que connaissance experte comme statistique amènent facilement à de tels intervalles. Pour une tâche donnée, les bornes supérieures et inférieures sur la probabilité de chaque conséquence donnent les contraintes sur les modèles possibles.

La figure 2 illustre ces contraintes pour trois conséquences. Le triangle est ici un simplex de probabilité représentant toutes les distributions de probabilité à trois conséquences ($Pr(out_i) = 1$ au sommet out_i). Le trapèze horizontal donne la contrainte de l'intervalle lié à out_i. Les modèles possibles sont définis par l'intersection des trois trapèzes.

Une caractéristique intéressante de l'ensemble résultant est sa convexité, ce qui est utile pour un problème d'optimisation. Pour cette tâche, l'adversaire doit trouver le "pire" point dans cet ensemble.

De même, les (PO)MDPs incertains sont souvent modélisés en spécifiant des intervalles pour les probabilités de transition :

$$T(s, a, s') \in [Pr^{\min}(s'|s, a), Pr^{\max}(s'|s, a)].$$

Il est important d'observer qu'il n'y a pas d'équivalence directe entre les modéles incertain pour DTP ou MDP. En effet, puisqu'une tâche du problème de planification peut être déclenchée dans diverses situations, changer la distribution de probabilité sur

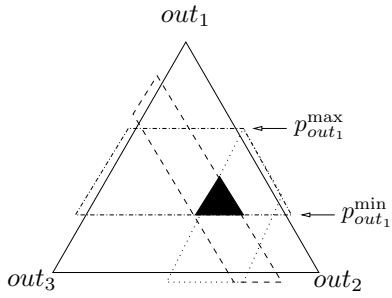

FIG. 2 – Le triangle noir représente toutes les distributions de probabilité possibles pour les conséquences d'une tâche.

ses conséquences altèrera plusieurs distributions de probabilités $T(s, a, \cdot)$. Ainsi, pour qu'un MDP incertain représente correctement un problème de DTP incertain, il faut ajouter des contraintes reliant les distributions $T(s, a, \cdot)$.

Problème

Pour obtenir un plan *robuste*, il faut trouver le meilleur plan face aux pires modèles possibles. En notant l'ensemble des modèles possibles par M, cela amène à résoudre :

$$\arg\max_{\pi\in\Pi} \min_{m\in M} R(\pi, m). \tag{4}$$

On peut efficacement résoudre cette équation pour un MDP incertain par programmation dynamique si l'hypothèse suivante est vérifiée :

Hypothèse 1 : *Les distributions de probabilité $T(s, a, \cdot)$ sont indépendantes d'une paire état-action à l'autre.*

Toutefois, nous venons de voir qu'une bonne traduction d'un problème de planification incertain en un MDP incertain requiert l'ajout de contraintes qui rompent cette hypothèse. Celà fait perdre l'intérêt pour les MDP, et nous incite à regarder vers d'autres techniques d'optimisation, avec la difficulté que nous traitons ici un jeu.

Les règles du jeu

Ce jeu est un jeu à deux joueurs et à somme nulle, un joueur (le "planificateur") cherchant une politique maximisant la récompense moyenne, et le second (l'"adversaire") cherchant un modèle minimisant cette même récompense moyenne. L'adversaire a les contraintes suivantes :

- Si les conséquences de deux tâches différentes sont décidées en même temps, elles doivent l'être de manière indépendante. Ainsi, on peut factoriser l'adversaire en une "équipe adverse" : un adversaire par tâche.
- La distribution de probabilité sur les conséquences d'une tâche ne doit pas dépendre de l'état courant. Chaque adversaire doit être aveugle (pas d'observation), ce qui justifie le caractère partiellement observable de notre problème.

3 Algorithmes

Comme expliqué en sections 1 et 2, le cadre de théorie des jeux et l'observabilité partielle suggèrent d'utiliser des algorithmes de recherche de politique plutôt que la programmation dynamique. C'est aussi la direction suivie par FPG (Aberdeen, 2005) pour concevoir un planificateur passant facilement à l'échelle, mais avec la différence que FPG effectue une factorisation du côté du planificateur quand nous la plaçons du côté adverse.

