Représentations graphiques de fonctions et processus décisionnels Markoviens factorisés .

Jean-Christophe Magnan

THÈSE DE DOCTORAT DE
l'UNIVERSITÉ PIERRE ET MARIE CURIE - PARIS VI

Spécialité

Informatique

École Doctorale d'Informatique, de Télécommunication et d'Électronique de Paris

Présentée par

M. Jean-Christophe Magnan

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE - PARIS VI

Sujet de la thèse :

Représentations Graphiques de Fonctions et Processus Décisionnels de Markov Factorisés

soutenue le 2 Février 2016

devant le jury composé de :

M. Christophe Gonzales	(LIP6, Paris)	Directeur de Thèse
M. Pierre-Henri Wuillemin	(LIP6, Paris)	Co-encadrant
M. Régis Sabbadin	(INRA, Toulouse)	Rapporteur
M. Abdel-Illah Mouaddib	(GREYC, Caen)	Rapporteur
M. Olivier Sigaud	(ISIR, Paris)	Examinateur
M. Florent Teichteil-Koenigsbuch	(Airbus, Toulouse)	Examinateur

En planification théorique de la décision, le cadre des Processus Décisionnels Markoviens Factorisés (Factored Markov Decision Process, FMDP) a produit des algorithmes efficaces de résolution des problèmes de décisions séquentielles dans l'incertain. L'efficacité de ces algorithmes repose sur des structures de données telles que les ARBRES DE DÉCISION ou les Diagrammes de Décision Algébriques (ADDs). Ces techniques de planification sont utilisées en *Apprentissage par Renforcement* par l'architecture **SDYNA** afin de résoudre des problèmes inconnus de grandes tailles. Toutefois, l'état de l'art des algorithmes d'apprentissage, de programmation dynamique et d'apprentissage par renforcement utilisés par **SDYNA**, requière que le problème soit spécifié uniquement à l'aide de variables binaires et/ou utilise des structures améliorables en termes de compacité. Dans ce manuscrit, nous présentons nos travaux de recherche visant à élaborer et à utiliser une structure de donnée plus efficace et moins contraignante, et à l'intégrer dans une nouvelle instance de l'architecture **SDYNA**.

Dans une première partie, nous présentons l'état de l'art de la modélisation de problèmes de décisions séquentielles dans l'incertain à l'aide de FMDP. Nous abordons en détail la modélisation à l'aide d'ARBRE DE DÉCISION et d'ADDs. Puis nous présentons les RGFORs, nouvelle structure de données que nous proposons dans cette thèse pour résoudre les problèmes inhérents aux ADDs. Nous démontrons ainsi que les RGFORs s'avèrent plus efficaces que les ADDs pour modéliser les problèmes de grandes tailles.

Dans une seconde partie, nous nous intéressons à la résolution des problèmes de décision dans l'incertain par *programmation dynamique*. Après avoir introduit les principaux algorithmes de résolution, nous nous attardons sur leurs variantes dans le domaine factorisé. Nous précisons les points de ces variantes factorisées qui sont améliorables. Nous décrivons alors une nouvelle version de ces algorithmes qui améliore ces aspects et utilise les RGFORs précédemment introduits.

Dans une dernière partie, nous abordons l'utilisation des FMDPs en *Apprentissage par Renforcement*. Puis nous présentons un nouvel algorithme d'apprentissage dédié à la nouvelle structure que nous proposons. Grâce à ce nouvel algorithme, une nouvelle instance de l'architecture **SDYNA** est proposée, se basant sur les RGFORs : l'instance **SPIMDDI**. Nous testons son efficacité sur quelques problèmes standards de la littérature. Enfin nous présentons quelques travaux de recherche autour de cette nouvelle instance. Nous évoquons d'abord un nouvel algorithme de gestion du compromis exploration-exploitation destiné à simplifier l'algorithme **F-RMAX**. Puis nous détaillons une application de l'instance **SPIMDDI** à la gestion d'unités dans un jeu vidéo de stratégie en temps réel.

Table des matières

Liste des Figures

Liste des Tableaux

Liste des Algorithmes

Abreviations

IA	**I**ntelligence **A**rtificielle
MDP	**M**arkov **D**ecision **P**rocess
FMDP	**F**actored **M**arkov **D**ecision **P**rocess
HMDP	**H**ierarchical **M**arkov **D**ecision **P**rocess
RMDP	**R**elationnal **M**arkov **D**ecision **P**rocess
VI	**V**alue **I**teration
PI	**P**olicy **I**teration
SVI	**S**tructured **VI**
SPI	**S**tructured **PI**
SPUDD	**S**tochastic **P**lanning **U**sing **D**ecision **D**iagrams
SPUMDD	**S**tochastic **P**lanning **U**sing **M**ulti-valued **D**ecision **D**iagrams
ITI	**I**ncremental **T**ree **I**nduction
ITI+DD	**I**ncremental **T**ree **I**nduction and **D**ecision **D**iagrams
IOTI	**I**ncremental **O**rdered **T**ree **I**nduction
IMDDI	**I**ncremental **M**ulti-valued **D**ecision **D**iagrams **I**nduction
SDYNA	**S**tructured **D**yna
SPITI	**S**tochastic **P**lanning and **I**ncremental **T**ree **I**nduction
SPIMDDI	**S**tochastic **P**lanning and **I**ncremental **M**ulti-valued **D**ecision **D**iagrams **I**nduction
DAG	**D**irected **A**cyclic **G**raph
BN	**B**ayesian **N**etwork
RGFOR	**R**eprésentation **G**raphique de **F**onction **O**rdonnée et **R**éduite
ADD	**A**lgebraic **D**ecision **D**iagrams
BDD	**B**inary **D**ecision **D**iagrams
MDD	**M**ulti-valued **D**ecision **D**iagrams

Introduction

Considérons un cycliste qui cherche à se rendre à sa destination. Ce cycliste évolue dans un environnement dynamique et stochastique : la route, avec la circulation et les piétons à gérer. Il dispose d'un nombre d'actions pour agir : tourner, accélérer et décélérer, etc. À l'aide de ces actions, le cycliste doit parvenir à sa destination tout en s'adaptant aux changements imprévisibles de l'environnement. En Intelligence Artificielle, ce type de problème très étudié est couramment dénommé *Problème de Décisions Séquentielles dans l'Incertain*. Ces problèmes se caractérisent par un agent qui évolue dans un environnement stochastique pour y accomplir un objectif. Pour y parvenir, l'agent dispose d'actions qui lui permettent d'altérer l'environnement.

Pour résoudre ces problèmes, les méthodes standards (**PI** [Howard, 1960], **VI** [Bellman, 1957]) s'appuient sur une modélisation du problème sous forme de processus décisionnel markovien (MDP, [Puterman, 2005]). Cet outil mathématique pose formellement le problème en définissant clairement les configurations du système sous-jacent, les actions entreprenables par l'agent et les évolutions possibles du système. S'appuyant sur ces MDPs, les algorithmes **PI** et **VI** sont capables d'extraire efficacement des solutions optimales à ces problèmes, à savoir quels actions entreprendre pour atteindre l'objectif.

Malédiction de la dimension

Dans cette étude, nous nous intéressons à une difficulté assez handicapante pour la résolution des *problèmes de décisions séquentielles dans l'incertain* : la *malédiction de la dimension* [Bellman, 1961]. Identifié par Bellman alors qu'il travaillait sur des problèmes d'optimisation dynamique, cette notion met en exergue les difficultés qui apparaissent lorsque les espaces de travail deviennent de grande dimension. Cet obstacle affecte nos outils de trois façons différentes.

Malédiction de la dimension et problèmes de décisions séquentielles dans l'incertain

Premièrement, le formalisme MDP conventionnel échoue à modéliser les problèmes de grandes tailles. En effet, l'approche classique nécessite de caractériser complètement toutes les configurations et évolutions des systèmes étudiés. Or plus un système est complexe, plus son MDP nécessite d'entrées pour décrire complètement et correctement le système. Au bout du compte, la modélisation d'un problème complexe requière un espace mémoire beaucoup trop grand pour les ordinateurs contemporains.

Ensuite, les algorithmes de résolutions deviennent inutilisables sur les problèmes de tailles très importantes. En effet, ces algorithmes reposent sur l'énumération de toutes les configurations et évolutions des systèmes étudiés. Pour des systèmes complexes, la somme des calculs alors requis est beaucoup trop importante pour envisager d'exécuter ces algorithmes sur des architectures informatiques contemporaines.

Une dernière difficulté concerne les connaissances requises pour modéliser ces problèmes. En effet, les méthodes de résolution standards demandent une connaissance parfaite des systèmes étudiés. Or pour des systèmes complexes, cette connaissance est difficile voir impossible à acquérir. Une solution consiste alors à apprendre empiriquement et automatiquement les modèles de ces systèmes. Pour un système donnée, cet apprentissage requière de visiter plusieurs fois chaque configuration, et de tenter toutes les évolutions possibles depuis chacun de ces états plusieurs fois. Néanmoins, pour un système complexe, cet apprentissage est difficile à mettre en place, voir inenvisageable du fait du trop grand nombre de configurations à explorer.

Généraliser

Des solutions efficaces à ces difficultés existent autour du concept d'*abstraction* ([Boutilier et al., 1999], [Guestrin and Gordon, 2002], [Boutilier, 2001]). Les techniques d'abstraction consistent à agréger les configurations du système étudié qui évoluent de manière similaires et à les traiter comme une seule et même configuration abstraite. Ce regroupement a deux effets bénéfiques. D'une part, moins de configurations, et d'évolutions à partir de celles-ci, sont à modéliser. En effet, les états qui se comportent de manière similaire sont "résumés" en un état abstrait. Les modèles sont alors plus simples à mémoriser. D'autre part, les calculs nécessaires sont alors allégés. En effet, seules les configurations abstraites sont à prendre en considération dans les calculs.

Plus récemment, des solutions ont été proposées pour reconnaître ces configurations abstraites empiriquement ([Degris et al., 2006a]). Ces algorithmes proposent ainsi un

cadre de travail où l'agent explore le système, permettant ainsi à divers sous-algorithmes de collecter les données nécessaires à l'extraction des configurations abstraites et à la recherche des meilleurs actions à entreprendre pour ces configurations. D'autres algorithmes ([Strehl et al., 2007], [Chakraborty and Stone, 2011]) ont par ailleurs été proposés pour rendre l'exploration du système plus pertinente afin d'améliorer l'extraction de configurations abstraites.

Contributions

Dans cette thèse, nous nous intéressons spécifiquement au cadre factorisé ([Boutilier et al., 1999]). Dans ce cadre, un ensemble de variables est utilisé pour caractériser les configurations du système et leurs évolutions. Le processus d'abstraction, et donc la caractérisation des états abstraits, se base alors sur ces variables. L'état de l'art des algorithmes et des structures de données utilisées dans ce cadre présente quelques défauts qui atténuent son efficacité. L'objectif de cette thèse est de mettre en place un nouvel algorithme d'*apprentissage par renforcement* dans le cadre des MDPs factorisés s'appuyant sur une version améliorée de la structure de données proposée dans l'état de l'art.

Amélioration de la représentation La structure de données utilisées dans l'état de l'art impose une contrainte forte sur les variables utilisées pour modéliser le système : ces variables caractérisant le système doivent être binaires. De fait, là où une variable multimodale aurait été pratique pour décrire un aspect du système, il faut la transformer en un ensemble de variables binaires. Nous montrons que, moyennant quelques adaptations, les solutions actuelles de modélisation peuvent en réalité être étendues à un cadre multimodal. Cette extension facilite grandement les choix de modélisation, et s'avère encore plus efficace en terme d'espace mémoire requis.

Amélioration de la recherche de solution L'algorithme considéré comme le meilleur dans l'état de l'art pour la recherche de solution optimale impose une contrainte sur la manière dont est modélisé le système. Or le modèle est particulièrement sensible à cette contrainte : la taille mémoire qu'il requière dépend de cette contrainte. Nous nous intéressons donc à un nouvel algorithme qui n'impose plus cette contrainte. Nous verrons que ce retrait améliore la taille requise pour la modélisation et le temps de recherche de la solution optimale.

Amélioration de l'apprentissage Les solutions existantes pour la découverte et l'apprentissage du système ne permettent pas de manipuler les structures de données optimales pour la représentation du problème. De plus, les algorithmes d'exploration existants ne s'appuient pas sur ces structures. Notre travail ici est donc de fournir un algorithme d'apprentissage qui permet l'apprentissage de ces structures de données. De plus, nous introduisons quelques réflexions sur l'adaptation des algorithmes d'exploration aux nouvelles structures.

Problèmes standards

Pour illustrer les travaux présentés dans ce manuscrit, nous confronterons nos algorithmes aux algorithmes de l'état de l'art sur quelques problèmes standards de la littérature. Ces problèmes, bien que jouets, permettent de juger sans trop de difficultés de la qualité des solutions obtenues.

Le problème COFFEE ROBOT [Dearden and Boutilier, 1997]

Le premier modèle que nous utiliserons est le problème COFFEE ROBOT. Dans ce problème, un robot a pour tâche d'apporter du café à son propriétaire. Pour y parvenir, ce robot doit se rendre à la boutique du coin pour y acheter du café puis le ramener au bureau. Néanmoins, sur le trajet, qui se fait en extérieur, le robot peut se faire tremper à cause d'une averse ; ce qui n'est pas nécessairement une bonne chose pour ses circuits. Le robot doit donc l'éviter autant que possible. Avant de quitter le bureau, le robot a toutefois la possibilité de prendre un parapluie pour éviter ce désagrément. La tâche du robot est donc d'approvisionner son maître en café sans se faire arroser durant l'opération.

Ce problème est de petite taille ; il est donc facile d'analyser et de valider ou non les solutions trouvées par les algorithmes. De plus, comme nous le développerons par la suite, le système étudié dans ce problème se décrit à l'aide de variables uniquement binaires. Ce problème servira donc à nous assurer que nos propositions d'amélioration se comportent bien comme les versions originales dans le cas binaire.

Le problème FACTORY [Dearden and Boutilier, 1997]

Les problèmes FACTORY plutôt, puisqu'il en existe plusieurs versions, sont des problèmes décrivant la fabrication de pièces dans une chaîne d'assemblage. L'agent, qui contrôle cette chaîne, doit parvenir à assembler deux pièces après avoir effectué divers

traitements sur celles-ci : les polir, les peindre, les percer, les tailler, etc. Certains traitements peuvent en défaire d'autre, par exemple polir les pièces après les avoir peintes efface la peinture qui a été mise. De plus, la qualité de finition n'est pas nécessairement la même pour tous les assemblages de pièces ; les traitements requis changent en conséquence. Enfin, la qualité et la disposition des outils nécessaire à la réalisation des traitements sont variables, et changent la stratégie à suivre pour assembler les pièces en conséquent. L'agent doit donc trouver à partir de ces différentes contraintes la séquences optimale de traitements à faire subir aux pièces pour les assembler.

Ce problème, et ses multiples variantes qui rajoutent différents traitements à faire sur les pièces, est intéressant car il nous confronte directement au fléau de la dimensionnalité. En effet, ce problème sous sa forme standard ne peut être stocké en mémoire et demande des adaptations pour pouvoir être manipulé comme nous le verrons par la suite. Ce problème permettra donc de vérifier l'efficacité de nos solutions sur les problèmes qui représentent le cœur de notre cible : les problèmes de grandes tailles.

Le problème TAXI [Dietterich, 1998]

Le problème TAXI est un problème de navigation : l'agent est un chauffeur de taxi qui doit aller chercher un client et l'emmener à sa destination. Les points de récupération et de dépôts sont des points remarquables dans la ville : Maison, Travail, Club, Cinéma. Le chauffeur doit se rendre en ces différents points en naviguant dans la ville. Toutefois, sa position actuelle est déterminée par le carrefour où il se trouve. Le quadrillage des rues est à l'américaine : les rues orientées est-ouest sont numérotée de 0 à 4 suivant leur position sur l'axe nord-sud, les rues orientées nord-sud sont numérotée de 0 à 4 suivant leur position sur l'axe est-ouest. Un carrefour est donc décrit comme l'intersection de la rue tant suivant l'axe est-ouest et de la rue tant suivant l'axe nord-sud. De plus, le chauffeur doit faire attention à son réservoir qui se vide assez rapidement. Le chauffeur peut néanmoins faire un détour par une station-service pour refaire le plein.

Ce problème nous intéresse car, outre qu'il est de grande taille, c'est un problème multimodal. Toutes les variables utilisées pour décrire l'évolution du problème sont en effet définies sur des domaines discrets de cardinalités supérieures à deux. Nous verrons par la suite que ce genre de problèmes pose des difficultés pour les algorithmes de l'état de l'art, difficultés que nous résolvons dans ce manuscrit. Ce problème nous permet donc de juger de l'efficacité de nos solutions dans ce cadre.

Structure du manuscrit

Le mémoire s'articule en trois parties.

La première partie s'intéresse à la modélisation des *problèmes de décisions séquentielles dans l'incertain* de grandes tailles, et à l'amélioration des structures utilisées pour modéliser ces problèmes. Dans un premier chapitre, les solutions existantes pour modéliser ces problèmes sont abordées, en particulier le cadre des MDPs factorisés. Le second chapitre développe nos travaux sur l'amélioration des structures de données utilisées par l'état de l'art pour modéliser les MDPs factorisés.

La seconde partie s'intéresse à la résolution des *problèmes de décisions séquentielles dans l'incertain* avec la nouvelle structure présentée dans la partie précédente. Dans un premier chapitre sont présentés les algorithmes de recherche de solutions optimales, ainsi que leurs adaptations aux problèmes de grandes tailles dans le cadre des MDPs factorisés. Dans un second chapitre, nous développons nos travaux sur l'amélioration d'un des sous-algorithmes utilisés dans la recherche de solutions optimales dans les MDPs factorisés, ainsi que sur l'intégration de la structure de données introduite précédemment aux algorithmes de recherche de solutions optimales.

La dernière partie porte sur l'élaboration d'un nouvel algorithme d'*apprentissage par renforcement* dans le cadre des MDPs factorisés se servant de la nouvelle structure de données et des nouveaux algorithmes de recherche de solutions optimales. Dans un premier chapitre, nous présentons le cadre général de l'*apprentissage par renforcement*, et l'adaptation aux MDPs factorisés. Dans le chapitre suivant, nous parlons de **IMDDI**, un algorithme d'apprentissage de la structure nouvellement introduite. Grâce à ce nouvel algorithme, nous pouvons mettre un algorithme d'*apprentissage par renforcement* exploitant ces nouveaux algorithmes et structure de données. Dans un dernier chapitre, nous présentons quelques travaux d'extension de l'algorithme d'*apprentissage par renforcement* ainsi mis en place. Nous présentons ainsi un nouvel algorithme d'exploration du système. Nous parlons aussi d'une application concrète de ces travaux dans la création d'une I.A. pour le jeu vidéo Starcraft II.

Enfin nous concluons en introduisant des perspectives de recherches futures.

ANR LARDON

Tous ces travaux de recherche s'intègre et ont été financé par l'A.N.R. LARDONS. LARDONS signifie "Learning And Reasoning for Deciding Optimally using Numerical

and Symbolic information" [1]. Ce projet a réuni des chercheurs intéressés d'un côté par les problèmes de décision dans l'incertain et la logique, d'un autre côté par l'apprentissage, le raisonnement et la décision.

LARDONS est fondée par l'Agence Nationale de Recherche sous la subvention ANR-10-BLAN-0215. Elle a commencé officiellement le 13 Décembre 2010 et s'est terminée le 12 Juin 2015.

1. Apprendre et Raisonner pour Décider Optimalement en utilisant des informations Numériques et Symboliques

Première partie

Processus Décisionnels de Markov et Représentations Factorisées

Chapitre 1

Modélisation des problèmes de décision dans l'incertain

Dans ce chapitre, nous allons d'abord aborder en détail la modélisation de problèmes de décision dans l'incertain à l'aide de Processus Décisionnels de Markov . Nous allons voir quel est le formalisme mathématique mis en œuvre pour modéliser ces problèmes. À l'issue de cette présentation, nous poserons les hypothèses restrictives sur les problèmes étudiés dans le cadre de cette thèse. Ensuite, nous étudierons les difficultés liées aux problèmes de grandes tailles, ainsi que les solutions existantes pour y remédier.

1.1 Processus Décisionnels de Markov

Les *processus décisionnels de Markov* (MDPs[1], Puterman [2005]) sont des outils mathématiques de modélisation. Ils décrivent dans un langage formel des systèmes dynamiques dans lesquels un agent au moins interagit avec un environnement stochastique. Dans COFFEE ROBOT, il s'agit du robot qui évolue entre le bureau est la boutique du coin. Dans TAXI, il s'agit du chauffeur qui doit trouver son client dans la ville et l'amener à destination. Ces outils sont utilisés pour poser des problèmes de décisions séquentielles dans l'incertain.

L'aspect décisionnel est porté par l'agent. Il est en effet amené à prendre itérativement des décisions qui vont influer sur le système. Par exemple, dans TAXI, le chauffeur doit choisir à chaque intersection entre poursuivre sur sa trajectoire initiale ou changer de direction, changeant alors la position du véhicule. Les décisions de l'agent doivent l'amener à réaliser un objectif. Dans COFFEE ROBOT, il s'agit de délivrer du café au propriétaire. Dans FACTORY, il faut terminer l'assemblage des deux pièces.

1. MDP : Markov Decision Process

L'aspect incertain est induit par la nature stochastique de l'environnement. Les conséquences des prises de décision de l'agent ne sont alors pas nécessairement connues à l'avance. Ainsi, dans COFFEE ROBOT, il peut pleuvoir ou non. S'il se met à pleuvoir alors que l'agent traverse la rue, celui-ci sera trempé.

Comme nous le verrons dans la Partie II, la modélisation sous forme de MDP permet de trouver les décisions optimales à prendre par l'agent.

1.1.1 Formalisation

Le formalisme MDP s'accompagne de plusieurs notions que nous allons détailler.

États

Le système modélisé est caractérisé par un certain nombre d'éléments (le taxi ou le client dans TAXI par exemple). Chaque élément a un ensemble de configurations dans lesquels il peut être (le client de TAXI peut être à 5 endroits dans la ville). Le recoupement de la configuration courante de tous ces éléments du système permet alors de définir de manière unique un *état*[2] s du système. Dans TAXI, un état possible[3] est *"Le taxi, dont le réservoir n'a plus que 7 unités, est à l'intersection de la 2ème et de la 3ème ; le passager attend au Travail et souhaite se rendre au Théâtre."* L'ensemble de ces états devient l'*espace d'états* $\mathbb{S}$.

Ci-dessous est donné le nombre d'états dans lesquels peuvent être nos problèmes clefs :

COFFEE ROBOT :	$\lvert\mathbb{S}\rvert = 64$ états,
TAXI :	$\lvert\mathbb{S}\rvert = 7000$ états,
FACTORY :	$\lvert\mathbb{S}\rvert = 55296$ états.

Actions

L'agent est régulièrement appelé à prendre des décisions. Chaque décision consiste à choisir une *action*[4] a à effectuer parmi un ensemble d'actions nommé l'*espace d'actions* $\mathbb{A}$.

Les actions permettent de *"contrôler"* l'état courant du système. Grâce à elles, l'agent modifie en effet la configuration actuelle de certains éléments du système. Par

2. Nous noterons aussi s la variable d'état du système. Nous notons $|\mathbb{S}|$ la taille de $\mathbb{S}$ lorsque celle-ci est définissable.

3. État illégal (ou impossible) : Notons qu'un sous-ensemble d'états de $\mathbb{S}$ peut être considéré comme *illégal* (son complémentaire dans $\mathbb{S}$ étant alors *légal*). Un état *illégal* est un état dans lequel le système ne peut être.

4. Nous noterons a la variable d'action de l'agent.

exemple, l'action *"Aller au nord"* permet au chauffeur du problème TAXI de déplacer le véhicule d'une intersection vers le nord. Dès que l'agent a choisi une action, celle-ci est immédiatement effectuée.

La notion de contrôle sur le système doit toutefois être prise avec du recul. D'une part, la nature stochastique de l'environnement fait que l'action peut échouer, entraînant l'inertie du système ; ou elle peut avoir des conséquences inattendues. Par exemple, dans la FACTORY, souder deux pièces qui étaient déjà collées entraîne leur séparation, là où nous nous attendons à ce qu'elles soient toujours assemblées. D'autre part, certaines actions dans certaines configurations ne changent rien. Typiquement, pour le robot de COFFEE ROBOT, acheter du café alors qu'il n'est pas à la boutique le laisse sans café.

Pour chacun des problèmes clés, l'espace d'action [5] de l'agent est :

COFFEE ROBOT : $\mathbb{A} = \{$Bouger, Prendre le parapluie, Acheter du café, Délivrer le café$\}$,

TAXI : $\mathbb{A} = \{$Aller au nord, à l'est, au sud, à l'ouest, Prendre le passager, Déposer le passager, Faire le plein$\}$,

FACTORY : $\mathbb{A} = \{$Peindre A, B, Tailler A, B, Creuser A, B, Polir A, B, Coller, Souder$\}$.

Temporalité

La régularité à laquelle se font les prises de décisions est elle-aussi modélisée. L'*espace des temps* $\mathbb{T}$ définit ainsi les temps t auxquels sont prises les décisions. À chaque instant t, alors que le système est dans un état s_t [6], l'agent doit choisir une action a_t. Cette action est alors immédiatement effectuée par l'agent.

Mettre en place un tel espace est une question difficile. L'action a_t entreprise à l'instant t doit pouvoir se terminer correctement. Mais il faut aussi une certaine réactivité pour s'adapter à des changements qui pourraient survenir et compromettre le système. Il s'agit donc de trouver une bonne granularité du pas de temps.

Autre particularité de $\mathbb{T}$, l'agent peut se voir donner une limite de temps T pour atteindre le but final (dans TAXI, déposer le client à sa destination en moins de 10 pas de temps par exemple). Le problème est alors dit à *Horizon Fini*. Dans le cas contraire, nous considérons que le problème est à *Horizon Infini*. L'existence ou non de cette borne a des répercussions sur la recherche des décisions optimales. En particulier, la question de pouvoir atteindre le but depuis tout état se pose dans le cas des problèmes à *Horizon Fini*.

5. Action inaccessible : Il est possible qu'en certaines parties de l'espace d'états $\mathbb{S}$, l'agent n'ait plus accès à certaines actions de $\mathbb{A}$. Ces actions sont alors dites inaccessibles.

6. Nous noterons s_t et a_t les variables d'état et d'action à l'instant t.

Transitions

Chaque fois que le système avance d'un instant t à l'instant $t+1$, il effectue une *transition*.

Une question importante dans le formalisme MDP est de savoir dans quel état s_{t+1} sera le système à l'issu de la transition. Deux facteurs influent sur cette issue. Le premier est le contrôle que l'agent exerce sur le système au travers de son action. Le second est la nature stochastique de l'environnement ; de fait, il existe toujours une incertitude sur l'état d'arrivée. Pour déterminer dans quel état arrive le système à la fin d'une transition, le formalisme MDP considère que le système modélisé satisfait deux hypothèses.

La première est qu'il est possible de définir une distribution de probabilité conditionnelle sur l'ensemble des états futurs $s_{t+1} \in \mathbb{S}$ sachant l'ensemble des états passés et des actions choisies. Nous ne nous intéressons alors plus à savoir dans quel état nous allons arriver, mais quelle est la probabilité $P(s_{t+1} | \mathcal{H}_t)$ avec $\mathcal{H}_t = \{s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0\}$ d'arriver en un état s_{t+1} donné sachant tous les états visités précédemment et les actions choisies en ces états.

Il est difficile d'établir cette distribution de probabilité car elle dépend de l'historique $\mathcal{H}_t$. Toutefois, la seconde hypothèse nous permet de simplifier ce problème. Elle stipule que la *propriété de Markov* [Markov, 1953] est vérifié par le système.

Définition 1.1 (Propriété de Markov)**.**

Un système stochastique satisfait la propriété de Markov si la distribution de probabilité conditionnelle sur ses états futurs sachant son état courant et ses états passés est indépendante de ses états passés sachant son état courant. En d'autres termes :

$$s_{t+1} \perp\!\!\!\perp \mathcal{H}_{t-1} \mid s_t \tag{1.1}$$

La distribution de probabilité sur l'évolution du système ne dépend alors plus que de l'état courant et de l'action choisie.

$$\forall\ s_t \in \mathbb{S},\ a_t \in \mathbb{A},\ s_{t+1} \in \mathbb{S},\ P(s_{t+1} | \mathcal{H}_t) = P(s_{t+1} | s_t, a_t) \tag{1.2}$$

Notons que cette hypothèse n'est pas toujours vérifiable. Il faut alors avoir recours à des modèles plus généraux qui prennent en compte l'historique comme les Processus Décisionnels de Markov Partiellement Observables (POMDPs, [Cassandra et al., 1994]).

Par la suite, nous appellerons fonction *Transition*, la fonction $\mathcal{T}$ qui, étant donné l'état à l'instant t, l'action choisie et l'état à l'instant $t+1$ [7], nous retourne la probabilité que cette transition ait eu lieu. Ainsi, nous avons :

$$\begin{array}{rccl} \mathcal{T}: & \mathbb{S} \times \mathbb{A} \times \mathbb{S} & \longrightarrow & [0,1] \\ & (s_t, a_t, s_{t+1}) & \longmapsto & P(s_{t+1} \mid s_t,\ a_t) \end{array} \tag{1.3}$$

Récompenses

La dernière notion à introduire sont les *récompenses*. Rappelons qu'un MDP est utilisé pour trouver quelles actions un agent devrait effectuer pour résoudre le problème de décision. Un mécanisme est donc nécessaire pour démarquer les états, actions et transitions qui constituent l'objectif à atteindre du reste. La fonction *Récompense* $\mathcal{R}$ remplie cette tâche.

Il y a plusieurs façons de définir cette fonction. Mais le principe est toujours le même : associer une récompense $r \in \mathbb{R}$ à un fait important. Comme nous le verrons dans la Partie II, résoudre le problème de décision implique alors de chercher à maximiser la somme des récompenses collectées.

La première manière de définir la fonction *Récompense* est de la définir par rapport à l'état courant :

$$\begin{array}{rccl} \mathcal{R}: & \mathbb{S} & \longrightarrow & \mathbb{R} \\ & s & \longmapsto & \mathcal{R}(s) \end{array} \tag{1.4}$$

L'agent est alors récompensé lorsqu'il atteint un certain état. Par exemple, dans le problème TAXI, ce sera d'être en tout état où la position du client correspond à sa destination.

La seconde approche est de récompenser les actions effectuées en des états particuliers.

$$\begin{array}{rccl} \mathcal{R}: & \mathbb{S} \times \mathbb{A} & \longrightarrow & \mathbb{R} \\ & (s, a) & \longmapsto & \mathcal{R}(s, a) \end{array} \tag{1.5}$$

Dans TAXI, il s'agira de récompenser le chauffeur lorsque celui-ci dépose le client alors que sont taxi est à la bonne destination.

La dernière façon de définir est de l'associer à des transitions :

$$\begin{array}{rccl} \mathcal{R}: & \mathbb{S} \times \mathbb{A} \times \mathbb{S} & \longrightarrow & \mathbb{R} \\ & (s', a, s) & \longmapsto & \mathcal{R}(s, a, s') \end{array} \tag{1.6}$$

7. Pour alléger l'écriture, lorsqu'il sera question des variables d'action a et d'état s aux instants t et $t+1$, nous noterons $a_t = a$, $s_t = s$ et $s_{t+1} = s'$, ainsi que $a_{t+1} = a'$.

Dans TAXI, il s'agit de récompenser les transitions où le passager passe du taxi à sa destination.

Comme le montre les exemples, les trois manières de définir la fonction *Récompense* décrivent des faits similaires. Ces trois définitions sont donc interchangeables. Le choix d'une définition plutôt que d'une autre se fait donc pour des raisons de commodité.

Les valeurs attribuées sont généralement des grandeurs positives pour les faits d'intérêts. La valeur nulle est donnée aux cas restants. Certains problèmes distribuent toutefois des valeurs négatives pour des faits particulièrement indésirés. C'est le cas dans TAXI lorsque le chauffeur cherche à déposer le client autre part qu'à sa destination. Éliciter correctement cette fonction *Récompense* est un vrai problème.

Dans la suite de cette étude, nous privilégierons la modélisation sous forme du couple état-action.

Le modèle

Une définition formelle d'un MDP découle de ces notions :

Définition 1.2 (Processus Décisionnel de Markov)**.**

Un MDP modélise un système sous la forme d'un tuple $\langle \mathbb{S}, \mathbb{A}, \mathbb{T}, \mathcal{T}, \mathcal{R} \rangle$ où :

- $\mathbb{S}$: l'espace d'états.
- $\mathbb{A}$: l'espace d'actions.
- $\mathbb{T}$: l'espace de temps.
- $\mathcal{T}$: la fonction *Transition*.
- $\mathcal{R}$: la fonction *Récompense*.

La Figure 1.1 montre une représentation graphique de ce modèle.

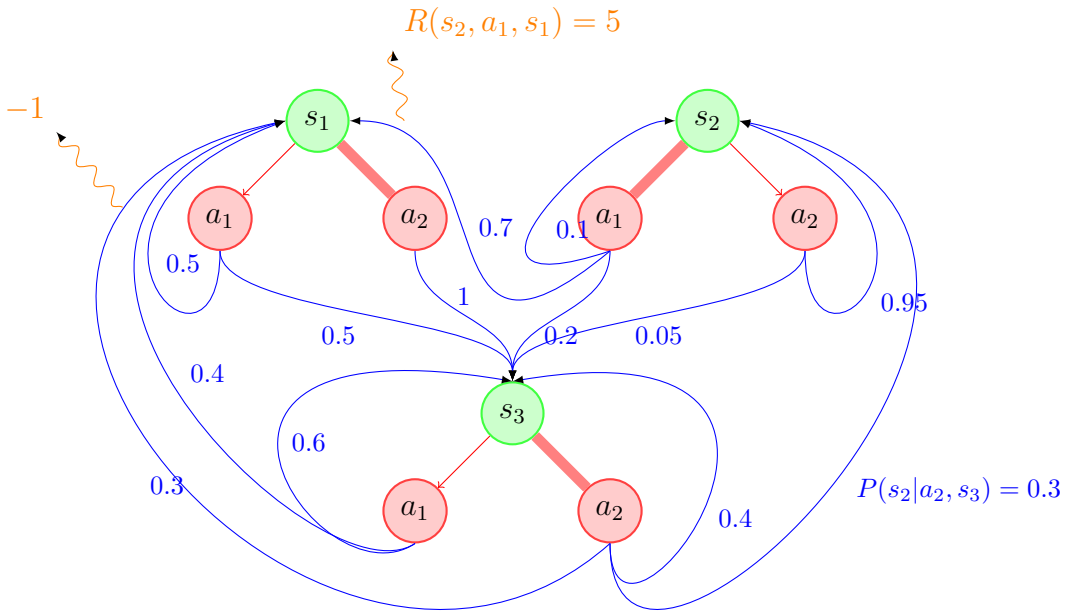
1.1.2 Hypothèses de cette thèse

Dans le reste de cette thèse, nous allons nous intéresser à des MDPs plus spécifiques. Aussi nous allons faire plusieurs hypothèses restrictives.

Espaces Discrets Nous supposerons dans cette étude que les espaces d'états $\mathbb{S}$, d'actions $\mathbb{A}$ et de temps $\mathbb{T}$ sont discrets.

Espaces Finis Nous supposerons aussi que les espaces d'états $\mathbb{S}$ et d'actions $\mathbb{A}$ sont finis.

Sous ces deux conditions, les fonctions *Transition* $\mathcal{T}$ et *Récompense* $\mathcal{R}$ sont représentables à l'aide de tables multidimensionnelles (cf. Figures 1.1b et 1.1c). La fonction



(a) *Représentation Graphique du MDP*

$\mathcal{T}(s, a, s') =$

a_1	s'_1	s'_2	s'_3
s_1	0.5	0.0	0.5
s_2	0.7	0.1	0.2
s_3	0.4	0.0	0.6
a_2	s'_1	s'_2	s'_3
s_1	0.0	0.0	1.0
s_2	0.3	0.3	0.4
s_3	0.0	0.95	0.05

(b) *Fonction Transition* $\mathcal{T}$

$\mathcal{R}(s, a, s') =$

a_1	s'_1	s'_2	s'_3
s_1	0	0	0
s_2	5	0	0
s_3	0	0	0
a_2	s'_1	s'_2	s'_3
s_1	0	0	0
s_2	0	0	0
s_3	-1	0	0

(c) *Fonction Récompense* $\mathcal{R}$

FIGURE 1.1 – *Une représentation graphique (a) du* MDP *d'un système simple avec les fonctions* Transition $\mathcal{T}$ *(b) et récompense* Récompense $\mathcal{R}$ *(c) détaillées sous forme de tables.*

Transition, qui dépend de 3 variables (s, s' et a), est généralement découpée en sous-tables. Une sous-table par action possible pour l'agent est alors créée (soient $|\mathbb{A}|$ tables).

Horizon Infini Nous ferons l'hypothèse que nous travaillons à horizon infini. Autrement dit, l'agent itère indéfiniment ; il n'existe pas de date T à laquelle le processus est interrompu. Cette hypothèse nous servira dans la Partie II.

Stationnarité Dernière hypothèse importante : nous supposerons que les fonctions *Transition* et *Récompense* sont *stationnaires*. Ces fonctions n'évoluent alors pas au cours du temps. En d'autre terme la récompense donnée pour avoir effectué telle action en tel état sera toujours la même ; la probabilité de se trouver en tel état après avoir fait telle action en tel état est invariante.

1.1.3 Fléau de la dimension

Avec cette formalisation, nous rencontrons déjà quelques problèmes liés à la malédiction de la dimension. Ces difficultés sont liées à l'espace d'états $\mathbb{S}$. En effet, pour des problèmes réalistes, cet espace est souvent multi-dimensionnel. Sa taille devient alors rapidement un problème, et ce dès la modélisation du problème.

Un premier challenge apparaît lors de la définition des probabilités de transition et de la fonction *Récompense*. En effet, au vu du très grand nombre d'états possibles du système, il est complexe, voir impossible, d'énumérer toutes les transitions possibles, de déterminer leur probabilité, et éventuellement, si elles sont pertinentes, de leur associer une récompense. L'apprentissage par renforcement factorisé que nous étudierons en Partie III traite ces difficultés.

La seconde difficulté est plus informatique. En effet, pour stocker la fonction *Transition* en mémoire, $|\mathbb{A}|$ matrices de dimension $|\mathbb{S}| \times |\mathbb{S}|$ sont généralement utilisées. La complexité en taille du modèle est donc en $\mathcal{O}(|\mathbb{A}| \times |\mathbb{S}|^2)$. Il est aisé de voir que ces matrices deviennent problématiques pour des problèmes de grandes tailles. Pour FACTORY, par exemple, cela signifie qu'il faut stocker 14 matrices de 55296 par 55296. À 8 octets la valeur numérique (nombre d'octets généralement nécessaires pour stocker un nombre flottant), il faut près de 367 Go pour disposer du problème en mémoire. Et ce n'est pas la version la plus complexe d'un problème relativement simple.

Dans le reste de cette partie, nous allons voir quelles solutions existent pour résoudre ce problème.

1.2 Abstraction

Pour gérer l'espace d'états devenant trop large, les chercheurs se sont intéressés aux procédés d'*Abstraction*. Souvent, plusieurs états se comportent de manière identique d'un point de vue mathématique : les probabilités de transition ou les récompenses pour ces états sont les mêmes. Ces états pourraient alors être regroupés en un seul état. La place requise pour stocker ces fonctions seraient dès lors moins importantes. Plusieurs implémentations de ce concept existe.

1.2.1 MDPs Hiérarchiques

Une première implémentation repose sur une décomposition hiérarchique du système. Dans les MDPs hiérarchiques (HMDP [8],[Parr, 1998], [Hauskrecht et al., 1998], [Guestrin and Gordon, 2002]), le système est morcelé en sous-systèmes organisés autour de tâches à résoudre. Ces sous-systèmes peuvent à leurs tours être découpés en sous-systèmes si une tâche a besoin d'être redécoupée, d'où la structure hiérarchique. Exploitant le paradigme *Diviser pour conquérir*, ces modélisations cherchent des solutions optimales à des problèmes simples pour ensuite les recombiner et avoir la solution optimale du problème plus complexe.

Chaque tâche est vue comme une macro-action pour le MDP parent. Toutefois ces macro-actions prennent du temps à se résoudre (plus d'un pas de temps). Il est donc nécessaire d'élargir la notion de transition pour ces parents : celle-ci se définit entre un instant t et un instant $t+N$ (si l'action met N pas de temps à se résoudre en moyenne). Pour modéliser ces problèmes nous n'utilisons alors plus tout à fait des MDPs mais des semi-MDPs qui prennent en compte ce paramètre N pour chaque action.

Un MDP est utilisé pour chaque tâche à résoudre. Chaque MDP est seulement concerné par la portion de l'espace d'états originel nécessaire pour accomplir la tâche. Dans le problème TAXI, le chauffeur à trois tâches principales à résoudre : *prendre le client*, *déposer le client*, et *faire le plein*. Si nous nous intéressons à la tâche *faire le plein*, celle-ci est déconnectée du client. Seul la position du taxi et le niveau de réservoir restent pertinent. Il s'ensuit une réduction de l'espace d'état à 350 états au lieu des 7000 initiaux pour le MDP dédié à cette tâche. Une première *abstraction* s'opèrent donc au niveau des sous-MDPs.

Une seconde *abstraction* est possible sur les états du MDP parent afin de réduire la taille de son espace d'état [Dietterich, 2000]. Par exemple, dans TAXI, pour la tâche *faire le plein*, il est possible d'abstraire la position du véhicule à 2 états : le véhicule est

8. HMDP : Hierarchical MDP

à la station service, le véhicule n'est pas à la station service. Le MDP parent modélise alors moins d'états.

1.2.2 MDPs Relationnels

Une autre forme d'*abstraction*, basée sur la représentation des connaissances du système et sur l'utilisation de la logique du premier ordre, a aussi été étudiée. Dans les MDPs Relationnels (RMDP [9],Boutilier [2001],Wang et al. [2007]), le système est modélisé à partir des objets et des liens qui les unissent. Chaque relation est formalisée sous la forme d'un prédicat, et ces prédicats sont manipulés à l'aide de la logique du 1er ordre.

Chaque état est caractérisé par une conjonction des prédicats vrais en cet état. Les états sont donc définis sur un nombre non-constant de prédicats. Une première forme d'*abstraction* s'obtient en acceptant des prédicats logiques du 1er ordre. Un état abstrait est alors défini par une conjonction ayant un ou plusieurs prédicats avec variables. De facto, ils couvrent plusieurs états.

La description des fonctions *Transition* et *Récompense* exploite cette représentation. En définissant des états abstraits par introduction de variables, il est alors possible de donner par exemple la même récompense pour plusieurs états concrets.

La représentation des transitions exploitent aussi les opérateurs STRIPS probabilistes [Mcdermott et al., 1993]. Dans le formalisme des opérateurs STRIPS, les prédicats restent inchangés si leur évolution n'est pas précisée dans la description de l'action. Ce mécanisme est appelé persistance. Les prédicats qui évoluent sont regroupés en conjonction. Tout état qui satisfait alors cette conjonction est donc concerné par cet opérateur. Ces opérateurs permettent donc de décrire les transitions pour plusieurs états concrets. D'où une réduction de la taille nécessaire pour décrire la fonction *Transition* du MDP.

Des définitions élaborées sont requises pour les HMDPs et les RMDPs. Dans beaucoup de problèmes, cette connaissance ne peut pas être acquise. Toutefois, ces modélisations mettent en évidences que les états concrets du système se démarquent les uns des autres sur un certain nombre de détails du système qu'il est possible d'exploiter pour réduire la taille de la description du système. Les MDPs Factorisés sont eux aussi basés sur de telles caractérisations.

9. RMDP : Relationnal MDP

1.3 Représentations factorisées

Pour améliorer la compacité des fonctions *Transition* $\mathcal{T}$ et *Récompense* $\mathcal{R}$, une solution est donc de passer en représentation factorisée. Cette transformation est au cœur de cette thèse, aussi allons-nous la voir en détail.

1.3.1 Décomposition de l'espace d'états $\mathbb{S}$

Comme le montre les HMDPs et les RMDPs, des simplifications peuvent être effectuées lorsqu'on s'intéresse aux divers éléments composant le système représenté. Ainsi, dans TAXI, connaître la position du client lorsque le véhicule se déplace n'apporte pas plus d'informations sur l'évolution de la position du véhicule. La fonction *Transition* serait alors grandement simplifiée en évitant de spécifier l'évolution du véhicule pour chaque position du client.

Or la représentation sous forme d'espace d'état $\mathbb{S}$ des configurations du système, qui rend atomique chaque état, ne permet pas de faire cette simplification. Sous cette représentation, *"le véhicule est arrivé à l'intersection de la 2ème et de la 3ème alors que le client est à bord"* est irrémédiablement différent de *"le véhicule est arrivé à l'intersection de la 2ème et de la 3ème alors que le client est au théâtre"*. Pour pouvoir découper les états et isoler les informations pertinentes (*"le véhicule est arrivé à l'intersection de la 2ème et de la 3ème "*), il faut pouvoir décomposer l'espace d'état.

Cette décomposition s'appuie sur un ensemble de variables $\mathbb{X}$[10]. Chaque variable de cet ensemble sert à décrire un élément caractéristique du système. Par exemple, le problème TAXI a une variable pour la position longitudinale du Taxi et une autre pour sa position latitudinale, ainsi qu'une pour la position du client. Pour les problèmes clés, les ensembles de variables sont donnés dans la Table 1.1.

Chaque instance de ces variable (e.g. Position Client = Théâtre) correspond à une configuration possible de la caractéristique représentée (position du client dans la ville). Une instanciation de toutes les variables de $\mathbb{X}$ permet d'identifier une configuration du système. De fait, l'espace d'état $\mathbb{S}$ est décomposable à l'aide de cet ensemble :

10. De manière générale, nous noterons X_i une variable définie sur un domaine fini discret $\mathcal{D}om\{X_i\}$ de cardinal $|X_i|$. Lorsque X_i sera considérée instanciée (i.e. assignée à une valeur donnée de $\mathcal{D}om\{X_i\}$), elle sera notée x_i. Pour alléger l'écriture, lorsque nous évoquerons la variable aux instants t et $t+1$, nous noterons $X_i = X_{i,t}$ et $X'_i = X_{i,t+1}$.

De plus nous noterons $|\mathbb{X}|$, la taille de l'ensemble $\mathbb{X}$, et $|\mathcal{D}om\{\mathbb{X}\}|$ la taille du domaine $\mathcal{D}om\{\mathbb{X}\} = \mathcal{D}om\{X_1\} \times \ldots \times \mathcal{D}om\{X_n\}$.

Définition 1.3 (Décomposabilité de l'espace d'état $\mathbb{S}$)**.**

Un espace d'état $\mathbb{S}$ est *décomposable* si et seulement si $\exists$ un ensemble de variables finies et discrètes $\mathbb{X} = \{X_1, \ldots, X_n\}$ qui caractérise sans équivoque $\mathbb{S}$.

L'espace d'état $\mathbb{S}$ est alors en bijection avec $\mathcal{D}om\{\mathbb{X}\}$. À chaque état $s \in \mathbb{S}$ correspond une instanciation unique de toutes les variables de $\mathbb{X}$, et à chaque instanciation de toutes les variables de $\mathbb{X}$ correspond un unique état $s \in \mathbb{S}$.

Une extension des MDPs qui utilise l'ensemble des variables $\mathbb{X}$ en lieu et place de l'espace d'états $\mathbb{S}$ peut alors être définie :

Définition 1.4 (Processus Décisionnel de Markov Factorisé, Boutilier et al. [1999])**.**

Un Processus Décisionnel de Markov Factorisé (FMDP [11]) modélise donc un système sous la forme d'un tuple $\langle \mathbb{X}, \mathbb{A}, \mathbb{T}, \mathcal{T}, \mathcal{R} \rangle$ où :

- $\mathbb{X}$: l'ensemble de variables caractérisants le système,
- $\mathbb{A}$: l'espace d'actions [12],
- $\mathbb{T}$: l'espace de temps,
- $\mathcal{T}$: la fonction *Transition*,
- $\mathcal{R}$: la fonction *Récompense*,

Grâce à ce changement de représentation, les informations pertinentes (*"le véhicule est arrivé à l'intersection de la 2ème et de la 3ème "*) peuvent être isolée des informations non importantes (*"le client est à bord"* ou *"le client est au théâtre"*). Les *états abstraits* se définissent alors comme des instanciations partielles de l'ensemble de variables $\mathbb{X}$ ($X = 2$; $Y = 3$ pour notre fil rouge par exemple) ; ces états abstraits couvrent l'ensemble des états concrets formulables en instanciant les variables restantes ($PassengerPos$, $PassengerDest$ et $Tank$, soient 280 instanciations/états concrets possibles). Par la suite, nous appellerons CONTEXTE un tel état abstrait.

Supposons que, pour un CONTEXTE donné, les fonctions *Transition* ou *Récompense* sont constantes quelle que soit l'instanciation des variables restantes. En ne stockant que ce CONTEXTE et la valeur alors assumée par les fonctions *Transition* ou *Récompense*, une économie d'espace est réalisée. En effet, il est alors redondant de stocker l'ensemble des états couverts par ce CONTEXTE avec cette même valeur. En appliquant cette astuce à l'ensemble des CONTEXTES pour lesquels les fonctions *Transition* ou *Récompense* sont constantes, l'espace requis pour mémoriser ces fonctions alors grandement réduit.

Toutefois, un autre mécanisme de simplification est applicable auparavant : l'indépendance entres variables. Les fonctions *Transition* et *Récompense* peuvent en effet

11. FMDP : Factored MDP
12. Certains modèles de FMDP admettent aussi un factorisation de l'espace d'actions.

Table 1.1 – *Ensembles de variables* $\mathbb{X}$ *utilisés pour décomposer les problèmes* COFFEE ROBOT, FACTORY *et* TAXI.

Nom	Description	Domaine	Taille
COFFEE ROBOT			64
Hrc	Le robot a du café ?	{Yes,No}	2
Hoc	Le proprio a du café ?	{Yes,No}	2
R	Pleut-il ?	{Yes,No}	2
W	Le robot est mouillé ?	{Yes,No}	2
U	Le robot a un parapluie ?	{Yes,No}	2
Loc	Position du robot	{Office,Shop}	2
FACTORY			55296
$SkilledLab$	Ouvrier Qualifié ?	{Yes,No}	2
$QualityNeeded$	Niveau de Qualité Désiré	{HighQ,LowQ}	2
$SprayGun$	Établi avec SprayGun ?	{Yes,No}	2
$Connected$	Connexion des pièces	{Bonne, Mauvaise, Aucune}	3
$ASmooth$	Pièce A Polie ?	{Yes,No}	2
$BSmooth$	Pièce B Polie ?	{Yes,No}	2
$AShaped$	Pièce A Taillée ?	{Yes,No}	2
$BShaped$	Pièce B Tailée ?	{Yes,No}	2
$Glue$	Pièces Collées ?	{Yes,No}	2
$APainted$	Pièce A peinte ?	{Bien, Mal, Non}	3
$BPainted$	Pièce B peinte ?	{Bien, Mal, Non}	3
$Bolts$	Pièces Soudées ?	{Yes,No}	2
$ADrilled$	Pièces Percées ?	{Yes,No}	2
$BDrilled$	Pièces Percées ?	{Yes,No}	2
TAXI			7000
X	Position Longitudinale du Taxi	{1,..,5}	5
Y	Position Latitudinale du Taxi	{1,..,5}	5
$PassengerPos$	Position Client	{H,W,C,Th,Ta}	5
$PassengerDest$	Destination Client	{H,W,C,Th}	4
$Tank$	Niveau Réservoir	{0,..,14}	14

être totalement indépendantes de certaines variables étant donné certaines autres variables pour certaines actions. Par exemple l'avancement du véhicule dans TAXI est complètement déconnecté de la position du client et sa destination. En exploitant ces indépendances, la taille des tables stockant les fonction *Transition* et *Récompense* est réductible.

Nous allons maintenant voir comment se mettent en place de telles factorisations.

1.3.2 Factorisation des fonctions *Transition* et *Récompense*

Lorsque $\mathbb{S}$ est décomposable à partir d'un ensemble $\mathbb{X}$, les fonctions *Transition* et *Récompense* peuvent être reformulées comme fonctions des variables de $\mathbb{X}$:

$$\mathcal{T}(s,\ a,\ s') = \mathcal{T}(X_1, \ldots, X_n,\ a,\ X'_1, \ldots, X'_n) \tag{1.7}$$

et

$$\mathcal{R}(s,\ a) = \mathcal{R}(X_1, \ldots, X_n,\ a) \tag{1.8}$$

Grâce à ce changement de variables dans les fonctions *Transition* et *Récompense*, les indépendances entre variables inhérentes au système sont exploitables afin de factoriser le MDP.

Factorisation de la fonction *Transition* $\mathcal{T}$

Nous avons donc :

$$\begin{aligned}\mathcal{T}(s,\ a,\ s') &= P(s' \mid s,\ a) \\ &= P(X'_1, \ldots, X'_n \mid X_1, \ldots, X_n,\ a)\end{aligned}$$

Par soucis de simplicité de la présentation, nous ferons l'hypothèse que, à tout instant $t+1$, les variables de $\mathbb{X}$ sont indépendantes entre elles conditionnellement aux variables de $\mathbb{X}$ à l'instant t. La factorisation est aussi réalisable dans le cas où certaines variables sont corrélées mais requière alors quelques adaptations [Boutilier, 1997]. Avec cette propriété, la distribution de probabilités jointe conditionnelle sur les variables de $\mathbb{X}'$ se réécrit comme le produit des distributions de probabilités marginales conditionnelles des variables de $\mathbb{X}'$. Nous avons alors :

Propriété 1 (Indépendances Conditionnelles I)**.**
Si $\forall\ X'_i \in \mathbb{X}', X'_j \in \mathbb{X}'$,

$$X'_i \perp\!\!\!\perp X'_j \mid \mathbb{X}$$

Alors :

$$\mathcal{T}(s,\ a,\ s') = \prod_{i=1}^{n} P(X'_i \mid X_1, \ldots, X_n, a) \tag{1.9}$$

Grâce à cette réécriture, un gain notable a été fait. Par exemple, dans le problème TAXI, nous sommes passés d'une table de 448 millions d'entrées à 5 tables de 784 000

entrées pour la plus grande [13]. Nous sommes néanmoins toujours tributaires de toutes les variables à l'instant t, donc de tout l'espace d'états.

Toutefois, l'exploitation des indépendances conditionnelles va à nouveau nous permettre de réduire le nombre de paramètres dont dépend chaque probabilité marginale. Mais nous allons nous servir d'une autre propriété des indépendances conditionelles :

Propriété 2 (Indépendances Conditionnelles II)**.**

Soient :

— $X'_i \in \mathbb{X}'$,

— *un ensemble* $\text{PARENTS}(X'_i) \subseteq \mathbb{X}$

— *et son complémentaire dans* $\mathbb{X}$ *:* $\overline{\text{PARENTS}(X'_i)}$.

Si :

$$X'_i \perp\!\!\!\perp \overline{\text{PARENTS}(X'_i)} \mid \text{PARENTS}(X'_i)$$

Alors :

$$\begin{aligned} P(X'_i \mid X_1, \ldots, X_n) &= P(X'_i \mid \text{PARENTS}(X'_i),\ \overline{\text{PARENTS}(X'_i)}) \\ &= P(X'_i \mid \text{PARENTS}(X'_i)) \end{aligned} \tag{1.10}$$

La distribution de probabilité sur la variable X'_i ne dépend alors plus que d'un nombre réduit de variables appelées PARENTS de X'_i. Et, comme $|\text{PARENTS}(X'_i)| \leq |\mathbb{X}|$, un gain substantiel lors du stockage mémoire de la distribution de probabilité est possible.

Il est important de noter que l'ensemble $\text{PARENTS}(X'_i)$ des PARENTS de X'_i dépend fortement de l'action a entreprise par l'agent. L'évolution de la caractéristique modélisée par X'_i peut en effet dépendre pour certaines actions de caractéristiques supplémentaires par rapport à l'ensemble de caractéristique dont elle dépend en temps normal. Par exemple, dans COFFEE ROBOT, en temps normal (lorsque le robot est au bureau ou à la boutique) la probabilité d'être trempé à l'instant $t+1$ (variable W') ne dépend que de s'il est mouillé ou non à l'instant t (variable W). Mais lorsqu'il entreprend l'action de traverser la rue, la probabilité d'être mouillé dépend alors aussi de s'il pleut (variable R) et de si le robot a un parapluie (variable U, et alors $\text{PARENTS}_{Bouger}(W') = \{W, R, U\}$). Il est donc nécessaire de définir pour chaque action et pour chaque variable à $t+1$ l'ensemble des variables à t dont la variable dépend.

13. Au total, 3 920 000 entrées sont donc nécessaires contre les 448 000 000 entrées requises initialement.

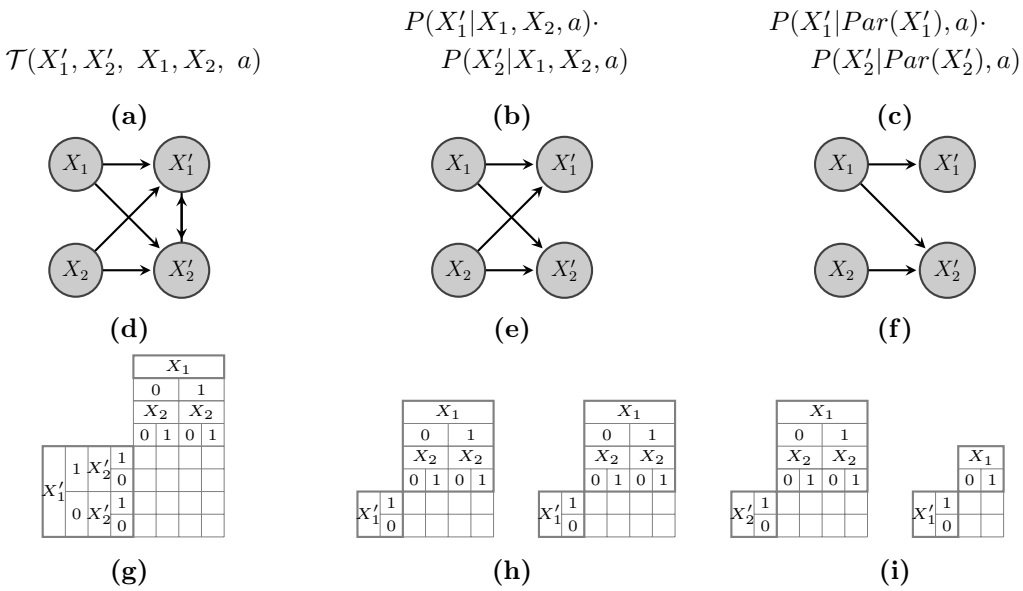


FIGURE 1.2 – *Dans cet exemple, la même distribution de probabilités conditionnelle sur X'_1 et X'_2 sachant X_1, X_2 et a est représentée : comme (a) une distribution de probabilités conditionnelle jointe, (b) un produit de distributions de probabilités conditionnelles marginales, et (c) en exploitant les indépendances conditionnelles.*
Les figures (d), (e) et (f) montrent l'évolution du 2TBN associé au fur et à mesure que sont prises en compte les diverses indépendances conditionnelles.
Enfin, les figures (g), (h) et (i) montrent la réduction de la table utilisée pour stocker la distribution de probabilités. Finalement, nous n'avons besoin de conserver que 12 valeurs au lieu des 16 initiales.

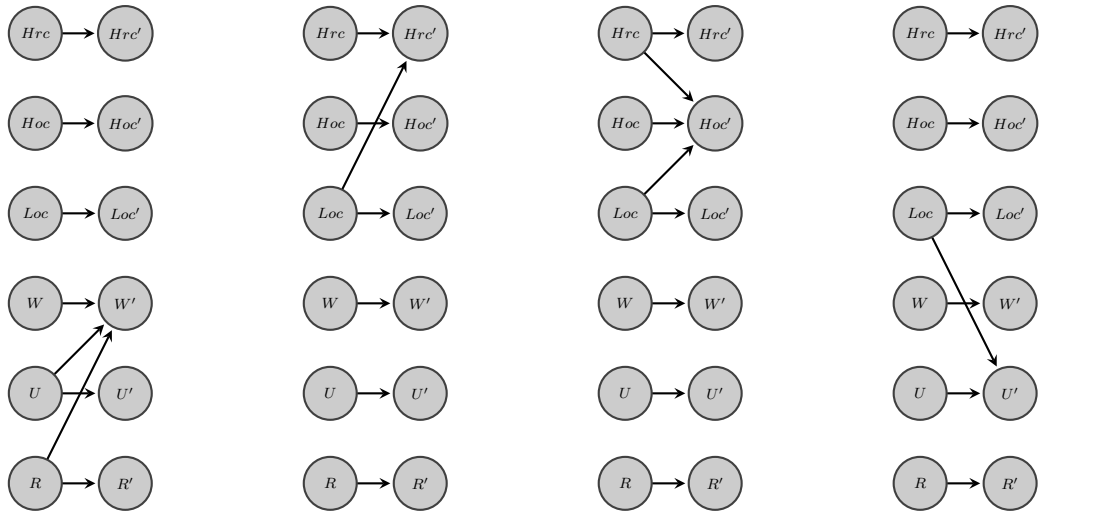
Finalement, nous avons donc :

$$\mathcal{T}(s,\ a,\ s') \ = \ \prod_{i=1}^{n} P(X'_i \mid \text{PARENTS}_a(X'_i), a) \tag{1.11}$$

Ainsi, une large distribution de probabilités sur un trop grand nombre d'états est factorisable en quelques distributions de probabilités dépendantes de beaucoup moins de paramètres. La Figure 1.2 illustre cette factorisation.

La fonction *Transition* est alors représentable à l'aide de $|\mathbb{A}|$ réseaux bayésiens dynamiques à 2 tranches de temps (2TBNs, [Murphy, 2002; Dean and Kanazawa, 1989]). Ces modèles graphiques représentent les dépendances conditionnelles qui existent entre un ensemble de variable $\mathbb{X}$ à un instant t et cet ensemble de variables à l'instant $t+1$ ($\mathbb{X}'$). Un 2TBN est un graphe $G(N, A)$ dirigé sans cycle (DAG [14]) avec un ensemble de nœuds N_t associés aux variables à l'instant t et un ensemble de nœuds N_{t+1} associés aux variables à l'instant $t+1$. Chaque arc (n_i, n_j) d'un 2TBN symbolise la dépendance de la variable n_j par rapport à la variable n_i. Enfin, à chaque variable à $t+1$ s'adjoint

14. DAG : Directed Acyclic Graph



(a) *Bouger* **(b)** *Acheter Café* **(c)** *Délivrer Café* **(d)** *Prendre Parapluie*

FIGURE 1.3 – *Les 2T*BN*s utilisés pour factoriser la fonction* Transition *du problème* COFFEE ROBOT. *Chaque 2T*BN *est associé à une action effectuable par l'agent. Finalement, pour disposer des 16384 valeurs de la fonction originelle, nous n'avons besoin que de stocker 36 (Bouger, a) + 28 (Acheter Café, b) + 36 (Délivrer Café, c) + 28 (Prendre Parapluie, d) = 128 valeurs.*

une table de probabilités conditionnelles (CPT [15]) décrivant la loi de probabilité de cette variable sachant ses parents.

De tels représentations réduisent de façon significative la consommation de mémoire. Pour l'illustrer, reprenons le problème COFFEE ROBOT. Pour représenter la loi jointe pour l'action *Bouger*, il faut $64 \times 64 = 4096$ valeurs. Toutefois, une analyse des indépendances conditionnelles nous indiquent que :

— Hrc' ne dépend que de Hrc,
— Hoc' ne dépend que de Hoc,
— R' ne dépend que de R,
— U' ne dépend que de U,
— W' dépend de R, U et W,
— Loc' ne dépend que de Loc.

Le 2TBN n'a alors besoin que de 5 tables de 4 valeurs et d'une table de 16 valeurs. En tout, il ne faut donc que 36 valeurs pour disposer des 4096 initiales.

15. CPT : Conditionnal Probabilities Table

Factorisation de la fonction récompense *Récompense* $\mathcal{R}$

La fonction *Récompense* peut être décomposée de façon similaire. D'une part, il est possible [16] d'utiliser la décomposition additive (similairement à l'Indépendance Conditionnelle I (1)). D'autre part, les différents termes de cette décomposition ne dépendent pas nécessairement de toutes les variables (similairement à l'Indépendance Conditionnelle II (2)).

Guestrin et al. [2003] formalise cette décomposition. Dans cet article est introduit la notion de PORTÉE, similaire aux PARENTS pour les distributions de probabilités.

Définition 1.5 (PORTÉE)**.**

Une fonction f a un ensemble PORTÉE $(f) \subseteq \mathbb{X}$ si et seulement si :

f : PORTÉE(f) $\rightarrow \mathbb{R}$

En d'autres termes, la PORTÉE d'une fonction multivariable f est l'ensemble des variables strictement nécessaires pour la définir (comme les PARENTS pour une distribution conditionnelle).

Cette notion de PORTÉE appliquée à chaque terme de la décomposition additive permettent de réduire le nombre de paramètres dont ces termes dépendent. Par exemple, dans le problème COFFEE ROBOT, la décomposition additive de la fonction *Récompense* $\mathcal{R}_{DelC}$ lorsque nous nous intéressons à l'action *Delivrer Café* permet d'isoler deux fonctions $\mathcal{R}^1_{DelC}$ et $\mathcal{R}^2_{DelC}$. Puis l'analyse des PORTÉE montrent que PORTÉE $(\mathcal{R}^1_{DelC}) = \{Hrc, Loc, Hoc\}$ et PORTÉE $(\mathcal{R}^2_{DelC}) = \{W\}$.

