Apprentissage par renforcement rapide pour des grands ensembles d'actions en utilisant des codes correcteurs d'erreur

Gabriel Dulac-Arnold, Ludovic Denoyer, Philippe Preux, Patrick Gallinari

Apprentissage par renforcement rapide pour des grands ensembles d'actions en utilisant des codes correcteurs d'erreur

Gabriel Dulac-Arnold[1], Ludovic Denoyer[1], Philippe Preux[2], Patrick Gallinari[1]

[1] UPMC-LIP6, Case 169
4 Place Jussieu
Paris 75005 — France
firstname.lastname@lip6.fr

[2] LIFL (UMR CNRS) & INRIA Lille Nord-Europe
Villeneuve d'Ascq — France
firstname.lastname@inria.fr

Abstract : L'utilisation de l'apprentissage par renforcement (AR) pour la résolution de problèmes réalistes se heurte à des questions de passage à l'échelle. La plupart des algorithmes d'AR sont incapables de gérer des problèmes avec des centaines, voire des milliers d'actions, ce qui en limite l'application dans la pratique. Nous considérons le problème d'AR dans le cadre de l'apprentissage supervisé où la politique optimale est obtenue sous la forme d'un classeur multi-classes, l'ensemble des classes correspondant à l'ensemble des actions du problème. Nous introduisons l'utilisation de codes correcteurs d'erreurs (CCE) dans ce contexte et proposons deux nouvelles méthodes pour réduire la complexité de l'apprentissage en utilisant des approches à base de *rollouts*. La première de ces méthodes consiste à introduire un classeur basé sur des CCE comme classeur multi-classes, ce qui réduit la complexité de l'apprentissage de $\mathcal{O}(A^2)$ à $\mathcal{O}(A\log(A))$. Ensuite, nous proposons une seconde méthode qui met à profit le dictionnaire de codage du CCE pour découper le PDM initial en $\mathcal{O}(\log(A))$ PDM à 2 actions. Cette seconde méthode réduit la complexité de l'apprentissage de $\mathcal{O}(A^2)$ à $\mathcal{O}(\log(A))$ ce qui permet de traiter en des temps très raisonnables des problèmes avec un grand nombre d'actions. Nous terminons avec une démonstration expérimentale de l'intérêt de notre approche.

1 Introduction

L'objectif de l'apprentissage par renforcement (AR) et plus généralement des processus de décision séquentielle, est d'apprendre une politique optimale permettant d'optimiser une certaine fonction objectif pour un agent interagissant avec son environnement. On suppose ici que cet environnement peut se modéliser à l'aide d'un processus de décision de Markov. En AR, la dynamique de l'environnement est inconnue, de même que la fonction de récompense immédiate. Cela entraîne que pour obtenir une politique optimale, l'apprenant doit interagir avec son environnement. Quoique bien compris d'un point de vue théorique, l'AR doit toujours affronter de nombreuses questions pratiques liées à la complexité de l'environnement ; l'une de ces difficultés pratiques est constituée par un ensemble d'actions fini, de grande taille (100, 1000, 10000 par exemple). Actuellement, l'utilisation d'approximateurs de fonctions pour représenter cette politique et généraliser l'apprentissage à l'ensemble des états est une approche très commune. Cependant, le cas du grand ensemble d'actions a été beaucoup moins exploré à ce jour et demeure un défi.

Quand le nombre d'actions, noté A, n'est ni restreint à quelques unités, ni infini non dénombrable, la situation est difficile. Dans le cas d'un espace d'actions continu, des hypothèses de régularité peuvent être émises sur les conséquences des actions, par exemple une propriété de Lipschitz (Lazaric *et al.*, 2007; Bubeck *et al.*, 2011; Negoescu *et al.*, 2011). Cependant, les situations dans lesquelles l'ensemble des actions est fini et de l'ordre de quelques dizaines à quelques milliers sont abondantes : les échecs, le jeu de Go et nombre de problèmes de planification sont de ceux-ci. Dans le cas général où les conséquences des actions n'ont pas de régularité particulière, il n'est pas possible de généraliser les conséquences d'une action aux actions qui lui sont similaires, ou proches ; le sous-échantillonnage n'est alors pas possible.

Dans cet article, nous présentons un algorithme qui sous-échantillonne adéquatement l'ensemble des actions. S'appuyant sur des idées issues de la classification supervisée multi-classes, nous introduisons une approche originale qui réduit significativement la complexité de l'apprentissage dans le cas de grands ensembles d'actions. En affectant un code multi-bits à chaque action, nous créeons des partitions binaires de l'ensemble d'actions en utilisant des codes correcteurs d'erreurs[1] (CCE) (Dietterich & Bakiri, 1995). Notre approche est ancrée sur l'algorithme *Rollout Classification Policy Iteration* (RCPI) (Lagoudakis & Parr, 2003), algorithme bien connu pour son efficacité sur des problèmes réels. Nous commençons par proposer une manière simple de réduire le coût de l'apprentissage d'une politique optimale en tirant parti de l'uage de CCE. Ensuite, nous étendons cette idée et proposons une nouvelle méthode d'AR qui permet de trouver une politique optimale en résolvant un ensemble de problèmes de décison de Markov (PDM) ayant deux actions indépendemment les uns des autres. Tandis que notre première méthode — RCPI avec CCE (ERCPI) — réduit la omplexité globale de l'apprentissage de $\mathcal{O}(A^2)$ à $\mathcal{O}(A\log(A))$, notre seconde méthode — RCPI binaire (BRCPI) — la réduit encore bien plus, passant à $\mathcal{O}(\log(A))$.

Ce papier est organisé comme suit : nous définissons le problème d'AR et nos notations dans la section 2.1, puis nous introduisons l'algorithme RCPI et les CCE dans les sections 2.2 et 2.3 respectivement. Nous présentons l'idée générale de notre travail dans la section 3. Nous montrons comment RCPI peut être étendu en utilisant des CCE à la section 3.2 et expliquons en détail comment un PDM peut être factorisé pour accélérer RCPI pendant la phase d'apprentissage dans la section 3.3. Une analyse comparative de la complexité des algorithmes est fournie en section 3.4. Des résultats expérimentaux sont fournis sur deux problèmes dans la section 4. Enfin, des travaux en lien avec les nôtres sont décrits en section 5.

2 Pré-requis

Dans cette section, nous présentons les trois éléments clés pour comprendre la contribution décrite dans la suite du papier : les problèmes de décision de Markov, l'algorithme *Rollout Classification Policy Iteration* et les codes correcteurs d'erreurs.

2.1 Processus et problèmes de décision de Markov

Nous rappelons ici quelques éléments sur ces processus. Nous ne cherchons pas à expliquer ici ce que sont ces concepts, mais essentiellement à introduire les notations que nous utilisons par la suite. Le lecteur est invité à consulter un texte de référence comme le livre de M. Puterman Puterman (1994) s'il rssent le besoin, ou l'envie, d'en savoir plus.

