Apprentissage de représentation et auto-organisation modulaire pour un agent autonome

Bruno Scherrer

Département de formation doctorale en informatique **École doctorale IAEM Lorraine**
UFR STMIA

Apprentissage de représentation et auto-organisation modulaire pour un agent autonome

THÈSE

présentée et soutenue publiquement le 6 janvier 2003

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1
(spécialité informatique)

par

Bruno SCHERRER

Composition du jury

Président :	René Schott, Professeur, Université Henri Poincaré - Nancy I.
Rapporteurs :	Leslie Pack Kaelbling, Professor of Computer Science and Engineering, MIT AI Lab. Raja Chatila, Directeur de Recherche, CNRS.
Directeurs :	Frédéric Alexandre, Directeur de Recherche, INRIA. François Charpillet, Directeur de Recherche, INRIA.

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Résumé

Cette thèse étudie l'utilisation d'algorithmes connexionnistes pour résoudre des problèmes d'apprentissage par renforcement. Les algorithmes connexionnistes sont inspirés de la manière dont le cerveau traite l'information : ils impliquent un grand nombre d'unités simples fortement interconnectées, manipulant des informations numériques de manière distribuée et massivement parallèle. L'apprentissage par renforcement est une théorie computationnelle qui permet de décrire l'interaction entre un agent et un environnement : elle permet de formaliser précisément le problème consistant à atteindre un certain nombre de buts via l'interaction.

Nous avons considéré trois problèmes de complexité croissante et montré qu'ils admettaient des solutions algorithmiques connexionnistes : 1) L'apprentissage par renforcement dans un petit espace d'états : nous nous appuyons sur un algorithme de la littérature pour construire un réseau connexionniste ; les paramètres du problème sont stockés par les poids des unités et des connexions et le calcul du plan est le résultat d'une activité distribuée dans le réseau. 2) L'apprentissage d'une représentation pour approximer un problème d'apprentissage par renforcement ayant un grand espace d'états : nous automatisons le procédé consistant à construire une partition de l'espace d'états pour approximer un problème de grande taille. 3) L'auto-organisation en modules spécialisés pour approximer plusieurs problèmes d'apprentissage par renforcement ayant un grand espace d'états : nous proposons d'exploiter le principe "diviser pour régner" et montrons comment plusieurs tâches peuvent être réparties efficacement sur un petit nombre de modules fonctionnels spécialisés.

Mots-clés: intelligence artificielle, apprentissage par renforcement, connexionnisme, processus décisionnels de Markov

Abstract

This thesis studies the use of connectionist algorithms for solving reinforcement learning problems. Connectionist algorithms are inspired by the way information is processed by the brain : they rely on a large network of highly interconnected simple units, which process numerical information in a distributed and massively parallel way. Reinforcement learning is a computational theory that describes the interaction between an agent and an environment : it enables to precisely formalize goal-directed learning from interaction.

We have considered three problems, with increasing complexity, and shown that they can be solved with connectionist algorithms : 1) Reinforcement learning in a small state space : we exploit a well-known algorithm in order to build a connectionist network : problem paramaters are stored into weighted units and connections and planning is the result of a distributed activity in the network. 2) Learning a representation for approximating a reinforcement learning problem with a large state space : we provide an algorithm for automatically building a state space partition in order to approximate a large problem. 3) Self-organization of specialized modules for approximating various reinforcement problems with a large state space : we exploit a "divide and conquer" approach and show that various tasks can efficiently be spread over a little number of specialized functional modules.

Keywords: artificial intelligence, reinforcement learning, connexionism, Markov decision processes

Remerciements

Je voudrais ici remercier tous ceux qui, de près ou de loin, ont contribué à la concrétisation de cette thèse.

Je pense tout d'abord à Leslie Kaelbling, Raja Chatila et René Schott qui ont accepté de rapporter ce travail. Je suis d'autant plus reconnaissant de la qualité de leurs lectures, de leurs remarques, et de leurs nombreuses suggestions que le calendrier que je leur ai imposé ne leur a probablement pas rendue la tâche très facile.

Je remercie Frédéric Alexandre et François Charpillet de m'avoir accueilli dans les équipes CORTEX et MAIA, soutenu à maintes reprises, et fait confiance pendant ces trois années. Je pense également à tous les membres des équipes CORTEX et MAIA qui ont toujours été présents quand il le fallait. Je remercie particulièrement Yann, Hervé, Nicolas, Laurent, Alain et Olivier pour leurs multiples relectures.

Je tiens à rendre hommage à Martine Kuhlmann et à Nadine Beurné qui, par leur aide permanente, infaillible et consciencieuse, ont joué un rôle certain dans le bon déroulement de ces trois années et des derniers jours précédant la soutenance.

Pour finir, je salue tous ceux qui, par leurs conseils, leur bonne humeur, et leur présence, ont enrichi les années qui viennent de s'écouler : Régine, Stephanie, Yann, Alain, Laurent, Nicolas & Sophie, Evelyne, Bernard, Nicolas & Estelle, Marie-Pierre, Michelle, Béa, Etienne, François, Nicolas C., Mathieu, Oto, Julien, Khalid, Hervé, Ferran, Bertrand M., Christophe A., Vincent M., Vincent T., Christophe L., Jean-Charles et j'en oublie sûrement (qu'ils me pardonnent !)...

Table des matières

Table des figures

Introduction

Un cerveau est pour bien des tâches largement plus *performant* qu'un ordinateur. Considérons par exemple le traitement d'informations visuelles : un bébé d'un an est bien plus rapide et précis pour reconnaître des objets, des visages, que le meilleur système de reconnaissance visuelle d'intelligence artificielle tournant sur l'ordinateur parallèle le plus rapide [Hertz *et al.*, 1991]. Ce constat est aujourd'hui encore d'actualité.

Plus généralement, tout cerveau animal ou humain a des caractéristiques intéressantes. Un cerveau est :

- **autonome** : il permet l'adaptation à des situations inconnues par apprentissage. En particulier il n'a pas besoin d'être programmé ;
- **robuste** : il traite des informations floues, probabilistes et bruitées ;
- **multi-tâche** : il peut gérer plusieurs objectifs ; il n'est pas spécifique à un problème particulier ;
- **anytime** : la qualité des réponses augmente avec le temps alloué pour répondre ;
- **dynamique** : la plupart des traitements n'ont ni début, ni fin. Ils s'inscrivent dans une interaction continue avec un milieu variable ;
- **tolérant aux pannes** : des cellules nerveuses meurent tous les jours sans que cela affecte significativement les performances ;
- **massivement parallèle** : c'est un réseau d'unités relativement simples (les neurones) fonctionnant en parallèle.

Cette thèse tente de transposer ces propriétés dans un système informatique. Pour ce faire, le travail présenté dans ce mémoire s'appuie particulièrement sur deux composantes actuelles de l'intelligence artificielle (IA) :

- L'**apprentissage par renforcement** (A/R) : il s'agit d'une théorie computationnelle générique qui intègre les qualités **autonomie** et **robustesse** à l'aide d'un formalisme mathématique : les **processus décisionnels de Markov** (PDM).
- Le **connexionnisme** : ce domaine propose une solution de remplacement au modèle computationnel traditionnel de Von Neumann en s'inspirant du paradigme de calcul distribué du cerveau. Entre autres, il s'intéresse à la construction et l'étude d'algorithmes **massivement parallèles** et **tolérants aux pannes**.

Contexte scientifique

Avant de rentrer dans les détails de nos travaux, nous proposons aux lecteurs une courte présentation de l'A/R et du connexionnisme. Nous en profitons au passage pour situer notre problématique par rapport à celles des équipes CORTEX et MAIA du Loria, cadre de travail de cette thèse.

L'intelligence artificielle

Le but originel de l'intelligence artificielle (IA), domaine né dans les années 50, était de construire des systèmes informatiques qui égaleraient voire dépasseraient l'homme dans de nombreuses activités réputées intelligentes comme raisonner, utiliser le langage ou résoudre des problèmes. Les résultats n'ayant jamais atteint ces espérances, ce domaine a aujourd'hui pour objectif plus raisonnable de construire des systèmes informatiques intelligents dans une acception très générale.

Le domaine de l'IA comporte de nombreuses branches que l'on peut grossièrement répartir en deux catégories : les *problèmes* et les *techniques*. Les problèmes sont plutôt des domaines d'application alors que les techniques se veulent plus des méthodes abstraites et génériques. Les figures 1 et 2 proposent une liste représentative des problèmes et des techniques de l'IA (notre présentation est ici fortement inspirée de [Crabbe *et al.*, 1997]).

La diversité de ce domaine fait qu'il est difficile de le cerner. On peut combiner des problèmes avec des techniques et ainsi faire la distinction entre le traitement du langage naturel à l'aide de réseaux de neurones artificiels et le traitement du langage naturel à l'aide de modèles stochastiques. On peut combiner des techniques et par exemple utiliser des réseaux de neurones pour coder des probabilités conditionnelles de modèles stochastiques. Enfin, on peut combiner les problèmes : la reconnaissance de formes peut être vue comme une forme d'apprentissage automatique.

Il est également usuel de voir l'ensemble de ces branches sous deux angles : l'IA classique et l'IA numérique. L'IA classique exploite des informations généralement codées sous forme symbolique et est réputée pour les raisonnements *déductifs* (du général vers le particulier). A contrario, l'IA numérique traite des informations de nature numérique et connaît de grands succès dans les raisonnements *inductifs* (du particulier vers le général). Il est généralement admis que pour qu'un système soit vraiment intelligent, il doive disposer à la fois de facultés déductives et inductives, de capacités symboliques et numériques.

A ses débuts, l'IA était considérée comme entièrement séparée de disciplines de la recherche opérationnelle telles que la théorie du contrôle ou encore les statistiques. L'IA concernait alors la logique, les symboles mais pas les nombres. Les travaux reposaient essentiellement sur des programmes en LISP et il n'y avait ni algèbre linéaire, ni équations différentielles, ni statistiques. Depuis une vingtaine d'année, le domaine a progressivement évolué et s'est ouvert aux méthodes numériques. Les chercheurs de l'IA moderne acceptent les approches statistiques ou les algorithmes de contrôle comme des méthodes alternatives, ou simplement comme des outils utiles à leurs besoins.

La recherche opérationnelle implique une méthodologie particulièrement rigoureuse : elle essaie d'obtenir des résultats précis sur des problèmes très contraints. L'IA des débuts était souvent plus aventureuse : elle étudiait des problèmes plus complexes d'une manière forcément plus expérimentale. A égale distance entre ces deux domaines se situent aujourd'hui des champs d'études passionnants : ils abordent des problèmes complexes et les étudient de manière rigoureuse. Les deux domaines qui concernent cette thèse, l'A/R et le connexionnisme, en font partie.

L'apprentissage par renforcement

L'introduction de l'A/R que nous faisons ici est très fortement inspirée de celle qu'on trouve dans [Sutton et Barto, 1998]. Nous conseillons au lecteur qui souhaiterait approfondir ce domaine d'une manière générale de consulter directement cet ouvrage de présentation à la fois générale et précise. Une description plus formelle/mathématique des problèmes de ce domaine peut être

Programmation automatique	Réalisation de programmes à partir de la description des fonctions à remplir. Sorte de "supercompilateurs", ces systèmes doivent automatiquement générer des programmes à partir d'instructions écrites dans un langage proche du langage naturel.
Résolution de problèmes	Résolution de problèmes à base de combinatoire comme les jeux de réflexion. Le travail consiste à modérer au maximum l'"explosion combinatoire" à l'aide entre autres d'heuristiques.
Démonstration automatique de théorèmes	Automatisation totale ou partielle de démonstration de théorèmes mathématiques.
Représentation et extraction des connaissances	Pour être interprétées par un système informatique, les connaissances générales et diverses doivent être converties en données. Deux problèmes fondamentaux se posent : quelle forme doivent prendre les données et comment les extraire ?
Fouille de données	Intimement liée à la représentation et à l'extraction des connaissances, il s'agit ici de traiter des bases de données de très grandes tailles. Ceci implique entre autres de faire des classifications, des segmentations (ou clusters), des raisonnements à partir de cas.
Traitement automatique du langage naturel	Problèmes de traduction automatique (cet objectif pose aujourd'hui encore beaucoup de problèmes et on parle plutôt d'assistance à la traduction) et d'interprétation automatique du langage.
Reconnaissance de formes	Transformation du langage parlé (information auditive) en texte écrit, reproduction artificielle du processus de vision humain par une machine.
Robotique autonome	Intersection de l'IA et de la robotique traditionnelle, cette branche vise à construire des robots (mobiles) qui se comportent intelligemment.
Apprentissage automatique (Machine Learning)	Construction de systèmes informatiques qui sont capables d'apprendre à partir d'exemples et/ou d'expériences propres.

FIG. 1: Quelques grands problèmes de l'intelligence artificielle

Systèmes à base de connaissances	Encore appelés systèmes experts, ils constituent des systèmes de diagnostic de haut niveau qui sont capables d'égaler l'expert humain d'un domaine précis.
Modèles stochastiques	Ensemble de techniques qui permettent de structurer et d'exploiter des informations de nature probabiliste.
Réseaux de neurones artificiels	Etude de programmes qui sont structurés et qui fonctionnent à la manière des cerveaux humain et animal.
Planification	Techniques consistant à dériver une séquence de décisions afin d'atteindre un but à partir d'une position initiale et d'un ensemble d'actions élémentaires.

FIG. 2: Quelques grandes techniques de l'intelligence artificielle

consultée dans [Bertsekas et Tsitsiklis, 1996].

Un domaine informatique

L'idée que l'on *apprend en interagissant* avec notre environnement est probablement la première qui vienne à l'esprit lorsqu'on pense à la nature du phénomène d'apprentissage. Quand un enfant joue, remue ses bras, regarde autour de lui, il n'a pas de professeur, mais il est en relation sensori-motrice directe avec son environnement. Il peut observer, mémoriser les conséquences de ses actes, et utiliser toutes ces informations pour agir de manière autonome. Tout au long de la vie, ce genre d'interactions est une source majeure de connaissances à propos de notre environnement et de nous-même. Quand nous apprenons à conduire une voiture ou à mener une conversation, nous sommes conscients de la façon avec laquelle notre environnement répond à ce que nous faisons ; en adaptant notre comportement, nous pouvons avoir plus ou moins d'influence sur ce qui va se passer.

Les chercheurs de l'A/R étudient un *modèle computationnel* de l'apprentissage via l'interaction : plutôt que de modéliser directement la façon dont les hommes et les animaux apprennent, ils étudient des situations idéalisées et simplifiées afin de trouver et d'évaluer des méthodes informatiques d'apprentissage. C'est une perspective fondamentalement informatique. Le but est de construire des architectures informatiques pour résoudre des problèmes d'apprentissage d'intérêt scientifique. L'évaluation de ces constructions se fait à l'aide d'analyses mathématiques ou d'expériences computationnelles. Celle-ci permet de répondre aux questions suivantes : quel problème résoud-on ? avec quelle précision ? avec quelle vitesse ? Par rapport aux autres problèmes abordés par la branche "apprentissage automatique" de l'IA, l'A/R se concentre particulièrement sur l'*apprentissage par l'interaction afin d'atteindre des buts.*

Caractéristiques fondamentales de l'apprentissage par renforcement

Apprendre par renforcement, c'est apprendre quoi faire (comment réagir face à une situation) de sorte à maximiser un signal numérique de récompense. En particulier :

- On ne dit pas à l'apprenant quelles actions il doit faire, comme dans la plupart des paradigmes de l'apprentissage automatique, mais il doit découvrir par lui même quelles sont les actions qui lui fournissent le maximum de récompenses en les essayant. Un agent apprenant

par renforcement progresse par *essais et erreurs*.

- L'interaction entre l'apprenant et l'environnement a un effet sur la récompense immédiate mais aussi sur les prochaines situations possibles et par conséquent, sur toutes les récompenses qui suivent. Le caractère retardé des récompenses pose le problème de déterminer le rôle d'une action particulière dans la globalité d'un comportement ; on parle souvent du *credit assignment problem*.

Le domaine de l'A/R n'est pas caractérisé par la nature des algorithmes qui sont mis en jeu mais par un *problème d'apprentissage*. Tout algorithme qui répond bien à ce problème est considéré comme un algorithme de l'A/R. Nous détaillerons formellement ce problème dans la première partie de ce manuscrit. Nous en donnons seulement ici une idée intuitive : un agent doit interagir avec un environnement pour atteindre un but. Pour ce faire, l'agent doit être capable :

- de *percevoir l'état de l'environnement*, au moins dans une certaine mesure
- d'*agir*, d'exécuter des actions qui font évoluer l'état de l'environnement.

L'agent doit également avoir un ou plusieurs *buts* qui sont liés à l'état de l'environnement. La formalisation du problème de l'A/R permet de définir ces trois notions (perception, action, but) d'une manière aussi simple que possible sans toutefois les rendre triviales.

L'A/R est très différent de l'apprentissage supervisé qui est la forme la plus étudiée en apprentissage automatique (en particulier en reconnaissance de formes et dans les réseaux de neurones artificiels). L'apprentissage supervisé suppose qu'un professeur fournit les bonnes réponses au problème posé. C'est une forme fondamentale d'apprentissage, mais elle seule n'est pas adaptée au problème de l'apprentissage par interaction. Dans les problèmes interactifs, il est souvent difficile d'avoir des exemples de comportements qui sont à la fois corrects et représentatifs de toutes les situations que l'agent peut rencontrer. Dans des domaines où l'homme n'est pas expert, c'est-à-dire précisément dans les domaines où l'on attend des résultats de l'apprentissage automatique, un agent doit être capable d'apprendre à partir de sa propre expérience.

L'un des problèmes fondamentaux de l'A/R est le *compromis entre l'exploration et l'exploitation*. Pour obtenir un maximum de récompenses, un agent apprenant par renforcement préfèrera des actions qu'il a essayé dans le passé et qui se sont avérées efficaces pour obtenir des récompenses. Néanmoins, pour découvrir de telles actions, il aura dû accomplir des actions qu'il ne connaissait pas a priori. L'agent doit exploiter ce qu'il connaît déja afin d'obtenir des récompenses, mais il doit également explorer de sorte à faire des choix encore meilleurs dans le futur. Le dilemme exploration/exploitation (ou court-terme/long-terme) fait que si l'agent utilise exclusivement l'un des deux aspects de l'alternative, il a de fortes chances d'avoir une faible quantité de récompenses [Kumar, 1985]. L'agent doit essayer un maximum d'actions et progressivement favoriser celles qui sont les meilleures. Dans des problèmes bruités/stochastiques, chaque action doit être essayée plusieurs fois pour évaluer correctement l'espérance de récompenses associée. Le dilemme exploration/exploitation est propre à l'A/R. On ne le trouve généralement pas dans les autres formes d'apprentissage automatique comme l'apprentissage supervisé.

Une caractéristique centrale de l'A/R est que son formalisme aborde le problème d'atteindre un but via l'interaction dans un environnement bruité comme un *tout*. Ceci est à mettre en contraste avec d'autres approches qui ne s'attaquent qu'à l'un des sous-problèmes sans forcément voir comment il se combine avec les autres. Par exemple, la plus grande part des recherches actuelles en apprentissage automatique concerne l'apprentissage supervisé sans qu'il soit précisément explicité pourquoi une telle capacité est finalement utile. Bien que ces approches donnent lieu à des résultats très utiles, leur focalisation sur un sous-problème bien spécifique constitue une réelle limitation.

L'A/R se veut une réponse à cette limitation. Tout agent apprenant par renforcement a un but explicite, perçoit son environnement et choisit des actions qui influencent cet environnement.

De plus, on suppose souvent qu'un tel agent doit opérer malgré une incertitude significative à propos de son environnement. Quand l'A/R implique la planification, il doit faire l'équilibre entre :

- la planification à proprement parler,
- les prises de décision en temps réel,
- la façon dont un modèle de l'environnement est acquis ou amélioré.

Quand l'apprentissage par renforcement implique une forme d'apprentissage supervisé, c'est dans le but précis de déterminer les caractéristiques de l'environnement qui sont utiles et celles qui ne le sont pas. La démarche de l'A/R n'exclut pas le fait d'isoler des sous-problèmes spécifiques afin de les étudier précisément. Elle insiste néanmoins sur la nécessité de considérer ces sous-problèmes comme partie intégrante d'un tout : un agent ayant des buts, en interaction avec un environnement potentiellement bruité.

Illustrations

Une bonne manière de comprendre l'A/R est de décrire quelques exemples et applications qui ont guidé son développement :

- Un joueur d'échecs joue un coup. Son choix combine à la fois la planification (anticipation d'attaques ou de contre-attaques) et un jugement immédiat et intuitif de la valeur de certaines positions ou de certains coups.
- Un système de contrôle règle automatiquement les paramètres d'une opération de raffinerie de pétrole en temps réel. Ce système optimise le compromis coût/qualité à partir de références suggérées par des experts humains, sans toutefois s'y conformer strictement.
- Une gazelle qui vient de naître se tient sur ses pattes tremblottantes. Une heure plus tard, elle court à 50 km/h.
- Un robot mobile décide s'il va entrer dans une pièce pour rechercher des détritus ou s'il va commencer par retourner vers la zone où il peut se recharger. Il prend cette décision en tenant compte du temps et de la facilité avec lesquels il a atteint la zone de recharge dans le passé.

Ces quatre exemples partagent des caractéristiques basiques. Tous impliquent l'interaction entre un agent qui prend des décisions et un environnement ; l'agent cherche à atteindre un but particulier malgré des incertitudes concernant son environnement. Les décisions de l'agent influent sur le prochain état de l'environnement (par exemple la prochaine disposition des pièces sur l'échiquier, la prochaine position du robot), et donc aussi sur ses opportunités dans le futur. A tout instant, un choix requiert de tenir compte des conséquences à long terme des actions, c'est-à-dire met en jeu une certaine forme de planification.

En même temps, dans chacun de ces exemples, l'effet des actions n'est pas complètement prédictible : l'agent doit fréquemment observer son environnement pour vérifier que tout se passe comme prévu ; éventuellement il peut corriger le tir. Tous ces exemples impliquent des objectifs qui sont explicites. L'agent peut déterminer s'il a atteint un but ou s'il se trouve dans une situation dramatique à partir de la situation courante : le joueur d'échecs peut constater qu'il a fait mat, le robot voit que sa batterie est vide.

Enfin, dans tous ces exemples, l'agent peut utiliser son expérience pour améliorer ses performances. Le joueur d'échecs peut affiner ses intuitions quant à la position actuelle des pièces. Les connaissances que l'agent possède au départ influencent ce qui sera utile ou facile à apprendre ; néanmoins c'est l'interaction avec l'environnement qui lui permettra d'ajuster son comportement pour être efficace.

La recherche en A/R aujourd'hui

La théorie computationnelle sur laquelle se fonde l'A/R permet de bien comprendre les tenants et les aboutissants de l'apprentissage via l'interaction avec un environnement. Des algorithmes "de base" ont été proposés pour des problèmes véritablement idéalisés et simplifiés. Depuis une vingtaine d'années, les efforts de recherche de cette communauté visent à utiliser l'approche A/R pour des problèmes d'IA de plus en plus complexes et réalistes. Quelques résultats impressionnants et ponctuels ont été atteints, par exemple dans les jeux de réflexion : un programme utilisant l'A/R a appris, en jouant contre lui-même, à égaler les meilleurs joueurs humains de back-gammon [Tesauro, 1995]. Dans la même idée, une thèse de l'équipe Maia qui se terminait à mon arrivée au Loria visait à appliquer les techniques de l'A/R à la robotique autonome [Laroche, 2000]. L'optique d'aborder des problèmes de plus en plus complexes et réalistes à l'aide de l'A/R fait partie des objectifs de l'équipe Maia du Loria. Cette thèse a pour ambition de participer à ce mouvement.

Le connexionnisme

Il est saisissant de remarquer que la structure d'un cerveau est fondamentalement différente de celle d'un ordinateur. Ce constat motive depuis longtemps un grand nombre de travaux sur ce qu'on appelle les réseaux de neurones artificiels ou (nous préférerons cette appellation par la suite) le connexionnisme.

Réseaux de neurones, connexionnisme et neuromimétique

Les termes "réseaux de neurones", "connexionnisme", et "neuromimétique"[1] sont parfois l'objet d'amalgames plus ou moins justifiés. Pour commencer, nous reproduisons ci-dessous un extrait de [Touzet, 1992] qui propose un choix pour les sémantiques respectives des termes connexionnisme et réseau de neurones. :

> " [...] les réseaux de neurones artificiels ne désignent que les modèles manipulés ; ce n'est ni un domaine de recherche, ni une discipline scientifique. Connexionnisme et neuromimétique sont tous deux des domaines de recherche à part entière, qui manipulent chacun des modèles de réseaux de neurones artificiels, mais avec des objectifs différents. L'objectif poursuivi par les ingénieurs et chercheurs connexionnistes est d'améliorer les capacités de l'informatique en utilisant des modèles aux composants fortement connectés. Pour leur part, les neuromiméticiens manipulent des modèles de réseaux de neurones artificiels dans l'unique but de vérifier leurs théories biologiques du fonctionnement du système nerveux central."