Factorisation dans Fonctions *Transition* et *Récompense* dans un MDP

Finalement, la factorisation des deux fonctions est procédé comme suit. Tout d'abord, les deux fonctions sont découpées par actions. Pour la fonction *Transition*, ses $|\mathbb{A}|$ sous-fonctions sont ensuite factorisées en $|\mathbb{X}|$ distributions marginales chacune (une distribution marginale par variables de $\mathbb{X}$). Pour cette fonction, il y a donc en tout $|\mathbb{A}| \times |\mathbb{X}|$ tables de tailles variables [17]. Pour la fonction *Récompense*, le nombre de sous-tables dépend de la décomposition additive (si celle-ci a lieu).

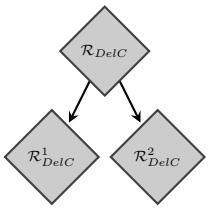
Ces factorisations tendent à minimiser la mémoire utilisée pour stocker le modèle. Toutefois, elles ne sont pas sans poser problèmes. En effet, si nous voulons créer des fonctions composées comme nous allons être appelés à le faire dans la Partie II, nous

16. Alors que la représentation sous forme de 2TBN de la fonction *Transition* est largement utilisée dans les algorithmes de planification, la décomposition additive de la fonction *Récompense* $\mathcal{R}$ l'est moins.

17. Car dépendant de la taille de chaque ensemble PARENTS.

Hrc	Loc	Hoc	W	$\mathcal{R}_{DelC}$
No	Bureau	No	No	1
No	Bureau	No	Yes	0
No	Bureau	Yes	No	1
No	Bureau	Yes	Yes	0
No	Boutique	No	No	1
No	Boutique	No	Yes	0
No	Boutique	Yes	No	1
No	Boutique	Yes	Yes	0
Yes	Bureau	No	No	10
Yes	Bureau	No	Yes	9
Yes	Bureau	Yes	No	1
Yes	Bureau	Yes	Yes	0
Yes	Boutique	No	No	1
Yes	Boutique	No	Yes	0
Yes	Boutique	Yes	No	1
Yes	Boutique	Yes	Yes	0

(a)



(b)

W	$\mathcal{R}^2_{DelC}$
No	1
Yes	0

(c)

Hrc	Loc	Hoc	$\mathcal{R}^1_{DelC}$
No	Bureau	No	0
No	Bureau	Yes	0
No	Boutique	No	0
No	Boutique	Yes	0
Yes	Bureau	No	9
Yes	Bureau	Yes	0
Yes	Boutique	No	0
Yes	Boutique	Yes	0

(d)

FIGURE 1.4 – *Décomposition et factorisation de la fonction* Récompense $\mathcal{R}$ *pour l'action Délivrer Café. Comme* PORTÉE $(\mathcal{R}_{DelC})$= $\{Hrc, Loc, Hoc, W\}$, *la Table (a) nous montre les valeurs de* $\mathcal{R}_{DelC}$ *pour ces variables uniquement. Le graphe (b) nous montre qu'elle est décomposable en deux fonctions* $\mathcal{R}^1_{DelC}$ *et* $\mathcal{R}^2_{DelC}$ *de* PORTÉES *différentes. On a alors les deux Tables (d) et (c) qui résument en 10 valeurs une table initiale de 64 valeurs.*

ne pourrons pas combiner les 2T-BNs et les décompositions additives ensemble. La question de stocker les fonctions composées se pose donc.

De plus, comme PARENTS$(X_i) \subseteq \mathbb{X}$ et PORTÉE $(f) \subseteq \mathbb{X}$, il y a toujours les cas problématiques où PARENTS$(X_i) = \mathbb{X}$ et PORTÉE $(f) = \mathbb{X}$. Les fonctions dépendent alors de toutes les variables. De larges tables doivent alors être stockées en mémoire. L'exploitation des indépendances contextuelles constitue une bonne parade à ces difficultés.

1.3.3 Exploiter les indépendances contextuelles

Une étude plus attentive des fonctions *Transition* et *Récompense* montre que lorsque certains éléments du système sont dans des configurations particulières, ces fonctions assument toujours les mêmes valeurs quelle que soit la situation des autres éléments du système. Ces situations intéressantes sont modélisables à l'aide de la notion de CONTEXTE :

Hrc	Loc	Hoc	$\mathcal{R}^1_{DelC}$
No	Bureau	No	0
No	Bureau	Yes	0
No	Boutique	No	0
No	Boutique	Yes	0
Yes	Bureau	No	9
Yes	Bureau	Yes	0
Yes	Boutique	No	0
Yes	Boutique	Yes	0

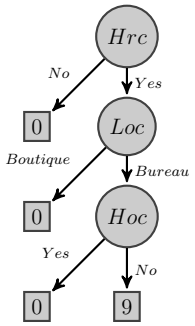
(a)

$\Rightarrow$

Hrc	Loc	Hoc	$\mathcal{R}^1_{DelC}$
No	-	-	0
-	Boutique	-	0
Yes	Bureau	No	9
-	-	Yes	0

(b)

$\Rightarrow$



(c)

FIGURE 1.5 – *La fonction $\mathcal{R}^1_{DelC}$ (c.f. Figure 1.4) peut avoir 8 valeurs possibles (a). Cependant, la fonction récompense montre quelques indépendances contextuelles : par exemple, dans le contexte où le robot n'a pas de café ($Hrc = 0$), la fonction est constante et est égale à 0. Exploiter ces indépendances permettent de la résumer à 4 valeurs (b). Les arbres de décision capturent efficacement ces indépendances (c).*

Définition 1.6 (CONTEXTE)**.**

Un CONTEXTE $\mathbf{c}$ est une instanciation complète d'un sous-ensemble $\mathbf{C} \subseteq \mathbb{X}$.

À un tel CONTEXTE $\mathbf{c}$ est associé de manière inhérente l'ensemble $\{\mathbf{x_c} \in \mathcal{D}om\{\mathbb{X}\} \mid \forall X_i \in \mathbb{X},\ X_i \in \mathbf{C} \Longrightarrow \mathbf{c}[X_i] = \mathbf{x_c}[X_i]\}$ des instanciations complètes de $\mathbb{X}$ compatibles avec ce CONTEXTE. Pour un FMDP, un CONTEXTE définit donc un ensemble d'états qui ont pour particularité d'avoir tous en commun la même configuration d'un sous-ensemble d'éléments du système. Par exemple, dans le problème TAXI, le CONTEXTE $\langle$*PassengerDest*=Theater, *PassengerPos*=Work$\rangle$ regroupe tous les états où le client est au travail et souhaite se rendre au théâtre.

Indépendances Contextuelles

L'indépendance contextuelle d'une fonction f de plusieurs variables à domaines discrets se définit alors comme :

Définition 1.7 (Indépendance Contextuelle)**.**

Soient :

— une fonction f sur un ensemble de variables $\mathbb{X}$,

— un sous-ensemble $\mathbf{C}$ de $\mathbb{X}$, et $\overline{\mathbf{C}}$ son complémentaire dans $\mathbb{X}$,

— un CONTEXTE $\mathbf{c} \in \mathcal{D}om\{\mathbf{C}\}$,

$f \perp\!\!\!\perp \overline{\mathbf{C}} \mid \mathbf{c}$ si $\forall$ instanciation $\overline{\mathbf{c}} \in \mathcal{D}om\{\overline{\mathbf{C}}\}$, $f(\mathbf{c}, \overline{\mathbf{c}}) = cste$

Par exemple, dans Figure 1.5, lorsque $\langle Hrc = No\rangle$, la fonction $\mathcal{R}^1_{DelC}$ prend la valeur 0 quelque soient les instanciations de *Loc* et *Hoc*. Dans le CONTEXTE $\langle Hrc = No\rangle$,

$\mathcal{R}^1_{DelC}$ est donc indépendante de Loc et Hoc (en plus d'être indépendante de W, R et U).

L'intérêt de telles indépendances est immédiat : il n'est alors nécessaire de stocker qu'une seule fois la valeur assumée par la fonction pour les $|\mathcal{D}om\{\overline{\mathbf{C}}\}|$ instanciations que peut prendre le complémentaire $\overline{\mathbf{C}}$. Le gain en place est alors en $|\mathcal{D}om\{\overline{\mathbf{C}}\}| = \prod_{X_i \in \overline{\mathbf{C}}} |\mathcal{D}om\{X_i\}|$.

Une fonction est entièrement définissable à l'aide d'un ensemble de CONTEXTES si, pour chacun des CONTEXTES de cet ensemble, elle est constante. Par exemple, pour $\mathcal{R}^1_{DelC}$, les CONTEXTES $\langle Hrc = No\rangle$, $\langle Loc = Boutique\rangle$, $\langle Hoc = Yes\rangle$ et $\langle Hrc = Yes \wedge Loc = Bureau \wedge Hrc = No\rangle$ suffisent à caractériser les valeurs prises par $\mathcal{R}^1_{DelC}$ en toutes situations (cf Figure 1.5b). Néanmoins, l'ensemble de CONTEXTES utilisé pour définir une fonction présentent potentiellement des redondances : un même sous-ensemble d'états est caractérisé par plusieurs CONTEXTES de l'ensemble. Par exemple, plusieurs états sont compris à la fois dans le CONTEXTE $\langle Hrc = No\rangle$ et dans le CONTEXTE $\langle Loc = Boutique\rangle$: tous ceux où le robot est au magasin mais n'a pas encore de café. Pour écarter ces redondances, les CONTEXTES choisis doivent être disjoints.

Deux CONTEXTES $\mathbf{c}_1$ et $\mathbf{c}_2$ sont dits disjoints s'il n'existe pas d'instanciation complète $\mathbf{x}$ de $\mathbb{X}$ telle que $\mathbf{x}$ est compatible avec $\mathbf{c}_1$ et avec $\mathbf{c}_2$. Soit alors un ensemble $\mathcal{C}$ de CONTEXTES deux à deux disjoints tel que pour tout instanciation complète de $\mathbb{X}$, il existe dans $\mathcal{C}$ un CONTEXTE compatible ; par définition, cet ensemble $\mathcal{C}$ forme une partition de $\mathcal{D}om\{\mathbb{X}\}$. Une fonction f définit sur $\mathcal{D}om\{\mathbb{X}\}$ est alors entièrement définissable à l'aide de cet ensemble $\mathcal{C}$, à la condition que f soit constante pour chaque CONTEXTE de $\mathcal{C}$.

L'ensemble $\mathcal{C}$ de CONTEXTES deux à deux disjoints définissant f est construit de manière récursive. En choisissant une première variable X_1, créer $|\mathcal{D}om\{X_1\}|$ CONTEXTES tel que dans le CONTEXTE $\mathbf{c}_i$, $\mathbf{c}_i[X_1] = x_i$, $x_i \in \mathcal{D}om\{X_1\}$. Puis vérifier chacun de ces CONTEXTES. Si dans le CONTEXTE $\mathbf{c}_k$, f n'est pas constante, choisir une autre variable X_2 et créer $|\mathcal{D}om\{X_2\}|$ nouveaux CONTEXTES tel que dans le nouveau CONTEXTE $\mathbf{c}_l$, on ait $\mathbf{c}_l[X_1] = x_k$ et $\mathbf{c}_l[X_2] = x_l$, $x_l \in \mathcal{D}om\{X_2\}$. Ce processus est répété pour tout CONTEXTE jusqu'à ce que f soit constante pour le CONTEXTE considéré.

ARBRES DE DÉCISION et Indépendances Contextuelles

Ce processus de partitionnement récursif du domaine de définition de la fonction f se représente très bien à l'aide d'*Arbres de Décisions* (cf. Figure 1.5c). Les ARBRES

DE DÉCISION sont des graphes dirigés sans cycles (DAG), composés de nœuds internes associés à des variables et de nœuds terminaux associés aux valeurs alors assumées par la fonction f représentée à l'aide de cette structure. Chaque arc sortant d'un nœud interne correspond à une instanciation possible de la variable liée. Ces arbres sont *enracinés* ; en d'autres termes, un seul nœud est sans parent et est alors appelé racine de cet arbre. Ce nœud est nécessairement interne dès lors que l'arbre possède plus d'un nœud [18].

Propriété 3. *Tout chemin depuis cette racine jusqu'à un nœud terminal correspond à un* CONTEXTE.

Démonstration. En effet, suivre un chemin implique de sélectionner des arcs. Or chaque arc est une instanciation de la variable lié au nœud de départ de cet arc. Il en résulte donc que suivre un chemin de bout en bout implique d'instancier les variables rencontrées sur ce chemin. □

De fait, ce partitionnement ne capture pas efficacement les factorisations sous forme de disjonction de CONTEXTES. Par exemple, la fonction $\mathcal{R}^1_{DelC}$ est constante pour la disjonction $\langle Hrc = No \vee Loc = Boutique \vee Hoc = Yes \rangle$. Pour pouvoir inclure une telle disjonction dans le graphe, il faut autant de CONTEXTES que de littéraux dans la disjonction originelle mais tel que chaque CONTEXTE s'écrit comme un chemin depuis la racine. Dans notre exemple, ces CONTEXTES sont $\langle Hrc = No \rangle$, $\langle Hrc = Yes \wedge Loc = Boutique \rangle$, $\langle Hrc = Yes \wedge Loc = Boutique \wedge Hoc = No \rangle$ et $\langle Hrc = Yes \wedge Loc = Boutique \wedge Hoc = Yes \rangle$ (cf. Figure 1.5c). Le problème est qu'alors, dans l'arbre, il y a autant de duplications d'un même sous-arbre qu'il y a de littéraux dans la disjonction originelle (duplication du nœud terminal 0 dans notre exemple). L'arbre a alors tendance à croître de manière exponentielle.

Boutilier et al. [1995] introduit l'usage des ARBRES DE DÉCISION dans les FMDPs. En s'appuyant sur les factorisations exploitant les indépendances entre variables, Boutilier et al. [1995] propose d'utiliser $|\mathbb{A}| \times |\mathbb{X}|$ ARBRES DE DÉCISION pour représenter la fonction *Transition* de chaque variable pour chaque action, ainsi que $|\mathbb{A}|$ ARBRES DE DÉCISION pour les fonctions *Récompense* associées à chaque action. Toutefois, le fait que les ARBRES DE DÉCISION atteignent très vite une taille très grande limite leur efficacité, en particulier lors de la manipulation des arbres dans la planification (cf. Partie II).

18. Et est nécessairement une feuille dans le cas contraire.

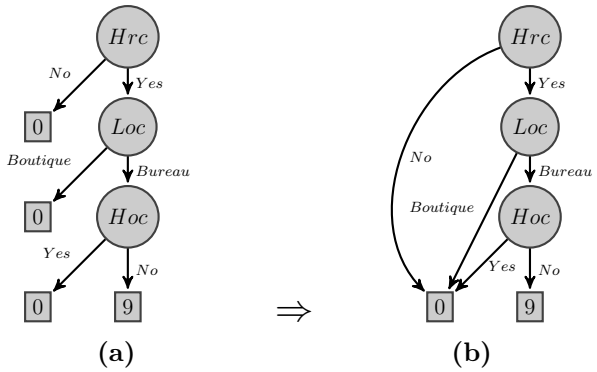


FIGURE 1.6 – *La fonction $\mathcal{R}^1_{DelC}$ (c.f. Figure 1.4) représentée à l'aide (a) d'un* ARBRE DE DÉCISION *puis (b) d'un* ADD *après fusion des sous-parties isomorphes (le nœud* 0 *en l'occurence).*

Diagrammes de Décision Algébriques

Pour palier à cette croissance exponentielle, Hoey et al. [1999] propose alors d'utiliser les *Diagrammes de Décision Algébriques* (ADD [19], [Bahar et al., 1993]) pour représenter les fonctions. Les ADDs (1.6b) sont une généralisation des *Diagrammes de Décision Binaires* (BDDs [20], [Bryant, 1986]) utilisés pour représenter des fonctions réelles à variables booléennes ($f : \mathbb{B}^n \to \mathbb{R}$).

Formellement, les ADDs peuvent être vu comme des ARBRES DE DÉCISION : un ensemble de nœuds terminaux et de nœuds internes, une racine unique, et un ensemble d'arc symbolisant les instanciations des variables liées aux nœuds parents. Toutefois, ils intègrent deux différences importantes : ils sont ordonnés et réduits. Le caractère ordonné signifie que sur tout chemin depuis la racine, les variables sont toujours rencontrées dans le même ordre. Le caractère réduit implique que les sous-arbres isomorphes sont fusionnés ensemble. Le caractère ordonné se justifie par le fait qu'il rend alors la réduction de l'arbre ordonné en un ADD exécutable en un temps polynomial.

La fusion des sous-graphes isomorphes qui s'opère au cours de la réduction permet de prévenir la croissance exponentielle des arbres. De plus, les algorithmes de manipulation des ADDs sont encore plus efficaces que ceux de manipulation des arbres. D'une part la présence d'un ordre sur les variables permet de faire des prédictions sur les variables encore présentes sur les chemins explorés et donc de procéder à des optimisations. D'autre part, la réduction facilitent l'exécution des algorithmes de manipulation sur deux aspects à la fois. De par la taille réduite, un algorithme polynomial en la taille de l'arbre sera plus rapide sur son format réduit. Mais aussi, les sous-arbres isomorphes qui ont été fusionnés ne sont visités qu'une fois par ces algorithmes (cf Partie II).

19. ADD : Algebraic Decision Diagram
20. BDD : Binary Decision Diagram

1.3.4 Observations

Les ADDs sont donc l'état de l'art de la modélisation des FMDPs. Les avantages offerts par ces structures sont multiples : ces structures sont capables de représenter de manière compacte les diverses fonctions d'un FMDP, et leurs manipulations se fait en temps polynomial. Notons toutefois que, comme les ARBRES DE DÉCISION, ces structures ne capturent pas non plus les factorisations sous forme de disjonctions de CONTEXTES.

Des ADDs pour quelles fonctions ?

Les deux indépendances (conditionnelles et contextuelles) sont utilisées pour factoriser la fonction *Transition* $\mathcal{T}$ à l'aide d'ADDs. Nous factorisons donc la fonction *Transition* $\mathcal{T}$ en plusieurs fonctions $\mathcal{T}_{X_i}$ (passage de la loi jointe en un produit de lois marginales). Nous avons ainsi une fonction $\mathcal{T}_{X_i}$ par variable X_i de $\mathbb{X}$. Toutefois, comme les ensembles PARENTS dépendent de l'action entreprise par l'agent lors de l'action, ces fonctions $\mathcal{T}_{X_i}$ se voient découpées en $|\mathbb{A}| \times |\mathbb{X}|$ fonctions $\mathcal{T}_{A,X_i}$. Finalement, nous avons $|\mathbb{A}| \times |\mathbb{X}|$ ADDs pour stocker toutes les probabilités de transition possibles.

Pour la fonction *Récompense*, comme évoqué, la décomposition additive est rarement utilisée. Dans la majorité des cas, il suffit de $|\mathbb{A}|$ ADDs pour représenter la fonction *Récompense* (si les récompenses sont attribuées par couple état-action).

Remarques

Bien qu'état de l'art [Hoey et al., 1999], les FMDPs s'appuyant sur des ADDs pour la modélisation des fonctions doivent faire face à quelques restrictions. Premièrement, les ADDs représentent seulement des fonctions à variables booléennes. Ce qui implique que $\mathbb{X}$ ne soit constitué que de variables booléennes. Par conséquent, toute variable multimodale doit être encodée à l'aide de variables binaires. Cet encodage entraîne un premier problème ; l'espace d'état est artificiellement augmenté. En effet, pour coder une variable X_i à $|X_i|$ modalité, il faut au minimum $\lceil log_2|X_i| \rceil$ variables binaires. Or ces $\lceil log_2|X_i| \rceil$ variables encodent alors $2^{\lceil log_2|X_i| \rceil}$ possibles modalités. Ce qui signifie que $2^{\lceil log_2|X_i| \rceil} - |X_i|$ modalités sont artificiellement créées et n'ont pas de sens concret.

En plus de créer des modalités inexistantes, ces variables binaires augmentent la taille des ADDs. Or comme nous le verrons dans la Partie II, les algorithmes que nous serons amenés à utiliser ont une complexité en la taille des représentations graphiques. Cet encodage, qui augmente donc artificiellement la complexité de ces algorithmes, est donc problématique.

De plus, la taille d'un ADD dépend fortement de l'ordre utilisé pour l'ordonnancer. En effet, l'ordonnancement rend certes plus facilement détectable les sous-arbres isomorphes. Mais cette facilité de détection se fait au prix de la compacité de ces sous-arbres. En effet l'ordre commun imposé à tous ces sous-arbres ne nous permet pas nécessairement de représenter chacun d'entre eux de manière optimale. Le pire cas est un sous-arbre de taille exponentielle, entraînant une taille exponentielle pour l'ADD parent. La Figure 1.7 illustre ce problème. La recherche d'un ordre qui minimise l'ADD est NP-difficile [Friedman and Supowit, 1990] et par conséquence seul des algorithmes de recherche de solutions approchées [Drechsler and Becker, 1998] sont efficaces. Augmenter le nombre de variables rend alors encore plus complexe la tâche de trouver un ordre optimal.

Enfin, dans la Partie III, nous serons amenés à vouloir apprendre les fonctions en s'appuyant sur ces ADDs. Devoir gérer l'encodage lors de cet apprentissage rendra la tâche encore plus complexe.

Offrir la possibilité de manipuler directement des variables multimodales est donc une nécessité. La première contribution de cette thèse est de pouvoir s'affranchir des ADDs booléens sans perdre en puissance de factorisation.

$$f(X_1, X_2, X_3, X_4, X_5, X_6) = X_1 \wedge X_4 \vee X_2 \wedge X_5 \vee X_3 \wedge X_6$$

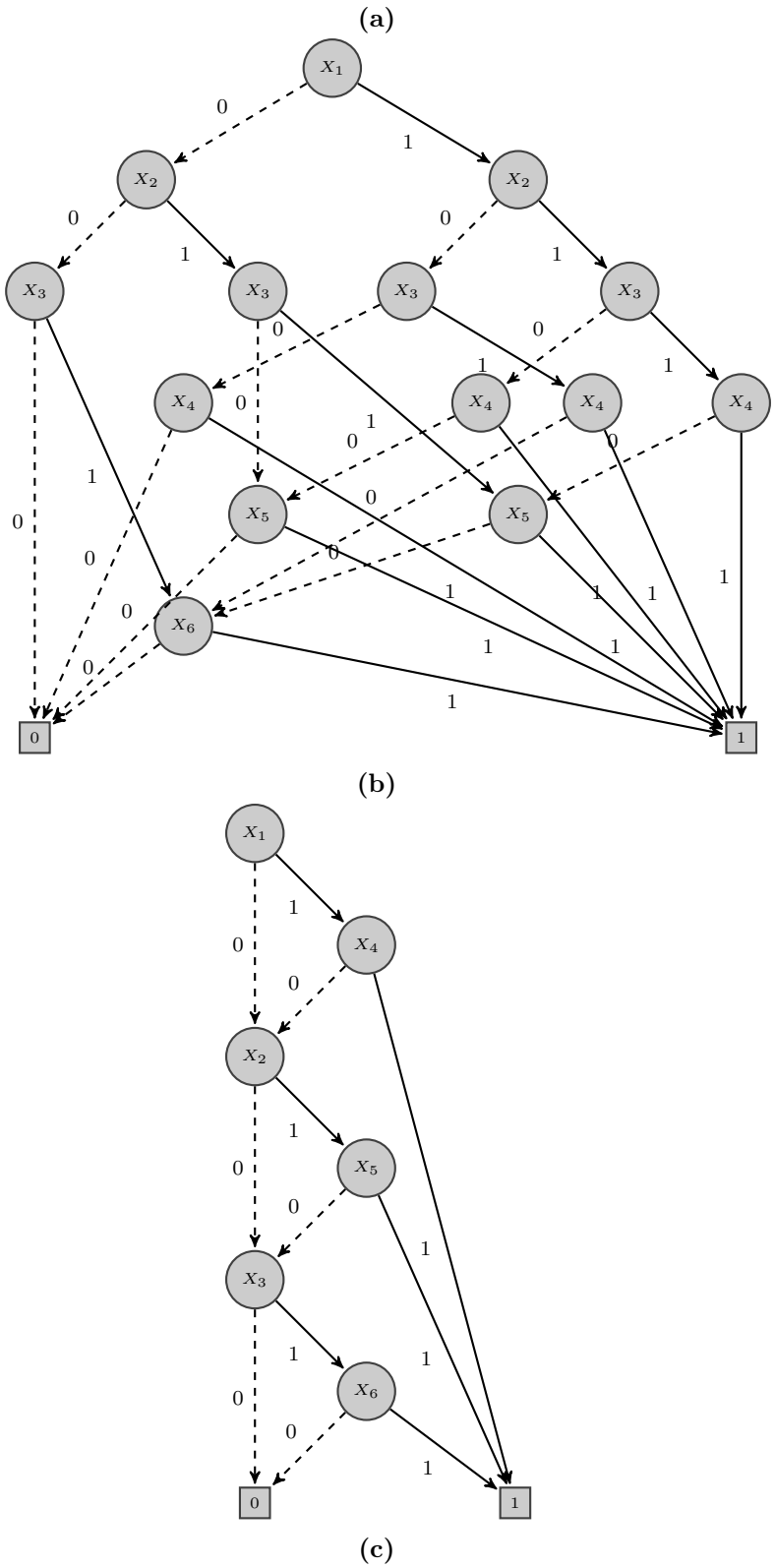


FIGURE 1.7 – *Une même fonction booléenne (a) représentée avec (b) un* ADD *mal ordonnée et (c) un* ADD *correctement ordonné.*

Chapitre 2

Améliorations de la représentation factorisée

Dans le chapitre précédent, nous avons présenté en détails les MDPs. Nous avons aussi parlé des difficultés liées aux tailles des espaces d'états $\mathbb{S}$: ces tailles, généralement très grandes pour les problèmes réels, rendent l'élaboration et la manipulation des MDPs difficiles voir impossibles. Nous avons vu plusieurs techniques pour contourner ces difficultés. En particulier, nous avons détaillé les techniques reposant sur la factorisation dont la plus efficace : la représentation sous forme d'ADDs.

Toutefois les ADDs, bien qu'état de l'art, s'accompagnent de limitations quant aux variables utilisables. En effet, seules des variables binaires sont insérables dans ces représentations graphiques. La solution suivie jusqu'ici est de binariser les variables multimodales. Néanmoins cette solution n'est pas idéale. Outre une mauvaise compression du modèle dans certaines situations, cette représentation pose problème pour le cadre d'apprentissage vers lequel nous nous orientons. Dans ce chapitre, nous proposons plutôt de généraliser les ADDs aux variables multimodales, et nous nous intéressons aux difficultés soulevées par cette généralisation.

2.1 Graphes de décision et variables multimodales

La littérature regorge de structure de données étendant les ADDs aux variables multimodales, notamment dans le domaine de l'apprentissage. Ainsi, Kohavi [1994a] et Kohavi and Li [1995] s'intéressent aux variables multimodales dans le cadre de l'apprentissage d'arbre de décisions. En effet, pour certains problèmes, les représentations arborescentes binaires apprises sont nécessairement de taille exponentielle. En cause, des fonctions booléennes contenues dans ces problèmes comme *majority*, *parity* et *m of*

n dont la représentation sous forme d'arbre binaire est exponentielle. Ces problèmes deviennent de taille polynomiale lorsque des variables multimodales sont utilisées et que l'on autorise la fusion des sous-arbres isomorphes ; la structure obtenue est nommée un *Oblivious read-Once Decision Graph*. Ces structures sont assignées à la représentation de fonctions *k-catégorisés* dans le but de faire de la classification.

En parallèle, Oliver [1992] utilise une structure identique, le *Decision Graph*, également pour la classification. Un telle représentation est choisie car elle est s'accorde bien avec le *Principe du Message de Longueur Minimal* [Wallace and Boulton, 1968]. Cette reformulation du *Principe de Parcimonie* stipule que de tous les modèles qui "expliquent" aussi bien une base de données, le modèle le plus petit est probablement le meilleur. Ces travaux, repris par Oliveira and Sangiovanni-Vincentelli [1995] et Oliveira and Sangiovanni-Vincentelli [1996], montrent que ces structures minimisent la taille des modèles appris tout en ayant une erreur de prédiction équivalente à celle de C4.5.

Enfin, le diagramme de décision multimodale (MDD [1]) de Srinivasan et al. [1990] est la représentation graphique la plus en phase avec nos objectifs. En effet, cette structure est une généralisation des BDDs aux fonctions à variables multimodales et à valeurs dans des domaines discrets et finis. Pour présenter le MDD, Srinivasan et al. [1990] utilise une structure de données plus générale et plus intéressante : la Représentation Graphique de Fonction (RGF). Toutefois Srinivasan et al. [1990] se limite dans sa présentation aux fonctions à valeurs dans des domaines discrets et finis.

Nous allons voir qu'il est possible d'étendre les RGFs à n'importe quel domaine. De fait, une RGF est une généralisation aussi bien des MDDs que des ARBRES DE DÉCISION et des ADDs et remplit le rôle de représentation compacte des fonctions d'un FMDP.

2.2 Représentations Graphiques de Fonction

Avant de donner la définition formelle d'une RGF, il est nécessaire de définir plusieurs objets mathématiques.

2.2.1 Prérequis

Nous manipulons toujours un ensemble de variables discrètes et finies $\mathbb{X} = \{X_1, \ldots, X_n\}$. Pour un système modélisé à l'aide d'un FMDP, ces variables représentent les configurations possibles des caractéristiques du système. Toutefois la définition d'une RGF telle

1. MDD : MultiValued Decision Diagram

que nous allons faire s'inscrit dans un cadre plus large. Cet ensemble peut donc avoir un autre sens suivant le problème dans lequel il est utilisé.

Soit f la fonction :

$$\begin{array}{rccc} f: & \mathcal{D}om\{\mathbb{X}\} & \longrightarrow & \mathbb{E} \\ & (X_1, \ldots, X_n) & \longmapsto & f(X_1, \ldots, X_n) \end{array} \tag{2.1}$$

Nous sommes à la recherche d'une représentation efficace de cette fonction sous forme de graphe. Il n'y a pas de restriction sur l'ensemble $\mathbb{E}$ qui peut aussi bien être $\mathbb{R}$ que l'ensemble $\mathbb{P}$ des distributions de probabilités d'une variable Y conditionnellement à un ensemble $\mathbb{X}$, ou encore l'espace d'action $\mathbb{A}$ comme nous le verrons en Partie II.

La première notion à introduire est la *restriction* de fonction qui permet de définir à partir d'une fonction f une nouvelle fonction $f|$ pour laquelle un sous-ensemble de variables de $\mathbb{X}$ assume une valeur constante dans f.

Définition 2.1 (Restriction d'une fonction)**.**

La restriction $f|_{X_i=b}$ d'une fonction f pour $X_i = b$ est la fonction :

$$\begin{array}{rccc} f|_{X_i=b}: & \mathcal{D}om\{\mathbb{X} \setminus \{X_i\}\} & \longrightarrow & \mathbb{E} \\ & (X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n) & \longmapsto & f(X_1, \ldots, X_{i-1}, b, X_{i+1}, \ldots, X_n) \end{array}$$

Il est possible d'enchaîner les restrictions. En particulier [2] :

$$f|_{X_l=a,\ X_m=b,\ X_n=c} = f(\ldots, a, \ldots, c, \ldots, b, \ldots)$$

Des CONTEXTES sont donc naturellement et directement utilisables pour restreindre une fonction f.

Comme nous avons vu dans le chapitre précédent, l'usage des ensembles PARENTS et PORTÉE est le même : établir le sous-ensemble de variables dont une fonction f dépend réellement. Nous allons donc ici généraliser cette notion :

Définition 2.2 (SUPPORT d'une fonction)**.**

Le SUPPORT de f est le sous-ensemble de variables dont f dépend réellement, i.e.,

$$\text{SUPPORT}(f) = \{X_i \in \mathbb{X} \,|\, \exists u, v \in \mathcal{D}om\{X_i\} \text{ t.q. } f|_{X_i=u} \neq f|_{X_i=v}\} \tag{2.2}$$

Par définition, nous avons donc :

— $\text{SUPPORT}(f) = \text{PARENTS}(Y)$ si f décrit la distribution de probabilités conditionnelle d'une variable Y sachant $\mathbb{X}$,

2. En assimilant une fonction sans paramètre à une fonction constante.

— SUPPORT(f) = PORTÉE(f) si f est une fonction de plusieurs variables.

Remarquons que, les *restrictions* étant des fonctions, elles ont leur propre SUPPORT. Le SUPPORT d'une fonction et celui d'une de ses restrictions sont alors en relation :

Propriété 4 (SUPPORT et Restriction)**.**

$\forall X_i$, SUPPORT$(f|_{X_i=b}) \subseteq$ SUPPORT$(f) \setminus \{X_i\}$.

En effet, le support de la restriction écarte toutes variables qui deviennent non pertinentes en plus de X_i.

Enfin, une notion à étendre est celle de sous-arbre. En effet, la notion de sous-arbre n'est pas applicable directement dans des ADDs. En effet, deux sous-arbres fusionnés dans ces structures peuvent à leur tour avoir des sous-arbres fusionnés. Il s'agit alors de sous-graphes. Toutefois cette notion est beaucoup trop large pour ne pas être ambiguës. En effet, un sous-graphe peut inclure n'importe quel ensemble de nœuds et d'arcs. Or nous voulons caractériser spécifiquement un sous-graphe isomorphe à un arbre enraciné.

Définition 2.3 (Graphe des Descendants)**.**

Soit $G(N, A)$ un graphe dirigé sans circuit (DAG). Le graphe des Descendants d'un nœud $n \in N$ de G est le graphe dirigé sans circuit $G^{\downarrow n}(N_{G^{\downarrow n}}, A_{G^{\downarrow n}})$ définit récursivement :

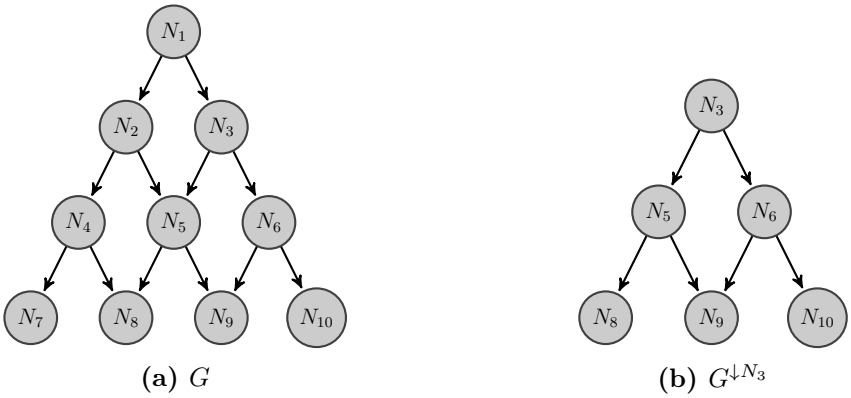
— Si $\nexists m \in N$ t.q. $\exists (n, m) \in A$,
Alors $N_{G^{\downarrow n}} = \{n\}$ et $A_{G^{\downarrow n}} = \{\emptyset\}$

— Sinon :

◦ $N_{G^{\downarrow n}} = \{n\} \cup \bigcup_{\{m \in N \mid \exists (n,m) \in A\}} N_{G^{\downarrow m}}$, et

◦ $A_{G^{\downarrow n}} = \bigcup_{\{m \in N \mid \exists (n,m) \in A\}} [\{(n, m)\} \cup A_{G^{\downarrow m}}]$

La Figure 2.1 illustre l'extraction d'un tel sous-graphe.



(a) G **(b)** $G^{\downarrow N_3}$

FIGURE 2.1 – *Extraction du graphe des Descendants* $G^{\downarrow N_3}$ *de* N_3

Notons que le sous-arbre d'un arbre est un graphe des Descendants pour le nœud racine de ce sous-arbre.

2.2.2 Définition d'une REPRÉSENTATION GRAPHIQUE DE FONCTION

Nous pouvons maintenant définir une RGF comme :

Définition 2.4 (RGF)**.**

Un DAG $G_f(N, A)$ est une RGF pour la fonction f définie en Equation 2.1 si :

— lorsque f est constante t.q. :

$$\begin{array}{rccc} f: & \mathcal{D}om\{X_1\} \times \ldots \times \mathcal{D}om\{X_n\} & \longrightarrow & \mathbb{E} \\ & (X_1, \ldots, X_n) & \longmapsto & e \end{array}$$

Alors G_f est constitué d'un nœud unique n_r ($N = \{n_r\}$) ; De plus, l'élément $e \in \mathbb{E}$ est assigné à n_r ($n_r.val = e$) ;

— lorsque f n'est pas constante,

Alors G_f a un unique nœud $n_r \in N$ sans parents ; De plus :

- $\exists$ une variable $X_r \in$ SUPPORT(f) assignée à n_r ($n_r.var{=}X_r$) ;
- $\forall x_r \in \mathcal{D}om\{X_r\}$, $\exists!$ $n_{x_r} \in N$ t.q. :
  - $(n_r, n_{x_r}) \in A$;
  - le graphe des Descendants $G^{\downarrow n_{x_r}}$ est une RGF pour $f|_{n_r.var=x_r}$.

La Figure 2.2 illustre cette définition.

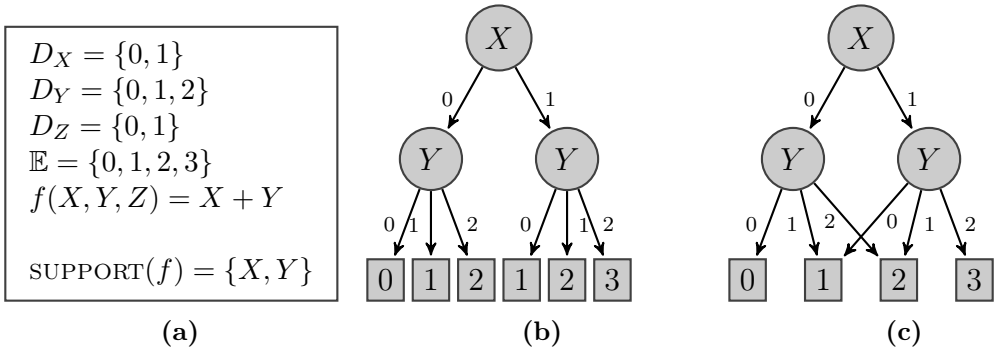


FIGURE 2.2 – *2* RGF*s pour la même fonction.*

Dans une RGF $G_f(N, A)$ d'une fonction f, tout nœud $n \in N$ t.q. n est associé à un élément de $\mathbb{E}$, et est donc sans enfant, est nommé nœud *terminal* ou encore *feuille*. Par opposition, tout autre nœud est dénommé nœud *non terminal* ou encore nœud *interne*.

La fonction **FILS** retourne le fils d'un nœud interne associé à la modalité donnée en argument : $n.\textbf{FILS}(x_i) = n_{x_i}$.

Cette définition s'applique très bien aux ARBRES DE DÉCISION et aux ADDs. En particulier, elle permet de conserver l'équivalence entre chemin du graphe et instanciation de CONTEXTE pour la fonction représentée. Ainsi en un nœud n de G_f, f est restreinte à la fonction $f|_{\text{CONTEXTE}(n)}$; et le graphe des Descendants $G^{\downarrow n}$ est une RGF pour $f|_{\text{CONTEXTE}(n)}$.

Comme l'illustre la Figure 2.2, une même fonction a plusieurs RGFs possibles pour la représenter. Toutefois, elles partagent toutes la même propriété de compacité : aucune variable non pertinente apparaît sur aucun des chemins de la RGF. Cette propriété vient de l'utilisation du SUPPORT(.) pour désigner les variables installables en un nœud interne.

La définition 2.4 introduit également les fusions de sous-graphes. En effet, la clause *"le graphe des Descendants* $G^{\downarrow n_{x_i}}$ *est une* RGF *pour* $f|_{n_r.var=x_i}$*"* permet à plusieurs nœuds de partager un même graphe de Descendants. La seule condition à vérifier est que pour deux nœuds n_1 et n_2,

$$G^{\downarrow n_1.\textbf{FILS}(x_1)} = G^{\downarrow n_2.\textbf{FILS}(x_2)} \iff f|_{n_1.var=x_1} = f|_{n_2.var=x_2}$$

Par exemple, dans la Figure 2.2c, $f|_{X=0,Y=2} = f|_{X=1,Y=1} = 2$. Cette définition, en plus de servir de base à une définition des ADDs, laisse envisager d'autres modèles compacts où les sous-arbres isomorphes sont regroupables même si aucun ordre global sur les branches n'est imposé. Ces modèles resteraient alors exploitables dans le cadre des FMDPs.

2.2.3 REPRÉSENTATION GRAPHIQUE DE FONCTION ORDONNÉE et RÉDUITE

La Définition 2.4 permet de définir des ARBRES DE DÉCISION ayants des nœuds aussi bien associés à des variables binaires que des variables multimodales. Toutefois, cette définition n'impose pas la fusion de sous-graphes isomorphes pour éviter toute redondance d'information. Cette définition est donc incomplète pour décrire un ADD. Il faut lui rajouter deux autres définitions.

Définition 2.5 (RGF Réduite)**.**

Une RGF G_f est réduite $\iff \forall$ nœuds $n \neq n'$,

$$f|_{\text{CONTEXTE}(n)} \neq f|_{\text{CONTEXTE}(n')}$$

Quand une RGF est réduite, si deux graphes de Descendants sont isomorphes alors ils sont nécessairement fusionnés.

Définition 2.6 (RGF Ordonnée)**.**

Une RGF G_f est *ordonnée* $\iff \exists$ un ordre complet $\succ_{G_f}$ sur SUPPORT(f) t.q. $\forall$ nœuds n_1, n_2 non terminaux de G_f,

$$n_2 \in G^{\downarrow n_1} \iff n_1.var \succ_{G_f} n_2.var$$

Quand une RGF est ordonnée, l'algorithme pour la réduire est polynomial. Il suffit en effet alors de vérifier pour chaque nœud que :

1. tous ses fils ne sont pas un seul et même nœud,
2. $\nexists$ un autre nœud lié à la même variable et ayant les mêmes fils.

Cette vérification se fait nœud par nœud, en remontant dans l'ordre $\succ_{G_f}$.

Algorithme 1 : RÉDUIRE : Réduction d'une RGF Ordonnée en une RGFOR

Données : $G_f(N, A)$ une RGF Ordonnée d'une fonction f, $\succ_{G_f}$ l'ordre suivi par G_f

début
  pour $\forall X_i \in$ SUPPORT(f), *en remontant* $\succ_{G_f}$ **faire**
    pour $\forall$ *nœud* $n \in N$ *t.q.* $n.var = X_i$ **faire**
      Soit $x_0 \in \mathcal{D}om\{X_i\}$;
      si $\forall\ x_i \in \mathcal{D}om\{X_i\}$, $n.\textbf{FILS}(x_i) = n.\textbf{FILS}(x_0)$ **alors**
        pour $\forall p \in N$ *t.q.* $\exists (p, n) \in A$ **faire**
          $A \leftarrow A \setminus \{(p, n)\} \bigcup \{(p, n.\textbf{FILS}(x_0))\}$;
          $N \leftarrow N \setminus \{n\}$;
      si $\exists\ n' \in N$ *t.q.* $\forall\ x_i \in \mathcal{D}om\{X_i\}$, $n.\textbf{FILS}(x_i) = n'.\textbf{FILS}(x_i)$ **alors**
        pour $\forall p \in N$ *t.q.* $\exists (p, n') \in A$ **faire**
          $A \leftarrow A \setminus \{(p, n')\} \bigcup \{(p, n)\}$;
          $N \leftarrow N \setminus \{n'\}$;

Résultat : G_f Ordonné et Réduit

L'Algorithme **RÉDUIRE** (1) donne les détails. La Figure 2.3 illustre les deux situations où un nœud $N_{redondant} \in N$ est redondant et peut être retiré du graphe.

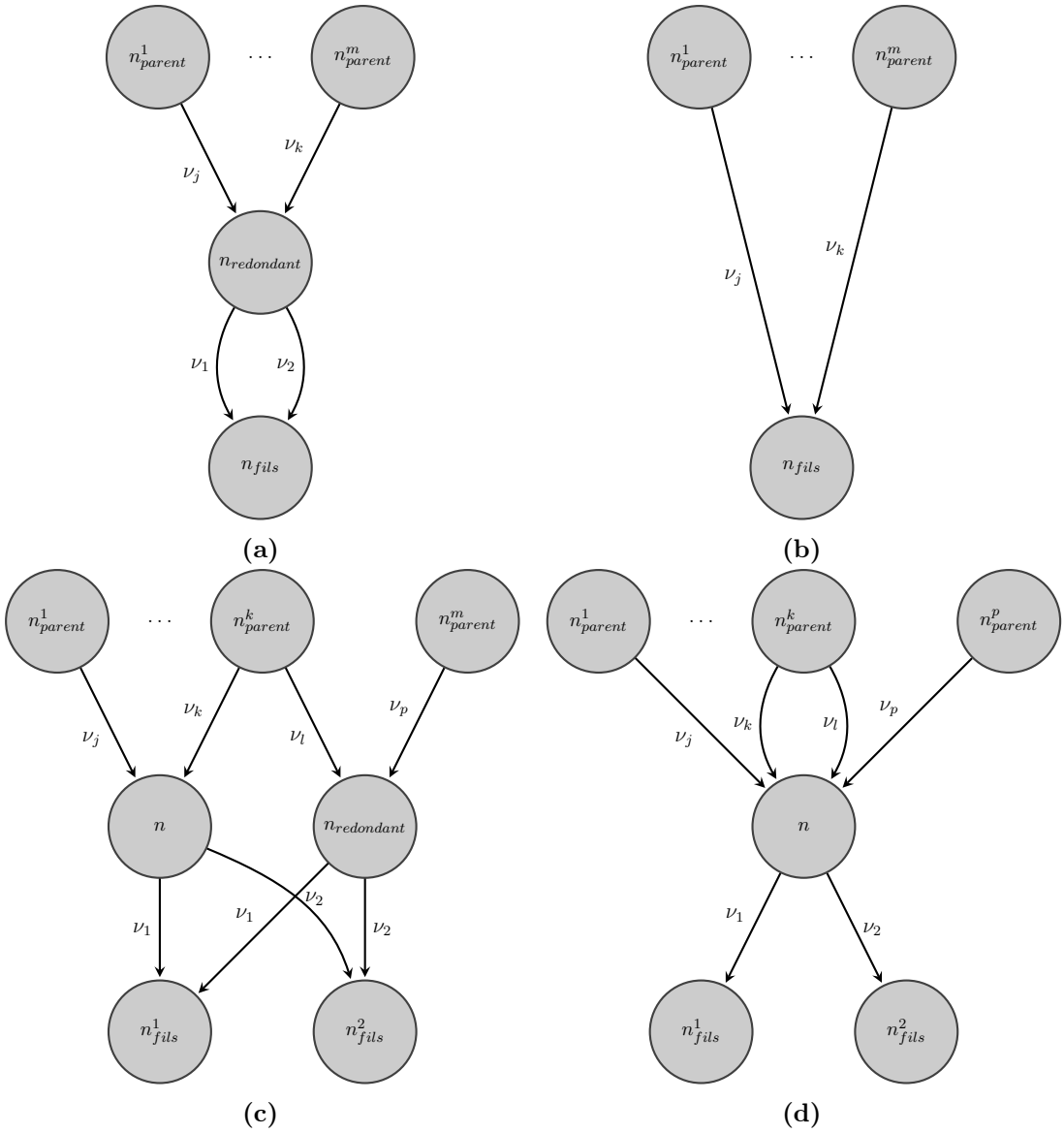


FIGURE 2.3 – *Cas possible de redondance de nœud lors de la réduction d'une* RGF *Ordonnée. Le premier cas (a) est celui où un nœud à tous ses fils pointant vers un même nœud fils. Ce nœud est alors remplacer par des arcs allant de ses parents à son unique fils (b).*
Dans l'autre cas (c), 2 nœuds liés à la même variable (n et $n_{redondant}$) ont les même fils pour les mêmes modalités de la variable. L'un des deux nœuds est retiré du graphe (d). Ses parents pointent alors vers l'autre nœud

Ces trois définitions nous permettent de définir un ADD comme une RGFOR. Mais plus intéressant encore, elles permettent d'intégrer naturellement des représentations compactes avec variables multimodales au cadre de travail factorisé. En effet, les RGFORs reposent sur les notions que nous avons exploitées pour factoriser les FMDPs.

2.3 Minimisation de la taille des RGFORs

Avant de valider l'intégration de RGFORs dans le cadre factorisé des MDPs pour modéliser un problème, il nous faut évoquer le problème de la minimisation de ces RGFORs. Comme le montre la Figure 1.7, les ADDs, et par extension les RGFORs, sont sensibles à l'ordonnancement des variables. Or trouver un ordonnancement optimal est NP-difficile [Friedman and Supowit, 1990]. Des heuristiques existent pour trouver un ordre efficace sur des BDDs. Ces heuristiques donnent aussi de bons résultats avec les ADDs. Aussi avons-nous cherché à les appliquer aux RGFORs.

2.3.1 Heuristiques de recherche d'ordre global efficace

Les heuristiques sont divisées en deux catégories. La première ([Fujita et al., 1988; Malik et al., 1988; Minato et al., 1990]) est plus communément appelée *Ordonnancement des Variables Spécifiquement à l'Application* (ASVO [3]). Elle est utilisée en Conception de Circuit pour trouver un ordre efficace en se basant sur les caractéristiques des circuits représentés. Elle ne peut donc pas réellement nous convenir. D'une part, il faudrait pouvoir généraliser les caractéristiques exploitables d'un circuit à n'importe quelle situation modélisée par un MDP, ce qui n'a pas de sens. D'autre part, elle ne s'inscrit pas dans notre objectif à long terme qui est de découvrir le modèle *ad hoc*. Ces heuristiques demandent une connaissance précise du modèle à l'avance.

La seconde catégorie ([Fujita et al., 1991; Ishiura et al., 1991; Rudell, 1993]), en revanche, nous intéresse plus. Communément appelée *Heuristique d'Ordonnancement Dynamique des Variables* (DVOH [4]), elle ne requière aucune connaissance a priori de la fonction représentée, et cherche par permutation à trouver un ordre efficace. Deux algorithmes principaux existent dans cette catégorie : **PERMUTATION FENÊTRÉE** et **TAMISAGE**.

Dans l'algorithme **PERMUTATION FENÊTRÉE** [Fujita et al., 1991], une fenêtre de taille fixe m est déplacée sur les n variables à ordonner. Pour chaque position de la fenêtre, par exemple les m premières variables, toutes les permutations possibles des m variables (soit $!m$) sont testées. La permutation de meilleur gain en réduction de taille est mémorisée. Ce processus est répété pour les $n - m + 1$ positions possibles de la fenêtre. La permutation de meilleur gain est alors effectuée. Puis l'algorithme repart de la position originelle de la fenêtre et répète ce processus. Il s'arrête lorsque plus aucun gain n'est possible.

3. ASVO : Application Specific Variable Ordering
4. DVOH : Dynamic Variable Ordering Heuristic

Algorithme 2 : **TAMISAGE** : Minimisation d'une RGFOR

Données : $G_f(N, A)$ une RGFOR, $\succ_{G_f}$ l'ordre suivi par G_f

```
1  début
2      Établir ≻_sifting en classant par ordre décroissant les variables suivant le
         nombre de nœuds qui leur sont associés dans G_f;
3      pour ∀X_i ∈ SUPPORT(f), en suivant ≻_Sifting faire
4          TailleMin ← |G_f|;
5          MeilleurPosX ←Position Courante de X_i ;
6          tant que ∃X_j adjacent t.q. X_j ≻_{G_f} X_i faire
7              PERMUTER(X_j,X_i);
8              si TailleMin > |G_f| alors
9                  TailleMin ← |G_f|;
10                 MeilleurPosX ← Position Courante de X_i ;
11         tant que ∃X_j adjacent t.q. X_i ≻_{G_f} X_j faire
12             PERMUTER(X_j,X_i);
13             si TailleMin > |G_f| alors
14                 TailleMin ← |G_f|;
15                 MeilleurPosX ← Position Courante de X_i ;
16         Amener par permutation X_i à MeilleurPosX;
```

Résultat : G_f de taille plus petite

L'Algorithme **TAMISAGE** (2) cherche pour toute variable la meilleure position dans l'ordre actuelle. L'algorithme échange donc la position de cette variable avec toutes les autres variables. À la fin, il la place à la position de gain maximale en taille. Puis le processus est répété pour toutes les variables. L'ordre dans lequel l'algorithme teste les variables est déterminé initialement : elles sont sélectionnées par ordre décroissant du nombre de nœuds associés dans le graphe. Cet algorithme, et ses extensions, est l'état de l'art de la minimisation heuristique des BDDs. De plus, St-Aubin et al. [2000] souligne son efficacité pour minimiser les ADDs employés dans les FMDPs. Aussi est-ce celui-là que nous avons cherché à adapter.

2.3.2 La méthode PERMUTER

L'Algorithme **TAMISAGE** (2) [Rudell, 1993], tout comme l'algorithme **PERMUTATION FENÊTRÉE**, repose sur la méthode **PERMUTER** qui échange deux variables adjacentes dans l'ordre $\succ_{G_f}$ et modifie le graphe en conséquence pour qu'il respecte ce nouvel ordre. La Figure 2.4a montre que dans le cas binaire, cette permutation se fait très simplement. Toutefois, en variables multimodales, les choses ne sont pas aussi

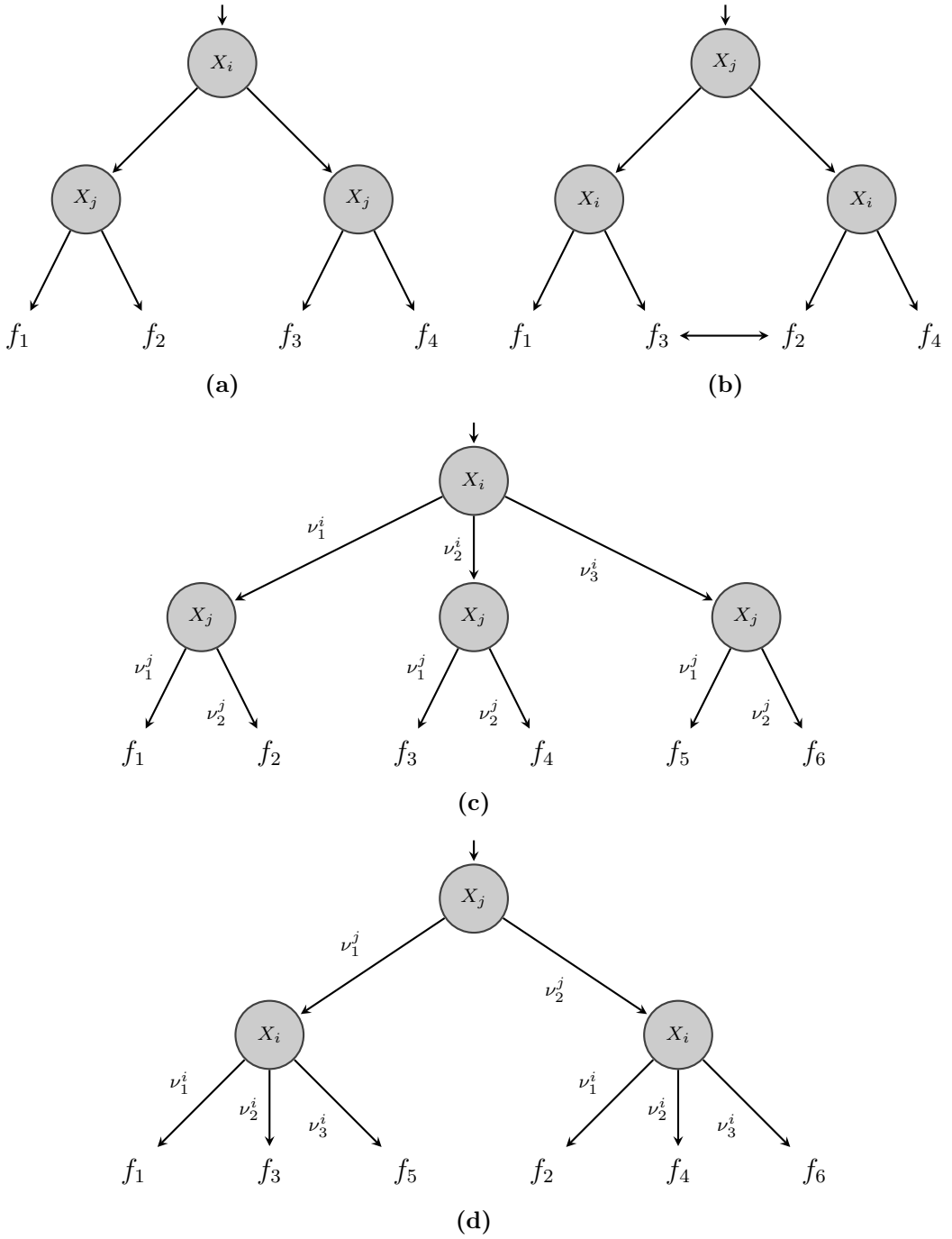


FIGURE 2.4 – *Permutation de variables dans une* RGF *Ordonnée. Le cas binaire (a ⟶ b) consiste à simplement échanger deux nœuds fils. Le cas multimodale (c ⟶ d) demande plus de travail.*

simples (c.f. Figure 2.4c), en particulier lorsque pour les deux variables adjacentes X_i et X_j que nous cherchons à permuter, $|X_i| \neq |X_j|$.

Pour que la permutation soit valide, il faut s'assurer que tout chemin $n_i \xrightarrow{x_i} n_j \xrightarrow{x_j} n_f$ soit remplacé par un chemin $n_i \xrightarrow{x_j} n \xrightarrow{x_i} n_f$. Nous garantissons alors qu'en tout

Algorithme 3 : PERMUTER : Échange de deux variables adjacentes d'une RGFOR

Données : $G_f(N, A)$ une RGFOR, X_i et X_j deux variables adjacentes à échanger tel que $X_i \succ_{G_f} X_j$ initialement

```
1  début
2      pour ∀n ∈ N t.q. n.var = X_i faire
3          Mémoriser anciens enfants ;  // La fonction FILS PRECEDENT permet alors
             de retrouver ces enfants
4          n.var ← X_j;
5          pour ∀x_j ∈ Dom{X_j} faire
6              Insérer un nœud n_i t.q. n_i.var = X_i;
7              pour ∀x_i ∈ Dom{X_i} faire
8                  n_i.FILS(x_i) ← n.FILS PRECÉDENT(x_i).FILS(x_j) ;
9              n.FILS(x_j) ← n_i;
10     Échanger X_i et X_j dans ≻_{G_f};
```

Résultat : G_f avec X_i et X_j échangées

nœud n_f se trouve toujours la même restriction. Cette permutation s'opère donc localement au niveau des nœuds associés aux variables X_i et X_j.

Dans un premier temps, la variable associée à tout nœud n_i est modifiée : la variable associée devient X_j. Ensuite, tous les arcs (n_i, n_j) sont retirés car ils ne sont plus valides. Si un nœud n_j devient sans parent alors il est retiré du graphe, ainsi que tout arc (n_j, n_f). Des nœuds n associés X_i sont enfin insérés ainsi que les arcs nécessaires pour créer les chemins de remplacement.

L'Algorithme **PERMUTER** (3) décrit la méthode **PERMUTER** adaptée aux variables multimodales.

2.4 Utilisation des RGFORs dans un FMDP

Représenter les fonctions *Transition* et *Récompense* d'un FMDP à l'aide de RGFORs avec variables multimodales se fait de la même manière qu'avec des ADDs (sans phase de binarisation toutefois). Les deux fonctions sont donc "découpées" par actions, la fonction *Transition* étant aussi "découpée" par variables, puis ces diverses sous-fonctions sont modélisées à l'aide de RGFORs. Nous allons voir dans quelles proportions cette nouvelle représentation est plus avantageuse que les ADDs.

2.4.1 Procédures de Test

Les RGFORs multimodales sont comparées aux ADDs. Les tailles prisent en mémoire par les deux modèles dépendant fortement de choix d'implémentation, nous privilégions ici le nombre de nœuds internes requis par chaque structure de données comme mesure. Dans un premier temps nous faisons ces comparaisons sur des modèles générés aléatoirement, puis nous observons comment les problèmes clés sont représentés. Pour permettre une comparaison juste, nous avons appliqué l'Algorithme **TAMISAGE** (2) aux deux représentations afin d'obtenir des graphes de tailles aussi petites que possible.

Génération aléatoire de modèle

Pour la phase d'étude de modèles générés aléatoirement, nous avons d'abord créé des RGFORs puis nous les avons binarisées. L'algorithme de génération en lui-même repose sur une exploration *profondeur d'abord* récursive, sauf qu'au lieu d'explorer les fils, ceux-ci sont générés à chaque appel récursif. Ainsi, en chaque nœud, en commençant par la racine, une variable est sélectionnée dans une séquence de variables générée en amont. [5] Ensuite pour chaque modalité de cette variable, un fils est créé. Puisque nous sommes en création profondeur d'abord, nous nous occupons immédiatement du fils créé (par appel récursif) avant de poursuivre la création des autres fils. Deux possibilités alors : soit nous créons un nœud terminal, soit nous créons un nœud interne.

La probabilité de création d'un nœud terminal est fonction du nombre de variables encore exploitables dans la séquence de variables. Dans le cas où il ne reste plus de variable, cette probabilité est bien évidemment de un. Dans le cas où il est décidé d'installer un nœud terminal, un nouveau test aléatoire est effectué : il s'agit en effet de décider si nous utilisons un nœud terminal déjà installé où si nous en créons un nouveau (avec une nouvelle valeur donc). Ceci doit être fait afin de simuler le fait que plusieurs contextes peuvent rendre f constant à une même valeur.

La création d'un nœud fils interne est similaire. Dans un premier temps, une variable à attacher à ce nouveau nœud est choisie. Puis, si cette variable a déjà d'autres nœuds associés, un test est donc effectué pour savoir si nous réutilisons ou non un de ces nœuds. Cet artifice permet de simuler les sous-graphes isomorphe fusionnés. Si nous décidons de prendre un nœud déjà existant, celui-ci est choisi aléatoirement parmi ceux existants et correspondants. Auquel cas, un arc est tiré depuis le parent vers ce nœud. L'appel récursif se termine alors, et on retourne au parent pour lui attribuer un autre

5. Notons qu'alors toute variable qui précède cette variable n'est pas utilisée dans les appels récursifs descendants de l'appel courant.

fils pour une autre modalité. Dans le cas contraire, nous sommes dans la situation décrite plus haut.

Deux méta-paramètres conditionnent cette génération des graphes. Le premier est le nombre de variables utilisées. L'algorithme de génération dispose d'une séquence de variables dans laquelle il sélectionne la prochaine variable à installer dans le graphe. Pour disposer d'une plus grande variété de graphes, la taille de cette séquence est variable. En effet, la profondeur maximale du graphe est modulée par ce nombre de variables puisque dans une RGF, chaque variable n'est rencontrée qu'au plus une et une seule fois dans tout chemin donné depuis la racine.

Le second méta-paramètre est la taille du domaine de chaque variable. Là encore nous avons fixé une limite supérieure et laissé l'algorithme choisir aléatoirement la taille du domaine de chaque variable à sa création. Là où le précédent paramètre jouait sur la profondeur du graphe, ce paramètre joue sur la largeur du graphe puisque pour une variable donnée X_i, chaque nœud associé a au plus $|X_i|$ fils.

Binarisation

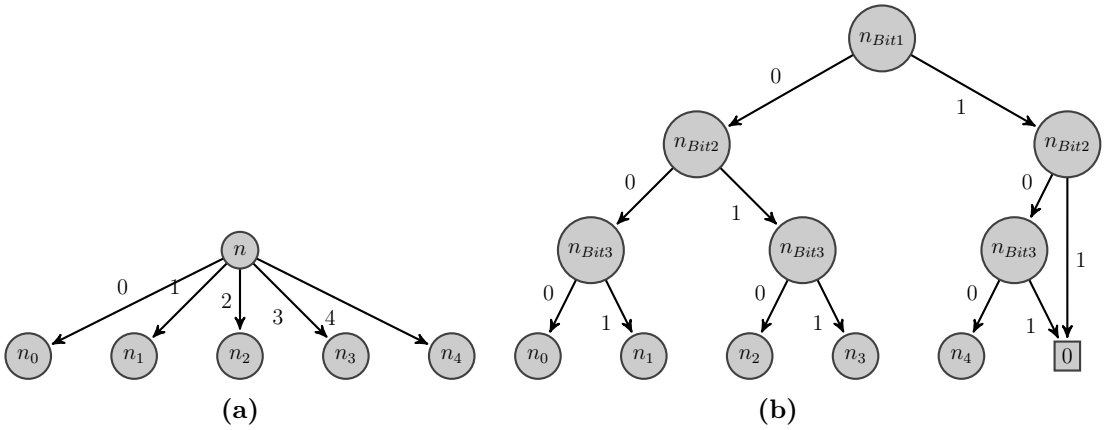


FIGURE 2.5 – *Binarisation d'un nœud associé à une variable multimodale*

Une fois la RGFOR générée, il s'agit de binariser les variables qui la compose pour obtenir la version ADD. Comme pour la permutation de deux variables, cette opération peut se faire niveau par niveau sans affecter le reste du graphe. Il s'agit en effet de juste faire en sorte que la binarisation d'un nœud amène correctement à chacun de ses fils.

Dans un premier temps, pour transformer une variable X_i en un ensemble de variables binaires, chaque modalité de X_i est associée à un unique nombre entre 0 et $|\mathcal{D}om\{X_i\}| - 1$. Par exemple, la variable $PassengerPos$ du problème TAXI a cinq modalités. À ces cinq modalités sont associées des nombres (index) entre 0 et 4 pour les

identifier rapidement. Le nombre attribué le plus grand sert alors de base à la binarisation.