Un processus de décision de Markov, noté $\mathcal{M}$, est défini par un quadruplet $\mathcal{M} = (\mathcal{S}, \mathcal{A}, T, R)$, où :

- $\mathcal{S}$ est l'ensemble des états, avec $s \in \mathcal{S}$ dénotant un état du PDM ;
- $\mathcal{A}$ est l'ensemble des actions possibles, avec $a \in \mathcal{A}$ dénotant une action ;
- $T : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ est la fonction de transition du PDM : elle définit la probabilité d'aller de l'état s à l'état s' après avoir effectuer l'action a: $T(s', s, a) = P(s'|s, a)$;
- $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ est la fonction de récompense immédiate, définie par la récompense reçue en moyenne lorsque l'action a est effectuée dans l'état s ; la récompense immédiate effectivement perçue sera notée r.

Dans cet article, nous supposons que l'ensemble d'actions est le même dans tous les états.

Définissons une politique déterministe $\pi : \mathcal{S} \to \mathcal{A}$. Dans ce papier, et sans restriction de généralité, nous considérons que la fonction objectif que l'agent doit optimiser est l'espérance de la somme γ-dépréciée des récompenses immédiates, à partir d'un état appartenant à ensemble donné D : $J_\pi(s) = \mathbb{E}[\sum_{k\geq 0} \gamma^k r_{t+k} | s_t = s \in D, \pi]$.

La performance d'une politique est mesurée par rapport à cette fonction objectif J_π. Le but de l'AR est de trouver une politique optimale π^* qui maximise cette fonction objectif : $\pi^* = argmax_\pi J_\pi$.

[1] ce terme correspond à notre traduction de *Erroc Correcting Output Code*. Quoique proche, il ne s'agît pas exactement de la même idée que celle des codes correcteurs d'erreurs utilisés pour la transmission d'information (CRC, *cf.* `http://fr.wikipedia.org/wiki/Code_correcteur`).

Dans un problème d'AR, l'agent connaît $\mathcal{S}$ et $\mathcal{A}$, mais ne connaît pas la dynamique de son environnement T ni R.

Dans ce papier, nous supposons que l'agent peut démarrer depuis n'importe quel état du PDM et qu'il peut réaliser autant de trajectoires que l'on veut pour réaliser l'apprentissage d'une politique optimale. Ce sont là des hypothèses très communes dans la littérature.

2.2 Rollout Classification Policy Iteration

Nous ancrons notre travail dans le cadre de l'algorithme RCPI introduit par Lagoudakis & Parr (2003). RCPI appartient à la famille des algorithmes d'itération sur les politiques approché (API). Ceux-ci améliorent leur estimation de la fonction Q — $Q_\pi(s,a) = \mathbb{E}[J_\pi(s)|\pi]$ au fil de leurs itérations. Nous utilisons l'indice k pour noter les itérations. À l'itération k, un algorithme API dispose d'une politique π_k ; il l'utilise pour simuler le comportement de l'agent dans son environnement et ainsi récolté des informations sur la fonction de transition et la fonctoin de récompense immédiate. Il utilse celles-ci pour approximer la fonction Q, approximation que nous notons donc $\tilde{Q}_k$, qui appartient à l'espace de fonctions considéré. Cela est obtenu avec une politique initiale (pouvant être aléatoire) π_0 et les itérations sont répétées jusqu'à ce que l'estimation $\tilde{Q}_k$ de Q soit considérée comme suffisamment bonne. $\tilde{Q}_k(s,a)$ est estimée en réalisant K *rollouts*, *i.e.* des simulations de Monte-Carlo utilisant π. À partir de ces échantillons, une régression est effectuée pour généraliser la fonction Q à l'ensemble de l'espace état-action et on obtient $\tilde{Q}_k$. La nouvelle politique π_{k+1} est alors la politique gloutonne par rapport à $\tilde{Q}_k$, c'est-à-dire celle qui choisit l'action ayant la plus grande valeur de $\tilde{Q}_k$ en chaque état.

Dans le cas de RCPI, au lieu d'utiliser un approximateur de fonctions qui estime $\tilde{Q}_k$, la meilleure action pour un état s donné est sélectionnée en utilisant un classeur, sans approximation explicite de Q. Cette estimation est souvent réalisée par un classeur binaire f_θ sur l'espace état-action tel que la nouvelle politique puisse s'écrire :

$$\pi_{k+1}(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, f_\theta(s,a). \tag{1}$$

L'ensemble d'apprentissage du classeur $\mathcal{S}^{\mathrm{T}}$ est engendré par l'échantillonnage de Monte Carlo du PDM, estimant l'action optimale pour chaque état échantillonné. Une fois engendrée, ces paires état-action (s,a) optimales sont utilisées pour entraîner un classeur supervisé ; l'état est interprêté comme un ensemble d'attributs et l'action a comme l'étiquette de la donnée/état. En d'autres termes, RCPI est un algorithme API qui utilise des simulations de Monte Carlo pour transformer le problème de RL en un prolème de classification multi-classes.

2.3 Codes correcteurs d'erreur

Introduits initialement dans Dietterich & Bakiri (1995) en classification supervisée quand le nombre de classes est important, les codes correcteurs d'erreur (CCE) ont été utilisés avec succès depuis un bon nombre d'années. Nous présentons briévement ces CCE, leur mise en œuvre en AR étant détaillée en section 3.2.

Soit un problème de classification supervisée multi-classes dont l'ensemble d'étiquettes est $\mathcal{Y}$, les $|\mathcal{Y}|$ étiquettes peuvent être encodées sous forme d'entiers dont la représentation binaire utilise $C = \log_2(|\mathcal{Y}|)$ bits. L'utilisation de CCE pour la classification supervisée associe un code binaire de longueur[2] $C = \gamma \log(|\mathcal{Y}|)$ où $\gamma \geq 1$.

Le principe de ces classeurs multi-classes avec CCE est d'apprendre à prédire un code plutôt que directement une étiquette. Cela transforme un problème à $|\mathcal{Y}|$ classes en un ensemble de $C = \gamma \log(|\mathcal{Y}|)$ problèmes de classification supervisée binaires (*i.e.* à deux classes). Une fois entraînée, la classe d'une donnée x est obtenue en fournissant cette donnée en entrée de chacun des classeurs et en concaténant leur sortie respective (un bit chacun) pour produire le codage de l'étiquette prédite : $code(x) = (f_{\theta_0}(x), \cdots, f_{\theta_C}(x))$. L'étiquette prédite est ainsi l'étiquette dont le code est le plus proche en termes de distance de Hamming. Notons que cette détermination du code le plus proche peut être réalisée en temps logarithmique en utilisant des approches à base d'arbres, tels des *k*-d trees Bentley (1975). Aussi, en utilisant des CCE pour la classification, la prédiction de la classe d'une donnée est de complexité $\mathcal{O}(log(|\mathcal{Y}|)$.