Ici, on acquiert de l'expérience en simulant le système. Dans ce but, nous utilisons une boucle de simulation générique dans nos différents algorithmes de planification robuste. Comme le montre l'algorithme 1, elle prend comme paramètres les fonctions que le planificateur et ses adversaires doivent utiliser pour apprendre : le renforcement d'OL-POMDP (*reinforcement :* Eq. (1)+(3)), la mise à jour du gradient de GPOMDP (`gradientUpdate` **:** Eq. (1)+(2)), ou rien (`nothing`).

Algorithm 1 SimulationLoop($algoPlan$, $algoOpp$)

```
1:  s = initial state
2:  while s.goal ≠ true do
3:      o = getObservation(s)
4:      a = plan.getAction(o)
5:      s' = findSuccessor(s, a)
6:      r = getReward(s, a, s')
7:      plan.algoPlan(r)
8:      for all a' ∈ A do
9:          opp(a').algoOpp(r)
10:     s ← s'
```

La simulation du système a lieu essentiellenemt dans la fonction `findSuccessor()` (voir algorithme 2), basée sur le même code que FPG. Etant donné un état s et une action a, elle échantillonne le prochain état à l'aide du modèle courant du domaine. Pour simplifier un peu le problème, seule une action peut être déclenchée à un instant donné (mais il peut y avoir plusieurs actions en cours d'exécution). Dans cet algorithme, c'est quand la conséquence d'une tâche est choisie que l'"adversaire" correspondant prend une décision (modifie son modèle).

Nous pouvons maintenant montrer comment la boucle de simulation est utilisée par deux algorithmes de planification robuste. L'idée clef est que l'instabilité de l'adversaire aide à stabiliser la politique du planificateur.

3.1 Apprentissages alternés

Le premier algorithme, *Sequential Robust Policy-Gradient* (seqRPG), est une traduction directe de l'équation 4 en un algorithme basé sur des montées de gradient pour politiques. Sa boucle externe effectue une montée de gradient pour optimiser la politique π (donc à maximiser $\min_{m \in M} R(\pi, m)$) à travers son vecteur de paramètres $\vec{\theta}$. A chaque itération de cette boucle :

Algorithm 2 findSuccessor(s :state, a :action)

```
1: s.addEvent(a, sample-outcome, s.time+a.duration)
2: for all f ∈ instantEffects(a) do
3:    s.processEffect(f)
4: for all f ∈ delayedEffects(a) do
5:    s.addEvent(f, task-effect, s.time+f.delay)
6: repeat
7:    if s.time > maximum makespan then
8:       s.failure=true
9:       return
10:   if s.operationGoalsMet() then
11:      s.goal=true
12:      return
13:   if ¬s.anyEligibleTasks() & s.noEvent() then
14:      s.failure=true
15:      return
16:   event = s.nextEvent()
17:   s.time = event.time
18:   if type(event) = task-effect then
19:      s.processEffect(event.effect)
20:   else if type(event) = sample-outcome then
21:      out=sampleOutcome(event)
22:      for all f ∈ instantEffects(out) do
23:         s.processEffect(f)
24:      for all f ∈ delayedEffects(out) do
25:         s.addEvent(f, task-effect, s.time+f.delay)
26: until s.isDecisionPoint()
```

– lignes 2-3 : une première boucle interne cherche le pire modèle pour la politique courante (c'est là que $\min_{m \in M} R(\pi, m)$ est calculé),
– lignes 4-6 : une seconde boucle interne estime le gradient par rapport à $\vec{\theta}$ de la récompense moyenne espérée, et
– ligne 7 : un pas suivant le gradient est effectué pour mettre à jour les paramètres de la politique.

Au début de l'apprentissage, il pourrait être utile de ne pas ré-initialiser l'estimation du gradient entre deux passages dans la boucle externe, puisque son évolution sera probablement continue. Toutefois, après un certain temps, le vecteur $\vec{\theta}$ va se mettre à osciller autour d'un point d'équilibre, la direction du gradient devenant très instable.