Une variable binaire va en effet être associée à chacun des bits nécessaires pour représenter ce nombre en base 2. La séquence de variables binaires obtenues contient donc $\lceil log_2|\mathcal{D}om\{X_i\}|\rceil$ variables. Par exemple, le nombre 4 en binaire se code sur trois bits donc la séquence a trois variables binaires. L'utilisation d'une séquence est ici importante. Il s'agit d'associer toujours la même variable pour le bit de poids fort, et la même pour le bit de poids faible.

Chaque nœud associée à la variable multimodale binarisée est alors remplacée par une RGFOR avec les spécificités suivantes :

— chaque nœud de cette RGFOR est associée à une des variables binaires utilisées pour la binarisation ;
— dans chaque chemin, les variables se succèdent dans l'ordre de la séquence susévoquée, donc de la variable "Bit poids fort" vers la variable "Bit poids faible" ;
— chaque chemin étant une instanciation des variables binaires, un nombre en base 10 est associable à cette instanciation. Si ce nombre est inférieur à la taille du domaine de la variable multimodale binarisée, alors il représente une des modalités de la variable multimodale. En conséquent, le chemin doit amener au nœud du graphe originel associé à cette modalité. Dans le cas contraire, il doit être associé à un nœud terminal par défaut défini à l'avance.

Ce processus est aisément programmable de manière récursive. La Figure 2.5 illustre le processus.

2.4.2 RGFORs vs ADDs

Pour établir les comparaisons qui suivent, nous avons fait varier les deux méta-paramètres "Nombre de Variables Maximal" et "Nombre de Modalités Maximal". Puis nous avons effectué mille générations de diagramme pour chaque paire de méta-paramètres. Chaque diagramme a ensuite été binarisé pour avoir la version binaire. Nous avons fait varier le nombre de variables entre 5 et 15 par incrément de 5, et le nombre de modalités entre 3 et 10 par incrément de 1.

La Table 2.1 montre le rapport Taille RGFOR/ Taille ADD en moyenne pour chaque couple de méta-paramètre.

Les résultats montrent que la réduction en taille suit les deux méta-paramètres. La principale source de gain est sur le nombre de modalités maximum : la réduction en taille dépasse les cinquante pourcents pour les cas où les variables ont dix modalités maximums. Ajoutons que l'écart-type se réduit lui aussi en fonction du nombre maximal

TABLE 2.1 – *Comparaison en taille des* RGFORs *multivaluées par rapport aux* ADDs

Nb Max Variables	Nb Max Modalités	Réduction moyenne
5	3	94.2% ±1.8%
5	4	70.21% ±11.07%
5	5	66.1% ±7.64%
5	6	58.49% ±7.85%
5	7	53.51% ±6.96%
5	8	51.39% ±6.02%
5	9	50.56% ±5.33%
5	10	47.53% ±5.44%
10	3	95.02% ±1.37%
10	4	67.39% ±10.27%
10	5	64.12% ±6.86%
10	6	54.22% ±6.97%
10	7	49.99% ±6.3%
10	8	47.58% ±5.42%
10	9	46.05% ±5.04%
10	10	43.27% ±5.28%
15	3	95.26% ±1.2%
15	4	66.96% ±9.22%
15	5	62.36% ±6.19%
15	6	52.31% ±7.21%
15	7	48.56% ±6.07%
15	8	45.91% ±5.36%
15	9	44.38% ±4.95%
15	10	41.49% ±5.12%

de modalités. Ces résultats soulignent clairement l'efficacité des RGFORs avec variables multivaluées en comparaison avec les ADDs. Même avec de petits problèmes (au sens du nombre de variables), jusqu'à deux fois moins de mémoire est à utiliser. Bien que marginal, plus le problème a de variables, plus les RGFORs s'avèrent là encore efficace en comparaison avec les ADDs.

2.4.3 Et pour les problèmes clés

Pour juger de l'efficacité des RGFORs multivaluées sur les ADDs dans les problèmes clés, nous avons comparé le nombre total de nœuds utilisés par chaque représentation pour modéliser les problèmes. Nous avons donc sommé le nombre de nœuds de tous les graphes nécessaires pour représenter les fonctions *Récompense* et *Transition*. Bien sur, une minimisation sur chaque graphe a été faite avant de faire la somme.

COFFEE ROBOT

Le problème COFFEE ROBOT n'est pas un problème pertinent pour mesurer l'efficacité de cette nouvelle représentation. En effet, COFFEE ROBOT est un problème entièrement binaire : toutes les variables sont booléennes. Par conséquence, la représentation en RGFORs de ce problème est la même que la représentation en ADDs. Il en resulte que le passage en RGFORs n'offre aucun avantage, ni aucun désavantage.

FACTORY

Le problème FACTORY utilise un mélange de variables binaires et de variables ternaires. De plus, comme déjà évoqué, ce problème a plusieurs versions ajoutant des variables supplémentaires. La Table 2.2 montre le rapport de taille pour les deux représentations sur les différentes versions du problème.

Un gain de 10% est fait sur la représentation en ADD. Ce faible gain s'explique par le peu de variables ternaires de ces problèmes. D'une part, la version multivaluée est alors quasiment similaire à la version binaire d'un point de vue structure ; en dehors des nœuds associés à des variables ternaires, les deux structures sont similaires. D'autre part, une variable ternaire se binarise à l'aide de deux variables binaires. Au pire cas, chaque nœud associé à une variable ternaire se transforme en un mini-graphe de trois nœuds dans la version binaire. D'où le faible gain en taille.

TABLE 2.2 – *Comparaison en taille des* RGFORs *multivaluées par rapport aux* ADDs

Problème	Taille RGFOR/ Taille ADD	Diff. espace états
Tiny-FACTORY	94, 7%	75%
FACTORY	86, 1%	42, 2%
FACTORY 0	87, 3%	42, 2%
FACTORY 1	88%	42, 2%
FACTORY 2	88, 4%	42, 2%
FACTORY 3	87, 5%	31, 6%

Deux observations sont toutefois à faire. Premièrement, à chaque passage à une nouvelle version où des variables ternaires sont introduites (Tiny-FACTORY $\longrightarrow$ FACTORY et FACTORY 2 $\longrightarrow$ FACTORY 3) une amélioration du gain est observé (86,1 contre 94,7 dans le premier cas, 87,5 contre 88 dans le deuxième cas). Ce résultat démontre donc la capacité des RGFORs de mieux représenter des variables multivaluées.

Néanmoins, lorsque uniquement des variables binaires sont introduites, des pertes de gain sont observées (87,3 contre 86,1 dans FACTORY $\longrightarrow$ FACTORY 0, et 87,3 contre 88 dans FACTORY 1). En effet, dans les deux types de représentations, certains graphes

existants sont alors agrandis par l'insertion de nœuds associés à ces nouvelles variables binaires. Des modifications équivalentes pour les deux versions ont donc lieu dans ces graphes. Ces modifications ont pour effet de diluer les gains apportés par la représentation multivaluée.

La dernière colonne de Table 2.2 compare la taille de $\mathbb{S}$ représenté à l'aide de variables multivaluées (taille alors égale au produit des domaines des variables multivaluées), et la taille de $\mathbb{S}$ après binarisation (égale à deux puissance le nombre de variables binaires). Chaque changement de version où une variable ternaire est insérée entraîne une inflation de la taille de l'espace d'états pour la version binaire. Le problème est que très vite (dès FACTORY), le nombre d'états concrets est inférieur au nombre d'états fictifs résultant de la binarisation : $57,7\%$ des états de FACTORY, FACTORY 0, FACTORY 1 et FACTORY 2 sont fictifs. Cette proportion d'états fictifs atteint les $69,4\%$ dans FACTORY 3. Cette modélisation à l'aide de variables binaires posent donc réellement problème, en particulier lorsque dans la Partie III nous chercherons à explorer l'espace d'états. La représentation en variables multivaluées, nécessairement fidèle à l'espace d'états d'origine, ne pose pas ce problème.

TAXI

Les variables du problème TAXI sont exclusivement multimodales. Une binarisation de tous les variables du problème est donc nécessaire pour représenter le problème à l'aide d'ADDs. Au total, la binarisation résulte en la création de quinze nouvelles variables. En particulier, la binarisation de la variable *Tank* introduit à elle seule quatre variables binaires pour encoder les quatorze modalités qu'elle peut assumer.

La déffirence de taille entre la version multivaluée et la version binaire est de 58%. La version multivaluée s'avère donc très efficace pour représenter des problèmes multivalués puisqu'elle permet une réduction de près de 50%. À cela s'ajoute le fait que l'espace d'états de la solution binaire génère $77,1\%$ d'états fictifs dans sa modélisation. Les représentations à l'aide d'RGFORs des FMDPs s'avèrent donc le meilleur choix pour des problèmes à variables multimodales.

2.5 Conclusion

Dons ce chapitre, nous avons donc présenté une solution pour la représentation des variables multimodales dans les ADDs : les RGFORs. Nous avons vu que la littérature s'intéressait déjà à ce type de représentation. Plusieurs travaux de recherche ont en effet utilisé ce type de structure sous plusieurs noms différents.

Dans ce chapitre, nous lui avons donné une formulation correspondant à nos besoins : à savoir représenter de manière compacte la factorisation d'une fonction de plusieurs variables discrètes à l'aide d'un ensemble de CONTEXTES partitionnant le domaine de définition de cette fonction. Nous avons tout d'abord donné la définition des RGFs qui adresse aussi bien des représentations sous forme d'ARBRES DE DÉCISION que sous forme d'ADDs ou de RGFORs multivaluées. Puis nous avons vu qu'en ajoutant les propriétés d'ordonnancement des variables et de réduction du graphe obtenu, nous obtenons les ADDs ou les RGFORs multivaluées.

Nous avons évoqué comment la réduction se faisait dans le cas multimodal. Nous avons utilisé un des algorithmes de minimisation des ADDs pour l'étendre aux RGFORs multivaluées.

Enfin nous avons comparé l'efficacité d'utiliser des RGFORs multivaluées en lieu et place d'ADDs. Nous avons vu que plus les variables ont un domaine grand, plus le gain en taille est important pour les RGFORs. Nous avons aussi vu que la représentation multivaluée est bien plus efficace que la représentation binaire pour nos problèmes clefs. Surtout nous avons noté que la représentation multivaluée n'entraîne pas une importe inflation de l'espace d'états (causé par la binarisation).

Par la suite nous privilégierons donc les RGFORs pour la représentation. En effet, nous allons voir que, d'une part, certains algorithmes de résolution des *problèmes de décisions séquentielles dans l'incertain* sont d'une complexité dépendante de la taille des graphes utilisés. Par conséquence, plus la représentation est compacte, plus ces algorithmes sont efficaces. D'autre part, les algorithmes d'apprentissage que nous utiliserons par la suite pour apprendre le modèle bénéficieront eux aussi de cette représentation. En effet, une représentation binaire implique une augmentation du nombre de variables, donc de la complexité de ces algorithmes d'apprentissage. De plus, l'agrandissement de l'espace d'états induit par la binarisation n'est pas sans poser problème, en particulier au niveau de la gestion du compromis exploration-exploitation.

Deuxième partie

Planification et Représentations Factorisées

Chapitre 3

Planification dans un MDP

Dans ce chapitre, nous allons voir comment se résolvent les *problèmes de décisions séquentielles dans l'incertain* posé à l'aide de MDP. Après avoir caractérisé toute solution au problème, nous allons voir quelles techniques sont mises en œuvre pour trouver une solution optimale. Enfin nous verrons comment ces techniques sont adaptées pour résoudre des problèmes de grandes tailles.

3.1 Caractérisation d'une solution à un problème de décision dans l'incertain

Les MDPs modélisent des *problèmes de décisions séquentielles dans l'incertain* (cf. Partie I). Dans ces problèmes, un agent est immergé dans un environnement stochastique et doit y accomplir une ou plusieurs tâches. Pour y parvenir, il dispose de plusieurs actions qui lui confèrent un certain contrôle sur son environnement. Le problème est alors de savoir quelles actions sont à entreprendre pour réaliser les objectifs fixés.

3.1.1 Politique

En cours d'exploitation, l'agent prend une décision, observe l'impact de cette décision sur l'environnement puis en prend une autre, et ainsi de suite. Ces décisions sont prises *itérativement*. Le problème n'est donc pas de trouver une *séquence* de décisions optimales à suivre quoiqu'il arrive. Il s'agit plutôt de trouver en tout état du système la ou les décisions optimales à prendre. Nous voulons indiquer à l'agent en tout état quelles sont les actions à effectuer : la *Politique* à suivre .
Formellement, une *Politique* peut être définie de deux manières. La première est d'associer une et une seule action à tout état : on parle alors de *Politique déterministe*.

Définition 3.1 (Politique Déterministe)**.**

Une *Politique déterministe* π est une fonction :

$$\begin{array}{rccc} \pi : & \mathbb{S} & \longrightarrow & \mathbb{A} \\ & s & \longmapsto & \pi(s) \end{array}$$

La *Politique* est alors utilisée de la manière suivante : en l'état s, l'agent effectue l'action $a = \pi(\mathrm{s})$ [1]. Le système transite alors en un nouvel état s' où l'agent effectuera l'action $a' = \pi(s')$, et ainsi de suite ...

La seconde manière de définir une *Politique* est d'associer à chaque couple état-action une probabilité d'exécution :

Définition 3.2 (Politique Stochastique)**.**

Une *Politique stochastique* [2] est une fonction définie comme :

$$\begin{array}{rccc} \pi : & \mathbb{S} \times \mathbb{A} & \longrightarrow & [0,1] \\ & (s,a) & \longmapsto & \pi(s,a) \end{array}$$

t.q.

$$\forall s \in \mathbb{S},\ \sum_{a \in \mathbb{A}} \pi(s,a) = 1$$

Ainsi, lorsque l'agent est en l'état s, il effectue un tirage aléatoire sur l'action à effectuer suivant la distribution de probabilité $\pi(s,a)$. Puis il effectue l'action retenue par le tirage. Il transite alors vers un nouvel état où il effectue un nouveau tirage, et ainsi de suite.

Tout comme les fonctions *Transition* et *Récompense*, une *Politique* est susceptible d'évoluer au cours du temps. L'action (ou la distribution de probabilités) choisie peut donc être changée en fonction du temps écoulé. À l'inverse, la *Politique* peut aussi être fixée définitivement :

1. Politique Déterministe et Chaînes de Markov : Lorsque l'agent suit une *Politique déterministe*, le système peut être modélisé plus simplement à l'aide d'une Chaîne de Markov [Norris, 1998]. L'espace d'état est toujours l'espace d'état $\mathbb{S}$. Les distributions de probabilité de transition sont alors celles associées à la *Politique* choisie : $\mathcal{T}(s,s') = \mathcal{T}(s,\pi(s),s')$.

2. $\Pi_{Deterministe} \subset \Pi_{Stochastique}$: La classe des *Politiques déterministe* est un sous-ensemble de la classe des *Politiques stochastiques*. En effet, il est possible d'associer à toute *Politique déterministe* une *Politique stochastique* t.q. $\forall s \in \mathbb{S}$, $\forall a \in \mathbb{A}$:
Si $a = \pi_{Deterministe}(s)$,
Alors $\pi_{Stochastique}(s,a) = 1$,
Sinon $\pi_{Stochastique}(s,a) = 0$.

Propriété 5 (Politique Stationnaire)**.** *Une* Politique stationnaire *est une* Politique *qui n'évolue pas au cours du temps.*

Dans ce manuscrit, nous ne nous intéresserons qu'aux *Politiques stationnaires déterministes*. Plus précisément, nous recherchons une *Politique stationnaire* et *déterministe* qui permet à l'agent d'atteindre le ou les objectifs fixés. Il s'agit donc de trouver la *Politique stationnaire* et *déterministe* optimale[3].

Pour pouvoir trouver cette *Politique* optimale, il faut d'abord pourvoir comparer deux *Politiques* et pouvoir dire que l'une est meilleure que l'autre. Il faut donc une mesure de performance des deux *Politiques* vis-à-vis des objectifs à atteindre.

3.1.2 Évaluer une *Politique* : la fonction *Valeur*

La fonction *Récompense* $\mathcal{R}$, introduite au Chapitre 1, est utilisée à cette fin : cette fonction sert à marquer les actions à faire, à distinguer les états à atteindre en leurs associant une récompense offerte alors à l'agent. Dès lors, une bonne *Politique* est une *Politique* à même de collecter plusieurs de ces récompenses.

Un critère de performance basé sur les récompenses

Le calcul de la performance d'une *Politique* π est donc basé sur le *gain cumulé* de récompenses obtenues par l'agent en suivant cette *Politique* π. Si le problème était un problème de décisions séquentielles dans le certain, ce *gain cumulé* se calculerait alors de la manière suivante :

$$\mathcal{G}^{\pi} = \mathcal{R}(s_0,\ \pi(s_0)) + \mathcal{R}(s_1,\ \pi(s_1)) + \ldots + \mathcal{R}(s_t,\ \pi(s_t)) + \ldots$$

Toutefois, deux problèmes se posent.

D'une part, ce critère ne prend pas en compte l'état d'origine du système. Or celui-ci a une influence sur le gain de récompense. Commencer le problème COFFEE ROBOT avec le robot qui a déjà du café en main n'est pas la même chose que commencer ce problème sans café en main. En effet, dans la deuxième situation de départ, il va falloir traverser la rue deux fois pour aller chercher du café et donc risquer d'être mouillé. Pour calculer la performance d'une *Politique*, il faut prendre en compte l'état de départ. La mesure de performance est donc une fonction de $\mathbb{S} \longrightarrow \mathbb{R}$ qui à chaque état associe la

3. Par la suite et par abus de langage, lorsque nous évoquerons le terme de *Politique*, nous parlerons d'une *Politique stationnaire* et *déterministe*.

performance de la *Politique* en partant de cet état :

$$\mathcal{G}_\pi(s_0) = \mathcal{R}(s_0,\ \pi(s_0)) + \mathcal{R}(s_1,\ \pi(s_1)) + \ldots + \mathcal{R}(s_t,\ \pi(s_t)) + \ldots$$

D'autre part, la nature stochastique de l'environnement n'est pas prise en compte. Or, il est très peu probable que deux tentatives effectuées avec le même état de départ donnent la même séquence de récompenses. Dans COFFEE ROBOT, en supposant que le robot doit aller chercher du café, dans une tentative il peut s'en sortir sans être trempé, et dans l'autre subir la pluie. Dès lors, pour prendre en compte l'incidence de l'environnement, il faut passer en estimation probabiliste ; il s'agit de calculer l'*espérance du gain cumulé* :

$$E[\mathcal{G}_\pi(s)] = E\left[\sum_{t=0}^{\infty} \mathcal{R}(s_t,\ \pi(s_t)) \mid s_0 = s\right] \tag{3.1}$$

Le problème de cette espérance est qu'elle peut diverger sous l'hypothèse d'horizon infini [4].

Le *facteur d'actualisation*

Cependant, la pertinence de prendre en compte des récompenses très éloignées dans le temps se pose. En effet, la nature stochastique de l'environnement a un certain impact sur les récompenses atteignables. Plus une récompense est loin, moins il est probable de l'obtenir à partir de l'état courant du fait des aléas du système. Donc prendre en compte une récompense future telle quelle ne semble pas pertinent.

Une solution est de dégrader les récompenses futurs pour les rendre comparable à l'instant courant. Le *facteur d'actualisation* agit en ce sens. Il s'agit d'une grandeur réelle γ comprise entre 0 et 1 qui dégrade plus ou moins la valeur accordée à une unité de récompense reçue à l'instant suivant : $\gamma_t \sim 1_{t+1}$. Appliquée récursivement, cette grandeur pondère donc la récompense reçue à l'instant $t+T$ par la valeur γ^T. Dès lors, le critère de performance devient une *espérance du gain pondéré cumulé* :

$$E[\mathcal{G}_\pi(s)] = E\left[\sum_{t=0}^{\infty} \gamma^t \cdot \mathcal{R}(s_t,\ \pi(s_t)) \mid s_0 = s\right] \tag{3.2}$$

Lorsque T devient très grand, γ^T tends vers 0, et donc, mécaniquement, l'influence de la récompense à l'instant $t+T$ diminue dans la somme.

4. Dans le cas de problème à horizon fini non borné et aléatoire, du type jeux et paris où le processus s'arrête bien à un moment mais celui-ci ne peut être défini initialement, cette espérance a une valeur finie et peut être (et est) employée.

Le *facteur d'actualisation* peut donc être vu comme un coefficient d'intérêt pour les récompenses en fonction de leurs éloignements temporels. La valeur de γ définit plus ou moins un horizon temporel au-delà duquel les récompenses cessent d'influer sur les décisions à prendre. Par exemple, pour $\gamma = 0$, seule la récompense immédiatement reçue est importante. L'agent est alors dit *myope*.

De plus, on peut démontrer que l'*espérance du gain pondéré cumulé*[5] existe alors :

Lemme 3.3 (Existence de l'espérance du gain pondéré cumulé)**.**

Pour $\gamma \in [0, 1[$*, si la fonction* Récompense $\mathcal{R}$ *est bornée,*

alors l'espérance $E\left[\sum_{t=0}^{\infty} \gamma^t \cdot \mathcal{R}(s_t,\ \pi(s_t)) \mid s_0\right]$ *existe dans* $\mathbb{R}$.

Démonstration.

Soient $\mathcal{R}_{min}$ et $\mathcal{R}_{max}$ t.q. $\forall s \in \mathbb{S}$,

$$\mathcal{R}_{min} \ \leq\ \mathcal{R}(s_t,\ \pi(s_t)) \ \leq\ \mathcal{R}_{max}$$

On a alors,$\forall s \in \mathbb{S}$, $\forall t$:

$$\gamma^t \cdot \mathcal{R}_{min} \ \leq\ \gamma^t \cdot \mathcal{R}(s_t,\ \pi(s_t)) \ \leq\ \gamma^t \cdot \mathcal{R}_{max}$$

Et en sommant :

$$\sum_{t=0}^{\infty} \gamma^t \cdot \mathcal{R}_{min} \ \leq\ \sum_{t=0}^{\infty} \gamma^t \cdot \mathcal{R}(s_t,\ \pi(s_t)) \ \leq\ \sum_{t=0}^{\infty} \gamma^t \cdot \mathcal{R}_{max}$$

Donc :

$$\mathcal{R}_{min} \cdot \sum_{t=0}^{\infty} \gamma^t \ \leq\ \sum_{t=0}^{\infty} \gamma^t \cdot \mathcal{R}(s_t,\ \pi(s_t)) \mid s_0 \ \leq\ \mathcal{R}_{max} \cdot \sum_{t=0}^{\infty} \gamma^t$$

Or $\sum_{t=0}^{\infty} \gamma^t$ est une série convergente pour $\gamma \in [0, 1[$ et converge vers $\frac{1}{1-\gamma}$.

Donc $\sum_{t=0}^{\infty} \gamma^t \cdot \mathcal{R}(s_t,\ \pi(s_t)) \mid s_0$ est bornée dans l'intervalle $[\frac{\mathcal{R}_{min}}{1-\gamma}, \frac{\mathcal{R}_{max}}{1-\gamma}]$.

5. En *Horizon Infini*, il existe une troisième mesure de performance d'une *Politique* : l'*espérance du gain cumulé moyen.*

$$\rho_\pi(s) = \lim_{n \longrightarrow \infty} E\left[\frac{1}{n} \sum_{t=0}^{n-1} \mathcal{R}(s_t,\ \pi(s_t)) \mid s_0 = s\right]$$

Ce critère mesure la performance moyenne d'une *Politique*. Ce critère est particulièrement adapté aux problèmes logistiques d'accès à une ressource.

Donc $E\left[\sum_{t=0}^{\infty} \gamma^t \cdot \mathcal{R}(s_t,\ \pi(s_t)) \mid s_0\right]$ existe. □

Fonction *Valeur* et calcul de la Fonction *Valeur*

L'*espérance du gain pondéré cumulé* permet d'évaluer la performance d'une *Politique*. Toutefois cette mesure se définit par rapport à un état initial. Pour évaluer correctement une *Politique*, sa performance doit être étudiée pour l'ensemble des états possibles. Soit donc la fonction *Valeur* qui associe à chaque état la performance d'une *Politique* donnée :

Définition 3.4 (Fonction Valeur)**.**

Soient un MDP $\langle \mathbb{S}, \mathbb{A}, \mathbb{T}, \mathcal{T}, \mathcal{R} \rangle$ et une *Politique* π associée à ce MDP,

Soit $\gamma \in [0, 1[$,

La fonction *Valeur* $\mathcal{V}_\pi$ est la fonction :

$$\begin{array}{rccl} \mathcal{V}_\pi : & \mathbb{S} & \longrightarrow & \mathbb{R} \\ & s & \longmapsto & E\left[\sum_{t=0}^{\infty} \gamma^t \cdot \mathcal{R}(s_t,\ \pi(s_t)) \mid s_0 = s\right] \end{array}$$

Calculer la fonction *Valeur* d'une *Politique* en l'état s'avère assez complexe. Il s'agit en effet de calculer l'espérance d'une série aléatoire. Il est tout de même possible de l'évaluer.

Pour calculer la fonction *Valeur* $\mathcal{V}_\pi$ pour une *Politique* π donnée, il est nécessaire de procéder à quelques transformations :

$$\begin{aligned} \forall s \in \mathbb{S},\ \mathcal{V}_\pi(s) &= E\left[\sum_{t=0}^{\infty} \gamma^t \cdot \mathcal{R}(s_t,\ \pi(s_t)) \mid s_0 = s\right] \\ &= E\left[\mathcal{R}(s_0,\ \pi(s_0)) + \sum_{t=1}^{\infty} \gamma^t \cdot \mathcal{R}(s_t,\ \pi(s_t)) \mid s_0 = s\right] \end{aligned}$$

Par linéarité de l'espérance mathématique :

$$\mathcal{V}_\pi(s) = E\left[\mathcal{R}(s_0,\ \pi(s_0)) \mid s_0 = s\right] + E\left[\sum_{t=1}^{\infty} \gamma^t \cdot \mathcal{R}(s_t,\ \pi(s_t)) \mid s_0 = s\right]$$

Or, par définition,

$$E\left[\sum_{t=1}^{\infty} \gamma^t \cdot \mathcal{R}(s_t,\ \pi(s_t)) \mid s_0 = s\right] = \gamma \cdot E\left[\mathcal{V}_\pi(s_1) \mid s_0 = s\right]$$

Donc :

$$\mathcal{V}_\pi(s) = E\left[\mathcal{R}(s_0,\ \pi(s_0)) \mid s_0 = s\right] + \gamma \cdot E\left[\mathcal{V}_\pi(s_1) \mid s_0 = s\right]$$

De plus, d'une part,

$$E\left[\mathcal{R}(s_0,\ \pi(s_0)) \mid s_0 = s\right] = \mathcal{R}(s,\ \pi(s))$$

Et d'autre part :

$$E\left[\mathcal{V}_\pi(s_1) \mid s_0 = s\right] = \sum_{s' \in \mathbb{S}} P(s' \mid s,\ \pi(s)) \cdot \mathcal{V}_\pi(s')$$

Finalement, nous avons :

$$\mathcal{V}_\pi(s) = \mathcal{R}(s,\ \pi(s)) + \gamma \cdot \sum_{s' \in \mathbb{S}} P(s' \mid s,\ \pi(s)) \cdot \mathcal{V}_\pi(s') \tag{3.3}$$

La fonction *Valeur* est donc la solution d'un système linaire de la forme $X = AX + B$. Dès lors, il est possible de la résoudre de manière algébrique. Toutefois, cette approche peut être d'une grande complexité et est soumise à des instabilités numériques [Higham, 2002].

Une autre approche est possible. Tout d'abord, observons que l'espace $\mathbb{V}$ des fonctions de $\mathbb{S}$ dans $\mathbb{R}$ est identifiable à l'espace vectorielle $\mathbb{R}^{|\mathbb{S}|}$. Ainsi toute fonction $\mathcal{V}$ de $\mathbb{V}$ peut être vu comme un vecteur de $\mathbb{R}^{|\mathbb{S}|}$ [6]. L'autre approche possible est basée sur l'*Opérateur de Bellmann* défini comme :

Définition 3.5 (*Opérateur de Bellmann* Γ_π d'une *Politique* π)**.**

L'*Opérateur de Bellmann* Γ_π d'une *Politique* π est l'application :

$$\begin{array}{rccc} \Gamma_\pi : & \mathbb{R}^{|\mathbb{S}|} & \longrightarrow & \mathbb{R}^{|\mathbb{S}|} \\ & \mathcal{U} & \longmapsto & \Gamma_\pi(\mathcal{U}) \end{array}$$

t.q.

$$\forall s \in \mathbb{S},\ \Gamma_\pi(\mathcal{U}(s)) = \mathcal{R}(s,\ \pi(s)) + \gamma \cdot \sum_{s' \in \mathbb{S}} P(s' \mid s,\ \pi(s)) \cdot \mathcal{U}(s')$$

La fonction *Valeur* d'une *Politique* π est donc par définition un des points fixes de l'*Opérateur de Bellmann* (c.f. Equation 3.3). Il se trouve qu'elle en est l'unique.

6. Aussi nous noterons $\mathcal{V}(s)$ l'image de s par la fonction $\mathcal{V}$, et $\mathcal{V}$ le vecteur d'images sur $\mathbb{R}^{|\mathbb{S}|}$.

Lemme 3.6.

Soient un MDP $\langle \mathbb{S}, \mathbb{A}, \mathbb{T}, \mathcal{T}, \mathcal{R} \rangle$ *et une* Politique π *associée à ce* MDP.

La fonction Valeur $\mathcal{V}_\pi$ *de la* Politique π *est l'unique point fixe de l'*Opérateur de Bellmann *associé. En d'autre terme :*

$$\exists! \mathcal{V} \in \mathbb{R}^{|\mathbb{S}|} \text{ t.q. } \mathcal{V} = \Gamma_\pi(\mathcal{V})$$

Et alors :

$$\mathcal{V} = \mathcal{V}_\pi$$

Pour démontrer ce lemme, il faut d'abord vérifier que l'*Opérateur de Bellmann* dispose de certaines propriétés. Tout d'abord, il doit être monotone :

Propriété 6 (Monotonicité de l'Opérateur de Bellmann)**.**

Soient $\mathcal{U} \in \mathbb{R}^{|\mathbb{S}|}$ *et* $\mathcal{V} \in \mathbb{R}^{|\mathbb{S}|}$,

$$\forall s \in \mathbb{S}, \mathcal{U}(s) \leq \mathcal{V}(s) \Longrightarrow \Gamma_\pi(\mathcal{U}(s)) \leq \Gamma_\pi(\mathcal{V}(s))$$

Démonstration.

Soient $\mathcal{U}$ et $\mathcal{V}$, deux vecteurs de $\mathbb{R}^{|\mathbb{S}|}$ t.q. $\mathcal{U} \leq \mathcal{V}$.

On a alors,$\forall s, s' \in \mathbb{S}$:

$$P(s' \mid s,\ \pi(s)) \cdot \mathcal{U}(s') \leq P(s' \mid s,\ \pi(s)) \cdot \mathcal{V}(s')$$

En sommant :

$$\sum_{s' \in \mathbb{S}} P(s' \mid s,\ \pi(s)) \cdot \mathcal{U}(s') \leq \sum_{s' \in \mathbb{S}} P(s' \mid s,\ \pi(s)) \cdot \mathcal{V}(s')$$

Et donc :

$$\mathcal{R}(s,\ \pi(s)) + \gamma \cdot \sum_{s' \in \mathbb{S}} P(s' \mid s,\ \pi(s)) \cdot \mathcal{U}(s') \leq \mathcal{R}(s,\ \pi(s)) + \gamma \cdot \sum_{s' \in \mathbb{S}} P(s' \mid s,\ \pi(s)) \cdot \mathcal{V}(s')$$

□

L'*Opérateur de Bellmann* doit aussi vérifier la γ-additivité :

Propriété 7 (γ-Additivité de l'Opérateur de Bellmann)**.**

Soient $U \in \mathbb{R}^{|\mathbb{S}|}$ *et* $\delta \in \mathbb{R}$,

$$\forall s \in \mathbb{S}, \Gamma_\pi(\mathcal{U}(s) + \delta) = \Gamma_\pi(\mathcal{U}(s)) + \gamma \cdot \delta$$

Démonstration.

Soit $\mathcal{U}$, un vecteur de $\mathbb{R}^{|\mathbb{S}|}$.

On a alors, $\forall s \in \mathbb{S}$:

$$\begin{aligned}\Gamma_\pi(\mathcal{U}(s)+\delta) &= \mathcal{R}(s,\ \pi(s)) + \gamma \cdot \sum_{s' \in \mathbb{S}} P(s' \mid s,\ \pi(s)) \cdot (\mathcal{U}(s')+\delta) \\ &= \mathcal{R}(s,\ \pi(s)) + \gamma \cdot \sum_{s' \in \mathbb{S}} P(s' \mid s,\ \pi(s)) \cdot \mathcal{U}(s') + \gamma \cdot \sum_{s' \in \mathbb{S}} P(s' \mid s,\ \pi(s)) \cdot \delta\end{aligned}$$

Or :

$$\sum_{s' \in \mathbb{S}} P(s' \mid s,\ \pi(s)) \cdot \delta = \delta$$

Donc :

$$\Gamma_\pi(\mathcal{U}(s)+\delta) = \mathcal{R}(s,\ \pi(s)) + \gamma \cdot \sum_{s' \in \mathbb{S}} P(s' \mid s,\ \pi(s)) \cdot \mathcal{U}(s') + \gamma \cdot \delta$$

□

Sous ces deux propriétés et en définissant la norme infinie comme :

Définition 3.7 (Norme infinie de $\mathbb{R}^{|\mathbb{S}|}$)**.**

Soit $\mathcal{U} \in \mathbb{R}^{|\mathbb{S}|}$,

$$\| \mathcal{U} \|_\infty = \max_{s \in \mathbb{S}} | \mathcal{U}(s) |$$

On montre alors que l'*Opérateur de Bellmann* est une application contractante :

Propriété 8 (Application contractante de l'espace métrique $(\mathbb{R}^{|\mathbb{S}|}, \| \|_\infty)$)**.**

*L'*Opérateur de Bellmann Γ_π *est une application contractante de l'espace métrique* $\mathbb{R}^{|\mathbb{S}|}$ *dotée de la norme infinie* $\| \|_\infty$.

En d'autre terme, soit $\mathcal{U} \in \mathbb{R}^{|\mathbb{S}|}$ *et* $\mathcal{V} \in \mathbb{R}^{|\mathbb{S}|}$,

$$\| \Gamma_\pi(\mathcal{U}) - \Gamma_\pi(\mathcal{V}) \|_\infty \leq \gamma \cdot \| \mathcal{U} - \mathcal{V} \|_\infty$$

Démonstration.

Soit $d = \| \mathcal{U} - \mathcal{V} \|_\infty = \max_{s \in \mathbb{S}} | \mathcal{U}(s) - \mathcal{V}(s) |$.

On a par définition, $\forall s \in \mathbb{S}$:

$$\mathcal{U}(s) - d \leq \mathcal{V}(s) \leq \mathcal{U}(s) + d$$

En utilisant les propriétés d'additivité (7) et de monotonicité (6) de l'Opérateur de Bellmann, il vient que :

$$\Gamma_\pi(\mathcal{U}(s)) - \gamma \cdot d \leq \Gamma_\pi(\mathcal{V}(s)) \leq \Gamma_\pi(\mathcal{U}(s)) + \gamma \cdot d$$

qui est alors équivalent à :

$$\mid \Gamma_\pi(\mathcal{U}(s)) - \Gamma_\pi(\mathcal{V}(s)) \mid \leq \gamma \cdot d$$

D'où :

$$\max_{s\in\mathbb{S}} \mid \Gamma_\pi(\mathcal{U}(s)) - \Gamma_\pi(\mathcal{V}(s)) \mid \leq \gamma \cdot \max_{s\in\mathbb{S}} \mid \mathcal{U}(s) - \mathcal{V}(s) \mid$$

Ou encore :

$$\parallel \Gamma_\pi(\mathcal{U}) - \Gamma_\pi(\mathcal{V}) \parallel_\infty \leq \gamma \cdot \parallel \mathcal{U} - \mathcal{V} \parallel_\infty$$

□

Dès lors, le théorème du point-fixe de *Banach* s'applique à l'*Opérateur de Bellmann.*

Théorème 3.8 (Théorème du point fixe de Banach)**.**

Soit (X, d) un espace métrique non vide complet avec une application contractante $\Gamma : X \longrightarrow X$. Alors Γ admets un unique point fixe $x^\star$ in X (i.e. $\Gamma(x^\star) = x^\star$).

Ce qui démontre que la fonction $\mathcal{V}_\pi$ est bien l'unique point fixe de l'opérateur Γ_π. Mais en plus, un autre moyen de calculer la fonction *Valeur* est dévoilé.

Corollaire 3.9 (Calcul du Point Fixe de Banach)**.**

$x^\star$ peut être trouvé comme suit : commencer avec un élément arbitraire $x_0 \in X$ et appliquer la suite $(x_n)_{n\in\mathbb{N}}$ définie par $x_n = \Gamma(x_{n-1})$, alors $x_n \longrightarrow x^\star$.

Ainsi, pour tout vecteur $\mathcal{V}_0$ de $\mathbb{R}^{|\mathbb{S}|}$, il suffit donc d'appliquer récursivement l'*Opérateur de Bellmann* de sorte que $\mathcal{V}_n = \Gamma_\pi(\mathcal{V}_{n-1})$ et alors après suffisamment d'itération, nous obtenons $\parallel \mathcal{V}_n - \mathcal{V}_\pi \parallel_\infty < \epsilon$, $\forall \epsilon \in [0, 1]$.

Cette méthode de calcul est donc une méthode approchée. Toutefois sa complexité n'est qu'en $\mathcal{O}(|\mathbb{S}|^2)$. Soit une complexité moindre que celle des algorithmes de résolutions de système linéaire [7].

7. Une troisième manière de calculer cette fonction *Valeur* existe. Il s'agit de poser le calcul sous forme de **P***rogramme* **L***inéaire* [Dantzig, 1963] :

$$\begin{aligned} &\text{Minimiser} && \sum_s \alpha(s) \cdot \mathcal{V}(s) \\ &\text{Sous contrainte :} && \mathcal{V}(s) \geq \mathcal{R}(s,\ \pi(s)) + \gamma \cdot \sum_{s'\in\mathbb{S}} P(s' \mid s,\ \pi(s)) \cdot \mathcal{V}(s') \\ & && \forall s \in \mathbb{S} \end{aligned}$$

3.2 Recherche de la *Politique* optimale

La fonction *Valeur* évalue donc la performance d'une *Politique* donnée. Nous disposons maintenant d'un critère pour rechercher une *Politique* optimale.

3.2.1 Fonction *Valeur* et Critère d'Optimalité

Rappelons que la fonction *Valeur* est représentable sous la forme d'un vecteur de $\mathbb{R}^{|\mathbb{S}|}$. Or il existe un *ordre partiel* sur $\mathbb{R}^{|\mathbb{S}|}$:

Propriété 9 (Ordre partiel sur $\mathbb{R}^{|\mathbb{S}|}$)**.**
$\forall \mathcal{U}, \mathcal{V} \in \mathbb{R}^{|\mathbb{S}|}, \mathcal{U} \leq \mathcal{V} \iff \forall s \in \mathbb{S}, \mathcal{U}(s) \leq \mathcal{V}(s)$

Nous pouvons donc comparer deux à deux les *Politiques* optimales pour déterminer quelle politique est la meilleure.

Le critère d'optimalité d'une *Politique* se définit alors ainsi :

Définition 3.10 (Critère d'Optimalité d'une *Politique*)**.**
Soient $\pi^\star$ une *Politique*, et $\mathcal{V}_{\pi^\star}$ sa fonction *Valeur* associée.
$\pi^\star$ est optimale $\iff \forall$ autre *Politique* π, $\mathcal{V}_\pi \leq \mathcal{V}_{\pi^\star}$

Ce critère d'optimalité permet de déterminer si une *Politique* est optimale ou non. Toutefois, il ne permet pas d'en déduire directement une *Politique* optimale. Pour trouver cette *Politique* optimale, des algorithmes de recherche sont nécessaires. Ce critère permettra alors de démontrer si l'algorithme est capable d'atteindre l'optimal.

Un algorithme de recherche tout simple serait alors de comparer toutes les *Politiques* possibles est de conserver les plus performantes. Toutefois la taille de l'espace des *Politiques* est en $|\mathbb{A}|^{|\mathbb{S}|}$. Cet algorithme *Force Brute*, qui aurait alors une complexité en $\mathcal{O}(|\mathbb{S}| \times |\mathbb{A}|^{2|\mathbb{S}|} \times |\mathbb{S}|^2)$, n'est donc pas envisageable.

3.2.2 *Opérateur d'Optimalité de Bellmann*

Les méthodes de résolution des *problèmes de décisions séquentielles dans l'incertain* les plus efficaces reposent sur la *programmation dynamique*.

Programmation Dynamique et Principe d'Optimalité

Cette méthode d'optimisation s'applique aux problèmes d'optimisation qui peuvent être décomposés en sous-problèmes de même nature. Pour un MDP, le problème d'optimisation est de trouver une *Politique* optimale. Ce problème peut être découpé en

sous-problèmes qui sont de trouver une *Politique* optimale pour chaque pas de temps t. Le *Principe d'Optimalité*, sur lequel s'appuie la *programmation dynamique*, s'énonce alors ainsi pour un MDP :

Principe 3.11 (Principe d'Optimalité [Bellman, 1957])**.**

Une Politique *Optimale a la propriété que quels que soient l'état initial et la décision initiale, les décisions restantes doivent rester une* Politique *optimale par rapport à l'état résultant de la première décision.*

En d'autres termes, si en l'état courant l'agent prend la meilleure action à suivre, puis se conforme à la *Politique* optimale, alors il suit la *Politique* optimale [8]. La *Politique* optimale est donc la *Politique* qui a pour propriété :

$$\forall s \in \mathbb{S},\ \mathcal{V}_{\pi^\star}(s) = \max_{a \in \mathbb{A}} \left[\mathcal{R}(s,\ a) + \gamma \cdot \sum_{s' \in \mathbb{S}} P(s' \mid s,\ a) \cdot \mathcal{V}_{\pi^\star}(s') \right] \tag{3.4}$$

Opérateur d'Optimalité de Bellmann

À partir de l'Equation 3.4, un nouvel opérateur peut être introduit : l'*Opérateur d'Optimalité de Bellmann* $\Gamma^\star$.

Définition 3.12 (Opérateur d'optimalité de Bellmann $\Gamma^\star$)**.**

L'*Opérateur d'Optimalité de Bellmann* $\Gamma^\star$ est l'application :

$$\begin{array}{rccc} \Gamma^\star : & \mathbb{R}^{|\mathbb{S}|} & \longrightarrow & \mathbb{R}^{|\mathbb{S}|} \\ & \mathcal{U} & \longmapsto & \Gamma^\star(\mathcal{U}) \end{array}$$

t.q.

$$\forall s \in \mathbb{S},\ \Gamma^\star(\mathcal{U}(s)) = \max_{a \in \mathbb{A}} \left[\mathcal{R}(s,\ a) + \gamma \cdot \sum_{s' \in \mathbb{S}} P(s' \mid s,\ a) \cdot \mathcal{U}(s') \right]$$

L'*Opérateur d'Optimalité de Bellmann* possède les propriétés de Monotonicité (6) et de γ-Additivité (7). Les démonstrations se font aisément en reprenant les démonstrations pour l'*Opérateur de Bellmann*. Il en découle que l'*Opérateur d'Optimalité de*

8. Horizon Fini et Backtracking : A horizon fini, l'exploitation du principe d'optimalité peut être vue comme un *backtracking* à partir de la date de fin T. On prend alors l'action qui donne la meilleure récompense directe. Puis on recule à $T-1$. À $T-1$, on prend l'action qui maximise la récompense immédiate et l'espérance sur la récompense à la date T, en considérant qu'à la date T, l'action qui donne la récompense maximale sera prise. Puis on recule à $T-2$. À $T-2$, on prend l'action qui maximise la récompense immédiate et l'espérance sur la récompense à la date $T-1$, avec les mêmes considérations sur l'action optimale prise à $T-1$. Et on répète ce processus jusqu'au temps $t=0$. En horizon infini, on peut considérer que ce processus est fait pour une date T très grande. On backtracke alors indéfiniment.

Bellmann est aussi une application contractante de l'espace métrique $(\mathbb{R}^{|\mathbb{S}|}, \| \|_\infty)$. Et donc, par application du théorème du point fixe de Banach, son unique point fixe, par définition, $\mathcal{V}_{\pi^\star}$.

Nous pouvons alors démontrer que $\mathcal{V}_{\pi^\star}$ vérifie bien le critère d'optimalité 3.10 :

Lemme 3.13. $\mathcal{V}_{\pi^\star} = \Gamma^\star(\mathcal{V}_{\pi^\star}) \Longrightarrow \mathcal{V}_{\pi^\star} \geq \mathcal{V}_\pi,\ \forall \pi \in \mathbb{A}^{|\mathbb{S}|}$

Nous n'avons pas intégré la preuve de ce lemme dans ce manuscrit. Nous renvoyons le lecteur à [Sigaud and Buffet, 2008] pour cette démonstration.

Attention toutefois : le théorème de Banach [Banach, 1922] permet de démontrer l'existence et l'unicité du point fixe de l'*Opérateur d'Optimalité de Bellmann*. Par conséquence, il démontre l'existence et l'unicité de la fonction *Valeur* $\mathcal{V}^\star$ optimale. Toutefois, ce résultat n'implique pas l'unicité des *Politiques* optimales. Ce résultat implique juste que $\forall \pi^\star,\ \mathcal{V}_{\pi^\star} = \mathcal{V}^\star$.

3.2.3 Itération de la *Valeur*

Le théorème de Banach ne permet pas uniquement de démontrer l'unicité du point fixe de l'*Opérateur d'Optimalité de Bellmann*. De la même manière que la fonction *Valeur* d'une *Politique* donnée est approchable par application récursive de l'*Opérateur de Bellmann*, la fonction *Valeur* optimale est estimable par application récursive de l'*Opérateur d'Optimalité de Bellmann*. En effet, la suite $(\mathcal{V}_n)_{n\in\mathbb{N}}$ de vecteur $\mathcal{V} \in \mathbb{R}^{|\mathbb{S}|}$ t.q. $\mathcal{V}_n = \Gamma^\star(\mathcal{V}_{n-1})$ converge vers $\mathcal{V}^\star$. En itérant suffisamment, il vient que $\| \mathcal{V}_n - \mathcal{V}^\star \|_\infty < \epsilon$, $\forall \epsilon \in [0, 1]$.

L'Algorithme VI (4) qui en découle est communément appelé *Itération de la Valeur* [Bellman, 1957], pusiqu'il consiste à faire converger sur plusieurs itérations une fonction *Valeur* initiale par application successive de l'*Opérateur d'Optimalité de Bellmann* . Pour initialiser cet algorithme, la fonction *Récompense* $\mathcal{R}$ est souvent choisi comme fonction initiale $\mathcal{V}_0$. L'*Itération de la Valeur* peut alors être vu comme un backtracking d'une date T infinie.

Le critère d'arrêt [Williams and Baird, 1993], qui est un autre corollaire du théorème de Banach, est :

$$\| \mathcal{V}_{n+1} - \mathcal{V}_n \|_\infty \leq \frac{\epsilon \cdot (1-\gamma)}{2 \cdot \gamma}$$

La fonction *Valeur* $\mathcal{V}_{n+1}$ est alors à $\frac{\epsilon}{2}$ de l'optimale $\mathcal{V}^\star$.

Algorithme 4 : VI : Itération de la *Valeur*

Données : un MDP $\langle \mathbb{S}, \mathbb{A}, \mathbb{T}, \mathcal{T}, \mathcal{R} \rangle$

début
  // Initialisation
  $\mathcal{V} \leftarrow \mathcal{R}$;
  $\Delta \leftarrow 1$;
  // Itération de la Valeur
  tant que $\Delta \geq \frac{\epsilon \cdot (1-\gamma)}{2 \cdot \gamma}$ **faire**
    $\mathcal{V}_{old} \leftarrow \mathcal{V}$;
    pour chaque $s \in \mathbb{S}$ **faire**
      pour chaque $a \in \mathbb{A}$ **faire**
        $\mathcal{Q}(s,a) \leftarrow \mathcal{R}(s,\ a) + \gamma \cdot \sum_{s' \in \mathbb{S}} P(s' \mid s,\ a) \cdot \mathcal{V}_{old}(s')$;
      $\mathcal{V}(s) \leftarrow \max_{a \in \mathbb{A}} [\mathcal{Q}(s,a)]$;
    // Critère atteint?
    $\Delta \leftarrow \| \mathcal{V} - \mathcal{V}_{old} \|_\infty$;
  // Récupération de la *Politique*
  pour chaque $s \in \mathbb{S}$ **faire**
    pour chaque a $\mathbb{A}$ **faire**
      $\mathcal{Q}(s,a) \leftarrow \mathcal{R}(s,\ a) + \gamma \cdot \sum_{s' \in \mathbb{S}} P(s' \mid s,\ a) \cdot \mathcal{V}(s')$;
    $\pi^\star(s) \leftarrow \operatorname{argmax}_{a \in \mathbb{A}} [\mathcal{Q}(s,a)]$;

Résultat : une *Politique* $\pi^\star$ ϵ-optimale

Afin d'effectuer l'étape de maximisation sur les actions de l'*Opérateur d'Optimalité de Bellmann*, il est nécessaire de calculer les quantités intermédiaires

$$\mathcal{R}(s,\ a) + \gamma \cdot \sum_{s' \in \mathbb{S}} P(s' \mid s,\ a) \cdot \mathcal{V}(s'), \forall a \in \mathbb{A}, s \in \mathbb{S}$$

Pour cela, une nouvelle fonction est introduite :

Définition 3.14 (Fonction *Valeur* d'Action $\mathcal{Q}_\pi$ [9])**.**

Soient un MDP $\langle \mathbb{S}, \mathbb{A}, \mathbb{T}, \mathcal{T}, \mathcal{R} \rangle$ et une *Politique* π associée à ce MDP,

Soit $\gamma \in [0, 1[$,

La fonction *Valeur* d'Action $\mathcal{Q}_\pi$ d'une *Politique* π est la fonction :

$$\begin{array}{rccl} \mathcal{Q}_\pi : & \mathbb{S} \times \mathbb{A} & \longrightarrow & \mathbb{R} \\ & (s,a) & \longmapsto & \mathcal{R}(s,\ a) + \gamma \cdot \sum_{s' \in \mathbb{S}} P(s' \mid s,\ a) \cdot \mathcal{V}_\pi(s') \end{array}$$

9. Nous noterons $\mathcal{Q}_a$ la restriction de $\mathcal{Q}_\pi$ à l'action a

De fait, nous avons aussi :

$$\mathcal{Q}^\star(s,a) = \mathcal{R}(s,\ a) + \gamma \cdot \sum_{s' \in \mathbb{S}} P(s' \mid s,\ a) \cdot \mathcal{V}^\star(s')$$
$$\mathcal{V}^\star(s) = \max_{a \in \mathbb{A}} \mathcal{Q}^\star(s,a)$$
$$\pi^\star(s) = \operatorname*{argmax}_{a \in \mathbb{A}} \mathcal{Q}^\star(s,a)$$

où $\mathcal{Q}^\star$ est la fonction *Valeur* d'action optimale.

La fonction *Valeur* d'Action mesure la performance de choisir l'action a en l'état s initial puis de s'en tenir à la *Politique* π. Cette fonction est utilisée dans la plupart des algorithmes de recherche de *Politique* optimale.

Lorsque le critère d'arrêt est atteint, il faut extraire une *Politique* ϵ-optimale. Les fonctions *Valeur* d'action sont recalculées une dernière fois avant de faire une étape de recherche d'*argument du maximum* pour trouver la meilleure action en tout état. L'étape de recherche d'*argument du maximum* est gloutonne. En effet, il peut y avoir pour $s \in \mathbb{S}$ plusieurs $a \in \mathbb{A}$ telles que $\mathcal{Q}(s,a)$ soit maximale. Néanmoins, comme l'algorithme élabore une *Politique déterministe*, une seule action est retenue pour chaque état. D'où le fait que la *Politique* obtenue est dite gloutonne [10].

Avec sa complexité en $\mathcal{O}(|\mathbb{A}| \times |\mathbb{S}|^2)$, l'Algorithme **VI** (4) est un des algorithmes principaux de recherche de *Politique* optimale. Toutefois, il ne s'agit que d'un algorithme approché. En effet, le critère d'arrêt implique que la fonction *Valeur* obtenue est ϵ-optimale.

3.2.4 Itération de la *Politique*

Howard [1960] propose une autre approche en manipulant cette fois-ci une *Politique* plutôt qu'une fonction *Valeur*. À chaque étape, l'algorithme *Itération de la* Politique (**PI**, 5) évalue d'abord les performances de la *Politique* courante puis construit une nouvelle *Politique* qui améliore ces performances.

L'évaluation des performances se fait par le calcul de la fonction *Valeur* de cette *Politique*. Les algorithmes de calcul vu à la sous-section 3.1.2 sont alors mis à l'œuvre. Cette phase est la plus coûteuse des deux phases puisqu'il est nécessaire soit de résoudre un système linéaire de taille $|\mathbb{S}|$ (complexité en $\mathcal{O}(|\mathbb{S}|^3)$), soit d'appliquer itérativement

10. Il est possible d'établir et d'obtenir par cet algorithme une *Politique* stochastique dans laquelle toutes les actions optimales ont une probabilité égale d'être choisies et toutes autres actions ont une probabilité nulle d'être choisies.

Algorithme 5 : PI : Itération de la *Politique*

Données : un MDP $\langle \mathbb{S}, \mathbb{A}, \mathbb{T}, \mathcal{T}, \mathcal{R} \rangle$

début
  // Initialisation
  pour chaque $s \in \mathbb{S}$ **faire**
    $\pi_{next}(s) \leftarrow \underset{a \in \mathbb{A}}{\text{argmax}} \ [\mathcal{R}(s,\ a)]$;
  $Instable \leftarrow$ true ;
  // Itéraltion de la *Politique*
  tant que $Instable$ **faire**
    $\pi \leftarrow \pi_{next}$;
    $Instable \leftarrow$ false ;
    // Performance de π
    Trouver $\mathcal{V}_\pi$ t.q. $\mathcal{V}_\pi \leftarrow \Gamma_\pi(\mathcal{V}_\pi)$;
    // Amélioration de π
    pour chaque $s \in \mathbb{S}$ **faire**
      pour chaque a $\mathbb{A}$ **faire**
        $\mathcal{Q}_\pi(s,a) \leftarrow \mathcal{R}(s,\ a) + \gamma \cdot \sum_{s' \in \mathbb{S}} P(s' \mid s,\ a) \cdot \mathcal{V}_\pi(s')$;
      $\pi_{next}(s) \leftarrow \underset{a \in \mathbb{A}}{\text{argmax}} \ [\mathcal{Q}_\pi(s,a)]$;
    // Critère Atteint ?
    si $\pi_{next} \neq \pi$ **alors**
      $Instable \leftarrow$ true ;

Résultat : une *Politique* $\pi^\star$ optimale

de l'*Opérateur de Bellmann* (où chaque itération est en $\mathcal{O}(|\mathbb{S}|^2)$ mais le nombre d'itérations est difficilement prédictible), soit de résoudre un *Programme Linéaire* (dont la complexité est aussi difficilement évaluable).

La seconde phase exploite directement l'*Opérateur d'Optimalité de Bellmann*. Néanmoins, puisqu'il s'agit de construire une nouvelle *Politique*, une recherche de l'*argument du maximum* est faite en lieu et place d'une maximisation. La nouvelle *Politique* obtenue de manière gloutonne reste meilleure que la *Politique* précédente en vertu de la propriété contractante de l'*Opérateur d'Optimalité de Bellmann*. Cette étape est de complexité moindre ($\mathcal{O}(|\mathbb{A}| \times |\mathbb{S}|^2)$).

L'*Itération de la* Politique, malgré une complexité plus importante, a une caractéristique intéressante : son critère d'arrêt. En effet, l'algorithme s'arrête lorsqu'il n'y a pas eu d'amélioration dans la *Politique* à une itération donnée. La *Politique* est alors optimale.

3.2.5 Amélioration de PI et VI

VI et **PI** sont les deux algorithmes classiques pour la recherche de Politique *optimale*. Toutefois, ils ne sont pas exempts de problèmes. En particulier, ces deux algorithmes reposent sur une mise à jour de la fonction *Valeur* ou de la *Politique* pour tout l'espace d'état $\mathbb{S}$ à chaque itération. Ce qui ne va pas sans poser de difficultés lorsque la taille de l'espace d'état augmente. Les méthodes d'abstraction vues au Chapitre 1 prennent alors tout leur sens. Toutefois, avant de passer aux méthodes d'abstraction, plusieurs améliorations peuvent déjà être apportées.

Méthode de mise à jour

La version présentée des algorithmes **VI** et **PI** suppose que deux tables sont utilisées à chaque itération (les tables $\mathcal{V}_n$ et $\mathcal{V}_{n+1}$ pour **VI**, les tables π_n et π_{n+1} pour **PI**). Les mises à jour consistent alors à modifier toutes les valeurs d'une des tables en fonction de toutes les valeurs de l'autre table. Cette méthode est dite *synchrone* [Sutton and Barto, 1998]. Bien qu'utile pour présenter les algorithmes et les preuves de convergence, elle est améliorable.

Dans un premier temps, il est possible de n'utiliser qu'une table. Les mises à jour des valeurs à chaque itération se font alors directement dans cette seule et unique table. Cette manière de procéder *en place* améliore la convergence des algorithmes. En effet, à chaque itération, les états non encore mis à jour profitent des mises à jour déjà effectuées sur les autres états. Ils jouiront donc de valeurs plus proches de l'optimale pour leurs mises à jour.

Sur cette méthode de mise à jour en place vient se greffer la méthode dite *asynchrone*. Dans les algorithmes exploitant cette méthode, les mises à jour se font alors sans ordre précis. Plus précisément, certaines régions de l'espace d'état peuvent être mises à jour plus fréquemment que d'autres, en fonction de leur importance. Sous la garantie que tous les états sont mis à jour régulièrement, ces algorithmes convergent toujours. Cette technique convient particulièrement aux problèmes où les états de départs possibles sont fixes et connus à l'avance ([Hansen and Zilberstein, 2001], [McMahan et al., 2005]). La résolution se concentre alors sur les états faisant le lien optimal entre états finaux et états de départ.

Itération de la *Politique* Généralisée

Les deux algorithmes **VI** et **PI** sont deux variantes d'un algorithme plus général : l'*Itération de la* Politique *Généralisée* [Sutton and Barto, 1998]. Cet algorithme,

dont **PI** est la retranscription littérale, alterne entre amélioration de la *Politique* à partir d'une fonction *Valeur* et estimation de la fonction *Valeur* associée à cette *Politique*. À un bout du spectre il y a donc **PI** où à chaque itération, le calcul exact de la fonction *Valeur* de la *Politique* courante est effectué pour ensuite améliorer cette *Politique* courante. Et à l'autre il y a **VI** qui calcule directement la fonction *Valeur* de la *Politique* optimale. Modified **PI** [Puterman and Shin, 1978] illustre ce compromis en ne cherchant pas à calculer exactement la fonction *Valeur* de la *Politique* courante, mais seulement une fonction approchée puis en inférant une nouvelle *Politique* à partir de celle-ci.

3.2.6 Autres méthodes de recherche

Les deux méthodes que nous venons de voir, **VI** et **PI**, et leurs améliorations sont des algorithmes basés sur la *Programmation Dynamique* (au travers de l'*Opérateur d'Optimalité de Bellmann*). Ces méthodes ne sont pas les seules pour rechercher la *Politique* optimale.

Programmation Linéaire

Tout comme le calcul de la fonction *Valeur*, le calcul de la fonction *Valeur* optimale peut se faire par *Programmation Linéaire* [Manne, 1960] :

$$\begin{aligned} &\text{Minimiser} && \sum_{s\in\mathbb{S}} \alpha(s)\cdot\mathcal{V}(s) \\ &\text{Sous contrainte :} && \mathcal{V}(s) \geq \mathcal{R}(s,\ a) + \gamma\cdot\sum_{s'\in\mathbb{S}} P(s'\mid s,\ a)\cdot\mathcal{V}(s) \\ & && \forall s\in\mathbb{S},\ \forall a\in\mathbb{A} \end{aligned}$$

Les coefficients $\alpha(s) > 0$ servent à marquer l'intérêt de l'état qui leur est associé. Ces coefficients sont fixés de telle sorte que leur somme soit normalisée($\sum_{s\in\mathbb{S}} \alpha(s) = 1$). Ces coefficients sont sans effet sur la solution optimale.

Comme pour les méthodes de résolution par *Programmation Dynamique*, la résolution par *Programmation Linéaire* implique une connaissance complète du MDP. Toutefois, cette méthode permet elle aussi d'obtenir une solution exacte. Cette méthode est de complexité équivalente aux algorithmes de *Programmation Dynamique*.

Apprentissage par renforcement

Plus un domaine de recherche qu'une méthode en elle-même, l'apprentissage par renforcement [Sutton and Barto, 1998] se consacre à l'étude des algorithmes d'apprentissage de la *Politique* optimale à partir d'expériences. Dans ce cadre de travail, il n'est

plus (nécessairement) question de planifier une *Politique* optimale à partir d'un modèle. Au contraire, il s'agit de laisser l'agent prendre des décisions par lui-même, en de déduire des observations faites en retour la *Politique* optimale.

La Partie III s'intéresse en détails à ces méthodes de recherche aussi nous ne les développons pas ici.

3.2.7 Fléau de la dimension

Le passage à l'échelle vers des problèmes de taille plus grande pose problème pour la planification aussi. En effet, la complexité d'un itération de **VI** en est $\mathcal{O}(|\mathbb{A}| \times |\mathbb{S}|^2)$, celle d'une itération de **PI** en $\mathcal{O}(|\mathbb{A}| \times |\mathbb{S}|^2 + |\mathbb{S}|^3)$. Par conséquent, plus l'espace d'état devient grand, plus ces algorithmes ont besoin de temps pour s'exécuter. À titre d'illustration, le problème FACTORY a un espace d'état de 55296 états et 10 actions. Pour **VI**, la complexité d'une itération est alors de 30 576 476 160 opérations. En admettant que nous ne passons que 10ns à faire chaque opération, il faut plus de 300 secondes pour une itération.

De plus, un nouveau problème se pose : comment stocker en mémoire la fonction *Valeur* et la *Politique*, ainsi que les divers artefacts de calcul (les fonctions *Valeur* d'action entre autre). Les techniques mises en œuvre pour factoriser les fonctions *Transition* et *Récompense* sont-elles utilisables pour ces nouvelles fonctions ?

Nous allons voir dans la section suivante que les représentations factorisées permettent de contourner efficacement ces problèmes générés par la taille imposante de l'espace d'état.

3.3 Planification dans un FMDP

La transformation d'un MDP en un FMDP repose sur la décomposition de son espace d'état $\mathbb{S}$ par un ensemble de variables $\mathbb{X}$ (cf. Partie I). Les fonctions *Transition* $\mathcal{T}$ et *Récompense* $\mathcal{R}$ s'écrivent alors comme des fonctions des variables de $\mathbb{X}$. Les indépendances entre les variables de $\mathbb{X}$ sont exploitées pour décomposer et simplifier ces fonctions. Celles-ci sont alors "résumées" à l'aide d'un ensemble d'instantiations partielles appelées CONTEXTES des variables de $\mathbb{X}$. Des représentations graphiques telles que les ARBRES DE DÉCISION ou les ADDs capturent efficacement ces CONTEXTES et permettent alors de stocker efficacement en mémoire les fonctions *Transition* et *Récompense*.

Nous allons voir que les *Politiques* et les fonctions *Valeur* se reformulent elles aussi à l'aide de $\mathbb{X}$. Elles sont donc elles aussi représentables à l'aide d'ARBRES DE DÉCISION ou

d'ADDs. Mais plus intéressant encore, nous allons voir que les algorithmes **VI** et **PI** sont modifiables pour être aptes à manipuler directement ces représentations graphiques. Chaque itération de ces algorithmes s'exécute alors sans avoir à parcourir tout l'espace d'état $\mathbb{S}$, ou plutôt l'ensemble $\mathcal{D}om\{\mathbb{X}\}$ puisque les deux ensembles sont équivalents.

3.3.1 Décomposition d'état, *Politiques* et fonctions *Valeur*

La décomposition de l'espace d'état implique qu'alors tout état s est en bijection avec une unique instantiation complète $\mathbf{x} \in \mathcal{D}om\{\mathbb{X}\}$ de toutes les variables de $\mathbb{X}$. Dès lors toute fonction de $\mathbb{S}$ dans un autre ensemble $\mathbb{E}$ est aussi une fonction de $\mathcal{D}om\{\mathbb{X}\}$ dans l'ensemble $\mathbb{E}$. Dès lors, on a :

Définition 3.15 (*Politique* et fonction *Valeur* factorisées)**.**

Soit un MDP $\langle \mathbb{S}, \mathbb{A}, \mathbb{T}, \mathcal{T}, \mathcal{R} \rangle$,

Soit $\mathbb{X}$ un ensemble de variables tel que $\mathbb{S}$ est décomposable par cet ensemble,

La *Politique* π est alors la fonction :

$$\begin{array}{rccc} \pi : & \mathcal{D}om\{X_1\} \times \ldots \times \mathcal{D}om\{X_n\} & \longrightarrow & \mathbb{A} \\ & (X_1, \ldots, X_n) & \longmapsto & \pi(X_1, \ldots, X_n) \end{array}$$

Sa performance peut être évaluée à l'aide de la fonction $\mathcal{V}_\pi$:

$$\begin{array}{rccc} \mathcal{V}_\pi : & \mathcal{D}om\{X_1\} \times \ldots \times \mathcal{D}om\{X_n\} & \longrightarrow & \mathbb{R} \\ & (X_1, \ldots, X_n) & \longmapsto & \mathcal{V}_\pi(X_1, \ldots, X_n) \end{array}$$

t.q.

$$\mathcal{V}_\pi(X_1, \ldots, X_n) = \mathcal{R}(X_1, \ldots, X_n,\ \pi(X_1, \ldots, X_n))$$

$$+\gamma \cdot \sum_{X'_1 \in \mathcal{D}om\{X_1\}} \ldots \sum_{X'_n \in \mathcal{D}om\{X_n\}} P(X'_1, \ldots, X'_n \mid X_1, \ldots, X_n,\ \pi(X_1, \ldots, X_n)) \cdot \mathcal{V}_\pi(X'_1, \ldots, X'_n)$$

De fait, la *Politique* et la fonction *Valeur* ont elles aussi leur SUPPORT duquel des CONTEXTES sont extractibles pour factoriser ces fonctions. Elles peuvent donc aussi bénéficier d'une RGF, donnant ainsi la possibilité de les stocker de façon réduite en mémoire. Ces fonctions sont produites algorithmiquement à partir des fonctions *Transition* et *Récompense*. Nous allons voir que leurs représentations structurées (ARBRES DE DÉCISION ou ADDs) sont elles aussi déductibles algorithmiquement.