Nous proposons au lecteur d'adhérer, tout au moins pendant la lecture de ce mémoire, à ce point de vue. Ainsi nous choisissons les positions suivantes. Le domaine "connexionnisme" propose des *contraintes structurelles* à l'informatique dans le but de réaliser des fonctionnalités informatiques pointues. Le domaine "neuromimétique" est une approche de modélisation des sciences du vivant. Finalement les réseaux de neurones sont des instances de modèles de ces deux domaines. Certains réseaux de neurones sont purement connexionnistes (ils n'ont pas de lien avec la réalité biologique). Certains sont simplement neuromimétiques (ils n'ont pas vocation à être utiles computationnellement). La figure 3 illustre les interactions entre ces différents domaines.

[1] On trouve aussi le terme "neuromimétisme".

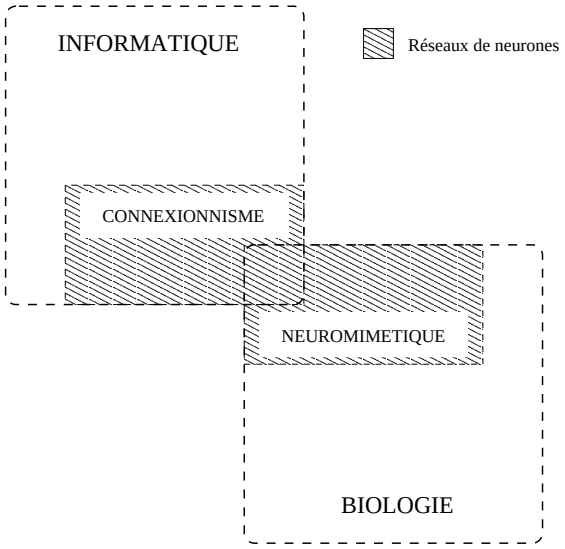

FIG. 3: **Réseaux de neurones, connexionnisme et neuromimétique.**

Qu'est-ce que le connexionnisme ?

Les chercheurs en informatique s'intéressent, nous venons de le dire, aux algorithmes connexionnistes. Même après la clarification que nous venons de donner, c'est un domaine qui est aujourd'hui encore assez mal défini. A ce sujet, il est amusant de citer [Bottou, 1991] qui a contribué à l'étude théorique de ce domaine :

> " L'intérêt pour ce domaine mal défini est tel, ces temps-ci, que le label "connexionniste" est plus un moyen d'obtenir des subsides qu'un critère scientifique. Se risquer à définir le connexionnisme, c'est assurément exclure quelque prétendant, fort honorable au demeurant, et donc s'attirer son ressentiment."

Nous nous limiterons donc à présenter quelques éléments de ce domaine. Pour ce faire, nous nous sommes fortement inspirés de l'introduction de la Foire Aux Questions [Sarle, 1997] sur les réseaux de neurones artificiels.

Structure des réseaux connexionnistes Il n'existe pas de définition universellement acceptée d'"algorithme connexionniste". Néanmoins, la plupart des chercheurs de la communauté s'accorderont probablement sur les propriétés structurelles (tirées de [Haykin, 1994]) suivantes :

- Un algorithme connexionniste est constitué de nombreux petits processeurs (ou unités), chacun ayant une petite quantité de mémoire locale ;
- ces unités sont connectées par des canaux de communication (connexions) qui transportent des informations numériques (par opposition à symboliques).

Certains [Nigrin, 1993] vont plus loin et ajoutent :

- Chaque unité opère seulement sur l'information locale (les entrées qu'elle reçoit via ses connexions) et le fait de façon asynchrone (il n'y a donc pas d'horloge générale pour le système).

Il est ici intéressant de remarquer que la dernière propriété, l'asynchronisme, est particulièrement forte. En pratique, assez peu de réseaux connexionnistes la vérifient. Par exemple, des

modèles qui ont fait la renommée du connexionnisme, comme le perceptron multi-couches [Rumelhart *et al.*, 1986] ou les cartes auto-organisatrices [Kohonen, 1988] ne sont pas asynchrones : ils s'appuient sur une horloge générale.

Nous conviendrons qu'un système qui vérifie toutes les propriétés que nous venons d'énoncer (avec l'éventuel bémol sur l'asynchronisme) est un algorithme connexionniste.

Deux propriétés courantes Jusqu'à présent, nous avons essentiellement insisté sur des caractéristiques structurelles. Il peut être immédiatement intéressant de justifier le fait de s'imposer ce genre de contraintes en énonçant les propriétés que l'on espère en retirer. Dans le domaine de l'IA, il existe de nombreux points de comparaison entre les systèmes connexionnistes et les "autres" systèmes (voir par exemple [Barnden, 1995] pour une liste plus exhaustive, ou encore [Hertz *et al.*, 1991] pour des analyses plus formelles). D'une manière générale, deux d'entre eux me semblent particulièrement pertinents.

Le premier concerne l'efficacité temporelle des systèmes informatiques. Lorsqu'on utilise un algorithme, un facteur essentiel est le temps requis pour que celui-ci fournisse des réponses aux problèmes qui lui sont présentés. Si la *théorie* de la complexité permet d'étudier le coût temporel d'un algorithme, les architectures parallèles ont pour objectif de le réduire *pratiquement*. Alors que de nombreux algorithmes sont difficilement parallélisables, les réseaux de neurones, de par leurs caractéristiques structurelles (distribuées, locales et parfois asynchrones) sont généralement considérés comme de bons candidats pour l'accélération matérielle. Si notre propos est un peu simpliste d'un point de vue général (paralléliser efficacement les réseaux connexionnistes constitue un domaine de recherche non trivial), une implantation récente [Boniface, 2000] du paradigme connexionniste apporte néanmoins de l'eau à notre moulin. On peut y voir par exemple qu'un algorithme comme les cartes auto-organisatrices jouit d'une accélération proche de l'accélération idéale (où la vitesse est divisée par le nombre de processeurs fonctionnant en parallèle). Sur des intervalles de temps courts, des réseaux connexionnistes parallélisés pourraient traiter une quantité énorme d'information et surpasser ainsi largement les capacités computationnelles d'autres approches.

Une deuxième qualité qui découle souvent des contraintes structurelles du connexionnisme concerne la qualité des réponses du système malgré l'occurrence d'événements imprévus. Imaginons que nous ayons implanté sur une plate-forme matérielle spécifique (par exemple une plate-forme parallèle embarquée) un réseau connexionniste. Considérons l'incident suivant : des unités arrêtent de fonctionner (par exemple parce que leur support matériel est abîmé). Suite à cette panne, il serait souhaitable que le système ne "plante" pas définitivement comme le font en général la plupart des programmes informatiques. Mieux encore, le système pourrait continuer à fonctionner et progressivement corriger les effets immédiats de l'incident. Nous dirons d'un système qui est capable de raisonnablement maintenir (voire récupérer) ses performances malgré des pannes internes qu'il est *tolérant aux pannes*. Les caractéristiques structurelles d'un réseau connexionniste font qu'il est un bon candidat pour la tolérance aux pannes.

Un enjeu entre connexionnisme et neuromimétique

A l'intersection du connexionnisme et de la neuromimétique se situent des travaux pluridisciplinaires passionnants : des chercheurs essaient de construire des réseaux de neurones artificiels qui se veulent une *explication opérationnelle* de l'efficacité du fonctionnement cérébral. Ces réseaux sont à la fois des modèles relativement plausibles de ce qui se passe dans le cerveau et constituent des algorithmes computationnellement puissants. Ils donnent des éclaircissements sur le fonctionnement cérébral et peuvent devenir des outils informatiques standards.

On peut dire que ce projet scientifique progresse selon deux mouvements complémentaires :
- une démarche ascendante : on s'inspire de données structurelles, de mécanismes très particuliers de certaines parties du cerveau pour en construire des artefacts informatiques. On analyse mathématiquement ou expérimentalement ces architectures pour déduire leurs propriétés computationnelles ;
- une démarche descendante : on formalise des problèmes qui semblent être résolus efficacement par le cerveau (par exemple "la reconnaissance d'un visage" ou, ce qui nous intéressera précisément dans cette thèse, "l'apprentissage par renforcement") et on montre qu'ils admettent des solutions qui sont à la fois connexionnistes et neuromimétiques.

L'équipe Cortex du Loria participe à la recherche dans ce domaine pluridisplinaire en suivant une démarche essentiellement ascendante. Dans cette thèse, nous espérons participer partiellement à ce travail pluridisplinaire en suivant une démarche descendante. L'une de nos contributions consiste à montrer que le problème de l'apprentissage par renforcement, qui semble être traité efficacement par le cerveau, peut être résolu sous forme connexionniste. Nous disons de notre participation qu'elle est *partielle* pour une raison précise. En tant qu'informaticien, nous ne pouvons prétendre trancher sur le caractère neuromimétique (c'est-à-dire biologiquement plausible) des architectures proposées. Néanmois, la mise sous forme connexionniste constitue une première étape pour éventuellement tendre vers cet objectif.

Plan de la suite du mémoire

Dans ce mémoire, nous nous intéressons au problème général de l'A/R et nous essayons de fournir des réponses de nature connexionniste. La complexité de résolution des problèmes de l'A/R dépend intrinsèquement de la taille de l'espace des "états" du problème modélisé. Ce constat nous a incité à articuler notre présentation en trois étapes :
- Nous commençons par présenter le problème de l'A/R. Lorsque l'espace d'états est *petit*, nous montrons qu'il est possible de lui donner une réponse connexionniste (**massivement parallèle** et **tolérante aux pannes**), **multi-tâche**, **anytime** et **dynamique**.
- Nous considérons ensuite le cas où l'espace d'états est *grand*. Nous présentons une méthode itérative et connexionniste pour approximer un problème de l'A/R ayant un grand espace d'états par un problème ayant un petit espace d'états. Ceci nous permet de conserver les propriétés de la première partie pour des problèmes d'A/R plus difficiles. Nous interprétons cette méthode comme un ***apprentissage de représentation***.
- Finalement, nous généralisons le problème résolu par l'*apprentissage de représentation* : nous proposons une méthode itérative et connexionniste pour approximer n problèmes de l'A/R ayant un grand espace d'états par m problèmes de petit espace d'états avec $m < n$. (Ainsi, la deuxième partie concerne le cas où $m = n = 1$.) Nous montrons que ce problème s'inscrit dans la problématique générale de la catégorisation par noyaux et nous le traitons avec la méthode des nuées dynamiques. Nous interprétons ce processus comme une ***auto-organisation en modules*** spécialisés.

Première partie

Apprentissage par renforcement et connexionnisme

Introduction

« One of the fundamental tasks of this new "neural networks" science is to demonstrate by mathematical analyses, computer simulations, and even working artificial sensory and control systems that is is possible to implement massively parallel information-processing functions using components the principles of which are not mysterious but already familiar from computer technology, communication science, and control engineering. There is nothing in the "neural network" area which were not known, in principle at least, from constructs already in use or earlier suggested. »

Self-Organization and Associative Memory, Teuvo Kohonen

Le cerveau biologique emploie des méthodes de résolution de problèmes connexionnistes, c'est-à-dire très différentes des ordinateurs actuels. Parmi les problèmes que le cerveau semble résoudre, un problème particulièrement motivant est celui formalisé par la théorie computationnelle de l'A/R. L'objet de cette première partie est de montrer que le problème de l'A/R admet une réponse qui est naturellement connexionniste.

Nous commençons par décrire la théorie computationnelle de l'A/R qui est au centre de nos travaux. Nous présentons ses fondements et son formalisme central : les processus décisionnels de Markov. Ce dernier permet de modéliser le problème d'un agent **autonome** devant prendre des décisions de manière **robuste** dans un environnement **stochastique**. Dans le cas où l'espace d'états est *petit*, nous décrivons en détail une réponse algorithmique : l'algorithme *Value Iteration*. Nous montrons que cet algorithme est **anytime**, **multi-tâche** et **dynamique**.

Dans un deuxième temps, nous proposons de construire une architecture de décision connexionniste qui permet de répondre au problème de l'A/R en nous appuyant particulièrement sur les caractéristiques intrinsèques de l'algorithme ***Value Iteration***. Nous argumentons que cette architecture a les qualités qu'on peut attendre des algorithmes connexionnistes : elle est **tolérante aux pannes** et **massivement parallèle**.

Chapitre 1

Les fondements de l'apprentissage par renforcement

La présentation que nous faisons ici de la théorie computationnelle de l'A/R s'inspire fortement de [Sutton et Barto, 1998], introduction générale et très complète. Nous ne présentons ici qu'un ensemble choisi d'éléments de la littérature (ceux que nous considérons comme les plus pertinents et en particulier ceux qui seront utiles à l'architecture que nous proposons). Nous renvoyons le lecteur à cet ouvrage pour une présentation plus globale.

Nous commençons par préciser les éléments essentiels du problème de l'apprentissage par renforcement. Nous détaillons le modèle d'interaction entre l'agent et l'environnement et nous formalisons la notion de récompense qui permet de définir les objectifs de l'agent.

Dans une deuxième section, nous présentons en détail le formalisme central de l'A/R : les processus décisionnels de Markov. Nous introduisons l'hypothèse essentielle de ce formalisme, l'hypothèse de Markov sur la dynamique d'interaction entre l'agent et l'environnement. Nous montrons que le problème que l'on cherche à résoudre se ramène au calcul d'une fonction dite fonction de valeur optimale.

Nous décrivons alors en détail l'algorithme *Value Iteration* qui permet de calculer cette fonction. Nous mettons particulièrement en valeur ses caractéristiques anytime et dynamique.

Finalement, nous faisons la distinction entre les problèmes de planification que traitent les processus décisionnels de Markov et le véritable problème de l'A/R qui ne présuppose pas que l'agent dispose d'un modèle de son interaction avec l'environnement. Nous présentons les deux grandes approches qui permettent de résoudre ce problème d'A/R. Nous discutons en particulier du dilemme exploration/exploitation.

1.1 Introduction au problème de l'A/R

L'A/R est un cadre formel qui propose d'étudier l'interaction entre un *agent* et un *environnement*. L'idée est d'exploiter cette interaction pour faire en sorte que l'agent apprenne à agir. Dans sa version la plus moderne, agent et environnement interagissent continuellement : l'agent effectue des actions et l'environnement répond à ces actions en présentant de nouvelles situations à l'agent. L'environnement envoie un signal numérique appelé récompense ou renforcement, signal que l'agent essaie de maximiser sur le long-terme. Le fait de spécifier des récompenses pour toutes les situations d'interaction envisageables permet de définir une tâche ou un problème à résoudre.

1.1.1 L'interface agent/environnement

D'une manière un peu plus précise, on suppose que l'agent et l'environnement interagissent à des instants discrets. A chaque instant t, l'agent est dans une situation $s_t \in S$ où S est un ensemble de situations possibles, et fait une action $a_t \in A$ où A est un ensemble d'actions possibles. L'instant d'après, l'agent reçoit un signal numérique de récompense $r_{t+1} \in \mathbb{R}$ qui rend compte de l'action qu'il vient de faire, et se retrouve dans une nouvelle situation s_{t+1}. Dans la littérature, on utilise fréquemment le vocable état pour désigner la notion de situation. Nous emploierons indifféremment ces deux termes dans la suite. La figure 1.1 schématise cette interaction agent-environnement. Il est important ici d'insister sur le caractère générique et

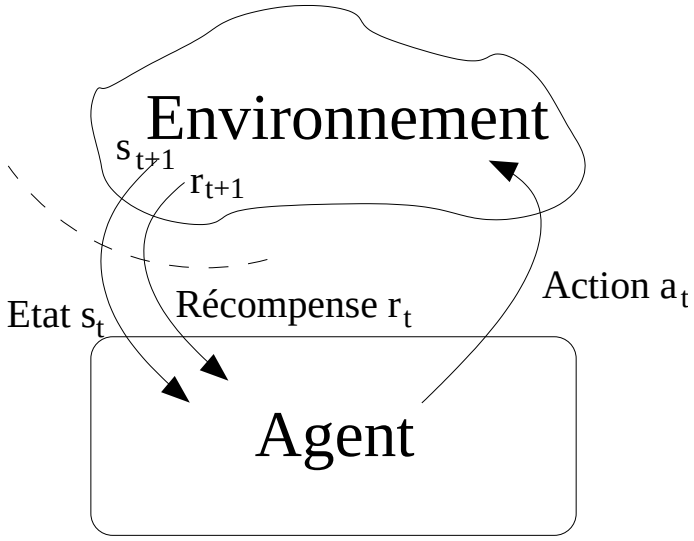

FIG. 1.1: **Interface entre l'agent et l'environnement.** A l'instant t, l'agent est dans l'état s_t et fait l'action a_t. A l'instant d'après, il reçoit un signal de récompense r_{t+1} et se retrouve dans un nouvel état s_{t+1}.

abstrait de ce cadre ; il peut en effet être utilisé pour décrire toutes sortes de problèmes. Les états peuvent prendre des formes très variées : il peut s'agir de sensations de bas niveau (données reçues par des capteurs) ou de descriptions abstraites de haut niveau (description symbolique de positions d'objets dans une pièce). L'état peut reposer sur une mémoire des sensations passées, être complètement subjectif ou mental. Par exemple, un agent peut être dans l'état où "il n'est pas vraiment sûr de la position spatiale d'un objet". De même, les actions peuvent être de bas niveau (comme la contraction d'un muscle) ou bien constituer des décisions de haut niveau (aller acheter un sandwich). Certaines actions peuvent être intérieures, mentales ou inconscientes (dans l'acception commune de ces termes) : une action peut consister à concentrer son attention sur un objet. D'une manière générale, les actions d'un système apprenant par renforcement sont des décisions que nous voulons lui faire prendre ; les états sont toutes les informations que nous connaissons qui pourraient être utiles pour prendre ces décisions.

Ainsi, ce serait une erreur de considérer l'interface de la figure 1.1 entre l'agent et l'environnement comme modélisant précisément la frontière physique d'un robot ou d'un animal. La notion d'agent dans l'A/R a un sens particulier. Du fait de l'utilisation possible d'actions et de perceptions intérieures, la frontière suggérée par le formalisme est en général plus étroite que celle de l'entité modélisée. Par exemple, les moteurs et les capteurs d'un robot doivent être considérés

comme partie de l'environnement plutôt que comme partie de l'agent. De la même façon, des récompenses "calculées" à l'intérieur d'un corps animal physique (par exemple le plaisir) doivent être considérées comme extérieures à l'agent. Si on utilise ce formalisme pour modéliser un animal, ses muscles, son squelette et les organes sensoriels doivent être considérés comme faisant partie de l'environnement. La notion d'agent que l'A/R introduit correspond alors uniquement au centre de la décision. Si on se sert de cette théorie pour modéliser un animal, l'agent joue le rôle de son cerveau.

1.1.2 Notion de récompense

Les récompenses sur le long-terme

L'objectif d'un agent est formalisé à travers le signal de récompense. A chaque instant, l'interaction produit une récompense r_t, valeur numérique bornée, qui mesure la justesse de réaction de l'agent. Le but de l'agent est alors de maximiser le "cumul" de ces récompenses sur le long-terme. Une manière naturelle pour prendre en compte le long-terme est de dire qu'à l'instant t, l'agent doit prendre la décision qui lui permet de maximiser la somme des récompenses qu'il recevra dans le futur :

$$R_t = r_{t+1} + r_{t+2} + .. + r_T \tag{1.1}$$

où T est un instant terminal qui met fin à l'interaction. Dans bien des cas cependant, l'interaction n'a pas de limite ($T = \infty$). Pour éviter que le critère ci-dessus ne diverge, on pourrait s'intéresser à la récompense future moyenne. Dans la plupart des paradigmes de l'A/R, on s'intéresse à un critère qui, s'il paraît a priori un peu plus compliqué, a des propriétés intéressantes (nous les verrons un peu plus loin). Il s'agit de la récompense pondérée à long-terme :

$$R_t = r_{t+1} + \gamma.r_{t+2} + \gamma^2.r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k.r_{t+k+1} \tag{1.2}$$

où $\gamma \in [0 : 1[$ est appelé *facteur de pondération*. Du point de vue de l'agent, $1 - \gamma$ peut être interprété comme la probabilité, à chaque instant, de mourir (de ne plus avoir de récompenses à l'avenir). En effet, on peut écrire :

$$R_t = r_{t+1} + \gamma.R_{t+1} + (1 - \gamma).0 \tag{1.3}$$

Ce facteur de pondération détermine la valeur présente d'une récompense future : une récompense reçue dans k pas de temps vaut γ^k fois ce qu'elle vaudrait si elle était reçue immédiatement. Ce critère présente donc un intérêt fonctionnel : il incite à maximiser les récompenses tout en diminuant le temps pour les obtenir. L'influence du temps sera d'autant plus négligeable que γ sera proche de 1. A contrario, si γ vaut 0 (l'agent agit comme s'il était persuadé de mourir à l'instant d'après), on dit que l'agent est *glouton* : son but revient alors, à chaque instant, à maximiser sa récompense immédiate sans se préoccuper de celles qui suivent[2].

De la définition d'une récompense instantanée

L'utilisation d'un signal de récompense (local dans le temps) pour définir une tâche peut paraître a priori limitant. C'est en effet beaucoup moins riche que d'utiliser un critère global (sur toute la trajectoire des états). De nombreuses applications dans la littérature ont néanmoins montré que cette approche était pratique et flexible. Pour s'en convaincre, évoquons quelques exemples

[2]Tel l'énorme M. Creosote (*Le sens de la vie, Monty Python*), qui explose de trop manger !

d'utilisation d'une récompense. Pour faire nager un robot artificiel, on peut donner à chaque instant une récompense qui est proportionnelle à sa vitesse [Coulom, 2002]. Pour construire un agent dont le but est de sortir d'un labyrinthe, on peut donner une récompense nulle la plupart du temps et $+1$ dès que l'agent sort effectivement du labyrinthe [Kaelbling *et al.*, 1996]. Pour éviter qu'un tel agent ne se cogne contre les murs, on pourra le sanctionner à chaque fois qu'il touche un mur en lui donnant une récompense fortement négative (par exemple -1000 [Laroche, 2000]). Pour un système qui devrait jouer à un jeu de réflexion comme le back-gammon, un choix judicieux de renforcement consiste à donner une récompense nulle pendant toute la partie, puis $+1$, 0 ou -1 après le dernier coup de chaque partie selon que le système a gagné, fait une partie nulle ou perdu [Tesauro, 1995]. Nous voyons dans ce dernier exemple que la récompense permet de définir un but qui peut être relativement difficile à atteindre : ce sera au système artificiel de trouver le (long) cheminement qui permet de l'atteindre.

1.2 Les processus décisionnels de Markov finis

Nous allons maintenant présenter un formalisme qui permet d'aborder efficacement une grande partie du problème de l'A/R. Avant de le présenter en détail, nous devons introduire un concept fondamental : l'hypothèse de Markov.

1.2.1 L'espace d'états et l'hypothèse de Markov

Dans les problèmes d'A/R, l'agent prend à chaque instant des décisions en se basant sur la situation courante, ou encore état courant. En effet, on dit que l'agent suit une *politique* qui est de la forme $\pi : S \to A$: à chaque état est associée (de manière éventuellement stochastique) une action. La question essentielle que nous abordons maintenant est la suivante : sachant que l'agent prend ses décisions ainsi, qu'est-ce qui constitue un *bon* espace d'états ? Comme nous l'avons déjà suggéré, la réponse est "toute information qui pourrait être utile à l'agent pour prendre ses décisions". Cela peut se formaliser encore plus précisément.

Il est naturel de supposer que l'état ne décrit pas exhaustivement la situation de l'environnement, mais qu'il est le fruit d'un pré-traitement. Par exemple, un robot navigateur aurait tout intérêt à utiliser un Global Positionning System (GPS) : l'ensemble des états serait alors les couples latitude-longitude que le robot peut atteindre. Dans ce cas, un traitement de haut niveau rend compte de la situation présente. Il n'y a cependant aucune raison de limiter la notion d'état à celle de perception immédiate : il peut s'agir d'une représentation complexe construite petit à petit, grâce à la séquence des perceptions de l'ensemble des interactions passées.

D'une manière idéale, il faut qu'un état résume de manière *complète* toutes les informations nécessaires à la décision perçues dans le passé. L'idée qu'un état résume complètement le passé est une hypothèse mathématique bien étudiée qu'on appelle hypothèse de Markov. Une manière équivalente de l'introduire est de dire que l'état du système à l'instant $t+1$ ne dépend plus que de celui à l'état t (et biensûr de l'action effectuée par l'agent). Ainsi, dans tout ce qui suit, nous ferons l'hypothèse suivante :

$$P(s_{t+1} = s' | s_0, a_0, s_1, a_1, .., s_t, a_t) = P(s_{t+1} = s' | s_t, a_t) \tag{1.4}$$

pour tout état s' et tout historique des événements passés : $s_0, a_0, s_1, a_1, .., s_t, a_t$.