Un défaut apparent de cet algorithme est qu'il ré-optimise le modèle à chaque itération, ce qui suggère d'importants temps de calcul.

Algorithm 3 Sequential Robust Policy-Gradient

```
1: while θ⃗ not converged do
2:    while model not converged do
3:       SimulationLoop( nothing(), reinforcement())
4:    ∇_θ⃗ R = 0
5:    while ∇_θ⃗ R not converged do
6:       SimulationLoop( gradientUpdate(), nothing())
7:    plan.followGradient()
```

3.2 Apprentissages simultanés

Le second algorithme, *Simultaneous RPG* (simRPG), cherche à réduire cette complexité temporelle. Comme décrit dans l'algorithme 4, il consiste à laisser tous les agents apprendre en-ligne simultanément. Mimant l'algorithme 3, son idée principale est que, pour que le plan converge, le pas d'apprentissage du planificateur α_{pla} doit décroître plus vite que celui des adversaires α_{opp} : $\alpha_{pla}(t)/\alpha_{opp}(t) \rightarrow 0$.

Algorithm 4 Simultaneous Robust Policy-Gradient

```
1: while θ⃗_pla and θ⃗_opp not converged do
2:    SimulationLoop( reinforcement(), reinforcement())
```

L'apprentissage plus rapide des adversaires assure que le planificateur est stabilisé autour d'un point d'équilibre. En effet, dès que le planificateur s'éloigne d'une telle position, les adversaires adaptent rapidement le modèle, incitant le planificateur à faire marche arrière. Avec un rapport $\alpha_{pla}(t)/\alpha_{opp}(t)$ fixe, cela amène à des oscillations périodiques (voir IGA avec des jeux à deux actions dans (Singh *et al.*, 2000) ou avec Roshambo dans (Buffet & Aberdeen, 2006)), même avec des pas de longueur décroissante. Un rapport décroissant est nécessaire pour attenuer progressivement les oscillations du côté du planificateur.

Une solution pour que ces pas d'apprentissages vérifient 1- l'hypothèse habituelle $\sum_{t=0}^{\infty} \alpha(t) > \infty$ et 2- $\sum_{t=0}^{\infty} \alpha(t)^2 < \infty$, est de les définir par $\alpha_{pla}(t) = t^{-\beta}$ et $\alpha_{opp}(t) = t^{-\beta'}$ avec $0 < \beta' < \beta < 1$.

4 Expérimentations

Ces expérimentations, conduites sur un Pentium IV à 2.8 GHz, ont pour objectif principal d'illustrer les comportements de seqRPG et simRPG, ainsi que de présenter des problèmes types de planifications robuste. La politique du planificateur est un réseau linéaire fournissant des probabilités grâce à une fonction softmax, les paramètres appris étant les poids (Aberdeen, 2005). Les politiques des adversaires sont des tables avec un paramètre par probabilité.[3] Comme pour FPG, l'algorithme n'énumère pas l'espace

[3]Pour des raisons pratiques (gérer les intervalles facilement), les adversaires utilisent l'algorithme GIGA (Zinkevich, 2003) au lieu d'OL-POMDP. GIGA ne peut être employé pour le planificateur car il ne permet

d'états, ce qui conduit à une faible consommation de mémoire.

Dans seqRPG, le critère d'arrêt des boucles internes est une durée limite de 1000 pas de simulation (pour apprendre le modèle *et* estimer le gradient). Dans chaque expérimentation, des statistiques sont collectées toutes les T itérations de la boucle externe (de seqRPG comme simRPG), et l'algorithme est exécuté pendant une durée fixe t (donnée en secondes). Enfin, les pas d'apprentissage α_{pla} et α_{opp} sont constants. Ces quatre valeurs sont données sous chaque graphique.

4.1 Matching pennies & problème de file d'attente

Matching pennies

Nous commençons avec un premier jeu de matrice transformé en problème de planification. C'est une version non symétrique de "matching pennies" utilisant la matrice de gain suivante :

$$\begin{bmatrix} -1 & +\frac{1}{2} \\ +1 & -\frac{1}{2} \end{bmatrix},$$

où le planificateur choisit la colonne. La stratégie optimale est de jouer 2/3 face et 1/3 pile (1ère et 2nde colonnes).