Pour les trouver, les CONTEXTES vont nous être d'une grande utilité. En effet, les algorithmes de planification factorisée cherchent à tirer partie d'un avantage offert par la factorisation des fonctions à l'aide des CONTEXTES. Rappelons que chacun

de ces CONTEXTES est un ensemble d'états pour lesquels la fonction représentée est constante (c.f. les indépendances contextuelles Section 1.3.3). Cette constance implique que tout résultat lié à un des états du CONTEXTE est applicable à tout autre état de ce CONTEXTE. Pour un CONTEXTE donné, les calculs ne sont donc à faire qu'une seule fois. Pour améliorer les performances des algorithmes **VI** et **PI**, et plus particulièrement contourner la très coûteuse visite de tout l'espace d'état, cette simplification des calculs offerte par les CONTEXTES est donc exploitée. Nous allons maintenant voir comment se met en place cette astuce pour combiner efficacement deux fonctions (la combinaison de fonctions est centrale dans **VI** et **PI**).

3.3.2 Combinaison de Fonctions et CONTEXTES

Une combinaison de fonctions est définie de la manière suivante :

Définition 3.16 (Combinaison de fonctions)**.**

Soient deux fonctions f_1 et f_2 définies sur un même ensemble de départ $\mathbb{E}$. La combinaison $\odot$[11] de ces deux fonctions consiste à créer une troisième fonction f_3 elle-aussi définie sur $\mathbb{E}$ et telle que :

$$\begin{array}{rccl} f_3: & \mathbb{E} & \longrightarrow & \mathbb{F}_3 = \mathbb{F}_1 \odot \mathbb{F}_2 \\ & e & \longmapsto & f_1(e) \odot f_2(e) \end{array}$$

Les fonctions définies à l'aide d'ensembles de CONTEXTES sont des fonctions constantes sur ces CONTEXTES. Aussi :

Proposition 3.17. *Si deux fonctions définies sur un même ensemble de variables $\mathbb{X}$ ont toutes deux un même* CONTEXTE *pour lequel elles sont constantes, alors l'algorithme de combinaison de ces deux fonctions peut et ne devrait faire le calcul de combinaison qu'une fois pour toutes les instanciations de $\mathcal{D}om\{\mathbb{X}\}$ compatibles avec ce* CONTEXTE *(et autant que possible le stocker qu'une seule fois).*

Trouver et exploiter ces CONTEXTES offrent un gain en temps de calcul considérable. Toutefois, de manière générale, deux fonctions combinées n'ont pas nécessairement des CONTEXTES exactement identiques pour lesquels elles sont constantes. En revanche, elles peuvent avoir des CONTEXTES compatibles entre eux, i.e. tel que si une variable de $\mathbb{X}$ est instanciée dans les deux CONTEXTES, elle soit instanciée à la même valeur.

Définition 3.18 (CONTEXTES compatibles)**.**

11. Nous notons $\odot$ l'opérateur binaire de combinaison qui peut être une addition, multiplication, maximisation ou tout autre combinaison de deux termes.

Soient deux CONTEXTES $\mathbf{c}_1$ et $\mathbf{c}_2$ des sous-ensembles de variables $\mathbf{C}_1$ et $\mathbf{C}_2$ de $\mathbb{X}$.

$$\mathbf{c}_1 \text{ et } \mathbf{c}_2 \text{ sont compatibles } \iff \forall X_i \in \mathbf{C}_1 \bigcap \mathbf{C}_2,\ \mathbf{c}_1[X_i] = \mathbf{c}_2[X_i]$$

où $\mathbf{c}_k[X_i] \in \mathcal{D}om\{X_i\}$ est l'instanciation x_i de X_i dans $\mathbf{c}_k$.

À partir de deux CONTEXTES compatibles, il est possible de définir un troisième CONTEXTE issu des deux premiers :

Définition 3.19 (Union de deux CONTEXTES)**.**

Soient deux CONTEXTES $\mathbf{c}_1$ et $\mathbf{c}_2$ des sous-ensembles de variables $\mathbf{C}_1$ et $\mathbf{C}_2$ de $\mathbb{X}$.

Si $\mathbf{c}_1$ et $\mathbf{c}_2$ sont compatibles ,

Alors l'union des deux CONTEXTES est le CONTEXTE $\mathbf{c}_u = \mathbf{c}_1 \bigcup \mathbf{c}_2$ de l'ensemble de variables $\mathbf{C}_1 \bigcup \mathbf{C}_2 \subseteq \mathbb{X}$ t.q. $\forall X_i \in \mathbf{C}_1$ (resp. $\mathbf{C}_2$), $\mathbf{c}_u[X_i] = \mathbf{c}_1[X_i]$ (resp. $\mathbf{c}_u[X_i] = \mathbf{c}_2[X_i]$).

Pour ce contexte, la propriété suivante est alors vraie :

Propriété 10 (Constance par rapport à l'union de deux CONTEXTES)**.**

Soient deux CONTEXTES $\mathbf{c}_1$ *et* $\mathbf{c}_2$ *des sous-ensembles de variables* $\mathbf{C}_1$ *et* $\mathbf{C}_2$ *de* $\mathbb{X}$ *tels que* $\mathbf{c}_1$ *et* $\mathbf{c}_2$ *sont compatibles.*

Soient deux fonctions f_1 *et* f_2 *telles que* $f_1 \perp\!\!\!\perp \overline{\mathbf{C}_1} \mid \mathbf{c}_1$ *et* $f_2 \perp\!\!\!\perp \overline{\mathbf{C}_2} \mid \mathbf{c}_2$.

$$\mathbf{c}_u = \mathbf{c}_1 \bigcup \mathbf{c}_2 \Longrightarrow f_1 \perp\!\!\!\perp \overline{\mathbf{C}_u} \mid \mathbf{c}_u \text{ et } f_2 \perp\!\!\!\perp \overline{\mathbf{C}_u} \mid \mathbf{c}_u$$

où $\overline{\mathbf{C}_u}$ *est le complémentaire de* $\mathbf{C}_1 \bigcup \mathbf{C}_2$ *dans* $\mathbb{X}$.

Et alors, f_1 *et* f_2 *sont constantes pour toutes instanciations des variables de* $\overline{\mathbf{C}_u}$.

Démonstration.

Nous avons $\mathbf{C}_1 \subseteq \mathbf{C}_u$ (resp. $\mathbf{C}_2 \subseteq \mathbf{C}_u$).

Donc $\overline{\mathbf{C}_u} \subseteq \overline{\mathbf{C}_1}$ (resp. $\overline{\mathbf{C}_u} \subseteq \overline{\mathbf{C}_2}$).

Or $f_1 \perp\!\!\!\perp \overline{\mathbf{C}_1} \mid \mathbf{c}_1$ (resp. $f_2 \perp\!\!\!\perp \overline{\mathbf{C}_2} \mid \mathbf{c}_2$).

Donc $f_1 \perp\!\!\!\perp \overline{\mathbf{C}_u} \mid \mathbf{c}_1$ (resp. $f_2 \perp\!\!\!\perp \overline{\mathbf{C}_u} \mid \mathbf{c}_2$). □

Deux CONTEXTES compatibles permettent donc de définir une troisième CONTEXTE pour lequel f_1 et f_2 sont constantes. Dès lors un algorithme efficace de combinaison de deux fonctions est un algorithme de parcours des ensembles de CONTEXTES qui définissent ces fonctions. Cet algorithme les parcoure à la recherche de CONTEXTES compatibles. Lorsqu'il en trouve deux, il en déduit l'union, et calcule la combinaison des deux fonctions par rapport à ce nouveau CONTEXTE.

Algorithme 6 : COMBINAISON DE FONCTION

Données : une fonctions f_1 de $\mathcal{D}om\{\mathbb{X}\} \longrightarrow \mathbb{E}_1$, f_2 de $\mathcal{D}om\{\mathbb{X}\} \longrightarrow \mathbb{E}_2$, et $\odot$ une opération de $\mathbb{E}_1 \times \mathbb{E}_2 \longrightarrow \mathbb{E}_r$

```
1 début
2     pour Chaque CONTEXTE c_1 de f_1 faire
3         pour Chaque CONTEXTE c_2 de f_2 faire
4             si c_1 et c_2 sont compatibles alors
5                 c_u ← c_1 ⋃ c_2 ;
6                 f_r(c_u) ← f_1(c_u) ⊙ f_2(c_u) ;
```

Résultat : f_r de $\mathcal{D}om\{\mathbb{X}\} \longrightarrow \mathbb{E}_r$ t.q. $\forall \mathbf{X} \in \mathcal{D}om\{\mathbb{X}\}$, $f_r(\mathbf{X}) = f_1(\mathbf{X}) \odot f_2(\mathbf{X})$

L'Algorithme **COMBINAISON DE FONCTION** (6) résume la combinaison de deux fonctions. Itérer sur les ensembles de CONTEXTES des deux fonctions à la recherche de CONTEXTES compatibles est a priori coûteux. Dans les faits, les RGFs permettent de faire cette recherche de manière très efficace.

3.3.3 Opérateur de Combinaison et Arbres de Décision

Les ARBRES DE DÉCISION sont relativement efficaces pour simplifier la représentation d'une fonction définie sur un ensemble de variables en exploitant les CONTEXTES (cf. Partie I). Boutilier et al. [2000] s'appuie sur cette représentation pour d'une part factoriser un MDP (cf Partie I) mais aussi pour rechercher plus efficacement une *Politique* optimale. Pour effectuer les combinaisons de fonctions nécessaires à cette recherche en exploitant les représentations des fonctions sous forme de ARBRES DE DÉCISION, il utilise deux algorithmes de manipulation de ces arbres : les algorithmes **AJOUTER ARBRE** et **SIMPLIFIER ARBRE** (illustré par la Figure 3.1).

AJOUTER ARBRE$(T_1, T_2, l, \odot)$ Cette méthode retourne un ARBRE DE DÉCISION reproduisant l'arbre T_1 à l'exception de sa feuille l remplacée par l'arbre T_2. Avant de faire la jonction, la valeur stockée en chaque feuille de T_2 est combinée avec la valeur en l par l'opération $\odot$ de combinaison. La feuille l est ensuite remplacée par la racine de T_2.

Toutefois cette méthode appliquée seule laisse l'arbre résultant avec des redondances. En effet, rien interdit à T_1 et à T_2 d'avoir des nœuds associés aux mêmes variables. En remplaçant simplement la feuille l de T_1 par la racine T_2, nous faisons

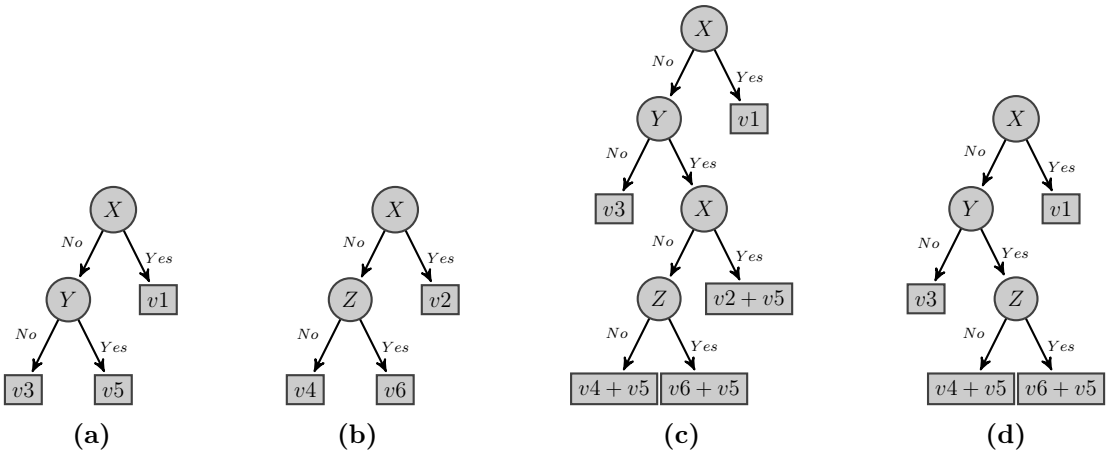


FIGURE 3.1 – *Soient T_1 (a) et T_2 (b). La procédure* **AJOUTER ARBRE**$(T_1, T_2, v5, +)$ *produit l'arbre T_3 (c) que la procédure* **SIMPLIFIER ARBRE** *réduit en l'arbre T_4 (d).*

alors potentiellement apparaître plus d'une fois une même variable, sur un chemin allant de la racine à une feuille de l'arbre nouvellement créé. Ces redondances de variables doivent être retirées.

SIMPLIFIER ARBRE(T) Cette méthode se charge donc de parcourir l'arbre et de retirer les parties redondantes. Principalement, elle détecte si plusieurs nœuds internes d'une même branche de l'arbre sont liés à la même variable. Si tel est le cas, tous ces nœuds sauf le nœud le plus haut dans l'arbre doivent être retirés. Ce retrait doit toutefois être fait de sorte que le nouvel arbre reste cohérent.

Ainsi, pour une variable v, en une branche b, soient n_h et n_r deux nœuds internes liés à v tel que n_r soit plus "bas" que n_h dans le graphe et donc soit redondant. Pour "descendre" de n_h vers n_r, l'arc x_h a été emprunté ; v est alors instanciée à la valeur x_h. Au nœud n_r, il faut donc "brancher" sur cette modalité x_h pour que l'arbre reste cohérent.

Retirer un nœud n_r consiste à insérer un arc de son parent à son fils n_r.**FILS**(x_h). Les autres fils, leurs sous-arbres associés, et le nœud n_r sont alors retirés du graphe.

La méthode **AJOUTER ARBRE** revient donc à placer en fin d'un CONTEXTE de T_1 tous les CONTEXTES de T_2 (graphiquement parlant). La méthode **SIMPLIFIER ARBRE** vient alors corriger la structure, ne conservant que les CONTEXTES compatibles. L'application cumulée de ces deux méthodes crée donc des unions de CONTEXTES et calcule les valeurs de combinaison de f_1 et f_2 pour ces unions. L'application de ces deux méthodes [12] à toute feuille de T_1 pour l'étendre par T_2 crée alors un nouvel ARBRE DE

12. Cette nouvelle méthode sera nommée **ÉTENDRE**$(T_1, T_2, \odot)$

DÉCISION qui est la combinaison des arbres T_1 et T_2 par $\odot$ (et donc des fonctions f_1 et f_2).

Cette méthode **ÉTENDRE** est assez coûteuse. En effet, chaque appel à **AJOUTER ARBRE** implique de dupliquer T_2 pour le placer en bas de T_1. L'arbre T_2 est alors copié en mémoire autant de fois que T_1 possède de feuilles. Or chaque copie est un parcours *Profondeur d'Abord* de T_2. Cette duplication est d'autant plus coûteuse que nous ne conservons finalement qu'une sous-partie de T_2 après détection [13] et retrait des redondances.

Pour ces raisons, Boutilier et al. [2000] suggère une autre méthode de combinaison : ordonner les deux arbres et les parcourir à la recherche de CONTEXTES compatibles. Cette méthode est une ébauche de l'algorithme de combinaison des BDDs [Bryant, 1986] et donc des ADDs. Nous y revenons plus loin.

3.3.4 Combinaison d'arbres et Recherche de Politique optimale

L'utilisation de l'*Opérateur d'Optimalité de Bellman*, dans **VI** ou **PI** pour converger vers la *Politique* optimale, se décompose en trois étapes :

1. Calcul l'espérance de la *récompense futur espérée*,
2. Ajout la récompense immédiate,
3. Maximisation (ou recherche d'argument du maximum) sur toutes les actions.

Les étapes 2.) et 3.) sont facilement transformables en combinaison d'ARBRES DE DÉCISION à l'aide de **ÉTENDRE** ; L'opération de combinaison est alors soit la maximisation, soit l'addition.

Le calcul d'espérance est lui plus complexe car il implique de calculer les quantités :

$$\rho(X_1,\ldots,X_n) = \gamma\cdot \sum_{X'_1\in\mathcal{D}om\{X_1\}} \ldots \sum_{X'_n\in\mathcal{D}om\{X_n\}} P(X'_1,\ldots,X'_n \mid X_1,\ldots,X_n,\ a)\cdot\mathcal{V}_\pi(X'_1,\ldots,X'_n)$$

Ce calcul pose donc deux difficultés : d'une part la recomposition de la distribution jointe de probabilité $P(X'_1,\ldots,X'_n \mid X_1,\ldots,X_n,\ a)$, d'autre part le calcul des sommes sur les domaines des variables à $t+1$ (et de facto leur élimination).

Recomposition de la distribution jointe

Pour stocker la fonction *Transition* en mémoire, celle-ci a été décomposée en plusieurs lois marginales stockées sous forme d'ARBRES DE DÉCISION (un ARBRE DE

13. Qui requière elle aussi un parcours en profondeur.

Algorithme 7 : PREGRESS : Recomposition de la distribution jointe de transition pour une action a

Données : $T[\mathcal{V}]^{\downarrow n_r}$, le sous-arbre de $T[\mathcal{V}]$, l'arbre représentant $\mathcal{V}$, ayant n_r pour racine, et a, l'action sur laquelle cette distribution jointe est calculée.

début
  // Si $T[\mathcal{V}]^{\downarrow n_r}$ est une feuille
  si $T[\mathcal{V}]^{\downarrow n_r} = \{l\}$ **alors**
    retourner {};
  pour $x_r \in \mathcal{D}om\{n_r.var\}$ **faire**
    $T[P_a, n_r.\textbf{FILS}(x_r)] \leftarrow \textbf{PREGRESS}(T[\mathcal{V}]^{\downarrow n_r.\textbf{FILS}(x_r)}, a)$; // où $T[\mathcal{V}]^{\downarrow n_r.\texttt{fils}(x_r)}$ est le graphe des Descendants pour le nœud $n_r.\texttt{fils}(x_r)$
  // Soit $T[P_{a,n_r.var}]$ est l'ARBRE DE DÉCISION représentant la fonction *Transition* de la variable $n_r.var$ pour l'action a
  $T[P_a, n_r] \leftarrow T[P_{a,n_r.var}]$;
  pour chaque *feuille* l *de* $T[P_{a,n_r.var}]$ **faire**
    $T_l \leftarrow \{\}$;
    pour $x_r \in \mathcal{D}om\{n_r.var\}$ **faire**
      // Soit alors $P^{(l)}$ la distribution de probabilité sur $n_r.var$ en la feuille l
      si $P^{(l)}(x_r) \neq 0$ **alors**
        $T_l \leftarrow \textbf{ÉTENDRE}(T_l, T[P_a, n_r.\textbf{FILS}(x_r)], \bigcup)$;
    $T[P_a, n_r] \leftarrow \textbf{AJOUTER ARBRE}(T[P_a, n_r], T_l, l, \bigcup)$;
  retourner $T[P_a, n_r]$;

Résultat : $T[P_a, n_r]$, l'arbre représentant la distribution de probabilités jointe de transition des variables présentent dans $T[\mathcal{V}]^{\downarrow n_r}$ pour l'action a

DÉCISION $T[P_{a,X_i}]$ par action et par variable). Pour pouvoir l'utiliser lors de l'application de l'*Opérateur d'Optimalité de Bellman*, il est nécessaire de la recomposer ; plus précisément de recomposer les ARBRES DE DÉCISION $T[P_a]$ liés à chaque action a.

Toutefois, calculer ces distributions jointes sur toutes les variables à $t+1$ n'est pas nécessaire en plus d'être coûteux en mémoire et en temps de calcul. En effet, le calcul de l'espérance implique de sommer sur le domaine des variables à $t+1$. Cette somme est de la forme $\sum_{x'_i} P(X'_i = x'_i) \cdot f(x'_i)$. Or si $f(X'_i)$ est constante et égale à v $\forall x'_i \in \mathcal{D}om\{X'_i\}$ [14], cette somme est égale à v puisque $\sum_{x'_i} P(X'_i = x'_i) = 1$. Par conséquence, seule la distribution jointe sur les variables du SUPPORT $(\mathcal{V})$ est à calculer.

Lors du parcours en profondeur de l'arbre $T[\mathcal{V}]$, la liste des variables rencontrées sur les différentes branches de l'arbre donne le SUPPORT $(\mathcal{V})$. Une fois ce support obtenu, la

14. En d'autres termes, f est indépendante de X'_i.

combinaison, pour chaque action, des arbres $T[P_{a,X_i}]$ donne l'arbre $T[P_a]$ représentant la probabilité jointe

$$P(\text{SUPPORT}(\mathcal{V}) \mid \bigcup_{X'_i \in \text{SUPPORT}(\mathcal{V})} \text{PARENTS}(X'_i),\ a)$$

pour l'action a. L'Algorithme **PREGRESS** (7) donne le détail de ces calculs.

L'opération de combinaison [15] effectuée au niveau des feuilles consiste à unir des tables de probabilités. En effet, chaque arbre $T[P_{a,X_i}]$ stocke en chaque feuille une distribution de probabilité $P(X_i)$ sous forme de table. L'union de tables pour les probabilités $P(X_i)$ et $P(X_j)$ consiste simplement à créer la table $P(X_i, X_j)$. Au final, chaque feuille de $T[P_a]$ est la table d'une distribution $P(\text{SUPPORT}(\mathcal{V}))$

Calcul de l'espérance

Une fois l'arbre représentant la distribution jointe obtenu, l'espérance est calculable. Ce calcul consiste en un nouveau parcours *Profondeur d'Abord* de $T[\mathcal{V}]$.

Cette fois-ci, le chemin suivi pour atteindre une feuille l est mémorisé. Soit $\mathbf{c}_l$ le CONTEXTE associé à ce chemin. Chaque fois qu'une feuille est atteinte dans $T[\mathcal{V}]$, toutes les feuilles de $T[P_a]$ sont mises à jour. En effet, pour chacune des feuilles de $T[P_a]$, plusieurs entrées de la table de probabilités associée sont compatibles avec $\mathbf{c}_l$. Pour chacune de ces entrées, la probabilité stockée est alors remplacée par le produit de celle-ci avec la valeur v_l de la feuille l de $T[\mathcal{V}]$.

Une fois $T[\mathcal{V}]$ entièrement exploré, toutes les entrées des tables aux feuilles de $T[P_a]$ ont été mises à jour par le produit $P(\mathbf{c}) \cdot \mathcal{V}(\mathbf{c})$. Pour chaque feuille, toutes les entrées de la table sont alors sommées (équivalent à faire la somme sur les variables à $t+1$) pour obtenir l'espérance sur les récompenses futures en cette feuille.

L'Algorithme **QREGRESS** (8) donne l'implémentation de ce calcul, et effectue les autres opérations nécessaires pour obtenir la fonction *Valeur* d'action $T[\mathcal{Q}_a]$ liée à l'action a (à savoir : multiplier par γ chaque feuille, et combiner par somme avec l'ARBRE DE DÉCISION $T[\mathcal{R}_a]$ donnant la fonction *Récompense* pour l'action courante).

Itération de la Valeur Structurée et Itération de la *Politique* Structurée

Grâce à ces algorithmes, l'Algorithme **VI** (4) est entièrement exécutable en manipulant uniquement des ARBRES DE DÉCISION. La recherche d'une *Politique* ϵ-optimale

15. Notée $\bigcup$.

Algorithme 8 : QREGRESS : Calcul de la fonction *Valeur* d'action pour l'action a

Données : $T[\mathcal{V}]$, l'arbre représentant $\mathcal{V}$, et a, l'action pour laquelle cette fonction est calculée.

début
  // Calcul de l'espérance sur les récompenses à $t+1$
  $T[P_a] \leftarrow$ **PREGRESS**$(T[\mathcal{V}], a)$;
  pour $\forall$ CONTEXTE **c** *de* $T[\mathcal{V}]$ **faire**
    pour $\forall$ *feuille* l *de* $T[P_a]$ **faire**
      pour $\forall$ *entrée* $\mathbf{e}_l$ *compatible avec* **c faire**
        Remplacer $P_l(\mathbf{e}_l)$ par $v_l(\mathbf{e}_l) = P_l(\mathbf{e}_l) \cdot \mathcal{V}(c)$;
  pour $\forall$ *feuille* l *de* $T[P_a]$ **faire**
    $v_l \leftarrow \gamma \cdot \sum_{\forall \text{ entrée } \mathbf{e}_l} v_l(\mathbf{e}_l)$;
    Remplacer la table de probabilité en l par v_l;
  // Calcul de $\mathcal{Q}_a$
  retourner $T[\mathcal{Q}_a] \leftarrow$ **ÉTENDRE**$(T[P_a], T[\mathcal{R}_a], +)$;

Résultat : $T[\mathcal{Q}_a]$, l'arbre représentant la nouvelle estimation de la fonction *Valeur* d'action optimale pour l'action a

se fait alors sans avoir à parcourir tout l'espace d'état à chaque itération. L'Algorithme **SVI** (9) donne le détail de cette implémentation exploitant les ARBRES DE DÉCISION comme représentations structurées.

L'Algorithme **PI** (5) demande quelques étapes supplémentaires pour être adapté aux arbres. Rappelons que **PI** (5) a deux étapes : une étape d'évaluation de la *Politique* actuelle et une étape d'amélioration de cette *Politique*. L'étape d'amélioration est similaire à l'étape de récupération gloutonne de la *Politique* dans **VI** et ne pose donc pas de problème.

L'étape d'évaluation de la performance en revanche est plus problématique. Trois possibilités sont disponibles pour faire ce calcul : résolution d'un système linéaire, application récursive de l'*Opérateur de Bellman* Γ_π associé à la *Politique* π et résolution d'un programme linéaire. La solution suivie par Boutilier et al. [1999] est de se servir de l'*Opérateur de Bellman* Γ_π.

En effet, cet opérateur est, comme l'*Opérateur d'Optimalité de Bellman*, adaptable aux ARBRES DE DÉCISION. Pour se faire, plusieurs arbres doivent être calculés : les arbres $T[P_{\pi,X_i}]$ représentant les distributions de probabilités de *Transition* $P(X'_i \mid \text{PARENTS}(X'_i), \pi(\text{PARENTS}(X'_i)))$ en suivant la *Politique* π, l'arbre $T[\mathcal{R}_\pi]$ représentant les récompenses reçues en suivant la *Politique* π, et un arbre $T[\mathcal{V}]$ représentant une estimation de la fonction *Valeur* $\mathcal{V}_\pi$ associée à la *Politique* π.

Algorithme 9 : SVI : Itération de la Valeur Structurée

Données : un FMDP où les fonctions *Transition* et *Récompense* sont factorisées à l'aide d'arbre

début
  `// Calcul de la fonction Valeur initiale`
  $T[\mathcal{V}] \leftarrow \{\}$;
  pour chaque $a \in \mathbb{A}$ **faire**
    $T[\mathcal{V}] \leftarrow$ **ÉTENDRE**$(T[\mathcal{V}], T[\mathcal{R}_a], \max)$;
  $T[\mathcal{V}_{old}] \leftarrow \{\}$;
  `// Itération de la valeur`
  tant que $\exists$ *feuille* $l \in$ **ÉTENDRE**$(T[\mathcal{V}], T[\mathcal{V}_{old}], -)$ *t.q.* $l.val \geq \frac{\epsilon \cdot (1-\gamma)}{2 \cdot \gamma}$ **faire**
    $T[\mathcal{V}_{old}] \leftarrow T[\mathcal{V}]$, $T[\mathcal{V}] \leftarrow \{\}$;
    pour chaque $a \in \mathbb{A}$ **faire**
      $T[\mathcal{Q}_a] \leftarrow$ **QREGRESS**$(T[\mathcal{V}_{old}], a)$;
      $T[\mathcal{V}] \leftarrow$ **ÉTENDRE**$(T[\mathcal{V}], T[\mathcal{Q}_a], \max)$;
  `// Récupération gloutonne de la Politique` ϵ`-optimale`
  $T[\pi] \leftarrow \{\}$;
  pour chaque $a \in \mathbb{A}$ **faire**
    $T[\pi] \leftarrow$ **ÉTENDRE**$(T[\pi], T[\mathcal{Q}_a], \text{argmax})$;

Résultat : $T[\pi]$ l'arbre représentant une *Politique* ϵ-optimale pour le FMDP

Ces nouveaux arbres sont induits à l'aide de la fonction **ÉTENDRE** et à partir de l'arbre $T[\pi]$ et de tous les arbres $T[P_{a,X_i}]$, $T[\mathcal{R}_a]$ et $T[\mathcal{Q}_a]$ respectivement [16]. Ainsi, chaque feuille l de $T[\pi]$ est un CONTEXTE dans lequel l'agent suit l'action a_l. Donc, dans ce CONTEXTE, les probabilités de *Transition* sont celles liées à l'action a_l. De même, la fonction *Valeur* est la fonction *Valeur* d'action liée à cette action. L'opérateur de combinaison $\triangleright$, que **ÉTENDRE**$(T[\pi], T, \triangleright)$ utilise pour effectuer ces associations, doit donc se comporter de la manière suivante :

— Si, pour la feuille l de l'arbre $T[\pi]$, l'arbre T est lié à l'action a_l,
— Alors cette feuille l est remplacée par l'arbre T (toutefois, les feuilles de T restent inchangées),
— Sinon ne rien faire (i.e., la feuille l est laissée intacte).

La Figure 3.2 illustre comment cet opérateur procède. L'arbre $T[\pi]$ est alors étendu avec chaque $T[\mathcal{Q}_a]$ pour obtenir $T[\mathcal{V}]$. De la même manière, $T[\pi]$ est alors étendu avec chaque $T[\mathcal{R}_a]$ pour obtenir $T[\mathcal{R}_\pi]$. Enfin, pour chaque X_i, l'arbre $T[\pi]$ est alors étendu avec les arbres $T[P_{a,X_i}]$ de chaque action pour obtenir $T[P_{\pi,X_i}]$.

L'estimation de la *Politique* se fait alors en appliquant itérativement l'Algorithme

16. Par exemple, $T[P_{\pi,X_i}]$ nécessite tous les arbres $T[P_{a,X_i}]$ et l'arbre $T[\pi]$.

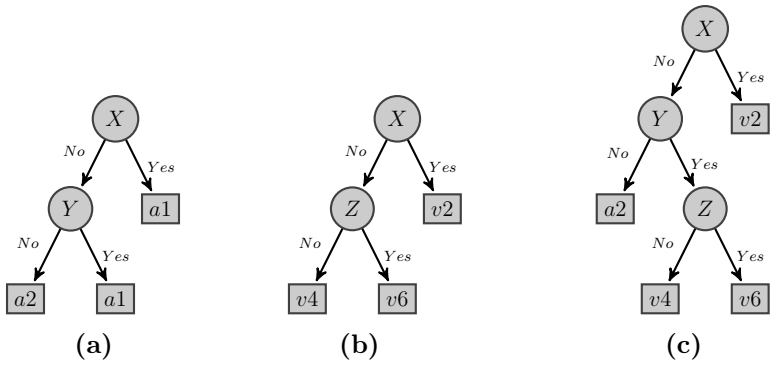


FIGURE 3.2 – *Soient $T[\pi]$ (a) et $T[\mathcal{Q}_{a1}]$ (b). La procédure* ***Etendre****$(T[\pi], T[\mathcal{Q}_{a1}], \triangleright)$ produit l'arbre T_3 (c). Cet arbre sera à son tour combiner avec $T[\mathcal{Q}_{a2}]$ par* ***Etendre****$(T_3, T[\mathcal{Q}_{a2}], \triangleright)$ pour obtenir $T[\mathcal{V}_\pi]$.*

QREGRESS (8) à l'arbre $T[\mathcal{V}]$. Les distributions de probabilité utilisées pour calculer la distribution jointe sur les variables à $t+1$ sont les arbres $T[P_{\pi,X_i}]$. L'arbre de récompense utilisé est naturellement $T[\mathcal{R}_\pi]$. Cet algorithme, nommé **SPE**, ne cherche qu'à calculer une estimation de la fonction *Valeur* liée à la *Politique*, aucun critère d'arrêt n'est donc prédéterminé. Ce peut être un critère d'ϵ-optimalité, ou un nombre fixe d'itération[17].

L'Algorithme **SPI** (11) s'exécute comme l'Algorithme **PI** (5). Il consiste à itérer jusqu'à stabilisation de la *Politique* qui est alors optimale. Chaque itération se déroule en deux phases : estimation de la *Politique* à l'aide de l'Algorithme **SPE** (10), puis maximisation sur les fonctions *Valeur* d'action.

Remarques

Ces deux algorithmes **SVI** et **SPI** permettent donc d'effectuer une recherche de *Politique* optimale sans avoir à énumérer l'espace d'états dans son entier. Ils exploitent en effet les CONTEXTES qui nous ont permis de factoriser le FMDP au départ. Utilisant plusieurs sous-algorithmes, ils construisent alors la solution optimale en ne manipulant que ces CONTEXTES (par l'intermédiaire d'arbre). Toutefois les sous-algorithmes utilisés en interne posent quelques soucis.

Premièrement, les méthodes **AJOUTER ARBRE** et **SIMPLIFIER ARBRE** reposent sur beaucoup de parcours de graphes. En effet, l'algorithme **AJOUTER ARBRE**, qui étend T_1 par T_2, impose de faire une duplication de T_2 pour l'installer. Cette duplication impose un premier parcours en profondeur. Ensuite, la méthode **SIMPLIFIER ARBRE** parcours l'arbre obtenu pour simplifier les zones redondantes. En plus d'un nouveau

17. Auquel cas c'est une GPI qui est effectué

Algorithme 10 : SPE : Estimation de la Politique Structurée

Données : un FMDP où les fonctions *Transition* et *Récompense* sont factorisées à l'aide d'arbre, $|\mathbb{A}|$ arbres $T[\mathcal{Q}_a]$ représentant une estimation des fonctions *Valeur* d'action, et $T[\pi]$ la *Politique* à estimer

début
  // Initialisation
  $T[\mathcal{V}] \leftarrow T[\pi]$; $T[\mathcal{R}\pi] \leftarrow T[\pi]$;
  pour chaque $X_i \in \mathbb{X}$ **faire**
    $T[P_{\pi,X_i}] \leftarrow T[\pi]$;
  pour chaque $a \in \mathbb{A}$ **faire**
    $T[\mathcal{V}] \leftarrow$ **ÉTENDRE**$(T[\mathcal{V}], T[\mathcal{Q}_a], \triangleright)$;
    $T[\mathcal{R}_\pi] \leftarrow$ **ÉTENDRE**$(T[\mathcal{V}], T[\mathcal{R}_a], \triangleright)$;
    pour chaque $X_i \in \mathbb{X}$ **faire**
      $T[P_{\pi,X_i}] \leftarrow$ **ÉTENDRE**$(T[P_{\pi,X_i}], T[P_{a,X_i}], \triangleright)$;
  // Estimation de $\mathcal{V}_\pi$
  tant que *Critère non atteint* **faire**
    $T[\mathcal{V}] \leftarrow$ **QREGRESS**$(T[\mathcal{V}], \pi)$;

Résultat : $T[\mathcal{V}]$ l'estimation de la fonction *Valeur* pour la *Politique* π

parcours, cette méthode rend inefficace la méthode précédente car des pans entiers de l'arbre dupliqué T_2 vont être élagués.

Deuxièmement, l'Algorithme **PREGRESS** (7) produit un arbre dont les feuilles sont des distributions de probabilités sur les variables à $t+1$. Le problème est que ces distributions sont stockées sous forme de tables multidimensionnelles. La taille de ces tables dépend directement des variables présentes dans SUPPORT (f) qui peut être égale à $\mathbb{X}$. Dans ce cas, tout le procédé de factorisation a servi à rien.

Enfin, la structure même utilisée est problématique. Les ARBRES DE DÉCISION sont facilement très larges, et possèdent donc un très grand nombre de feuilles. Sachant que les algorithmes impliquent de lourdes opérations en ces feuilles, la recherche de *Politique* optimale se révèle problématique même avec ces structures.

Les ADDs, et leurs algorithmes de combinaison, offrent une solution plus efficace.

3.3.5 Combinaison d'ADDs et Recherche de *Politique* optimal

Les ADDs sont une version réduite des ARBRES DE DÉCISION où les sous-graphes isomorphes sont fusionnés (cf. Chapitre 1). Pour faciliter la recherche et la fusion de ces sous-graphes, un ordre complet $\succ$ d'apparition des variables sur chaque branche est imposé. Cet ordre permet également de combiner deux ADDs efficacement. Comme cet algorithme de combinaison s'étend naturellement aux variables multivariées, nous

Algorithme 11 : SPI : Itération de la Politique Structurée

Données : un FMDP où les fonctions *Transition* et *Récompense* sont factorisées à l'aide d'arbre

début
  // Calcul de la *Politique* initiale
  $T[\pi] \leftarrow \{\}$;
  pour chaque $a \in \mathbb{A}$ **faire**
    $T[\pi] \leftarrow \textbf{ÉTENDRE}(T[\pi], T[\mathcal{R}_a], \text{argmax})$; $T[\mathcal{Q}_a] \leftarrow T[\mathcal{R}_a]$;
  $T[\pi_{old}] \leftarrow \{\}$;
  // Itération de la Politique
  tant que $T[\pi] \neq T[\pi_{old}]$ **faire**
    $T[\pi_{old}] \leftarrow T[\pi]$;
    // Estimation de $\mathcal{V}_\pi$
    $T[\mathcal{V}] \leftarrow \textbf{SPE}(all\ T[\mathcal{Q}_a], T[\pi])$;
    // Récupération gloutonne de la *Politique* ϵ-optimale
    $T[\pi] \leftarrow \{\}$;
    pour chaque $a \in \mathbb{A}$ **faire**
      $T[\mathcal{Q}_a] \leftarrow \textbf{QREGRESS}(T[\mathcal{V}], a)$;
      $T[\pi] \leftarrow \textbf{ÉTENDRE}(T[\pi], T[\mathcal{Q}_a], \text{argmax})$;

Résultat : $T[\pi]$ l'arbre représentant une *Politique* optimale pour le FMDP

allons donner l'algorithme générale de combinaison de deux RGFORs ayant un ordre similaire.

Énumération de CONTEXTES et exploration *profondeur d'abord*

L'algorithme de combinaison repose sur une exploration *profondeur d'abord* simultanée des deux RGFORs G_1 et G_2. Un algorithme de parcours *profondeur d'abord* est une simple procédure appelée récursivement et ayant comme argument un nœud du graphe exploré. En partant du nœud racine, cette méthode s'appelle à nouveau sur chacun des fils du nœud passé en argument. Cette succession d'appel ne stoppe que lorsque le nœud courant n'a aucun fils (et est donc une feuille).

Or, pour une RGF, appeler cette méthode sur le fils d'un nœud n revient à instancier la variable $n.var$ associée à ce nœud[18]. Cette variable reste instanciée durant tout l'appel. Des CONTEXTES sont donc instanciés au fur et à mesure. En particulier, lorsqu'un appel est fait sur une feuille, la fonction représentée est constante pour le CONTEXTE appelant. Les explorations *profondeurs d'abord* des RGFORs sont donc intéressantes car elles permettent d'énumérer un à un les CONTEXTES pour lesquels la fonction représentée est constante.

18. Elle est alors instanciée à $x_{n.var}$ t.q. $n.\textbf{FILS}(x_{n.var}) = n_{fils}$

Combinaison avec contrainte d'ordre global commun [Bryant, 1986]

La combinaison de deux diagrammes G_1 et G_2 repose donc sur une exploration *profondeur d'abord* simultanée des deux graphes. Ce parcours se fait par appels récursifs à la méthode **EXPLORATION ORDONNÉE** (12). Chaque appel est lancé en fournissant en argument une paire de nœuds $n_1 \in G_1$ et $n_2 \in G_2$ La méthode **COMBINAISON ORDONNÉE**$(G_1, G_2, \odot)$ initie cette exploration en appelant **EXPLORATION ORDONNÉE**$(G_1, G_2, G_1.racine, G_2.racine, \odot)$ sur les racines respectives de deux graphes. Soit $\succ$ l'ordre d'apparition des variables imposé à G_1, G_2 et G (la RGFOR retournée) pour la combinaison. Chaque appel récursif se déroule alors de la manière suivante.

Au début de l'appel, la fonction **EXPLORATION ORDONNÉE** (12) détermine comment ces nœuds devraient être explorés

1. si n_1 et n_2 sont tous deux terminaux, alors $n_1.val \odot n_2.val$ est évalué ;
2. sinon une variable à explorer X_{explo} est sélectionnée entre $n_1.var$ ou $n_2.var$ à l'aide de $\succ$ [19].

La fonction **EXPLORATION ORDONNÉE** (12) lance alors plusieurs appels successifs à elle-même ; un appel est effectué pour chaque modalité x_{explo} que X_{explo} peut assumer. À chacun de ces appels, pour chaque RGF, le nœud alors donné en argument est soit le nœud courant n si $n.var \neq X_{explo}$ ou si n est terminal, soit $n.$**FILS**(x_{explo}).

De fait, lorsque $n_1.var$ (resp. $n_2.var$) est aussi présente dans G_2 (resp. G_1), l'algorithme temporise automatiquement l'exploration de G_1. L'algorithme ne reprend l'exploration de ce graphe que lorsque :

— ou bien $n_1.var \succ n_2.var$ (resp. $n_2.var \succ n_1.var$),
— ou bien $n_1.var = n_2.var$.

Dans le cas où $n_1.var = n_2.var$, l'exploration se fait simultanément sur les deux. Ceci garantit que les CONTEXTES des deux graphes auront la même instanciation de $n_1.var$ et seront donc compatibles. Dans l'autre situation, du fait de l'existence de l'ordre, il est certain que $n_1.var$ n'est pas présente dans le Graphe des Descendants issu de n_2 de G_2. L'algorithme peut alors instancier $n_1.var$ sans risquer de rendre incompatible les deux CONTEXTES.

Lorsque n_1 et n_2 sont tous deux des feuilles, le procédé d'exploration entraîne que les CONTEXTES sont alors compatibles. La combinaison de $n_1.val$ et $n_2.val$ peut donc être calculée car elle est pertinente.

19. Cette manière de procéder permet de nous assurer que l'ordre $\succ$ est aussi respecté dans G.

Algorithme 12 : EXPLORATION ORDONNÉE : Fonction récursive d'exploration de deux RGFs ordonnés similairement

Données : G_1 et G_2 deux RGFs ordonnés, n_1 nœud de G_1 couramment exploré, n_2 nœud de G_2 couramment exploré, une opération de combinaison $\odot$

```
1  début
2      si G1.EST TERMINAL(n1) and G2.EST TERMINAL(n2) alors
3          retourner G.INSÉRER NŒUD TERMINAL(n1.val ⊙ n2.val)
4      sinon
5          X_explo ← n1.var ;
6          si n2.var ≻ n1.var alors
7              X_explo ← n2.var ;
8          n_G ← G.INSÉRER NŒUD INTERNE(X_explo) ;
9          pour x_explo ∈ Dom{X_explo} faire
10             n'_1 ← n_1 ;
11             si n1.var = X_explo alors
12                 n'_1 ← n1.FILS(x_explo) ;
13             n'_2 ← n_2 ;
14             si n2.var = X_explo alors
15                 n'_2 ← n2.FILS(x_explo) ;
16             n_G.FILS(x_explo) ← EXPLORATION ORDONNÉE(G1, G2, n'_1, n'_2, ⊙)) ;
17         retourner n_G ;
```

Résultat : une référence vers le nœud n_G de G installé au cours de cet appel récursif

À la fin de chaque appel à l'Algorithme **EXPLORATION ORDONNÉE** (12), un nouveau nœud n_G est inséré dans G[20]. Ce nœud n_G contient alors une valeur si n_1 et n_2 sont tous deux terminaux. Dans le cas contraire, n_G est associé à la variable en cours d'exploration X_{explo}. Les enfants de n_G sont alors les nœuds résultants des explorations faites suivant les modalités de X_{explo}. De cette manière, et petit à petit, toutes les unions de CONTEXTES compatibles détectées durant les sous-appels récursifs sont reconstituées dans le graphe résultat, et sont associées aux résultats leur correspondants.

La présence de l'ordre commun sur les variables rend l'algorithme assez simple. Quand G_1 et G_2 sont tous deux des arbres ordonnés, chaque paire de nœuds ne sera visitée au plus qu'une fois. La complexité de cet algorithme est donc en $\mathcal{O}(|G_1|.|G_2|)$.

Pour les RGFORs (BDDs, ADDs or MDDs), si nous n'y prenons pas garde, cette complexité n'est pas respectée. En effet, rappelons que les RGFORs se distinguent des ARBRES DE DÉCISION par la fusion des sous-arbres isomorphes. Il en résulte qu'un

20. Bien entendu, des vérifications de redondance sont effectuées à chaque insertion de nœud dans G.

même sous-arbre dans ces structures peut être visité plusieurs fois au cours d'une combinaison (autant de fois qu'il existe de CONTEXTES y amenant). Dans les situations où c'est une paire de sous-arbres (un de G_1, un de G_2) qui est à nouveau visitée, cette seconde visite est redondante. Rappelons en effet que dans tout sous-arbre fusionné la fonction représentée est indépendante du chemin qui y amène. Une conséquence directe est que la combinaison des deux fonctions de deux sous-arbres fusionnés donnés est toujours la même, quelque soit le chemin qui nous a amené à ces sous-arbres. Une économie d'opérations est donc possible. Concrètement, cela veut dire que si une paire de nœuds, de $G_1 \times G_2$, a déjà été explorée, il n'est pas nécessaire de l'explorer à nouveau. Le mécanisme d'élagage consiste donc à vérifier si la paire de nœuds passée en argument lors de l'appel récursif n'a pas déjà été utilisée.

Recherche de *Politique* optimale et manipulation d'ADDs

À partir de cet algorithme de combinaison, il est possible de produire l'algorithme **VI** en ne manipulant que des RGFORs. L'Algorithme **SPUDD** (13) [Hoey et al., 1999] l'utilise donc pour effectuer la recherche d'une *Politique* optimale dans un FMDP représenté à l'aide d'ADDs. Puisqu'il est basé sur **VI**, l'algorithme **SPUDD**, comme l'algorithme **SVI**, repose sur l'application itérative de l'*Opérateur d'Optimalité de Bellman*.

Calcul de l'espérance sur les récompenses futures Le calcul de l'*espérance sur les gains futurs* diffère de la version avec arbre. À nouveau, un ADD sera calculé par action a. À nouveau, la distribution jointe sera uniquement calculée sur les variables présentes dans l'$ADD\,[\mathcal{V}]$ représentant l'estimation courante de la fonction *Valeur*. Toutefois cette distribution n'est pas calculée directement.

Dans un premier temps, l'$ADD\,[\mathcal{V}]$ est décalé temporellement : chaque variable X_i est remplacée par la variable correspondante à $t+1$ X'_i. La méthode **DÉCALER** effectue ce changement de variables nécessaire pour que l'$ADD\,[\mathcal{V}]$ représente la fonction *Valeur* sur les états futurs $(\mathcal{V}(X'_1, \ldots, X'_n))$.

Ensuite l'$ADD\,[\mathcal{V}]$ est itérativement combiné [21] avec chaque $ADD\,[P_{a,X_i}]$ représentant les probabilités de *Transition* de la variable X_i pour l'action a. Après chaque combinaison, l'élimination de X'_i par sommation sur son domaine est immédiatement effectuée. Pour éliminer cette variable, en chaque nœud associé à X'_i du graphe obtenu, la procédure suivante est effectuée :

— Extraire les Graphes des Descendants associés à chaque nœud fils,

21. L'opérateur de combinaison est la multiplication

Algorithme 13 : SPUDD : Stochastic Planning Using Decision Diagrams

Données : un FMDP représenté à l'aide d'ADDs

début
  // Initialisation
  $ADD[\mathcal{V}] \leftarrow \{\}$; **pour chaque** $a \in \mathbb{A}$ **faire**
    $ADD[\mathcal{V}] \leftarrow$**combinaison ordonnée**$(ADD[\mathcal{V}], ADD[\mathcal{R}_a], \max)$;
  $ADD[\mathcal{V}_{old}] \leftarrow \{\}$; // Itération de la valeur
  tant que $\exists$ *une feuille* $l \in$**combinaison ordonnée**$(ADD[\mathcal{V}], ADD[\mathcal{V}_{old}], -)$ *t.q.* $l.val \geq \epsilon$ **faire**
    $ADD[\mathcal{V}_{old}] \leftarrow ADD[\mathcal{V}]$;
    décaler$(ADD[\mathcal{V}])$;
    pour chaque $a \in \mathbb{A}$ **faire**
      pour chaque $X'_i \in$ SUPPORT$(\mathcal{V})$ **faire**
        $ADD[\mathcal{Q}_a] \leftarrow$**combinaison ordonnée**$(ADD[\mathcal{V}], ADD[P_{a,X'_i}], \times)$;
        pour $\forall$ *nœud* $n_{X'_i}$ *associé à* X'_i **faire**
          $Temp \leftarrow \{\}$;
          pour chaque $x_i \in \mathcal{D}om\{X'_i\}$ **faire**
            $Temp \leftarrow$**combinaison ordonnée**$(Temp, ADD[\mathcal{Q}_a]^{\downarrow n_{X'_i}.\textbf{fils}(x_i)}, +)$;
          Remplacer $ADD[\mathcal{Q}_a]^{\downarrow n_{X'_i}}$ par $Temp$;
      $ADD[\mathcal{Q}_a] \leftarrow$**combinaison ordonnée**$(ADD[\mathcal{Q}_a], ADD[\mathcal{R}_a], +)$;
      $ADD[\mathcal{V}] \leftarrow$**combinaison ordonnée**$(ADD[\mathcal{V}], ADD[\mathcal{Q}_a], \max)$;
  // Récupération gloutonne de la *Politique* ϵ-optimale
  $ADD[\pi] \leftarrow \{\}$;
  pour chaque $a \in \mathbb{A}$ **faire**
    $ADD[\pi] \leftarrow$**combinaison ordonnée**$(ADD[\pi], ADD[\mathcal{Q}_a], \text{argmax})$;
Résultat : une référence vers le nœud de G installé au cours de cet appel récursif

— Combiner ces graphes [22].
— Remplacer le nœud associé à X'_i (et son Graphe des Descendants) par le graphe obtenu.

Toute la procédure (Multiplication + Somme) est alors répétée sur l'ADD obtenu mais pour une autre des X'_i encore présente dans le diagramme.

Finalement, toute feuille de l'ADD obtenu est multiplié par γ avant que cet ADD soit sommé avec l'$ADD[\mathcal{R}_a]$ pour donner la nouvelle estimation de $ADD[\mathcal{Q}_a]$. Puis les ADDs représentant les fonctions *Valeur* d'action seront maximisés entre eux pour donner

22. A l'aide d'une somme cette fois-ci

le nouvel $ADD\,[\mathcal{V}]$. Ce processus est répété jusqu'à convergence. L'Algorithme **SPUDD** (13) donne les détails de l'implémentation.

3.3.6 Observations sur SPUDD

S'appuyant sur la librairie CUDD [Somenzi] qui manipule de manière très performante les ADDs, **SPUDD** parvient à résoudre efficacement des problèmes de planification à plusieurs millions d'états (les versions augmentées de FACTORY). Mais comme nous l'avons déjà vu en Partie I, les ADDs ont quelques travers qui posent problèmes.

D'une part ils n'utilisent que des variables binaires. Les variables multivaluées doivent être encodées à l'aide de variables binaires pour que la représentation du FMDP adjoint soit exploitable par **SPUDD**. Or l'utilisation de variables binaires va augmenter la taille du graphe. Et par répercussion la complexité de la rechercher de *Politique* optimale puisque celle-ci repose sur la taille des ADDs.

D'autre part, la librairie CUDD en elle-même conduit à quelques compromis coûteux. En effet, les feuilles des ADDs doivent être obligatoirement des valeurs. Il en résulte que pour l'ADD représentant la probabilité de *Transition* de la variable X_i, seule la transition vers la modalité "vrai" [23] est stockée. La probabilité de transition pour l'autre modalité doit être calculée avant toute exécution de **SPUDD**. Le problème est que ce calcul produit un diagramme qui n'est absolument pas optimal en taille, car il duplique la structure originelle pour en fournir la négation.

Enfin, la combinaison implique que les diagrammes soient ordonnés similairement. Ceci permet de s'assurer que les CONTEXTES aux feuilles soient bien compatibles. Or la taille d'un ADD dépend fortement de l'ordre sur les variables utilisées. L'algorithme de combinaison impose donc aux deux diagrammes opérés un ordre qui n'est pas nécessairement optimal pour chacun d'entre eux. Plus problématique, pour être efficace, CUDD ordonne tous les diagrammes stockés en mémoire similairement. Le compromis à trouver pour ordonner les diagrammes de façon à ce qu'ils soient le plus minimal possible est donc encore plus contraint. D'autant que rien ne garantit qu'un ordre efficace pour tous les ADDs liés aux distributions de probabilités soit efficace pour les ADDs liés aux récompenses ; les fonctions *Récompense* et *Transition* sont en effet fondamentalement différentes.

Devant toutes les contraintes et limitations imposées par CUDD, n'est-il pas possible de s'en affranchir pour trouver des algorithmes et représentations plus efficaces. Le chapitre suivant présente nos études sur le sujet.

23. Puisque X_i est binaire, elle est valuée sur $\{faux, vrai\}$

Chapitre 4

Planification et RGFORs

Dans ce chapitre, nous proposons un nouvel algorithme de combinaison de RGFORs qui n'impose pas que les deux graphes combinés soient ordonnés de manière similaire. Nous allons voir en détails quels sont les problèmes soulevés par cette suppression de contrainte et comment nous les avons résolus. Nous évoquons aussi la complexité de ce nouvel algorithme. Puis nous présentons les expériences menées pour valider cet algorithme ainsi que les résultats obtenus. L'algorithme proposé dans ce chapitre a fait l'objet d'un publication [Magnan and Wuillemin, 2013a].

4.1 Combinaison de RGFORs sans ordre commun

Soient G_1, G_2 et G des RGFORs (BDDs, ADDs or MDDs) tel que $G = G_1 \odot G_2$. Nous recherchons un algorithme qui produit G à partir de G_1 et G_2 sans imposer que les ordres [1] $\succ_1$ de G_1 et $\succ_2$ de G_2 soient compatibles.

Comme nous avons pu le voir au chapitre précédent, le parcours en profondeur simultané des deux graphes est une bonne stratégie pour énumérer efficacement les CONTEXTES compatibles. Aussi nous cherchons à l'appliquer ici aussi. Toutefois, quelques difficultés vont devoir être résolues.

4.1.1 Exploration en profondeur simultanée et Variables Rétrogrades

Nous nommons **EXPLORER** (14) la nouvelle fonction d'exploration récursive. L'appel initial est effectué par la méthode **COMBINER** avec les racines de G_1 et G_2 en arguments. Soient $n_1 \in G_1$ et $n_2 \in G_2$ les nœuds visités lors d'un quelconque appel

1. Par abus de langage, lorsque nous parlerons d'*ordre*, nous ferons en réalité allusion à *ordre complet*.

récursif à **EXPLORER** (14). Maintenant que la contrainte d'ordre global commun est relâchée, trois ordres doivent être pris en compte : $\succ_1$ pour G_1, $\succ_2$ pour G_2 et $\succ_G$ pour G[2]. A chaque appel récursif, les situations suivantes restent valables :

1. si n_1 et n_2 sont tous deux terminaux, alors $n_1.val \odot n_2.val$ est évalué ;
2. sinon
   a. si n_1 (resp. n_2) est terminal alors $X_{explo} = n_2.var$ (resp. $n_1.var$),
   b. si $n_1.var \succ_1 n_2.var$ et $n_1.var \succ_2 n_2.var$ alors $X_{explo} = n_1.var$,
   c. si $n_2.var \succ_1 n_1.var$ et $n_2.var \succ_2 n_1.var$ alors $X_{explo} = n_2.var$.

   Mais la situation suivante doit maintenant être elle aussi prise en compte :
   d. $n_1.var \succ_1 n_2.var$ alors que $n_2.var \succ_2 n_1.var$.

Aucune règle nous permet *a priori* de décider alors laquelle des variables $n_1.var$ et $n_2.var$ est à explorer en premier. Pour résoudre ce problème d'ordre d'exploration, nous allons introduire le concept de variable rétrograde :

Définition 4.1 (Variable Rétrograde)**.**

Soient $\succ_1$ et $\succ_2$ deux ordres sur un ensemble de variables $\mathbb{X}$. Une variable $X_r \in \mathbb{X}$ est dite rétrograde dans $\succ_2$ par rapport à $\succ_1$ si $\exists\ X_p \in X$ t.q. $X_r \succ_1 X_p$ et $X_p \succ_2 X_r$.

Corollaire 4.2. *Si X_r est rétrograde dans $\succ_2$ par rapport à $\succ_1$ à cause de X_p, alors X_p est rétrograde dans $\succ_1$ par rapport à $\succ_2$ à cause de X_r.*

Nous notons $\Re_{1,2}$ l'ensemble des variables rétrogrades $\{X_i \in \mathbb{X},\ X_i$ rétrograde dans $\succ_2$ par rapport à $\succ_1\}$. $\mathcal{D}om\{\Re_{1,2}\}$ est le domaine de cet ensemble de variables. La taille de ce domaine est $|\mathcal{D}om\{\Re_{1,2}\}| = \prod_{X_r \in \Re_{1,2}} |X_r|$[3].

Une première solution pour résoudre ce problème d'exploration est de toujours attribuer la priorité à la même RGFOR. Nous décidons donc pour l'instant que G_1 est toujours explorée en premier. Dans la situation où $n_1.var \succ_1 n_2.var$ alors que $n_2.var \succ_2 n_1.var$, $X_{explo} = n_1.var$. Des stratégies plus complexes seront évoquées plus tard. Nous allons maintenant voir quelles difficultés ce choix engendre.

4.1.2 Exploration en profondeur simultanée et Ordre de Construction

L'absence de contrainte d'ordre implique que $\succ_G$ n'est plus nécessairement ordonné comme $\succ_1$ ou $\succ_2$. Néanmoins, rappelons qu'à chaque fin d'appel récursif, un nœud est inséré dans G. Sa variable associée (s'il est interne) est la variable explorée au cours de

2. Nous allons voir par la suite que c'est à l'algorithme lui-même d'établir cet ordre
3. Notons que de manière générale $\Re_{1,2} \neq \Re_{2,1}$ et donc que $|\mathcal{D}om\{\Re_{1,2}\}| \neq |\mathcal{D}om\{\Re_{2,1}\}|$.

l'appel récursif. Cette manière de construire G entraîne que ses variables se succèdent dans un ordre bien déterminé : l'ordre dans lequel les variables sont explorées.

La méthode **COMBINER** établit donc l'ordre de $\succ_G$ de telle sorte qu'il soit compatible avec cet ordre d'exploration avant de lancer l'exploration. Puisque nous avons posé que G_1 a la priorité pour l'exploration, l'ordre $\succ_G$ a les propriétés suivantes :

— (i) $\succ_G$ étend $\succ_1$;

— (ii) $\succ_G$ étend $\succ_{2\setminus\Re_{1,2}}$.

Ces deux propriétés suffisent pour bâtir $\succ_G$ à partir de $\succ_1$ et $\succ_2$.

Remarquons que ces propriétés entraînent que $\Re_{G,1} = \emptyset$ et $\Re_{G,2} = \Re_{1,2}$. De fait, l'ensemble des variables rétrogrades à surveiller est $\Re_{G,2}$. Nous savons, par définition, que $\forall X_r \in \Re_{G,2}$, $\exists X_p \in \mathbb{X}$ qui vérifie que $X_r \succ_G X_p$ mais que $X_p \succ_2 X_r$. Nous devons donc nous attendre à rencontrer dans G_2 des nœuds n_p tels que $n_p.var = X_p$ et tels que $\exists n_r \in N_{G_2^{\downarrow n_p}}$ et $n_r.var = X_r$.

Lemme 4.3. *Lors de n'importe quel appel récursif à* **EXPLORER** *(14), la variable X_r doit être en cours d'exploration pour que toute exploration de la variable X_p puisse débuter.*

Démonstration. Chaque appel récursif est lié à l'exploration d'une variable X_{explo}. À la fin d'un appel récursif, un nœud résultat n^G_{explo} est inséré dans G. Ce nœud est associé à X_{explo}.

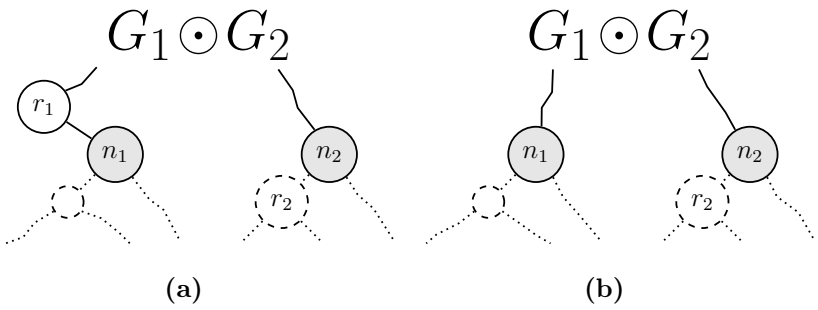
Supposons alors que $X_{explo} = X_p$ et que $\exists$ un nœud n_r dans les descendants de n_2 (nœud de G_2 pour l'appel courant) tel que $n_r.var = X_r$. L'appel récursif explorant alors X_r commencera après l'appel récursif explorant X_p mais se terminera avant.

Par construction, le nœud résultat n^G_r associé à X_r se situera parmi les descendants de n^G_{explo} associé à X_p. Nous aurons alors deux nœuds qui violent la contrainte d'ordre sur les variables puisque n^G_r est descendant de n^G_{explo} mais $n^G_r.var \succ_G n^G_{explo}.var$ ($X_r \succ_G X_p$).

Donc, pour vérifier que la contrainte $X_r \succ_G X_p$ dans G, toute exploration de X_r doit avoir commencé avant qu'une exploration de X_p débute. □

Notons qu'alors, dans G_2, un nœud associé à X_r sera rencontré. L'Algorithme **EXPLORER** (14) branche alors directement sur le fils lié à la modalité en cours d'exploration de X_r. Ceci a une incidence sur la complexité que nous analysons plus loin.

Ce lemme implique que l'Algorithme **EXPLORER** (14) doit anticiper une exploration sur X_r lorsqu'une exploration sur X_p est requise dans G_2. Une exploration de X_p en G_2 est requise si et seulement si $n_2.var = X_p$ et $n_2.var \succ_G n_1.var$. En dehors de ce cas, les explorations se font normalement. En particulier, X_r peut être rencontrée dans



(a) **(b)**

FIGURE 4.1 – *Dans le cas (a), aucune* exploration anticipée *n'est requise : X_r est en cours d'exploration sur r_1. Dans le cas (b), une* exploration anticipée *est requise : X_r n'a pas encore été explorée. Les nœuds sur lesquels se font l'exploration courante sont n_1 et n_2. $r_1.var = r_2.var = X_r$.*

G_1. Elle est alors normalement explorée, indifféremment de sa présence en un nœud de G_2.

Supposons maintenant qu'une exploration sur X_p soit nécessaire. Soient n_1 et n_2 les nœuds visités lors d'un appel à **EXPLORER** (14). Si le graphe des Descendants $G_2^{\downarrow n_2}$ contient un nœud n_r lié X_r, l'Algorithme **EXPLORER** (14) doit potentiellement anticiper une exploration de X_r. Au contraire, si $G_2^{\downarrow n_2}$ ne contient pas X_r, l'Algorithme **EXPLORER** (14) peut explorer normalement n_1 et n_2.

Si une exploration de X_p débute et qu'alors une exploration de X_r doit être anticipée, deux cas peuvent se produire :

a. *Si $X_r \in$ chemin exploré courant de G_1 alors X_r est déjà en cours d'exploration (cf. Figure 4.1a). Aucune* exploration anticipée *de X_r n'est alors requise.*

b. *Si $X_r \notin$ chemin exploré courant de G_1 alors X_r n'est pas en cours d'exploration (cf. Figure 4.1b). Une* exploration anticipée *de X_r est nécessaire avant d'explorer X_p dans G_2.*

Une *exploration anticipée* de X_r en un nœud n_2 revient à effectuer l'exploration normale d'un nœud n_r artificiellement inséré dans G_2 de telle manière que $n_r.var = X_r$ et que tous ses arcs sortants pointent sur n_2.

