APPRENTISSAGE PAR RENFORCEMENT DANS LES PROCESSUS DE DÉCISION MARKOVIENS FACTORISÉS

Thèse de Doctorat de l'Université Paris VI

Présentée par Thomas Degris
pour obtenir le grade de
Docteur de l'Université Paris VI

Spécialité : informatique

Soutenue le 26 avril 2007 devant le jury composé de :

Alain Dutech	(INRIA, Vandoeuvre les Nancy)	Examinateur
David Filliat	(ENSTA, Paris)	Examinateur
Frédérick Garcia	(INRA, Toulouse)	Rapporteur
Michael Littman	(Rutgers University, New Jersey)	Examinateur
Rémi Munos	(INRIA Futurs, Lille)	Rapporteur
Patrice Perny	(Université Paris VI)	Examinateur
Olivier Sigaud	(Université Paris VI)	Directeur de thèse

Résumé

Les méthodes classiques d'apprentissage par renforcement ne sont pas applicables aux problèmes de grande taille car elles impliquent l'énumération d'un trop grand nombre d'états. Les Processus de Décision Markovien Factorisés (FMDPs) permettent de représenter de tels problèmes de façon plus compacte en spécifiant leur structure. Des méthodes de planification, basées sur la programmation dynamique ou la programmation linéaire et adaptées au cadre des FMDPs, ont été proposées et obtiennent de bons résultats, même lorsque le nombre d'états est très grand. Cependant, la mise en oeuvre de ces méthodes nécessite que la structure soit spécifiée manuellement a priori.

Cette thèse étudie l'apprentissage automatique de la structure d'un problème d'apprentissage par renforcement représenté sous la forme d'un FMDP. À partir de l'expérience d'un agent dans son environnement, nous proposons d'utiliser des techniques d'apprentissage supervisé, en particulier l'induction d'arbres de décision, pour construire une représentation compacte du problème. Une fois le problème représenté, nous montrons qu'il est possible de réutiliser les méthodes de planification adaptées au cadre des FMDPs pour obtenir une solution efficace à celui-ci.

Nous proposons une étude empirique de cette approche en la mettant en oeuvre sur plusieurs problèmes stochastiques de grande taille classiques dans la littérature des FMDPs. Sur l'ensemble des tests étudiés, dans le cadre d'un apprentissage hors ligne puis en ligne, nous montrons que notre approche possède les capacités de généralisation et d'agrégation nécessaires lorsque le nombre d'états possibles est très grand. De plus, en appliquant nos outils au contrôle d'un personnage non joueur dans le jeu vidéo Counter-Strike©, nous montrons que les représentations construites par l'apprentissage peuvent être lisibles et manipulables par un opérateur humain.

Mots-clés : apprentissage par renforcement, induction d'arbres de décision, processus de décision markovien factorisé, jeu vidéo, exploration dirigée

Abstract

Classical reinforcement learning techniques are not adapted to solve large problems because they require to explicitly enumerate the possible states in the state space. Factored Markov Decision Processes (FMDPs) are a mathematical framework exploiting the structure of the problem to represent it compactly. Planning methods, based on dynamic programming or linear programming, have been adapted to FMDPs and show good results, even for very large problems. However, these methods require to manually specify the structure of the problem before to solve it.

This thesis propose to learn automatically the structure of a reinforcement learning problem represented as a FMDP. From the experience of an agent in its environment, we propose to use supervised learning techniques, more precisely induction of decision trees, to build a compact representation of the problem. Once the problem represented, we show that it is possible to reuse planning methods for FMDPs to compute an efficient solution for it.

We propose an empirical study of such approach by validating it on different stochastic large size problems taken from the FMDP literature. Both for off-line and on-line settings, we show that our approach exhibit aggregation and generalisation properties required for problems with a large number of states. Moreover, we show that the representations built by our approach are human readable by applying it on the Counter-Strike© video game.

Keywords : reinforcement learning, decision tree induction, factored markov decision processes, video game, directed exploration

Remerciements

En premier lieu (et pour me faire pardonner de les avoir oubliés lors de mon discours à ma soutenance), je remercie mon jury d'avoir pris le courage et le temps nécessaire pour se pencher sur mes élucubrations scientifiques. À commencer par mes rapporteurs, Frédérick Garcia et Rémi Munos : j'ai été honoré qu'ils aient lu avec assiduité (et dans les temps) ma thèse, malgré leurs contraintes professionnelles. De plus, avoir la participation de ces deux spécialistes dans les processus de décision markovien et l'apprentissage par renforcement dans ma thèse a été très important pour moi.

Bien que n'étant pas rapporteur, Alain Dutech s'est aussi particulièrement investi dans la relecture de ce mémoire et je le remercie tout particulièrement pour cela. En plus d'avoir suggéré plusieurs corrections, suggestions et remarques pour l'amélioration du manuscrit, j'ai eu la chance d'avoir plusieurs discussions approfondissant ma réflexion autour de mon travail.

Merci aussi à David Filliat que je soupçonne fortement (bien qu'il ne me l'ait jamais avoué) d'avoir influencé de façon positive la décision concernant mon financement. David m'a ensuite laissé libre de mon projet scientifique, au fil de mes lectures, des problématiques et de mes motivations et je lui en suis extrêmement reconnaissant.

Un grand merci à Michael Littman qui, je trouve, constitue un modèle dans sa façon d'aborder la recherche. D'une part, il dirige une équipe adressant des problématiques allant du calcul d'une borne de convergence à l'application robotique. D'autre part, il prend les choses avec toujours beaucoup de recul et d'humour (son mime de ϵ-greedy est à voir).

Merci à Patrice Perny pour sa participation en tant que président de mon jury de thèse et pour ses commentaires et questions qui ouvrent de nombreuses perspectives à mon travail.

Au cours des trois années nécessaires à la réalisation de ces travaux, l'encadrement d'Olivier Sigaud a été déterminant. Je le remercie tout particulièrement pour son engagement à faire en sorte que ma thèse se passe bien, de m'avoir fait confiance et encouragé sur de nombreux points et d'avoir partagé avec générosité son expérience de développeur, chercheur, enseignant et philosophe.

Je tiens tout particulièrement à souligner le rôle fondamental joué avec brio par Pierre-Henri Wuillemin à qui j'adresse mes plus chaleureux remerciements. Non seulement son savoir, communiqué avec beaucoup de patience, de pédagogie et d'enthousiasme pour le non-mathématicien que je suis, m'a grandement facilité l'accès à la littérature des FMDPs, mais en plus, beaucoup de nos discussions scientifiques ont eu des répercussions directes sur les résultats présentés dans cette thèse. À plusieurs moments, Pierre-Henri était là pour poser les bonnes questions, apporter les bonnes réponses et orienter mes travaux dans la bonne direction. Je lui en suis extrêmement reconnaissant. Je ne désespère pas de le convaincre que, finalement, les ADDs, ce n'est pas si mal.

Obtenir des résultats pour Counter-Strike© a nécessité un important travail de développement que je n'aurai pas eu le temps de fournir si je n'avais pas été aidé par de brillants collaborateurs.

Nicolas Despres a été le principal investigateur pour l'implémentation et la mise au point d'une architecture logicielle qui avait seulement été griffonnée sur un tableau. Jean-Philippe Dubus s'est frotté avec rigueur et détermination à l'interface de programmation de Counter-Strike©. Merci aussi à Guillaume Riby, Alix Mougenot et Rémy-Christophe Schermesser qui ont fait partie de l'équipe de développement de choc de Kodabot.

Une thèse représente un certain engagement personnel que j'ai pris plaisir à fournir, étant donné l'équipe de travail, l'Animatlab, dans laquelle j'étais. Un merci à Jean-Arcady Meyer, Agnès Guillot, Olivier Sigaud et Stéphane Doncieux pour former une équipe de recherche ouverte, stimulante et propice aux nouvelles idées. Un merci particulier à Thierry Gourdin avec qui j'ai partagé mon bureau dans la joie et la bonne humeur. Il s'est toujours porté volontaire pour subir, avec patience et curiosité, mes idées en génie logiciel et, en plus, a souvent supporté ma musique. Ces deux exemples illustrent quel admirable camarade et collègue se cachent derrière sa barbe. Merci à Gabriel Robert dont la thèse m'a beaucoup motivé et inspiré. Sous ses airs de joueur impénitent et d'animateur de la bonne humeur, il n'en reste pas moins un conseiller intelligent et pertinent doublé d'un travailleur consciencieux. Merci à Fabien Flacher dont la culture générale en IA (et pas qu'en IA d'ailleurs) m'étonnera toujours, pour nos débats politiques et de toujours titiller mon esprit de contradiction. Merci à Loïc Lachèze pour sa passion pour les templates en C++ et son talent à trouver des positions défensives imprenables dans ET. Merci aussi aux autres AnimatLabien d'avoir contribué à un cadre de travail agréable, je pense notamment à Vincent, Mehdi, Alexandra, Jean-Baptiste, Gildas, Benoît, Stéphane, et Steve à qui je souhaite une bonne continuation. Un merci spécial à Manu et Laurent pour leur enthousiasme, leurs relectures et leur respect des vieux : bonne chance pour vos thèses.

Je remercie aussi Christophe Marsala pour avoir partagé ses connaissances concernant les arbres de décisions et, d'une façon plus générale, l'équipe LOFTI pour avoir partagé son ambiance sympa. Un grand merci aussi à l'équipe administrative et technique du LIP6, Jacqueline Le Baquer, Ghislaine Mary, Thierry Langroy, Nicole Nardy, Christophe Bouder, Jean-Pierre Arranz et Vincent Cuzin, qui font tout leur possible pour que, dans de bonnes conditions, les chercheurs cherchent et les enseignants enseignent.

Je remercie aussi très chaleureusement Angelo Arleo qui m'a donné goût à la recherche (et, avec Valérie, goût aux voyages par la même occasion) et qui m'a permis d'être au bon endroit et au bon moment pour la réalisation de cette thèse. Il m'a enseigné avec patience et enthousiasme les rudiments du métier de chercheur et le résultat se retrouve directement dans mes articles et dans cette thèse. J'espère avoir été à la hauteur. Mon passage au Collège de France aura été une étape décisive pour le démarrage de cette thèse, merci aussi à Laure, Anne-Lyse et Éric.

Lors du déroulement de cette thèse, je dois avouer que j'avais du mal à penser à autre chose que l'apprentissage par renforcement, les FMDPs ou le développement. Heureusement, des personnes sympathiques étaient là pour m'encourager à me changer les idées et m'apporter ainsi des moments

rafraîchissant. Je remercie donc Stéphane et Amélie pour les promenades et les dégustations (désolé d'avoir mis autant d'enthousiasme à gratter votre mur) ; Alain et Lydia pour le roller et le Sénégal ; Caroline et Stéphane pour le tir à l'arc, le barbecue, les jeux vidéo et la geekattitude ; Marie-Jeanne pour ses talents d'interprète français/allemand, la piscine et son amour inconditionnel de la langue française (pas la peine d'aller chercher geekattitude dans le dictionnaire Marie-Jeanne : il n'existe pas) et Yvan pour l'escalade.

Un très, très, très (à multiplier par le nombre de couples état/action possibles dans le plus grand problème que j'ai utilisé dans cette thèse) grand merci à ma famille. En premier lieu, je dois beaucoup à mes parents qui m'ont, depuis toujours, supporté et encouragé dans ma passion pour l'informatique (en plus de m'éduquer, ce qui n'était sûrement pas une mince affaire). Leur fierté et leur admiration pour leurs enfants a toujours été un moteur et une motivation pour moi. Et puis un grand merci à Quentin, artiste dans l'âme, à la créativité et au talent sans limite, à Nathalie pour ses pâtisseries, ses barbes à papa et sa gaieté (et vive la Haute-Saône) et à Thibaut pour sa bonne humeur constante et sa passion des avions (et oui... au final, après avoir compté les points, le bilan est que c'est quand même l'A380 le plus beau). Une mention spéciale pour mon oncle Didier qui m'a fourni mes deux premiers ordinateurs (un Apple IIe et un IBM 286). Je profite de cette occasion pour remercier de façon chaleureuse et reconnaissante ma belle famille, Monique, Jean-Marie, Isabelle et Éric, notamment pour leur accueil généreux, leur disponibilité et leur gentillesse (avec une petite mention spéciale pour leur douche).

J'ai gardé le meilleur (ou plutôt la meilleure) pour la fin : Delphine. Comment la remercier pour sa patience sans faille, sa gentillesse constante, son soutien, sa sensibilité, son sens de l'organisation, son écoute, son courage, son intelligence et son calme ? (Ce sont les premières qualités qui me viennent en tête, si je continue de chercher, il y en a plein d'autres qui vont arriver.) Il faut imaginer un Thomas souvent dans ses pensées, préoccupé par les FMDPs, travaillant quasiment tout le temps, enthousiasmé quand ça marche et inquiété quand ce n'est pas le cas ; face à une Delphine sereine, attentive, disponible, encourageante et apaisante. De plus, à coté de cela, elle nous a organisé des supers vacances, des supers visites, des supers soirées et des supers week-ends, me mettant ainsi en vacances forcées pour mon plus grand bien. Bref, difficile d'expliquer comment et combien son soutien aura été important pour moi. Ce qui est sûr, c'est que je lui en serai éternellement reconnaissant.

Annexe des remerciements

J'ai pris l'habitude de travailler en écoutant de la musique (avec des contraintes telles que ne jamais écouter deux fois le même morceau pendant une journée). Merci donc aux auteurs, compositeurs et interprètes suivant qui m'ont accompagnés : Alain Souchon, Fredericks, Goldman et Jones, Laurent Voulzy, Yann Tiersen, Bach, Beethoven, Bizet, Chopin, Debussy, Handel, Mozart, Rossini, Satie, Schubert, Schumann, Dvorak, Tchaikovsky, Vivaldi (et tous leurs illustres interprètes anonymes), Daft Punk, Total Eclipse, Gotan Project, Christy Moore, Mary Black, Norah Jones, Keziah Jones, Brad Mehldau, Buddy Guy, Claude Bolling, Gonzales, Julien Lourau, Keith Jarret, Lincoln Abbey, Lisa Ekdahl, Michel Petrucciani, Sting, Enya, Angela McCluskey, Björk, Cat Stevens, Coldplay, David Gray, Deep Forest, Dido, Dire Straits, The Doors, Emiliana Torrini, Emilie Simon, Flotation Toy Warning, Garbage, Ghinzu, Girls in Hawaii, Goldfrapp, Gorillaz, Jimmy Hendrix, Jethro Tull, Madonna, Marianne Faithfull, Mark Knopfler, Moby, Morcheeba, Nick Drake, Paolo Conte, Piers Faccini, Pink Floyd, Placebo, Portishead, Radiohead, Shearwater, Simon and Garfunkel, Sinead O'Connor, Supertramp, Suzanne Vega, Syd Matters, Texas, Tracy Chapman, U2, The Who, Bruno Coulais, Cliff Martinez, Cyril Morin, Howard Shore, Joe Hisaishi, Kenji Kawai et Yamashiro Shoji.

Table des matières

1 Introduction **15**
1.1 Objectifs . . . 16
1.2 Méthodes . . . 17
1.2.1 Exploitation de la structure du problème . . . 18
1.2.2 Apprentissage de la structure du problème . . . 19
1.2.3 Exploration et généralisation . . . 19
1.3 Principales contributions . . . 20
1.4 Plan du mémoire . . . 21

2 Les Processus de Décision Markoviens **23**
2.1 Définition d'un MDP . . . 23
2.1.1 Politiques et fonctions de valeur . . . 25
2.1.2 Fonctions de valeur optimales . . . 27
2.2 Résolution d'un MDP . . . 27
2.2.1 Programmation dynamique . . . 28
2.2.2 Programmation linéaire . . . 31
2.3 Planification et apprentissage dans les MDPs . . . 32
2.3.1 L'algorithme Q-learning . . . 33
2.3.2 L'approche DYNA . . . 34
2.4 Synthèse . . . 36

3 Les Processus de Décision Markoviens Factorisés **37**
3.1 Les Processus de Decision Markoviens Factorisés . . . 37
3.1.1 Représentation de la fonction de transition . . . 39
3.1.2 Représentation de la fonction de récompense . . . 41
3.2 *Structured Value Iteration* et *Stucture Policy Iteration* . . . 43
3.2.1 Représentations . . . 44
3.2.2 Manipulations . . . 49
3.2.3 Calcul d'une fonction de valeur d'action sur une itération . . . 50

3.2.4 Construction d’une politique gloutonne 52
3.2.5 Les algorithmes SPI et SVI 53
3.3 L’algorithme *Stochastic Planning Using Decision Diagrams* 54
3.3.1 Représentations 55
3.3.2 Algorithmes 60
3.4 Programmation Linéaire Approchée dans un FMDP 61
3.4.1 Représentations 62
3.4.2 Manipulations 69
3.4.3 Calcul d’une fonction de valeur d’action sur une itération 71
3.4.4 Algorithmes 76
3.5 Synthèse 79

4 Apprentissage hors-ligne d’un FMDP 81
4.1 Apprentissage supervisé d’ensembles d’exemples 82
4.1.1 Induction d’arbres de décision 82
4.1.2 Mesure d’information pour des valeurs symboliques 84
4.1.3 Mesure d’information pour des valeurs réelles 86
4.2 Construction d’un FMDP et intégration des algorithmes de planification 86
4.2.1 Décomposition des observations en ensembles d’exemples 88
4.2.2 Algorithmes de construction de FMDPs 92
4.2.3 Intégration avec les algorithmes de planification 96
4.2.4 Réorganisation de règles exhaustives et mutuellement exclusives 99
4.3 Résultats 102
4.3.1 Incidence de la valeur du seuil 102
4.3.2 Incidence de la taille du problème 108
4.3.3 Incidence de la taille de l’échantillon d’observations 116
4.4 Synthèse 126

5 Apprentissage incrémental : l’approche SDYNA 129
5.1 L’approche SDYNA 130
5.2 Intégration de l’apprentissage dans SDYNA 131
5.2.1 Induction incrémentale d’arbres de décision 132
5.2.2 Apprentissage incrémental d’un FMDP 134
5.3 Intégration de la planification dans SDYNA 137
5.3.1 Intégration de l’algorithme SVI 138
5.3.2 Intégration de l’algorithme SPUDD 140
5.3.3 Intégration de la programmation linéaire approchée 141
5.4 Résultats 142

5.4.1 Le problème *Coffee Robot* 143
5.4.2 Le problème *Factory* 145
5.4.3 Le problème *Factory4* 148
5.4.4 Le problème *Ring* 150
5.5 Synthèse 152

6 Le compromis exploration/exploitation dans SDYNA 155
6.1 Algorithme d'apprentissage basé sur un modèle et avec exploration dirigée 156
6.1.1 Définition de l'apprentissage “efficace” 157
6.1.2 L'algorithme Explicit Explore or Exploit 157
6.1.3 L'algorithme R-MAX 159
6.1.4 Les algorithmes MBIE et MBIE-EB 160
6.1.5 Apprentissage d'un FMDP 162
6.2 Exploration dirigée dans l'architecture SDYNA 164
6.2.1 Problème de l'exploration lorsque la structure est inconnue 164
6.2.2 Bonus d'exploration de paramètres et bonus d'exploration de structure . . . 166
6.3 Résultats 168
6.3.1 Les problèmes *Linear* et *Expon* 169
6.3.2 Le problème *Factory* 170
6.4 Synthèse 172

7 Application au jeu vidéo Counter-Strike 175
7.1 Description du jeu 175
7.2 Définition et formalisation du problème 178
7.2.1 Définition des récompenses 178
7.2.2 Définition de l'ensemble d'états 179
7.2.3 Définition de l'ensemble d'actions 180
7.2.4 Définition des pas de temps 181
7.2.5 Remarques concernant le problème 182
7.3 Mise en œuvre de SDYNA 182
7.4 Résultats 183
7.4.1 Fonction de récompense 184
7.4.2 Fonction de transition 186
7.4.3 Politiques gloutonnes 189
7.5 Synthèse 193

8 Discussion 195
8.1 Apprentissage supervisé d'un FMDP 195
8.1.1 Contributions 196
8.1.2 Limitations 197
8.2 Planification dans les FMDPs 202
8.2.1 Contributions 203
8.2.2 Limitations 204
8.3 Apprentissage par renforcement dans les FMDPs 206
8.3.1 Contributions 206
8.3.2 Limitations 208

Conclusion et perspectives 209

Bibliographie 214

Chapitre 1

Introduction

Une entreprise planifiant sa production, une compagnie aérienne organisant ses vols, un robot autonome découvrant son environnement ou bien encore un joueur dans un jeu vidéo ont tous en commun la nécessité de choisir un comportement à suivre dans le but de maximiser un ou plusieurs critères. Ces environnements sont dynamiques puisqu'ils évoluent au cours du temps et incertains puisque cette évolution ne peut pas être prédite avec certitude. Suivant le problème, le ou les critères à maximiser, aussi appelés *récompense*, peuvent représenter le chiffre d'affaire d'une entreprise, une ressource telle qu'une puissance énergétique disponible ou bien encore un score dans un jeu vidéo.

L'ensemble de ces applications supposent l'implication d'un *agent* (i.e. un comité d'entreprise, un chef d'équipe, un robot, un joueur) prenant des décisions en fonction d'un ensemble de paramètres décrivant l'état courant du problème. En fonction de ces paramètres, le but de l'agent est de prendre des décisions adaptées au fur et à mesure de l'évolution du système pour maximiser ses récompenses sur le long terme dans un environnement à la fois dynamique et incertain.

La résolution de ce type de problèmes de façon automatique est inévitablement confrontée au problème de la taille de l'espace de recherche. En effet, la prise de décision nécessite souvent l'analyse de nombreux paramètres empêchant une énumération complète de l'ensemble des solutions possibles pour des problèmes soit de temps de calcul, soit d'espace mémoire.

Pour surmonter cette difficulté, ces dix dernières années ont vu apparaître de nouveaux cadres mathématiques permettant de définir et de représenter des problèmes de décision de grande taille. Ces formalismes utilisent le fait qu'un problème, bien qu'il soit grand, est souvent structuré ; c'est-à-dire qu'il présente des régularités pouvant être exploitées afin de pouvoir le représenter de façon compacte, malgré sa taille. Ainsi, de tels formalismes proposent des représentations permettant de diminuer à la fois l'espace mémoire et les temps de calculs requis pour traiter le problème.

Des algorithmes, exploitant ces représentations adaptées à certaines structures de problèmes, ont été développés dans le cadre de ces formalismes. En évitant une énumération exhaustive de l'ensemble des possibilités, ces algorithmes permettent de calculer les solutions de grands problèmes

auparavant inaccessibles.

Cependant, l'inconvénient majeur de ces techniques est qu'elles supposent de connaître complètement à la fois la dynamique et la structure du problème à résoudre. Étant donné la structure d'un problème, des méthodes d'apprentissage ont été proposées pour évaluer la dynamique de celui-ci. Cependant, à notre connaissance, il n'existait pas avant nos travaux de techniques permettant d'apprendre simultanément la structure et la dynamique d'un problème donné. D'une façon générale, cette thèse se fixe donc comme objectif l'apprentissage de la structure et de la dynamique de grands problèmes dans l'incertain, avec pour but la construction du comportement d'un agent maximisant une ou plusieurs récompenses, au fur et à mesure de l'apprentissage.

1.1 Objectifs

Nos travaux s'inscrivent dans le cadre de l'*apprentissage par renforcement*, dans lequel un agent a pour objectif d'apprendre à sélectionner ses décisions en fonction de sa situation et à partir d'un signal de récompense (ou de punition). Plus précisément, l'apprentissage par renforcement fait référence aux problèmes dans lesquels le comportement de l'agent doit s'adapter en fonction d'une mesure quantitative (la récompense qu'il obtient) caractérisant la décision (ou la suite de décisions) qui vient d'être exécutée.

Il se distingue donc de *l'apprentissage supervisé* où la bonne solution est donnée au mécanisme d'apprentissage. Ainsi, plutôt que de spécifier ce que l'agent doit ou ne doit pas faire, la récompense qu'il obtient spécifie ce qui est bien et ce qui ne l'est pas. L'agent doit donc apprendre par essais-erreurs les décisions (ou la suite de décisions) correctes à réaliser. Par conséquent, lorsqu'un agent prend une décision, il est nécessaire qu'il explore d'autres décisions afin de pouvoir comparer la récompense obtenue pour chacune d'entre elles, contrairement à l'apprentissage supervisé où la bonne décision est directement spécifiée.

Le problème est formalisé en définissant un *agent* interagissant avec son *environnement* en choisissant une *action*. Nous supposons que l'agent sélectionne ses actions parmi un ensemble fini d'actions possibles. De plus, nous supposons que le temps est décomposé en pas de temps et que l'agent sélectionne une seule action (puis l'exécute) pour chacun d'entre eux. L'action représente donc la décision de l'agent à chaque pas de temps.

L'environnement est décrit, d'une part, de façon qualitative par un ensemble de paramètres, ou *variables*, composant l'*état* courant du système. Nous supposons que les variables peuvent prendre leurs valeurs parmi un ensemble de valeurs finies. D'autre part, une *fonction de récompense* associe un nombre réel à chaque couple état/action possible dans le système. Cette fonction quantifie l'état courant du système et spécifie l'objectif à maximiser par l'agent.

Une *transition* est le passage d'un état courant du système à un nouvel état au pas de temps suivant. Elle est définie par l'action sélectionnée par l'agent et par la *fonction de transition* décrivant

la dynamique du système. Afin de représenter l'incertain, l'environnement est *stochastique*, c'est-à-dire qu'à partir d'un même état initial et d'une même action sélectionnée, l'agent peut arriver de façon aléatoire dans plusieurs états différents au pas de temps suivant. De plus, nous supposons que l'état du système au prochain pas de temps ne dépend que de l'état courant du système et de la dernière action sélectionnée par l'agent. De tels systèmes sont appelés *markoviens* ou satisfaisant *l'hypothèse de Markov*.

Dans le cadre de l'apprentissage par renforcement, notre objectif est donc de trouver une méthode automatique permettant, dans un premier temps, de construire une représentation du problème et, dans un deuxième temps, d'utiliser cette représentation pour exécuter un comportement, ou *politique*, maximisant les récompenses obtenues sur le long terme. Nous supposons les caractéristiques suivantes sur le problème : il est stochastique, markovien et possède un nombre d'états et un nombre d'actions possibles finis. De plus, nous nous intéresserons plus particulièrement aux problèmes de grande taille et dont à la fois leur structure et leur dynamique sont inconnues a priori.

1.2 Méthodes

Un problème d'apprentissage par renforcement tel que nous venons de le décrire peut être formalisé à l'aide d'un processus de décision markovien (Bellman, 1957), ou *Markov Decision Process* (MDP). Les MDPs sont un cadre mathématique permettant de modéliser et de résoudre des problèmes de décision dans les environnements stochastiques. À partir de ce cadre, deux problèmes peuvent être définis. Le premier est un problème de *planification*. Dans ce cas, la dynamique du système, ou *fonction de transition*, et la fonction de récompense associées sont supposées être connues à l'avance. À partir de ces fonctions, le problème consiste alors à trouver une politique optimale maximisant la récompense obtenue sur le long terme. Le deuxième problème est un problème d'*apprentissage par renforcement* (Sutton and Barto, 1998). Dans ce cas, la fonction de transition et la fonction de récompense sont inconnues et l'agent doit construire sa politique par essais-erreurs.

Dans le cadre de l'apprentissage par renforcement, deux approches sont envisageables. La première, appelée *apprentissage par renforcement direct*, consiste à construire la solution au problème directement, sans construire de représentation explicite des fonctions de transition et de récompense du problème. La deuxième, *l'apprentissage par renforcement indirect*, construit une représentation de ces fonctions et les utilise ensuite pour construire la solution au problème. Nos travaux visent à adapter des techniques de planification utilisant une représentation des fonctions de transition et de récompense à des problèmes d'apprentissage par renforcement où ces fonctions sont inconnues a priori. Ils se placent donc dans le cadre de la deuxième approche : l'apprentissage par renforcement indirect.

1.2.1 Exploitation de la structure du problème

Dans les deux problématiques que sont la planification et l'apprentissage par renforcement, l'état courant du système peut être caractérisé par l'assignation de valeurs à un ensemble de variables. Dans ce cas, le nombre d'états possibles du système croît exponentiellement avec le nombre de variables décrivant l'état. Cette propriété est appelée par Bellman (1957) "*the curse of dimensionality*", littéralement "la malédiction de la dimensionalité". Or, les MDPs considèrent les états d'un problème comme atomiques et requièrent par conséquent des représentations exhaustives pour définir une fonction. Une telle caractéristique rend les MDPs inadaptés pour les grands problèmes.

En effet, si l'on considère la fonction de récompense d'un problème, elle qualifie chaque action pour chaque état : par conséquent, elle associe à chaque couple état/action une valeur réelle. Une représentation exhaustive requiert donc l'énumération de tous les couples état/action possibles. Une telle représentation est impossible à partir seulement de quelques dizaines de variables représentant les états possibles du problème. Le problème est amplifié pour la description de la fonction de transition du problème puisque, dans un problème stochastique, lorsqu'un agent réalise une action dans un état, plusieurs transitions sont possibles et l'agent peut donc arriver dans plusieurs états. Par conséquent, il est nécessaire à chaque couple état/action d'associer l'ensemble des états de destination possibles.

Pour résoudre cette difficulté, de nouvelles représentations et techniques provenant de la communauté de planification ont été proposées. Ces techniques proposent d'exploiter la structure du problème à résoudre pour représenter de façon compacte les fonctions de transition et de récompense qui lui sont associées. Plus précisément, Boutilier et al. (1995) proposent d'utiliser des *réseaux bayésiens dynamiques* (Dean and Kanazawa, 1989), ou *Dynamic Bayesian Networks* (DBNs) pour représenter les fonctions de transition et de récompense d'un MDP.

Les DBNs sont un modèle graphique permettant de représenter les dépendances entre les variables d'un problème au cours du temps. En effet, l'évolution de chaque variable ne dépend souvent que d'un petit nombre d'autres variables dans le problème. Les DBNs permettent donc de représenter de façon compacte, ou *factorisée*, les fonctions de transition et de récompense d'un problème. De telles représentations permettent de diminuer à la fois la complexité et la place mémoire requise pour les calculs nécessaires à la résolution du problème.

Ainsi, un MDP représenté par ces fonctions de transition et de récompense compactes est appelé un Processus de Décision Markovien *Factorisé*, ou *Factored Markov Decision Process* (FMDP) (Boutilier et al., 1995). Dans le cadre des FMDPs, des nouvelles techniques de planification ont été proposées. Ces techniques exploitent les représentations compactes des fonctions de transition et de récompense pour *agréger* des états similaires et réduire ainsi les temps de calcul et l'espace mémoire nécessaire pour la résolution du problème. De plus, certaines d'entre elles utilisent des représentations *approchées* des différentes fonctions du problème, réduisant ainsi l'espace mémoire qui serait nécessaire pour une représentation exacte. Lorsque la structure du problème est adaptée,

ces techniques permettent de résoudre des problèmes de beaucoup plus grande taille.

1.2.2 Apprentissage de la structure du problème

A notre connaissance, il n'existait aucune méthode avant nos travaux utilisant le cadre des FMDPs et qui ne supposait pas la connaissance a priori de la structure du problème à résoudre. Plus précisément, les dépendances entre les variables pour la description des fonctions de transition et de récompense étaient supposées connues. Cette thèse se concentre donc sur l'utilisation des méthodes développées pour la résolution des problèmes de planification dans un FMDP pour résoudre des problèmes d'apprentissage par renforcement de grande taille et sans connaissance a priori de la structure du problème. L'idée principale est d'utiliser des techniques d'apprentissage supervisé pour construire une représentation structurée des fonctions de transition et de récompense à partir de l'expérience de l'agent dans son environnement.

Pour cela, nous utilisons le fait qu'à chaque nouvelle action exécutée par l'agent, il obtient une nouvelle *observation* de l'environnement dans lequel il agit. Cette observation est composée de l'ancien état de l'agent, l'action exécutée, le nouvel état de l'agent et la récompense immédiate. Nous proposons d'exploiter ces observations pour construire une représentation structurée, c'est-à-dire exhibant les dépendances entre les variables du problème, des fonctions de transition et de récompense du problème à résoudre. Au fur et mesure de leurs constructions, ces représentations peuvent être utilisées par des techniques de planification dans le cadre des FMDPs pour calculer de façon incrémentale une politique à exécuter par l'agent.

1.2.3 Exploration et généralisation

Pour résoudre un problème d'apprentissage par renforcement, il est nécessaire que le comportement de l'agent comporte, d'une part, une phase d'*exploration* et, d'autre part, une phase d'*exploitation*. Pendant la phase d'exploration, l'agent doit découvrir les caractéristiques pertinentes de l'environnement sans forcément attendre une récompense. Au contraire, pendant la phase d'exploitation, l'agent exploite les caractéristiques apprises de l'environnement afin d'obtenir une récompense attendue. Le mécanisme de gestion de ces deux phases antagonistes est appelé compromis exploration/exploitation.

Découvrir les caractéristiques pertinentes de l'environnement lors de la phase d'exploration devient un problème critique lorsque l'environnement de l'agent est grand. En effet, lorsque le nombre d'états possibles est grand, il n'est plus possible d'explorer de façon systématique toutes les transitions du problème, surtout lorsque celles-ci sont stochastiques. La solution consiste alors pour l'agent à être capable de *généraliser* à partir de son expérience. Ainsi, pour que l'agent puisse réagir correctement à une situation qu'il n'a pas encore rencontrée mais qui est similaire à une observation précédente dans son historique, nos travaux utilisent des algorithmes d'apprentissage

ayant une capacité de généralisation. Nous verrons que cette capacité est étroitement liée à la capacité d'agrégation des algorithmes de planification lors de la construction de la politique.

Enfin, de nouveaux algorithmes ont été proposés, aussi bien dans le cadre des MDPs que des FMDPs, pour résoudre le dilemme exploration/exploitation. De plus, quelques-uns de ces algorithmes présentent des bornes garantissant le temps d'apprentissage. Nous avons donc considéré la mise en œuvre de ces solutions dans le cadre des FMDPs et lorsque la structure du problème est inconnue.

1.3 Principales contributions

Cette thèse se concentre sur une approche destinée à résoudre par apprentissage par renforcement des grands problèmes markoviens, stochastiques et discrets. Les principales contributions des travaux présentés dans cette thèse portent sur les points suivants :

Apprentissage de la structure : nous proposons une méthode générale utilisant des techniques d'apprentissage supervisé pour construire, sous la forme d'un FMDP, une représentation factorisée du problème à résoudre. D'une part, notre méthode permet de bénéficier des travaux précédents concernant l'apprentissage supervisé et, plus particulièrement, l'apprentissage supervisé incrémental. D'autre part, nous montrons que cette méthode s'appuie sur les propriétés de généralisation de l'apprentissage, évitant ainsi une exploration exhaustive de l'environnement. Enfin, nous montrons que les structures de données construites sont lisibles et peuvent faciliter la compréhension du problème à résoudre, ainsi que sa solution.

Intégration d'algorithmes de planification incrémental : dans le cadre d'une approche générale, appelée SDYNA et s'inspirant des travaux antérieurs dans les MDPs, nous proposons l'utilisation d'algorithmes de planification afin d'exploiter les connaissances acquises au fur et à mesure de l'évolution de l'agent dans son environnement. Nous montrons que, dans le cadre des FMDPs, les algorithmes de planification sont capables d'exploiter la propriété de généralisation de l'apprentissage de deux façons. La première est l'agrégation d'états similaires afin de diminuer les coûts du calcul de la solution en temps et en mémoire. La deuxième est l'utilisation de la généralisation pour construire une politique adaptée dans des situations que l'agent n'a pas encore rencontrées.

Application au jeu vidéo : nous illustrons notre approche en l'appliquant à un problème réel, celui d'apprendre une politique pour un personnage non joueur dans un jeu vidéo. Nous montrons que l'algorithme est capable de trouver une solution au problème et que cette solution est suffisamment lisible pour être comprise par une tierce personne, non spécialiste ni du domaine de l'apprentissage par renforcement, ni du domaine du jeu vidéo.

Intégration d'algorithmes d'exploration : dans le cadre de l'apprentissage par renforcement,

plusieurs algorithmes d'exploration avec des résultats théoriques intéressants, notamment concernant leur vitesse de convergence, ont été proposés dans la littérature. Nous montrons que quelques-uns de ces algorithmes peuvent s'intégrer dans SDYNA et permettent d'améliorer certains résultats concernant l'exploration des problèmes. Cependant, nous mettrons en évidence plusieurs limitations d'une telle approche.

1.4 Plan du mémoire

Dans un premier temps, nous décrirons le cadre des MDPs et des solutions, principalement basées sur la représentation ou la construction d'une représentation des fonctions de transition et de récompense, qui ont été proposées dans ce cadre (chapitre 2). Dans un deuxième temps, nous décrirons le cadre des FMDPs et des solutions de planification pour résoudre des problèmes modélisés dans ce cadre (chapitre 3). Ensuite, nous décrirons comment une méthode d'apprentissage supervisé peut être utilisée pour construire de façon automatique un FMDP représentant le problème à résoudre (chapitre 4). Le chapitre 5 présente SDYNA et décrit l'intégration de méthodes incrémentales d'apprentissage et de planification afin de résoudre un problème d'apprentissage par renforcement en ligne. Nous proposerons l'intégration des méthodes d'exploration dans l'architecture SDYNA lors du chapitre 6. Ensuite, chapitre 7, nous présenterons des résultats de SDYNA dans un cadre applicatif, plus particulièrement celui du jeu vidéo Counter-Strike©. Enfin, chapitre 8, nous discuterons des contributions et des limitations de notre travail.

Chapitre 2

Les Processus de Décision Markoviens

Ce chapitre introduit le cadre des processus de décision markoviens (section 2.1) en décrivant l'utilisation de ce cadre sous deux aspects différents. Le premier suppose une connaissance a priori complète du problème à résoudre (c'est-à-dire des fonctions de transition et de récompense) et décrit des méthodes pour trouver une solution à ce problème, sans nécessiter une expérience dans l'environnement (section 2.2). Le deuxième suppose que cette connaissance concernant le problème est acquise par essais-erreurs lors de l'expérience de l'agent dans l'environnement (section 2.3).

2.1 Définition d'un MDP

Les processus de décision markoviens, ou *Markov Decision Processes* (MDPs), sont une façon naturelle de formaliser un problème de décision dans l'incertain dans lequel la propriété de Markov est vérifiée. Un MDP fini se définit par :

- un ensemble S fini d'états discrets ;
- un ensemble A fini d'actions discrètes ;
- une fonction de transition $T : S \times A \rightarrow \Pi(S)$ avec $\Pi(S)$ l'ensemble des distributions de probabilités $P(s_{t+1}|s_t, a_t)$ avec s_t l'état à l'instant t et a_t l'action réalisée à l'instant t ;
- une fonction de récompense $R : S \times A \rightarrow \mathbb{R}$ représentant la récompense $R(s, a)$ obtenue en faisant l'action a dans l'état s.

L'hypothèse de Markov se définit formellement par l'égalité :

$$P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0) = P(s_{t+1}|s_t, a_t) \tag{2.1}$$

avec s_t l'état de l'agent à l'instant t et a_t l'action réalisée par l'agent à l'instant t. Pour simplifier les notations, nous noterons $P(s_{t+1}|s_t, a_t) = P(s'|s, a)$ avec s' l'état de l'agent à l'instant $t+1$, s l'état de l'agent à l'instant t et a l'action réalisée par l'agent à l'instant t. On suppose que le problème est stationnaire (les fonctions de transition et de récompense ne varient pas au cours du temps).

Il est souvent naturel de décrire un problème par un ensemble de paramètres pouvant prendre différentes valeurs décrivant l'état courant du système. Dans ce cas, l'ensemble des états possibles S est décrit par un ensemble de variables aléatoires $X = \{X_1, \dots, X_n\}$ où chaque variable X_i peut prendre différentes valeurs dans son domaine $\underline{\text{Dom}}(X_i)$. Un état est donc une instanciation de X décrite sous la forme d'un vecteur $x = \{x_1, \dots, x_n\}$ de valeurs x_i avec $\forall i \; x_i \in \underline{\text{Dom}}(X_i)$. De plus, on utilise comme raccourci d'écriture $\underline{\text{Dom}}(X)$ pour décrire l'ensemble des instanciations possible des variables $X_i \in X$. L'espace d'état S du MDP est donc $S = \underline{\text{Dom}}(X)$.

L'exemple CoffeeRobot

Afin d'illustrer les représentations utilisées tout au long de ce mémoire, nous utiliserons un exemple élémentaire, appelé *CoffeeRobot*, décrit par Boutilier et al. (2000). Un robot doit aller acheter un café pour sa propriétaire restant au bureau. Quand il pleut, comme le robot doit sortir pour aller chercher le café, il doit se munir d'un parapluie lorsqu'il est au bureau, sinon il sera mouillé. Pour décrire l'état du système, six variables aléatoires binaires[1] ($\underline{\text{Dom}}(X_i) = \{0, 1\}$ correspondant respectivement à Faux et Vrai) sont utilisées :

1. $\mathcal{HOC}$: la propriétaire a-t-elle un café ?
2. $\mathcal{HRC}$: le robot a-t-il un café ?
3. $\mathcal{W}$: le robot est-t-il mouillé ?
4. $\mathcal{R}$: est-ce qu'il pleut ?
5. $\mathcal{U}$: le robot a-t-il un parapluie ?
6. $\mathcal{O}$: le robot est-t-il au bureau ?

Par exemple, le vecteur $[\mathcal{HOC}{=}0, \mathcal{HRC}{=}1, \mathcal{W}{=}0, \mathcal{R}{=}1, \mathcal{U}{=}0, \mathcal{O}{=}1]$ représente un état de ce problème dans lequel la propriétaire n'a pas de café, le robot a un café, le robot n'est pas mouillé, il pleut, le robot n'a pas de parapluie et le robot est au bureau. Ce problème étant composé de 6 variables binaires, son espace d'états contient $2^6 = 64$ états possibles.

Le robot dispose de quatre actions :

– $\mathcal{G}\mathrm{o}$: se déplacer vers le lieu opposé ;
– $\mathcal{B}\mathrm{uyC}$: acheter un café, que le robot obtient s'il est au café ;
– $\mathcal{D}\mathrm{elC}$: donner le café à sa propriétaire, qu'elle peut obtenir si le robot est au bureau et qu'il a un café ;
– $\mathcal{G}\mathrm{etU}$: prendre un parapluie, que le robot peut obtenir s'il est au bureau.

[1]Principalement pour des raisons de simplicité d'exposition, la plupart des exemples décrits dans ce manuscrit utilisent des variables binaires. Cependant, rien ne limite l'utilisation des structures de données et des algorithmes exposés dans ce manuscrit à des problèmes contenant des variables non binaires (voir, par exemple, le chapitre 7).

L'effet de ces actions peut être bruité afin de représenter les cas stochastiques. Par exemple, lorsque le robot donne la tasse de café à sa propriétaire, la propriétaire obtiendra son café avec une certaine probabilité. L'action peut mal se passer, par exemple, lorsque le robot renverse le café. Ainsi, lorsque le robot exécute l'action $\mathcal{D}$elC dans l'état $s = [\mathcal{HOC}{=}0, \mathcal{HRC}{=}1, \mathcal{W}{=}0, \mathcal{R}{=}1, \mathcal{U}{=}0, \mathcal{O}{=}1]$ (le robot a un café et est au bureau, la propriétaire n'a pas de café), la fonction de transition définit :

- $P([\mathcal{HOC}{=}1, \mathcal{HRC}{=}1, \mathcal{W}{=}0, \mathcal{R}{=}1, \mathcal{U}{=}0, \mathcal{O}{=}1] | s, \mathcal{D}\text{elC}) = 0.8$,
- $P([\mathcal{HOC}{=}0, \mathcal{HRC}{=}1, \mathcal{W}{=}0, \mathcal{R}{=}1, \mathcal{U}{=}0, \mathcal{O}{=}1] | s, \mathcal{D}\text{elC}) = 0.2$,
- 0.0 pour les autres probabilités.

Enfin, le robot reçoit une récompense de 0.9 lorsque la propriétaire a un café (0 lorsqu'elle n'a pas de café) ajoutée à 0.1 lorsqu'il est sec (et 0 lorsqu'il est mouillé). La récompense obtenue lorsque la propriétaire a un café est supérieure à la récompense obtenue par le robot lorsqu'il reste sec pour indiquer que le premier objectif est prioritaire sur le deuxième. Dans cet exemple, la fonction de récompense ne dépend pas de l'action réalisée par le robot.

La fonction de transition T et la fonction de récompense R définissent complètement le problème à résoudre. Cependant, ce formalisme ne décrit en rien l'action a à exécuter pour maximiser la récompense lorsque l'agent est dans l'état s. En effet, les fonctions T et R expriment simplement la probabilité pour l'agent de se retrouver dans l'état s' et d'obtenir une récompense $R(s, a)$, s'il effectue l'action a dans l'état s.

2.1.1 Politiques et fonctions de valeur

Une politique $\pi : S \rightarrow A$ stationnaire et déterministe est une fonction $\pi(s)$ indiquant le fait de faire l'action a dans l'état s. Une politique π définit donc le comportement de l'agent en spécifiant l'action $\pi(s)$ à réaliser pour chaque état s.

Puisqu'une récompense est associée à tous les couples état/action, il est possible d'associer une valeur $V_\pi(s)$ à une politique π et un état s indiquant la quantité de récompense que l'agent peut espérer lorsqu'il est dans l'état s, puis qu'il applique la politique π. On utilise le critère de la récompense actualisée définie par :

$$V_\pi(s) = E_\pi \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \middle| s_0 = s \right] \tag{2.2}$$

avec $\gamma \in [0, 1[$ un paramètre déterminant l'importance attribuée aux récompenses obtenues plus tard. La valeur $V_\pi(s)$ peut être décomposée en une somme ajoutant la récompense immédiate obtenue par l'agent dans l'état s à l'espérance des récompenses obtenues dans les états suivants si l'agent suit la politique π multipliée par le facteur γ. Plus formellement, $V_\pi(s)$ peut donc se réécrire de façon récursive (Howard, 1960) :

$$V_\pi(s) = R_\pi(s) + \gamma \sum_{s'} P_\pi(s'|s) V_\pi(s') \tag{2.3}$$

avec $R_\pi(s) = R(s, \pi(s))$ et $P_\pi(s'|s) = P(s'|s, \pi(s))$.

Dans le problème *Coffee Robot*, la figure 2.1 montre un exemple de politique $\pi(s)$ ainsi que sa fonction de valeur associée $V_\pi(s)$. Il y a 64 états possibles dans le problème, la définition de ces deux fonctions nécessite donc 64 lignes dans une représentation tabulaire. Cependant, il est d'ores et déjà possible de remarquer certaines régularités dans la représentation de ces deux fonctions. Par exemple, leurs dernières lignes sont identiques. Nous verrons lors du chapitre suivant comment de telles régularités peuvent être exploitées.

États s	$\pi(s)$
$s_0 = [\mathcal{HOC}{=}0,\mathcal{HRC}{=}0,\mathcal{W}{=}0,\mathcal{R}{=}0,\mathcal{U}{=}0,\mathcal{O}{=}0]$	$\mathcal{B}$uyC
$s_1 = [\mathcal{HOC}{=}0,\mathcal{HRC}{=}0,\mathcal{W}{=}0,\mathcal{R}{=}0,\mathcal{U}{=}0,\mathcal{O}{=}1]$	$\mathcal{G}$o
$s_2 = [\mathcal{HOC}{=}0,\mathcal{HRC}{=}0,\mathcal{W}{=}0,\mathcal{R}{=}0,\mathcal{U}{=}1,\mathcal{O}{=}0]$	$\mathcal{B}$uyC
...	...
$s_{62} = [\mathcal{HOC}{=}1,\mathcal{HRC}{=}1,\mathcal{W}{=}1,\mathcal{R}{=}1,\mathcal{U}{=}0,\mathcal{O}{=}1]$	$\mathcal{G}$o
$s_{62} = [\mathcal{HOC}{=}1,\mathcal{HRC}{=}1,\mathcal{W}{=}1,\mathcal{R}{=}1,\mathcal{U}{=}1,\mathcal{O}{=}0]$	$\mathcal{G}$o
$s_{63} = [\mathcal{HOC}{=}1,\mathcal{HRC}{=}1,\mathcal{W}{=}1,\mathcal{R}{=}1,\mathcal{U}{=}1,\mathcal{O}{=}1]$	$\mathcal{G}$o

(a)

États s	$V_\pi(s)$
$s_0 = [\mathcal{HOC}{=}0,\mathcal{HRC}{=}0,\mathcal{W}{=}0,\mathcal{R}{=}0,\mathcal{U}{=}0,\mathcal{O}{=}0]$	6.9
$s_1 = [\mathcal{HOC}{=}0,\mathcal{HRC}{=}0,\mathcal{W}{=}0,\mathcal{R}{=}0,\mathcal{U}{=}0,\mathcal{O}{=}1]$	6.3
$s_1 = [\mathcal{HOC}{=}0,\mathcal{HRC}{=}0,\mathcal{W}{=}0,\mathcal{R}{=}0,\mathcal{U}{=}1,\mathcal{O}{=}0]$	6.3
...	...
$s_{62} = [\mathcal{HOC}{=}1,\mathcal{HRC}{=}1,\mathcal{W}{=}1,\mathcal{R}{=}1,\mathcal{U}{=}0,\mathcal{O}{=}1]$	9.0
$s_{62} = [\mathcal{HOC}{=}1,\mathcal{HRC}{=}1,\mathcal{W}{=}1,\mathcal{R}{=}1,\mathcal{U}{=}1,\mathcal{O}{=}0]$	9.0
$s_{63} = [\mathcal{HOC}{=}1,\mathcal{HRC}{=}1,\mathcal{W}{=}1,\mathcal{R}{=}1,\mathcal{U}{=}1,\mathcal{O}{=}1]$	9.0

(b)

FIG. 2.1 – Exemple d'une politique $\pi(s)$ (figure a) et de sa fonction de valeur $V_\pi(s)$ (figure b) dans le problème *Coffee Robot*. $\pi(s_0) = \mathcal{B}$uyC indique que, lorsque l'agent est dans l'état s_0, alors il exécutera l'action $\mathcal{B}$uyC. $V_\pi(s_0) = 6.9$ indique que dans l'état s_0, la récompense espérée sur le long terme quand le robot exécute la politique $\pi(s)$ est de 6.9.

Plutôt que de travailler sur la valeur $V(s)$ associée à un état, il est souvent plus intéressant de travailler sur la valeur $Q_a^V(s)$, appelée fonction de *valeur d'action* ou de *qualité*, associée à un couple état/action (s, a) et à une fonction de valeur $V(s)$. L'équation 2.4 définit $Q_a^V(s)$ pour un état s et une action a :

$$Q_a^V(s) = R(s,a) + \gamma \sum_{s'} P(s'|s,a)V(s') \tag{2.4}$$

De plus, pour toute fonction de valeur V, il est possible de définir une politique gloutonne $\underline{\text{Greedy}}_V$ relative à V :

$$\underline{\text{Greedy}}_V(s) = \arg\max_a \left[R(s,a) + \gamma \sum_{s'} P(s'|s,a)V(s') \right] \tag{2.5}$$

L'opération consiste à maximiser à chaque pas de temps les récompenses sur le long terme en choisissant l'action avec la plus grande valeur escomptée. L'équation 2.5 peut s'écrire plus simplement en fonction des fonctions de valeur d'action $Q_a^V(s)$:

$$\underline{\text{Greedy}}_V(s) = \arg\max_a \left[Q_a^V(s) \right] \tag{2.6}$$

2.1.2 Fonctions de valeur optimales

Dans un problème d'apprentissage par renforcement, le but d'un agent est d'apprendre à maximiser la somme des récompenses obtenues par l'agent au cours de l'expérience. Ainsi, il est intéressant de considérer l'ensemble des politiques optimales π^* permettant à l'agent d'obtenir le maximum de récompense lors de l'expérience. On définit donc $\pi \geq \pi'$ si et seulement si $\forall s \in S : V_\pi(s) \geq V_{\pi'}(s)$. Une politique π^* est optimale si et seulement si $\forall \pi : \pi^* \geq \pi$. De plus, pour tout MDP, il existe au moins une politique optimale π^* stationnaire et déterministe (Puterman, 1996). On définit la fonction de valeur optimale V^* (correspondant à la valeur de l'ensemble des politiques optimales π^*) comme étant :

$$V^*(s) = \max_{\pi} [V_\pi(s)] \tag{2.7}$$

Il est possible d'exprimer l'équation 2.3 correspondant à la fonction de valeur optimale V^* :

$$V^*(s) = \max_{a \in A} \left[R(s,a) + \gamma \sum_{s'} P(s'|s,a) V^*(s') \right] \tag{2.8}$$

La fonction de valeur d'action optimale $Q^*_a(s)$ est définie comme étant la fonction de valeur d'action pour l'action a et relative à la fonction de valeur optimale V^* :

$$Q^*_a(s) = R(s,a) + \gamma \sum_{s'} P(s'|s,a) V^*(s') \tag{2.9}$$

Lorsque la fonction de valeur optimale V^* est connue, une politique optimale π^* peut être construite de façon gloutonne à partir de V^* (ou des fonctions de valeur d'action optimales Q^*_a) :

$$\pi^*(s) = \underline{\text{Greedy}}_{V^*}(s) \tag{2.10}$$

Le but d'un agent résolvant un problème d'apprentissage par renforcement dans un MDP fini est donc d'apprendre à exécuter un comportement le plus proche possible des politiques appartenant à l'ensemble des politiques optimales.

2.2 Résolution d'un MDP

Cette section décrit deux méthodes de planification pour calculer la fonction de valeur optimale V^* et une politique optimale π^* d'un MDP donné : la programmation dynamique et la programmation linéaire, décrites respectivement section 2.2.1 et 2.2.2. Ces deux approches supposent donc la connaissance a priori des fonctions de transition et de récompense du MDP.

2.2.1 Programmation dynamique

La programmation dynamique désigne un ensemble d'algorithmes permettant de calculer les politiques optimales π^* d'un MDP fini. Ces algorithmes reposent sur les hypothèses suivantes :

1. la fonction de transition T est connue ;
2. la fonction de récompense R est connue.

Les algorithmes de programmation dynamique permettent donc de trouver l'ensemble des solutions d'un MDP uniquement si celui-ci est parfaitement connu.

Généralement, deux étapes composent un algorithme de programmation dynamique : l'évaluation d'une politique et l'amélioration d'une politique.

Évaluation d'une politique π

L'évaluation d'une politique π consiste à calculer $V^\pi(s)\ \forall s \in S$. À partir de l'équation 2.3, il est possible d'écrire un algorithme incrémental permettant d'évaluer $V^\pi(s)$. Ainsi l'équation de mise à jour pour un état s lors d'une itération k de l'algorithme se définit par :

$$V_\pi^{k+1}(s) = R_\pi(s) + \gamma \sum_{s'} P_\pi(s'|s) V_\pi^k(s') \tag{2.11}$$

Lorsque k tend vers l'infini, la fonction $V_\pi^k(s)$ tend vers $V_\pi(s)$ (Puterman, 1996). L'algorithme 2.2 permet donc de connaître la fonction de valeur $V^\pi(s)$ pour la politique π lorsque les fonctions de transition et de récompense sont connues.

Amélioration d'une politique π

L'évaluation d'une politique π est nécessaire pour l'améliorer. En effet, l'algorithme d'évaluation d'une politique nous permet de comparer deux politiques afin de déterminer laquelle est la meilleure.

Ainsi, l'une des méthodes possibles pour améliorer une politique π est de tester si, pour un état s, en réalisant l'action a plutôt que $\pi(s)$ puis en se conformant à la politique π, $V_\pi(s)$ est améliorée. Il est alors possible de définir une nouvelle politique $\pi'(s)$ à partir d'une politique $\pi(s)$ et de sa fonction de valeur $V_\pi(s)$:

$$\begin{aligned} \pi'(s) &= \arg\max_a \left[R(s,a) + \gamma \sum_{s'} P(s'|s,a) V_\pi(s') \right] \\ &= \underline{\text{Greedy}}_{V_\pi}(s) \end{aligned} \tag{2.12}$$

Pour chaque état s, l'opération consiste donc à choisir de façon gloutonne l'action a pour la politique $\pi'(s)$ amenant à l'état s' possédant la valeur $V_\pi(s')$ la plus intéressante. Lorsque la fonction de valeur est la même pour la politique π et sa politique améliorée π', cela signifie qu'il n'est plus possible de les améliorer et donc que ces politiques sont optimales.

Entrée(s) : $\pi(s)$ **Sortie(s) :** $V_\pi(s)$

1. Initialiser V_π arbitrairement, (exemple : $\forall s \in S$, $V_\pi(s) = R_\pi(s)$)
2. Répéter :
   (a) $\Delta \leftarrow 0$
   (b) Pour chaque $s \in S$:
      i. $v \leftarrow V_\pi(s)$
      ii. $V_\pi(s) = R_\pi(s) + \gamma \sum_{s'} P_\pi(s'|s) V_\pi(s')$
      iii. $\Delta \leftarrow \max(\Delta, |v - V_\pi(s)|)$

   tant que $\Delta < \epsilon$ (avec ϵ un petit nombre positif)
3. Retourner $V_\pi(s)$

FIG. 2.2 – Algorithme itératif d'évaluation d'une politique

L'algorithme Policy Iteration

Une fois qu'une politique π a été améliorée pour obtenir la politique π', il est possible d'évaluer la politique π' en utilisant l'algorithme 2.2, puis de l'améliorer une nouvelle fois pour obtenir la politique π'' et ainsi de suite. Puisqu'un MDP fini a un nombre fini de politiques, cette méthode converge jusqu'à obtenir une politique optimale et une fonction de valeur optimale en un nombre fini d'itérations. Cette méthode est appelée *Policy Iteration*. L'algorithme complet est décrit dans la figure 2.3.

L'algorithme *Policy Iteration* permet donc, à partir de la définition d'un MDP et en alternant les phases d'évaluation et d'amélioration de politiques, de calculer une des politiques optimales et la fonction de valeur optimale de ce MDP.

L'algorithme Value Iteration

L'un des problèmes de l'algorithme *Policy Iteration* est qu'il est très coûteux en temps de calcul. En effet, à chaque itération, il est nécessaire de réaliser les opérations suivantes :

1. évaluer la politique π en cours sur l'ensemble des états $s \in S$ autant de fois qu'il est nécessaire pour que $V_\pi(s)$ converge ;
2. améliorer la politique π en cours sur l'ensemble des états $s \in S$.

L'étape d'évaluation de la politique π est particulièrement coûteuse puisqu'il est nécessaire de mettre à jour la fonction de valeur $V_\pi(s)$ sur l'ensemble des états possibles jusqu'à ce que cette

Entrée(s) : $\emptyset$ **Sortie(s) :** $V^*(s), \pi^*(s)$

1. Initialisation : choisir $V_\pi(s)$ et $\pi(s)$ de façon arbitraire

Évaluation de la politique :

2. Répéter
   (a) $\Delta \leftarrow 0$
   (b) Pour chaque $s \in S$:
      i. $v \leftarrow V_\pi(s)$
      ii. $V_\pi(s) \leftarrow R_\pi(s) + \gamma \sum_{s'} P_\pi(s'|s) V_\pi(s')$
      iii. $\Delta \leftarrow \max(\Delta, |v - V_\pi(s)|)$

   tant que $\Delta < \epsilon$ (avec ϵ un petit nombre positif)

Amélioration de la politique :

3. $politiqueStable \leftarrow true$
4. Pour chaque $s \in S$:
   (a) $b \leftarrow \pi(s)$
   (b) $\pi(s) \leftarrow \underline{\text{Greedy}}_{V_\pi}(s)$
   (c) Si $b \neq \pi(s)$ alors $politiqueStable \leftarrow false$

5. Si non$(politiqueStable)$ alors aller en 2
6. Retourner $V_\pi(s)$ et $\pi(s)$

FIG. 2.3 – Algorithme *Policy Iteration* (d'après Sutton and Barto (1998))

fonction converge. Cependant, pour certains problèmes, il n'est pas forcément nécessaire d'attendre la convergence $V_\pi(s)$ pour améliorer la politique π.

En effet, sans perdre les propriétés de convergence, il est possible de fixer un nombre quelconque d'itérations pour l'évaluation de la politique. Un cas particulier est lorsque ce nombre d'itérations est fixé à 1, l'algorithme s'appelle alors *Value Iteration*. De la même façon que l'algorithme *Policy Iteration*, l'algorithme *Value Iteration*, décrit figure 2.4, prend en entrée la définition d'un problème sous la forme d'un MDP (avec une fonction de récompense et une fonction de transition) et calcule la fonction de valeur optimale du problème ainsi qu'une politique optimale.

Étant donné un problème décrit en utilisant le formalisme d'un MDP fini, les deux algorithmes, *Policy Iteration* et *Value Iteration*, permettent donc de trouver la solution à ce problème en déterminant la politique optimale pour un agent pour ce problème. Cependant, ces deux algorithmes

Entrée(s) : $\emptyset$ **Sortie(s) :** $V^*(s), \pi^*(s)$

1. Initialiser V arbitrairement, (exemple : $\forall s \in S, V(s) = 0$)
2. Répéter
   (a) $\Delta \leftarrow 0$
   (b) Pour chaque $s \in S$:
      i. $v \leftarrow V(s)$
      ii. $V(s) \leftarrow \max_a [R(s,a) + \gamma \sum_{s'} P(s'|s,a)V(s')]$
      iii. $\Delta \leftarrow \max(\Delta, |v - V(s)|)$

   tant que $\Delta < \epsilon$ (avec ϵ un petit nombre positif)
3. $\pi^*(s) \leftarrow \underline{\text{Greedy}}_V(s)$
4. Retourner V et π^*

FIG. 2.4 – Algorithme *Value Iteration*

souffrent d'inconvénients majeurs ne permettant pas leur utilisation pour des problèmes réels :

1. elles sont coûteuses en temps de calcul et en mémoire puisqu'il est nécessaire, pour les deux algorithmes, de parcourir explicitement et plusieurs fois l'ensemble de l'espace d'états afin de mettre à jour les valeurs correspondantes à chaque état. Or, le nombre d'états croît de façon exponentielle en fonction du nombre de variables utilisées pour décrire l'espace d'états et du nombre de valeurs que peuvent prendre ces variables. Par exemple, ajouter une variable supplémentaire dans le problème *Coffee Robot* fait passer le nombre d'états possibles de 64 à 128 (soit 512 couples état/action), 2 variables supplémentaires de 128 à 256 (soit 1024 couples état/action), etc. Rapidement, l'espace d'états devient trop grand, soit en terme de mémoire requise, soit en temps de calcul, pour que ces algorithmes soient utilisables ;
2. les données du problème, c'est-à-dire la fonction de transition T et la fonction de récompense R doivent être connues a priori.

Cependant, ces algorithmes restent intéressants puisqu'ils proposent une solution exacte à un problème donné et leurs principes restent applicables dans d'autres contextes, comme nous le verrons dans les sections 2.3.2 et 3.2.

2.2.2 Programmation linéaire

Lorsque les fonctions de transition et de récompense sont connues, la programmation linéaire est une approche alternative à la programmation dynamique pour calculer la fonction de valeur

optimale d'un MDP. Un MDP peut se formuler sous la forme d'un programme linéaire de la façon suivante (Manne, 1960) :

$$\begin{aligned} \text{Déterminer} \quad & V(s), \forall s \in S; \\ \text{minimisant} \quad & \sum_s \alpha(s)V(s); \\ \text{et satisfaisant} \quad & V(s) \geq R(s,a) + \gamma \sum_{s'} P(s'|s,a)V(s'), \forall s \in S, \forall a \in A \end{aligned} \tag{2.13}$$

avec $\alpha(s) > 0$ la pondération d'intérêt de l'état s (*state relevance weight*) et dont la somme est, habituellement, normalisée ($\sum_s \alpha(s) = 1$). Il est intéressant de noter que la solution optimale obtenue avec la formulation de l'équation 2.13 ne dépend pas de la définition de $\alpha(s)$. Une fois la fonction de valeur optimale $V^*(s)$ calculée, une politique optimale peut ensuite être calculée en utilisant $\pi^*(s) = \underline{\text{Greedy}}_{V^*}(s)$.

Ainsi, étant donné un problème décrit en utilisant le formalisme d'un MDP, le programme linéaire de l'équation 2.13 permet de calculer une solution optimale. Cependant, cette approche souffre des mêmes inconvénients que la programmation dynamique :

1. les fonctions de transition T et de récompense R doivent être connues ;
2. la résolution est coûteuse en temps de calcul et en mémoire. En effet, dans le programme linéaire de l'équation 2.13, le nombre exponentiellement croissant d'états possibles dans le problème se retrouve à la fois dans le nombre de variables à déterminer, dans le nombre de termes de la somme à minimiser, le nombre de contraintes à satisfaire et le nombre de termes dans le produit de chaque contrainte.

Cependant, de même que les algorithmes de programmation dynamique, le principe reste intéressant puisqu'il permet de calculer une solution exacte à un problème donné. Nous verrons dans la section 3.4 que cette méthode, à l'aide de techniques supplémentaires, a été adaptée pour résoudre des problèmes de grande taille.

2.3 Planification et apprentissage dans les MDPs

Les techniques de planification telles que la programmation linéaire et la programmation dynamique font l'hypothèse que les fonctions de transition et de récompense du MDP à résoudre sont connues. Or cette hypothèse n'est pas adaptée à de nombreux problèmes. Cette section décrit des méthodes d'apprentissage par renforcement permettant de trouver la solution à un MDP par essais-erreurs lorsque les fonctions de transition et de récompense sont inconnues.

Dans un premier temps, nous décrirons dans la section 2.3.1 l'algorithme Q-LEARNING. Notre objectif étant de pouvoir utiliser des techniques de planification dans des problèmes d'apprentissage par renforcement, nous nous concentrons ensuite dans la section 2.3.2 sur l'approche DYNA qui a

été proposée par Sutton (1990). Cette approche nous intéresse particulièrement puisqu'elle intègre des techniques de planification avec de l'apprentissage.

2.3.1 L'algorithme Q-learning

L'algorithme Q-LEARNING (Watkins, 1989) propose d'estimer les fonctions de valeur d'action $Q_a(s)$ au fur et à mesure de l'expérience de l'agent dans son environnement. Il est important de noter que Q-LEARNING ne suppose pas la connaissance des fonctions de transition et de récompense du MDP à résoudre. De plus, il ne construit aucune représentation de ces fonctions lors de son apprentissage et évalue directement les fonctions optimales du MDP modélisant le problème. Une telle méthode est appelée apprentissage par renforcement *direct*, par opposition aux méthodes d'apprentissage par renforcement *indirect* qui calculent des fonctions optimales au problème à partir de fonctions de transition et de récompense qu'elles auront construites de façon explicite par apprentissage (telle que l'approche DYNA que nous présentons dans la section 2.3.2 suivante). L'algorithme est décrit figure 2.5.

Paramètre(s) : α

Initialisation : $\forall a \in A, \forall s \in S$ définir $Q_a(s)$ de façon arbitraire

À chaque pas de temps : pour un état s :

1. Choisir une action a en fonction des $Q_a(s)$ (par exemple en utilisant ϵ-greedy)
2. Exécuter a, observer s' et r
3. $Q_a(s) \leftarrow Q_a(s) + \alpha(r + \gamma \max_{a'} [Q_{a'}(s')] - Q_a(s))$

FIG. 2.5 – L'algorithme Q-LEARNING

Le paramètre α est appelé taux d'apprentissage et détermine l'importance de la correction réalisée sur la fonction $Q_a(s)$ lors d'une mise à jour. Ce paramètre peut varier au fur et à mesure de l'expérience. ϵ-greedy fait référence à une politique d'exploration : à chaque fois qu'une action doit être décidée, l'action ayant la valeur d'action la plus élevée est choisie la plupart du temps, avec une probabilité faible ϵ de choisir une action de façon aléatoire (l'action est alors sélectionnée suivant une distribution uniforme indépendante des valeurs d'action). À chaque pas de temps, la probabilité de choisir la meilleure action estimée est donc de $1-\epsilon$. La valeur ϵ est en général relativement faible ($0 < \epsilon < 0.2$).

L'algorithme Q-LEARNING apprend donc une estimation des fonctions de valeur d'action $Q_a(s)$ de façon incrémentale au fur et à mesure des observations $\langle s, a, s', r \rangle$ (étape 3, figure 2.5). Si chaque

action est exécutée dans chaque état un nombre infini de fois et que α tend vers 0, alors les fonctions de valeur d'action $Q_a(s)$ convergent vers $Q^*_a(s)$ (Watkins and Dayan, 1992; Tsitsiklis, 1994; Jaakkola et al., 1994).

Il est important de noter que l'algorithme Q-LEARNING suppose que les fonctions $Q_a(s)$ sont représentées de façon tabulaire, rendant ainsi l'algorithme difficile à mettre en œuvre sur des problèmes de grande taille. Enfin, Q-LEARNING ne permet pas de généraliser l'expérience de l'agent à des situations encore inconnues, imposant ainsi une exploration explicite de l'ensemble des couples état/action du problème.

2.3.2 L'approche DYNA

A première vue, les techniques de planification (section 2.2) dans le cadre des MDPs peuvent sembler opposées aux techniques d'apprentissage par renforcement telles que Q-LEARNING (section 2.3.1). En effet, les premières supposent une connaissance complète des fonctions de transitions et de récompenses et ne nécessitent donc pas d'expérimentation dans l'environnement. Les deuxièmes supposent ces fonctions inconnues et nécessitent donc un apprentissage par essais-erreurs pour construire une solution au MDP.

Cependant, Sutton (1990) propose une vue unifiée de ces deux approches à travers l'architecture DYNA. En effet, cette architecture intègre ensemble la prise de décision, l'apprentissage et la planification pour résoudre un problème d'apprentissage par renforcement dont les fonctions de transition et de récompense sont inconnues. D'une part, à partir de chaque observation de l'agent dans son environnement, l'apprentissage est utilisé pour construire de façon incrémentale une représentation du problème sous la forme d'une fonction $\hat{T}$ de transition et d'une fonction $\hat{R}$ de récompense d'un MDP. D'autre part, à chaque pas de temps, une phase de planification utilise les représentations $\hat{T}$ et $\hat{R}$ représentant le problème pour mettre à jour les fonctions de valeur d'action $Q_a(s)$. Enfin, les fonctions $Q_a(s)$ sont utilisées par l'agent pour prendre la prochaine décision.

Plusieurs méthodes existent lors de la phase de planification pour mettre à jour les fonctions de valeur d'action $Q_a(s)$. Nous nous concentrons sur l'algorithme DYNA-Q (Sutton, 1990; Sutton and Barto, 1998) utilisant la formule de mise à jour de l'algorithme Q-LEARNING (section 2.3.1). En supposant que l'environnement est déterministe, DYNA-Q est décrit figure 2.6.

La phase de prise de décision est similaire à l'algorithme Q-LEARNING : il s'agit de prendre une décision en fonction des fonctions de valeur d'action et d'une politique d'exploration donnée (étape 1). A l'instar de Q-LEARNING, DYNA-Q peut utiliser l'algorithme ϵ-greedy. La phase d'apprentissage est décomposée en deux étapes : d'une part, les fonctions $Q_a(s)$ sont mises à jour exactement de la même façon que Q-LEARNING (étape 3), d'autre part la connaissance apportée par l'observation de l'agent dans l'environnement est ajoutée aux représentations $\hat{T}$ et $\hat{R}$ des fonctions de transition et de récompense (étape 4). Enfin, une phase de planification plus ou moins complète,

Paramètre(s) : α, N

Initialisation : $\forall a \in A, \forall s \in S$ définir $Q_a(s)$ de façon arbitraire

À chaque pas de temps : pour un état s :

Décision :

1. Choisir une action a en fonction des $Q_a(s)$ (par exemple en utilisant ϵ-greedy)
2. Exécuter a, observer s' et r

Apprentissage :

3. $Q_a(s) \leftarrow Q_a(s) + \alpha(r + \gamma \max_{a'} [Q_{a'}(s')] - Q_a(s))$
4. Mettre à jour $\hat{T}(s,a)$ à partir de $\langle s, a, s' \rangle$ et $\hat{R}(s,a)$ à partir de $\langle s, a, r \rangle$

Planification :

5. Répéter N fois :
   (a) $s \leftarrow$ un état observé choisi aléatoirement
   (b) $a \leftarrow$ une action déjà exécutée dans s et choisie aléatoirement
   (c) Déterminer s' tel que $\hat{P}(s'|s,a) = 1$ (l'environnement est supposé déterministe)
   (d) $r \leftarrow \hat{R}(s,a)$
   (e) $Q_a(s) \leftarrow Q_a(s) + \alpha(r + \gamma \max_{a'} [Q_{a'}(s')] - Q_a(s))$

FIG. 2.6 – L'algorithme DYNA-Q

suivant la valeur de N, se déroule en choisissant de façon aléatoire des couples état/action déjà visités et en utilisant le modèle du problème, c'est-à-dire les représentations $\hat{T}$ et $\hat{R}$, pour mettre à jour la fonction $Q_a(s)$ correspondante (étape 5).

L'algorithme DYNA-Q décrit dans la figure 2.6 peut être généralisé pour les problèmes dans lesquels les fonctions de transition et de récompense sont stochastiques (Peng and Williams, 1992). Dans ce cas, les mises à jour sont pondérées par la probabilité estimée $\hat{P}(s'|s,a)$ de la fonction de transition $\hat{T}$. L'équation de mise à jour devient alors :

$$Q_a(s) \leftarrow Q_a(s) + \alpha \left(\hat{P}(s'|s,a) \left(r + \gamma \max_{a'} [Q_{a'}(s')] - Q_a(s) \right) \right) \tag{2.14}$$

$Q_a(s)$ est mise à jour lors de l'étape 5e pour tous les états s' tel que $\hat{P}(s'|s,a) > 0$.

Pour conclure, l'algorithme DYNA-Q fait les mêmes hypothèses que l'algorithme Q-LEARNING, c'est-à-dire que les fonctions de transition et de récompense sont inconnues. Aussi, il souffre du même inconvénient : l'algorithme est basé sur des représentations tabulaires pour représenter les fonctions du problème. Par conséquent, il n'est pas adapté à la résolution de grands problèmes puisqu'il nécessite une représentation explicite des couples état/action. De plus, il ne permet pas de

généraliser à partir de l'historique de l'agent.

2.4 Synthèse

Dans ce chapitre, nous avons décrit le cadre mathématique des MDPs. Ce cadre peut être utilisé pour la description de deux types de problématiques. La première suppose une connaissance complète et a priori des fonctions de transition et de récompense du problème. Dans ce cas, des méthodes de planification, basées soit sur la programmation dynamique, soit sur la programmation linéaire, peuvent être utilisées pour calculer une solution au problème.

La deuxième problématique correspond aux problèmes dans lesquels les fonctions de transition et de récompense du problème sont inconnues a priori. Dans ce cas, trouver une solution au MDP est un problème d'apprentissage par renforcement dans lesquels l'agent doit construire une solution par essais-erreurs. Nous avons décrit deux méthodes classiques pour la résolution de tels problèmes d'apprentissage, nommément Q-LEARNING, une méthode d'apprentissage par renforcement directe, et DYNA-Q une méthode d'apprentissage par renforcement indirect intégrant des techniques de planification avec l'apprentissage.

Chapitre 3

Les Processus de Décision Markoviens Factorisés

L'ensemble des solutions décrites dans le cadre des MDPs (chapitre 2), que ce soit pour la planification ou l'apprentissage par renforcement, partagent toutes une même limitation : elles ne sont pas adaptées à la résolution de problèmes de grande taille. Ce chapitre vise donc à décrire une extension des MDPs, appelée *Processus de Decision Markoviens factorisés* et permettant de représenter les fonctions de transition et de récompense d'un problème de façon compacte (section 3.1). Une fois le problème représenté de façon compacte, nous décrirons plusieurs méthodes de planification permettant de trouver les solutions optimales ou optimales approchées (section 3.2, 3.3 et 3.4), tout en exploitant la structure du problème afin d'éviter une énumération explicite de l'espace d'état. Ce chapitre n'aborde pas les méthodes d'apprentissage existantes dans ce cadre, qui seront présentées chapitre 6.

3.1 Les Processus de Decision Markoviens Factorisés

L'un des inconvénients majeurs du cadre des MDPs est la taille des représentations des fonctions du problème à résoudre, plus précisément les fonctions de transition et de récompense. En effet, les algorithmes et leurs représentations associées nécessitent une énumération explicite de l'ensemble des états possibles du problème pour ces fonctions. Ainsi, pour la résolution de problèmes dont l'espace d'états est grand, les MDPs ne sont pas adaptés parce qu'ils ne permettent ni de représenter ce problème par la description des fonctions de transition et de récompense, ni d'utiliser les algorithmes permettant d'obtenir la solution au problème.

Pour surmonter cette difficulté, une extension aux MDPs a été présentée par Boutilier et al. (1995, 1999) : les *Processus de Décision Markoviens Factorisés* (*Factored Markov Decision Processes* (FMDPs)). Étant donné la décomposition de l'espace d'états en un ensemble de variables

aléatoires, les principales contributions de ce cadre mathématique sont d'une part, de pouvoir *décomposer* les fonctions de transition et de récompense (respectivement de façon multiplicative et additive) et d'autre part, de pouvoir exploiter *les indépendances relatives aux fonctions* liées à la structure du problème pour la description et la manipulation des fonctions de ce problème. De plus, les FMDPs offrent un cadre approprié à l'utilisation, de façon complémentaire mais pas obligatoire, de deux autres propriétés liées à la structure d'un problème : *les indépendances relatives aux contextes* et *l'approximation additive*.

Les indépendances relatives aux fonctions expriment le fait que certaines définitions du problème ne dépendent pas systématiquement de toutes les autres variables du problème ou bien de l'action réalisée par l'agent. Par exemple, dans le problème *Coffee Robot*, la valeur de la variable $\mathcal{R}$ au prochain pas de temps et indiquant s'il pleut ou non ne dépend que de sa propre valeur au pas de temps courant. En effet, le fait qu'il va pleuvoir au prochain pas de temps est indépendant des variables telles que "est ce que le robot a le café ou pas ?" ou de l'action exécutée par l'agent. Le cadre des FMDPs permet d'exploiter cette propriété principalement dans la description des fonctions de transition et de récompense du problème et dans l'utilisation de ces fonctions par les algorithmes de planification (cette notion est formalisée par deux opérateurs, Parents et Scope, qui sont définis respectivement dans les sections 3.1.1 (page 40) et 3.1.2 (page 42)).

Les indépendances relatives aux contextes concernent le fait que pour représenter une fonction du problème à résoudre (quelle que soit la fonction), il n'est pas nécessaire de tester systématiquement l'ensemble des variables nécessaires à la représentation de cette fonction. Un contexte se définit de la façon suivante :

Définition 1 (Contexte) *Soit une fonction* $f : \{X_0, \ldots, X_n\} \rightarrow Y$. *Un contexte* $c \in \underline{Dom}(C)$ *est une instanciation d'un sous-ensemble de variables* $C = \{C_0, \ldots, C_j\}$ *tel que* $C \subseteq \{X_0, ..., X_n\}$. *Un contexte est noté* $(C_0 = c_0) \wedge \ldots \wedge (C_j = c_j)$ *ou* $C_0 = c_0 \wedge \ldots \wedge C_j = c_j$.

Par exemple, la description de la politique optimale dans le problème *Coffee Robot* nécessite l'utilisation de toutes les variables du problème. Cependant, dans le contexte $\mathcal{HOC} = 0 \wedge \mathcal{HRC} = 1 \wedge \mathcal{O} = 1$, c'est-à-dire lorsque la propriétaire n'a pas de café et que le robot a un café et qu'il est au bureau, il est possible de déterminer l'action optimale (l'action $\mathcal{D}$elC dans ce cas) sans avoir à tester d'autres variables telles que "est-ce-qu'il pleut ?" ou "le robot est-t-il mouillé ?".

Enfin, l'approximation additive d'un problème fait référence à la fonction de valeur qui peut, pour certains problèmes, être approchée par une combinaison linéaire, c'est-à-dire une somme pondérée, de fonctions de base plus simples, chacune ne dépendant que d'un petit nombre de variables. Par exemple, dans le problème *Coffee Robot*, nous verrons que la fonction de valeur du problème peut être approchée par deux fonctions de base, l'une étant relative au fait que la propriétaire ait un café ou non, la deuxième étant relative au fait que le robot soit mouillé ou non. De plus, nous verrons que cette approximation additive peut être exploitée dans la représentation de l'ensemble des fonctions de valeur du problème.

3.1.1 Représentation de la fonction de transition

Contrairement aux MDPs, les FMDPs supposent que l'ensemble des états possibles est nécessairement décomposé en un ensemble de variables aléatoires (décrit section 2.1). En effet, l'avantage d'une telle représentation est qu'il est possible de décomposer une probabilité $P(s'|s)$ en un produit de probabilités, puis d'exploiter les indépendances relatives aux fonctions décrivant l'état du système dans la description de la fonction de transition.