Dans ce chapitre, nous supposerons que l'ensemble des états S est une donnée a priori : nous éludons temporairement le processus qui consiste à le construire pour nous concentrer sur le problème du choix des actions. Dans l'ensemble de ce mémoire, nous supposerons de plus

que le mécanisme d'identification de l'état (le processus qui permet de déterminer à chaque instant, l'état de l'agent) est donné a priori. Illustrons ces hypothèses sur l'exemple du robot navigateur qui utilise un GPS : nous supposerons ici qu'un ensemble de lieux (latitude-longitude) caractéristiques est donné a priori et que la fonctionnalité GPS est fournie. Quand nous utiliserons le vocable "état", nous supposerons implicitement que l'hypothèse de Markov est vérifiée pour le modèle considéré.

1.2.2 Le formalisme PDM

Le formalisme des processus décisionnels de Markov permet de rassembler la plupart des idées et concepts que nous avons décrits jusqu'ici : états, actions, dynamique d'interaction vérifiant l'hypothèse de Markov et récompense.

Définition 1 (Processus décisionnel markovien)
Un processus de décision markovien (PDM) est un tuple $\langle S, A, T, R \rangle$ *:*

- S *est un ensemble d'états ;*
- A *est un ensemble d'actions ;*
- $T : S \times A \times S \rightarrow [0:1]$ *est une fonction appelée "fonction de transition" ; à tout instant, la probabilité de passer de l'état* s *à l'état* s' *en faisant l'action* a *est* $T(s, a, s')$*. Nous supposons ici implicitement que cette probabilité est stationnaire (elle ne varie pas au cours du temps)*[3]
- $R : S \times A \rightarrow I\!R$ *est la fonction de récompense ; elle associe un signal de renforcement borné à chaque couple état-action et définit implicitement des objectifs*[4]*.*

Illustration

Considérons un robot qui se déplace dans un environnement en forme de P et dont l'objectif est d'atteindre une zone but (voir figure 1.2).

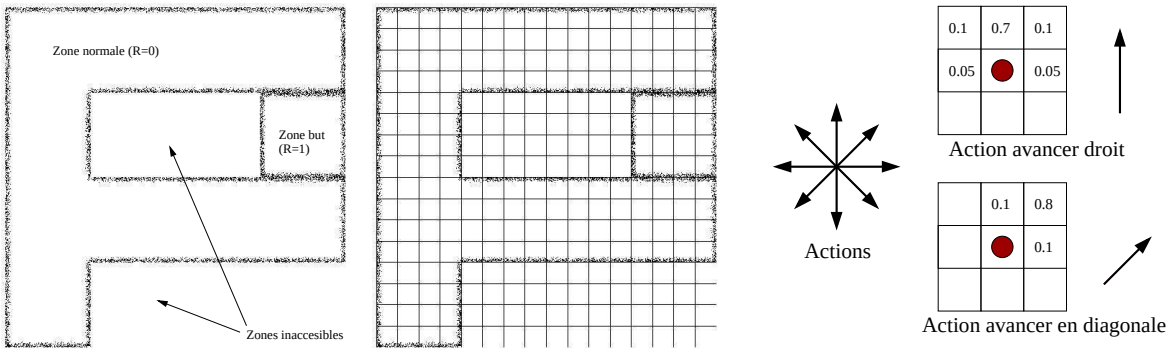

FIG. 1.2: **Navigation discrète.** A gauche : le problème que nous voulons modéliser à l'aide d'un PDM est celui d'un agent devant se déplacer dans un environnement simple. Au centre : nous représentons ce problème à la manière d'un damier ; les zones de l'environnement sont des cases ; à tout moment, l'agent est dans une et une seule de ces cases. A droite : l'agent peut se déplacer sur le damier dans les 8 directions cardinales. Ses mouvements sont légèrement bruités ; par exemple, lorsque l'agent se déplace vers le Nord-Est, la probabilité qu'il atteigne effectivement la case qui est au Nord-Est est 0.8.

[3]La stationnarité de la fonction de transition ne signifie pas nécessairement que l'environnement est statique, mais simplement que la façon dont un couple (état,action) permet de prédire ses successeurs ne varie pas au cours du temps.

[4]On trouve d'autres ensembles définitions pour la fonction récompense R. Elle est parfois définie sur S ou encore sur $S \times A \times S$. De même, R peut être choisie déterministe ou stochastique.

On peut modéliser ce problème à l'aide d'un PDM en définissant $16 \times 16 = 256$ zones élémentaires qui découpent l'environnement de manière régulière. On identifie alors l'état de l'agent-robot avec la zone où le robot se trouve (information qu'il peut obtenir à l'aide d'un GPS par exemple).

Le robot peut effectuer 9 actions différentes : se déplacer dans les 8 directions cardinales et ne pas bouger. On suppose que le résultat de chaque action est légèrement bruité. L'intégration d'un tel modèle de dynamique peut être résumé par deux types de transitions (on suppose alors que le déplacement est isotrope), le déplacement "droit" et le déplacement en "diagonale" (voir figure 1.2).

Afin de pénaliser le robot lorsqu'il se cogne à un mur, nous lui donnons une punition (récompense -1). Pour le motiver à atteindre la zone but, nous assignons à tous les états qui la constituent une récompense positive $+1$. Dans tous les autres états, la récompense est nulle.

1.2.3 Fonction de valeur associée à une politique stochastique

L'enjeu d'un tel formalisme est d'évaluer le critère que nous avons introduit pour juger du comportement de l'agent : la récompense pondérée à long-terme. La plupart des algorithmes de l'A/R sont basés sur l'estimation d'une *fonction de valeur*, fonction qui estime la quantité moyenne de récompenses futures en partant d'un état. Cette fonction dépend bien évidemment de la fonction politique $\pi : S \to A$ (éventuellement stochastique) de l'agent. La valeur d'un état s en suivant la politique π est :

$$V^{\pi}(s) = E_{\pi}\left[R_t | s_t = s\right] \tag{1.5}$$

où $E_{\pi}[.]$ désigne l'espérance mathématique sachant que l'agent suit la politique π et R_t est la récompense pondérée définie à l'équation 1.2. Cette définition a bien un sens dans la mesure où la fonction de valeur ne dépend pas du temps t : ceci est dû au fait que le processus d'évolution des états lorsqu'on fixe une politique est une chaîne de Markov stationnaire.

Une des propriétés fondamentales de cette fonction est le fait qu'elle vérifie une équation récursive bien particulière. Précisons la notation utilisée pour la politique π : si $\pi(s,a)$ est la probabilité d'effectuer l'action a dans l'état s, alors on a :

$$\begin{aligned} V^{\pi}(s) &= \sum_{a \in A} \pi(s,a).\,(\text{récompense immédiate} + \gamma.\text{récompense future}) \\ &= \sum_{a \in A} \pi(s,a).\left(R(s,a) + \gamma.\sum_{s' \in S} T(s,a,s').V^{\pi}(s')\right) \end{aligned} \tag{1.6}$$

Littéralement, la valeur d'un état est la récompense immédiate $R(s,a)$ *plus* les récompenses futures. Les récompenses futures sont les valeurs des états qui seront rencontrés à l'instant d'après selon les transitions $T(s,a,s')$. Le tout est évidemment pondéré par $\pi(s,a)$ pour prendre en compte le caractère stochastique de la décision. Cette propriété, aussi appelée équation de Bellman de V^{π}, peut être vue comme une équation de type point fixe $V^{\pi} = F(V^{\pi})$. Dans la mesure où $\gamma < 1$, on montre que la fonction F est une contraction, et que la solution de l'équation existe et est unique [Sutton et Barto, 1998].

Il existe une autre fonction de valeur, appelée *Q-fonction*, qui donne la valeur d'un couple état-action (s,a) dans le cadre d'une politique. En effet, elle vaut la quantité moyenne de récompenses reçues en partant d'un état, en effectuant une action, puis en suivant une politique π.

$$Q^{\pi}(s,a) = E_{\pi}\left[R_t | s_t = s, a_t = a\right] \tag{1.7}$$

C'est une fonction particulièrement intéressante car elle permet d'évaluer le choix d'une action dans le cadre d'une politique, et donc de remettre en question cette politique. D'une manière évidente, on a les relations suivantes entre les deux fonctions de valeur que nous venons d'introduire :

$$Q^\pi(s,a) = R(s,a) + \gamma . \sum_{s' \in S} T(s,a,s').V^\pi(s') \tag{1.8}$$

$$V^\pi(s) = \sum_{a \in A} \pi(s,a).Q^\pi(s,a) \tag{1.9}$$

Comment déterminer la fonction de valeur d'une politique π donnée ? L'équation 1.7 est un système de $|S|$ équations à $|S|$ inconnues. On pourrait résoudre ce système linéaire en inversant ce système. Il est souvent plus simple de considérer que c'est une équation de type point fixe. Un algorithme itératif, *Iterative Policy Evaluation*(voir l'algorithme 1.1) exploite cette idée pour calculer la fonction de valeur.

Algorithme 1.1 Iterative Policy Evaluation

Données : Un PDM $\langle S, A, T, R \rangle$ et une politique stochastique $\pi : S \to A$
But : Calculer la fonction de valeur V^π
Initialisation quelconque de V_0, $t = 0$.
répéter
$t \leftarrow t + 1$
pour tout $s \in S$ **faire**

$$V_t(s) = \sum_{a \in A} \pi(s,a). \left(R(s,a) + \gamma . \sum_{s' \in S} T(s,a,s').V_{t-1}(s') \right)$$

fin pour
jusqu'à $\max_{s \in S} |V_t(s) - V_{t-1}(s)| < \varepsilon$

Illustration

Nous illustrons cet algorithme sur le problème de navigation discrète que nous avons introduit un peu plus tôt en supposant que l'agent suit une politique complètement aléatoire. L'évolution de la fonction de valeur au cours des itérations est visible figure 1.3. On remarque que d'une manière logique, l'espérance de récompense est d'autant plus grande qu'on est loin des murs et qu'on est proche de la zone-but.

1.2.4 Fonction de valeur optimale

Calculer la valeur d'une politique est un premier pas. Trouver la meilleure (nous allons rapidement préciser la signification de "meilleure") est le problème que nous considérons ici. Résoudre le problème de l'A/R consiste en effet à trouver une politique qui maximise l'espérance de récompense à long-terme quel que soit l'état de départ : on cherche une politique $\pi^* : S \to A$ qui est telle que :

$$\forall s \in S, V^{\pi^*}(s) = \max_\pi V^\pi(s) \tag{1.10}$$

L'un des résultats fondamentaux des PDM est le suivant : parmi toutes les politiques optimales π, il en existe une qui est déterministe et stationnaire [Bellman, 1957]. L'opérateur *max* de l'équation 1.10 peut donc être limité aux politiques déterministes.

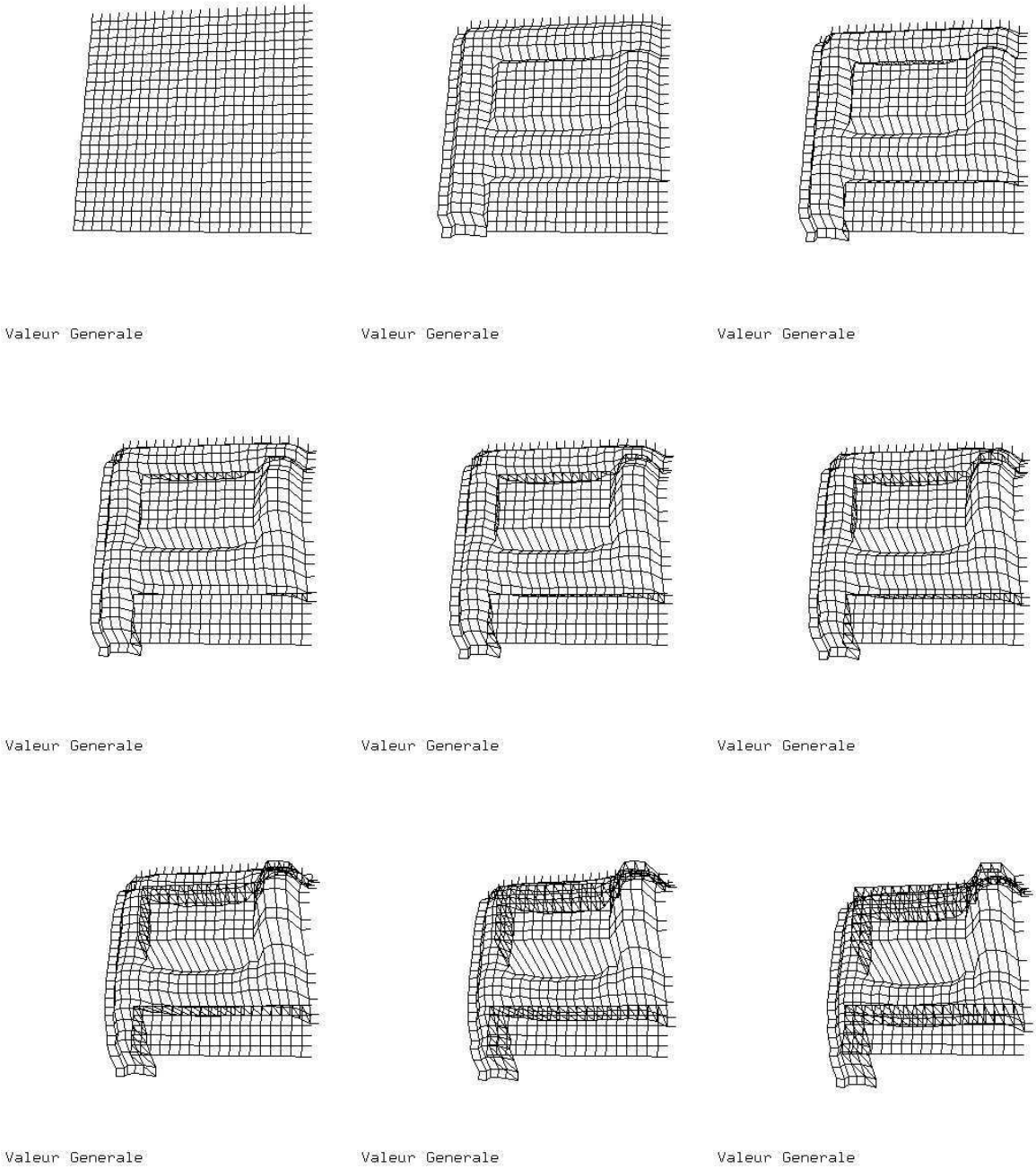

FIG. 1.3: **Fonction de valeur d'une politique aléatoire pendant l'algorithme Policy Evaluation aux instants** $t = 0,1,2,3,4,5,10,15,final$**.** Itération après itération, la valeur des états augmente près de la zone but (où il y a une récompense positive) et diminue près des murs (où il y a une récompense négative). Après convergence, la valeur en chaque point vaut l'espérance de récompenses dans le futur lorsqu'on suit une politique aléatoire.

Pour caractériser une solution de ce problème, il est usuel de s'appuyer sur la Q-fonction introduite plus haut. Une politique optimale π^* vérifie nécessairement l'équation suivante :

$$Q^{\pi^*}(s, \pi^*(s)) = \max_{a \in A} Q^{\pi^*}(s, a) \tag{1.11}$$

Littéralement, si la politique est optimale globalement (dans le temps), elle l'est aussi localement. Cette équation peut, de manière équivalente, s'écrire à l'aide de la fonction V :

$$V^{\pi^*}(s) = \max_{a \in A} \left[R(s, a) + \gamma . \sum_{s' \in S} T(s, a, s') . V^{\pi^*}(s') \right] \tag{1.12}$$

Une solution de cette équation est appelée *fonction de valeur optimale*. Cette relation, appelée équation de Bellman de la fonction de valeur optimale, est aussi une équation de type point fixe $V^{\pi^*} = G(V^{\pi^*})$. On montre que la solution existe et est unique [Bellman, 1957].

La plupart des algorithmes de l'A/R se concentrent sur le calcul de l'une des deux fonctions de valeur optimales (Q ou V). Si c'est V, on peut déduire son homologue Q à l'aide de l'équation 1.8. Une fois la fonction de valeur optimale Q^{π^*} calculée, on déduit immédiatement une[5] politique optimale ainsi :

$$\pi^*(s) \in \arg\max_{a \in A} Q^{\pi^*}(s, a) \tag{1.13}$$

Dans le calcul d'une politique optimale, l'essentiel est le calcul de la fonction de valeur optimale ; la politique s'en déduit alors immédiatement. Nous consacrons les deux sections qui suivent à la description détaillée d'un algorithme qui permet de calculer ces fonctions de valeur optimales.

1.3 L'algorithme *Value Iteration*

Comment calcule-t-on une politique optimale π^* ? Nous détaillons ici un algorithme issu de la programmation dynamique qui répond à cette question : *Value Iteration* [Bellman, 1957]. Il se concentre sur le calcul de la fonction de valeur optimale V^{π^*}. L'illustration que nous en faisons nous permet de mettre en valeur un certain nombre de propriétés intéressantes bien connues.

1.3.1 L'algorithme *Value Iteration*

L'algorithme *Value Iteration* exploite le fait que l'équation 1.12 est une équation de type point fixe unique (voir algorithme 1.2).

Il est aisé de voir qu'il converge en remarquant que le processus itératif est une contraction. On remarque en effet que :

$$\max_{s \in S} |V_{t+1}(s) - V_t(s)| \leq \gamma . \max_{s \in S} |V_t(s) - V_{t-1}(s)| \tag{1.14}$$

Comme l'équation 1.12 n'a qu'une solution, la limite du processus est nécessairement la fonction de valeur optimale. On déduit successivement la Q-fonction et une politique optimales en utilisant les équations 1.8 et 1.13. La condition d'arrêt de l'algorithme (différence entre 2 fonctions successives) est justifiée par le fait que :

$$\max_{s \in S} |V^{\pi^*}(s) - V_t(s)| \leq \frac{\gamma . \max_{s \in S} |V_t(s) - V_{t-1}(s)|}{1 - \gamma} < \frac{\gamma . \varepsilon}{1 - \gamma} \tag{1.15}$$

Pour obtenir cette dernière inégalité, il suffit en effet d'itérer l'inégalité 1.14, d'utiliser l'inégalité triangulaire et le fait que $\sum_{k=1}^{\infty} \gamma^k = \frac{\gamma}{1-\gamma}$ (voir par exemple [Williams et Baird, 1993]).

[5] Il peut y en avoir plusieurs qui sont équivalentes (qui ont la même fonction de valeur).

Algorithme 1.2 Value Iteration

Données : Un PDM $\langle S, A, T, R \rangle$
But : Calculer la fonction de valeur optimale V^{π^*}
Initialisation quelconque de V_0, $t = 0$.
répéter
$t \leftarrow t + 1$
pour tout $s \in S$ **faire**

$$V_t(s) = \max_{a \in A} \left(R(s,a) + \gamma . \sum_{s' \in S} T(s,a,s') . V_{t-1}(s') \right)$$

fin pour
jusqu'à $\max_{s \in S} |V_t(s) - V_{t-1}(s)| < \varepsilon$

1.3.2 Illustration et quelques qualités

Reprenons le problème de navigation discrète que nous avons décrit un peu plus tôt. Les figures 1.4 à 1.6 illustrent l'évolution de la suite V_t qui converge vers la fonction de valeur optimale et les politiques optimales qu'on peut déduire à chaque itération à l'aide de l'équation 1.13. Cet exemple va nous permettre de relever trois caractéristiques intéressantes.

Un algorithme anytime

Tout d'abord, on peut dire de cet algorithme qu'il est *anytime* : la qualité des politiques est croissante au cours des itérations et si l'on décide d'arrêter le processus itératif avant la fin on dispose d'une solution sous-optimale. En particulier, on connaît l'incertitude sur la fonction de valeur optimale à l'aide de l'équation 1.15. Avec les paramètres $\gamma = 0.99$ et $\varepsilon = 10^{-4}$, l'algorithme s'arrête au bout de 153 itérations et l'équation 1.15 nous permet de dire que $|V_{153} - V^{\pi^*}| < 10^{-2}$. En pratique, on remarque souvent qu'on obtient une politique optimale bien avant d'avoir une très bonne évaluation de la fonction de valeur optimale. Dans le cas présent, on observe que la politique optimale est atteinte au bout de 18 itérations. Cette rapidité de planification et le caractère *anytime* nous permettent de souligner une propriété fonctionnelle intéressante : contrairement à bon nombre de systèmes de planification qui séparent les phases de planification et d'exécution, l'utilisation d'un PDM avec un algorithme comme *Value Iteration* permet de les mélanger. Il est ainsi tout à fait possible d'exécuter le plan en même temps qu'on le calcule : on alterne alors itérations de l'algorithme *Value Iteration* et exécutions d'actions.

Une généralisation de l'algorithme des flots

Ensuite, il est intéressant d'avoir un regard qualitatif sur le processus qui consiste à calculer la fonction de valeur optimale. Comme le montre l'évolution de la fonction de valeur (figures 1.4 à 1.6), on observe que la source de motivation de l'agent (située dans la zone-but à atteindre) engendre et entretient un pic de valeur qui est *propagé* à travers l'espace d'états. L'algorithme *Value Iteration* est complètement analogue à l'algorithme des flots de Bellman-Ford [Bellman, 1958] qui calcule un plus court chemin dans un graphe. Pour être encore plus précis, le formalisme PDM est une stricte généralisation de la théorie des graphes et on montre que l'algorithme des flots est un cas particulier de *Value Iteration* [Gordon, 1999]. Un graphe peut en effet être représenté par un PDM déterministe où les actions correspondent aux arêtes du graphe. Si on

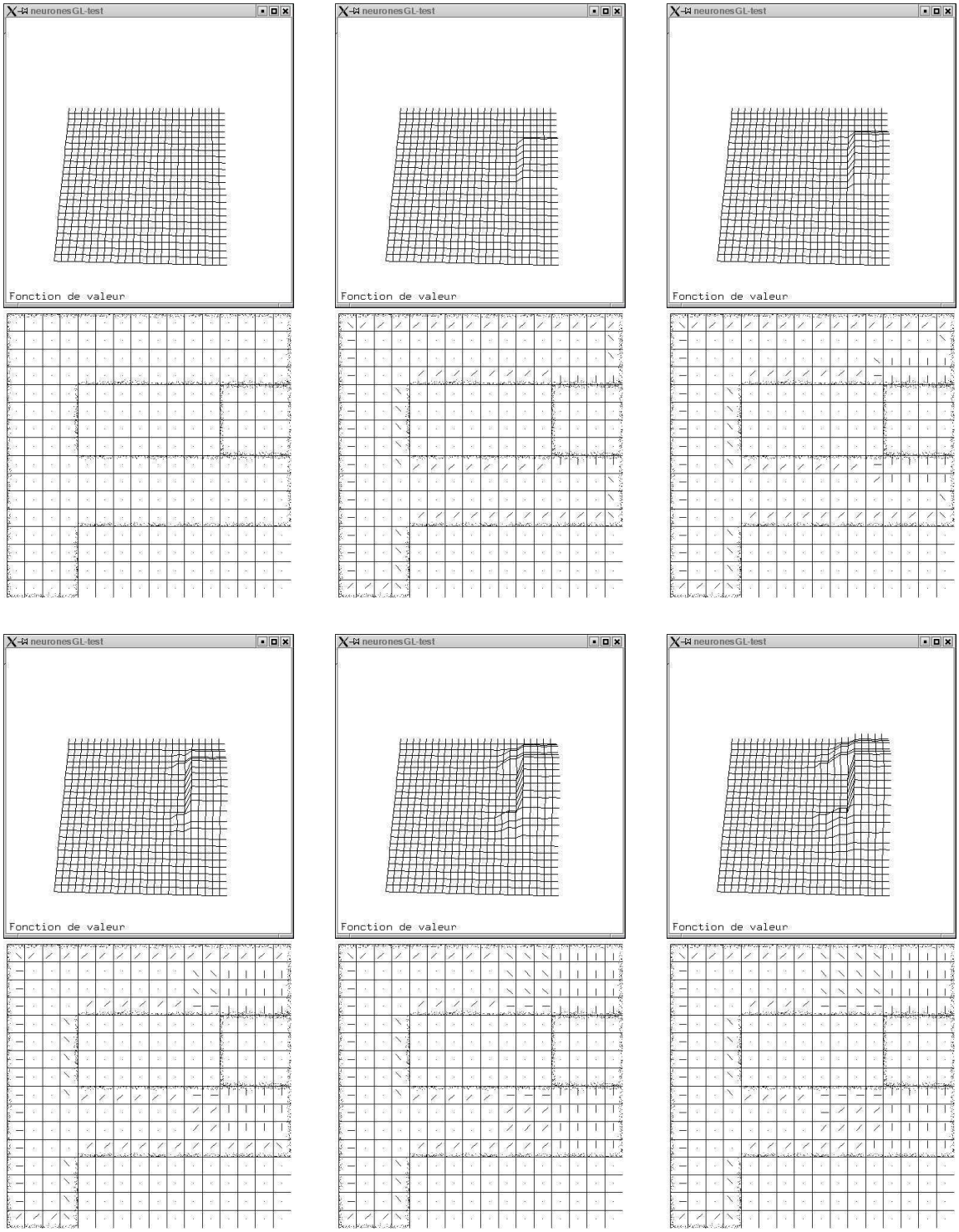

FIG. 1.4: **Fonction de valeur et plan pendant l'algorithme *Value Iteration* aux instants $t = 0, 1, 2, 3, 4, 5$.** Pour chaque instant, nous avons représenté la fonction de valeur et la politique qu'on en déduit à l'aide de l'équation 1.13. L'action à effectuer en chaque état est représentée par un segment qui part du centre de l'état et qui va dans la direction cardinale de l'action.