4.1.3 Élagage

Dans l'Algorithme **EXPLORATION ORDONNÉE** (12), le résultat de l'exploration d'une quelconque paire de nœuds donnée est toujours le même (cf. sous-sous-section 3.3.5). Les chemins empruntés depuis les racines des deux graphes pour arriver en ces nœuds n'influent pas sur cette exploration. Par conséquent, à chaque fin d'appel récursif à **EXPLORATION ORDONNÉE** (12), le nœud inséré dans G est conservé dans une table de hachage avec la paire (n_1, n_2) comme clé. L'opération d'élagage consiste

Algorithme 14 : EXPLORER : Fonction récursive d'exploration de deux RGFORs non ordonnées similairement

Données : G_1 et G_2 deux RGFs ordonnés, n_1 nœud de G_1 couramment exploré, n_2 nœud de G_2 couramment exploré, et $Inst_{\Re_{G,2}}$ la table indiquant l'instanciation courante des variables rétrogrades

début
  si G_1.**EST TERMINAL**(n_1) **and** G_2.**EST TERMINAL**(n_2) **alors**
    retourner G.**INSÉRER NŒUD TERMINAL**$(n_1.val \odot n_2.val)$;
  // Si la variable en n_2 est retrograde, il faut passer
  si $n_2.var \in \Re_{G,2}$ **alors**
    retourner EXPLORER$(G_1, G_2, n_1, n_2$.**FILS**$(Inst_{\Re_{G,2}}[n_2.var]))$
  // Selection variable à explorer
  $X_{explo} \leftarrow n_1.var$;
  si $n_2.var \succ n_1.var$ **alors**
    $X_{explo} \leftarrow n_2.var$;
  // Vérification qu'aucune exploration anticipée n'est à faire
  si $\exists n_r$ *t.q.* $n_r \in n_2.Descendants$ **and** $n_r.var \in \Re_{G,2}$ **and** $n_r.var \notin$ *chemin courant de* G_1 **alors**
    $X_{explo} \leftarrow n_r.var$;
  // Exploration
  $n_G \leftarrow G$.**INSÉRER NŒUD INTERNE**(X_{explo}) ;
  pour $x_{explo} \in Dom(X_{explo})$ **faire**
    $n'_1 \leftarrow n_1$;
    si $n_1.var = X_{explo}$ **alors**
      $n'_1 \leftarrow n_1$.**FILS**(x_{explo}) ;
    $n'_2 \leftarrow n_2$;
    si $n_2.var = X_{explo}$ **alors**
      $n'_2 \leftarrow n_2$.**FILS**(x_{explo}) ;
    n_G.**FILS**$(x_{explo}) \leftarrow$ **EXPLORER**$(G_1, G_2, n'_1, n'_2))$;
  retourner n_G ;

Résultat : une référence vers le nœud n_G de G installé au cours de cet appel récursif

alors à s'assurer que la paire courante de nœuds visités n'a pas déjà son entrée dans la table de hachage. Si tel est le cas, l'appel récursif se termine immédiatement par le retour du nœud associé. Ceci garantit une complexité en $\mathcal{O}(|G_1| \cdot |G_2|)$, puisque chaque paire de nœud n'est visitée qu'une et une seule fois au plus.

Dans l'Algorithme **EXPLORER** (14), l'élagage ne peut être effectué aussi efficacement. En effet, les visites consécutives d'une même paire de nœud n'aboutissent pas nécessairement au même résultat. Supposons pour la paire courante que (n_1, n_2), $n_2.var = X_p$ et que $X_r \in G_2^{\downarrow n_2}$. Avec ou sans exploration anticipée, X_r est instanciée

à une certaine valeur avant que l'exploration ne commence sur n_2. Comme évoqué ci-dessus, quand tout nœud lié à X_r est rencontré dans G_2, l'exploration branche automatiquement sur le nœud fils lié à la valeur courante de X_r. Une conséquence directe est qu'une partie seulement de $G_2^{\downarrow n_2}$ est alors explorée. L'Algorithme **EXPLORER** (14) doit alors explorer à nouveau $G_2^{\downarrow n_2}$ pour chaque valeur que X_r peut prendre. Cette exploration doit se faire depuis n_2 pour que l'ordre de construction des nœuds dans G respecte $\succ_G$.

Toutefois, toute nouvelle exploration du sous-graphe pour une même valeur donnée de X_r produit le même résultat. Un élagage peut donc être mis en place pour réduire la complexité. La clé utilisée dans la table de hachage doit néanmoins être modifiée. En plus de la paire de nœuds visités, la clé doit préciser pour quelle valeur des variables rétrogrades présentes dans le graphe des Descendants de n_2 la visite a été faite.

4.1.4 Complexité

Les multiples explorations dues aux variables rétrogrades affectent la complexité de l'Algorithme **EXPLORER** (14). Cet algorithme voit toujours sa complexité dépendre de la taille des deux représentations graphiques (comme l'Algorithme **EXPLORATION ORDONNÉE** (12)). Toutefois la complexité devient aussi dépendante de la taille du domaine des variables rétrogrades $\mathcal{D}om\{\Re_{1,2}\}$. Cette complexité est donc en $\mathcal{O}(|G_1| \cdot |G_2| \cdot |\mathcal{D}om\{\Re_{1,2}\}|)$.

Mais cette complexité *pire cas* doit être pondérée. Premièrement, l'Algorithme **EXPLORER** (14) ne ré-explore le sous-graphe d'un nœud n_2 que si nécessaire. Or cette complexité *pire cas* considère que ces ré-explorations sont nécessaires dès la racine du graphe G_2. Une borne supérieure plus précise de la complexité est difficile à établir ; elle demande une analyse topologique de G_1 et G_2.

Deuxièmement G_1 et G_2 ont maintenant leurs propres ordres qui peuvent être optimaux. Leurs tailles respectives sont donc plus petites que celles imposées par l'Algorithme **EXPLORATION ORDONNÉE** (12). La complexité *de base* n'est donc pas la même.

Troisièmement, $\succ_G$ est constructible de différentes manières. En effet, l'approche présentée (où $\succ_G$ est ordonné de sorte à ne laisser les variables rétrogrades apparaîtrent que dans G_2), n'est pas la seule. D'autres manières d'ordonnancer $\succ_G$ existent ; certaines d'entre elles peuvent améliorer significativement la taille de l'espace des variables rétrogrades, et donc la complexité de l'algorithme. Nous nous sommes intéressés à quelques-unes de ces approches que nous allons maintenant voir plus en détail.

Du choix de l'ordre pour $\succ_G$

Dans un premier temps, nous avons posé que $\succ_G$ étendait $\succ_1$. Or nous aurions pu tout autant décider que $\succ_G$ était construit de sorte à étendre $\succ_2$ au lieu de $\succ_1$; auquel cas, les variables rétrogrades apparaîtraient dans G_1. Ainsi, nous avons le choix de l'ensemble de variables rétrogrades à gérer, soit $\Re_{1,2}$ soit $\Re_{2,1}$. Or comme $|\mathcal{D}om\{\Re_{1,2}\}| \neq |\mathcal{D}om\{\Re_{2,1}\}|$, un compromis peut être trouvé ici. Nous nommons cette méthode ***Ordo1***.

Cette manière d'établir l'ordre impose toutefois de potentiellement choisir comme rétrogrades des variables qui ont un grand nombre de modalités. Rappelons alors qu'une variable X_r n'est rétrograde que s'il existe une autre variable X_p tel que l'une précède l'autre dans un ordre, et vice-versa dans l'autre. Cette autre variable peut être de plus petit domaine. Une question se pose alors : est-ce plus intéressant de prendre cette variable de plus petit domaine quand bien même cela fait apparaître une variable rétrograde dans l'autre graphe ? Une autre approche pour construire l'ordre final serait donc de choisir la variable de plus petit domaine lorsque deux variables sont en conflit. Cette méthode est identifiée comme la méthode ***Ordo2***.

Il est possible de peaufiner encore un peu plus ce choix. Rappelons que l'élagage se fait de sorte que si, pour une paire de nœuds donnée, il n'y pas de variables rétrogrades dans leurs descendants, l'élagage *de base*[4] marche. Les graphes des Descendants de ces paires de nœuds ne sont alors à explorer (et ne sont explorés) qu'une fois.

Ainsi, une nouvelle mesure pertinente est le nombre de nœuds à visiter plusieurs fois avant d'atteindre ces nœuds visitables qu'une fois. Les nœuds à revisiter sont les nœuds liés aux variables comprises entre les deux variables en conflit dans les deux ordres[5]. Déterminer exactement combien de nœuds sont à revisiter demande une analyse topologique. Toutefois, un majorant de ce nombre de visites à refaire est calculable assez simplement : en considérant que les RGFORs sont des arbres, le nombre d'explorations à refaire est égal au produit des domaines des variables comprises entre X_r et X_p incluses.

Soient alors $\mathbb{E}_r = \{X_i \in \mathbb{X}, X_r \succ_1 X_i \succ_1 X_p\}$ et $\mathbb{E}_p = \{X_i \in \mathbb{X}, X_p \succ_2 X_i \succ_2 X_r\}$. Le critère de décision est le suivant : si $|\mathbb{E}_r| \leq |\mathbb{E}_p|$, alors la variable X_r est explorée en premier (donc $X_r \succ_G X_p$). Dans le cas contraire, X_p aura précédence. Cette méthode est nommée ***Ordo3***.

Dans la section expérimentation, nous testons ces approches.

4. A savoir : la paire de nœud a-t-elle déjà été visitée ou non.
5. Tout $X_i \in \mathbb{X}$ t.q. $X_r \succ_1 X_i \succ_1 X_p$ ou $X_p \succ_2 X_i \succ_2 X_r$.

4.1.5 Intégration dans un algorithme de planification

Maintenant que nous disposons d'un algorithme effectuant la combinaison de deux RGFORs sans imposer que ces deux RGFORs soient ordonnées de la même manière, nous pouvons définir des algorithmes de recherche de *Politique* optimale qui exploitent cet algorithme.

SPUMDD : SVI avec des RGFORs multivaluées et l'Algorithme EXPLORER (14)

La version de l'algorithme **VI** adaptée aux RGFORs multivaluées est relativement simple à mettre en place. Comme l'algorithme **SPUDD** avec les ADDs, cette version doit gérer trois opérations de combinaison de graphes : les multiplications successives de la fonction *Valeur* estimée par les distributions de probabilités , l'ajout de la récompense, et les maximisations (ou recherche d'argument du maximum) des fonctions *Valeur* d'action. L'Algorithme **EXPLORER** (14) ci-dessus est utilisable (et est utilisé) pour effectuer ces combinaisons.

Une petite modification est toutefois apportée pour la combinaison par multiplication. En effet, cette combinaison intervient dans le calcul de l'espérance des récompenses sur les états futurs. L'élimination des variables à $t+1$ suit alors ces multiplications.

Dans la version **SPUDD**, cette élimination consiste à extraire, pour chaque nœud n_{suppr} lié à la variable X'_{suppr} à éliminer, les graphes des Descendants des nœuds fils, puis à sommer ces graphes entre eux. Le graphe résultat est alors inséré dans le graphe en lieu et place de $G^{\downarrow n_{suppr}}$. Outre qu'il faille éventuellement supprimer des nœuds dans la structure après ce retrait, cette approche implique aussi de faire plusieurs parcours en profondeur des graphes des Descendants, en particulier si plusieurs modalités de la variable éliminée mènent au même sous-graphe.

L'élimination des variables est réalisable plus tôt, lors de la multiplication en elle-même. Rappelons qu'il est possible d'éliminer une variable marquée à $t+1$ immédiatement après que le produit de la fonction *Valeur* par sa probabilité de *Transition* ait été calculé [6]. En ordonnant intelligemment les variables lors de la multiplication, il est possible de maintenir les variables marquées à $t+1$ à la fin [7] ; en particulier nous pouvons (et allons) maintenir la variable à supprimer en toute fin.

L'algorithme de combinaison est alors légèrement modifié : lors de l'appel récursif explorant la dernière variable (qui est donc à éliminer), l'algorithme procède toujours à la visite des nœuds fils [8]. Dans la version canonique de l'Algorithme **EXPLORER** (14),

6. L'hypothèse d'indépendance entre les variables à $t+1$ le permet.
7. Le graphe lié à la fonction *Valeur* n'est alors pas nécessairement ordonné optimalement.
8. Ces nœuds sont nécessairement terminaux

Algorithme 15 : EXPLORER ET ELIMINER : Fonction récursive d'exploration de deux RGFORs non ordonnées similairement avec suppression de variable.

Données : G_1 et G_2 deux RGFs ordonnés, n_1 nœud de G_1 couramment exploré, n_2 nœud de G_2 couramment exploré, X'_{suppr} la variable à supprimer

```
1  début
2      si (G1.EST TERMINAL(n1) or n1.var = X'_suppr) and
         (G2.EST TERMINAL(n2) or n2.var = X'_suppr) alors
3          cumul ← 0 ;
4          pour x_suppr ∈ Dom(X'_suppr) faire
5              n'1 ← n1 ;
6              si ! G1.EST TERMINAL(n1) alors
7                  n'1 ← n1.FILS(x_suppr) ;
8              n'2 ← n2 ;
9              si ! G2.EST TERMINAL(n2) alors
10                 n'2 ← n2.FILS(x_suppr) ;
11             cumul ⊕= n'1.val ⊙ n'2.val ;
12         retourner G.INSÉRER NŒUD TERMINAL(cumul);
           // Le reste de l'algorithme est similaire à l'Algorithme EXPLORER (14) à
              partir de la ligne 5
```

Résultat : une référence vers le nœud de G installé au cours de cet appel récursif

chacune de ces visites devrait alors se finir par l'insertion d'un nœud terminal dans le graphe résultat avec le résultat de la combinaison. Toutefois, dans cette situation-ci, le résultat est récupéré et cumulé avec les résultats des visites des autres nœuds fils. Ce cumul, une fois tous les fils visités, représente la somme obtenue après élimination de la variable et est donc inséré dans le graphe résultat en lieu et place d'un nœud interne associé à cette variable. L'Algorithme **EXPLORER ET ELIMINER** (15) donne le détail de cette nouvelle version appelée par **COMBINER ET ELIMINER**[9].

Ainsi, moyennant un ordonnancement pas nécessairement optimal, il est possible d'effectuer directement l'élimination des variables à $t+1$. Plusieurs parcours en profondeur sont alors évités, d'où un potentiel gain en temps. Dans la section expérimentation, nous testons cette nouvelle approche.

SPIMDD : SPI avec des RGFORs et l'Algorithme EXPLORER (14)

L'algorithme **PI** est aussi adaptable aux RGFORs. Toutefois, comme pour l'algorithme **SPI**, il s'agit plutôt d'une G**PI** car nous ne calculons pas exactement la fonction *Valeur* associée à la *Politique* courante à chaque itération. Nous recherchons plutôt

9. L'appel initial se fait avec les racines des deux graphes.

Algorithme 16 : SPUMDD : Planification stochastique manipulant des RGFORs multivaluées

Données : un FMDP représenté à l'aide de RGFORs multivaluées

début
  // Initialisation
  $RGFOR[\mathcal{V}] = \{\}$;
  pour chaque $a \in \mathbb{A}$ **faire**
    $RGFOR[\mathcal{V}] = \textbf{COMBINER}(RGFOR[\mathcal{V}], RGFOR[\mathcal{R}_a], \max)$;
  $RGFOR[\mathcal{V}_{old}] = \{\}$;
  // Itération de la valeur
  tant que $\exists$ *une feuille* $l \in \textbf{COMBINER}(RGFOR[\mathcal{V}], RGFOR[\mathcal{V}_{old}], -)$ *t.q.* $l.val \geq \epsilon$ **faire**
    $RGFOR[\mathcal{V}_{old}] = RGFOR[\mathcal{V}]$;
    DÉCALER$(RGFOR[\mathcal{V}])$;
    pour chaque $a \in \mathbb{A}$ **faire**
      pour chaque $X'_i \in \text{SUPPORT}(\mathcal{V})$ **faire**
        Ordonnancer $RGFOR[\mathcal{Q}_a]$ de sorte que X'_i soit à la dernière position dans $\succ_{RGFOR[\mathcal{Q}_a]}$;
        $RGFOR[\mathcal{Q}_a] = \textbf{COMBINER ET ELIMINER}(RGFOR[\mathcal{V}], RGFOR[P_{a,X'_i}], \times, X'_i)$;
      $RGFOR[\mathcal{Q}_a] = \textbf{COMBINER}(RGFOR[\mathcal{Q}_a], RGFOR[\mathcal{R}_a], +)$;
      $RGFOR[\mathcal{V}] = \textbf{COMBINER}(RGFOR[\mathcal{V}], RGFOR[\mathcal{Q}_a], \max)$;
  // Récupération gloutonne de la *Politique* ϵ-optimale
  $RGFOR[\pi] = \{\}$;
  pour chaque $a \in \mathbb{A}$ **faire**
    $RGFOR[\pi] = \textbf{COMBINER}(RGFOR[\pi], RGFOR[\mathcal{Q}_a], \text{argmax})$;

Résultat : une référence vers le nœud de G installé au cours de cet appel récursif

une estimation des valeurs prises par cette fonction *Valeur*. Puis nous améliorons la *Politique* à partir de cette estimation.

L'estimation de la fonction *Valeur* se calcule à l'aide de l'*opérateur de Bellman*. Ce qui implique que nous devons d'abord disposer des fonctions *Récompense* et *Transition* liées à la politique suivie et d'une estimation de départ de la fonction *Valeur* (donnée par les fonctions *Valeur* d'action dont on a déduit la *Politique*). L'opérateur $\triangleright$, vu dans la description de **SPI**, va à nouveau nous servir.

Pour la fonction *Récompense*, il s'agira simplement de combiner la RGFOR représentant la *Politique* courante avec les RGFORs des fonctions *Récompense* liées à chaque action par l'intermédiaire de cet opérateur. L'opérateur a alors pour tâche d'interrompre ou non une exploration. Ainsi, supposons, pour une action a donnée, un appel récursif avec en paramètre une feuille de la RGFOR liée à la politique optimale. Deux

Algorithme 17 : SPEMDD : Estimation de la Politique Structurée avec RGFORs multivaluées

Données : un FMDP où les fonctions *Transition* et *Récompense* sont factorisées à l'aide de RGFORs multivaluées, $|\mathbb{A}|$ RGFORs $RGFOR[\mathcal{Q}_a]$ représentant une estimation des fonctions *Valeur* d'action, et $RGFOR[\pi]$ la politique à estimer.

début
  // Initialisation
  $RGFOR[\mathcal{R}_\pi] = RGFOR[\pi]$;
  $RGFOR[\mathcal{V}] = RGFOR[\pi]$;
  pour chaque $X_i \in \mathbb{X}$ **faire**
    $RGFOR\left[P_{\pi,X'_i}\right] = RGFOR[\pi]$;
  pour chaque $a \in \mathbb{A}$ **faire**
    $RGFOR[\mathcal{R}_\pi] = \textbf{COMBINER}(RGFOR[\mathcal{R}], RGFOR[\mathcal{R}_a], \triangleright)$;
    $RGFOR[\mathcal{V}] = \textbf{COMBINER}(RGFOR[\mathcal{V}], RGFOR[\mathcal{Q}_a], \triangleright)$;
    pour chaque $X_i \in \mathbb{X}$ **faire**
      $RGFOR\left[P_{\pi,X'_i}\right] = \textbf{COMBINER}(RGFOR\left[P_{\pi,X'_i}\right], RGFOR[P_{a,X_i}], \triangleright)$;
  // Estimation de $\mathcal{V}_\pi$
  tant que *Critère non atteint* **faire**
    DÉCALER$(RGFOR[\mathcal{V}])$;
    pour chaque $X'_i \in \text{SUPPORT}(\mathcal{V})$ **faire**
      Ordonnancer $RGFOR[\mathcal{V}]$ de sorte que X'_i soit à la dernière position dans $\succ_{RGFOR[\mathcal{V}]}$;
      $RGFOR[\mathcal{V}] = \textbf{COMBINER ET ELIMINER}(RGFOR[\mathcal{V}], RGFOR[P_{\pi,X_i}], \times, X'_i)$;
    $RGFOR[\mathcal{V}] = \textbf{COMBINER}(RGFOR[\mathcal{V}], RGFOR[\mathcal{R}_\pi], +)$;

Résultat : $RGFOR[\mathcal{V}]$ l'estimation de la fonction *Valeur* pour la *Politique* π

cas sont à prendre en compte. Dans le premier, la feuille désigne comme action a. L'exploration se poursuit alors normalement. En particulier, lorsque la feuille du second graphe est atteinte, sa valeur est directement insérée dans le graphe résultat. Dans le second cas, la feuille désigne une autre action que a. L'appel récursif se termine alors prématurément. Toutefois, la feuille est dupliquée dans le diagramme résultat, pour une combinaison future avec le bon diagramme.

Le même procédé est appliqué pour le calcul des fonctions *Transition* et de la fonction *Valeur*. Pour chaque variable, la RGFOR représentant la *Politique* est combinée avec les RGFORs représentant les probabilités de *Transition* de la variable pour chaque action. Le résultat est alors la RGFOR représentant la fonction *Transition* de la variable pour la *Politique* courante. De même, la fonction *Valeur* est le résultat de

Algorithme 18 : SPIMDD : Itération de la Politique Structurée avec RGFORs multivaluées

Données : un FMDP où les fonctions *Transition* et *Récompense* sont factorisées à l'aide d'arbre

début
  // Calcul de la politique initiale
  $RGFOR[\pi] = \{\}$;
  pour chaque $a \in \mathbb{A}$ **faire**
    $RGFOR[\pi] = \textbf{COMBINER}(RGFOR[\pi], RGFOR[\mathcal{R}_a], \text{argmax})$;
    $RGFOR[\mathcal{Q}_a] = RGFOR[\mathcal{R}_a]$;
  $RGFOR[\pi_{old}] = \{\}$;
  // Itération de la Politique
  tant que $RGFOR[\pi] \neq RGFOR[\pi_{old}]$ **faire**
    $RGFOR[\pi_{old}] = RGFOR[\pi]$;
    // Estimation de $\mathcal{V}_\pi$
    $RGFOR[\mathcal{V}] = \textbf{SPEMDD}(all\ RGFOR[\mathcal{Q}_a], RGFOR[\pi])$;
    // Récupération gloutonne de la *Politique* ϵ-optimale
    $RGFOR[\pi] = \{\}$;
    pour chaque $a \in \mathbb{A}$ **faire**
      $RGFOR[\mathcal{Q}_a] = \textbf{QREGRESS}(RGFOR[\mathcal{V}], a)$;
      $RGFOR[\pi] = \textbf{COMBINER}(RGFOR[\pi], RGFOR[\mathcal{Q}_a], \text{argmax})$;

Résultat : $RGFOR[\pi]$ l'arbre représentant une *Politique* optimal pour le FMDP

la combinaison de la RGFOR représentant la *Politique* avec les RGFORs représentant les fonctions *Valeur* d'action qui nous ont permis d'établir cette *Politique*. L'Algorithme **SPEMDD** (17) donne les détails de la procédure.

La fonction *Valeur* associée à la politique courante se calcule alors comme se calculent les fonctions *Valeur* d'action dans **SPUMDD**. Bien que relativement simple à mettre en place, cet algorithme n'a pas été implémenté et donc testé. L'Algorithme **SPIMDD** (18) donne les détails de la procédure.

4.2 Expériences et Observations

L'évaluation de l'Algorithme **EXPLORER** (14) s'est faite en deux étapes. Dans un premier temps nous avons observé ses performances pures, en le comparant à l'Algorithme **EXPLORATION ORDONNÉE** (12) sur des combinaisons de RGFORs générées aléatoirement à l'aide de l'algorithme vu au Chapitre 2. Puis nous nous sommes intéressés à son apport dans l'algorithme de planification **SPUMDD**.

Nous avons implémenté **SPUMDD** (et donc **EXPLORER**) en utilisant la librairie C++ aGrUM developpée au LIP6 [10]. Puisque l'implémentation standard de **SPUDD** exploite la librairie CUDD très bien optimisée pour les ADD (mais pas pour les RGFORs multivaluées), nous avons codé notre propre version de **SPUDD** (et donc **EXPLORATION ORDONNÉE**) dans le but de rester comparable d'un point de vue temps d'exécution. Cette version s'appuie donc sur les mêmes structures de données que l'algorithme **SPUMDD**.

4.2.1 Validation de EXPLORER

Dans la première étape, nous nous intéressons à ses performances de manière générale et non dans le contexte de la planification. Ainsi, nous avons repris l'algorithme de génération de RGFORs vu en Chapitre 2, et nous avons généré des paires de RGFORs que nous avons combinées avec les Algorithmes **EXPLORER** (14) et **EXPLORATION ORDONNÉE** (12).

Méthodologie

Comme pour les expériences du Chapitre 2, deux méta-paramètres sont utilisés pour contrôler les expérimentations, à savoir : le nombre de modalités maximum par variable, qui module la largeur des RGFORs générées, et le nombre maximal de variables par RGFOR, qui conditionne sa profondeur. Ainsi plus ses deux grandeurs augmentent, plus la complexité théorique des algorithmes évalués augmente. Pour chaque couple de méta-paramètres, mille paires d'RGFORs ont été générées.

Nous avons comparé les deux algorithmes en nombre d'appels récursifs et en taille des diagrammes. Le nombre d'appels récursifs nous indique directement la complexité pratique de la combinaison des deux RGFORs manipulées. La taille nous permet de comparer le gain obtenu en ordonnant individuellement chaque RGFOR par rapport à la taille nécessaire pour ordonner les deux RGFORs similairement. À chaque combinaison sont donc récupérés les nombres d'appels récursifs et les tailles.

Pour chaque paire d'RGFORs générées en accord avec les valeurs actuelles des deux méta-paramètres, les combinaisons sont effectuées en trois phases. Dans la première phase, les deux RGFORs sont minimisées individuellement : nous cherchons à minimiser leurs tailles sans les contraindre à être ordonnées similairement. Puis elles sont combinées avec l'Algorithme **EXPLORER** (14). Trois combinaisons sont alors effectuées afin de tester les trois méthodes d'ordonnancement : ***Ordo1***, ***Ordo2*** et ***Ordo3***.

10. Laboratoire d'Informatique de Paris VI

Comparaison en nombre d'appels récursifs (**EXPLORATION ORDONNÉE** + ADDs vs **EXPLORER**)				
NbVar	NbModa	***Ordo1***	***Ordo2***	***Ordo3***
5	3	140.43% 117.07% – 164.11%	141.2% 119.15% – 206.34%	126.83% 106.87% – 147.09%
5	4	85.25% 65.88% – 113.19%	88.95% 67.53% – 117.92%	69.89% 55.71% – 87.81%
5	5	89.31% 70.27% – 119.67%	92.92% 72.52% – 122.31%	80.46% 64.56% – 100.95%
5	6	76.18% 58.95% – 104.72%	78.75% 59.98% – 106.71%	68.32% 53.87% – 87.24%
5	7	70.48% 54.24% – 97.26%	73.08% 55.81% – 99.78%	65.25% 51.76% – 83.88%
10	3	118.05% 103.35% – 150.99%	170.94% 129.5% – 239.83%	107.07% 100.0% – 125.68%
10	4	69.08% 56.91% – 91.13%	106.07% 70.22% – 159.41%	62.98% 52.25% – 79.23%
10	5	72.69% 60.21% – 95.53%	108.73% 70.55% – 177.29%	70.45% 58.14% – 86.88%
10	6	64.21% 51.7% – 93.78%	93.85% 61.56% – 178.13%	59.43% 48.6% – 77.85%
10	7	64.24% 49.15% – 94.73%	96.39% 58.64% – 188.17%	58.51% 46.77% – 74.6%
15	3	109.69% 100.0% – 125.92%	182.08% 134.15% – 276.54%	110.56% 100.0% – 124.2%
15	4	64.33% 53.84% – 79.68%	115.04% 73.02% – 193.31%	64.53% 53.22% – 77.3%
15	5	68.04% 57.59% – 92.06%	133.33% 77.66% – 242.4%	67.45% 56.25% – 82.6%
15	6	60.86% 49.1% – 81.65%	126.84% 66.05% – 293.6%	57.32% 48.1% – 72.48%
15	7	74.28% 51.79% – 100.0%	-	55.71% 45.2% – 72.47%

TABLE 4.1 – *Comparaison en nombre d'appels récursifs entre les algorithmes* **EXPLORATION ORDONNÉE** + ADD*s et* **EXPLORER**. *Le tableau montre le rapport (***EXPLORER***)/(***EXPLORATION ORDONNÉE** + ADD*s), plus précisément la médiane, et à côté, en plus petit, les premiers et derniers quartiles.*

La seconde phase consiste à les minimiser de manière globale. Les deux RGFORs sont donc dans un premier temps ordonnées similairement, puis l'Algorithme **TAMISAGE** (2) est appliqué. Toutefois, celui-ci a été légèrement modifié de sorte à opérer sur les deux diagrammes : les variables sont déplacées simultanément sur les deux RGFORs, et la somme des tailles des deux RGFORs est utilisée comme critère de décision. Les deux diagrammes sont ensuite à nouveaux combinés par l'Algorithme **EXPLORATION ORDONNÉE** (12). Cette combinaison se fait donc entre RGFORs multivaluées ordonnées similairement. Nous effectuons cette combinaison pour voir les performances de l'Algorithme **EXPLORATION ORDONNÉE** (12) sur des RGFORs multivaluées.

Enfin, la dernière phase consiste à transformer les RGFORs en ADDs. Les variables sont alors binarisées, puis les graphes modifiés en conséquence comme vu au Chapitre 2. Les ADDs obtenus sont à nouveau minimisés de manière globale puis combinés par l'Algorithme **EXPLORATION ORDONNÉE** (12).

Résultats

Comparativement à l'algorithme de combinaison d'ADDs originel (**EXPLORATION ORDONNÉE** + ADDs), notre nouvel algorithme **EXPLORATION ORDONNÉE** montre une certaine supériorité (cf. Table 4.1). Pour les méthodes d'ordonnancement ***Ordo1*** et

Comparaison en nombre d'appels récursifs (EXPLORATION ORDONNÉE vs EXPLORER)				
NbVar	NbModa	***Ordo1***	***Ordo2***	***Ordo3***
5	3	140.43% 117.07% – 164.11%	141.2% 119.15% – 206.34%	126.83% 106.87% – 147.09%
5	4	141.52% 115.37% – 178.47%	149.56% 122.22% – 189.49%	125.42% 107.88% – 152.24%
5	5	142.85% 115.06% – 182.95%	148.23% 121.72% – 190.2%	136.99% 112.82% – 165.45%
5	6	147.37% 116.5% – 187.75%	151.18% 120.37% – 192.29%	135.99% 112.16% – 164.07%
5	7	141.14% 114.94% – 183.64%	150.28% 118.62% – 194.67%	134.45% 110.93% – 162.62%
10	3	118.05% 103.35% – 151.47%	170.94% 129.5% – 239.83%	107.07% 100.0% – 125.68%
10	4	121.06% 102.38% – 153.06%	175.47% 125.76% – 259.09%	111.37% 100.0% – 134.15%
10	5	118.04% 101.27% – 151.71%	176.72% 117.76% – 286.73%	110.68% 98.98% – 134.73%
10	6	117.15% 100.0% – 157.85%	181.62% 105.88% – 316.42%	111.92% 98.29% – 140.93%
10	7	119.25% 98.33% – 172.63%	172.84% 100.82% – 343.4%	110.04% 91.61% – 142.21%
15	3	109.56% 100.0% – 125.92%	182.08% 134.15% – 276.54%	110.56% 100.0% – 124.2%
15	4	109.67% 100.0% – 135.61%	200.79% 131.76% – 322.13%	109.11% 100.0% – 129.43%
15	5	110.7% 100.0% – 140.95%	205.56% 118.08% – 375.34%	108.03% 98.62% – 130.48%
15	6	109.48% 97.33% – 143.74%	220.77% 110.99% – 495.39%	104.63% 91.4% – 126.13%
15	7	100.0% 100.0% – 130.28%	-	103.36% 82.67% – 128.38%

TABLE 4.2 – *Comparaison en nombre d'appels récursifs (***EXPLORATION ORDONNÉE** *vs* **EXPLORER***). Le tableau montre le rapport (***EXPLORER***)/(***EXPLORATION ORDONNÉE***), plus précisément la médiane, et à côté, en plus petit, les premier et dernier quartiles.*

Ordo3, notre algorithme demande globalement moins d'appels récursifs (jusqu'à moitié moins d'appels). Seule la méthode ***Ordo2*** s'avère particulièrement inefficace (du double au triple d'appels requis). Notons par ailleurs que plus le nombre de variables augmente, plus notre algorithme s'avère efficace. Les seuls cas où l'algorithme de combinaison d'ADDs est plus efficace sont les situations où le nombre maximal de modalité par variable est minimal.

Toutefois, les RGFORs multivaluées manipulées par notre algorithme sont alors quasiment similaires à leur version binaire. Dans cette configuration, comparer une combinaison de RGFORs multivaluées sans contrainteset une combinaison d'ADDs revient à comparer une combinaison de RGFORs multivaluées sans contrainteset une combinaison de RGFORs multivaluées avec contrainte. Or, comme le montre la Table 4.2, l'efficacité de la combinaison sans contraintesde RGFORs multivaluées est plus nuancée lorsqu'on la compare à une combinaison avec contraintes de RGFORs multivaluées.

Ainsi, pour les combinaisons employant la technique d'ordonnancement ***Ordo1*** par exemple, bien que leur efficacité augmente avec le nombre maximal de variables, ces combinaisons restent au mieux 10% moins efficaces. La technique d'ordonnancement ***Ordo2*** s'avère totalement inefficace : elle conduit à des combinaisons demandant plus du double d'appels récursifs, voire au quadruple. De plus, le nombre d'appels récursifs pour les combinaisons reposant sur cette technique ne fait qu'augmenter avec le nombre

maximal de variables. Finalement, seule la technique d'ordonnancement ***Ordo3*** tire son épingle du jeu. Même si elle implique plus d'appels récursifs, sa médiane est à peine 10% au-dessus de la médiane de la combinaison avec contrainte. De plus, le premier quartile passe sous la barre des 100%, indiquant que 25% des combinaisons sans contraintes effectuées ont eu une complexité inférieure aux combinaisons avec contraintes.

Comparaison en taille des RGFORs combinées			
NbVar	NbModa	**EXPLORER** vs **EXPLORATION ORDONNÉE** + ADDs	**EXPLORER** vs **EXPLORATION ORDONNÉE**
5	3	88.89% 80.77% – 93.75%	95.65% 86.36% – 100.0%
5	4	59.26% 48.39% – 69.57%	94.12% 80.95% – 100.0%
5	5	55.0% 41.94% – 63.89%	92.0% 75.93% – 100.0%
5	6	43.2% 30.97% – 52.78%	90.32% 70.97% – 100.0%
5	7	37.83% 25.94% – 48.89%	89.39% 68.18% – 100.0%
10	3	88.46% 75.0% – 93.94%	93.1% 78.57% – 100.0%
10	4	54.19% 41.07% – 64.0%	86.36% 68.57% – 100.0%
10	5	47.62% 33.33% – 58.21%	82.86% 60.7% – 97.31%
10	6	35.87% 23.05% – 47.37%	77.98% 52.11% – 95.56%
10	7	29.35% 17.23% – 41.76%	72.41% 43.68% – 93.33%
15	3	89.66% 77.78% – 95.0%	94.44% 81.25% – 100.0%
15	4	55.41% 42.37% – 64.52%	88.89% 70.97% – 100.0%
15	5	48.49% 34.56% – 57.89%	84.75% 61.84% – 100.0%
15	6	36.63% 23.5% – 47.37%	78.57% 50.9% – 96.55%
15	7	39.44% 23.69% – 60.24%	90.12% 55.61% – 100.0%

TABLE 4.3 – *Comparaison en taille entre d'une part les combinaisons de* RGFORs *multivaluées sans contrainteset d'autre part les combinaisons d'*ADDs *(donc contraintes) et combinaisons de* RGFORs *multivaluées contraintes. Le tableau montre les rapports (*RGFORs *multivaluées sans contrainte)/(*ADD*) et (*RGFORs *multivaluées sans contrainte)/(*RGFORs *multivaluées avec contrainte), plus précisément la médiane, et à côté, en plus petit, les premier et dernier quartiles.*

Cette légère perte en temps de calcul est toutefois compensée par un gain en mémoire (cf Table 4.3). D'une part, la combinaison de RGFORs multivaluées sans contraintespar rapport à une combinaison d'ADDs permet de manipuler des diagrammes jusqu'à 5 fois plus petits. D'autre part, même comparativement aux combinaisons avec contrainte, les combinaisons sans contraintes manipulent des diagrammes 20% plus petits en moyenne.

En conclusion, tout d'abord, des trois techniques d'ordonnancement, la méthode ***Ordo3*** s'avère la plus intéressante : sa complexité empirique est quasi semblable à la complexité des combinaisons avec contraintes. Aussi nous allons privilégier celle-ci pour la planification. La comparaison en taille des diagrammes combinés montre que

l'Algorithme **EXPLORER** (14) permet réellement un gain en mémoire. Ce gain sera à notre avantage lors de l'utilisation de cet algorithme pour la planification. En effet, moins de mémoire consommée implique des opérations de manipulation en mémoire plus courtes. Ce temps supplémentaire gagné compense alors le temps perdu en calcul supplémentaire. Comme nous allons le voir, cet avantage permet à **SPUMDD** d'être plus efficace. Notons enfin qu'une représentation multivaluée est définitivement plus intéressante en termes de complexité de calcul qu'une représentation binaire. En effet, que ce soit avec ou sans contraintes, la combinaison d'RGFORs multivaluées est jusqu'à 40% moins complexe que la combinaison d'ADDs.

4.2.2 Validation de SPUMDD

L'algorithme de combinaison **EXPLORER** que nous proposons ayant des performances comparables à celles de l'algorithme de combinaison **EXPLORATION ORDONNÉE** utilisé par **SPUDD**, nous nous intéressons donc maintenant à son utilisation dans le cadre d'une planification à l'aide de **SPUMDD**.

Algorithmes évalués

SPUMDD est donc comparé à l'état de l'art **SPUDD**. Une version multivaluée de **SPUDD** a aussi été implémentée afin de tester l'efficacité de l'Algorithme **EXPLORER** (14) en comparaison de l'Algorithme **EXPLORATION ORDONNÉE** (12). Elle sera notée **SPUDD-multi**. Nous avons aussi implémenté les différentes versions du calcul de l'espérance dans **SPUMDD** : combinaison puis élimination contre combinaison et élimination (soit la version **SPUMDD**). La version combinaison puis élimination sera notée **SPUMDD-proj**. Au total, nous avons donc 4 algorithmes testés. Pour les algorithmes exploitant l'Algorithme **EXPLORER** (14), la technique d'ordonnancement ***Ordo3*** est utilisée puisqu'elle offre les meilleurs résultats.

Ces algorithmes se basant sur **VI** pour la planification, il nous faut fixer un ϵ comme critère d'arrêt. Nous avons fixé comme valeur $\epsilon = 0,01$.

Grandeurs mesurées

Dans ces expérimentations, nous nous intéressons à la taille des graphes calculés. En effet, la complexité des opérations de combinaison dépend de la taille des graphes. La taille de la fonction *Valeur* est particulièrement importante vue qu'elle intervient dans tous les calculs de chaque itération.

Temps de Planification (en s)				
Problème	**SPUMDD**	**SPUMDD-proj**	**SPUDD-multi**	**SPUDD**
COFFEE ROBOT	**0,7**	1,4	**0,7**	**0,7**
Tiny - FACTORY	**0,45**	1	**0,45**	**0,45**
FACTORY	105	**103**	249	771
FACTORY 0	152	**118**	295	1 038
FACTORY 1/2	543	**213**	835	3 160
FACTORY 3	2 712	**903**	3 303	10 696
Taxi	**13**	21	74	151

TABLE 4.4 – *Résultats en temps de planification obtenus sur* **SPUDD**, **SPUMDD-*proj***, **SPUDD-*multi*** *et* **SPUMDD** *pour les problèmes classiques. Le temps indiqué est en seconde.* FACTORY *1 et 2 sont affichées ensemble car elles requièrent le même temps moyen de calcul. Rappelons que ces deux problèmes ne diffèrent que sur une variable qui ne change rien à la planification.*

Nous nous sommes aussi intéressés au nombre d'appels récursifs dans les algorithmes Algorithme **EXPLORER** (14) et Algorithme **EXPLORATION ORDONNÉE** (12). Nous voulons en effet savoir quel algorithme est le moins coûteux en calcul au total.

Enfin nous nous sommes intéressés à la durée totale pour effectuer une planification. En effet, en plus des algorithmes de combinaison, d'autres tâches sont consommatrices de temps : les minimisations de graphes, les destructions de graphes, etc. Nous cherchons à savoir quel algorithme est le plus efficace pour planifier. Pour obtenir un temps moyen, 25 planifications de chaque problème sont effectuées puis la moyenne est calculée.

Minimisation

En dehors de la phase de calcul de l'espérance, pour les deux algorithmes, les RGFORs sont minimisées régulièrement. Pour **SPUMDD**, dès qu'une nouvelle RGFOR est produite en résultat d'une combinaison, nous cherchons à la minimiser. Pour **SPUDD**, dès que deux diagrammes sont combinés, ils sont d'abord minimisés en utilisant la version multigraphe de l'algorithme de minimisation.

Résultats

En ouverture de cette analyse, il est important de noter qu'en terme de nombre d'itérations, les 4 algorithmes se comportent similairement : ils mettent tous le même nombre d'itérations à atteindre le critère d'arrêt, pour chaque problème.

Nombre d'appels récursifs des algorithmes de combinaison				
Problème	**SPUMDD**	**SPUMDD-proj**	**SPUDD-multi**	**SPUDD**
COFFEE ROBOT	**135 291**	269 033	255 572	255 572
Tiny - FACTORY	**102 560**	107 462	103 413	112 976
FACTORY	32 877 254	**24 311 965**	47 768 179	91 810 798
FACTORY 0	48 107 501	**25 883 058**	89 726 814	116 740 264
FACTORY 1/2	101 858 451	**49 362 517**	103 261 545	199 972 440
FACTORY 3	246 426 709	**82 327 149**	293 341 010	485 106 756
Taxi	**11 710 665**	27 086 389	33 321 119	17 354 486

TABLE 4.5 – *Résultats en nombre d'appels récursifs obtenus sur* **SPUDD**, **SPUMDD-*proj***, **SPUDD-*multi*** *et* **SPUMDD** *pour les problèmes classiques.* FACTORY *1 et 2 sont affichées ensemble car elles requièrent le même temps moyen de calcul. Rappelons que ces deux problèmes ne diffèrent que sur une variable qui ne change rien à la planification.*

En termes de temps (cf Table 4.4), les représentations multivaluées s'avèrent toutes plus efficaces pour la planification que **SPUDD**. Hormis le problème COFFEE ROBOT, binaire par nature et donc avantageux pour **SPUDD**, les algorithmes **SPUMDD**, **SPUMDD-proj** et **SPUDD-multi** planifient la plupart des problèmes en moins de temps que **SPUMDD**. Ce constat est d'autant plus vrai que la différence entre la taille du domaine multivarié et la taille du domaine binarisé est importante. Ainsi **SPUMDD-proj** est dix fois plus efficace à résoudre TAXI ou FACTORY 3 que **SPUDD**.

Une analyse plus focalisée sur les trois algorithmes utilisant des représentations multivaluées montrent que les algorithmes utilisant des combinaisons sans contraintes sont globalement plus efficaces. Seuls les petits problèmes COFFEE ROBOT et Tiny-FACTORY sont à l'avantge de **SPUDD-multi**.

En revanche, il est difficile de départager **SPUMDD-proj** de **SPUMDD**. En effet, autant **SPUMDD** est plus performant sur les problèmes COFFEE ROBOT, Tiny-FACTORY et TAXI, autant **SPUMDD-proj** est beaucoup plus efficace sur les autres problèmes FACTORY.

Une analyse en nombre d'appels récursifs (cf. Table 4.5) montre que pour les problèmes FACTORY, **SPUMDD-proj** est bien plus efficace. Le contre-coup d'une organisation des variables de telle sorte que la variable à éliminer soit en bas dans **SPUMDD** explique pourquoi il est plus long que **SPUMDD-proj** : Cette réorganisation implique une taille plus grande, donc une complexité de combinaison plus grande.

Une étude attentive de la Table 4.5 montre que pour certains problèmes, **SPUMDD** et **SPUDD-multi** ont un même nombre total d'appels récursifs. Or l'analyse en temps montre qu'à chaque fois, **SPUMDD** est plus rapide. L'explication vient de la phase de

Comparaison en taille de la fonction *Valeur* multivaluée et binaire		
Problème	Multivalué	Binaire
COFFEE ROBOT	**24**	**24**
Tiny - FACTORY	**12**	17
FACTORY	**377**	814
FACTORY 0	**384**	864
FACTORY 1/2	**736**	2088
FACTORY 3	**814**	1907
Taxi	**178**	845

TABLE 4.6 – *Comparaison en taille de la fonction* Valeur *avec une représentation binaire et une représentation multivaluée.*

minimisation, où **SPUMDD-proj** et **SPUMDD** sont plus efficaces que **SPUDD-multi** et **SPUDD**. Réorganiser des paires de diagrammes est beaucoup plus coûteux que réorganiser un seul diagramme. L'exemple TAXI en est particulièrement symptomatique : **SPUMDD** et **SPUDD** sont les plus efficaces en terme d'appels récursifs, pourtant **SPUDD** s'avère finalement dix fois plus long à converger. Cela ne peut pas être uniquement dû au calcul de l'espérance, qui se fait par combinaison puis élimination, car **SPUMDD-proj** et **SPUDD-multi** eux aussi reposent sur cette approche. Or il apparaît aussi que **SPUDD-multi** est beaucoup plus lent que **SPUMDD-proj** alors que ces deux algorithmes ne diffèrent que sur la manière d'ordonnancer les RGFORs. Il est donc évident qu'ordonner similairement deux graphes est beaucoup plus coûteux qu'en minimiser un, d'où un avantage supplémentaire pour **SPUMDD**.

En résumé, ces expériences montrent clairement l'avantage à utiliser un algorithme de combinaison sans contraintespour la planification. Que ce soit en temps de planification ou nombre d'appels récursifs, de tels algorithmes requièrent bien moins d'appels récursifs, et les temps de minimisation pour accélérer les calculs sont bien plus courts.

Enfin, d'un point de vue consommation mémoire, comme le montre la Table 4.6, les représentations multivaluées sont plus intéressantes. En effet, en dehors de COFFEE ROBOT qui est un cas purement binaire, les représentations multivaluées manipulent une fonction *Valeur* beaucoup plus petite que sa version binaire.

4.3 Conclusion

Dans cette partie, nous avons vu quelles méthodes étaient mise en œuvre pour résoudre les problèmes de décision dans l'incertain. Ainsi nous avons défini quelle forme

prenait la solution de tels problèmes : une *Politique* optimale qui à chaque état lui associe la (ou les) meilleure(s) action(s) à prendre. Nous avons aussi introduit la fonction *Valeur* qui permet de comparer deux *Politiques*. Cette fonction *Valeur* sert de guide dans les deux algorithmes principaux de recherche de *Politique* optimale : **VI** et **PI**.

Bien entendu, cette recherche d'une *Politique* optimale souffre d'un problème de complexité lors du passage à l'échelle. La recherche de *Politique* optimale à l'aide de **VI** ou **PI** devient non envisageable. Toutefois, comme nous avons pu le voir, les représentations factorisées vues dans la Partie I permettent d'énumérer plus facilement l'espace d'états, qui est la principale difficulté soulevée par le passage à l'échelle.

Ainsi, **SVI** et **SPI**, puis **SPUDD**, se sont avérés capables de trouver la *Politique* optimale de problèmes toujours plus grands. Toutefois l'algorithme le plus performant des trois, à savoir **SPUDD**, impose une contrainte importante sur la modélisation du problème : toute variable utilisée doit être binaire. Cette contrainte pose problème lorsque le système se modélise à l'aide de variables multimodales, augmentant ainsi la complexité et du modèle et des calculs. Les RGFORs multivaluées que nous avons présentée en Partie I résolvent ce problème. Plus spécifiquement, il est possible de trouver la *Politique* optimale à l'aide de ces représentations en utilisant les mêmes algorithmes que pour **SPUDD**.

Toutefois, il nous est apparu qu'un des sous-algorithmes utilisés par **SPUDD** posait problème : l'algorithme de combinaison d'ADDs (et donc de RGFORs multivaluées). Pièce centrale de l'algorithme **SPUDD**, il impose que les ordres sur les variables soient identiques (ou du moins compatibles) pour les deux graphes combinés. Or, l'ordre sur les variables est un des problèmes majeurs des RGFORs en général (cf. Partie I) car il conditionne la taille de ces RGFORs. Et la complexité de l'algorithme de combinaison est dépendante directement de la taille des deux graphes combinés.

Pour cette raison, nous avons recherché, et proposons, un nouvel algorithme de combinaison de RGFORs qui n'impose plus que les graphes soient orientés de la même manière. Ainsi, les deux RGFORs combinées peuvent être ordonnées différemment et avoir donc leur taille la plus minimale possible lors de la combinaison. Toutefois, nous avons aussi vu que le retrait de cette contrainte se fait au prix d'une augmentation de la complexité de la combinaison. Bien que notre nouvel algorithme semble légèrement moins efficace du fait de cette augmentation de complexité, son intégration dans des algorithmes de planification montre qu'il s'avère plus avantageux finalement. Sur les problèmes testés, la combinaison sans contraintess'avère en effet être d'une complexité moindre, manipuler des modèles de tailles moindres (et donc avoir un temps de gestion mémoire moindre), et être plus efficace à rechercher la taille minimale des RGFORs

manipulées.

Nous disposons maintenant de deux outils qui vont grandement nous aider dans la dernière partie de cette thèse. Notre prochain objectif est en effet de pouvoir apprendre les FMDPs à partir d'expérience, et de planifier sur ces modèles appris. Les RGFORs multivaluées d'une part vont faciliter l'apprentissage des modèles en nous évitant de devoir nous poser des questions sur la binarisation du modèle. L'algorithme **SPUMDD** réduira lui les temps de planification, permettant à l'algorithme principal de se concentrer plus efficacement sur les tâches d'exploration et d'apprentissage.

Troisième partie

Apprentissage Par Renforcement et Représentations Factorisées

Chapitre 5

Apprentissage par Renforcement

Jusqu'à présent, le modèle utilisé pour la planification était supposé connu. Les fonctions de *Transition* et de *Récompense* étaient parfaitement définies. Les variables caractérisant le système étaient correctement déterminées. Les SUPPORTS des fonctions *Transition* et *Récompense* étaient identifiés, et les représentations structurées bien établies. Toutefois, ce cadre idéal est difficilement accessible aux problèmes réels.

De nombreux problèmes ne sont généralement pas aussi simplement modélisables. D'une part, la nature même du système étudié rend les fonctions de *Transition* et de *Récompense* impossibles à modéliser, comme dans la prédiction de séries temporelles par exemple. D'autre part, les problèmes de très grandes tailles ont nécessairement un très grand nombre de transitions. Connaître parfaitement chaque transition afin d'y associer la correcte probabilité de *Transition* est virtuellement impossible.

L'existence de ces difficultés amène donc à l'implémentation d'algorithmes capables d'apprendre le modèle et/ou sa solution à partir d'un flux d'observations expérimentales. L'*Apprentissage par Renforcement* [Sutton and Barto, 1998] se propose ainsi de découvrir la *Politique* optimale d'un problème par essai-erreur. Dans ce chapitre, nous allons voir en détail comment se mettent en place de telles méthodes de découverte de la *Politique* optimale.

5.1 Apprentissage par Renforcement : un tour d'horizon

Les méthodes d'*Apprentissage par Renforcement* reposent sur la construction d'une estimation de la *Politique* optimale. Cette estimation est élaborée en étudiant les transitions vécues par le système. Dans ces méthodes, l'agent mène donc des expériences [1] afin d'explorer l'environnement et de "comprendre" comment les actions qu'il peut entreprendre altèrent celui-ci. Le but est d'estimer, à terme, quelles sont les meilleures actions à entreprendre en chaque état et de les exploiter. Une grande difficulté de l'*Apprentissage par Renforcement* est de garantir une exploration suffisante du système pour pouvoir affirmer que les tâches estimées optimales le sont réellement.

5.1.1 Méthodes d'approximation

Pour bâtir une estimation de la *Politique* optimale à partir des expérimentations effectuées par l'agent et des observations recueillies en retour, il existe plusieurs techniques.

Méthodes de Monte-Carlo Dans les méthodes de Monte-Carlo [Bertsekas and Tsitsiklis, 1996], l'agent génère un grand nombre de trajectoires [2]. Les transitions et récompenses observées dans chacune de ces trajectoires sont ensuite utilisées pour corriger l'estimation de la fonction *Valeur* pour chacun des états visités dans la trajectoire. Ces méthodes ne sont pas par nature incrémentales car elles nécessitent d'avoir effectué toute une trajectoire pour mettre à jour les états de la trajectoire.

Méthode par différence temporelle A contrario, les méthodes d'apprentissage par différence temporelle (**TD**(λ) [Sutton, 1988], **SARSA** [Rummery and Niranjan, 1994], **Q-LEARNING** [Watkins and Dayan, 1992], et les architectures acteurs-critiques [Witten, 1977; Barto et al., 1983]) sont incrémentales. Ces méthodes reposent sur la mise à jour de l'estimation de la fonction *Valeur* (ou des fonctions *Valeur* d'action) à l'aide de règles de correction ne prenant en compte que la dernière transition observée. Par exemple, pour la transition de l'état s vers l'état s' après avoir effectué l'action a et obtenu la récompense r, la règle de mise à jour pour l'algorithme **Q-LEARNING** est la

1. L'observation des expériences consiste à noter en quel état s' le système arrive lorsque l'agent effectue l'action a en l'état courant s, ainsi que l'éventuelle récompense r collectée.
2. Une trajectoire est un ensemble de transition depuis un état initial quelconque jusqu'à un état but atteint (ou un nombre d'itérations atteint).

suivante :

$$\hat{\mathcal{Q}}(s,a) = \hat{\mathcal{Q}}(s,a) + \alpha \cdot (r + \gamma \cdot \max_{a' \in \mathbb{A}} \hat{\mathcal{Q}}(s',a') - \hat{\mathcal{Q}}(s,a))$$

Comme le montre cette règle, les méthodes par différence temporelle effectuent une correction de l'estimation actuelle de la fonction *Valeur* ou la fonction *Valeur* d'action par l'estimation de l'*espérance de gains cumulés* recevable d'après l'observation tout juste obtenue ($r+\gamma \cdot \max_{a' \in \mathbb{A}} \hat{\mathcal{Q}}(s',a')$ dans **Q-LEARNING**). Le coefficient α d'apprentissage, qui diminue au cours de l'apprentissage, pondère l'importance de la correction. L'intérêt de ces méthodes d'apprentissage est donc double. D'une part, leurs règles de correction incrémentales permettent de rectifier les estimations faites dès réception d'une nouvelle observation. D'autre part, elles ne nécessitent pas de connaissance *a priori* du modèle puisqu'elles manipulent exclusivement les fonctions *Valeur* et *Valeur* d'action.

5.1.2 Méthodes Directes ou Indirectes

Ces méthodes d'approximation peuvent, ou non, bâtir et se servir d'une représentation du monde au cours de toute exécution pour affiner leurs estimations. Suivant qu'un algorithme d'*Apprentissage par Renforcement* exploite ou non un tel modèle, nous parlons de méthode directe (sans modèle) ou indirecte (avec modèle). Si l'algorithme est indirect, le modèle est appris empiriquement et constitue donc une approximation du monde réel. Ce modèle est alors utilisé ou bien pour effectuer une planification par *Programmation Dynamique* (**VI** ou **PI**), ou bien pour pratiquer des expériences virtuelles, moins coûteuses que les expériences réelles à faire.

Ainsi, l'architecture **DYNA** [Sutton, 1990] se bâtit une représentation d'une monde à partir d'expériences réelles. Cette représentation du monde est utilisée pour effectuer des expériences virtuelles. Ces expériences, sans incidences sur le système réel, permettent de déduire hypothétiquement où mèneraient les actions choisies. Une méthode d'approximation s'en sert alors pour corriger itérativement l'estimation de la *Politique* optimale. Les expériences réelles, effectuées moins souvent que les expériences hypothétiques, sont elles aussi exploitées par la méthode d'approximation [3].

Dans l'instance **DYNA-PI**, une méthode acteur-critique effectue la mise à jour de l'estimation de la *Politique* optimale. Toutefois, cette instance s'adapte mal aux changements dans le monde réel (non stationnarité du modèle) : il lui faut un certain nombre d'expériences pour "s'en apercevoir" et s'y adapter. Pour palier à ce problème, Sutton [1990] montre qu'utiliser la méthode du **Q-LEARNING** s'avère plus efficace pour

3. En plus d'être utilisée par l'algorithme d'apprentissage du modèle.

prendre en compte ces modifications. **DYNA-Q** s'avère ainsi beaucoup plus réactif aux problèmes non stationnaires.

Algorithme 19 : Architecture globale de **DYNA-Q**

pour chaque *pas de temps* t **faire**
  Choisir expérience : réelle ou virtuelle;
  Choisir état de départ s (si expérience réelle, s = état courant);
  Choisir Action a;
  Effectuer a **and** Observer prochain état s' et récompense r.;
  si *Expérience réelle* **alors**
    Mettre à jour modèle;
  $\hat{\mathcal{Q}}(s,a) = \hat{\mathcal{Q}}(s,a) + \alpha \cdot (r + \gamma \cdot \max_{a' \in \mathbb{A}} \hat{\mathcal{Q}}(s',a') - \hat{\mathcal{Q}}(s,a))$;

5.1.3 Compromis Exploration-Exploitation

Un agent utilisant une méthode d'*Apprentissage par Renforcement* fait face à un dilemme. D'une part, il doit acquérir les informations nécessaires à la découverte de la *Politique* optimale. Or, l'acquisition de ces informations passe nécessairement par l'exécution d'actions qui ne sont pas optimales. D'autre part, il doit effectuer les actions qu'il estime optimales afin de résoudre le problème de décisions séquentielles dans l'incertain qui lui est attribué à la base. Ses choix d'actions sont donc contraints par les deux nécessités d'explorer le système et d'exploiter ses connaissances afin d'engranger des récompenses. Satisfaire ces deux contraintes s'avère complexe.

Les preuves de convergence des méthodes d'approximation vues ci-dessus exigent que toutes les transitions soient testées [4]. Le choix d'action doit donc être aussi uniforme que possible afin de garantir que toutes les transitions soient testées. Mais il doit aussi privilégier autant que possible les actions qui semblent optimales afin de capitaliser les récompenses. Deux catégories de méthodes existent pour résoudre ce dilemme : les méthodes dirigées et les méthodes non-dirigées.

Les méthodes non-dirigées

Les méthodes non-dirigistes utilisent très peu d'informations d'apprentissage pour choisir la prochaine action à effectuer ; cette action est en grande partie sélectionnée de manière aléatoire. Ainsi, dans l'approche ϵ-**GREEDY**, l'action à suivre d'après la *Politique* estimée optimale est retenue avec une probabilité $1-\epsilon$ [5]. Dans le cas contraire,

4. En pratique, une exploration partielle suffit à obtenir une *Politique* satisfaisante.
5. $\epsilon \in [0,1]$

une action est aléatoirement prise dans $\mathbb{A}$. Ce choix aléatoire d'action dans $\mathbb{A}$ se fait alors de manière uniforme.

L'approche **SOFTMAX** maintient une *Politique* stochastique suivant une distribution de Boltzmann. Ainsi, en chaque état $s \in \mathbb{S}$, la probabilité d'effectuer l'action $a \in \mathbb{A}$ est :

$$\pi_T(s,a) = \frac{\exp(-\frac{\hat{\mathcal{Q}}(s,a)}{T})}{\sum_{a'\mathbb{A}} \exp(-\frac{\hat{\mathcal{Q}}(s,a')}{T})}$$

Le paramètre T a une incidence directe sur l'exploration. Pour T très grand, la distribution sur les actions est quasi-uniforme ; elles sont alors sélectionnées de façon homogène. Pour T très proche de 0, l'action qui sera la plus probablement choisie est l'action ayant la plus forte *Valeur* d'action. Ainsi, le paramètre T est diminué au cours de l'apprentissage, de sorte que l'exploration laisse progressivement la place à l'exploitation.

Les méthodes dirigées

Les méthodes dirigistes utilisent a contrario des informations d'apprentissage afin de maintenir à jour les heuristiques qu'elles emploient pour choisir les actions. La plupart de ces heuristiques consistent à altérer la *Valeur* d'action $\hat{\mathcal{Q}}(s,a)$ des couples état-action insuffisamment explorés afin d'en augmenter l'intérêt (l'agent cherche les couples état-action à forte *Valeur* d'action).

Certaines de ces heuristiques ont été étudiées et des garanties théoriques ont pu être établies sur leur convergence. Ainsi, en définissant la notion de complexité d'échantillon d'exploration comme étant :

Définition 5.1 (Complexité de l'échantillon d'exploration)**.**

Soit une trajectoire $h = (s_0, a_0, s_1, \ldots)$. Soit $\epsilon > 0$. La complexité de l'échantillon d'exploration d'un algorithme d'*Apprentissage par Renforcement* **A** vis-à-vis de la trajectoire h est le nombre d'itérations T tel que $\forall$ itération $t < T$, la *Politique* $\hat{\pi}_t$ estimée par **A** pour l'état courant s_t n'est pas ϵ-optimale (ou plus formellement $|\hat{\mathcal{V}}_{\hat{\pi}_t}(s_t) - \mathcal{V}^\star(s_t)| < \epsilon$).

Kakade et al. [2003] définit un algorithme d'*Apprentissage par Renforcement* comme étant efficace s'il satisfait la définition :

Définition 5.2 (Algorithme PAC-MDP)**.**

Un algorithme d'*Apprentissage par Renforcement* **A** est défini comme étant PAC-MDP [6] [7] efficace si, $\forall \epsilon > 0$ et $\delta \in [0, 1]$, les complexités algorithmiques et d'échantillon d'exploration de **A** sont inférieures à un polynôme s'exprimant à l'aide des quantités $|\mathbb{S}|$, $|\mathbb{A}|$, $\frac{1}{\epsilon}$, $\frac{1}{\delta}$ et $\frac{1}{1-\gamma}$ avec une probabilité supérieure à $1 - \delta$.

Plusieurs algorithmes de la littérature satisfont à cette définition [8]. Dans l'algorithme $\mathbf{E}^3$ [Kearns and Singh, 2002] [9] , deux *Politiques* sont maintenues à jour. L'une représente la *Politique* optimale estimée, l'autre est une *Politique* calculée de sorte à favoriser les couples état-action insuffisamment visités. Pour un état donné, ces *Politiques* ne sont calculées que lorsque l'état a été visité un nombre minimal de fois [10]. La *Politique* d'exploration est utilisée tant que la fonction *Valeur* liée à la *Politique* d'exploitation n'est pas à un $\frac{\epsilon}{2}$ de la fonction *Valeur* optimale. Lorsque, pour un état donné, la *Politique* d'exploitation a atteint le seuil, l'algorithme bascule sur cette *Politique*. À chaque instant, l'algorithme explore ou exploite donc de façon explicite. L'inconvénient majeur de cet algorithme est qu'il nécessite que la fonction *Valeur* optimale soit déjà connue. Il reste néanmoins le premier algorithme à avoir été démontré PAC-MDP.

L'algorithme **R-MAX**[Brafman and Tennenholtz, 2003] procède en donnant un a priori positif aux couples états-actions inconnus [11]. Ainsi pour ces couples, la *Valeur* d'action est égale à la récompense maximale ($Rmax$) récupérable dans ce problème. L'algorithme effectue une planification par *Programmation Dynamique* avec les valeurs modifiées pour les couples états-actions insuffisamment visités ; cette planification est donc artificiellement orientée en partie vers la découverte des couples inconnus. Au contraire de l'algorithme $\mathbf{E}^3$, cet algorithme utilise une seule *Politique* pour à la fois explorer et exploiter. Il explore ou exploite donc de façon implicite.

Enfin, l'algorithme **MBIE** [12] [Strehl and Littman, 2005] repose sur l'utilisation d'intervalles de confiance sur les valeurs estimées. Ainsi, pour chaque couple état-action, un intervalle est fixé autour de la valeur estimée de *Récompense* récupérable et de la probabilité de *Transition*. Cet intervalle est fonction du nombre de fois que le couple a été testé. Une version modifiée de **VI** est ensuite utilisée pour exploiter la borne supérieure (donc optimiste) de l'estimation.

6. PAC-MDP : Probabely Approximately Correct in MDP
7. Le terme PAC est emprunté à la littérature sur l'apprentissage supervisé (cf. [Valiant, 1984])
8. Ces algorithmes utilisent un modèle (pour la fonction *Transition* principalement) appris pour gérer leur heuristique. Ce sont donc des approches indirectes.
9. *Explicit Explore or Exploit* : Exploration ou Exploitation Explicite
10. Le nombre minimal de visite est un paramètre de l'algorithme.
11. Un couple état-action inconnu est, là aussi, un couple état-action n'ayant pas été testé un nombre minimal de fois, ce nombre minimal étant fixé à l'avance.
12. Model-Based Interval Estimation : Estimation d'intervalle basée sur le modèle.

5.2 Méthodes avec Modèle et Représentations Factorisées : l'architecture SDYNA

Bien entendu, toutes ces méthodes rencontrent des difficultés lors du passage à l'échelle. En premier lieu, les fonctions *Valeur* et *Valeur* d'actions à maintenir à jour sont de plus en plus grandes. Pour des espaces d'états très larges, il devient alors impossible de maintenir les tables associées en mémoire (cf. Chapitre 1). Des techniques d'abstraction doivent alors être mises en place. Ensuite, le trop grand nombre d'états induit qu'il devient inenvisageable de tous les visiter, et donc d'acquérir une connaissance totale du système. Une exploration complète étant impossible, une généralisation de ce qui a pu être expérimenté doit être mise en place et exploitée pour planifier une *Politique* optimale. Pour satisfaire à ces deux points, les FMDPs sont un choix naturel : c'est une technique standard d'abstraction qui permet des généralisations lorsqu'elle utilise des représentations structurées [13].