Les figures 3 et 4 montrent l'évolution de : la probabilité que le planificateur joue face ou pile (`planner1` et `planner2`), la probabilité que l'adversaire joue face (`opp`) et la récompense moyenne (`R`).

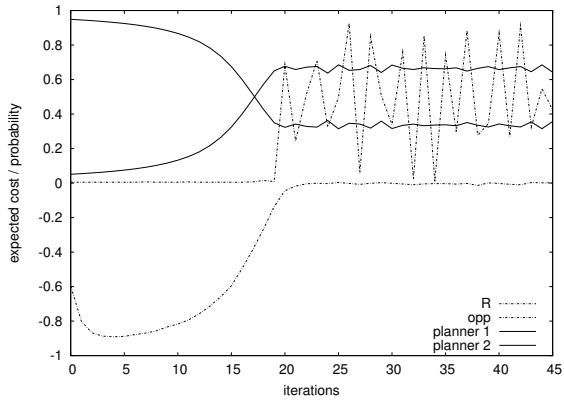

FIG. 3 – Une exécution de seqRPG sur "matching pennies"
$\alpha_{pla} = .1 \qquad \alpha_{opp} = .1 \qquad T = 10 \qquad t = 60s$

Les deux cas montrent une première phase pendant laquelle le planificateur se déplace lentement vers sa position d'équilibre (la stratégie de l'équilibre de Nash), et une seconde phase pendant laquelle il oscille lentement, étant stabilisé par les oscillations plus fortes de l'adversaire. La durée de la première phase (environ 30s pour seqRPG et 5s

pas d'utiliser une fonction d'approximation.

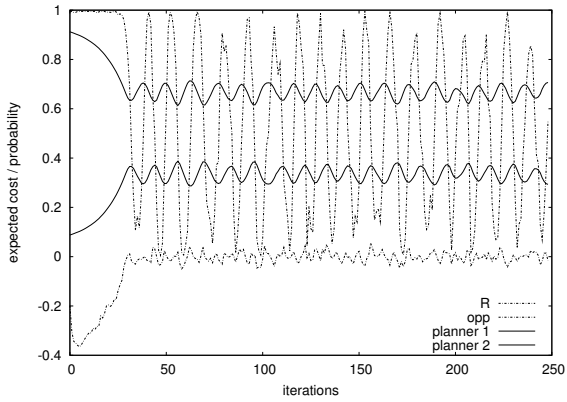

FIG. 4 – Une exécution de simRPG sur "matching pennies"
$\alpha_{pla} = .0001 \qquad \alpha_{opp} = .1 \qquad T = 1000 \qquad t = 20s$

pour simRPG) illustre le fait que l'algorithme simultané converge en général plus vite (confirmé dans toutes les autres expérimentations).

Problème de file d'attente

Ce second problème amène à une situation assez similaire à celle de "matching pennies", mais dans un problème de planification robuste pratique. Celui-ci met en jeu deux files d'attentes de jobs : Q_1 et Q_2, chacun d'une capacité de 5 jobs et initialement vide. A chaque pas de temps, le planificateur doit envoyer un job à l'une de ces files alors qu'un autre est pris dans une file pour être exécuté (Q_1 avec probabilité p), la file choisie pouvant être vide. L´adversaire contrôle p. Personne ne connait le contenu des files.

La récompense est $+.1$ pour chaque job consommé, et -1 pour chaque job perdu à cause d'une file pleine. Un état but est atteint quand un job est perdu ou après 50 pas de simulations, de sorte que 10 jobs au plus peuvent être exécutés.

Les deux algorithmes apprennent le comportement approprié équilibrant l'utilisation des deux files. Mais, comme on peut le voir sur les courbes de la figure 5, l'apprentissage est notablement ralenti (par rapport à "matching pennies"). C'est lié à la durée bien plus longue du plan, laquelle rend le renforcement des décisions plus difficile pour le planificateur comme pour l'adversaire.