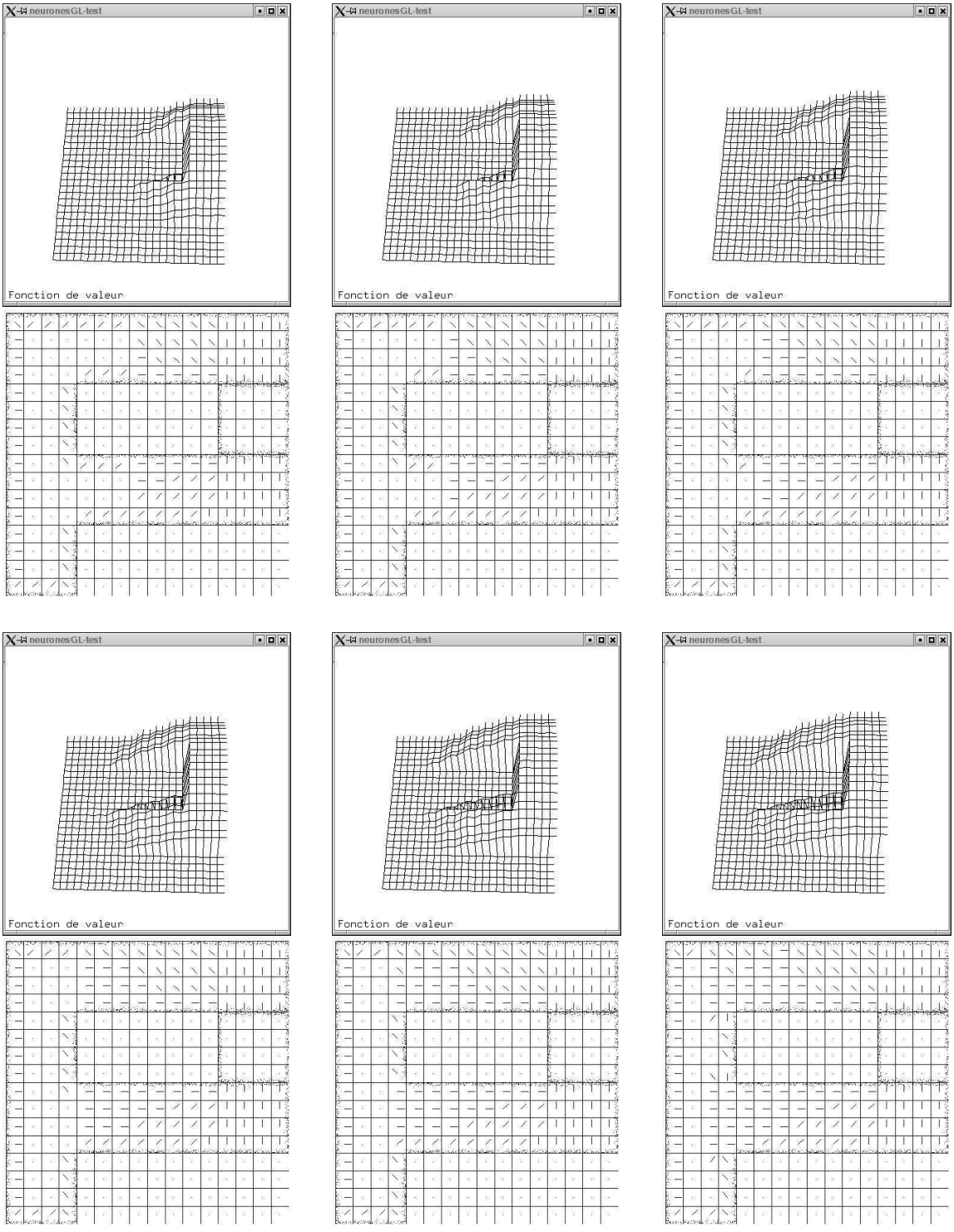

FIG. 1.5: **Fonction de valeur et plan pendant l'algorithme *Value Iteration* aux instants $t = 6, 7, 8, 9, 10, 11$.** Voir la figure 1.4 pour la légende.

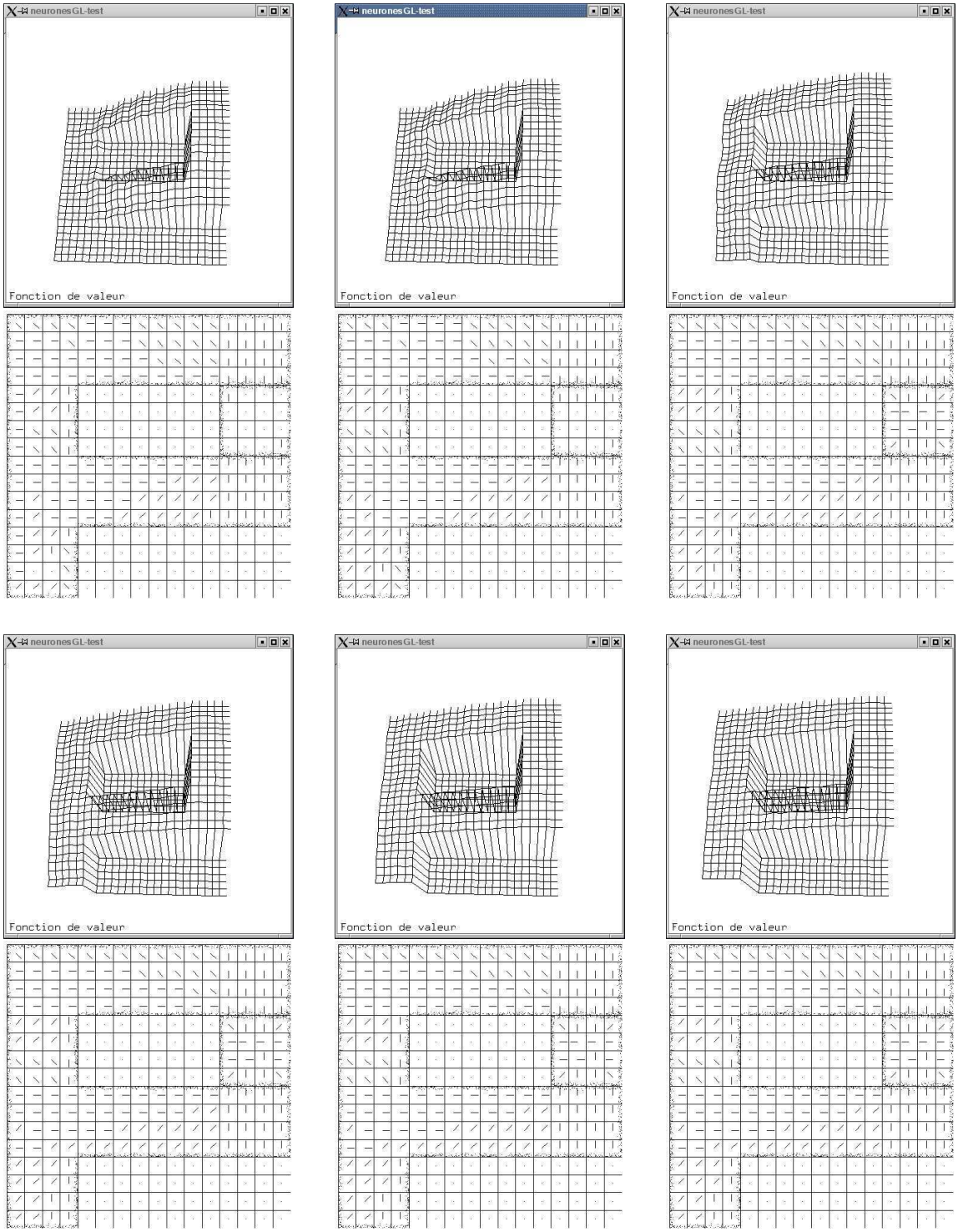

FIG. 1.6: **Fonction de valeur et plan pendant l'algorithme *Value Iteration* aux instants** $t = 12, 13, 18, 23, 28, final$**.** Voir la figure 1.4 pour la légende.

donne une récompense de 1 au nœud but du graphe et une récompense nulle à tous les autres nœuds, la distance de tout nœud au but (qui est l'objet du calcul de l'algorithme des flots) se déduit simplement de la fonction de valeur optimale à l'aide de la relation $d = \frac{\log(V.(1-\gamma))}{\log(\gamma)}$. En effet, si un état est à la distance d du but, sa valeur est tout simplement $\sum_{k=d}^{\infty} \gamma^k = \frac{\gamma^d}{1-\gamma}$ (récompenses de valeur 1 à partir de $t = d$).

Planification indépendante des conditions initiales

Une autre propriété particulièrement intéressante des PDM est la suivante : la politique qui maximise l'espérance de récompense pondérée à long-terme le fait à partir de *n'importe quel* état de départ. Ainsi, une fois *une* politique calculée, on peut l'utiliser pour agir à partir de *tous* les états. La figure 1.7 illustre ceci en montrant une centaine de trajectoires simulées en suivant la politique optimale à partir de points tirés au hasard uniformément. Cette figure illustre également l'influence des paramètres récompenses et du facteur de pondération γ sur les plans effectivement calculés. Par exemple, si on augmente significativement l'ampleur de la punition lorsque le robot touche un mur, on voit que celui-ci aura tendance, dans son plan, à rendre son chemin plus sûr (en gardant une distance de sécurité). Il en va de même lorsqu'on fait varier γ. Un γ petit donne beaucoup plus d'importance aux récompenses très proches dans le temps et moins à celles qui sont lointaines (par exemple la motivation d'atteindre la zone but) : le robot sera ainsi plus prudent. On peut dire que plus γ est proche de 1, plus l'horizon sur lequel l'agent raisonne pour prendre ses décisions est grand. Le réglage de ces paramètres est en général fait de manière empirique.

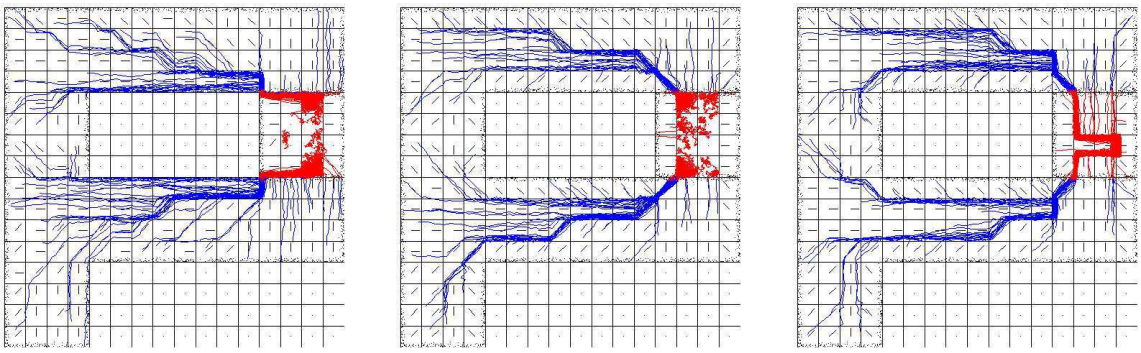

FIG. 1.7: **Exemple de trajectoires induites par la planification avec différentes valeurs de γ et de punition.** De gauche à droite nous avons utilisé les couples de paramètres suivants : ($\gamma = 0.99$, punition$= -1$), ($\gamma = 0.5$, punition$=-1$), et ($\gamma = 0.99$, punition$= -1000$).

1.3.3 Complexité temporelle

Si on note m le nombre moyen de transitions possibles à partir de chaque état (dans le pire des cas $m = |S|$), la complexité moyenne par itération de l'algorithme *Value Iteration* est en $O(|A|.|S|.m)$. On montre de plus (voir [Littman *et al.*, 1995] [Littman, 1996]) que d'une manière générale, le nombre d'itérations nécessaires pour trouver une politique optimale est polynomial en $|S|$, $|A|$, γ et B où B est le nombre de bits total pour représenter les données du problème. Le calcul d'une borne plus précise du nombre d'itérations requis pour converger vers la politique optimale est aujourd'hui encore un problème ouvert. Dans certains cas particuliers, comme celui

où la dynamique exclut tout cycle, le nombre d'itérations est borné par $|S|$ [Bertsekas et Tsitsiklis, 1996].

Si tous les paramètres jouent un rôle dans la complexité, la taille $|S|$ de l'espace d'états est généralement considérée comme le paramètre le plus crucial. Plus il y a d'états et plus le nombre moyen de transitions m peut être grand, plus une itération de *Value Iteration* prend du temps, et plus on risque d'avoir besoin de nombreuses itérations pour propager la valeur dans l'espace tout entier. Il est en pratique possible (avec la puissance de calcul des ordinateurs récents) de résoudre des PDM ayant jusqu'à 1 million d'états [Sutton et Barto, 1998]. Pour illustration, le calcul de la fonction de valeur optimale dans le problème de navigation discrète ($|S| = 256$, $m = 10$, $|A| = 9$, $\gamma = 0.99$ et $\varepsilon = 10^{-4}$)) a duré moins de 3 secondes sur une machine séquentielle actuelle[6]. Un problème analogue ayant plus de 100000 états a requis sur cette même machine un peu moins de 10 minutes. Nous verrons un peu plus loin que l'utilisation du parallélisme permet d'améliorer significativement ces performances.

1.4 Planification dynamique et *Value Iteration*

Nous avons vu que l'algorithme *Value Iteration* permet de faire de la planification. Nous allons montrer que si on supprime sa condition d'arrêt, nous pouvons le considérer comme un algorithme dynamique : il peut alors s'adapter à d'éventuelles fluctuations des paramètres du modèle. Les propriétés que nous avons décrites jusqu'ici au sujet de l'algorithme *Value Iteration* sont bien connues dans la littérature. L'idée selon laquelle il s'agit d'un algorithme dynamique est à notre connaissance originale.

L'algorithme *Value Iteration* présente deux caractéristiques intéressantes :

- c'est un processus itératif invariable : toutes les itérations (en début comme en fin de planification) sont identiques ;
- la convergence vers la fonction de valeur est indépendante des conditions initiales : on peut utiliser n'importe quelle fonction de valeur initiale, par exemple, on peut partir de celle calculée pour une tâche autre que celle qu'on est en train de résoudre.

Nous proposons de montrer ici que ces deux caractéristiques rendent ce système dynamique : il tend toujours vers la fonction de valeur optimale correspondant à la tâche courante, même lorsque celle-ci change en cours de route. Nous allons illustrer cette capacité d'adaptation dans deux cas :

- Variation discontinue de la récompense : un agent aura à successivement résoudre des tâches qui peuvent être relativement différentes les unes des autres. Chacune sera définie par une fonction de récompense. A des instants choisis, on passera d'une tâche à une autre en changeant instantanément et globalement la récompense.
- Variation continue de la récompense : la récompense rendra compte d'un besoin (comme le niveau des batteries pour un robot). La récompense variera continuellement au cours du temps.

Nous ne traitons pas le cas (complètement analogue par ses conséquences) où la fonction de transition T varie.

1.4.1 Changements rapides de la fonction récompense

Reconsidérons le problème de navigation discrète que nous avons introduit un peu plus tôt (page 21). Supposons cette fois-ci qu'il n'y a pas seulement une zone but mais trois (voir la figure

[6] Processeur Pentium III, 800Mhz, 192Mo de RAM.

1.8). Nous définissons trois fonctions récompenses (R_1, R_2, R_3) qui correspondent chacune à l'un

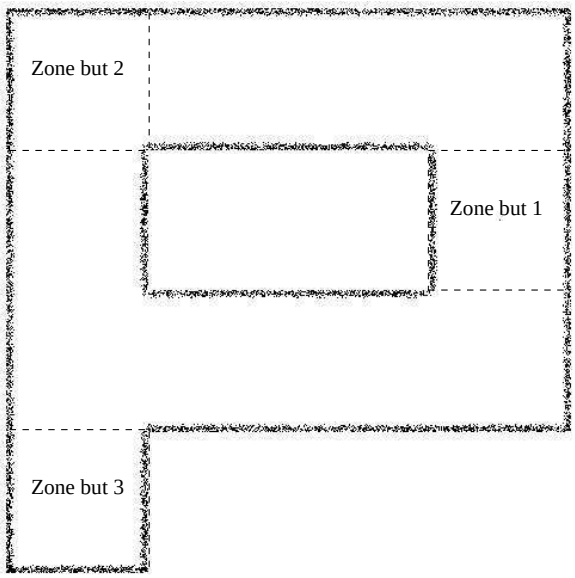

FIG. 1.8: **Navigation discrête multi-tâche.** Dans ce problème, on distingue trois zones buts. Une affectation des valeurs de récompense (R_1, R_2, R_3) dans chacune des zones buts définit une tâche. Par exemple ($R_1 = 0$, $R_2 = 1$, $R_3 = 1$) définit la tâche "aller soit dans la zone but 2 soit dans la zone but 3.

des buts :

$$\begin{cases} R_i(s,a) = 1 & \text{si } s \text{ est dans la zone but } i \\ R_i(s,a) = -1 & \text{si le couple } (s,a) \text{ amène l'agent vers un mur} \\ R_i(s,a) = 0 & \text{dans les autres cas} \end{cases} \tag{1.16}$$

Nous allons voir ce qui se passe si, à des instants particuliers, nous changeons totalement la fonction récompense. Supposons que l'agent ait planifié (calculé la fonction de valeur) pour aller à la zone but 1 (associée à la récompense R_1). Remplaçons soudainement (à l'instant t_0) R_1 par R_2. Au bout de 10 itérations, remplaçons de nouveau R_2 par R_3. La figure 1.9 illustre l'évolution de $|V_t - V_{t-1}|$, qui est proportionnelle à une borne supérieure de l'erreur pour le calcul de la fonction de valeur courante (cf. équation 1.15 page 25).

Si on observe sur ce graphe des pics d'erreur à chaque fois qu'on présente une nouvelle récompense, on voit que ceux-ci tendent tout de suite à disparaître : autrement dit, le calcul de la fonction de valeur s'adapte continuellement au changement de tâche. La figure 1.10 complète l'illustration de ce processus d'adaptation en montrant l'évolution de la fonction de valeur et de la politique calculée dans les itérations qui suivent t_0 (le passage de R_1 à R_2). La fonction de valeur et le plan sont mis à jour dès l'apparition d'une nouvelle tâche ; on observe de nouveau la propagation de valeur et du plan à partir de la zone but.

1.4.2 Evolution continue et interactive de la fonction récompense

Considérons à présent, sur le même exemple, le cas où la fonction de récompense est continuellement variable. Pour ce faire, supposons qu'elle est liée à trois besoins évoluant au cours du temps. Nous noterons l'ampleur numérique à l'instant t des trois besoins respectivement $b_1^{(t)}$, $b_2^{(t)}$

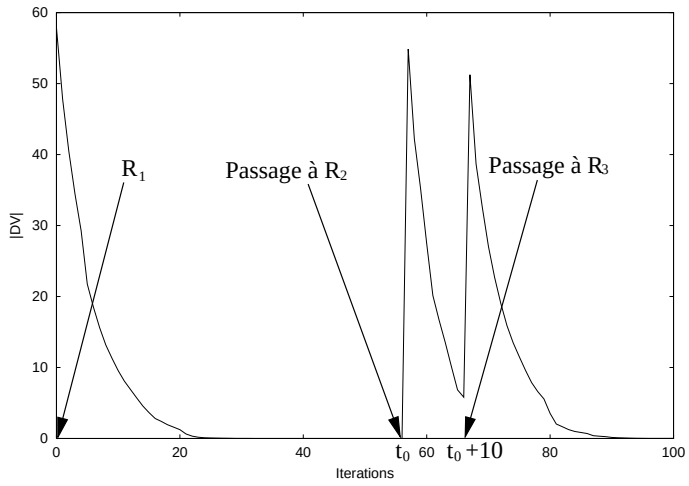

FIG. 1.9: **Variation de la fonction de valeur au cours du temps lorsqu'on change subitement de fonction récompense.** A des instants particuliers qui sont indiqués sur le graphique, la fonction récompense du problème change. En ordonnée, nous avons tracé la variation de la fonction de valeur qui est un indicateur de l'erreur commise par rapport à la fonction de valeur optimale ($|V_{t+1} - V_t| \propto |V^* - V_t|$). L'algorithme *Value Iteration* peut être vu comme un système dynamique qui tend en permanence vers un attracteur implicitement défini par les paramètres R et T du PDM. Quand on change ces paramètres, cela change l'attracteur et le calcul impliqué par *Value Iteration* s'adapte.

et $b_3^{(t)}$. La fonction de récompense est alors définie comme suit :

$$\begin{cases} R^{(t)}(s,a) = b_i^{(t)} & \text{si } s \text{ est dans la zone but } i \\ R^{(t)}(s,a) = -1 & \text{si le couple } (s,a) \text{ amène l'agent vers un mur} \\ R^{(t)}(s,a) = 0 & \text{sinon} \end{cases} \tag{1.17}$$

Supposons de plus que l'interaction de l'agent avec l'environnement joue un rôle dans l'évolution des besoins. Chacun des 3 besoins peut être satisfait si l'agent se rend dans la zone but correspondante : le besoin correspondant diminue alors très vite jusqu'à être finalement rassasié. Le reste du temps, les besoins augmentent lentement. Ainsi la seule façon de les contenir tous est de passer régulièrement dans chacune des 3 zones. Par exemple l'évolution des besoins peut suivre les lois suivantes :

$$\begin{cases} b_i^{(t+1)} = \max(0, b_i^{(t)} - .25) & \text{si l'agent est dans la zone but } i \\ b_i^{(t+1)} = b_i^{(t)} + 0.02 & \text{sinon} \end{cases} \tag{1.18}$$

Le protocole expérimental consiste à alterner à chaque itération t :
- une demande de déplacement de l'agent déduite de l'action optimale courante ;
- une mise à jour de besoins ;
- une itération de l'algorithme *Value Iteration* sur tous les états.

Les figures 1.12 et 1.13 illustrent l'évolution de la fonction de valeur pendant les 400 premières itérations (au début il n'y a aucun besoin). La figure 1.11 décrit la sémantique des dessins employés : la position de l'agent est représentée par une petite boule sur la surface. Les niveaux des 3 besoins sont donnés par la taille de boules en bas du graphique. La plupart du temps, les

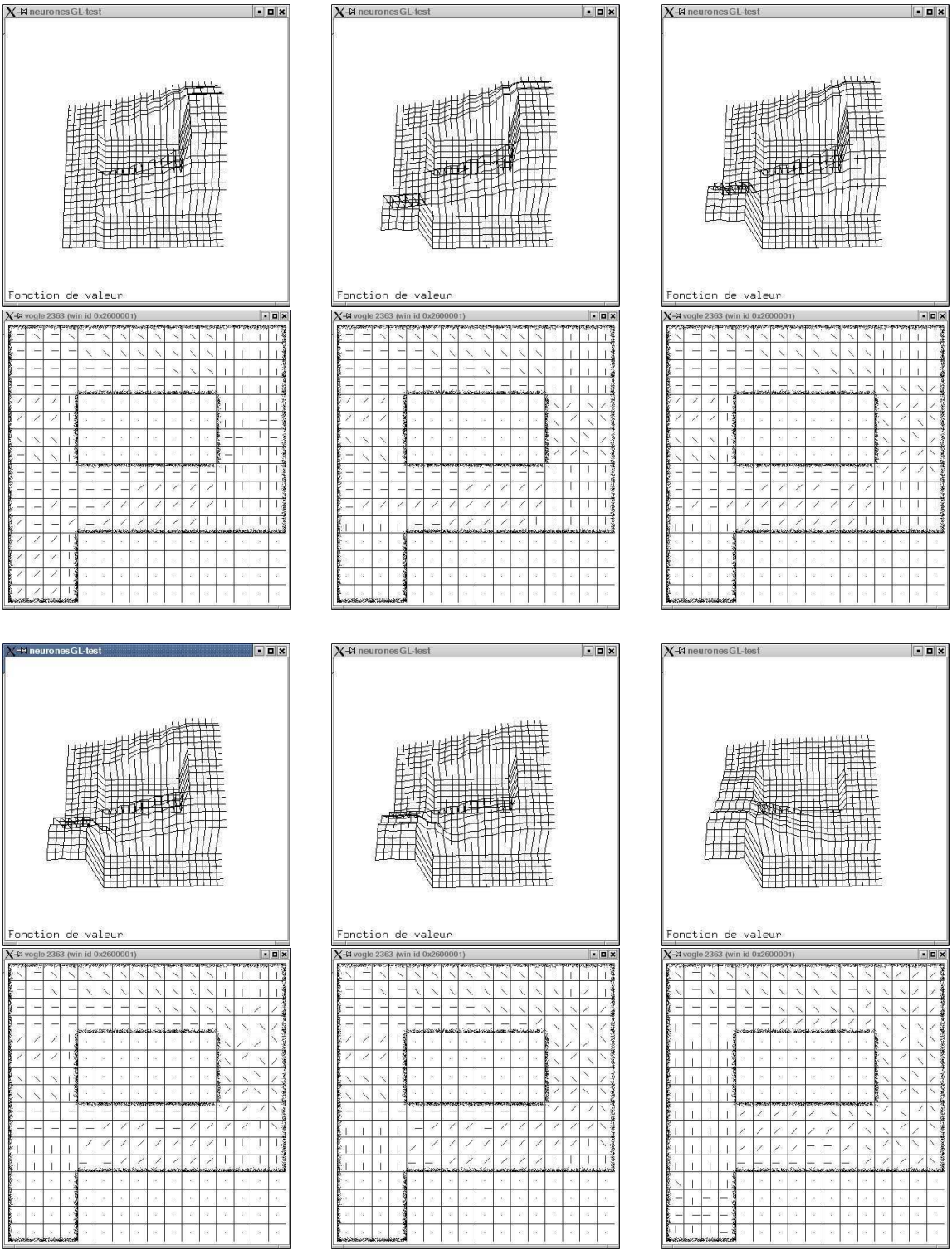

FIG. 1.10: **Adaptation de la fonction de valeur et du plan lors d'un changement de récompense.** Les instants présentés sont $t = t_0$, $t_0 + 1$, $t_0 + 2$, $t_0 + 3$, $t_0 + 4$, $t_0 + 10$ où t_0 est l'instant où l'on a changé la fonction récompense. Voir la figure 1.4 page 27 pour la représentation de la politique.

besoins augmentent doucement ce qui se traduit par une augmentation de la fonction de valeur dans les zones correspondantes. Lorsque l'agent atteint une zone but, le besoin correspondant est rapidement satisfait (le diamètre de la boule correspondante diminue jusqu'à disparaître)[7].