Par exemple, admettons qu'un espace d'état soit décrit avec trois variables binaires X, Y et Z. Pour énumérer l'ensemble des combinaisons possibles $P(s'|s)$, il est nécessaire de décrire une table contenant $2^{2*3} = 64$ entrées. En décomposant la probabilité $P(s'|s)$ en un produit de probabilités, on obtient :

$$\begin{aligned} P(s'|s) &= P(X', Y', Z'|s) \\ &= P(X'|s)P(Y'|s, X')P(Z'|s, X', Y') \end{aligned}$$

avec X représentant la valeur de la variable X au pas de temps t et, X' la valeur de la variable X au pas de temps $t + 1$. De plus, si les relations de dépendance entre les variables sont connues, par exemple, chacune des variables X, Y et Z ne dépend que de sa valeur dans l'état précédent sauf la variable Y qui dépend aussi de X dans l'état précédent, alors $P(s'|s)$ peut s'écrire de façon plus compacte :

$$\begin{aligned} P(s'|s) &= P(X'|s)P(Y'|s, X')P(Z'|s, X', Y') \\ &= P(X'|X)P(Y'|Y, X)P(Z'|Z) \end{aligned}$$

En agrégeant les états pour lesquels la fonction de transition est identique, seules $2^1 + 2^2 + 2^1 = 8$ entrées sont nécessaires et réparties en trois tables différentes, une pour chaque variable (correspondant respectivement à la description des distributions de probabilités $P(X'|X)$, $P(Y'|Y, X)$ et $P(Z'|Z)$).

Ainsi, les indépendances relatives aux fonctions liées à la structure du problème sont mises en évidence et permettent ainsi d'agréger certaines régularités dans la description de la fonction de transition. De plus, elles correspondent à une représentation intuitive consistant à décrire l'effet de chaque action sur la valeur de chacune des variables du problème. Cette représentation de la fonction de transition est formalisée en utilisant le cadre des réseaux bayésiens dynamiques (Boutilier et al., 1995).

Les réseaux bayésiens dynamiques

Les réseaux bayésiens (Pearl, 1988) sont un formalisme permettant de représenter graphiquement des dépendances (ou indépendances) entre des variables. Les variables constituent les nœuds

d'un graphe orienté, les relations de dépendance probabiliste entre deux variables sont représentées par un arc entre les deux nœuds représentant chacun des variables. Les réseaux bayésiens dynamiques (Dean and Kanazawa, 1989) (*Dynamic Bayesian Networks (*DBNs*)*) sont des réseaux bayésiens permettant de représenter les données temporelles engendrées par des processus stochastiques.

En faisant l'hypothèse que le problème observé est stationnaire (donc la fonction de transition T du MDP ne varie pas au cours du temps), il est possible de représenter T avec des DBNs faisant seulement apparaître deux pas de temps successifs. Dans ce cas, les DBNs sont composés de deux groupes de nœuds :

1. le groupe de nœuds représentant l'ensemble des variables de l'espace d'état à l'instant t ;
2. le groupe de nœuds représentant l'ensemble des variables de l'espace d'état à l'instant $t + 1$.

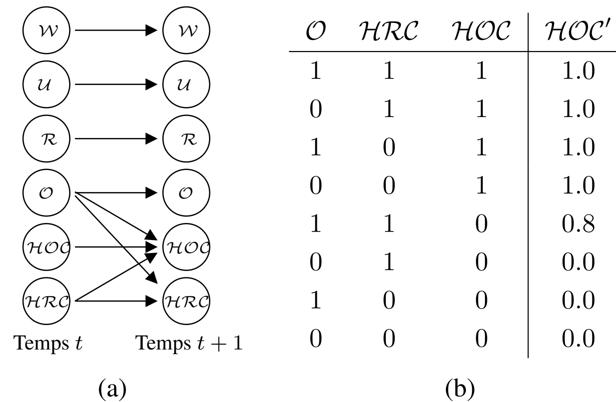
Les arcs indiquent alors les dépendances entre les variables à l'instant t et les variables à l'instant $t + 1$ ou encore des dépendances entre les variables à l'instant $t + 1$ (ces arcs sont appelés arcs synchrones). Dans ce cas particulier, un DBN est quelquefois appelé *2 Time Bayesian Network*. Pour la suite de ce mémoire, nous ferons l'hypothèse que les arcs synchrones ne sont pas nécessaires pour décrire le modèle des transitions du problème.

Il est alors possible de représenter complètement la fonction de transition en utilisant un DBN par variable et par action. L'action exécutée par l'agent peut aussi être considérée comme une variable à l'instant t. Dans ce cas, seul un DBN par variable suffit (Boutilier and Goldszmidt, 1996).

Modèle factorisé de la fonction de transition

La figure 3.1 montre la représentation de l'effet de l'action $\mathcal{D}$elC sur l'ensemble des états. Le DBN (figure 3.1(a)) permet de constater facilement que, pour l'action $\mathcal{D}$elC, la variable $\mathcal{HOC}$ ne dépend que des variables $\mathcal{O}$, $\mathcal{HRC}$ et $\mathcal{HOC}$ au pas de temps précédent et est indépendante des autres variables du problème. On définit $\underline{\text{Parents}}_\tau(X'_i)$ l'ensemble des parents de la variable X'_i dans le DBN τ. Cet ensemble peut être partitionné en deux sous-ensembles $\underline{\text{Parents}}^t_\tau(X'_i)$ et $\underline{\text{Parents}}^{t+1}_\tau(X'_i)$ représentant respectivement l'ensemble des parents au temps t et l'ensemble des parents au temps $t + 1$. Pour la suite de ce manuscrit, nous supposons l'absence d'arc synchrone, donc $\underline{\text{Parents}}^{t+1}_\tau(X'_i) = \emptyset$ et $\underline{\text{Parents}}_\tau(X'_i) = \underline{\text{Parents}}^t_\tau(X'_i)$. Dans l'exemple de la figure 3.1, nous avons $\underline{\text{Parents}}_{\mathcal{D}\text{elC}}(\mathcal{HOC}') = \{\mathcal{O}, \mathcal{HRC}, \mathcal{HOC}\}$.

Afin de quantifier l'effet d'une action sur l'espace d'états, on spécifie la probabilité $P_\tau(X'_i|x)$ pour chaque instanciation possible $x \in \underline{\text{Dom}}(\underline{\text{Parents}}_\tau(X'_i))$. Chaque réseau d'action est donc quantifié par un ensemble de *Distributions de Probabilités Conditionnelles*, ou *Conditional Probability Distributions*. Une telle distribution pour une variable X'_i est notée $P_\tau(X'_i|\underline{\text{Parents}}_\tau(X'_i))$. Une pro-



$\mathcal{O}$	$\mathcal{HRC}$	$\mathcal{HOC}$	$\mathcal{HOC}'$
1	1	1	1.0
0	1	1	1.0
1	0	1	1.0
0	0	1	1.0
1	1	0	0.8
0	1	0	0.0
1	0	0	0.0
0	0	0	0.0

FIG. 3.1 – Représentation (partielle) de la fonction de transition T pour le problème *Coffee Robot*. La figure (a) représente les dépendances entre les variables pour l'action $\mathcal{D}$elC sous la forme d'un DBN. La figure (b) définit la distribution de probabilités conditionnelle $P_{\mathcal{D}\text{elC}}(\mathcal{HOC}'|\mathcal{O}, \mathcal{HRC}, \mathcal{HOC})$ sous forme tabulaire.

babilité $P_\tau(s'|s)$ de la fonction de transition peut alors être définie de façon compacte :

$$P_\tau(s'|s) = \prod_i P_\tau(x_i^{s'}|x^s) \tag{3.1}$$

avec $x_i^{s'}$ l'instanciation de la variable X_i' dans l'état s' et x^s l'instanciation des variables appartenant à $\underline{\text{Parents}}_\tau(X_i')$.

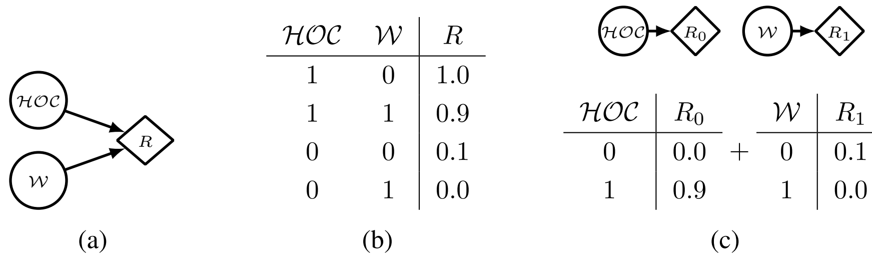
La figure 3.1(b) représente sous forme tabulaire et pour l'action $\mathcal{D}$elC la distribution de probabilités conditionnelle $P_{\mathcal{D}\text{elC}}(\mathcal{HOC}'|\mathcal{O}, \mathcal{HRC}, \mathcal{HOC})$ dans le problème *Coffee Robot*. Les colonnes $\mathcal{O}$, $\mathcal{HRC}$ et $\mathcal{HOC}$ représentent la valeur de ces variables à l'instant t, la colonne $\mathcal{HOC}'$ représente la probabilité pour la variable $\mathcal{HOC}$ d'avoir la valeur Vrai au temps $t+1$.

La décomposition multiplicative et l'exploitation des indépendances relatives aux fonctions dans la description du modèle des transitions et dans le calcul des probabilités qui en découlent sont les principales contributions des FMDPs par rapport aux MDPs. Ces deux propriétés sont exploitées par l'ensemble des algorithmes décrits dans le cadre des FMDPs.

3.1.2 Représentation de la fonction de récompense

Pour spécifier complètement un MDP, il est nécessaire de décrire la fonction R de récompense. Une représentation similaire à la description de la fonction de transition peut être utilisée. En effet, la fonction de récompense d'un MDP peut, d'une part, être décomposée de façon additive et, d'autre part, ne dépend pas nécessairement de toutes les variables d'état du problème.

Par exemple, dans le problème *Coffee Robot*, la fonction de récompense, représentée par un losange dans la figure 3.2, ne dépend que des variables $\mathcal{HOC}$ et $\mathcal{W}$ et elle est indépendante des actions réalisées par le robot ou bien des autres variables du problème.



$\mathcal{HOC}$	$\mathcal{W}$	R
1	0	1.0
1	1	0.9
0	0	0.1
0	1	0.0

$\mathcal{HOC}$	R_0
0	0.0
1	0.9

$+$

$\mathcal{W}$	R_1
0	0.1
1	0.0

FIG. 3.2 – Représentation de la fonction de récompense $R(s)$

La table de la figure 3.2(b) spécifie que le meilleur état pour le robot est lorsque sa propriétaire a un café et que le robot est sec tandis que le pire cas est lorsque sa propriétaire n'a pas de café et que le robot est mouillé. On observe la préférence donnée au cas où l'utilisateur possède un café et le robot est mouillé par rapport au cas où l'utilisateur n'a pas de café et le robot est sec.

Boutilier et al. (2000) définissent la fonction de récompense du problème *Coffee Robot* en faisant la somme des deux critères du problème, "la propriétaire a un café" et "le robot est mouillé". Pourtant, ces deux critères sont indépendants. Afin de profiter de la décomposition additive de cette fonction de récompense, Guestrin et al. (2003b) proposent de formaliser la fonction de récompense d'un FMDP en une somme de plusieurs *fonctions de récompense localisées* (*localized reward functions*).

Pour le problème *Coffee Robot*, on peut définir la fonction de récompense comme la somme de deux fonctions de récompenses localisées : la première associée à la variable $\mathcal{HOC}$ et la deuxième associée à la variable $\mathcal{W}$ et représentant respectivement les deux critères "la propriétaire a un café" et "le robot est mouillé".

Guestrin et al. (2003b) formalisent cette notion en définissant tout d'abord la notion de *scope* d'une fonction f localisée (notée $\underline{\text{Scope}}(f)$). Le scope d'une fonction f localisée est défini tel que :

Définition 2 (scope) *Soit une fonction* $f : \{X_1, \ldots, X_n\} \mapsto \mathbb{R}$, *on a* $\underline{\text{Scope}}(f) = C$ *définissant le* scope *de* f *si* $f : \underline{\text{Dom}}(C) \mapsto \mathbb{R}$ *avec* $C \subseteq \{X_1, \ldots, X_n\}$.

Soit une fonction f tel que $\underline{\text{Scope}}(f) = C$ avec $C \subseteq X$, on utilise la notation $f(x)$ comme raccourci pour noter $f(x[C])$ avec $x[C]$ représentant l'instanciation des variables appartenant à C dans l'instanciation x. Le scope d'une fonction f permet ainsi de mettre en évidence les indépendances relatives à f[1].

Il est maintenant possible de définir le concept de fonction de récompense localisée. Soit un ensemble de fonctions localisées $R_1^a, \ldots, R_r^a$ avec le scope de chaque fonction R_i^a restreint à un

[1] La notion de scope d'une fonction est similaire à la notion de parent utilisée pour la définition des distributions de probabilités conditionnelles de la fonction de transition.

groupe $C_i^a \subseteq \{X_1, \ldots, X_n\}$. La récompense associée au fait d'exécuter l'action a dans un état s est alors définie telle que :

$$R^a(s) = \sum_{i=1}^{r} R_i^a(s[C_i^a]) \tag{3.2}$$

$$= \sum_{i=1}^{r} R_i^a(s) \tag{3.3}$$

Ainsi, pour reprendre l'exemple de *Coffee Robot*, le problème est défini par deux fonctions de récompenses R_1 et R_2 définies dans la figure 3.2(c) et correspondant respectivement aux deux critères "la propriétaire a un café" et "le robot est mouillé". On a $\underline{\text{Scope}}(R_1) = \{\mathcal{HOC}\}$ et $\underline{\text{Scope}}(R_2) = \{\mathcal{W}\}$. On utilise $R_1(s)$ comme raccourci pour représenter $R_1(s[\mathcal{HOC}])$, avec $s[\mathcal{HOC}]$ représentant l'instanciation de $\mathcal{HOC}$ dans s.

Bien que l'ensemble des algorithmes décrits dans le cadre des FMDPs exploitent les indépendances relatives aux fonctions de récompense du problème, tous n'exploitent pas la décomposition additive de la fonction de récompense. De plus, tous les problèmes ne présentent pas une telle décomposition.

Nous venons de décrire le formalisme des FMDPs permettant de mettre en évidence la structure d'un problème dans sa description. Ainsi, à partir d'un FMDP complètement spécifié et modélisant un problème à résoudre, plusieurs méthodes de planification ont été proposées et permettent de calculer une fonction de valeur optimale ou optimale approchée ainsi qu'une politique optimale ou optimale approchée. Les sections suivantes décrivent plusieurs de ces méthodes, notamment les algorithmes SPI et SVI dans la section 3.2, l'algorithme SPUDD dans la section 3.3 et une approche basée sur la programmation linéaire dans la section 3.4.

3.2 *Structured Value Iteration* et *Stucture Policy Iteration*

Les deux algorithmes, Structured Value Iteration (SVI) et Structured Policy Iteration (SPI) (Boutilier et al., 2000) ont été les premiers algorithmes adaptant les algorithmes de programmation dynamique au formalisme des FMDPs, démontrant ainsi les avantages (et les inconvénients) de ce formalisme. En plus des indépendances spécifiques aux fonctions utilisées dans la décomposition des fonctions de transition et de récompense, les algorithmes SPI et SVI utilisent une représentation structurée afin d'exploiter les indépendances relatives au contexte dans la représentation des différentes fonctions du problème.

Par exemple, nous pouvons constater que, dans l'exemple *Coffee Robot*, lorsque la propriétaire a déjà un café, alors il n'est pas nécessaire d'évaluer si le robot a un café ou s'il est au bureau

pour déterminer si la propriétaire aura un café au prochain pas de temps. Ainsi, la distribution de probabilités de la variable aléatoire $\mathcal{HOC}'$ dans le contexte $\mathcal{HOC} = 1$, c'est-à-dire “la propriétaire a un café”, est indépendante des variables $\mathcal{HRC}$ et $\mathcal{O}$ au pas de temps précédent, c'est-à-dire “le robot a-t'il un café ?” et “le robot est-t'il au bureau ?”, bien que ces deux variables soient nécessaires pour définir complètement la distribution de probabilités de $\mathcal{HOC}'$.

Pour exploiter ce type de régularités, Boutilier et al. (2000) suggèrent plusieurs notations pour représenter les fonctions du FMDP à résoudre, telles que les règles (Poole, 1997), les listes de décision (Rivest, 1987) ou les graphes de décision booléens (Bryant, 1986). SPI et SVI sont présentés en utilisant les arbres de décision (Quinlan, 1993), principalement à cause de leur simplicité. Nous verrons qu'ils présentent d'autres avantages (section 4). Nous verrons aussi deux autres méthodes de résolution dans les FMDPs et utilisant d'autres représentations (section 3.3 et 3.4).

3.2.1 Représentations

Les arbres de décision représentent une fonction en partitionnant son espace d'entrée. Un arbre de décision est composé de :

nœuds intérieurs (ou nœuds de décision) : ils représentent un test sur une variable de l'espace d'entrée. Ils sont parents d'autres nœuds dans l'arbre et définissent la structure de la partition de l'espace d'entrée.

branches : elles connectent un nœud intérieur parent à un nœud enfant. Elles représentent la valeur de la variable testée au nœud intérieur parent vers le nœud enfant.

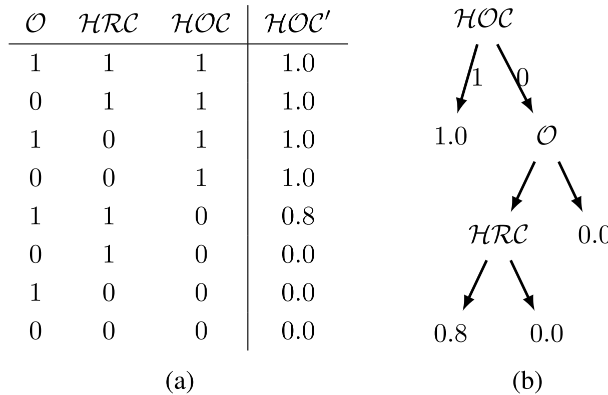
feuilles : elles représentent les nœuds terminaux de l'arbre et sont associées à la valeur de la fonction dans la partition définie par l'ensemble des tests des nœuds intérieurs qui sont les parents de la feuille.

Dans le cadre de SPI et SVI, les arbres de décision sont utilisés pour représenter l'ensemble des fonctions du FMDP à résoudre. Une fonction F représentée avec un arbre de décision est notée $\mathsf{Tree}[F]$. Concernant la notation dans les figures, les arbres sont représentés en utilisant la convention suivante : pour un nœud de décision testant une variable X booléenne, les branches de gauche et de droite sont associées respectivement à $X = 1$ et $X = 0$. Lorsque la variable n'est pas booléenne, alors la valeur de X est indiquée sur chaque branche.

Représentation de la fonction de transition

Dans le problème *Coffee Robot*, la description sous forme tabulaire de la distribution de probabilités $P_{\mathcal{D}\text{elC}}(\mathcal{HOC}'|\mathcal{O}, \mathcal{HRC}, \mathcal{HOC})$, rappelée figure 3.3(a), fait apparaître plusieurs régularités pouvant être agrégées. Par exemple, on peut remarquer que, comme décrit ci-dessus, dans le contexte $\mathcal{HOC} = 1$, la probabilité que $\mathcal{HOC}'$ soit vrai est égale à 1, quelle que soit la valeur des

deux autres variables $\mathcal{O}$ et $\mathcal{HRC}$ appartenant à l'ensemble $\underline{\text{Parents}}_{\mathcal{D}\text{elC}}(\mathcal{HOC}')$: lorsque la propriétaire a un café, alors il est certain qu'elle aura un café au prochain pas de temps. Les arbres de décision permettent de représenter de façon compacte ce type de régularités.



$\mathcal{O}$	$\mathcal{HRC}$	$\mathcal{HOC}$	$\mathcal{HOC}'$
1	1	1	1.0
0	1	1	1.0
1	0	1	1.0
0	0	1	1.0
1	1	0	0.8
0	1	0	0.0
1	0	0	0.0
0	0	0	0.0

(a)

(b)

FIG. 3.3 – Représentation sous la forme tabulaire de la distribution de probabilités conditionnelle $P_{\mathcal{D}\text{elC}}(\mathcal{HOC}'|\mathcal{O},\mathcal{HRC},\mathcal{HOC})$ (figure a) et sous la forme d'un arbre de décision (figure b). La feuille notée 0.8 signifie que la probabilité pour la variable $\mathcal{HOC}'$ d'être vraie est : $P_{\mathcal{D}\text{elC}}(\mathcal{HOC}'|\mathcal{O} = 1,\mathcal{HRC} = 1,\mathcal{HOC} = 0) = 0.8$. Ainsi, certaines régularités sont agrégées, comme par exemple les probabilités $P_{\mathcal{D}\text{elC}}(\mathcal{HOC}'|\mathcal{HOC} = 1) = 1.0$.

Un arbre de décision $\text{Tree}\,[P_\tau(X'|\underline{\text{Parents}}_\tau(X'))]$ représentant la distribution de probabilités conditionnelle $P_\tau(X'|\underline{\text{Parents}}_\tau(X'))$ est composée de :

nœuds intérieurs : représentent un test sur une variable $X_j \in \underline{\text{Parents}}_\tau(X')$;

branches : représentent une valeur $x_j \in \underline{\text{Dom}}(X_j)$ de la variable X_j testée au nœud parent et définissant le sous espace représenté par le nœud enfant connecté à la branche.

les feuilles : représentent la distribution de probabilités $P_f(X'|x[X_j])$, avec $x[X_j]$ l'ensemble des instanciations des variables $X_j \in \underline{\text{Parents}}_\tau(X')$ testées dans les nœuds parents de la feuille f dans l'arbre.

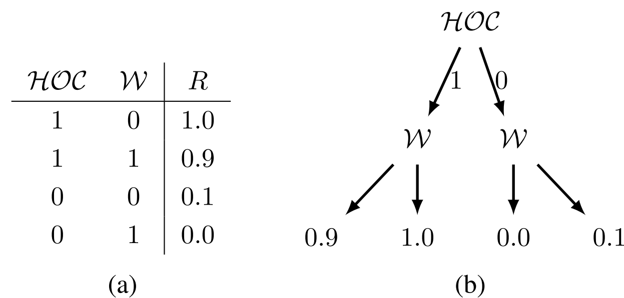
L'interprétation d'un tel arbre est directe : la distribution de probabilités d'une variable X' pour une instanciation x est donnée par la feuille de l'unique branche dont l'instanciation (partielle) des valeurs des tests des nœuds de décision est consistante avec x.

La figure 3.3(b) représente $\text{Tree}\,[P_{\mathcal{D}\text{elC}}(\mathcal{HOC}'|\mathcal{O},\mathcal{HRC},\mathcal{HOC})]$: la distribution de probabilités conditionnelle $P_{\mathcal{D}\text{elC}}(\mathcal{HOC}'|\mathcal{O},\mathcal{HRC},\mathcal{HOC})$ sous la forme d'un arbre de décision. Les valeurs aux feuilles indiquent la probabilité que la variable $\mathcal{HOC}'$ soit vraie. On peut alors remarquer qu'une représentation en arbre de décision, pour la définition de $P_{\mathcal{D}\text{elC}}(\mathcal{HOC}')$, est plus compacte qu'une représentation tabulaire puisqu'elle exploite les indépendances relatives aux contextes : 4 feuilles sont nécessaires à la représentation de la fonction alors que 8 entrées sont nécessaires pour décrire

la même fonction sous forme tabulaire. Nous verrons que cette factorisation est utilisée par les algorithmes de planification SPI et SVI.

Représentation de la fonction de récompense

La représentation d'une fonction de récompense avec des arbres de décision est très similaire à la représentation d'une distribution de probabilités. En effet, la signification des nœuds intérieurs et des branches est la même. Seule change l'étiquette attachée aux feuilles de l'arbre puisqu'elle représente des nombres réels plutôt que des distributions de probabilités.



$\mathcal{HOC}$	$\mathcal{W}$	R
1	0	1.0
1	1	0.9
0	0	0.1
0	1	0.0

FIG. 3.4 – Définition de la fonction de récompense $R(s)$ avec une représentation tabulaire (figure a) et un arbre de décision (figure b). La feuille notée 0.9 signifie $R(\mathcal{HOC} = 1, \mathcal{W} = 1) = 0.9$.

La figure 3.4 représente la fonction de récompense pour la problème *Coffee Robot* et compare la représentation tabulaire $R(s)$ avec l'arbre de décision $\text{Tree}\,[R(s)]$. On peut constater qu'aucune indépendance relative aux contextes n'est utilisable puisque le nombre de feuilles dans l'arbre est égale au nombre de lignes nécessaires à la définition de la fonction avec une représentation tabulaire.

Les algorithmes SPI et SVI ne permettent pas d'exploiter la décomposition additive d'une fonction de récompense telle qu'elle a été définie dans la section 3.1.2.

Représentation d'une politique

Une politique $\pi(s)$ peut aussi être représentée sous la forme d'un arbre de décision $\text{Tree}\,[\pi(s)]$. La figure 3.5 représente une politique stationnaire déterministe $\text{Tree}\,[\pi(s)]$ dans le problème *Coffee Robot*.

L'espace d'état du problème *Coffee Robot* est composé de 6 variables binaires. Une description tabulaire de π aurait donc nécessité $2^6 = 64$ entrées. L'arbre $\text{Tree}\,[\pi]$ ne contient que 8 feuilles (15 nœuds au total). Sur le problème *Coffee Robot*, l'utilisation d'arbres de décision pour représenter une politique permet donc d'exploiter des indépendances relatives aux contextes telles que, lorsque la propriétaire n'a pas de café, que le robot est au bureau et qu'il a un café, il n'est pas nécessaire de

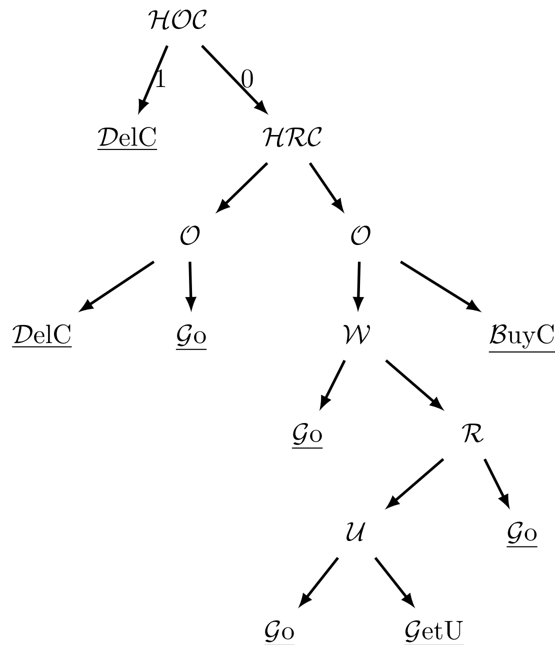


FIG. 3.5 – Représentation d'une politique $\pi(s)$ sous la forme d'un arbre de décision Tree $[\pi(s)]$. La feuille notée $\underline{\mathcal{B}\text{uyC}}$ signifie $\pi(\mathcal{HOC} = 0, \mathcal{HRC} = 0, \mathcal{O} = 0) = \underline{\mathcal{B}\text{uyC}}$.

connaître la valeur des variables telles que “est-ce-qu'il pleut ?” pour déterminer la meilleure action à réaliser.

Lors de l'exécution d'une politique dans un environnement, une telle représentation se révèle avantageuse lorsque déterminer la valeur d'une variable a un coût (par exemple en terme de temps de calcul). En effet, elle permet de n'avoir à déterminer que la valeur des variables strictement nécessaires à l'exécution de la politique de façon spécifique à l'état courant de l'agent. Une telle propriété permet ainsi d'économiser l'évaluation des variables inutiles.

Enfin, l'utilisation d'un arbre de décision pour la description d'une politique permet d'effectuer un nombre minimum de tests pour déterminer l'action à réaliser pour l'agent. Dans le pire cas, pour un problème décrit avec N variables, seuls N tests sont nécessaires pour déterminer l'action retournée par la politique. Cependant, l'espace mémoire requis pour une telle représentation reste, dans le pire cas, exponentielle en fonction du nombre de variables composant l'espace d'états du problème.

Représentation d'une fonction de valeur

Naturellement, la fonction de valeur V_π d'une politique π peut aussi se représenter sous la forme d'un arbre de décision Tree $[V_\pi]$. La sémantique d'un tel arbre est quasiment identique à celle d'un arbre de décision représentant une fonction de récompense : un nœud de décision représente une variable, une branche représente la valeur de la variable testée au nœud de décision parent et les

feuilles représentent la valeur de la fonction de valeur pour la partition délimitée par les tests de ses parents. La figure 3.6 représente la fonction de valeur de la politique Tree $[\pi]$ représentée figure 3.5.

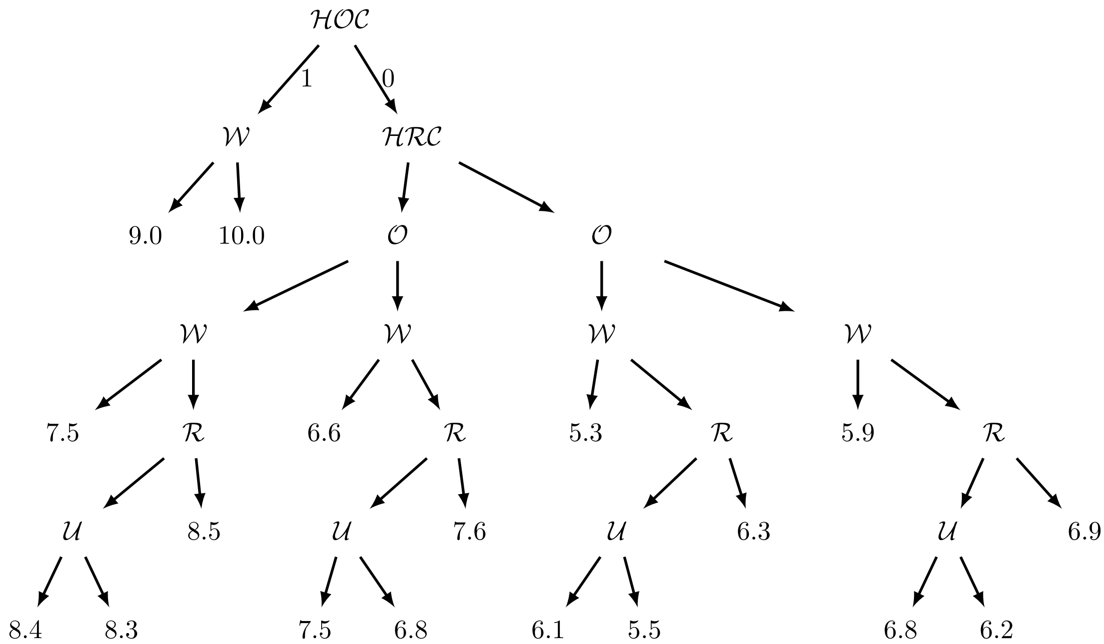


FIG. 3.6 – Représentation de la fonction de valeur $V_\pi(s)$ de la politique π sous la forme d'un arbre de décision Tree $[V_\pi(s)]$ pour le problème *Coffee Robot*. La feuille notée 10.0 signifie $V_\pi(\mathcal{HOC} = 1, \mathcal{W} = 0) = 10.0$.

L'arbre Tree $[V_\pi]$ ne contient que 18 feuilles (35 nœuds au total) alors qu'une représentation tabulaire aurait nécessitée 64 entrées. Sur le problème *Coffee Robot*, une représentation sous la forme d'arbre de décision permet donc d'exploiter les indépendances relatives aux contextes. Par exemple, la valeur $V_\pi(\mathcal{HOC} = 1, \mathcal{W} = 0)$ de la politique π, lorsque la propriétaire a un café et que le robot est sec, ne dépend pas des autres variables du problème. Une telle propriété peut être considérée comme l'agrégation de plusieurs états. Ainsi, lors du calcul itératif d'une fonction de valeur, il n'est nécessaire de calculer qu'une seule fois la mise à jour de la valeur d'une feuille pour mettre à jour la valeur de tous les états représentés par cette feuille.

Cependant, il est possible de constater sur la fonction de valeur Tree $[V_\pi]$ qu'une telle représentation ne permet pas d'exploiter certaines régularités présentes dans la définition de V_π. En effet, on peut remarquer, par exemple, que la structure des sous-arbres composés des variables $\mathcal{R}$, $\mathcal{W}$, $\mathcal{U}$ et $\mathcal{O}$ est identique. Nous verrons qu'une approximation additive de la fonction de valeur (que nous présenterons section 3.4.1, page 66) permet d'exploiter une telle symétrie, contrairement à une représentation telle que les arbres de décision.

Enfin, dans le pire cas, c'est-à-dire lorsque la fonction de valeur de la politique évaluée est différente pour tous les états possibles, la taille de la représentation augmente exponentiellement

avec le nombre de variables composant l'espace d'états du problème.

3.2.2 Manipulations

Le principe de base des algorithmes SPI et SVI est d'adapter les algorithmes *Policy Iteration* et *Value Iteration* aux arbres de décision. Ainsi, plutôt que d'avoir à calculer une mise à jour de la valeur de chaque état possible lors d'une itération, comme c'est le cas pour *Policy Iteration* et *Value Iteration*, SVI et SPI calculent cette mise à jour pour chaque feuille d'un arbre de décision, permettant ainsi de réduire le coût des calculs lorsque plusieurs états sont agrégés et représentés par une feuille.

Pour effectuer cette opération, SPI et SVI utilisent trois opérateurs différents sur les arbres (Boutilier et al., 2000) :

Simplification (noté $\underline{\text{Simplification}}(T)$) : opérateur supprimant les sous-arbres identiques et les nœuds de décision redondants dans l'arbre T (opérateur illustré figure 3.7).

Ajout d'un arbre (noté $\underline{\text{Append}}(T_1, T_2)$) : opérateur ajoutant l'arbre T_2 à chaque feuille l_i de l'arbre T_1 en utilisant l'opérateur $\underline{\text{Append}}(T_1, l_i, T_2)$. Cet opérateur ajoute la structure de l'arbre T_2 à la feuille l_i en utilisant une opération de combinaison $F(l_i, T_2)$ pouvant être l'une de ces fonctions : l'union, la somme ou le maximum de la feuille l_i et des feuilles l_2 appartenant à T_2 (opérateur illustré figure 3.8).

Union d'arbres (noté $\underline{\text{Merge}}(\{T_1, \ldots, T_n\})$) : opérateur construisant un seul arbre contenant l'ensemble des partitions apparaissant dans tous les arbres $T_1, \ldots, T_n$ de l'ensemble à fusionner. Cette opération peut être définie de façon récursive et en utilisant l'opération d'ajout d'un arbre :

$$\begin{aligned} \underline{\text{Merge}}(\{T_1\}) &: T_1 \\ \underline{\text{Merge}}(\{T_1, \ldots, T_i\}) &: \underline{\text{Append}}(T_i, \underline{\text{Merge}}(\{T_1, \ldots, T_{i-1}\})) \end{aligned}$$

Les figures 3.7 et 3.8 montrent respectivement un exemple de simplification d'un arbre contenant deux sous-arbres identiques et d'ajout d'un arbre à un autre. L'ajout d'arbre permet de calculer le résultat d'un opérateur mathématique sur plusieurs arbres. La simplification permet de diminuer la taille de l'arbre obtenu (dans ce cas en supprimant les sous-arbres identiques), une fois le résultat d'un opérateur calculé. Nous montrerons, section 4.2.4 (page 99), que la simplification peut éventuellement être complétée (ou remplacée) par une réorganisation de l'arbre.

Ces opérateurs de base sur les arbres permettent de redéfinir les opérations réalisées sur des représentations tabulaires par la programmation dynamique et ainsi, lorsque la fonction représentée le permet, d'accélérer les calculs nécessaires en exploitant les indépendances relatives aux contextes.

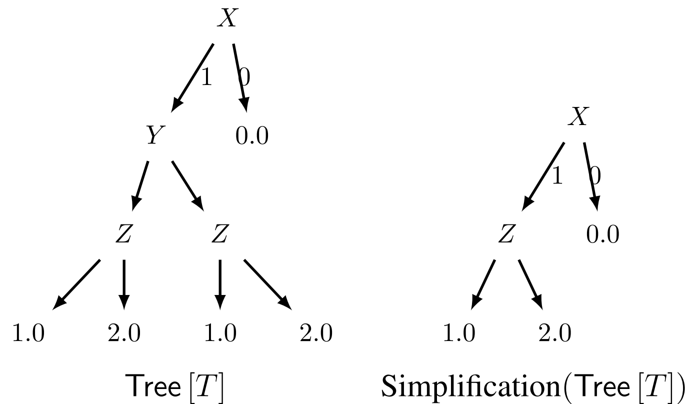


FIG. 3.7 – Illustration de l'opération de simplification sur un arbre Tree $[T]$ contenant deux sous-arbres identiques.

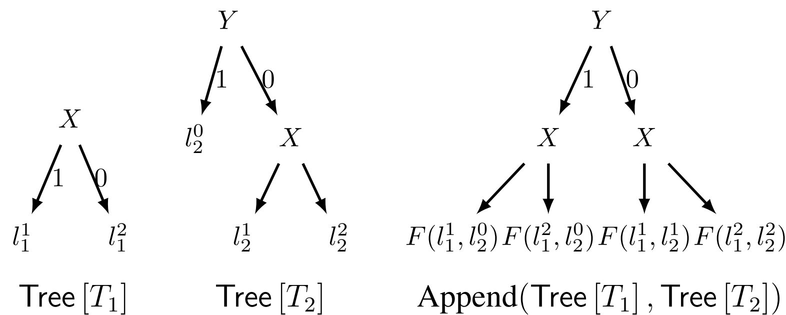


FIG. 3.8 – Illustration de l'opération d'ajout de l'arbre Tree $[T_2]$ à l'arbre Tree $[T_1]$ et en utilisant l'opération de combinaison F.

3.2.3 Calcul d'une fonction de valeur d'action sur une itération

L'une des opérations de base de la programmation dynamique est le calcul d'une fonction de valeur d'action à partir d'une fonction de valeur, suivant l'équation 2.4 (page 26) :

$$Q_a^V(s) = R(s,a) + \gamma \sum_{s'} P(s'|s,a)V(s')$$

SPI et SVI réalisent cette opération à l'aide de l'opérateur Regress défini figure 3.9. L'algorithme reprend les différentes étapes du calcul de l'équation.

La principale difficulté dans le calcul de l'équation 2.4 est le calcul de l'espérance de la fonction de valeur $\sum_{s'} P(s'|s,a)V(s')$ tout en exploitant la structure du problème. Pour cela, une représentation factorisée Tree $\left[P_a^V\right]$ de la fonction de transition dont la structure est dépendante de la structure de l'arbre de décision Tree $[V]$ représentant la fonction de valeur est tout d'abord calculée via l'opérateur PRegress (étape 1). Une représentation factorisée Tree $\left[P_a^V V\right]$ de l'espérance de la fonction de valeur $\sum_{s'} P(s'|s,a)V(s')$ est ensuite calculée à partir de Tree $\left[P_a^V\right]$ et de la fonction de valeur Tree $[V]$ (étape 2). Le produit $\gamma \sum_{s'} P(s'|s,a)V(s')$ est calculé lors de l'étape 3 en mul-

---

Entrée(s) : $\text{Tree}[V], a$ **Sortie(s) :** $\text{Tree}[Q_a^V]$

---

1. $\text{Tree}[P_a^V] \leftarrow \underline{\text{PRegress}}(\text{Tree}[V], a)$
2. Construire $\text{Tree}[P_a^V V]$ de la façon suivante : pour chaque branche b parente de la feuille l_b et appartenant à l'arbre $\text{Tree}[P_a^V]$, faire :
   (a) Soit P^b la distribution de probabilités jointe obtenue à partir du produit de chaque distribution de probabilités de chaque variable présente dans la feuille l_b
   (b) Calculer $v_b = \sum_{b' \in \text{Tree}[V]} P^b(b')V(b')$ avec : b' les branches de l'arbre $\text{Tree}[V]$, $P^b(b')$ la probabilité que l'instanciation des variables associées à la branche b' soit vraie étant donné P^b et $V(b')$ la valeur contenue par la feuille l'_b associée à la branche b' dans l'arbre $\text{Tree}[V]$
   (c) Définir v_b comme étant le contenu la feuille l_b
3. $\text{Tree}[P_a^V V] \leftarrow \gamma \cdot \text{Tree}[P_a^V V]$ (en multipliant chaque feuille par γ)
4. $\text{Tree}[Q_a^V] \leftarrow \underline{\text{Append}}(\text{Tree}[R], \text{Tree}[P_a^V V])$ (en utilisant l'addition comme opérateur de combinaison)
5. Retourner $\text{Tree}[Q_a^V]$

FIG. 3.9 – L'algorithme $\underline{\text{Regress}}(\text{Tree}[V], a)$.

tipliant la valeur de toutes les feuilles de $\text{Tree}[P_a^V V]$ par γ. Enfin, une représentation factorisée de $\text{Tree}[Q_a^V]$ est obtenue en ajoutant la fonction de récompense $\text{Tree}[R]$ lors de l'étape 4.

Concernant l'opérateur PRegress, il calcule une représentation factorisée $\text{Tree}[P_a^V]$ de la fonction de transition étant donnée une fonction de valeur $\text{Tree}[V]$. La représentation en arbre de décision de la fonction de valeur $\text{Tree}[V]$ définit une partition de l'espace d'état. À partir de cette partition, l'opérateur PRegress calcule une nouvelle partition sous la forme d'un arbre de décision $\text{Tree}[P_a^V]$ dont les feuilles contiennent pour chaque région de $\text{Tree}[V]$ la probabilité d'arriver dans cette région après avoir exécuté l'action a. L'opérateur est décrit figure 3.10.

L'algorithme PRegress est un algorithme récursif décomposé principalement en deux phases. La première consiste à construire les arbres $\text{Tree}\left[P_a^{V_{x_i}}\right]$ représentant une définition factorisée de la fonction de transition pour chacune des partitions $\text{Tree}[V_{x_i}]$ définies par l'ensemble des branches $x_i \in \underline{\text{Dom}}(X)$ du nœud racine X de l'arbre (étape 3). À partir d'une représentation factorisée $\text{Tree}[P_a](X'|s)$ du modèle de transition de la variable X pour l'action a, la deuxième phase consiste à faire l'union des arbres $\text{Tree}\left[P_a^{V_{x_i}}\right]$ pour chacune des feuilles où la probabilité $P(X' = x_i)$ est strictement positive (étape 5). Nous invitons le lecteur à consulter l'article de Boutilier et al. (2000) pour de plus amples explications concernant les opérateurs Regress et PRegress.

Entrée(s) : $\text{Tree}[V], a$ **Sortie(s) :** $\text{Tree}\left[P_a^V\right]$

1. Si $\text{Tree}[V]$ contient seulement une feuille alors retourner l'arbre vide $\text{Tree}\left[P_a^V\right]$.
2. $X \leftarrow$ la variable testée à la racine de $\text{Tree}[V]$
   $\text{Tree}[P_a](X'|s) \leftarrow$ la fonction de transition de X pour l'action a
3. $\forall x_i \in \underline{\text{Dom}}(X)$:
   (a) Soit $\text{Tree}[V_{x_i}]$ le sous-arbre de $\text{Tree}[V]$ attaché à la racine par la branche telle que $X = x_i$
   (b) $\text{Tree}\left[P_a^{V_{x_i}}\right] \leftarrow \underline{\text{PRegress}}(\text{Tree}[V_{x_i}], a)$
4. $\text{Tree}\left[P_a^V\right] \leftarrow \text{Tree}[P_a](X'|s)$
5. Pour toute feuille $l \in \text{Tree}\left[P_a^V\right]$, contenant la distribution de probabilités P^l :
   (a) $\text{Tree}[P_l] \leftarrow \underline{\text{Merge}}(\{\text{Tree}\left[P_a^{V_{x_i}}\right] : \forall x_i \text{ tel que } P^l(x_i) > 0\})$ (en utilisant l'union comme opérateur de combinaison)
   (b) Ajouter $\text{Tree}[P_l]$ à la feuille l (en utilisant l'union comme opérateur de combinaison)
6. Retourner $\text{Tree}\left[P_a^V\right]$

FIG. 3.10 – L'algorithme $\underline{\text{PRegress}}(\text{Tree}[V], a)$.

3.2.4 Construction d'une politique gloutonne

Une fois qu'il est possible de calculer une fonction de valeur d'action pour une étape à partir d'une fonction de valeur, il est naturel de redéfinir les autres opérateurs utilisés par la programmation dynamique, tel que l'opérateur Greedy, en utilisant des représentations structurées telles que les arbres de décision. L'opérateur Greedy est redéfini figure 3.11.

Entrée(s) : $\text{Tree}[V]$ **Sortie(s) :** $\text{Tree}[\pi]$

1. $\forall a \in A : \text{Tree}\left[Q_a^V\right] \leftarrow \underline{\text{Regress}}(\text{Tree}[V], a)$
2. $\text{Tree}[\pi] \leftarrow \underline{\text{Merge}}(\{\text{Tree}\left[Q_a^V\right] : \forall a \in A\})$ en utilisant la maximisation comme opérateur de combinaison
3. Retourner $\text{Tree}[\pi]$

FIG. 3.11 – L'opérateur $\underline{\text{Greedy}}(\text{Tree}[V])$ défini pour utiliser des arbres de décision.

Le calcul est décomposé en deux phases simples. À partir de la représentation structurée d'une fonction de valeur V, la première phase consiste à calculer une représentation structurée des fonc-

tions de valeur d'action Tree $\left[Q_a^V\right]$ pour chaque action. La deuxième phase construit une politique gloutonne en choisissant, pour chaque sous-partie de l'espace définie dans l'un des arbres Tree $\left[Q_a^V\right]$, l'action ayant la meilleure valeur d'action.

3.2.5 Les algorithmes SPI et SVI

À partir des opérateurs Regress et Greedy, il est maintenant possible de réécrire les algorithmes de programmation dynamique pour utiliser des arbres de décision comme représentation structurée. Nous commençons par décrire dans la figure 3.12 l'algorithme *Structured Policy Evaluation* (SPE) utilisant des représentations sous la forme d'arbres de décision et correspondant à l'algorithme d'évaluation de la politique (voir figure 2.2, page 29).

Entrée(s) : Tree $[\pi]$ **Sortie(s) :** Tree $[V_\pi]$

---

1. Tree $[V_\pi] \leftarrow$ Tree $[R]$
2. Répéter tant que le critère de fin n'est pas satisfait :
   (a) Pour chaque action a apparaissant dans Tree $[\pi]$: Tree $\left[Q_a^{V_\pi}\right] \leftarrow$ Regress(Tree $[V_\pi]$, a)
   (b) Construire Tree $[V_\pi]$ à partir de Tree $[\pi]$ en remplaçant chaque feuille l_a contenant l'action a par les valeurs contenues dans Tree $\left[Q_a^{V_\pi}\right]$
3. Retourner Tree $[V_\pi]$

FIG. 3.12 – L'algorithme d'évaluation de la politique SPE

L'algorithme SPE se décompose en deux phases : on calcule tout d'abord les différentes fonctions de valeur d'action Tree $\left[Q_a^{V_\pi}\right]$ pour chacune des actions apparaissant dans la politique Tree $[\pi]$, puis on construit la fonction de valeur de la politique Tree $[V_\pi]$ en remplaçant les feuilles Tree $[\pi]$ par les fonctions de valeur d'action correspondantes. Le critère d'arrêt utilisé pour cet algorithme est identique au critère d'arrêt utilisé pour l'algorithme défini avec une représentation tabulaire.

Nous avons décrit les deux algorithmes d'évaluation de la politique, SPE et, d'amélioration de la politique Greedy pour utiliser les arbres de décision. Il est donc maintenant possible de réécrire l'adaptation de l'algorithme *Policy Iteration* pour utiliser les arbres de décision. L'algorithme *Structured Policy Iteration* (SPI) est décrit figure 3.13.

De la même façon que l'algorithme *Policy Iteration*, l'algorithme SPI alterne les phases d'évaluation de la politique (étape 2) avec celle d'amélioration de la politique (étape 4).

Enfin, il est aussi possible de redéfinir l'algorithme *Value Iteration* pour utiliser les arbres de décision en s'appuyant sur les opérateurs précédemment définis. L'algorithme *Structured Value*

Entrée(s) : $\emptyset$ **Sortie(s) :** $\textsf{Tree}\,[V^*]$, $\textsf{Tree}\,[\pi^*]$

1. Initialisation : choisir $\textsf{Tree}\,[\pi]$ de façon arbitraire

Évaluation de la politique :

2. $\textsf{Tree}\,[V_\pi] \leftarrow$ SPE (π)

Amélioration de la politique :

3. $\textsf{Tree}\,[\pi'] \leftarrow \textsf{Tree}\,[\pi]$
4. $\textsf{Tree}\,[\pi] \leftarrow \underline{\text{Greedy}}(\textsf{Tree}\,[V_\pi])$

5. Si $\textsf{Tree}\,[\pi'] \neq \textsf{Tree}\,[\pi]$ alors aller en 2
6. Retourner $V_\pi(s)$ et $\pi(s)$

FIG. 3.13 – L'algorithme *Structured Policy Iteration* (SPI)

Iteration (SVI) est décrit figure 3.14.

De la même façon que l'algorithme *Value Iteration*, l'algorithme SVI calcule tout d'abord une représentation structurée $\textsf{Tree}\,[V]$ de la fonction de valeur optimale (étape 2) et en déduit ensuite une représentation structurée $\textsf{Tree}\,[\pi]$ de la politique optimale (étape 3).

3.3 L'algorithme *Stochastic Planning Using Decision Diagrams*

Dans certains problèmes, la fonction de valeur possède des symétries qui ne sont pas exploitées par les arbres de décision, notamment lorsque la fonction est strictement identique dans plusieurs contextes disjoints. L'algorithme présenté par Hoey et al. (1999) nommé *Stochastic Planning Using Decision Diagrams* (SPUDD) propose d'utiliser des diagrammes de décision algébriques, *Algebraic Decision Diagrams*, (ADD) décrits par Bahar et al. (1993) pour représenter les fonctions d'un FMDP. De façon semblable à SPI, SPUDD exploite les indépendances relatives à la fois aux fonctions et aux contextes.

L'utilisation d'ADDs plutôt que les arbres de décision propose deux avantages supplémentaires. Le premier permet de mieux factoriser certaines fonctions en exploitant le fait que certaines sous-parties d'une partition de l'espace sont semblables les unes aux autres, alors que les contextes les caractérisants sont disjoints. De plus, les variables utilisées dans un ADD sont ordonnées. Bien que trouver un ordre optimal des variables à tester pour représenter une fonction de façon la plus compacte possible est un problème difficile, Hoey et al. (2000) montrent que plusieurs heuris-

---

Entrée(s) : $\emptyset$ **Sortie(s) :** $\mathsf{Tree}\left[V^*\right], \mathsf{Tree}\left[\pi^*\right]$

---

1. Initialisation : choisir $\mathsf{Tree}\left[V\right]$ de façon arbitraire
2. Répéter tant que le critère de fin n'est pas satisfait :
   (a) pour chaque action $a \in A$: $\mathsf{Tree}\left[Q_a^V\right] \leftarrow \underline{\text{Regress}}(\mathsf{Tree}\left[V\right], a)$
   (b) $\mathsf{Tree}\left[V\right] \leftarrow \underline{\text{Merge}}(\{\mathsf{Tree}\left[Q_a^V\right] : \forall a \in A\})$ en utilisant la maximisation comme opérateur de combinaison
3. $\mathsf{Tree}\left[\pi^*\right] \leftarrow \underline{\text{Greedy}}(\mathsf{Tree}\left[V\right])$
4. Retourner $\mathsf{Tree}\left[\pi^*\right]$ et $\mathsf{Tree}\left[V\right]$

FIG. 3.14 – L'algorithme *Structured Value Iteration* (SVI)

tiques peuvent être utilisées pour trouver un ordre permettant de représenter les fonctions de façon suffisamment compacte pour accélérer nettement les calculs. Le deuxième avantage est que cet ordonnancement est utilisé pour accélérer les calculs réalisés sur les fonctions représentées. Ces deux avantages permettent d'améliorer les algorithmes de programmation dynamique aussi bien pour l'espace mémoire requis pour représenter les fonctions du FMDP que dans la complexité des différents opérateurs utilisés pour manipuler ces fonctions.

3.3.1 Représentations

Les ADDs sont une généralisation des diagrammes de décision binaires (BDDs) ou *Binary Decision Diagrams* (Bryant, 1986). Les BDDs sont une représentation compacte représentant des fonctions $\mathbb{B}^n \rightarrow \mathbb{B}$ de n variables binaires vers une valeur binaire. Les ADDs généralisent les BDDs pour représenter des fonctions réelles $\mathbb{B}^n \rightarrow \mathbb{R}$ de n variables binaires vers une valeur réelle. Un ADD est composé de :

nœuds intérieurs (ou nœuds de décision) : ils représentent un test sur une variable binaire de l'espace d'entrée. Ils sont le parent de deux branches correspondant respectivement au fait que la variable testée est égale à Vrai ou Fausse.

branches : elles connectent un nœud intérieur parent à un nœud enfant.

feuilles : elles représentent les nœuds terminaux du diagramme et sont associées à la valeur de la fonction dans l'un des sous-espaces définis par l'ensemble des tests des nœuds intérieurs parents de la feuille.

Contrairement à un arbre de décision, les nœuds intérieurs et les feuilles d'un ADD peuvent avoir plusieurs parents. Une fonction F représentée avec un ADD est notée $\mathsf{ADD}\left[F\right]$. La convention

suivante est utilisée pour représenter un ADD dans une figure : pour un nœud de décision testant une variable X, les branches dessinées en trait plein et pointillé sont associées respectivement à $X = 1$ et $X = 0$.

Les ADDs possèdent plusieurs propriétés intéressantes. D'une part, pour un ordre de variables donné, chaque fonction distincte n'a qu'une seule représentation. D'autre part, la taille de la représentation de nombreuses fonctions peut être réduite grâce à la réutilisation de sous-graphe identique au sein de la description. Enfin, il existe des méthodes optimisées pour la plupart des opérations de base, notamment la multiplication, l'addition ou bien la maximisation.

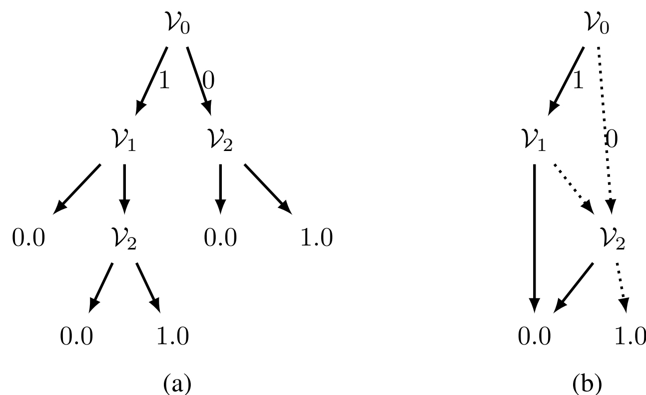


FIG. 3.15 – Comparaison des représentations d'une fonction F sous la forme d'une arbre de décision Tree $[F]$ et d'un diagramme de décision algébrique ADD $[F]$.

La figure 3.15 montre l'exemple d'une même fonction F représentée par un arbre de décision et par un ADD. Elle illustre le fait que les arbres de décision, contrairement aux ADDs, ne sont pas adaptés pour la représentation de certaines fonctions, notamment les fonctions disjonctives. Ainsi, alors que la représentation Tree $[F]$ contient 5 feuilles différentes (et 4 nœuds intérieurs), la représentation ADD $[F]$ n'en contient que 2 (plus 3 nœuds intérieurs). La mise à jour de cette fonction dans le cas de SPI nécessitera donc 5 calculs de mise à jour différents alors que SPUDD ne réalisera que 2 calculs.

Cependant, l'utilisation des ADDs impose principalement deux contraintes sur le FMDP à résoudre. Premièrement, il est nécessaire que les variables du FMDP soient toutes binaires, les ADDs ne représentant que des fonctions $\mathbb{B}^n \rightarrow \mathbb{R}$. Pour les problèmes contenant des variables à plus de deux valeurs, il est toujours possible de décomposer ces variables avec de nouvelles variables (binaires). Deuxièmement, les algorithmes basés sur les ADDs supposent que, au sein de la structure de données, les tests sur les variables sont ordonnées. Lorsque ces deux contraintes sont satisfaites, il est possible de représenter l'ensemble des fonctions du FMDP à résoudre en utilisant des ADDs.

Représentation de la fonction de transition

De façon similaire à SPI, SPUDD utilise une représentation basée sur les DBNs pour représenter la fonction de transition du problème à résoudre. La différence se situe au niveau de la représentation des distributions de probabilités conditionnelles quantifiant le DBN et associées à chaque nœud X'. En effet, contrairement à SPI qui utilise des arbres de décision pour représenter les distributions de probabilités conditionnelles, SPUDD utilise des ADDs.

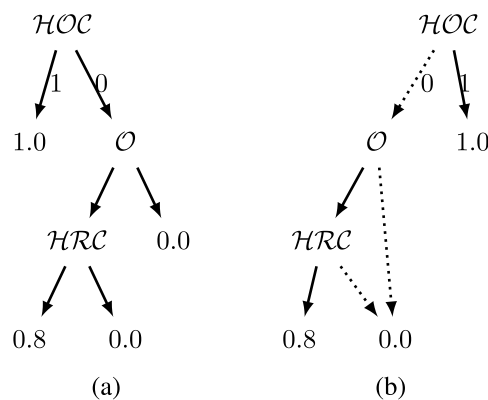


FIG. 3.16 – Représentation de la distribution de probabilités conditionnelle $P_{\mathcal{D}\text{elC}}(\mathcal{HOC}')$ sous la forme d'un arbre de décision (figure a) et d'un ADD (figure b). Le nœud terminal dans l'ADD notée 0.0 signifie que la probabilité pour la variable $\mathcal{HOC}'$ d'être vraie est : $P_{\mathcal{D}\text{elC}}(\mathcal{HOC}'|\mathcal{O} = 1, \mathcal{HRC} = 0, \mathcal{HOC} = 0) = 0.0$ et $P_{\mathcal{D}\text{elC}}(\mathcal{HOC}'|\mathcal{O} = 0, \mathcal{HOC} = 0) = 0.0$.

La figure 3.16 compare la représentation sous la forme d'un arbre de décision et d'un ADD de la distribution de probabilités conditionnelle $P_{\mathcal{D}\text{elC}}(\mathcal{HOC}')$ utilisée pour quantifier le DBN correspondant à l'action $\mathcal{D}$elC dans le problème *Coffee Robot*.

La représentation de la fonction $P_{\mathcal{D}\text{elC}}(\mathcal{HOC}')$ est un peu réduite : alors que Tree $[P_{\mathcal{D}\text{elC}}(\mathcal{HOC}')]$ nécessite 4 feuilles, ADD $[P_{\mathcal{D}\text{elC}}(\mathcal{HOC}')]$ nécessite 3 feuilles ; le nombre de nœuds intérieurs est le même. L'interprétation de la représentation en ADD est identique à celle utilisant un arbre de décision. Le nombre réel contenu dans une feuille indique la probabilité que la variable soit vraie au prochain pas de temps.

Représentation d'une fonction de récompense

La description de la fonction de récompense est très similaire entre SPI et SPUDD : la valeur contenue dans une feuille indique la récompense obtenue par l'agent lorsqu'il est dans le contexte spécifié par l'ensemble des tests des parents de la feuille.

La figure 3.17 compare les représentations de la fonction de récompense de *Coffee Robot* sous la forme d'un arbre de décision et d'un ADD. Comme SPI, SPUDD utilise les indépendances relatives aux fonctions puisque seules les variables $\mathcal{HOC}$ et $\mathcal{W}$ sont utilisées. De plus, on peut constater

que la représentation en ADD de cette fonction ne change rien concernant la taille de celle-ci. Enfin, comme SPI, SPUDD n'exploite pas la décomposition additive présente dans la définition de la fonction de récompense de *Coffee Robot*.

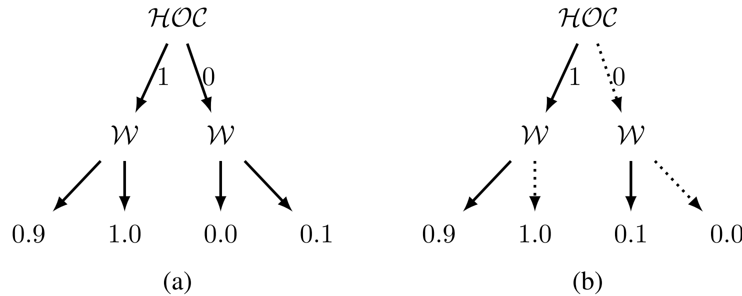


FIG. 3.17 – Représentation de la fonction de récompense R du problème *Coffee Robot* sous la forme d'un arbre de décision (figure a) et d'un ADD (figure b).

Représentation d'une politique

Plutôt que de contenir des nombres réels, il est bien entendu naturel de définir des ADDs permettant de contenir des actions aux feuilles et, d'une façon générale, des ADDs permettant de représenter des fonctions symboliques $F : \mathbb{B}^n \to \mathcal{A}$ de n variables binaires vers un ensemble $\mathcal{A}$ de valeurs symboliques. Pour éviter de multiplier les notations, nous utiliserons aussi la notation ADD pour désigner une telle structure de données. Les ADDs représentant des fonctions symboliques sont utilisés essentiellement pour représenter une politique dans le FMDP à résoudre

L'action contenue à une feuille indique donc l'action spécifiée par la politique représentée dans le contexte défini par l'ensemble des tests des parents de la feuille. Il est important de noter que le nombre de feuilles dans un ADD pour représenter une politique est inférieur ou égal au nombre d'actions réalisables par l'agent : en effet, un ADD ne contient systématiquement qu'une feuille par action utilisée par la politique.

La figure 3.18 compare la représentation de la même politique $\pi(s)$ dans le problème *Coffee Robot* sous la forme d'un arbre de décision et d'un ADD. Le nombre de nœuds intérieurs est identique entre les deux représentations, indiquant qu'aucun sous-arbre n'est identique dans $\mathsf{Tree}\,[\pi]$. Concernant le nombre de feuilles, on observe bien qu'il est limité dans la représentation en ADD au nombre d'actions possibles dans le problème *Coffee Robot* : $\mathsf{ADD}\,[\pi]$ ne contient que 4 feuilles alors que $\mathsf{Tree}\,[\pi]$ en contient 8.

Cet exemple montre que cette représentation est notamment adaptée pour la représentation de fonctions disjonctives : dans le contexte $\mathcal{HOC} = 0 \wedge \mathcal{HRC} = 0 \wedge \mathcal{O} = 1$, c'est-à-dire quand la propriétaire et le robot n'ont pas de café et que le robot est au bureau, l'action définie par π est d'aller au café (action $\mathcal{G}$O), sauf pour le cas particulier $\mathcal{R} = 0$, $\mathcal{W} = 0$ et $\mathcal{U} = 0$, c'est-à-

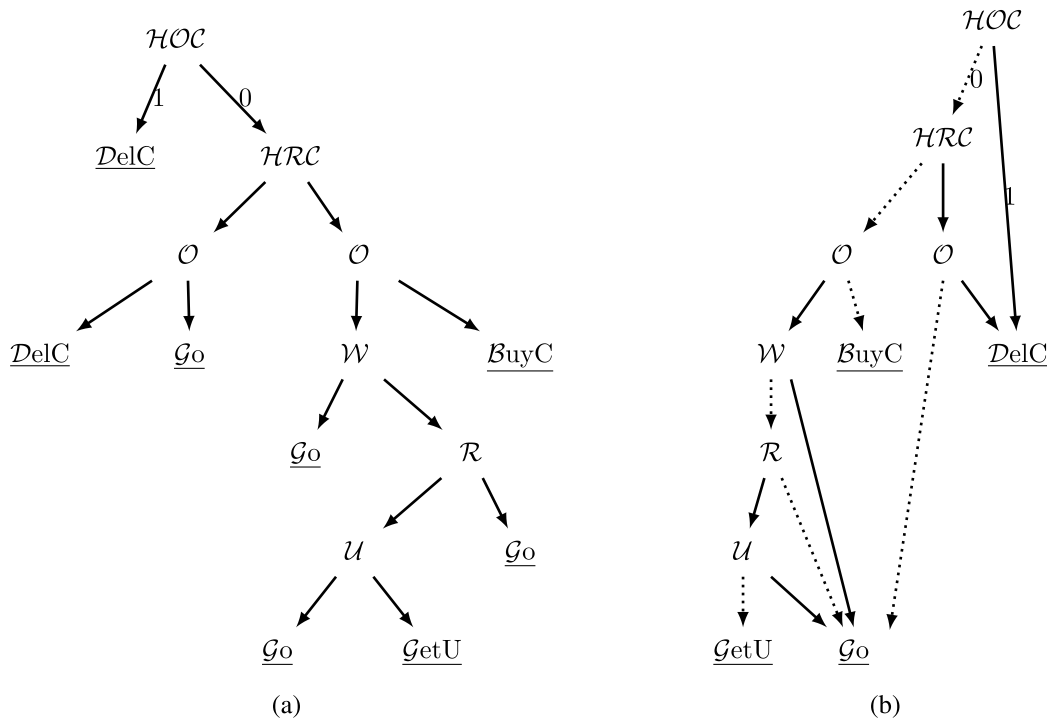


FIG. 3.18 – Représentation d'une politique $\pi(s)$ sous la forme d'un arbre de décision (figure a) et d'un ADD (figure b). Dans l'ADD, le nœud terminal noté $\underline{\mathcal{D}\mathrm{elC}}$ signifie $\pi(\mathcal{HOC} = 1) = \underline{\mathcal{D}\mathrm{elC}}$ et $\pi(\mathcal{HOC} = 0, \mathcal{HRC} = 1, \mathcal{O} = 1) = \underline{\mathcal{D}\mathrm{elC}}$.

dire où il pleut, que le robot n'est pas mouillé et qu'il n'a pas de parapluie. Uniquement dans ce cas particulier et pour ce contexte, l'action définie par π est d'aller chercher un parapluie (action $\mathcal{G}\mathrm{etU}$). La représentation Tree $[\pi]$ nécessite 4 feuilles pour représenter cette exception alors que ADD $[\pi]$ ne nécessite que 2 feuilles. Évidemment, alors que le gain est limité sur un problème tel que *Coffee Robot*, de telles factorisations peuvent se révéler extrêmement utiles pour un problème de grande taille.

Bien que la représentation d'une politique sous la forme d'un ADD puisse diminuer l'espace requis pour représenter la fonction, il ne permet en aucun cas de diminuer le nombre de tests nécessaires pour déterminer l'action spécifiée par la politique. En effet, un ADD permet de faire pointer plusieurs branches vers un même sous-graphe. Mais les tests seront quand même effectués dans ce sous-graphe.

Représentation d'une fonction de valeur

Enfin, la sémantique de la représentation d'une fonction de valeur V_π d'une politique π sous la forme d'un ADD est similaire à celle utilisée pour représenter une fonction de récompense : la valeur contenue à une feuille représente la récompense espérée sur le long terme en exécutant la

politique π en commençant à l'un des états contenus par l'une des partitions définies par l'ensemble des tests réalisés par les parents de cette feuille.

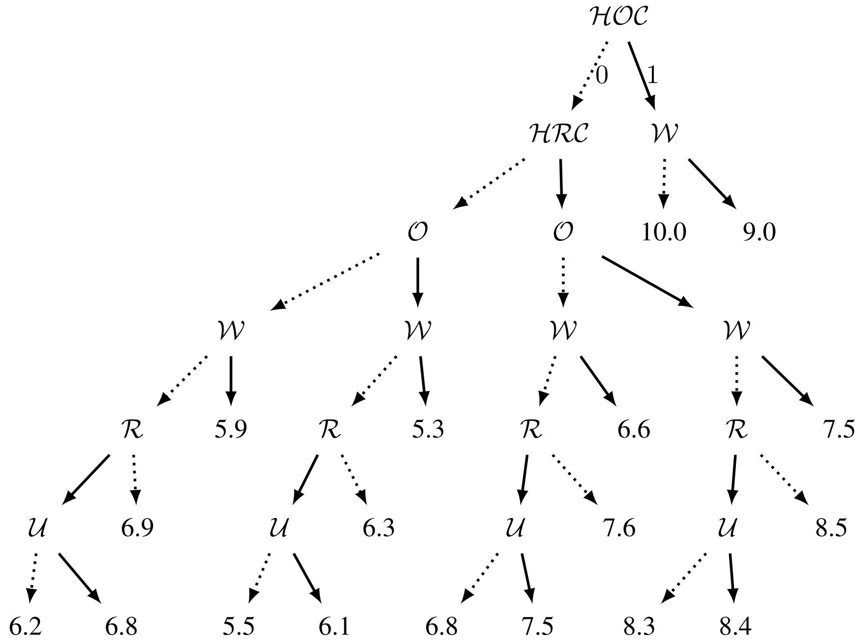


FIG. 3.19 – Représentation de la fonction de valeur $V_\pi(s)$ de la politique π sous la forme d'un ADD pour le problème *Coffee Robot*. La feuille notée 10.0 signifie $V_\pi(\mathcal{HOC} = 1, \mathcal{W} = 0) = 10.0$.

La figure 3.19 représente sous la forme d'un ADD la fonction de valeur ADD $[V_\pi]$ de la politique π représentée figure 3.18. En comparant à la même fonction de valeur représentée en utilisant un arbre de décision (figure 3.6, page 48), on peut constater que, pour cette fonction en particulier, la représentation en ADD est strictement identique : le nombre de feuilles et de nœuds intérieurs est le même. En effet, Tree $[V_\pi]$ ne contient aucun sous arbre identique et aucune factorisation supplémentaire à l'aide d'un ADD n'est donc possible.

De plus, on peut remarquer que SPUDD souffre du même défaut que SPI, c'est-à-dire que la représentation ADD $[V_\pi]$ ne permet pas d'exploiter certaines régularités présentes dans la définition de V_π dans le problème *Coffee Robot*. De même que pour la représentation Tree $[V_\pi]$ basée sur un arbre de décision, ADD $[V_\pi]$ est composé des différents sous-graphes regroupant les variables $\mathcal{R}$, $\mathcal{W}$, $\mathcal{U}$ et $\mathcal{O}$. Bien que ces sous-graphes possèdent exactement la même structure, les valeurs contenues aux feuilles sont différentes et ils ne peuvent donc pas être regroupés.

3.3.2 Algorithmes

De la même façon que pour SPI et SVI, la plupart des opérateurs sur les fonctions sont redéfinis et optimisés pour manipuler des ADDs. L'algorithme SPUDD reprend le principe de l'algorithme

Value Iteration pour l'adapter aux ADDs en supposant que toutes les variables du FMDP à résoudre sont binaires et que les variables sont préalablement ordonnées.

Les travaux sur SPUDD ont été prolongés avec APRICODD (St-Aubin et al., 2000) qui est une implémentation de SPUDD avec plusieurs améliorations. Premièrement, plusieurs étapes du calcul de l'équation 2.4 sont optimisées afin de permettre à l'utilisateur de pouvoir paramétrer un compromis temps de calcul sur espace mémoire nécessaire. De plus, il est possible de calculer des fonctions valeur approchées en spécifiant, soit une taille maximale de l'ADD représentant la fonction de valeur, ou bien une erreur maximale de la représentation (Hoey et al., 2000). Enfin, APRICODD propose plusieurs méthodes de réorganisation automatiques des variables afin d'éviter à l'utilisateur d'avoir à le spécifier manuellement. La dernière version d'APRICODD est disponible sur Internet[2].

3.4 Programmation Linéaire Approchée dans un FMDP

Nous avons vu qu'une alternative à la programmation dynamique pour résoudre un MDP est l'utilisation de la programmation linéaire (section 2.2.2). L'utilisation de cette technique à la résolution d'un FMDP est l'aboutissement de nombreux travaux commencés par Koller and Parr (1999, 2000) puis menés principalement par Guestrin (Guestrin et al., 2001; Guestrin, 2003; Guestrin et al., 2003b).

Nous avons vu que le programme linéaire généré pour résoudre un MDP pose un problème de complexité à la fois dans la fonction à optimiser, les variables à déterminer et le nombre de contraintes. Ces problèmes sont résolus en exploitant deux idées principales reposant principalement sur les indépendances relatives aux fonctions et la décomposition additive de la fonction de récompense.

La première idée exploite une représentation approchée de la fonction de valeur, plus précisément une combinaison linéaire de fonctions de base (Schweitzer and Seidmann, 1985), pour diminuer d'une part la complexité de la définition de la fonction à optimiser et du nombre de variables à déterminer et, d'autre part pour accélérer le calcul de la génération des contraintes. La deuxième idée propose d'utiliser un algorithme de décomposition des contraintes afin de pouvoir représenter l'ensemble des contraintes du programme linéaire de façon compacte.

Ces deux idées sont exploitées par deux algorithmes différents proposés par Guestrin et al. (2003b). Le premier est une reformulation de l'algorithme *Policy Iteration* utilisant la programmation linéaire pour la phase d'évaluation de la politique. Le deuxième algorithme part directement du programme linéaire de l'équation 2.13 et propose la construction directe d'un programme linéaire afin d'évaluer la fonction de valeur optimale du FMDP à résoudre. Pour un même ensemble de fonctions de base, l'algorithme basé sur *Policy Iteration* permet de calculer des politiques plus per-

[2] http://www.cs.toronto.edu/~jhoey/spudd

formantes que le deuxième. Cependant, l'algorithme reprenant directement le programme linéaire de l'équation 2.13 présente les avantages d'être à la fois simple à implémenter et beaucoup plus rapide à l'exécution (Guestrin et al., 2003b), permettant, le cas échéant, de rajouter des fonctions de base supplémentaires pour obtenir des meilleures politiques tout en restant rapide en temps de calcul. Nous nous concentrerons donc principalement sur ce dernier.

3.4.1 Représentations

Principalement deux représentations sont utilisées dans l'utilisation de la programmation linéaire telle qu'elle est proposée par Guestrin. La première représentation est une représentation tabulaire classique et permet d'exploiter uniquement les propriétés d'indépendance relative aux fonctions et de décomposition additive du problème. La deuxième représentation est une représentation structurée basée sur des règles (Zhang and Poole, 1999) permettant en plus d'utiliser les indépendances relatives aux contextes au sein d'une fonction. Bien que Guestrin et al. (2003b) montrent que, pour certains problèmes, une représentation tabulaire est plus rapide qu'une représentation structurée, nous pensons que les représentations structurées sont mieux adaptées pour représenter des problèmes réels, justement parce qu'elles exploitent les indépendances relatives aux contextes. De plus, le pire des cas des représentations structurées est souvent moins mauvais que le pire des cas des représentations tabulaires en terme de temps de calcul (St-Aubin et al., 2000; Guestrin, 2003).

Deux avantages sont avancés par Guestrin et al. (2003b) pour justifier l'utilisation des règles plutôt qu'une autre représentation telle que les arbres de décision ou les ADDs. Premièrement, cette représentation est bien adaptée à leur technique de décomposition des contraintes du programme linéaire. Deuxièmement, contrairement aux arbres de décision ou aux ADDs, les règles utilisées pour décrire une fonction peuvent ne pas être exclusives. Deux types de règles sont distinguées : les règles de probabilité, ou *probability rules* et, les règles de valeur, ou *value rules*. Les règles de probabilité sont utilisées pour représenter la fonction de transition alors que les règles de valeur sont utilisées pour définir les fonctions de récompense ainsi que les fonctions de valeur. Ces deux types de règles et leurs utilisations dans le cadre de la programmation linéaire approchée dans un FMDP sont décrits dans la suite de cette section, d'après Guestrin et al. (2003b). Une fonction $F(x)$ représentée avec un ensemble de règles est notée $\mathsf{Rules}\,[F]$.

Représentation de la fonction de transition

Le premier type de règles est utilisé pour représenter la fonction de transition, plus précisément les distributions de probabilités conditionnelles quantifiant les DBNs. Une règle correspond à un ou plusieurs contextes dans la distribution ayant la même probabilité. Nous commençons par définir la consistance entre deux contextes :

Définition 3 (Consistance entre deux contextes) *Soit $C \subseteq \{X, X'\}$, $c \in \underline{Dom}(C)$, $B \subseteq \{X, X'\}$ et $b \in \underline{Dom}(B)$. On dit que les deux contextes b et c sont consistants s'ils ont tous les deux les mêmes valeurs pour toutes les variables appartenant à l'intersection $C \cap B$.*

Ainsi, des contextes possédant des variables avec des valeurs identiques sont définis comme étant *consistants*. Les probabilités ayant la même valeur et des contextes consistants sont représentées avec des règles de probabilité :

Définition 4 (Règle de probabilité) *Une règle de probabilité $\eta = |c : p|$ est une fonction $\eta : \{X, X'\} \mapsto [0, 1]$ avec le contexte $c \in \underline{Dom}(C)$, $C \subseteq \{X, X'\}$ et $p \in [0, 1]$ et tel que $\eta(s, x') = p$ si les instanciations s et x' sont consistantes avec c, ou sinon est égal à 1.*

Deux règles sont dites consistantes si leurs contextes respectifs sont consistants. On définit maintenant un ensemble de règles de probabilité pour définir complètement une distribution de probabilités conditionnelle :

Définition 5 (Ensemble de règles de probabilité) *Un ensemble de règles P_a d'une distribution de probabilités conditionnelle est une fonction $P_a : (\{X_i'\} \cup X) \mapsto [0, 1]$ composée des règles de probabilité $\{\eta_1, \dots, \eta_m\}$ dont les contextes sont mutuellement exclusifs et exhaustifs. On définit : $P_a(x_i'|x) = \eta_j(x, x_i')$ avec η_j l'unique règle appartenant à P_a dont le contexte c_j est consistant avec (x_i', x). De plus, on a nécessairement : $\forall s \in S : \sum_{x_i'} P_a(x_i'|s) = 1$.*

Il est possible de définir $\underline{\text{Parents}}_a(X_i')$ comme l'union des variables appartenant aux contextes des règles appartenant à $P_a(X_i')$.

A l'instar des arbres de décision, les ensembles de règles de probabilité permettent d'exploiter les indépendances relatives aux contextes. De plus, les arbres de décision forment une partition complète d'un espace. Il est donc facile de définir un ensemble de règles mutuellement exclusives et exhaustives à partir d'un arbre de décision, comme le montre la figure 3.20 pour définir $P_{\mathcal{D}\text{elC}}(\mathcal{HOC}')$.

La probabilité $P_{\mathcal{D}\text{elC}}(\mathcal{HOC}' = 1 | \mathcal{HOC} = 0, \mathcal{O} = 1, \mathcal{HRC} = 1) = 0.8$ est représentée par la règle correspondante $|\mathcal{HOC} = 0 \wedge \mathcal{O} = 1 \wedge \mathcal{HRC} = 1 \wedge \mathcal{HOC}' = 1 : 0.8|$. On peut remarquer que, pour les tests concernant les variables X au temps t, le contexte de cette règle correspond aux tests réalisés dans l'arbre de décision pour atteindre la feuille 0.8. De plus, la variable X_i' au temps $t+1$ appartiennent aussi au contexte de la règle. Une distribution de probabilités conditionnelle $F(x)$ représentée avec un ensemble de règles de probabilité est notée $\text{Rules}_\text{p}[F]$.

Représentation de la fonction de récompense

Pour représenter la fonction de récompense d'un FMDP, on définit les règles de valeur :

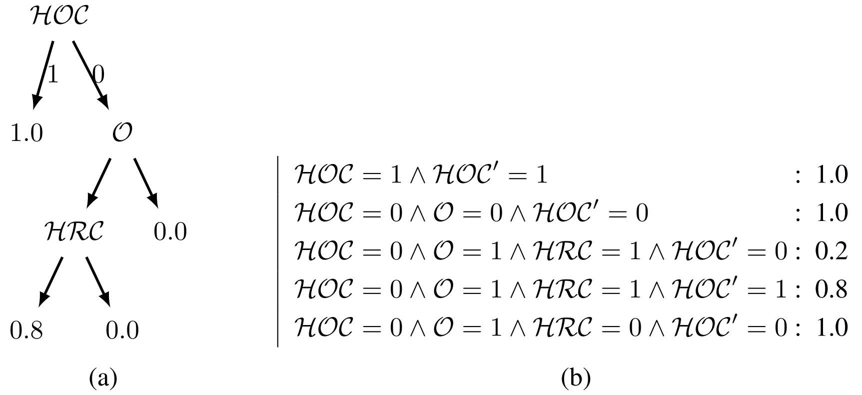


FIG. 3.20 – Représentation de la distribution de probabilités conditionnelle $P_{\mathcal{D}\mathrm{elC}}(\mathcal{HOC}')$ sous la forme d'un arbre de décision et d'un ensemble de règles. La règle $|\mathcal{HOC} = 0 \wedge \mathcal{O} = 1 \wedge \mathcal{HRC} = 1 \wedge \mathcal{HOC}' = 1 : 0.8|$ définit $P_{\mathcal{D}\mathrm{elC}}(\mathcal{HOC}' = 1 | \mathcal{HOC} = 0, \mathcal{O} = 1, \mathcal{HRC} = 1) = 0.8$.

Définition 6 (Règle de valeur) *Une règle de valeur $\rho = |c : v|$ est une fonction $\rho : X \rightarrow \mathbb{R}$ telle que $\rho(x) = v$ lorsque x est consistant avec le contexte c et 0 sinon.*

On note que le scope d'une règle de valeur est $\underline{\mathrm{Scope}}(\rho) = C$ avec C l'ensemble des variables instanciées dans le contexte c de la règle $\rho = |c : v|$.