[2]Différentes méthodes existent pour générer ces codes. En pratique, on utilise généralement un codage redondant avec $\gamma \approx 10$.

	b_1	b_2	b_3
a_1	$+$	$+$	$-$
a_2	$-$	$+$	$-$
a_3	$+$	$-$	$+$
a_4	$-$	$+$	$+$
a_5	$+$	$-$	$-$

Figure 1: Un example de matrice de codage de 5 actions en $C = 3$ bits. Par exemple, le code de l'action 1 est $(+, +, -)$.

3 ERCPI & BRCPI

Nous présentons notre contribution en deux étapes, la seconde s'appuyant sur la première. Nous commençons par expliquer comment des CCE peuvent être aisément intégrés dans une politique à base de classeur et poursuivons en montrant comment la matrice de codage des CCE peut être utilisée pour factoriser RCPI en un algorithme d'apprentissage de complexité bien inférieure à RCPI.

3.1 Idée générale

L'idée générale de nos deux algorithmes tournent autour de l'utilisation de CCE pour représenter un ensemble d'actions $\mathcal{A}$. Cette approche affecte un code de $C = \gamma \log(A)$ bits à chacune des A actions. Les codes sont organisés dans une *matrice de codage*, idée illustrée à la figure 1, matrice que nous notons $\mathbf{M}^c$. Chaque ligne de celle-ci correspond au codage binaire d'une action, tandis que chaque colonne est une dichotomie particulière de l'ensemble des actions, correspondant donc à un bit particulier du codage b_i. En effet, chaque colonne est une projection de l'ensemble des A actions dans un espace à 2 dimensions. Nous notons $\mathbf{M}^c_{[a,*]}$ la a^{th} ligne de $\mathbf{M}^c$, laquelle correspond donc au codage binaire de l'action. $\mathbf{M}^c_{[a,i]}$ correspond au bit b_i du codage de l'action a.

L'idée principale consiste à considérer que chaque bit correspond à une sous-politique binaire dénotée π_i[3]. En combinant ces sous-politiques, nous construisons la politique originale π que l'on cherche à apprendre :

$$\pi(s) = \underset{a \in \mathcal{A}}{\operatorname{argmin}}\, d_{\mathrm{H}}(\mathbf{M}^c_{[a,*]}, (\pi_1(s), \cdots, \pi_C(s)), \tag{2}$$

où $\mathbf{M}^c_{[a,*]}$ est le codage binaire de l'action a et d_{H} dénote la distance de Hamming. Pour un état donné s, chaque sous-politique fournit une action binaire $\pi_i(s) \in \{-, +\}$. L'ensemble de sous-politiques fournit donc un vecteur binaire de taille C. $\pi(s)$ sélectionne l'action a dont le code binaire minimise la distance de Hamming avec la concaténation de ces C bits.

Nous proposons deux variantes de RCPI qui diffèrent dans la manière où chaque sous-politique est apprise. ERCPI remplace la définition traditionnelle de la politique π par la définition donnée dans l'éq. (2). Pour sa part, BRCPI apprend indépendemment chacune des sous-politiques en considérant que chacune est apprise pour un PDM ayant 2 actions. L'algorithme résultant est très rapide.

3.2 ERCPI

ERCPI s'appuie sur la définition d'une politique donnée par l'équation (2) afin de diminuer la complexité de l'apprentissage de RCPI. Les C sous-politiques — $\pi_{i \in [1,C]}$ — sont apprises simultanément sur le PDM original en ajoutant une étape de codage à RCPI, comme cela est décrit dans l'algorithme 1. Comme tout algorithme d'amélioration de politique, ERCPI effectue itérativement les étapes suivantes :

Simulation : cela consiste à effectuer des simulations Monte Carlo des politiques courantes $\pi_{k,i}$ afin d'estimer la qualité d'un ensemble de paires état-action. De ces simulations, un ensemble d'exemples d'apprentissage $\mathcal{S}^{\mathrm{T}}_k$ est généré dans lequel les données sont les états, les étiquettes étant l'action estimée comme la meilleure dans cet état.

Apprentissage : Pour chaque bit b_i, $\mathcal{S}^{\mathrm{T}}_k$ est utilisé pour créer le jeu d'apprentissage d'un problème à 2 classes $\mathcal{S}^{\mathrm{T}}_{k,i}$. Chacun de ces jeux d'exemples $\mathcal{S}^{\mathrm{T}}_{k,i}$ est alors utilisé pour entraîner un classeur $f_{\theta_{k,i}}$ duquel

[3]Ici, nous l'indiquons pas l'indice k de l'itération de l'API. En toute rigueur, on devrait noter $\pi_{k,i}$ mais cet alourdissemnt de notation est inutile ici.

une sous-politique $\pi_{k+1,i}$ sera déduite, selon l'éq. (1). Finalement, les C sous-politiques sont combinées pour fournir une politique améliorée π_{k+1} selon l'éq. (2).

ERCPI est présenté dans l'Alg. 1.

La fonction de **Rollout** utilisée par ERCPI est la même que celle utilisée par RCPI — π_k est utilisée pour estimer la valeur d'une certaine paire état-action $\tilde{Q}_k(s,a)$.

Algorithm 1: ERCPI

Data:
$\mathcal{S}_R$: ensemble d'état uniformément échantillonné ; $\mathcal{M}$: PDM ; π_0: politique initiale ; K: nombre de trajectoires ; T: longueur maximale d'une trajectoire.

1 $\pi \leftarrow \pi_0$
2 $k \leftarrow 0$
3 **repeat**
4 $\mathcal{S}_k^{\mathrm{T}} \leftarrow \emptyset$
5 **foreach** $s \in \mathcal{S}_R$ **do**
6 **foreach** $a \in A$ **do**
7 $\tilde{Q}_k(s,a) \leftarrow \mathbf{Rollout}(\mathcal{M}, s, a, K, \pi)$
8 **end**
9 $a^* \leftarrow \mathrm{argmax}_{a \in \mathcal{A}} \tilde{Q}_\pi(s,a)$
10 **if** $\forall a \neq a^*, \tilde{Q}_k(s,a) \ll \tilde{Q}_k(s,a^*)$ **then**
11 $\mathcal{S}_k^{\mathrm{T}} \leftarrow \mathcal{S}_k^{\mathrm{T}} \cup \{(s,a^*)\}$
12 **end**
13 **end**
14 **foreach** $i \in [1, C]$ **do**
15 $\mathcal{S}_{k,i}^{\mathrm{T}} \leftarrow \emptyset$
16 **foreach** $(s,a) \in \mathcal{S}_k^T$ **do**
17 $a_i \leftarrow \mathbf{M}^c_{[a,i]}$
18 $\mathcal{S}_{k,i}^{\mathrm{T}} \leftarrow \mathcal{S}_{k,i}^{\mathrm{T}} \cup (s, a_i)$
19 **end**
20 $f_{\theta_i} \leftarrow \mathbf{Train}(\mathcal{S}_{k,i}^{\mathrm{T}})$
21 $\pi_{k+1,i}$ depuis f_{θ_i} selon l'éq. (1)
22 **end**
23 π' comme défini par l'éq. (2)
24 $\pi_{k+1} \leftarrow \alpha(\pi_k, \pi')$
25 $k \leftarrow k+1$
26 **until** $\pi_k \sim \pi_{k-1}$;
27 **return** π_k

Jusqu'à la ligne 12 de l'Algorithme 1, ERCPI est identique à RCPI, la seule nuance étant que seule la meilleure paire (s, a^*) est conservée, ce qui est néanmoins classique quand RCPI est utilisé avec un classeur multiclasses.