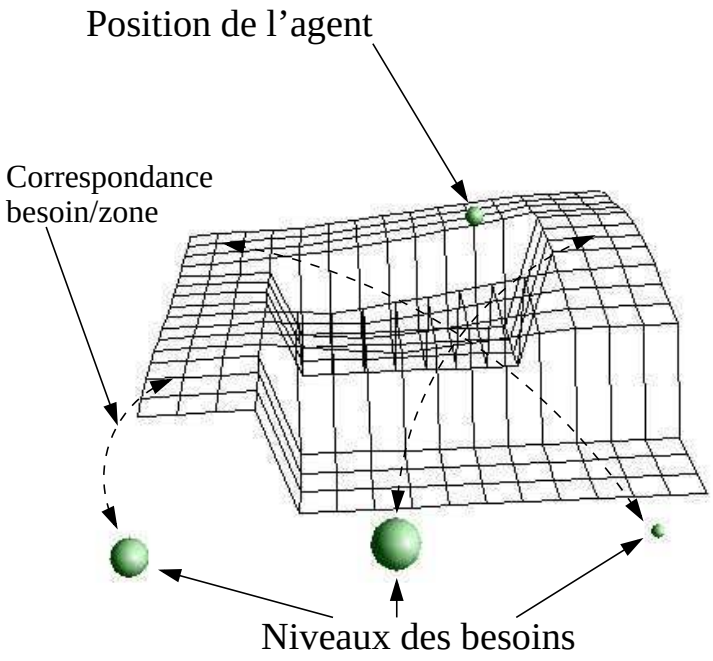

FIG. 1.11: **Légende pour les figures 1.12 et 1.13.** Les niveaux des trois besoins sont représentés par les diamètre des boules qui sont en bas du graphique. La position de l'agent est indiquée par une petite boule sur la surface.

Sur l'exemple que nous avons décrit, on observe que la fonction de valeur s'adapte vite aux fluctuations des besoins. La figure 1.14 montre que très rapidement, le comportement de l'agent tend à limiter le niveau maximum de l'ensemble des trois besoins. Pour ce faire, nous pouvons même remarquer que l'agent semble suivre une stratégie qui consiste à passer toujours dans le même ordre d'une zone à la suivante.

1.4.3 Conditions nécessaires à une bonne dynamique

Dans l'expérience artificielle que nous avons menée, l'efficacité avec laquelle la fonction de valeur s'adapte, ou encore "oublie" une tâche et passe à la suivante est liée à la combinaison de plusieurs facteurs. Les deux principaux facteurs[8] sont la taille de l'espace d'états et le facteur de pondération γ.

[7]Une démonstration animée d'un problème analogue peut être visualisée à l'URL suivante *http ://www.loria.fr/equipes/maia/demos/demos_bruno/*

[8]Il est plus difficile d'analyser dans le cas général l'influence de la vitesse d'évolution des paramètres R et T. L'existence de l'opérateur max dans l'équation caractéristique de la fonction de valeur optimale fait qu'une variation minime de ces paramètres peut engendrer des grosses différences sur la fonction de valeur.

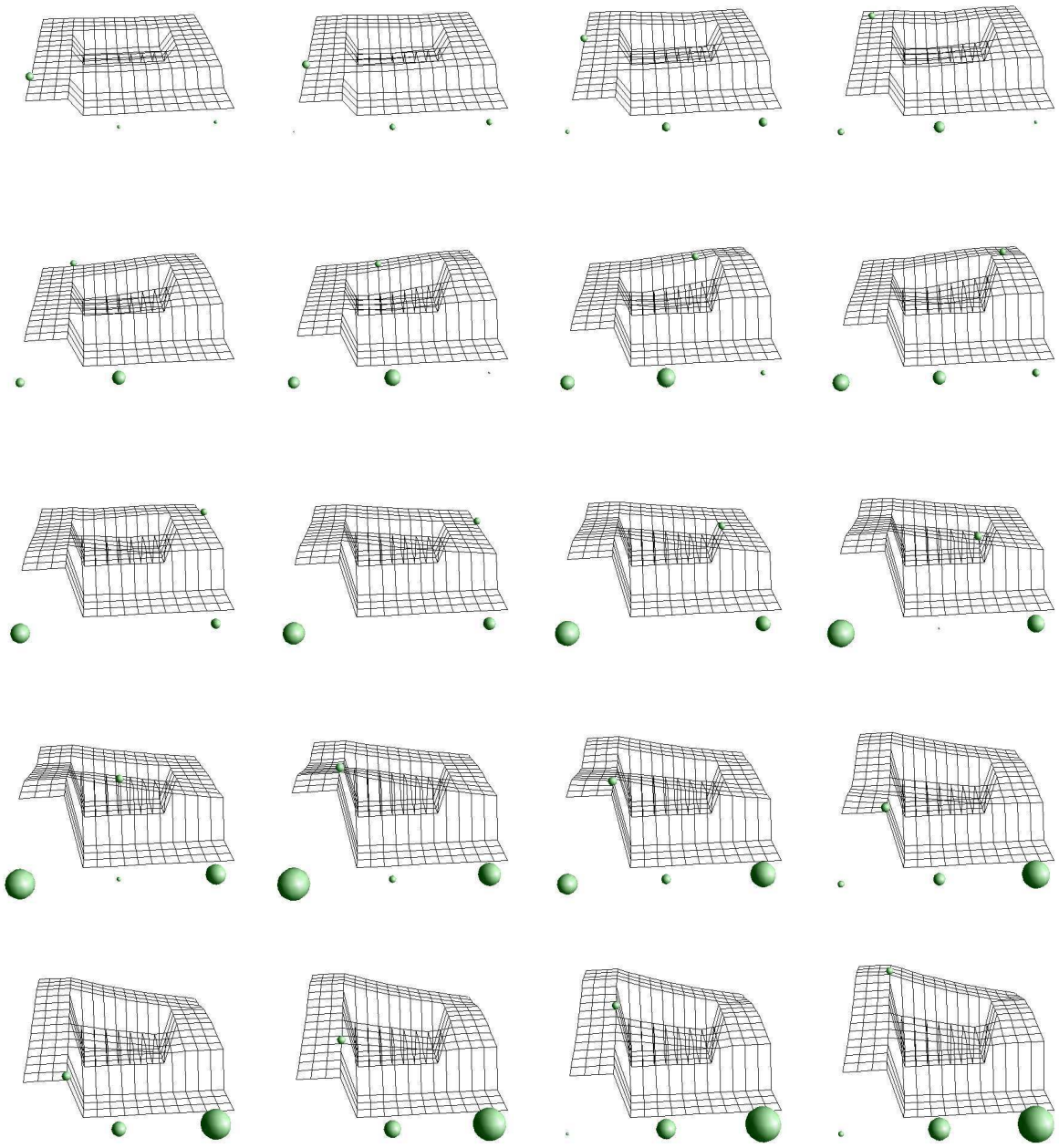

FIG. 1.12: **Evolution de la fonction de valeur lors de la variation continue et interactive de la récompense à** $t = 0, 10, 20, .., 190$**.** L'agent est confronté à trois besoins qui correspondent chacun à une zone but. Ces besoins augmentent lentement au cours du temps. Lorsque l'agent atteint une des zones buts, le besoin correspondant diminue rapidement jusqu'à devenir nul. La dynamique de ce système peut se décrire comme une boucle : besoin, récompense, fonction de valeur, politique, besoin. Les besoins courants définissent la fonction de récompense. La fonction de récompense fait évoluer le calcul de la fonction de valeur. De chaque fonction de valeur on peut déduire une politique. L'interaction impliquée par la politique influence à son tour les besoins.

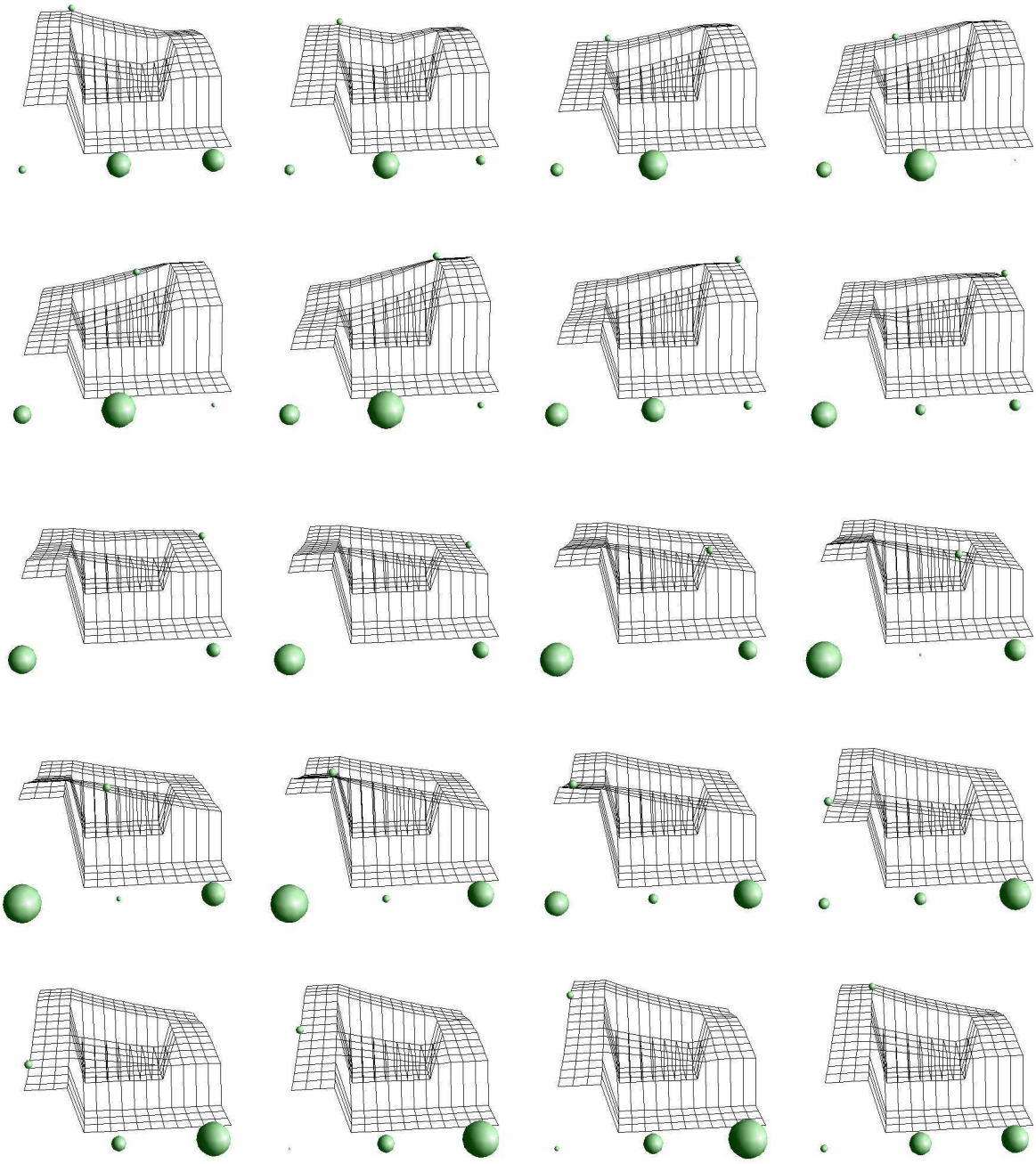

FIG. 1.13: **Evolution de la fonction de valeur lors de la variation continue et interactive de la récompense à** $t = 200, 210, 220, .., 390$**.** Voir la figure précédente.

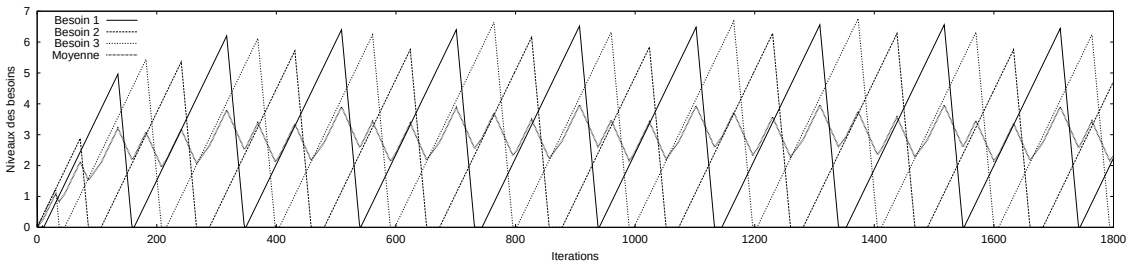

FIG. 1.14: **Evolution des besoins au cours de l'expérience sur l'adaptation à une fonction récompense continuellement variable.** Ce graphique représente le niveau de chaque besoin ainsi que la moyenne de ces niveaux au cours du temps. On observe que les besoins sont successivement rassasiés. Leur niveau moyen reste dans un intervable constant.

Taille de l'espace et nouveau plan

Il n'est pas nécessaire de considérer le processus que nous venons de décrire comme une adaptation. On peut simplement le voir comme une perpétuelle planification. Comme nous l'avons dit au sujet de la complexité, la taille de l'espace d'états joue un rôle crucial dans la planification. Il joue donc aussi un rôle crucial pour l'adaptation. La vitesse avec laquelle un changement des paramètres pourra être correctement répercuté dépendra du temps nécessaire pour propager la valeur et donc de la taille de l'espace d'états.

Facteur de pondération et oubli

Le paramètre γ joue également un rôle crucial dans l'adaptation du système. Plus γ est proche de 1 et plus l'adaptation prendra du temps. Ceci peut s'expliquer intuitivement en rappelant l'interprétation de γ en terme d'horizon de planification développée page 30 : plus γ est proche de 1, plus l'horizon de planification est lointain et plus le temps nécessaire pour oublier une tâche afin de passer à la suivante sera long.

En pratique il faut corréler la taille de l'espace d'états, la vitesse de variation de la récompense et l'horizon de planification de l'agent. Dans les expériences que nous avons faites sur l'adaptation, les valeurs $|S| = 256$, $\gamma = 0.95$ se sont avérées bien adaptées.

1.5 L'approche A/R

Dans la présentation des PDM que nous venons de faire, nous avons montré comment calculer le meilleur comportement (celui qui a la plus grande valeur). Ce calcul nécessite la donnée d'un modèle de l'interaction agent/environnement (S, A, T, et R). Calculer les actions à faire pour chaque état qu'on pourrait rencontrer est une forme de plan. En ce sens, le fait d'exploiter un PDM comme nous l'avons fait jusqu'ici pour trouver une politique optimale relève plus d'une problématique de *planification* que d'*apprentissage*.

Dans le domaine computationnel de l'A/R, on suppose généralement que les paramètres T et/ou R sont (au moins partiellement) inconnus. C'est dans ce cadre qu'apparaissent les caractéristiques que nous avons présentées dans l'introduction de ce mémoire. L'A/R impose à l'agent d'interagir, de faire des essais-erreurs afin de découvrir les caractéristiques de son interaction avec l'environnement. Lorsque l'interaction entre l'agent et l'environnement se fait en ligne (c'est-à-dire lorsque l'agent doit "se déplacer" dans son environnement et observer les résultats),

un dilemme important apparaît : celui entre l'exploration du monde (pour améliorer ses connaissances et mieux planifier) et l'exploitation de ses connaissances actuelles (pour obtenir plus de récompenses).

L'objet de cette dernière section est de décrire les problèmes et les solutions que l'approche A/R rajoute au problème traité par les PDM : l'évaluation des paramètres et le dilemme exploration/exploitation.

1.5.1 Estimation des paramètres : avec ou sans représentation interne ?

Comment peut-on faire pour calculer une politique si la fonction de récompense R et/ou la dynamique T sont inconnues ? Les travaux existants exploitent le principe fondamental suivant : utiliser des interactions (des échantillons de R et T) pour estimer ces fonctions inconnues.

Selon qu'elles utilisent ou non une *représentation interne*, on range habituellement les approches qui abordent ce problème d'apprentissage (modèle non complètement connu) dans deux catégories. Par représentation interne, on entend tout objet informatique qui représente une estimation des données inconnues (R et/ou T).

- L'*apprentissage direct* propose de calculer directement la politique optimale sans évaluer *explicitement* ni la fonction de transition T ni la fonction de récompense R. La plupart des algorithmes consiste à appliquer des techniques dites d'approximation stochastique [Jaakkola *et al.*, 1994] pour estimer la Q-fonction optimale Q^{π^*} directement en utilisant uniquement des échantillons de R et T. On en déduit alors une politique optimale à l'aide de l'équation 1.13. L'algorithme *Q-learning* [Watkins, 1989] a rendu célèbre cette approche. On voit dans sa description (algorithme 1.3) qu'on met directement à jour l'estimation de Q.

Algorithme 1.3 Q-Learning

Données : Un PDM $\langle S, A, T, R\rangle$

But : Calculer la fonction de valeur optimale Q^{π^*}

Initialisation quelconque de Q, d'un état initial s et d'une politique d'exploration π.

répéter indéfiniment

Expérimenter : choisir une action a selon une politique quelconque et observer la récompense r et l'état suivant s'.

$Q(s,a) \leftarrow (1-\alpha).Q(s,a) + \alpha.(r + \gamma.\max_{a' \in A} Q(s',a'))$

$s \leftarrow s'$

fin répéter

- L'*apprentissage indirect* consiste à construire un modèle du monde (des estimations de R et T) et à exploiter ce dernier à l'aide d'algorithmes de planification comme *Value Iteration*. L'algorithme 1.4 montre une manière naturelle (il s'agit de l'estimateur standard moyen) pour mettre à jour des estimations R et T à partir d'expériences (s,a,s'). Le principe de maintenir de telles estimations a été initialement introduit via l'architecture DYNA [Sutton, 1991]. L'idée était initialement la suivante : avoir un modèle (même approché) du monde permet d'effectuer un certain nombre d'expériences "mentalement" et donc de converger bien plus vite que les algorithmes de l'apprentissage direct (comme le *Q-Learning*). L'utilisation pratique d'une telle architecture est par exemple illustrée dans [Peng et Williams, 1993] [Sutton, 1990].

Si l'apprentissage direct présente l'avantage immédiat de ne pas avoir à stocker en mémoire des estimations de T et R, il met en général beaucoup plus de temps pour converger vers une

Algorithme 1.4 Estimation des paramètres R et T à partir d'échantillons $(s_0, a_0, r_0, s_1, ...)$

$R(s,a) \leftarrow \frac{\sum_{(s_i,a_i)=(s,a)} r_i}{\#(s,a)}$
$T(s,a,s') \leftarrow \frac{\#(s,a,s')}{\#(s,a)}$
où $\#(x)$ désigne le nombre de fois qu'on trouve la séquence x dans les échantillons.

politique de qualité. Nous tombons ici sur un compromis bien connu de l'informatique : le compromis mémoire utilisée/vitesse de calcul. L'approche directe favorise la mémoire tandis que l'approche indirecte favorise la vitesse.

1.5.2 Le dilemme exploration/exploitation

Généralement, l'approche A/R signifie de plus que l'agent doit apprendre à agir le long d'*une seule* trajectoire (on pourrait dire au cours de son unique vie). L'approche A/R est ainsi un peu plus qu'un problème de planification. A l'origine, un agent ne sait rien ou presque, et petit à petit, il doit à la fois récupérer des informations sur son interaction avec l'environnement et les exploiter. A tout instant, l'agent est confronté à un dilemme. Il peut *exploiter* ses connaissances imparfaites afin d'obtenir *tout de suite* la plus grande récompense. Ou encore il peut *explorer* d'autres possibilités, élargir ses connaissances, pour trouver comment avoir encore plus de récompense *plus tard*. Le dilemme exploration/exploitation (ou court-terme/long-terme) fait que si l'agent utilise exclusivement l'un des deux aspects de l'alternative, il a de fortes chances d'avoir une faible quantité de récompenses [Kumar, 1985]. Le cas où les paramètres R et T sont stationnaires est celui qui a été le plus étudié. Nous présentons les principales réponses qui traitent ce problème. Nous discutons ensuite du cas où les paramètres peuvent varier.

Dans un modèle stationnaire

Nous présentons ici quelques réponses données au dilemme exploration/exploitation dans des problèmes d'A/R stationnaires. On distingue des approches ad hoc et des approches formelles. Notre courte présentation s'inspire ici fortement de [Fabiani *et al.*, 2001].

Des méthodes particulèrement simples et ad hoc sont proposées par [Bertsekas et Tsitsiklis, 1996] [Kaelbling *et al.*, 1996] :

- On alterne régulièrement politique aléatoire (exploration) et politique gourmande (exploitation) : pendant N_1 instants consécutifs, on tire uniformément l'action dans A, puis pendant N_2 on choisit la politique donnée par la *Q-fonction* courante.
- On effectue un tirage semi-uniforme : on suit la meilleure action courante avec une probabilité $1-\tau$ et on tire uniformément l'action dans A avec la probabilité τ.
- On tire l'action dans A selon une distribution de Boltzmann basée sur les *Q-valeurs* $Q_t(s,a)$ courantes en introduisant une quantité θ (souvent appelée température). Dans ce cas, la probabilité de choisir l'action a à l'instant t est :

$$P_\theta(a_t = a) = \frac{e^{\frac{Q_t(s,a)}{\theta}}}{\sum_{a' \in A} e^{\frac{Q_t(s,a')}{\theta}}} \tag{1.19}$$

  Une telle approche favorisera les actions qui ont une valeur importante sans pour autant empêcher l'exploration.

Le choix des paramètres ainsi introduits (N_1, N_2, τ, ou θ), de même que leur évolution au cours du temps, est alors crucial pour obtenir de bons résultats. Pratiquement, il est usuel de

commencer par une forte exploration ($N_1 \ll N_2$, $\tau \ll 1$, $\theta \gg 1$) et de progressivement tendre vers l'exploitation ($N_1 \gg N_2$, $\tau \gg 1$, $\theta \ll 1$).

Les méthodes formelles, plus sophistiquées, s'inspirent de résultats théoriques concernant le même dilemme pour un problème mathématique : le bandit-manchot[9] à k bras. L'idée est qu'on peut voir un PDM comme un ensemble de machines à sous indépendantes où chaque bras correspond à un couple état-action (voir [Kaelbling *et al.*, 1996] pour une description précise). La récompense obtenue en faisant l'action a dans l'état s correspond au gain de la machine correspondante. Le manque de connaissances statistiques sur le gain qu'on peut obtenir sur chacune des machines est traduit par un bonus (d'exploration) que l'on rajoute à la valeur $R(s, a)$. Ce bonus peut être défini localement [Kaelbling, 1993]. Dans [Meuleau et Bourgine, 1999], les auteurs montrent que le fait de propager ce bonus d'états en états (selon la dynamique du monde) permet d'améliorer sensiblement les performances.