Il est maintenant possible de définir une fonction comme un ensemble de règles de valeur :

Définition 7 (Ensemble de règles de valeur) *Un ensemble de règles de valeur représentant une fonction $f : X \mapsto \mathbb{R}$ est composé de l'ensemble des règles de valeur $\{\rho_1, \ldots, \rho_n\}$ telles que $f(x) = \sum_{i=1}^{n} \rho_i(x)$ avec $\forall i$: $\underline{\mathit{Scope}}(\rho_i) \subseteq X$.*

Une fonction F représentée avec un ensemble de règles de valeur est notée $\mathsf{Rules}_{\mathsf{v}}[F]$. De plus, on suppose qu'une récompense $R(s, a)$ peut s'écrire sous la forme d'une somme de fonctions de récompense dont le scope est limité :

$$R(s, a) = \sum_j r_j^a(s) \tag{3.4}$$

Cette représentation permet de représenter de façon naturelle des fonctions en exploitant à la fois des indépendances relatives aux contextes et une décomposition additive, comme le montre la figure 3.21.

Comme nous l'avons décrit dans la section 3.1.2, la fonction de récompense du problème *Coffee Robot* peut être décomposée en une somme de deux fonctions dont le scope n'est restreint qu'à une seule variable du problème. Plusieurs représentations peuvent être utilisées pour représenter les fonctions composant la fonction de récompense, notamment une forme tabulaire, d'arbres de décision ou d'un ensemble de règles de valeur.

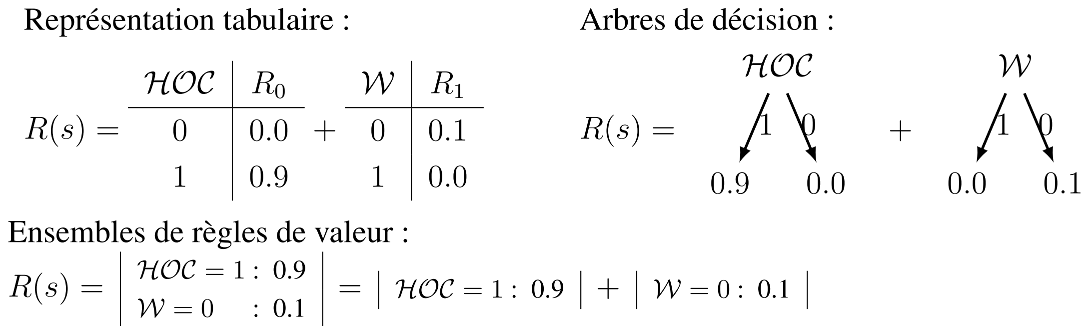


FIG. 3.21 – Représentation de la fonction de récompense R décomposée en une somme de fonction de récompense dont le scope est restreint à une seule variable du problème.

La figure 3.21 montre que deux configurations sont possibles, soit en regroupant les règles au sein d'un même ensemble pour ne définir qu'une seule fonction, soit en séparant les règles dans deux ensembles différents pour définir deux fonctions différentes. Enfin, on remarque que, même sur cet exemple simple, les règles de valeur permettent d'exploiter les indépendances relatives aux contextes pour décrire la fonction de récompense du problème *Coffee Robot*, contrairement aux arbres de décision. En effet, les arbres de décision requièrent la représentation des feuilles contenant la valeur 0, ce qui n'est pas le cas des règles.

Représentation d'une politique

Pour représenter une politique π de façon compacte, Guestrin et al. (2003b) reprennent une technique présentée par Koller and Parr (2000). Plutôt que d'utiliser un arbre de décision $\mathsf{Tree}[\pi]$ ou un ADD $\mathsf{ADD}[\pi]$ pour représenter une définition structurée de π, une action par défaut est choisie a priori dans le FMDP et la politique est représentée comme une liste de décision ordonnées. Chaque élément de la liste est composé de trois informations différentes : un contexte indiquant si la décision peut être prise étant donné un état s, l'action à exécuter si la décision est prise et enfin le bonus indiquant la récompense espérée supplémentaire pour cette décision comparée à la récompense espérée de l'action par défaut. Le dernier élément de la liste est toujours l'action par défaut, associée à un contexte vide (pour que ce dernier élément représente la décision par défaut à prendre si aucun autre n'est consistant avec l'état) et un bonus de 0. Une politique π représentée sous la forme d'une liste de décision est notée $\mathsf{List}[\pi]$. La figure 3.22 montre l'exemple d'une politique dans le problème *Coffee Robot* dont l'action par défaut est $\mathcal{G}$O.

On peut remarquer que la politique représentée figure 3.22 n'est pas simplifiée. En effet, par exemple, la règle 3 peut être agrégée avec la règle 1 puisque ces deux règles ont le même contexte (la règle 3 ne sera jamais utilisée puisque la règle 1 sera nécessairement utilisée avant). De plus, contrairement aux arbres de décision ou aux ADDs, le nombre de tests réalisés pour déterminer l'action à exécuter peut être supérieur au nombre de variables décrivant le problème.

	Contexte	Action	Bonus
0	$\mathcal{HOC}=0 \wedge \mathcal{HRC}=1 \wedge \mathcal{W}=0 \wedge \mathcal{R}=1 \wedge \mathcal{U}=0 \wedge \mathcal{O}=1$	$\mathcal{D}$elC	2.28
1	$\mathcal{HOC}=0 \wedge \mathcal{HRC}=0 \wedge \mathcal{W}=0 \wedge \mathcal{R}=1 \wedge \mathcal{U}=0 \wedge \mathcal{O}=0$	$\mathcal{B}$uyC	1.87
2	$\mathcal{HOC}=0 \wedge \mathcal{HRC}=1 \wedge \mathcal{W}=0 \wedge \mathcal{R}=1 \wedge \mathcal{U}=1 \wedge \mathcal{O}=1$	$\mathcal{D}$elC	1.60
3	$\mathcal{HOC}=0 \wedge \mathcal{HRC}=1 \wedge \mathcal{W}=1 \wedge \mathcal{O}=1$	$\mathcal{D}$elC	1.45
4	$\mathcal{HOC}=0 \wedge \mathcal{HRC}=1 \wedge \mathcal{W}=0 \wedge \mathcal{R}=0 \wedge \mathcal{O}=1$	$\mathcal{D}$elC	1.44
5	$\mathcal{HOC}=0 \wedge \mathcal{HRC}=0 \wedge \mathcal{W}=0 \wedge \mathcal{R}=1 \wedge \mathcal{U}=1 \wedge \mathcal{O}=0$	$\mathcal{B}$uyC	1.27
6	$\mathcal{HOC}=0 \wedge \mathcal{HRC}=0 \wedge \mathcal{W}=1 \wedge \mathcal{O}=0$	$\mathcal{B}$uyC	1.18
7	$\mathcal{HOC}=0 \wedge \mathcal{HRC}=0 \wedge \mathcal{W}=0 \wedge \mathcal{R}=0 \wedge \mathcal{O}=0$	$\mathcal{B}$uyC	1.18
8	$\mathcal{HOC}=1 \wedge \mathcal{W}=0 \wedge \mathcal{R}=1 \wedge \mathcal{U}=0$	$\mathcal{D}$elC	0.84
9	$\mathcal{HOC}=0 \wedge \mathcal{HRC}=0 \wedge \mathcal{W}=0 \wedge \mathcal{R}=1 \wedge \mathcal{U}=0 \wedge \mathcal{O}=1$	$\mathcal{G}$etU	0.18
10	$\mathcal{HOC}=1 \wedge \mathcal{W}=0 \wedge \mathcal{R}=1 \wedge \mathcal{U}=1$	$\mathcal{D}$elC	0.09
11	$\emptyset$	$\mathcal{G}$o	0.00

FIG. 3.22 – Représentation d'une politique $\pi(s)$ sous la forme d'une liste de décision List $[\pi]$ (avec $\mathcal{G}$o l'action par défaut).

Enfin, pour certains problèmes de grande taille, quelle que soit la méthode de planification utilisée, une représentation explicite de la politique optimale, même factorisée, est impossible puisqu'il est nécessaire pour chaque état d'évaluer toutes les variables du problème afin de déterminer la meilleure action à réaliser par l'agent. La méthode que nous reprenons de Guestrin et al. (2003b) et que nous présentons dans ce manuscrit ne nécessite pas une telle représentation explicite de la politique.

Représentation de la fonction de valeur

Nous avons vu qu'un MDP pouvait s'écrire sous la forme d'un programme linéaire de la façon suivante (section 2.2.2, équation 2.13, page 32) :

$$\begin{aligned} &\text{Déterminer} && V(s), \forall s \in S; \\ &\text{minimisant} && \sum_s \alpha(s) V(s); \\ &\text{et satisfaisant} && V(s) \geq R(s,a) + \gamma \sum_{s'} P(s'|s,a) V(s'), \forall s \in S, \forall a \in A \end{aligned} \quad (3.5)$$

Cependant, une telle représentation pose un problème de complexité, aussi bien dans le nombre de variables à déterminer, dans le nombre de termes de la somme de la fonction objectif, que dans le nombre de contraintes.

Une solution pour éviter l'explosion combinatoire concernant le nombre de variables à déterminer et le nombre de termes dans la fonction à minimiser est l'approximation de la fonction de valeur par une *combinaison linéaire* proposée par Bellman et al. (1963). L'espace des fonctions de valeur approchées $\tilde{V} \in \mathcal{H} \subseteq \mathbb{R}^n$ est défini via un ensemble de *fonctions de base*, ou *basis functions*, dont

le scope est limité à un petit nombre de variables :

Définition 8 (Fonction de Valeur Linéaire) *Une fonction de valeur linéaire $\tilde{V}$ sur un ensemble de fonctions de base $H = \{h_0, \ldots, h_k\}$ est une fonction telle que $\tilde{V}(s) = \sum_{j=1}^{k} w_j h_j(s)$ avec $w \in \mathbb{R}^k$.*

Cette approximation peut être utilisée pour redéfinir le programme linéaire simplement en remplaçant la fonction de valeur à déterminer par son approximation (Schweitzer and Seidmann, 1985) :

$$\begin{array}{rl} \text{Déterminer} & w_1, \ldots, w_k; \\ \text{minimisant} & \sum_s \alpha(s) \sum_i w_i h_i(s); \\ \text{et satisfaisant} & \sum_i w_i h_i(s) \geq \\ & R(s,a) + \gamma \sum_{s'} P(s'|s,a) \sum_i w_i h_i(s'), \forall s \in S, \forall a \in A \end{array} \tag{3.6}$$

Ainsi, plutôt que de déterminer la fonction de valeur dans l'espace complet des fonctions de valeur, l'espace de recherche est réduit à l'espace des valeurs pour l'ensemble des coefficients utilisés dans la combinaison linéaire. De plus, nous verrons section 3.4.3 que le fait de limiter le scope des fonctions de base permet d'exploiter les indépendances relatives aux fonctions de base.

On peut donc remarquer que le nombre de variables à déterminer du programme linéaire n'est plus le nombre d'états possibles mais le nombre de coefficients dans l'approximation linéaire. Cependant, le nombre de termes dans la fonction à minimiser et le nombre de contraintes sont toujours égaux aux nombres d'états dans le problème. Nous verrons comment réduire cette complexité section 3.4.4.

Pour un tel programme, une solution existe si une fonction de base constante est incluse à l'ensemble des fonctions de base (Schweitzer and Seidmann, 1985). Nous supposons donc qu'une telle fonction h_0 telle que $h_0(s) = 1, \forall s \in S$ est systématiquement incluse à l'ensemble des fonctions de base. De plus, il est important de noter que le choix des pondérations d'intérêt $\alpha(s)$ influe sur la qualité de l'approximation (de Farias and Van Roy, 2001).

En plus de la diminution de la complexité du programme linéaire, une telle approximation de la fonction de valeur permet d'exploiter à la fois les indépendances relatives aux fonctions et certaines régularités de la structure de la fonction de valeur. Dans le problème *Coffee Robot*, la figure 3.23 montre un exemple de décomposition additive de la fonction de valeur approchée permettant d'exploiter une régularité que des représentations telles que les arbres de décision et les ADDs ne pouvaient pas utiliser.

La définition d'une fonction valeur Tree $[V]$ du problème est décomposée en deux fonctions de base Tree $[h_1]$ et Tree $[h_2]$[3] et permet une approximation de Tree $[V_\pi]$ dont l'erreur est inférieure à 1.

[3]Ces deux fonctions Tree $[h_1]$ et Tree $[h_2]$ ont été obtenues à partir de l'arbre de décision représentant la fonction

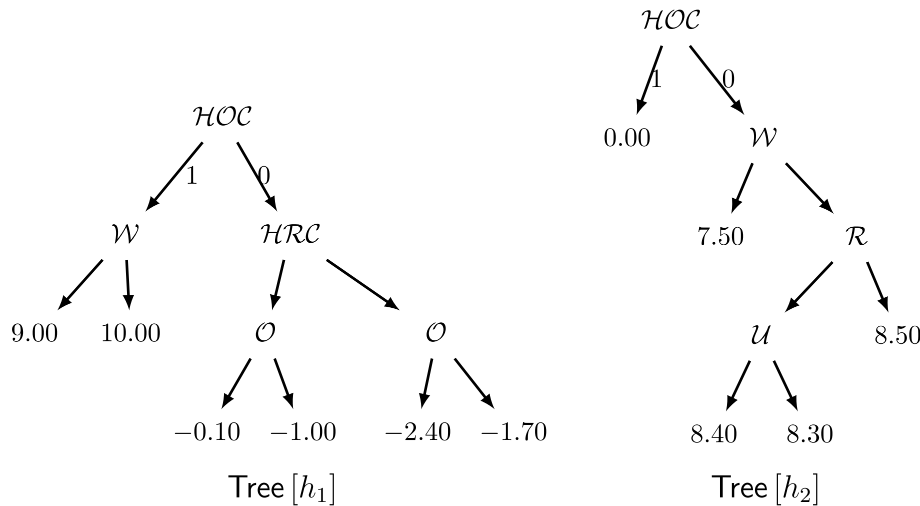


FIG. 3.23 – Exemple de décomposition de la fonction de valeur du problème *Coffee Robot* sous la forme de deux arbres de décision représentant deux fonctions de base permettant de calculer la politique $\pi(s)$ (figure 3.22). La fonction de valeur optimale approchée est : $\tilde{V}^*(s) = 0.63 \cdot \text{Tree}\,[h_0] + 0.94 \cdot \text{Tree}\,[h_1] + 0.96 \cdot \text{Tree}\,[h_2]$. L'arbre Tree $[h_0]$ n'est pas illustré puisqu'il définit une fonction constante et ne contient donc qu'une seule feuille égale à 1.

La propriété de décomposition additive est exploitée puisque, plutôt que de contenir 18 feuilles, cette représentation ne nécessite que 11 feuilles pour les deux arbres (soit 20 nœuds au total, au lieu de 35 nœuds pour Tree $[V_\pi]$). Cette décomposition contient deux fonctions de base (trois en comptant la fonction constante h_0), donc trois coefficients, w_0, w_1 et w_2, sont à déterminer dans le programme linéaire 3.6.

Une décomposition supplémentaire peut être obtenue en décomposant les arbres Tree $[h_1]$ et Tree $[h_2]$ en deux ensembles de fonctions de base pour associer un coefficient w_i à chacune des feuilles de ces deux arbres. Une telle décomposition est montrée figure 3.24 pour l'arbre Tree $[h_2]$ où, pour chacune de ces feuilles l_i, un nouvel arbre Tree $[h_{2_i}]$ est construit contenant 0.0 à toutes les feuilles l_j sauf pour la feuille l_i dont la valeur est à pondérer par w_i. Cependant, comme nous l'avons décrit section 3.3.1, les arbres de décision ne sont pas adaptés à la représentation de fonctions disjonctives telles que les fonctions Tree $[h_{2_i}]$ utilisées pour décomposer Tree $[h_2]$. L'utilisation de règles de valeur reste la représentation la plus compacte, comme le montre la figure 3.25 où les arbres de décision Tree $[h_{2_i}]$ sont réécrits sous la forme de règles de valeur $\text{Rules}_\text{v}\,[h_{2_i}]$.

Enfin, lorsque la fonction de récompense possède une décomposition additive, comme c'est le cas dans le problème *Coffee Robot*, il semble naturel que la fonction de valeur du problème possède également cette propriété. Cependant, ces deux propriétés ne sont pas nécessairement corrélées. En effet, bien qu'une fonction de récompense puisse ne présenter aucune décomposition additive, une combinaison linéaire de fonctions de base peut quand même permettre de déterminer avec une faible

de valeur Tree $[V_\pi]$ dans le problème *Coffee Robot*, figure 3.6 (page 48).

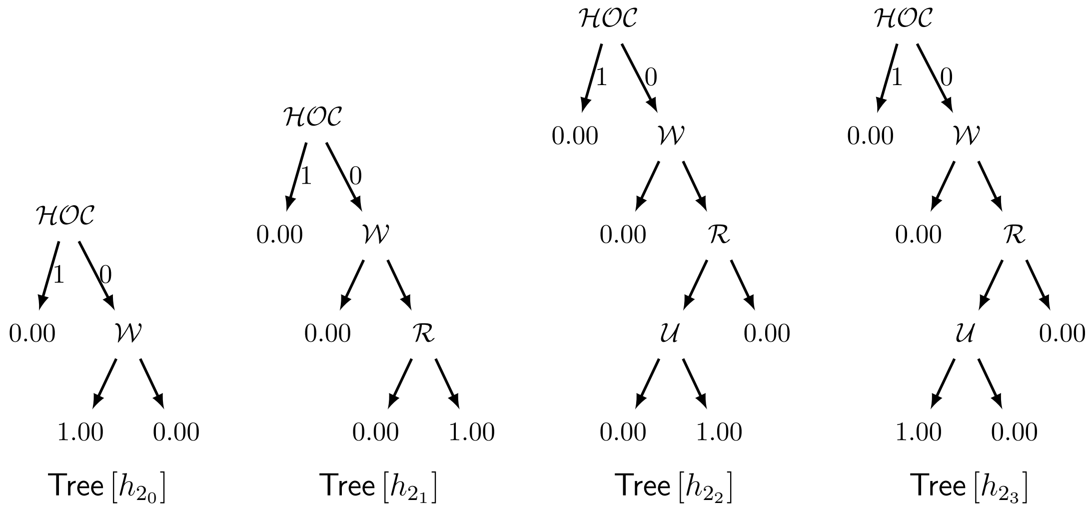


FIG. 3.24 – Décomposition de Tree $[h_2]$ en une combinaison linéaire $\sum_i w_{2_i}\text{Tree}\,[h_{2_i}]$.

$$
\begin{aligned}
\text{Rules}_\text{v}\,[h_{2_0}] &= \left|\, \mathcal{HOC}=0 \wedge \mathcal{W}=1:\ 1.0 \,\right| \\
\text{Rules}_\text{v}\,[h_{2_1}] &= \left|\, \mathcal{HOC}=0 \wedge \mathcal{W}=0 \wedge \mathcal{R}=0:\ 1.0 \,\right| \\
\text{Rules}_\text{v}\,[h_{2_2}] &= \left|\, \mathcal{HOC}=0 \wedge \mathcal{W}=0 \wedge \mathcal{R}=1 \wedge \mathcal{U}=1:\ 1.0 \,\right| \\
\text{Rules}_\text{v}\,[h_{2_3}] &= \left|\, \mathcal{HOC}=0 \wedge \mathcal{W}=0 \wedge \mathcal{R}=1 \wedge \mathcal{U}=0:\ 1.0 \,\right|
\end{aligned}
$$

FIG. 3.25 – Réécriture des fonctions Tree $[h_{2_i}]$ sous la forme plus compacte de règles de valeur Rules$_\text{v}$ $[h_{2_i}]$.

erreur d'approximation les fonctions de valeur du problème. Une telle représentation est donc plus générale que les représentations sous forme d'arbre de décision ou d'ADDs proposées par SPI ou SPUDD (Guestrin et al., 2003b). Réciproquement, des représentations compactes des fonctions de transition et de récompense n'impliquent pas non plus une représentation compacte de la fonction de valeur (Koller and Parr, 1999; Mundhenk et al., 2000; Liberatore, 2002).

3.4.2 Manipulations

De même que pour les arbres de décision et les ADDs, il est nécessaire de redéfinir les opérations telles que l'addition, la multiplication ou la maximisation afin d'exploiter les indépendances relatives aux contextes lorsque les fonctions sont représentées avec un ensemble de règles. Cette section définit ces opérations pour les deux types de règles, les règles de probabilité et les règles de valeur, en commençant par les règles ayant le même contexte :

Définition 9 (Produit et somme de règles de même contexte) *Soit* $\rho_1 = |c : v_1|$ *et* $\rho_2 = |c : v_1|$ *deux règles ayant le même contexte* c*. Le produit des règles* ρ_1 *et* ρ_2 *est défini tel que* $\rho_1 \cdot \rho_2 = |c : v_1 \cdot v_2|$*. La somme des règles* ρ_1 *et* ρ_2 *est défini tel que* $\rho_1 + \rho_2 = |c : v_1 + v_2|$*.*

Nous définissons maintenant l'opérateur de maximisation sur une variable Y d'un ensemble de règles, qui, à l'exception de Y, partagent le même contexte :

Définition 10 (Maximisation de règles de même contexte) *Soit Y une variable avec $\underline{Dom}(Y) = \{y_0, \ldots, y_k\}$. Soit, pour tout $i \in \{1, \ldots, k\}$, une règle ρ_i tel que $\rho_i = |c \wedge Y = y_i : v_i|$. Alors l'opérateur de maximisation d'une fonction* $\mathsf{Rules}[f]$ *tel que* $\mathsf{Rules}[f] = \rho_0 + \ldots + \rho_k$ *sur la variable Y est défini tel que* $\max_Y(f) = |c : max_i(v_i)|$.

Une fois cette opérateur appliqué sur la fonction f, on peut remarquer que la variable Y a été maximisée et qu'elle ne fait plus partie du scope de la fonction $\max_Y(f)$.

La définition de ces trois opérateurs suppose que le contexte des règles est identiques. Afin de pouvoir les utiliser sur l'ensemble des règles, il est nécessaire de définir un nouvel opérateur :

Définition 11 (Partitionnement de règles) *Soit $\rho = |c : v|$ une règle dont le scope est $\underline{Scope}(C)$. Soit Y une variable avec $\underline{Dom}(Y) = \{y_0, \ldots, y_k\}$. La partition $\underline{Split}(\rho\angle Y)$ de la règle ρ sur la variable Y est définie de la façon suivante :*

$$\underline{Split}(\rho\angle Y) = \begin{cases} \{\rho\} \text{ si } Y \in \underline{Scope}(C); \\ \{|c \wedge Y = y_i : v| \quad | \quad y_i \in \underline{Dom}(Y)\} \text{ sinon.} \end{cases}$$

L'opération de partitionnement ajoute donc une nouvelle variable dans le scope d'une fonction. L'exemple suivant illustre l'opérateur de partitionnement sur l'une des règles de la fonction de récompense de *Coffee Robot* :

$$\underline{\mathrm{Split}}(|\mathcal{W} = 0 : 0.1|\angle\mathcal{HOC}) = \left| \begin{array}{l} \mathcal{W} = 0 \wedge \mathcal{HOC} = 0 : \ 0.1 \\ \mathcal{W} = 0 \wedge \mathcal{HOC} = 1 : \ 0.1 \end{array} \right|.$$

En faisant l'analogie avec les arbres de décision, l'opérateur consisterait à remplacer une feuille de l'arbre par un nœud de décision et à placer la valeur contenue dans la feuille à chacune des nouvelles feuilles ayant comme parent le nouveau nœud de décision.

Afin de pouvoir appliquer les opérateurs d'addition, de multiplication et de maximisation sur des ensembles quelconques de règles, nous pouvons utiliser l'opérateur de partitionnement récursivement de la façon suivante :

Définition 12 (Partitionnement récursif de règles) *Soit une règle $\rho = |c : v|$. Soit b un contexte tel que $b \in \underline{Dom}(B)$. Le partitionnement récursif $\underline{Split}(\rho\angle b)$ de ρ sur le contexte b est défini de la façon suivante :*

$$\underline{Split}(\rho\angle b) = \begin{cases} \{\rho\} \text{ si } c \text{ n'est pas consistant avec } b; \\ \{\rho\} \text{ si } \underline{Scope}(B) \subseteq \underline{Scope}(C); \\ \{\underline{Split}(\rho_i\angle b) | \rho_i \in \underline{Split}(\rho\angle Y)\}, \text{ avec } Y \in \underline{Scope}(B) - \underline{Scope}(C) \text{, sinon.} \end{cases}$$

Une fois le résultat de l'opérateur calculé, une seule règle, dont le contexte est $c \wedge b$, est consistante avec le contexte b. Le résultat des opérateurs d'addition et de multiplication sur deux ensembles de règles peut donc être calculé en partitionnant récursivement les règles consistantes pour qu'elles partagent le même contexte puis les remplacer par le résultat des opérateurs dans ce contexte. L'exemple suivant illustre une addition sur deux ensembles de règles extrais de l'exemple utilisé pour l'opérateur Append, figure 3.8 (page 50), sur les arbres de décision :

$$\begin{aligned}
\text{Rules}\,[T_1] + \text{Rules}\,[T_2] &= \left|\begin{array}{l} X=1:\ l_1^1 \\ X=0:\ l_1^2 \end{array}\right| + \left|\begin{array}{ll} Y=1 & :\ l_2^0 \\ Y=0 \wedge X=1 & :\ l_2^1 \\ Y=0 \wedge X=0 & :\ l_2^2 \end{array}\right| \\
&= \left|\begin{array}{l} X=1 \wedge Y=0:\ l_1^1 \\ X=1 \wedge Y=1:\ l_1^1 \\ X=0 \wedge Y=0:\ l_1^2 \\ X=0 \wedge Y=1:\ l_1^2 \end{array}\right| + \left|\begin{array}{l} Y=1 \wedge X=0:\ l_2^0 \\ Y=1 \wedge X=1:\ l_2^0 \\ Y=0 \wedge X=1:\ l_2^1 \\ Y=0 \wedge X=0:\ l_2^2 \end{array}\right| \quad \begin{array}{l}\big(\underline{\text{Split}}(\text{Rules}\,[T_1]\,\angle Y)+ \\ \underline{\text{Split}}(\text{Rules}\,[T_2]\,\angle X)\big)\end{array} \\
&= \left|\begin{array}{l} Y=1 \wedge X=0:\ l_1^2 + l_2^0 \\ Y=1 \wedge X=1:\ l_1^1 + l_2^0 \\ Y=0 \wedge X=1:\ l_1^1 + l_2^1 \\ Y=0 \wedge X=0:\ l_1^2 + l_2^2 \end{array}\right|
\end{aligned}$$

Un dernier opérateur utilisé dans les algorithmes proposés par Guestrin et al. (2003b) est un opérateur de maximisation, RuleMax sur un ensemble de règles et par rapport à une variable donnée. Il est décrit figure 3.26. L'algorithme commence par ajouter des règles de valeur égales à 0 pour garantir qu'une règle existe pour l'ensemble des valeurs de la variable à maximiser. Dans un deuxième temps, les règles consistantes sont ajoutées les unes aux autres. Enfin, pour les règles dont le contexte est identique, le résultat de la maximisation est ajouté au résultat $\text{Rules}_{\text{v}}\,[g]$. Sinon, le contexte des règles est partitionné afin d'être égal aux contextes des autres règles consistantes.

3.4.3 Calcul d'une fonction de valeur d'action sur une itération

De la même façon que les autres algorithmes de planification, il est nécessaire de calculer une fonction de valeur d'action à partir d'une fonction de valeur suivant l'équation 2.4 (page 26) :

$$Q_a^V(s) = R(s,a) + \gamma \sum_{s'} P(s'|s,a) V(s')$$

Cependant, ce calcul doit être réalisé en remplaçant la fonction de valeur V par son approximation $\tilde{V}$. L'équation devient alors :

$$Q_a^{\tilde{V}}(s) = R(s,a) + \gamma \sum_{s'} P(s'|s,a) \sum_i w_i h_i(x) \tag{3.7}$$

Entrée(s) : $\mathsf{Rules_v}\,[f]\,, B$ avec B la variable à maximiser **Sortie(s) :** $\mathsf{Rules_v}\,[g]$

---

1. Soit $g = \{\}$
2. Pour tout $b_i \in \underline{\mathrm{Dom}}(B)$: ajouter à $\mathsf{Rules_v}\,[f]$ la règle $|B = b_i : 0|$
3. Tant qu'il existe deux règles consistantes $\rho_1 = |c_1 : v_1|$ et $\rho_2 = |c_2 : v_2|$:
   Si $c_1 = c_2$, alors remplacer ces deux règles par $|c_1 : v_1 + v_2|$, sinon remplacer ces règles par $\{\underline{\mathrm{Split}}(\rho_1 \angle c_2) \cup \underline{\mathrm{Split}}(\rho_2 \angle c_1)\}$
4. Tant que $\mathsf{Rules_v}\,[f]$ n'est pas vide :
   S'il existe des règles tels que $|c \wedge B = b_i : v_i|, \forall b_i \in \underline{\mathrm{Dom}}(B)$:
   Alors : enlever ces règles de $\mathsf{Rules_v}\,[f]$ et ajouter la règle $|c : \max_i v_i|$ à g
   Sinon : sélectionner deux règles $\rho_i = |c_i \wedge W = b_i : v_i|$ et $\rho_j = |c_j \wedge B = b_j : v_j|$ tel que c_i et c_j soient consistants (mais non identique) puis les remplacer par $\{\underline{\mathrm{Split}}(\rho_i \angle c_j) \cup \underline{\mathrm{Split}}(\rho_j \angle c_i)\}$
5. Retourner $\mathsf{Rules_v}\,[g]$

FIG. 3.26 – L'algorithme $\underline{\mathrm{RuleMax}}(\mathsf{Rules_v}\,[f]\,, B)$.

Afin d'éviter l'explosion combinatoire dans le calcul de cette équation, il est possible d'exploiter l'indépendance relative aux fonctions, notamment en décomposant le produit des probabilités par la récompense attendue au prochain pas de temps (Koller and Parr, 1999) :

$$\begin{aligned} Q_a^{\tilde{V}}(s) &= R(s,a) + \gamma \sum_{s'} P(s'|s,a) \sum_i w_i h_i(s') \\ &= R(s,a) + \gamma \sum_i w_i \sum_{s'} P(s'|s,a) h_i(s') \\ &= R(s,a) + \gamma \sum_i w_i g_i^a(s) \end{aligned} \tag{3.8}$$

avec $g_i^a(s) = \sum_{s'} P(s'|s,a) h_i(s')$. Cette décomposition traduit le fait que la récompense espérée peut être calculée de façon indépendante pour chaque fonction de base composant l'approximation de la fonction de valeur.

Nous supposons que chaque fonction de base h_i est décrite par un ensemble de règles de valeur telle que $h_i(s) = \sum_j \rho_j^{h_i}(s)$ avec $\rho_j^{h_i} = |c_j^{h_i} : v_j^{h_i}|$. Pour une fonction de base h_i donnée, il est donc possible de décomposer le calcul de la récompense espérée $g_i^a(s)$ pour chaque règle (Guestrin et al., 2003b) :

$$\begin{aligned} g_i^a(s) &= \sum_{s'} P(s'|s,a) h_i(s') \\ &= \sum_{s'} P(s'|s,a) \sum_j \rho_j^{h_i}(s) \end{aligned}$$

$$= \sum_j \sum_{s'} P(s'|s,a)\rho_j^{h_i}(s)$$
$$= \sum_j P(c_j^{h_i}|s,a)v_j^{h_i} \quad (3.9)$$

L'algorithme BackProjRule, décrit dans la figure 3.27, permet de calculer la récompense espérée $P(c_j^{h_i}|s,a)v_j^{h_i}$ en exploitant les indépendances relatives au contexte $c_j^{h_i}$. Il est composé de trois étapes. En premier lieu, les distributions de probabilités consistantes avec la règle ρ sont sélectionnées et ajoutées à $\mathcal{P}$. Ensuite, les règles consistantes au sein de cet ensemble sont multipliées pour former un ensemble de règles de probabilité mutuellement exclusives. Enfin, ces règles de probabilité sont multipliées avec la valeur v de ρ pour former l'ensemble g de règles de valeurs. Le calcul de $g_i^a(s)$ sous la forme de règles de valeur se définit donc ainsi : $\mathsf{Rules_v}[g_i^a] = \sum_j \underline{\text{BackProjRule}}(\rho_j^{h_i}, a)$.

Entrée(s) : $\mathsf{Rules_p}[P], a, \rho$ avec : $\rho = |c : v|$, $C = \underline{\text{Scope}}(\rho)$ et $c \in \underline{\text{Dom}}(C)$ **Sortie(s) :** g

---

1. Construire l'ensemble $\mathcal{P}$ de règles de probabilité tel que :
   $\mathcal{P} = \{\eta_j \in P(X'_i|\underline{\text{Parents}}(X'_i))|X'_i \in C$ et c est consistant avec $c_j\}$
2. Enlever les variables X' des contextes de toutes les règles de $\mathcal{P}$
3. Tant qu'il existe deux règles consistantes $\eta_1 = |c_1 : p_1|$ et $\eta_2 = |c_2 : p_2|$:
   Si $c_1 = c_2$, alors remplacer ces deux règles par $|c_1 : p_1 \cdot p_2|$, sinon remplacer ces règles par l'ensemble $\{\underline{\text{Split}}(\eta_1 \angle c_2) \cup \underline{\text{Split}}(\eta_2 \angle c_1)\}$
4. Construire l'ensemble g des règles de valeur tel que :
   $\forall \eta_i \in \mathcal{P} : g = g \cup \{|c_i : p_i \cdot v|\}$
5. Retourner g

FIG. 3.27 – L'algorithme BackProjRule(ρ, a).

Il est important de noter que cet algorithme effectue le même calcul que les étapes 1 et 2 de l'algorithme Regress(Tree $[V]$, a) de SPI, figure 3.9 (page 51). Enfin, la représentation $\mathsf{Rules_v}\left[Q_a^{\tilde{V}}\right]$ sous la forme de règle de valeur de la fonction $Q_a^{\tilde{V}}(s)$ s'obtient en multipliant la valeur de chaque règle de $\mathsf{Rules_v}[g_i^a]$ par γ puis en ajoutant à l'ensemble de règles les règles de valeur $\mathsf{Rules_v}[R]$.

Construction d'une politique gloutonne

Une fois qu'il est possible de calculer l'équation 2.4 permettant de définir une fonction de valeur d'action à partir d'une fonction de valeur, il est possible de définir l'opérateur Greedy construisant

une politique gloutonne à partir de l'ensemble des fonctions de valeur d'action. L'algorithme proposé par Guestrin et al. (2003b) construit une liste de décision telle que celle décrite figure 3.22.

Entrée(s) : d, $\text{Rules}_\text{v}\left[Q_d\right]$, $\text{Rules}_\text{v}\left[Q_a\right]$ avec d l'action par défaut **Sortie(s) :** $\text{List}\left[\pi\right]$

1. Définir $\text{List}\left[\pi\right] = \{\}$
2. Pour chaque action $a \in A$ différente de l'action par défaut d :
   (a) Calculer le bonus δ_a d'exécuter l'action a : $\text{Rules}_\text{v}\left[\delta_a\right] = \text{Rules}_\text{v}\left[Q_a\right] - \text{Rules}_\text{v}\left[Q_d\right]$
   (b) Ajouter les contextes pour lesquelles le bonus est positif :
   $\forall \rho_i = |c_i : v_i|$: si $v_i > 0$ alors $\text{List}\left[\pi\right] = \text{List}\left[\pi\right] \cup \langle c_i, a, v_i\rangle$
3. Ajouter l'action par défaut : $\text{List}\left[\pi\right] = \text{List}\left[\pi\right] \cup \langle \emptyset, d, 0\rangle$
4. Trier $\text{List}\left[\pi\right]$ par ordre décroissant des bonus associés à chaque décision
5. Retourner $\text{List}\left[\pi\right]$

FIG. 3.28 – L'algorithme DecisionList$(d, \text{Rules}_\text{v}\left[Q_d\right], \text{Rules}_\text{v}\left[Q_a\right])$.

Pour représenter de façon compacte une politique, nous supposons qu'une action d par défaut peut être définie et pour laquelle nous avons un modèle de transition par défaut (Koller and Parr, 2000). Une fonction de récompense, pouvant s'écrire comme une somme $\sum_{i=1}^{r} R_i(s)$ de fonctions dont le scope est restreint, est aussi associée à l'action par défaut. Pour chaque action a différente de l'action par défaut, un modèle de transition composé uniquement des variables ayant une distribution de probabilités différente de l'action par défaut est défini. De plus, une récompense supplémentaire R^a est ajoutée pour définir la récompense totale $R^a + \sum_{i=1}^{r} R_i(s)$ obtenue lorsque l'action a est exécutée. La fonction de valeur d'action associée à l'action par défaut est définie de la façon suivante :

$$Q_d(s) = \sum_{i=1}^{r} R_i(s) + \sum_i w_i g_i^d(s) \tag{3.10}$$

La fonction de valeur d'une action a différente de l'action par défaut est définie de la façon suivante :

$$Q_a(s) = R^a(s) + \sum_{i=1}^{r} R_i(s) + \sum_i w_i g_i^a(s) \tag{3.11}$$

Le bonus $\delta_a(s)$ obtenu en exécutant l'action a plutôt que l'action d dans l'état s peut donc être calculé en utilisant :

$$\begin{aligned} \delta_a(s) &= Q_a(s) - Q_d(s) \\ &= R^a(s) + \sum_i w_i \left(g_i^a(s) - g_i^d(s)\right) \end{aligned} \tag{3.12}$$

À partir de ce calcul, l'algorithme DecisionList décrit figure 3.28 construit ensuite une politique gloutonne sous la forme d'une liste de décision $\mathsf{List}\,[\pi]$. L'algorithme commence d'abord par calculer, pour tous les contextes, les bonus des différentes actions (par rapport à l'action par défaut). Lorsque ce bonus est strictement positif, alors il est ajouté à la liste de décision. L'action par défaut est ensuite ajoutée dans la liste de décision avec un bonus de 0. Enfin, la liste est triée par ordre décroissant afin de pouvoir sélectionner la première décision pour laquelle le contexte est consistant avec l'état et ayant le meilleur bonus.

À partir du calcul des fonctions de valeur d'action (algorithme BackProjRule) et de la construction d'une politique gloutonne (algorithme DecisionList), il est possible de définir l'algorithme Greedy, décrit figure 3.29, qui construit une politique gloutonne à partir d'une fonction de valeur. De la même façon que dans les autres méthodes de planification, les fonctions de valeur d'action sont calculées à partir d'une fonction de valeur donnée en utilisant l'algorithme BackProjRule. L'opération de maximisation est ensuite réalisée avec l'algorithme DecisionList.

Entrée(s) : $\mathsf{Rules_p}\,[P]$, $\mathsf{Rules_v}\,[R]$, $\mathsf{Rules_v}\,[V]$, d avec d l'action par défaut **Sortie(s) :** $\mathsf{List}\,[\pi]$

---

1. Pour chaque l'action par défaut d, calculer :
   $\mathsf{Rules_v}\,[Q_d] = \mathsf{Rules_v}\,[R] + \sum_i w_i \sum_j \underline{\text{BackProjRule}}(\rho_j^{h_i}, d)$
2. Pour chaque action $a \in A$ différente de l'action par défaut, calculer :
   $\mathsf{Rules_v}\,[Q_a] = \mathsf{Rules_v}\,[R^a] + \mathsf{Rules_v}\,[R] + \sum_i w_i \sum_j \underline{\text{BackProjRule}}(\rho_j^{h_i}, a)$
3. $\mathsf{List}\,[\pi] \leftarrow \underline{\text{DecisionList}}(d, \mathsf{Rules_v}\,[Q_d], \mathsf{Rules_v}\,[Q_a])$
4. Retourner $\mathsf{List}\,[\pi]$

FIG. 3.29 – L'algorithme Greedy$(d, \mathsf{Rules_v}\,[V])$.

Les algorithmes proposés par Guestrin et al. (2003b) que nous présentons dans ce manuscrit permettent d'aborder des problèmes pour lesquels la taille de représentation de la politique augmente de façon exponentielle avec le nombre de variables d'état du problème (quelle que soit la méthode de représentation utilisée). L'une des raisons est que ces algorithmes ne nécessitent pas une représentation explicite de la politique et ne nécessitent donc pas l'utilisation des algorithmes Greedy ou DecisionList décrits dans cette section.

3.4.4 Algorithmes

Nous avons vu section 3.4.1 que le programme linéaire utilisant une combinaison linéaire pour restreindre l'espace de recherche de la fonction de valeur se définit ainsi :

$$\begin{array}{rl} \text{Déterminer} & w_1, \ldots, w_k; \\ \text{minimisant} & \sum_s \alpha(s) \sum_i w_i h_i(s); \\ \text{et satisfaisant} & \sum_i w_i h_i(s) \geq \\ & R(s,a) + \gamma \sum_{s'} P(s'|s,a) \sum_i w_i h_i(s'), \forall s \in S, \forall a \in A \end{array} \quad (3.13)$$

Le nombre de variables à déterminer dans ce programme est donc réduit à k (plutôt que d'être égal au nombre d'états du problème). Cependant, le nombre de termes dans la somme de la fonction objectif et le nombre de contraintes sont toujours égaux au nombre d'états possibles dans le problème. Dans un premier temps, nous commençons par décrire comment la complexité de la fonction objectif est réduite. Dans un deuxième temps, nous décrirons comment le nombre de contraintes est lui aussi réduits, notamment en exploitant la structure du problème.

Fonction objectif

La fonction objectif s'écrit $\sum_s \alpha(s) \sum_i w_i h_i(s)$. Les pondérations d'intérêt $\alpha(s)$ peuvent être considérées comme une distribution sur l'espace d'état avec $\alpha(s) > 0$ et $\sum_s \alpha(s) = 1$. En réarrangeant les termes, il est possible de calculer cette distribution pour chaque fonction de base, en fonction du scope de celle-ci :

$$\begin{aligned} \sum_s \alpha(s) \sum_i w_i h_i(s) &= \sum_i w_i \sum_s \alpha(s) h_i(s) \\ &= \sum_i w_i \sum_{c_i \in C_i} \alpha(c_i) h_i(c_i) \end{aligned} \quad (3.14)$$

La nouvelle formulation du programme linéaire est donc :

$$\begin{array}{rl} \text{Déterminer} & w_1, \ldots, w_k; \\ \text{minimisant} & \sum_i w_i \sum_{c_i \in C_i} \alpha(c_i) h_i(c_i); \\ \text{et satisfaisant} & \sum_i w_i h_i(s) \geq \\ & R(s,a) + \gamma \sum_{s'} P(s'|s,a) \sum_i w_i h_i(s'), \forall s \in S, \forall a \in A \end{array} \quad (3.15)$$

Pour l'ensemble de nos expérimentations et, comme suggéré par Guestrin et al. (2003b), nous avons utilisé une distribution uniforme pour l'ensemble de l'espace d'états, donc $\alpha(s) = \frac{1}{|S|}$, et pouvant être calculé pour un contexte c donné : $\alpha(c) = \frac{1}{|C|}$.

Ensemble des contraintes

Pour résoudre la difficulté concernant le nombre de contraintes nécessaires, de Farias and Van Roy (2004) proposent une analyse de l'erreur induite dans le calcul de la solution à partir d'un échantillon de l'ensemble de contraintes. La solution proposée par Guestrin et al. (2003b) suggère plutôt de décomposer l'ensemble de contraintes afin d'exploiter les indépendances relatives aux fonctions pour diminuer le nombre de contraintes nécessaires au calcul de la solution. La technique est décrite dans la suite de cette section.

Dans un premier temps, il est nécessaire de réarranger l'écriture des contraintes du programme linéaire 3.13 :

$$\begin{aligned}
& \textstyle\sum_i w_i h_i(s) && \geq R(s,a) + \gamma \sum_{s'} P(s'|s,a) \sum_i w_i h_i(s') \\
\iff & \textstyle\sum_i w_i h_i(s) && \geq R(s,a) + \gamma \sum_i w_i g_i^a(s) \\
\iff & \quad 0 && \geq R(s,a) + \sum_i w_i [\gamma g_i^a(s) - h_i(s)] \\
\iff & \quad 0 && \geq \sum_i w_i [\gamma g_i^a(s) - h_i(s)] + \sum_j r_j^a(s)
\end{aligned}$$

avec $g_i^a(s) = \sum_{s'} P(s'|s,a) h_i(s')$ (section 3.4.3, équation 3.8) et en supposant que la fonction de récompense se décompose sous la forme factorisée $R(s,a) = \sum_j r_j^a(s)$ (section 3.4.1, équation 3.4) dont le scope de chaque fonction r_j est limité à un petit ensemble de variables. Le programme linéaire devient alors :

$$\begin{aligned}
\text{Déterminer} \quad & w_1, \ldots, w_k; \\
\text{minimisant} \quad & \sum_i w_i \sum_{c_i \in C_i} \alpha(c_i) h_i(c_i); \\
\text{et satisfaisant} \quad & 0 \geq \sum_i w_i [\gamma g_i^a(s) - h_i(s)] + \sum_j r_j^a(s), \forall s \in S, \forall a \in A
\end{aligned} \tag{3.16}$$

L'ensemble des contraintes de ce programme peut ensuite être remplacé par une seule contrainte non linéaire par action :

$$\begin{aligned}
\text{Déterminer} \quad & w_1, \ldots, w_k; \\
\text{minimisant} \quad & \sum_i w_i \sum_{c_i \in C_i} \alpha(c_i) h_i(c_i); \\
\text{et satisfaisant} \quad & 0 \geq \max_s \{\sum_i w_i [\gamma g_i^a(s) - h_i(s)] + \sum_j r_j^a(s)\}, \forall a \in A
\end{aligned} \tag{3.17}$$

Afin de pouvoir résoudre ce programme linéaire, Guestrin et al. (2003b) proposent une méthode pour générer un nouvel ensemble équivalent de contraintes linéaires à partir d'une contrainte non

linéaire de la forme générale : $\phi \geq \max_x \sum_i w_i b_i(x) - \sum_j c_j(x)$ avec w_i les variables libres à déterminer. Les fonctions b_i sont appelées fonctions de base, les fonctions c_j sont appelées fonctions cibles. La méthode s'appuie sur des techniques d'élimination de variables dans les réseaux bayésiens et utilise la structure du FMDP via les indépendances relatives aux fonctions et la décomposition additive des fonctions. Elle est décrite figure 3.30.

Entrée(s) : $\mathcal{B} = \{b_0, \dots, b_k\}, \mathcal{C} = \{c_0, \dots, c_k\}$, un ordre d'élimination O **Sortie(s) :** l'ensemble de contraintes Ω

---

1. Soit $\mathcal{F} = \{\}$
2. Pour tout $b_i \in \mathcal{B}$:
   (a) Pour tout contexte $z \in$ Scope(b_i) :
      i. Créer une nouvelle variable $u_z^{f_i}$
      ii. $\Omega = \Omega \cup \{u_z^{f_i} = w_i b_i(z)\}$
      iii. Définir $f_i(z) = u_z^{f_i}$
   (b) $\mathcal{F} = \mathcal{F} \cup \{f_i\}$
3. Pour tout $c_i \in \mathcal{C}$:
   (a) Pour tout contexte $z \in$ Scope(c_i) :
      i. Créer une nouvelle variable $u_z^{f_i}$
      ii. $\Omega = \Omega \cup \{u_z^{f_i} = c_i(z)\}$
      iii. Définir $f_i(z) = u_z^{f_i}$
   (b) $\mathcal{F} = \mathcal{F} \cup \{f_i\}$
4. Pour toute variable X_l de la liste ordonnée O :
   (a) Construire $\{e_0, \dots, e_L\}$ tel que $e_i \in \mathcal{F}$ et $X_l \in$ Scope(e_i).
   (b) $e \leftarrow$ RuleMax$(\sum_{i=0}^{L} e_i, X_l)$
   (c) Pour tout $z \in$ Scope(e) : $\Omega = \Omega \cup \{u_z^e \geq \sum_{j=1}^{L} u_{(z,x_l)}^{e_j}\}, \forall x_l \in$ Dom(X_l)
   (d) $\mathcal{F} = \mathcal{F} \cup e \setminus \{e_0, \dots, e_L\}$
5. $\Omega = \Omega \cup \{\phi \geq \sum_{e_i \in \mathcal{F}} e_i\}$
6. Retourner Ω

FIG. 3.30 – L'algorithme FactoredLP$(\mathcal{C}, \mathcal{B}, O)$ générant un ensemble de contraintes linéaires équivalent à $\phi \geq \max_x \sum_i w_i c_i(x) - \sum_j b_j(x)$.

L'algorithme FactoredLP$(\mathcal{B}, \mathcal{C}, O)$ est principalement constitué de deux étapes. En premier lieu, les fonctions présentes dans les ensembles $\mathcal{B}$ et $\mathcal{C}$ sont ramenées à une représentation commune de la forme $f_i(s) = \sum_z u_z^{f_i}(s)$ et ajoutées dans un ensemble $\mathcal{F}$. Lors de la deuxième étape, étant donné

un ordre de variables O, les variables sont éliminées en remplaçant dans $\mathcal{F}$ les fonctions possédant la variable en train d'être éliminée dans leur scope par une fonction représentant le maximum.

Il est maintenant possible de définir l'algorithme FactoredLPA qui, à partir de représentations factorisées des fonctions de transition et de récompenses, d'un ensemble de fonctions de base et d'un ordre d'élimination des variables, permet de calculer une approximation de la fonction de valeur optimale d'un FMDP. FactoredLPA est décrit figure 3.31.

Entrée(s) : $\mathsf{Rules_p}[P], \mathsf{Rules_v}[R], H = \{\mathsf{Rules_v}[h_0], \ldots, \mathsf{Rules_v}[h_k]\}, O$ **Sortie(s) :** $\{w_0, \ldots, w_k\}$, $\{\forall a \in A, \forall h_i \in H : g_i^a\}$

---

1. Pour tout $a \in A$: pour tout $h_i \in H$: $g_i^a \leftarrow \underline{\text{BackProjRule}}_a(\mathsf{Rules_v}[h_i])$
2. Pour tout $h_i \in H$: $\alpha_i \leftarrow \sum_{c_i \in C_i} \alpha(c_i) h_i(c_i)$
3. Soit $\Omega = \{\}$
4. Pour tout $a \in A$: $\Omega = \Omega \cup \underline{\text{FactoredLP}}(\{\gamma g_0^a - h_0, \ldots, \gamma g_k^a - h_k\}, R^a, O)$
5. $\Omega = \Omega \cup \{\phi = 0\}$
6. Résoudre le programme linéaire suivant : Déterminer $\{w_0, \ldots, w_k\}$ minimisant $\sum_i w_i \alpha_i$ et satisfaisant Ω
7. Retourner $\{w_0, \ldots, w_k\}$

FIG. 3.31 – L'algorithme FactoredLPA calculant une approximation de la fonction de valeur optimale dans un FMDP.

L'algorithme FactoredLPA commence par calculer l'ensemble des fonctions g_i^a puis des α_i pour chaque fonctions de base et chaque action. Ensuite, l'ensemble des contraintes pour chaque action du problème est généré via l'algorithme FactoredLP. Enfin, le programme linéaire permet d'obtenir la valeur des coefficients $\{w_0, \ldots, w_k\}$ composant une représentation approchée de la fonction de valeur optimale du FMDP.

3.5 Synthèse

Nous avons tout d'abord présenté le cadre mathématique des FMDPs. Nous avons vu que ce cadre permettait d'exploiter différentes propriétés de la structure des problèmes de grande taille pour pouvoir les représenter, puis les résoudre. Les deux principales propriétés exploitées sont la décomposition multiplicative de la fonction de transition et l'indépendance relative aux fonctions du problème. Cette dernière est mise en évidence par l'utilisation de DBNs pour représenter les fonctions de transition et de récompense. Suivant les algorithmes de planification utilisés, les deux

autres propriétés exploitées pour la représentation et la résolution des problèmes sont les indépendances relatives aux contextes et une approximation additive de la fonction de valeur. Une fois le cadre mathématique décrit, nous avons décrit trois familles d'algorithme de planification. Toutes utilisent la décomposition multiplicative de la fonction de transition et l'indépendance relative aux fonctions.

Deux d'entre elles, regroupant les algorithmes SPI et SVI d'une part, et l'algorithme SPUDD d'autre part, s'appuient sur la programmation dynamique. Elles utilisent des représentations structurées, telles que les arbres de décision et les ADDs, pour exploiter les indépendances relatives aux contextes et représenter de façon compacte les fonctions de certains problèmes.

La troisième famille s'appuie sur la programmation linéaire pour résoudre le problème. Elle utilise la décomposition additive de la fonction de récompense d'une part et, une représentation approchée de la fonction de valeur optimale du problème d'autre part. Cette approximation est une combinaison linéaire de fonctions de base ne dépendant que d'un petit nombre de variables. Les indépendances relatives aux fonctions et, éventuellement, les indépendances relatives aux contextes lorsqu'une représentation structurée est utilisée, sont ensuite exploitées pour diminuer le nombre de contraintes du programme linéaire.

Chapitre 4

Apprentissage hors-ligne d'un FMDP

Dans le cadre des FMDPs, les algorithmes de planification permettent de trouver des solutions à des problèmes de grande taille en exploitant la structure du problème connue a priori. Dès lors, ces méthodes ne sont pas directement utilisables sur un problème d'apprentissage par renforcement pour lequel cette structure est inconnue. Afin d'élargir le cadre d'application de ces algorithmes, plus particulièrement de pouvoir les exploiter sur l'ensemble des problèmes d'apprentissage par renforcement discrets et satisfaisant l'hypothèse de Markov, ce chapitre présente deux nouvelles méthodes d'apprentissage permettant, à partir d'un échantillon d'observations de l'agent dans son environnement, de construire un FMDP représentant le problème d'apprentissage par renforcement à résoudre.

Plus précisément, ces deux méthodes construisent des représentations factorisées des fonctions de transition et de récompense d'un FMDP représentant le problème, en exploitant à la fois les indépendances relatives aux fonctions et les indépendances relatives aux contextes. Une fois le FMDP construit, les algorithmes de planification peuvent être utilisés afin de calculer une solution au problème. Concernant l'algorithme de planification présenté section 3.4 et utilisant la programmation linéaire, nous supposerons que les fonctions de base utilisées pour approcher la fonction de valeur et la décomposition additive de la fonction de récompense sont connues a priori.

Les méthodes et les résultats que nous présentons dans ce chapitre concernent un apprentissage hors-ligne : notre but est de construire un FMDP à partir seulement d'un échantillon d'observations d'un agent quelconque dans l'environnement. Bien qu'un tel apprentissage se révèle souvent utile en pratique, notre objectif principal reste l'apprentissage en ligne de la structure des FMDPs que nous présenterons au chapitre 5.

Dans un premier temps, nous expliquons dans la section 4.1 comment la construction d'un FMDP se ramène à un problème d'apprentissage supervisé. Puis nous décrivons une famille d'algorithmes d'apprentissage supervisé adaptés à un tel apprentissage : l'induction d'arbres de décision. Ensuite, la section 4.2 présente l'utilisation de tels algorithmes pour construire un FMDP complet puis l'intégration de cet apprentissage avec les méthodes de planification décrites lors du chapitre

précédent. Enfin, la section 4.3 présente les résultats obtenus par les deux méthodes d'apprentissage que nous proposons sur les problèmes classiques de la littérature des FMDPs.

4.1 Apprentissage supervisé d'ensembles d'exemples

Pour pouvoir réutiliser les techniques de planification des FMDPs, il est nécessaire de construire une représentation des fonctions de transition et de récompense conforme au formalisme. Nous rappelons que la fonction de transition est une fonction permettant d'obtenir les distributions de probabilités $P(s'|s, a)$. Or, à chaque expérience de l'agent dans son environnement, celui-ci observe un état s dans lequel il était, l'action a qu'il a réalisée et un nouvel état s' dans lequel il se trouve au pas de temps courant. À partir de cette observation $\langle s, a, s' \rangle$, il est donc possible d'apprendre la fonction de transition $T(s, a)$ en utilisant des techniques d'apprentissage supervisé. De même, la fonction de récompense est une fonction $R : S \times A \to \mathbb{R}$. À chaque expérience de l'agent dans son environnement, celui-ci observe un état s, son action a et, la récompense r qu'il obtient. Ainsi, à partir de l'observation $\langle s, a, r \rangle$ et en utilisant des techniques d'apprentissage supervisé, il est possible d'apprendre la fonction de récompense $R(s, a)$.

Afin de pouvoir être utilisée dans le cadre de l'apprentissage de la structure d'un FMDP, il est nécessaire que la méthode d'apprentissage soit capable de déterminer de façon automatique les indépendances relatives aux fonctions du problème. De plus, afin de pouvoir s'intégrer avec les algorithmes de planification décrits dans le chapitre 3, la méthode d'apprentissage doit aussi être capable de construire des représentations structurées pour déterminer de façon automatique les indépendances relatives aux contextes. Dans le domaine de l'apprentissage supervisé, une famille d'algorithmes satisfaisant l'ensemble de ces contraintes est constituée par les algorithmes d'induction d'arbres de décision.

La suite de cette section introduit d'une façon générale l'induction d'arbres de décision dans le cadre de l'apprentissage supervisé, indépendamment de l'apprentissage des FMDPs. L'approche que nous proposons pour l'apprentissage des FMDPs et l'intégration de ces méthodes avec les algorithmes de planification est décrite dans la section 4.2 (page 86).

4.1.1 Induction d'arbres de décision

L'objectif d'un algorithme d'induction d'arbres de décision est de construire une représentation structurée $\mathsf{Tree}[F]$ d'une fonction $F : \mathcal{A} \to \zeta$ à partir d'un ensemble d'exemples $\mathcal{E} = \{\langle \mathfrak{a}_0, \varsigma_0 \rangle, \ldots, \langle \mathfrak{a}_n, \varsigma_n \rangle\}$ avec $\mathfrak{a}_i \in \mathcal{A}$ et $\varsigma_i \in \zeta$. Une instance $\mathfrak{a} \in \mathcal{A}$ est composée d'un ensemble d'attributs $\{\mathcal{V}_0, \ldots, \mathcal{V}_n\}$. On note $\mathfrak{a}[\mathcal{V}_i]$ la valeur de l'attribut $\mathcal{V}_i$ dans l'exemple $\langle \mathfrak{a}, \varsigma \rangle$ avec $\mathfrak{a}[\mathcal{V}_i] \in \underline{\mathrm{Dom}}(\mathcal{V}_i)$ et $\underline{\mathrm{Dom}}(\mathcal{V}_i)$ un ensemble fini d'éléments. Lorsque ζ est un ensemble fini et discret, l'apprentissage de la fonction F est un problème de *classification*, lorsque ζ est l'ensemble $\mathbb{R}$,

l'apprentissage de F est un problème de *régression*.

Le principe de l'apprentissage d'arbres de décision est de découper l'espace en plusieurs partitions, chacune représentant un sous-problème de la fonction à apprendre. Dans le cadre de l'apprentissage supervisé, nous présentons dans cette section les algorithmes de construction d'arbres (Breiman et al., 1984) utilisés aussi bien pour la classification, tels que ID3 (Quinlan, 1983, 1986) ou C4.5 (Quinlan, 1993), que pour la régression, tels que les arbres de régression des moindres carrés (Breiman and Breiman, 1984). À partir d'un ensemble d'exemples, l'algorithme BuildTree, décrit figure 4.1, construit de façon récursive un arbre de décision.

Entrée(s) : $\mathcal{E} = \{\langle \mathfrak{a}_0, \varsigma_0 \rangle, \ldots, \langle \mathfrak{a}_n, \varsigma_n \rangle\}$, une mesure d'information $\mathcal{M}$, un nœud k de Tree $[F]$ **Sortie(s) :** Tree $[F]$

Si tous les exemples $\langle \mathfrak{a}_i, \varsigma_i \rangle \in \mathcal{E}$ pointent sur la même valeur ς $(\forall i, 0 \leq i \leq n, \varsigma_i = \varsigma)$:

Alors : transformer k en une feuille contenant ς

Sinon :

1. Soit $\mathcal{V}_i \leftarrow$ SelectAttr$(\mathcal{M}, \mathcal{E})$
2. Transformer k en un nœud de décision testant $\mathcal{V}_i$
3. $\forall \nu \in$ Dom$(\mathcal{V}_i)$: $\mathcal{E}_\nu \leftarrow \{e = \langle \mathfrak{a}, \varsigma \rangle | e \in \mathcal{E}$ et $\mathfrak{a}[\mathcal{V}_i] = \nu\}$
4. $\forall \nu \in$ Dom$(\mathcal{V}_i)$: BuildTree$(\mathcal{E}_\nu, k_\nu)$, avec k_ν le nœud enfant de k correspondant à $\mathcal{V}_i = \nu$

FIG. 4.1 – L'algorithme BuildTree construisant un arbre à partir d'un ensemble d'exemples $\mathcal{E}$.

L'algorithme BuildTree commence par vérifier si les exemples appartenant à l'ensemble $\mathcal{E}$ d'exemples pointent tous vers la même valeur ς. Dans ce cas, il n'est pas nécessaire de continuer la construction de l'arbre et une feuille contenant cette valeur est installée. Dans le cas contraire, avec l'opérateur SelectAttr, on utilise une mesure d'information $\mathcal{M}$ (nous décrirons les mesures d'information dans les sections 4.1.2 et 4.1.3 suivantes) pour sélectionner le nouvel attribut $\mathcal{V}_i$ testé au nœud de décision dans l'arbre (étape 1). Une fois le nœud de décision installé, le test de l'attribut $\mathcal{V}_i$ partitionne l'ensemble $\mathcal{E}$ des exemples en plusieurs sous-ensembles $\mathcal{E}_\nu$ d'exemples, chacun correspondant à une valeur de $\mathcal{V}_i$ différente (étape 3). L'algorithme est donc ensuite appelé récursivement pour chaque branche $\mathcal{V}_i = \nu$ avec le sous-ensemble $\mathcal{E}_\nu$ d'exemples correspondant (étape 4).

L'algorithme BuildTree développe l'arbre de décision jusqu'à ce que chaque feuille soit pure, c'est-à-dire jusqu'à ce que les exemples appartenant à l'ensemble $\mathcal{E}$ à apprendre pointent tous sur la même valeur. Ce comportement peut poser un problème de sur-apprentissage, notamment lorsque l'échantillon d'exemples contient des erreurs ou du bruit. Pour éviter un tel problème, deux techniques peuvent être utilisées : le pré-élagage et le post-élagage.

La première solution, le pré-élagage, consiste à cesser l'installation de nouveaux tests lorsque

la pureté des exemples à apprendre est considérée comme suffisante (par exemple, lorsque les exemples contenus dans l'ensemble d'exemples pointent de façon majoritaire sur une même valeur). Les avantages de cette méthode sont, premièrement, qu'elle est simple à mettre en œuvre et, deuxièmement peu coûteuse en temps de calcul. Son principal inconvénient est qu'elle est "myope", c'est-à-dire qu'elle ne prend en compte qu'un critère local calculé à partir des exemples d'une feuille. Par conséquent, elle peut empêcher le développement d'une branche (d'une profondeur strictement supérieure à 1) qui serait nécessaire.

La deuxième solution, le post-élagage, ou *pruning*, consiste tout d'abord à construire l'arbre de décision complètement puis, dans un deuxième temps, chercher à le simplifier. Un critère de qualité est alors utilisé afin d'évaluer le compromis entre la taille de l'arbre et l'erreur commise. Bien que cette méthode soit à la fois plus efficace et plus valide d'un point de vue théorique, elle nécessite un développement complet de l'arbre et donc est coûteuse en temps de calcul et en mémoire.

Enfin, l'algorithme BuildTree nécessite une mesure d'information $\mathcal{M}$ afin de sélectionner, via la fonction SelectAttr, l'attribut testé à un nœud de décision lors de son installation. Dans le cadre d'induction d'arbres de décision pour la classification, une mesure d'information pour les valeurs symboliques est utilisée. Dans le cadre d'induction d'arbres de régression, la mesure d'information pour les valeurs réelles est utilisée. Ces deux types de mesure d'information sont décrits respectivement dans les sections 4.1.2 et 4.1.3.

4.1.2 Mesure d'information pour des valeurs symboliques

Nous traitons tout d'abord le cas où l'espace de sortie ζ de la fonction $F : \mathcal{A} \to \zeta$ à apprendre est discret et fini. Dans le cadre de l'apprentissage supervisé, ce problème peut se ramener à un problème de classification. Un grand nombre de mesures ont été proposées afin de sélectionner le meilleur attribut à tester dans un nœud de décision. Nous pouvons notamment citer *gain* et *gain ratio* (Quinlan, 1993), *l'information mutuelle* ou *l'entropie croisée* (Cover, 1991), *Kolmogorov-Smirnoff* (Friedman, 1977) ou encore le critère du χ^2 (Quinlan, 1986). Nous avons choisi d'utiliser ce dernier principalement parce qu'il permet d'effectuer un test statistique afin de différencier deux distributions de probabilités (Saporta, 1990). Ainsi, comme cela a été suggéré par Quinlan (1986) et Utgoff (1986), ce test peut être utilisé également dans le cadre d'un pré-élagage.

Calcul du critère

Pour un ensemble d'exemples $\mathcal{E}$ et pour un attribut $\mathcal{V}$, le critère du χ^2, noté $\mathcal{M}_{\chi^2}$ dans les algorithmes, se calcule de la façon suivante :

$$\chi^2 = \sum_{\varsigma \in \zeta} \sum_{\nu \in \underline{\mathrm{Dom}}(\mathcal{V})} \frac{\left(n_{\mathcal{E}}^{\varsigma,\nu} - n_{\mathcal{E}}^{\nu} \cdot \frac{n_{\mathcal{E}}^{\varsigma}}{n_{\mathcal{E}}}\right)^2}{n_{\mathcal{E}}^{\nu} \cdot \frac{n_{\mathcal{E}}^{\varsigma}}{n_{\mathcal{E}}}} \tag{4.1}$$

avec $n_\mathcal{E}$ le nombre d'exemples dans $\mathcal{E}$, $n_\mathcal{E}^\varsigma$ le nombre d'exemples dans $\mathcal{E}$ ayant pour valeur ς, $n_\mathcal{E}^\nu$ le nombre d'exemples dans $\mathcal{E}$ dont l'attribut $\mathcal{V} = \nu$ et $n_\mathcal{E}^{\varsigma,\nu}$ le nombre d'exemples dans $\mathcal{E}$ ayant pour valeur ς et dont l'attribut $\mathcal{V} = \nu$.

Ainsi, ce critère est utilisé par l'algorithme de construction d'arbres BuildTree en tant que mesure d'information (fonction SelectAttr) en sélectionnant l'attribut $\mathcal{V}$ pour lequel la valeur χ^2 est la plus grande (sélectionnant ainsi l'attribut séparant les distributions pour lesquelles la probabilité qu'elles soient différentes est la plus grande).

Étude du critère lorsque le nombre d'exemples augmentent

Pour un ensemble $\mathcal{E}$ donné de $n_\mathcal{E}$ exemples, il est possible de considérer une interprétation probabiliste du calcul de χ^2. Nous pouvons calculer la probabilité que la fonction F soit égale à ς :

$$\hat{P}(\varsigma) = \frac{n_\mathcal{E}^\varsigma}{n_\mathcal{E}} \tag{4.2}$$

De même, nous pouvons calculer la probabilité que la fonction F soit égale à ς lorsqu'un attribut $\mathcal{V}$ est égal à ν :

$$\hat{P}(\varsigma|\mathcal{V} = \nu) = \frac{n_\mathcal{E}^{\varsigma,\nu}}{n_\mathcal{E}^\nu} \tag{4.3}$$

Il est alors possible de réécrire le critère du χ^2 en termes probabilistes (Saporta, 1990) :

$$\begin{aligned}
\chi^2 &= \sum_{\varsigma\in\zeta} \sum_{\nu\in\underline{\text{Dom}}(\mathcal{V})} \frac{\left(n_\mathcal{E}^{\varsigma,\nu} - n_\mathcal{E}^\nu \cdot \frac{n_\mathcal{E}^\varsigma}{n_\mathcal{E}}\right)^2}{n_\mathcal{E}^\nu \cdot \frac{n_\mathcal{E}^\varsigma}{n_\mathcal{E}}} \\
&= \sum_{\varsigma\in\zeta} \sum_{\nu\in\underline{\text{Dom}}(\mathcal{V})} \frac{\left(n_\mathcal{E}^\nu \cdot \frac{n_\mathcal{E}^{\varsigma,\nu}}{n_\mathcal{E}^\nu} - n_\mathcal{E}^\nu \cdot \frac{n_\mathcal{E}^\varsigma}{n_\mathcal{E}}\right)^2}{n_\mathcal{E}^\nu \cdot \frac{n_\mathcal{E}^\varsigma}{n_\mathcal{E}}} \\
&= \sum_{\varsigma\in\zeta} \sum_{\nu\in\underline{\text{Dom}}(\mathcal{V})} \frac{\left(n_\mathcal{E}^\nu \cdot \hat{P}(\varsigma|\mathcal{V} = \nu) - n_\mathcal{E}^\nu \cdot \hat{P}(\varsigma)\right)^2}{n_\mathcal{E}^\nu \cdot \hat{P}(\varsigma)} \\
&= \sum_{\varsigma\in\zeta} \sum_{\nu\in\underline{\text{Dom}}(\mathcal{V})} n_\mathcal{E}^\nu \frac{\left(\hat{P}(\varsigma|\mathcal{V} = \nu) - \hat{P}(\varsigma)\right)^2}{\hat{P}(\varsigma)}
\end{aligned} \tag{4.4}$$

Nous pouvons remarquer une propriété intéressante du critère χ^2 lorsque le nombre d'exemples $n_\mathcal{E}^\nu$ augmente. En effet, pour deux probabilités $\hat{P}(\varsigma)$ et $\hat{P}(\varsigma|\mathcal{V} = \nu)$ données, nous avons le terme $\frac{\left(\hat{P}(\varsigma|\mathcal{V}=\nu) - \hat{P}(\varsigma)\right)^2}{\hat{P}(\varsigma)}$ égale à une constante strictement positive lorsque $\hat{P}(\varsigma) \neq \hat{P}(\varsigma|\mathcal{V} = \nu)$ et, égale 0 lorsque $\hat{P}(\varsigma) = \hat{P}(\varsigma|\mathcal{V} = \nu)$. Nous pouvons alors en déduire :

$$\lim_{n_\mathcal{E}^\nu \to +\infty} \chi^2 = \begin{cases} +\infty & \text{si } \hat{P}(\varsigma) \neq \hat{P}(\varsigma|\mathcal{V} = \nu) \\ 0 & \text{si } \hat{P}(\varsigma) = \hat{P}(\varsigma|\mathcal{V} = \nu) \end{cases} \tag{4.5}$$

Nous verrons section 4.3 que cette limite offre des propriétés intéressantes lors de l'apprentissage.

4.1.3 Mesure d'information pour des valeurs réelles

Nous traitons maintenant le cas où la fonction F à apprendre est une fonction $F : \mathcal{A} \rightarrow \mathbb{R}$. Bien que l'espace de sortie de la fonction soit continu, l'espace d'entrée $\mathcal{A}$ reste quant à lui discret et fini. Dans le cadre de l'apprentissage supervisé, ce problème se ramène à un problème de régression. Principalement, deux critères ont été proposés : le critère des moindres carrés et le critère de la moindre déviation absolue (Breiman and Breiman, 1984; Breiman et al., 1984; Torgo, 2000). Nous nous sommes intéressé plus particulièrement au premier critère, le deuxième étant moins sensible aux erreurs dans l'échantillon mais plus coûteux en temps de calcul.

Pour décrire la façon dont se calcule le critère des moindres carrés, noté $\mathcal{M}_{LS}$ dans les algorithmes, nous commençons par définir la valeur $k_{\mathcal{E}}$ représentant la moyenne d'un ensemble $\mathcal{E}$ d'exemples $\langle \mathfrak{a}, \varsigma \rangle$ avec $\varsigma \in \mathbb{R}$:

$$k_{\mathcal{E}} = \frac{1}{n_{\mathcal{E}}} \cdot \sum_{\varsigma \in \mathcal{E}} \varsigma \tag{4.6}$$

où $n_{\mathcal{E}}$ est le nombre d'exemples dans $\mathcal{E}$. On calcule maintenant la variance, appelée *erreur de régression*, ou *fitting error* et, notée $E_{\mathcal{E}}$, associée à l'ensemble d'exemples :

$$E_{\mathcal{E}} = \frac{1}{n_{\mathcal{E}}} \cdot \sum_{\varsigma \in \mathcal{E}} (\varsigma - k_{\mathcal{E}})^2 \tag{4.7}$$

On peut maintenant définir l'erreur $E_{\mathcal{E}}^{\mathcal{V}}$ associée à un test sur un attribut $\mathcal{V}$:

$$E_{\mathcal{E}}^{\mathcal{V}} = \sum_{\nu \in \mathcal{V}} \frac{n_{\mathcal{E}}^{\nu}}{n_{\mathcal{E}}} \cdot E_{\mathcal{E}_{\mathcal{V}=\nu}} \tag{4.8}$$

avec $n_{\mathcal{E}}^{\nu}$ représentant le nombre d'exemples avec $\mathcal{V} = \nu$ et $E_{\mathcal{E}_{\mathcal{V}=\nu}}$ l'erreur de régression définie par l'équation 4.7 et calculée sur l'ensemble $\mathcal{E}_{\mathcal{V}=\nu}$ regroupant les exemples dont $\mathcal{V} = \nu$.

Enfin, pour pouvoir utiliser ce critère avec l'algorithme de construction d'arbres BuildTree en tant que mesure d'information (fonction SelectAttr), on sélectionne l'attribut $\mathcal{V}$ de la façon suivante :

$$\underline{\text{SelectAttr}}(\mathcal{E}) = \arg\max_{\mathcal{V}} \left[E_{\mathcal{E}} - E_{\mathcal{E}}^{\mathcal{V}} \right] \tag{4.9}$$

A un nœud de décision, l'attribut $\mathcal{V}$ sélectionné est donc celui pour lequel la différence entre son erreur de régression $E_{\mathcal{E}}^{\mathcal{V}}$ et l'erreur de régression $E_{\mathcal{E}}$ de l'ensemble $\mathcal{E}$ est maximum.

4.2 Construction d'un FMDP et intégration des algorithmes de planification

À partir de l'expérience de l'agent dans son environnement, nous avons vu qu'il était envisageable d'apprendre les fonctions de transition et de récompense du problème. Nous proposons

maintenant une approche simple et originale pour utiliser l'apprentissage supervisé. Cette approche utilise l'induction d'arbres de décision, présentée lors de la section précédente, pour construire un FMDP permettant de représenter un problème d'apprentissage par renforcement dont les fonctions de transition et de récompense sont inconnues a priori. Une fois que le problème d'apprentissage par renforcement est représenté dans ce formalisme, les techniques de planification décrites au chapitre 3 peuvent être utilisées afin de calculer une solution au problème.

L'utilisation d'arbres de décisions comme représentation structurée pour les fonctions de transition et de récompense du FMDP permet à la fois d'exploiter les indépendances relatives aux contextes et d'en déduire de façon implicite les indépendances relatives aux fonctions.

En effet, comme le montre la figure 4.2, les ensembles $\underline{\text{Parents}}_a(X_i)$ et $\underline{\text{Scope}}(R_i)$ s'obtiennent, respectivement à partir d'une distribution de probabilités conditionnelle $\text{Tree}\,[P_a(X_i|s)]$ et d'une fonction localisée $\text{Tree}\,[R_i]$, en construisant l'ensemble des variables X_i testées dans au moins un nœud de décision de ces arbres. Par conséquent, il est possible d'obtenir facilement les DBNs pour chaque distribution de probabilités conditionnelle $P_a(X_i|s)$ et pour chaque fonction de récompense localisée R_i. Les arbres de décision sont donc une représentation structurée extrêmement intéressante dans le cadre de l'apprentissage de FMDPs.

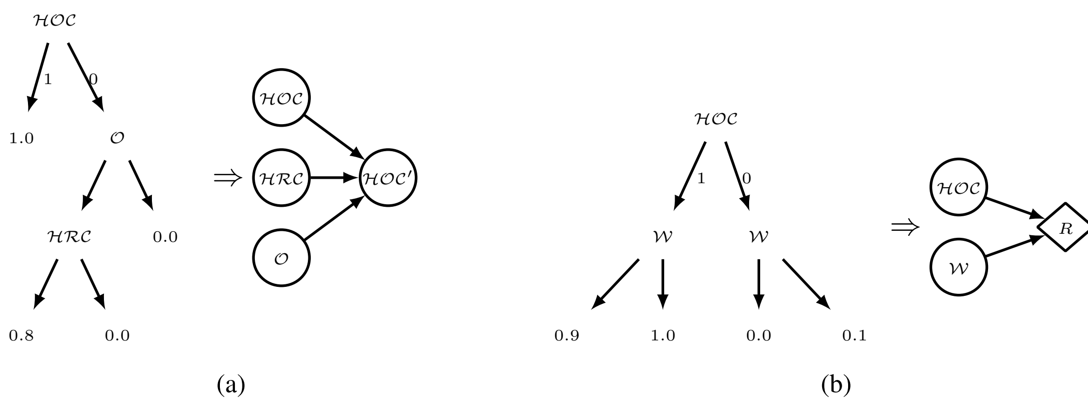


FIG. 4.2 – Obtention de $\underline{\text{Parents}}_{\mathcal{D}\text{elC}}(\mathcal{HOC})$ à partir de $\text{Tree}\,[P_{\mathcal{D}\text{elC}}(\mathcal{HOC}|s)]$ (figure a) et de $\underline{\text{Scope}}(R)$ à partir de $\text{Tree}\,[R]$ (figure b) dans le problème *Coffee Robot*.

Nous commençons par expliquer section 4.2.1 comment un échantillon d'observations peut être décomposé en ensembles d'exemples utilisables par des algorithmes d'induction d'arbres de décision. Nous proposons ensuite, section 4.2.2, deux algorithmes pour construire un FMDP à partir des algorithmes d'induction d'arbres de décision. Enfin, nous montrons section 4.2.3 comment ces algorithmes s'intègrent avec les méthodes de planification dans les FMDPs. Dans un autre registre, la section 4.2.4 montre comment l'induction d'arbres de décision peut être utilisée pour réorganiser des fonctions représentées par une partition de contextes.

4.2.1 Décomposition des observations en ensembles d'exemples

Notre but est, à partir d'un échantillon d'observations, de construire un FMDP sans supposer a priori qu'elle est sa structure. Il est donc nécessaire de construire la représentation de deux fonctions différentes du FMDP : sa fonction de transition et sa fonctions de récompense.

Nous commençons tout d'abord par expliquer comment nous décomposons un échantillon d'observations en plusieurs ensembles d'exemples et les adaptations nécessaires de l'algorithme d'induction d'arbres de décision BuildTree afin de construire une représentation de la fonction de transition. Dans un deuxième temps, nous décrirons cette décomposition pour l'apprentissage de la fonction de récompense.

Ensembles d'exemples pour l'apprentissage de la fonction de transition

A chaque essai de l'agent dans son environnement, l'agent peut observer une transition $\langle s, a, s' \rangle$. Nous supposons qu'un état s est décrit par un ensemble de variables aléatoires $\{X_1, \ldots, X_n\}$ et donc que la transition observée représente une transition $\langle \{x_1, \ldots, x_n\}, a, \{x'_1, \ldots, x'_n\} \rangle$ entre l'instanciation $\{x_1, \ldots, x_n\}$ de ces variables à l'instant t et l'instanciation $\{x'_1, \ldots, x'_n\}$ à l'instant $t + 1$.

Nous rappelons que la fonction à apprendre est, pour chaque variable aléatoire X_i composant l'espace d'état, une distribution de probabilités conditionnelle $P_a(X'_i|s)$. Or, cette distribution de probabilités peut être représentée par une fonction stochastique $F^a_{X_i}$ de S vers Dom(X_i). Le problème se ramène donc à un problème proche de celui de la classification : une représentation $\mathsf{Tree}[F]$ peut être construite à partir d'un ensemble d'exemples $\langle \mathfrak{a} = \{x_1, \ldots, x_n\}, \varsigma = x'_i \rangle$ extrait à partir des transitions observées par l'agent lorsqu'il a exécuté l'action a. Les attributs de l'exemple sont constitués par l'état s à l'instant t de l'agent. La valeur de l'exemple correspond à la valeur x'_i de la variable X_i dans l'état s'. Une deuxième décomposition consiste à considérer l'action exécutée par l'agent comme un attribut de l'exemple à apprendre, au même titre que les variables aléatoires X_i composant l'espace d'état. Dans ce cas, la fonction à apprendre est aussi une fonction stochastique F_{X_i} de $S \times A$ vers Dom(X_i).

Utilisation de l'algorithme de construction d'arbres de décision

Nous pouvons tout d'abord envisager d'utiliser l'algorithme d'induction d'arbres de décision BuildTree (figure 4.1, page 83) pour apprendre les fonctions stochastiques $F^a_{X_i}$ et F_{X_i} représentant les distributions de probabilités conditionnelles de la fonction de transition. Cependant, nous rappelons que l'algorithme BuildTree n'arrête le développement de l'arbre que lorsque l'ensemble des exemples associés à une feuille est pur. Comme le montre un exemple figure 4.3, ce comportement n'est absolument pas adapté à l'apprentissage d'une fonction F stochastique puisque, bien que les

exemples de l'ensemble d'exemples proviennent de la même distribution de probabilités conditionnelle, le fait que ceux-ci contiennent des valeurs différentes implique l'installation de nœuds de décision supplémentaires, jusqu'à ce que tous les attributs soient testés.