La différence principale d'ERCPI vis-à-vis de RCPI apparaît à partir de la ligne 13 : le jeu d'exemples d'entraînement $\mathcal{S}_k^{\mathrm{T}}$ y est décomposé selon C ensembles d'actions binaires et chaque sous-politique $\pi_{k,i}$ est apprise. À la ligne 16, on remplace l'étiquette originale de l'état s par son étiquette binaire dans l'ensemble d'actions de $\pi_{k,i}$ — cela correspond au bit b_i du code de l'action a.

La fonction **Train** apparaissant à la ligne 19 correspond à l'entraînement d'un classeur binaire à partir du jeu d'exemples $\mathcal{S}_{k,i}^{\mathrm{T}}$ ce qui fournit la sous-politique $\pi_{k,i}$. Après ce pas, une politique globale π' est définie selon l'éq. (2). Afin d'assurer la stabilité de l'algorithme, la nouvelle politique π_{k+1} est un mélange de l'ancienne politique π_k avec celle nouvellement obtenue par le classeur π' (*cf.* ligne 23) : pour chaque état, on prend l'action déterminée par π_k avec une probabilité α, celle déterminée par π' avec probabilité $1-\alpha$: c'est ce que l'on note avec la fonction $\alpha(\pi_k, \pi')$.

3.3 BRCPI

ERCPI découpe le problème d'amélioration de politique en C problèmes, mais l'entraînement nécessite toujours π_k et requiert donc l'ensemble des sous-politiques. De plus, pour chaque état, les A actions doivent être évaluées par simulation Monte Carlo (*cf.* Alg. 1, ligne 5). Pour réduire la complexité de cet algorithme, nous proposons d'apprendre les C sous-politiques — $\pi_{k,i \in [1,C]}$ — **indépendemment**, transformant le PDM initial en C sous-PDM, chacun correspondant à un environnement dans lequel une certaine politique $\pi_{k,i}$ est exécutée.

Chacune des politiques binaires $\pi_{k,i}$ possède sa propre représentation de l'ensemble d'actions, définie par la colonne correspondante dans la matrice de codage $\mathbf{M}^c$. Pour l'entraînement, la sélection de la meilleure action doit être adaptée dans cet ensemble d'actions binaire et chaque choix effectué par l'une des politiques $\pi_{k,i}$ doit être combinée avec celui des autres politiques pour déterminer l'action choisie dans l'ensemble original d'actions.

Soient $\mathcal{A}_i^+, \mathcal{A}_i^- \subset \mathcal{A}$ les ensembles d'actions associés aux $\pi_{k,i}$, tels que :

$$\begin{aligned} \mathcal{A}_{k,i}^+ &= \{a \in \mathcal{A} \mid \mathbf{M}^c_{[a,i]} = \text{`}+\text{'}\} \\ \mathcal{A}_{k,i}^- &= \mathcal{A} \setminus \mathcal{A}_i^+ = \{a \in \mathcal{A} \mid \mathbf{M}^c_{[a,i]} = \text{`}-\text{'}\}. \end{aligned} \tag{3}$$

Pour un i donné, $\mathcal{A}_{k,i}^+$ est l'ensemble des actions originales correspondant à la sous-action $+$ et $\mathcal{A}_{k,i}^-$ est l'ensemble des actions originales correspondant à la sous-action $-$.

On peut maintenant définir C nouveaux PDM à ensemble d'actions binaires que nous nommons des sous-PDM et les notons $\mathcal{M}_{i\in[1,C]}$. Ils sont définis à partir du PDM original comme suit :

- $\mathcal{S}_i = \mathcal{S}$: le même ensemble d'états que le PDM original ;
- $\mathcal{A}_i = \{+, -\}$.
- $T_i = T(s', s, a)P(a|a_i) = P(s'|s, a)P(a|a_i)$, où $P(a|a_i)$ est la probabilité de choisir l'action $a \in \mathcal{A}^{a_i}$, sachant que la soous-action appliquée sur le sous-PDM $\mathcal{M}_i$ est $a_i \in \{+, -\}$. Nous considérons que $P(a|+)$ est uniforme pour $a \in \mathcal{A}^+$ et nulle pour $a \in \mathcal{A}^-$, et vice versa. $P(s'|s, a)$ est la fonction de transition du PDM original.
- $R_i(s, a_i) = \sum_{a \in \mathcal{A}_i^{a_i}} P(a|a_i)R(s, a)$.

Chacun de ces sous-PDM représente l'environnement dans lequel une politique binaire agit. Chacun de ces sous-PDM est défini indépendemment des autres ; aussi, nous considérons ces sous-PDM comme définissant des problèmes de RL distincts.

Ceci étant posé, nous proposons de transformer RCPI appliqué sur le PDM original en C nouveaux sous-PDM résolu par RCPI, chacun cherchant une politique optimale π_i pour son propre $\mathcal{M}_i$. Une fois que toutes les sous-politiques ont été obtenues, elles peuvent être utilisées durant la phase de sélection de l'action de la manière décrite en section 3.2.

L'avantage principal de cette approche est que puisque chacun des $\gamma \log(A)$ sous-problèmes traités par l'algorithme 2 est modélisé par un PDM à ensemble d'actions binaire, augmenter le nombre d'actions dans le PDM original augmente le nombre de sous-problème logarithmiquement, sans augmenter la complexité chacun de ces sous-problèmes — voir la section 3.4.

Discutons quelques détails de BRCPI tel qu'il est décrit par l'algorithme 2. BRCPI ressemble beaucoup à RCPI si ce n'est qu'au lieu de boucler sur les A actions à la ligne 6, BRCPI échantillonne seulement $\tilde{Q}$ pour les actions $+$ et $-$. Cependant, la boucle interne est exécutée $C = \gamma \log(A)$ fois, comme on le voit à la ligne 1 de l'algorithme 2.

Dans la fonction **Rollout** (ligne 7), si $\pi_{k,i}$ choisit la sous-action '$+$', l'action a_i du PDM original est échantillonnée depuis $\mathcal{A}_i^+$ selon $P(a|a_i)$ et la fonction de transition du PDM est appliquée en utilisant cette action. Cela estime bien la récompense moyenne de choisir l'action $+$ dans l'état s.