4.2 Roshambo

Ici, nous considérons le jeu pierre-papier-ciseaux avec la matrice de gains habituelle :

$$\begin{bmatrix} 0 & +1 & -1 \\ -1 & 0 & +1 \\ +1 & -1 & 0 \end{bmatrix}.$$

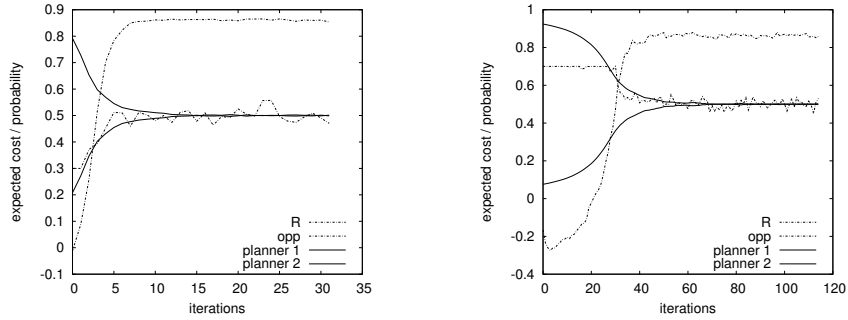

FIG. 5 – Une exécution de seqRPG et simRPG sur le problème de file d'attente
$\alpha_{pla} = 1.|.0001 \qquad \alpha_{opp} = .1|.1 \qquad T = 10|1000 \qquad t = 600|120s$

La figure 6 montre la trajectoire des politiques des deux joueurs avec des pas d'apprentissage différents pour les adversaires. Le triangle supérieur gauche est le simplex pour l'adversaire, et l'inférieur gauche est, en miroir, un simplex pour le planificateur. Plus les adversaires sont rapides, plus ils réagissent vite à la politique du planificateur, laquelle se stabilise.

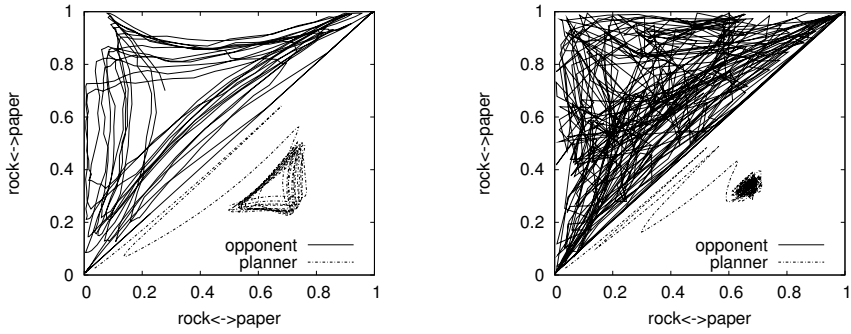

FIG. 6 – Deux exécutions de simRPG. Seul α_{opp} change.
$\alpha_{pla} = .0001 \qquad \alpha_{opp} = .01|.1 \qquad T = 1000 \qquad t = 90s$

4.3 Problème avec 2 maxima

Ce dernier problème illustre le fait qu'il peut y avoir plusieurs optima locaux. Comme illustré sous la forme d'un MDP sur la figure 7, le planificateur n'a pas de choix d´action, et le modèle n'a qu'un paramètre p. a_0, a_1 et a_2 ayant pour coûts respectifs c_0, c_1 et 0, le coût-au-but optimal est (en fonction de p) :

$$V_{s_0}(p) \quad = \quad \frac{c_0}{p} + \frac{c_1}{(1-p)}.$$

TAB. 1 – Pourcentage d'exécutions convergeant vers $p = .1$
($c_0 = -.1 \quad \alpha_{pla} = .0001 \quad \alpha_{opp} = .1 \quad T = 1000 \quad t = 20s$)

	$c_1 = c_0$	$c_1 = 2c_0$	$c_1 = 3c_0$
Théorie	50.0%	41.4%	36.6%
Expérience	47.2%	39.0%	36.0%

Pour passer à l'optimisation d'une récompense moyenne, on fixe un horizon (50 pas de simulation), de sorte que $R \simeq V_{s_0}(p)/50$. Ainsi, des maxima locaux sont obtenus pour $p = 0$ ou $p = 1$. Pour des raisons pratiques, on restreint $p \in [.1, .9]$.