Algorithme 20 : Architecture globale de **SDYNA**

```
pour chaque pas de temps t faire
    s ← état courant;
    a ← Décision en fonction de π_t(s);      // π_t est la stratégie actuelle
    Effectuer a et observer s' et r;
    Apprentissage "Factorisé" incrémental d'une observation o = (s, a, s');
    Planification "Factorisée" → π_{t+1};
```

L'architecture générale **SDYNA** [Degris et al., 2006b](cf. Algorithme 20) fournit un cadre de travail exploitant les représentations factorisées pour pratiquer de l'*Apprentissage par Renforcement* dans des problèmes de grandes tailles. Une instance de cette architecture nécessite le choix d'une structure de données pour modéliser le problème (ARBRE DE DÉCISION, ADD, etc.). Afin de construire expérimentalement une estimation des fonctions *Transition* et *Récompense* du FMDP, l'instance doit proposer un algorithme d'apprentissage supervisé incrémental adapté à la structure de données choisie. De plus, une version de **PI** ou **VI** adaptée à la structure retenue doit être fournie pour effectuer la planification d'une *Politique* optimale par *Programmation Dynamique* à partir de l'estimation du FMDP. Cette architecture propose donc de mettre en place un *Apprentissage par Renforcement* basé sur un modèle appris mais exploitant des techniques de planification par *Programmation Dynamique*.

13. Au travers des CONTEXTES

5.2.1 L'algorithme SPITI

SPITI, une instanciation de cette architecture proposée dans le même article, utilise les ARBRES DE DÉCISION comme représentation en s'appuyant sur l'algorithme **SVI** [Boutilier et al., 1999] pour l'étape de planification (ligne 6) et sur l'algorithme **ITI** [Utgoff et al., 1997] pour l'induction des ARBRES DE DÉCISION durant la phase d'apprentissage (ligne 5). Dans **SPITI**, la prise de décision (ligne 3) est un ϵ-**GREEDY** qui, au lieu d'utiliser la *Politique* optimale courante (exploitation), peut essayer de nouvelles actions non optimales afin d'améliorer la connaissance du modèle (exploration). Nous allons voir plus en détail comment est faite l'implémentation **SPITI** de **SDYNA**.

Contenu d'une observation

L'un des objectifs de **SPITI** est donc d'apprendre les fonctions de *Transition* et de *Récompense* d'un FMDP. Rappelons qu'un MDP sous forme factorisée a une fonction *Transition* par variable et par action, et une fonction *Récompense* par action. Par conséquence, $|\mathbb{A}|\times(|\mathbb{X}|+1)$ processus d'apprentissage sont lancés pour construire des estimations de toutes ces fonctions. Ces processus suivent tous un même algorithme : **ITI**, en l'occurrence dans **SPITI**.

Chaque processus est alimenté par un flux d'observations qui décrivent les transitions vécues par le système. Pour fournir l'intégralité des informations nécessaires à l'apprentissage supervisé du FMDP, chaque observation se présente sous la forme d'un vecteur

$$o = \langle x_1, \ldots, x_m, a, x'_1, \ldots, x'_n, r \rangle$$

décrivant les informations suivantes :

— l'état précédent, renseigné par les instanciations de variables $x_1, \ldots, x_m$,
— le nouvel état du système, précisé par les instanciations de variables $x'_1, \ldots, x'_n$,
— l'action a effectuée lors de la transition,
— et la récompense r collectée.

Chaque processus n'a besoin que d'une sous-partie du vecteur o. Ainsi le processus qui apprend la *Transition* de la variable X_i pour l'action a n'aura besoin que du sous-vecteur

$$o_{a,X_i} = \langle x_1, \ldots, x_m, x'_i \rangle$$

De même, le processus apprenant la fonction *Récompense* pour l'action a exploitera le sous-vecteur

$$o_{a,\mathcal{R}} = \langle x_1, \ldots, x_m, r \rangle$$

Bien entendu, tout processus apprenant une fonction *Transition* ou *Récompense* pour l'action a_1 n'a pas à étudier les observations o liées à d'autres actions (donc telles que $o[a] \neq a_1$). Un processus maître doit donc chaperonner l'apprentissage et s'assurer que chaque processus d'apprentissage reçoit les observations correspondantes à l'action pour laquelle le processus apprend sa fonction. La tâche du processus principale est donc d'envoyer aux bons sous-processus les bonnes informations.

Apprentissage d'ARBRES DE DÉCISION

Plusieurs algorithmes d'apprentissage d'ARBRES DE DÉCISION depuis un corpus Ω d'observations existent dans la littérature : **CART** [Breiman et al., 1984a], **C4.5** [Quinlan, 1993], etc. Le principe d'induction d'un ARBRE DE DÉCISION est globalement le même dans tous ces algorithmes : séparer récursivement une base de données en plusieurs sous-ensembles à l'aide d'un ensemble de variables (plus communément appelées attributs ou tests). Dans notre cadre de travail, cet ensemble de variables est l'ensemble $\{X_1, \ldots, X_m\}$ décrivant l'état du système avant une transition.

Pour séparer un ensemble d'observation, le processus sélectionne une variable. La séparation de la base de données se fait alors suivant les modalités assumées dans chaque observation par cette variable. Ainsi, si la variable X_i est choisie, $|X_i|$ sous-bases sont créées. Chaque sous-base Ω_{x_i} reçoit les observations o où $o[X_i] = x_i$. Le processus de séparation est ensuite répété sur ces sous-bases. Il s'arrête lorsque aucune variable n'est utilisable pour effectuer la séparation [14].

Pour sélectionner la variable suivant laquelle se fait cette séparation, une mesure est utilisée pour détecter le partitionnement le plus pertinent. Cette mesure compare la base originelle aux sous-bases obtenues. Le ratio du gain d'information, par exemple, mesure combien l'entropie a diminué dans les sous-bases comparativement à l'entropie dans la base initiale. D'autres statistiques peuvent être employées : l'indice de Gini, le test de Kolmogorov-Smirnoff, le test du χ^2, etc. Quel que soit le test employé, le principe reste le même : la variable qui a la meilleure performance suivant ce test est la variable choisie pour effectuer la séparation.

La structure de l'ARBRE DE DÉCISION appris suit le partitionnement de la base intiale : à chaque nœud n de l'ARBRE DE DÉCISION est associé un des sous-ensembles Ω_n générés. Les observations de ce sous-ensemble sont compatibles avec l'instanciation des variables en n [15]. La variable installée au nœud n est la variable sélectionnée pour

14. Soit aucune variable n'est pertinente pour séparer la base, soit il n'y a plus de variable disponible
15. Rappel : tout chemin de la racine vers un nœud d'un ARBRE DE DÉCISION est une instanciation de CONTEXTE. Ainsi les observations stockées en un nœud n sont compatible avec ce CONTEXTE.

subdiviser la sous-base Ω_n. De fait, les ensembles d'observations placés aux feuilles de l'arbre forment une partition de l'ensemble de toutes les observations Ω. De même, l'ensemble d'observations associé à la racine est Ω.

Il est aisé de voir que ce processus d'apprentissage s'intègre bien à notre cadre de travail : le partitionnement de la base d'observation initiale revient à partitionner l'espace d'état et à ranger les observations par CONTEXTES compatibles. Les bases de données aux feuilles nous donnent alors toutes les informations nécessaires pour connaître les valeurs assumées par la fonction. De plus, seules les variables appartenant au SUPPORT de la fonction apprise devraient théoriquement être sélectionnées pour les séparations. En effet, par définition, ce sont les seules à conditionner le comportement de la fonction apprise. En pratique, comme dans les cas d'apprentissage plus généraux, il n'est pas impossible qu'une autre variable soit sélectionnée par effet de bruit sur les données.

Le problème de ce méta-algorithme d'apprentissage est qu'il est statique. Dans le cas où une seule nouvelle observation est ajoutée à la base, il est nécessaire de refaire toute l'induction de l'arbre depuis la base de données initiale. Or dans notre cadre de travail, nous sommes typiquement dans une situation où les observations arrivent dynamiquement, sous forme d'un flux d'information. Il est nécessaire d'adapter l'algorithme pour le rendre incrémental. L'algorithme **ITI** [Utgoff et al., 1997] propose une telle adaptation.

ITI : Apprentissage Incrémental d'ARBRES DE DÉCISION

Algorithme 21 : ITI : Induction Incrémentale d'ARBRE DE DÉCISION

Données : une observation $o = (x_1, \ldots, x_m, Y)$, un ARBRE DE DÉCISION T à mettre à jour

1 **début**
2 Mettre à jour les statistiques de T avec o ; `// cf. Algorithme Ajouter Exemple (22)`
3 Mettre à jour la structure de T avec o ; `// cf. Algorithme Mettre à Jour arbre de décision (23)`

Résultat : un ARBRE DE DÉCISION pour $P(Y|X_1, \ldots, X_m)$

Pour rendre l'apprentissage incrémental, l'algorithme **ITI** se garde en permanence la possibilité de réviser toute sélection de variable en tout nœud de l'arbre généré, et, lorsqu'une telle révision est requise, l'algorithme met à jour l'arbre de telle sorte que le coût total de cette révision soit inférieur au coût d'une ré-induction complète de l'arbre. Ainsi, lorsqu'une nouvelle observation arrive, l'ensemble des bases compatibles

avec cette observation sont mises à jour, ainsi que les statistiques liées. Lorsque, pour un nœud donné, la mesure associée à une variable donnée devient meilleure que la mesure associée à la variable actuellement en place, les deux variables sont échangées. Il y a donc deux phases dans cet algorithme : une mise à jour des bases suivie d'une mise à jour structurelle.

La mise à jour des bases est directe. En partant de la racine, à chaque nœud n rencontré, sa base Ω_n et les mesures associées sont mises à jour en fonction de la nouvelle observation o. Puis l'algorithme descend en sélectionnant le nœud fils $n.\textsc{child}(o[n.var])$.

Algorithme 22 : ITI - AJOUTER EXEMPLE

Données : l'observation $o = \{x_1, \ldots, x_m, y\}$ ajoutée dans l'ARBRE DE DÉCISION T

```
1  début
2      Nœud n ← racine de T;
3      répéter
4          Ajouter o à Ω_n;
5          pour chaque variable X_i ∈ 𝕏 faire
6              Mettre à jour statistique p_n^i en n pour X_i;
7          Marquer n;
8          si n n'est pas une feuille alors
9              n ← n.FILS(o[X_i]);
10     jusqu'à n est une feuille;
```

La mise à jour structurelle est un peu plus technique. Rappelons que celle-ci doit être de coup moindre qu'une ré-induction complète de l'arbre pour être valable. La solution retenue par Utgoff et al. [1997] pour installer en un nœud donné la meilleure variable est de procéder par *transposition récursive*. Cette méthode permet de conserver en partie la structure du sous-arbre en n qui, malgré ce changement de variable, est éventuellement toujours pertinente.

La méthode de transposition vérifie d'abord que tous les fils du nœud n où la variable X_{select} doit être installée ont X_{select} pour variable. Si tel n'est pas le cas, la méthode de transposition s'appelle récursivement sur ces nœuds fils. L'appel est terminal lorsque le nœud courant est associé à X_{select} ou est une feuille. Pour le second cas, la feuille est séparée suivant X_{select}.

Lorsque tous les fils du nœud courant sont associés à la variable X_{select}, un échange de variable a lieu localement. Cet échange se fait comme l'échange de variable vu au

Algorithme 23 : ITI - METTRE À JOUR ARBRE DE DÉCISION

Données : un ARBRE DE DÉCISION T après l'ajout d'une nouvelle observation o (avec l'Algorithme **ITI - AJOUTER EXEMPLE** (22)), n le nœud courant mis à jour

```
1  début
2      Soit 𝕏_n : {X_i ∈ 𝕏 | ∀ nœud n_p du chemin amenant en n, X_i ≠ n_p.var};
3      répéter
4          X_select = meilleure variable à installer en n;
5          𝕏 = 𝕏 \ {X_select};
           // Installer X_select au noeud n
6          si n.var ≠ X_select alors
7              TRANSPOSER(T,n,X_select);          // cf Algorithme transposer (24)
8          pour chaque x_{n.var} ∈ Dom{n.var} faire
9              si n.FILS(x_{n.var}) est marqué alors
10                 ITI - METTRE À JOUR ARBRE DE DÉCISION(T, n.FILS(x_{n.var}));
11     jusqu'à 𝕏_n = ∅ or ∄X_i ∈ 𝕏_n t.q. X_i peut être installée en n;
12     si ! T.EST TERMINAL(n) alors
13         Transformer n en feuille;
14     Démarquer n;
```

Chapitre 2. La seule différence est qu'il n'affecte que le nœud n et ses nœuds fils ; les autres nœuds du graphe restent inchangés.

Au cours de cet échange, seules les bases de données et les statistiques des nœuds fils doivent être recalculées. En effet, le nœud n reste inchangé d'un point de vue informations contenues dans sa base de données. De même, les base de données de ses petits-fils sont elles aussi non affectées par l'opération. Seules les bases filles doivent être refaites pour devenir cohérentes avec la nouvelle séparation de la base Ω_n suivant la variable X_i. Toutefois, pour reconstituer ces bases filles, il n'est pas nécessaire de parcourir toute la base Ω_n pour la séparer. En effet, l'union des bases petites-filles est plus efficace. [16]

La Figure 5.1 illustre cette mise à jour structurelle.

Pour plus d'efficacité dans ses deux phases de mise à jour, lors de l'ajout d'une nouvelle observation, les nœuds compatibles avec l'observation (et donc visités lors de l'ajout de l'observation) sont marqués. En effet, seuls ces nœuds sont susceptibles de changer de variable du fait de cette nouvelle observation. Ensuite, durant la phase de

16. Suivant la mesure utilisée pour la sélection de variable, il peut même ne pas être nécessaire d'unir ces sous-bases. En effet, certains tests, comme celui du χ^2, nécessitent une table de contingence. Cette table devient la seule information pertinente à maintenir en chaque nœud. Cette table en un nœud donné s'obtient aisément à partir des tables filles (par addition).

Algorithme 24 : TRANSPOSER : Installation de variable en un nœud

Données : un ARBRE DE DÉCISION T, n un nœud de T, et X_{select} la variable à installer en n

```
1  début
2      si n.var ≠ X_select alors
3          si n est une feuille alors
4              Transformer n en nœud interne associé à X_select;
5              Créer ∀x_select ∈ Dom{X_select} un nœud fils n.FILS(x_select);
6              Subdiviser Ω_n en |Dom{X_select}| bases Ω_{n.FILS(x_select)};
7          sinon
8              pour chaque x_{n.var} ∈ Dom{n.var} faire
9                  TRANSPOSER(T, n.FILS(x_{n.var}), X_select);   // Tous les fils de n
                   ont X_select pour variable à la fin de cette boucle.
10             Échanger localement n.var avec X_select;          // cf.
                Algorithme permuter (3)
11             Mettre à jour les statistiques des nœuds fils à partir des statistiques
                des nœuds petit-fils;
12         Marquer n;
```

mise à jour structurelle, seuls les nœuds marqués sont vérifiés. Néanmoins, tout nœud subissant une transposition de variable au cours de cette phase est marqué à son tour. En effet, la variable qui lui était associée ne l'est plus et rien ne garantit que la variable attribuée lors de la transposition soit la plus pertinente.

Cette mise à jour coûte moins qu'une ré-induction complète du graphe car seule une partie est réellement mise à jour. Le seul cas où ce mécanisme entraîne une ré-induction complète est éventuellement lors d'un changement de variable pour le nœud racine.

SPITI : Planification et Apprentissage avec des ARBRES DE DÉCISION

Dans **SPITI**, pour chaque $P(X'_i \mid a, \mathbb{X})$ [17] et pour chaque fonction récompense $\mathcal{R}(\mathbb{X}, a)$, un ARBRE DE DÉCISION est appris incrémentalement. Degris et al. [2006a] utilisent dans **SPITI** une version de **ITI** où le critère de sélection de variable est le test du χ^2 [18] au lieu du ratio de gain d'information et indique deux raisons majeurs pour ce choix : White and Liu [1994] montre que le test du χ^2 n'a pas de biais vis à vis des variables multimodales. De plus, le test du χ^2 permet à **SPITI** de faire du pré-élagage : une feuille ne sera pas développée si aucune variable ne satisfait au test du χ^2.

17. La découverte de chaque PARENTS(X'_i) et de chaque PORTÉE($\mathcal{R}$) est laissée à l'algorithme d'apprentissage.

18. Pour l'apprentissage des fonctions *Récompense*, un test de régression au sens des moindres carrés est utilisé.

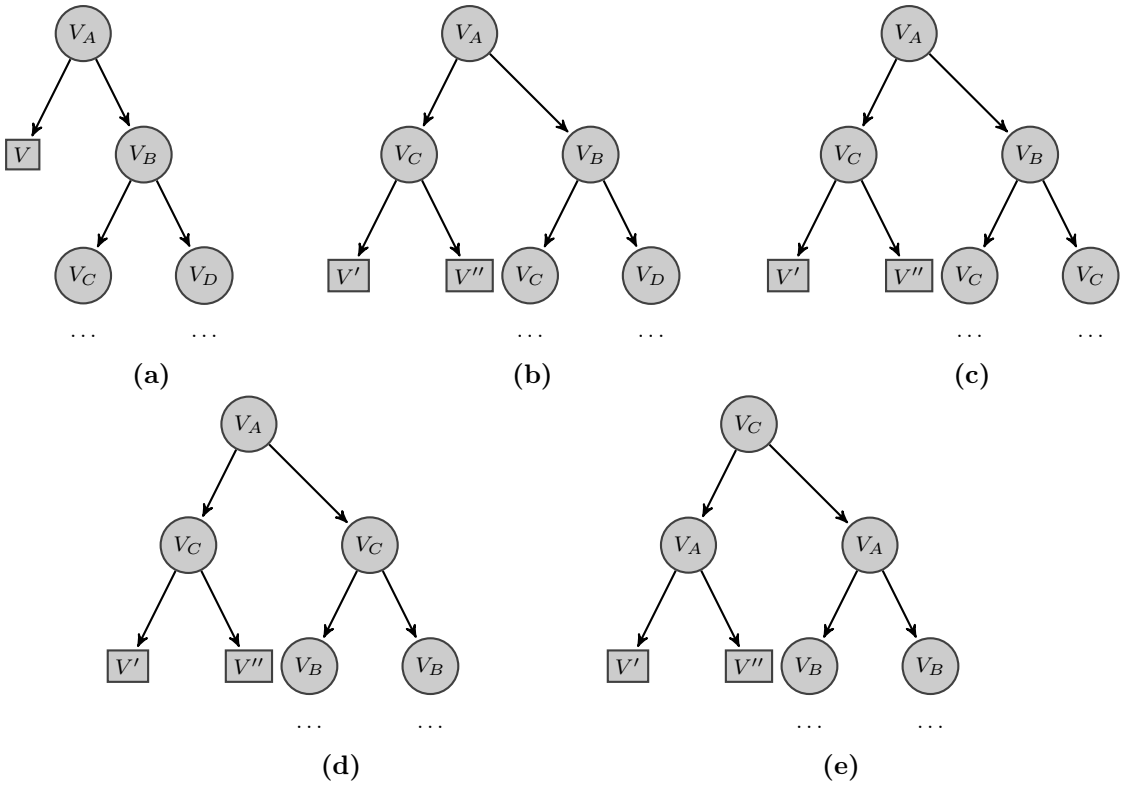


FIGURE 5.1 – *Transposition de la variable V_C au nœud racine occupé par V_A (a $\longrightarrow$ e). L'étape 1 (b) consiste à subdiviser le nœud fils terminal à gauche de la racine pour faire apparaître V_C dans cette branche. L'étape 2 (c) voit V_C transposé au nœud fils droit du nœud associé à V_B. L'étape 3 (d) est celle où V_B est transposée avec V_C.*

Dans les ARBRES DE DÉCISION représentant les $P(X'_i \mid A, \mathbb{X})$, les feuilles sont associées à une distribution de probabilités sur X'_i (voir Figure 5.2). La distribution de probabilités de X'_i en une feuille l est estimée à partir des fréquences de X'_i dans Ω_l. Dans l'ARBRE DE DÉCISION représentant $\mathcal{R}(\mathbb{X}, a)$, à chaque feuille est associée une valeur réelle : la récompense moyenne observée dans la base d'observation.

À intervalle fixe en nombre d'observations ajoutées, l'algorithme **SPITI** lance une planification pour mettre à jour son estimation de la *Politique* optimale. Cette planification utilise une version modifiée de **SVI**. En effet, chaque ajout d'observation ne modifie que relativement le modèle. Lancer alors une recherche complète de la *Politique* optimale toutes les quelques observations est bien trop coûteux pour le gain faible en précision de la fonction *Valeur*. Pour cette raison, la version modifiée n'effectue qu'un nombre défini à l'avance d'itération de la valeur.

À partir de l'estimation courante de la *Politique* optimale, le moteur de décision choisit l'action à prendre en appliquant la méthode ϵ-**GREEDY**.

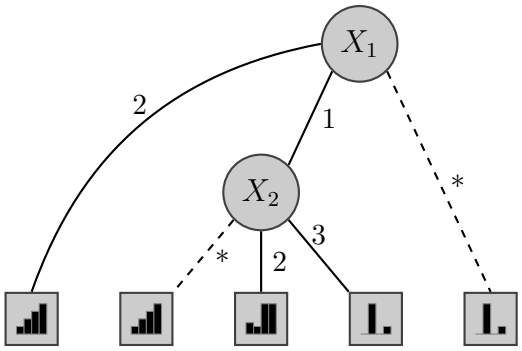


FIGURE 5.2 – *Un* ARBRE DE DÉCISION *représentant une distribution de probabilité* $P(Y|X_1, X_2)$. *Les feuilles contiennent les distributions de probabilité* Y. *Par ailleur, cet* ARBRE DE DÉCISION *indique que* $P(Y|X_1 = 2) = P(Y|X_1 = 1, X_2 \notin \{2, 3\})$ *et que* $P(Y|X_1 \notin \{1, 2\}) = P(Y|X_1 = 1, X_2 = 3)$. *Ces feuilles sont équivalentes et devraient être fusionnées.*

5.2.2 Architecture SDYNA avec RGFORs multivaluées

Dans le cadre de cette thèse, nous nous sommes naturellement intéressés à la mise en place d'une instance de **SDYNA** qui exploiterait des RGFORs multivaluées ; ces dernières s'avèrent en effet plus intéressantes ques les ARBRES DE DÉCISION pour les problèmes de grandes tailles. Dans cette optique, l'algorithme de planification dédié à cette structure que nous proposons d'utiliser est l'algorithme **SPUMDD** vu au Chapitre 4. De plus, pour la prise de décision, nous restons sur une approche ϵ-**GREEDY**. Néanmoins, nous nous retrouvons face à une difficulté pour l'algorithme d'apprentissage incrémental : il n'existe pas à notre connaissance d'algorithme incrémental d'apprentissage de RGFORs multivaluées.

Plusieurs articles s'intéressent à l'apprentissage de RGFORs. L'approche proposée dans [Oliver, 1993] et [Oliveira and Sangiovanni-Vincentelli, 1994] repose sur l'induction d'un arbre en utilisant l'un des algorithmes de l'état de l'art (**CART**, **C4.5**) dans un premier temps. Puis cet arbre est réordonné de telle manière que la recherche de sous-parties isomorphes soit simplifiée. Enfin, ces sous-parties isomorphes sont fusionnées en utilisant comme critère le principe de Minimum Description Lenght.

Kohavi [1994b] et Kohavi and hsin Li [1995] proposent une approche récursive : en suivant un ordre fixé à l'avance des variables, il procède à une décomposition des observations de la base d'apprentissage suivant les modalités que peut prendre la variable de l'appel récursif courant. Cette décomposition se poursuit jusqu'à ce que, en toute feuille, la fonction représentée assume la même valeur pour toutes les observations présentes. Une opération de réduction est alors effectuées pour obtenir un diagramme de décision.

Ces algorithmes ne peuvent s'appliquer à un apprentissage en ligne puisque les arbres sont déduits de bases de données fixes. Ainsi, ajouter de nouvelles observations

à la base de données demande d'apprendre un tout nouvel arbre. Ces approches ne sont donc pas de bonnes candidates pour une architecture **SDYNA**. Être capable de mettre à jour l'arbre, et sa version réduite en RGFOR, sans avoir à parcourir à nouveau toute la base de données serait un grand atout qui n'a pas encore été proposé à notre connaissance. Dans le chapitre suivant, nous proposons un tel algorithme.

Chapitre 6

Apprentissage incrémental et Instance SDYNA pour RGFORs

Dans le chapitre précédent, nous avons vu comment les techniques d'*Apprentissage par Renforcement* s'implémentaient dans le cadre factorisé. Ainsi, en fonction de la structure de données choisie pour représenter le modèle, ces implémentations reposent sur l'utilisation d'un algorithme d'apprentissage incrémental et d'un algorithme de planification exploitant cette structure. L'instance **SPITI** par exemple exploite les ARBRES DE DÉCISION comme structure de données. Ces derniers sont maintenus à jour grâce à l'algorithme Algorithme **ITI** (21). La *politique* optimale est estimée en utilisant une version modifiée de l'Algorithme **SVI** (9). Toutefois, de manière générale, les ARBRES DE DÉCISION sont moins efficaces que les RGFORs. Il est donc légitime de rechercher une instance **SDYNA** manipulant des RGFORs.

Ce chapitre présente nos travaux de recherche sur l'algorithme **SPIMDDI** : l'extension de **SDYNA** aux RGFORs multivaluées, et donc aux méthodes de planification les plus efficaces. Nous allons dans un premier temps décrire **IMDDI**, notre proposition pour un algorithme d'apprentissage incrémental d'RGFORs. Puis nous présenterons la mise en place de l'algorithme **SPIMDDI** ainsi qu'une analyse de ses performances. Ces travaux ont fait l'objet de deux publications [Magnan and Wuillemin, 2015a,b]

6.1 IMDDI : Algorithme d'apprentissage incrémental de RGFORs multivaluées

Dans une architecture **SDYNA**, l'agent capte un flot continu d'informations sur son environnement au fur et à mesure qu'il l'expérimente. Apprendre une nouvelle RGFOR

pour chaque nouvelle observation reçue est évidemment trop coûteux. Un apprentissage incrémental doit donc être mis en place afin d'exploiter de manière optimale ce flux. Tout algorithme réalisant cet apprentissage doit alors être capable de mettre à jour la structure de données qu'il apprend au lieu de la reconstruire complètement lorsque de nouvelles informations lui parviennent.

Dans cette section, nous décrivons **IMDDI**, un nouvel algorithme réalisant un tel apprentissage incrémental de RGFORs. Dans un premier temps, cette présentation se focalisera sur l'estimation d'une distribution de probabilités pour une variable multimodale Y conditionnellement à un ensemble de variables multimodales $\mathbb{X} = \{X_1, \ldots, X_m\}$ (cf Figure 5.2). En effet, apprendre le modèle factorisé des *Transitions* consiste à apprendre plusieurs distributions de probabilités conditionnelles : une par variable caractérisant le système à $t+1$ sachant les variables à l'instant t [Boutilier et al., 1999]. Nous verrons que l'apprentissage d'autres fonctions telle que la fonction $\mathcal{R}$ repose sur des algorithmes similaires.

6.1.1 Apprentissage d'une distribution de probabilités conditionnelles

Considérer une distribution de probabilités sur des variables discrètes comme une fonction multimodale permet de la représenter par un graphe de fonction. La Figure 6.1 montre ainsi différents graphes de fonction pour une même distribution de probabilités.

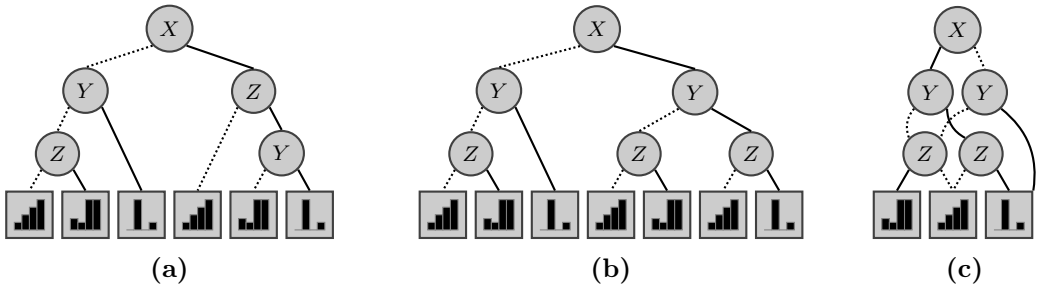


FIGURE 6.1 – *Une distribution de probabilité représentée à l'aide de (a) un* ARBRE DE DÉCISION*, (b) un* ARBRE DE DÉCISION ORDONNÉ *($X \succ Y \succ Z$) et (c) un* MDD *(utilisant le même ordre global).*

Pour rappel, il y a deux différences majeures entre un ARBRE DE DÉCISION (Figure 6.1a) et une RGFOR (Figure 6.1c). Premièrement, une RGFOR est contrainte par un ordre global sur les variables. Les variables doivent apparaître sur chaque branche de la RGFOR en accord avec cet ordre global. Cet ordre a naturellement un fort impact sur la compacité de la RGFOR. Nous appelons ARBRE DE DÉCISION ORDONNÉ un ARBRE DE

DÉCISION ayant cette contrainte d'ordre global sur ses variables (Figure 6.1b). Deuxièmement, une RGFOR peut réduire sa taille en fusionnant ses sous-arbres isomorphes. Ainsi, les nœuds de la RGFOR représentée en Figure 6.1c qui ont plusieurs parents représentent les racines de sous-arbres fusionnés. La complexité de réduction d'un ARBRE DE DÉCISION ORDONNÉ T en une RGFOR est en $O(|T| \cdot \log |T|)$ où $|T|$ est le nombre de nœuds dans T [Bryant, 1986].

Dans le cadre de l'apprentissage non-incrémental de RGFOR, l'approche proposée par Oliver [1993] consiste à simplement mixer un algorithme d'apprentissage d'ARBRE DE DÉCISION avec un algorithme de transformation de ARBRE DE DÉCISION en RGFOR. Un premier algorithme incrémental pour apprendre une RGFOR (nommé **ITI**+DD dans le reste de l'article) est proposé ici et suit cette approche : i) simplement utiliser un algorithme incrémental comme **ITI** pour apprendre un ARBRE DE DÉCISION, puis ii) choisir un ordre global sur les variables et construire l'ARBRE DE DÉCISION ORDONNÉ conséquent, et finalement iii) construire la RGFOR par fusion de nœuds. Dans **ITI**+DD, les étapes ii) et iii) sont donc effectuées à chaque nouvelle observation. L'étape iii) est en général peu coûteuse. Par contre, l'étape ii) (recherche d'un ordre global pour une RGFOR minimisant sa taille) est NP-difficile [Friedman and Supowit, 1990].

Un point décisif de l'algorithme **IMDDI** consiste donc en une heuristique d'estimation de l'ordre optimal qui permet de se passer de cette étape coûteuse : nous proposons l'algorithme **IOTI**, une version modifiée de **ITI** qui manipule un ARBRE DE DÉCISION ORDONNÉ en lieu et place d'un ARBRE DE DÉCISION. De cet arbre ordonné, et seulement si nécessaire, la RGFOR est déduite (étape iii). L'Algorithme **IMDDI** (25) présente ces étapes pour toute observation o fournie par un flux en ligne.

Algorithme 25 : IMDDI : Induction Incrémental de RGFOR pour $P(Y|X_1, \ldots, X_m)$

Données : une observation $o = \langle x_1, \ldots, x_m, y \rangle$

```
1 début
2     Mettre à jour l'ARBRE DE DÉCISION ORDONNÉ avec o;
      // Cette étape est réalise par l'algorithme IOTI : cf. Algorithmes Ajouter
         Exemple (26) et Mettre à Jour ARBRE DE DÉCISION ORDONNÉ (27)
3     si mise à jour requise alors
4         Réduire l'arbre en RGFOR ;
          // étape iii) : cf. Algorithme Réduire arbre de décision ordonné (28)
```

Résultat : une RGFOR pour $P(Y|X_1, \ldots, X_m)$

6.1.2 IOTI : Induction Incrémentale d'Arbre de Décision Ordonné

L'algorithme **IOTI** est basé sur **ITI**. Toutefois, cet algorithme a été modifié afin de mettre à jour la structure tout en prenant en compte l'ordre global. Étant donnée la nouvelle observation o à intégrer dans le modèle, les points critiques à gérer sont : i) comment o modifie l'ordre global, ii) comment mettre à jour la structure en accord avec l'ordre global, et enfin iii) comment sélectionner une variable en chaque nœud de l'arbre. Ces modifications sont décrites dans les sous-sections suivantes.

Sélection de Variables

L'algorithme **ITI** n'a pas de restriction concernant la mesure utilisée pour la sélection de variables. Dans Utgoff et al. [1997], le Ratio du Gain d'Information est employé mais les auteurs rappellent que tout autre test est tout aussi valable. Degris et al. [2006b] propose de se servir de la statistique du χ^2.

Dans notre cas d'étude, nous devons gérer des variables multimodales. La mesure employée ne doit donc pas avoir de biais envers ces variables. Nous proposons pour **IOTI** d'utiliser la statistique G [Mingers, 1989]. Cette statistique n'a en effet pas de biais pour les variables multimodales [White and Liu, 1994]. De plus, d'une part, le test G est similaire au test du χ^2 si le nombre d'échantillons est suffisamment grand ; d'autre part, Dunning [1993] argumente que la statistique G gère mieux les occurrences rares que la statistique du χ^2.

Pour utiliser la statistique G comme mesure pour la sélection de variables, nous procédons de la façon suivante. Soit n le nœud auquel nous voulons soit installer une variable (si n est une feuille), soit nous assurer que la variable actuellement installée est la plus pertinente. Soit $\mathbb{X}_n$ l'ensemble des variables potentiellement candidates en ce nœud n[1]. Pour chaque variable $X_i \in \mathbb{X}_n$, n garde un tableau de contingence Ω_n comptant pour chaque couple de modalité (x_i, y) des variables X_i et Y le nombre d'observations passées en ce nœud pour lesquelles X_i et Y étaient instanciées à x_i et y.

En posant :

— $\omega_n^{(x_i,y)}$ le nombre d'observations en Ω_n pour $X_i = x_i$ et $Y = y$,

— $\omega_n = |\Omega_n|$,

— $\omega_n^{(\cdot,y)}$ et $\omega_n^{(x_i,\cdot)}$ les totaux marginaux ($\omega_n^{(\cdot,y)} = \sum_{x_i} \omega_n^{(x_i,y)}$),

la statistique G pour la variable X_i est alors calculée ainsi :

$$G_n^i = 2 \cdot \sum_{x_i \in Dom X_i} \sum_{y \in Dom Y} \omega_n^{(x_i,y)} \ln \frac{\omega_n^{(x_i,y)}}{e_n^{(x_i,y)}} \tag{6.1}$$

1. Rappelons que chaque variable ne peut apparaître qu'une fois dans tout chemin

où $e_n^{(x_i,y)} = \frac{\omega_n^{(\cdot,y)} \cdot \omega_n^{(x_i,\cdot)}}{\omega_n}$ [2] . La statistique G suit une distribution du χ^2 au degré de liberté $\nu = (|X_i| - 1)(|Y| - 1)$. Dans ce cas-ci, la statistique G mesure à quel point les observations que nous avons faites sont décorrélées des observations que nous aurions pu faire si X_i et Y étaient indépendantes. Ainsi, plus cette statistique G est grande, plus X_i et Y sont liées ; et donc plus il semble pertinent d'installer X_i au nœud n.

Ajoutons que, comme la statistique du χ^2, la statistique G a une propriété intéressante : elle peut être utilisée comme critère de pré-élagage pour empêcher l'arbre de trop grandir. Pour se faire, il suffit de regarder la p-valeur associée à la statistique G calculée. La p-valeur sert en effet d'indicateur de validité. Si la p-valeur n'excède pas un certain seuil, alors l'hypothèse nulle ne devrait pas être rejetée. Dans notre cas, l'hypothèse nulle est que X_i et Y ne sont pas fortement dépendantes : X_i ne devrait pas être installée en n.

IOTI utilise donc la p-valeur comme mesure pour la sélection de variables. La variable ayant la plus faible [3] p-valeur sera retenue [4] . Si cette p-valeur est inférieure à un seuil fixé τ_1 [5], nous installons la variable au nœud n. Dans le cas contraire, et si le nœud n n'est pas une feuille, n est transformé en feuille. Soulignons le fait que ce seuil a le même sens que le niveau de fiabilité pour un test d'indépendance du χ^2 (ou G).

Ajouter une nouvelle observation o

Pour rappel, nous considérons qu'une observation o consiste en une instanciation de toutes les variables observées $\{X_1, \cdots, X_m, Y\}$. Par construction, il existe un unique chemin de la racine à une feuille de l'ARBRE DE DÉCISION ORDONNÉ qui est compatible avec o. Ajouter o à l'ARBRE DE DÉCISION ORDONNÉ consiste à mettre à jour les bases

2. Dans le cas où $\omega_n^{(x_i,y)}$ est nul (il n'y a pas eu d'observations où $X_i = x_i$ et $Y = y$), la quantité $\omega_n^{(x_i,y)} \ln \frac{\omega_n^{(x_i,y)}}{e_{x_i,y}}$ est considérée nulle, en vertu de $\lim_{x \longrightarrow 0} x \ln x = 0$.

3. Pour un tableau de contingence donné, la p-valeur donne la probabilité d'avoir observé cette distribution sachant les variables indépendantes. Or, comme déjà évoqué, plus la statistique G est forte, moins les variables *se comportent* de manière indépendante. Donc moins il est probable qu'elles le soient. D'où la recherche d'une p-valeur faible.

4. Pour que la p-valeur soit statistiquement signifiante, il faut s'assurer d'avoir suffisamment d'observations au nœud où la sélection de variables a lieu. Ce qui implique d'une part d'avoir un nombre total d'échantillons en ce nœud supérieur à la vingtaine, et d'autre part que chaque paire de modalité (X_i, y) pour une variable X_i donnée ait un nombre minimum de cinq observations. Dans le cas où ces deux critères sont vérifiés pour X_i, alors la p-valeur associée est considérée comme valable. Dans le cas contraire, la p-valeur est considérée nulle par la suite.

Le premier critère est aisément vérifiable. Le second est en revanche plus difficile, voire impossible à satisfaire dans notre cadre de travail. En effet, dans les cas où les transitions sont déterministes, certains couples de modalités n'auront pas d'observations affiliées puisque ces situations ne peuvent se produire. Pour palier à ce problème et décider si les statistiques sont suffisantes, un nombre palier moyen d'observations est fixé. Si le nombre total d'observations est supérieur à $Palier * NbCouplesPossibles$, nous considérons avoir suffisamment d'informations au nœud pour avoir une p-valeur valable.

5. En pratique, ce seuil τ_1 est fixé à 0,05.

Ω_n et les statistiques de chaque nœud n de ce chemin. Avec X_n la variable installée en un nœud interne n, p_n^i la p-valeur de la variable $X_i \in \mathbb{X}_n$ au nœud n, $n.\textbf{FILS}(x_n)$ le nœud fils de n pour la modalité x_n de X_n, l'Algorithme **IOTI - AJOUTER EXEMPLE** (26) décrit formellement cette mise à jour de la structure de l'ARBRE DE DÉCISION ORDONNÉ.

Algorithme 26 : IOTI - AJOUTER EXEMPLE : Ajout d'une nouvelle observation o

Données : l'observation $o = \langle x_1, \ldots, x_m, y \rangle$ ajoutée dans l'ARBRE DE DÉCISION ORDONNÉ T

```
1 début
2     Nœud n ← racine de T;
3     répéter
4         Ajouter o à Ω_n;
5         pour chaque variable X_i ∈ 𝕏_n faire
6             Mettre à jour p_n^i;
7         si n n'est pas une feuille alors
8             nœud n ← n.FILS(o[X_n]);
9     jusqu'à n est une feuille;
```

Mettre à jour l'ARBRE DE DÉCISION ORDONNÉ

Une fois que les statistiques ont été mises à jour en accord avec la nouvelle observation, l'étape suivante consiste à réviser la topologie de l'arbre. En effet, du fait de l'insertion de la nouvelle observation, les variables précédemment installées ne sont plus nécessairement les variables pertinentes à installer.

Cette opération est rendue plus complexe par l'existence d'un ordre global (comparativement à la version **ITI**). Étant un algorithme incrémental, **IOTI** infère un ordre global assez efficace tout en se gardant la possibilité de le remettre à jour. Pour mettre à jour l'ARBRE DE DÉCISION ORDONNÉ, **IOTI** doit donc i) trouver un (aussi bon que possible) ordre global, ii) s'assurer que cet ordre est respecté sur chaque branche et iii) assurer que la meilleure variable possible soit installée en chaque nœud de l'ARBRE DE DÉCISION ORDONNÉ. Une dernière nécessité est que, pour toute observation qui ne change pas la structure, cette opération soit aussi peu coûteuse que possible.

Pour remplir ces conditions, la stratégie que nous proposons est de vérifier variable par variable la pertinence de sa position dans l'ordre global courant. Pour décider quelle variable est la suivante dans l'ordre global, un score est attribué à chaque variable restante. Ces scores se servent des statistiques G maintenues en chaque nœud, mais agrègent uniquement les statistiques d'un sous-ensemble de nœuds spécifique : la

frontière $\mathcal{B}$. Cette frontière $\mathcal{B}$ est l'ensemble des nœuds où la variable couramment installée est actuellement remise en question. Cette frontière est initialisée au singleton qu'est la racine de l'arbre ; elle contiendra les feuilles de l'ARBRE DE DÉCISION ORDONNÉ à la fin de la mise à jour.

IOTI calcule les scores agrégés en sommant les p-valeurs obtenues en chaque nœud $n \in \mathcal{B}$ pondérées par la proportion d'observations présentes en ces nœuds ($|\Omega_n|/|\Omega|$). La variable minimisant ce score est alors choisie comme variable suivante dans l'ordre global.

Algorithme 27 : IOTI - METTRE À JOUR ARBRE DE DÉCISION ORDONNÉ : Mise à jour de la structure de l'ARBRE DE DÉCISION ORDONNÉ

Données : un ARBRE DE DÉCISION ORDONNÉ T après l'ajout d'un o (avec l'Algorithme **IOTI - AJOUTER EXEMPLE** (26))

début
  $\mathcal{B} = \{$ racine R de $T\}$; `// frontière`
  $\mathcal{F} = \mathbb{X}$; `// Variables non insérées dans la mise à jour de l'ordre global`
  répéter
    pour chaque *variable* $X_i \in \mathcal{F}$ **faire**
      $p^i = \sum_{n \in \mathcal{B}} \frac{\omega_n}{|\Omega|} \cdot p_n^i$;
    $X_{select} \leftarrow \arg\min_{X_i \in \mathcal{F}} p^i$;
    $\mathcal{B}' \leftarrow \mathcal{B}$;
    pour chaque $n \in \mathcal{B}$ **faire**
      si $p_n^{select} \leq \tau_1$ **alors**
        TRANSPOSER(T, n, X_{select});
        $\mathcal{B}' \leftarrow \mathcal{B}' \setminus \{n\} \cup \bigcup_{x_{select} \in \mathcal{D}om\{X_{select}\}} n.\textbf{FILS}(x_{select})$
    $\mathcal{B} \leftarrow \mathcal{B}'$;
    $\mathcal{F} \leftarrow \mathcal{F} \setminus \{X_{select}\}$;
  jusqu'à $\mathcal{F} = \emptyset$ *or* $\nexists X_i \in \mathcal{F}$ *et* $n \in \mathcal{B}$ *t.q.* $p_n^i \leq \tau_1$ *(X_i peut être installée en n)*;

Suite à cette sélection, tous les nœuds de la frontière sont visités un-à-un. Pour chaque nœud de la frontière, la variable choisie est installée en ce nœud si la p-valeur est au-deçà du seuil fixé τ_1. Cette installation se fait de la même manière que dans l'algorithme **ITI**. Si la variable est installée, alors le nœud est retiré de la frontière, remplacé par tous ses fils. Notons qu'une fois cette itération terminée, tout nœud de la frontière (présent ou à venir) ne pourra plus choisir d'installer cette variable, ceci afin d'éviter toute violation de la contrainte d'ordonnancement.

Le critère d'arrêt de cet Algorithme **IMDDI - METTRE À JOUR ODT** (27) doit prendre en compte deux possibilités : ou bien toutes les variables de $\mathbb{X}$ ont été insérées dans l'ordre global ; ou bien aucune des variables restantes ne peut être installée en aucun des nœuds de la frontière courante (c.-à-d. toutes les p-valeurs sont au-delà du seuil). Quand l'algorithme se termine, la frontière contient toutes les feuilles de l'ARBRE DE DÉCISION ORDONNÉ. Par conséquent, tout nœud de la frontière qui n'est pas une feuille à ce moment-là doit être élagué.

Il est simple de vérifier que si la nouvelle observation ne change pas la structure de l'ARBRE DE DÉCISION ORDONNÉ, les seuls calculs effectués par l'Algorithme **IMDDI - METTRE À JOUR ODT** (27) sont les estimations de poids en chaque nœud de l'arbre.

6.1.3 De IOTI à IMDDI : réduction d'un ARBRE DE DÉCISION ORDONNÉ en RGFOR

Une fois l'ARBRE DE DÉCISION ORDONNÉ généré, il faut le réduire en une RGFOR. Dans cette étape, l'algorithme d'apprentissage va donc fusionner les sous-parties isomorphes du graphe. Cette fusion se fait à l'aide de l'Algorithme **RÉDUIRE** (1) polynomial en la taille de l'arbre à réduire si cet arbre est ordonné.

Pour rappel, cet algorithme commence donc par regrouper ensemble les feuilles qui ont des distributions de probabilités similaires. Puis, pour chaque variable X_i, en remontant dans l'ordre global, et pour chaque nœud n lié à X_i, deux vérifications sont faites :

- si $\exists n'$ lié aussi à X_i et tel que $\forall x_i \in Dom(X_i)$, $n.\textbf{FILS}(x_i) = n'.\textbf{FILS}(x_i)$ alors n et n' sont fusionnés.
- si $\forall$ modalité $x_i^a, x_i^b \in Dom(X_i)$, $n.\textbf{FILS}(x_i^a) = n.\textbf{FILS}(x_i^b)$ alors n est un nœud redondant et est remplacé par des arcs reliant ses parents à son unique fils.

La première phase de cet algorithme, qui consiste à agréger les feuilles semblables ensemble doit être analysée plus spécifiquement. Si les feuilles des RGFORs étaient des valeurs scalaires, fusionner ces feuilles serait simple. Toutefois, pour des distributions de probabilités conditionnelles, chaque feuille est une distribution sur la variable Y. Il nous faut donc un test pour déterminer si, pour tout couple de feuilles, les probabilités de distribution sont suffisamment similaires pour que les deux feuilles soient fusionnées.

Ici encore nous utilisons la statistique G, non pour un test de dépendance forte mais pour un test d'ajustement. Soit donc u et v les deux feuilles que nous envisageons de fusionner, soient $\omega_u^{(y)}$ (resp. $\omega_v^{(y)}$) le nombre d'observations en u (resp. v) pour la modalité $y \in \mathcal{D}om\{Y\}$ et $\omega_u = |\Omega_u|$ (resp. $\omega_v = |\Omega_v|$) le nombre total d'observations en

u (resp. v). Agréger u et v produirait une nouvelle feuille w. Il vient que $\forall y \in \mathcal{D}om\{Y\}$, $\omega_w^{(y)} = \omega_u^{(y)} + \omega_v^{(y)}$ et que $\omega_w = \omega_u + \omega_v$.

Pour déterminer si oui ou non u et v devraient être regroupées en w, nous calculons une statistique G pour les deux feuilles.

$$\forall l \in \{u, v\}, G_l = 2 \cdot \sum_y \omega_l^{(y)} \ln \frac{\omega_l^{(y)}}{e_l^{(y)}} \tag{6.2}$$

où $e_l^{(y)} = \omega_w^{(y)} \frac{\omega_l}{\omega_w}$ [6]. Ces statistiques mesurent non pas combien les deux distributions en u et v sont similaires, mais combien elles sont chacune similaires à la distribution w que nous obtiendrions si nous prenions la décision de les regrouper.

Là encore, nous avons recours aux p-valeurs p_u et p_v associées à ces statistiques G, le degré de liberté étant $|\mathcal{D}om\{Y\}| - 1$. Ainsi, si p_u et p_v sont supérieures [7] à un certain seuil τ_2 [8], alors leurs distributions de probabilité sont suffisamment similaires à la distribution de probabilité en w. Il est alors pertinent de remplacer u et v par w.

Algorithme 28 : IMDDI - RÉDUIRE ARBRE DE DÉCISION ORDONNÉ : Fusionner les sous-graphe isomorphes

Données : un ARBRE DE DÉCISION ORDONNÉ T

```
début
    répéter
        (u*, v*) = arg max_{(u,v)∈L²} (min(p_u, p_v));
        si min(p_{u*}, p_{v*}) ≥ τ2 alors
            Fusionner u* et v*;
    jusqu'à ∄(u, v) ∈ L² t.q. min(p_{u*}, p_{v*}) ≥ τ2;
    pour chaque X_i ∈ 𝕏, en remontant dans l'ordre global faire
        pour chaque paire de nœuds internes n, n' ayant X_i comme variable installée faire
            si ∀x_i ∈ Dom(X_i), n.FILS(x_i) = n'.FILS(x_i) alors
                n_{X_i} et n'_{X_i} sont fusionnés ;
        pour chaque n ayant X_i comme variable installée faire
            si ∀x_i^a, x_i^b ∈ Dom(X_i), n.FILS(x_i^a) = n.FILS(x_i^b) alors
                Remplacer n par des arcs allant de ses parents à son unique fils.
```

6. notons que nous devons réduire proportionnellement les quantités $\omega_w^{(y)}$ pour être comparables aux quantités $\omega_l^{(y)}$.

7. Cette fois-ci, nous travaillons sur la similitude de deux distributions. Par conséquent, plus deux distributions sont similaires, plus la statistique G associée est faible. La p-valeur se rapproche alors de l'unité.

8. En pratique ce seuil est fixé à 0,9.

L'algorithme de fusion des feuilles de l'ARBRE DE DÉCISION ORDONNÉ fonctionne de manière gloutonne. Tout d'abord, un couple de p-valeurs (p_u, p_v) est calculé pour chaque couple possible de feuilles (u, v). Il est facile de trouver le meilleur candidat $(u^\star, v^\star)$ à la fusion (par exemple à l'aide d'un tas). Si $p_{u^\star}$ et $p_{v^\star}$ sont l'une et l'autre plus grandes que le seuil τ_2 donné, alors ces deux nœuds sont groupés en un nœud w. Ensuite, pour tout couple (w, l), où l est une autre feuille de l'ARBRE DE DÉCISION ORDONNÉ, les p-valeurs sont calculées. Le processus reprend alors à partir de l'étape de sélection d'une paire de nœuds ; le critère d'arrêt est l'absence de fusion lors d'une itération.

L'Algorithme **IMDDI - RÉDUIRE ARBRE DE DÉCISION ORDONNÉ** (28) présente la réduction complète d'un ARBRE DE DÉCISION ORDONNÉ en RGFOR. Il faut noter que cet algorithme est assez gourmand en temps. Toutefois, il est utilisé assez rarement car, très régulièrement, l'ARBRE DE DÉCISION ORDONNÉ ne changera pas (comme décrit dans l'Algorithme **IMDDI** (25)) ou l'algorithme changera marginalement la structure, laissant l'ordre global inchangé. Les expérimentations confirment ce comportement attendu.

6.1.4 Étude de la complexité d'IMDDI

Étant un algorithme d'apprentissage incrémental composé de plusieurs sous-algorithmes, **IMDDI** a une complexité globale difficile à évaluer. Toutefois quelques propriétés peuvent être analysées. Avec Y la variable d'intérêt, et $\mathbb{X}$ l'ensemble des variables expliquant Y, **IMDDI** essaye d'apprendre $P(Y|\mathbb{X})$ sous la forme d'une RGFOR en utilisant un ARBRE DE DÉCISION ORDONNÉ T via **IOTI**.

Propriété 11 (Ajouter une observation – Algorithme **IOTI - AJOUTER EXEMPLE** (26))**.** *La complexité pour prendre en compte une nouvelle observation o dans l'*ARBRE DE DÉCISION ORDONNÉ *est en $\mathcal{O}(|\mathbb{X}|^2 \times m^2)$, où m est le nombre de modalités de la variable de plus grand domaine.*

Démonstration. Pour ajouter une observation o à l'ARBRE DE DÉCISION ORDONNÉ, l'algorithme effectuera une recherche profondeur d'abord jusqu'à atteindre une feuille. La hauteur de l'arbre est en $\mathcal{O}(|\mathbb{X}|)$ puisque chaque variable n'apparaît qu'au plus une fois seulement sur chaque chemin liant la racine à une feuille. Toutefois, durant l'ajout de o, en chaque nœud du chemin, l'algorithme met à jour la statistique G pour chaque variable de $\mathbb{X}_n$ avec $|\mathbb{X}_n| \leq |\mathbb{X}|$. Le calcul de cette statistique requière d'énumérer une à une les modalités de chaque variable de $\mathbb{X}_n$ et de Y. La mise à jour des statistiques G est alors en $\mathcal{O}(|\mathbb{X}| \times m^2)$. D'où une complexité en $\mathcal{O}(|\mathbb{X}| \times |\mathbb{X}| \times m^2)$. □

Le comportement d'**IOTI** dépend fortement des informations apportées par l'observation o : il peut arriver qu'ajouter o à l'arbre T n'induise pas de changement structurel de T. Toutefois, certaines insertions d'observation o vont impliquer des modifications. La complexité sera alors différente. Il faut noter que même si la modalité d'une variable v dans o n'a jamais été observée, la complexité donnée ci-dessus ne change pas.

Propriété 12 (Mettre à jour T sans modification – Algorithme **IOTI - METTRE À JOUR ARBRE DE DÉCISION ORDONNÉ** (27))**.** *La complexité pour mettre à jour l'arbre sans changement structurel est en* $\mathcal{O}(|T| \odot |\mathbb{X}|)$.

Démonstration. Puisque aucun changement structurel ne se produira sur la frontière de nœud à chaque itération, chaque nœud n de l'ARBRE DE DÉCISION ORDONNÉ T participera une fois seulement aux calculs des scores agrégés aux frontières. Le nœud n fera alors un calcul par variable de $\mathbb{X}_n$ avec (comme ci-dessus) $|\mathbb{X}_n| \leq |\mathbb{X}|$. □

Propriété 13 (Installer une nouvelle variable en un nœud – Algorithme **IOTI - METTRE À JOUR ARBRE DE DÉCISION ORDONNÉ** (27))**.** *La complexité d'installer une nouvelle variable* v *en un nœud* n *donné (en supposant que ses noeuds fils soient associés à* v*) est en* $\mathcal{O}(|\Omega_n| + |\mathbb{X}| \times m^2)$.

Démonstration. L'installation d'une nouvelle variable v en un nœud n requière de créer un nouvel ensemble de $(Dom(V))$ fils au nœud n. Soit $n.\textbf{FILS}(i)$ le fils pour la i-ème modalité que peut assumer la nouvelle variable v installée en n. Ce fils $n.\textbf{FILS}(i)$ inclut maintenant toutes les observations d'Ω_n où v est égale à sa i-ème modalité. Pour trouver ces observations, il faut parcourir Ω_n.

Il faut aussi calculer ($\leq |\mathbb{X}|$) p-valeurs pour chaque nœud fils. □

Notons qu'il n'est pas nécessaire de subdiviser la base Ω_n en tout nœud n du graphe. En effet, il n'y a besoin que des tables de contingence pour calculer les statistiques G. Ainsi, celles du nœud n s'obtiennent directement puisqu'elles restent inchangées (elles ne dépendent pas de la variable installée). De plus les tables des nœuds fils s'obtiennent facilement en sommant les tables des nœuds petits-fils. Seule la subdivision de nœuds feuilles implique qu'il est nécessaire de parcourir les bases d'observations associées. En effet, par construction, ces nœuds feuilles n'ont pas de petits-fils ; il faut donc parcourir la base de donnée pour construire les tables de contingence des fils (un seul parcours suffit alors).

Une autre partie importante d'**IMDDI** est la construction de la RGFOR depuis l'ARBRE DE DÉCISION ORDONNÉ.

Propriété 14 (Réduire T en une RGFOR – Algorithme **IMDDI - RÉDUIRE ARBRE DE DÉCISION ORDONNÉ** (28))**.** *La complexité de la réduction de l'*ARBRE DE DÉCISION ORDONNÉ *est en* $\mathcal{O}(|T|^2)$.

Démonstration. Comme Bryant [1986] le démontre, la fusion d'un ARBRE DE DÉCISION ORDONNÉ où les feuilles ont des valeurs exactes est en $\mathcal{O}(|T|log|T|)$. Toutefois **IMDDI** agrège des distributions de probabilités similaires sur les feuilles et non des valeurs exactes. Ce regroupement doit donc être découpé en deux parties. Une partie est la fusion des nœuds internes. Cette partie reste en $\mathcal{O}(|T|log|T|)$. La fusion des feuilles est différente dans **IMDDI**. En effet, initialement, il faut évaluer deux p-valeurs pour chaque paire de feuilles ($2 \times m \times |L|^2$ valeurs sont alors calculées, où L est l'ensemble des noeuds feuilles de T). Ensuite, après chaque regroupement, il faut estimer les p-valeurs pour la feuille nouvellement créée et ce avec toutes les autres feuilles restantes ($m \times |L|$ valeurs sont alors calculées). Toutefois il n'y aura au plus que $|L|$ fusions (on ne peut pas faire plus de fusions qu'il n'existe de feuilles). Ainsi, cette étape calcule au plus $2 \times m \times |L|^2 + m \times |L| * |L|$ valeurs. Tout compte fait, la complexité de cette phase est en $\mathcal{O}(m \times |L|^2)$. En considérant que $|L| < |T|$, la complexité totale de cette partie est en $\mathcal{O}(|T|log|T|) + \mathcal{O}(m \times |T|^2) = \mathcal{O}(|T|^2)$ [9]. □

Cette analyse en complexité montre que **IMDDI** est un algorithme dédié à l'apprentissage en-ligne : la taille de la base d'observations Ω n'entre jamais en compte dans les complexités sus-évoquées. Seule la complexité d'un sous-algorithme dépend d'une sous-partie Ω_n installée en ce nœud. Notons aussi que les deux sous-algorithmes les plus coûteux (installer et fusionner) ne se déroulent pas nécessairement à chaque fois qu'une nouvelle observation o est prise en compte. Comme la section suivante le démontrera, la réévaluation de la RGFOR ne se produit effectivement que marginalement.

6.1.5 Découverte de modalités

Une caractéristique intéressante de cet algorithme (de même que de l'algorithme **ITI**) est qu'aucune hypothèse n'est à faire initialement sur la taille du domaine des variables. Ces domaines sont ajustables dynamiquement en cours d'exécution de l'apprentissage. Ainsi, l'algorithme est initialisé avec toutes les variables ayant un domaine de taille 2. Puis, au fur et à mesure des réceptions d'observations, les variables ayant un domaine plus grand sont corrigées, ainsi que les statistiques associées à ces variables en tout nœud du graphe.

9. Largement surévaluée puisque $m \times |L| \ll T$.

Un bémol est que, pour une variable donnée, il est nécessaire de remettre alors en question tout nœud actuellement associé à cette variable. En effet, du fait de l'apparition d'une nouvelle modalité, les p-valeurs associées à la variable changent. Ces p-valeurs ne sont alors plus nécessairement les meilleures. Il est alors nécessaire de sélectionner une nouvelle meilleure variable en ces nœuds. Toutefois, cette vérification est automatiquement faite lors de la phase de mise à jour. Néanmoins, si la variable modifiée est la variable Y dont l'algorithme apprend la distribution, tous les nœuds du graphe doivent être remis en question ; toutes les mesures se basent en effet en partie sur cette variable.

6.1.6 Apprentissage en ligne de la fonction *Récompense*

A priori, l'algorithme **IMDDI** n'est pas applicable pour l'apprentissage de fonctions *Récompense*. En effet, une RGFOR représentant une distribution de probabilités conditionnelles et une RGFOR représentant une fonction à valeurs dans $\mathbb{R}$ diffèrent principalement par l'objet mathématique manipulé aux feuilles : dans un cas ce sont des distributions de probabilités, dans l'autre des valeurs réelles. Dans un cas, nous manipulons un domaine discret, et dans l'autre un domaine continu. De fait, le principal problème dans l'adaptation de **IMDDI** à des apprentissages de fonctions à valeurs réelles est que les mesures utilisées (la statistique G pour des tests d'ajustement et de dépendance) ne sont pas applicables.

L'approche moindres carrés

Dans **SPITI**, Degris [2007] utilise l'erreur au sens des moindres carrés [Breiman et al., 1984b] comme mesure de sélection [10]. Le calcul de cette mesure se fait en plusieurs étapes :

1) Tout d'abord, en définissant m_n comme la moyenne des valeurs r_n observées dans la base de données Ω_n du nœud n, une première grandeur à calculer car nécessaire pour la sélection de variables est la dispersion des données dans la base initiale :

$$\mathcal{D}_n = \frac{1}{\omega_n} \cdot \sum_{r_n \in \Omega_n} (r_n - m_n)^2$$

2) Ensuite, la sélection de la variable X_i pour le noeud n aboutira à la création de $|X_i|$ bases de données Ω_{x_i}, toutes extraites de Ω_n. Soit alors m_{x_i} la moyenne des valeurs

10. Degris [2007] suggère que la moindre déviation absolue [Breiman et al., 1984a] serait aussi employable bien que plus coûteuse en calcul.

r observées dans la sous-base Ω_{x_i}. La dispersion des données dans cette sous-base est alors :

$$\mathcal{D}_{x_i} = \frac{1}{\omega_{x_i}} \cdot \sum_{r_{x_i} \in \Omega_{x_i}} (r_{x_i} - m_{x_i})^2$$

La dispersion totale $\mathcal{D}_{X_i}$ résultant de la séparation de la base Ω suivant les modalités de la variable X_i est alors :

$$\mathcal{D}_{X_i} = \sum_{x_i \in \mathcal{D}om\{X_i\}} \frac{\omega_{x_i}}{\omega_n} \mathcal{D}_{x_i}$$

3) Le choix de variable pertinent s'oriente sur la variable qui minimise la dispersion dans ses sous-bases par rapport à la base originelle :

$$V = \arg\max_{X_i \in \mathbb{X}} (\mathcal{D}_n - \mathcal{D}_{X_i})$$

Le problème de cette méthode est qu'il n'existe pas de critère de pré-élagage. En conséquence, l'algorithme fait d'abord croître les arbres jusqu'à atteindre des feuilles pures, ou jusqu'à ce que toutes les variables soient installées. Une phase de post-élagage est donc à effectuer. Le danger est alors de se retrouver avec des arbres de taille exponentielle.

De plus, lors de la fusion des feuilles pour inférer la RGFOR associée, bien qu'une méthode des moindres carrés pourrait être mise en place, l'inexistence d'un critère de pré-élagage résulte en une fusion de toutes les feuilles suivie d'un retour sur toutes les fusions faites pour juger de leur pertinence.

Pour ces raisons, nous n'avons pas cherché à utiliser cette mesure de sélection.

L'approche "structurée"

Dans notre approche, nous nous sommes servi de l'hypothèse de structure sur les données. Une conséquence directe est que la fonction *Récompense* n'est plus définie sur le domaine continu $\mathbb{R}$ mais sur un ensemble fini et discret de valeurs de $\mathbb{R}$. Dès lors, il est possible de définir une variable aléatoire R de domaine ce sous-ensemble. Nous pouvons alors manipuler cette variable comme toute autre variable aléatoire Y.

Grâce à cette astuce, l'apprentissage de la RGFOR représentant la fonction *Récompense* est le même que celui d'une fonction *Transition*. Plus spécifiquement, la sélection est toujours la même : choisir la meilleure variable pour l'ordre global à partir de l'agrégation de statistiques G pour les nœuds de la frontière, puis vérifier pour chaque nœud

de la frontière s'il désire l'installer. De même, la fusion des feuilles s'opère similairement. Si deux feuilles s'ajustent très bien à la feuille obtenue par fusion d'après leur statistique G, cette feuille résultante remplace les deux feuilles.

Reste toutefois que les feuilles apprises sont des distributions de probabilités et non des valeurs réelles. Or pour l'algorithme de planification, il nous faut des feuilles à valeurs réelles. Néanmoins, chaque modalité de R est une récompense obtenable et donc une valeur réelle : R est une variable aléatoire réelle. Donc il existe une espérance mathématique associée à cette variable ; pour chaque feuille l, cette espérance est alors la récompense moyenne m_l de la base Ω_l associée.

6.2 Validation Expérimentale d'IMDDI

Pour évaluer les performances de **IMDDI**, nous l'avons comparé avec **ITI**, qui est l'algorithme d'apprentissage de l'instance **SPITI** de l'architecture **SDYNA** basé sur des ARBRES DE DÉCISION. Toutefois, **ITI** est un algorithme qui produit des ARBRES DE DÉCISION non-ordonnés. Nous avons donc aussi comparé **IMDDI** à la version *augmentée* de **ITI** présentée plus haut : **ITI**+DD. Pour rappel, dans cette version, après qu'**ITI** ait mis à jour l'arbre appris, nous réordonnons une copie de cet arbre. La même heuristique que dans **IMDDI** est utilisée pour réordonner l'arbre afin de le réduire en RGFOR.

6.2.1 Méthodologie

Ces trois algorithmes ont été implémentés en PYTHON. Nous les avons testés en leur faisant apprendre des représentations factorisées de distributions $P(Y|\mathbb{X})$. Mais pour réaliser cet apprentissage, il nous a fallu des flux de données (des bases d'apprentissage) suivant des distributions $P(Y|\mathbb{X})$ données.