Dans un modèle non-stationnaire

L'algorithme *Value Iteration* est dynamique : nous avons montré qu'il était capable de planifier même si les paramètres du modèle sont en train de varier, autrement dit lorsque le modèle est non-stationnaire. Dans le cas où les paramètres ne sont pas directement accessibles, leur non-stationnarité complique significativement le dilemme exploration/exploitation. Lorsque le modèle est non stationnaire, l'agent apprenant par renforcement doit prendre en compte le caractère récent ou non de ses expériences. Plus le temps se sera écoulé, plus les paramètres auront pu évoluer.

Il existe dans la littérature assez peu de travaux qui abordent le problème de l'A/R avec des paramètres non stationnaires. La seule architecture de la littérature qui à notre connaissance, propose une réponse (heuristique) au dilemme exploration/exploitation dans le cas non-stationnaire est l'architecture Dyna-Q+ [Sutton, 1990]. Dans la même philosophie que ce que nous avons présenté un peu plus haut, le désir d'exploration est ici traduit par un bonus de récompense pour chaque paire état-action. En particulier, si une transition n'a pas été essayée pendant un temps δt, la récompense qui lui est associée est augmentée de $\kappa.\sqrt{\delta t}$ avec κ petit. Ceci encourage l'agent à continuer de tester toutes les transitions et même à planifier de longues séquences d'actions pour effectuer tous ces tests de manière optimale.

Résumé du chapitre

Nous avons introduit le problème de l'A/R. Un agent en interaction avec un environnement doit maximiser sur le long terme les récompenses qu'il reçoit. Le formalisme PDM reprend les éléments qui sont essentiels pour décrire un tel problème : les *états* S, les *actions* A, les *transitions* d'états T vérifiant l'hypothèse de Markov et les *récompenses* R. Ces quatre éléments permettent une description générique des problèmes d'A/R. Lorsqu'un PDM est défini, le problème consistant à trouver un comportement optimal, c'est-à-dire une politique maximisant les récompenses sur le long terme, se ramène au calcul d'une fonction sur l'espace des états : la *fonction de valeur optimale*.

Nous avons présenté en détail l'algorithme *Value Iteration* qui permet de calculer une telle fonction. Nous avons montré qu'il avait un certain nombre de qualités appréciables. Il met en jeu des calculs arithmétiques simples (additions, multiplications, et opérateur *max*). Il est *anytime* : plus le temps alloué est grand et plus son calcul de la fonction de valeur optimale est précis. Il

[9] *Bandit Problem* en anglais.

est *dynamique* et *multi-tâche* : si les paramètres R ou T évoluent au cours du temps, le calcul effectué par *Value Iteration* s'adapte à ces variations. Nous avons en particulier souligné que la taille de l'espace d'états est un paramètre crucial pour contenir la complexité et pour assurer les caractéristiques dynamique et multi-tâche.

Nous avons fait la distinction entre deux déclinaisons du problème posé par le formalisme PDM : la *planification* met en jeu un PDM dont on connaît tous les paramètres tandis que l'*apprentissage* ne présuppose pas la connaissance de la dynamique d'interaction et de la fonction récompense. Nous avons distingué les deux grandes approches pour répondre au problème plus difficile de l'apprentissage : l'*approche directe* et l' *approche indirecte*. Le choix entre les deux relève du compromis informatique mémoire/rapidité. Nous avons enfin mis en valeur l'une des difficultés centrales du problème d'apprentissage : le dilemme exploration/exploitation.

Chapitre 2

Une réponse connexionniste au problème de l'A/R

Le chapitre précédent nous a permis de formaliser le problème de l'A/R. Ce chapitre va le présenter sous un angle original : celui du connexionnisme. Nous allons en effet montrer que nous pouvons sans gros effort (il s'agit pour l'essentiel d'une réécriture de la partie précédente) proposer une architecture connexionniste générique qui répond au problème de l'A/R. Nous le faisons dans le cas où l'espace d'états est petit. Nous supprimerons cette hypothèse dans la deuxième partie du mémoire.

Nous commençons par présenter et justifier un choix structurel important pour cette architecture. Nous argumentons 1) qu'un espace d'états petit et 2) l'éventuelle gestion de plusieurs tâches suggèrent d'aborder le problème de l'A/R avec une approche indirecte : le problème de l'A/R se découpe alors en l'estimation des paramètres du modèle *et* la planification à partir de ces estimations.

Dans un deuxième temps, nous décrivons l'architecture que nous proposons sous une forme connexionniste, c'est-à-dire en termes d'unités et de connexions. Les estimations des fonctions T et R (propre à l'approche indirecte) se traduisent respectivement par un apprentissage des connexions et un apprentissage des unités. L'algorithme *Value Iteration* s'y intègre sous la forme d'une loi d'évolution d'activité au sein du réseau.

Finalement, nous justifions l'appellation connexionniste. Nous montrons qu'une telle architecture est naturellement parallélisable et donc significativement accélérable. Puis nous analysons et évaluons expérimentalement sa tolérance aux pannes.

2.1 Pour une approche indirecte

Comme nous l'avons vu précédemment (section 1.5.1), le problème de l'A/R peut être résolu avec une approche directe ou une approche indirecte. Dans la première approche, l'agent calcule directement la fonction de valeur optimale. Dans l'approche indirecte, l'agent construit une estimation des paramètres R et T et utilise une technique de planification du PDM alors implicitement défini. Nous allons ici donner deux arguments en faveur de l'approche indirecte.

2.1.1 Un argument général lié à la taille de l'espace d'états

Le choix entre l'approche directe et indirecte peut être vu, nous l'avons dit à la section 1.5.1, comme un compromis mémoire/vitesse de calcul. Dans le cas où l'espace d'états est fini,

il est généralement plus efficace d'avoir une approche indirecte : en effet, lorsqu'il n'est plus envisageable (du point de vue temps de convergence) d'utiliser une approche directe, il n'y a pas de problème de mémoire avec l'approche indirecte et les ordinateurs d'aujourd'hui. On peut considérer qu'à partir de 10^5 états, l'algorithme standard de l'approche directe Q-Learning devient impratiquable[10]. Si on a 10^5 états et 100 transitions possibles en moyenne, il faut stocker 10^7 probabilités de transitions par action soit environ 70 Mégaoctets de RAM par action. C'est donc à la portée des ordinateurs actuels. Sans rien ôter aux mérites des travaux fondateurs sur l'approche directe _ les avancées théoriques considérables qu'elles ont induites lorsqu'on utilise par exemple des approximateurs de fonctions [Sutton, 1996] dans le cas infini suffisent à les justifier mais cela sort de notre propos (nombre fini d'états) _ il est préférable d'utiliser une approche indirecte lorsqu'on a un espace d'états fini. Ceci est d'ailleurs d'autant plus vrai que l'espace d'états est petit.

2.1.2 Un argument lié à la multiplicité des tâches

La deuxième raison que nous invoquons pour le choix de l'approche indirecte découle de la volonté qu'elle puisse être multi-tâche. Dans le jargon de l'A/R, cette volonté se traduit ainsi : on envisage la possibilité que les paramètres R et T puissent varier. Dans ce cas le modèle sous-jacent au problème que nous traitons serait un PDM ayant plusieurs fonctions transitions et récompenses : $\langle S, A, (T_1, .., T_n), (R_1, .., R_n)\rangle$.

Qu'on choisisse une approche directe ou indirecte, l'agent doit apprendre (mémoriser des régularités du monde) et planifier (exploiter ces régularités). Comme nous l'avons évoqué à la section 1.5.1 page 39, le maintien d'une représentation interne peut accélérer la convergence vers la fonction de valeur optimale pour *une seule* tâche (T_i, R_i). Lorsqu'on en a *plusieurs*, et particulièrement lorsque celles-ci ne sont pas complètement décorrélées, la représentation interne apprise pour une tâche peut servir pour les suivantes. Ainsi, tâche après tâche, cette représentation interne est de plus en plus précise et la résolution (qui se résume à la découverte de ce qui a changé puis à une adaptation conséquente de la planification), est de plus en plus rapide. A contrario, l'approche directe ne capitalise pas les expériences mais s'en sert uniquement pour estimer la *Q-fonction* optimale de la tâche courante ; lorsqu'on change de fonction récompense, il faut alors relancer l'algorithme : en particulier il faut réinitialiser le coefficient d'apprentissage α (voir algorithme 1.3 page 39).

Pour illustration, la figure 2.1, schématise l'avantage en terme de temps de l'approche indirecte sur l'approche directe lorsque plusieurs tâches partagent la même fonction de transition : plus le nombre de tâches augmente et plus cet avantage est décisif. Dans la mesure où l'estimation de la dynamique T d'une tâche sert pour les suivantes on peut considérer que le maintien de sa représentation interne permet un transfert d'apprentissage entre des tâches, et donc une accélération significative sur le long terme.

2.2 Description du réseau

Nous passons maintenant à la description précise de notre architecture sous la forme de réseau connexionniste. Nous définissons les unités et les connexions qui la constituent. Les unités correspondent aux états d'un PDM et les connexions aux transitions possibles entre ces états. Nous définissons ensuite les lois d'apprentissage des connexions et des unités : celles-ci corres-

[10] A notre connaissance l'utilisation du Q-Learning dans la littérature n'atteint jamais ce nombre (on trouve 10000 dans [Smart, 2002]).

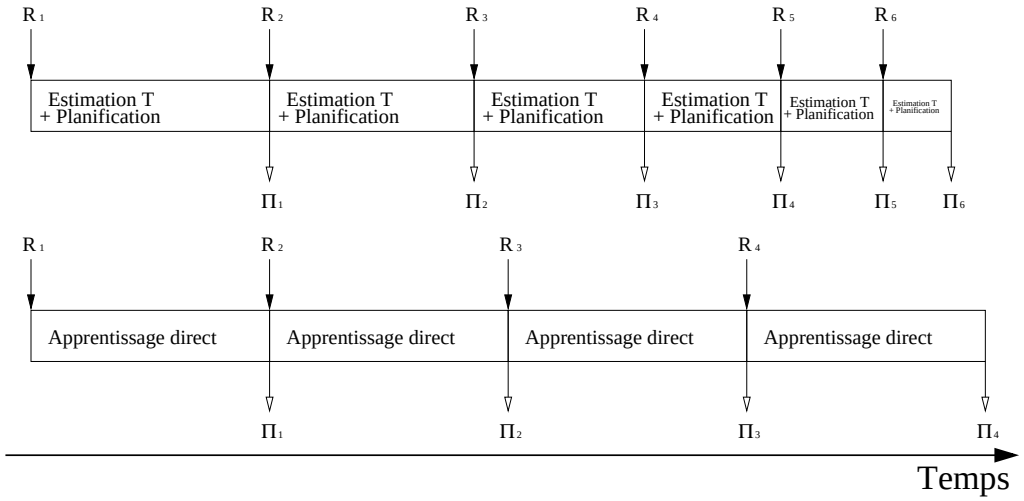

FIG. 2.1: **Comparaison de l'apprentissage direct (en bas) et de l'apprentissage indirect (en haut) pour la gestion de plusieurs fonctions récompenses.** Avec l'apprentissage indirect, les tâches définies par les fonctions récompenses sont résolues de plus en plus vite : La capitalisation de l'estimation de la fonction de transition T faite pour la tâche 1 sert au calcul de la tâche 2 et ainsi de suite...

pondent à l'estimation des paramètres T et R. Nous présentons enfin la valeur d'un état comme l'activité d'une unité. L'algorithme *Value Iteration* nous permet de spécifier la loi d'évolution de cette activité.

2.2.1 Définition des unités et des connexions

L'architecture que nous proposons s'appuie sur le formalisme PDM. A toute situation ou état du système $s \in S$, nous associons (nous identifions) une unité. Deux unités pourront communiquer à l'aide de connexions orientées *typées* : il y a un type pour chaque action $a \in A$. Notons $c^a_{ss'}$ la connexion de type $a \in A$ qui va de s à s'. Le réseau que nous proposons est complètement connecté : de toute unité partent des connexions de *tous les* types vers *toutes les* unités. Ce réseau contient donc $|S|$ unités et $|A|.|S|^2$ connexions. Il s'agit d'un réseau complètement connecté. Un exemple est illustré figure 2.2.

Toute unité s possède 5 variables :
- E : un booléen qui vaut $Vrai$ si et seulement si l'unité correspond à l'état du système ;
- $R[|A|]$: un tableau de $|A|$ réels pour stocker la récompense lorsqu'on fait l'action a ;.
- $Q[|A|]$: un tableau de $|A|$ réels pour stocker les *Q-valeurs* ;
- V : un réel qui correspond à la valeur de l'unité ;
- π : un élément de A correspondant à l'action à faire lorsque l'unité correspond à l'état du système..

Toute connexion $c^a_{ss'}$ est caractérisée par son poids :
- T : un réel pour stocker l'estimation de la probabilité de transition $T(s, a, s')$.

Dans la suite, nous utiliserons la notation $o.x^{(t)}$ pour désigner (à la manière des pointeurs informatiques) la variable x de l'objet o à l'instant t.

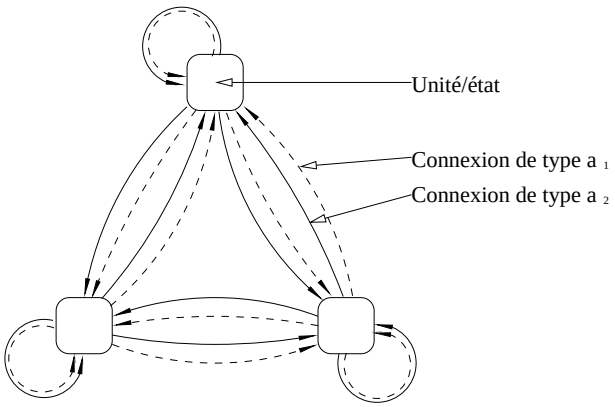

FIG. 2.2: **Un exemple de réseau comportant 3 états et 2 actions.** A chacun des 3 états correspond une unité. De chaque unité partent des connexions de 2 types (un type par action) vers toutes les unités. Ce réseau comporte donc $3 \times 2 \times 3 = 18$ connexions.

2.2.2 Lois d'apprentissage des connexions

Le vocable "apprentissage" peut avoir des significations diverses. Lorsqu'on parle de réseau connexionniste, il fait généralement référence aux lois de modification des poids des connexions (par analogie aux lois de modification des forces synaptiques dans le cerveau). Les connexions de notre réseau vont servir de support de codage pour l'estimation de la fonction de transition. Nous allons décrire la façon dont nous pouvons simplement faire cette estimation.

La loi d'apprentissage que nous proposons est décrite par l'algorithme 2.1. Ce genre de loi

Algorithme 2.1 Mise à jour du poids de la connexion $c^a_{ss'}$ à l'instant t

si $s.E^{(t-1)} = Vrai$ et $a^{(t-1)} = a$ **alors**
  $c^a_{ss'}.T \leftarrow (1-\alpha) \times c^a_{ss'}.T$
  si $s'.E^{(t)} = Vrai$ **alors**
    $c^a_{ss'}.T \leftarrow c^a_{ss'}.T + \alpha$
  fin si
fin si

est généralement qualifiée de loi *hebbienne* dans la mesure où elle est similaire à l'hypothèse faite par Hebb [Hebb, 1949] à propos de la façon dont les poids synaptiques évoluent dans le cerveau en fonction de l'expérience. Ce dernier suggère que la force d'une synapse change proportionnellement aux activités des neurones qu'elle relie. En termes connexionnistes, nous dirions que le poids d'une connexion augmente lorsque les unités qu'elle relie sont excitées à des instants proches et diminue sinon. C'est le cas pour la loi qui nous intéresse ici : lorsqu'une connexion $c^a_{ss'}$ détecte le déclenchement de l'action a, elle augmente le poids de la connexion $(+\alpha.(1-T))$ si les états qu'elle relie ont été successivement activés ($s.E^{(t-1)} = Vrai$ et $s'.E^{(t)} = Vrai$) ; si seul l'état s était activé, le poids de la connexion diminue légèrement ($-\alpha.T$).

Il est important de noter que la loi d'apprentissage ne remet pas en cause la cohérence des poids en tant que matrice de transition (la somme des probabilités de transitions doit toujours

être égale à 1). En effet :

$$\sum_{s'} c^a_{ss'}.T^{(t)} = (1-\alpha).\sum_{s'} c^a_{ss'}.T^{(t-1)} + \alpha \quad (2.1)$$

$$= 1 \text{ à condition que } \sum_{s'} c^a_{ss'}.T^{(t-1)} = 1 \quad (2.2)$$

Si à $t = 0$, on part de poids cohérents (par exemple tous égaux à $\frac{1}{|A|.|S|}$), ces poids le resteront.

Cette loi d'apprentissage met en jeu un paramètre α que nous appelons taux d'apprentissage. La loi que nous proposons ainsi que l'utilisation d'un coefficient d'apprentissage se retrouvent dans des algorithmes connexionnistes de classification de la littérature qui impliquent eux aussi une estimation de variable aléatoire. Une bonne synthèse de ces algorithmes peut être consultée dans [Fritzke, 1997]. Nous allons montrer que le caractère stationnaire ou non du problème que l'on cherche à résoudre permet de choisir adéquatement ce taux d'apprentissage. L'analyse computationnelle que nous faisons ci-après est naturellement inspirée de celle qu'on trouve dans [Fritzke, 1997].

Problème non stationnaire : un taux d'apprentissage constant

Pendant l'apprentissage, $c^a_{ss'}.T$ peut être vu comme la probabilité moyenne de transition $s \xrightarrow{a} s'$ si on donne une importance variable aux échantillons sur lesquels on se base. Plus un échantillon sera récent (dans la liste des mises à jour de $c^a_{ss'}.T$), plus il aura d'importance dans cette moyenne. Pour cette moyenne en effet, le poids de l'échantillon qui a donné lieu à la dernière mise à jour est 1, celui de l'échantillon précédent est $(1-\alpha)$, celui d'avant $(1-\alpha)^2$, etc. Celui qui a donné lieu à la i^{eme} dernière mise à jour est $(1-\alpha)^i$. Lorsque le nombre de mises à jour m est grand, on montre en effet [Fritzke, 1997] que :

$$c^a_{ss'}.T \simeq \frac{\sum_{i=0}^{m-1}(1-\alpha)^i.\delta^{(m-i)}}{\sum_{i=0}^{m-1}(1-\alpha)^i} \quad (2.3)$$

avec

$$\begin{cases} \delta^{(i)} = 1 & \text{si le poids a été renforcé à la } i^{eme} \text{ mise à jour} \\ \delta^{(i)} = 0 & \text{sinon} \end{cases} \quad (2.4)$$

Ainsi, le poids d'une connexion *reflète* l'ensemble des transitions qui la concernent tout en les *oubliant* progressivement au cours du temps. Ce processus ne s'arrête jamais : chaque nouvel échantillon de transition peut provoquer des changements significatifs sur le poids. Ce genre de systèmes peut en théorie suivre une probabilité non stationnaire. Dans le cas où elle est stationnaire, les prototypes finissent par devenir quasi-stationnaires autour d'un minimum local de la distortion. Le choix du taux d'apprentissage relève d'un compromis : plus il est grand, plus cette position de quasi-équilibre est atteinte rapidement, mais plus les fluctuations autour de l'équilibre seront grandes. Ce compromis est illustrée figure 2.3.

Problème stationnaire : un taux d'apprentissage variable

Si on est sûr que les probabilités de transition sont stationnaires, on peut utiliser un taux d'apprentissage décroissant lentement vers 0. Par exemple, si à la i^{eme} mise à jour on prend $\alpha = \frac{1}{i}$, il est facile de voir qu'on tombe sur l'estimateur standard de la probabilité de transition que nous décrivions pour l'approche indirecte (voir algorithme 1.4 page 40) :

$$c^a_{ss'}.T = \frac{\#(s,a,s')}{\#(s,a)} \quad (2.5)$$

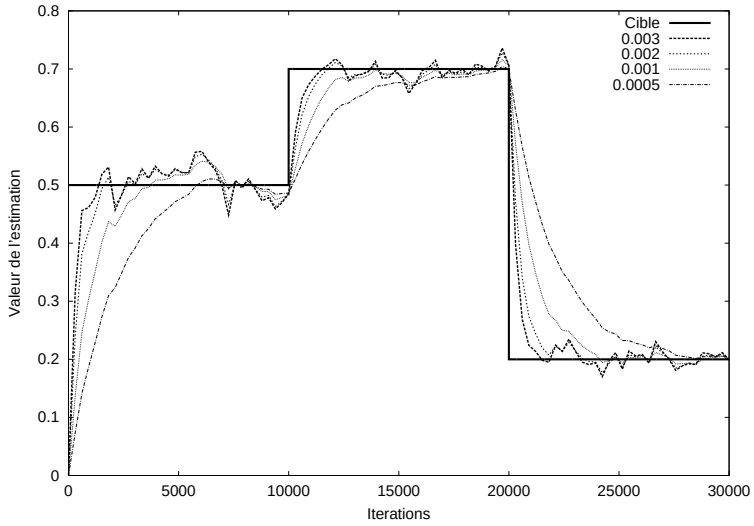

FIG. 2.3: **Apprentissage du poids d'une probabilité avec différents taux d'apprentissage constants.** Ce graphique montre l'évolution de plusieurs estimations d'une variable aléatoire. Cette variable aléatoire a un paramètre cible qui change à deux reprises : il vaut successivement $p = 0.5, p = 0.7, p = 0.2$. Plus le taux d'apprentissage est grand et plus l'estimation tend rapidement vers la cible. A contrario, plus ce taux est grand et plus l'estimation variera autour de la valeur cible. On observe des fluctuations semblables pour chacune des courbes car tous les apprentissages ont été réalisés à partir des mêmes échantillons.

où $\#(s,a)$ et $\#(s,a,s')$ sont respectivement le nombre de fois qu'on a effectué l'action a dans l'état s et le nombre de fois qu'alors, on est arrivé en s'. Cette convergence est illustrée figure 2.4

La figure 2.5 montre les connexions de poids non négligeables après apprentissage pour le problème de navigation discrète.

2.2.3 Lois d'apprentissage des unités

La technique d'estimation de la fonction de transition se transpose aisément à l'estimation de la fonction de récompense. Sa description est donnée par l'algorithme 2.2.

Algorithme 2.2 Mise à jour de la variable R de l'unité s à l'instant t

si $s.E^{(t)} = Vrai$ **alors**
  $s.R \leftarrow (1-\alpha) \times s.R + \alpha \times r_t$
fin si

A chaque fois que le système reçoit une récompense r_t, il met à jour la variable R de l'unité/état courante. Si la fonction de récompense cible est déterministe, on utilisera un taux d'apprentissage $\alpha = 1$. Si elle est stochastique, on fera comme pour l'estimation de T : on choisira un taux d'apprentissage constant si R est non stationnaire ou un taux d'apprentissage variable $\alpha = \frac{1}{i}$ pour la i^{eme} mise à jour si R est stationnaire. Les enjeux concernant le choix de ce taux d'apprentissage α sont les mêmes que pour l'estimation de T. Nous renvoyons le lecteur à la discussion que nous avons faite pour l'estimation de T par les connexions.

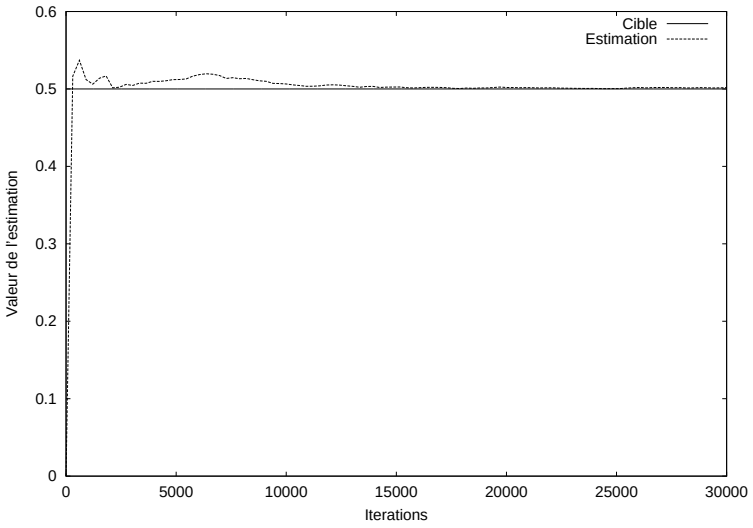

FIG. 2.4: **Evolution de l'estimation avec un taux d'apprentissage décroissant.** Lorsqu'une variable aléatoire a un paramètre cible constant, un taux d'apprentissage décroissant permet de tendre vers la valeur exacte. Ici le paramètre cible vaut $p = 0.5$

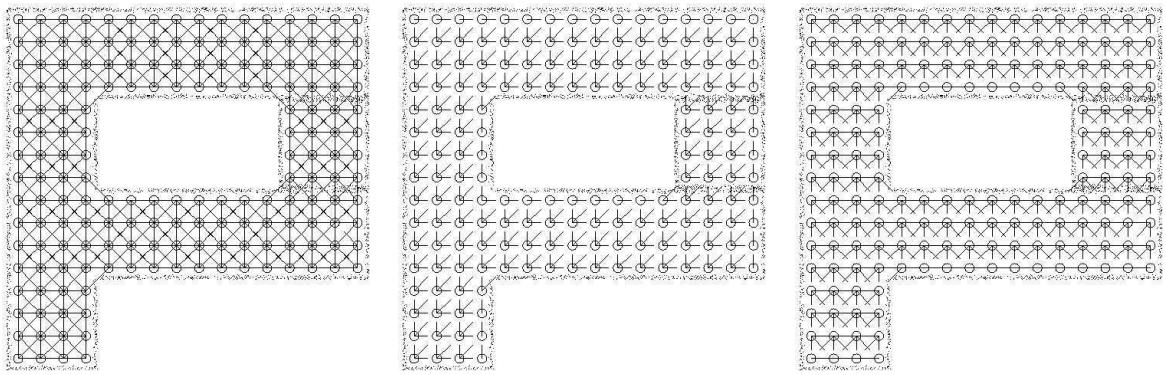

FIG. 2.5: **Connexions ayant un poids non nul dans le problème de navigation discrète.** De gauche à droite, nous avons représenté toutes les connexions, les connexions correspondant uniquement à l'action Nord-Est, et les connexions correspondant uniquement à l'action Sud. Après apprentissage de la matrice de transition, une connexion $c^a_{ss'}$ du réseau a un poids non nul si la transition $s \xrightarrow{a} s'$ est possible.