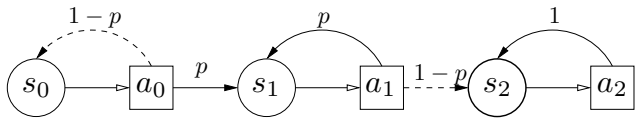

FIG. 7 – Problème avec 2 optima déterministes

La figure 8 montre une exécution de seqRPG et simRPG, seules la récompense moyenne (R) et p (opp) étant représentés. Dans le cas présent, les deux algorithmes convergent vers le même optimum local : $p = .1$.

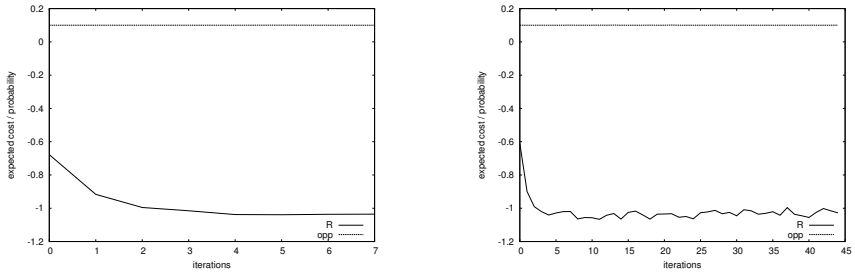

FIG. 8 – Une exécution de seqRPG (gauche) et simRPG (droite)
$\alpha_{pla} = .1|.0001 \qquad \alpha_{opp} = .01|.1 \qquad T = 10|1000 \qquad t = 60s|20s$

Pour vérifier le comportement de simRPG, nous avons rassemblé des statistiques sur le nombre de fois où il converge vers chaque côté de l'intervalle. Nous avons effectué trois séries de 500 exécutions durant 20 secondes chacune, et chacune se terminant avec $p = .1 \pm .005$ ou $p = .9 \pm .005$. Une fonction de coût différente a éte utilisée dans chaque série. Comme le montrent les résultats (table 1), cette expérience réfléchit correctement la probabilité théorique de converger vers $p = .1$.

5 Discussion et conclusion

Les résultats expérimentaux confirment que la montée de gradient robuste pour politique rend possible la convergence du plan à une position d'équilibre, qu'il s'agisse de la version séquentielle ou simultanée de l'algorithme. Et, comme prédit, la premiére est nettement plus gourmande en temps que la seconde.

Ces expérimentations ont été conduites sur des problèmes choisis pour illustrer des difficultés type de la planification robuste, nécessitant des politiques stochastiques ou avec plusieurs optima. L'algorithme devrait bien passer à l'échelle, en particulier si l'on adopte la formulation factorisée de FPG (Aberdeen, 2005). Mais des expérimentations supplémentaires sur de gros problèmes seront nécessaires pour confirmer ce point.

Une difficulté dans notre cadre générique est que, comme dans la section 4.3, il peut y avoir plusieurs optima locaux. C'est différent quand l'hypothèse 1 est vérifiée, puisqu'il n'y a alors qu'une classe de pires modèles équivalents contre lesquels tous les plans robustes font aussi bien.

Supposant que l'on a un optimum global, il est possible de converger vers le pire modèle en échangeant les rôles entre le planificateur et ses adversaires dans seqRPG ou simRPG. L'idée est, ayant trouvé un vecteur $\vec{\theta}^*$ correspondant à un plan robuste, de contraindre le vecteur $\vec{\theta}$ dans un voisinage de $\vec{\theta}^*$ et de stabiliser désormais le modèle à l'aide d'une politique instable (et non plus l'inverse).

Enfin, cette approche ne semble pas spécifique à la planification robuste avec des modèles incertains et pourrait être utilisée aussi bien en planification avec adversaires. Cela conduit à une nouvelle direction à explorer : l'extension de cette approche pour des jeux à somme général avec plusieurs agents planifiant chacun avec un objectif différent.