Comme nous l'avons vu en section 3.2, chacune des $\mathcal{A}_i$ est une projection différente de l'ensemble d'action original dans un ensemble binaire. Chacun des classseurs $\pi_{k,i}$ fournit donc une décision considérant un découpage différent de l'ensemble d'actions. Certains découpages peuvent n'avoir aucun sens par rapport au PDM en cours de résolution ; dans ce cas, les récompenses immédiates attendus des actions sélectionnées selon cette sous-politique particulière $\pi_{k,i}$ $\mathcal{A}_i^+$ et $\mathcal{A}_i^-$ sont égales. Cela ne pose pas de problème dans la mesure où cette sous-politique particulière sera simplement un bruit qui sera corrigé par des sous-politiques plus pertinentes.

3.4 Complexité des algorithmes RCPI, ERCPI et BRCPI

Dans cette section, nous étudions le coût de calcul des algorithmes proposés, ERCPI et BRCPI et nous les comparons au coût de RCPI.

Algorithm 2: BRCPI

Data:
$\mathcal{S}_R$: ensemble d'état uniformément échantillonné ; $\mathcal{M}$: PDM ; π_0: politique initiale ; K: nombre de trajectoires ; T: longueur maximale d'une trajectoire ; C: nombre de PDM binaires.

foreach $i \in C$ **do**
  $k \leftarrow 0$
  $\pi_{k,i} \leftarrow \pi_0$
  repeat
    $\mathcal{S}_k^{\mathrm{T}} \leftarrow \emptyset$
    foreach $s \in \mathcal{S}_R$ **do**
      foreach $a \in \{+, -\}$ **do**
        $\tilde{Q}_k(s,a) \leftarrow \mathbf{Rollout}(\mathcal{M}_i, s, a, K, \pi_{k,i})$
      end
      $a^* \leftarrow \mathrm{argmax}_{a \in \mathcal{A}} \tilde{Q}_k(s,a)$
      if $\forall a \neq a^*, \tilde{Q}_k(s,a) \ll \tilde{Q}_k(s,a^*)$ **then**
        $\mathcal{S}^{\mathrm{T}} \leftarrow \mathcal{S}^{\mathrm{T}} \cup \{(s,a^*)\}$
      end
    end
    $f_{\theta_i} \leftarrow \mathbf{Train}(\mathcal{S}^{\mathrm{T}})$
    π_i' depuis f_{θ_i} selon l'éq. (1)
    $\pi_{k+1,i} \leftarrow \alpha(\pi_{k,i}, \pi_i')$
    $k \leftarrow k+1$
  until $\pi_{k,i} \sim \pi_{k-1,i}$;
  return π *comme défini par l'éq.* (2) *en utilisant les* $\pi_{k,i}$
end

Table 1: Coût d'une itération de RCPI-OVA, ERCPI, and BRCPI. S est le nombre d'états, A est le nombre d'actions, K est le nombre de rollouts, T est la longueur de la trajectoire, $\mathcal{C}(S,A)$ est le coût d'apprentissage du classeur pour S états et A actions.

Algorithme	Coût des simulations	Coût d'apprentissage
RCPI-OVA	$SAK(TA)$	$A.\mathcal{C}(S)$
ERCPI	$SAK(T\gamma\log(A))$	$\gamma\log(A).\mathcal{C}(S)$
BRCPI	$\gamma\log(A)\,(2SK(2T))$	$\gamma\log(A)\mathcal{C}(S)$

Afin de définir ce coût, nous considérons que $\mathcal{C}(S,A)$ est le temps mis pour apprendre un classeur multi-classes à partir de S exemples avec A étiquettes possibles ; $\mathcal{I}(A)$ est alors le coût de classification d'une donnée.

Le cout de calcul d'une itération de RCPI et de ERCPI est composé d'un coût de simulation — qui correspond au temps passé à effectuer des simulations Monte Carlo de la politique courante — et un temps d'apprentissage qui correspond au temps passé à apprendre le classeur qui définiera la nouvelle politique[4]. Ce coût prend la forme générale suivante :

$$\text{Coût} = SAK \times T\mathcal{I}(A) + \mathcal{C}(S,A), \tag{4}$$

où $T\mathcal{I}(A)$ est le coût de simulation d'une trajectoire de longueur T, $SAK \times T\mathcal{I}(A)$ est le coût d'exécution les K simulatoins de Monte Carlo sur S états en testant A actions possible actions et $\mathcal{C}(S,A)$ est le coût d'apprentissage du classeur correspondant[5].

La différence principale entre RCPI et ERCPI provient des valeurs de $\mathcal{I}(A)$ et $\mathcal{C}(S,A)$. Quand on compare ERCPI à RCPI en utilisant un classeur multiclasse un-contre-tous (RCPI-OVA) — dans ce cas, nous entraînons un classeur binaire par action possible, les 2 classes étant d'une part l'action considéré, d'autre part toutes les autres — on constate aisément que ERCPI réduit à la fois $\mathcal{I}(A)$ et $\mathcal{C}(S,A)$ par un facteur $\frac{A}{\log A}$ (*cf.* table 1).

Quand on considère l'algorithme BRCPI, $\mathcal{I}$ et $\mathcal{C}$ sont réduits comme dans le cas d'ERCPI. Cependant, le coût de simulation est réduit également puisque BRCPI apprend un ensemble de politiques pour des

[4]En practique, quand il y a de nombreuses actions, le coût des simulations est significantivement plus élevée que le temps d'apprentissage, lequel est alors ignoré (Lazaric *et al.*, 2010).

[5]Nous ne tenons pas compte du coût en temps des transitions dans le PDM.

Table 2: Complexité *vs.* le nombre d'actions possibles.

Algortihme	RCPI-OVA	ERCPI	BRCPI
Complexité	$\mathcal{O}(A^2)$	$\mathcal{O}(A\log(A))$	$\mathcal{O}(\log(A))$

problèmes (sous-PDM) n'ayant que 2 actions possibles, ces problèmes étant au nombre de $\gamma\log(A)$. Pour chacun de ces sous-problèmes, le coût de simulatio nest $2SK(2T)$ puisque le nombre d'actions possibles est 2. le cout d'apprentissage correspond à l'apprentissage de $\gamma\log(A)$ classeurs binaires, ce qui entraîne un coût très faible — *cf.* la table 1. La complexité résultante par rapport au nombre d'actions est présenté dans la table 2. On constate que cette complexité est seulement logarithmique pour BRCPI. De plus, il est important de noter que les sous-problèmessont indépendants les uns des autres, donc BRCPI peut très facilement être parallélisé. Pour illustrer ces complexités, les temps de calculs sont indiqués dans la section expérimentale.