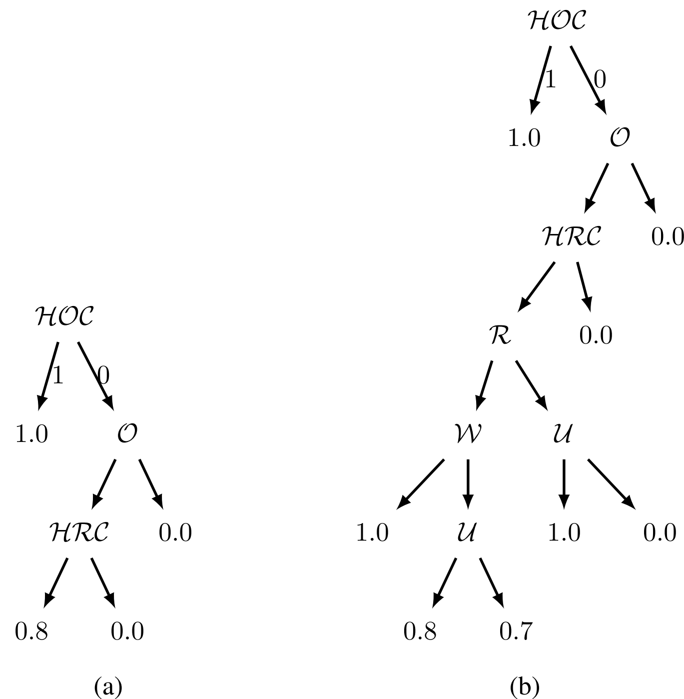


FIG. 4.3 – L'apprentissage de la distribution de probabilités conditionnelle $P_{\mathcal{D}\text{elC}}(\mathcal{HOC}'|s)$ (figure a) par l'algorithme BuildTree a pour conséquence le développement inutile de branches pour les distributions non déterministes (figure b), comme c'est le cas dans cet exemple pour la feuille $P_{\mathcal{D}\text{elC}}(\mathcal{HOC}'|\mathcal{O} = 1, \mathcal{HRC} = 1, \mathcal{HOC} = 0) = 0.8$.

La figure 4.3 illustre un résultat possible (figure 4.3(b)) de l'algorithme BuildTree après l'apprentissage d'un échantillon d'exemples calculés à partir de la définition de la distribution de probabilités conditionnelle $P_{\mathcal{D}\text{elC}}(\mathcal{HOC}'|s)$, représenté figure 4.3(a). Les feuilles pour lesquelles $P_{\mathcal{D}\text{elC}}(\mathcal{HOC}'|s) = 0$ ou $P_{\mathcal{D}\text{elC}}(\mathcal{HOC}'|s) = 1$ ne posent pas de problèmes puisqu'elles sont déterministes. Cependant, pour les probabilités différentes de 0 ou 1, comme par exemple la feuille $P_{\mathcal{D}\text{elC}}(\mathcal{HOC}'|\mathcal{O} = 1, \mathcal{HRC} = 1, \mathcal{HOC} = 0) = 0.8$, l'algorithme installe des nœuds de décision inutiles testant les variables $\mathcal{W}$, $\mathcal{U}$ et $\mathcal{R}$ jusqu'à ce que tous les exemples appartiennent à la même valeur, ou bien que l'arbre soit complètement développé.

Ainsi, l'algorithme BuildTree de construction d'arbres de décision ne peut pas être directement utilisé pour l'apprentissage des distributions de probabilités conditionnelles $P_a(X'_i|s)$: il est nécessaire d'ajouter un élagage afin d'éviter le sur-apprentissage lorsque le problème possède une fonction de transition stochastique. L'algorithme BuildTreeS, décrit figure 4.4, est une adaptation de l'algorithme BuildTree et a été adapté pour ce type de problème.

Entrée(s) : $\mathcal{E} = \{\langle \mathfrak{a}_0, \varsigma_0 \rangle, \ldots, \langle \mathfrak{a}_n, \varsigma_n \rangle\}$, une mesure d'information $\mathcal{M}$, un nœud k de Tree $[F]$ **Sortie(s) :** Tree $[F]$

1. Soit $\mathcal{V}_i \leftarrow$ SelectAttr$(\mathcal{M}, \mathcal{E})$
2. $\forall \nu \in$ Dom$(\mathcal{V}_i)$: $\mathcal{E}_\nu \leftarrow \{e = \langle \mathfrak{a}, \varsigma \rangle | e \in \mathcal{E}$ et $\mathfrak{a}[\mathcal{V}_i] = \nu\}$
3. **Si** IsDiffSig$(\mathcal{M}, \{\forall \nu \in$ Dom$(\mathcal{V}_i)$:$\mathcal{E}_\nu\})$ est faux :

   Alors : transformer k en une feuille contenant : Aggregate$(\mathcal{M}, \{\forall \nu \in$ Dom$(\mathcal{V}_i)$:$\mathcal{E}_\nu\})$

   Sinon :

   (a) Transformer k en un nœud de décision testant $\mathcal{V}_i$

   (b) $\forall \nu \in$ Dom$(\mathcal{V}_i)$: BuildTreeS$(\mathcal{E}_\nu, k_\nu)$, avec k_ν le nœud enfant de k correspondant à $\mathcal{V}_i = \nu$

FIG. 4.4 – L'algorithme BuildTreeS construisant un arbre à partir d'un ensemble d'exemples $\mathcal{E}$.

Celui-ci contient principalement deux modifications. Premièrement, l'algorithme effectue un pré-élagage afin d'éviter le développement inutile de branches pour l'apprentissage de fonctions stochastiques. Par conséquent, après avoir sélectionné le meilleur attribut $\mathcal{V}$ à installer, un test vérifie si les ensembles $\mathcal{E}_\nu$ distribués aux nœuds enfants sont significativement différents. Ce test est réalisé par l'opérateur IsDiffSig et dépend de la mesure d'information utilisée. Deuxièmement, lorsqu'une feuille est installée dans l'arbre, son contenu est défini par l'opérateur Aggregate qui agrège l'ensemble $\mathcal{E}$ d'exemples présents à la feuille (par exemple une distribution de probabilités). Une feuille est installée lorsque les ensembles $\mathcal{E}_\nu$ ne sont pas considérés comme significativement différents (parce que l'ensemble d'exemples est pur par exemple) par l'opérateur IsDiffSig. Dans le cas contraire, c'est-à-dire lorsque la différence entre les ensembles $\mathcal{E}_\nu$ est considérée comme significative, un nœud de décision testant l'attribut $\mathcal{V}$ est créé et les ensembles $\mathcal{E}_\nu$ sont distribués dans les branches, de la même façon que l'algorithme BuildTree.

Dans le cadre de l'apprentissage de la fonction de transition, les deux opérateurs IsDiffSig et Aggregate s'implémentent à l'aide du critère χ^2. En effet, l'opérateur IsDiffSig compare la valeur $\chi^2_{\mathcal{E}_\nu}$ pour les ensembles $\mathcal{E}_\nu$ à un seuil τ_{χ^2} fixé a priori. Ainsi, si $\chi^2_{\mathcal{E}_\nu} \geq \tau_{\chi^2}$, alors les ensembles sont considérés comme significativement différents et un nœud de décision testant l'attribut $\mathcal{V}$ est alors installé. Dans le cas contraire, une feuille est installée et son contenu est défini par Aggregate. Nous rappelons que la fonction F à apprendre est une distribution de probabilités conditionnelles $P_a(X'_i|s)$. L'opérateur Aggregate retourne donc la distribution de probabilités $\forall x_i \in$ Dom(X_i) : $P(x_i)$, à partir de l'ensemble d'exemples $\mathcal{E}$ (calculé d'après l'équation 4.2).

Ensemble d'exemples pour l'apprentissage des fonctions de récompenses localisées

Nous avons indiqué comment un échantillon d'observations pouvait être décomposé pour l'apprentissage de la fonction de transition du FMDP. Nous décrivons une méthode similaire afin de construire une représentation de la fonction de récompense.

L'expérience de l'agent constitue une suite d'observations $\langle s, a, r\rangle$. Il paraît alors naturel de construire un ensemble d'exemples de la forme $\langle \mathfrak{a} = \{x_1, \ldots, x_n\}, \varsigma = r\rangle$, ou bien de la forme $\langle \mathfrak{a} = \{x_1, \ldots, x_n, a\}, \varsigma = r\rangle$ si l'action est aussi considérée comme un attribut de l'exemple. Cependant, les algorithmes d'induction d'arbres de régression tels que nous les avons décrit section 4.1.1 ne permettent pas de découvrir la structure additive d'une fonction, comme cela est illustré figure 4.5 avec la fonction de récompense de *Coffee Robot*.

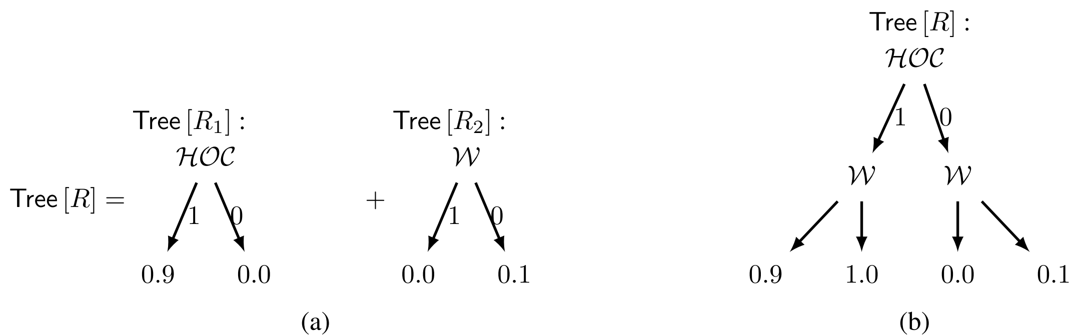


FIG. 4.5 – Alors que la fonction de récompense du problème *Coffee Robot* peut être décrit sous la forme de deux fonctions de récompense localisées Tree $[R_1]$ et Tree $[R_2]$ (figure a), l'algorithme d'induction d'arbres de régression ne peut construire qu'un seul arbre Tree $[R]$ représentant la somme des fonctions R_1 et R_2 (figure b) à partir d'exemples de la forme $\langle \mathfrak{a} = \{x_1, \ldots, x_n\}, \varsigma = r\rangle$, par exemple.

En effet, pour apprendre une fonction de type $A + B$, un algorithme d'induction d'arbres de régression construira un arbre Tree $[A + B]$ représentant l'ensemble des combinaisons $\underline{\text{Dom}}(A) \times \underline{\text{Dom}}(B)$, plutôt que de construire deux arbres Tree $[A]$ + Tree $[B]$. Nous supposons donc que nous connaissons a priori la décomposition additive de la fonction de récompense du problème d'apprentissage par renforcement.

Par conséquent, l'observation de la récompense r est composé d'un ensemble d'observations de récompenses r_i à partir duquel il est possible, pour chaque récompense localisée, de construire un ensemble d'exemples $\langle \mathfrak{a} = \{x_1, \ldots, x_n\}, \varsigma = r_i\rangle$ (ou $\langle \mathfrak{a} = \{x_1, \ldots, x_n, a\}, \varsigma = r_i\rangle$ si l'action est inclue dans les attributs) afin d'utiliser l'induction d'arbres de régression pour construire une représentation de chaque fonction de récompense localisée Tree $[R_i]$.

Les problèmes de la littérature des FMDPs utilisent tous des fonctions de récompenses déterministes pour lesquels l'algorithme <u>BuildTree</u> de construction d'arbres décrit figure 4.1 peut être directement utilisé avec la mesure des moindres carrés (section 4.1.3). Lorsqu'une fonction R est

stochastique, l'algorithme stochastique de construction d'arbres peut aussi être utilisé avec la mesure des moindres carrés. Dans ce cas, les différentes valeurs des exemples peuvent être agrégés (opérateur Aggregate) en calculant la moyenne des valeurs avec l'équation 4.6 (page 86). Le test déterminant si deux ensembles d'exemples sont significativement différents (opérateur IsDiffSig) peut être évalué de différentes façons. La section 7.3 (page 182) consacrée à la mise en œuvre de nos travaux dans un problème réel concernant le jeu vidéo décrit l'une d'entre elles.

4.2.2 Algorithmes de construction de FMDPs

À partir de la décomposition précédemment décrite des observations de l'agent en ensembles d'exemples, il devient possible d'utiliser les algorithmes d'induction d'arbres de décisions pour construire les différentes distributions de probabilités conditionnelles de la fonction de transition ainsi que les récompenses localisées de la fonction de récompense. Pour cela, nous proposons deux nouveaux algorithmes pour la construction d'un FMDP, nommément BuildFMDP et BuildFMDPnAT.

Ces deux méthodes utilisent les algorithmes de construction d'arbres de décision afin d'apprendre, à partir d'un échantillon d'observations de l'agent dans son environnement, les représentations utilisées pour la définition complète d'un FMDP.

Cependant, elles se différencient par la façon dont l'action exécutée par l'agent est utilisée pour construire la fonction de transition. Le premier algorithme, BuildFMDP, construit une fonction de transition constituée d'un arbre Tree $[P_a(X'_i|s)]$ par action a et par variable X_i. En revanche, afin d'obtenir un modèle plus compacte lorsque certaines variables du problème ne dépendent pas de l'action réalisée par l'agent (comme par exemple la variable $\mathcal{R}$, représentant "est-ce-qu'il pleut ?", dans le problème *Coffee Robot*), l'algorithme BuildFMDPnAT construit une fonction de transition constituée d'un arbre Tree $[P(X'_i|s,a)]$ par variable X_i. Nous commençons par décrire l'algorithme BuildFMDP.

Un arbre par action par variable : l'algorithme BuildFMDP

L'algorithme BuildFMDP construit un FMDP représenté par, d'une part, une fonction de transition constitué d'un arbre Tree $[P_a(X'_i|s)]$ par action a et par variable X_i et, d'autre part, une fonction de récompense constitué d'un arbre Tree $[R_i(s,a)]$ par récompense localisée R_i. L'algorithme est décrit dans la figure 4.6.

L'algorithme commence par l'apprentissage de la fonction de transition en construisant les différents ensembles $\mathcal{E}^a_{X_i}$ pour chaque distribution de probabilités conditionnelle $P(X'_i|s,a)$ (étape 2), puis construit la représentation structurée correspondante Tree $[P_a(X'_i|s)]$ en utilisant l'algorithme d'induction d'arbres de décision BuildTreeS (étape 3). Ensuite, de façon similaire, l'algorithme construit une représentation de la fonction de récompense en commençant par construire les diffé-

Entrée(s) : Un ensemble d'observations $\mathcal{O} = \{\langle s, a, s', r\rangle\}$ **Sortie(s) :** Un FMDP représenté par : $\forall a \in A, \forall X_i \in X$: $\mathsf{Tree}\,[P_a(X'_i|s)]$ et $\forall R_i \in R$: $\mathsf{Tree}\,[R_i]$

1. Initialisation des ensembles d'exemples : $\forall a \in A, \forall X_i \in X$: $\mathcal{E}^a_{X_i} = \emptyset$
2. Faire $\forall o = \langle s, a, s'\rangle \in \mathcal{O}$: faire $\forall X_i \in X$:
   (a) Soit $e = \langle \mathfrak{a} = s, \varsigma = x'_i\rangle$ avec x'_i la valeur de X_i dans s'
   (b) $\mathcal{E}^a_{X_i} \leftarrow \{e\} \cup \mathcal{E}^a_{X_i}$
3. $\forall a \in A, \forall X_i \in X$: $\mathsf{Tree}\,[P_a(X'_i|s)] \leftarrow$ BuildTreeS$(\mathcal{E}^a_{X_i}, \mathcal{M}_{\chi^2})$
4. Faire $\forall o = \langle s, a, r\rangle \in \mathcal{O}$: faire $\forall r_i \in r$:
   (a) Soit $e = \langle \mathfrak{a} = \{s, a\}, \varsigma = r_i\rangle$ avec r_i la valeur de la récompense localisée R_i dans r
   (b) $\mathcal{E}_{R_i} \leftarrow \{e\} \cup \mathcal{E}_{R_i}$
5. $\forall R_i \in R$: $\mathsf{Tree}\,[R_i]$: $\mathsf{Tree}\,[R_i] \leftarrow$ BuildTree$(\mathcal{E}_{R_i}, \mathcal{M}_{LS})$
6. Retourner $\forall a \in A, \forall X_i \in X$: $\mathsf{Tree}\,[P_a(X'_i|s)]$ et $\forall R_i \in R$: $\mathsf{Tree}\,[R_i]$

FIG. 4.6 – L'algorithme BuildFMDP construit un FMDP à partir d'un ensemble d'observations de l'agent dans son environnement.

rents ensembles d'exemples $\mathcal{E}_{R_i}$ puis utilise l'algorithme d'induction d'arbres de régression pour construire $\mathsf{Tree}\,[R_i]$.

À partir d'un échantillon d'observations, l'algorithme BuildFMDP retourne donc, d'une part, l'ensemble des distributions de probabilités conditionnelles $\{\forall a \in A, \forall X_i \in X : \mathsf{Tree}\,[P_a(X'_i|s)]\}$ définissant complètement la fonction de transition, d'autre part l'ensemble des fonctions localisées $\{\forall R_i \in R : \mathsf{Tree}\,[R_i]\}$ définissant complètement la fonction de récompense du FMDP représentant le problème d'apprentissage par renforcement à résoudre.

Un arbre par variable : l'algorithme BuildFMDPnAT

Bien que fonctionnant sur un principe similaire, l'algorithme BuildFMDPnAT décrit figure 4.7 propose une méthode alternative pour construire un FMDP à partir d'un ensemble d'observations. En effet, la fonction de transition du FMDP construit est composée d'un arbre $\mathsf{Tree}\,[P(X'_i|s, a)]$ par variable X_i.

L'algorithme BuildFMDPnAT d'apprentissage d'un FMDP est décrit figure 4.7. La différence avec l'algorithme BuildFMDP se situe principalement au niveau de l'apprentissage de la fonction de transition (étape 3). En effet, plutôt que de construire un ensemble d'exemples par arbre $\mathsf{Tree}\,[P_a(X'_i|s)]$ par action a et par variable X_i, l'algorithme construit des exemples en ajoutant l'action réalisée par l'agent dans la liste $\mathfrak{a} = \{s, a\}$ des attributs constituant un ensemble d'exemples

---

Entrée(s) : Un ensemble d'observations $\mathcal{O} = \{\langle s, a, s', r\rangle\}$ **Sortie(s) :** Un FMDP représenté par : $\forall X_i \in X$: $\mathsf{Tree}\,[P(X_i'|s,a)]$ et $\forall R_i \in R$: $\mathsf{Tree}\,[R_i]$

---

1. Initialisation des ensembles d'exemples : $\forall X_i \in X : \mathcal{E}_{X_i} = \emptyset$
2. Faire $\forall o = \langle s, a, s'\rangle \in \mathcal{O}$: faire $\forall X_i \in X$:
   (a) Soit $e = \langle \mathfrak{a} = \{s, a\}, \varsigma = x_i'\rangle$ avec x_i' la valeur de X_i dans s'
   (b) $\mathcal{E}_{X_i} \leftarrow \{e\} \cup \mathcal{E}_{X_i}$
3. $\forall X_i \in X$: $\mathsf{Tree}\,[P(X_i'|s,a)] \leftarrow$ BuildTreeS$(\mathcal{E}_{X_i}, \mathcal{M}_{\chi^2})$
4. Faire $\forall o = \langle s, a, r\rangle \in \mathcal{O}$: faire $\forall r_i \in r$:
   (a) Soit $e = \langle \mathfrak{a} = \{s, a\}, \varsigma = r_i\rangle$ avec r_i la valeur de la récompense localisée R_i dans r
   (b) $\mathcal{E}_{R_i} \leftarrow \{e\} \cup \mathcal{E}_{R_i}$
5. $\forall R_i \in R$: $\mathsf{Tree}\,[R_i]$: $\mathsf{Tree}\,[R_i] \leftarrow$ BuildTree$(\mathcal{E}_{R_i}, \mathcal{M}_{LS})$
6. Retourner $\forall X_i \in X$: $\mathsf{Tree}\,[P(X_i'|s,a)]$ et $\forall R_i \in R$: $\mathsf{Tree}\,[R_i]$

FIG. 4.7 – L'algorithme BuildFMDPnAT construit un FMDP, avec un seul arbre $\mathsf{Tree}\,[P(X_i'|s,a)]$ représentant les probabilités conditionnelles d'une variable X_i, à partir d'un ensemble d'observations de l'agent dans son environnement.

par variable X_i. L'apprentissage de la fonction de récompense est identique.

Nous pensons que l'utilisation d'un arbre par variable permet de représenter des fonctions de transition de façon plus compacte, plus particulièrement lorsque certaines transitions du problème ne dépendent pas de l'agent. Par exemple, dans le problème *Coffee Robot*, nous avons notamment cité précédemment la variable $\mathcal{R}$ représentant “est-ce-qu'il pleut ?” dans le problème. Aucune action du robot ne lui permet d'agir sur la valeur de cette variable. Par conséquent, sa distribution de probabilités $P_a(\mathcal{R}|s)$ est la même pour toutes les actions du problème et peut donc être représentée par une seule distribution de probabilités $P(\mathcal{R}|s)$ ne dépendant pas de l'action exécutée par le robot.

De même, une telle similarité peut exister dans des contextes pour certaines distributions de probabilités conditionnelles. Par exemple, dans le problème *Coffee Robot*, la variable $\mathcal{W}$, correspondant à “le robot est-t-il mouillé ?”, dépend de l'action du robot : s'il pleut et que le robot choisit l'action $\mathcal{G}$o sans parapluie, alors il sera mouillé avec une probabilité importante. Cependant, dans le contexte où le robot est déjà mouillé, comme il n'a aucune action pour se sécher, la distribution de probabilités $P(\mathcal{W}|s)$ ne dépend pas de l'action effectuée par le robot. Dans ce contexte, cette distribution de probabilités est la même pour toutes les actions et peut donc être représentée que par une seule distribution pour toutes les actions.

Il est important de noter que les algorithmes de planification dans les FMDPs tels que nous les avons décrit au chapitre 3 n'utilisent pas une telle représentation de la fonction de transition. Par conséquent, afin de pouvoir réutiliser ces algorithmes avec la représentation construite par l'algorithme BuildFMDPnAT, il est nécessaire de pouvoir construire un arbre $\mathsf{Tree}\left[P_a(X'_i|s)\right]$ à partir de $\mathsf{Tree}\left[P(X'_i|s,a)\right]$ pour une action a donnée.

Cette opération est effectuée avec l'opérateur $\underline{\text{Extract}}(T,X,x)$, supplémentaire aux opérateurs définis section 3.2.2. Pour un arbre T, une variable X et une valeur $x \in \underline{\text{Dom}}(X)$, l'opérateur $\underline{\text{Extract}}(T,X,x)$ construit un nouvel arbre en supprimant les partitions pour lesquelles la variable X est différente de la valeur x. Cette opération peut être effectuée en remplaçant tous les nœuds de décision de l'arbre testant la variable X par leur branche pour laquelle $X = x$. La figure 4.8 montre un exemple d'extraction de la distribution de probabilités conditionnelle $\mathsf{Tree}\left[P_{\mathcal{D}\mathrm{elC}}(\mathcal{HOC}'|s)\right]$ à partir de la distribution de probabilités conditionnelle $\mathsf{Tree}\left[P(\mathcal{HOC}'|s,a)\right]$.

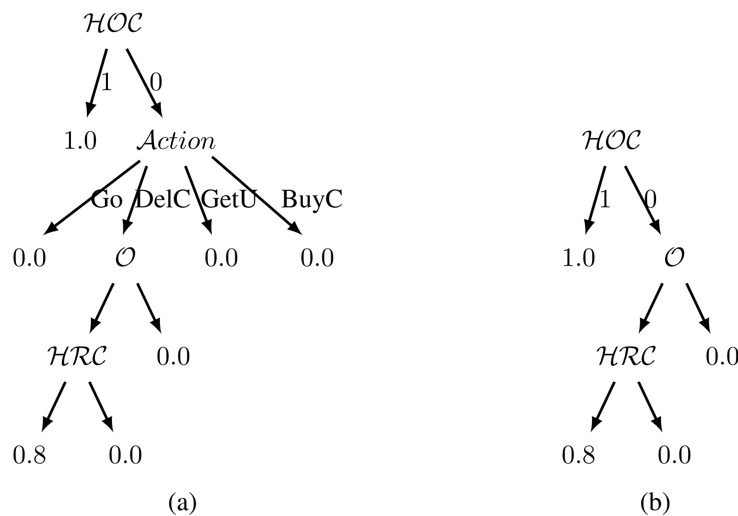


FIG. 4.8 – Extraction de la distribution de probabilités conditionnelle $\mathsf{Tree}\left[P_{\mathcal{D}\mathrm{elC}}(\mathcal{HOC}'|s)\right]$ (figure b) à partir de la distribution de probabilités conditionnelle $\mathsf{Tree}\left[P(\mathcal{HOC}'|s,a)\right]$ (figure a) avec l'opérateur $\underline{\text{Extract}}(\mathsf{Tree}\left[P(\mathcal{HOC}'|s,a)\right], \mathcal{A}ction, \mathcal{D}\mathrm{elC})$.

Enfin, dans le cadre de l'apprentissage, il est important de noter qu'une différence existe concernant le calcul des distributions de probabilité ne dépendant pas de l'action. En effet, dans l'algorithme BuildFMDP, le fait de construire un ensemble d'exemples par action et par variable partitionne l'ensemble des observations. Cette partition a pour conséquence de distribuer les exemples vers l'arbre correspondant à l'action qui a été exécutée, même lorsque ceux ci sont utilisés pour évaluer la probabilité d'une transition ne dépendant pas de l'action. Ce n'est pas le cas de l'algorithme BuildFMDPnAT qui ne construit qu'un ensemble d'exemples par variable.

Gestion des valeurs manquantes

Lorsque l'algorithme BuildFMDPnAT est utilisé, ou bien lorsque les variables du problème peuvent prendre plus de 2 valeurs différentes avec BuildFMDP, un problème peut se poser lorsqu'un nœud de décision est installé et qu'aucun exemple ne correspond à l'une des branches partant de ce nœud. Ce cas est illustré figure 4.9 pour un arbre Tree $[P(\mathcal{HOC}'|s,a)]$ qui serait construit avec l'algorithme BuildFMDPnAT et pour lequel, par exemple, les actions $\mathcal{G}$o et $\mathcal{G}$etU n'auraient jamais été exécutées lorsque la propriétaire n'avait pas de café (c'est-à-dire $\mathcal{HOC} = 0$).

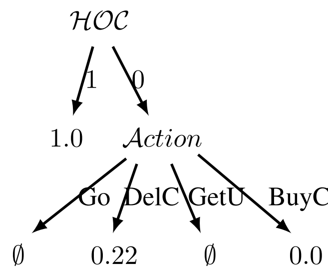


FIG. 4.9 – Arbre Tree $[P(\mathcal{HOC}'|s,a)]$ incomplet pour lequel deux feuilles (correspondant à $\mathcal{HOC} = 0$ et $\mathcal{A}ction = \mathcal{G}$o ou $\mathcal{A}ction = \mathcal{G}$etU) n'ont aucun exemple associé.

De tels arbres contenant des feuilles vides posent problème lors de l'intégration avec les algorithmes de planification puisqu'elles ne correspondent à aucune probabilité pouvant être utilisée lors du calcul des fonctions de valeur d'action à partir d'une fonction de valeur (plus particulièrement lors du calcul de l'équation 2.4 par l'algorithme Regress (page 51) utilisé par SPI et SVI et, par l'algorithme BackProjRule (page 73) dans le calcul des fonctions g_i^a). Pour contourner ce problème, nous proposons plusieurs solutions pouvant être utilisées.

Pour un nœud associé à un ensemble d'exemples $\mathcal{E}$, la première solution consiste à interdire l'installation d'un attribut pour lequel il existe des valeurs sans exemple. Avant d'installer un test sur l'attribut $\mathcal{V}$, on vérifie donc que $\forall \nu \in$ Dom$(\mathcal{V}_i) : |\{e = \langle \mathfrak{a}, \varsigma \rangle | e \in \mathcal{E}$ et $\mathfrak{a}[\mathcal{V}_i] = \nu\}| > 0$, c'est-à-dire que chaque valeur ν de l'attribut $\mathcal{V}$ est associée à un ensemble d'exemples dont le cardinal est strictement supérieur à 0. Une deuxième solution consiste à assigner à chaque feuille sans exemple une distribution de probabilités calculée à partir de l'ensemble d'exemples $\mathcal{E}$ du nœud parent.

Les résultats présentés dans la suite de ce chapitre ont été obtenus en utilisant la première solution. Dans le cadre de l'apprentissage incrémental, le chapitre 5 présente une troisième alternative à ce problème.

4.2.3 Intégration avec les algorithmes de planification

Les algorithmes d'apprentissage BuildFMDP et BuildFMDPnAT présentés dans la section précédente construisent un FMDP représentant complètement le problème d'apprentissage par renfor-

cement à résoudre. À partir de ce FMDP, il est alors possible d'utiliser l'une des méthodes de planification dans les FMDPs, décrites au chapitre 3, afin de déterminer une solution au problème. Pour cela, nous distinguons deux approches différentes, suivant si la méthode de planification exploite ou non la décomposition additive de la fonction de récompense.

Dans un premier temps, nous décrirons l'algorithme BatchSolveNoAD qui intègrent l'apprentissage avec des algorithmes de planifications n'exploitant pas la décomposition additive de la fonction de récompense, tels que les algorithmes SPI, SVI et SPUDD. Dans ce cas, la solution au problème d'apprentissage par renforcement est la représentation structurée d'une politique.

Dans un deuxième temps, nous décrirons l'algorithme BatchSolve qui intègrent l'apprentissage avec la planification basée sur la programmation linéaire et qui exploite la décomposition additive de la fonction de récompense. Dans ce cas, la solution au problème d'apprentissage par renforcement est la représentation structurée d'une fonction de valeur. Ainsi la représentation explicite d'une politique, impossible pour certains problèmes, est évitée.

L'algorithme BatchSolveNoAD

L'algorithme BatchSolveNoAD intègre les fonctions d'apprentissage d'un FMDP décrites précédemment avec des méthodes de planification n'exploitant pas la décomposition additive de la fonction de récompense. L'ensemble des fonctions construites par apprentissage et définissant le FMDP utilisent des représentations sous la forme d'arbres de décision. Contrairement à SPI et SVI où cette représentation est utilisée directement par les algorithmes, SPUDD utilise des ADDs pour représenter les fonctions du problème (section 3.3.1, page 55). Il est possible de construire un BDD à partir d'un arbre de décision représentant une fonction binaire (Hoey et al., 2000). Cette méthode est généralisable à d'autres fonctions et une implémentation de celle-ci est disponible avec la version de SPUDD disponible sur Internet. Nous invitons le lecteur intéressé par les détails de cet algorithme à consulter l'article de Hoey et al. (2000) et le code source de SPUDD.

Concernant la fonction de récompense, les algorithmes SPI, SVI et SPUDD ne permettent pas d'exploiter sa décomposition additive. Afin de pouvoir les utiliser avec la représentation construite par les algorithmes d'apprentissage d'un FMDP, il est donc nécessaire de calculer préalablement la somme des fonctions de récompense localisées $R = \sum_i R_i$. L'algorithme BatchSolveNoAD de la figure 4.10 propose l'intégration de ces trois algorithmes de planification avec les méthodes d'apprentissage d'un FMDP décrites ci-dessus.

L'algorithme BatchSolveNoAD commence par construire le FMDP $\mathcal{F}$ à partir de l'expérience de l'agent dans son environnement représenté par l'échantillon $\mathcal{O}$ d'observations. Ensuite, en utilisant l'opérateur Merge sur les arbres de décision défini section 3.2.2, la somme des fonctions de récompense localisées est calculée, pour construire le FMDP $\mathcal{F}'$. Enfin, un algorithme de planification tel que SPI, SVI ou SPUDD est utilisé pour en déduire la politique optimale du FMDP $\mathcal{F}'$ représentant le problème d'apprentissage par renforcement à résoudre.

Entrée(s) : Un ensemble d'observations $\mathcal{O} = \{\langle s, a, s', r \rangle\}$ **Sortie(s) :** V^*, π^*

---

1. Utiliser un algorithme d'apprentissage (BuildFMDP ou BuildFMDPnAT) pour construire, à partir de $\mathcal{O} = \{\langle s, a, s', r \rangle\}$, le FMDP :
   $\mathcal{F} = \{\forall X_i \in X : \text{Tree}\,[P(X'_i|s,a)]$ et $\forall i \in [1,n] : \text{Tree}\,[R_i]\}$
2. $\text{Tree}\,[R] \leftarrow$ Merge($\{\forall i \in [1,n] : \text{Tree}\,[R_i]\}$) (en utilisant l'addition comme opérateur de combinaison)
3. Utiliser un algorithme de planification (SPI, SVI ou SPUDD) pour calculer V^* et π^* à partir du FMDP :
   $\mathcal{F}' = \{\forall X_i \in X : \text{Tree}\,[P(X'_i|s,a)]$ et $\text{Tree}\,[R]\}$
4. Retourner V^* et π^*

FIG. 4.10 – L'algorithme BatchSolveNoAD intégrant les algorithmes d'apprentissage d'un FMDP avec les algorithmes de planification SPI, SVI ou SPUDD n'exploitant pas la décomposition additive de la fonction de récompense.

On peut remarquer que l'algorithme BatchSolveNoAD ne nécessite aucune connaissance a priori des indépendances relatives aux fonctions et des indépendances relatives aux contextes requises par les algorithmes de planification SPI, SVI et SPUDD. Ainsi, l'ensemble de ces indépendances sont découvertes par l'apprentissage lors de la construction du FMDP.

L'algorithme BatchSolve

Bien qu'étant légèrement différente, l'intégration des algorithmes d'apprentissage d'un FMDP avec les algorithmes de planification proposés par Guestrin et al. (2003b) et utilisant la programmation linéaire est tout aussi directe. En effet, nous rappelons que, sans nécessité de transformations particulières, un arbre peut être considéré comme un ensemble de règles mutuellement exclusives et exhaustives (voir section 3.4.1). L'algorithme BatchSolve, décrit figure 4.11, propose une telle intégration. Nous rappelons que, contrairement à l'algorithme BatchSolveNoAD décrit précédemment, l'algorithme BatchSolve permet d'utiliser des méthodes de planification exploitant la décomposition additive de la fonction de récompense.

De la même façon que BatchSolveNoAD, l'algorithme BatchSolve commence par construire un FMDP représentant le problème d'apprentissage par renforcement à partir de l'ensemble d'observations de l'agent dans son environnement. Ensuite, afin d'obtenir une approximation de la fonction de valeur optimale $\hat{V}^*$ du FMDP, l'algorithme FactoredLPA est appelé. Cependant, contrairement aux algorithmes SPI, SVI ou SPUDD, FactoredLPA ne retourne pas la politique optimale correspondant aux problèmes.

En effet, alors que, pour certains problèmes, une approximation additive permet d'obtenir des

Entrée(s) : Un ensemble d'observations $\mathcal{O} = \{\langle s, a, s', r\rangle\}$, un ensemble de fonctions de bases $H = \{\mathsf{Rules_v}[h_0], \ldots, \mathsf{Rules_v}[h_k]\}$, un ordre d'élimination des variables O
Sortie(s) : $\hat{V}^*, \{\forall a \in A : \hat{Q}^*_a\}$

1. Utiliser un algorithme d'apprentissage (BuildFMDP ou BuildFMDPnAT) pour construire, à partir de $\mathcal{O} = \{\langle s, a, s', r\rangle\}$, le FMDP : $\mathcal{F} = \{\forall X_i \in X : \mathsf{Tree}[P(X'_i|s,a)]$ et $\forall i \in [1,n] : \mathsf{Tree}[R_i]\}$
2. $\{\{w_0, \ldots, w_k\}, \{\forall a \in A, \forall h_i \in H : g^a_i\}\} \leftarrow$ FactoredLPA$(\mathcal{F}, H, O)$
3. Calculer $\hat{V}^*$ à partir de H et $\{w_0, \ldots, w_k\}$
4. Pour tout $a \in A$: calculer $\hat{Q}^*_a$ à partir de $\{\forall h_i \in H : g^a_i\}$ et $\forall i \in [1,n] : \mathsf{Tree}[R_i]$
5. Retourner $\hat{V}^*$ et $\{\forall a \in A : \hat{Q}^*_a\}$

FIG. 4.11 – L'algorithme BatchSolve intégrant les algorithmes d'apprentissage d'un FMDP avec l'algorithme de planification FactoredLPA exploitant la composition additive du problème.

représentations très compactes des fonctions de valeur, une représentation explicite de la politique peut nécessiter un espace mémoire augmentant de façon exponentielle avec le nombre de variables d'état, rendant par conséquent une telle représentation impossible pour les grands problèmes. Ainsi, pour éviter une telle représentation, l'algorithme FactoredLPA retourne l'ensemble des fonctions g^a_i à partir duquel l'ensemble des fonctions de valeur d'action $\hat{Q}^*_a$ est construit (en utilisant l'équation 3.8, page 72). Pour connaître la meilleure action à exécuter pour un état s donné, la récompense espérée $\hat{Q}^*_a(s)$ est calculée pour chaque action. L'action ayant la récompense espérée la plus élevée est sélectionnée.

Il est important de noter que, à l'instar de BatchSolveNoAD, l'algorithme BatchSolve ne nécessite aucune connaissance a priori concernant les indépendances relatives aux fonctions pour les fonctions de transitions et de récompenses. Cependant, afin de pouvoir exploiter une approximation additive de la fonction de valeur, il requiert la définition d'un ensemble de fonctions de base, un ordre d'élimination des variables et la décomposition additive de la fonction de récompense.

4.2.4 Réorganisation de règles exhaustives et mutuellement exclusives

L'algorithme de construction d'arbres de décision BuildTree peut être utilisé dans un autre cadre que celui de l'apprentissage des FMDPs. En effet, les algorithmes SPI et SVI calculent des représentations sous la forme d'arbres de décision de la fonction de valeur optimale $\mathsf{Tree}[V^*]$ et de la politique optimale $\mathsf{Tree}[\pi^*]$. Cependant, rien ne garantit que la représentation de ces fonctions est optimales, c'est-à-dire que l'ordre des variables utilisé dans les arbres est optimale.

Il est possible d'utiliser l'algorithme BuildTree de construction d'arbres de décision figure 4.1 pour réorganiser de tels arbres et, d'une façon plus générale, toute fonction composée d'un ensemble de règles exhaustives et mutuellement exclusives (telles que les ensembles de règles ou bien les représentations tabulaires). Nous proposons l'algorithme BuildTreeF, décrit figure 4.12 et illustré figure 4.13, pour effectuer une telle réorganisation.

Entrée(s) : Une fonction F représentée par un ensemble $F = \{|c_1 : \kappa_1|, \ldots, |c_n : \kappa_n|\}$ de règles exhaustives et mutuellement exclusives, une mesure d'information $\mathcal{M}$
Sortie(s) : $\mathsf{Tree}[F]$

---

1. Initialiser l'ensemble d'exemples $\mathcal{E} = \emptyset$
2. Pour toute règle $|c : \kappa| \in F$, faire :
   (a) Soit l'exemple (incomplet) $e \leftarrow \langle \mathfrak{a} = c, \varsigma = \kappa \rangle$
   (b) $\mathcal{E} \leftarrow \mathcal{E} \cup \{e\}$
3. $\mathsf{Tree}[F] \leftarrow$ BuildTree$(\mathcal{E}, \mathcal{M})$
4. Retourner $\mathsf{Tree}[F]$

FIG. 4.12 – L'algorithme BuildTreeF construit un arbre étant donné un ensemble de règles exhaustives et mutuellement exclusives.

À partir d'un l'ensemble $F = \{|c : \kappa|\}$ de règles exhaustives et mutuellement exclusives définissant une fonction F, (représentant, par exemple, l'arbre $\mathsf{Tree}[F]$ à réorganiser), l'algorithme BuildTreeF commence par construire un ensemble d'exemples représentant l'ensemble des partitions utilisées par la fonction F. Lorsque la fonction F utilise des indépendances relatives aux contextes (par exemple, la branche d'un arbre ne testant pas tous les attributs possibles pour accéder à une feuille), certains attributs $\mathcal{V}$ des exemples construits n'ont alors aucune valeur associée. L'arbre est ensuite construit à partir de cet ensemble d'exemples. Afin de respecter le partitionnement de la fonction F, un test supplémentaire est ajouté à la mesure d'information permettant de sélectionner l'attribut à tester à un nœud de décision : un attribut $\mathcal{V}$ n'est sélectionné que si et seulement si le nombre d'exemples pour lesquels l'attribut $\mathcal{V}$ a une valeur associée est égal au nombre total d'exemples dans l'ensemble d'exemples du nœud courant en train d'être créé.

Les figures 4.13(a) et 4.13(b) représentent respectivement une fonction F sous la forme d'un arbre de décision $\mathsf{Tree}[F]$ et d'un ensemble de règles (équivalent) $\mathsf{Rules}[F]$. L'algorithme de réorganisation BuildTreeF établit l'ensemble $\mathcal{E}_F$ d'exemples correspondant au partitionnement utilisé par la représentation de F (figure 4.13(c)). À partir de cet ensemble, l'algorithme BuildTree construit ensuite un nouvel arbre de décision $\mathsf{Tree}[F]$ représentant la même fonction F mais de

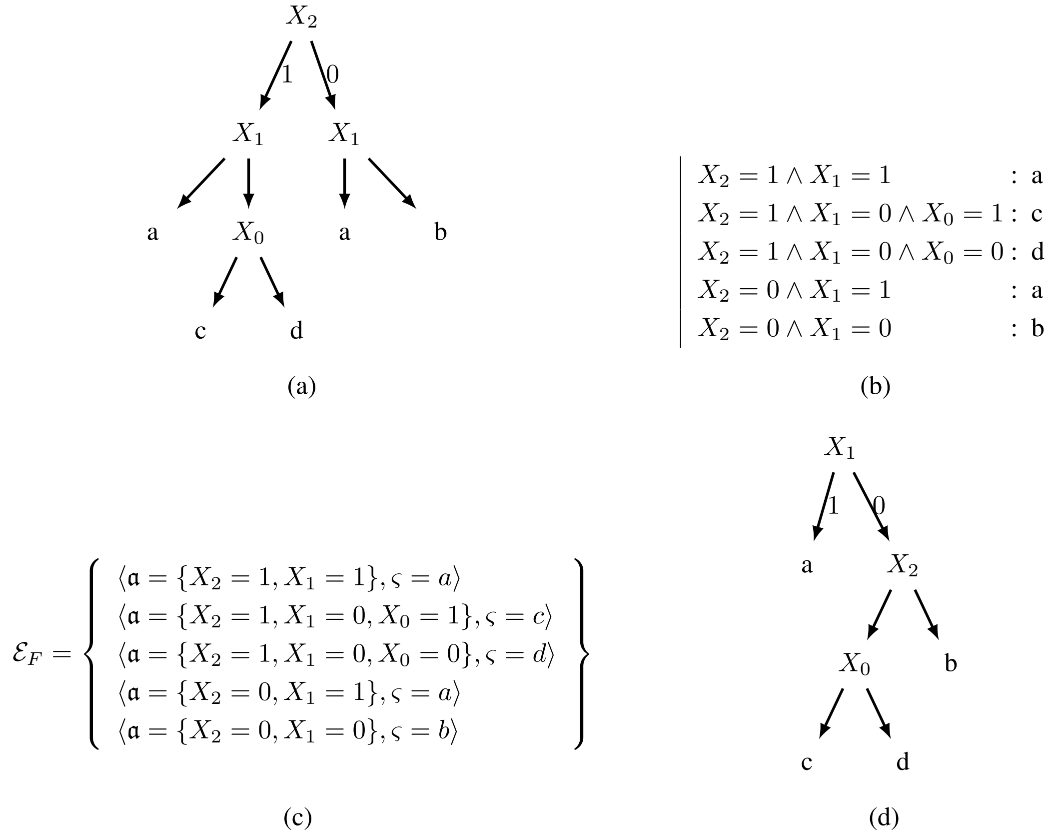


FIG. 4.13 – À partir d'une fonction F définie sous la forme d'un arbre de décision (figure a) ou d'un ensemble de règles (figure b), l'algorithme BuildTreeF construit un ensemble $\mathcal{E}_F$ d'exemples (figure c) puis appelle BuildTree pour reconstruire F sous la forme d'un arbre de décision à partir de l'ensemble $\mathcal{E}_F$ d'exemples (figure d).

façon plus compacte (figure 4.13(d)).

Il est important de noter que l'algorithme BuildTreeF ne modifie aucunement la définition de la fonction F à réorganiser. Si le partitionnement effectué par la représentation de F ne peut être modifié, l'algorithme BuildTreeF retournera alors la même représentation. Dans le cas contraire, bien que BuildTreeF puisse construire une représentation $\mathsf{Tree}\,[F]$ différente, sa définition restera strictement identique.

Suivant la mesure d'information utilisée, l'algorithme BuildTreeF peut être utilisé pour réorganiser différentes fonctions. Une mesure d'information pour des valeurs symboliques sera utilisée pour réorganiser une politique $\mathsf{Tree}\,[\pi]$, alors qu'une mesure d'information pour des valeurs réelles sera utilisée pour réorganiser une fonction de valeur optimale$\mathsf{Tree}\,[V]$. Enfin, l'algorithme BuildTree s'intègre naturellement avec les algorithmes SPI et SVI, notamment pour remplacer directement l'opérateur Simplification réalisé sur les arbres (section 3.2.2, page 49).

4.3 Résultats

Cette section valide de façon expérimentale les algorithmes d'apprentissage d'un FMDP, nommément BuildFMDP et BuildFMDPnAT, leur intégration avec les algorithmes de planification présentés chapitre 3, ainsi que l'algorithme de réorganisation BuildTreeF dans le cadre de son utilisation avec l'algorithme SVI.

Afin de pouvoir exécuter des politiques dans les environnements utilisés dans la littérature des FMDPs, nous introduisons la notion d'*épisode*. En effet, dû à la nature de certain problème, l'agent peut se retrouver dans un état où il n'y a plus rien à faire, rendant difficile la génération de trajectoires ou bien l'évaluation de politiques. Par exemple, dans le problème *Coffee Robot*, pour toutes les actions, lorsque la propriétaire a un café, la probabilité qu'elle ait son café au pas de temps suivant est égale à 1 (le problème ne permet donc pas que la propriétaire puisse boire, jeter ou perdre son café). Ainsi, une fois que la propriétaire a son café, le robot n'a plus rien à faire. Afin d'éviter de telle situation, l'expérience de l'agent est découpée en épisode d'une durée T_e déterminée a priori. À chaque fin d'épisode, l'état de l'agent est réinitialisé. Sauf indication contraire, nous utilisons $T_e = 15$ pas de temps pour les expériences de ce chapitre.

Concernant les paramètres utilisés, l'ensemble des résultats de ce chapitre ont été obtenus en utilisant une valeur $\gamma = 0.99$ et, sauf indication contraire, une valeur de seuil pour l'apprentissage de $\tau_{\chi^2} = 30$. Nous discuterons du nombre de degrés de liberté associé à ce seuil dans la section 4.3.1 suivante. Sauf indication contraire, tous les algorithmes ont été implémentés avec le langage Java. Les résultats des expériences sont des moyennes de 10 exécutions et ont été calculés sur des PCs avec des processeurs Intel Pentium IV. La machine virtuelle Java utilisait un tas d'une limite de 1Go.

En premier lieu, la section 4.3.1 commence par analyser l'incidence de la valeur du seuil τ_{χ^2}, utilisé dans l'apprentissage de la fonction de transitions, sur la taille des représentations construites et la qualité de la politique obtenue par les algorithmes de planification. Ensuite, la section 4.3.2 utilise des problèmes de taille variable afin d'étudier l'incidence de la taille du problème sur la taille des représentations du FMDP, le temps requis pour les construire et l'incidence sur la qualité de la politique. Enfin, la section 4.3.3 utilise des problèmes de la littérature des FMDPs, considérés comme représentatifs de problèmes réels, afin d'analyser l'incidence de la taille de l'échantillon d'observations de l'agent dans son environnement sur la taille des représentations et la qualité de la politique calculée par les algorithmes de planification.

4.3.1 Incidence de la valeur du seuil

Les algorithmes d'apprentissage BuildTreeS et BuildFMDPnAT n'ont qu'un seul paramètre : le seuil τ_{χ^2}, utilisé par l'algorithme BuildTreeS afin de distinguer deux ensembles d'exemples significativement différents lors de l'apprentissage de la fonction de transition (cf. section 4.2.1).

Nous commençons donc par étudier l'incidence de la valeur ce seuil sur l'apprentissage. Pour cela, un échantillon d'observations dans le problème *Coffee Robot* est généré à partir d'une trajectoire exécutée par un agent utilisant une politique aléatoire avec une distribution uniforme sur l'ensemble des actions du problèmes. À partir d'un échantillon donné, plusieurs FMDPs sont générés, chacun avec une valeur du seuil τ_{χ^2} différente. Nous comparons ensuite plusieurs propriétés des FMDPs ainsi construits, notamment la taille de la représentation de leurs fonctions de transition et la qualité de leur politique optimale comparée à la politique optimale du problème.

Valeur du seuil et taille de l'échantillon

Dans un premier temps, nous proposons de comparer la taille de la représentation de la fonction de transition construit en fonction du seuil τ_{χ^2} et de la taille de l'échantillon d'observation. La figure 4.14 a été obtenue en utilisant l'algorithme BuildFMDPnAT et a pour but de montrer une vue globale du comportement de l'algorithme BuildTreeS de construction d'arbres de décision.

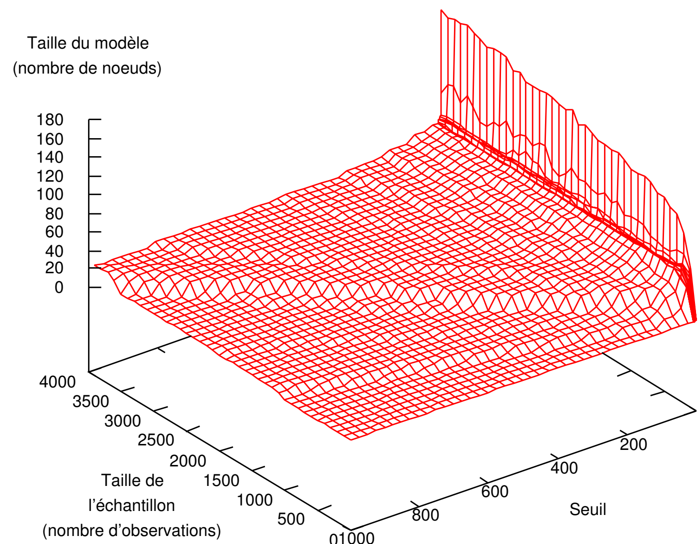


FIG. 4.14 – Taille de la représentation de la fonction de transitions construites en fonction du seuil τ_{χ^2} et de la taille de l'échantillon d'observations. On peut notamment observer que, plus la valeur de τ_{χ^2} est élevée, plus le nombre d'exemples nécessaires pour distinguer deux distributions de probabilités est élevé.

Nous pouvons distinguer trois zones différentes, correspondantes à trois grandes différences de taille du FMDP construit. La première zone correspond à une valeur de seuil τ_{χ^2} inférieure à 10. Cette zone regroupe les modèles pour lesquels le nombre de nœuds est supérieur à 100 dès que la taille de l'échantillon d'observations dépasse plusieurs dizaines d'exemples. Les deux autres zones sont séparées par une frontière commune découpant l'espace par une diagonale partant de $\tau_{\chi^2} > 10$ avec une taille d'échantillon inférieure à 500 et allant jusqu'à $\tau_{\chi^2} = 1000$ avec une taille d'échantillon de 4000 observations. Ces deux zones représentent les modèles contenant, respectivement,

environ 60 nœuds et, environ 10 nœuds.

Cette figure illustre le comportement de la limite du test χ^2, présentée dans la section 4.1.2 (page 84). Nous rappelons que cette limite montre que la valeur du test χ^2 augmente avec le nombre d'exemples dans les distributions lorsque les deux distributions de probabilités sont différentes.

Ceci est illustré par la frontière séparant les modèles contenant environ 60 nœuds et ceux contenant environ 10 nœuds. En effet, cette frontière met en valeur le fait que plus le seuil τ_{χ^2} est grand, plus le nombre d'observations nécessaires pour distinguer deux distributions de probabilités (et donc installer un nœud de décision dans l'arbre plutôt qu'une feuille) est élevé.

Si l'on considère la probabilité associée à la valeur du seuil τ_{χ^2}, on utilise un degré de liberté de 1 puisque les variables de *Coffee Robot* sont binaires. Lorsque la valeur χ^2 est supérieure à 10, la probabilité que deux distributions de probabilités soient significativement différentes est de 0.99, et lorsqu'elle est supérieur à 20, alors la probabilité est de 0.9999. On observe donc que, pour des petites variations de probabilités entre différentes valeurs de seuil τ_{χ^2}, l'impact sur le modèle construit pour un nombre de pas de temps donné peut être très important. C'est la raison pour laquelle nous utiliserons dans la suite de ce manuscrit directement la valeur du seuil τ_{χ^2} et non la probabilité associée.

Si l'on considère un nombre infini d'exemples, tous les FMDPs construits auront la même fonction de transition, quelque soit la valeur du seuil τ_{χ^2}. Par conséquent, τ_{χ^2} reflète plus un paramètre spécifiant la certitude avec laquelle un nœud de décision est installé dans l'arbre, plutôt qu'un paramètre indiquant la valeur de l'approximation avec laquelle la fonction de transition est construite.

Bien sûr, la taille d'un FMDP ne donne aucune indication sur la qualité avec laquelle ce FMDP représente le problème d'apprentissage par renforcement à résoudre, notamment quelle performance on peut attendre d'une politique optimale de ce FMDP dans le problème à résoudre. Dans ce but, nous décrivons dans la section suivante une mesure supplémentaire : l'erreur relative d'une politique.

Définition de l'erreur relative d'une politique

Afin d'estimer la qualité d'une politique π calculée à partir du FMDP appris par rapport à une politique optimale, nous pouvons calculer la différence de sa fonction de valeur V_π avec la fonction de valeur optimale V^*. Nous définissons donc l'erreur relative d'une politique, notée ξ_π, comme la moyenne de l'erreur relative $\Delta V = (V^* - V_\pi)/V^*$ entre sa fonction de valeur V_π et la fonction de valeur optimale V^* (avec nécessairement $V^* \geq V_\pi$).

Nous avons implémenté le calcul de l'erreur relative d'une politique uniquement avec les algorithmes de planification SPI et SVI, donc en utilisant les arbres de décision comme représentations

factorisées (Degris et al., 2006a). Ainsi, étant donné une politique $\text{Tree}[\pi]$, nous calculons sa fonction de valeur $\text{Tree}[V_\pi]$ avec l'algorithme SPE (section 3.2.5, page 53) et en utilisant les fonctions de transition et de récompense définissant le problème à résoudre.

Ensuite, à partir de l'ensemble $S_{\Delta V} = \{\text{Tree}[V^*], \text{Tree}[V_\pi]\}$, la différence relative $\text{Tree}[\Delta V_\pi]$ est calculée avec l'opérateur $\underline{\text{Merge}}(S_{\Delta V})$ et en utilisant comme fonction de combinaison l'erreur relative ΔV. Enfin, l'erreur relative ξ_π de la politique $\text{Tree}[\pi]$ est calculée de la façon suivante :

$$\xi_\pi = \frac{\sum_{l \in \text{Tree}[\Delta V_\pi]} \Delta V_l \cdot S_l}{\prod_{i \in |X|} |\underline{\text{Dom}}(X_i)|} \tag{4.10}$$

avec ΔV_l le contenu de la feuille l et S_l la taille du sous-espace d'états caractérisé par l.

Cette mesure représentant une erreur relative par rapport à une politique optimale, une valeur ξ_π proche de 0 pour une politique π signifie que la performance de cette politique est proche de celle de la politique optimale. Au contraire, une valeur ξ_π proche de 1 signifie que la politique π est mauvaise comparée à une politique optimale.

Bien que nous n'ayons implémenté cet algorithme que pour les algorithmes SPI et SVI et en utilisant les arbres de décision comme représentation, l'erreur relative d'une politique utilisant des représentations factorisées est facilement transposable aux autres algorithmes de planification utilisant d'autres structures de données. Cependant, il est important de noter qu'une représentation explicite de la politique est nécessaire, rendant son utilisation impossible pour les grands problèmes où la représentation de la politique croît de façon exponentielle avec le nombre de variables d'état.

Incidence du seuil sur l'erreur relative de la politique

Afin d'analyser l'incidence du seuil τ_{χ^2} sur l'erreur relative de la politique, un protocole similaire à celui décrit ci-dessus est utilisé sur le problème *Coffee Robot* et une version bruitée de ce problème. La version bruitée de *Coffee Robot* reprend la définition du problème et ajoute un bruit de 5% sur toutes les transitions déterministes. Ainsi, une probabilité conditionnelle $P(X' = 1|s) = 1.0$ deviendra $P(X' = 1|s) = 0.95$ (et $P(X' = 0|s) = 0.05$), de même pour une probabilité $P(X' = 0|s) = 1.0$ qui deviendra $P(X' = 0|s) = 0.95$ (et $P(X' = 1|s) = 0.05$).

À partir d'une politique aléatoire, une trajectoire dans le problème est générée. À partir de cette trajectoire et des deux algorithmes d'apprentissage $\underline{\text{BuildFMDP}}$ et $\underline{\text{BuildFMDPnAT}}$, différentes valeurs du seuil τ_{χ^2} sont utilisées pour construire un FMDP $\tilde{\mathcal{F}}$ représentant le problème d'apprentissage. À partir de $\tilde{\mathcal{F}}$, nous utilisons l'algorithme SVI pour calculer la politique $\pi^*_{\tilde{\mathcal{F}}}$ optimale associée. Ensuite, l'erreur relative de cette politique $\xi_{\pi^*_{\tilde{\mathcal{F}}}}$ est estimée en utilisant la méthode décrite ci-dessus. La figure 4.15 montre les résultats pour $\tau_{\chi^2} \in [0; 20]$ et pour un échantillon de 500 observations (correspondant à la trajectoire d'une politique aléatoire de 500 pas de temps).

Plusieurs observations peuvent être effectuées à partir de ces résultats. Premièrement, pour les valeurs du seuil τ_{χ^2} très faible ($\tau_{\chi^2} \leq 4$), bien que la taille des modèles construits soit très impor-

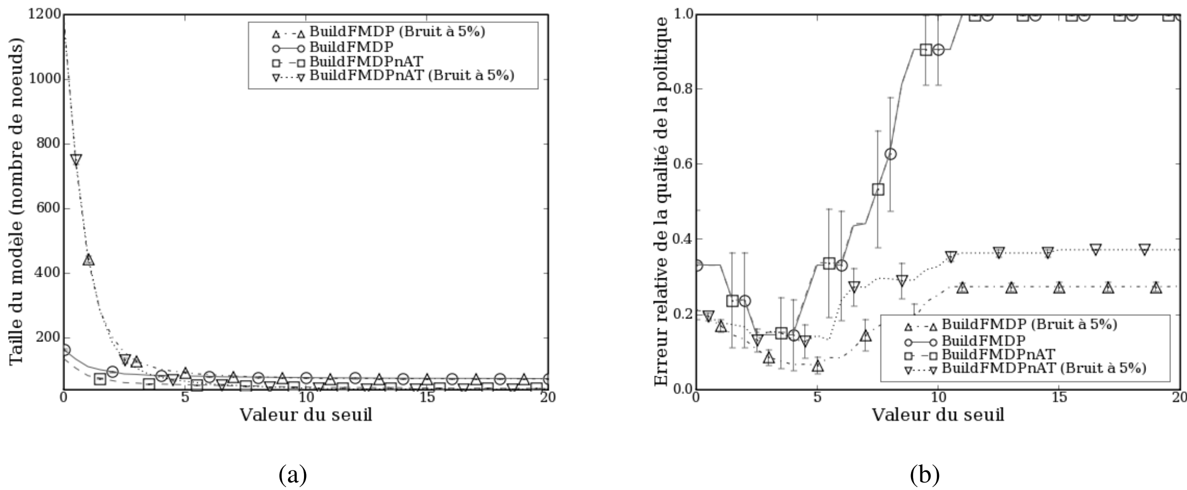


FIG. 4.15 – Taille de la représentation de la fonction de transition (figure a) et erreur relative de la politique calculée à partir de cette représentation (figure b) à partir d'une trajectoire de 500 pas de temps. La taille du modèle et la qualité de la politique sont très sensibles à la valeur du seuil τ_{χ^2}.

tante (dépassant les 1000 nœuds), l'erreur relative $\xi_{\pi^*_{\tilde{\mathcal{F}}}}$ est, au mieux, similaire aux modèles plus compacts du problème correspondant à des valeurs $\tau_{\chi^2} > 4$, au pire plus importante. En effet, nous pouvons observer que cette erreur est minimale pour une valeur approximative de $\tau_{\chi^2} \approx 5$. Deuxièmement, lorsque le seuil dépasse cette valeur, la taille du modèle construit continue de diminuer légèrement, alors que l'erreur relative $\xi_{\pi^*_{\tilde{\mathcal{F}}}}$ augmente. Troisièmement, notons que l'algorithme BuildFMDP permet d'obtenir des politiques avec une erreur relative moins élevée que celle obtenue à partir de l'algorithme BuildFMDPnAT qui, cependant, utilise une représentation plus compacte.

La figure 4.16 montre des résultats à partir du même protocole mais utilisant un échantillon de 4000 observations (correspondant à la trajectoire d'une politique aléatoire de 4000 pas de temps). Plusieurs différences notables sont à observer par rapport aux résultats obtenus à partir de 500 observations.

En premier lieu, les erreurs relatives des politiques obtenues pour le problème *Coffee Robot* bruité ont nettement diminué. De plus, à la fois la taille de la représentation de la fonction de transition et l'erreur relative des politiques sont nettement moins sensibles à la valeur du seuil τ_{χ^2}. En effet, pour $\tau_{\chi^2} > 5$, le nombre de nœuds utilisé pour représenter le problème reste stable (environ 55 nœuds pour le problème *Coffee Robot* et, 80 nœuds pour la version bruitée), de même que l'erreur relative des politiques qui reste inférieure à 3%, indiquant que le FMDP appris est représentatif du problème *Coffee Robot* (bruité ou non) à résoudre.

Enfin, un point commun avec les résultats obtenus pour 500 observations est le comportement des algorithmes pour des valeurs du seuil τ_{χ^2} faible. En effet, alors que la taille de la représentation

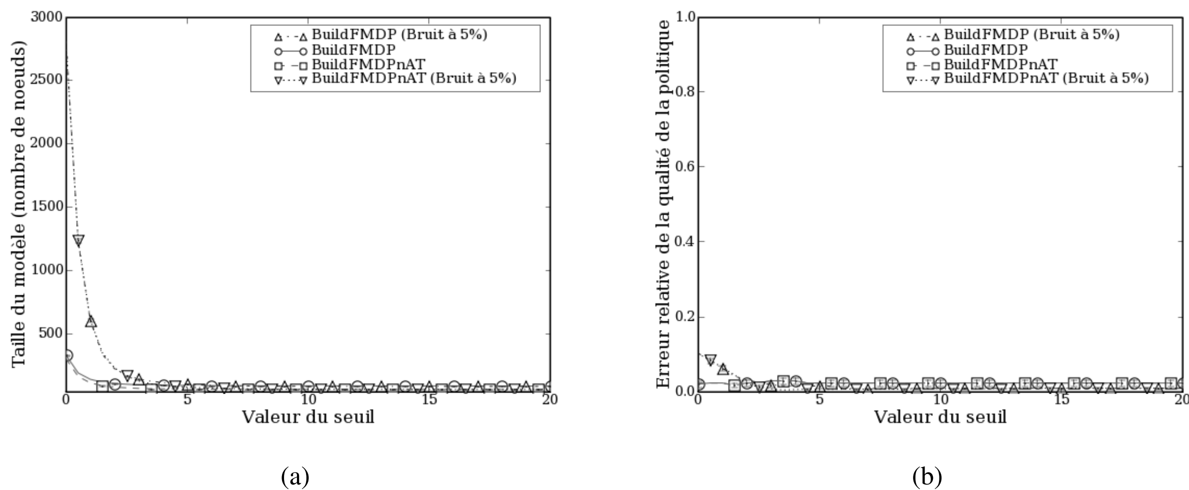


FIG. 4.16 – Taille de la représentation de la fonction de transition (figure a) et erreur relative de la politique calculée à partir de cette représentation (figure b) à partir d'une trajectoire de 4000 pas de temps. La taille du modèle et la qualité de la politique sont très sensibles à la valeur du seuil τ_{χ^2} lorsque celui-ci a une valeur faible ($\tau_{\chi^2} < 10$).

de la fonction de transition augmente de façon importante, l'erreur relative reste identique pour le problème *Coffee Robot* et, augmente lorsque le problème est bruité. Enfin, notons que l'algorithme BuildFMDPnAT construit une représentation plus compacte que l'algorithme BuildFMDP et, contrairement aux résultats précédents, obtient des politiques d'une qualité similaire à celles obtenues avec BuildFMDP.

Ces résultats illustrent donc que la valeur du seuil τ_{χ^2} a le même impact sur la taille de la représentation que sur la qualité des politiques optimales calculées à partir du FMDP. En effet, lorsque la valeur de τ_{χ^2} est trop élevée et que le nombre d'observation à apprendre est trop faible, la certitude associée au test effectué par un nœud de décision pour séparer les distributions de probabilités dans la fonction de transition est trop faible. Par conséquent, certains nœuds ne sont pas installés, dégradant ainsi la politique calculée à partir de ce FMDP.

Cependant, nous avons observé que, pour les même valeurs de seuil, avec des observations supplémentaires, la politique calculée à partir du FMDP s'améliorent sensiblement. En effet, la valeur du test χ^2 augmente lorsque le nombre d'exemples dans les distributions augmente (voir la limite du test χ^2 section 4.1.2, page 84). Par conséquent, des nœuds supplémentaires sont installés dans les arbres utilisés pour représenter la fonction de transition, améliorant son exactitude. Une fois de plus, ces résultats montrent que le seuil τ_{χ^2} représente un paramètre indiquant une certitude avec laquelle deux distributions peuvent être séparées, plutôt que la définition d'une approximation

de la représentation de la fonction de transition.

De plus, il est important de noter que lorsque le seuil τ_{χ^2} est très faible, limitant ainsi le pré-élagage réalisé par le test statistique, un grand nombre de nœuds de décision inutiles est installé. L'installation de tels nœuds a un impact négatif sur la qualité de la politique puisque, à cause du nombres supplémentaires de probabilités à évaluer, un plus grand nombre d'exemples est nécessaire. Cette propriété est notamment illustré dans le problème *Coffee Robot* bruité avec un petit échantillon d'observations (figure 4.15). Le même résultat est obtenu sur un autre problème dans Degris et al. (2006a).

4.3.2 Incidence de la taille du problème

Afin d'analyser l'incidence de la taille du problème sur la taille du modèle construit par les algorithmes d'apprentissage, le temps nécessaire pour construire les représentations et l'erreur relative de la politique pour une taille d'échantillon fixée, nous utilisons deux problèmes, nommés *Linear* et *Expon*, définis par Boutilier et al. (2000) afin d'illustrer le pire cas et le meilleur cas pour les algorithmes SPI, SVI et SPUDD.

Définition des problèmes

Ces deux problèmes comportent n variables binaires $X_1, \ldots, X_n$ et n actions. L'état terminal d'un épisode est défini lorsque toutes les variables X_k sont égales à Vrai. La récompense du problème est égale à 1 dans cet état terminal. L'ensemble des états initiaux regroupe tous les états possibles, sauf l'état terminal. Comme le montre la figure 4.17, la différence entre les deux problèmes *Linear* et *Expon* se situe au niveau de la définition de la fonction de transition.

Pour le problème *Linear* (figure 4.17(a)), l'action a_k assigne la variable X_k à Vrai si toutes les variables X_i, avec $i < k$, précédentes sont égales à Vrai, et toutes les variables X_i, avec $i > k$, suivantes à Faux. Ainsi, si l'état du problème *Linear* est considéré comme un nombre binaire, la politique optimale consiste alors à sélectionner l'action a_k assignant à Vrai le bit X_k de plus haut poids ayant l'ensemble de ces prédécesseurs X_i, avec $i < k$, à Vrai (figure 4.18(a)). De plus, ce problème peut être bruité (Boutilier et al., 2000) : un bruit de k% indique une chance de k% que l'action a_k assigne la variable X_{k-1} à Faux. Le problème *Linear* correspond au meilleur cas pour les algorithmes SPI, SVI et SPUDD puisque la représentation de la fonction de valeur optimale ne requiert que $n + 1$ feuilles. Le même nombre de feuilles est nécessaire pour définir une politique optimale à ce problème.

Concernant le problème *Expon* (figure 4.17(b)), l'action a_k assigne la variable X_k à Vrai si toutes les variables X_i, avec $i < k$, précédentes sont égales à Vrai. De plus, l'action a_k assigne toutes les variables X_i, avec $i < k$, à l'état Faux. Ainsi, si l'état du problème *Expon* est considéré comme un nombre binaire, de la même façon que pour le problème *Linear*, la politique optimale consiste

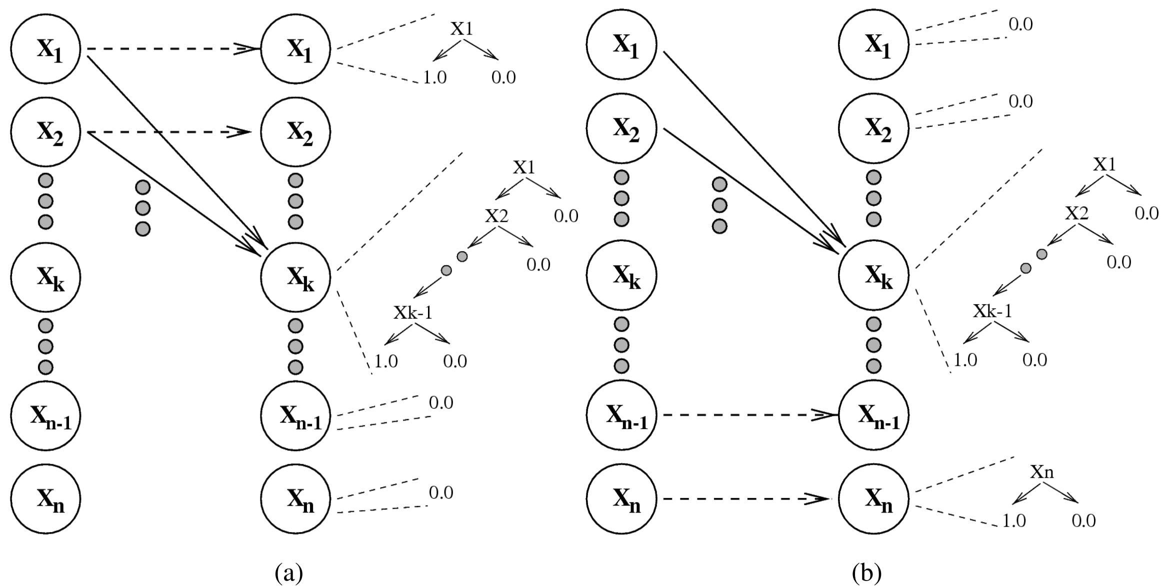


FIG. 4.17 – Description des distributions de probabilités conditionnelles pour la k^{ième} action dans les problèmes *Linear* (figure a) et *Expon* (figure b) avec n variables. Figure extraite de Boutilier et al. (2000) (page 40 et 42).

à sélectionner l'action a_k assignant à Vrai le bit X_k de plus haut poids ayant l'ensemble de ces prédécesseurs X_i, avec $i < k$, à Vrai (figure 4.18(b)). Cependant, contrairement au problème *Linear*, une telle politique passera par tous les nombres binaires de façon séquentielle. Par conséquent, puisque nous utilisons le critère de récompense actualisée pour définir la fonction de valeur, celle-ci nécessite une valeur différente pour chaque nombre binaire. Le problème *Expon* correspond donc au pire cas pour les algorithmes SPI, SVI et SPUDD puisque la représentation de la fonction de valeur optimale requiert 2^n feuilles. Notons qu'au contraire, à l'instar du problème *Linear*, seulement $n+1$ feuilles sont requises pour définir une politique optimale à ce problème. Enfin, pour le problème *Expon*, le bruit est ajouté de la façon suivante : un bruit de k% indique une chance de k% que l'action a_k assigne la variable X_{k-1} à faux. Cependant, l'action a_k assigne quand même toutes les variables X_i, avec $i < k$, à l'état faux. Dans ce cas, il est alors nécessaire de recommencer toute la séquence de nombres afin d'arriver à un état similaire.

Incidence de la taille des problèmes sur la taille du modèle appris et temps de calcul requis par l'apprentissage

Afin d'analyser l'incidence de la taille des problèmes sur la taille du modèle appris et le temps de calcul requis par l'apprentissage, le protocole suivant est utilisé : pour une taille donnée de problème, un échantillon de 20 000 observations est généré à partir de la trajectoire d'un agent exécutant une politique aléatoire dans les deux problèmes *Linear* et *Expon*, sans bruit et avec un

Linear :

	Action	État
t		0000
t+1	0	0001
t+2	1	0011
t+3	2	0111
t+4	3	1111
t+5	3	1111
	...	

(a)

Expon :

	Action	État
t		0000
t+1	0	0001
t+2	1	0010
t+3	0	0011
t+4	2	0100
t+5	0	0101
	...	

(b)

FIG. 4.18 – Exécution des politiques optimales dans les problèmes *Linear* (figure a) et *Expon* (figure b) avec 4 variables.

bruit de 20%. Pour une valeur $\tau_{\chi^2} = 30$, les deux algorithmes d'apprentissage BuildFMDP et BuildFMDPnAT sont utilisés pour générer un FMDP à partir duquel une politique optimale est calculée. Les figures 4.19 et 4.20 montrent, respectivement, le nombre de nœuds utilisés et le temps nécessaire à l'apprentissage pour construire un modèle de la fonction de transition des problèmes *Linear* et *Expon*.

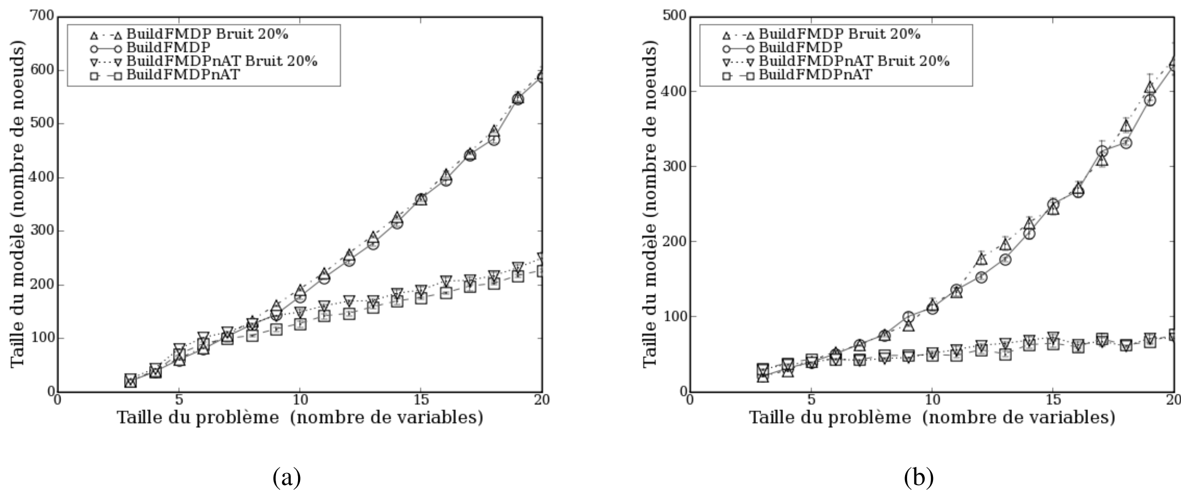


FIG. 4.19 – Incidence de la taille des problèmes *Linear* (figure a) et *Expon* (figure b) sur la taille du modèle. Pour ces deux problèmes, alors que le nombre d'états possibles croît de façon exponentielle, la taille du modèle augmente mais avec une complexité moindre.

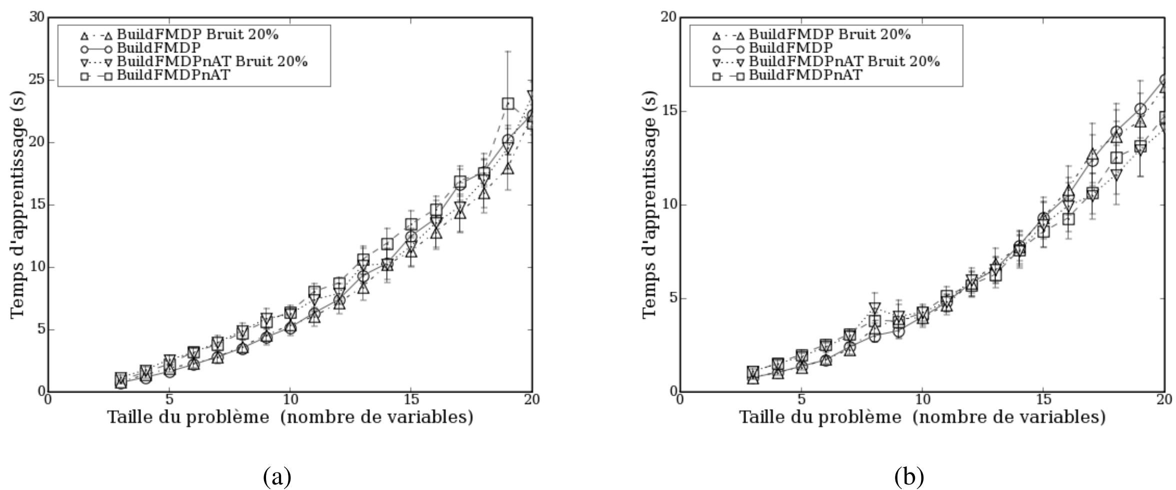


FIG. 4.20 – Incidence de la taille des problèmes *Linear* (figure a) et *Expon* (figure b) sur le temps de calcul de l'apprentissage. Pour ces deux problèmes, alors que le nombre d'états possibles croît de façon exponentielle, le temps d'apprentissage augmente mais avec une complexité moindre.

Pour le problème *Linear*, alors que la taille du problème croît de façon exponentielle de 4 à 1 048 576 états possibles, cette augmentation ne se retrouve ni dans la taille du modèle appris (qui varie de 10 à 600 nœuds, figure 4.19(a)), ni dans le temps nécessaire à l'apprentissage (qui varie de 1s à 25s, figure 4.20(a)).

On peut observer que la présence de bruit dans le problème a peu d'influence sur le temps d'apprentissage (au plus une seconde) et sur le taille du modèle (moins de 50 nœuds). De plus, une différence notable apparaît entre les deux algorithmes BuildFMDP et BuildFMDPnAT. En effet, pour un temps d'apprentissage légèrement supérieur (maximum 2 secondes), le modèle construit par l'algorithme BuildFMDPnAT est plus compact que celui de BuildFMDP, respectivement moins de 250 nœuds contre plus de 550 nœuds.

Des résultats similaires sont obtenus pour le problème *Expon* concernant la taille du modèle appris (figure 4.19(b)) et le temps nécessaire à l'apprentissage (figure 4.20(b)).

Incidence de la taille des problèmes sur l'erreur relative de la politique

La figure 4.21 montre l'influence de la taille de ces deux problèmes sur l'erreur relative de la politique. Pour le problème *Linear*, on peut observer que l'erreur relative de la politique optimale générée à partir du FMDP appris est rapidement égale à 1 (figure 4.21(a)), à partir de problèmes composés de 8 variables binaires ($2^8 = 256$ états possibles). Rapidement, les méthodes d'apprentissage BuildFMDP et BuildFMDPnAT ne permettent donc plus de construire un FMDP représentatif du problème et permettant de trouver une solution au problème d'apprentissage par renforcement (malgré la taille de l'échantillon de 20 000 observations).

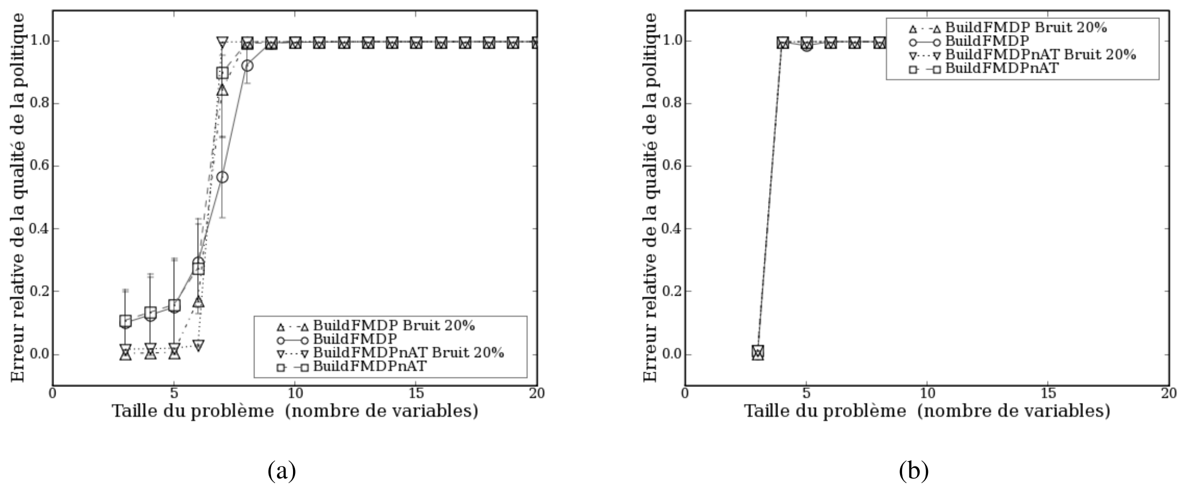


FIG. 4.21 – Incidence de la taille des problèmes *Linear* (figure a) et *Expon* (figure b) sur l'erreur relative de la politique. À partir de tailles de problème petites, la politique générée à partir du FMDP appris a une erreur relative de 1. L'apprentissage est donc incomplet et ne permet donc pas de calculer une solution permettant de résoudre le problème.