2.2.4 Lois d'évolution de l'activité des unités

Une fois que l'on dispose des estimations des fonctions T et R, la théorie des PDM nous permet de déduire la mise à jour des variables Q et V des unités afin de globalement calculer la fonction de valeur et la politique optimale. L'algorithme *Value Iteration* devient alors l'algorithme 2.3. Toute connexion du réseau $c^a_{ss'}$ a une entrée qui récupère la variable $s'.V$ et une sortie qui

Algorithme 2.3 Mise à jour des variables d'une unité s à l'instant t

pour tout $a \in A$ **faire**
  $Q[a] \leftarrow R[a] + \gamma . \sum_{s'} c_{ss'}.out$
fin pour
$V = \max_{a \in A} Q[a]$
$\pi = \arg\max_{a \in A} Q[a]$

peut être lue par s. Le flux d'information de la connexion $c^a_{ss'}$ est donc orienté de s' vers s. A tout instant, une telle connexion module l'information qu'elle reçoit par son poids. La sortie (notée $c^a_{ss'}.out$) de $c^a_{ss'}$ est calculée à partir de son entrée $c^a_{ss'}.in$:

$$c^a_{ss'}.out = c^a_{ss'}.in \times c^a_{ss'}.T = s'.V \times c^a_{ss'}.T \tag{2.6}$$

Cette loi linéaire de modulation se retrouve dans la plupart des réseaux connexionnistes de la littérature.

L'unité s met à jour chacune de ses *Q-valeurs* $Q[a]$ en sommant l'estimation de la récompense immédiate $R[a]$ et les valeurs des connexions rentrantes modulées par le coefficient γ. La fonction de valeur et la politique courante sont déduites comme précédemment. Bien que la formulation soit légèrement différente, le processus itératif que nous donnons ici est strictement équivalent à celui que nous avons décrit en détail à la section 1.2.4. Nous ne ferons donc pas plus de commentaires sur le calcul du plan.

2.2.5 Mécanisme de sélection de l'action

Nous avons déjà noté que la convergence vers la politique optimale pouvait être rapide. Nous savons également que la qualité des plans calculés augmente progressivement. Enfin, il est plus que raisonnable de supposer que le temps nécessaire pour exécuter une action du plan (interagir avec le monde) est largement supérieur à celui nécessaire pour faire une itération de planification (un cycle de fonctionnement des unités). Pour toutes ces raisons, un agent s'appuyant sur une telle architecture peut commencer à agir même si la planification n'a pas convergé vers la politique optimale courante. Afin d'exécuter le plan calculé par le réseau à un moment donné, l'action que l'agent doit effectuer est celle calculée par l'unique unité qui correspond à l'état courant : l'action à suivre est donc $s.\pi$ où s est l'unique unité telle que $s.E = Vrai$.

2.3 Une architecture massivement parallèle et tolérante aux pannes

Tous les éléments et calculs impliqués dans une approche indirecte de l'A/R, l'estimation des fonctions T et R et l'application de l'algorithme de résolution *Value Iteration*, s'intègrent naturellement dans l'architecture connexionniste que nous avons présentée. Nous allons à présent montrer que cette architecture "mérite" l'appellation connexionniste, c'est-à-dire qu'elle a les qualités couramment attendues des algorithmes connexionnistes : le parallélisme et la tolérance aux pannes.

2.3.1 Un parallélisme massif

Nous proposons de voir dans quelle mesure le réseau que nous proposons est parallélisable, c'est-à-dire pourrait s'implanter sur une machine parallèle pour les deux processus centraux : l'apprentissage et la planification/adaptation. Nous nous intéressons particulièrement à la question de la synchronisation.

Parallélisation des processus d'apprentissage

L'apprentissage de la matrice de transition et de la fonction récompense impliquent l'exploration du monde. A moins de supposer que l'agent a le don d'ubiquité, cette exploration est par nature complètement séquentielle. Ensuite, comme nous l'avons déjà fait au sujet de la planification, il est pragmatiquement raisonnable de supposer que le temps nécessaire pour interagir avec l'environnement est bien plus grand que celui requis pour effectivement apprendre (mettre à jour la connexion concernée par une transition consiste en 3 conditions et 2 opérations arithmétiques). En conséquence, il semble ici qu'il n'y a aucune raison de vouloir accélérer le processus d'apprentissage par parallélisation. Néanmoins, dans l'optique où l'on souhaiterait intégrer ce processus d'apprentissage dans une architecture complètement distribuée, la contrainte essentielle qu'il serait judicieux de supprimer est la nécessité de synchronisation (via une horloge globale à temps discret) qu'implique l'apprentissage des connexions : une connexion doit en effet comparer les excitations des unités qu'elle relie à deux instants consécutifs t et $t+1$. Il existe dans la littérature un certain nombre de propositions pour remédier à cette synchronisation discrète. Par exemple, on peut utiliser la notion de trace d'activité [Reiss et Taylor, 1991] : une unité qui a été active (et qui ne doit plus l'être) présente pendant un certain temps une activité résiduelle, ce qui permet aux connexions concernées pour l'apprentissage de "tranquillement" faire la corrélation temporelle. Des mises en œuvre d'un tel mécanisme peuvent être consultées dans [Frezza-Buet, 1999] et [Rougier, 2000].

Parallélisation du processus de planification

Sur une machine séquentielle le temps d'une itération de l'algorithme *Value Iteration* est en $O(|S|^2.|A|)$. Sur une machine parallèle en hypercube ayant $|S|^2.|A|$ processeurs, [Bertsekas et Tsitsiklis, 1989] montrent qu'une itération de l'algorithme *Value Iteration* peut passer en $O(\log(|A|.|S|))$. On peut même aller un peu plus loin. On peut montrer [Bertsekas, 1983] que toutes les équations du type point fixe comme celle qui sous-tend l'algorithme *Value Iteration* (équation 1.12 page 25) peuvent être résolues de manière complètement distribuée et asynchrone : les mises à jour des *Q-valeurs* et de V peuvent se faire dans n'importe quel ordre (sur s et a) ; la convergence vers la solution optimale est assurée à la simple condition que tous les couples (s, a) continuent de donner lieu à des mises à jour. Ainsi, il est possible d'implémenter l'algorithme *Value Iteration* sur des systèmes multi-processeurs ayant des délais de communication entre processeurs sans néanmoins utiliser d'horloge globale [Bertsekas, 1983]. Dans l'architecture que nous avons proposée, nous avons réduit le grain de calcul à l'échelle d'un état du PDM. On montre [Williams et Baird, III, 1990] qu'on peut construire des algorithmes qui sont assurés de converger vers le point fixe avec un grain encore plus fin : la somme $\sum_{s'} T(s, a, s').V(s')$ impliquée dans la mise à jour itérative peut elle-même être découpée en plusieurs étapes qui n'ont pas besoin d'être synchronisées. Le processus de planification, local, distribué et complètement asynchrone est donc un excellent candidat pour une parallélisation massive.

2.3.2 Tolérance aux pannes

Intéressons-nous enfin à la capacité de l'architecture proposée à continuer de donner des réponses relativement satisfaisantes si certains de ses constituants deviennent inopérants. Le cas où des connexions deviendraient inopérantes n'étant pas significativement différent[11], nous nous limitons au seul cas où des unités arrêtent de fonctionner.

Mort d'une unité et mécanisme d'identification des états

Avant d'aborder l'effet d'une panne sur les processus d'apprentissage et de planification, nous devons faire une hypothèse sur la façon dont le mécanisme d'identification de l'état courant (qui agit en entrée de notre architecture) réagit à la disparition d'une unité. En particulier, que se passe-t-il lorsque le système se trouve dans l'état correspondant à l'unité décédée ? Bien souvent [Kohonen, 1988] [Fritzke, 1997] [Rougier, 2000] (et ce d'autant plus qu'il s'agit d'une architecture distribuée et parallèle) on considère que la détermination de l'état courant est un processus de compétition entre les unités du système. A tout moment, chaque unité est candidate pour dire "je représente l'état courant". En pratique, on évalue pour chacune un score de représentativité de l'état réel du système et on choisit celle qui le représente le mieux (celle qui a le meilleur score). Ainsi, lorsqu'une unité vient à mourir, nous considèrerons qu'elle disparaît simplement de la compétition et que ce sont des unités qui lui étaient proches (au sens de la représentativité) qui se partagent la situation qu'elle codait.

Tolérance aux pannes du processus d'apprentissage

Lorsqu'une unité meurt, les probabilités de transition dans le réseau changent. Par exemple, la probabilité d'atteindre l'unité décédée à partir de n'importe quelle autre unité devient nulle. Ainsi, la mort d'une unité est équivalente, du point de vue des unités restantes, à la non-stationnarité de la fonction de transition. Nous avons montré (section 2.2.2 et figure 2.3 en particulier) qu'utiliser un taux d'apprentissage constant permettait de suivre une matrice de transition non-stationnaire. En conséquence, le processus d'apprentissage, s'il utilise un taux constant, corrige petit à petit les erreurs induites par la disparition d'une unité.

Tolérance aux pannes du processus dynamique de planification

Pour rapidement comprendre pourquoi le processus de planification est lui aussi tolérant aux pannes, nous donnons ici une explication imagée. Un PDM peut être vu comme une carte (dans le problème de navigation discrète, c'est exactement une carte) : si certaines unités meurent, cela est intuitivement équivalent au fait que la carte devient plus floue dans les zones où le décès a eu lieu. Après ré-apprentissage, la planification a lieu comme précédemment, à la différence que la carte sur laquelle elle se base est un petit peu moins précise.

La figure 2.6 illustre expérimentalement l'évolution des performances au fur et à mesure que l'on détruit des unités pour le problème que nous avons détaillé (la navigation discrète).

Nous avons procédé de manière itérative à :

- la suppression d'une unité ;
- le ré-apprentissage des paramètres (par exploration aléatoire) ;
- la planification ;
- des simulations de 200 trajectoires partant de points tirés uniformément pour évaluer les performances du réseau.

[11] La mort d'une unité est fonctionnellement identique à la mort de toutes ses connexions.

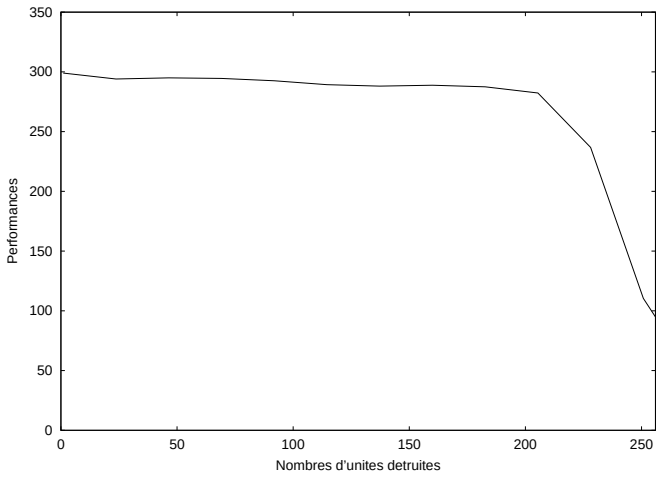

FIG. 2.6: **Illustration de tolérance aux pannes pour le problème de navigation discrète.** Cette figure montre l'évolution des performances (évaluées par simulation) au fur et à mesure que l'on supprime des unités. L'architecture que nous proposons est tolérante aux pannes dans la mesure où les performances se maintiennent longtemps.

On observe que les performances se maintiennent, même après un grand nombre d'unités détruites. Dans ce cas précis, la baisse tardive des performances peut être expliquée par le fait qu'avec très peu d'unités, l'agent est toujours capable de trouver un chemin vers la zone but.

Résumé du chapitre

A la fin du chapitre précédent, nous avons introduit le problème de l'A/R et nous avons montré qu'on pouvait lui apporter une réponse algorithmique selon deux approches :

- L'approche directe : on utilise des échantillons de R et T pour calculer directement la fonction de valeur optimale ;
- L'approche indirecte : on découpe le problème de l'A/R en deux. D'un côté, on fait l'estimation des paramètres R et T et de l'autre, on planifie à l'aide de ces paramètres estimés.

Dans ce chapitre, nous avons commencé par argumenter en faveur de l'approche indirecte, en particulier lorsque l'espace d'états est petit et lorsqu'on espère un transfert d'apprentissage entre plusieurs tâches.

Nous avons ensuite montré que tous les calculs impliqués par une approche indirecte d'un petit problème d'A/R s'intégrent naturellement dans une architecture *connexionniste*. Les *unités* de cette architecture sont les états du PDM. Les *connexions* sont le support de la fonction décrivant la dynamique du problème T. La récompense R est estimée de manière répartie par l'ensemble des unités. Enfin, la fonction de valeur optimale est une activité distribuée dans le réseau et l'algorithme *Value Iteration* est une loi de propagation de cette activité.

Nous avons insisté sur le caractère connexionniste de cette vision des PDM en argumentant qu'une telle architecture était intrinsèquement parallèle et tolérante aux pannes.

Conclusion

Cette première partie nous a permis de poser les bases de la théorie computationnelle de l'A/R. Nous avons décrit son formalisme central, les PDM, et nous avons considéré une réponse algorithmique particulière : *Value Iteration*. L'utilisation conjointe de l'approche indirecte et de l'algorithme *Value Iteration* nous ont permis de construire une architecture connexionniste qui répond aux problèmes de l'A/R ayant un petit espace d'états.

Au final, nous avons une architecture qui présente les qualités que nous annoncions au début du mémoire. Elle est :

- **autonome** : elle peut faire une estimation des paramètres T et R caractérisant son interaction avec l'environnement ; elle calcule un plan/comportement optimal à l'aide de l'algorithme *Value Iteration* ;
- **robuste** : les récompenses, les transitions sont des informations de nature stochastique ;
- **anytime** : plus il y a de temps, plus les estimations de R et T sont bonnes ; plus il y a de temps et plus le calcul de la fonction de valeur optimale effectué par *Value Iteration* est précis.
- **multi-tâche** et **dynamique** : lorsque les fonctions T et R (définissant le problème à résoudre) évoluent, une perpétuelle exploration permet d'adapter leur estimation ; ces changements de paramètres sont dynamiquement pris en compte par le calcul de planification effectué par *Value Iteration* ;
- **massivement parallèle** : il s'agit d'un réseau d'unités relativement simples fonctionnant en parallèle ; l'apprentissage de T et R se fait à l'aide de règles locales ; les calculs impliqués par la planification sont complètement distribués et ne requièrent pas de synchronisation explicite ;
- **tolérant aux pannes** : on peut supprimer des unités sans que cela affecte de manière immédiatement dramatique les performances ; en particulier la ré-estimation des paramètres T et R après la mort d'une unité permet dans une certaine mesure de compenser les baisses de performances.

D'une manière générale, il est important de souligner que le fonctionnement efficace de l'architecture que nous avons présentée est dépendant de la taille du problème considéré : plus l'espace d'états est grand et plus l'estimation précise des fonctions R et T sera longue ; de même plus l'espace d'états est grand et plus les caractéristiques dynamique et multi-tâche risquent de s'estomper. La suite de ce mémoire va proposer des solutions concernant ces limitations : nous allons proposer des extensions de cette architecture dans le cas où l'espace d'états est grand.

Deuxième partie

Apprentissage d'une représentation interne

Introduction

> « L'homme n'a jamais pu se passer de grilles. Devant le désordre apparent du monde, il lui fallut chercher les termes signifiants, ceux qui, associés entre eux, rendaient son action sur le milieu plus efficace, lui permettaient de survivre. Devant l'abondance infinie des objets et des êtres, il a recherché entre eux des relations, et devant l'infinie mobilité des choses, il a cherché des invariances. »
>
> *La nouvelle grille, Henri Laborit*

Des problèmes complexes tels que ceux de la vie réelle peuvent être décrits dans le formalisme de l'A/R. Dans l'introduction de ce mémoire, nous donnions quelques pistes pour comprendre comment l'A/R pourrait permettre de modéliser un joueur d'échec qui progresse par l'expérience, ou une gazelle qui apprend à marcher. La différence entre ces problèmes véritablement motivants et ceux que nous avons considérés dans la première partie est simple : les problèmes difficiles ont un grand espace d'états : la complexité de résolution croît avec le nombre d'états. L'un des efforts actuels de recherche en A/R consiste à aborder des problèmes de plus en plus difficiles et donc en particulier des problèmes qui ont des espaces d'états très grands.

Le travail présenté dans ce chapitre part du désir de contribuer à cet effort. Nous y présentons une démarche générale pour aborder des problèmes d'A/R significativement plus difficiles que ceux que nous avons décrits dans la partie précédente. Par rapport aux nombreuses approches de la littérature actuelle, il est important de souligner ce que la nôtre a de particulier. Pour cela nous la mettons en perspective de la première partie de ce mémoire. Lorsque l'espace d'états est petit, nous avons vu que nous pouvions donner une réponse au problème de l'A/R qui a de nombreuses qualités, en particulier les qualités courantes du connexionnisme (parallélisation et tolérance aux pannes). Dans ce chapitre, nous souhaitons résoudre des problèmes d'A/R ayant un grand espace d'états *tout en gardant* les qualités que nous avions pour les petits espaces d'états.

Voici la démarche que nous proposons : si l'espace d'états est trop grand pour appliquer l'approche que nous avons décrite jusqu'ici, l'agent apprenant par renforcement pourrait utiliser une représentation simplifiée/approchée du problème. Mieux encore, il pourrait la construire de manière autonome, c'est-à-dire inférer un petit ensemble représentatif d'états, qui résument adéquatement l'ensemble des vrais états. Le problème que nous abordons précisément dans cette

partie est celui de la construction autonome d'une telle représentation. Si cette représentation est assez compacte, tout ce que nous avons décrit dans la partie précédente pourra être appliqué.

Précisons ce que nous voulons dire par "construire une représentation simplifiée et compacte d'un problème". Il est important de distinguer deux sortes d'états : les vrais états du problème à résoudre (qui sont très nombreux) et les états une fois le problème simplifié (qu'on veut en petit nombre). Pour une raison qui sera rapidement claire, nous appelons les états du problème simplifié des "macro-états". Ce sont eux qui définissent ce que nous appelons la "représentation simplifiée du problème". L'ensemble des macro-états est une partition de l'ensemble des états : tout macro-état rassemble un ou plusieurs vrais états. Cette approche est également connue sous le nom d'agrégation d'états : on dit alors que tous les vrais états d'un macro-état sont agrégés.

L'idée de transformer un problème complexe de type PDM en agrégeant les états de sorte à retomber sur une formalisation finie et de petite taille n'est pas nouvelle. C'est une approximation typique pour aborder des problèmes d'A/R ayant un grand espace d'états. C'est par exemple une des approches qu'a suivie [Laroche, 2000] pour la robotique mobile ; dans son cas néanmoins, l'agrégation est faite à la main et les principes utilisés sont propres au problème traité : la navigation robotique. Il existe des propositions de calcul automatique d'une agrégation pour des PDM pas ou peu spécifiques à une application donnée. Parmi celles-ci, on dénote plusieurs approches empiriques [McCallum, 1995] [Dutech, 1999] [Finton, 2002]. Les auteurs de [Munos et Moore, 2000] ont récemment proposé une approche formelle qui pose ce problème dans le cadre général de l'approximation de PDM : trouver une agrégation se formalise comme un problème de minimisation d'une erreur d'approximation. Le caractère formel de cette approche permet à notre avis de bien cerner la puissance et les limitations de l'approche. En particulier, cela assure que nos développements sont génériques, c'est-à-dire qu'ils peuvent s'appliquer à n'importe quel problème d'A/R. Pour toutes ces raisons, c'est cette approche formelle que nous considérons et que nous détaillons dans la suite.

Cette partie est organisée en deux chapitres. Dans le premier chapitre, nous reprenons la méthodologie introduite par [Munos et Moore, 2000] pour étudier l'erreur induite par une approximation. Nous l'adaptons au cas de l'approximation par agrégation. Dans le deuxième chapitre, nous donnons un exemple de mise en œuvre de cette analyse et décrivons pratiquement une technique d'amélioration de l'agrégation. Nous l'appliquons à plusieurs problèmes de contrôle dont l'espace d'états est infini.

Chapitre 3

Fondements théoriques de l'approximation d'un PDM par agrégation

Lorsqu'on utilise une approximation pour résoudre un problème d'A/R, cela peut induire des erreurs sur le calcul de la fonction de valeur optimale. Il est important de maîtriser ces erreurs pour que la politique correspondante soit néanmoins de bonne qualité. Des travaux récents [Munos et Moore, 2000] proposent une méthodologie générale pour analyser et réduire efficacement cette erreur. Leur démarche comporte trois étapes que nous reprenons et adaptons au cas d'une approximation par agrégation d'états.

Cette démarche va successivement nous amener à étudier les trois questions suivantes :

- Comment estimer l'erreur commise sur la fonction de valeur optimale lorsqu'on utilise un modèle agrégé ?
- L'erreur commise dans certaines zones de l'espace d'états induit une erreur dans d'autres zones ; comment mesurer ces dépendances ?
- Comment faire évoluer une agrégation de sorte à diminuer l'erreur d'approximation ?

3.1 La mesure d'une approximation

Nous commençons par présenter la méthodologie générale introduite dans [Munos et Moore, 2000] afin de dériver une borne supérieure de l'erreur commise sur le calcul de la fonction de valeur si on utilise un modèle approché plutôt que le vrai modèle. Nous l'appliquons ensuite au cas particulier qui nous intéresse, c'est-à-dire celui où l'approximation est une agrégation d'états. Nous illustrons ces développements théoriques à l'aide de deux problèmes de contrôle.

3.1.1 Cas général

Nous introduisons quelques notations importantes puis nous présentons les principaux résultats de [Munos et Moore, 2000].