4 Expérimentations

Dans ce papier, notre but est de gérer, pratiquement, des grands nombres d'actions non corrélées. Aussi, la meilleure démonstration de cette capacité est de proposer une évaluation expérimentale de ERCPI et BRCPI. Dans cette section, nous montrons que BRCPI exhibe des temps de calcul très faibles, permettant de passer de journées de calculs à des heures, voire moins.

4.1 Protocole

Nous évaluons nos algorithmes sur deux problèmes d'AR : le fameux *Mountain Car* et un problème discret qui nous permet d'investiguer les performances des algorithmes proposés.

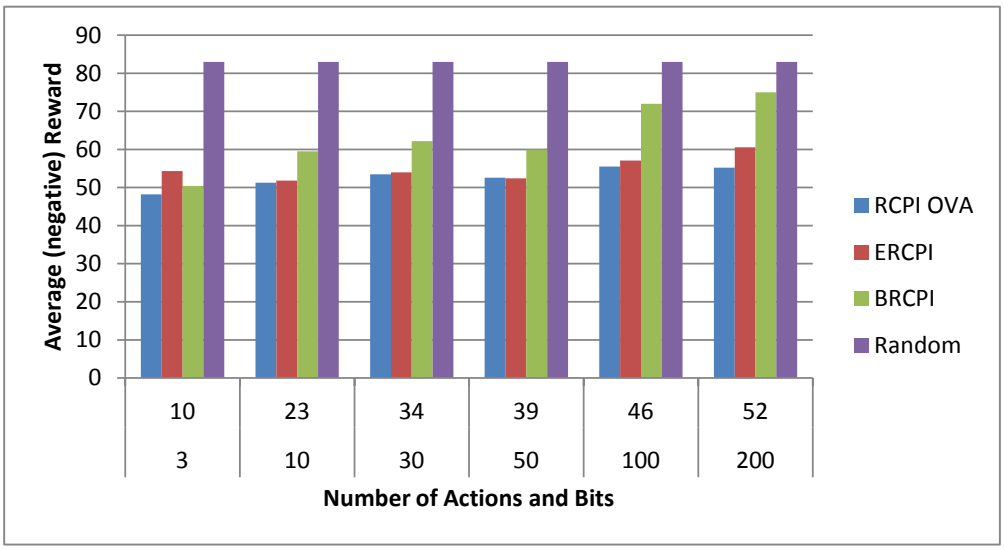

Figure 2: *Mountain Car*: récompense moyenne (valeur négative : plus la récompense est faible, mieux c'est) obtenue par différents algorithmes sur 3 exécutions avec des nombres d'actions différents. Sur l'axe des abscisses, la première ligne correspond à $\gamma\log(A)$ tandis que la seconde ligne indique le nombre d'actions A.

Le *Mountain Car* est bien connu dans la communauté AR. Sa définition varie quelque peu selon les auteurs, mais il est très généralement basé sur un nombre faible (2 ou 3) d'actions. Cependant, les actions peuvent également être définies sur un domaine continu lequel rend le problème un peu plus " réaliste ". Dans nos expériences, nous discrétisons l'intervalle d'accélérations possibles pour obtenir un ensemble d'actions discret. Nous faisons varier la discrétisation de grossière à fine, ce qui permet d'étudier l'effet du nombre d'actions possibles sur les performances des algorithmes comparés. L'espace d'états est continu ; il est géré par des techniques de pavages comme dans (Sutton, 1996). La récompense immédiate à chaque pas de temps est -1 et un épisode a une durée maximale de 100 pas de temps. Le retour (récompenses cumulées) mesure donc la capacité de la politique obtenue à amener la voiture au sommet de la montagne le plus vite possible.

Le second problème que nous nommerons *Maze*, est un monde en grille de taille 50x50 dans lequel un apprenant doit aller du côté gauche vers le côté droit de la grille. À chaque cellule de la grille est associée une récompense immédiate négative quand elle est atteinte, soit -1, -10, ou -100. Dansle cas le plus simple, l'agent peut choisir d'aller soit vers le haut, le bas ou la droite, résultant en un PDM à 3 actions. Nous construisons des ensembles d'actions plus complexes en générant toutes les séquences de ces 3 actions élémentaires de longueur donnée. Ainsi pour une longueur 2, les 6 actions possibles sont haut-haut, haut-bas, haut-droite, ... Contrairement au *Mountain Car*, il n'existe pas de notion de similarité entre actions dans ce problème, similarité étant entendue par rapport aux conséquences. Chaque état est représenté par un vecteur de caractéristiques qui contient les informations à propos des différents types de cellules qui se situent dans une grille 5x5 centrées sur la position courante de l'agent. La récompense globale obtenue correspond à la capacité de l'agent d'aller du bord gauche au bord droit, en évitant les cellules ayant une récompense très négative.

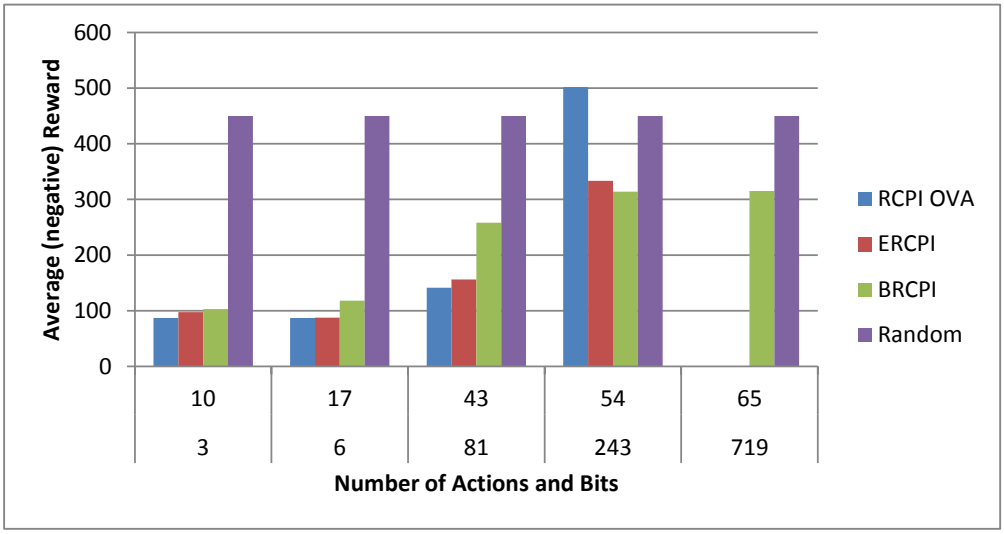

Figure 3: *Maze* : récompense moyenne (valeur négative : plus c'est petit, mieux c'est) obtenue par les différents algoritmes sur 3 *mazes* différent générés aléatoirement, avec des nombre d'actions différents. Sur l'axe des abscisses, la première ligne correspond à $\gamma \log(A)$ tandis que la seconde indique le nombre d'actions A. RCPI-OVA et ERCPI n'ont pas pu s'exécuter pour 719 actions. Notons que pour 243 actions, RCPI-OVA apprend une très mauvaise politique.