Le phénomène est amplifié pour le problème *Expon*. En effet, l'erreur relative de la politique optimale générée atteint 1 (figure 4.21(b)) à partir du problème de taille 5 ($2^5 = 32$ états possibles), indiquant que même pour des petits problèmes, notre méthode d'apprentissage ne permet pas de construire une politique proche de la politique optimale. De plus, on peut observer une différence entre les politiques générées à partir des FMDPs construits par les deux algorithmes BuildFMDP et BuildFMDPnAT. En effet, dès que le problème a une taille supérieure à 5, la politique générée à partir du modèle appris avec l'algorithme BuildFMDPnAT est inopérante, ce qui n'est pas le cas de la politique générée à partir du modèle appris par BuildFMDP qui reste opérationnelle jusqu'au problème de taille 8.

La mesure de l'erreur relative de la politique optimale générée à partir du FMDP appris indique à quel point cette politique sera performante dans le problème donné, comparée à une politique optimale de ce problème. Cependant, nous pouvons constater que, pour certains problèmes, notamment *Linear* et *Expon*, cette information est très pauvre. En effet, figure 4.21, cette mesure est quasiment binaire, indiquant uniquement si la politique calculée à partir du FMDP appris peut être opérationnelle ou pas.

Bien que cette information soit utile, son problème majeur est qu'elle ne différencie pas l'apprentissage de la fonction de transition avec l'apprentissage de la fonction de récompense. En effet, alors que l'apprentissage de la fonction de transition peut être exact, si celui de la fonction de récompense n'est pas correct, alors la politique générée sera inopérante et aura une erreur relative proche de 1. Or, il peut être intéressant de distinguer ce cas avec celui où l'apprentissage de la fonc-

tion de transition et de la fonction de récompense sont incorrectes. C'est la raison pour laquelle nous introduisons une nouvelle mesure complémentaire, notée $\mathcal{Q}_{\chi^2}$, comparant la fonction de transition du FMDP appris à la fonction de transition définissant le problème d'apprentissage.

Exactitude de l'apprentissage de la fonction de transitions

La mesure d'exactitude $\mathcal{Q}_{\chi^2}$ (Degris et al., 2006a) est complémentaire à la mesure de l'erreur relative de la politique, puisqu'elle permet de quantifier l'apprentissage de façon indépendante de la méthode de planification utilisée et de la fonction de récompense. D'un point de vue général, cette mesure représente la moyenne des probabilités d'accepter ou non l'hypothèse d'indépendance pour chaque couple état/action entre la distribution de probabilités théorique et celle apprise. En utilisant les arbres de décision comme représentations des distribution de probabilités conditionnelles, cette mesure se calcule de la façon suivante :

$$\mathcal{Q}_{\chi^2} = \frac{\sum_{a \in A} \sum_{X_i \in X} \sigma_{a,i}}{|A| \cdot \prod_{X_i \in X} |\underline{\text{Dom}}(X_i)|} \tag{4.11}$$

avec $|S|$ représentant le cardinal de l'ensemble S et $\sigma_{a,i}$ défini d'après l'algorithme présenté figure 4.22.

Alors que le produit $|A| \cdot \prod_{X_i \in X} |\underline{\text{Dom}}(X_i)|$ représente le nombre de couples état/action dans le problème, la variable $\sigma_{a,i}$ représente la somme des probabilités d'indépendance $Q(\chi^2_{(l_{\mathcal{F}}, l_{\tilde{\mathcal{F}}})})$ pour une action a donnée et une variable X_i donnée. La somme $\sum_{a \in A} \sum_{X_i \in X} \sigma_{a,i}$ représente donc la somme des probabilités d'indépendance pour l'ensemble des couples état/action du problème.

Afin d'exploiter les indépendances relatives aux contextes, $\sigma_{a,i}$ est calculée en utilisant l'opérateur $\underline{\text{Merge}}$ sur les arbres de décision (cf. section 3.2.2). On peut remarquer que l'opérateur $\underline{\text{Merge}}$ est seulement utilisé afin de calculer, via la fonction de combinaison, la somme $\sigma_{a,i}$ sur l'ensemble des partitions de $\text{Tree}\left[P_a^{\mathcal{F}}(X_i'|s)\right]$ et $\text{Tree}\left[P_a^{\tilde{\mathcal{F}}}(X_i'|s)\right]$. L'arbre de décision résultant de l'opérateur n'est pas utilisé. Ainsi, pour chaque distribution de probabilités $P_a^{\tilde{\mathcal{F}}}(X_i')$ à évaluer, le calcul de $Q(\chi^2_{(l_{\mathcal{F}}, l_{\tilde{\mathcal{F}}})})$ est effectué avec la distribution de probabilités théorique $P_a^{\mathcal{F}}(X_i')$ correspondante. Cette probabilité est alors ajoutée à la somme $\sigma_{a,i}$ en pondérant par la taille de l'espace d'état (correspondant au nombre de couples état/action) représenté par $P_a^{\tilde{\mathcal{F}}}(X_i')$.

Enfin, la probabilité d'accepter ou de rejeter l'hypothèse d'indépendance entre les deux distributions $P_a^{\tilde{\mathcal{F}}}(X_i')$ et $P_a^{\tilde{\mathcal{F}}}(X_i')$ est calculée par une approximation de la distribution de probabilités du χ^2 (Press et al., 1992). Lorsque ce calcul est impossible, principalement lorsque la distribution de probabilités $P_a^{\tilde{\mathcal{F}}}(X_i')$ de l'échantillon contient des valeurs de X_i' non présentes dans la distribution de probabilités théorique $P_a^{\mathcal{F}}(X_i')$, alors on définit $Q(\chi^2_{(l_{\mathcal{F}}, l_{\tilde{\mathcal{F}}})}) = 0$. Ce cas peut arriver lorsqu'un nœud de décision n'a pas été installé pour séparer l'espace pour, par exemple, des transitions déterministes avec $X_i' = 0$ et $X_i' = 1$.

Entrée(s) : Le FMDP $\mathcal{F}$ définissant le problème, le FMDP $\tilde{\mathcal{F}}$ à évaluer, une variable X_i, une action a **Sortie(s) :** Somme des valeurs d'exactitudes $\sigma_{a,i}$

---

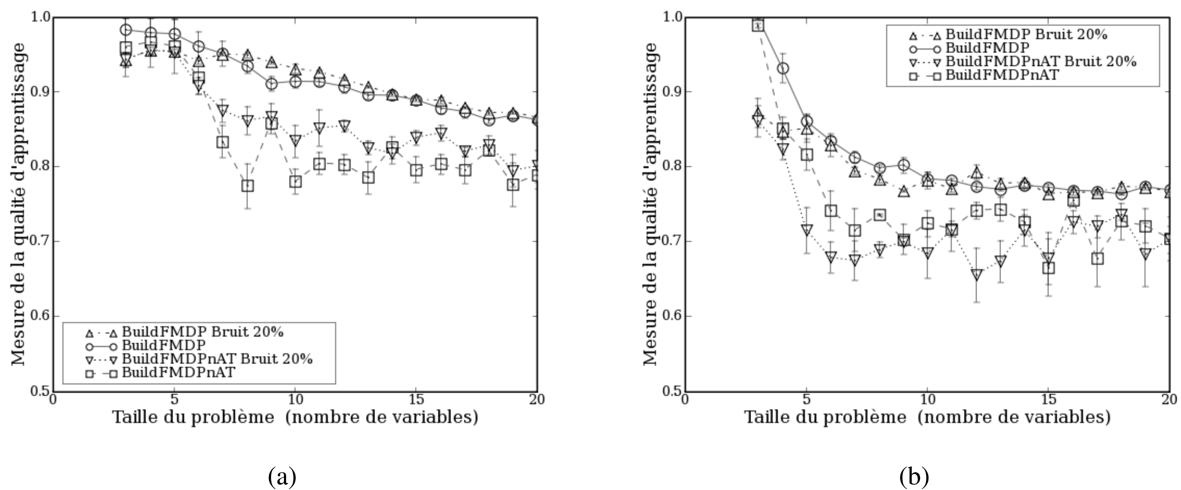
1. $\sigma_{a,i} = 0$
2. Soit Tree $\left[P_a^{\mathcal{F}}(X'_i|s)\right]$ l'arbre représentant la distribution de probabilités conditionnelle $P_a^{\mathcal{F}}(X'_i|s)$ dans le FMDP $\mathcal{F}$ définissant le problème
3. Soit Tree $\left[P_a^{\tilde{\mathcal{F}}}(X'_i|s)\right]$ l'arbre représentant la distribution de probabilités conditionnelle $P_a^{\tilde{\mathcal{F}}}(X'_i|s)$ dans le FMDP $\tilde{\mathcal{F}}$ à évaluer
4. Merge($\{$Tree $\left[P_a^{\mathcal{F}}(X'_i|s)\right]$, Tree $\left[P_a^{\tilde{\mathcal{F}}}(X'_i|s)\right]\}$) en utilisant comme fonction de combinaison :
   (a) Soit la feuille $l_{\mathcal{F}} \in$ Tree $\left[P_a^{\mathcal{F}}(X'_i|s)\right]$ et contenant la distribution de probabilités $P_a^{\mathcal{F}}(X'_i)$ théorique
   (b) Soit la feuille $l_{\tilde{\mathcal{F}}} \in$ Tree $\left[P_a^{\tilde{\mathcal{F}}}(X'_i|s)\right]$ et contenant la distribution de probabilités $P_a^{\tilde{\mathcal{F}}}(X'_i)$ de l'échantillon
   (c) Soit $Q(\chi^2_{(l_{\mathcal{F}},l_{\tilde{\mathcal{F}}})})$ la probabilité d'accepter l'hypothèse d'indépendance, calculé à partir du test statistique χ^2 avec les distributions $P_a^{\tilde{\mathcal{F}}}(X'_i)$ et $P_a^{\tilde{\mathcal{F}}}(X'_i)$
   (d) Soit $S_{l_{\tilde{\mathcal{F}}}}$ la taille de l'espace d'états représenté par la feuille $l_{\tilde{\mathcal{F}}}$
   (e) $\sigma_{a,i} = \sigma_{a,i} + S_{l_{\tilde{\mathcal{F}}}} \cdot Q(\chi^2_{(l_{\mathcal{F}},l_{\tilde{\mathcal{F}}})})$
5. Retourner $\sigma_{a,i}$

FIG. 4.22 – Calcul de $\sigma_{a,i}$ utilisé dans la mesure d'exactitude d'une fonction de transitions.

Incidence de la taille des problèmes sur l'exactitude de la fonction de transition apprise

La figure 4.23 illustre l'incidence de la taille des problèmes *Linear* et *Expon* sur l'exactitude de la fonction de transition construite par apprentissage. Le protocole utilisé est identique à celui mis en œuvre pour les analyses précédentes.

Pour le problème *Linear* (figure 4.23(a)), nous pouvons remarquer que l'influence de la taille du problème est bien moindre que dans le résultat précédent, concernant l'erreur relative de la politique dans ce problème (figure 4.21(a), page 112). D'une part, bien que la taille du problème augmente de façon exponentielle, la mesure décroît linéairement. Ce résultat, mis en relation avec le résultat précédent concernant l'erreur relative de la politique, indique ainsi que le problème d'apprentissage se situe donc plutôt au niveau de la fonction de récompense, plutôt que de la fonction de transition. D'autre part, on peut remarquer que, pour les deux méthodes d'apprentissage BuildFMDP et BuildFMDPnAT, le bruit dans le problème a peu d'effet sur la qualité de l'apprentissage. Enfin, on peut noter une différence nette entre l'exactitude de l'apprentissage de l'algorithme BuildFMDP qui est meilleur que celui de BuildFMDPnAT. Cette différence peut être en rapport avec la différence observée sur l'erreur relative des politiques 4.21 entre les deux algorithmes.



(a) (b)

FIG. 4.23 – Incidence de la taille des problèmes *Linear* (figure a) et *Expon* (figure b) sur l'exactitude de la fonction de transition apprise par les algorithmes BuildFMDP et BuildFMDPnAT. Alors que la taille des problèmes augmente exponentiellement, une telle variation ne s'observe pas sur la qualité de l'apprentissage.

Concernant le problème *Expon* (figure 4.23(b)), des observations similaires peuvent être réalisées. On peut noter que la différence entre les deux algorithmes d'apprentissage est plus importante, même si cette différence ne se retrouve pas dans les résultats concernant l'erreur relative de la politique. On peut aussi noter un effet un peu plus important du bruit sur la qualité de l'apprentissage.

Les résultats concernant l'apprentissage d'un FMDP dans les problèmes *Expon* et *Linear* font apparaître plusieurs propriétés et problématiques intéressantes. La première d'entre elles est le fait que les algorithmes BuildFMDP et BuildFMDPnAT sont capables d'exploiter la structure du problème lors de la construction d'un FMDP. Par conséquent, la complexité de l'apprentissage de la fonction de transition d'un problème d'apprentissage par renforcement ne dépend plus du nombre de couples état/action existant dans ce problème mais plutôt de sa structure.

Deuxièmement, la contradiction existant entre les résultats concernant l'erreur relative de la politique (l'erreur de la politique devient importante même pour des problèmes de petite taille) et l'exactitude de l'apprentissage (l'exactitude du modèle appris est peu sensible à la taille du problème, alors que la taille de l'échantillon reste fixe) illustre un problème bien connu en apprentissage : la pertinence de l'échantillon à partir duquel l'algorithme apprend.

En effet, dans les problèmes *Linear* et *Expon*, une seule récompense est présente dans le problème et ne peut être obtenu qu'à partir d'un seul état. Dans le cadre d'un apprentissage hors-ligne, comme c'est le cas pour les résultats présentés dans cette section, si cet état n'est pas présent dans l'échantillon, alors la représentation de la fonction de récompense sera nécessairement inadéquate,

quelque soit l'algorithme d'apprentissage utilisé. Par conséquent, même si la fonction de transition du problème a été apprise correctement, la politique générée à partir d'un FMDP ayant une mauvaise fonction de récompense, sera elle aussi mauvaise. La mauvaise qualité des échantillons d'observation peut aussi bien venir de la nature du problème que du fait que les échantillons sont obtenus à partir de la trajectoire d'un agent exécutant une politique purement aléatoire. Dans le cadre d'un apprentissage en ligne, trouver la récompense du problème est directement relié au problème de l'exploration dirigée dans lequel nous reviendrons au chapitre 6 (page 155).

4.3.3 Incidence de la taille de l'échantillon d'observations

Les problèmes *Expon* et *Linear*, utilisés dans la section précédente, représente des cas extrêmes, aussi bien concernant la représentation de leur fonction de valeur optimale que pour l'exploration nécessaire afin de résoudre ces problèmes. Les résultats présentés dans ce chapitre concernent des problèmes considérés comme "typiques" des problèmes réels. Afin d'analyser l'incidence de la taille de l'échantillon d'observations sur l'apprentissage, nous utilisons les trois méthodes de planification SVI, SPUDD et la programmation linéaire. Le but de cette analyse est de déterminer la qualité de la politique obtenue à partir d'un FMDP construit en fonction de la taille de l'échantillon d'observations.

Le protocole expérimental est similaire aux précédents : un échantillon d'observations est généré à partir de la trajectoire d'un agent exécutant une politique aléatoire. Pour plusieurs longueurs de cette trajectoire, on calcule les résultats suivants à partir du résultat des algorithmes BuildFMDP et BuildFMDPnAT : le temps requis par les algorithmes d'apprentissage pour construire le FMDP représentant le problème, la taille de la représentation de la fonction de transitions et l'exactitude de la fonction de transitions du FMDP appris.

Nous n'utilisons pas l'erreur relative de la politique puisque nous n'avons pas, à la fois pour des raisons techniques et pour des raisons de complexité trop importante pour certains problèmes, implémenté son calcul pour les algorithmes de planification SPUDD et FactoredLPA. L'erreur relative de la politique n'étant pas disponible et, afin d'évaluer la qualité de la politique générée à partir d'un FMDP appris, nous avons substitué le calcul de cette erreur par le calcul de la récompense actualisée, étant donné une politique. À partir d'un FMDP $\tilde{\mathcal{F}}$ appris, une politique optimale $\pi^*_{\tilde{\mathcal{F}}}$ est calculée. Ensuite, cette politique est évaluée dans le problème pendant $T = 4000$ pas de temps. On retient alors la récompense actualisée R^γ_T obtenue par $\pi^*_{\tilde{\mathcal{F}}}$ au bout de cette trajectoire. À un pas de temps t, la récompense actualisée est calculée de la façon suivante :

$$R^\gamma_t = r_t + \gamma R^\gamma_{t-1} \tag{4.12}$$

avec r_t la récompense immédiate obtenue par l'agent à l'instant t et R^γ_{t-1} la récompense actualisée obtenue au pas de temps précédent. On peut remarquer que, contrairement à la mesure de l'erreur

relative de la politique, le calcul de R_T^γ ne nécessite pas une représentation explicite de $\pi^*_{\tilde{\mathcal{F}}}$ qui peut être calculée en ligne à partir des fonctions de valeur d'action.

Le problème *Factory*

Nous commençons par le problème *Factory*, utilisé avec l'algorithme de planification SVI. Ce problème, extrait de la littérature concernant l'ordonnancement de tâches, est utilisé par Dearden and Boutilier (1997) pour illustrer les performances des algorithmes de planification sur un cas typique.

Le problème concerne un robot qui doit assembler deux pièces A et B l'une avec l'autre. En fonction des compétences, des outils disponibles et de la demande, les deux pièces doivent être travaillées (nettoyées, polies, peintes, ...) avant d'être connectées. L'espace d'action du robot comporte 14 actions stochastiques qui sont :

- $\mathcal{S}\mathrm{hapeA}$, $\mathcal{S}\mathrm{hapeB}$: modeler l'objet A, B ;
- $\mathcal{D}\mathrm{rillA}$, $\mathcal{D}\mathrm{rillB}$: percer l'objet A, B ;
- $\mathcal{D}\mathrm{ipA}$, $\mathcal{D}\mathrm{ipB}$: tremper (peindre) l'objet A, B ;
- $\mathcal{S}\mathrm{prayA}$, $\mathcal{S}\mathrm{prayB}$: vaporiser (peindre) l'objet A, B ;
- $\mathcal{H}\mathrm{andPaintA}$, $\mathcal{H}\mathrm{andPaintB}$: peindre à la main l'objet A, B ;
- $\mathcal{P}\mathrm{olishA}$, $\mathcal{P}\mathrm{olishB}$: polir l'objet A, B ;
- $\mathcal{B}\mathrm{olt}$: boulonner les deux objets ensemble ;
- $\mathcal{G}\mathrm{lue}$: coller les deux objets ensemble.

L'ensemble de l'espace d'états comporte 17 variables binaires qui sont :

- $\mathcal{T}$: une pièce de qualité est requise ;
- $\mathcal{C}$: les objets A et B sont connectés ;
- $\mathcal{CW}$: les objets A et B sont bien connectés ;
- $\mathcal{AP}$, $\mathcal{BP}$: l'objet A, B est peint ;
- $\mathcal{APW}$, $\mathcal{BPW}$: l'objet A, B est bien peint ;
- $\mathcal{ASH}$, $\mathcal{BSH}$: l'objet A, B a la bonne forme ;
- $\mathcal{ASM}$, $\mathcal{BSM}$: l'objet A, B est poli ;
- $\mathcal{ADR}$, $\mathcal{BDR}$: l'objet A, B est percé ;
- $\mathcal{BO}$: le robot a une clé ;
- $\mathcal{GL}$: le robot a de la colle ;
- $\mathcal{SG}$: le robot a un pistolet à peinture ;
- $\mathcal{SL}$: un ouvrier qualifié est présent.

Le problème est donc composé de $2^{17} * 14 = 1\,835\,008$ couples état/action. La récompense obtenue par le robot dépend de la demande. Par exemple, si une pièce finale de bonne qualité est demandée, alors le robot a une récompense plus importante si l'objet est, par exemple, peint à la main plutôt que vaporisé, avant d'être boulonné.

Réorganisation de la fonction de valeur

Afin de déterminer si l'utilisation de la réorganisation de la fonction de valeur est profitable, nous avons testé son utilisation avec l'algorithme SVI sur les problèmes *Coffee Robot* et *Factory*. Les résultats sont montrés figure 4.24.

	Non ordonnée	Ordonnée
Coffee Robot	68ms	125ms
Factory	2004s	1090s

(a)

	Non ordonnée	Ordonnée
Coffee Robot	35 nœuds	35 nœuds
Factory	6233 nœuds	2835 nœuds

(b)

FIG. 4.24 – Temps de calcul de l'algorithme SVI (figure a) et taille de la fonction de valeur optimale (figure b) sur les problèmes *Coffee Robot* et *Factory*. Lorsqu'aucune réorganisation n'est possible, l'algorithme de réorganisation nécessite un plus grand temps de calcul. Lorsqu'une réorganisation importante peut être effectuée, alors le temps de calcul diminue, de même que la taille de la fonction de valeur optimale.

On peut constater que, pour le problème *Coffee Robot*, aucune réorganisation n'est possible puisque l'arbre de décision représentant la fonction de valeur optimale possède le même nombre de nœuds avec ou sans réorganisation. Pour ce problème, l'algorithme SVI avec réorganisation nécessite plus de temps de calcul. Au contraire, pour le problème *Factory*, une réorganisation importante de la fonction de valeur est possible puisque SVI avec réorganisation construit un arbre de 2835 nœuds contre 6233 nœuds pour SVI sans réorganisation. Cette amélioration a un effet sur la place mémoire requise pour stocker la structure de donnée, mais aussi sur le temps de calcul puisque SVI avec réorganisation nécessite deux fois moins de temps pour converger que l'algorithme SVI sans réorganisation. Ainsi, pour les résultats présentés ci-dessous, nous utiliserons SVI avec réorganisation.

Résultats de l'apprentissage

La figure 4.25 représente le temps de calcul utilisé par les algorithmes d'apprentissage pour la construction (figure 4.25(a)) et la taille de la représentation de la fonction de transition (figure 4.25(b)) du FMDP appris. Pour les deux algorithmes BuildFMDP et BuildFMDPnAT, nous pouvons observer que leur temps de calcul respective est, d'une part, similaire et, d'autre part augmente linéairement avec la taille de l'échantillon à apprendre. Concernant la taille du modèle, on

remarque une augmentation rapide du nombre de nœuds (pour les échantillons de moins de 6000 observations) puis une stabilisation pour des échantillons de plus grande taille. De plus, on peut remarquer que la taille du modèle construit par BuildFMDPnAT contient un peu plus de 600 nœuds alors que le modèle construit par BuildFMDP en contient quasiment 800.

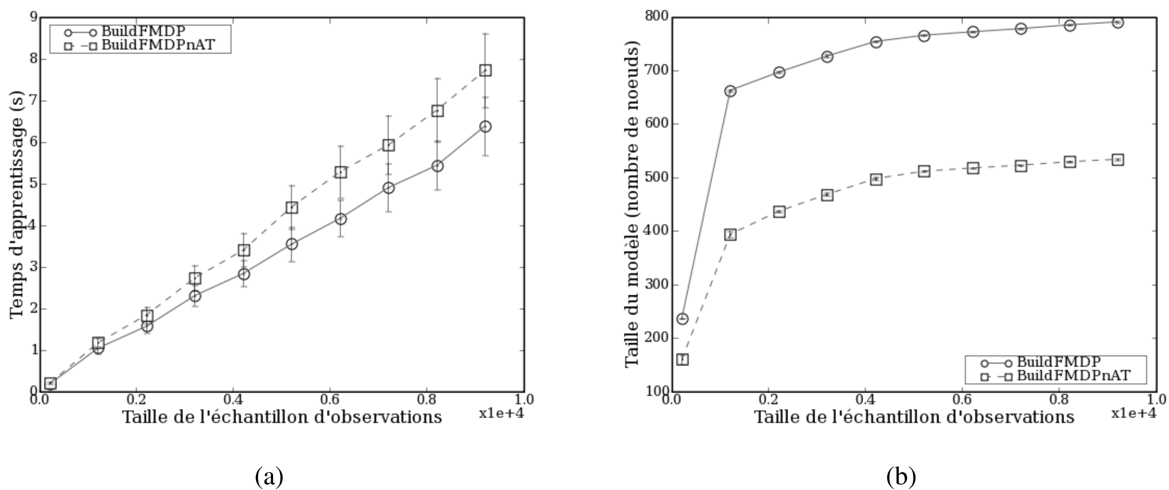
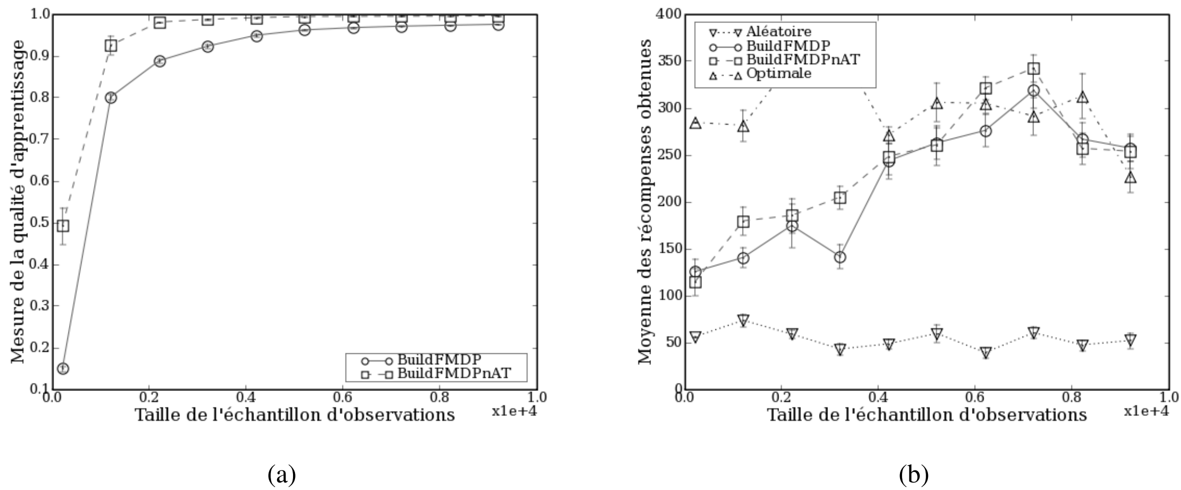


FIG. 4.25 – Temps de calcul pour les algorithmes BuildFMDP et BuildFMDPnAT (figure a) et taille de la représentation de la fonction de transition (figure b) apprise sur le problème *Factory*. Alors que le temps de calcul augmente linéairement avec la taille de l'échantillon, l'augmentation de la taille du modèle décroît avec la taille de cet échantillon.

La figure 4.26 représente l'exactitude du modèle (figure 4.26(a)) construit par les algorithmes d'apprentissage ainsi que la récompense actualisée de la politique optimale calculée à partir du FMDP appris (figure 4.26(b)). On peut notamment observer que, malgré une taille de modèle plus petite, l'exactitude du modèle du FMDP construit par BuildFMDPnAT est meilleure que celle du modèle construit par BuildFMDP. Cette différence se retrouve de façon moins nette avec la récompense actualisée pour laquelle l'erreur standard est très importante. On peut cependant observer que les deux politiques optimales générées à partir des FMDPs appris sont meilleures qu'une politique aléatoire, sans pour autant atteindre la performance de la politique optimale du problème. De plus, on peut remarquer que l'amélioration de la récompense actualisée requiert un échantillon de plus grande taille comparée à l'amélioration de l'exactitude du modèle.



(a) (b)

FIG. 4.26 – Incidence de la taille de l'échantillon sur l'exactitude du modèle (figure a) et de la récompense actualisée sur le problème *Factory* et l'algorithme SVI (figure b). Alors que l'exactitude du modèle augmente rapidement, la récompense actualisée nécessite des échantillons de plus grande taille.

Les résultats concernant le problème *Factory* montrent une propriété essentielle dans l'apprentissage de grands problèmes d'apprentissage par renforcement : la généralisation. En effet, bien que la taille de l'échantillon d'observation, $1,0 \cdot 10^4$ observations, par rapport aux nombres de couples état/action existant dans le problème, $1,8 \cdot 10^6$ couples, soit petite, les récompenses obtenues par la politique optimale calculée à partir du FMDP appris sont proches de celles obtenues par la politique optimale du problème. Ainsi, la politique calculée à partir du FMDP appris propose des actions pertinentes pour des états n'appartenant pas à l'échantillon d'observations.

Cette généralisation s'obtient à partir de deux propriétés complémentaires exhibées par les algorithmes que nous utilisons. D'une part, les algorithmes d'induction d'arbres de décision construisent les fonctions du FMDP en généralisant à partir des exemples appartenant à l'échantillon d'observations. D'autre part, les algorithmes de planification dans les FMDPs agrègent les états similaires ensembles, travaillant ainsi sur des partitions de l'espace d'états, même si il existe dans ces partitions des états qui n'apparaissent pas dans l'échantillon d'observations.

Nous pouvons aussi remarquer que la représentation de la fonction de transition construite par l'algorithme BuildFMDPnAT (constituant seulement un arbre $\mathsf{Tree}\,[P(X'_i|s,a)]$ par variable) est plus compacte que la représentation utilisée par l'algorithme BuildFMDP (constituant un arbre $\mathsf{Tree}\,[P_a(X'_i|s)]$ par action et par variable). Ceci n'est pas au détriment de la représentation du problème puisque l'exactitude de la fonction de transition construite par ces algorithmes et les performances obtenues par les politiques correspondantes sont similaires. Par conséquent, la représentation utilisée par BuildFMDPnAT exploite le fait que certaines transitions du problème ne dépendent pas de l'action exécutée par l'agent.

Le problème *Factory4*

Le deuxième problème que nous traitons se nomme *Factory4* et est utilisé avec l'algorithme de planification SPUDD, permettant de traiter un problème de plus grande taille que SVI. Ce problème reprend le principe de *Factory* mais avec des variables supplémentaires (au total 28 variables) et une action supplémentaire (au total 15 actions), soit $2^{28} * 15 = 4\,026\,531\,840$ couples état/action. Les variables supplémentaires sont :

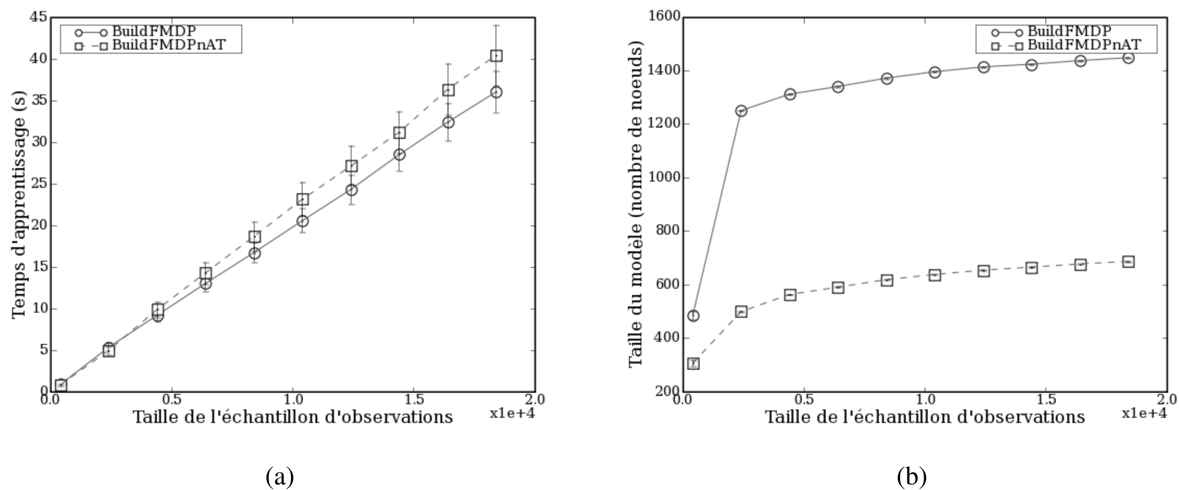
- $\mathcal{CL}$: le robot a un serre-joint ;
- $\mathcal{DR}$: le robot a une foreuse ;
- $\mathcal{MO}$: le robot a un étau ;
- $\mathcal{BR}$: le robot a des brosses ;
- $\mathcal{LA}$: de la laque est disponible ;
- $\mathcal{AW}$: le robot a une machine à souder à l'arc électrique ;
- $\mathcal{SW}$: le robot a une machine à souder par résistance ;
- $\mathcal{SWL}$: un ouvrier qualifié pour une soudure à l'arc électrique est présent ;
- $\mathcal{BIT}$: le robot a une mèche pour la foreuse ;
- $\mathcal{AWE}$: le robot a le matériel nécessaire pour une soudure à l'arc électrique ;
- $\mathcal{SWE}$: le robot a le matériel nécessaire pour une soudure par résistance.

La seule action ajoutée est l'action $\mathcal{W}$elD consistant à souder deux pièces ensembles. Nous avons utilisé la version de SPUDD en C++ disponible sur Internet[1].

La figure 4.27 représente le temps de calcul utilisé par les algorithmes d'apprentissage pour la construction (figure 4.27(a)) et la taille de la représentation de la fonction de transition (figure 4.27(b)) du FMDP appris. Les résultats sont très semblables entre ce problème et le précédent : d'une part, le temps d'apprentissage des algorithmes augmente linéairement avec la taille de l'échantillon, d'autre part la taille du modèle construit se stabilise rapidement. Enfin, on remarque aussi la différence entre les deux algorithmes BuildFMDPnAT et BuildFMDP. Le premier, en utilisant un peu moins de temps de calcul (entre 0 et 5 secondes), construit un modèle nécessitant moins de nœuds que le deuxième (moins de 800 nœuds contre plus de 1400 nœuds).

La figure 4.28 représente l'exactitude du modèle (figure 4.28(a)) construit par les algorithmes d'apprentissage ainsi que la récompense actualisée de la politique optimale (figure 4.28(b)) calculée à partir du FMDP appris. Ainsi, on observe que, malgré une représentation plus compacte et un temps de calcul pour l'apprentissage plus court, le modèle appris par BuildFMDPnAT est considéré comme plus exact que celui de BuildFMDP. Cependant, contrairement aux résultats précédents, cette différence ne se retrouve pas dans la récompense actualisée où l'on ne distingue pas de différence nette entre leurs politiques associées. Enfin, notons que dans les deux cas, pour l'échantillon de 20 000 observations, les performances des deux politiques sont proches de la politique optimale

[1] http://www.cs.toronto.edu/~jhoey/spudd



(a) (b)

FIG. 4.27 – Temps de calcul pour les algorithmes BuildFMDP et BuildFMDPnAT (figure a) et taille de la représentation de la fonction de transition (figure b) apprise sur le problème *Factory4*. Alors que le temps de calcul augmente linéairement avec la taille de l'échantillon, l'augmentation de la taille du modèle décroît avec la taille de cet échantillon.

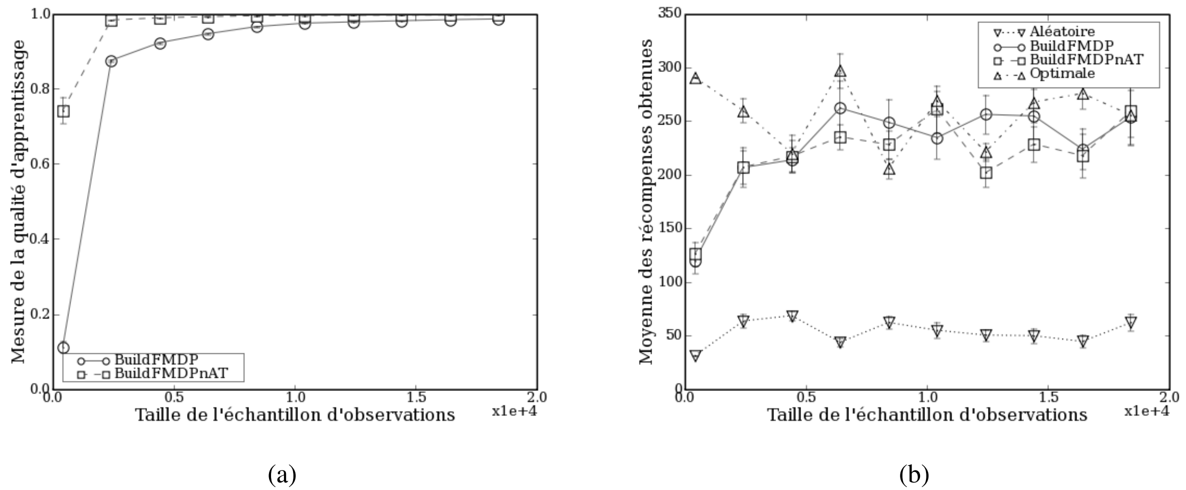
du problème.

Bien que le problème *Factory4* soit plus grand que le problème *Factory* ($4,0 \cdot 10^9$ couples état/action contre $1,8 \cdot 10^6$), les résultats obtenus sont similaires, soulignant ainsi que la complexité de l'apprentissage dépend plus de la structure du problème (les deux problèmes sont similaires) que de leur taille. De plus, nous pouvons remarquer que, bien que les algorithmes d'apprentissage construisent un FMDP en utilisant des arbres de décisions, cette représentation reste adaptée à l'algorithme SPUDD qui utilise des ADDs pour représenter les fonctions du problème.

Le problème *Ring*

Enfin, nous terminons cette analyse par un problème utilisé par Guestrin et al. (2003b) pour illustrer sa méthode (basée sur la programmation linéaire) avec des problèmes de planification possédant notamment une forte décomposition additive de la fonction de récompense. Le problème *Ring* représente un ensemble de n machines connectées les unes aux autres pour former un anneau unidirectionnel. Toutes les transitions du problèmes étant stochastiques, ce problème ne nécessite pas la définition d'état initiaux ou terminaux.

Chaque machine X_i peut être opérationnelle (noté $X_i = 1$) ou non (noté $X_i = 0$). Lorsqu'une machine n'est pas opérationnelle, la probabilité que les machines qui lui sont connectées ne soient pas opérationnelles au prochaine pas de temps augmente nettement. Un administrateur système



(a) (b)

FIG. 4.28 – Incidence de la taille de l'échantillon sur l'exactitude du modèle (figure a) et la récompense actualisée (figure b) sur le problème *Factory4* et l'algorithme SPUDD. L'exactitude du modèle et la récompense actualisée augmente rapidement pour les deux algorithmes d'apprentissage BuildFMDP et BuildFMDPnAT.

est chargé de surveiller ce réseau de machines. Pour cela, il peut redémarrer chaque machine ou bien ne rien faire. Ainsi, la distribution de probabilités conditionnelle $P(X'_i)$ d'une machine X_i lorsque l'action est autre que de redémarrer cette machine est définie figure 4.29. Si l'administrateur système choisit de redémarrer une machine X_i, alors $P(X'_i = 1) = 1$. Enfin, l'administrateur système peut choisir de ne rien faire. Le problème est donc composé de n variables binaires et de $n + 1$ actions.

X_{i-1}	X_i	X'_i
0	0	0.05
0	1	0.5
1	0	0.09
1	1	0.9

FIG. 4.29 – Distribution de probabilités conditionnelle $P(X'_i)$ d'une machine X_i lorsque l'action sélectionnée est autre que de redémarrer cette machine

La fonction de récompense de ce problème est fortement additive puisqu'elle est décomposée en une somme de n fonctions de récompense localisée. Chaque fonction de récompense R_i est associée à une machine X_i et est égale à 1 lorsque la machine est opérationnelle, sinon 0. Pour des raisons pratiques, une récompense de 2 au lieu de 1 est associée à l'une des machines afin de définir une préférence sur l'une d'entre elles (et donc éviter que la fonction de valeur puisse être approchée par une somme de fonctions de base ayant chacune un coefficient identique).

$$\begin{vmatrix} X_i = 0 \wedge X_{i-1} = 0 : \ 0.0495 \\ X_i = 0 \wedge X_{i-1} = 1 : \ 0.0891 \\ X_i = 1 \wedge X_{i-1} = 0 : \ 0.495 \\ X_i = 1 \wedge X_{i-1} = 1 : \ 0.891 \end{vmatrix}$$

FIG. 4.30 – Définition d'une fonction de base h_i associée à une variable X_i.

De plus, nous supposons que la décomposition additive de la fonction valeur est connue. Par conséquent, nous utilisons une fonction de base h_i par variable X_i, celle-ci est définie figure 4.30. La politique notée *Optimale* est la politique gloutonne associée à l'approximation de la fonction de valeur calculée avec l'algorithme FactoredLPA en utilisant l'ensemble de fonctions de base h_i (voir figure 4.30) et les fonctions exactes de récompense et de transition définissant le problème.

Concernant la taille du problème, nous avons testé nos algorithmes avec $n = 40$ (soit $2^{40} * 41 = 45\ 079\ 976\ 738\ 816$ couples état/action). Le programme linéaire généré est résolu avec l'algorithme du simplex utilisé depuis l'application `glpsol` inclue dans le paquetage *GNU Linear Programming Kit*[2]. Enfin, pour des raisons d'implémentation, nous avons substitué l'algorithme BackProjRule(ρ, a) (figure 3.27, page 73) par le calcul équivalent des étapes 1 et 2 de l'algorithme Regress$(\mathsf{Tree}\,[V]\,, a)$ de SPI (figure 3.9, page 51) qui utilise des arbres de décision plutôt que des règles.

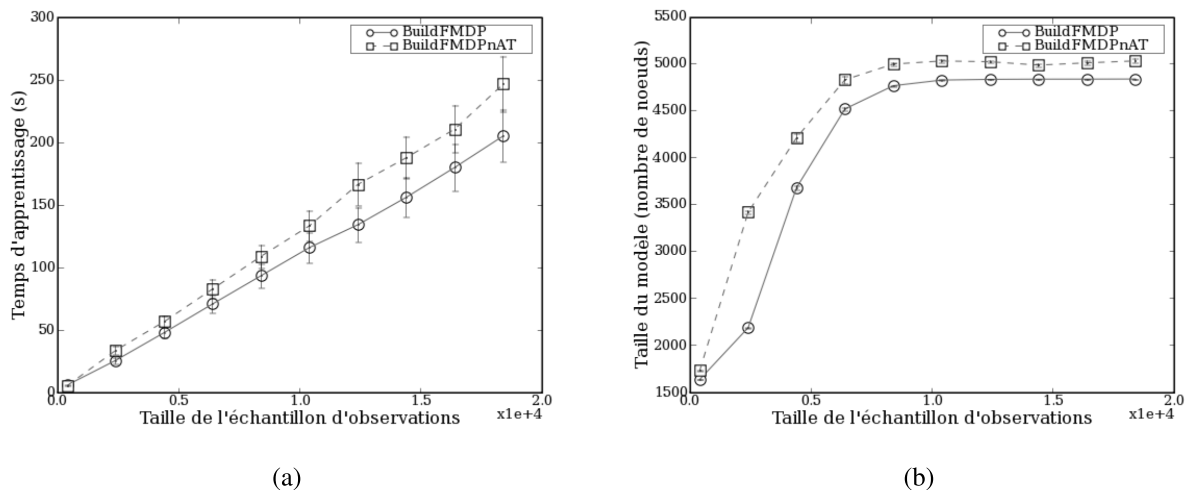
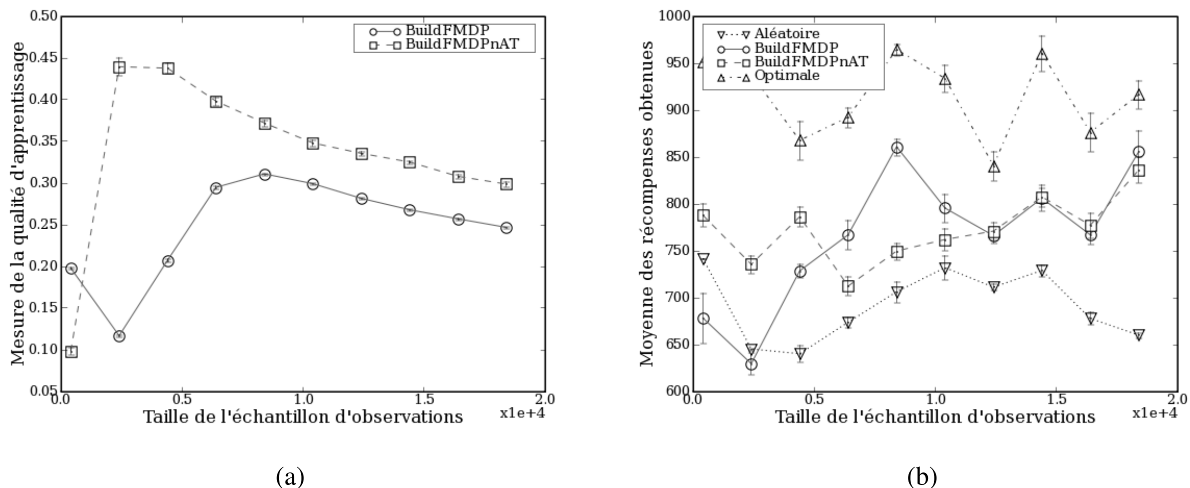


FIG. 4.31 – Temps de calcul (figure a) pour les algorithmes BuildFMDP et BuildFMDPnAT et taille de la représentation de la fonction de transition apprise (figure b) sur le problème *Ring*.

La figure 4.31 représente le temps de calcul utilisé par les algorithmes d'apprentissage (figure 4.31(a)) et la taille de la représentation de la fonction de transition (figure 4.31(b)) du FMDP

[2]http ://www.gnu.org/software/glpk/glpk.html

appris. On peut observer que l'apprentissage effectué par BuildFMDPnAT construit un modèle de la fonction de transition de taille similaire à celle construite par l'algorithme BuildFMDP (environ 5000 nœuds) au prix d'un temps de calcul plus élevé (environ 50 secondes). De plus, on note pour les deux algorithmes que le temps d'apprentissage augmente de façon linéaire avec la taille de l'échantillon à apprendre.



(a) (b)

FIG. 4.32 – Incidence de la taille de l'échantillon sur l'exactitude du modèle (figure a) et la récompense actualisée (figure b) sur le problème *Ring* et l'algorithme FactoredLPA.

La figure 4.32 représente l'exactitude du modèle (figure 4.32(a)) construit par les algorithmes d'apprentissage ainsi que la récompense actualisée de la politique optimale (figure 4.32(b)) calculée à partir du FMDP appris. On peut observer que, comparé aux deux problèmes précédents, le modèle appris est considéré comme de moins bonne qualité (une moyenne de probabilité de moins de 0.4 contre plus de 0.8 pour les problèmes précédents). De plus, on observe une différence importante entre les deux algorithmes BuildFMDPnAT et BuildFMDP. Cette différence se retrouve partiellement avec la mesure de la récompense actualisée où l'algorithme BuildFMDPnAT progresse beaucoup plus vite que BuildFMDP pour atteindre un seuil sur lequel il est rejoint ensuite. Enfin, on peut observer que l'exactitude de la fonction de transition pour les deux algorithmes d'apprentissage décroît à partir d'un échantillon d'observations dont la taille est supérieur à 10 000.

Les résultats de ce problème se distingue des deux problèmes précédents sur plusieurs points. En premier lieu, les deux algorithmes BuildFMDPnAT et BuildFMDP construisent des représentations de la fonction de transition de taille similaire, bien qu'utilisant une représentation différente. Ceci vient du fait que, pour ce problème, peu de transition ne dépendent pas de l'action choisie par

l'agent. En effet, les variables du problème sont constituées par l'état des différentes machines du réseau, chacune pouvant être affecté par une action.

Cependant, on peut observer une différence significative entre les deux représentations lorsque l'échantillon d'observations est de petite taille. Cette différence peut être expliquée par le fait que l'algorithme BuildFMDPnAT, contrairement à BuildFMDP, ne partitionne pas les ensembles d'exemples pour chaque action existant dans le problème, permettant ainsi d'obtenir une évaluation des distributions de probabilité plus exactes lorsque le nombre d'exemples disponible pour l'apprentissage est faible.

De plus, nous pouvons aussi observer que l'exactitude de l'apprentissage décroît au fur et à mesure que le nombre d'observations augmente à partir d'une certaine taille de l'échantillon d'observations. Ceci peut être expliqué par le fait que le nombre d'exemples est assez important pour que le pré-élagage autorise l'installation d'un nouveau nœud de décision, mais que ce nombre d'exemples ne soit pas suffisant, soit pour choisir un test adapté à la séparation des distributions de probabilités, soit pour évaluer de façon correcte les probabilités aux feuilles sous ce nœud de décision (par exemple, si un nœud de décision testant l'action réalisée par l'agent est installé, il contiendra 41 feuilles sur lesquelles seront répartis les exemples présents au nœud de décision).

Enfin, nous pouvons noter que l'apprentissage est plus lent que dans les problèmes précédents. Cependant, pour une taille d'échantillon extrêmement petite, $2,0 \cdot 10^4$ observations, comparé à la taille du problème, $4,5 \cdot 10^{13}$ couples état/action, l'apprentissage permet d'obtenir des politiques dont les performances sont nettement meilleures qu'une politique purement aléatoire.

4.4 Synthèse

Dans ce chapitre, nous avons présenté deux nouveaux algorithmes d'apprentissage hors-ligne d'un FMDP, nommément BuildFMDPnAT et BuildFMDP. Ces deux algorithmes utilisent principalement l'induction d'arbres de décision pour construire des représentations structurées des fonctions de transition et de récompense. De plus, nous avons explicité comment les représentations construites par ces algorithmes pouvaient être utilisées avec des algorithmes de planification dans les FMDPs. Afin de qualifier l'apprentissage, nous avons introduit deux mesures, l'erreur relative d'une politique et l'exactitude de la fonction de transition, chacune permettant d'exploiter la structure du problème pour être évaluée.

Les résultats montrent que les algorithmes d'induction d'arbres de décisions sont particulièrement bien adaptés pour l'apprentissage des FMDPs. En effet, premièrement, il est facile de déduire à partir des arbres de décision construit par l'apprentissage aussi bien les indépendances relatives aux fonctions du problème que les indépendances relatives aux contextes. La structure du problème peut donc ainsi être facilement exploitée par les algorithmes de planification.

Deuxièmement, la propriété de généralisation des algorithmes d'induction d'arbres de décision

est exploitée par les algorithmes de planification qui, en agrégeant des états similaires, calculent des politiques pertinentes dans des états n'appartenant pas à l'échantillon d'apprentissage. Par conséquent, la complexité de l'apprentissage ne dépend principalement que de la structure du problème, et moins de sa taille : pour certains problèmes, un échantillon d'observations de petite taille est suffisant pour que l'apprentissage puisse améliorer nettement la performance d'un agent, bien que le problème soit de grande taille.

Nous avons introduit deux algorithmes d'apprentissage distincts, chacun utilisant une représentation différente de la fonction de transition. Pour représenter la distribution de probabilités conditionnelle d'une variable pour une action donnée, le premier construit un arbre de décision par variable et par action, alors que le deuxième ne construit qu'un arbre par variable et inclus l'action dans les tests pouvant être installés dans l'arbre. Bien que cette deuxième représentation nécessite un coût de calcul supplémentaire, nous pensons qu'elle est plus adaptée pour les grands problèmes pour deux raisons.

La première est qu'elle permet d'exploiter le fait que certaines transitions dans le problème ne dépendent pas de l'action choisie par l'agent et construit donc une représentation de la fonction de transition pouvant être beaucoup plus compacte que la première méthode. La deuxième raison est qu'elle nécessite moins d'exemples pour estimer les probabilités ne dépendant pas de l'action, permettant ainsi d'avoir une meilleure estimation de certaine transition à partir d'un échantillon d'observations plus petit.

L'apprentissage d'un FMDP est réalisé en utilisant un algorithme d'induction d'arbres de décision effectuant un pré-élagage pour l'apprentissage de fonctions stochastiques, telles que les distributions de probabilités conditionnelles de la fonction de transition. Ce pré-élagage est paramétré par la valeur seuil τ_{χ^2} constituant l'unique paramètre de l'apprentissage. Nous avons illustré de façon expérimentale que ce seuil constitue un paramètre réglant la certitude avec laquelle deux distributions sont séparées, et non une approximation des distributions de probabilités conditionnelles. Ce résultat confirme le résultat théorique concernant la limite de la valeur χ^2 lorsque le nombre d'exemples augmente. Par conséquent, pour une taille d'échantillon infinie, l'apprentissage construira le même FMDP, quelle que soit la valeur du seuil.

Enfin, nous avons souligné une difficulté concernant n'importe quel algorithme d'apprentissage hors-ligne : la pertinence des observations appartenant à l'échantillon. Nous avons notamment montré que pour apprendre un problème d'apprentissage par renforcement, l'absence de certaines observations pouvait être dramatique pour la qualité de la politique optimale calculée à partir du FMDP appris, alors que l'apprentissage de la fonction de transition peut être de bonne qualité.

Les travaux présentés dans ce manuscrit sont, à notre connaissance, les premiers travaux concernant l'apprentissage de la structure des FMDPs et l'intégration d'un tel apprentissage avec les algorithmes de planification existant dans la littérature. L'apprentissage de FMDPs concerne l'apprentissage de DBNs dans le cas particulier où seulement deux pas de temps sont considérés. L'appren-

tissage de la structure de DBNs dans un cadre plus général a déjà fait l'objet de quelques études (Chickering et al., 1997; Friedman and Goldszmidt, 1998). À partir d'un échantillon d'observations, ces algorithmes construisent de façon explicite la structure générale des DBNs, puis de façon locale, les structures quantifiant le réseau. Cependant, nous rappelons que notre objectif est un apprentissage incrémental du FMDP : contrairement aux algorithmes d'induction d'arbres de décision, ces méthodes s'adaptent mal à un tel apprentissage.

Chapitre 5

Apprentissage incrémental : l'approche SDYNA

Le chapitre précédent concerne l'apprentissage hors-ligne d'un FMDP. On suppose qu'un échantillon d'observations a été constitué. À partir de cet échantillon, on construit un FMDP qui sera utilisé par un algorithme de planification pour générer une politique. Une fois cette politique calculée, elle est constante et ne permet donc pas à l'agent de s'adapter lors de l'exécution de celle-ci dans son environnement, par exemple lorsque l'agent se trouve dans des états qui ne sont pas similaires à ceux qui étaient présents dans l'échantillon d'observation.

Pour qu'un agent soit capable d'adapter sa politique, il est nécessaire de prendre en compte au fur et à mesure les observations de celui-ci dans son environnement. Par conséquent, plutôt que de calculer la politique d'un agent à partir d'un échantillon d'observations, l'apprentissage et la planification doivent mettre à jour celle-ci à partir d'un flux d'observations.

Une solution simple pour effectuer cette mise à jour consiste à construire le FMDP et la politique optimale associée à chaque pas de temps. Cependant, une telle méthode est extrêmement coûteuse : par exemple, les résultats du chapitre précédent, section 4.3.3, montrent que le temps d'apprentissage augmente linéairement avec la taille de l'échantillon. De plus, elle ne permet pas de bénéficier des calculs réalisés précédemment afin que le temps de calcul nécessaire à la mise à jour ne dépende principalement que des changements induits par la (ou les) nouvelle(s) observation(s), contrairement à une approche incrémentale.

Nous proposons dans ce chapitre une approche générale pour résoudre un problème d'apprentissage par renforcement de façon incrémentale : l'approche SDYNA. SDYNA reprend l'idée principale que nous avons présentée dans le chapitre précédent, c'est-à-dire utiliser l'apprentissage supervisé pour construire un FMDP, puis utiliser la planification pour construire une politique à partir de ce FMDP. Cependant, nous nous concentrons sur l'utilisation d'algorithmes incrémentaux dans le cadre d'un apprentissage à partir d'un flux d'observations afin de pouvoir résoudre un problème

d'apprentissage par renforcement en ligne.

Dans un premier temps, nous présentons dans la section 5.1 l'approche générale SDYNA. SDYNA est principalement composée d'une phase d'apprentissage incrémental, présentée section 5.2 et, d'une phase de planification incrémentale, présentée section 5.3. Nous présentons ensuite section 5.4 les résultats de SDYNA dans certains des problèmes présentés dans le chapitre précédent.

5.1 L'approche SDYNA

A priori, la problématique de planification, où les fonctions de récompense et de transition du problème sont connues, semble être opposée avec la problématique d'apprentissage par renforcement, où les fonctions de récompense et de transition du problème ne sont pas connues. Cependant, nous avons vu dans le chapitre consacré aux MDPs (section 2.3.2, page 23) que Sutton and Barto (1998) a proposé un cadre unifié, DYNA, où ces deux problématiques sont complémentaires et peuvent être utilisées ensemble afin d'améliorer les performances d'un algorithme dans le cadre de l'apprentissage par renforcement.

L'approche DYNA, plus particulièrement les instanciations DYNA-PI et DYNA-Q (Sutton, 1990), a été proposée dans le cadre des MDPs. Par conséquent, ces algorithmes ne sont pas utilisables directement avec des problèmes d'apprentissage par renforcement de grande taille, notamment parce qu'ils nécessitent une énumération exhaustive de l'espace d'états.

Ainsi, de façon similaire à l'approche DYNA (cf. section 2.3.2, page 34), nous proposons l'approche *Structured* DYNA[1] (SDYNA). À l'instar de DYNA, SDYNA intègre la prise de décision, l'apprentissage et la planification afin de résoudre un problème d'apprentissage par renforcement en ligne et pour lequel la structure des fonctions de transition et de récompense sont inconnues. Afin de pouvoir traiter des problèmes de grande taille, contrairement à DYNA, SDYNA utilise des représentations factorisées dans le cadre du formalisme des FMDPs.

SDYNA est présenté figure 5.1, avec $\mathsf{Fact}\left[F\right]$ la représentation factorisée de la fonction F, Acting l'algorithme de prise de décision à partir d'un ensemble de fonctions de valeur d'action $\{\mathsf{Fact}\left[Q^t_a\right], \forall a \in A\}$, UpdateModel l'algorithme d'apprentissage supervisé incrémental du FMDP $\hat{\mathcal{F}}_t$ et IncPlan l'algorithme de planification. SDYNA est donc composé de trois phases différentes.

La première concerne la prise de décision : l'agent est dans un état s et l'algorithme Acting choisit une action a à réaliser dans l'environnement à partir des connaissances et de la politique actuelle de l'agent, représentés par l'ensemble des fonctions de valeur d'action $\{\mathsf{Fact}\left[Q^t_a\right], \forall a \in A\}$. Après avoir exécuté l'action a dans l'état s, l'agent est dans un nouvel état s' et a obtenu la récompense r, constituant ainsi une nouvelle observation $\langle s, a, s', r\rangle$. Lors de la deuxième phase, cette observation est intégrée au FMDP $\hat{\mathcal{F}}_{t-1}$ par l'algorithme d'apprentissage supervisé incrémental UpdateModel

[1]Nous utilisons la même convention que pour les noms SPI (*Structured Policy Iteration*) et SVI (*Structured Value Iteration*) qui nomment respectivement la version factorisée des algorithmes *Value Iteration* et *Policy Iteration*.

Paramètre(s) : $\text{Fact}\,[F]$ la représentation factorisée de la fonction F, les algorithmes Acting, UpdateModel et IncPlan

Initialisation : Initialiser le FMDP $\hat{\mathcal{F}}_0 = \{\forall X_i \in X : \text{Fact}\,[P(X'_i|s,a)]$ et $\forall i \in [1,n] : \text{Fact}\,[R_i]\}$, l'ensemble des fonctions de valeur d'action $\{\text{Fact}\,[Q_a^0]\,, \forall a \in A\}$ et la fonction de valeur $\text{Fact}\,[V_0]$.

À chaque pas de temps : pour un état s :

Décision :

1. $a \leftarrow \underline{\text{Acting}}(s, \{\text{Fact}\,[Q_a^{t-1}]\,, \forall a \in A\})$
2. Exécuter a, observer s' et r

Apprentissage :

3. $\hat{\mathcal{F}}_t \leftarrow \underline{\text{UpdateModel}}(\hat{\mathcal{F}}_{t-1}, \langle s, a, s', r\rangle)$

Planification :

4. $\{\text{Fact}\,[V_t]\,, \{\text{Fact}\,[Q_a^t]\,, \forall a \in A\}\} \leftarrow \underline{\text{IncPlan}}(\hat{\mathcal{F}}_t, \text{Fact}\,[V_{t-1}])$

FIG. 5.1 – L'algorithme SDYNA.

qui construit une version mise à jour $\hat{\mathcal{F}}_t$ du FMDP. Ensuite, pour la troisième phase, le FMDP $\hat{\mathcal{F}}_t$ est utilisé pour calculer un nouvel ensemble de fonctions de valeur d'action $\{\text{Fact}\,[Q_a^t]\,, \forall a \in A\}$ et une nouvelle fonction de valeur $\text{Fact}\,[V_t]$ à partir de la fonction de valeur calculée au pas de temps précédent $\text{Fact}\,[V_{t-1}]$ avec l'algorithme IncPlan.

Nous pouvons remarquer que SDYNA ne nécessite pas de représentation explicite de la politique, à l'instar de DYNA-Q. Ainsi, la phase de décision, réalisée par l'algorithme Acting, peut exécuter une politique gloutonne, à partir de l'ensemble des fonctions de valeur d'action $\{\text{Fact}\,[Q_a^t]\,, \forall a \in A\}$, en sélectionnant, pour un état s donné, l'action pour laquelle la valeur d'action $\text{Fact}\,[Q_a^t(s)]$ est la plus élevée. Afin de gérer le compromis exploration/exploitation, d'autres algorithmes peuvent être utilisés, notamment ϵ-greedy ou *softmax* (Sutton and Barto, 1998). Les deux sections suivantes présentent en détail les algorithmes UpdateModel (section 5.2) et IncPlan (section 5.3).

5.2 Intégration de l'apprentissage dans SDYNA

Nous avons présenté lors du chapitre 4 une méthode de décomposition d'une observation de l'agent dans son environnement $\langle s, a, s', r\rangle$ en plusieurs exemples afin de réutiliser des techniques d'apprentissage supervisé pour construire un FMDP. Bien que SDYNA requière l'utilisation d'algorithmes incrémentaux, cette méthode reste valide. Cependant, les algorithmes exposés lors du chapitre précédent, plus particulièrement les algorithmes d'induction d'arbres de décision, ne peuvent pas être utilisés directement puisqu'ils ne sont pas incrémentaux. Nous commençons donc par dé-

crire section 5.2.1 les algorithmes existant dans la littérature et concernant l'induction incrémentale d'arbres de décision. La section 5.2.2 décrit les algorithmes d'apprentissage d'un FMDP adapté du chapitre précédent pour apprendre de façon incrémentale à partir d'un flux d'exemples.

5.2.1 Induction incrémentale d'arbres de décision

Cette section présente, de la même façon que dans la section 4.1.1, les algorithmes incrémentaux de construction d'arbres pour la classification (Schlimmer and Fisher, 1986; Utgoff, 1986, 1988; Utgoff et al., 1997) et pour la régression. Ainsi, plutôt que de construire un arbre $\mathsf{Tree}[F]$ représentant la fonction F à apprendre à partir d'un ensemble d'exemples $\mathcal{E} = \{\langle \mathfrak{a}_i, \varsigma_i \rangle\}$, l'objectif est de mettre à jour la représentation $\mathsf{Tree}[F]$ à partir d'un flux d'exemples $\langle \mathfrak{a}_t, \varsigma_t \rangle$.

Paramètre(s) : une mesure d'information $\mathcal{M}$

Initialisation : Initialiser $\mathsf{Tree}[F]$
Soit k le nœud courant, $\mathcal{E}_k = \{\langle \mathfrak{a}_k, \varsigma_k \rangle\}$ les exemples présents au nœud k
A chaque exemple $\langle \mathfrak{a}_t, \varsigma_t \rangle$ **:**

1. Mettre à jour la mesure d'information $\mathcal{M}$ en ajoutant $\langle \mathfrak{a}_t, \varsigma_t \rangle$ à $\mathcal{E}_k$
2. **Si** tous les exemples $\{\langle \mathfrak{a}_j, \varsigma_j \rangle\} \in \mathcal{E}_k$ pointent sur la même valeur ς ($\forall \langle \mathfrak{a}_j, \varsigma_j \rangle \in \mathcal{E}_k : \varsigma_j = \varsigma$) :
   Alors : transformer k en une feuille contenant ς
   Sinon :
   (a) Soit $\mathcal{V} \leftarrow$ SelectAttr($\mathcal{M}, \mathcal{E}$)
   (b) **Si** k ne teste pas $\mathcal{V}$ (k est une feuille ou un nœud de décision testant un autre attribut que $\mathcal{V}$) :
      Alors :
      i. Transformer k en un nœud de décision testant $\mathcal{V}$
      ii. Supprimer les sous-arbres de k (s'ils existent)
      iii. $\forall \langle \mathfrak{a}_j, \varsigma_j \rangle \in \mathcal{E}_k$: UpdateTree($\langle \mathfrak{a}_j, \varsigma_j \rangle, k_{\mathfrak{a}_j[\mathcal{V}]}$) avec $k_{\mathfrak{a}_j[\mathcal{V}]}$ le nœud enfant de k correspondant à la branche $\mathfrak{a}_j[\mathcal{V}]$ (ajouter la branche si nécessaire)
      Sinon : UpdateTree($\langle \mathfrak{a}_t, \varsigma_t \rangle, k_{\mathfrak{a}_t[\mathcal{V}]}$) avec $k_{\mathfrak{a}_t[\mathcal{V}]}$ le nœud enfant de k correspondant à la branche $\mathfrak{a}_t[\mathcal{V}]$ (ajouter la branche si nécessaire)

FIG. 5.2 – L'algorithme UpdateTree récursif utilisé pour mettre à jour un arbre à partir d'un flux d'exemples.

La version incrémentale de l'algorithme BuildTree (figure 4.1, page 83), notée UpdateTree, est décrite figure 5.2 (Utgoff, 1986). L'algorithme commence par tester si l'ensemble des exemples $\mathcal{E}_k$ présents au nœud k est pure, c'est-à-dire si tous les exemples pointent sur la même valeur (étape 2). Si c'est le cas, alors une feuille est installée. Sinon, un attribut $\mathcal{V}$ est sélectionné par la mesure

d'information $\mathcal{M}$ via l'opérateur SelectAttr pour être installé au nœud k. Si k était une feuille, elle est remplacée par un nœud de décision. Si k était un nœud de décision testant un autre attribut que $\mathcal{V}$, alors les sous-arbres de k sont supprimés. Dans ces deux derniers cas, un nouveau nœud de décision testant l'attribut $\mathcal{V}$ est installé et l'ensemble de tous les exemples (y compris le dernier exemple $\langle \mathfrak{a}_t, \varsigma_t \rangle$) est distribué sur les nœuds enfants de k (étape 2(b)iii). Enfin, si k était déjà un nœud de décision testant l'attribut $\mathcal{V}$, alors aucune modification n'est nécessaire pour le nœud k et l'exemple $\langle \mathfrak{a}_t, \varsigma_t \rangle$ est simplement distribué au nœud enfant correspondant à l'exemple (étape 2b(Sinon)).

Concernant la fonction de mesure utilisée, à chaque fois qu'un exemple est ajouté, elle est évaluée et sélectionne un attribut. Aucune modification spécifique au cadre incrémental n'est nécessaire, les mesures basées sur le critère du χ^2 (section 4.1.2, page 84) et des moindres carrés (section 4.1.3, page 86), respectivement pour la classification et la régression, peuvent être utilisées sans modification.

Plusieurs alternatives ont été proposées, notamment ID5 et ID5R (Utgoff, 1986, 1988). Ces algorithmes présentent plusieurs avantages par rapport à l'algorithme UpdateTree. D'une part, ils sont peu coûteux en temps de calcul lorsque tous les attributs ont un score de la fonction de mesure équivalent, contrairement à UpdateTree qui risque de détruire puis reconstruire l'arbre à chaque nouvel exemple. D'autre part, ils ne requièrent pas de maintenir à chaque niveau de l'arbre la liste des exemples[2] en remplaçant l'étape 2(b)ii de suppression des sous-arbres de l'algorithme UpdateTree par un opérateur de réorganisation. Cependant, ces algorithmes ne conviennent pas pour l'apprentissage de fonctions stochastiques telles que nous en avons besoin pour l'apprentissage de la fonction de transition, notamment puisqu'il est difficile d'y intégrer un pré-élagage.

Une autre alternative concerne l'algorithme ITI (Utgoff et al., 1997). ITI propose notamment la gestion des valeurs manquantes dans les attributs d'une part et, les attributs à valeur continue d'autre part. Les algorithmes d'induction d'arbres de décision tels que nous les avons présentés jusqu'à maintenant installent un nœud de décision pour tester un attribut $\mathcal{V}$. Par conséquent, un nœud de décision contient une branche pour chaque valeur $\nu \in \text{Dom}(\mathcal{V})$. Au contraire, ITI installe à un nœud de décision un test de type "$\mathcal{V} = \nu$?", pour une valeur pour un attribut. Bien qu'une telle représentation puisse représenter certaines fonctions de façon plus compacte[3], elle nécessite des algorithmes plus complexes pour être manipulée par les opérateurs que nous avons présentés sur les arbres (section 3.2.2, page 49). De plus, nous supposons que pour un état donné à un pas de temps donné, la valeur de toutes les variables aléatoires composant l'état de l'agent est connue : la gestion de valeurs manquantes dans les attributs n'est donc pas nécessaire. De même, nous supposons que notre espace d'état est fini et discret : la gestion d'attributs à valeur continue n'est donc pas

[2]Notons qu'il est toujours possible avec l'algorithme UpdateTree de stocker les exemples uniquement aux feuilles de l'arbre et non à chaque nœud de l'arbre. En effet, lorsqu'un sous-arbre est détruit, il est possible de reconstruire la base d'exemples à la racine de ce sous-arbre en regroupant les exemples présents à ses feuilles.

[3]Notons que pour des arbres binaires, les deux représentations sont équivalentes.

nécessaire.

Enfin, une optimisation intéressante a été proposée par Kalles and Morris (1996). L'algorithme d'induction d'arbres de décision incrémental UpdateTree évalue la mesure d'information à chaque mise à jour, lorsqu'un exemple est ajouté. Or, lorsque l'écart des scores des différents attributs est important, plusieurs exemples sont nécessaires pour remettre en cause l'ordre calculé par la fonction de mesure entre les différents attributs. Ainsi, Kalles and Morris (1996) propose de calculer le nombre d'exemples nécessaires pour un éventuel changement d'ordre des attributs et ainsi de recalculer la mesure d'information une fois ce nombre d'exemples atteint. Ce calcul a été proposé pour la mesure d'information *gain*, nous n'avons pas adapté cette technique pour le critère du χ^2.

5.2.2 Apprentissage incrémental d'un FMDP

L'apprentissage incrémental d'un FMDP est très similaire à l'apprentissage d'un FMDP à partir d'un échantillon d'observations. La décomposition d'observations en exemples utilisée est la même que celle que nous avons proposée lors du chapitre précédent (section 4.2.1, page 88). Cette section commence par décrire l'algorithme d'induction d'arbres de décision incrémental à partir d'un flux d'exemples pour l'apprentissage des distributions de probabilités conditionnelles dans le cadre de la mise à jour de la fonction de transition du FMDP. Nous proposons ensuite les deux algorithmes de mise à jour d'un FMDP à partir d'un flux d'observations, nommément UpdateFMDP et UpdateFMDPnAT, les équivalents incrémentaux des deux algorithmes d'apprentissage hors-ligne BuildFMDP et BuildFMDPnAT et correspondant à l'algorithme UpdateModel de mise à jour du modèle dans SDYNA.

Induction incrémentale d'arbres de décision pour l'apprentissage de distributions de probabilités conditionnelles

De la même façon qu'avec l'algorithme d'induction d'arbres de décision BuildTreeS (section 4.2, page 86), nous utilisons le pré-élagage pour apprendre les distributions de probabilités conditionnelles de la fonction de transition du FMDP et éviter ainsi le développement inutile de l'arbre pour les transitions stochastiques.

L'algorithme incrémental d'induction d'arbres de décision UpdateTreeS pour l'apprentissage de fonctions stochastiques (Schlimmer and Fisher, 1986) est décrit figure 5.3. Il reprend les mêmes étapes que l'algorithme UpdateTreeS. Cependant, plutôt que de tester si l'ensemble d'exemples $\mathcal{E}_k$ présent au nœud k est pur, la fonction de mesure est utilisée pour déterminer si la différence entre les ensembles d'exemples distribués aux branches est significative (via l'opérateur IsDiffSig défini section 4.2, page 86). De plus, le contenu d'une feuille est défini à partir de l'ensemble d'exemples $\mathcal{E}_k$ par l'opérateur d'agrégation Aggregate (défini section 4.2, page 86). Enfin, le test réalisé par IsDiffSig peut être vrai pour un ensemble d'exemples puis devenir faux avec des exemples supplé-

Paramètre(s) : une mesure d'information $\mathcal{M}$

Initialisation : Initialiser Tree $[F]$

Soit k le nœud courant, $\mathcal{E}_k = \{\langle \mathfrak{a}_k, \varsigma_k \rangle\}$ les exemples présents au nœud k

A chaque exemple $\langle \mathfrak{a}_t, \varsigma_t \rangle$ **:**

1. Mettre à jour la mesure d'information $\mathcal{M}$ en ajoutant $\langle \mathfrak{a}_t, \varsigma_t \rangle$ à $\mathcal{E}_k$
2. **Si** IsDiffSig$(\mathcal{M}, \{\forall \nu \in$ Dom$(\mathcal{V}_i) : \mathcal{E}_\nu\})$ est faux

   Alors : transformer k en une feuille contenant : Aggregate$(\mathcal{M}, \{\forall \nu \in$ Dom$(\mathcal{V}_i) : \mathcal{E}_\nu\})$ (détruire les sous-arbres si nécessaire)

   Sinon :

   (a) Soit $\mathcal{V} \leftarrow$ SelectAttr$(\mathcal{M}, \mathcal{E})$

   (b) **Si** k ne teste pas $\mathcal{V}$ (k est une feuille ou un nœud de décision testant un autre attribut que $\mathcal{V}$) :

   Alors :

   i. Transformer k en un nœud de décision testant $\mathcal{V}$

   ii. Supprimer les sous-arbres de k (s'ils existent)

   iii. $\forall \langle \mathfrak{a}_j, \varsigma_j \rangle \in \mathcal{E}_k$: UpdateTreeS$(\langle \mathfrak{a}_j, \varsigma_j \rangle, k_{\mathfrak{a}_j[\mathcal{V}]})$ avec $k_{\mathfrak{a}_j[\mathcal{V}]}$ le nœud enfant de k correspondant à la branche $\mathfrak{a}_j[\mathcal{V}]$ (ajouter la branche si nécessaire)

   Sinon : UpdateTreeS$(\langle \mathfrak{a}_t, \varsigma_t \rangle, k_{\mathfrak{a}_t[\mathcal{V}]})$ avec $k_{\mathfrak{a}_t[\mathcal{V}]}$ le nœud enfant de k correspondant à la branche $\mathfrak{a}_t[\mathcal{V}]$ (ajouter la branche si nécessaire)

FIG. 5.3 – L'algorithme UpdateTreeS récursif utilisé pour mettre à jour un arbre à partir d'un flux d'exemples (Schlimmer and Fisher, 1986).

mentaires. Ainsi, un nœud de décision à la racine d'un sous-arbre peut être détruit pour devenir une feuille.

Un arbre par action par variable : l'algorithme UpdateFMDP

L'algorithme UpdateFMDP, décrit figure 5.4, met à jour un FMDP à chaque nouvelle observation de l'agent dans son environnement. On utilise un arbre par action et par variable pour représenter chaque distribution de probabilités conditionnelle $P_a(X'_i|s)$. Par conséquent, la décomposition d'une observation en exemples est la même que pour l'algorithme BuildFMDP (section 4.2.2, page 92) : à partir d'une observation $\langle s, a, s', r \rangle$, on construit pour chaque variable X_i correspondant à l'action qui a été effectuée un exemple $\langle \mathfrak{a} = s, \varsigma = s'[X_i] \rangle$ afin d'apprendre la distribution de probabilités conditionnelle $P_a(X'_i|s)$ (étape 1). De même, pour apprendre chaque récompense R_i, on construit un exemple $\langle \mathfrak{a} = \{s, a\}, \varsigma = r[R_i] \rangle$ (étape 2).

Paramètre(s) :

Initialisation : $\forall a \in A, \forall X_i \in X$: $\text{Tree}\left[P_a(X_i'|s)\right]$ et $\forall R_i \in R$: $\text{Tree}\left[R_i\right]$

A chaque observation $\langle s, a, s', r\rangle$ **:**

1. Faire $\forall X_i \in X$:
   (a) Soit $e = \langle \mathfrak{a} = s, \varsigma = s'[X_i]\rangle$
   (b) UpdateTreeS($\mathcal{M}_{\chi^2}$, $\text{Tree}\left[P_a(X_i'|s)\right], e$)
2. Faire $\forall R_i \in R$:
   (a) Soit $e = \langle \mathfrak{a} = \{s, a\}, \varsigma = r[R_i]\rangle$
   (b) UpdateTree($\mathcal{M}_{LS}$, $\text{Tree}\left[R_i\right], e$)

FIG. 5.4 – L'algorithme UpdateFMDP met à jour un FMDP à partir d'un flux d'observations de l'agent dans son environnement (dans SDYNA, correspond à l'algorithme de mise à jour du modèle UpdateModel).

Un arbre par variable : l'algorithme UpdateFMDPnAT

De façon similaire, l'algorithme UpdateFMDPnAT, décrit figure 5.5, met à jour un FMDP à chaque nouvelle observation de l'agent dans son environnement. On utilise un arbre par variable pour représenter chaque distribution de probabilités conditionnelle $P(X_i'|s, a)$: l'action est donc considérée comme un attribut. Par conséquent, la décomposition d'une observation en exemple est la même que pour l'algorithme BuildFMDPnAT (section 4.2.2, page 92) : à partir d'une observation $\langle s, a, s', r\rangle$, on construit pour chaque variable X_i un exemple $\langle \mathfrak{a} = \{s, a\}, \varsigma = s'[X_i]\rangle$ afin d'apprendre la distribution de probabilités conditionnelle $P(X_i'|s, a)$ (étape 1). De même, pour apprendre chaque récompense R_i, on construit un exemple $\langle \mathfrak{a} = \{s, a\}, \varsigma = r[R_i]\rangle$ (étape 2).

Pour que le FMDP construit par l'algorithme UpdateFMDPnAT, contenant seulement un arbre $\text{Tree}\left[P(X_i'|s, a)\right]$ par variable X_i, puisse être utilisé par les algorithmes de planification, on utilise l'opérateur Extract(T, X, x) (section 4.2.2, page 92) pour extraire les distributions de probabilités conditionnelles $\text{Tree}\left[P_a(X_i'|s)\right]$ pour chaque action à partir de $\text{Tree}\left[P(X_i'|s, a)\right]$, de la même façon que pour l'algorithme BuildFMDPnAT.

Nous pouvons remarquer que l'apprentissage par les algorithmes UpdateTree et UpdateTreeS s'effectue par une mise à jour des représentations de ces fonctions sous la forme d'arbres de décision à chaque nouvelle observation de l'agent. Ainsi, il n'est pas utile de construire des ensembles d'exemples à partir d'un échantillon d'observations.

Paramètre(s) :

Initialisation : $\forall X_i \in X$: Tree $[P(X'_i|s)]$ et $\forall R_i \in R$: Tree $[R_i]$

A chaque observation $\langle s, a, s', r \rangle$ **:**

1. Faire $\forall X_i \in X$:
   (a) Soit $e = \langle \mathfrak{a} = \{s, a\}, \varsigma = s'[X_i] \rangle$
   (b) UpdateTreeS($\mathcal{M}_{\chi^2}$, Tree $[P(X'_i|s, a)]$, e)
2. Faire $\forall R_i \in R$:
   (a) Soit $e = \langle \mathfrak{a} = \{s, a\}, \varsigma = r[R_i] \rangle$
   (b) UpdateTree($\mathcal{M}_{LS}$, Tree $[R_i]$, e)

FIG. 5.5 – L'algorithme UpdateFMDPnAT construit un FMDP à partir d'un ensemble d'observations de l'agent dans son environnement (dans SDYNA, correspond à l'algorithme de mise à jour du modèle UpdateModel).

5.3 Intégration de la planification dans SDYNA

La section précédente présente les algorithmes UpdateFMDP et UpdateFMDPnAT pour mettre à jour à chaque nouvelle observation le FMDP utilisé par l'agent pour représenter le problème à résoudre. Une fois que cette mise à jour a été effectuée, il est nécessaire de mettre à jour la politique de l'agent afin que celle-ci prenne en compte les modifications réalisées dans le FMDP. Bien qu'il soit possible d'utiliser directement les algorithmes de planification dans les FMDPs présentés lors du chapitre 3 et comme nous l'avons proposé avec l'apprentissage hors-ligne lors du chapitre 4, plusieurs raisons incitent à les adapter à l'apprentissage incrémental réalisé par SDYNA.