Conclusion

Des travaux récents montrent que l'incertitude du modèle est un problème important en planification. Toutefoism la plupart des approches sont basées sur une hypothèse non satisfaite par de nombreux cadres réalistes. Nous avons proposé deux algorithmes trouvant des plans robustes, alors que des algorithmes de montée de gradient usuels peuvent générer des comportements oscillant en self-play. Nous faisons la démonstration de seqRPG et simRPG sur des problèmes incertains types afin de fournir une meilleure compréhension de ces problèmes comme des deux algorithmes employés.

Acknowledgements

Merci à Frédérick Garcia pour une discussion des plus bénéfiques, ainsi qu'aux relecteurs pour leurs commentaires des plus constructifs.

National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology and the Arts, and the Australian Research Council through Backing Australia's Ability and the ICT Centre of Excellence program. This work is also supported by the Australian Defence Science and Technology Organisation.

Références

ABERDEEN D. (2005). Policy-gradient methods for planning. In *Advances in Neural Information Processing Systems 19 (NIPS'05)*.

ABERDEEN D., THIÉBAUX S. & ZHANG L. (2004). Decision-theoretic military operations planning. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS'04)*.

BAGNELL J., NG A. Y. & SCHNEIDER J. (2001). *Solving Uncertain Markov Decision Problems*. Rapport interne CMU-RI-TR-01-25, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA.

BAXTER J. & BARTLETT P. (2001). Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research*, **15**, 319–350.

BAXTER J., BARTLETT P. & WEAVER L. (2001). Experiments with infinite-horizon, policy-gradient estimation. *Journal of Artificial Intelligence Research*, **15**, 351–381.

BOWLING M. & VELOSO M. (2002). Multiagent learning using a variable learning rate. *Artificial Intelligence*, **136**, 215–250.

BUFFET O. & ABERDEEN D. (2005). Robust planning with (L)RTDP. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05)*.

BUFFET O. & ABERDEEN D. (2006). *Policy-Gradient for Robust Planning*. Rapport interne. [extended version of CAp'06 paper].

CASSANDRA A. R. (1998). *Exact and Approximate Algorithms for Partially Observable Markov Decision Processes*. PhD thesis, Brown University, Department of Computer Science, Providence, RI.

CHANG Y.-H. & KAELBLING L. (2001). Playing is believing : the role of beliefs in multi-agent learning. In *Advances in Neural Information Processing Systems 14 (NIPS'01)*.

HOFFMANN J. & NEBEL B. (2001). The FF planning system : Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, **14**, 253–302.

LITTLE I., ABERDEEN D. & THIÉBAUX S. (2005). Prottle : A probabilistic temporal planner. In *Proceedings of the Twentieth American National Conference on Artificial Intelligence (AAAI'05)*.

MAUSAM & WELD D. (2005). Concurrent probabilistic temporal planning. In *Proceedings of the Fifteenth International Conference on Planning and Scheduling (ICAPS'05)*.

NILIM A. & GHAOUI L. E. (2004). Robustness in Markov decision problems with uncertain transition matrices. In *Advances in Neural Information Processing Systems 16 (NIPS'03)*.

SINGH S., KEARNS M. & MANSOUR Y. (2000). Nash convergence of gradient dynamics in general-sum games. In *Proceedings of the Sixteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI'00)*, p. 541–548.

TESAURO G. (1994). *TD-Gammon, a self-teaching backgammon program, achieves master-level play*. Rapport interne, IBM, Thomas J. Watson Research Center, Yorktown Heights, NY 10598.

WILLIAMS R. (1992). Simple statistical gradient-following algorithms for connectionnist reinforcement learning. *Machine Learning*, **8**(3), 229–256.

YOUNES H. L. S. & SIMMONS R. (2004). Policy generation for continuous-time stochastic domains with concurrency. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS'04)*.

ZINKEVICH M. (2003). Online convex programming and generalized infinitesimal gradient ascent. In *Proceedings of the Twentieth International Conference on Machine Learning (ICML'03)*.