Dans les deux problèmes, les états d'apprentissage et de test sont sélectionnés uniformément parmi les états possibles. Nous avons choisi d'échantillonner $S = 1000$ états pour chaque problème ; le nombre de trajectoires effectuées pour chaque paire état-action est $K = 10$. Le classeur utilisé est un perceptron à

sortie *hinge-loss*, entraîné par 1000 itérations de desecnte de gradient stochastique. Les codes correcteurs d'erreur ont été générés en utilisant une procédure aléatoire classique, décrite dans (Berger, 1999). La valeur de α pour le mélange de la nouvelle politique avec l'ancienne a été fixé à 0.5.

4.2 Résultats

Les récompenses moyennes obtenues après convergence des 3 algorithmes sont présentés sur les figures 3 et 2, pour différents nombres d'actions. La récompense moyenne d'une politique aléatoire est également indiquée. Tout d'abord, nous voyons que RCPI-OVA et ERCPI obtiennent les mêmes performances sur ces deux problèmes, excepté pour *Maze* avec 243 actions. Cela peut s'expliquer par le fait que les politiques trouvées par RCPI-OVA ne sont pas capables de traiter des problèmes avec un grand nombre de classes, quand cela implique de résoudre des problèmes de classification binaires avec très peu d'exemples positifs. Dans ce contexte, les classeurs à base de CCE sont connus pour avoir un meilleur comportement. BRCPI atteint de moins bonnes performances que RCPI-OVA et ERCPI. En effet, BRCPI apprend des sous-politiques binaires indépendantes qui, quand utilisées ensemble pour reconstruire une politique pour le PDM original, ne correspondent qu'à une politique sous-optimale. Notons que même face à un grand nombre d'actions, BRCPI est capable d'apprendre une politique pertinente — en particulier, *Maze* avec 719 actions montre que la politique engendrée par BRCPI est significativement meilleure qu'une politique aléatoire, tandis que les autres algorithmes ne sont tout simplement pas capables, dans la pratique, de traiter ce problème. C'est très intérssant car cela implique que BRCPI est capable de trouver une politique satisfaisante là où les limites pratiques de mise en œuvre des autres algorithmes sont dépassées.

La table 3 indique les temps de calcul pour une itération de chacun des algorithmes pour *Mountain Car* avec 100 actions. ERCPI est plus rapide que RCPI d'un facteur $1{,}4$ tandis que BRCPI est $12{,}5$ fois plus rapide que RCPI, $23{,}5$ plus rapide si l'on ne tient compte que du coût de simulation. Cela explique pourquoi la figure 3 ne montre pas les performances de RCPI et ERCPI sur le problème *maze* avec 719 actions : dans ce contexte, une itération de ces algorithmes prend des journées de calcul, alors que quelques heures suffisent à BRCPI. Enfin, notons que les gains en temps de clacul augentent avec le nombre d'actions du PDM.

Enfin, la figure 4 indique les performances de BRCPI en fonction du nombre de trajectoires ; on peut voir qu'une meilleure politique peut être trouvée en augmentant le nombre de trajectoires, K. De plus, même si on utilise une valeur plus grande pour K, le temps d'exécution de BRCPI demeure plus bas que RCPI-OVA et ERCPI.

Table 3: Temps (en secondes) pour une itération — en distinguant simulation et apprentissage — des différents algorithmes. Ces temps sont mesurés sur un processeur Xeon-X5690 équipé d'une carte GPU TESLA M2090 pour $K = 10$ et $S = 1000$. Le gain total ainsi que le gain sur les simulations par rapport à RCPI-OVA sont indiqués dans la colonne de droite.

Mountain Car - 100 Actions - 46 bits				
	Simulation	Apprentissage	Total	Gain
RCPI-OVA	$4{,}312$	380	$4{,}698$	$\times 1{,}0$
ERCPI	$3{,}188$	190	$3{,}378$	$\times 1{,}4 (\times 1{,}35)$
BRCPI	184	190	374	$\times 12{,}5 (\times 23{,}5)$

5 Travaux en relation

Rollout Classification Policy Iteration (Lagoudakis & Parr, 2003) est un algorithme d'apprentissage par renforcment adapté aux très grands espaces d'états. La phase Monte Carlo de RCPI peut être très coûteuse et un ensemble d'approches ont été proposées pour améliorer cet aspect (Dimitrakakis & Lagoudakis, 2008). Récemment, RCPI a été évaluée théoriquement (Lazaric *et al.*, 2010). L'efficacité bien connue de cet algorithme pour traiter des problèmes réels d'une part et son inadaptation à traiter des problèmes ayant beaucoup d'actions d'autre part, ont motivé ce travail.

L'apprentissage par renforcement a depuis longtemps traité de très grand espace d'états (voire des espaces infinis, continus) en généralisant la fonction valeur sur l'espace d'états (voir notamment des travaux

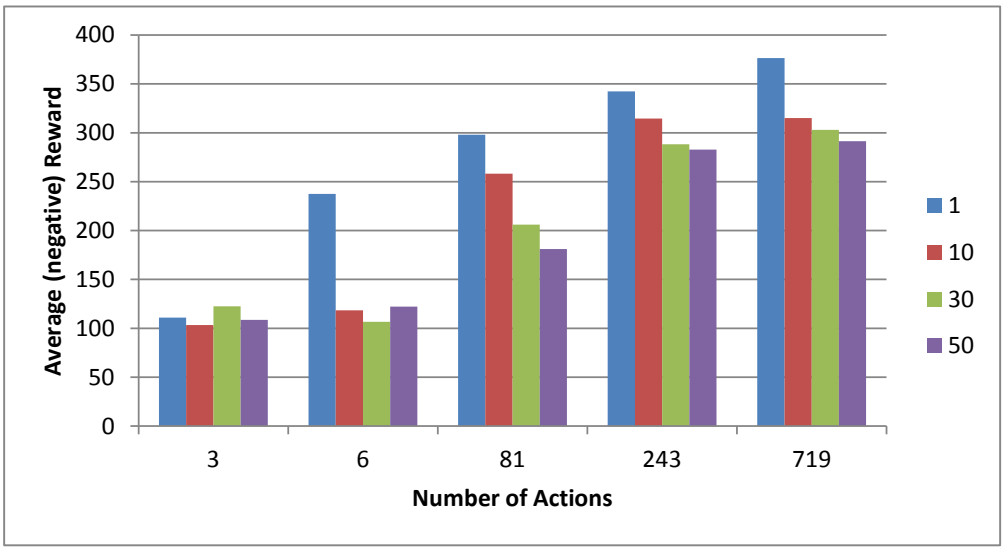

Figure 4: *Maze* : récompense moyenne obtenue par BRCPI pour $K \in \{1, 10, 30, 50\}$ (rappelons que plus la valeur est négative, mieux c'est).

précurseurs comme (Tham, 1994; Tesauro, 1992)). Le premier, Tesauro a introduit la notion de *rollouts* (Tesauro & Galperin, 1997), s'appuyant sur les techniques de Monte Calo pour explorer un grand espace d'états et d'actions. La gestion de grands espaces d'états a également été considérée via des méthodes d'échantillonnage ou de descente de gradient sur Q (Negoescu *et al.*, 2011; Lazaric *et al.*, 2007), mais ces approches supposent une fonction Q-fonction sympathique, ce qui est une attente optimiste pour des problèmes réels.