D'une part, certains algorithmes de planification, plus particulièrement SPI, SVI et SPUDD, construisent une représentation explicite de la politique gloutonne de l'agent. Cette représentation est inutile puisqu'il est plus coûteux de la construire que d'utiliser les fonctions de valeur d'actions pour déterminer pour l'état courant la meilleure action estimée à effectuer. De plus, calculer une politique gloutonne peut être inutile pour d'autres politiques d'exploration que ϵ-greedy (par exemple, une politique d'exploration *softmax* (Sutton and Barto, 1998)).

D'autre part, les algorithmes de planification calculent une évaluation de la fonction de valeur optimale du FMDP représentant le problème jusqu'à ce que celle-ci ait convergée. Or ces itérations peuvent ne pas être utiles puisque le FMDP sera mis à jour à la prochaine observation, remettant en cause la fonction de valeur optimale qui aura été calculée. De plus, la valeur courante de la fonction de valeur est souvent suffisante pour déterminer la meilleure action à réaliser par l'agent bien avant

que le critère de convergence soit satisfait : il n'est donc pas forcément nécessaire d'attendre la fin de la convergence pour que l'agent détermine l'action à réaliser lors de la phase de décision dans SDYNA.

Enfin, les algorithmes de planification que nous avons présentés jusqu'à présent ne sont pas incrémentaux. Par conséquent, ils ne permettent pas de réutiliser les résultats qui auraient été déterminés lors de calculs antérieurs au pas de temps précédent.

Il est donc nécessaire d'adapter les algorithmes de planification que nous avons présentés lors du chapitre 3 afin qu'ils puissent être utilisés dans le cadre incrémental de SDYNA. Les sections suivantes présentent les adaptations que nous proposons pour les algorithmes SVI (section 5.3.1), SPUDD (section 5.3.2) et FactoredLPA (section 5.3.3).

5.3.1 Intégration de l'algorithme SVI

Entrée(s) : $\hat{\mathcal{F}}_t, \mathsf{Tree}\left[V_{t-1}\right]$ **Sortie(s) :** $\mathsf{Tree}\left[V_t\right], \{\mathsf{Tree}\left[Q_a^t\right], \forall a \in A\}$

---

1. Pour chaque action $a \in A$: $\mathsf{Tree}\left[Q_a^t\right] \leftarrow$ Regress($\mathsf{Tree}\left[V_{t-1}\right], a$)
2. $\mathsf{Tree}\left[V_t\right] \leftarrow$ Merge($\{\mathsf{Tree}\left[Q_a^t\right] : \forall a \in A\}$) en utilisant la maximisation comme opérateur de combinaison
3. Retourner $\mathsf{Tree}\left[V_t\right]$ et $\{\mathsf{Tree}\left[Q_a^t\right], \forall a \in A\}$

FIG. 5.6 – Algorithme IncSVI : version incrémentale de l'algorithme SVI (dans SDYNA, correspond à l'algorithme IncPlan de mise à jour des fonctions de valeurs d'action $\{\mathsf{Tree}\left[Q_a^t\right], \forall a \in A\}$).

Nous commençons par proposer une version incrémentale de l'algorithme SVI, que nous appelons IncSVI et qui est décrit figure 5.6. Afin d'éviter un temps de calcul trop important, l'algorithme IncSVI n'effectue qu'une seule itération de l'algorithme SVI. À partir de la fonction de valeur $\mathsf{Tree}\left[V_{t-1}\right]$ calculée lors du pas de temps précédent et de la version courante du FMDP $\hat{\mathcal{F}}_t$, l'ensemble des fonctions de valeur d'action $\{\mathsf{Tree}\left[Q_a^t\right]\}$ est mis à jour (étape 1). Ces fonctions sont utilisées pour construire une nouvelle fonction de valeur $\mathsf{Tree}\left[V_t\right]$ (étape 2) et retournées pour être utilisées afin de déterminer la prochaine décision de l'agent (algorithme Acting dans SDYNA). La fonction valeur $\mathsf{Tree}\left[V_t\right]$ est aussi retournée pour être utilisée lors du prochain pas de temps. Si le FMDP $\hat{\mathcal{F}}$ construit par l'apprentissage devient stationnaire, alors l'algorithme IncSVI convergera vers la fonction de valeur optimale $\mathsf{Tree}\left[\hat{V}^*\right]$ de $\hat{\mathcal{F}}$ représentant le problème à résoudre.

À partir de l'algorithme IncSVI décrit figure 5.6, plusieurs variantes peuvent être utilisées. Nous en avons principalement développé deux que nous avons utilisées pour obtenir les résultats présentés par la suite (section 5.4, page 142). La première concerne les valeurs manquantes dans

les arbres construits par l'apprentissage. La deuxième concerne une heuristique utilisée lorsque la structure de la fonction de récompense change.

Gestion des valeurs manquantes

Nous avons vu section 4.2.2 (page 96) que la construction des arbres $\mathsf{Tree}\left[P_a(X_i|s)\right]$ pouvait engendrer des feuilles vides si aucune précaution n'était prise. Les dispositions présentées section 4.2.2 peuvent directement être utilisées dans le cadre de SDYNA et d'un apprentissage incrémental sans nécessité d'adaptation spécifique. Nous présentons une solution supplémentaire, adaptée seulement à un apprentissage en ligne, afin de gérer ce problème dans le cadre de l'intégration de l'algorithme SVI dans SDYNA.

Concernant l'algorithme SVI, une feuille vide d'un arbre $\mathsf{Tree}\left[P_a(X_i|s)\right]$ est gênant si la variable X_i est utilisée dans la fonction de valeur. En effet, lors du calcul des fonctions de valeur d'action par l'algorithme Regress($\mathsf{Tree}\left[V\right], a$) (figure 3.9, page 51), l'espérance de la fonction de valeur $\sum_{s'} P(s'|s,a)V(s')$, représentée par l'arbre $\mathsf{Tree}\left[P_a^V V\right]$, est calculée à partir de la représentation factorisée $\mathsf{Tree}\left[P_a^V\right]$ de la fonction de transition. Or, si un ou plusieurs arbres $\mathsf{Tree}\left[P_a(X_i|s)\right]$ contiennent des feuilles vides, alors cette représentation $\mathsf{Tree}\left[P_a^V\right]$ contiendra aussi des feuilles vides, empêchant de calculer la somme pondérée $v_b = \sum_{b' \in \mathsf{Tree}[V]} P^b(b')V(b')$ contenue aux feuilles de $\mathsf{Tree}\left[P_a^V V\right]$.

Contrairement à un apprentissage hors-ligne, l'apprentissage en ligne permet à un agent d'adapter sa politique pendant son expérience, notamment de corriger des erreurs d'évaluation de valeurs espérées lorsque celles-ci sont trop optimistes. Ainsi, le problème des valeurs manquantes peut simplement être résolu en attribuant une valeur optimiste à la somme pondérée v_b lorsque la probabilité $P^b(b')$ fait référence à des feuilles vides de $\mathsf{Tree}\left[P_a^V\right]$ afin d'encourager l'agent à obtenir la donnée manquante. Par exemple, on assigne à v_b la valeur maximum $\hat{R}_{\text{MAX}}/(1-\gamma)$, avec $\hat{R}_{\text{MAX}}$ la récompense maximum que l'agent a obtenu, lorsque la probabilité $P^b(b')$ ne peut pas être calculée. Nous verrons qu'une telle approche constitue la base de certaines méthodes d'exploration dirigée (voir le chapitre 6 dédié à ce sujet). Les résultats présentés section 5.4 utilisent l'algorithme de planification incrémental IncSVI adapté de SVI exploitant cette heuristique.

Changement de structure de la fonction de récompense

Au cours de l'apprentissage, la structure de la fonction de récompense $\mathsf{Tree}\left[R\right]$ peut changer par l'algorithme de mise à jour UpdateTree. Or, la structure de la fonction de valeur $\mathsf{Tree}\left[V\right]$ est étroitement liée à celle de la fonction de récompense. Ainsi, lorsque la structure de la fonction de récompense $\mathsf{Tree}\left[R\right]$ change suite à une mise à jour par l'algorithme d'apprentissage, souvent, des changements très importants s'effectuent aussi dans la structure de la fonction valeur.

Ainsi, nous avons observé que, lorsque la structure de la fonction de récompense a changé, la

mise à jour de la fonction valeur Tree $[V]$ se traduit par une augmentation importante de la taille de Tree $[V]$ pour ensuite revenir à une taille plus modeste correspondant à la structure de la fonction valeur optimale dans le FMDP appris.

Afin d'éviter ce sur-coût de calcul, nous proposons l'heuristique suivante dans une version modifiée de l'algorithme de planification incrémental IncSVI : lorsque la structure de la fonction de récompense Tree $[R]$ a changé lors de la dernière phase d'apprentissage, alors les fonctions de valeur d'actions Tree $[Q_a^t]$ sont calculées à partir d'une fonction de valeur réinitialisée Tree $[V_0]$ plutôt que la fonction de valeur au pas de temps précédent Tree $[V_{t-1}]$.

Bien qu'elle telle heuristique puisse économiser de l'espace mémoire et du temps de calcul, elle a un impact négatif sur la qualité de la politique de l'agent pendant quelques pas de temps. En effet, l'algorithme IncSVI n'effectue qu'une seule itération par pas de temps alors que plusieurs itérations seraient nécessaires pour obtenir une politique adaptée au FMDP appris.

5.3.2 Intégration de l'algorithme SPUDD

L'intégration de l'algorithme SPUDD est très similaire à celle de SVI. Cependant, pour des contraintes purement techniques de programmation, une différence existe dans notre implémentation par rapport à l'adaptation incrémentale de SVI : la politique de l'agent est construite à chaque pas de temps. La version incrémentale de l'algorithme SPUDD est décrite dans la figure 5.7.

Entrée(s) : $\hat{\mathcal{F}}_t$, ADD $[V_{t-1}]$ **Sortie(s) :** ADD $[V_t]$, ADD $[\pi_t]$

---

1. Pour chaque action $a \in A$: ADD $[Q_a^t] \leftarrow$ Regress(ADD $[V_{t-1}]$, a)
2. ADD $[V_t] \leftarrow$ Merge($\{$ADD $[Q_a^t] : \forall a \in A\}$) en utilisant la maximisation comme opérateur de combinaison
3. ADD $[\pi_t] \leftarrow$ Greedy(ADD $[V_t]$)
4. Retourner ADD $[\pi_t]$, ADD $[V_t]$

FIG. 5.7 – Algorithme IncSPUDD : version incrémentale de l'algorithme SPUDD (dans SDYNA, correspond à l'algorithme IncPlan).

A l'instar de l'algorithme IncSVI, l'algorithme effectue une itération de *Value Iteration* en commençant par calculer la fonction de valeur d'action ADD $[Q_a^t]$ pour chaque action à partir de la fonction de valeur ADD $[V_{t-1}]$ calculée au pas de temps précédent (étape 1). Ensuite, une opération de maximisation sur l'ensemble de ces fonctions de valeur d'action est réalisée pour calculer une nouvelle fonction de valeur ADD $[V_t]$ (étape 2). Enfin, à partir de la fonction de valeur ADD $[V^t]$, une politique gloutonne ADD $[\pi_t]$ est calculée (étape 3). Ainsi, l'algorithme de décision Acting de

SDYNA utilise la politique ADD $[\pi_t]$ pour prendre sa décision plutôt que d'utiliser l'ensemble des fonctions de valeur d'action.

Lors du chapitre 3, nous avons décrit l'ensemble des opérateurs Regress, Merge et Greedy dans le cadre de SVI (section 3.2, page 43), en utilisant les arbres comme structures de données pour représenter les fonctions du problème. Bien qu'ayant une implémentation sensiblement différente, nous n'avons pas décrit ces opérateurs tels qu'ils sont utilisés dans le cadre de SPUDD et de l'utilisation d'ADDs comme structure de données. Cependant, aussi bien dans SPUDD que dans SVI, ces opérateurs représentent les mêmes opérations et sont, par conséquent, utilisés de la même façon. Nous invitons donc le lecteur intéressé par les détails de ces opérateurs à consulter les références correspondantes (Hoey et al., 1999, 2000; St-Aubin et al., 2000).

5.3.3 Intégration de la programmation linéaire approchée

L'intégration de la programmation linéaire approchée dans le cadre de SDYNA pose plusieurs difficultés supplémentaires par rapport aux approches basées sur la programmation dynamique telle que SVI et SPUDD. Outre le fait qu'un ensemble de fonctions de base doit être défini a priori, chaque mise à jour effectuée par l'apprentissage dans le FMDP modifie l'ensemble de contraintes du programme linéaire, rendant ainsi les solutions précédentes difficilement réutilisables. Par conséquent, l'algorithme d'intégration de la programmation linéaire approchée dans SDYNA que nous proposons constitue plus une démonstration de la faisabilité du concept plutôt qu'une véritable adaptation de cette approche à un cadre incrémental.

L'algorithme IncLP, décrit 5.8, utilise le fait que lorsque la structure du FMDP ne change pas lors de l'apprentissage, alors la solution qui a été calculée lors du pas de temps précédent peut être utilisée pour résoudre le programme linéaire généré. À l'inverse, afin d'éviter d'avoir à calculer une solution à chaque changement de structure effectué par l'apprentissage, une période tampon T_M est utilisée pour attendre que le FMDP appris $\hat{\mathcal{F}}_t$ se stabilise. Si aucune stabilisation dans la structure ne se produit au bout d'une période limite T_P, alors une phase de planification est déclenchée afin de mettre à jour la politique de l'agent. Lorsque la structure du FMDP ne change pas, on évite un temps de calcul trop important par pas de temps en restreignant la fréquence de mise à jour à une période définie par T_{MIN}.

Ainsi, l'algorithme IncLP commence par vérifier si la structure du FMDP $\hat{\mathcal{F}}_t$ a changé par rapport à celle de $\hat{\mathcal{F}}_{t-1}$ (étape 1). Si c'est le cas la solution du dernier programme linéaire généré est réinitialisée. Ensuite, lors de l'étape 2, si la structure du FMDP n'a pas changé pendant au moins T_M pas de temps et que la dernière phase de planification a été effectuée il y a plus de T_{min} pas de temps, ou que cette phase de planification a été réalisée il y a plus de T_P pas de temps, alors la fonction de valeur $\tilde{V}_t$ et les fonctions de valeur d'action $\tilde{Q}^t_a$ sont mises à jour par l'algorithme FactoredLPA($\hat{\mathcal{F}}_t$). Dans le cas contraire, alors la fonction de valeur $\tilde{V}_t$ et les fonctions de valeur d'action $\tilde{Q}^t_a$ ne sont

---

Entrée(s) : $\hat{\mathcal{F}}_t, \tilde{V}_{t-1}$ **Sortie(s) :** $\tilde{V}_t, \{\tilde{Q}^t_a, \forall a \in A\}$

---

1. **Si** la structure du FMDP $\hat{\mathcal{F}}_t$ a été modifié lors de l'apprentissage au dernier pas de temps (structure de $\hat{\mathcal{F}}_t \neq$ structure de $\hat{\mathcal{F}}_{t-1}$) :

   Alors :

   (a) lastModif $\leftarrow t$

   (b) Réinitialiser la solution du dernier programme linéaire généré

2. **Si** $(($t$ -$ lastModif $> T_M)$ **ou** $(t -$ lastPlanning $< T_P))$ **et** $(t -$ lastPlanning $> T_{MIN})$:

   Alors :

   (a) lastPlanning $\leftarrow t$

   (b) $\{\tilde{V}_t, \{\tilde{Q}^t_a, \forall a \in A\}\} \leftarrow$ FactoredLPA$(\hat{\mathcal{F}}_t)$ en réutilisant la solution du dernier programme linéaire généré si elle n'a pas été réinitialisée.

   Sinon : $\{\tilde{V}_t, \{\tilde{Q}^t_a, \forall a \in A\}\} \leftarrow \{\tilde{V}_{t-1}, \{\tilde{Q}^{t-1}_a, \forall a \in A\}\}$

3. Retourner $\tilde{V}_t$ et $\{\tilde{Q}^t_a, \forall a \in A\}$

FIG. 5.8 – Algorithme IncLP : version incrémentale de la programmation linéaire approchée (dans SDYNA, correspond à l'algorithme IncPlan de mise à jour des fonctions de valeurs d'action $\{\tilde{Q}^t_a, \forall a \in A\}$).

pas mises à jour et les valeurs au pas de précédent sont utilisées sans modification.

5.4 Résultats

Les résultats que nous présentons dans cette section illustrent de façon expérimentale la mise en œuvre de SDYNA utilisant les algorithmes UpdateFMDP et UpdateFMDPnAT (section 5.2.2, page 134) de mises à jour d'un FMDP, les algorithmes de planifications incrémentaux IncSVI (section 5.3.1, page 138), IncSPUDD (section 5.3.2, page 140) et IncLP (section 5.3.3, page 141) et une politique d'exploration ϵ-greedy, sur un ensemble de problèmes exposés lors du chapitre précédent. Afin de nommer les agents basés sur l'approche SDYNA en fonction des algorithmes qu'ils utilisent, nous utilisons la convention suivante :

- l'algorithme utilisé pour la mise à jour du modèle détermine le préfixe du nom de l'agent, avec le préfixe U pour l'algorithme UpdateFMDP et UNAT pour l'algorithme UpdateFMDPnAT,
- l'algorithme utilisé pour la planification détermine le suffixe du nom de l'agent, avec le suffixe SVI pour l'algorithme IncSVI, SPUDD pour l'algorithme IncSPUDD et LP pour l'algorithme IncLP.

Par exemple, l'agent nommé USVI désigne un agent utilisant l'algorithme UpdateFMDP pour mettre à jour son modèle et l'algorithme IncSVI pour mettre à jour ses fonctions de valeur d'action, un agent nommé UNATLP désigne un agent utilisant l'algorithme UpdateFMDPnAT pour mettre à jour son modèle et l'algorithme IncLP pour mettre ses fonctions de valeur d'action. Concernant les paramètres des agents SDYNA, de même que pour le chapitre précédent, nous utilisons une valeur de seuil $\tau_{\chi^2} = 30$ pour l'ensemble des problèmes traités. Les expériences se déroulent avec les même conditions expérimentales. Concernant l'algorithme d'exploration ϵ-greedy, nous utilisons $\epsilon = 0.1$. De plus, lorsque la meilleure action doit être sélectionnée et que plusieurs d'entre elles sont considérées comme meilleures et équivalentes, alors l'une d'entre elle est choisie de façon aléatoire avec une distribution uniforme.

Afin de comparer l'approche SDYNA à une approche d'apprentissage par renforcement avec modèle, nous avons aussi testé une version stochastique de l'algorithme DYNA-Q, telle que nous l'avons décrite section 2.3.2 (page 34) sur l'ensemble des problèmes. La même politique d'exploration que SDYNA est utilisée, c'est-à-dire ϵ-greedy avec $\epsilon = 0.1$. Enfin, les fonctions de valeur d'action $Q_a(s)$ sont initialisées avec la valeur 0.

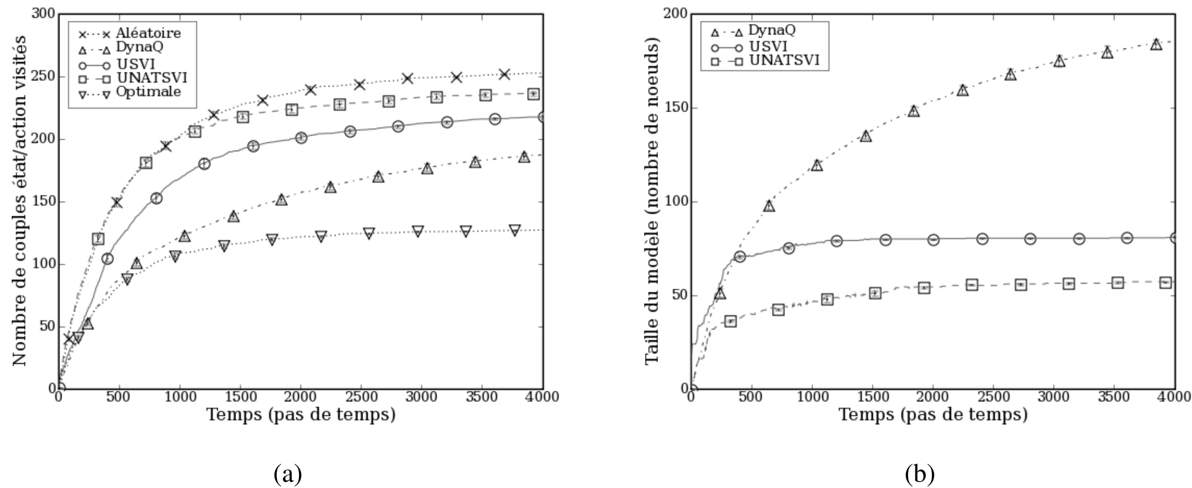
Enfin, les algorithmes DYNA-Q et ceux basés sur l'approche SDYNA sont comparés à deux agents étalons exécutant pour le premier une politique optimale et pour le deuxième une politique aléatoire.

5.4.1 Le problème *Coffee Robot*

Le premier problème que nous traitons est celui de *Coffee Robot* défini section 2.1 (page 24). Le problème est constitué de 6 variables binaires et 4 actions (256 couples état/action) et représente donc un petit problème d'apprentissage par renforcement stochastique. De même que pour le chapitre précédent, la position de l'agent est réinitialisée à une position aléatoire dans le problème tous les 15 pas de temps. Nous utilisons les deux agents basés sur l'approche SDYNA suivants : USVI et UNATSVI. Le premier utilisant l'algorithme UpdateFMDP pour mettre à jour le modèle, le deuxième utilisant l'algorithme UpdateFMDPnAT. Les deux utilisent l'algorithme IncSVI pour la planification.

La figure 5.9 montre le nombre de couples état/action visités par les agents (figure 5.9(a)) et la taille du modèle construit par les algorithmes d'apprentissage (figure 5.9(b)). On observe ainsi que DYNA-Q est l'agent explorant le moins l'ensemble des couples état/action du problème, bien qu'utilisant le même algorithme d'exploration que les agents SDYNA. Concernant les agents USVI et UNATSVI, ils explorent plus que DYNA-Q et visitent donc un nombre plus important de couples état/action, mais sans atteindre l'ensemble des couples état/action possibles du problème. En comparant la taille des représentations de la fonction de transition des agents, on remarque une nette différence : moins de 100 nœuds pour les agents basé sur l'approche SDYNA contre plus de 180

nœuds pour DYNA-Q. On observe aussi que la taille de la représentation de la fonction de transition pour l'agent DYNA-Q correspond au nombre de couples état/action visités par cet agent.



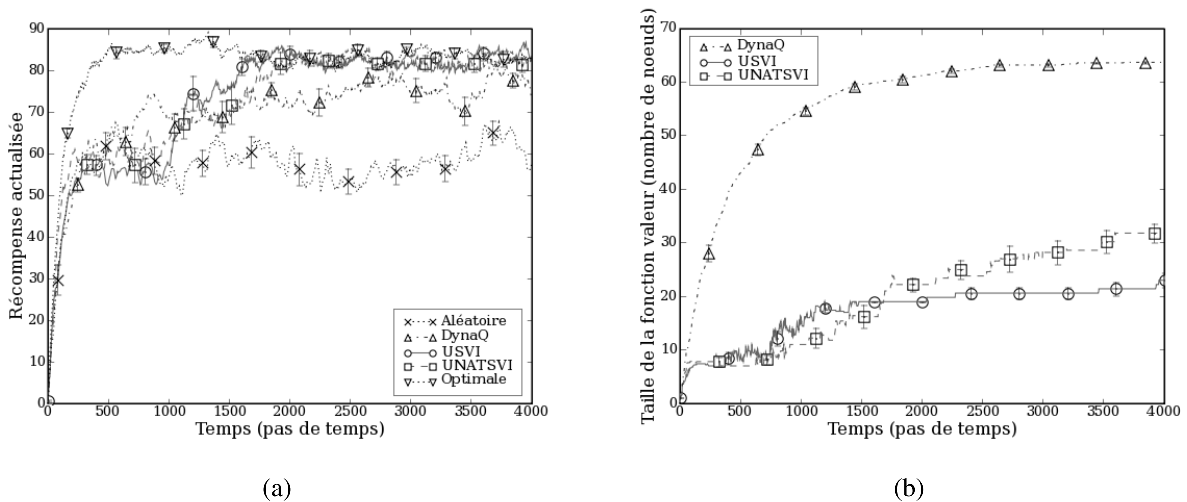
(a) (b)

FIG. 5.9 – Problème *Coffee Robot* : nombre de couples état/action visités par les agents (figure a) et taille de la fonction de transition construite par apprentissage (figure b). Bien que le nombre de couples état/action visités par l'agent DYNA-Q soit inférieur à celui des agents SDYNA, la fonction de transition construite par DYNA-Q est moins compacte que la fonction de transition du FMDP construit par les algorithmes UpdateFMDP et UpdateFMDPnAT.

La figure 5.10 montre la récompense actualisée obtenue par chacun des agents (figure 5.10(a)) et la taille de leur fonction de valeur (figure 5.10(b)). D'une part, les trois agents USVI, UNATSVI et DYNA-Q arrivent rapidement à exécuter une politique meilleure que la politique aléatoire (environ après 1000 pas de temps). Cependant, contrairement aux deux agents SDYNA, la récompense actualisée obtenue par DYNA-Q reste inférieure à celle obtenue par la politique optimale. D'autre part, la taille de représentation de la fonction de valeur est plus compacte concernant les agents USVI et UNATSVI (moins de 40 nœuds) comparé à DYNA-Q dont la taille de la représentation correspond au nombre d'états dans le problème (64 nœuds).

Sur un petit problème d'apprentissage par renforcement tel que *Coffee Robot*, les performances des agents DYNA et SDYNA sont quasiment similaires. La légère différence existante concernant la récompense actualisée obtenue par DYNA-Q peut être expliquée par le nombre moins important de couples état/action visités par DYNA-Q, comparés aux agents SDYNA. Nous rappelons que la politique d'exploration est exactement la même (ϵ-greedy) pour l'agent DYNA-Q et les agents SDYNA.

Les couples état/action supplémentaires visités par les agents SDYNA peuvent provenir du fait que, tant que certains tests ne sont pas installés dans les arbres, certaines actions peuvent être considérées équivalentes alors qu'elles ne le sont pas. Dès lors, lorsque la meilleure action doit être



(a) (b)

FIG. 5.10 – Problème *Coffee Robot* : récompense actualisée obtenue par les agents (figure a) et taille de la fonction de valeur calculée par l'algorithme de planification de l'agent (figure b). Alors que l'algorithme DYNA-Q construit une fonction de valeur de la même taille que le nombre de couples état/action visités, les agents SDYNA construisent une représentation plus compacte de la fonction de valeur et construisent une meilleure politique.

choisie lors de la sélection de l'action, des actions considérées comme étant équivalentes seront choisies avec une distribution uniforme, participant ainsi à l'exploration. Les fonctions de valeur d'action de DYNA-Q étant initialisées à 0, DYNA-Q aura plutôt tendance à sélectionner la dernière action essayée, si l'état à déjà été rencontré, limitant ainsi l'exploration. Des résultats supplémentaires, notamment concernant l'algorithme DYNA-Q avec une initialisation optimiste des fonctions de valeur de d'action, sont disponibles dans Degris et al. (2006b).

Enfin, il est important de noter que les représentations structurées utilisées par les agents SDYNA permettent, d'une part, aux algorithmes d'apprentissage de représenter de façon plus compacte la fonction de transition et, d'autre part, à l'algorithme de planification de construire une représentation compacte de la fonction de valeur, bien que les agents SDYNA visitent un plus grand nombre de couples état/action que l'agent DYNA-Q.

5.4.2 Le problème *Factory*

Le deuxième problème que nous traitons est le problème *Factory* défini lors du chapitre précédent, section 4.3.3 (page 117). Le problème est constitué de 17 variables binaires et 14 actions (1 835 008 couples état/action). De même que pour le chapitre précédent, la position de l'agent est réinitialisée à une position aléatoire dans le problème tous les 15 pas de temps. Nous utilisons les même agents SDYNA que pour le problème précédent, nommément USVI et UNATSVI : le premier

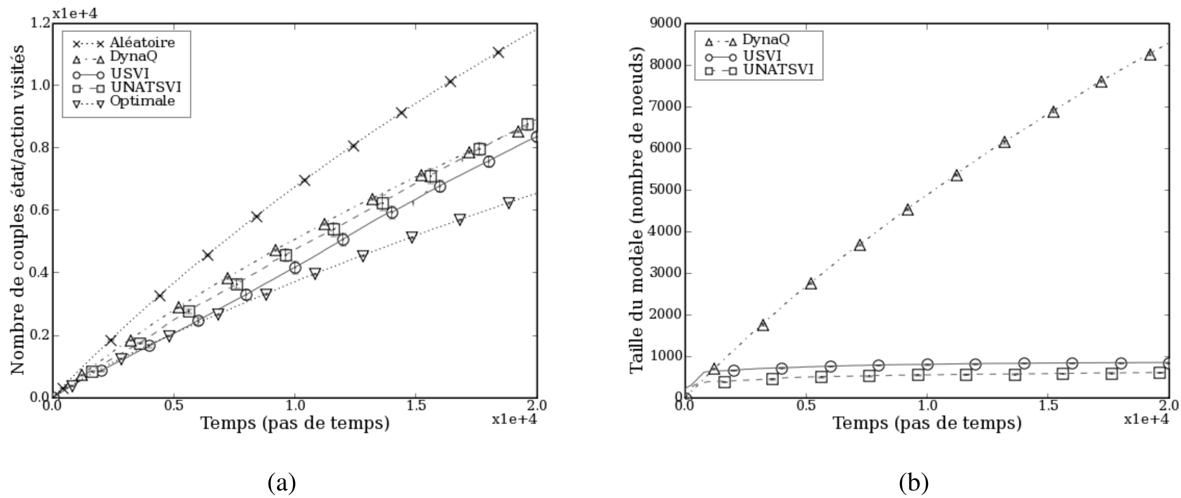
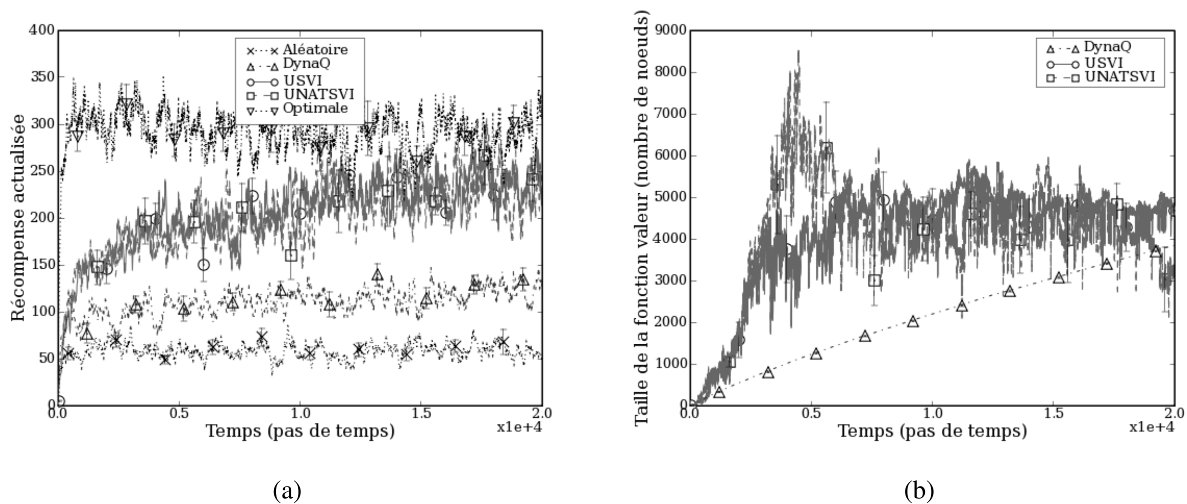


FIG. 5.11 – Problème *Factory* : nombre de couples état/action visités par les agents (figure a) et taille de la fonction de transition construite par apprentissage (figure b). Alors que la fonction de transition construite par DYNA-Q croît au fur et à mesure des couples état/action visités dans le problème, la taille de la fonction de transition du FMDP construite par les algorithmes UpdateFMDP et UpdateFMDPnAT se stabilise, même lorsque de nouveaux couples état/action sont essayés.

utilisant l'algorithme UpdateFMDP pour mettre à jour le modèle, le deuxième utilisant l'algorithme UpdateFMDPnAT. Les deux utilisent l'algorithme IncSVI pour la planification.

La figure 5.11 représente le nombre de couples état/action visités par les agents (figure 5.11(a)) et la taille du modèle construit par les algorithmes d'apprentissage (figure 5.11(b)). On observe ainsi que les agents DYNA-Q, USVI et UNATSVI explorent un nombre semblable de couples état/action (un peu plus de 8000 nœuds). Une fois de plus, la taille de la représentation de la fonction de transition construite par ces agents est sensiblement différente : alors que DYNA-Q construit une représentation de taille égale au nombre de couples état/action visités (donc une taille de plus de 8000 nœuds), les agents USVI et UNATSVI construisent des représentations beaucoup plus compactes de moins de 1000 nœuds.

La figure 5.12 montre la récompense actualisée obtenue par chacun des agents (figure 5.12(a)) et la taille de leur fonction de valeur (figure 5.12(b)). D'une part, on peut donc observer que les trois agents USVI, UNATSVI et DYNA-Q arrivent rapidement à exécuter une politique meilleure qu'une politique aléatoire. Cependant, aucun des agents n'exécutent une politique équivalente à la politique optimale. En comparant les récompenses obtenues par la politique de l'agent DYNA-Q et celle des agents USVI et UNATSVI, on remarque une nette différence, à l'avantage des agents basés sur l'approche SDYNA, après 5000 pas de temps. Cette différence de qualité dans les politiques s'obtient au prix d'une taille de la fonction de valeur rapidement plus importante que celle de DYNA-Q : l'arbre représentant la fonction de valeur contient environ 5000 nœuds alors que DYNA-



(a) (b)

FIG. 5.12 – Problème *Factory* : récompense actualisée obtenue par les agents (figure a) et taille de la fonction valeur calculée par l'algorithme de planification de l'agent (figure b). Alors que l'algorithme DYNA-Q exécute une politique légèrement meilleure qu'une politique aléatoire, les agents SDYNA améliorent nettement leur politique par rapport à celle de DYNA-Q ou d'une politique aléatoire, au prix d'une représentation de la fonction de valeur plus importante en taille, mais se stabilisant avec le temps.

Q utilise une représentation tabulaire arrivant à moins de 4000 lignes à la fin de l'expérience.

Les résultats obtenus pour ce problème illustrent bien les propriétés de généralisation importante des algorithmes d'induction d'arbres de décision utilisés avec un algorithme de planification dans les FMDPs. En effet, alors que les agents SDYNA visitent un nombre similaire de couple état/action que DYNA-Q, la récompense actualisée obtenue par ces agents est nettement meilleure.

La mauvaise performance de l'algorithme DYNA-Q peut être expliqué par le fait que l'algorithme utilise une représentation atomique des états. Par conséquent, une observation effectuée par l'agent ne concerne qu'une transition dans le problème. Au contraire, le formalisme des FMDPs et la généralisation utilisée par l'induction d'arbre de décision a pour conséquence qu'une observation effectuée par l'agent concerne chacune des variables du problème, dans des contextes différents pouvant être très généraux, c'est-à-dire représentant une grande partie de l'espace d'état. Ainsi, l'information apportée par l'observation ne concerne plus seulement la transition que l'agent vient d'effectuer mais, potentiellement, un grand nombre de transitions dans le problème.

Cette généralisation provoque un effet non négligeable pour la planification : en comparant les tailles des fonctions de valeur, on remarque que celles des agents SDYNA sont beaucoup plus importantes que celle de DYNA-Q. Cependant, on peut remarquer que la taille de la représentation tabulaire de DYNA-Q augmente linéairement avec le nombre de couples état/action visités par l'agent. Par conséquent, on peut s'attendre à ce que la taille de la représentation de la fonction de

valeur de DYNA-Q continue de croître au fur et à mesure que de nouveaux couples état/action sont visités par l'agent, contrairement à la représentation structurée utilisée par SDYNA qui se stabilise après 6 000 pas de temps.

5.4.3 Le problème *Factory4*

Le troisième problème que nous traitons est le problème *Factory4* défini lors du chapitre précédent, section 4.3.3 (page 121). Le problème est constitué de 28 variables binaires et 15 actions ($4,0 \cdot 10^9$ couples état/action). De même que pour le chapitre précédent, la position de l'agent est réinitialisée à une position aléatoire dans le problème tous les 15 pas de temps. Nous utilisons les agents SDYNA suivant : USPUDD et UNATSPUDD : le premier utilisant l'algorithme UpdateFMDP pour mettre à jour le modèle, le deuxième utilisant l'algorithme UpdateFMDPnAT. Les deux utilisent l'algorithme IncSPUDD pour la planification.

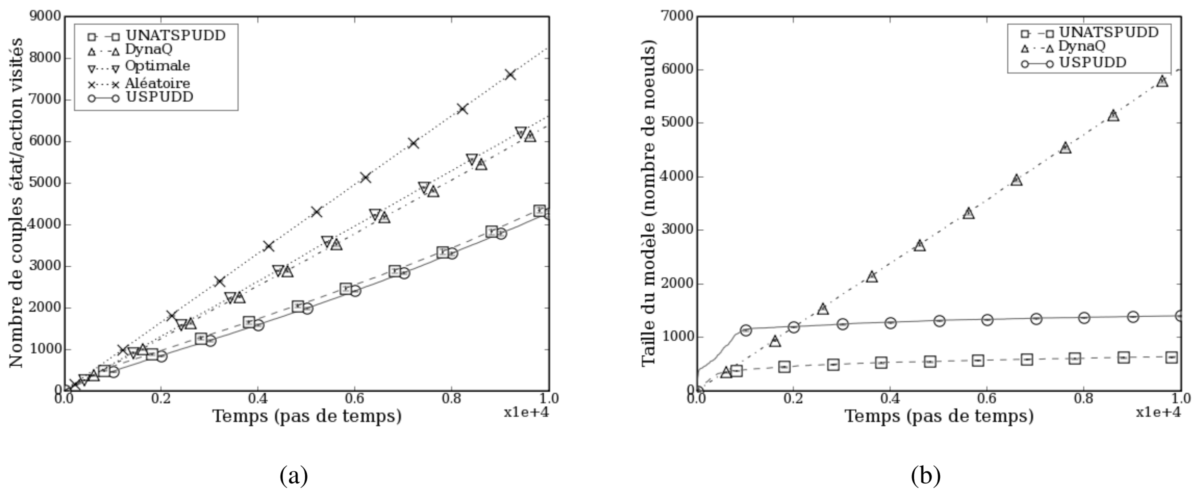


FIG. 5.13 – Problème *Factory4* : nombre de couples état/action visités par les agents (figure a) et taille de la fonction de transition construite par apprentissage (figure b). Alors que la fonction de transition construite par DYNA-Q croît au fur et à mesure des couples état/action visités dans le problème, la taille de la fonction de transition du FMDP construite par les algorithmes UpdateFMDP et UpdateFMDPnAT se stabilise alors que de nouveaux couples état/action sont essayés.

La figure 5.13 représente le nombre de couples état/action visités par les agents (figure 5.13(a)) et la taille du modèle construit par les algorithmes d'apprentissage (figure 5.13(b)). En premier lieu, on peut remarquer que les agents USPUDD et UNATSPUDD explorent un nombre moins important de couples état/action (moins de 5000 couples état/action) que DYNA-Q ou que l'agent exécutant une politique optimale. En deuxième lieu, la taille de la représentation de la fonction de transition construite par ces agents est, de même que pour les problèmes précédents, sensiblement différente :

alors que DYNA-Q construit une représentation de taille d'environ 6000 nœuds, les agents USPUDD et UNATSPUDD construisent des représentations beaucoup plus compactes de moins de 2000 nœuds.

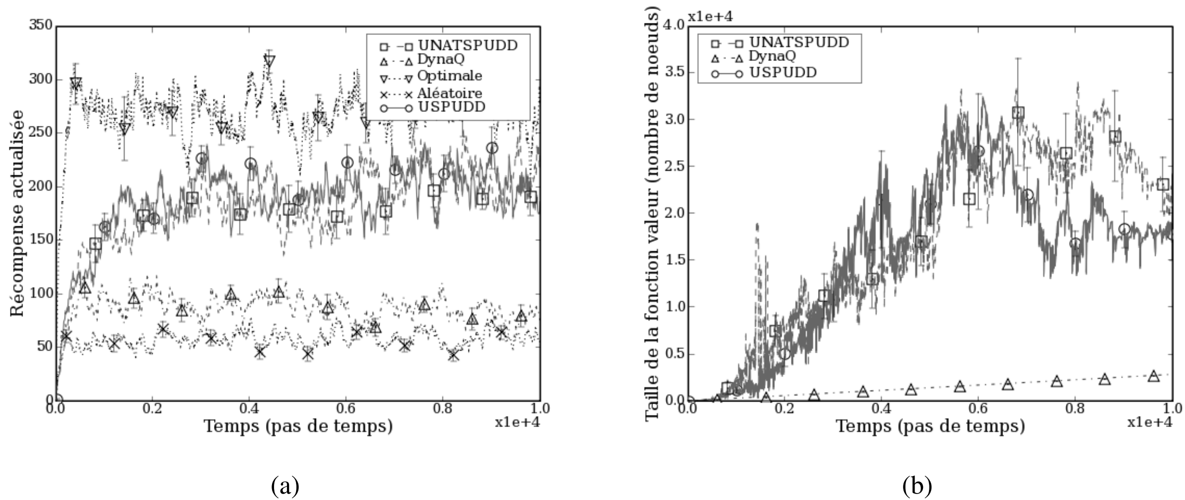


FIG. 5.14 – Problème *Factory4* : récompense actualisée obtenue par les agents (figure a) et taille de la fonction valeur calculée par l'algorithme de planification de l'agent (figure b). Alors que l'algorithme DYNA-Q exécute une politique légèrement meilleure qu'une politique aléatoire, les agents SDYNA améliorent nettement leur politique par rapport à celle de DYNA-Q ou d'une politique aléatoire, au prix d'une représentation de la fonction de valeur plus importante en taille, mais se stabilisant avec le temps.

La figure 5.14 montre la récompense actualisée obtenue par chacun des agents (figure 5.14(a)) et la taille de leur fonction de valeur (figure 5.14(b)). De même que pour le problème précédent, on peut observer que les trois agents USVI, UNATSVI et DYNA-Q arrivent rapidement à exécuter une politique meilleure qu'une politique aléatoire. On remarque cependant une nette différence en comparant les récompenses obtenues par la politique de l'agent DYNA-Q et celle des agents USVI et UNATSVI : alors que les agents SDYNA arrivent après 5000 pas de temps à obtenir des récompenses se rapprochant de la politique optimale, la progression de l'agent DYNA-Q s'arrête rapidement (avant 2000 pas de temps) pour se stabiliser sur une politique plus éloignée de la politique optimale. Néanmoins, une différence importante de la taille des représentations utilisées par les algorithmes est à noter : alors que la représentation tabulaire de la fonction de valeur utilisée par DYNA-Q contient moins de 6000 nœuds à la fin de l'expérience, les ADDs utilisés par l'algorithme IncSPUDD pour représenter la fonction de valeur peut compter plus de 20 000 nœuds.

De même que pour les résultats concernant le problème *Factory*, l'utilisation de la version incrémentale de SPUDD dans SDYNA montre la capacité de généralisation de l'apprentissage. Comme nous l'avions déjà souligné lors du chapitre concernant l'apprentissage hors-ligne pour ce même

problème (section 4.3.3, page 121), l'apprentissage est aussi rapide que dans le problème précédent alors que le problème *Factory4* est plus grand ($4,0 \cdot 10^9$ couples état/action contre $1,8 \cdot 10^6$), soulignant ainsi le fait que la complexité de l'apprentissage dépend plus de la structure du problème que de sa taille.

5.4.4 Le problème *Ring*

Enfin, le dernier problème que nous traitons est le problème *Ring* défini lors du chapitre précédent, section 4.3.3 (page 122). Le problème est constitué de 40 variables binaires et 41 actions ($4,5 \cdot 10^{13}$ couples état/action). Nous utilisons les mêmes fonctions de base que le chapitre précédent, définies dans la figure 4.30 (page 124). Les agents SDYNA, ULP et UNATLP, sont testés lors de l'expérimentation. Ils utilisent respectivement l'algorithme UpdateFMDP et UpdateFMDPnAT pour mettre à jour le modèle. Les deux utilisent l'algorithme IncLP pour la planification, avec comme paramètres $T_M = 100$, $T_P = 1500$ et $T_{MIN} = 50$.

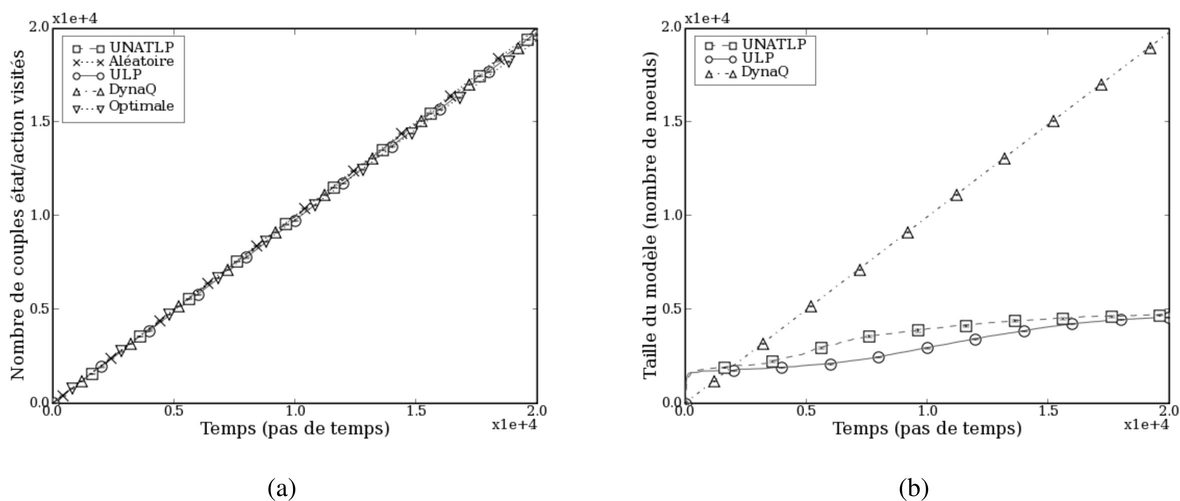
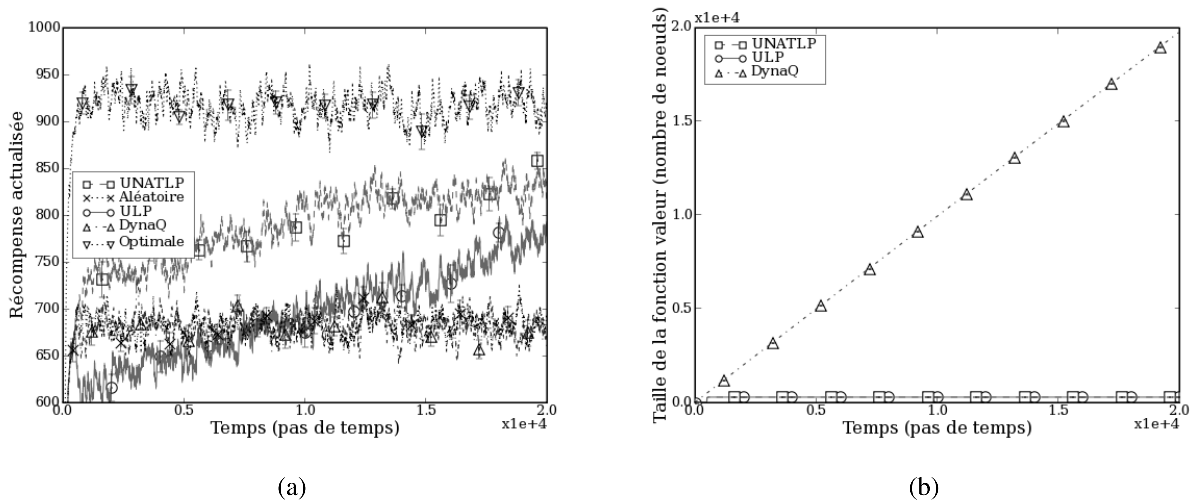


FIG. 5.15 – Problème *Ring* : nombre de couples état/action visités par les agents (figure a) et taille de la fonction de transition construite par apprentissage (figure b). Alors que la fonction de transition construite par DYNA-Q croît au fur et à mesure des couples état/action visités dans le problème, la taille de la fonction de transition du FMDP construit par les algorithmes UpdateFMDP et UpdateFMDPnAT se stabilise alors que de nouveaux couples état/action sont essayés.

La figure 5.15 représente le nombre de couples état/action visités par les agents (figure 5.15(a)) et la taille du modèle construit par les algorithmes d'apprentissage (figure 5.15(b)). D'une part, on peut remarquer que l'ensemble des agents DYNA-Q, ULP, UNATLP, l'agent exécutant une politique optimale et l'agent exécutant une politique purement aléatoire explorent tous un nombre de couples état/action équivalent au nombre de pas de temps dans l'environnement (environ 20 000 couples).

D'autre part, on retrouve un résultat similaire aux résultats précédents concernant la taille de la représentation de la fonction de transition construite par DYNA-Q, comparé à l'approche SDYNA : alors que DYNA-Q construit une représentation tabulaire d'environ 20 000 nœuds, les agents ULP et UNATLP construisent des représentations beaucoup plus compactes d'environs 5000 nœuds.



(a) (b)

FIG. 5.16 – Problème *Ring* : récompense actualisée obtenue par les agents (figure a) et taille de la fonction valeur calculée par l'algorithme de planification de l'agent (figure b). Alors que l'algorithme DYNA-Q construit une fonction de valeur de la même taille que le nombre d'états visités et exécute une politique qui n'est pas meilleure qu'une politique aléatoire, les agents SDYNA utilisent une représentation très compacte de la fonction de valeur (approchée) et améliorent nettement leur politique par rapport à celle de DYNA-Q ou d'une politique aléatoire.

La figure 5.16 montre la récompense actualisée obtenue par chacun des agents (figure 5.16(a)) et la taille de leur fonction de valeur (figure 5.16(b)). Contrairement aux problèmes précédents, on peut observer que l'agent DYNA-Q n'exécute pas une politique meilleure que la politique aléatoire, même après 20 000 pas de temps. Au contraire, les deux agents, ULP et UNATLP, de l'approche SDYNA arrivent rapidement à exécuter une politique meilleure qu'une politique aléatoire sans toutefois atteindre les performances d'une politique optimale. Concernant la taille de la représentation des fonctions de valeurs, la représentation tabulaire utilisée par l'algorithme DYNA-Q augmente au fur et à mesure que l'algorithme découvre de nouveaux états (plus de 19 000 nœuds), contrairement à la représentation utilisée par l'algorithme IncLP qui se restreint à la taille des fonctions de base prédéfinies (voir figure 4.30, page 124).

Les résultats concernant ce problème montrent clairement que l'architecture SDYNA est capable d'apprendre des représentations permettant d'exploiter la décomposition additive de la fonction de

récompense d'un problème lors de la phase de planification. En effet, la fonction de valeur du problème *Ring* a une représentation exacte augmentant de façon exponentielle avec le nombre de variables du problème. La représentation approchée de la fonction de valeur utilisée par la programmation linéaire (supposant une connaissance a priori des fonctions de base) permet donc de représenter de façon approchée cette fonction de valeur en utilisant une représentation très compacte. Ces résultats montrent qu'une telle représentation s'utilise très bien avec l'apprentissage effectué par SDYNA. En effet, de même que pour les problèmes précédents, la généralisation de l'induction d'arbres de décision est exploitée par la planification permettant ainsi d'améliorer nettement la politique d'un agent après seulement quelques centaines de pas de temps alors que le nombre d'états possibles dans le problème est de l'ordre du milliard.

Il est important de remarquer que pour tous les algorithmes, y compris l'agent exécutant la politique optimale approchée ou celui exécutant une politique aléatoire, un nombre similaire de couples état/action est visité et correspond au nombre de pas de temps de l'agent dans l'environnement. En effet, le problème est suffisamment stochastique et grand pour que, quelque soit la politique utilisée, un agent n'exécute pas deux fois la même action dans le même état. Un agent tel que DYNA-Q n'a donc pas eu d'autre choix que d'explorer au cours de toute la durée de l'expérience.

Enfin, nous pouvons noter une nette différence concernant la récompense actualisée obtenue par les deux agents SDYNA utilisant chacun une représentation de la fonction de transition différente. Nous rappelons que la fonction de transition de l'agent ULP est composée d'un arbre par action et par variable. Par conséquent, les observations effectuées par l'agent sont partitionnées sur chacune des actions, soit 41 pour ce problème. Cette partition n'est pas nécessaire pour l'agent UNATLP qui construit une fonction de transition composée d'un arbre par variable qui, quelque soit l'action exécutée par l'agent et contrairement à l'agent ULP, est mis à jour à chaque pas de temps.

5.5 Synthèse

Dans ce chapitre, nous avons présenté un cadre général, SDYNA, reprenant le concept de l'architecture DYNA, en l'adaptant aux FMDPs. À l'instar de DYNA, SDYNA intègre une phase d'apprentissage des fonctions représentant le problème d'apprentissage par renforcement et une phase de planification, calculant une solution à ce problème.

Dans le cadre de l'apprentissage, nous avons présenté deux nouveaux algorithmes d'apprentissage incrémental d'un FMDP, nommément UpdateFMDP et UpdateFMDPnAT. Ces algorithmes utilisent la même décomposition des observations de l'agent que les algorithmes d'apprentissage BuildFMDP et BuildFMDPnAT présentés lors du chapitre précédent. Cependant, ils sont adaptés au cadre de l'apprentissage incrémental de SDYNA. Ils reposent sur des travaux antérieurs concernant l'apprentissage incrémental d'arbres de décision et permettent donc de mettre à jour en ligne un FMDP. Dans le cadre de la planification, nous avons adapté les algorithmes SVI, SPUDD et l'ap-

proche par la programmation linéaire, les intégrant ainsi dans SDYNA. Enfin, nous avons validé de façon expérimentale ces algorithmes sur les problèmes classiques de la littératures concernant la planification dans les FMDPs et pour lesquels nous supposions que la structure était inconnue.

Nous avons montré que pour ces problèmes, les agents basés sur le cadre SDYNA montraient d'importante capacité de généralisation, permettant à l'agent d'améliorer rapidement son comportement, même lorsque le problème était de grande taille. Ainsi, ces agents utilisent pleinement les capacités de généralisation des algorithmes d'induction d'arbres de décision et les capacités d'agrégation des algorithmes de planification pour s'adapter rapidement au problème. Un tel apprentissage dépend alors plus de la structure du problème que de la taille de celui-ci.

Il est important de noter que notre adaptation de la programmation linéaire dans le cadre incrémental n'est pas véritablement satisfaisante puisqu'elle réutilise très peu d'informations calculées lors de la phase de planification précédente. Cependant, les résultats montrent que cette approche peut être très prometteuse concernant son utilisation dans les grands problèmes.

D'une façon générale, peu d'algorithmes d'apprentissage par renforcement ont été proposés pour résoudre de grands problèmes discrets. Nous pouvons notamment citer l'algorithme UTREE proposé par McCallum (1995, 1996). Cet algorithme est intéressant puisqu'il construit une représentation structurée des fonctions de valeur d'action, sous la forme d'un arbre de décision, et stockant aux feuilles, les observations de l'agent correspondant au contexte défini par les parents de la feuille. Ces exemples constituent un modèle utilisé par l'algorithme pour mettre à jour l'arbre avec l'algorithme *Value Iteration*. L'algorithme installe un nouveau nœud dans l'arbre lorsqu'un test statistique détermine que les distributions de valeur d'action sont différentes à une feuille. Dans ce cas, un test sur une variable de l'espace d'état pour un état précédent de l'agent appartenant à une suite d'observations est sélectionné, permettant ainsi de résoudre certain problème où l'hypothèse de Markov n'est pas satisfaite. Enfin, il est important de noter qu'une telle représentation permet une certaine généralisation, notamment pour les tous les états agrégés dans une même feuille de l'arbre des fonctions de valeur d'action. L'algorithme UTREE est particulièrement bien adapté aux problèmes ayant un grand nombre de variable d'état pouvant être modélisé par un MDP possédant un nombre plus limité d'état.

Si l'on compare cette approche à celle utilisée par SDYNA, on note que l'algorithme UTREE n'utilise pas de représentation factorisée des fonctions de transition et de récompense. C'est pourquoi, d'une part, UTREE n'est pas adapté pour l'apprentissage de MDPs de grande taille, d'autre part la généralisation est limitée à la partition construite à un pas de temps donné par l'algorithme : UTREE a besoin au minimum d'un exemple par feuille pour que la représentation de la fonction de valeur soit complète, ce qui n'est pas le cas de l'approche SDYNA. Cependant, plusieurs idées utilisées par l'algorithme UTREE peuvent être reprise et adaptées à SDYNA, comme par exemple pour la résolution des problèmes où l'hypothèse de Markov n'est pas satisfaite.

Chapitre 6

Le compromis exploration/exploitation dans SDYNA

Ces dix dernières années ont été fructueuses concernant la recherche sur le compromis exploration/exploitation en apprentissage par renforcement dans les MDPs finis. Le compromis exploration/exploitation fait référence au dilemme pour un agent entre, d'un côté, utiliser la connaissance acquise afin d'obtenir une récompense espérée connue et, de l'autre côté, explorer l'environnement afin de découvrir de nouvelles transitions mais sans attendre de récompense à court terme (Thrun, 1992; Dayan and Sejnowski, 1996; Singh et al., 2000).

Dans le chapitre précédent, afin de gérer ce compromis, nous avons utilisé l'algorithme ϵ-greedy comme méthode de sélection de l'action aussi bien avec DYNA qu'avec SDYNA. Nous rappelons que l'algorithme ϵ-greedy consiste à choisir la meilleure action connue la plupart du temps, avec une probabilité faible ϵ de choisir une action de façon aléatoire. Cette méthode présente l'avantage d'être à la fois simple et intuitive.

Cependant, elle présente l'inconvénient d'être une méthode d'exploration *non-dirigée* (Thrun, 1992), c'est-à-dire que l'exploration est purement aléatoire, sans prendre en compte les fonctions de valeur d'action ou bien la connaissance actuelle concernant les fonctions de transition et de récompense. Ainsi, pour les problèmes tels que *Linear* et *Expon*, décrits section 4.3.2 (page 108), la récompense n'est accessible que pour un seul état sur l'ensemble des états possibles et cet état n'est accessible que par une seule transition possible. Par conséquent, nous avons pu constater figure 4.21 (section 4.3.2, page 112), en utilisant l'erreur relative de la politique, qu'une exploration purement aléatoire peut être rapidement déficiente à trouver la récompense dans le problème. Par exemple, dans le problème *Expon*, la récompense n'est pas découverte après 20 000 pas de temps avec une politique purement aléatoire alors que le problème n'est composé que de 4 variables binaires et 4 actions (soit seulement 64 couples état/action).

Afin de résoudre un tel problème, plusieurs algorithmes d'exploration *dirigée* (Thrun, 1992) ont

été proposés ces dernières années. Ces algorithmes d'exploration dirigée utilisent une heuristique déterminant les couples état/action à explorer en priorité. De plus, certain d'entre eux ont été analysés formellement et offrent des garanties théoriques quant au temps d'exploration de l'algorithme nécessaire à l'apprentissage.

Dans ce chapitre, nous ne chercherons pas à proposer une solution originale au problème de l'exploration dans un problème d'apprentissage par renforcement, mais plutôt de formuler des questions concernant l'intégration des méthodes d'exploration existantes dans la littérature dans le cadre de SDYNA. Celui-ci commence donc, section 6.1, par décrire la formalisation d'un algorithme d'apprentissage par renforcement qualifiée "d'efficace". Dans un deuxième temps, un aperçu des principaux algorithmes d'apprentissage par renforcement gérant le compromis exploration/exploitation, aussi bien dans le cadre factorisé que non factorisé sera présenté dans cette même section. La section 6.2 présente un point de départ pour l'intégration de cette famille d'algorithmes dans l'architecture SDYNA. La section 6.3 présente les résultats d'une telle intégration, notamment sur les problèmes *Linear* et *Expon* (décrits précédemment dans la section 4.3.2, page 108).

6.1 Algorithme d'apprentissage basé sur un modèle et avec exploration dirigée

Plusieurs algorithmes d'apprentissage par renforcement utilisant un modèle pour diriger l'exploration ont été proposés, à commencer par DYNA-Q+ (Sutton and Barto, 1998). L'algorithme DYNA-Q+ reprend l'algorithme DYNA-Q auquel est ajouté un bonus aux couples état/action à explorer. Sans être exhaustive, cette section décrit plusieurs algorithmes d'apprentissage par renforcement, certain reprenant d'une façon ou d'une autre l'idée de DYNA-Q+, avec une stratégie d'exploration dirigée et basée sur la construction d'un modèle des fonctions de transition et de récompense. De plus, certains de ces algorithmes ont été formalisés et des analyses théoriques concernant leur vitesse d'apprentissage ont été proposées.

Ainsi, sans rentrer dans les détails des analyses formelles[1], cette section commence par décrire la notion d'algorithme "efficace" (section 6.1.1). Puis elle décrit plusieurs algorithmes d'apprentissage par renforcement basés sur un modèle, notamment l'algorithme *Explicit, Exploit and Explore* (E^3) proposé par Kearns and Singh (1998), l'algorithme R-MAX proposé par Brafman and Tennenholtz (2003) et l'algorithme *Model Based Interval Estimation* (MBIE) proposé par Strehl and Littman (2005), respectivement dans les sections 6.1.2, 6.1.3 et 6.1.4. Enfin, nous décrirons les méthodes qui ont été adaptées dans le cadre des FMDPs section 6.1.5.

[1]Le lecteur intéressé par le détail de ces analyses pourra consulter les articles cités.

6.1.1 Définition de l'apprentissage "efficace"

Afin de caractériser l'efficacité d'un algorithme d'apprentissage par renforcement, Strehl (2007) propose de formaliser la notion d'apprentissage "efficace". Dans ce but, il ajoute deux nouveaux paramètres à l'algorithme d'apprentissage par renforcement. Le premier paramètre, ϵ, définit la performance désirée de l'algorithme, c'est-à-dire la distance à laquelle la fonction de valeur de la politique optimale apprise par l'algorithme doit être de la fonction de valeur optimale dans le problème. Le deuxième paramètre, δ, définit une mesure de confiance de l'apprentissage, c'est-à-dire la certitude que l'on cherche à obtenir concernant la performance de l'algorithme d'apprentissage. Ces deux paramètres se retrouvent dans l'ensemble des algorithmes que nous présentons dans cette section. Une valeur plus faible de ces paramètres nécessitera plus d'exploration de la part de l'algorithme d'apprentissage puisque les distances requises sont plus contraignantes. D'après Kakade (2003), on définit maintenant la complexité de l'échantillon d'exploration (*sample complexity of exploration*).

Définition 13 (Complexité de l'échantillon d'exploration) *Pour tout* $\epsilon > 0$, la complexité de l'échantillon d'exploration *d'un algorithme* $\mathcal{A}$ *est le nombre de pas de temps* t *tel que la politique* $\mathcal{A}_t$ *de l'algorithme à un instant* t *n'est pas* ϵ*-optimale pour l'état courant* s_t *à l'instant* t*, c'est-à-dire que* $V^{\mathcal{A}_t}(s_t) < V^*(s_t) - \epsilon$.

À partir de cette définition, il est possible de définir la notion d'algorithme d'apprentissage par renforcement "efficace".

Définition 14 (Algorithme PAC-MDP) *Un algorithme* $\mathcal{A}$ *est défini comme étant un algorithme* PAC-MDP *efficace (*PAC-MDP *signifiant* Probably Approximately Correct in Markov Decision Processes*) si, pour tout* $\epsilon > 0$ *et* $0 \leq \delta < 1$*, la complexité de l'échantillon d'exploration de l'algorithme* $\mathcal{A}$ *peut s'exprimer sous la forme d'un polynôme en fonction du nombre d'états* $|S|$*, du nombre d'actions* A*, de* $1/\epsilon$*, de* $1/\delta$ *et de* $1/(1-\gamma)$*, avec une probabilité supérieure à* $1-\delta$*.*

Dans la suite du manuscrit, nous dirons qu'un algorithme est PAC lorsque celui-ci est PAC-MDP efficace. Enfin, notons que la terminologie PAC (pour *Probably Approximately Correct*) est empruntée à l'apprentissage supervisé (Valiant, 1984) et traduit par "Probablement Approximativement Correct" en français (Cornuéjols and Miclet, 2002).

6.1.2 L'algorithme Explicit Explore or Exploit

L'algorithme *Explicit Explore or Exploit* (E^3) proposé par Kearns and Singh (1998) est un algorithme d'apprentissage par renforcement PAC reposant sur une méthode de gestion du compromis exploration/exploitation. Comme la plupart des autres algorithmes PAC, E^3 construit de façon interne les fonctions de transition et de récompense pour les utiliser ensuite afin de calculer une politique optimale.

Paramètre(s) : ϵ, m_{known}, $V^*(s)$

Initialisation : $S \leftarrow \emptyset$ (S représentant l'ensemble des états connus)

À chaque pas de temps : pour un état s :

1. **Si** l'état courant s de l'agent n'est pas dans S ($s \notin S$), **alors** : choisir l'action la moins explorée
2. **Si** l'état s a été exploré m_{known} fois, **alors** : $S \leftarrow S \cup \{s\}$ (il devient connu)
3. **Si** l'état courant s de l'agent est dans S, **alors** :
   (a) calculer deux politiques π_r et π_e :
   π_r : exploite le modèle des transitions et cherche à maximiser les récompenses cumulées par l'agent pendant T pas de temps
   π_e : cherche à maximiser la probabilité d'arriver dans un état n'appartenant pas à l'ensemble des états connus S
   (b) **Si** $V^{\pi_r}(s) > V^*(s) - \epsilon/2$:
   Alors : l'agent exécute la politique π_r
   Sinon : l'agent exécute la politique π_e

FIG. 6.1 – L'algorithme E^3

La figure 6.1 décrit l'algorithme E^3. L'ensemble des états possibles du MDP est partitionné en deux ensembles possibles : les états connus (représentés par l'ensemble S) et les états inconnus. De plus, l'algorithme E^3 construit un modèle des transitions permettant de calculer une politique optimale et sa fonction de valeur associée. Or, comme l'algorithme fait l'hypothèse que la fonction de valeur optimale $V^*(s)$ est connue, l'heuristique utilisée peut se résumer par : si le modèle des transitions est suffisamment connu pour que la fonction de valeur de la politique optimale calculée à partir de ce modèle soit proche de la fonction de valeur optimale (à ϵ près), alors exécuter cette politique (étape 3b, politique π_r), sinon continuer d'explorer l'environnement (étape 3b, politique π_e). Tant que l'état courant est considéré comme inconnu, c'est-à-dire qu'il a été visité moins de m_{known} fois et donc qu'il n'appartient pas à S, l'algorithme choisit l'action qui a été la moins testée.

Ainsi, le nom de l'algorithme vient du fait que le compromis exploration/exploitation est géré de façon explicite dans cet algorithme. Celui-ci possède deux inconvénients majeurs : il suppose la connaissance a priori de la fonction de valeur optimale ; de plus, il nécessite la résolution de deux MDPs (le MDP maximisant la récompense de l'agent et le MDP utilisé pour l'exploration) à chaque pas de temps.

6.1.3 L'algorithme R-MAX

L'algorithme R-MAX a été proposé par Brafman and Tennenholtz (2003). Il formalise une technique très largement utilisée dans le domaine de l'apprentissage par renforcement consistant à avoir un a priori optimiste sur les couples états/actions inconnus du problème. Cette technique était déjà connue auparavant et avait été décrite dans de nombreux travaux (Sutton, 1990; Kaelbling, 1993; Sutton and Barto, 1998; Kaelbling et al., 1996). Cependant, Brafman and Tennenholtz (2003) démontrent que l'algorithme R-MAX est PAC et rendent ainsi légitime l'initialisation optimiste des couples état/action inconnus. La figure 6.2 décrit l'algorithme R-MAX dans le cadre des MDPs (l'algorithme est décrit par Brafman and Tennenholtz (2003) dans le cadre plus général des jeux stochastiques, ou *Stochatic Games* (Shapley, 1953)).

Paramètre(s) : R_{max}

Initialisation : initialiser le MDP $\hat{\mathcal{M}}$ avec :
- $S_{\hat{\mathcal{M}}}$ l'ensemble des états tel que : $S_{\hat{\mathcal{M}}} \leftarrow S \cup \{s_{R_{max}}\}$ avec $s_{R_{max}}$ un état fictif
- $A_{\hat{\mathcal{M}}}$ l'ensemble des actions tel que : $A_{\hat{\mathcal{M}}} \leftarrow A$
- $R_{\hat{\mathcal{M}}}$ la fonction de récompense telle que : $\forall s \in S_{\hat{\mathcal{M}}}, \forall a \in A : R_{\hat{\mathcal{M}}}(s,a) \leftarrow R_{max}$
- $T_{\hat{\mathcal{M}}}$ la fonction de transition telle que : $\forall s \in S_{\hat{\mathcal{M}}}, \forall a \in A : P(s_{R_{max}}|s,a) = 1$

À chaque pas de temps : pour un état s :

1. Calculer une politique optimale $\pi^*_{\hat{\mathcal{M}}}$ dans le MDP $\hat{\mathcal{M}}$
2. Exécuter l'action a choisie par la politique $\pi^*_{\hat{\mathcal{M}}}$, observer le nouvel état s'
3. **Si** le nouvel état courant s' est marqué comme inconnu, **alors** :
   (a) mettre à jour les informations de récompense concernant $\langle s, a, r\rangle$
   (b) mettre à jour les informations de transition concernant $\langle s, a, s'\rangle$
   (c) **Si** le couple état/action s, a est considéré comme connu, **alors** :
      i. ajouter les informations de récompense au MDP $\hat{\mathcal{M}}$
      ii. ajouter les informations de transition au MDP $\hat{\mathcal{M}}$
      iii. marquer le couple s, a comme connu.

FIG. 6.2 – L'algorithme R-MAX dans le cadre des MDPs

A l'instar de l'algorithme E^3, R-MAX se base sur la construction d'une politique optimale à partir d'une fonction de transition incomplète. Cependant, contrairement à E^3, il n'y a pas de différence explicite entre le fait d'explorer et d'exploiter. L'algorithme R-MAX est décrit figure 6.2.

En premier lieu, R-MAX est initialisé en construisant un MDP $\hat{\mathcal{M}}$ constitué de l'ensemble des

actions possibles et de l'ensemble des états possibles avec en plus un nouvel état fictif $s_{R_{max}}$. De plus, la fonction de récompense $R_{\hat{\mathcal{M}}}$ est initialisée en associant la récompense maximale à tous les couples état/action (y compris l'état fictif $s_{R_{max}}$). La fonction de transition $T_{\hat{\mathcal{M}}}$ est initialisée en définissant pour l'ensemble des couples état/action du problème une transition déterministe vers l'état fictif.

Ensuite, dans un état s, l'agent exécute une action a définie par une politique optimale $\pi^*_{\hat{\mathcal{M}}}$ dans le MDP $\hat{\mathcal{M}}$ (étape 1). Après avoir maintenu les informations concernant l'observation $\langle s, a, s', r\rangle$ de l'agent, si le couple état/action (s, a) est considéré comme connu, alors seulement les informations sont ajoutées au MDP $\hat{\mathcal{M}}$ (étape 3c). Ainsi, contrairement à E^3, R-MAX utilise un seul MDP à partir duquel est calculée une politique permettant à l'agent à la fois d'explorer et d'exploiter.

R-MAX propose donc une méthode permettant de gérer de façon élégante le compromis exploration/exploitation. Bien qu'il soit plus simple, il souffre cependant du même inconvénient concernant sa complexité : il est nécessaire de calculer à chaque pas de temps une politique optimale. Afin de diminuer les calculs nécessaires à la résolution d'un MDP à chaque pas de temps, Strehl and Littman (2006b) proposent l'algorithme RDTP-RMAX, lequel s'appuie sur la programmation dynamique temps réel (*Real-time Dynamic Programming*) proposé par Barto et al. (1995) pour ne mettre à jour les informations ne concernant qu'un seul état et montrent que la propriété PAC de l'algorithme est conservée.

6.1.4 Les algorithmes MBIE et MBIE-EB

Les algorithmes MBIE et MBIE-EB (Strehl and Littman, 2005) sont une adaptation de l'algorithme *Interval Estimation* (IE) initialement proposé par Kaelbling (1993) pour le problème des bandits-manchots. Plusieurs généralisations de l'algorithme IE aux MDPs ont été proposées et ont montré des résultats expérimentaux intéressants (Kaelbling, 1993; Wiering and Schmidhuber, 1998; Strehl and Littman, 2004). L'algorithme *Model Based Interval Estimation* (MBIE), proposé par Strehl and Littman (2005), est aussi une généralisation de l'algorithme IE aux MDPs pour lequel il est montré que l'algorithme est PAC. Enfin, l'algorithme MBIE-EB (*Model Based Interval Estimation with Exploratory Bonus*) est une simplification de l'algorithme MBIE.

L'algorithme MBIE

L'algorithme MBIE fonctionne de la façon suivante : à partir des observations de l'agent dans l'environnement, l'algorithme maintient à jour le MDP $\hat{\mathcal{M}}$ composé de :

– S l'ensemble des états ;
– A l'ensemble des actions ;
– $R_{\hat{\mathcal{M}}}$ une estimation de la fonction de récompense calculée à partir de la moyenne des récom-

penses obtenues pour chaque couple état/action :

$$R_{\hat{\mathcal{M}}}(s,a) = \frac{\sum_{i=1}^{\underline{n}(s,a)} r_i}{\underline{n}(s,a)} \tag{6.1}$$

avec $\underline{n}(s,a)$ le nombre de fois que l'agent a exécuté l'action a dans l'état s et r_i la i^{ème} récompense dans la suite de récompense $[r_1, r_2, \ldots, r_{\underline{n}(s,a)}]$ obtenue par l'agent en effectuant l'action a dans l'état s ;

– $T_{\hat{\mathcal{M}}}$ une estimation de la fonction de transition définie à partir de la probabilité observée d'atteindre l'état s' en effectuant l'action a dans l'état s :

$$P_{\hat{\mathcal{M}}}(s'|s,a) = \frac{\underline{n}(s,a,s')}{\underline{n}(s,a)} \tag{6.2}$$

avec $\underline{n}(s,a)$ le nombre de fois que l'agent a exécuté l'action a dans l'état s et $\underline{n}(s,a,s')$ le nombre de fois que l'agent est arrivé dans l'état s' après avoir effectué l'action a dans l'état s.

Deux intervalles de confiance, nommément $CI(R_{\hat{\mathcal{M}}})$ et $CI(P_{\hat{\mathcal{M}}})$, sont définis respectivement pour les fonctions de récompense $R_{\hat{\mathcal{M}}}$ et de transition $T_{\hat{\mathcal{M}}}$. L'intervalle de confiance $CI(R_{\hat{\mathcal{M}}}(s,a))$ de la fonction de récompense $R_{\hat{\mathcal{M}}}$ pour un couple état/action (s,a) est défini tel que : $CI(R_{\hat{\mathcal{M}}}(s,a)) = [R_{\hat{\mathcal{M}}}(s,a) - \epsilon^R_{\underline{n}(s,a)}, R_{\hat{\mathcal{M}}}(s,a) + \epsilon^R_{\underline{n}(s,a)}]$ avec :

$$\epsilon^R_{\underline{n}(s,a)} = \sqrt{\frac{\ln(2/\delta_R)R_{max}}{2\underline{n}(s,a)}} \tag{6.3}$$

L'intervalle de confiance $CI(P_{\hat{\mathcal{M}}}(\cdot|s,a))$ de la fonction de transition pour un couple état/action a, s est défini tel que : $CI(P_{\hat{\mathcal{M}}}(\cdot|s,a)) = \{P(\cdot|s,a) \mid \parallel P_{\hat{\mathcal{M}}}(\cdot|s,a) - P(\cdot|s,a) \parallel_1 \leq \epsilon^P_{\underline{n}(s,a)}\}$ avec :

$$\epsilon^P_{\underline{n}(s,a)} = \sqrt{\frac{2(\ln(2^{|S|} - 2) - \ln(\delta_T))}{\underline{n}(s,a)}} \tag{6.4}$$

$P(\cdot|s,a)$ une distribution de probabilités appartenant à l'intervalle de confiance $CI(P_{\hat{\mathcal{M}}}(\cdot|s,a))$ et $\parallel x(\cdot) \parallel_1 = \sum_i |x(i)|$ la norme L_1. Enfin, notons que δ_R et δ_T sont des valeurs calculées en fonction du paramètre δ (cf. section 6.1.1) défini a priori.

À partir de ces intervalles de confiance, une nouvelle fonction de valeur d'action $\tilde{Q}(a,s)$ est définie en étant systématiquement optimiste, c'est-à-dire en utilisant uniquement les bornes supérieures des intervalles de confiance. Plus formellement, la fonction de valeur d'action $\tilde{Q}(a,s)$ est définie telle que :

$$\tilde{Q}(s,a) = \max_{R_{\hat{\mathcal{M}}}(s,a) \in CI(R_{\hat{\mathcal{M}}})} R_{\hat{\mathcal{M}}}(s,a) + \max_{P(s'|s,a) \in CI(P_{\hat{\mathcal{M}}}(\cdot|s,a))} \gamma \sum_{s'} P(s'|s,a) \max_{a'} \tilde{Q}(s',a') \tag{6.5}$$

Enfin, Strehl and Littman (2004, 2005) montrent comment effectuer ce calcul, notamment en réutilisant l'algorithme *Value Iteration*, pour déterminer l'ensemble des fonctions de valeur d'action utilisées lorsque l'agent prend une décision. Nous renvoyons le lecteur à ces articles pour connaître la méthode proposée par les auteurs.

L'algorithme MBIE-EB

L'algorithme MBIE-EB (Strehl and Littman, 2006a,b) est une variation de l'algorithme MBIE. À l'instar de MBIE, l'algorithme est PAC. Son principal avantage, comparé à MBIE, est qu'il est très simple à décrire et implémenter. Ainsi, l'algorithme maintient à jour un MDP $\hat{\mathcal{M}}$ composé du n-uplet $\langle S, A, R_{\hat{\mathcal{M}}}, T_{\hat{\mathcal{M}}} \rangle$ et pour lequel on calcule la fonction de valeur d'action $\tilde{Q}(a, s)$ telle que :

$$\tilde{Q}(s,a) = R_{\hat{\mathcal{M}}}(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} \tilde{Q}(s',a') + \frac{\beta}{\sqrt{\underline{n}(s,a)}} \quad (6.6)$$

avec β un paramètre de l'algorithme et $\underline{n}(s,a)$ le nombre de fois que l'action a a été exécutée dans l'état s. Ainsi, l'équation 6.6 peut être résolue avec n'importe quelle technique résolvant un MDP.

Enfin, tout comme les algorithmes R-MAX et E^3, les algorithmes MBIE et MBIE-EB doivent résoudre un MDP à chaque pas de temps. De la même façon qu'ils ont adapté l'algorithme R-MAX en s'appuyant sur la programmation dynamique temps réel, Strehl and Littman (2006b) proposent aussi l'algorithme RDTP-MBIE, utilisant l'équation 6.6 pour mettre à jour les informations ne concernant qu'un seul état à chaque pas de temps. À l'instar de RDTP-RMAX, les auteurs montrent que l'algorithme RDTP-MBIE conserve sa propriété PAC.

6.1.5 Apprentissage d'un FMDP

Plusieurs des algorithmes présentés précédemment ont été adaptés dans le cadre des FMDPs. Le premier d'entre eux à avoir été proposé est l'algorithme DBN-E^3 (Kearns and Koller, 1999), qui est l'adaptation de l'algorithme E^3 aux FMDPs.

L'algorithme DBN-E^3

L'algorithme DBN-E^3 suppose que la structure du problème est connue, plus précisément l'ensemble des DBNs pour chacune des variables et des actions. Ainsi, plutôt que d'avoir un temps d'apprentissage dans le pire cas dépendant du nombre de couples état/action dans le problème, comme c'est le cas pour l'algorithme E^3, Kearns and Koller (1999) montrent que le nombre d'observations nécessaires à l'apprentissage de l'algorithme DBN-E^3 peut s'exprimer sous la forme d'un polynôme ne dépendant que du nombre de paramètres quantifiant les DBNs. L'algorithme DBN-E^3 exploite donc les indépendances relatives aux fonctions spécifiées par les DBNs afin de diminuer la complexité de l'apprentissage. Aucune évaluation empirique n'a été effectuée, seule une analyse théorique de DBN-E^3 est proposée.

Les algorithmes f-RMAX et f-IE

Récemment, Strehl (2007) a proposé une adaptation des algorithmes R-MAX et MBIE pour les FMDPs, nommément *factored* R-MAX (f-RMAX) et *factored* IE (f-IE). Pour décrire et analyser ces

algorithmes, les indépendances relatives aux fonctions sont supposées a priori connues mais aussi, dans un cadre plus général, les indépendances relatives aux contextes. De plus, il suppose que la fonction de récompense est connue.