Dans un premier temps, il faut donc des distributions $P(Y|\mathbb{X})$. Dans les expérimentations qui suivent, trois approches ont été suivies pour générer aléatoirement ces fonctions $P(Y|\mathbb{X})$:

ARBRE DE DÉCISION : la fonction $P(Y|\mathbb{X})$ est générée aléatoirement directement sous la forme d'un ARBRE DE DÉCISION, les nœuds internes étant associés aux variables de $\mathbb{X}$ et les feuilles sont des distributions sur la variable Y. La tâche d'apprentissage est donc de retrouver cet arbre.

RGFOR : De même que pour l'approche ARBRE DE DÉCISION, cette approche génère aléatoirement une RGFOR. La tâche d'apprentissage est donc de retrouver une RGFOR l'approchant.

Réseau Bayésien : Toutes les variables de $\mathbb{X}$ et Y sont reliées dans un réseau bayésien généré aléatoirement. Cette approche perturbe la tâche d'apprentissage car il n'est pas certain que la distribution de Y par rapport à ses parents possède alors des indépendances contextuelles : l'apprentissage aboutira peut-être à un arbre complet.

Toutes ces approches induisent des dépendances conditionnelles ; Y ne dépend pas nécessairement de toutes les variables de $\mathbb{X}$. Il faut noter que seules les fonctions $P(Y|\mathbb{X})$ suivant les deux premiers modèles présenteront des dépendances contextuelles explicites.

Pour chaque modèle de fonction, 20 instances sont aléatoirement générées (soient 20 BNs, 20 ARBRES DE DÉCISION, 20 RGFORs). Avant chaque création de fonction, l'ensemble des variables $\mathbb{X}$ est réinitialisé aléatoirement. Le nombre de modalités de chaque variable de $\mathbb{X}$ varie d'une fonction à l'autre, la plage de variation étant entre 2 et 10. Enfin, les fonctions générées n'utilisent qu'une sous-partie de $\mathbb{X}$: les variables restantes agissent comme du bruit dans l'apprentissage.

Pour obtenir des observations à partir de ces fonctions, la troisième approche est la plus simple puisque le réseau bayésien propose la loi jointe de l'ensemble des variables. Il suffit donc de générer des observations en suivant cette loi jointe. Pour les autres types de fonctions, ils nous faut d'abord instancier $\mathbb{X}$. Une modalité est donc tirée aléatoirement pour chaque variable de $\mathbb{X}$ et ce de manière uniforme et indépendante. Ensuite, une modalité pour Y est tirée aléatoirement suivant la distribution de $P(Y|\mathbb{X})$ correspondant à l'instanciation obtenue de $\mathbb{X}$. Pour chaque fonction créée, deux bases de 20 000 et 10 000 observations sont ainsi constituées.

La première base est utilisée pour l'apprentissage du modèle alors que la seconde est utilisée pour la validation : il s'agit principalement de comparer les vraisemblances des modèles appris. Cette vraisemblance est calculée en prenant, pour chaque observation de la base de test, la probabilité de la modalité assumée par Y sachant l'instanciation des variables de $\mathbb{X}$. Les résultats suivants sont obtenus en moyennant sur les 20 modèles générés pour chaque approche.

6.2.2 Qualité des modèles appris

Le premier critère de comparaison proposé est la log-vraisemblance du modèle appris par rapport à la base de validation. Cette vraisemblance nous assure qu'**IMDDI** ne dégrade pas trop la qualité de la distribution de probabilité apprise par rapport à **ITI** et **ITI**+DD. Les résultats de la Table 6.1 montrent qu'en moyenne, il n'y a pas de perte

TABLE 6.1 – *Comparaison en taille et vraisemblance (moyenne ± écart-type) entre* **IMDDI** *et* **ITI** *et* **ITI**+DD

	IMDDI vs **ITI**	
	Taille	Log vraisemblance
RGFORs	63.34% ±9,51%	100.96% ±1,53%
ARBRES DE DÉCISION	69.44% ±15.44%	101.93% ±1.89%
BNs	60.51% ±8.73%	100.69% ±1.25%
	IMDDI vs **ITI**+DD	
	Taille	Log vraisemblance
RGFORs	101.15% ±3,46%	100.01% ±0.04%
ARBRES DE DÉCISION	101.52% ±15.22%	100.03% ±0.09%
BNs	105.53% ±5.83%	99.94% ±0.39%

de la qualité des solutions obtenues. Ainsi, la stratégie **IMDDI** est aussi pertinente que **ITI** ou **ITI**+DD de ce point de vue.

Le second paramètre de comparaison est la taille (en nombre de nœuds) des RGFORs et ARBRES DE DÉCISION appris. En effet, il s'agit de montrer que **IMDDI** fournit des représentations plus compactes que les arbres qu'**ITI** génère. Ce critère est important car les complexités des algorithmes de calcul de *Politique* optimale dépendent crucialement de ce paramètre. Les résultats de la Table 6.1 montrent qu'en moyenne les modèles appris par **IMDDI** sont en effet plus compacts que les arbre d'**ITI**. Avec une diminution de 40% du nombre de nœuds pour une vraisemblance quasiment identique, ces expériences valident l'intérêt de la modélisation de domaines structurés en RGFORs plutôt qu'en ARBRES DE DÉCISION.

Néanmoins, les résultats très proches de **IMDDI** et **ITI**+DD suivant les 2 critères précédents ne permettent pas de montrer la pertinence de notre algorithme d'apprentissage incrémental **IMDDI** par rapport à la méthode plus basique de construction **ITI**+DD.

6.2.3 Efficacité temporelle

Le dernier critère sur lequel **IMDDI** et **ITI**+DD ont été confrontés est le temps. Plus précisément, il s'agit de comparer les temps de prise en compte de nouvelles observations. Cette étude a été faite en deux temps.

Dans une première étape, nous avons observé le temps de prise en compte d'une nouvelle observation lorsque celle-ci n'induit pas de modification de la structure finale. La Figure 6.2 montre clairement que le temps passé à réordonner l'arbre dans la stratégie **ITI**+DD le rend complètement inefficace. La phase de ré-ordonnancement de l'arbre

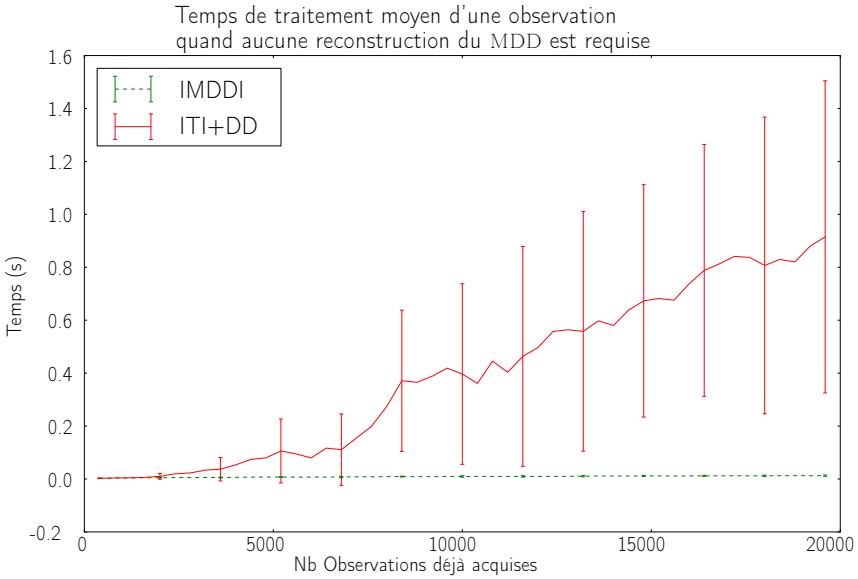


FIGURE 6.2 – *Temps de prise en compte d'une nouvelle observation par* **IMDDI** *et* **ITI**+DD. *Avant de créer ce graphe, les itérations où l'ajout de l'observation a conduit à une modification de la* RGFOR *ont été écartées. Pour les deux algorithmes, ces temps sont mesurés lors des apprentissages des bases générées à partir de* BN*s. On trace alors les moyennes et écart-types des temps de prises en compte des* i^{emes} *observations durant les* 20 *apprentissages de chaque algorithme.*

coûte en effet de plus en plus de temps à **ITI**+DD au fur et à mesure que sa taille augmente.

Comme mentionné dans la section précédente, la phase de réduction a la plus forte complexité. Néanmoins,en moyenne, seulement 15% des ajouts d'observations implique une reconstruction de la RGFOR pour **IMDDI** et **ITI**+DD durant l'apprentissage de bases générées avec les BNs. Cette moyenne tombe à 7% quand l'apprentissage est fait sur des bases obtenues à partir d'ARBRES DE DÉCISION et d'RGFORs.

On pourrait objecter que la révision du RGFORs nous dépouillerait ainsi de l'avantage gagné à éviter les ré-ordonnancements de **ITI**+DD. La Figure 6.3 décrit l'écart relatif entre les temps totaux pris par **IMDDI** et **ITI**+DD pour acquérir une nouvelle observation. La courbe centrale montre qu'en moyenne, en dépit de la reconstruction de la RGFORs, l'algorithme **IMDDI** est 1,5 fois plus rapide que l'algorithme **ITI**+DD.

La conclusion de cette première phase expérimentale est donc double : d'une part, la modélisation en RGFOR permet effectivement une compacité plus importante du modèle appris, gain qui aura un impact fort au moment de la recherche de stratégie optimale. D'autre part, l'algorithme d'apprentissage incrémental direct **IMDDI** se révèle beaucoup plus rapide qu'un algorithme basé sur un apprentissage incrémental d'arbre suivi d'une induction de la RGFOR à partir de cet arbre.

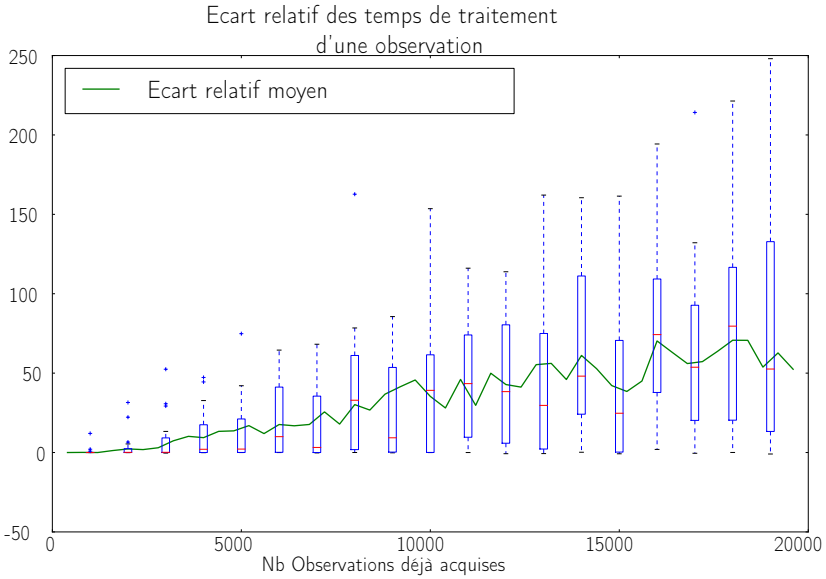


FIGURE 6.3 – *Évolution de l'écart relatif* $\frac{T_{ITI+DD}-T_{IMDDI}}{T_{IMDDI}}$ *(où* T_X *est le temps total pour l'algorithme X pour intégrer la nouvelle observation). Pour établir ce graphe, nous nous sommes servis des modèles appris à partir des bases* BNS*. Les extrémités hautes et basses des boites montrent le premier et le dernier quartile. La ligne interne est la médiane. Les moustaches sont les extrêmes.*

6.3 SPIMDDI : Apprentissage et Planification exploitant des RGFORs dans un cadre SDYNA

L'algorithme de planification **SPUMDD** (16) et l'algorithme d'apprentissage incrémental **IMDDI** (25) sont conçus pour pouvoir s'intégrer naturellement dans une instance de l'architecture **SDYNA**. Il est aisé de mixer ces deux algorithmes itératifs pour obtenir un algorithme d'apprentissage par renforcement qui manipule exclusivement des RGFORS.

Ainsi, dans cette nouvelle instance que nous nommerons **SPIMDDI**, plusieurs processus utilisant l'algorithme **IMDDI** tournent en parallèle pour apprendre les différentes distributions de probabilités de *Transition*– une par variable et par action. De même, plusieurs processus apprennent les fonctions *Récompense* associées à chaque action.

L'algorithme **SPUMDD** exploite ensuite ces RGFORS mises à jour régulièrement pour planifier une *Politique*. A l'instar de **SVI** dans **SPITI**, la condition d'arrêt de **SPUMDD** est modifiée pour que **SPUMDD** ne fasse qu'un nombre limité d'itérations. Puis une *Politique* est extraite (sous forme de RGFOR) et remplace l'actuelle *Politique*. C'est sur cette nouvelle *Politique* que se repose alors le moteur de décision pour prendre ses décisions.

6.3.1 Validation Expérimentale

Afin de pouvoir comparer notre approche au plus juste, nous avons implémenté **SPIMDDI** et **SPITI** dans la même infrastructure logicielle en C++ (basée sur la librairie aGrUM). Puis nous avons confronté **SPIMDDI** et **SPITI** sur les 3 problèmes clés COFFEE ROBOT, FACTORY et TAXI. Les deux algorithmes ont été comparés suivant 4 critères d'analyse : le nombre de couples état-action visités, la taille des modèles obtenus, la taille de la *Politique* optimale estimée et la récompense actualisée[11]. Les courbes sont obtenues en moyennant sur 20 expériences de 4000 trajectoires, une trajectoire étant composée de 25 transitions. Une réinitialisation aléatoire de l'état du système a lieu avant chaque début de trajectoire. Dans un premier temps, pour la gestion du compromis exploration-exploitation, nous nous sommes limités à une approche ϵ-**GREEDY** : il s'agit en effet ici de démontrer et valider l'intérêt d'une approche basée sur des RGFORs.

Pour les critères "Nombre de couples état-action visités" et "Récompense actualisée", nous avons aussi mesuré les performances d'un agent qui agirait toujours aléatoirement et d'un agent qui disposerait de la stratégie optimale. Par ailleurs pour les tailles des modèles et de la *Politique* optimale, nous indiquons les tailles optimales.

COFFEE ROBOT

Le premier problème sur lequel nous avons comparés **SPIMDDI** et **SPITI** est le problème COFFEE ROBOT. Pour rappel, COFFEE ROBOT est un petit problème à 64 états et 4 actions. Au total, 256 couples état-action existent donc dans ce problème. Notre banc d'essai effectuant 4000 trajectoires de 25 transitions, soient 100 000 transitions en tout, les deux algorithmes devraient théoriquement pouvoir visiter l'ensemble des couples état-action.

Ce qui est le cas (cf. Figure 6.4) : les deux algorithmes atteignent en moyenne les 256 couples visités vers la 2000ème trajectoire. Remarquons d'ailleurs que les deux algorithmes convergent similairement vers cette limite, ce qui dénote que l'algorithme ϵ-**GREEDY** se comporte similairement sur les deux algorithmes d'un point de vue gestion du compromis exploration-exploitation. De plus, **SPIMDDI** et **SPITI** convergent moins rapidement vers ce seuil que la stratégie purement aléatoire, ce qui dénote que ces deux algorithmes sont aussi en phase d'exploitation de leur *Politique* optimale estimée.

11. $R_n = r_n + \gamma \cdot R_{n-1}$

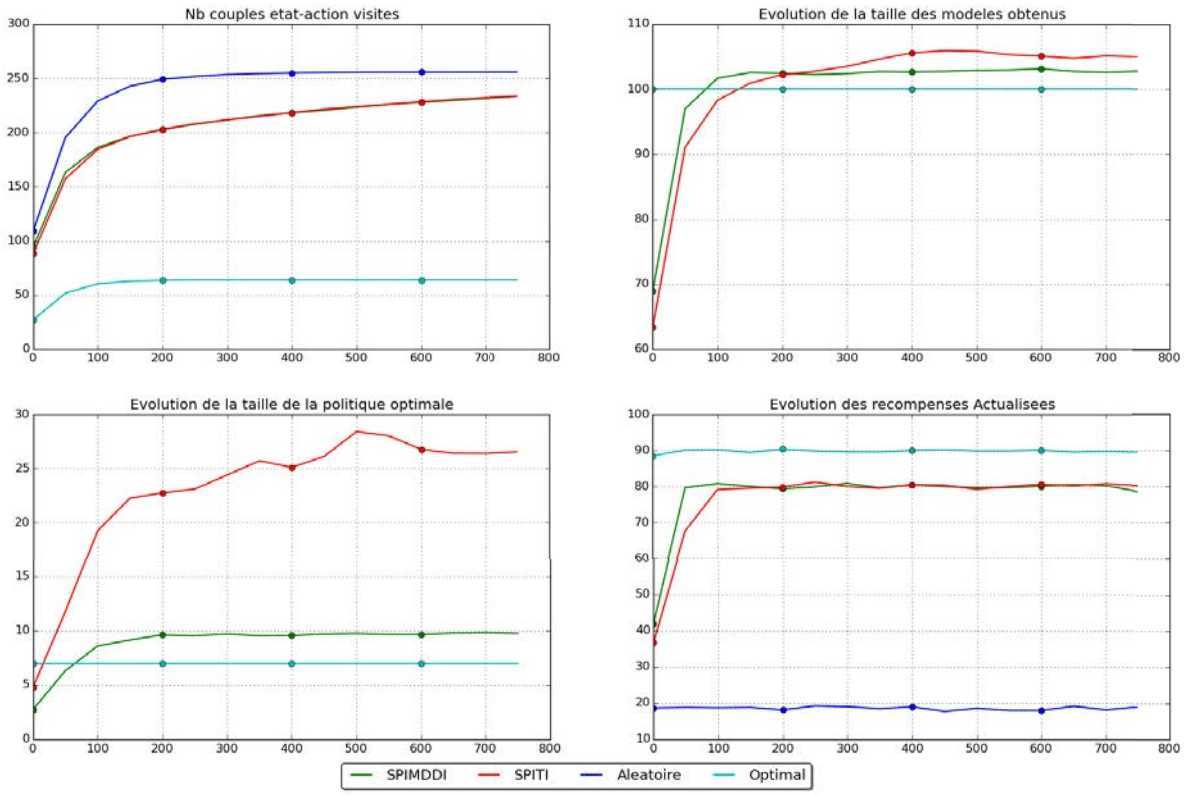


FIGURE 6.4 – *Performances de* **SPIMDDI** *(en vert) comparées à celles de* **SPITI** *(en rouge) et, s'il y a lieu, des stratégies optimale (en cyan) et aléatoire pure (en bleu indigo) sur le problème* COFFEE ROBOT. *De haut en bas, de gauche à droite, les performances étudiées sont le nombre de couples état-action visités, la taille des modèles appris, la taille de la* Politique *optimal induite, et l'évolution de la récompense actualisée.*

La similitude dans l'évolution des courbes de récompense actualisée montre que les deux algorithmes planifient une *Politique* similaire. Cette *Politique* estimée est vraisemblablement la *Politique* optimale puisque **SPIMDDI** et **SPITI** parviennent à faire pratiquement aussi bien que l'agent ayant la *Politique* optimale. La légère marge d'erreur est imputable au moteur de décision ϵ-**GREEDY** qui choisit de temps à autre une action suboptimale, donc retardant l'acquisition de récompense.

Du point de vue de la taille des modèles appris, **SPIMDDI** et **SPITI** se comportent à peu près similairement ; un léger avantage est à accorder à **SPIMDDI** qui parvient à apprendre un modèle de taille quasi optimale. La raison de cette similitude est que le problème COFFEE ROBOT est fortement arborescent. Ainsi, dans le modèle d'origine, la plupart des RGFORs sont des ARBRES DE DÉCISION dont les feuilles ont été fusionnées. Remarquons toutefois que **SPIMDDI** converge plus rapidement dénotant un pouvoir généralisateur plus fort. Sur la taille de la *Politique* optimale estimée, **SPIMDDI** montre

clairement sa supériorité avec une taille proche de celle de la *Politique* optimale.

FACTORY

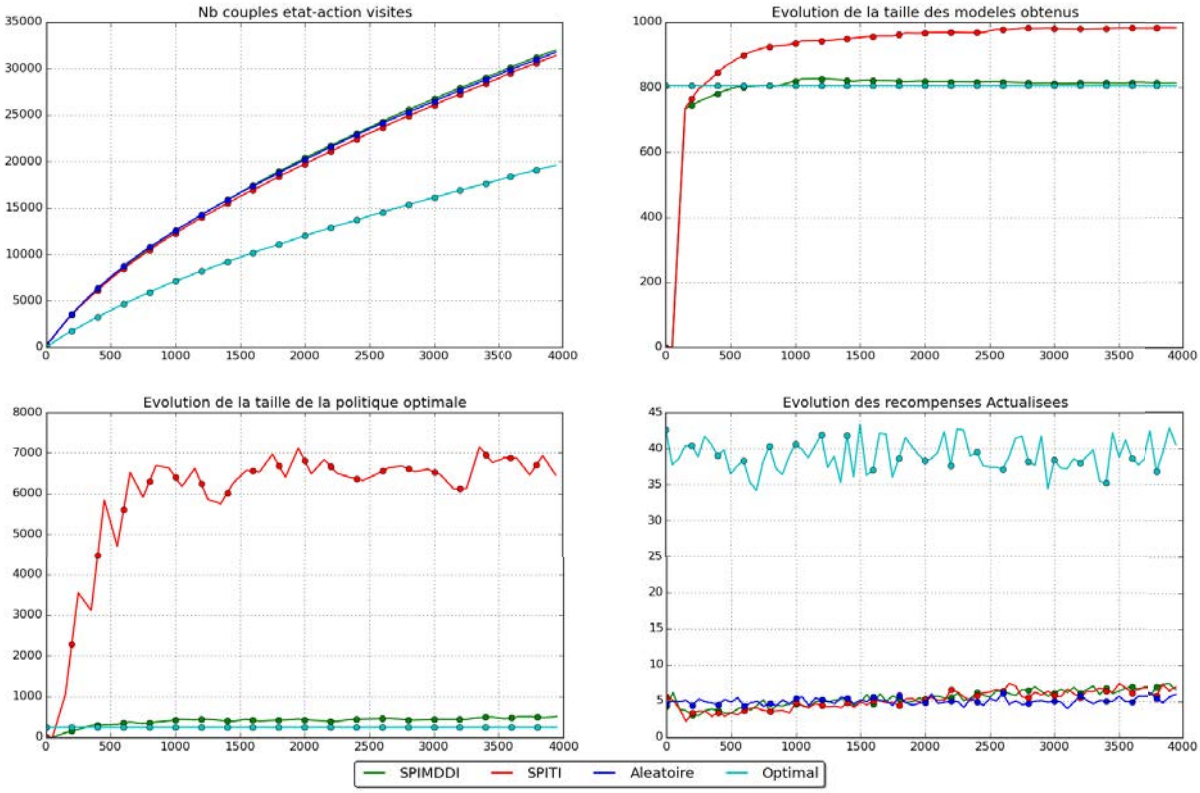


FIGURE 6.5 – *Performances de* **SPIMDDI** *(en vert) comparées à celles de* **SPITI** *(en rouge) et, s'il y a lieu, des stratégies optimale (en cyan) et aléatoire pure (en bleu indigo) sur le problème* FACTORY*. De haut en bas, de gauche à droite, les performances étudiées sont le nombre de couples état-action visités, la taille des modèles appris, la taille de la* Politique *optimal induite, et l'évolution de la récompense actualisée.*

Problème plus conséquent, FACTORY dispose de 55 296 états et de 14 actions, soient 774 144 couples état-action. Notre banc d'essai ne permet donc pas aux deux algorithmes de visiter toutes les transitions. Le graphique comparant les deux algorithmes sur le critère du nombre de couples état-action visités (cf. Figure 6.5) l'illustre d'ailleurs très bien : les deux algorithmes n'ont pas encore convergé.

Ce qui ne les empêche pas néanmoins de converger en taille des modèles appris et de la *Politique* optimale. Sur ces deux critères, **SPIMDDI** s'avère plus performant que **SPITI**. En particulier, pour les tailles des *Politiques* optimales estimées, la *Politique* estimée par **SPITI** atteint une taille 20 fois supérieur à la taille de la vraie solution optimale, tandis que **SPIMDDI** affiche une taille comparable.

Néanmoins, comme le montre l'évolution des récompenses actualisées, ces deux *Politiques* sont loin d'être des estimations fiables de la *Politique* optimale. Les récompenses actualisées stagnent en effet aux alentours de la valeur 5, avec une légère tendance à croître toutefois. De plus, si nous observons la courbe des couples état-action visités, nous pouvons voir que **SPIMDDI** et **SPITI** se comportent comme la stratégie aléatoire ; ils sont donc encore dans une phase d'exploration du système. Ce résultat démontre qu'en dépit d'une capacité à généraliser (la taille des modèles est strictement inférieure aux nombres de couples état-action visités), ces algorithmes ne sont pas efficaces si une méthode d'exploration efficace de l'espace d'états n'est pas mise en place.

TAXI

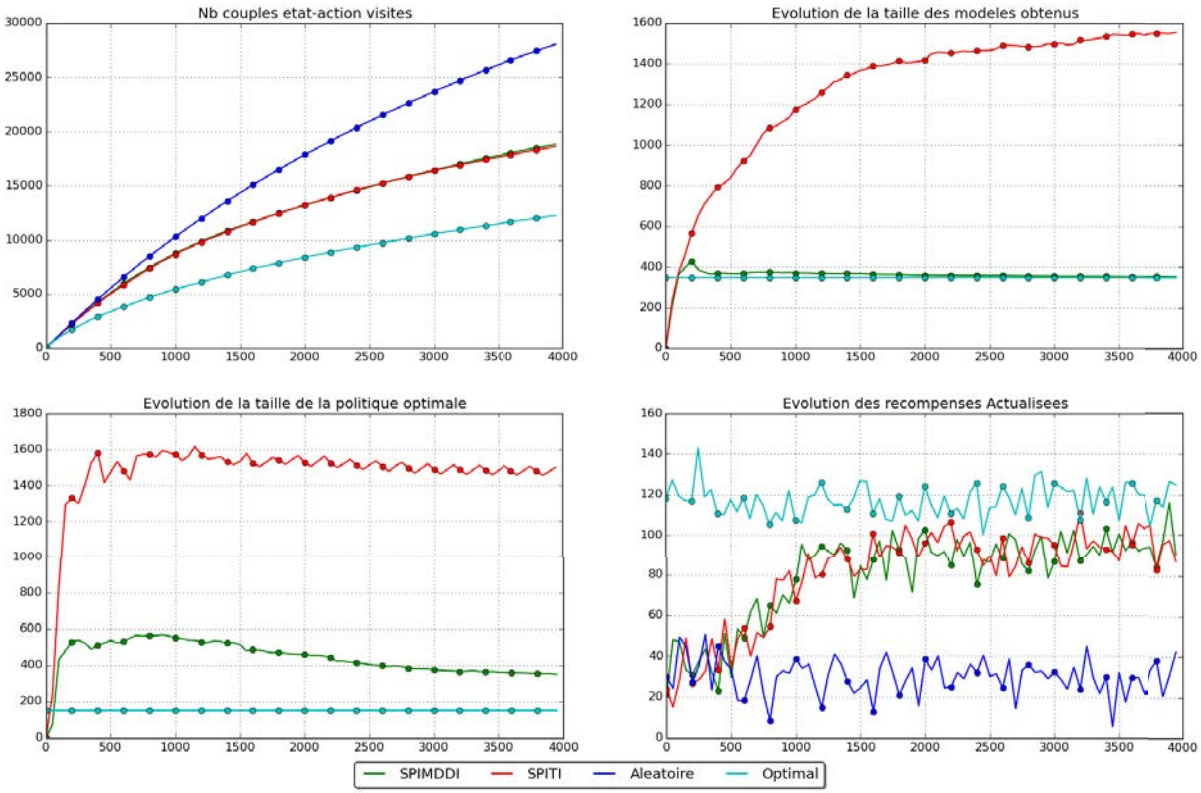


FIGURE 6.6 – *Performances de* **SPIMDDI** *(en vert) comparées à celles de* **SPITI** *(en rouge) et, s'il y a lieu, des stratégies optimale (en cyan) et aléatoire pure (en bleu indigo) sur le problème* TAXI*. De haut en bas, de gauche à droite, les performances étudiées sont le nombre de couples état-action visités, la taille des modèles appris, la taille de la* Politique *optimal induite, et l'évolution de la récompense actualisée.*

Le dernier problème, TAXI, nous permet de nous confronter à un problème purement multimodal. La particularité de ce type de problème est que chaque variable requière des statistiques en nombre suffisant pour chacune de leurs modalités afin d'avoir des tests fiables ; ce besoin conséquent de statistiques retarde davantage le moment où l'on considère comme significative chacune des variables et donc le moment où l'on envisage de l'installer dans le graphe. Il en résulte que les RGFORs sur des problèmes multivariés sont plus longues à construire et requièrent davantage d'exploration. Par conséquent, le moment où l'algorithme exploite davantage qu'il n'explore est plus tardif.

Ce problème a 7000 états et donne à l'agent le choix entre 7 actions ; le nombre de couples état-action à visiter atteint donc les 49 000. Sur notre banc d'essai, l'agent peut donc potentiellement visiter toutes les transitions. Comme le montre le net décrochement sur la courbe des nombres de couples état-action visités par rapport à la stratégie purement aléatoire, les deux algorithmes sont clairement entrés dans une phase d'exploitation.

L'analyse des résultats montre aussi que l'algorithme **SPIMDDI** se comporte beaucoup mieux d'un point de vue taille du modèle que l'algorithme **SPITI** : les structures manipulées par **SPITI** sont jusqu'à 4 fois plus grandes. D'un point de vue efficacité de la *Politique* apprise, les deux algorithmes semblent se comporter similairement. Sur la courbe des récompenses actualisées, tous deux parviennent à 10% de la stratégie optimale ; en tenant compte de l'influence de ϵ-**GREEDY**, les deux algorithmes semblent avoir trouvé une *Politique* quasi optimale.

6.3.2 Conclusion

SPIMDDI s'avère être un algorithme très intéressant. D'un point de vue apprentissage, l'algorithme **IMDDI** parvient à apprendre des structures plus petites que **ITI**, ce qui le rend utile pour la seconde partie de **SPIMDDI**, à savoir la planification. En effet, planifier avec des RGFORs est beaucoup plus rapide qu'avec des arbres, d'où un gain de temps pour continuer d'explorer-exploiter le système.

De ce point de vue là par ailleurs, **SPIMDDI** se comporte similairement à **SPITI**. Ceci dénote donc que **SPIMDDI** apprend les mêmes fonctions *Récompense* et *Transition* que **SPITI** puisque la planification est similaire dans les deux algorithmes. Notons toutefois que sur les problèmes très grands, ces deux algorithmes ne parviennent pas à avoir une connaissance suffisante du système pour avoir une stratégie optimale. Cette lacune nous a amené à rechercher un meilleur algorithme de gestion du compromis qu'ϵ-**GREEDY** dont nous présentons les premiers résultats dans le chapitre suivant.

6.4 Synthèse

Pour pouvoir effectuer de l'*apprentissage par renforcement* à partir d'une instance de **SDYNA** utilisant des RGFORs, nous avions besoin d'un algorithme d'apprentissage incrémental de RGFOR. Pour résoudre ce problème, nous avons présenté **IMDDI**. Nous avons vérifié expérimentalement que **IMDDI** apprend aussi efficacement que l'algorithme **ITI** d'apprentissage incrémental d'ARBRE DE DÉCISION. Nous avons montré que la stratégie employée par **IMDDI**, à savoir apprendre un ARBRE DE DÉCISION ORDONNÉ puis le réduire, était plus efficace qu'apprendre un arbre puis l'ordonner et le réduire (stratégie **ITI**+DD).

Nous avons ensuite vu comment intégrer **IMDDI** à une instance **SDYNA**, à savoir **SPIMDDI**. Nous avons comparé **SPIMDDI** avec **SPITI**, la version pour ARBRE DE DÉCISION de **SDYNA**. D'un point de vue efficacité d'apprentissage (exploration des couples état-action, récompense récoltée), les deux algorithmes se comportent similairement. Toutefois, **SPIMDDI** s'avère bien plus efficace pour représenter le modèle de manière compact (il atteint même le modèle compact optimal pour certains problèmes).

Chapitre 7

Amélioration et Application de SPIMDDI

Dans ce chapitre, nous présentons deux travaux de recherche que nous avons initiés au cours de cette thèse. Le premier travail s'intéresse à l'amélioration de la gestion du compromis exploration-exploitation en vue de fournir à **SPIMDDI** une garantie de convergence. Nous cherchons ainsi à adapter un des algorithmes PAC-MDP existants pour le cadre factorisé et structuré.

Le deuxième travail porte sur l'utilisation de **SPIMDDI** dans un cadre appliqué. En effet, jusqu'à présent, **SPIMDDI** a été éprouvé sur des problèmes jouets. Même si certains d'entre eux (FACTORY et TAXI) sont de très grandes tailles, les transitions et récompenses de ces problèmes sont parfaitement identifiées. Or **SPIMDDI** est conçu pour être utilisé sur des problèmes où nous ne disposons pas de ce genre de connaissances. Nous nous sommes donc intéressés à son utilisation dans la gestion d'une I.A. pour le jeu vidéo Starcraft II.

Ces deux axes, en plus d'être intéressants en soi, nous permettent de valider que notre approche et son implémentation sont aptes à être exploitées et améliorées.

7.1 Vers un algorithme PAC-FMDP

Un des principaux écueils de **SPITI**, et donc de **SPIMDDI**, est qu'il ne présente pas de garantie de convergence. Une des raisons est que le moteur de décision, l'algorithme ϵ-**GREEDY**, ne permet pas d'établir une telle preuve de convergence. Nous nous sommes donc naturellement intéresser à la création d'une heuristique s'appuyant sur l'architecture **SDYNA** et présentant une preuve de convergence. La contribution que nous présentons ici initie cette recherche et propose quelques résultats préliminaires.

7.1.1 Algorithmes PAC-FMDP

Créer une instance de **SDYNA** présentant une preuve de convergence passe en partie par l'intégration dans l'architecture d'un moteur de décision permettant d'établir une telle garantie. Un tel algorithme de gestion du compromis exploration-exploitation doit néanmoins être adapté aux représentations factorisées. Des travaux existent déjà dans la littérature pour gérer le compromis dans les représentations factorisées. Ces algorithmes définissent une nouvelle classe empruntant à la classe d'algorithme PAC-MDP (cf. Définition 5.2) la notion d'*apprentissage par renforcement* efficace tout en l'adaptant aux représentations factorisées :

Définition 7.1 (Algorithme PAC-FMDP [Strehl, 2007])**.**

Un algorithme $\mathcal{A}$ d'*apprentissage par renforcement* est défini comme étant PAC-FMDP efficace si, $\forall \epsilon > 0$ et $\delta \in [0, 1]$, la complexité algorithmique d'une itération de $\mathcal{A}$ et la complexité d'échantillon d'exploration de $\mathcal{A}$ sont inférieure à un polynôme s'exprimant à l'aide des quantités $|\mathbb{X}|$, $\max_{X_i \in \mathbb{X}} |\mathcal{D}om\{X_i\}|$, I[1], $|\mathbb{A}|$, $\frac{1}{\epsilon}$, $\frac{1}{\delta}$ et $\frac{1}{1-\gamma}$ avec une probabilité supérieur à $1 - \delta$.

Concrètement, la plupart de ces méthodes sont des transpositions des algorithmes PAC-MDP décrit dans le Chapitre 6. Ainsi, l'algorithme **F-E^3**[Kearns and Koller, 1999] est une version factorisée de **E^3** ; **F-IE** et **F-RMAX**[Strehl, 2007] adapte respectivement **MBIE** et **RMAX**.

Nous nous sommes intéressés à l'algorithme **F-RMAX** ; principalement car son heuristique est plutôt simple à mettre en place et que plusieurs variantes ont déjà été proposées dans la littérature. Dans cet algorithme, en l'état courant s, l'action choisie est l'action dont la *Valeur* d'action estimée pour l'état s est maximale. Ces estimations sont calculées de la manière suivante :

$$\hat{\mathcal{Q}}(s,a) = \begin{cases} \frac{R_{max}}{1-\gamma} \text{ si } \exists X_i \text{ t.q. } n(I(s,a,X_i)) < m_i; \\ \mathcal{R}(s,a) + \gamma \cdot \sum_{s' \in \mathbb{S}} \hat{P}(s' \mid s,a) \max_{a' \in \mathbb{A}} \hat{\mathcal{Q}}(s',a') \end{cases} \tag{7.1}$$

avec R_{max} la récompense maximale connue initialement, $\mathcal{R}(s,a)$ la fonction *Récompense* connue initialement, $I(s,a,X_i)$ le CONTEXTE consistant avec $\hat{P}(X_i \mid s,a)$, $n(I(s,a,X_i))$ le nombre de couples état-action visités compatibles avec ce CONTEXTE et m_i le seuil au-delà duquel le CONTEXTE est considéré comme suffisamment exploré.

Cet algorithme, en dépit de sa preuve de convergence, pose quelques difficultés dans notre cadre de travail. D'une part il suppose R_{max} et $\mathcal{R}(s,a)$ connues, ce qui est

1. I est l'ensemble des indépendances entre variables du FMDP.

contraire à nos hypothèses de départ. D'autre part, il nécessite de connaître toutes les indépendances conditionnelles et contextuelles pour établir les statistiques requises pour gérer le compromis exploration-exploitation. Là encore, comme nous supposons n'avoir aucune connaissance pré-établie du système, ce type de savoir nous est inaccessible. Enfin, cet algorithme utilise une représentation non factorisée pour les fonctions *Valeur* d'action. Nous ne pouvons pas nous permettre une telle représentation du fait de la taille supposée de l'espace d'états.

Les variantes successives de cet algorithme, **SLF-RMAX**[Strehl et al., 2007], **MET-RMAX**[Diuk et al., 2009] et **LSE-RMAX**[Chakraborty and Stone, 2011], traitent le problème de la connaissance a priori des indépendances du modèle. Ces solutions attribuent toutes une limite maximum aux nombres de PARENTS que peut avoir une variable dans sa distribution de probabilité de *Transition*[2]. **SLF-RMAX** cherche alors pour chaque variable l'ensemble de PARENTS qui maximise la vraisemblance de la distribution de probabilité de la variable sachant ces PARENTS[3]. Cette recherche demande un échantillon d'observations de taille conséquente[4]. **MET-RMAX** et **LSE-RMAX** améliore cette complexité en fixant un ordre de préférence dans les variables parentes pour une variable donnée : ainsi si la taille maximale de l'ensemble PARENTS est de K, seules les K premières variables de cet ordre pourront être choisies.

Ces méthodes, en plus de particulièrement contraindre le choix des variables parentes, ne traitent pas les autres problèmes que l'intégration de **F-RMAX** dans notre cadre de travail pose.

7.1.2 ADAPTIVE F-RMAX

Notre proposition cherche à résoudre globalement les trois problèmes précédemment évoqués. En premier lieu, la fonction *Récompense*, étant inconnue dans notre cadre de travail, est apprise par l'algorithme d'apprentissage. C'est donc une version estimée de la fonction *Récompense* qui sera utilisée dans la partie planification par la suite. De plus, ne connaissant pas la fonction *Récompense*, la récompense R_{max} accessible dans le système nous est inconnue. Par conséquent, nous utiliserons la valeur $\hat{R}_{max}$ qui est, à chaque instant t, la plus forte récompense reçue observée.

Notre solution au problème des indépendances utilise les ARBRES DE DÉCISION ORDONNÉS appris. En effet, si chaque ARBRE DE DÉCISION ORDONNÉ était fidèle à la fonction qu'il représente, sa structure nous indiquerait immédiatement quelles sont

2. Pour chaque action bien entendu
3. L'ensemble de PARENTS peut être de taille inférieur à la taille maximale fixée.
4. La complexité de l'échantillon d'observation dépend de la limite maximale de parents.

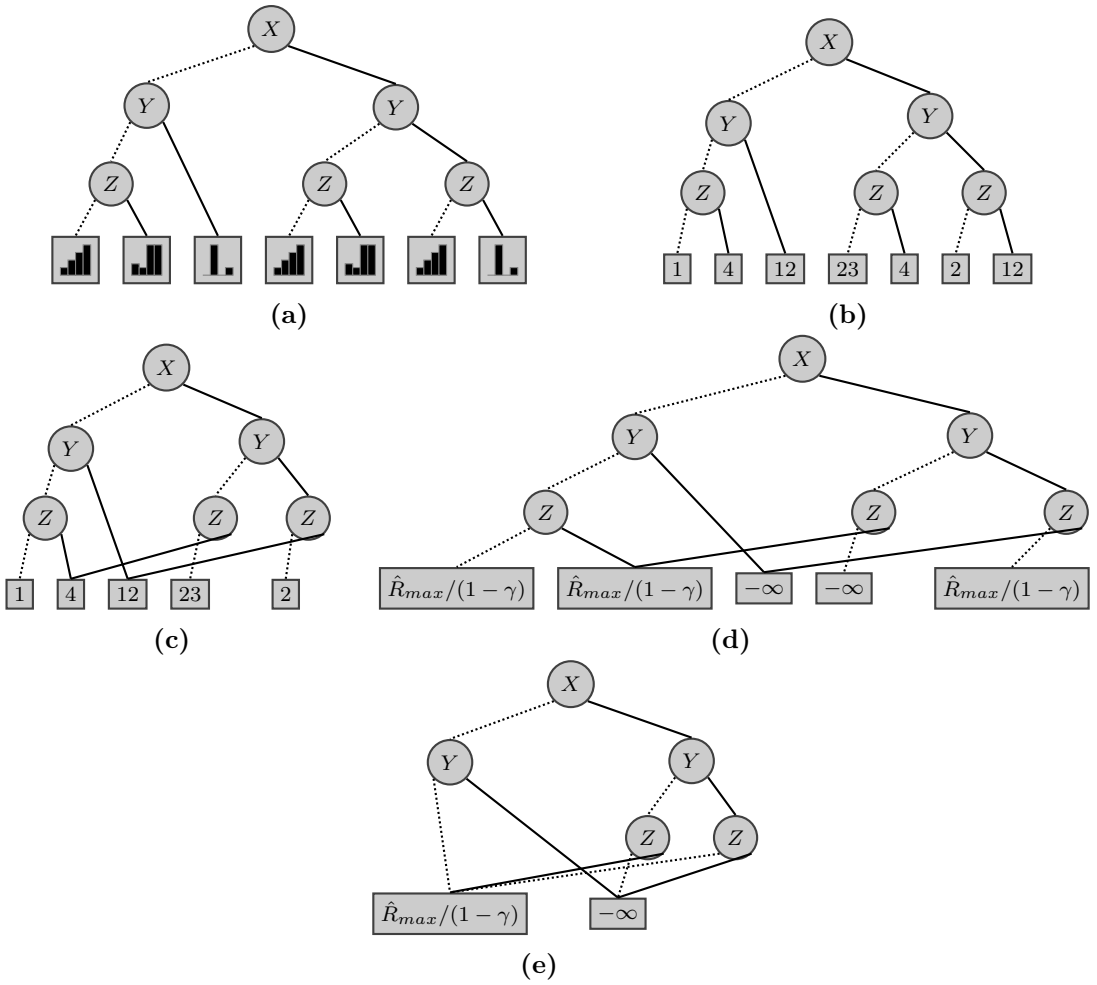


FIGURE 7.1 – *De l'*ARBRE DE DÉCISION ORDONNÉ *associée à* P_{a,X_i}*(a), un* ARBRE DE DÉCISION ORDONNÉ *avec le décompte en chaque feuille est extrait (b). Après avoir été réduit (c), les feuilles de cette* RGFOR *sont remplacées par* $\hat{R}_{max}/(1-\gamma)$ *si le* CONTEXTE *a insuffisamment été visité, par* $-\infty$ *sinon (d). Enfin cette* RGFOR *est réduite (e).*

les indépendances dans la fonction représentée. Ainsi, pour l'ARBRE DE DÉCISION ORDONNÉ représentant la distribution de probabilité de *Transition* de X_i pour l'action a, à toute feuille l seraient liés plusieurs $I(s, a, X_i)$ (autant que d'états compatibles avec le CONTEXTE amenant en cette feuille). Il nous suffirait alors de regarder le nombre d'observations en cette feuille l pour connaître directement $n(I(s, a, X_i))$.

Toutefois, nous n'avons pas cette représentation fidèle. Néanmoins, le nombre d'observations en chaque feuille reste un indicateur intéressant. Il nous permet de savoir en effet si suffisamment d'exemple sont présents en une feuille donnée pour que les statistiques qui y sont effectuées soient pertinentes ; ce qui est le but des $n(I(s, a, X_i))$ à la base. Aussi nous allons nous servir de la taille des bases de données en chaque feuille pour guider l'exploration. L'idée est d'avoir suffisamment de statistiques en chaque feuille pour pouvoir affirmer qu'il n'existe effectivement pas de variables parmi celles

restantes en cette feuille pour la raffiner. Ces décomptes vont donc être récupérés et utilisés par l'algorithme de planification.

Pour se faire, dans un premier temps, pour chaque ARBRE DE DÉCISION ORDONNÉ, les tailles des bases de données en ses feuilles sont extraites. Cette extraction se fait de sorte à garder la structure apprise : il s'agit donc d'une copie avec remplacement aux feuilles des distributions de probabilités par les tailles des bases (cf. Figures 7.1a et 7.1b). Les ARBRES DE DÉCISION ORDONNÉS obtenus sont alors convertis en RGFORs par réduction directe[5](cf. Figure 7.1c). Notons $RGFOR[\mathcal{N}_{a,X_i}]$ la RGFOR donnant la taille des bases de données aux feuilles de l'ARBRE DE DÉCISION ORDONNÉ estimant P_{a,X_i} et $RGFOR[\mathcal{N}_{a,\mathcal{R}}]$ la RGFOR donnant la taille des bases de données aux feuilles de l'ARBRE DE DÉCISION ORDONNÉ estimant $\mathcal{R}_a$.

Ensuite, pour chaque action, nous allons combiner les RGFORs correspondantes par minimisation. La RGFOR $RGFOR[\mathcal{N}_a]$ obtenue donnera ainsi, pour tous les CONTEXTES nécessaires aux définitions des fonctions liées à cette action, le nombre de visites effectuées. Les feuilles de $RGFOR[\mathcal{N}_a]$ sont alors modifiées suivant la règle suivante : si le nombre de visites en cette feuille est inférieur à un seuil, cette feuille reçoit alors la valeur $\hat{R}_{max}/(1-\gamma)$, sinon elle reçoit $-\infty$ (cf Figure 7.1d).

Puis vient la phase de planification où ces RGFORs vont être utilisées. L'algorithme de planification reprend les grandes lignes de **SPUMDD**. Toutefois, lorsque qu'une $RGFOR\left[\hat{\mathcal{Q}}_a\right]$ vient d'être calculée, les $RGFOR[\mathcal{N}_a]$ et $RGFOR\left[\hat{\mathcal{Q}}_a\right]$ sont combinées par maximisation de sorte que la RGFOR obtenue attribue la valeur maximale aux CONTEXTES insuffisamment explorés, et la valeur normale aux autres. Le reste de la planification suit le processus normal. La planification ira ainsi implicitement vers des états associés à des CONTEXTES peu visités, et sinon maximisera les gains.

L'Algorithme **SPUMDD** +**ADAPTIVE F-RMAX** (29) donne l'implémentation de la version modifiée de **SPUMDD**.

Le dernier écueil est résolu de facto. En effet, puisque nous utilisons **SPUMDD** pour la planification, les fonctions *Valeur* d'actions sont directement établies sous forme de RGFORs et donc compactes. Nous n'avons donc pas à nous préoccuper du problème de représentation.

7.1.3 Expérimentations

Pour tester l'efficacité de cette nouvelle heuristique, nous avons repris le banc d'essai vu à la section précédente. À nouveau, 20 tests ont été faits par problèmes, chaque test

5. Étant donné que nous manipulons des valeurs directement, cette phase est courte

Algorithme 29 : SPUMDD +ADAPTIVE F-RMAX : Planification stochastique manipulant des RGFORs multivaluées et gérant implicitement le compromis exploration-exploitation

Données : un FMDP représenté à l'aide de RGFORs multivaluées

début
  // Initialisation
  $RGFOR[\mathcal{V}] \leftarrow \{\}$;
  pour chaque $a \in \mathbb{A}$ **faire**
    $RGFOR[\mathcal{V}] \leftarrow$ **COMBINER**$(RGFOR[\mathcal{V}], RGFOR[\mathcal{R}_a], \max)$;
    $RGFOR[\mathcal{N}_a] \leftarrow \{\}$;
    Extraire et convertir la $RGFOR[\mathcal{N}_{a,\mathcal{R}}]$;
    COMBINER$(RGFOR[\mathcal{N}_a], RGFOR[\mathcal{N}_{a,\mathcal{R}}], \min)$;
    pour chaque $X'_i \in$ SUPPORT$(\mathcal{V})$ **faire**
      Extraire et convertir la $RGFOR[\mathcal{N}_{a,X_i}]$;
      $RGFOR[\mathcal{N}_a] \leftarrow$ **COMBINER**$(RGFOR[\mathcal{N}_a], RGFOR[\mathcal{N}_{a,X_i}], \min)$;
  $RGFOR[\mathcal{V}_{old}] \leftarrow \{\}$;
  // Itération de la valeur
  tant que *critère d'arrêt non atteint* **faire**
    $RGFOR[\mathcal{V}_{old}] \leftarrow RGFOR[\mathcal{V}]$;
    $RGFOR[\mathcal{V}] \leftarrow \{\}$;
    DÉCALER$(RGFOR[\mathcal{V}_{old}])$;
    pour chaque $a \in \mathbb{A}$ **faire**
      pour chaque $X'_i \in$ SUPPORT$(\mathcal{V}_{old})$ **faire**
        Ordonnancer $RGFOR[\mathcal{Q}_a]$ de sorte que X'_i soit à la dernière position dans $\succ_{RGFOR[\mathcal{Q}_a]}$;
        $RGFOR[\mathcal{Q}_a] \leftarrow$ **COMBINER ET ELIMINER**$(RGFOR[\mathcal{V}_{old}], RGFOR\left[P_{a,X'_i}\right], \times, X'_i)$;
        $RGFOR[\mathcal{Q}_a] \leftarrow$ **COMBINER**$(RGFOR[\mathcal{Q}_a], RGFOR[\mathcal{R}_a], +)$;
        $RGFOR[\mathcal{Q}_a] \leftarrow$ **COMBINER**$(RGFOR[\mathcal{Q}_a], RGFOR[\mathcal{N}_a], \max)$;
        $RGFOR[\mathcal{V}] \leftarrow$ **COMBINER**$(RGFOR[\mathcal{V}], RGFOR[\mathcal{Q}_a], \max)$;
  // Récupération gloutonne de la *Politique*
  $RGFOR[\pi] \leftarrow \{\}$;
  pour chaque $a \in \mathbb{A}$ **faire**
    $RGFOR[\pi] \leftarrow$ **COMBINER**$(RGFOR[\pi], RGFOR[\mathcal{Q}_a], \text{argmax})$;

Résultat : une *Politique* π mêlant implicitement exploration et exploitation

effectuant 4000 trajectoires. Chaque trajectoire est constituée de 25 prises de décisions. Nous avons uniquement comptés cette fois-ci le nombre de couples état-action visité et la récompense actualisée.

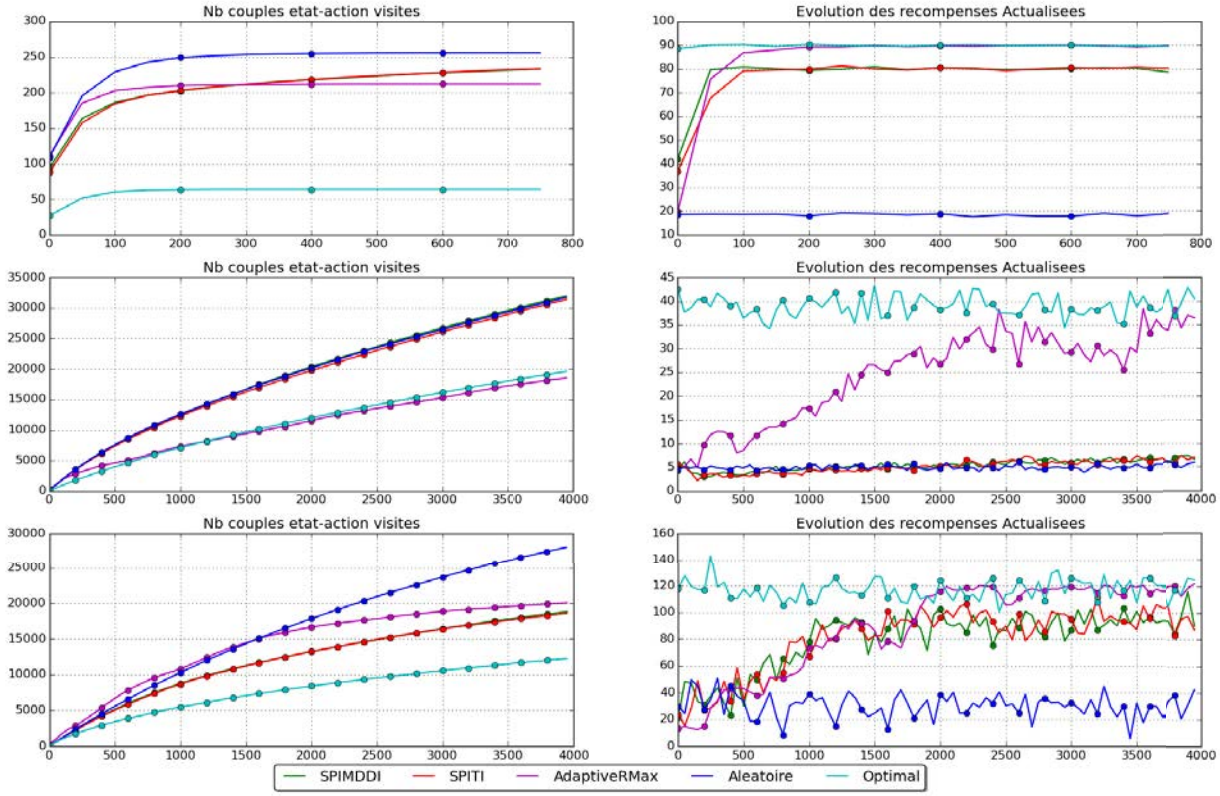


FIGURE 7.2 – *De haut en bas, les résultats pour les problèmes* COFFEE ROBOT, FACTORY *et* TAXI. **SPIMDDI + ADAPTIVE F-RMAX** *(en magenta) est comparé aux autres algorithmes (***SPIMDDI + ϵ-GREEDY***,* **SPITI***, stratégies optimale et aléatoire).*

La Figure 7.2 montre les résultats obtenus. **ADAPTIVE F-RMAX** s'avère être assez efficace puisque, grâce à cette heuristique, dans les trois problèmes, **SPIMDDI** atteint l'efficacité de la stratégie optimale. En particulier, dans le problème FACTORY, **SPIMDDI + ADAPTIVE F-RMAX** parvient à avoir une stratégie pratiquement optimale alors qu'il n'a visité que 3% des couples état-action existants. En comparaison, à 3%, **SPITI** et **SPIMDDI + ϵ-GREEDY** peinent à faire mieux que la stratégie aléatoire.

De plus, nous pouvons remarquer que dans les problèmes COFFEE ROBOT et TAXI, **SPIMDDI + ADAPTIVE F-RMAX** fait aussi bien que la stratégie optimale ; en comparaison, **SPITI** et **SPIMDDI + ϵ-GREEDY** ont tout deux leurs récompenses actualisées 10% inférieurs à la récompense actualisée de la stratégie optimale (et de **SPIMDDI + ADAPTIVE F-RMAX**). Ce décalage est dû à **ϵ-GREEDY** qui 10% du temps choisi une action suboptimale.

D'un point de vue nombre de couples état-action visités, nous constatons que **SPIMDDI + ADAPTIVE F-RMAX** converge plus rapidement que **SPITI** ou **SPIMDDI +**

ϵ-**GREEDY**. Ainsi, dans COFFEE ROBOT, **ADAPTIVE F-RMAX** n'explore pas la totalité de l'espace des couples état-action : l'heuristique plafonne à 212 états, marquant ainsi le passage d'une *Politique* d'exploration à une *Politique* d'exploitation pure. De même, dans TAXI, après avoir initialement visité bien plus de couples que **SPIMDDI** + ϵ-**GREEDY** et **SPITI**, **SPIMDDI** + **ADAPTIVE F-RMAX** tend à visiter moins de couples. Là aussi, ceci dénote le passage à une *Politique* d'exploitation. Les nouveaux couples découverts sont alors le fruit des réinitialisations aléatoires des situations de départ de chaque trajectoire.

7.1.4 Observations

SPIMDDI + **ADAPTIVE F-RMAX** parait donc prometteur. L'étape suivante est de garantir la convergence de **SPIMDDI** avec cette heuristique. Pour y parvenir, il faut d'une part vérifier que **IMDDI** est PAC, mais aussi s'assurer la complexité de l'échantillon d'observation puisse être définie et soit polynomiale dans les quantités données par la Définition 7.1 d'algorithme PAC-FMDP.

S'assurer que **IMDDI** est PAC peut impliquer de modifier **IMDDI**. En effet, bien qu'il existe des preuves de convergence pour l'apprentissage d'arbre de décision [Kim and Koehler, 1996], **IMDDI** a deux particularités qui peuvent rendre cette preuve non accessible : la sélection de variable et l'utilisation d'un pré-élagage dans l'induction de l'ARBRE DE DÉCISION ORDONNÉ, et la fusion probabiliste des feuilles dans la conversion en RGFOR. Mettre le premier aspect en accord avec un apprentissage PAC peut requérir de changer le test utilisé pour la sélection de variables. Le nouveau test devrait néanmoins ne pas être biaisé envers les variables multimodales. Il peut aussi être nécessaire de supprimer le pré-élagage et de le remplacer par un post-élagage. Utgoff et al. [1997] indique comment un élagage en fin d'induction peut être mis en place. La fusion probabiliste des feuilles peut aussi poser problème dans la mesure où elle constitue une approximation supplémentaire apportée à la structure apprise.

La complexité de l'échantillon d'observation est à définir. Un point de départ pour obtenir cette preuve est la preuve de l'algorithme **SLF-RMAX**. En effet, même s'il impose une certaine profondeur aux arbres appris, cet algorithme cherchent à apprendre les CONTEXTES nécessaires pour définir les probabilités de *Transition*. La principale différence entre **ADAPTIVE F-RMAX** et **SLF-RMAX** est que **ADAPTIVE F-RMAX** s'accorde à approfondir certains CONTEXTES là où **SLF-RMAX** reste rigide.

7.1.5 Conclusion

Bien qu'il reste des recherches à effectuer pour aboutir à une preuve de convergence, cette étude reste très intéressante. En effet, elle permet de montrer qu'avec un moteur de décision efficace, **SPIMDDI** parvient à estimer une solution efficace aux problèmes posés, même si ceux-ci sont de grande taille. De plus, elle permet de montrer combien **SPIMDDI** est modulable et adaptable ; les adaptations requises pour utiliser **ADAPTIVE F-RMAX** demandent juste de modifier quelques étapes de l'algorithme de planification, et d'insérer une étape d'extraction des décomptes entre la phase d'apprentissage et la phase de planification. De plus, les algorithmes nécessaires pour intégrer **ADAPTIVE F-RMAX** à **SPIMDDI**[6] sont déjà existant dans **SPIMDDI**.

7.2 Application au jeu vidéo Starcraft

Les problèmes utilisés, bien que de tailles conséquentes, ont été choisi pour leur formalisation simple, cette simplicité permettant une analyse des résultats. Ainsi, ces problèmes étaient posés de sorte que les transitions du système soit intelligible. Les modélisations sous forme FMDP étaient par conséquent facile à réaliser. Les variables caractérisant le système étaient clairement identifiées. Décrire la transition de chaque variable sous forme de RGF ne posait pas de problème. Par ailleurs, la *Politique* optimale s'imposait d'elle-même à un esprit humain. Ainsi, nous disposions d'une grande perspicacité sur ces problèmes étudiés.

Les problèmes réels sont rarement aussi simple à comprendre. La plupart du temps, nous ne disposons que d'une connaissance de surface des mécanismes régulant l'évolution du système étudié, complexe par nature. Dès lors, il nous est impossible de décrire son évolution et encore moins d'établir une *Politique* optimale. Toutefois, il est intéressant de déterminer si un algorithme d'*apprentissage par renforcement* est à même de dégager une *Politique* si ce n'est optimale, au moins efficace, en dépit de notre manque de connaissance sur les mécanismes du système. Les jeux vidéo sont intéressant pour ce type de recherche : ils allient complexité et accessibilité, ce qui les rend très pratiques pour des recherches sur l'*apprentissa- ge par renforcement*.

Ainsi, le comportement optimal à suivre par le joueur n'est pas évident. Ils demandent plusieurs essais et analyses pour déterminer les actions optimales, même pour un joueur possédant les capacités intellectuelles d'un être humain. Leur répétabilité est un atout dans cette optique : les processus d'apprentissage nécessitent de jouer sur plusieurs parties pour apprendre le modèle. Cette rejouabilité s'alliant souvent à une

6. A savoir, l'algorithme **COMBINER** principalement.

réinitialisation aléatoire de la situation de départ permet de mieux explorer l'espace des couples actions-transitions. Ajoutons que les jeux vidéo sont des systèmes relativement "chaotiques" au sens où une légère modification sur la situation initiale conduit à des parties entièrement différentes. Notons que cette répétabilité à de plus un coût faible.

Un autre atout est l'absence de risque sévère pour le système. La seule issue contraignante en cas de faute grave est la défaite dans une partie. Or celle-ci n'entraîne pas de destruction du système. L'algorithme d'*apprentissage par renforcement* peut donc travailler sans que l'on doit se poser de question sur d'éventuelles situations dangereuses.

Enfin, beaucoup de jeux vidéo permettent de faire du "modding". Cette activité consiste à créer soi-même du contenu additionnel pour ces jeux. En particulier, il est possible de programmer par soi-même l'I.A. de ces jeux à partir des signaux émis par le jeux pour nous renseigner sur son évolution.

Pour toutes ces raisons, les jeux vidéo sont particulièrement intéressants pour tester en "grandeur nature" des algorithmes d'*apprentissage par renforcement*.

7.2.1 Description du jeu

Starcraft II[7] est un jeu vidéo de stratégie temps réel. Dans ce type de jeux, le joueur doit atteindre un objectif à long terme (en l'occurrence défaire tous ses adversaires) et dispose pour y parvenir d'une large gamme d'actions à entreprendre : récolte de ressources, construction de bâtiments, d'unités, combats, recherche de technologies améliorant la production, le combat, etc. L'accent n'est pas mis sur l'adresse du joueur mais sur la planification de l'action. De plus, contrairement aux jeux de stratégie tour par tour (comme les Échecs ou le Go), les actions du joueur sont concomitantes à celles de ses adversaires. Enfin, le joueur joue soit contre une I.A. (en solo généralement), soit contre d'autres joueurs humains (ou assimilés comme tels) (partie en réseau).

Dans Starcraft II, le joueur a le choix de la civilisation qu'il joue entre les Terrans, les Zergs et les Protoss ; la race de départ influe sur les options offertes au joueur pour remporter la victoire (unités, batiments et technologies différentes). Pour gagner la partie, le joueur doit détruire tout moyen de perdurer de ses adversaires : détruire leurs bases, leurs bâtiments produisant des unités et leurs unités. Pour y parvenir, les unités du joueur, ainsi que certains de ses bâtiments, doivent *attaquer* les unités et bâtiments adverses. Une unité attaquante inflige à intervalles réguliers[8] des dégâts[8] d'un certain type[8]. Notons que certaines unités attaquent à distance tandis que d'autres doivent être au contact pour porter des coups. Ces dégâts ont pour effet de retirer petit

7. http://eu.battle.net/sc2/fr/
8. Dépendant de l'unité

à petit des points de la réserve de points de vies ou de structures de l'unité ou bâtiment attaqué. Remarquons que le bâtiment ou l'unité attaquée a une résistance [8]aux dégâts reçus, plus ou moins importante suivant le type des dégâts. Lorsque cette réserve arrive à zéro, l'unité est tuée, le bâtiment est détruit.

Engager l'ennemi et le vaincre est donc l'aspect central de ce jeu. Toutefois l'affrontement est le résultat d'un long travail de préparation en amont. Pour constituer sa force de frappe, le joueur doit mettre en place une collecte de ressources qui lui serviront à produire des unités. À l'aide de ces ressources, le joueur a le choix entre construire des bâtiments de production [9], construire des unités ou effectuer des recherches pour améliorer la production, les unités ou la collecte de ressources. En parallèle, le joueur contrôle chaque unité et chaque bâtiment produit. Il peut ainsi déplacer les unités pour effectuer du repérage : découvrir de nouvelles zones d'extraction de ressources, des positions ennemis, des unités ennemis. Il peut également leur demander d'engager les ennemis repérés. Son contrôle sur les bâtiments est un peu plus restreint : si le bâtiment est de production, il peut demander la production d'unités ; si le bâtiment est militaire, il peut demander à celui-ci d'attaquer des unités ennemis à portée (ou des bâtiments si ceux-ci sont à portée).

Le jeu se déroule sur une seule et unique carte générée de manière aléatoire au début de la partie. Cette génération aléatoire inclus les emplacements et tailles des nœuds de ressources que devront exploiter les joueurs pour construire leurs bâtiments et unités et pour effectuer leurs recherches. Les joueurs sont tous placés à des endroits différents de la carte et doivent débusquer leurs adversaires pour engager le combat. Les cartes incluent du relief tels des montagnes ou des cours d'eaux qui bloquent les déplacements des joueurs et qui offrent autant de points stratégiques pour harceler l'ennemi sans en pâtir.