Il existe des travaux qui réduisent logarithmiquement la complexité en fonction du nombre d'actions, mais ceux-ci imposent une certaine forme de recherche binaire sur l'ensemble d'actions (Pazis & Lagoudakis, 2011; Pazis & Parr, 2011). Ces approches complètent le PDM avec une structure de recherche sur l'ensemble des actions, plaçant ainsi la complexité de l'ensemble d'actions dans l'espace d'états. Bien que nous n'ayons pas été inspirés pas ces travaux, ceux-ci sont similaires aux nôtres dans leur esprit. Néanmoins, aucun ne propose de solution pour accélérer l'apprentissage comme le fait BRCPI

Pour leur part, les codes correcteurs d'erreur ont été introduits par Dietterich and Bakiri (1995) dans le cadre de la classification supervisée multi-classes. Bien que non traitée dans cet article, la construction du codage est un élément clé dans les performances des classeurs basés sur des CCE (Beygelzimer *et al.*, 2008). Dans notre cas, nous avons utilisé des codes générés aléatoirement, mais le codage peut être appris à partir d'un ensemble d'exemples (Crammer & Singer, 2002) ou en utilisant une métrique *a priori* sur les classes (Cissé *et al.*, 2011).

6 Conclusion

Nous nous sommes intéressés à l'apprentissage par renforcement dans le cas où le nombre d'actions est grand, sans être approximable par l'infini.

Nous avons proposé deux nouveaux algorithmes capables de trouver de bonnes politiques tout en étant bien plus rapide que l'algorithme RCPI. Le premier, ERCPI, est basé sur l'utilisation de codes correcteurs d'erreurs dans RCPI, tandis que le second, BRCPI, décompose le PDM original en un ensemble de sous-PDM à 2 actions pour lesquels l'apprentissage peut être très rapide. Alors qu'ERCPI obtient des résultats équivalents, voire des performances meilleures que l'algorithme classique RCPI-Un-contre-tous pour un

coût en calcul moindre, BRCPI calcule des sous-politiques très rapidement, même quand le nombre d'actions est très élevé. La complexité des algorithmes proposés est en $\mathcal{O}(A\log(A))$ pour ERCPI et en $\mathcal{O}(\log(A))$ pour BRCPI, à comparer avec la complexité de RCPI qui est en $\mathcal{O}(A^2)$. Notons que l'on peut également utiliser BRCPI pour rapidement découvrir une bonne politique et utiliser ensuite ERCPI pour l'améliorer ; cette solution pratique n'a pas été étudiée dans ce papier.

Ce travail ouvre de nombreuses perspectives de recherche : tout d'abord, les performances de BRCPI dépendent directement de la qualité du codage des actions et il est donc particulièrement intéressant de travailler sur ce point. D'un point de vue théorique, nous envisageons d'étudier la relation entre les performances des sous-politiques calculées par BRCPI et la performance de la politique agrégée. Enfin, le fait que nos algorithmes soient capables de traiter des PDM avec des milliers d'actions ouvre plusieurs perspectives d'un point de vue applicatif et peut nous permettre de trouver de bonnes solutions pour des problèmes qui n'ont pas été étudiés auparavant à cause de leur trop grand nombre d'actions.

References

BENTLEY J. L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM*, **18**(9), 509–517.

BERGER A. (1999). Error-correcting output coding for text classification. In *Workshop on Machine Learning for Information Filtering, IJCAI '99*.

BEYGELZIMER A., LANGFORD J. & ZADROZNY B. (2008). Machine learning techniques—reductions between prediction quality metrics. *Performance Modeling and Engineering*, p. 3–28.

BUBECK S., MUNOS R., STOLTZ G., SZEPESVÁRI C. *et al.* (2011). X-armed bandits. *Journal of Machine Learning Research*, **12**, 1655–1695.

CISSÉ M., ARTIERES T. & GALLINARI P. (2011). Learning efficient error correcting output codes for large hierarchical multi-class problems. In *Workshop on Large-Scale Hierarchical Classification ECML/PKDD '11*, p. 37–49.

CRAMMER K. & SINGER Y. (2002). On the Learnability and Design of Output Codes for Multiclass Problems. *Machine Learning*, **47**(2), 201–233.

DIETTERICH T. & BAKIRI G. (1995). Solving multiclass learning problems via error-correcting output codes. *Jo. of Art. Int. Research*, **2**, 263–286.

DIMITRAKAKIS C. & LAGOUDAKIS M. G. (2008). Rollout sampling approximate policy iteration. *Machine Learning*, **72**(3), 157–171.

LAGOUDAKIS M. G. & PARR R. (2003). Reinforcement learning as classification: Leveraging modern classifiers. In *Proc. of ICML '03*.

LAZARIC A., GHAVAMZADEH M. & MUNOS R. (2010). Analysis of a classification-based policy iteration algorithm. In *Proc. of ICML '10*, p. 607–614.

LAZARIC A., RESTELLI M. & BONARINI A. (2007). Reinforcement Learning in Continuous Action Spaces through Sequential Monte Carlo Methods. In *Proc. of NIPS '07*.

NEGOESCU D., FRAZIER P. & POWELL W. (2011). The knowledge-gradient algorithm for sequencing experiments in drug discovery. *INFORMS J. on Computing*, **23**(3), 346–363.

PAZIS J. & LAGOUDAKIS M. G. (2011). Reinforcement Learning in Multidimensional Continuous Action Spaces. In *Proc. of Adaptive Dynamic Programming and Reinf. Learn.*, p. 97–104.

PAZIS J. & PARR R. (2011). Generalized Value Functions for Large Action Sets. In *Proc. of ICML '11*, p. 1185–1192.

PUTERMAN M. L. (1994). *Markov decision processes — Discrete Stochastic Dynamic Programming*. Probability, and mathematical statistics. John Wiley & Sons.

SUTTON R. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Proc. of NIPS '96*, p. 1038–1044.

TESAURO G. (1992). Practical issues in temporal difference learning. *Machine Learning*, **8**, 257–277.

TESAURO G. & GALPERIN G. R. (1997). On-Line Policy Improvement Using Monte-Carlo Search. In *Proc. of NIPS '97*, p. 1068–1074.

THAM C. (1994). *Modular on-line function approximation for scaling up reinforcement learning*. PhD thesis, University of Cambridge.