De la même façon que la notion d'algorithme d'apprentissage par renforcement "efficace" a été définie dans le cadre des MDPs, il est possible de définir cette notion dans les FMDPs :

Définition 15 (Algorithme PAC-FMDP) *Un algorithme $\mathcal{A}$ est défini comme étant* PAC-FMDP *(pour* Probably Approximately Correct in Factored Markov Decision Processes*) efficace si, pour tout* $\epsilon > 0$ *et* $0 \leq \delta < 1$*, la complexité de l'échantillon d'exploration de l'algorithme $\mathcal{A}$ peut s'exprimer sous la forme d'un polynôme en fonction du nombre maximum* $\max_i |\underline{Dom}(X_i)|$ *de valeurs possibles pour une variable X_i, du nombre d'actions A, de $1/\epsilon$, de $1/\delta$ et de $1/(1-\gamma)$, avec une probabilité supérieur à $1-\delta$.*

Ainsi, Strehl (2007) montre que les algorithmes f-RMAX et f-IE satisfont cette propriété et donc qu'ils sont PAC-FMDP.

L'algorithme f-RMAX est très similaire à l'algorithme R-MAX. Il peut être décrit formellement en définissant les fonctions de valeur d'action calculées à partir du FMDP maintenu par l'algorithme :

$$Q(s,a) = \begin{cases} R_{max}/(1-\gamma) \text{ si } \exists X_i \text{ tel que } n(D(s,a,X_i)) < m_i; \\ R(s,a) + \gamma \sum_{s'} \hat{T}(s'|s,a) \max_{a'} Q(s',a') \text{ sinon.} \end{cases} \tag{6.7}$$

avec R_{max} la récompense maximale dans le problème, $D(s,a,X_i)$ le contexte de la distribution de probabilités conditionnelle $P(X_i'|s,a)$ consistant avec le couple état/action (s,a), $n(D(s,a,X_i))$ le nombre d'exemples consistants avec le contexte $D(s,a,X_i)$ et m_i le nombre d'exemples requis pour qu'une transition soit définie comme étant connue.

Ainsi, tant que le nombre d'exemples requis m_i pour estimer la probabilité $\hat{P}(X_i'|s,a)$ à partir d'un couple état/action (s,a) n'a pas été atteint pour au moins une variable X_i', alors les probabilités estimées consistantes avec le couple (s,a) sont ignorées et la valeur maximum est assignée à la fonction de valeur d'action pour ce couple.

De même, l'algorithme f-IE est très similaire à l'algorithme MBIE-EB et peut être décrit formellement en définissant les fonctions de valeur d'action :

$$Q(s,a) = \begin{cases} R_{max}/(1-\gamma) \text{ si } \exists X_i \text{ tel que } c_i = 0; \\ R(s,a) + \gamma \sum_{s'} \hat{T}(s'|s,a) \max_{a'} Q(s',a') + eb(c_1,\ldots,c_n) \text{ sinon.} \end{cases} \tag{6.8}$$

avec $c_i = D(s,a,X_i)$ et $eb(c_1,\ldots,c_n) : \mathbb{Z}^n \rightarrow \mathbb{R}$ une fonction déterminant un bonus d'exploration et définie telle que :

$$eb(c_1,\ldots,c_n) = max_{(X_i,j)\in D(s,a)} \frac{\beta_i}{\sqrt{c_i}} \tag{6.9}$$

où $D(s,a)$ est l'ensemble des probabilités $\{P(X_i'|s,a)\}$ consistant avec le couple état/action (s,a) et un ensemble de constantes β_i, $i = 1,\ldots,n$ déterminées a priori. Ainsi, de la même façon que

pour l'algorithme MBIE-EB, un bonus d'exploration est ajouté pour les transitions dont l'estimation est calculée à partir de peu d'exemples. De même que l'algorithme DBN-E^3, aucune évaluation empirique n'est proposée par l'auteur.

Exploration dirigée et programmation linéaire

Bien que la plupart des travaux adaptant des techniques d'exploration dirigée dans le contexte des FMDPs proposent des analyses formelles afin de démontrer l'efficacité théorique de ces algorithmes, peu d'évaluations pratiques ont été proposées, principalement à cause de la difficulté de leur implémentation.

Ainsi, Guestrin et al. (2002a) proposent d'utiliser la programmation linéaire telle que nous l'avons décrit section 3.4 (page 61) afin de résoudre les FMDPs construits à partir des adaptations des algorithmes R-MAX et E^3. À l'instar des travaux précédents, les indépendances relatives aux fonctions sont supposées connues.

De plus, Guestrin et al. (2002a) proposent une méthode supplémentaire d'exploration dirigée très similaire à l'algorithme MBIE. En effet, pour les transitions estimées du FMDP, un intervalle de confiance est calculé. Plutôt que de donner un bonus aux transitions pour lesquelles il y a peu d'exemples estimant cette transition, un bonus d'exploration plus important est calculé pour les intervalles dont l'écart entre la borne inférieure et la borne supérieure est important.

Par conséquent, de la même façon que la famille d'algorithme MBIE, par opposition aux algorithmes E^3 et R-MAX, les observations de l'agent sont prises en compte au fur et à mesure qu'elles sont obtenues. Les résultats obtenus tendent à montrer qu'une telle approche permet d'obtenir de meilleurs résultats qu'avec l'implémentation proposée des algorithmes E^3 et R-MAX (Guestrin et al., 2002a).

6.2 Exploration dirigée dans l'architecture SDYNA

Les résultats concernant l'apprentissage hors-ligne, présentés section 4.3.2 (page 108), et utilisant les problèmes *Linear* et *Expon* montrent que, pour certains problèmes, même s'ils sont de petites tailles, un algorithme dirigeant l'exploration est nécessaire. Après avoir précisé, section 6.2.1, la problématique de l'exploration lors de l'apprentissage d'un FMDP dont la structure est inconnue, nous proposons l'intégration d'un algorithme d'exploration dirigée dans l'architecture SDYNA dans la section 6.2.2.

6.2.1 Problème de l'exploration lorsque la structure est inconnue

L'ensemble des méthodes d'exploration dirigée dans les FMDPs supposent que la structure du problème est connue a priori (Kearns and Koller, 1999; Guestrin et al., 2002a; Strehl, 2007). Or,

le but des travaux présentés dans cette thèse est d'apprendre la structure du problème au fur et à mesure de l'apprentissage. Nous supposons donc que cette connaissance n'est pas accessible a priori. Par conséquent, une nouvelle difficulté apparaît.

En effet, l'ensemble des algorithmes d'exploration dirigée dans les FMDPs utilisent tous une technique similaire : en fonction de la structure du problème connu a priori, un ensemble de distributions de probabilités conditionnelles composant la fonction de transition est défini comme étant inconnu. Ensuite, l'algorithme explore l'environnement jusqu'à ce que ces distributions de probabilités soient estimées connues, ou bien lorsque qu'une politique satisfaisante a été atteinte.

Lorsque la structure est inconnue a priori, le nombre de distributions de probabilités conditionnelles à estimer n'est pas connu a priori. Ainsi, bien qu'une distribution de probabilités conditionnelle puisse être estimée à partir d'un nombre d'exemples important (et donc pouvant être considérée comme connue), il est peut être nécessaire de continuer de l'explorer afin de découvrir un sous-espace dans lequel la distribution de probabilités serait significativement différente. Or, pour découvrir l'ensemble de la structure du problème, il est nécessaire que l'agent essaye l'ensemble des couples état/action du problème. Par conséquent, un algorithme d'exploration dirigée dans le cadre de SDYNA, c'est-à-dire lorsque la structure est inconnue, ne peut être que PAC-MDP et non PAC-FMDP.

Une telle complexité peut rapidement être problématique dans les grands problèmes. En effet, dans le pire cas, le temps d'apprentissage de l'agent pourra s'exprimer sous la forme d'un polynôme fonction du nombre de couples état/action dans le problème. Cependant, le nombre de couple état/action d'un problème croît de façon exponentielle avec le nombre de variables et d'actions décrivant ce problème.

Par exemple, dans un problème tel que *Expon*, la nature du problème fait qu'il est nécessaire que l'agent teste tous les couples état/action. Le temps d'apprentissage est donc au moins de 10 240 essais (en réalité beaucoup plus puisque, en fonction de la nature du problème, le temps d'apprentissage s'exprimera sous la forme d'un polynôme de 10 240). Cependant, dans le problème *Factory* (défini section 4.3.3, page 117), il n'est pas envisageable d'avoir un temps d'apprentissage nécessitant plus de 1 835 008 essais (le problème étant composé de 17 variables binaires et 14 actions) alors même qu'une stratégie d'exploration purement aléatoire en nécessite moins de 10 000 pour obtenir une politique proche de la politique optimale (cf. figure 4.26, page 120).

On recherche donc une méthode d'exploration dirigée permettant, non seulement de gérer le compromis exploration/exploitation, mais aussi de gérer, une fois qu'un espace a été identifié, jusqu'à quel point cet espace doit être fouillé afin de découvrir de nouveaux sous-espaces à explorer. Dans le cadre de l'architecture SDYNA, nous proposons dans la section suivante un point de départ pour la définition d'un tel algorithme.

6.2.2 Bonus d'exploration de paramètres et bonus d'exploration de structure

L'idée principale que nous présentons est l'utilisation de deux bonus complémentaires pour l'exploration. D'une part, le bonus d'exploration de paramètres utilise la structure du problème découverte et estime la part d'exploration nécessaire concernant les distributions de probabilités conditionnelles dans le FMDP construit par l'algorithme. D'autre part, le bonus d'exploration de structure ne tient pas compte de la structure connue du problème et ajoute un bonus aux couples état/action qui n'ont pas encore été essayés dans le problème.

La valeur de ces deux bonus peut être déterminée en fonction de méthodes déjà décrites dans la littérature. Plus particulièrement, la suite de cette section décrit, d'une part, l'utilisation de l'algorithme f-IE avec l'algorithme IncSVI (figure 5.6, page 138), afin de calculer le bonus d'exploration de paramètres et, d'autre part, l'utilisation de l'algorithme R-MAX pour le bonus d'exploration de la structure.

Bonus d'exploration de paramètres

L'algorithme f-IE s'adapte facilement aux algorithmes de planification dans les FMDPs. Pour une telle adaptation, il est nécessaire de modifier, non pas l'algorithme IncSVI, mais plutôt l'algorithme Regress (figure 3.9, page 51) utilisé par IncSVI pour calculer l'équation 2.4 de mise à jour des fonctions de valeur d'action. La figure 6.3 décrit l'algorithme RegressFIE permettant de calculer l'équation 6.8 (page 163) de mise à jour des fonctions de valeur d'action de l'algorithme f-IE.

Les deux équations de mise à jour étant semblables (seul un bonus d'exploration est ajouté pour l'algorithme f-IE), l'algorithme RegressFIE reprend les mêmes étapes que l'algorithme Regress, seule la dernière opération consacrée au calcul du bonus est ajoutée (étape 5).

Ainsi, pour chaque feuille l_b appartenant à la branche b de l'arbre $\mathsf{Tree}\left[Q_a^V\right]$, on calcule son bonus d'exploration associé en déterminant, pour chaque distribution de probabilités conditionnelle $\mathsf{Tree}\left[\hat{P}_a\right](X_i'|s)$, le nombre d'exemples dont le contexte soit consistant avec les variables testées dans la branche b. Ensuite, le bonus d'exploration maximum est ajouté à la valeur déjà contenue par la feuille l_b.

Bonus d'exploration de la structure

Le problème majeur concernant le bonus d'exploration de la structure est de représenter de façon structurée l'ensemble des couples état/action qui n'ont pas encore été visités afin de pouvoir leur assigner un bonus d'exploration, de la même façon que pour l'algorithme R-MAX. Afin de pouvoir intégrer une telle exploration au sein de l'architecture SDYNA, nous proposons de modifier la mesure de régression utilisée pour l'apprentissage de la fonction de récompense. Ainsi, plutôt que

Entrée(s) : $\mathsf{Tree}[P], \mathsf{Tree}[R], \mathsf{Tree}[V], a$ **Sortie(s) :** $\mathsf{Tree}\left[Q_a^V\right]$

1. $\mathsf{Tree}\left[P_a^V\right] \leftarrow$ PRegress($\mathsf{Tree}[V], a$)
2. Construire $\mathsf{Tree}\left[P_a^V V\right]$ de la façon suivante : pour chaque branche b parente de la feuille l_b et appartenant à l'arbre $\mathsf{Tree}\left[P_a^V\right]$, faire :
   (a) Soit P^b la distribution de probabilités jointe obtenue à partir du produit de chaque distribution de probabilités de chaque variable présente dans la feuille l_b
   (b) Calculer $v_b = \sum_{b' \in \mathsf{Tree}[V]} P^b(b')V(b')$ avec : b' les branches de l'arbre $\mathsf{Tree}[V]$, $P^b(b')$ la probabilité que l'instanciation des variables associées à la branche b' soit vraie étant donné P^b et $V(b')$ la valeur contenue par la feuille l'_b associée à la branche b' dans l'arbre $\mathsf{Tree}[V]$
   (c) Définir v_b comme étant le contenu la feuille l_b
3. $\mathsf{Tree}\left[P_a^V V\right] \leftarrow \gamma \cdot \mathsf{Tree}\left[P_a^V V\right]$ (en multipliant chaque feuille par γ)
4. $\mathsf{Tree}\left[Q_a^V\right] \leftarrow$ Append($\mathsf{Tree}[R], \mathsf{Tree}\left[P_a^V V\right]$) (en utilisant l'addition comme opérateur de combinaison)
5. Pour chaque branche b parente de la feuille l_b appartenant à $\mathsf{Tree}\left[Q_a^V\right]$:
   (a) Calculer $eb(b) = \max_{\forall X_i} \frac{\beta_i}{n_{X_i}(b)}$ avec $n_{X_i}(b)$ le nombre d'exemples dans la branche b de l'arbre $\mathsf{Tree}\left[\hat{P}_a\right](X'_i|s)$
   (b) Soit $v = v_b + eb(b)$ avec v_b le contenu de la feuille l_b
   (c) Définir v comme étant le contenu la feuille l_b
6. Retourner $\mathsf{Tree}\left[Q_a^V\right]$

FIG. 6.3 – L'algorithme RegressFIE($\mathsf{Tree}[V], a$).

d'utiliser le critère des moindres carrés (section 4.1.3, page 86), nous proposons un nouveau critère de façon à, d'une part, réutiliser l'algorithme incrémental de construction d'arbres de décision et, d'autre part, intégrer naturellement l'algorithme d'exploration avec SDYNA.

Pour décrire la façon dont se calcule ce nouveau critère d'exploration de la structure, noté $\mathcal{M}_{SE}$, nous commençons par définir la valeur $k_\mathcal{E}$ associée à un ensemble $\mathcal{E}$ d'exemples $\langle \mathfrak{a}, \varsigma \rangle$ avec $\varsigma \in \mathbb{R}$ et associée à un nœud k dans l'arbre :

$$k_\mathcal{E} = \frac{\sum_{\varsigma \in \mathcal{E}} \varsigma + \beta_s \left|\complement_{\mathcal{E}_k}\mathcal{E}\right| w_s}{n_\mathcal{E} + \left|\complement_{\mathcal{E}_k}\mathcal{E}\right| w_s} \tag{6.10}$$

avec $n_\mathcal{E}$ le nombre d'exemples dans $\mathcal{E}$, $\mathcal{E}_k$ l'ensemble des exemples consistant avec le contexte associé au nœud k et $\left|\complement_{\mathcal{E}_k}\mathcal{E}\right|$ le cardinal de l'ensemble $\complement_{\mathcal{E}_k}\mathcal{E}$ représentant l'ensemble complémentaire de l'ensemble des exemples visités $\mathcal{E}$ dans l'ensemble des exemples possibles $\mathcal{E}_k$ (l'ensemble $\complement_{\mathcal{E}_k}\mathcal{E}$ représente donc l'ensemble des exemples possibles non visités). Enfin, β_s et w_s sont deux paramètres de l'exploration représentant respectivement le bonus ($\beta_s > 0$) et la pondération ($w_s \geq 0$)

associés aux exemples non visités. On définit maintenant l'erreur de régression, notée $E_{\mathcal{E}}$, associée à l'ensemble d'exemples $\mathcal{E}$ et au nœud k :

$$E_{\mathcal{E}} = \frac{\sum_{\varsigma \in \mathcal{E}} (\varsigma - k_{\mathcal{E}})^2 + (\left|\complement_{\mathcal{E}_k}\mathcal{E}\right| - k_{\mathcal{E}})^2 w_s}{n_{\mathcal{E}} + \left|\complement_{\mathcal{E}_k}\mathcal{E}\right| w_s} \tag{6.11}$$

Les autres équations de la mesure peuvent être utilisées telles quelles, de même les algorithmes d'apprentissage de la récompense et des FMDPs que nous avons décrits précédemment.

6.3 Résultats

Les résultats que nous présentons dans cette section porte sur l'algorithme d'exploration que nous venons de proposer appliqué à deux types de problèmes requérant des types d'exploration différents :

Linear* et *Expon (défini section 4.3.2, page 108) : la récompense pour ces deux problèmes ne peut être obtenue que dans un seul état du problème (lorsque toutes les variables sont égales à Vrai) accessible à partir d'une seule transition. Par conséquent, une exploration systématique de tous les couples état/action est nécessaire.

Factory (défini section 4.3.3, page 117) : plusieurs récompenses pour ce problème sont accessibles pour de nombreux états du problème et accessibles après de nombreux couples état/action. Cependant, le problème est de grande taille. Par conséquent, une exploration systématique est contre-productive et difficile.

Pour traiter ces problèmes, nous utiliserons le même agent SDYNA d'exploration dirigée composé de :

Apprentissage (fonction UpdateModel) : l'algorithme de mise à jour du modèle UpdateFMDPnAT et utilisant la mesure de régression pour l'exploration $\mathcal{M}_{SE}$ (cf. section 6.2.2, équations 6.10 et 6.11) pour l'apprentissage de la fonction de récompense ;

Planification (fonction IncPlan) : l'algorithme IncSVI utilisant l'opérateur RegressFIE (cf. section 6.2.2, figure 6.3) ;

Décision (fonction Acting) : un algorithme glouton (sélectionnant la meilleure action à chaque pas de temps).

Dans chaque problème, nous testerons deux agents dont les paramètres testeront deux compromis différent entre l'exploration de la structure et l'exploration des paramètres. Le premier agent, nommé UNATSVIES, donnera plus d'importance à l'exploration de la structure. Le deuxième agent, nommé UNATSVIEP, donnera plus d'importance à l'exploration des paramètres. Nous préciserons la valeur exacte des paramètres pour chacun des problèmes dans les sections respectives.

Les agents d'exploration dirigée sont comparés à un agent SDYNA, nommément UNATSVI, utilisant l'algorithme de mise à jour du modèle UpdateFMDPnAT (avec une mesure de régression des moindres carrés), l'algorithme de planification IncSVI et l'algorithme ϵ-greedy à la fois pour la décision et en tant que méthode d'exploration non dirigée. Les sections 6.3.1 et 6.3.2 comparent donc les résultats des agents à exploration dirigée et UNATSVI sur les problèmes *Expon* et *Linear* d'une part, et *Factory* d'autre part.

6.3.1 Les problèmes *Linear* et *Expon*

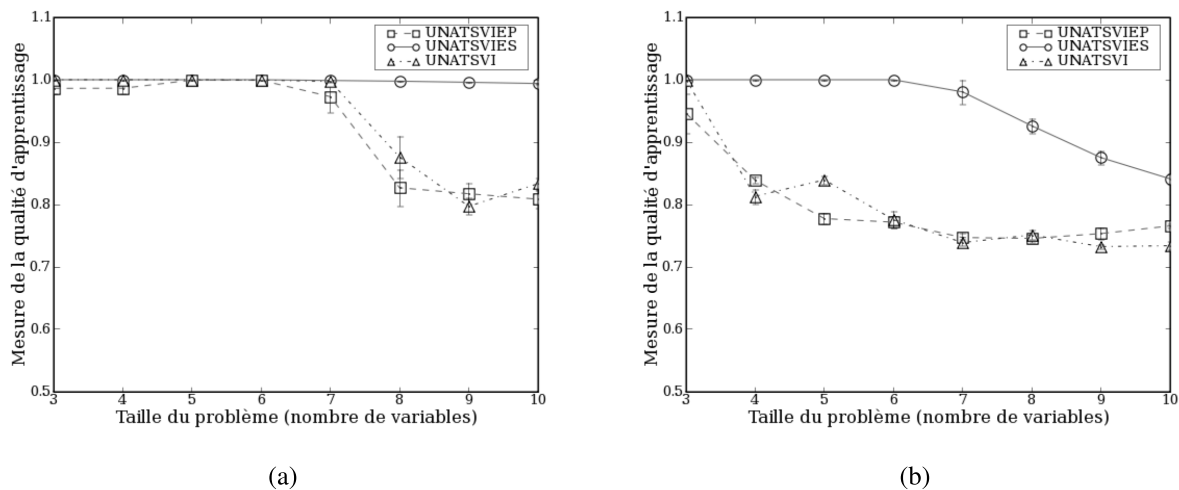
Nous avons vu, lors de la section 4.3.2 (page 108) concernant l'apprentissage hors-ligne, qu'un agent effectuant une exploration aléatoire (non dirigée) obtenait de mauvaises performances sur les problèmes *Linear* et *Expon*. Ces deux problèmes sont très adaptés à l'étude de méthodes d'exploration puisqu'ils requièrent une exploration exhaustive de l'environnement. Nous utilisons les paramètres suivants pour les agents dont l'exploration est dirigée :

- UNATSVIES : nous utilisons $\beta_s = 100$ et $w_s = 1$ pour l'exploration de la structure et $\forall X_i \in X$, $\beta_i = 0$ pour l'exploration des paramètres ;
- UNATSVIEP : nous utilisons $\beta_s = 0$ et $w_s = 0$ pour l'exploration de la structure et $\forall X_i \in X$, $\beta_i = 1$ pour l'exploration des paramètres.

Le protocole utilisé est similaire à celui utilisé lors du chapitre 4.3.2 : pendant 20 000 pas de temps, les agents UNATSVIEP, UNATSVIES et UNATSVI sont exécutés dans l'environnement. Une fois cette phase terminée, on calcule l'exactitude $\mathcal{Q}_{\chi^2}$ de la fonction de transition apprise par les agents (section 4.22, page 114) ainsi que l'erreur relative de la politique gloutonne ξ_π calculée à partir des modèles construits (section 4.3.1, page 104). Aucun bruit n'est ajouté dans les problèmes.

La figure 6.4 illustre l'incidence de la taille des problèmes *Linear* et *Expon* sur l'exactitude de la fonction de transition construite par apprentissage. Pour le problème *Linear* (figure 6.4(a)), nous pouvons remarquer qu'une différence importante apparaît pour un problème de taille 8 entre la qualité du modèle appris par UNATSVIES et celle des agents UNATSVIEP et UNATSVI, à la faveur du premier. Dû à la nature du problème et à la difficulté de son exploration, cette différence apparaît beaucoup plus tôt (dès la taille 4) pour le problème *Expon* (figure 6.4(b)). Cependant, on peut remarquer que la qualité de l'apprentissage baisse de façon importante pour l'agent UNATSVIES, bien qu'elle reste meilleure comparée à celle de UNATSVIEP et UNATSVI.

La figure 6.5 montre l'influence de la taille des deux problèmes sur l'erreur relative de la politique. Pour les deux problèmes, *Linear* (figure 6.5(a)) et *Expon* (figure 6.5(b)), on peut observer que l'agent UNATSVIES, par l'exploration de la structure, obtient de nettement meilleurs résultats que les agents UNATSVIEP et UNATSVI. En effet, pour le problème *Linear*, alors que la qualité de la politique est extrêmement mauvaise pour les agents UNATSVIEP et UNATSVI, dés que le problème a une taille de 7 (896 couples état/action), la qualité de la politique de l'agent UNATSVIES



(a) (b)

FIG. 6.4 – Incidence de la taille des problèmes *Linear* (figure a) et *Expon* (figure b) sur l'exactitude de la fonction de transition apprise par les agents UNATSVIES, UNATSVIEP et UNATSVI : l'exploration de la structure donne un avantage certain à l'agent UNATSVIES qui construit un modèle significativement plus exact que celui construit par les deux autres agents dés lors que la taille du problème augmente.

reste proche de la politique optimale. Pour le problème *Expon*, on observe un résultat similaire : la qualité de la politique est extrêmement mauvaise pour les agents UNATSVIEP et UNATSVI pour un problème de taille 4 (64 couples état/action), la qualité de la politique de l'agent UNATSVIES reste proche de la politique optimale jusqu'à un problème de taille 7 (896 couples état/action).

Ces résultats montrent l'intérêt d'utiliser des techniques d'exploration dirigée telle que nous les avons décrites lors de la section 6.1 (page 156), plus particulièrement l'exploration de la structure du problème. En effet, l'agent UNATSVI utilisant ϵ-greedy comme politique d'exploration et l'agent UNATSVIEP explorant seulement les paramètres du problème obtiennent des résultats rapidement mauvais lorsque la taille du problème augmente. L'agent UNATSVIES, en explorant la structure du problème, obtient de meilleurs résultats concernant l'exactitude du modèle dans tous les cas, et de meilleurs résultats concernant la qualité de sa politique gloutonne, lorsqu'il trouve l'unique récompense existante dans le problème.

6.3.2 Le problème *Factory*

Nous utilisons le problème *Factory* pour tester notre approche dans un grand problème. Malgré sa taille ($1,8 \cdot 10^6$ couples état/action) et contrairement à un problème tel que *Expon*, une exploration exhaustive n'est pas nécessaire puisque plusieurs récompenses sont accessibles à partir de nombreux états.

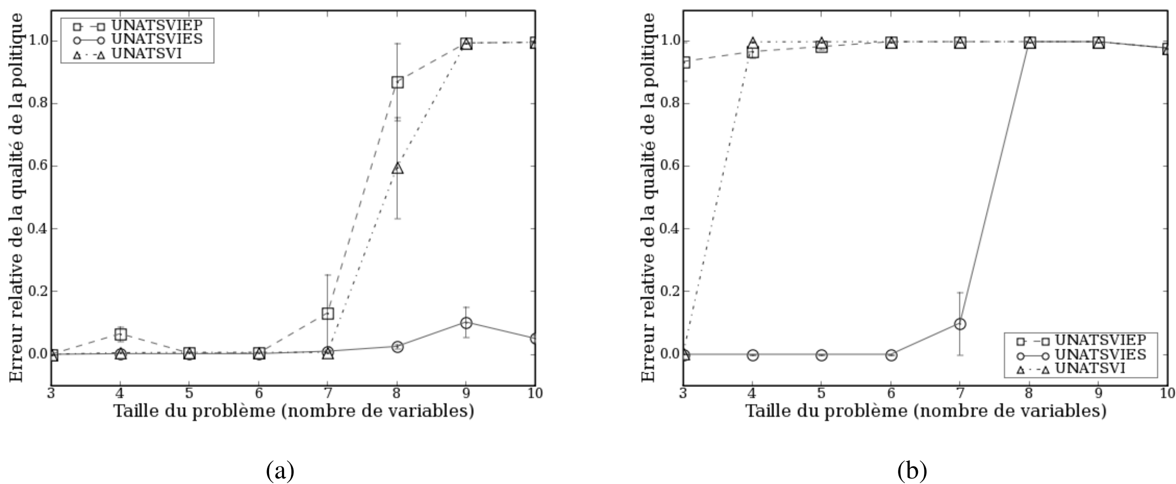


FIG. 6.5 – Incidence de la taille des problèmes *Linear* (figure a) et *Expon* (figure b) sur l'erreur relative de la politique gloutonne calculée à partir des modèles construits par les agents UNATSVIES, UNATSVIEP et UNATSVI : la politique de l'agent UNATSVIES explorant la structure du problème est significativement meilleure que celle construite par les deux autres agents dés lors que la taille du problème augmente.

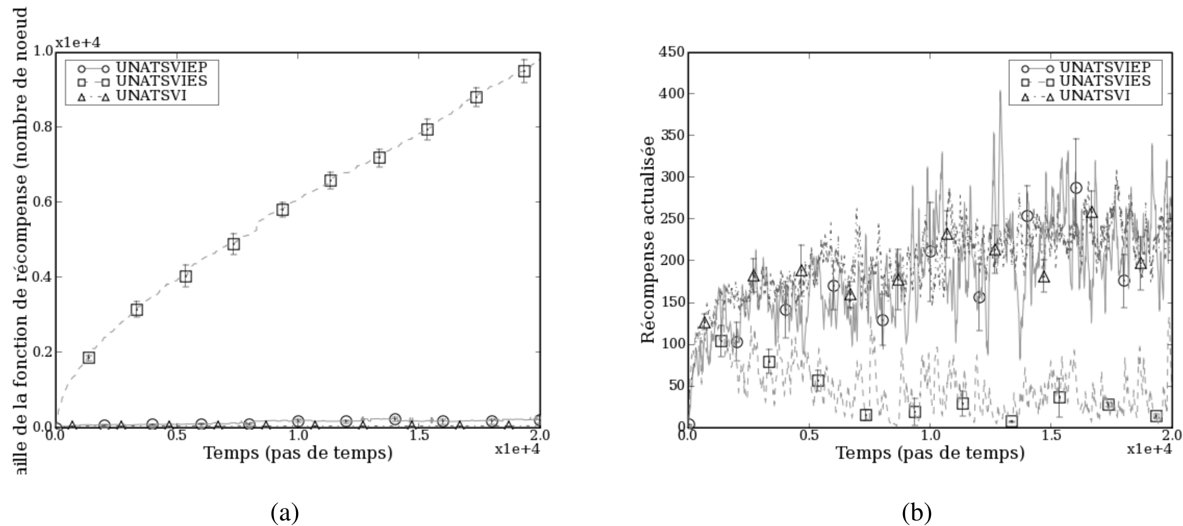
Nous utilisons le même protocole expérimental pour les résultats du chapitre 5, c'est-à-dire que l'état de l'agent est réinitialisé tous les 15 pas de temps. Dans ce problème nous avons testé les agents UNATSVIES et UNATSVIEP avec les valeurs de paramètres pour les bonus d'exploration de la structure et des paramètres du problème suivants :

– UNATSVIEP : nous utilisons $\beta_s = 0$ et $w_s = 0$ pour l'exploration de la structure et $\forall X_i \in X$, $\beta_i = 1$ pour l'exploration des paramètres ;
– UNATSVIES : nous utilisons $\beta_s = 0.01$ et $w_s = 0.01$ pour l'exploration de la structure et $\forall X_i \in X$, $\beta_i = 1$ pour l'exploration des paramètres.

Nous avons volontairement utilisé des petites valeurs pour les paramètres β_s et w_s de l'agent UNATSVIES afin de limiter l'exploration de la structure du problème. Ces agents sont comparés à un agent UNATSVI. Nous avons réalisé seulement une expérience pour obtenir les résultats présentés.

La figure 6.6 représente la taille de la fonction de récompense (figure 6.6(a)) construit par les différents agents UNATSVIES, UNATSVIEP et UNATSVI et la récompense actualisée obtenue (figure 6.6(b)) par ces mêmes agents. On remarque ainsi que la fonction de récompense construite par l'agent UNATSVIES (avec $w_s > 0$) atteint une taille de quasiment 10 000 nœuds (contre environ 100 nœuds pour les autres agents). De plus, on peut observer que cet agent obtient moins de récompenses actualisées que les agents UNATSVIEP et UNATSVI.

Les résultats présentés pour le problème *Factory* illustrent l'influence de l'exploration dirigée,



(a) (b)

FIG. 6.6 – Problème *Factory* : taille de la fonction de récompense (figure a) et de récompenses actualisées obtenues (figure b) construites par les agents UNATSVIES et UNATSVIEP, comparé à un agent UNATSVI. Alors que l'agent UNATSVIES obtient une faible récompense actualisée, il construit une représentation de la fonction de récompense de grande taille, contrairement à l'agent UNATSVIEP dont les performances sont similaires à celle obtenues par UNATSVI.

soit pour explorer la structure, soit pour explorer les paramètres. Ils montrent notamment, contrairement aux résultats concernant les problèmes *Expon* et *Linear*, que l'exploration de la structure est coûteuse, même lorsque le bonus attribué à son exploration est faible, puisque, d'une part, il obtient moins de récompenses actualisées que les autres agents et d'autre part, il nécessite une représentation d'une taille importante non compatible avec la résolution de problèmes de grande taille.

6.4 Synthèse

Dans ce chapitre, nous avons tout d'abord présenté plusieurs algorithmes d'exploration dirigée, à la fois dans le cadre des MDPs et des FMDPs, offrant des garanties quant aux propriétés de convergence.

Dans l'objectif d'intégrer ces méthodes au cadre SDYNA, nous avons souligné une difficulté importante concernant la mise en pratique de ces algorithmes dans des grands problèmes. Plus précisément, certain problème nécessitent une exploration exhaustive de l'environnement (par exemple le problème *Expon*) afin de découvrir complètement la structure de celui-ci. Au contraire, d'autres problèmes (par exemple le problème *Factory*) ne nécessitent pas une telle exploration et un agent avec une exploration non-dirigée obtient rapidement de bonnes performances.

Par conséquent, dans le cadre de l'apprentissage de la structure d'un problème et de façon complémentaire au compromis exploration/exploitation, nous pensons qu'un nouveau compromis existe entre, d'un côté, supposer que la structure apprise du problème est suffisante pour le représenter et, de l'autre côté, continuer d'explorer des sous-espace définis par cette structure pour découvrir de nouvelles distributions de probabilités conditionnelles.

Afin de gérer ce compromis, nous avons proposé d'utiliser deux types de bonus : un bonus d'exploration de la structure et un bonus d'exploration des paramètres de la structure. Nous calculons ces deux bonus à partir des algorithmes d'exploration existant dans la littérature. Pour le bonus d'exploration de la structure, nous avons décrit une adaptation de l'algorithme R-MAX, un algorithme d'exploration dirigé dans les MDPs, utilisé avec l'algorithme de planification IncSVI. Pour le bonus d'exploration des paramètres de la structure, nous utilisons l'algorithme f-IE, un algorithme d'exploration dirigée dans les FMDPs.

Nous avons testé notre algorithme avec deux agents, le premier basé principalement sur l'exploration de la structure, le deuxième sur l'exploration des paramètres de la structure. Ces deux agents ont chacun montré leur limite respective. Le premier agent, basé sur l'exploration de la structure, obtient de bons résultats sur le problème *Expon* mais une mauvaise performance sur un grand problème tel que *Factory*. Le deuxième agent, basé sur l'exploration des paramètres de la structure, obtient de bons résultats sur un grand problème tel que *Factory* mais une mauvaise performance sur le problème *Expon*.

D'une part, ces résultats illustrent la nécessité de faire un compromis pour l'exploration de la structure. D'autre part, ils soulignent la difficulté de représenter les fonctions nécessaires à l'exploration lorsque la structure du problème est inconnue. L'algorithme que nous avons proposé permet de définir le type d'exploration adapté au problème. Cependant, les résultats montrent que seule les comportements extrêmes entre l'exploration de la structure et l'exploration des paramètres sont efficaces suivant la nature du problème. Par conséquent, des travaux supplémentaires sont nécessaires pour obtenir un meilleur compromis et de meilleurs résultats pour l'exploration des problèmes de grande taille lorsque la structure est inconnue.

Chapitre 7

Application au jeu vidéo Counter-Strike

Afin de valider l'approche SDYNA sur un problème réel, nous avons utilisé le jeu vidéo comme plate-forme de test. Les jeux vidéos présentent l'avantage de proposer des problèmes réels dans un contexte de développement et de tests facilement contrôlable et peu coûteux (Sigaud, 2004; Robert, 2005). Nous avons choisi le jeu Counter-Strike© [1], une extension du jeu Half-Life© [2] développée par Valve© [3], principalement à cause de sa popularité dans le milieu du jeu vidéo.

Les résultats que nous présentons dans cette section ont pour but, non pas de comparer les performances de SDYNA à une autre approche dans les jeux vidéos, mais plutôt de donner une illustration de l'applicabilité de SDYNA à un problème réel. Par conséquent, contrairement aux résultats présentés précédemment, les résultats que nous montrerons ne seront pas quantitatifs mais qualitatifs. Par cette expérimentation, notre but est de répondre à des questions telles que : dans quelle mesure peut-on utiliser SDYNA sur un problème réel ? quels changements sont nécessaires à SDYNA pour être utilisé dans un problème réel ? les représentations construites par l'apprentissage sont-elles compréhensibles et/ou manipulables ?

Cette section est structurée de la façon suivante : nous commençons, section 7.1, par décrire le jeu Counter-Strike© d'une façon générale. Dans la section 7.2, nous décrivons le problème posé pour SDYNA, de même que la formalisation que nous proposons pour la résolution de ce problème. La mise en œuvre de SDYNA sera décrite section 7.3. La section 7.4 décrit les résultats que nous avons obtenus lors des expérimentations dans le jeu. Nous discutons ces résultats dans la section 7.5.

7.1 Description du jeu

Counter-Strike© est un jeu de tir subjectif, ou *First-Person Shooter*, dans lequel deux équipes, des terroristes et des antiterroristes, s'affrontent dans des courtes parties de quelques minutes. La

[1] http://www.counter-strike.net
[2] http://planethalflife.gamespy.com
[3] http://www.valvesoftware.com

figure 7.1(a) illustre un exemple de vue à la première personne du joueur dans le jeu. Suivant la carte utilisée, les deux équipes ont des objectifs différents. Pour nos expériences, nous avons utilisé la carte *de_dust*, montrée figure 7.2.

(a) (b)

FIG. 7.1 – Différentes captures d'écran du jeu Counter-Strike©. Figure a : vue du joueur dans le jeu. Figure b : un joueur de l'équipe terroriste installant la bombe sur le site de bombe A.

Sur cette carte, l'équipe terroriste apparaît au lieu marqué **T** (voir figure 7.2) et doit poser une bombe sur l'un des deux sites de bombe existants dans la carte (marqué **SA** et **SB** sur la carte de la figure 7.2). Les joueurs de l'équipe antiterroriste démarrent au lieu marqué **A** et doivent empêcher les joueurs de l'équipe terroriste de poser la bombe. Plusieurs conditions peuvent causer la fin de la partie en cours :

- le temps limite pour effectuer la mission a été dépassé ;
- la bombe a été posée par l'équipe terroriste et a explosé ;
- la bombe a été posée par l'équipe terroriste et a été désamorcée par l'équipe antiterroriste ;
- tous les membres d'une des deux équipes sont “morts”.

A la fin de chaque partie, chaque joueur de chaque équipe reçoit de l'argent en fonction de son score et de celui de son équipe et peut s'acheter, au début de la partie suivante, de nouvelles armes, de l'équipement supplémentaire (tel qu'un gilet pare-balles), des grenades (explosives, aveuglantes, ...) ou bien le garder pour la prochaine partie.

Au cours de la partie, un joueur peut donc se déplacer (sur toute la carte), sélectionner une arme, tirer, ramasser des munitions, des armes ou la bombe, laissées au sol par d'autres joueurs et lancer des grenades. Une seule bombe existe lors d'une partie. Le joueur de l'équipe terroriste portant la bombe peut l'amorcer sur l'un des deux sites de bombe (figure 7.1(b)). Une fois que la bombe est

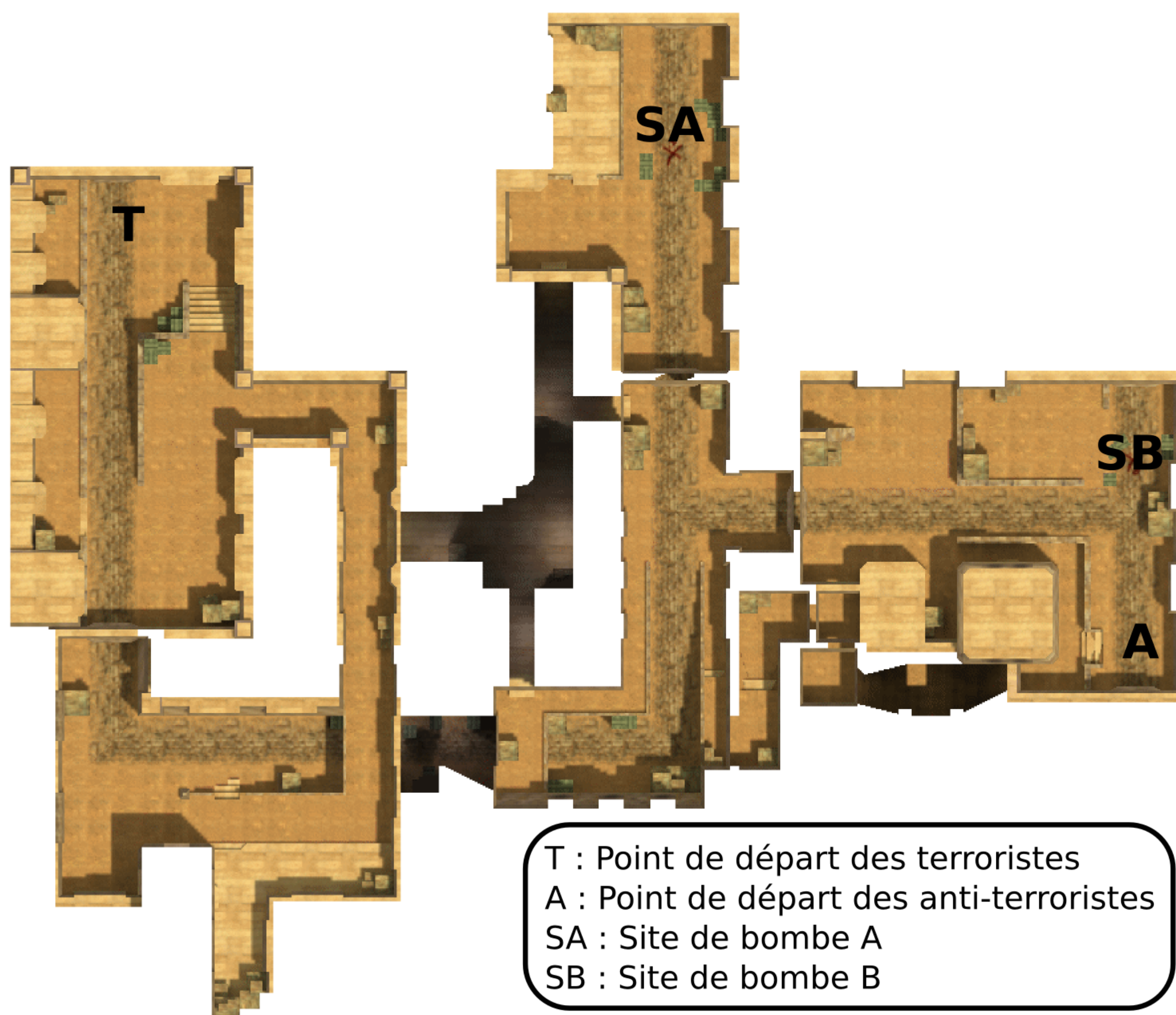


FIG. 7.2 – La carte *de_dust*. L'équipe terroriste doit déposer une bombe sur l'un des deux sites de bombe. L'équipe antiterroriste doit l'en empêcher.

posée, les joueurs de l'équipe antiterroriste peuvent la désamorcer avant que celle-ci explose.

Le joueur démarre une partie avec 100 points de vie. À chaque fois qu'il est touché (par un autre joueur, par une grenade ou bien par la bombe lorsqu'elle explose), il perd un certain nombre de points de vie (dépendant de l'endroit où il a été touché). Lorsque le nombre de points de vie est à 0, le joueur est mort et doit attendre le début de la partie suivante pour pouvoir jouer à nouveau. De plus, il n'est pas possible pour un joueur de récupérer des points de vie lors d'une partie. Un joueur qui n'a pas perdu tous ses points de vie à la fin d'une partie conserve son matériel pour la partie suivante.

L'équipe terroriste gagne soit parce que la bombe a explosé, soit parce que tous les joueurs de l'équipe antiterroriste ont été éliminés. L'équipe antiterroriste gagne soit parce que la bombe a été posée (par l'équipe adverse) puis désamorcée, soit parce que tous les joueurs de l'équipe terroriste

ont été éliminés ou bien encore soit parce que le temps limite pour effectuer la mission est dépassé.

Enfin, à chaque fois qu'une équipe gagne, elle marque un point. Chaque joueur peut connaître aussi le nombre d'ennemis qu'il a "tué" et le nombre de fois qu'il est mort.

7.2 Définition et formalisation du problème

Le problème que nous définissons pour tester l'approche SDYNA concerne le contrôle de *Personnage Non Joueur* (PNJ), c'est-à-dire des personnages du jeu (de l'équipe terroriste ou antiterroriste) contrôlés par l'ordinateur. Dans ses premières versions, Counter-Strike© était un jeu qui se jouait exclusivement en réseau, c'est-à-dire que les joueurs pouvaient y jouer soit sur Internet, soit en constituant un réseau local[4]. Plusieurs extensions non officielles (développées par des volontaires) ont été proposées pour que l'ordinateur puisse contrôler des PNJs et qu'un joueur puisse jouer tout seul, contre l'ordinateur. Nous nous posons donc la question de savoir si SDYNA pourrait être une aide au développement de telles extensions, plus précisément : est-ce-qu'un agent SDYNA peut apprendre à contrôler un PNJ dans Counter-Strike© ?

Afin de limiter le travail de développement important que nécessite une telle expérimentation, nous avons limité plusieurs paramètres dans le jeu. Notre but est d'apprendre à contrôler un PNJ de l'équipe terroriste et un PNJ de l'équipe antiterroriste. Un PNJ de l'équipe terroriste doit apprendre à aller poser la bombe et à savoir quand utiliser son arme. Nous n'avons pas eu le temps de développer le code nécessaire pour qu'un PNJ de l'équipe antiterroriste puisse apprendre à désamorcer la bombe. Par conséquent, un PNJ de cette équipe doit principalement apprendre à savoir quand utiliser son arme. De plus, les PNJs n'utilisent qu'une seule arme. Enfin, l'argent gagné lors de la mission précédente est utilisé de façon automatique pour acheter des munitions uniquement.

La structure de cette section suit plus ou moins la méthodologie que nous avons utilisée pour réaliser nos tests de SDYNA dans Counter-Strike©. Section 7.2.1, nous commencerons donc par définir les différentes récompenses que l'agent pourra obtenir dans le jeu. Section 7.2.2, nous définirons les perceptions de l'agent dans son environnement. Section 7.2.3, nous définirons les différentes actions qu'un agent pourra exécuter dans l'environnement. Enfin, section 7.2.4, nous terminerons par définir la notion de pas de temps.

7.2.1 Définition des récompenses

A travers les récompenses que le PNJ peut obtenir dans le jeu, on définit ce que l'agent SDYNA doit apprendre. Or, un agent SDYNA contrôlant un PNJ de l'équipe terroriste aura trois objectifs différents à remplir :

[4]Ce n'est plus le cas de la nouvelle version de Counter-Strike©, basée sur le moteur du jeu Half-Life© 2 et pour lequel il est possible de jouer contre l'ordinateur.

1. rester en vie ;
2. éliminer le plus possible des joueurs de l'équipe adverse ;
3. poser la bombe, s'il la possède.

On associe donc une récompense correspondant à chacun de ces objectifs :

1. -1 lorsque le PNJ est tué (0 sinon) ;
2. +10 lorsque le PNJ tue un adversaire (0 sinon) ;
3. +1 lorsque la bombe est posée (0 sinon).

On procède de la même façon concernant l'agent SDYNA contrôlant un PNJ de l'équipe antiterroriste. Celui-ci a deux objectifs :

1. rester en vie
2. éliminer le plus possible des joueurs de l'équipe adverse.

Il obtient donc comme récompense :

1. -1 lorsque le PNJ est tué (0 sinon) ;
2. +10 lorsque le PNJ tue un adversaire (0 sinon).

Contrairement aux autres récompenses, la récompense obtenue par le PNJ de l'équipe terroriste lorsque la bombe est posée ne correspond pas au score obtenu par son équipe. En effet, les récompenses associées à "rester en vie" et "éliminer le plus possible des joueurs de l'équipe adverse" correspondent directement aux scores du joueur, respectivement le nombre de fois que le PNJ est mort et le nombre de fois qu'il a tué un adversaire.

Il aurait été plus difficile de donner une récompense associée aux scores de l'équipe puisque, comme nous l'avons décrit lors de la section 7.1 (page 175), une équipe peut gagner pour plusieurs raisons différentes : lorsque tous les joueurs de l'équipe adverse ont été éliminés ou bien lorsque la mission a été accomplie. Pour que l'apprentissage d'une telle récompense soit possible, nous aurions été obligé d'ajouter un grand nombre de variables aléatoires dans l'espace d'état, par exemple pour indiquer le nombre de joueurs restant dans l'équipe adverse. Nous avons donc simplifié le problème en attribuant une récompense pour un PNJ de l'équipe terroriste lorsque la bombe était posée.

7.2.2 Définition de l'ensemble d'états

Afin de pouvoir représenter le problème sous la forme d'un FMDP, nous décrivons l'espace d'état des agents SDYNA par un ensemble de variables aléatoires. Chaque variable aléatoire correspond à une perception de l'agent de son environnement, c'est-à-dire une observation concernant soit l'état courant du jeu, soit l'état personnel du PNJ. Dans la suite de ce chapitre, nous utiliserons le mot perception plutôt que variable aléatoire.

Les perceptions communes aux PNJs des deux équipes dans le jeu sont :

– Position courante : cette perception indique au PNJ sa position courante dans la carte. Les valeurs possibles sont : Site de bombe A, Site de bombe B, Point de départ des terroristes, Point de départ des antiterroristes et Ailleurs ;
– A été touché : cette perception indique si le PNJ vient d'être touché. Les valeurs possibles sont : Oui et Non ;
– Munitions : cette perception indique au PNJ s'il lui reste des munitions. Les valeurs possibles sont : Oui et Non ;
– Santé : cette perception indique au PNJ s'il est encore vivant dans le jeu. Les valeurs possibles sont : Mort et Vivant ;
– Cible en vue : cette perception indique au PNJ s'il voit une cible sur laquelle il pourrait tirer. Les valeurs possibles sont : Oui et Non.

Pour que les PNJ de l'équipe terroriste puissent apprendre à poser la bombe, nous rajoutons deux perceptions supplémentaires :

– Possession de la bombe : cette perception indique au PNJ s'il possède la bombe ou non. Les valeurs possibles sont : Oui et Non ;
– Statut de la bombe : cette perception indique aux PNJs de l'équipe si la bombe a été posée. Les valeurs possibles sont : Posée et Non posée. De plus, elle indique au PNJ portant la bombe si, actuellement, il peut la poser. Dans ce cas, la valeur supplémentaire Peut être posée est utilisée.

Ainsi, la taille des espaces d'état sont de 80 et 480 états respectivement pour les PNJs de l'équipe des antiterroristes et des terroristes. Pour les PNJs de l'équipe des terroristes, certains états sont impossibles. Par exemple, la perception de l'état de la bombe ne peut pas égale à Peut être posée lorsque le PNJ n'est pas sur un site de bombe.

Par conséquent, comparé aux problèmes abordés dans les chapitres précédents, le problème que nous tentons de résoudre est de (très) petite taille. Cependant, nous verrons section 7.2.5 qu'il présente plusieurs difficultés supplémentaires, le rendant ainsi très informatif concernant la robustesse de l'approche SDYNA dans un problème réel.

7.2.3 Définition de l'ensemble d'actions

On définit les différentes interactions d'un PNJ dans le jeu en définissant l'ensemble des actions qu'il peut effectuer. Les actions communes aux PNJs des deux équipes dans le jeu sont :

– Tirer : tire sur l'un des adversaires présents dans le champ de vision du PNJ, sinon tire dans le vide ;
– Aller au départ des terroristes : se déplace vers le point de départ des joueurs de l'équipe terroriste ;

– Aller au départ des antiterroristes : se déplace vers le point de départ des joueurs de l'équipe antiterroriste ;
– Aller sur le site de bombe A : se déplace vers le site de bombe A ;
– Aller sur le site de bombe B : se déplace vers le site de bombe B.

Une action supplémentaire est ajoutée pour les PNJs de l'équipe terroriste afin qu'ils puissent armer la bombe sur l'un des deux sites de bombe :

– Poser la bombe : arme la bombe si le PNJ est sur un site de bombe et qu'il porte la bombe, sinon, cette action n'a aucun effet.

Toutes ces actions sont des actions de haut niveau supposant l'existence d'actions bas niveau afin de pouvoir être exécutées. Par exemple, pour exécuter une action de type "Aller à", nous supposons l'existence d'un graphe représentant la carte que nous utilisons avec un algorithme A^* afin de planifier la trajectoire de l'agent pour qu'il puisse atteindre son objectif. De même, pour l'action Tirer, nous supposons l'existence d'un algorithme permettant de sélectionner la cible et d'orienter le PNJ vers celle-ci, puis de tirer.

7.2.4 Définition des pas de temps

Un problème important dans la formalisation est la définition d'un pas de temps pour l'apprentissage. Ainsi, un PNJ doit pouvoir : d'une part, exécuter des actions de courte durée, par exemple l'exécution de l'action Tirer dure une ou deux secondes ; d'autre part, exécuter des actions de longue durée, par exemple l'exécution d'une action de déplacement peut durer plusieurs dizaines de secondes ; et enfin, avoir des réflexes, par exemple lorsqu'il détecte une cible potentielle, son temps de réaction doit être autour de 100ms. De plus, l'état du PNJ et l'action qu'il effectue sont rafraîchis par le serveur environ toutes les 10ms (cette fréquence n'est pas constante et dépend étroitement de la charge du serveur).

Il est important de noter que ces différences d'échelle dans le temps pose un problème uniquement pour l'apprentissage et non pour l'exécution d'une politique de l'agent qui serait déjà connue. En effet, nous rappelons qu'un agent SDYNA apprend à partir d'une observation $\langle s, a, s', r \rangle$ de l'environnement, c'est-à-dire une transition entre un état précédent s et un état courant s', après avoir exécuté l'action a et reçu la récompense r. Pour un apprentissage tel que celui de SDYNA, la difficulté concerne donc la détection de la fin de l'exécution de l'action a.

Pour cela, nous avons utilisé une heuristique simple : la fin d'une action est définie comme étant soit un changement de l'état du PNJ (c'est-à-dire la modification de la valeur d'une des perceptions), soit après un temps maximum, $T_{max} = 750$ rafraîchissements, pour lequel l'état de l'agent n'a pas changé. La valeur T_{max} a été fixée en fonction du temps nécessaire (approximatif) pour effectuer la plus longue action possible. Dans notre cas, nous avons compté le nombre de rafraîchissements exécutés pendant que l'agent part du point de départ de l'équipe terroriste et arrive au point de

départ de l'équipe antiterroriste.

7.2.5 Remarques concernant le problème

A partir de la définition de ce problème, nous pouvons souligner plusieurs de ses caractéristiques. En premier lieu, contrairement à tous les problèmes qui ont été traités jusqu'à présent, le problème dans le jeu Counter-Strike© tel qu'il a été défini ci-dessus ne satisfait pas l'hypothèse de Markov. En effet, par exemple, si l'on considère la façon dont la perception Munitions a été formalisée, on peut remarquer que sa valeur au pas de temps suivant ne dépend pas seulement de sa valeur au pas de temps courant et de l'action réalisée par l'argent, elle dépend aussi du nombre de fois que l'action Tirer a été exécutée dans l'historique de l'agent. Par conséquent, l'égalité suivante :

$$P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0) = P(s_{t+1}|s_t, a_t)$$

définissant l'hypothèse de Markov (equation 2.1, page 23) n'est plus respectée.

Cependant, nous pensons que SDYNA reste adaptée à la résolution de ce problème. En effet, toutes les informations nécessaires pour que l'agent puisse décider une action pertinente sont présentes dans l'ensemble des perceptions décrivant l'état courant s de l'agent. Il n'est donc pas nécessaire de consulter l'historique de l'agent pour que la meilleure action puisse être déterminée.

En deuxième lieu, nous pouvons remarquer que les perceptions et les actions que nous avons définies ne sont pas dépendantes de la carte que nous utilisons. En effet, toutes les cartes du même type que *de_dust* (la carte que nous utilisons pour nos expériences) possèdent deux sites de bombe ainsi que deux points de départ pour chaque équipe. Par conséquent, l'apprentissage réalisé sur une carte est directement réutilisable sur une autre carte du même type dans le jeu Counter-Strike©. Cette propriété vient du fait que nous utilisons une représentation haut niveau et abstraite de la carte basée sur aucune information spécifique de celle-ci (comme par exemple sa topologie).

Enfin, une dernière remarque concernant le problème concerne le bruit présent dans celui-ci. D'une part, certaines actions choisies par l'agent sont soumises à des paramètres de l'environnement empêchant leur bon déroulement lors de leur exécution par le PNJ dans le jeu. Nous avons, par exemple, remarqué plusieurs fois que lorsque deux PNJs se croisaient, ils pouvaient rester bloqués l'un contre l'autre, empêchant ainsi le déroulement correct des actions de navigation. D'autre part, les agents doivent apprendre en présence d'ennemis dont le comportement évolue au fil du temps.

7.3 Mise en œuvre de SDYNA

Pour pouvoir mettre en œuvre SDYNA pour la résolution du problème que nous avons défini dans le jeu Counter-Strike©, il est nécessaire de prendre en compte une différence notable avec les problèmes précédents sur lesquels SDYNA a été testé jusqu'à présent. Cette différence concerne

la récompense obtenue par un PNJ lorsque celui-ci élimine un joueur. En effet, contrairement aux autres récompenses définies jusqu'à présent, elle est stochastique. Lorsqu'un PNJ commence à tirer sur une cible qu'il voit, plusieurs issues peuvent terminer le combat, telles que la cible est morte, le PNJ est mort (il s'est fait toucher par sa cible ou un autre joueur) ou encore il peut ne plus avoir de munitions.

Afin de pouvoir apprendre des fonctions stochastiques pour construire une représentation de la fonction de transition d'un FMDP représentant le problème, nous avons utilisé un pré-élagage basé sur la mesure d'information pour des valeurs symboliques (voir section 4.2.1, page 88 et section 5.2.2, page 134). Nous utilisons une technique similaire pour l'apprentissage des fonctions de récompense du problème Counter-Strike©. Ainsi, un nœud de décision doit satisfaire deux critères pour être installé :

1. un critère évaluant une certitude : un nœud de décision testant l'attribut $\mathcal{V}$ est installé si :

$$\forall \nu \in \underline{\text{Dom}}(\mathcal{V}) : n_{\mathcal{E}}^{\nu} \geq N_{\nu} \tag{7.1}$$

   avec $n_{\mathcal{E}}^{\nu}$ représentant le nombre d'exemples de l'ensemble $\mathcal{E}$ tel que $\mathcal{V} = \nu$ et N_{ν} le nombre d'exemples requis dans chacune des branches pour qu'un nœud de décision soit installé (on utilise $N_{\nu} = 4$).

2. un critère évaluant l'approximation de l'apprentissage : un test est installé si au moins l'une de ses branches est pure. Ce critère suppose qu'il n'existe pas de bruit dans la récompense obtenue par l'agent et n'est pas adapté à l'apprentissage de fonctions stochastiques en général.

Dans le cadre de l'apprentissage incrémental de la fonction de récompense, nous utilisons donc l'algorithme d'induction d'arbres de décision incrémental UpdateTreeS (figure 5.3, section 5.2.2, page 135).

Enfin, l'agent SDYNA que nous utilisons est composé de l'algorithme UpdateFMDPnAT (figure 5.5, section 5.2.2, page 137) pour la mise à jour du FMDP, de l'algorithme IncSVI (figure 5.6, section 5.3.1, page 138) pour la planification et l'algorithme ϵ-greedy (section 2.3.1, page 33) pour la sélection de l'action. Nous utilisons exactement les même paramètres que pour les problèmes précédents, c'est-à-dire une valeur de $\gamma = 0.99$, un seuil pour l'apprentissage des distributions de probabilités conditionnelles de $\tau_{\chi^2} = 30$ et $\epsilon = 0.1$ pour l'algorithme d'exploration ϵ-greedy.

7.4 Résultats

Le protocole expérimental que nous utilisons est similaire à celui utilisé dans le chapitre 5. L'équipe des terroristes est composée de deux PNJs utilisant chacun **le même** agent SDYNA décrit ci-dessus. Ainsi, l'apprentissage est partagé par les deux PNJs de l'équipe. L'équipe des antiterroristes est composée d'un seul joueur utilisant lui aussi l'agent SDYNA décrit ci-dessus. L'avantage est

ainsi donné aux PNJs de l'équipe terroriste. Une partie du jeu dure cinq minutes. Enfin, les résultats présentés dans cette section ont été obtenus après un temps d'apprentissage de 2h08 pour l'agent SDYNA contrôlant les PNJs de l'équipe des terroristes, et de 2h38 pour l'agent SDYNA contrôlant les PNJs de l'équipe des antiterroristes. Ces durées d'expérience correspondent à la fois au temps réel et au temps écoulé dans le jeu. En effet, nous n'avions pas la possibilité d'accélérer le serveur de jeu.

Cette section décrit plusieurs fonctions du FMDP ainsi construit par les agents lors de l'apprentissage et de la planification. Nous commençons par décrire, section 7.4.1, les fonctions de récompense. Section 7.4.2, nous décrirons plusieurs distributions de probabilités conditionnelles extraites de la fonction de transition. Enfin, nous décrirons plusieurs politiques gloutonnes construites par la planification dans la section 7.4.3.

7.4.1 Fonction de récompense

Le but de l'apprentissage supervisé effectué par SDYNA est la construction d'un FMDP représentant le problème à résoudre. Concernant les fonctions de récompense de ce FMDP, l'apprentissage construit une représentation (sous la forme d'un arbre de décision dans notre cas) qui, en fonction des perceptions de l'agent et de l'action qu'il exécute, associe la récompense obtenue par l'agent. Nous rappelons que cet apprentissage est réalisé par l'algorithme incrémental d'induction d'arbres de décision UpdateTreeS (figure 5.3, page 135) en utilisant la mesure des moindres carrés (section 4.1.3, page 86).

La figure 7.3 montre le résultat d'un tel apprentissage pour les PNJs de l'équipe terroriste pour chaque objectif : “poser la bombe”, “rester en vie” et “éliminer le plus possible des joueurs de l'équipe adverse” illustrés respectivement dans les figures 7.3(a), 7.3(b), 7.3(c).

Ainsi, on peut constater que l'objectif “éliminer le plus possible des joueurs de l'équipe adverse” associant une récompense de +10 lorsque le PNJ tue un adversaire et 0 sinon, a été représenté par l'apprentissage en fonction des perceptions de l'agent et de l'action qu'il exécute. Plus précisément, l'agent a appris qu'il reçoit une récompense non nulle (différente de +10 puisqu'il ne tue pas sa cible systématiquement) lorsqu'il exécute l'action Tirer, qu'il voit une cible et qu'il lui reste des munitions.

Un apprentissage similaire a été effectué pour les autres objectifs du PNJ. Pour l'objectif “rester en vie”, l'apprentissage a associé une récompense de -1 lorsque la perception de l'état de santé du PNJ indique que celui-ci est mort. Enfin, pour l'objectif “poser la bombe”, l'apprentissage a associé une récompense de +1 lorsque la perception du statut de la bombe indique que celle-ci est posée. On remarque que, pour ces deux objectifs, la récompense obtenue par l'agent ne dépend pas d'une action en particulier mais plutôt de son état courant et de celui de la partie.

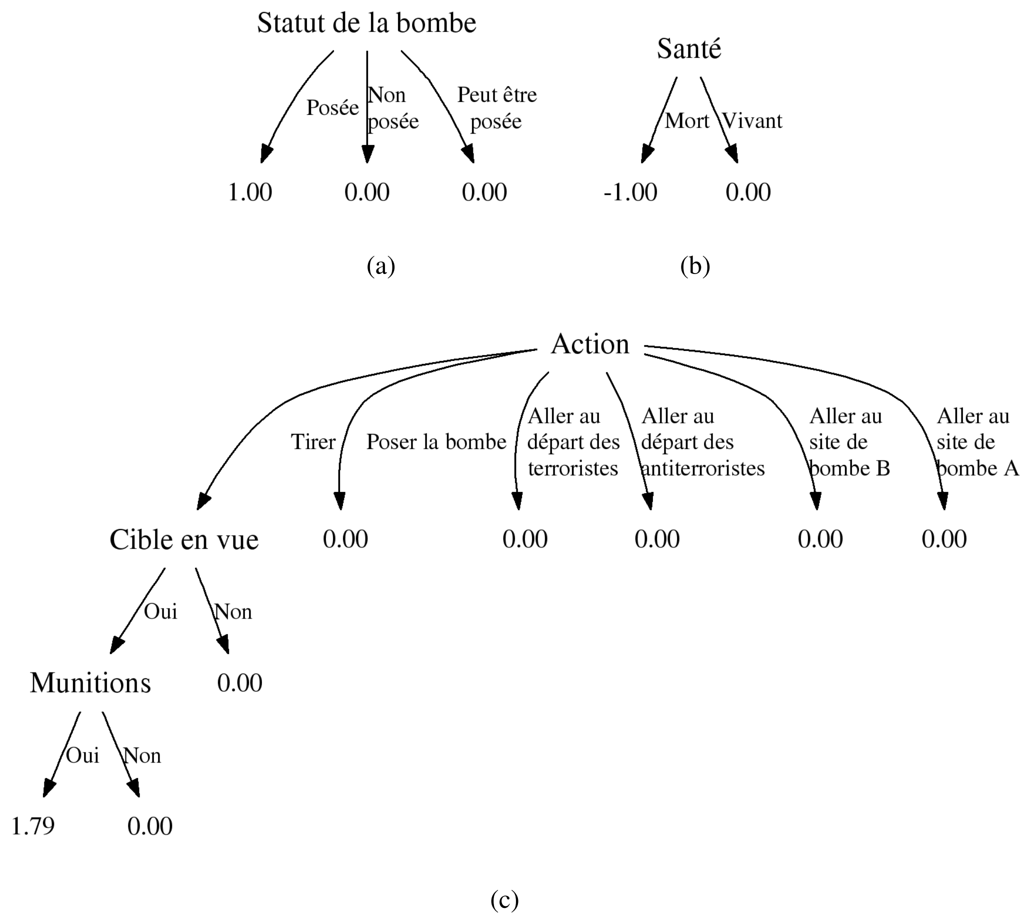


FIG. 7.3 – Résultat de l'apprentissage d'un PNJ de l'équipe des joueurs terroristes pour les récompenses associées aux objectifs "poser la bombe" (figure a), "rester en vie" (figure b) et "éliminer le plus possible des joueurs de l'équipe adverse" (figure c). Dans la figure c, la feuille marquée 1.79 indique que l'agent estime que s'il exécute l'action Tirer, qu'il voit une cible et qu'il a des munitions, alors il obtiendra une récompense de 1.79.

Ces résultats montrent que, d'une part, l'agent SDYNA a construit des représentations pertinentes des récompenses obtenues par les PNJs dans le jeu et, d'autre part, que les représentations construites par l'induction d'arbres de décision sont directement intelligibles par un utilisateur.

Lors de la définition d'un problème d'apprentissage par renforcement, il est nécessaire de définir les récompenses obtenus par l'agent, comme nous l'avons fait section 7.2.1 (page 178) pour ce problème, afin de déterminer quels sont les objectifs de l'agent dans son environnement. Par conséquent, l'apprentissage des fonctions de récompense pourrait sembler moins nécessaire que celui des fonctions de transition.

Cependant, l'apprentissage de ces fonctions supprime une contrainte importante pour la définition des objectifs. En effet, il n'est pas nécessaire que ceux-ci soient exprimées en fonction des différentes perceptions de l'agent dans son environnement. Ainsi, l'apprentissage permet de

construire les fonctions de récompense caractérisant les objectifs du problème à partir des corrélations observées entre les perceptions et les récompenses obtenues dans l'environnement.

7.4.2 Fonction de transition

Le but de l'apprentissage supervisé effectué par SDYNA est la construction d'un FMDP représentant le problème à résoudre. Concernant les fonctions de transition de ce FMDP, l'apprentissage doit construire une représentation (sous la forme d'un arbre de décision dans notre cas) d'une distribution de probabilités conditionnelle pour chaque perception et pour chaque action de l'agent. En d'autres termes, l'apprentissage construit une fonction indiquant pour chaque perception quelle sera la valeur de cette perception au prochain pas de temps, en fonction des perceptions et de l'action de l'agent au pas de temps courant. Nous rappelons que l'apprentissage est effectué par l'algorithme UpdateFMDPnAT (figure 5.5, page 137). Par conséquent, un arbre par perception est construit (et non un arbre par perception et par action). Nous rappelons aussi que chaque arbre est mis à jour avec l'algorithme incrémental d'induction d'arbres de décision UpdateTreeS (figure 5.3, page 135) en utilisant le critère du χ^2 (section 4.1.2, page 84). Cette section présente quelques exemples caractéristiques de distributions de probabilités conditionnelles construites par l'apprentissage pour les PNJs appartenant à l'équipe des terroristes.

La figure 7.4 montre le résultat de l'apprentissage pour la perception Cible en vue. L'arbre construit n'est composé que d'un seul nœud de décision testant la valeur de la perception indiquant la position courante de l'agent. D'après l'apprentissage, on peut donc voir que le fait de voir une cible au prochain pas de temps ne dépend pas de l'action exécutée par l'agent. De plus, on peut lire que l'agent estime qu'il est plus probable de voir une cible au prochain pas de temps lorsque sa position courante est Ailleurs (avec une probabilité de 0.35) ou Site de bombe B (avec une probabilité de 0.33) plutôt que Point de départ des antiterroristes (probabilité de 0.07).

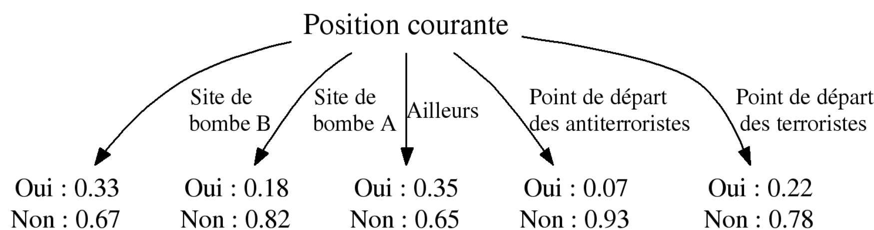


FIG. 7.4 – Résultat de l'apprentissage d'un PNJ de l'équipe des joueurs terroristes pour la distribution de probabilités conditionnelle de la variable Cible en vue. La feuille la plus à droite indique que l'agent estime qu'il a une probabilité de 0.22 de voir une cible au prochain pas de temps (et une probabilité de 0.78 de ne pas en voir) lorsque sa position courante est le point de départ des terroristes.

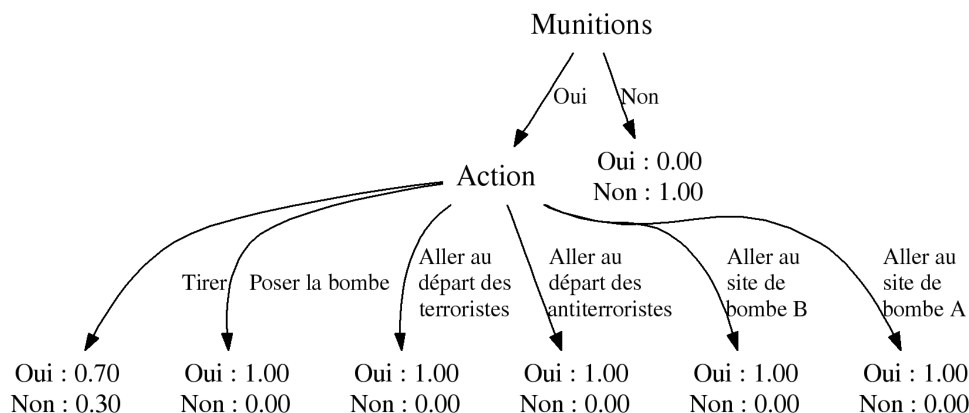


FIG. 7.5 – Résultat de l'apprentissage d'un PNJ de l'équipe des joueurs terroristes pour la distribution de probabilités conditionnelle de la variable Munitions. La feuille la plus à gauche indique que l'agent estime qu'il a une probabilité de 0.3 de ne plus avoir de munitions au prochain pas de temps (et une probabilité de 0.7 d'en avoir encore) lorsqu'il a des munitions et qu'il exécute l'action Tirer.

La figure 7.5 montre le résultat de l'apprentissage pour la perception Munitions. L'arbre construit est composé de deux nœuds de décision testant la valeur de la perception au pas de temps courant et l'action exécutée par l'agent. On remarque que lorsque l'agent n'a plus de munitions, alors il a une probabilité de 1.0 de ne pas en avoir au prochain pas de temps, représentant ainsi le fait qu'il n'est pas possible dans l'expérience que nous avons menée de récupérer des munitions. Lorsque l'agent a des munitions, on remarque aussi que l'action Tirer est la seule action ne garantissant pas qu'il en aura au prochain pas de temps (avec une probabilité de 0.3 que l'agent n'ait plus de munitions au prochain pas de temps), traduisant ainsi le fait qu'exécuter l'action Tirer coûte des munitions (ce qui n'est pas le cas des autres actions).

La figure 7.6 montre le résultat de l'apprentissage pour la perception Statut de la bombe, une variable représentant l'état courant de l'environnement plutôt que l'état interne du PNJ. On remarque en premier lieu que lorsque la bombe est posée, alors il y a une probabilité de 1.0 qu'elle soit posée au prochain pas de temps, illustrant ainsi que la bombe n'a jamais été désamorcée (puisque les antiterroristes n'en ont pas la capacité).

L'arbre indique aussi que pour qu'un agent puisse poser la bombe, il est nécessaire qu'elle ne soit pas posée, que sa position courante soit sur un site de bombe (A ou B) et qu'il possède la bombe. Lorsque la bombe peut être posée (seul l'agent possédant la bombe peut obtenir une telle valeur pour la perception Statut de la bombe), alors on observe que seule l'action Poser la bombe est associée à une probabilité non nulle que la bombe soit posée au prochain pas de temps.

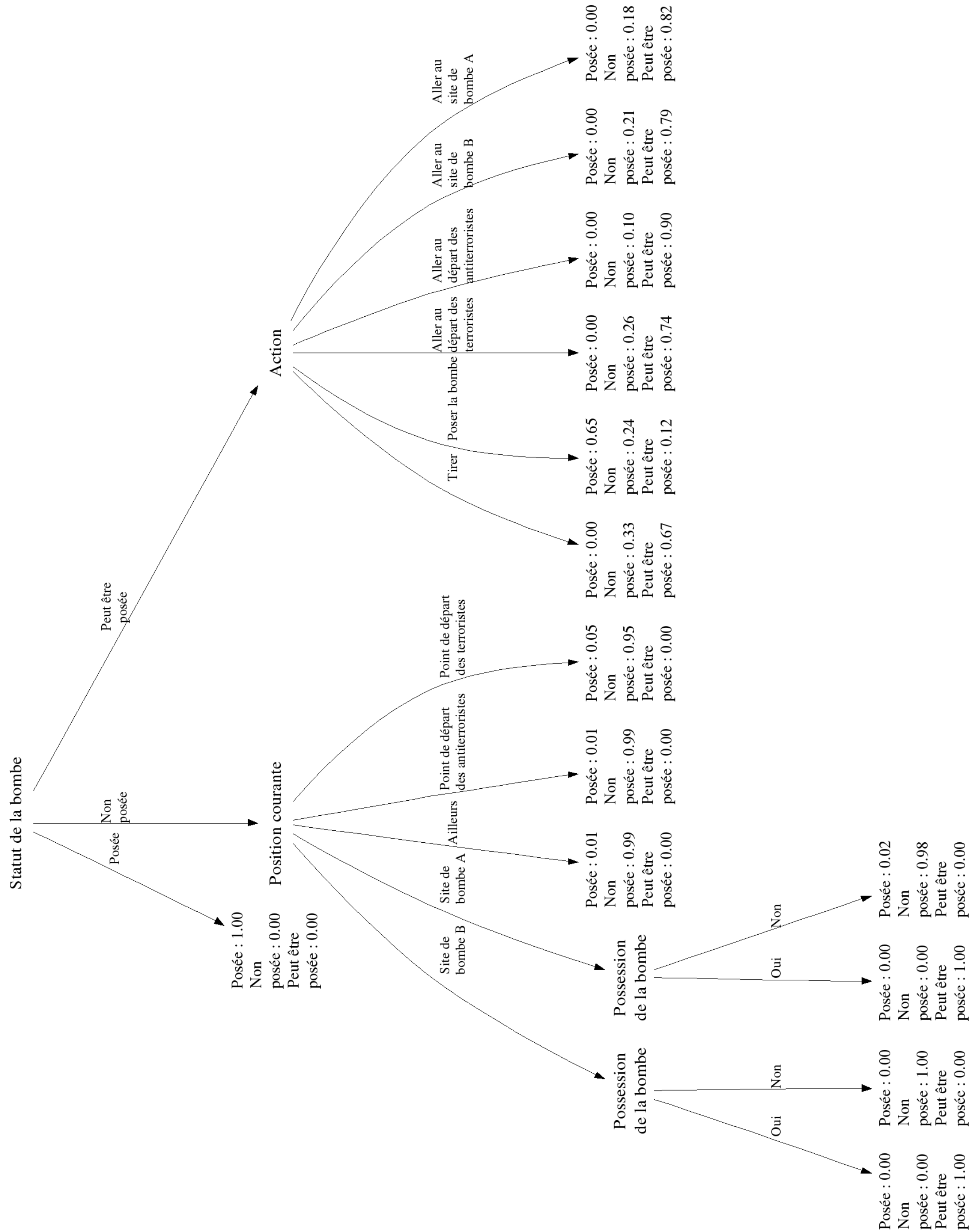


FIG. 7.6 – Résultat de l'apprentissage d'un PNJ de l'équipe des joueurs terroristes pour la distribution de probabilités conditionnelle de la variable Statut de la bombe. La feuille la plus à droite indique que l'agent estime qu'au prochain pas de temps, lorsque la bombe peut être posée (par cet agent) et que l'agent exécute l'action Aller sur le site de bombe A, il y a une probabilité nulle que la bombe soit posée, une probabilité de 0.18 que la bombe ne soit pas posée et une probabilité de 0.82 qu'elle puisse être posée (par cet agent).

Lorsque la bombe n'est pas posée et que la position courante d'un agent est, par exemple, Ailleurs, on note que la probabilité que la bombe soit posée au prochain pas de temps est non nulle. La perception Statut de la bombe représentant un état de l'environnement et non de l'agent, cette probabilité indique qu'un autre PNJ de l'équipe des terroristes a posé la bombe.

Enfin, on remarque que l'action exécutée par l'agent est seulement testée lorsque la perception Statut de la bombe de l'agent est égale à Peut être posée. Ainsi, le reste de l'arbre (correspondant aux autres valeurs de la perception) est commun à toutes les actions.

A l'instar de l'apprentissage de la fonction de récompense, ces résultats montrent que, d'une part, l'agent SDYNA a construit une représentation pertinente de la fonction de transition représentant le problème à résoudre dans le jeu et, d'autre part, que les représentations construites par l'induction d'arbres de décision sont directement intelligibles par un utilisateur.

De plus, l'arbre de décision représentant la perception Statut de la bombe illustre le gain de place réalisée lorsque la fonction de transition du FMDP représentant le problème n'est composée que d'un arbre par variable. De plus, il illustre le fait que la lecture de la représentation de la distribution de probabilités conditionnelle peut être facilitée. En effet, si un arbre par action et par variable avait été construit, la partie de l'arbre non dépendante de l'action aurait été reproduite pour toutes les actions possibles dans le problème, rendant ainsi sa lecture plus difficile.

7.4.3 Politiques gloutonnes

Une fois que l'apprentissage a défini les fonctions de transition et de récompense du FMDP représentant le problème, il est possible d'utiliser la planification pour calculer une politique pour l'agent. Les résultats présentés dans cette section illustrent des politiques gloutonnes (c'est-à-dire sélectionnant la ou les meilleures actions) calculées à partir du FMDP appris par l'agent SDYNA contrôlant les PNJs de l'équipe terroriste. Au cours de l'expérimentation, nous avons utilisé l'algorithme incrémental de planification IncSVI pour déterminer l'action exécutée par les PNJs dans le jeu. Cet algorithme ne calcule pas de représentation explicite de la politique. Afin de pouvoir les montrer dans cette section, nous avons calculé ces politiques en utilisant l'algorithme SVI avec le FMDP construit par l'apprentissage. Une fois la politique calculée, l'arbre de décision la représentant est réorganisé avec l'algorithme BuildTreeF (section 4.2.4, page 99) pour une meilleure lisibilité. Une feuille marquée "Pas de sélection" dans un arbre de décision représentant une politique signifie que toutes les actions possibles sont considérées comme équivalentes par l'agent.

La figure 7.7 représente la politique gloutonne calculée à partir des fonctions de transition et de récompense construites au cours de l'apprentissage des PNJs de l'équipe des terroristes. Cette politique a pour but de maximiser l'objectif "poser la bombe". Les récompenses associées aux deux autres objectifs, "rester en vie" et "éliminer le plus possible des joueurs de l'équipe adverse" sont

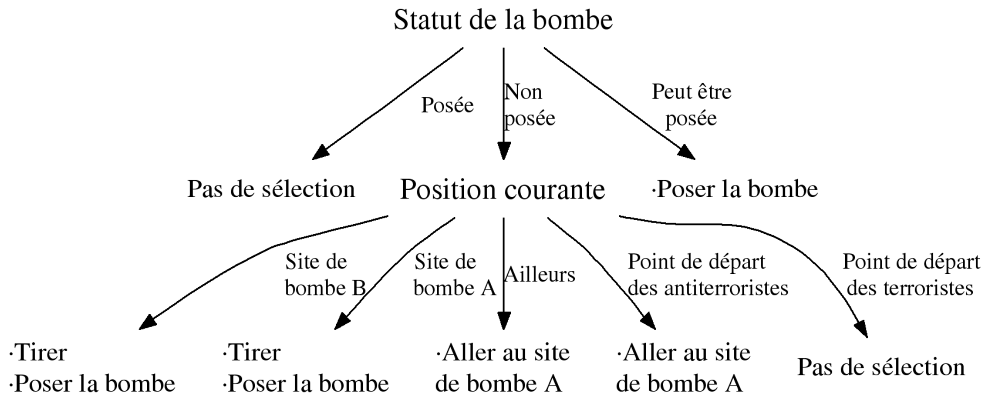


FIG. 7.7 – Résultat du calcul de la politique d'un PNJ de l'équipe des joueurs terroristes pour maximiser la récompense associée à l'objectif "poser la bombe" à partir de l'apprentissage. La feuille la plus à gauche signifie que lorsque la bombe n'est pas posée et la position courante du PNJ est le site de bombe B, alors les deux meilleures actions sont Tirer et Poser la bombe.

ignorées.