La carte, pour l'ensemble des joueurs, est inexplorée initialement : l'ensemble de la carte est en noir à l'exception de ce qui est en ligne de vue de la base initiale et des premières unités du joueur. Un autre aspect du jeu est donc l'exploration de cette carte pour trouver ressources et adversaires. Cette action est faite en déplaçant simplement toute unité du joueur vers une zone inconnue : dès lors qu'elle est dans le champ de vision de l'unité, elle est découverte. Par la suite, lorsqu'une zone explorée n'est plus de la zone de vue d'une unité, elle s'obscurcit et le joueur ne peut plus savoir ce qui s'y passe : le brouillard de guerre. Pour débusquer l'ennemi, le joueur doit donc envoyé ses troupes en patrouille pour lever ce brouillard de guerre et détecter les mouvements ennemis.

9. qui servent eux-mêmes à la production d'unités

Piloter un I.A. dans ce type de jeux est un défi. La variété d'actions possibles à chaque instant rend toute projection dans l'avenir complexe du fait de l'ensemble des possibles à chaque instant. Toute planification y est donc difficile, d'autant plus que les répercutions vont loin : par exemple, détruire une unité en soi n'est pas très intéressant pour le but final, mais détruire un collecteur de ressources adverse est très intéressant car l'approvisionnement en ressources de l'adversaire (qui est le nerf de la guerre) est directement impacté par cette action. Ce choix permanent d'actions aux conséquences difficilement prédictibles s'inscrit naturellement dans le cadre des processus décisionnels markoviens. Nous cherchons donc à créer une I.A. qui piloterait un camp à Starcraft II.

7.2.2 Mise en place

Mettre en place une I.A. qui contrôle tous les aspects du jeu est très complexe. En effet, Starcraft II, comme beaucoup de STR, demande de gérer plusieurs concepts assez différents.

D'une part, ce jeu requière une macro-gestion des ressources. Celle-ci sont en effet collectées et distribuées de manière globale ; il n'y pas de notion de stocks de ressources collectées situées à des endroits donnés et à déplacer pour être utilisées [10]. Le joueur pose juste ses collecteurs et récupère les ressources [11]. Les ressources collectées sont directement à la disposition de tous les bâtiments de construction et de recherche ; il n'est donc pas nécessaire de les acheminer à ces diverses structures. Aucune logistique est donc à mettre en place.

À l'opposé, les unités sont micro-gérées. Le jeu laisse ainsi à la charge du joueur de diriger ses unités sur le champ de bataille. Rappelons que le joueur se doit d'explorer la carte pour découvrir les ressources, patrouiller pour débusquer l'ennemi et se déplacer pour être à porter d'attaque. Dans la simple gestion de son ensemble d'unité, le joueur fait déjà face à une variété de prise de décisions pouvant l'amener à la victoire ou à la défaite.

Nous nous sommes intéressés (dans une première étape) donc à la gestion de ces unités, laissant de côté les activités de collecte, production, et recherche. Comme il s'agit de micro-gestion, nous cherchons à développer une I.A. assurant la gestion d'une unité. Toutefois, pour multiplier les expériences en parallèle, lors des expérimentations, nous pilotons en parallèle plusieurs unités afin de récolter plus d'observations de transitions.

10. Comme dans d'autre jeux tels que les jeux vidéos de la série Anno (https://fr.wikipedia.org/wiki/Anno_%28s%C3%A9rie%29) par exemple.

11. Éventuellement, il peut être nécessaire de protéger ses collecteurs, mais cette protection rentre dans le second aspect du jeux.

Ainsi un seul FMDP est appris par l'algorithme d'*apprentissage par renforcement*, mais toutes les transitions vécues par toutes les unités d'un camp sont utilisées pour établir ce FMDP.

La situation de départ consiste en deux camps composés de 7 unités et d'une base. L'un des camps est piloté par l'I.A. du jeu, l'autre par notre algorithme d'apprentissage. Les deux camps doivent détruire la base adverse ou défaire toutes les unités ennemis pour remporter la victoire. La carte sur laquelle évolue les unités est de petite taille et est générée aléatoirement en début de chaque partie. Il n'y a pas de relief à l'exception d'éventuels cours d'eau.

Récompenses

Plusieurs récompenses sont offertes lorsque l'I.A. achève des objectifs qui nous semblent pertinents pour la réussite où l'échec de la partie. En premier lieu, les issus du match sont récompensées : une défaite entraîne une pénalité de -7, une victoire entraîne une récompense de 23 points.

Ensuite, perdre une unité est une mauvaise chose en soit. Des ressources ont été dépensée pour produire l'unité perdues ; si l'unité meurt, cette dépense s'avère perdue. De plus, la capacité offensive de notre camp s'amenuise avec cette perte. Toutefois, les assauts en masses de structures défensives impliquent nécessairement des pertes pour parvenir à détruire ces structures. Toute pénalité infligée ne doit pas être trop importante pour rendre cette options inintéressante (au sens de la maximisation de l'espérance des gains cumulés). Ainsi, lorsqu'une unité se fait tuer, elle reçoit une pénalité de -1.

L'exploration d'une nouvelle zone de la carte est récompensée de 5 points. Cette activité est en effet très importante dès le début du jeu, ne serait-ce que pour localiser les ressources nécessaires au développement économique. Or nous n'avons pas de gestion économique mise en place, donc pas de moyen pour récompenser sur le long terme la découverte des nœuds de ressources. Cette récompense permet de le simuler.

Espace d'états $\mathbb{S}$

Les senseurs utilisés pour indiquer la situation d'une unité se regroupe en deux grandes catégories : d'une part les senseurs renseignant sur l'état interne de l'unité, d'autre part les senseurs donnant des informations sur son environnement. Le senseur détaillant l'état interne d'une unité est le suivant :

— **Points de vie** renseigne sur la réserve de santé de l'unité :
  - *Full* : la réserve de vie est plein,

- *High* : la réserve de vie est supérieure à 50 % de sa capacité maximale,
- *Low* : la réserve de vie est en dessous des 50 %,
- *Zero* : la réserve de vie est vide (et donc que l'unité est morte).

Pour décrire l'état de l'environnement immédiat d'une unité, plusieurs senseurs sont utilisés :

— **État Cible** indique le statut d'une éventuelle cible :
  - *Aucune* : aucune unité ennemi n'est ciblé,
  - *Engagée* : l'unité porte des coups à une unité adverse ciblé,
  - *Hors de portée* : l'unité adverse ciblé n'est pas à portée de coups,

— **Localisation** indique de manière abstraite la position de l'unité sur la carte :
  - *Proche Base Allié* : l'unité est à portée de vue de la base alliée,
  - *Proche Base Ennemie* : l'unité est à portée de vue de la base ennemie,
  - *Proche Troupe Allié* : des unités alliées sont en vue,
  - *Proche Troupe Ennemi* : des unités ennemies sont en vue,
  - *Frontière* : l'unité est à côté d'une zone inexplorée,
  - *Limbes* est la résultante d'une difficulté technique [12],
  - *Défaut* est une situation qui ne rentre pas dans ces autres cas [13],

— **Nombre d'Unités Alliés**,

— **Sous le Feu** : l'unité est attaquée ou non,

— **Nombre d'Unités Ennemies en Vue** : nombre d'unités ennemis que l'unité aperçoit directement.

Au bout du compte, la taille de l'espace d'états $|\mathbb{S}|$ est de 2352 états avec 7 unités sur la carte (le nombre d'unités utilisé pour les expérimentations) ; cette modélisation en fait donc un problème d'assez grande taille.

Espace d'Actions $\mathbb{A}$

L'I.A. disposait d'une large choix d'actions à ordonner à ses unités de faire. Ces actions se déclinent en plusieurs catégories. Dans un premier lieu, il y a les actions de déplacement qui consiste à déplacer l'unité sur la carte :

— **Explorer** : l'unité se dirige vers la plus proche zone encore non explorée,

— **Patrouiller** : l'unité fait un aller-retour entre sa zone de départ et une zone définie aléatoirement,

— **Venir en renfort** : l'unité rejoint une autre unité subissant une attaque,

12. En effet, lorsqu'une unité meurt, le jeu nous fait perdre toute information sur sa situation, d'où cet artifice.

13. Seul sur la carte et loin d'une frontière donc.

— **Aller en dernière position ennemie connue** : l'unité se dirige vers la zone où un ennemi a été vu pour la dernière fois,
— **Tracker** : si l'ennemi passe hors de portée ou disparaît, l'unité part dans la direction où il a été pour la dernière fois vu.

Ensuite, nous avons les actions d'engagement :

— **Attaquer** : L'unité attaque l'unité adverse la plus proche en vue (qui devient automatiquement sa cible),
— **Fuire** : L'unité s'éloigne de la cible.

L'option **Ne rien faire** est laissée à l'unité.

Ces actions sont de haut-niveau. Typiquement, les actions de déplacement utilisent un algorithme $A^\star$ [Hart et al., 1968] pour effectuer le déplacement du point de départ au point d'arrivée. De même, l'action Attaquer effectue une sélection automatique de l'ennemi le plus proche [14]. L'action Fuire cherche à faire s'éloigner au plus vite l'unité de la zone de combat. Une recherche de la meilleure direction pour s'éloigner des ennemis est donc effectuée.

Au total, le système ainsi modélisé dispose de 18816 couples états-actions pour la version à 7 unités.

7.2.3 Résultats

Une fois ce modèle élaborée et les différentes actions et différents senseurs mis en place pour suivre l'évolution du jeu et agir en conséquence, nous avons couplé ces éléments avec **SPIMDDI**. La Figure 7.3 montre le dispositif expérimental constitué du jeu qui s'exécute dans sa propre fenêtre et des divers programmes nécessaires au suivi des signaux émis par le jeu et à l'exécution de **SPIMDDI**. Pour établir les résultats qui suivent, nous avons exécuté plusieurs sessions de 60 parties successives sans réinitialiser le modèle appris par **SPIMDDI** entre chaque partie. Au terme de ces 60 parties, le modèle appris et la *Politique* optimale estimée sont exportés. Les observations qui suivent se basent sur ces résultats.

Apprentissage des fonctions *Transition*

Au terme des 60 parties d'apprentissage, **SPIMDDI** a appris des fonctions élaborées. Ainsi, soit la Figure 7.4 représentant l'évolution de la variable PV lors de l'action **Attaquer**. Notons d'abord que l'évolution des *Points de Vie*, d'après ce qui a été appris, dépend des *Points de Vie* (PV) à l'état précédent, du *Nombre d'Unités Alliés* (NUA),

14. Il est tout à fait envisageable d'améliorer l'intelligence de cette attaque en ciblant l'unité la plus proche et ayant le moins de point de vie.

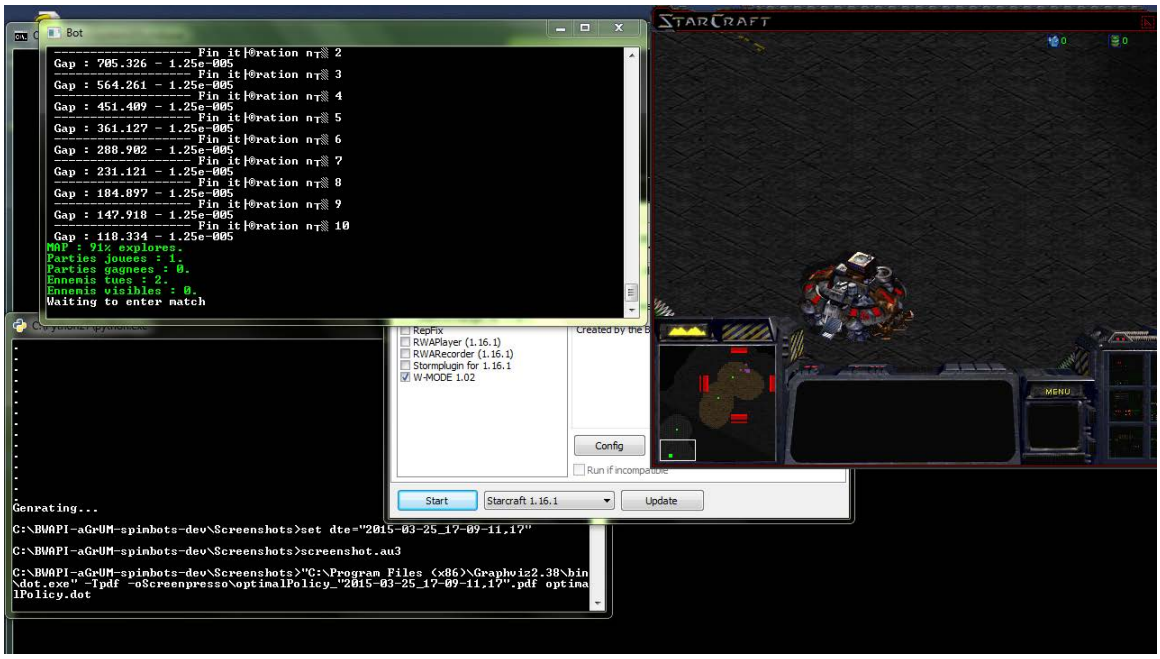


FIGURE 7.3 – *Un aperçu du cadre expérimentale pour Starcraft II. La fenêtre en haut à droite est la fenêtre du jeu en lui-même. Les autres fenêtres sont des shell windows nécessaire pour exécuter notre I.A.*

de la *Localisation* (L) et de si l'unité est *Sous le Feu* (SLF) ennemi ou non ; la sélection de ces variables est plutôt pertinente. D'une part, il parait normale que l'évolution des *Points de Vie* soit conditionnée à leur valeur précédente. Sa position sur la carte est aussi important : être isolé n'affecte pas *a priori* l'espérance de vie, en revanche se trouver à proximité d'ennemis l'est. De même, si l'unité est sous le coup d'une attaque, il est logique de s'attendre à une diminution de sa réserve de vie. Enfin, moins il y a d'unités alliés sur la carte, plus il est probable que les unités ennemies restantes attaquent en masse nos unités éparses.

Le graphe obtenu est bien une représentation compacte du problème. D'une part, toutes les variables ne sont pas présentent sur chacun des chemins depuis la racine, il y a donc bien abstraction et regroupement en CONTEXTE. D'autre part, quelques regroupements de CONTEXTES existent. En effet, chaque arc ayant plusieurs modalités indiquées signifie que pour le deux modalités le graphe des Descendants associé est similaire. De même, quelques nœuds possèdent plusieurs parents, dénotant ainsi des regroupements de CONTEXTES. **SPIMDDI** semble donc bien capable de dégager des CONTEXTES et de les regrouper dans ce problème.

Notons toutefois un problème lié aux états impossibles : l'estimation des probabilités de *Transition* pour des états impossibles. Par exemple, dans la Figure 7.4, si nous considérons le CONTEXTE $\langle$ PV = FULL, NUA = 0 $\rangle$, nous pouvons observer que la

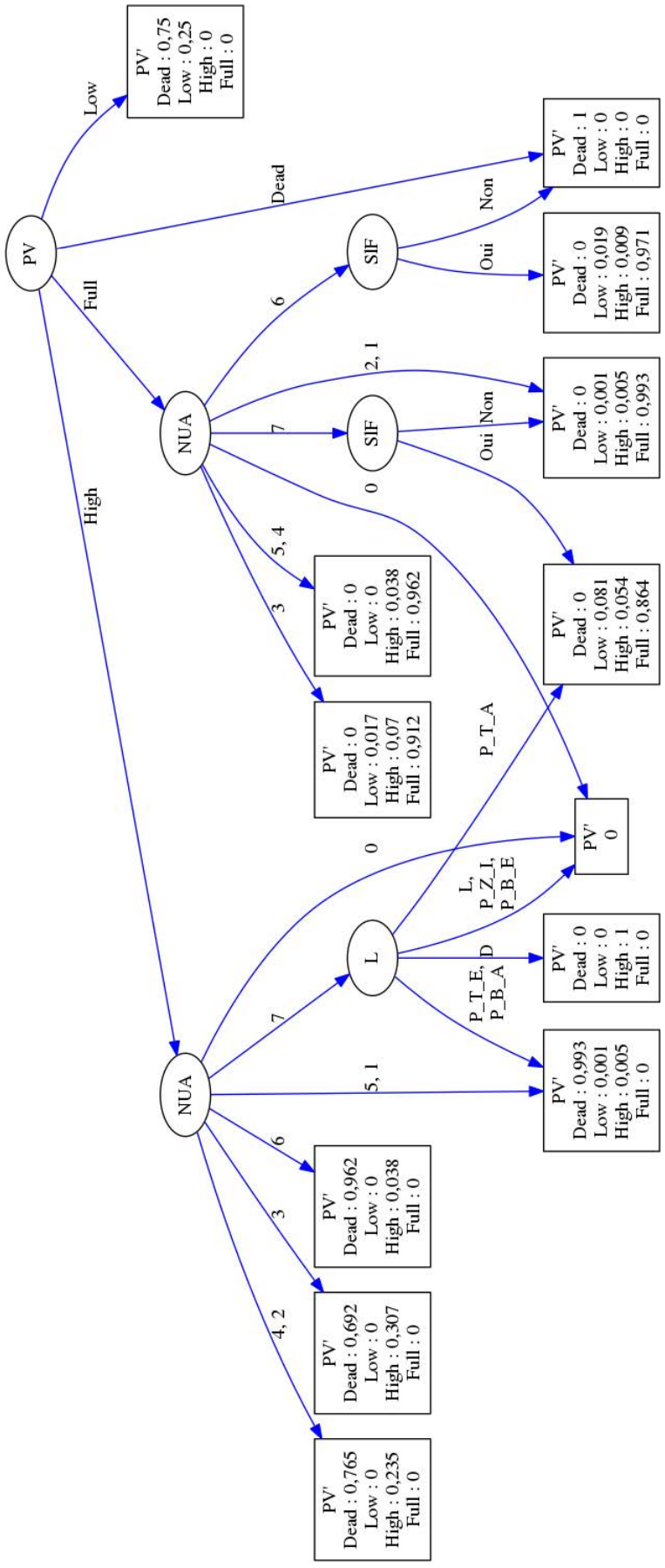


FIGURE 7.4 – *La fonction* Transition *pour la variable PV lors de l'action* ***Attaquer*** *apprise au terme d'une série d'expériences. Les valeurs indiquées pour les distributions de probabilités ont été tronquées à 3 chiffres après la virgule*

feuille associée indique 0 comme seule valeur plutôt qu'une distribution de probabilité. En d'autre terme, la probabilité de transition de *Points de Vie* de la modalité HIGH vers toute modalité est nulle. Ce CONTEXTE est associé à l'ensemble des états tels que plus aucune unité n'est en vie (NUA = 0) mais que l'unité considérée est à FULL en *Points de Vie* ; ces états sont impossible puisqu'il n'est pas possible d'être mort et d'avoir une réserve de vie remplie (PV à FULL). Aucune observation empirique n'est donc faisable pour établir une distribution de probabilité cohérente à associer à ce CONTEXTE. Le problème alors est que la valeur en cette feuille ne constitue pas une distribution de probabilité. Les calculs de planification effectués par la suite seront donc faussés pour ces états.

Toutefois, des solutions existent pour gérer ces états impossibles [Kozlova et al., 2009]. Ces solutions sont facilement intégrables à **SPIMDDI** ; en effet, ces solutions reposent sur des modifications de l'algorithme **SPUMDD** tout à fait applicables aux RGFORs. De plus, nous disposons d'un moyen de détection de ces états lors de l'apprentissage : toutes les probabilités de transition associées sont nulles. La gestion des états impossibles par **SPIMDDI** est donc possible.

Apprentissage des fonctions *Récompense*

La Figure 7.5 montre la représentation graphique de la fonction *Récompense* apprise pour l'action **Explorer**. Comme nous pouvions nous y attendre, la *Localisation* intervient. Toutefois, trois autres variables interviennent : le *Nombre d'Unités Alliés*, les *Points de Vie* et si l'unité est *Sous le Feu* ou non. La raison est que si l'unité meurt lors de son exploration, elle décroche la pénalité de mort. De même, si c'est la dernière unité à mourir, il est logique de recevoir la pénalité de partie perdue à l'issu de l'action.

La fonction apprise est donc en réalité la somme de plusieurs sous-fonction récompenses (Chapitre 1). Ces fonctions peuvent être retrouvées par décomposition additive, mais sont apprise sous la forme de cette fonction composée. Comme l'illustre la Figure 7.5, il est difficile d'analyser la pertinence de l'apprentissage en conséquence. Les feuilles indiquent en effet la moyenne des récompenses obtenues et non le détail.

De plus, la fonction apprise est nécessairement plus complexe que les fonctions qui la composent. Cette fonction est alors plus difficile à manipuler en mémoire. Au mieux de notre connaissance, il n'existe toutefois pas de technique pour apprendre les fonctions composantes directement.

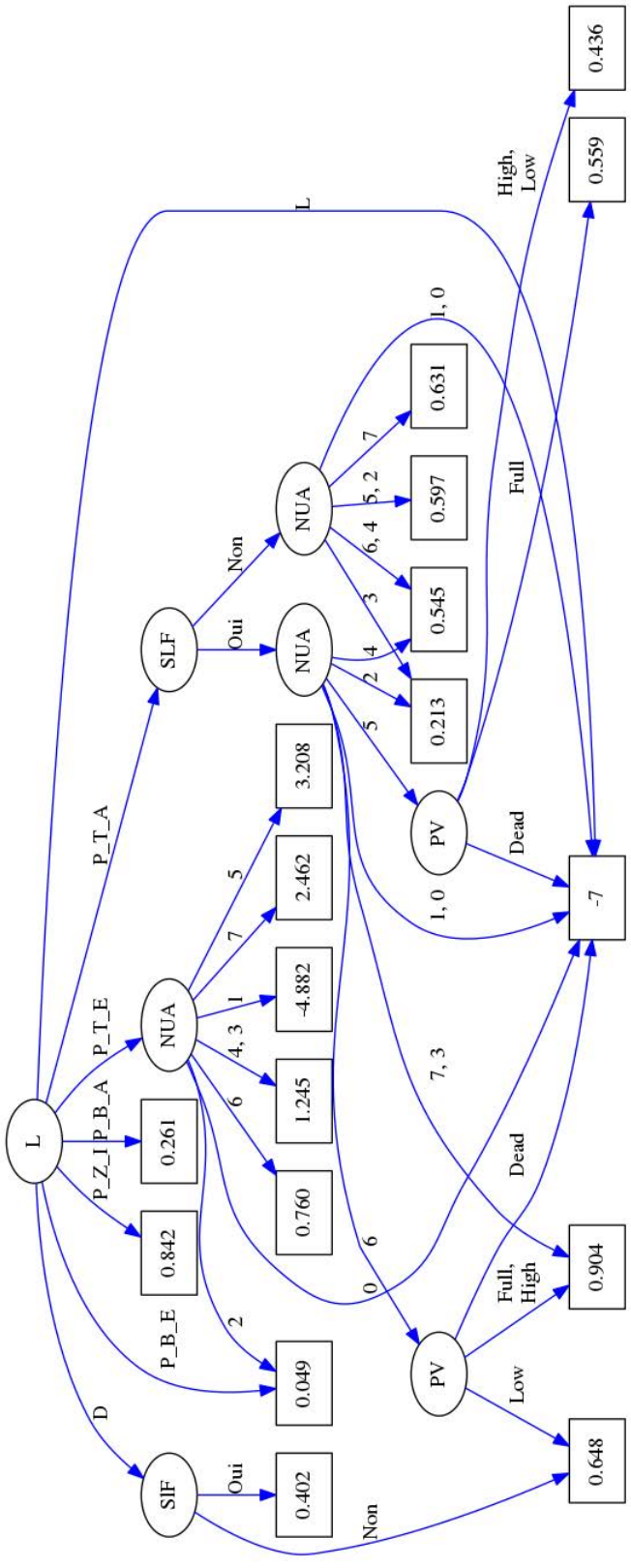


FIGURE **7.5** – *La fonction* Récompense *l'action* ***Explorer*** *apprise au terme d'une série d'expériences.*

Évolution de la *Politique* optimale estimée

La Figure 7.6 illustre l'évolution de la *Politique* optimale estimée au cours des 60 parties. Néanmoins, nous n'avons mis que les *Politiques* optimales estimées à la fin de la première manche et à la fin de la vingtième partie. En effet, la toute dernière *Politique* estimée, au bout de 60 parties, est bien trop grande (240 nœuds en tout) pour pouvoir être affichée.

L'évolution d'une *Politique* à 5 nœuds vers une *Politique* à 240 nœuds montre clairement que le modèle appris se complexifie, de même que la solution optimale au problème posé. Cette solution reste néanmoins plus compacte que l'espace d'état originel. En effet, d'une part plusieurs nœuds ont plus d'un parent, d'autre part plusieurs arcs sont associés à plusieurs modalités.

ADAPTIVE F-RMAX et le problème de suicide planifié

Dans cette étude, nous avons aussi chercher à voir ce que donnait une meilleure gestion du compromis exploration-exploitation. Aussi avons-nous appliqué notre proposition d'algorithme **ADAPTIVE F-RMAX**. En effet, ce cadre d'application est le cadre typique pour **ADAPTIVE F-RMAX** : l'absence de connaissance sur le modèle de la part de l'utilisateur est complète, l'heuristique doit donc faire de son mieux par elle-même.

En terme de nombre de victoires pures, **ADAPTIVE F-RMAX** s'avère très mauvais [15] pour des raisons intéressantes. Le fait est qu'**ADAPTIVE F-RMAX** s'avère plutôt efficace pour explorer les nouvelles régions de l'espace des couples état-action. Mais la modélisation retenue fait qu'**ADAPTIVE F-RMAX** agit contre l'objectif de victoire. En effet, la modélisation a un gros écueil : la moitié des états sont des états dans lesquels les unités sont mortes. Néanmoins, **ADAPTIVE F-RMAX** ne sait pas que ces états sont contre-productifs pour l'objectif de victoire, du moins pas immédiatement. Pour lui, il s'agit juste de nouvelles régions prometteuses à découvrir. Il planifie donc de sorte à aller explorer ces "endroits prometteurs". Or y aller implique de tuer les unités, et donc de perdre. Par conséquent, **ADAPTIVE F-RMAX** planifie sa défaite pour pouvoir aller explorer ces zones.

Ce problème n'est pas simple à résoudre. Modifier la modélisation n'est pas possible car retirer la notion de mort de l'unité reviens à supprimer un ensemble de configuration du système. Une solution possible serait de laisser le programme persévérer dans l'erreur jusqu'à ce qu'il parvienne à la conclusion que ces états ne sont pas intéressants. Dans le cadre de la gestion d'une I.A. pour un jeu vidéo, cette solution n'est pas un

15. 2 victoires contre 14 pour ϵ-**GREDDY**

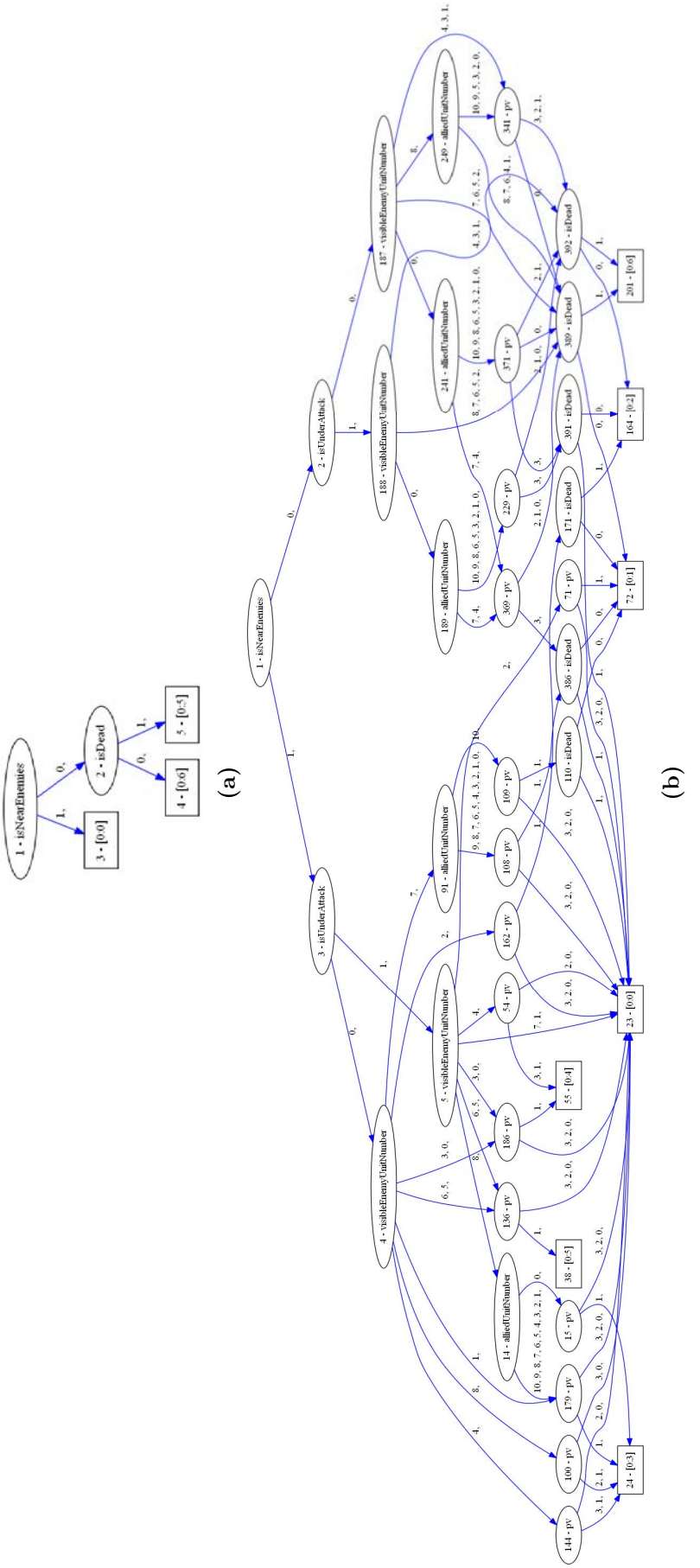


FIGURE 7.6 – *Evolution de la* Politique *optimale estimée après une partie (a), et après une vingtaine (b).*

problème ; ces morts d'unités n'ont pas de conséquences fatales pour le système. Néanmoins, pour d'autres applications, ce problème pourrait être plus délicat à résoudre et rejoint la problématique de l'exploration dans un espace dangereux [Hans et al., 2008; Moldovan and Abbeel, 2012].

7.2.4 Conclusion

Cette étude illustre bien la capacité de **SPIMDDI** à dégager une structure d'un problème a priori très complexe. Comme le montre les diagrammes présentés ici en exemple, une partie seulement des variables est utilisée pour décrire chaque transition et chaque récompense. **SPIMDDI** possèdent donc une capacité d'abstraction. De plus, il apparaît que **SPIMDDI** est capable de procéder à des regroupements de sous-CONTEXTES (nœuds ayant plusieurs parents ou plusieurs fois le même fils). Enfin, la *Politique* optimale estimée est elle-aussi une représentation compacte.

En outre, elle ouvre plusieurs pistes de recherche.

Une première piste dégagée est celle des états impossibles. Comme le montre la fonction *Transition* de la variable PV pour l'action **Attaquer**, l'algorithme **IMDDI** attribue une probabilité nulle ou une récompense nulle aux CONTEXTES impossibles. Lors de l'apprentissage de fonction *Transition*, certaines feuilles obtenues ne sont alors pas des distributions de probabilité. Toutefois des solutions existent pour gérer ces CONTEXTES et sont adaptables à **SPIMDDI**.

Une autre difficulté soulevée par cette étude est l'apprentissage de la fonction *Récompense*. En effet, **IMDDI** apprend une fonction *Récompense* qui cherche à expliquer l'ensemble des récompenses obtenues. Par exemple, dans la fonction *Récompense* associée à l'exploration, la manière dont les récompenses liées à l'exploration et à la mort de l'unité sont obtenues sont apprises ensemble. La représentation obtenue est en conséquence peu claire, et peu potentiellement être lourde à gérer en mémoire. La question se pose alors de mettre en place un apprentissage exploitant une décomposition additive des récompenses.

Enfin, le problème de l'exploration d'états dangereux pour l'agent se pose. En effet, comme nous avons pu l'expérimenter avec **ADAPTIVE F-RMAX**, **SPIMDDI** va logiquement attribué une forte récompense à tous les CONTEXTES nouvellement découverts pour les explorer plus avants. Dans cette étude, cela implique de suicider les unités pour mieux explorer les CONTEXTES où ces unités sont détruites. Dans ce cadre, perdre quelques parties pour mieux explorer ces états n'est pas dramatique. Mais, dans des situations où explorer des états dangereux s'avère fatales pour l'agent, des solutions pour contourner cette problématique doivent être trouvées.

7.3 Synthèse

Dans ce chapitre ont donc été présentés deux travaux de recherche s'appuyant sur l'architecture **SDYNA** et sur les RGFORs. Le premier, l'algorithme **ADAPTIVE F-RMAX**, consiste en une adaptation de l'algorithme de gestion du compromis exploitation-exploration **F-RMAX** à notre cadre de travail où les hypothèses de bases nécessaires à l'utilisation de **F-RMAX** ne peuvent pas être satisfaites. Les expériences menées sur **SPIMDDI** + **ADAPTIVE F-RMAX** montrent que l'algorithme **ADAPTIVE F-RMAX** permet de faire une exploration efficace de l'espace des couples état-action même si les hypothèses de **F-RMAX** ne sont pas satisfaite. Le deuxième travail de recherche, l'utilisation de **SPIMDDI** pour gérer une I.A. de jeux vidéos, a permis de voir comment **SPIMDDI** se comportait sur un problème réel.

Ces deux travaux illustrent la simplicité d'utilisation et l'adaptabilité de **SPIMDDI**. En effet, l'algorithme **ADAPTIVE F-RMAX** requière juste de récupérer certaines données d'apprentissage et de rajouter une ligne de code aux algorithmes de planification déjà existants. De même, l'utilisation de **SPIMDDI** pour Starcraft II requiert simplement de spécifier quelles variables décrivent les états du système et quelles actions sont envisageables. Ces spécifications ne nécessitent pas de modifier **SPIMDDI** pour l'utiliser. L'algorithme **SPIMDDI** s'avère donc être un algorithme d'*apprentissage par renforcement* efficace et pratique pour résoudre des problèmes de grandes tailles.

Discussion

Les travaux présentés dans cette thèse s'intéressent à l'amélioration du cadre factorisé des MDPs, très utile pour modéliser les *problèmes de décisions séquentielles dans l'incertain* de grandes tailles. Ce cadre se sert d'un ensemble de variables pour caractériser chaque état du système étudié. En exploitant les indépendances conditionnelles et contextuelles sur cet ensemble, les fonctions *Transition* et *Récompense* de tout FMDP sont "compressées".

Elles peuvent alors être stockées en mémoires à l'aide de représentations structurées de données telles que des ARBRES DE DÉCISION ou des ADDs. Ces structures de données sont des graphes dirigés sans cycle tel que seul un nœud soit sans parents (la racine). Chacun de ces graphe possède un ensemble de nœuds internes associés à une des variables du problème et un ensemble de nœuds terminaux associés aux valeurs prises par la fonction représentée. Chaque chemin du graphe depuis la racine vers une feuille est une instanciation des variables rencontrées dans ce chemin. Cette instanciation partielle, aussi appelée CONTEXTE, est telle que la fonction représentée est constante pour toute instanciation des variables restantes ; sa valeur assumée est alors celle contenue dans la feuille.

Grâce à ces représentations, les problèmes sont plus facilement stockables en mémoire et manipulables ; la résolution des problèmes de décisions s'en trouve accélérée. L'algorithme **SPUDD** parvient ainsi à trouver en quelques minutes la solution à des problèmes de décision à plusieurs millions d'états possibles.

Transversalement, ces mêmes structures de données sont utilisées dans le cadre de l'*Apprentissage par Renforcement*. En se basant uniquement sur l'observation des transitions vécues par le système, les algorithmes d'*Apprentissage par Renforcement* recherchent empiriquement une solution au problème de décisions posé. Dans l'architecture **SDYNA**, le FMDP du problème est appris sous forme d'un ensemble de représentations structurées grâce aux observations. Puis un algorithme de planification se sert de ce modèle estimé pour en déduire une solution au problème de décision initiale. Cette architecture est très intéressante puisqu'elle permet d'apprendre le modèle d'un

système sur lequel nous savons très peu de choses (en dehors des variables qui le caractérise), et parvient à en dégager des indépendances contextuelles et conditionnelles du problème [16].

Toutefois, antérieurement à cette thèse, l'état de l'art des FMDPs et de l'architecture **SDYNA** présentait quelques choix de conception qui soulevaient des contraintes et des limitations. Nous nous sommes donc intéressés durant cette thèse à lever ces contraintes.

Amélioration de la représentation

Le premier écueil de l'état de l'art concernait la modélisation des problèmes dans les FMDPs. Dans l'état de l'art, pour stocker en mémoire les fonctions de transitions et de récompenses des FMDPs, plusieurs ADDs étaient utilisés comme structure de données. Ces structures permettaient en effet de tirer profit des indépendances conditionnelles et contextuelles pour réduire la place nécessaire en mémoire pour décrire ces fonctions. De plus, contrairement aux ARBRES DE DÉCISION [17], ces structures ne présentaient pas de redondances. Néanmoins, elles obligeaient les FMDPs à utiliser uniquement des variables binaires pour décrire le problème. Chaque variable multimodale devait alors être convertie en un ensemble de variables binaires pour être utilisable. La modélisation s'en trouvait alors plus complexe.

Contributions

Nous proposons une structure plus générale qui englobe les ADDs dans sa définition : les RGFORs. Ces structures de données sont capable de modéliser les FMDPs aussi efficacement que les ADDs mais en plus ne restreignent pas le domaine des variables à un domaine binaire. Ce retrait de contrainte entraîne une réduction de la taille requise en mémoire pour stocker ces structures. Cette structure s'avère donc plus intéressante pour modéliser les FMDPs de grande taille.

Toutefois, cette généralisation nécessite la mise à jour de quelques algorithmes de manipulation des ADDs pour s'adapter à un cadre multimodale. Ainsi, la fonction d'échange de variables adjacentes dans l'ordre global a due être redéfinie pour manipuler des nœuds ayant plus de deux fils. Rappelons que cette fonction est centrale dans les méthodes de réduction de la taille des RGFORs. L'autre fonction importante qui

16. Du moins, un sous-ensemble d'indépendances
17. Une autre structure de données utilisée pour modéliser les FMDPs en exploitant les indépendances conditionnelles et contextuelles

a due être adaptée à un cadre multimodale est la fonction de combinaison de RGFORs (cf. section suivante).

Perspectives

Dans un premier temps, il est nécessaire d'améliorer la librairie logicielle utilisée pour les RGFORs. La librairie logicielle développée autour de cette nouvelle structure de donnée est loin d'être optimale techniquement. La librairie CUDD s'avère plus performante à manipuler les ADDs que notre librairie à manipuler les RGFORs. En l'état actuel, **SPUDD** avec l'aide de CUDD est plus rapide pour trouver les *Politique* optimales. Toutefois, CUDD est une librairie qui existe depuis plus de vingt ans. La communauté VLSI-CAD [18] a beaucoup aidé à son développement car les BDDs (dont les ADDs ne sont qu'une généralisation) sont nécessaire dans ce domaine de la conception matérielle. En comparaison, les RGFORs ont une très petite communauté d'utilisateurs.

Par la suite, plusieurs pistes de recherche existent pour améliorer et étendre le champ d'action des RGFORs. Ainsi, une manière possible d'améliorer la représentation est d'introduire l'utilisation de nœuds internes liés à plusieurs variables [Breslow and Aha, 1997]. Dans ce type de nœud, les arcs sortant sont liés aux valeurs prises par une fonction sur les variables liées au nœud (généralement une clause booléenne). Cette manière de procéder permet de réduire le nombre de nœuds nécessaires. Dans notre cadre d'étude, toute la difficulté est de définir des algorithmes de combinaison sur ces structures afin de pouvoir faire de la planification. Pour rappel, les algorithmes de combinaison reposent sur une énumération des CONTEXTES nécessaires et suffisants pour définir les deux fonctions combinées. Il s'agit donc de savoir ce que deviennent ces CONTEXTES avec la nouvelle représentation, et comment les énumérer.

Un autre axe de recherche est l'introduction de variables continues, ou du moins discrète mais de domaine infinie. Des travaux existent déjà sur l'utilisation de telles variables dans les FMDPs [Kveton, 2006]. Tout le problème est de savoir comment les intégrer à des représentations structurées.

Planification

Nous avons vu que l'algorithme **SPUDD**, état de l'art de pour la recherche de la *Politique* optimale dans les FMDPs, présentait deux défauts. Le premier était de s'appuyer sur une représentation à l'aide d'ADDs pour la planification. Or ces représentations ne

18. Very Large Scale Integration - Computer Assisted Design

sont pas nécessairement les plus aptes à représenter les FMDPs possédant des variables multimodales. La deuxième était de se servir d'un algorithme de combinaison de ces ADDs qui imposait que les ordres sur les variables des deux ADDs combinés soient similaire. Or, d'une part, la complexité de cet algorithme de combinaison des ADDs dépend de la taille des ADDs combinés, et, d'autre part, la taille des ADDs est sensible à la manière dont les variables sont ordonnées. Il en résulte que **SPUDD** utilise une structure qui n'est pas la meilleure qui soit, et utilise dessus un algorithme qui force les ADDs à ne pas nécessairement avoir leur taille minimale, sachant que cet algorithme voit sa complexité dépendre de cette taille.

Contribution

Notre contribution a donc été double. D'une part, nous proposons l'utilisation d'une structure de données plus simple aux travers des RGFORs (cf. contribution précédente). L'algorithme de combinaison d'ADDs a dû alors être modifié pour pouvoir opérer sur des nœuds ayant plusieurs fils (pour les variables multimodales). Cette modification consiste principalement à faire en sorte que les parcours en profondeur utilisés dans cet algorithme se fassent bien sur tous les fils d'un nœud donné.

D'autre part, nous proposons aussi un nouvel algorithme de combinaison qui retire la contrainte d'ordre. Cet algorithme s'appuie lui aussi sur des parcours en profondeur des deux RGFORs combinées. Cet algorithme doit néanmoins prendre quelques dispositions pour s'assurer que l'exploration permette une construction intègre du graphe résultat. Ces dispositions impliquent un surcoût en complexité. Toutefois, l'algorithme **SPUMDD** qui combine donc des RGFORs à l'aide de ce nouvel algorithme de combinaison s'avère plus efficace pour la planification que **SPUDD**.

Perspective

Plusieurs pistes sont à explorer pour améliorer la rapidité et la convergence des algorithmes.

Planifier les combinaisons Un futur amélioration possible vient de la manière dont sont combiné les RGFORs dans **SPUMDD**. En effet, dans l'implémentation actuelle, pour chaque action, la RGFOR représentant la fonction *Valeur* de l'itération précédente est multiplié par les RGFORs associées à chaque variable. Au total il y a donc $|\mathbb{A}| \times |\mathbb{X}|$ combinaisons. Toutefois il est possible de faire diminuer ce nombre. En effet, d'une action à l'autre, la RGFOR associée à une variable est potentiellement la même. Par

exemple, dans COFFEE ROBOT, la RGFOR associée à la variable R (pleut-il ?) ne change pas d'une action à l'autre car elle ne dépend absolument pas des actions entreprise par le robot. Nous pourrions donc gagner du temps en ne multipliant qu'une seule fois la RGFOR représentant la fonction *Valeur* de l'itération précédente avec la RGFOR associée à la variable R. De même, la variable *Loc* associée à la position du robot n'a qu'une action pour laquelle sa RGFOR diffère : l'action $Move$. Nous pourrions donc d'un côté multiplier la RGFOR représentant la fonction *Valeur* avec la RGFOR associée à la transition par défaut [19] de la variable *Loc*, et de l'autre côté multiplier la RGFOR représentant la fonction *Valeur* avec la RGFOR associée à la transition pour l'action $Move$ de la variable *Loc*. Deux combinaisons seraient alors nécessaires au lieu de quatre. Identifier les ensembles d'actions pour lesquels plusieurs variables ont une RGFORs constante permettrait ainsi d'économiser sur les combinaisons à faire et donc de gagner du temps. Ce problème est *a priori* NP-Difficile.

Combinaisons en place Dans les MDP (cf. Chapitre 3), une amélioration possible pour accélérer la convergence était de faire les mise à jour en place : une seule table était utilisée pour représenter la fonction valeur à l'itération n et à l'itération $n+1$ (cf. sous-section* 3.2.5). Une amélioration envisageable des algorithmes de planification structurés est de ne manipuler qu'une RGFOR pour la fonction *Valeur* (ou du moins qu'une RGFOR pour chaque fonction *Valeur* d'action). Cette nouvelle approche requière un nouvel algorithme de combinaison d'RGFORs qui, au lieu de produire une nouvelle RGFOR en sortie, modifie une des deux RGFORs combinées.

Apprentissage en ligne

Dans l'état de l'art de l'architecture **SDYNA**, l'algorithme **SPITI** permettait d'apprendre à partir des observations récoltées le modèle d'un FMDP à l'aide d'ARBRES DE DÉCISION comme structure de données pour représenter les fonctions *Transition* et *Récompense*. L'algorithme d'apprentissage incrémentale **ITI** était donc chargé de bâtir les différents arbres requis pour apprendre le modèle au fur et à mesure que les observations arrivaient. Puis une version modifiée de l'algorithme **SVI**, n'utilisant que la boucle principale et itérant un nombre limité de fois, planifiait une *Politique* optimale d'après le modèle couramment estimé. Le principal problème de cette approche était de se reposer sur des ARBRES DE DÉCISION pour apprendre le modèle, et par conséquent, sur **SVI** pour la planification. Or, cette structure est moins performante pour modéliser

19. Donc pour les actions $BuyC$, $DelC$ et $GetU$

le problème et la planification que les RGFORs. Nous nous sommes donc naturellement intéressés à créer une instance de **SDYNA** utilisant les RGFORs pour modéliser le problème.

Contribution

Pour y parvenir, nous avons dû proposer un algorithme d'apprentissage incrémental de RGFORs multimodales. En effet, pour être utilisable dans une architecture **SDYNA**, et permettre donc de faire de l'apprentissage par renforcement, il manquait aux RGFORs un algorithme d'apprentissage incrémental. L'algorithme **IMDDI** proposé permet de le faire. Cet algorithme, comme l'algorithme **ITI** avec les ARBRES DE DÉCISION, maintient à jour les RGFORs apprises en accord avec les observations reçues petit à petit.

Ce algorithme permet la mise en place de l'instance **SPIMDDI** de **SDYNA**. Cette instance utilise l'algorithme **IMDDI** pour l'apprentissage incrémental, et l'algorithme **SPUMDD** pour la planification. Les expérimentations que nous avons menées nous ont permis de constater que cette nouvelle instance apprend des modèles plus compactes que **SPITI** sans pour autant ni dégrader la qualité de la *Politique* optimale estimée ni affecter la manière dont l'espace des couples état-action est exploré.

Perspectives

Dans le Chapitre 7, nous avons évoqué plusieurs axes de recherche autour de **SPIMDDI**, et de l'architecture **SDYNA** plus généralement. Nous les rappelons ici succinctement.

Vers un algorithme PAC-FMDP Comme nous l'avons évoqué au Chapitre 6, il n'y pas de preuve de convergence pour l'algorithme **SPIMDDI** en lui-même. Néanmoins quelques adaptations peuvent permettre d'y parvenir. D'une part, il s'agirait de rendre l'algorithme **IMDDI** PAC. Pour se faire, la mesure de sélection des variables en chaque nœud est probablement à changer avec une mesure qui permette une preuve de convergence. Le pré-élagage fait est aussi probablement à remplacer par du post-élagage qui devra être fait avant toute fusion des feuilles. D'autre part, il est nécessaire de remplacer l'algorithme de gestion du compromis exploration-exploitation ϵ-**GREEDY**. En effet, celui-ci ne permet pas non plus de preuve de convergence. La version de l'algorithme R-Max que nous proposons en Section 7.1.2 parait néanmoins prometteuse.

Gestion des états impossibles L'étude sur Starcraft nous a permis de voir que IMDDI ne gère pas correctement les états impossibles. Ainsi, pour les fonctions *Transition*, pour tout CONTEXTE impossible, IMDDI cherche à associer une distribution de probabilité de transition pour la variable étudié comme pour n'importe quel autre CONTEXTE. Néanmoins, ce CONTEXTE étant impossible, aucune observation n'est acquise permettant d'établir la distribution. Il en résulte que IMDDI confère une probabilité de transition nulle sur toutes les modalités de la variable étudiée. En conséquent, la feuille obtenue ne contient pas une distribution de probabilité, ce qui fausse les calculs de planification effectué par la suite. Toutefois, nous disposons ainsi d'un moyen de détecter les CONTEXTES impossibles. Cet outil de détection serait utile si une méthode de gestion de ces CONTEXTES lors de la planification existait.

Apprentissage par décomposition additive IMDDI apprend une fonction *Récompense* qui explique l'ensemble des récompenses collectées pour chaque action. Toutefois, bien souvent, cette fonction résulte de l'addition de plusieurs plus sous-fonctions. Par exemple, dans COFFEE ROBOT la fonction récompense est composée d'une récompense suivant que le robot est mouillé ou non, et d'une récompense suivant que le propriétaire ait eu son café ou non. Dans le cas de problème standard la fonction obtenue reste simple et compréhensible. Néanmoins, comme le montre l'exemple de Starcraft, la fonction obtenue est potentiellement plus complexe et plus délicate à interpréter. De plus, il n'est pas impossible que la fonction composée soit particulièrement complexe, rendant sa manipulation lors de la planification difficile. Pour ces deux raisons, la recherche de méthodes d'apprentissage par décomposition additive se justifie.

Gestion du compromis exploration-exploitation et états dangereux Notre expérimentation de ADAPTIVE F-RMAX sur Starcraft a soulevé un autre problème : la gestion du compromis exploration-exploitation en présence d'états dangereux pour le système. En effet, certains algorithmes de gestion du compromis, à l'instar de ADAPTIVE F-RMAX, donne un bonus aux CONTEXTES nouvellement découverts afin de planifier de les explorer plus avant. Un bonus sera apporté aux états nouvellement découverts, même si ceux-ci représentent un danger pour l'agent ou le système. Par conséquent, l'agent ira naturellement vers ces états dangereux. De fait une nouvelle composante est ajouté au problème d'exploration en espace dangereux : l'algorithme d'exploration apporte un bonus aux états dangereux.

Bibliographie

Ronald A. Howard. Dynamic programming and markov processes, 1960.

Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition, 1957.

M.L. Puterman. *Markov decision processes : discrete stochastic dynamic programming*. Wiley series in probability and statistics. Wiley-Interscience, 2005. ISBN 9780471727828.

R. Bellman. *Adaptive Control Processes*. Princeton University Press, 1961.

Craig Boutilier, Thomas Dean, and Steve Hanks. Decision-theoretic planning : Structural assumptions and computational leverage. *JAIR*, 11 :1–94, 1999.

Carlos Guestrin and Geoffrey Gordon. Distributed planning in hierarchical factored mdps. In *In Proceedings if the Eighteenth Conference on Uncertainty in Artificial Intelligence*, pages 197–206, 2002.

Craig Boutilier. Symbolic dynamic programming for first-order mdps. In *In IJCAI*, pages 690–700. Morgan Kaufmann, 2001.

Thomas Degris, Olivier Sigaud, and Pierre-Henri Wuillemin. Chi-square Tests Driven Method for Learning the Structure of Factored MDPs. In *Proceedings of the 22nd conference on Uncertainty in Artificial Intelligence*, pages 122–129. AUAI Press, 2006a.

Er L. Strehl, Carlos Diuk, and Michael L. Littman. Efficient structure learning in factored-state mdps. In *Proceedings of the Twenty-Second National Conference on Artificial Intelligence (AAAI-07*, 2007.

Doran Chakraborty and Peter Stone. Structure learning in ergodic factored mdps without knowledge of the transition function's in-degree. In *Proceedings of the Twenty Eighth International Conference on Machine Learning (ICML'11)*, June 2011.

Richard Dearden and Craig Boutilier. Abstraction and approximate decision-theoretic planning. *Artificial Intelligence*, 89(1–2) :219 – 283, 1997. ISSN 0004-3702. doi : http://dx.doi.org/10.1016/S0004-3702(96)00023-9.

Thomas G. Dietterich. The MAXQ method for hierarchical reinforcement learning. In Jude W. Shavlik, editor, *Proceedings of the Fifteenth International Conference on Machine Learning (ICML 1998), Madison, Wisconsin, USA, July 24-27, 1998*, pages 118–126. Morgan Kaufmann, 1998.

A. A. Markov. The theory of algorithms. *Journal of Symbolic Logic*, 18(4) :340–341, 1953.

Anthony R Cassandra, Leslie Pack Kaelbling, and Michael L Littman. Acting optimally in partially observable stochastic domains. In *AAAI*, volume 94, pages 1023–1028, 1994.

Ronald Edward Parr. Hierarchical control and learning for markov decision processes, 1998.

Milos Hauskrecht, Nicolas Meuleau, Leslie Pack Kaelbling, Thomas Dean, and Craig Boutilier. Hierarchical solution of markov decision processes using macro-actions. In *In Proc. of Uncertainty in Artificial Intelligence (UAI*, pages 220–229, 1998.

Thomas G Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *J. Artif. Intell. Res.(JAIR)*, 13 :227–303, 2000.

Chenggang Wang, Saket Joshi, and Roni Khardon. First order decision diagrams for relational mdps. In *In Proceedings of the International Joint Conference of Artificial Intelligence*, pages 1095–1100, 2007.

Steve Hanks Drew Mcdermott, Steve Hanks, and Drew Mcdermott. Modeling a dynamic and uncertain world i : Symbolic and probabilistic reasoning about change. *Artificial Intelligence*, 66 :1–55, 1993.

Craig Boutilier. Correlated action effects in decision theoretic regression. In *uai97*, pages 30–37, 1997.

Kevin Patrick Murphy. Dynamic bayesian networks : Representation, inference and learning, 2002.

Thomas Dean and Keiji Kanazawa. A model for reasoning about persistence and causation. *Comput. Intell.*, 5(3) :142–150, dec 1989. ISSN 0824-7935.

Carlos Guestrin, Daphne Koller, Ronald Parr, and Shobha Venkataraman. Efficient solution algorithms for factored mdps. *Journal of Artificial Intelligence Research (JAIR)*, 19 :399–468, 2003.

Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Exploiting structure in policy construction. In *IJCAI-95, pp.1104–1111*, 1995.

Jesse Hoey, Robert St-aubin, Alan Hu, and Craig Boutilier. Spudd : Stochastic planning using decision diagrams. In *In Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, pages 279–288. Morgan Kaufmann, 1999.

R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications, 1993.

Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35 :677–691, 1986.

S. J. Friedman and K. J. Supowit. Finding the optimal variable ordering for binary decision diagrams. *IEEE Trans. Comput.*, 39(5) :710–713, may 1990. ISSN 0018-9340. doi : 10.1109/12.53586.

Rolf Drechsler and Bernd Becker. *Binary Decision Diagrams - Theory and Implementation.* Springer, 1998. ISBN 978-0-7923-8193-8.

Ron Kohavi. Bottom-up induction of oblivious read-once decision graphs. In *Machine Learning : ECML-94*, pages 154–169. Springer, 1994a.

Ron Kohavi and Chia-Hsin Li. Oblivious decision trees, graphs, and top-down pruning. In *IJCAI*, pages 1071–1079. Citeseer, 1995.

Jonathan Oliver. *Decision graphs : an extension of decision trees.* Citeseer, 1992.

Christopher S Wallace and David M Boulton. An information measure for classification. *The Computer Journal*, 11(2) :185–194, 1968.

Arlindo Oliveira and Alberto Sangiovanni-Vincentelli. Inferring reduced ordered decision graphs of minimal description length. In *Proceedings of the 12th International Conference on Machine Learning (ML95), Morgan Kaufmann, San Francisco.* Citeseer, 1995.

Arlindo L Oliveira and Alberto Sangiovanni-Vincentelli. Using the minimum description length principle to infer reduced ordered decision graphs. *Machine Learning*, 25 (1) :23–50, 1996.

Arvind Srinivasan, Timothy Kam, Sharad Malik, and Robert K. Brayton. Algorithms for discrete function manipulation. In *ICCAD'90*, pages 92–95, 1990.

Masahiro Fujita, Hisanori Fujisawa, and Nobuaki Kawato. Evaluation and improvements of boolean comparison method based on binary decision diagrams. In *Int'l Conf. on CAD*, pages 2–5. Citeseer, 1988.

Sharad Malik, Albert R Wang, Robert K Brayton, and Alberto Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *Computer-Aided Design, 1988. ICCAD-88. Digest of Technical Papers., IEEE International Conference on*, pages 6–9. IEEE, 1988.

Shin-ichi Minato, Nagisa Ishiura, and Shuzo Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE*, pages 52–57. IEEE, 1990.

Masahiro Fujita, Yusuke Matsunaga, and Taeko Kakuda. On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In *Proceedings of the conference on European design automation*, pages 50–54. IEEE Computer Society Press, 1991.

Nagisa Ishiura, Hiroshi Sawada, and Shuzo Yajima. Minimazation of binary decision diagrams based on exchanges of variables. In *ICCAD*, volume 91, pages 472–475, 1991.

Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 42–47. IEEE Computer Society Press, 1993.

Robert St-Aubin, Jesse Hoey, and Craig Boutilier. Apricodd : Approximate policy construction using decision diagrams. In *NIPS*, pages 1089–1095, 2000.

J.R. Norris. *Markov Chains*. Number n 2008 in Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 1998. ISBN 9780521633963.

Nicholas J Higham. *Accuracy and stability of numerical algorithms*. Siam, 2002.

George B. Dantzig. *Linear programming and extensions.* Rand Corporation Research Study. Princeton Univ. Press, Princeton, NJ, 1963.

Olivier Sigaud and Olivier Buffet. *Processus décisionnels de Markov en intelligence artificielle*, volume 1 - principes généraux et applications of *IC2 - informatique et systèmes d'information.* Lavoisier - Hermes Science Publications, 2008.

Stefan Banach. Sur les opérations dans les ensembles abstraits et leur application aux équations intégrales. *Fundamenta Mathematicae*, 3(1) :133–181, 1922.

Ronald J Williams and Leemon C Baird. Tight performance bounds on greedy policies based on imperfect value functions. Technical report, Citeseer, 1993.

R.S. Sutton and A.G. Barto. *Reinforcement Learning : An Introduction.* A Bradford book. Bradford Book, 1998. ISBN 9780262193986.

Eric A. Hansen and Shlomo Zilberstein. Lao-star : A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129(1–2) :35 – 62, 2001. ISSN 0004-3702. doi : http://dx.doi.org/10.1016/S0004-3702(01)00106-0.

H Brendan McMahan, Maxim Likhachev, and Geoffrey J Gordon. Bounded real-time dynamic programming : Rtdp with monotone upper bounds and performance guarantees. In *Proceedings of the 22nd international conference on Machine learning*, pages 569–576. ACM, 2005.

Martin L Puterman and Moon Chirl Shin. Modified policy iteration algorithms for discounted markov decision problems. *Management Science*, 24(11) :1127–1137, 1978.

Alan S Manne. Linear programming and sequential decisions. *Management Science*, 6 (3) :259–267, 1960.

Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1) :49–107, 2000.

Fabio Somenzi. CUDD : BDD package, University of Colorado, Boulder. http://vlsi.colorado.edu/~{}fabio/CUDD/.

Jean-Christophe Magnan and Pierre-Henri Wuillemin. Improving Decision Diagrams for Decision Theoretic Planning. In *The Twenty-Sixth International FLAIRS Conference*, pages 621–626. The AAAI Press, May 2013a.

Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1st edition, 1996. ISBN 1886529108.

Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1) :9–44, 1988.

Gavin A Rummery and Mahesan Niranjan. On-line q-learning using connectionist systems. 1994.

Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4) : 279–292, 1992.

Ian H Witten. An adaptive optimal controller for discrete-time markov environments. *Information and control*, 34(4) :286–295, 1977.

Andrew G Barto, Richard S Sutton, and Charles W Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *Systems, Man and Cybernetics, IEEE Transactions on*, (5) :834–846, 1983.

Richard S. Sutton. 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, 1990.

Sham Machandranath Kakade et al. *On the sample complexity of reinforcement learning*. PhD thesis, University of London, 2003.

Leslie G Valiant. A theory of the learnable. *Communications of the ACM*, 27(11) : 1134–1142, 1984.

Michael Kearns and Satinder Singh. Near-optimal reinforcement learning in polynomial time. *Machine Learning*, 49(2-3) :209–232, 2002.

Ronen I Brafman and Moshe Tennenholtz. R-max-a general polynomial time algorithm for near-optimal reinforcement learning. *The Journal of Machine Learning Research*, 3 :213–231, 2003.

Alexander L Strehl and Michael L Littman. A theoretical analysis of model-based interval estimation. In *Proceedings of the 22nd international conference on Machine learning*, pages 856–863. ACM, 2005.

Thomas Degris, Olivier Sigaud, and Pierre-Henri Wuillemin. Learning the Structure of Factored Markov Decision Processes in Reinforcement Learning Problems. In

Proceedings of the 23rd International Conference on Machine Learning, pages 257–264. ACM, 2006b.

Paul E. Utgoff, Neil C. Berkman, and Jeffery A. Clouse. Decision tree induction based on efficient tree restructuring. *Machine Learning*, 29(1) :5–44, 1997. ISSN 0885-6125. doi : 10.1023/A:1007413323501.

L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984a.

J. Ross Quinlan. *C4.5 : Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. ISBN 1-55860-238-0.

Allan P. White and Wei Zhong Liu. Technical note : Bias in information-based measures in decision tree induction. *Mach. Learn.*, 15(3) :321–329, jun 1994. ISSN 0885-6125. doi : 10.1023/A:1022694010754.

Jonathan J. Oliver. Decision graphs - an extension of decision trees, 1993.

Arlindo L. Oliveira and Alberto Sangiovanni-Vincentelli. Inferring reduced ordered decision graphs of minimal description length. In *proceedings of the 12th international conference on machine learning*, pages 421–429. Morgan Kaufmann, 1994.

Ron Kohavi. Bottom-up induction of oblivious read-once decision graphs. In Francesco Bergadano and Luc Raedt, editors, *Machine Learning : ECML-94*, volume 784 of *Lecture Notes in Computer Science*, pages 154–169. Springer Berlin Heidelberg, 1994b. ISBN 978-3-540-57868-0. doi : 10.1007/3-540-57868-4_56.

Ron Kohavi and Chia hsin Li. Oblivious decision trees, graphs, and top-down pruning. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1071–1077. Morgan Kaufmann, 1995.

Jean-Christophe Magnan and Pierre-Henri Wuillemin. On-line Learning of Multi-valued Decision Diagrams. In *Proceedings of the Twenty-Eighth International Florida Artificial Intelligence Research Society Conference*, pages 576–580, May 2015a. Best student paper Award.

Jean-Christophe Magnan and Pierre-Henri Wuillemin. Imddi et spimddi : apprentissage incrémental de diagrammes de décisions pour une architecture sdyna. In *Actes JFPDA*, pages 45–60, 2015b.

John Mingers. An empirical comparison of selection measures for decision-tree induction. *Machine Learning*, 3(4) :319–342, 1989. ISSN 0885-6125. doi : 10.1007/BF00116837.

Ted Dunning. Accurate methods for the statistics of surprise and coincidence. *Comput. Linguist.*, 19(1) :61–74, mar 1993.

Thomas Degris. *Apprentissage par renforcement dans les processus de décision Markoviens factorisés*. PhD thesis, 2007. Thèse de doctorat dirigée par Sigaud, Olivier Informatique Paris 6 2007.

L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth International Group, Belmont, CA, 1984b.

A.L. Strehl. Model-based reinforcement learning in factored-state mdps. In *Approximate Dynamic Programming and Reinforcement Learning, 2007. APRL 2007. IEEE International Symposium on*, pages 103–110, April 2007. doi : 10.1109/ADPRL.2007.368176.

Michael Kearns and Daphne Koller. Efficient reinforcement learning in factored mdps. In *IJCAI*, volume 16, pages 740–747, 1999.

Carlos Diuk, Lihong Li, and Bethany R Leffler. The adaptive k-meteorologists problem and its application to structure learning and feature selection in reinforcement learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 249–256. ACM, 2009.

Hyunsoo Kim and Gary J. Koehler. Pac-learning a decision tree with pruning. *European Journal of Operational Research*, 94(2) :405 – 418, 1996. ISSN 0377-2217. doi : http://dx.doi.org/10.1016/0377-2217(95)00174-3.

Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2) :100–107, 1968.

Olga Kozlova, Olivier Sigaud, Pierre-Henri Wuillemin, and Christophe Meyer. Considering unseen states as impossible in factored reinforcement learning. In *Machine Learning and Knowledge Discovery in Databases*, pages 721–735. Springer, 2009.

Alexander Hans, Daniel Schneegaß, Anton Maximilian Schäfer, and Steffen Udluft. Safe exploration for reinforcement learning. In *ESANN*, pages 143–148, 2008.

Teodor Mihai Moldovan and Pieter Abbeel. Safe exploration in markov decision processes. *arXiv preprint arXiv :1205.4810*, 2012.

Leonard A Breslow and David W Aha. Simplifying decision trees : A survey. *The Knowledge Engineering Review*, 12(01) :1–40, 1997.

Branislav Kveton. *Planning in hybrid structured stochastic domains*. PhD thesis, University of Pittsburgh, 2006.

Michael L Littman, Thomas L Dean, and Leslie Pack Kaelbling. On the complexity of solving markov decision problems. In *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence*, pages 394–402. Morgan Kaufmann Publishers Inc., 1995.

Jean-Christophe Magnan and Pierre-Henri Wuillemin. Improving Decision Diagrams for Decision Theoretic planning. In *Florida Artificial Intelligence Research Society Conference*, pages 621–626, may 2013b.

Alexander L Strehl and Michael L Littman. An analysis of model-based interval estimation for markov decision processes. *Journal of Computer and System Sciences*, 74(8) :1309–1331, 2008.