En premier lieu, on constate que lorsque le PNJ peut poser la bombe, alors la meilleure action est Poser la bombe. Une fois que la bombe a été posée, alors l'objectif a été atteint et aucune action n'est considérée comme étant meilleure qu'une autre. Lorsque la bombe n'a pas été posée, alors la meilleure action dépend de la position courante du PNJ. En effet, si le PNJ est sur un site de bombe, alors il considère que toutes les actions n'ayant aucun effet sur sa position, c'est-à-dire les actions Tirer et Poser la bombe sont équivalentes. Au contraire, lorsque le PNJ se situe Ailleurs ou au Point de départ des antiterroristes, alors la meilleure action est Aller sur le site de bombe A. Ce n'est pas le cas lorsque le PNJ est à son point de départ et où toutes les actions sont considérées comme équivalentes.

La figure 7.8 représente une autre politique gloutonne calculée à partir des fonctions de transition et de récompense construites au cours de l'apprentissage des PNJs de l'équipe des terroristes. Cette politique a pour but de maximiser l'objectif "éliminer le plus possible des joueurs de l'équipe adverse". Les récompenses associées aux deux autres objectifs, "rester en vie" et "poser la bombe" sont ignorées.

On observe que lorsque le PNJ n'a plus de munitions, alors toutes les actions sont considérées comme équivalentes. Au contraire, lorsque le PNJ a des munitions et qu'il voit une cible, alors il considère que la meilleure action est Tirer. Dans le cas où le PNJ a des munitions et qu'il ne voit pas de cible, alors la meilleure action dépend de sa position courante. Nous avons vu, lors de la description de la distribution de probabilités conditionnelle de la perception Cible en vue dans la figure 7.4 (page 186), que la probabilité de voir une cible au pas de temps suivant était plus importante lorsque la position courante du PNJ était Ailleurs.

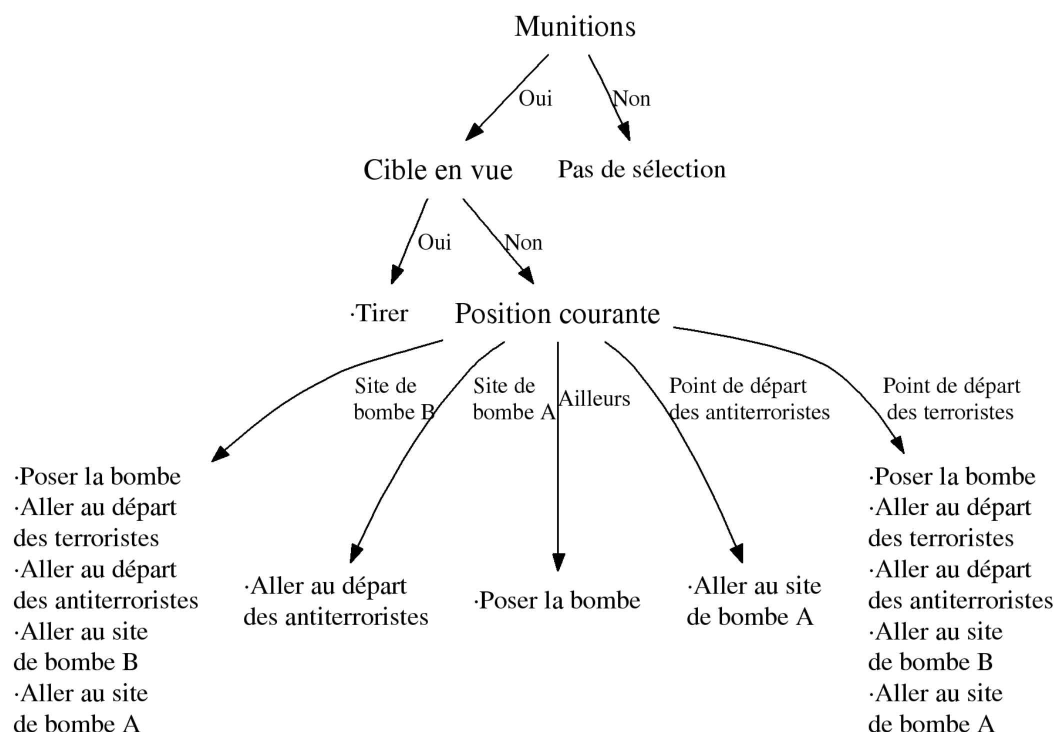


FIG. 7.8 – Résultat du calcul de la politique d'un PNJ de l'équipe des joueurs terroristes pour maximiser la récompense associée à l'objectif "éliminer le plus possible des joueurs de l'équipe adverse" à partir de l'apprentissage.

C'est la raison pour laquelle, lorsque le PNJ a des munitions mais ne voit pas de cible, la meilleure action estimée est Poser la bombe lorsque l'agent est Ailleurs, c'est-à-dire la seule action n'ayant pas d'effet ni sur la position courante de l'agent, ni sur ses munitions. Pour les lieux Site de bombe B et Point de départ des terroristes, toutes les actions sont considérées équivalentes, sauf l'action Tirer qui a un effet sur les munitions. Enfin, pour les lieux Point de départ des antiterroristes et Site de bombe A, la probabilité de voir une cible au prochain pas de temps n'est pas suffisante et la meilleure action correspond à la probabilité la plus élevée d'être Ailleurs au prochain pas de temps.

La figure 7.9 représente la politique gloutonne calculée à partir des fonctions de transition et en considérant l'ensemble des fonctions de récompense construites au cours de l'apprentissage des PNJs de l'équipe des terroristes. Cette politique a pour but de maximiser la somme des récompenses associées à chacun des objectifs du PNJ dans le jeu.

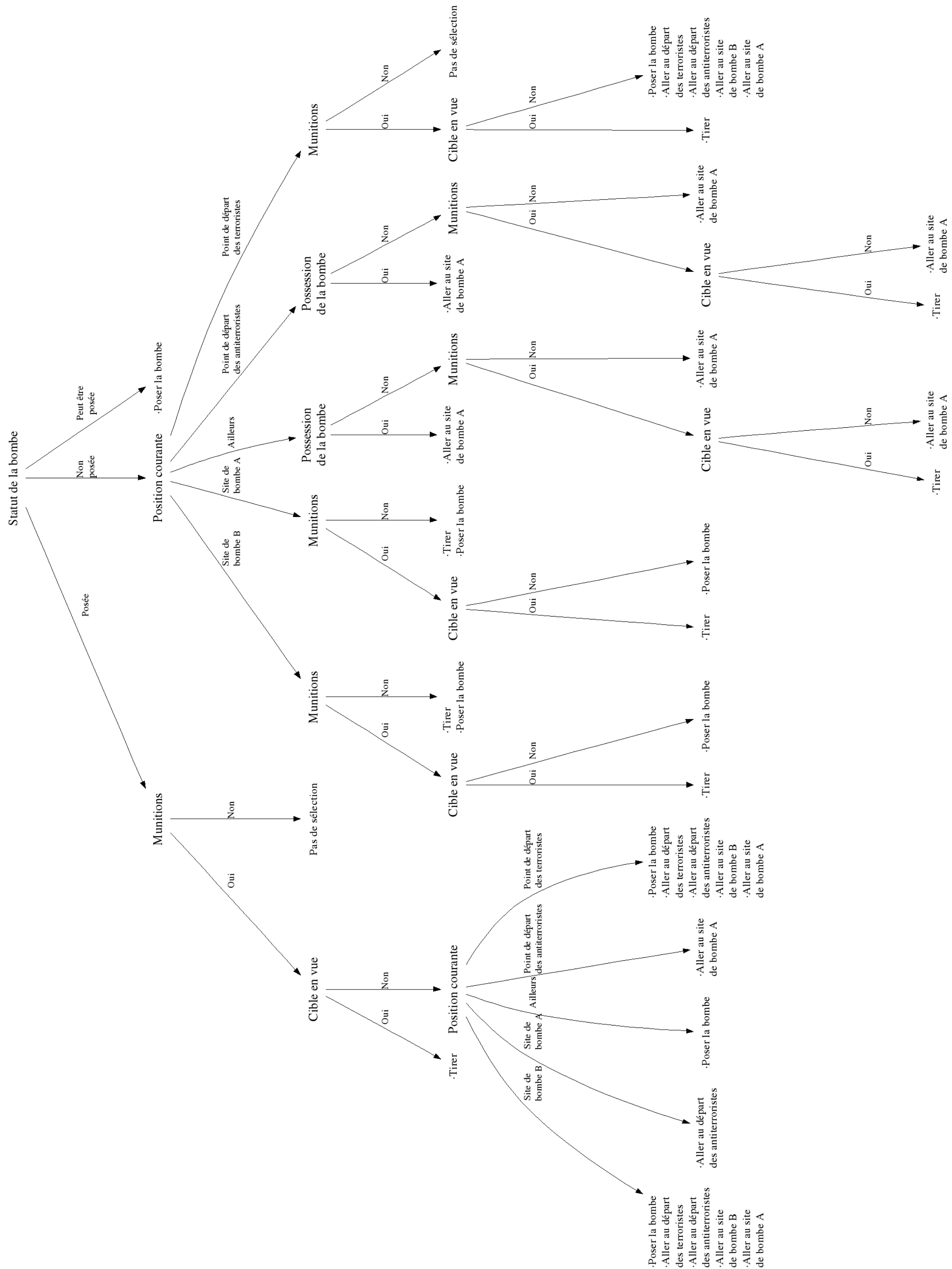


FIG. 7.9 – Résultat du calcul de la politique d'un PNJ de l'équipe des joueurs terroristes pour maximiser la somme de l'ensemble des récompenses associées à l'ensemble des objectifs du PNJ.

On remarque que l'on retrouve les mêmes meilleures actions que celles observées dans les politiques précédentes. Un premier exemple est lorsque le PNJ peut poser la bombe, alors la meilleure action est Poser la bombe. Un deuxième exemple est le fait que l'on retrouve à plusieurs endroits dans l'arbre le fait que lorsque le PNJ a une cible et qu'il a des munitions, alors la meilleure action sélectionnée est Tirer. On peut remarquer que la meilleure action considérée par l'agent est Aller sur le site de bombe A lorsque la bombe n'est pas posée, que le PNJ est Ailleurs et qu'il est en possession de la bombe. La perception Cible en vue n'étant pas testée, l'agent préfère donc aller poser la bombe plutôt que tirer sur un adversaire s'il en voyait un.

Ces résultats montrent que, d'une part, les algorithmes de planification permettent de construire des représentations intelligibles des politiques gloutonnes de l'agent, et d'autre part que l'apprentissage s'est révélé suffisamment pertinent pour que la politique de l'agent soit adaptée au problème que celui-ci doit résoudre.

Lorsque l'apprentissage est incomplet, alors l'agent n'est pas capable de discriminer une action par rapport à une autre, comme c'est le cas pour la politique maximisant l'objectif "poser la bombe" et que l'agent est au point de départ des terroristes (voir figure 7.7). Dans ce cas, les actions seront choisies de façon aléatoire, constituant ainsi de nouveaux exemples utilisés par l'apprentissage.

Enfin, nous pouvons remarquer qu'une représentation sous la forme d'arbre de décision, en plus de la lisibilité, permet de faire des économies importantes concernant les temps de calcul requis pour décider de la prochaine action du PNJ (lorsque l'apprentissage est arrêté). En effet, par exemple, lorsque le PNJ peut poser la bombe, le statut de la bombe est la seule perception à avoir été évaluée pour déterminer l'action à réaliser par l'agent, il n'est pas nécessaire de déterminer la valeur des autres perceptions.

7.5 Synthèse

Ce chapitre avait pour but d'illustrer la mise en œuvre de SDYNA sur un problème réel. Pour cela, nous avons choisi de tester l'apprentissage de SDYNA sur le problème consistant à contrôler un PNJ dans le jeu vidéo Counter-Strike©. Dans un premier temps, nous avons expliqué le déroulement du jeu. Dans un deuxième temps, nous avons formalisé le problème sous la forme d'un petit problème d'apprentissage par renforcement. Ensuite, nous avons décrit l'instance SDYNA que nous avons utilisé pour résoudre ce problème et valider celle-ci au cours d'une expérience d'environ 2h00 (temps réel et virtuel) dans le jeu vidéo.

Nous avons montré que, pour ce problème, l'instance de SDYNA était capable d'apprendre une représentation pertinente du problème sous la forme d'un FMDP, celui-ci étant défini par les fonctions de récompense et de transition. De plus, nous avons montré que ces représentations pouvaient

facilement être interprétées par un utilisateur, facilitant ainsi la compréhension du problème et la compréhension du comportement de l'agent.

De plus, nous avons aussi montré que le FMDP construit par l'apprentissage permettait, par l'utilisation d'un algorithme de planification, de construire une représentation explicite de la politique gloutonne de l'agent. Cette politique permet de représenter la ou les actions considérées comme étant meilleures par l'agent, résumant ainsi le comportement d'un PNJ dans le jeu. Nous avons constaté que, d'une part, SDYNA permettait de construire des politiques adaptées au problème posé dans le jeu vidéo, d'autre part, que ces représentations, à l'instar des fonctions de récompense et de transition, étaient intelligibles et, enfin, qu'elles permettaient de faire des économies de calcul en évitant l'évaluation de perceptions dans certains contextes.

Ces travaux proposent donc un exemple de l'application des FMDPs dans un jeu vidéo, démontrant ainsi l'applicabilité de ceux-ci dans des problèmes réels. Nous connaissons peu de travaux ayant testés les FMDPs sur des applications réelles. Nous pouvons notamment citer les travaux proposés par Forsell and Sabbadin (2006) qui utilisent une nouvelle représentation, appelée Processus Décisionnel de Markov multidimensionnels sur Graphe (PDMG) permettant de représenter les indépendances relatives aux fonctions. En s'appuyant sur cette représentation, ils proposent un algorithme, basé sur la programmation linéaire, permettant de résoudre le problème de façon approchée. Leurs travaux sont appliqués à la gestion de parcelles forestières afin de minimiser les dégâts occasionnés par une tempête. Dans un autre domaine, Guestrin et al. (2003a) utilise les MDPs relationnels, *Relational* MDPs (RMDPs), pour résoudre un problème de planification dans le jeu de stratégie temps réel Stratagus[5]. Contrairement à notre approche, ces travaux supposent une connaissance complète a priori de la structure du problème. Concernant l'apprentissage dans les jeux vidéo d'une façon générale, nous invitons le lecteur intéressé à consulter les travaux présentés par Robert (2005).

[5] http://www.stratagus.org/

Chapitre 8

Discussion

Les travaux présentés dans cette thèse combinent des techniques d'apprentissage supervisé avec des techniques de planification pour la résolution de problèmes d'apprentissage par renforcement. Ainsi, la discussion proposée dans ce chapitre est organisée d'après les trois problématiques abordées dans cette thèse : l'apprentissage supervisé des fonctions de transition et de récompense d'un FMDP est discuté section 8.1, la planification pour la construction des fonctions de valeur du FMDP est discutée section 8.2 et l'apprentissage par renforcement dans les FMDPs est discuté section 8.3. Lors du déroulement de la thèse, dans le cadre de SDYNA, nous avons essayé d'étudier ces trois problématiques plutôt que de nous concentrer que sur l'une d'entre elles. C'est la raison pour laquelle, pour chacune des sections, nous commencerons par commenter les contributions que nous avons présentées puis nous énumérerons les limitations existantes dans ces solutions, en indiquant des pistes de recherche lorsque cela est possible.

8.1 Apprentissage supervisé d'un FMDP

L'approche proposée dans cette thèse pour la résolution de problèmes d'apprentissage par renforcement repose sur l'utilisation de techniques d'apprentissage supervisé pour construire un modèle du problème sous la forme d'un FMDP, et sur l'utilisation de ce modèle par la planification pour trouver une solution au problème d'apprentissage par renforcement. Bien que plusieurs travaux récents aient été proposés pour l'apprentissage des paramètres du FMDP (voir section 6.1.5), l'approche décrite dans ce manuscrit est, à notre connaissance, la première à proposer l'apprentissage de la structure. Par ces travaux, nous pensons avoir apporté plusieurs contributions, décrites section 8.1.1. Cependant, nous pensons que cette approche n'est pas encore complètement aboutie et qu'elle souffre de plusieurs limitations que nous décrirons lors de la section 8.1.2.

8.1.1 Contributions

Utilisation de techniques d'apprentissage supervisé

Notre première contribution est de proposer une méthode pour l'apprentissage, hors-ligne ou en ligne, d'un FMDP. Pour cela, nous avons utilisé deux méthodes de décomposition des observations d'un agent en exemples utilisables par des algorithmes d'apprentissage supervisé. Nos algorithmes présentent l'avantage de nécessiter peu d'adaptations des techniques existantes, plus particulièrement l'induction d'arbres de décision. On peut donc parfaitement envisager l'utilisation d'autres techniques ayant été développées dans ce domaine de recherche, comme l'utilisation d'autres mesures d'information telles que *gain ratio* (Quinlan, 1993) ou le critère de la moindre déviation absolue (Torgo, 2000).

Plus généralement, d'autres méthodes d'apprentissage peuvent être considérées. Cependant, ces méthodes doivent permettre d'extraire la structure du problème, ce qui n'est pas le cas de toutes les méthodes d'apprentissage supervisé. Plus précisément, un algorithme d'apprentissage d'une distribution de probabilités conditionnelle ou d'une fonction de récompense doit être capable de définir pour ces fonctions les indépendances relatives à la fonction apprise et, lorsque des algorithmes de planification dans les FMDPs tels que SVI ou SPUDD sont utilisés, les indépendances relatives aux contextes.

Apprentissage de distribution de probabilités conditionnelle

L'une des principales difficultés dans l'apprentissage d'un FMDP est, d'une part, l'apprentissage de la structure des DBNs associés à chaque variable (et chaque action suivant la représentation utilisée), d'autre part quantifier ces DBNs par une représentation structurée. Une de nos contributions est donc d'avoir montré que cet apprentissage peut s'effectuer de façon simultanée, sans représentation explicite des DBNs, par l'induction d'arbres de décision avec l'utilisation d'un élagage, afin d'éviter le développement superflu de branches lorsque les transitions sont stochastiques.

Dans le cadre de l'apprentissage d'un DBN, cette technique est une heuristique permettant de limiter les tests statistiques à effectuer pour différencier les distributions de probabilités. Ainsi, seules les dépendances éventuelles sur une variable sont tout d'abord testées à la racine d'un arbre. C'est uniquement lorsqu'un nœud de décision est installé que les tests statistiques pour une dépendance supplémentaire sont réalisés. Par conséquent, l'induction d'arbres de décision utilisant un test statistique pour le pré-élagage permet non seulement d'identifier la structure d'un problème mais utilise aussi cette structure pour limiter la complexité des calculs nécessaires à l'apprentissage.

Similarités entre l'apprentissage hors-ligne et en ligne

La méthode d'apprentissage hors-ligne d'un FMDP que nous avons proposée est basée sur l'induction d'arbres de décision. Par ailleurs, plusieurs recherches ont été effectuées concernant l'induction d'arbres de décision afin d'apprendre en ligne une fonction à partir d'un flux d'exemples. Nous avons montré que ces approches pouvaient aussi être réutilisées pour l'apprentissage en ligne d'un FMDP.

De plus, les deux types d'apprentissage hors-ligne et en ligne étant similaires, la plupart des études que nous avons présentées pour l'apprentissage hors-ligne concernent directement l'apprentissage en ligne. Par exemple, les résultats sur l'étude du paramètre déterminant si la différence entre deux distributions de probabilités est significative sont les mêmes dans les cadres de l'apprentissage en ligne et de l'apprentissage hors-ligne.

Nous avons exploité cette propriété à plusieurs reprises, par exemple lors de la mise au point de SDYNA dans le jeu Counter-Strike©. En effet, les expériences dans le jeu Counter-Strike© pouvaient s'avérer longues puisque nous n'avions pas de moyen d'accélérer le jeu. Or, lors d'une expérience, nous enregistrions la trajectoire d'un agent (sous la forme d'une suite d'observation). Hors-ligne, cette trajectoire nous permettait ensuite de tester des paramètres de l'apprentissage du FMDP très rapidement, sans avoir à relancer une nouvelle expérience dans le jeu. Une fois les paramètres ajustés, une nouvelle expérience dans le jeu était lancée, économisant ainsi plusieurs heures d'expérimentations.

Enfin, en plus d'être suffisamment similaires pour utiliser les mêmes paramètres, à la fois l'apprentissage hors-ligne et en ligne sont robustes à une grande variété de problèmes. En effet, par exemple, concernant l'apprentissage de la fonction de transition, dans tous les problèmes traités dans ce manuscrit, y compris le problème réel dans Counter-Strike©, nous avons pu utiliser la même famille d'algorithmes d'induction d'arbres de décision en utilisant exactement la même valeur de paramètre dans le cadre du pré-élagage.

8.1.2 Limitations

Existe-t-il une preuve de convergence ?

La réponse est clairement non : le fait d'utiliser pour le pré-élagage un test statistique comparant des distributions de probabilité sur une seule variable empêche l'induction d'arbres de décisions de pouvoir apprendre certaines fonctions. Un exemple typique est le cas d'une fonction de type "ou exclusif", comme le montre la figure 8.1. Si, pour séparer deux distributions de probabilités pour des contextes différents, il est nécessaire de tester plusieurs variables, alors il est possible que le pré-élagage empêche l'installation des nœuds de décision nécessaires puisque le test statistique réalisé par celui-ci est "myope".

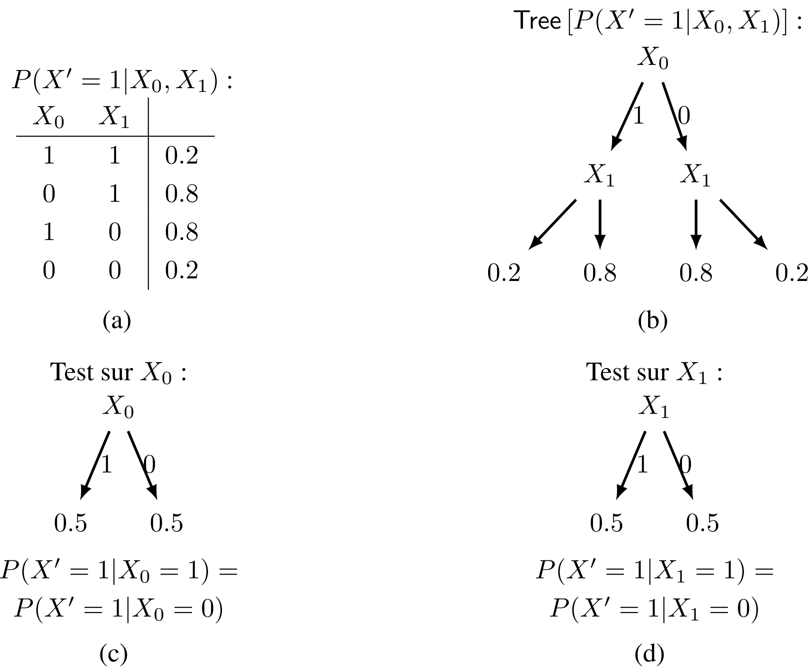


FIG. 8.1 – Problème de l'apprentissage d'une fonction "ou exclusif" définie sous la forme tabulaire dans la figure (a) et sous la forme d'un arbre de décision figure (b). Le pré-élagage empêche l'installation d'un test sur les variables X_0 ou X_1 au nœud de décision à la racine puisque le test sur une variable sépare des distributions de probabilités $P(X' = 1|X_0, X_1)$ identiques, comme le montre les figures (c) et (d).

Pour reprendre l'exemple de la fonction "ou exclusif" définie figure 8.1(a), il est nécessaire de tester les deux variables X_0 et X_1 comme le montre la figure 8.1(b). Pour définir la variable X_i testée pour le premier nœud de décision de l'arbre, le test statistique utilisé pour le pré-élagage compare les distributions de probabilités $P(X|X_i = x_i)$ avec $\forall i \; x_i \in \underline{\text{Dom}}(X_i)$ avec la distribution de probabilités $P(X)$. Dans le cas de notre exemple, ce sont les distributions de probabilités $P(X'|X_1 = 1)$ et $P(X'|X_1 = 0)$ d'une part, $P(X'|X_0 = 1)$ et $P(X'|X_0 = 0)$ d'autre part, qui sont comparées à la distribution de probabilités $P(X)$, illustré par les figures 8.1(c) et 8.1(d). La nature de la fonction "ou exclusif" fait que ces distributions de probabilités sont identiques. Par conséquent, le premier nœud de décision de l'arbre n'est pas installé, empêchant ainsi l'installation des nœuds de décision testant la deuxième variable et nécessaires à la discrimination des différentes distributions de probabilités.

Il est important de noter que le pré-élagage n'est pas nécessaire pour l'apprentissage d'une fonction de transition complètement déterministe. Dans un tel cas, le critère permettant de continuer à développer l'arbre est l'existence de feuilles non pures (c'est-à-dire contenant plusieurs valeurs différentes) et non un test statistique. C'est la raison pour laquelle, bien que la mesure d'information ne puisse pas déterminer le meilleur test à installer, l'arbre pourra se développer. Cependant, des tests sur des variables non corrélées avec les distributions de probabilité seront probablement

installées.

Lorsque le problème est stochastique, alors plusieurs pistes peuvent être envisagées. En effet, il pourrait être envisageable d'utiliser des techniques classiques (Bauer and Kohavi, 1999) concernant l'induction d'arbres de décision, notamment le BAGGING (Breiman, 1996) et le BOOSTING (Freund and Schapire, 1997). Plusieurs développements très intéressants concernant l'intégration de ces algorithmes dans le cadre d'un apprentissage incrémental ont été proposés par Fern and Givan (2003) qui, de plus, décrivent plusieurs améliorations importantes de l'algorithme incrémental d'induction d'arbres de décision, telles que le post-élagage. Ces améliorations sont directement applicable dans le cadre de SDYNA. Enfin, plusieurs travaux concernant l'approche PAC dans le cadre de l'induction d'arbres de décisions ont été proposés (Auer et al., 1995; Decatur, 1997; Pichuka et al., 2007) et pourraient être le point de départ de nouvelles recherches.

Apprentissage des arcs synchrones

Dans les travaux concernant l'apprentissage des FMDPs, nous n'avons pas considéré l'apprentissage des arcs synchrones dans les DBNs, c'est-à-dire que nous avons supposé que toutes les variables au temps $t + 1$ étaient toutes indépendantes les unes des autres. Or, certaines actions d'un problème peuvent avoir un effet qui est corrélé sur plusieurs variables pour un même pas de temps. Par exemple, pour l'action $\mathcal{D}$elC (le robot donne le café à sa propriétaire) du problème *Coffee Robot*, la variable $\mathcal{HOC}$, indiquant si la propriétaire a un café, peut directement être corrélée avec la variable $\mathcal{HRC}$ indiquant si le robot a un café. Concernant la planification, des méthodes existent pour calculer la solution à des problèmes avec de telles corrélations (Boutilier, 1997), notamment pour les algorithmes SPI et SVI (Boutilier et al., 2000) et l'approche basée sur la programmation linéaire proposée par Guestrin et al. (2003b).

Une approche possible afin d'apprendre la structure des DBNs pourvus d'arcs synchrones est d'ajouter à l'ensemble des attributs des exemples appris par les algorithmes d'induction d'arbres de décision la valeur des autres variables au temps $t + 1$. Ainsi, pour l'apprentissage des distributions de probabilités conditionnelles d'une variable X_i, plutôt que de former des exemples de type $\langle \mathfrak{a} = \{x_1, \ldots, x_n\}, \varsigma = x'_i \rangle$, nous pouvons envisager l'utilisation d'exemples de type $\langle \mathfrak{a} = \{x_1, \ldots, x_n\} \cup \{x'_j | \forall X_j \in X \text{ et } j \neq i\}, \varsigma = x'_i \rangle$. Dans ce cas, l'utilisation d'une heuristique telle que préférer les variables au temps t plutôt que $t + 1$ pourrait être nécessaire. Des recherches supplémentaires sont donc nécessaires pour vérifier la validité d'un tel apprentissage.

Branches multi-valuées

Les arbres de décision construits lors de l'apprentissage d'un FMDP sont composés de nœuds de décision, de branches et de feuilles. À partir d'un nœud de décision, une branche par valeur de la variable testée est installée. Cette représentation concerne les algorithmes d'induction d'arbres de

décision aussi bien en ligne que hors-ligne, tels que nous les avons présentés dans les chapitres 4 et 5. Cette représentation peut entraîner un surcoût important concernant l'apprentissage de fonctions qui utilisent des variables pouvant prendre de nombreuses valeurs, c'est-à-dire possédant un domaine dont le cardinal est important. C'est le cas, par exemple, pour le problème *Ring*.

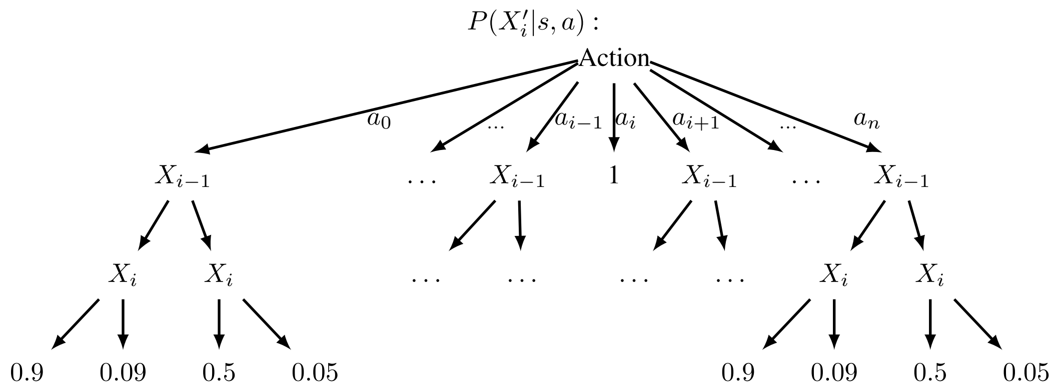


FIG. 8.2 – Utilisation d'un arbre de décision avec une valeur par branche pour un nœud de décision : pour certaines fonctions, dans ce cas la distribution de probabilités conditionnelle $P(X'_i|s,a)$ dans le problème *Ring*, une telle représentation nécessite un temps d'apprentissage long et une place importante en mémoire.

Lorsque l'action est considérée comme un attribut, le problème ayant 41 actions, un nœud de décision possédant 41 branches est installé dans les distributions de probabilités conditionnelles de chacune des 40 variables, comme le montre la figure 8.2. Ainsi, dans ce problème et avec une telle représentation, il est nécessaire d'apprendre 40 fois la même distribution de probabilités, identique pour toutes les actions ne consistant pas à redémarrer la machine i. La séparation de ces distributions de probabilités identiques a deux conséquences :

- un surcoût concernant la mémoire nécessaire pour représenter la fonction : la même structure de données est répété de façon inutile pour chaque branche du nœud de décision ;
- un surcoût concernant l'apprentissage : plus d'exemples sont nécessaires puisque, lorsqu'un exemple est appris, il ne met à jour que les données correspondant au contexte de celui-ci ; par exemple, lorsque l'agent choisit une action a_j tel que $i \neq j$, alors l'exemple mettra seulement à jour la distribution de probabilités correspondant à $P(X_i|a_j)$.

Nous pensons donc qu'il est possible de construire une structure de données exploitant de telles régularités.

En effet, comme le montre par exemple la figure 8.3, la même distribution de probabilités conditionnelle $P(X'_i|s,a)$ peut être représentée avec un arbre de décision dont les branches contiennent une ou plusieurs valeurs de la variable testée au nœud de décision. De cette façon, deux économies substantielles sont réalisées :

- une économie concernant la mémoire nécessaire pour représenter la fonction : la même structure de données est partagée par plusieurs contextes exploitant ainsi certaines régularités et

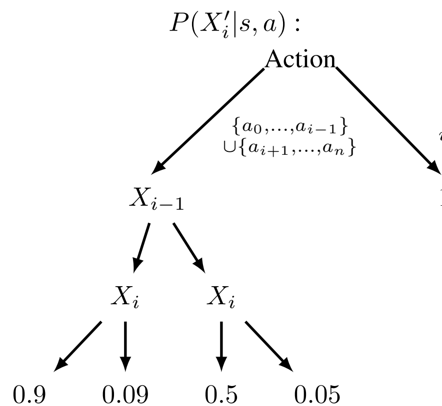


FIG. 8.3 – Utilisation d'un arbre de décision avec plusieurs valeurs par branche pour un nœud de décision : pour certaines fonctions, dans ce cas la distribution de probabilités conditionnelle $P(X_i'|s,a)$ dans le problème *Ring*, une telle représentation peut économiser un temps d'apprentissage important et une place importante en mémoire.

évitant une redondance de l'information inutile ;

– une économie concernant l'apprentissage : un exemple peut mettre à jour des probabilités dont *les contextes ne sont pas consistants* avec celui-ci, pouvant ainsi accélérer l'estimation de la distribution ; par exemple, lorsque l'agent choisit une action a_j tel que $i \neq j$, alors l'exemple mettra à jour la probabilité concernant l'action a_j mais aussi toutes les actions $a_{j'}$ tel que $i \neq j'$ de l'agent pour la distribution $P(X_i'|s,a)$.

Au niveau de l'apprentissage, la principale modification à effectuer concerne l'installation des nœuds de décision. Plutôt que de seulement choisir une variable à tester, il est nécessaire de définir quels sont les différents ensembles de valeurs correspondant aux branches du nœud de décision. Par conséquent, cette agrégation supplémentaire a un coût de calcul.

Une telle représentation n'ajouterait aucune difficulté théorique pour être utilisée avec les algorithmes de planification dans les FMDPs puisqu'il est facile de développer un arbre pour revenir à une représentation correspondant à une valeur par branche. Cependant, il serait sûrement intéressant de faire en sorte que cette agrégation puisse être exploitée lors de la planification. Dans le cadre de SPUDD, nous rappelons que l'algorithme ne manipule que des variables binaires. Les variables non binaires sont donc converties en variables binaires : cette représentation pourrait éventuellement guider cette conversion.

Sélection multi-critères des variables

Lorsqu'une variable est installée à un nœud de décision, le seul critère utilisé est celui donné par la mesure d'information utilisée par l'algorithme d'induction d'arbres de décision. L'algorithme UTREE d'apprentissage par renforcement proposé par McCallum (1995) utilise une mesure d'infor-

mation pour installer les tests directement dans les fonctions de valeur calculées pour le problème.

Deux objectifs différents sont donc visés : dans le cadre de l'apprentissage d'un FMDP, on recherche la variable discriminant avec le plus de pertinence une distribution de probabilités conditionnelle dans la fonction de transition, dans le cadre de l'algorithme UTREE, on recherche la variable discriminant avec le plus de pertinence la valeur de la fonction de valeur. Ainsi, le premier recherche une fonction de transition compacte, alors que le deuxième recherche une fonction de valeur compacte.

À partir de ces deux méthodes, une solution intermédiaire pourrait être envisagée dans le cadre de SDYNA. En effet, lorsqu'on représente des fonctions de valeur sous la forme d'arbres de décision, il est possible d'utiliser une mesure d'information (par exemple, la mesure des moindres carrés présentée section 4.1.3, page 86) pour les valeurs réelles afin d'évaluer l'importance d'une variable dans la représentation d'une fonction de valeur. Cette étape serait similaire à celle réalisée pour le nœud racine de l'arbre construit par l'algorithme de réorganisation BuildTreeF présenté dans la section 4.2.4. Une fois chaque variable utilisée dans la fonction de valeur qualifiée, cette mesure d'information peut devenir un critère additionnel lors de la sélection d'une variable dans une distribution de probabilités de la fonction de transition.

Ainsi, dans des problèmes contenant de nombreuses variables dont certaines peuvent être inutiles pour la résolution du problème, une telle technique itérative pourrait éviter à l'algorithme d'apprentissage d'un FMDP de rajouter des dépendances relatives aux fonctions impliquant l'ajout de nombreuses variables dans le calcul de la fonction de valeur, alors que les variables utilisées pour les autres dépendances suffisent à la résolution du problème.

8.2 Planification dans les FMDPs

Dans cette thèse, nous avons proposé plusieurs algorithmes permettant l'apprentissage d'un FMDP hors-ligne ou de façon incrémentale. Le FMDP ainsi construit est ensuite utilisé par des techniques de planification afin de calculer une solution au problème d'apprentissage par renforcement à résoudre. Par ces travaux, nous pensons donc avoir apporté des contributions, présentées section 8.2.1, sur deux thèmes : la relation entre l'apprentissage et la planification d'une part, l'adaptation des méthodes de planification au cadre incrémental d'autre part. Puis, section 8.2.2, nous soulignerons un certain nombre de limites de l'utilisation des méthodes de planification dans les FMDPs avec l'apprentissage, plus particulièrement dans le cadre de l'apprentissage incrémental de SDYNA.

8.2.1 Contributions

Approche validée avec toutes les méthodes de planification

La question ayant initié les travaux présentés dans ce manuscrit était de savoir s'il était possible d'utiliser les algorithmes de planification dans les FMDPS, donc adaptés aux grands problèmes, dans des problèmes dont la structure était inconnue. Une fois nos algorithmes d'apprentissage développés, il nous a donc semblé important de montrer de façon expérimentale que cet apprentissage pouvait être utilisé avec les principaux algorithmes de planification dans les FMDPS, c'est-à-dire SPI, SVI, SPUDD et l'approche basée sur la programmation linéaire.

C'est la raison pour laquelle nous pensons qu'une contribution très importante de notre travail est de montrer que l'ensemble des algorithmes de planification dans les FMDPS peuvent être utilisés lorsque la structure du problème, c'est-à-dire les indépendances relatives aux fonctions et les indépendances relatives aux contextes, est inconnue. Plus précisément, nous avons montré que les méthodes de planification dans les FMDPS étaient suffisamment générales, c'est-à-dire nécessitant peu de connaissances a priori, pour pouvoir être utilisées sur des problèmes dont la représentation est partielle et/ou se construit au fur et à mesure. Nos travaux ont donc utilisé cette robustesse pour élargir le domaine d'application de ces méthodes à tous les problèmes d'apprentissage par renforcement discrets.

La généralisation de l'apprentissage est exploitée par la planification

La propriété de généralisation des algorithmes d'induction d'arbres de décision est directement utilisée par l'algorithme d'apprentissage des FMDPS : pour certaines transitions, il permet de prédire correctement les distributions de probabilités conditionnelles pour des variables du problème alors que toutes les transitions n'ont pas été visitées, de même pour la fonction de récompense. Une contribution importante de nos travaux est d'avoir montré de façon empirique que la généralisation de l'apprentissage pour les fonctions de transition et de récompense du problème permettait, d'une part, à l'algorithme de planification de calculer une action pertinente pour des états qui n'auraient pas été visités, d'autre part d'agréger les états dont les valeurs d'action sont similaires.

Une telle propriété est fondamentale pour la résolution de grands problèmes d'apprentissage par renforcement. Premièrement, l'association des algorithmes d'induction d'arbres de décision avec la planification permet de généraliser l'expérience de l'agent à de nouveaux états non visités, évitant ainsi une exploration exhaustive de l'environnement impossible dans les grands problèmes. Deuxièmement, l'agrégation des états réalisés par la planification permet de représenter par de façon structurée des solutions aux problèmes, évitant ainsi une énumération exhaustive des couples état/action impossibles dans les grands problèmes. Par conséquent, l'efficacité des méthodes d'apprentissage d'un FMDP que nous avons proposées dépend principalement de la structure du problème plutôt que de sa taille. C'est la raison pour laquelle nous pensons que notre approche, d'une façon générale,

est adaptée pour la résolution de grands problèmes d'apprentissage par renforcement.

Planification incrémentale

Une autre de nos contributions est l'utilisation de la planification dans les FMDPs pour la résolution de problèmes d'apprentissage par renforcement et leur intégration au sein de l'architecture SDYNA, c'est-à-dire dans le cadre d'un apprentissage incrémental. Nous avons notamment montré que les algorithmes SVI, SPUDD et l'approche basée sur la programmation linéaire étaient directement utilisables dans le cadre d'un apprentissage incrémental.

De plus, nous avons mis en évidence le fait que des techniques classiques, telles qu'une initialisation optimiste des valeurs d'action inconnues, étaient facilement utilisables dans le cadre de SDYNA.

Enfin, il nous semble important de souligner que, dans le cadre de la planification basée sur la programmation dynamique, la propriété de généralisation de l'apprentissage, utilisée par la planification, est exploitée, même lorsque, entre deux pas de temps, une seule itération est effectuée. D'une façon générale, utiliser une technique de planification dans le cadre d'un apprentissage incrémental pose toujours la question du temps attribué aux calculs pour la mise à jour des fonctions de valeur d'action entre deux pas de temps. Nous discutons de cela dans la section suivante.

8.2.2 Limitations

Temps de calcul

Pour les algorithmes de planification basés sur la programmation dynamique et mis en œuvre dans le cadre de SDYNA, nous avons proposé les algorithmes IncSVI et IncSPUDD. Dans ces algorithmes, nous avons proposé de faire une seule itération de l'algorithme *Value Iteration*. Ainsi, les fonctions de valeur d'action et la fonction de valeur sont mises à jour pour l'ensemble des états existant dans le problème.

Plusieurs algorithmes existant dans la littérature proposent un compromis différent entre le temps de calcul et le nombre de mises à jour des fonctions de valeur. Nous pouvons notamment citer les algorithmes de programmation dynamique temps réel (*Real-time Dynamic Programming*) proposé par Barto et al. (1995), ou bien, dans le cadre l'approche DYNA, l'algorithme *Prioritized Sweeping* proposé simultanément et de façon indépendante par Peng and Williams (1993) et Moore et al. (1993).

Intégrer de telles méthodes au sein de SDYNA pourrait se révéler intéressant. En effet, la combinaison de la généralisation de l'apprentissage et l'agrégation de la planification permettrait de mettre à jour un grand nombre d'état simultanément, sans nécessiter une mise à jour exhaustive de l'espace d'état, comme c'est le cas actuellement avec les algorithmes IncSVI et IncSPUDD.

Des résultats préliminaires non présentés dans cette thèse laissent supposer que le gain en temps de calcul serait très important, alors que la perte concernant la récompense actualisée obtenue serait faible, constituant ainsi un meilleur compromis concernant le temps de calcul pour chaque pas de temps et le nombre d'états mis à jour.

Concernant l'approche basée sur la programmation linéaire, notre adaptation des algorithmes proposés par Guestrin et al. (2003b) dans le cadre de SDYNA est plutôt primitive. En effet, la méthode que nous avons proposée consiste à mettre à jour la fonction de valeur de temps en temps, en réutilisant la solution calculée à la dernière mise à jour, si possible. Le fait d'utiliser un programme linéaire complique une telle adaptation. En effet, il est difficile de prévoir les effets des modifications dans le modèle lors de l'apprentissage sur les contraintes du programme linéaire et de garantir que la solution calculée lors d'une résolution précédente satisfait le nouvel ensemble de contraintes.

Décomposition additive

Afin de pouvoir exploiter la décomposition additive d'un problème d'apprentissage par renforcement, nous avons supposé deux hypothèses différentes. La première concerne la décomposition de la récompense que l'agent reçoit en un ensemble de récompenses. La deuxième hypothèse suppose que les fonctions de base composant l'approximation linéaire de la fonction de valeur du problème soient connues.

Alors que la première hypothèse nous semble réaliste et applicable dans un problème d'apprentissage par renforcement, la deuxième suppose une connaissance a priori du problème très importante et peut ne pas être adaptée à des problèmes réels.

Des approches ont déjà été proposée pour apprendre des représentations non structurées des fonctions de base (Patrascu et al., 2002; Kveton and Hauskrecht, 2006a). De plus, comme cela est suggéré par Poupart et al. (2002) et Guestrin (2003), il est possible d'envisager la construction d'une représentation structurée des fonctions de base, notamment par l'utilisation des algorithmes de planification dans les FMDPs basés sur la programmation dynamique tels que SPI ou SPUDD. En effet, ces algorithmes calculent de façon complètement automatique la structure des fonctions de valeur. Il doit donc être possible de partir de la structure des fonctions de valeur ainsi calculées pour définir un ensemble de fonctions de base exploitant les indépendances relatives aux contextes et permettant une bonne approximation de la fonction de valeur optimale du problème. Une version préliminaire d'un tel algorithme, non présentée dans cette thèse, a été utilisée pour déterminer les fonctions de base dans le problème *Ring* (figure 4.30, page 124).

8.3 Apprentissage par renforcement dans les FMDPs

8.3.1 Contributions

Nous pensons que SDYNA apporte une contribution importante pour la résolution de grands problèmes d'apprentissage par renforcement stochastiques discrets. En effet, il permet d'un côté, d'exploiter des algorithmes d'apprentissage supervisé pour représenter le problème à résoudre sous la forme d'un FMDP et, de l'autre côté, d'exploiter des algorithmes de planification adaptées à la résolution de grands problèmes. Nous avons montré sur plusieurs problèmes que cette approche permettait de découvrir, hors-ligne ou en ligne, la structure d'un problème et permettait donc d'obtenir de bonnes performances, même lorsque la taille du problème était importante. D'une façon plus générale, nous pensons que l'une des contributions des travaux que nous avons présentés est de souligner plusieurs avantages de la construction d'un modèle du problème pour la résolution d'un problème d'apprentissage par renforcement, indépendamment de l'approche SDYNA.

Lisibilité des représentations utilisées

En effet, le premier avantage est l'information apportée à l'utilisateur par l'apprentissage sur le problème à résoudre. Dans le cadre de SDYNA, nous avons vu que les arbres de décision pouvaient représenter simplement les différents composants des fonctions de transition et de récompense définissant le FMDP (voir chapitre 7). Non seulement cette information est facile d'accès, mais en plus, elle renseigne l'utilisateur sur le problème qu'il a défini. À plusieurs reprises, lors de la mise en œuvre de SDYNA dans Counter-Strike©, nous avons détecté des problèmes de programmation concernant les perceptions et les actions que nous avions développées à partir des représentations construites par l'apprentissage.

Utilisation de la connaissance d'un expert

Le deuxième avantage concerne les différentes interactions possibles entre l'algorithme d'apprentissage par renforcement et un expert du problème à résoudre. Dans ce domaine, une des solution est, par exemple, de simplifier la tâche de l'apprentissage en spécifiant la structure du problème à résoudre, comme c'est le cas des algorithmes d'apprentissage par renforcement dans les FMDPs (voir section 6.1.5, page 162). Dans notre cas, nous avons supposé qu'aucune connaissance de ce type n'était a priori disponible. Cependant, l'utilisation d'une telle connaissance est complètement compatible avec l'apprentissage effectué par SDYNA qui, par exemple, peut être restreint à seulement certaines distributions de probabilités conditionnelles.

Ce type d'utilisation de la connaissance d'un expert se distingue d'autres approches où, par exemple, l'expert définit une première politique à exécuter pour l'agent. La tâche de l'apprentissage

sera alors d'ajuster et de compléter cette politique. Dans notre cas, l'utilisation de la connaissance de l'expert se situe au niveau de la définition du problème, et non dans sa résolution.

Par exemple, dans le cadre du jeu vidéo, une carte est souvent associée à un graphe topologique que l'agent utilise pour planifier la trajectoire d'un PNJ. Sans nécessité d'apprentissage, ce graphe peut directement être utilisé pour construire la distribution de probabilités conditionnelles associée à la perception de la position courante de l'agent. À partir de cette construction, l'apprentissage peut, par exemple pour une perception indiquant la présence d'une cible, l'utiliser pour compléter la représentation du problème.

Restriction du domaine de l'apprentissage

Un troisième et dernier avantage dans la construction d'un modèle tel que celui de SDYNA est la possibilité pour l'utilisateur, par exemple lorsque l'apprentissage montre des résultats satisfaisants, de choisir sur quelle perception (ou quelle sous-partie de la distribution de probabilités conditionnelle modélisant cette perception) l'apprentissage peut continuer. Une telle possibilité intéresse directement les utilisateurs souhaitant qu'un agent puisse s'adapter à un problème tout en gardant un comportement global satisfaisant.

Ainsi, dans le cadre du jeu vidéo Counter-Strike©, une fois que l'agent a appris les représentations nécessaires pour pouvoir remplir ses objectifs (notamment tirer seulement lors de la présence d'un adversaire, ou poser la bombe lorsque le PNJ est sur un site de bombe), l'apprentissage peut être limité seulement à la perception détectant la présence d'une cible et stoppé pour toutes les autres perceptions de l'agent. Ainsi, tout en gardant les comportements de base du jeu (poser la bombe ou tirer sur son adversaire), l'agent s'adaptera en fonction, par exemple, de la présence des adversaires sur la carte.

Il est aussi possible de définir des limitations de ce type avec un algorithme d'apprentissage par renforcement direct, c'est-à-dire ne construisant pas de représentation du problème sous la forme d'une fonction de transition et de récompense. Dans ce cas, la définition de ces limites ne concernera seulement qu'une sous-partie de l'espace des couples état/action de l'agent. Au contraire, dans un cadre de SDYNA, limiter l'apprentissage de l'agent à une seule perception par exemple ne définit pas à quelle sous-partie de l'espace des couples état/action cette restriction s'applique, ceci étant calculé automatiquement par les algorithmes de planification.

Robustesse de l'architecture

En testant SDYNA sur un problème réel, nous avons pu montrer que cette architecture permettait de trouver une solution à ce problème, bien que l'hypothèse de Markov ne soit plus satisfaite et que le problème comporte du bruit dans l'exécution des actions. En utilisant les mêmes algorithmes avec les mêmes paramètres que les problèmes définis dans la littérature concernant la planification dans

les FMDPs (hormis une légère modification concernant l'apprentissage de la fonction de récompense), nous avons montré, d'une part, que SDYNA était suffisamment robuste pour s'adresser à des problèmes réels et, d'autre part, que les problèmes théoriques étaient représentatifs des problèmes réels. Ceci confirme le fait que les problèmes théoriques utilisés dans ce manuscrit en particulier et dans la littérature en général sont de bon jeux d'essais pour les algorithmes d'apprentissage par renforcement.

8.3.2 Limitations

Exploration dans les grands problèmes

La première limitation de SDYNA dans le cadre de l'apprentissage par renforcement est, comme nous l'avons montré au chapitre 6, la difficulté d'exploiter la structure du problème pour gérer le compromis exploration/exploitation dans les grands problèmes. En effet, les algorithmes PAC-MDP ont une complexité polynomiale en fonction du nombre de couples état/action existant dans le problème, mais ce nombre est exponentiel en fonction du nombre d'actions et de variables d'état décrivant le problème à résoudre. Afin de diminuer cette complexité, des algorithmes PAC-FMDP, ayant une complexité polynomiale en fonction du nombre de paramètres décrivant le modèle, ont été proposés. Cependant, ces algorithmes supposent de connaître la structure du problème, ce que nous ne supposons pas.

Bien que nous ayons proposé une solution permettant de s'adapter à plusieurs types de problèmes, elle ne se révèle que partiellement satisfaisante. Nous pensons que cette question sera un enjeu majeur des futurs algorithmes d'apprentissage par renforcement dans les grands problèmes dont la structure est inconnue.

Problème non stationnaire

Dans les travaux présentés dans ce manuscrit, nous n'avons pas traité les problèmes non stationnaire, c'est-à-dire dont la dynamique et/ou les récompenses obtenues par l'agent évoluent au cours du temps, ni de la façon dont une architecture telle que SDYNA se comporterait pour la résolution de ce type de problème. De plus, dans les algorithmes que nous avons proposés, nous n'avons pas supposé de restriction de l'espace mémoire requis pour stocker les observations et les exemples utilisés par l'apprentissage pour la construction des fonctions de transition et de récompense du FMDP représentant le problème.

Limiter l'espace mémoire et donc limiter le nombre d'observations et d'exemples utilisables par l'apprentissage pourrait être une façon de gérer des problèmes non stationnaires. En effet, lorsque le nombre d'exemples maximum est atteint, il est nécessaire d'en enlever, c'est-à-dire "d'oublier" une observation. Pour gérer un problème non stationnaire, il serait possible de sélectionner l'exemple

le plus ancien à chaque oubli.

Pour de tels problèmes, une nouvelle complication concernant l'exploration apparaît : parce que le modèle de l'environnement de l'agent peut être faux, il devient nécessaire de réévaluer les paramètres du modèle qui n'ont pas été testés depuis longtemps. Pour gérer un tel problème, l'algorithme DYNA-Q + (Sutton and Barto, 1998) ajoute un bonus dans le calcul des valeurs d'action aux couples état/action qui n'ont pas été testés depuis longtemps. Une solution similaire pourrait être envisagée dans le cadre de SDYNA.

Problèmes réels de grande taille

Le problème réel avec lequel nous avons testé SDYNA était de petite taille, bien que SDYNA ait été conçu et testé sur des problèmes de grande taille. La principale raison est un manque matériel de temps pour effectuer les développements nécessaires à la définition des perceptions et des actions de chaque PNJ. Nous ne voyons aucune difficulté théorique ou pratique importante qui pourrait empêcher l'utilisation de SDYNA sur des problèmes réels dont la taille est similaire aux problèmes théoriques que nous avons présentés dans ce manuscrit.

Si les problèmes réels s'avèrent particulièrement bruités, il est probable qu'un apprentissage plus robuste devra être utilisé (voir section 8.1.2). Pour de tels problèmes, des méthodes d'approximation de la fonction de valeur similaire à celle décrite pour l'algorithme SVI (Boutilier et al., 2000), pour l'algorithme SPUDD (St-Aubin et al., 2000) ou bien une approche basée sur la programmation linéaire pourront être envisagées.

Conclusion et perspectives

L'objectif des travaux décrits dans ce manuscrit était d'utiliser sur des problèmes d'apprentissage par renforcement de grande taille et dont la structure est inconnue des méthodes de planification dans les FMDPs, c'est-à-dire des méthodes de planification initialement conçues pour être appliquées sur des problèmes de grande taille mais dont la structure est connue. Par une telle approche, nos travaux visaient à, d'une part, élargir le champ d'application des méthodes de planification aux problèmes dont la structure est inconnue, d'autre part, proposer une nouvelle famille de solutions pour la résolution de grands problèmes d'apprentissage par renforcement stochastiques et discrets.

Nous avons commencé par décrire le cadre des MDPs. Ce cadre permet de représenter de façon unifiée aussi bien des problèmes de planification que des problèmes d'apprentissage par renforcement. Dans ce cadre, nous avons tout d'abord décrit deux approches pour la planification, la programmation dynamique et la programmation linéaire, qui calculent une politique optimale à partir de la définition complète du problème. Nous avons aussi décrit deux algorithmes d'apprentissage par renforcement, Q-LEARNING et DYNA-Q, qui ne supposent pas que la définition était connue. Nous avons notamment souligné le fait que DYNA-Q permettait d'intégrer la planification avec l'apprentissage pour la résolution du problème.

Le principal inconvénient des MDPs est qu'ils ne permettent pas de représenter de grands problèmes. Pour cela, nous avons décrit le cadre des FMDPs qui utilise la structure du problème pour pouvoir le représenter de façon compacte. Nous avons distingué plusieurs types de structures pouvant être exploitées à partir d'une décomposition des fonctions de transition et de récompense : les indépendances relatives aux fonctions, les indépendances relatives aux contextes et l'approximation additive. Nous avons ensuite décrit plusieurs méthodes de planification, basées soit sur la programmation dynamique, soit sur la programmation linéaire, et capables d'exploiter la structure du problème pour représenter de façon compacte, lorsque c'est possible, les fonctions caractéristiques du problème.

Pour pouvoir exploiter la structure du problème, les méthodes de planification dans les FMDPs supposent que cette structure est connue a priori. Or, cette hypothèse n'est pas adaptée à de nombreux problèmes d'apprentissage par renforcement. Pour pouvoir apprendre cette structure à partir d'un ensemble d'observations de l'environnement, nous avons tout d'abord proposé une méthode

décomposant des observations en plusieurs ensembles d'exemples. Ces ensembles d'exemples sont utilisés ensuite par des méthodes d'apprentissage supervisés, plus précisément des algorithmes d'induction d'arbres de décision, pour construire une représentation du problème sous la forme d'un FMDP. Une fois le FMDP construit par l'apprentissage, nous avons montré qu'il était possible d'utiliser les méthodes de planification pour calculer une solution au problème d'apprentissage par renforcement, alors que sa structure n'était pas connue.

Ainsi, dans le cadre de l'apprentissage par renforcement hors-ligne, nous avons montré que notre approche était adaptée à la résolution de problèmes de grande taille principalement pour deux raisons. La première est l'utilisation de la propriété de généralisation des algorithmes d'induction d'arbres de décision évitant ainsi un ensemble exhaustif d'observations pour l'apprentissage. La deuxième est l'utilisation des algorithmes de planification dans les FMDPs permettant ainsi de représenter de façon compacte les fonctions qui peuvent l'être. La difficulté de l'apprentissage et de la résolution d'un problème d'apprentissage dépend alors de façon plus étroite de la complexité de sa structure, plutôt que de sa taille.

Dans le cadre de l'apprentissage par renforcement en ligne dans les FMDPs, nous avons proposé l'architecture SDYNA qui, à l'instar de l'architecture DYNA, intègre la planification et l'apprentissage. Nous avons tout d'abord décrit les algorithmes d'induction d'arbres de décision en ligne et montré que ceux ci pouvaient être utilisés de façon similaire. De plus, nous avons adapté les algorithmes de planification dans les FMDPs afin de pouvoir réutiliser la solution calculée lors d'un pas de temps précédent d'une part, et de limiter les calculs entre chaque pas de temps d'autre part. Nous avons montré que SDYNA permettait d'obtenir de meilleurs résultats qu'une approche telle que DYNA-Q, surtout lorsque la taille du problème d'apprentissage par renforcement était grande.

Afin d'améliorer les performances de SDYNA sur certains problèmes nécessitant une exploration dirigée de la part de l'agent pour sa résolution, nous avons proposé puis testé un algorithme dans le cadre de SDYNA intégrant des méthodes d'exploration dirigée existant dans la littérature. Nous avons montré que, bien que cet algorithme permette certaines améliorations pour des problèmes où l'exploration est difficile, il illustre le fait que l'utilisation des méthodes d'exploration dirigée pour des problèmes de grande taille ne nécessitant par une exploration exhaustive est difficile lorsque la structure du problème est inconnue.

Nous avons ensuite testé SDYNA sur un problème réel. Ce problème avait pour objectif le contrôle d'un PNJ dans le jeu Counter-Strike©. Par une étude qualitative, nous avons montré que SDYNA permettait de construire des représentations et des solutions pertinentes du problème, malgré la présence de bruit, le fait que l'hypothèse de Markov ne soit pas satisfaite, et que le problème soit dynamique. Cependant, notre représentation du problème était de petite taille et nous n'avons malheureusement pas eu le temps matériel de valider SDYNA sur un problème réel de grande taille.

Pour conclure, nous pensons avoir clairement établi, notamment par la présentation de l'architecture SDYNA dans le cadre des FMDPs, que les techniques d'apprentissage supervisé d'une part, et

les techniques de planification d'autre part, peuvent être utilisées de façon complémentaire et permettent de résoudre de nouveaux problèmes inaccessibles auparavant. Notre approche permet de combiner à la fois des propriétés de généralisation, d'agrégation et d'approximation qui en font une solution adaptée à la résolution de problèmes d'apprentissage par renforcement de grande taille, stochastiques et discrets. Pour cela, les FMDPs constituent un cadre mathématique idéal pour faire collaborer deux domaines de recherche, l'apprentissage supervisé et la planification, afin d'exploiter la structure d'un problème pour la résolution de grands problèmes. Cependant, nous pensons que des recherches supplémentaires sont à effectuer principalement dans le domaine de l'application de SDYNA sur des problèmes réels de grande taille et concernant la gestion du compromis exploration/exploitation.

Perspectives

L'approche que nous avons proposée, SDYNA, a été décrite dans le cadre des FMDPs. Ce cadre fait plusieurs hypothèses sur la nature du problème à résoudre, comme par exemple l'hypothèse de Markov. Proposer une version de SDYNA dans un cadre plus général que celui des MDPs permettrait d'élargir le champ d'application de celui-ci.

En premier lieu, il pourrait être intéressant de considérer des problèmes où il est nécessaire d'utiliser l'historique de l'agent pour résoudre certaines ambiguïtés qui pourraient exister dans l'état courant de l'agent, lorsque l'hypothèse de Markov n'est pas satisfaite. Pour cela, il serait nécessaire d'exprimer SDYNA dans un cadre plus général adapté à ce type de problème : les MDPs Partiellement Observables (POMDPs), ou *Partially Observable* MDPs (Sondik, 1971; Kaelbling et al., 1998). Des solutions dans ce cadre autoriseraient l'utilisation de variables d'état antérieur à l'état courant de l'agent, aussi bien dans la représentation du problème, que dans les fonctions nécessaires à sa résolution.

Deuxièmement, SDYNA suppose que les actions et que les variables d'états sont discrètes. Or, ce n'est pas le cas de certains problèmes qui possèdent certaines variables continues (l'espace d'état devient alors infini) mais aussi certaines actions continues (l'espace d'action devient alors infini). Des travaux tels que Guestrin et al. (2004); Kveton and Hauskrecht (2006b); Kveton (2006) utilisent le cadre des FMDPs hybrides, ou *hybrid factored* MDP qui étend le cadre des FMDPs aux variables et actions continues. D'une façon similaire à celle que nous avons décrite dans ce manuscrit, la fonction de valeur est approchée par une combinaison linéaire de fonctions de base, chacune ne dépendant que d'un petit nombre de variables du problème. Un programme linéaire est ensuite généré, en utilisant la structure du problème, pour déterminer la valeur des pondérations de chacune des fonctions de base. Cette approche suppose de connaître la structure du problème. Une telle méthode, combinée avec un apprentissage supervisé étant capable de gérer des variables discrètes ou continues, pourraient éventuellement être une alternative supplémentaire aux méthodes d'appren-

tissage par renforcement dans les problèmes avec des variables et des actions continues (Bertsekas and Tsitsiklis, 1996; Munos and Moore, 2002).

Enfin, le fait de pouvoir traiter de grands problèmes s'avèrent particulièrement utile dans les problèmes d'apprentissage par renforcement multi-agents, dont la complexité dépend directement du nombre d'agents dans le problème. Pour cela, des cadres mathématiques ont été proposés, notamment les MDPs multi-agents (*Multiagent* MDP) (Boutilier, 1999), ou encore les MDPs décentralisés partiellement observables (*DECentralized Partially Observable* MDP) (Bernstein et al., 2003). Nous pouvons aussi citer les travaux de Guestrin et al. (2002b) qui utilise, une fois de plus, la structure du problème et une représentation approchée de la fonction de valeur pour résoudre de grands problèmes de planification multi-agent. Dans un cadre similaire à celui de SDYNA, ces méthodes de planification combinées avec un apprentissage supervisé capable d'extraire la structure du problème pourraient éventuellement être une solution viable pour la résolution de grands problèmes d'apprentissage par renforcement multi-agent.

Bibliographie

Auer, P., Holte, R., and Maass, W. (1995). Theory and applications of agnostic PAC-learning with small decision trees. In *Proceedings of the Twelfth International Conference on Machine Learning*, volume 2129. 199

Bahar, R., Frohm, E., Gaona, C., Hachtel, G., Macii, E., Pardo, A., and Somenzi, F. (1993). Algebraic Decision Diagrams and their Applications. In *IEEE/ACM International Conference on CAD*, pages 188–191, Santa Clara, California. 54

Barto, A., Bradtke, S., and Singh, S. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1-2) :81–138. 160, 204

Bauer, E. and Kohavi, R. (1999). An Empirical Comparison of Voting Classification Algorithms : Bagging, Boosting, and Variants. *Machine Learning*, 36(1) :105–139. 199

Bellman, R., Kalaba, R., and Kotkin, B. (1963). Polynomial Approximation - a New Computational Technique in Dynamic Programming. *Math. Comp.*, 17(8) :155–161. 66

Bellman, R. E. (1957). *Dynamic Programming*. Princeton University Press, Princeton, New Jersey. 17, 18

Bernstein, D., Givan, R., Immerman, N., and Zilberstein, S. (2003). The Complexity Of Decentralized Control Of Markov Decision Processes. *Mathematics of Operations Research*, 27(4) :819–840. 214

Bertsekas, D. P. and Tsitsiklis, D. (1996). *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA. 214

Boutilier, C. (1997). Correlated Action Effects in Decision Theoretic Regression. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence*, pages 30–37. 199

Boutilier, C. (1999). Sequential optimality and coordination in multiagent systems. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, volume 1, pages 478–485. 214

Boutilier, C., Dearden, R., and Goldszmidt, M. (1995). Exploiting Structure in Policy Construction. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1104–1111, Montreal. 18, 37, 39

Boutilier, C., Dearden, R., and Goldszmidt, M. (2000). Stochastic Dynamic Programming with Factored Representations. *Artificial Intelligence*, 121(1) :49–107. 24, 42, 43, 44, 49, 51, 108, 109, 199, 209

Boutilier, C. and Goldszmidt, M. (1996). The Frame Problem and Bayesian Network Action Representations. In *Proceedings of the Eleventh Biennial Canadian Conference on Artificial Intelligence (AI '96)*, pages 69–83, Toronto, CA. 40

Boutilier, C., T., D., and Hanks, S. (1999). Decision-Theoretic Planning : Structural Assumptions and Computational Leverage. *Journal of Artificial Intelligence Research*, 11 :1–94. 37

Brafman, R. and Tennenholtz, M. (2003). R-MAX- A general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, 3(2) :213–231. 156, 159

Breiman, B. and Breiman, L. (1984). *Classification and Regression Trees*. Chapman & Hall/CRC. 83, 86

Breiman, L. (1996). Bagging Predictors. *Machine Learning*, 24(2) :123–140. 199

Breiman, L., Friedman, J., Olshen, R., and Stone, C. (1984). *Classification And Regression Trees*. Chapman & Hall, Inc., New York. 83, 86

Bryant, R. E. (1986). Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8) :677–691. 44, 55

Chickering, D. M., Heckerman, D., and Meek, C. (1997). A Bayesian Approach to Learning Bayesian Networks with Local Structure. In *Proceedings of the 13th International Conference on Uncertainty in Artificial Intelligence*, pages 80–89. 128

Cornuéjols, A. and Miclet, L. (2002). *Apprentissage Artificiel : Concepts et Algorithmes, chapter Apprentissage de réflexes par renforcement*. Eds. Eyrolles, France. 157

Cover, T. M. (1991). Universal Portfolios. *Mathematical Finance*, 1(1) :1–29. 84

Dayan, P. and Sejnowski, T. (1996). Exploration Bonuses and Dual Control. *Machine Learning*, 25(1) :5–22. 155

de Farias, D. and Van Roy, B. (2001). The Linear Programming Approach to Approximate Dynamic Programming. *Operations Research*, 51(6) :850–856. 67

de Farias, D. and Van Roy, B. (2004). On Constraint Sampling in the Linear Programming Approach to Approximate Dynamic Programming. *Mathematics of Operations Research*, 29(3) :462–478. 77

Dean, T. and Kanazawa, K. (1989). A Model for Reasoning about Persistence and Causation. *Computational Intelligence*, 5 :142–150. 18, 40

Dearden, R. and Boutilier, C. (1997). Abstraction and Approximate Decision Theoretic Planning. *Artificial Intelligence*, 89(1) :219–283. 117

Decatur, S. (1997). PAC Learning with Constant-Partition Classification Noise and Applications to Decision Tree Induction. In *Proceedings of the Fourteenth International Conference on Machine Learning*, volume 112, pages 113–115. 199

Degris, T., Sigaud, O., and Wuillemin, P. (2006a). Chi-square Tests Driven Method for Learning the Structure of Factored MDPs. In *Proceedings of the 22nd Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 122–129, Cambridge, MA, USA. 105, 108, 113

Degris, T., Sigaud, O., and Wuillemin, P. (2006b). Learning the Structure of Factored Markov Decision Processes in Reinforcement Learning Problems. In *Proceedings of the 23rd International Conference on Machine Learning (ICML)*, pages 257–264, Pittsburgh, Pennsylvania, USA. 145

Fern, A. and Givan, R. (2003). Online Ensemble Learning : An Empirical Study. *Machine Learning*, 53(1) :71–109. 199

Forsell, N. and Sabbadin, R. (2006). Algorithme de résolution approchée basé sur la programmation linéaire pour les processus décisionnels de markov sur graphe. In *Actes de la conférence JFPDA'06*, pages 89–96, Toulouse, France. 194

Freund, Y. and Schapire, R. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1) :119–139. 199

Friedman, J. H. (1977). A Recursive Partitioning Decision Rule for Nonparametric Classification. *IEEE Transactions on Computers*, C-26 :404–408. 84

Friedman, N. and Goldszmidt, M. (1998). Learning Bayesian Networks with Local Structure. In *Learning and Inference in Graphical Models*. M. I. Jordan ed. 128

Guestrin, C. (2003). *Planning Under Uncertainty in Complex Structured Environments*. PhD thesis, Computer Science Department, Stanford University, USA. 61, 62, 205

Guestrin, C., Hauskrecht, M., and Kveton, B. (2004). Solving factored MDPs with continuous and discrete variables. In *Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 235–242. AUAI Press Arlington, Virginia, United States. 213

Guestrin, C., Koller, D., Gearhart, C., and Kanodia, N. (2003a). Generalizing Plans to New Environments in Relational MDPs. In *International Joint Conference on Artificial Intelligence (IJCAI-03)*. 194

Guestrin, C., Koller, D., and Parr, R. (2001). Max-norm Projections for Factored MDPs. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2001)*, pages 673–680. 61

Guestrin, C., Koller, D., Parr, R., and Venkataraman, S. (2003b). Efficient Solution Algorithms for Factored MDPs. *Journal of Artificial Intelligence Research*, 19 :399–468. 42, 61, 62, 65, 66, 69, 71, 72, 74, 75, 76, 77, 98, 122, 199, 205

Guestrin, C., Patrascu, R., and Schuurmans, D. (2002a). Algorithm-Directed Exploration for Model-Based Reinforcement Learning in Factored MDPs. In *ICML-2002 The Nineteenth International Conference on Machine Learning*, pages 235–242. 164

Guestrin, C., Venkataraman, S., and Koller, D. (2002b). Context specific multiagent coordination and planning with factored MDPs. *AAAI 8th National Conference on Artificial Intelligence, Edmonton, Canada, July*. 214

Hoey, J., St-Aubin, R., Hu, A., and Boutilier, C. (1999). SPUDD : Stochastic Planning using Decision Diagrams. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, pages 279–288. Morgan Kaufmann. 54, 141

Hoey, J., St-Aubin, R., Hu, A., and Boutilier, C. (2000). Optimal and Approximate Stochastic Planning using Decision Diagrams. Technical Report TR-00-05, University of British Columbia. 54, 61, 97, 141

Howard, R. A. (1960). *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, Massachusetts. 25

Jaakkola, T., Jordan, M. I., and Singh, S. P. (1994). On the Convergence of Stochastic Iterative Dynamic Programming Algorithms. *Neural Computation*, 6(6) :1185–1201. 34

Kaelbling, L., Littman, M., and Cassandra, A. (1998). Planning and Acting in Partially Observable Stochastic Domains. *Artificial Intelligence*, 101(1-2) :99–134. 213

Kaelbling, L. P. (1993). *Learning in Embedded Systems*. The MIT Press. 159, 160

Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). Reinforcement Learning : a Survey. *Journal of Artificial Intelligence Research*, 4 :237–285. 159

Kakade, S. M. (2003). *On the Sample Complexity of Reinforcement Learning*. PhD thesis, Gatsby Computational Neuroscience Unit, University College London. 157

Kalles, D. and Morris, T. (1996). Efficient Incremental Induction of Decision Trees. *Machine Learning*, 24(3) :231–242. 134

Kearns, M. and Koller, D. (1999). Efficient Reinforcement Learning in Factored MDPs. In *Sixteenth International Joint Conference on Artificial Intelligence (IJCAI)*, volume 99, pages 740–747. 162, 164

Kearns, M. and Singh, S. (1998). Near-Optimal Reinforcement Learning in Polynomial Time. In *Proceedings of the 15th International Conference on Machine Learning*, pages 260–268. 156, 157

Koller, D. and Parr, R. (1999). Computing Factored Value Functions for Policies in Structured MDPs. In *Proceedings Sixteenth International Joint Conference on Artificial Intelligence (IJ-CAI)*, pages 1332–1339. 61, 69, 72

Koller, D. and Parr, R. (2000). Policy Iteration for Factored MDPs. In *Proceedings of the 16th Annual Conference on Uncertainty in AI (UAI)*, pages 326–334. 61, 65, 74

Kveton, B. (2006). *Planning In Hybrid Structured Stochastic*. PhD thesis, University of Pittsburgh. 213

Kveton, B. and Hauskrecht, M. (2006a). Learning Basis Functions in Hybrid Domains. In *Proceedings of the 21st National Conference on Artificial Intelligence*, pages 1161–1166. 205

Kveton, B. and Hauskrecht, M. (2006b). Solving Factored MDPs with Exponential-Family Transition Models. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling*, pages 114–120. 213

Liberatore, P. (2002). The size of MDP factored policies. *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI 2002)*, pages 267–272. 69

Manne, A. S. (1960). *Linear Programming and Sequential Decisions*. Cowles Foundation for Research in Economics at Yale University. 32

McCallum, A. (1996). Learning to Use Selective Attention and Short-Term Memory in Sequential Tasks. In *Simulation of Adaptive Behabiour, From Animals to Animats*, volume 4, pages 315–324. 153

McCallum, A. K. (1995). *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, Department of Computer Science, University of Rochester, USA. 153, 201

Moore, A., Atkeson, C., et al. (1993). Prioritized sweeping : Reinforcement learning with less data and less real time. *Machine Learning*, 13(1) :103–130. 204

Mundhenk, M., Goldsmith, J., Lusena, C., and Allender, E. (2000). Complexity of finite-horizon Markov decision process problems. *Journal of the ACM (JACM)*, 47(4) :681–720. 69

Munos, R. and Moore, A. (2002). Variable Resolution Discretization in Optimal Control. *Machine Learning*, 49(2) :291–323. 214

Patrascu, R., Poupart, P., Schuurmans, D., Boutilier, C., and Guestrin, C. (2002). Greedy Linear Value-Approximation for Factored Markov Decision Processes. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 285–291. 205

Pearl, J. (1988). *Probabilistic Reasonning in Intelligent Systems : Networks of Plausible Inference*. Morgan Kaufmann, San Mateo. 39

Peng, J. and Williams, R. (1993). Efficient Learning and Planning Within the Dyna Framework. *Adaptive Behavior*, 1(4) :437. 204

Peng, J. and Williams, R. J. (1992). Efficient learning and planning within the dyna framework. In Meyer, J.-A., Roitblat, H. R., and Wilson, S. W., editors, *Proceedings of the 2nd International Conference on Simulation of Adaptive Behavior, Hawaii*, pages 437–454. 35

Pichuka, C., Bapi, R., Bhagvati, C., Pujari, A. K., and Deekshatulu, B. L. (2007). A Tighter Error Bound For Decision Tree Learning Using Pac Learnability. In *International Joint Conference on Artificial Intelligence (IJCAI07)*, page To Appear. 199

Poole, D. (1997). The Independent Choice Logic for Modelling Multiple Agents under Uncertainty. *Artificial Intelligence*, 94(1-2) :7–56. 44

Poupart, P., Boutilier, C., Patrascu, R., and Schuurmans, D. (2002). Piecewise Linear Value Function Approximation for Factored MDPs. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 292–299. 205

Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T. (1992). *Numerical Recipes : The Art of Scientific Computing*. Cambridge University Press. 113

Puterman, M. L. (1996). *Markov Decision Processes : Discrete Stochastic Dynamic Programming.* John Wiley and Sons, New York. 27, 28

Quinlan, J. (1983). Learning Efficient Classification Procedures and their Application to Chess End Games. *Machine Learning : An Artificial Intelligence Approach*, 1 :463–482. 83

Quinlan, J. R. (1986). Induction of Decision Trees. *Machine Learning*, 1(1) :81–106. 83, 84

Quinlan, J. R. (1993). *C4.5 : Programs for Machine Learning*. Morgan Kaufmann, San Mateo. 44, 83, 84, 196

Rivest, R. L. (1987). Learning Decision Lists. *Machine Learning*, 2 :229–246. 44

Robert, G. (2005). *MHiCS, une Architecture de Sélection de l'Action Motivationnelle et Hiérarchique à Systèmes de Classeurs pour Personnages Non Joueurs Adaptatifs*. PhD thesis, Laboratoire Informatique de Paris VI, France. 175, 194

Saporta, G. (1990). *Probabilités, analyse des données et statistique*. Paris : Editions Technip. 84, 85

Schlimmer, J. and Fisher, D. (1986). A Case Study of Incremental Concept Induction. *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 496–501. 132, 134, 135

Schweitzer, P. and Seidmann, A. (1985). Generalized Polynomial Approximations in Markovian Decision Processes. *Journal of Mathematical Analysis and Applications*, 110 :568–582. 61, 67

Shapley, L. (1953). Stochastic Games. *Proceedings of the National Academy of Sciences*, 39(10) :1095–1100. 159

Sigaud, O. (2004). *Comportements Adaptatifs pour des Agents dans des Environnements Informatiques Complexes*. Habilitation à Diriger des Recherches de l'Université PARIS VI. 175

Singh, S., Jaakkola, T., Littman, M., and Szepesvári, C. (2000). Convergence Results for Single-Step On-Policy Reinforcement-Learning Algorithms. *Machine Learning*, 38(3) :287–308. 155

Sondik, E. (1971). *The Optimal Control of Partially Observable Markov Processes*. PhD thesis, Stanford University, California. 213

St-Aubin, R., Hoey, J., and Boutilier, C. (2000). APRICODD : Approximate Policy Construction Using Decision Diagrams. In *NIPS*, pages 1089–1095. 61, 62, 141, 209

Strehl, A. (2007). Model-Based Reinforcement Learning in Factored MDPs. In *Proceedings of the IEEE Symposium on Approximate Dynamic Programming*, page To appear. 157, 162, 163, 164

Strehl, A. and Littman, M. (2004). An Empirical Evaluation of Interval Estimation for Markov Decision Processes. In *Proceedings of the 16th IEEE International on Tools with Artificial Intelligence Conference (ICTAI 2004)*, pages 128–135. 160, 161

Strehl, A. and Littman, M. (2005). A theoretical analysis of model-based interval estimation. In *Proceedings of the Twenty-second International Conference on Machine Learning (ICML-05)*, pages 857–864. 156, 160, 161

Strehl, A. and Littman, M. (2006a). An Analysis of Model-Based Interval Estimation for Markov Decision Processes. *Paper submitted in July 2006 to Elsevier Science*. 162

Strehl, A. and Littman, M. (2006b). Incremental Model-based Learners With Formal Learning-Time Guarantees. In *Proceedings of the 22nd Conference on Uncertainty in Artificial Intelligence (UAI 2006)*, pages 485–493. 160, 162

Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224. San Mateo, CA. Morgan Kaufmann. 33, 34, 130, 159

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning : An Introduction*. MIT Press. 17, 30, 34, 130, 131, 137, 156, 159, 209

Thrun, S. (1992). The role of exploration in learning control. In White, D. and Sofge, D., editors, *Handbook for Intelligent Control : Neural, Fuzzy and Adaptive Approaches*. Van Nostrand Reinhold, Florence, Kentucky 41022. 155

Torgo, L. (2000). Inductive Learning of Tree-based Regression Models. *AI Communications*, 13(2) :137–138. 86, 196

Tsitsiklis, J. N. (1994). Asynchronous Stochastic Approximation and Q-learning. *Machine Learning*, 16 :185–202. 34

Utgoff, P. (1986). Incremental Induction of Decision Trees. *Machine Learning*, 4 :161–186. 84, 132, 133

Utgoff, P. (1988). ID5 : an Incremental ID3. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 107–120, Ann Arbor. 132, 133

Utgoff, P. E., Nerkman, N. C., and Clouse, J. A. (1997). Decision Tree Induction Based on Efficient Tree Restructuring. *Machine Learning*, 29(1) :5–44. 132, 133

Valiant, L. (1984). A theory of the learnable. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 436–445. ACM Press New York, NY, USA. 157

Watkins, C. J. C. H. (1989). *Learning with Delayed Rewards*. PhD thesis, Psychology Department, University of Cambridge, England. 33

Watkins, C. J. C. H. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3) :279–292. 34

Wiering, M. and Schmidhuber, J. (1998). Efficient model-based exploration. *Proceedings of the Sixth International Conference on Simulation of Adaptive Behavior : From Animals to Animats*, 6 :223–228. 160

Zhang, T. and Poole, D. (1999). On the Role of Context-specific Independence in Probabilistic Reasoning. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 1288–1293, Stockholm. 62