Notations

Considérons un PDM $\mathcal{M} = \langle S, A, T, R \rangle$ que nous appelons le vrai modèle. Considérons un deuxième PDM $\widehat{\mathcal{M}} = \langle S, A, \widehat{T}, \widehat{R} \rangle$ qui approxime[12] le premier PDM $\mathcal{M}$. Nous avons vu dans la partie précédente que les fonctions de valeurs optimales étaient les points fixes d'un opérateur qu'on peut appeler "opérateur de Bellman". Soit B^* l'opérateur de Bellman dans le *vrai* modèle. Pour toute fonction $W : S \to \mathbb{R}$, $B^*.W$ est une nouvelle fonction de $S \to \mathbb{R}$ qui est telle que :

$$[B^*.W](s) = \max_a \left(R(s,a) + \gamma . \sum_{s'} T(s,a,s').W(s') \right)$$

De façon analogue, soit $\widehat{B}^*$ l'opérateur de Bellman dans le modèle approché :

$$[\widehat{B}^*.W](s) = \max_a \left(\widehat{R}(s,a) + \gamma . \sum_{s'} \widehat{T}(s,a,s').W(s') \right)$$

Notons respectivement V^{π^*} et $\widehat{V}^{\widehat{\pi}^*}$ les fonctions de valeur optimale du vrai modèle $\mathcal{M}$ et du modèle approché $\widehat{\mathcal{M}}$. Par définition, ces fonctions sont les uniques points fixes des opérateurs que nous venons de définir : $B^*.V^{\pi^*} = V^{\pi^*}$ et $\widehat{B}^*.\widehat{V}^{\widehat{\pi}^*} = \widehat{V}^{\widehat{\pi}^*}$. Dans l'idéal, la vraie fonction de valeur V^{π^*} est ce que nous voudrions calculer. La fonction de valeur approchée $\widehat{V}^{\widehat{\pi}^*}$ est souvent celle qu'on a effectivement calculée. L'erreur induite en un point par l'utilisation du modèle approché peut être définie précisément et naturellement :

Définition 2 (Erreur d'approximation locale)
L'erreur d'approximation locale est la différence entre la fonction de valeur optimale approchée et la vraie fonction de valeur optimale :

$$E_{app}(s) = |V^{\pi^*}(s) - \widehat{V}^{\widehat{\pi}^*}(s)|$$

Bien souvent, on s'intéresse surtout à l'erreur d'approximation commise dans certaines zones de l'espace, par exemple celles correspondant à la distribution des états du système à $t = 0$:

Définition 3 (Erreur d'approximation initiale)
L'erreur d'approximation initiale est la somme des erreurs d'approximation locale pondérées par la distribution des états du système à $t = 0$:

$$E^{(0)}_{app} = \sum_{s \in S} E_{app}(s).p(s_0 = s) \tag{3.3}$$

Il apparaît (nous allons le voir dans la suite) que l'erreur d'approximation locale est intrinsèquement liée à une autre quantité qu'il est naturel d'appeler erreur d'interpolation locale :

Définition 4 (Erreur d'interpolation locale)
L'erreur d'interpolation locale est l'erreur induite si on utilise l'opérateur de Bellman approché $\widehat{B}^*$ au lieu du vrai opérateur de Bellman B^* sur la vraie fonction de valeur V^{π^*} :

$$E_{int}(s) = |\widehat{B}^*.V^{\pi^*}(s) - B^*.V^{\pi^*}(s)| \tag{3.4}$$

[12] Dans toute cette partie, nous utilisons la notation "chapeau" $\widehat{.}$ pour désigner les objets qui sont des approximations.

Lorsqu'on applique le *vrai* opérateur de Bellman B^* sur la *vraie* fonction de valeur optimale V^{π^*}, on obtient (par définition) de nouveau la *vraie* fonction de valeur optimale V^{π^*}. Si par contre on lui applique l'opérateur de Bellman approché $\widehat{B}^*$, on ne retombe pas forcément sur la *vraie* fonction de valeur optimale V^{π^*}. L'erreur d'interpolation a pour fonction de mesurer cette différence.

Propriétés

Nous allons exprimer les relations générales qui existent entre erreur d'interpolation et erreurs d'approximation. Comme les résultats que nous présentons ci-après ne sont pas originaux (ils proviennent de [Munos et Moore, 2000]), nous ne détaillons pas les démonstrations. Pour information, celles-ci sont néanmoins reproduites dans l'annexe A.

Si $\overline{E_{int}}(s)$ est une borne supérieure de $E_{int}(s)$, alors une borne supérieure $\overline{E_{app}}(s)$ de $E_{app}(s)$ est solution de l'équation de Bellman suivante :

$$\overline{E_{app}}(s) = \max_a \left(\gamma . \sum_{s'} \widehat{T}(s,a,s') . \overline{E_{app}}(s') \right) + \overline{E_{int}}(s) \tag{3.5}$$

Cette équation, qui caractérise une borne supérieure de l'erreur d'approximation à partir d'une borne de l'erreur d'interpolation, est strictement identique à celle qui caractérise la valeur optimale d'un PDM à partir de la fonction de récompense. Au lieu de propager la récompense à travers l'espace d'états, c'est ici l'erreur d'interpolation qui est propagée selon la dynamique approchée définie par $\widehat{T}$.

En pratique, on commence par évaluer une borne supérieure de l'erreur d'interpolation puis on utilise un algorithme identique à *Value Iteration* pour déduire une borne supérieure de l'erreur d'approximation. On peut alors immédiatement déduire une borne supérieure $\overline{E_{app}^{(0)}}$ de l'erreur d'approximation initiale :

$$\overline{E_{app}^{(0)}} = \sum_{s \in S} \overline{E_{app}}(s) . p(s_0 = s)$$

Nous aurons l'occasion d'illustrer ce procédé dans la suite de ce chapitre.

Le lien entre erreur d'interpolation et erreur d'approximation dévoilé par l'équation 3.5 ne dépend pas du type d'approximation qu'on utilise. En conséquence, les résultats que nous venons de présenter ont un corollaire général très intéressant : quelle que soit l'approximation utilisée, si l'erreur d'interpolation tend vers 0, alors l'erreur d'approximation tendra également vers 0. Autrement dit, quel que soit le moyen, si les paramètres approchés $\widehat{R}$ et $\widehat{T}$ tendent respectivement vers les vrais paramètres R et T, alors la fonction de valeur approchée tend vers la vraie fonction de valeur.

3.1.2 Application au cas de l'approximation par agrégation

Nous adaptons à présent ce que nous venons de décrire au cas particulier de l'approximation par agrégation. Nous commençons par introduire un certain nombre de notations et nous formalisons précisément ce qui se passe lorsqu'on effectue une agrégation. Nous dérivons le calcul d'une borne de l'erreur d'interpolation. Nous montrons enfin qu'on peut en déduire l'erreur d'approximation en résolvant une équation de type Bellman sur l'espace agrégé.

Notations

Soit un PDM $\mathcal{M} = \langle S, A, T, R \rangle$. Approximer ce PDM par agrégation signifie qu'on va définir un deuxième PDM dont l'espace d'états est une partition $\widehat{S}$ du vrai espace d'états S : $\widehat{\mathcal{M}} = \langle \widehat{S}, A, \widehat{T}, \widehat{R} \rangle$. Les éléments de $\widehat{S}$, qui sont des parties de S, sont appelés "états agrégés" ou "macro-états". Tout vrai état $s \in S$ appartient à un et un seul macro-état ; notons $\widehat{s} \in \widehat{S}$ le macro-état qui contient l'état $s \in S$.

L'agrégation des états implique que les paramètres $\widehat{T}$ et $\widehat{R}$ sont définis sur l'ensemble des macro-états $\widehat{S}$. Autrement dit, on a $\widehat{R} : \widehat{S} \times A \to \mathbb{R}$ et $\widehat{T} : \widehat{S} \times A \times \widehat{S} \to [0;1]$. Comme $\widehat{S}$ est une partition de S, il existe une manière naturelle de "plonger" ces paramètres[13] dans S :

$$\forall s_1 \in S, \forall a \in A, \widehat{R}(s_1, a) = \widehat{R}(\widehat{s_1}, a)$$

et

$$\forall (s_1, s_2) \in S^2, \forall a \in A, \widehat{T}(s_1, a, s_2) = \widehat{T}(\widehat{s_1}, a, \widehat{s_2})$$

Après "plongement" dans S, ces fonctions définies a priori dans $\widehat{S}$ sont des fonctions de S constantes sur tous les macro-états.

Calculs des erreurs

Nous allons dériver l'erreur d'interpolation induite par une agrégation et montrer en particulier qu'une fois une partition $\widehat{S}$ choisie, il y a une manière naturelle de choisir $\widehat{R}$ et $\widehat{T}$.

En utilisant une propriété connue de l'opérateur max[14], l'inégalité triangulaire, puis le fait que $|V^{\pi^*}(s)| \leq \frac{\max_{s,a} |R(s,a)|}{1-\gamma} = \frac{R_{max}}{1-\gamma}$, on peut développer une borne de l'erreur d'interpolation en un état s_1 :

$$\begin{aligned}
E_{int}(s_1) &= \left| \max_a \left(R(s_1, a) + \gamma . \sum_{s_2 \in S} T(s_1, a, s_2) . V^{\pi^*}(s_2) \right) \right. \\
&\quad \left. - \max_a \left(\widehat{R}(\widehat{s_1}, a) + \gamma . \sum_{s_2 \in S} \widehat{T}(\widehat{s_1}, a, \widehat{s_2}) . V^{\pi^*}(s_2) \right) \right| \\
&\leq \max_a \left| R(s_1, a) - \widehat{R}(\widehat{s_1}, a) + \gamma . \sum_{s_2 \in S} \left(T(s_1, a, s_2) - \widehat{T}(\widehat{s_1}, a, \widehat{s_2}) \right) . V^{\pi^*}(s_2) \right| \\
&\leq \max_a \left| R(s_1, a) - \widehat{R}(\widehat{s_1}, a) \right| + \gamma . \max_a \left(\sum_{s_2 \in S} \left| T(s_1, a, s_2) - \widehat{T}(\widehat{s_1}, a, \widehat{s_2}) \right| . \left| V^{\pi^*}(s_2) \right| \right) \\
&\leq \max_a \left| R(s_1, a) - \widehat{R}(\widehat{s_1}, a) \right| + \frac{\gamma . R_{max}}{1-\gamma} . \max_a \left(\sum_{s_2 \in S} \left| T(s_1, a, s_2) - \widehat{T}(\widehat{s_1}, a, \widehat{s_2}) \right| \right)
\end{aligned}$$

Alors on peut en déduire une borne supérieure de l'erreur d'interpolation $\overline{E_{int}}(\widehat{s_1})$ pour l'ensemble des états appartenant au macro-état $\widehat{s_1}$. On a en effet

$$\forall s_1 \in S, E_{int}(s_1) \leq \overline{E_{int}}(\widehat{s_1})$$

[13] Par commodité, nous utilisons la même notation pour la fonction définie sur $\widehat{S}$ et sa version après plongement dans S.

[14] Voir sa démonstration en annexe A

avec

$$\overline{E_{int}}(\widehat{s_1}) = \max_{s \in \widehat{s_1}} \left[\max_a \left| R(s,a) - \widehat{R}(\widehat{s_1},a) \right| + \frac{\gamma.R_{max}}{1-\gamma}. \max_a \left(\sum_{\widehat{s_2} \in \widehat{S}} \sum_{s' \in \widehat{s_2}} \left| T(s,a,s') - \widehat{T}(\widehat{s_1},a,\widehat{s_2}) \right| \right) \right]$$

L'erreur d'approximation est déduite par propagation de l'erreur d'interpolation. Plus l'erreur d'interpolation sera petite et plus l'erreur d'approximation le sera aussi. Des choix naturels pour $\widehat{R}$ et $\widehat{T}$ sont les suivants[15] :

$$\widehat{R}(\widehat{s},a) = \frac{1}{|\widehat{s}|}. \sum_{s \in \widehat{s}} R(s,a) \tag{3.6}$$

$$\widehat{T}(\widehat{s_1},a,\widehat{s_2}) = \frac{1}{|\widehat{s_1}|}. \sum_{(s,s') \in \widehat{s_1} \times \widehat{s_2}} T(s,a,s') \tag{3.7}$$

Une fois que ce choix est fait, on peut préciser la valeur de la borne supérieure de l'erreur d'interpolation :

$$\forall s_1 \in S, E_{int}(s_1) \leq \overline{E_{int}}(\widehat{s_1}) = \overline{\Delta R}(\widehat{s_1}) + \frac{\gamma.R_{max}}{1-\gamma}. \sum_{\widehat{s_2} \in \widehat{S}} \overline{\Delta T}(\widehat{s_1},\widehat{s_2}) \tag{3.8}$$

avec les notations suivantes :

$$\begin{cases} \overline{\Delta R}(\widehat{s}) = \frac{1}{|\widehat{s}|}. \max_{(s,s',a) \in \widehat{s} \times \widehat{s} \times A} |R(s,a) - R(s',a)| \\ \overline{\Delta T}(\widehat{s_1},\widehat{s_2}) = \frac{1}{|\widehat{s_1}|}. \max_{a \in A} \max_{(s_1,s'_1,a) \in \widehat{s_1} \times \widehat{s_1} \times A} |\sum_{s_2 \in \widehat{s_2}} T(s_1,a,s_2) - T(s'_1,a,s_2)| \end{cases}$$

Cette borne supérieure de l'erreur d'interpolatioin et la fonction de transition $\widehat{T}$ sont constantes sur chaque macro-état. L'équation 3.5 qui permet de caractériser une borne supérieure de l'erreur d'approximation est donc équivalente à l'équation ci-dessous (définie uniquement sur les macro-états) :

$$\overline{E_{app}}(\widehat{s_1}) = \max_a \left(\gamma. \sum_{\widehat{s_2}} \widehat{T}(\widehat{s_1},a,\widehat{s_2}).\overline{E_{app}}(\widehat{s_1}) \right) + \overline{E_{int}}(\widehat{s_1}) \tag{3.9}$$

L'équation 3.5 est définie sur le *vrai* espace d'états S. La borne supérieure de l'erreur d'approximation est dans le cas général aussi difficile à calculer que la vraie fonction de valeur optimale. Dans le cas agrégation, l'équation 3.9 montre que l'on peut caractériser une borne supérieure de l'erreur d'approximation par une équation définie sur l'espace agrégé $\widehat{S}$. Cette dernière est donc beaucoup plus facile à calculer. Plus précisément, cette borne est aussi facile à calculer que la fonction de valeur optimale *approchée*. Un algorithme comme *Value Iteration* permet de calculer cette borne supérieure de l'erreur d'approximation sur l'espace des macro-états. Par plongement de $\widehat{S}$ dans S, on peut facilement en déduire une borne de l'erreur d'approximation sur le *vrai* espace d'états.

Remarque concernant l'approche A/R

Dans le cadre de l'approche A/R, l'agent doit estimer les fonctions de transition et de récompense en explorant l'environnement. Lorsqu'il utilise un modèle agrégé, nous voyons ici que le

[15] Nous voulons avoir une erreur d'interpolation aussi petite que possible et nous devons vérifier des contraintes comme $\sum_{\widehat{s_2}} T(\widehat{s_1},a,\widehat{s_2}) = 1$.

calcul de ses paramètres $\widehat{T}$ et $\widehat{R}$ et de l'estimation de l'erreur d'interpolation E_{int} nécessitent de connaître des informations sur les variations des vraies fonctions R et T dans chacun des macro-états[16]. Ceci étant supposé, les quelques résultats que nous venons de présenter lui permettront de déduire l'erreur E_{app} qu'il commet sur la fonction de valeur optimale, c'est-à-dire sur l'objet essentiel qu'il cherche à calculer.

3.1.3 Illustration sur deux problèmes

Nous illustrons les notions d'erreur d'interpolation et d'erreur d'approximation que nous venons d'introduire sur deux exemples de PDM infinis qu'on approxime par agrégation régulière. Le premier, que nous appellerons "navigation continue" modélise un agent de type robot devant trouver son chemin dans un environnement. Le deuxième, "La voiture sur la colline"[17] est un problème de contrôle fréquemment rencontré dans la littérature A/R.

Exemple 1 : Navigation continue

Le problème "navigation continue" décrit une tâche de navigation analogue à celle que nous avons détaillée dans la partie précédente : un agent de type robot peut se déplacer dans un environnement qui comporte deux pièces et deux couloirs. Le mouvement de cet agent est légèrement bruité. Son objectif est d'atteindre une zone but (voir figure 3.1) .

Ce problème s'inscrit dans un espace continu à deux dimensions. En effet, il est décrit par un PDM dont l'espace d'états S est l'ensemble des positions dans l'environnement $s = (x, y)$ avec $0 \leq x \leq 10$ et $0 \leq y \leq 10$. Les actions sont des mouvements d'amplitude 0.1 dans 8 directions cardinales plus l'action "ne pas bouger". Soit $\overrightarrow{u_a}$ un vecteur déplacement correspondant à l'action a. Les fonctions R et T de ce PDM d'espace d'états infini sont implicitement définies par les règles suivantes :

- $s_{t+1} \leftarrow s_t + \overrightarrow{u_{a_t}} + \overrightarrow{b_t}$ où a_t est l'action effectuée au temps t et $\overrightarrow{b_t}$ est un vecteur bruit de direction cardinale aléatoire et d'amplitude 0.03.
- Si l'état suivant s_{t+1} se situe dans un mur, alors on revient à la position précédente et on donne une récompense négative : $s_{t+1} \leftarrow s_t$ et $r_{t+1} = -1$
- Si s_{t+1} appartient à la zone but, alors on donne une récompense positive : $r_{t+1} = 1$

Nous approximons ce PDM infini en faisant une agrégation par découpage régulier en 16×16 carrés de même taille (voir figure 3.2). Afin d'évaluer les fonctions $\widehat{R}$ et $\widehat{T}$ pour cette approximation, nous simulons les lois que nous avons décrites un peu plus haut sur de nombreuses trajectoires : ici nous nous arrangeons pour que chaque action soit essayée dans chaque macro-état au moins 25 fois. Nous en profitons pour estimer les variations $\overline{\Delta R}$ et $\overline{\Delta T}$ des vraies fonctions récompense et transition.

Nous déduisons de $\overline{\Delta R}$ et $\overline{\Delta T}$ l'erreur d'interpolation en tout macro-état. En utilisant un algorithme identique à *Value Iteration*, nous dérivons l'erreur d'approximation. La figure 3.3 représente graphiquement les résultats de ces calculs.

Ce qui nous intéresse dans cette illustration n'est pas tant l'ampleur des erreurs mais la différence de répartition entre l'erreur d'interpolation et l'erreur d'approximation, c'est-à-dire la différence avant et après la propagation de l'erreur. Avant la propagation, l'erreur se concentre près des zones de discontinuité du problème : les murs et les coins de l'environnement de navigation. Après propagation, l'erreur est répartie de manière à peu près homogène. Si le modèle

[16] Cette connaissance peut être liée à des hypothèses de régularité (i.e. R et T sont lipschhietziennes) et/ou une estimation.

[17] *Car on the hill* en anglais.

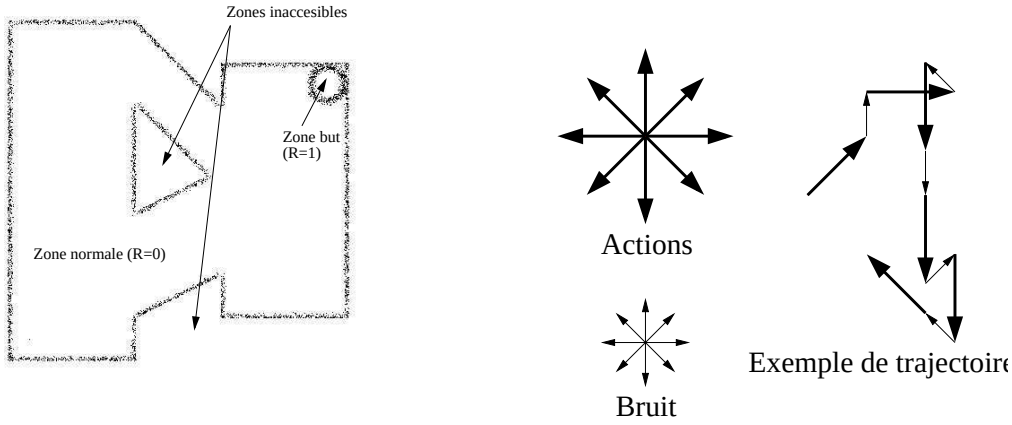

FIG. 3.1: **Une tâche de navigation continue.** Un agent de type robot doit se frayer un chemin dans un environnement contenant deux pièces et deux couloirs afin de rallier une zone but. Chacun de ses déplacements est bruité : à chaque mouvement se rajoute un bruit aléatoire. Par exemple, nous avons reproduit une trajectoire possible lorsque l'agent effectue successivement les actions Nord-Est, Est, Sud, Sud, Sud, Nord-Ouest.

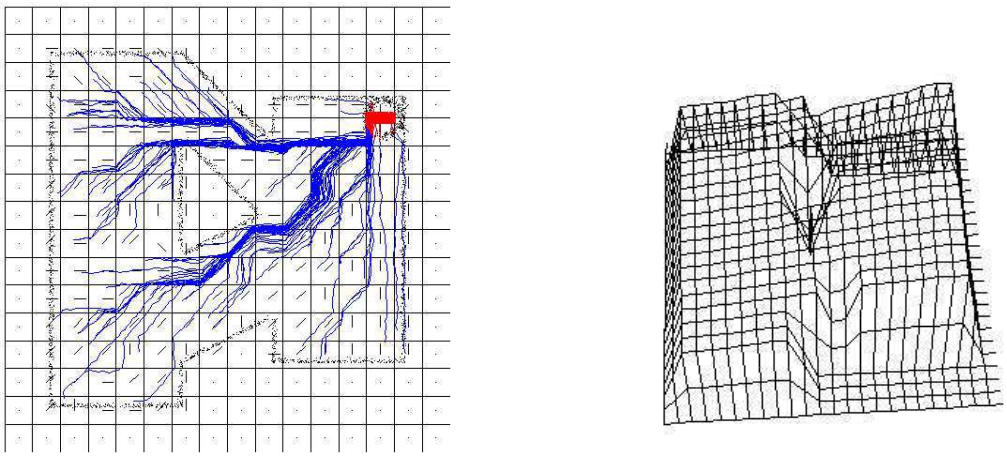

FIG. 3.2: **Approximation par agrégation du problème "Navigation continue".**. Une approximation par agrégation permet de calculer une fonction de valeur optimale approximative (à gauche). A partir de celle-ci, nous pouvons en déduire un plan et observer des exemples de trajectoires que l'agent emprunterait en suivant ce plan (à droite).

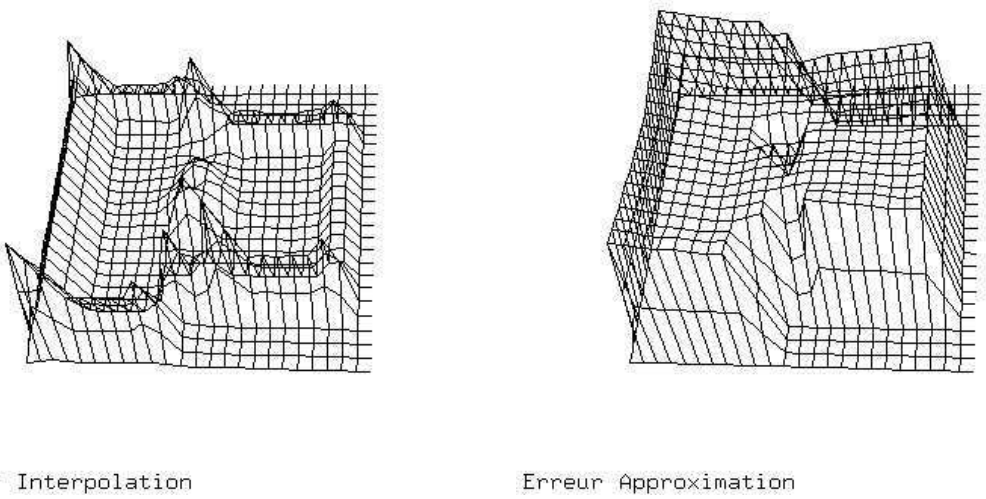

FIG. 3.3: **Bornes supérieures de l'erreur d'interpolation et de l'erreur d'approximation pour le problème "navigation continue".** L'erreur d'interpolation est élevée près des murs : c'est essentiellement à ces endroits que l'approximation sous forme d'agrégation fait de grosses erreurs sur R et T. L'erreur d'approximation sur la fonction de valeur est l'erreur d'interpolation une fois propagée : une erreur d'interpolation répartie dans peu d'endroits peut induire une erreur d'approximation relativement élevée partout.

agrégé implique une erreur sur le calcul de la fonction de valeur en certains points près des murs, on voit que celle-ci peut se répercuter sur les zones éloignées des murs.

Exemple 2 : La voiture sur la colline

Considérons une deuxième tâche qui consiste à amener un véhicule relativement peu puissant en haut d'une colline comme suggéré figure 3.4. C'est un problème typique de la littérature (on le retrouve par exemple dans [Sutton et Barto, 1998], [Munos, 1997a], [Baxter *et al.*, 1999]). La difficulté principale de ce problème est que la gravité est plus forte que le moteur de la voiture : même si la voiture est à pleine puissance, elle ne peut monter directement la pente. La seule solution consiste à osciller dans la cuvette (et donc à s'éloigner du but), de sorte à gagner assez d'énergie cinétique pour atteindre le but. Cette énergie doit cependant rester raisonnable : la voiture ne doit pas trop accélérer sinon elle tombe dans le ravin.

Ce problème est décrit à l'aide d'un PDM de taille infinie. L'espace d'états est l'ensemble des couples (x, v) correspondant respectivement à la position et la vitesse de la voiture. Ces deux variables sont bornées : $-1.2 \leq x \leq 0.6$ et $-0.07 \leq v \leq 0.07$. On considère que la voiture ne peut effectuer que deux actions, accélérer vers la droite ($a = +1$) ou vers la gauche ($a = -1$). La dynamique de ce système peut alors être décrite par les équations de la physique simplifiées suivantes [Sutton et Barto, 1998] :

$$\begin{cases} x_{t+1} &= x_t + v_t \\ v_{t+1} &= v_t + 0.001.a_t - 0.0025.cos(3.x_t) \end{cases} \tag{3.10}$$

La première équation est une version intégrée de la relation entre la position et la vitesse pour $dt = 1s$; la deuxième correspond à l'équation fondamentale de la dynamique (voir [Sutton et Barto,

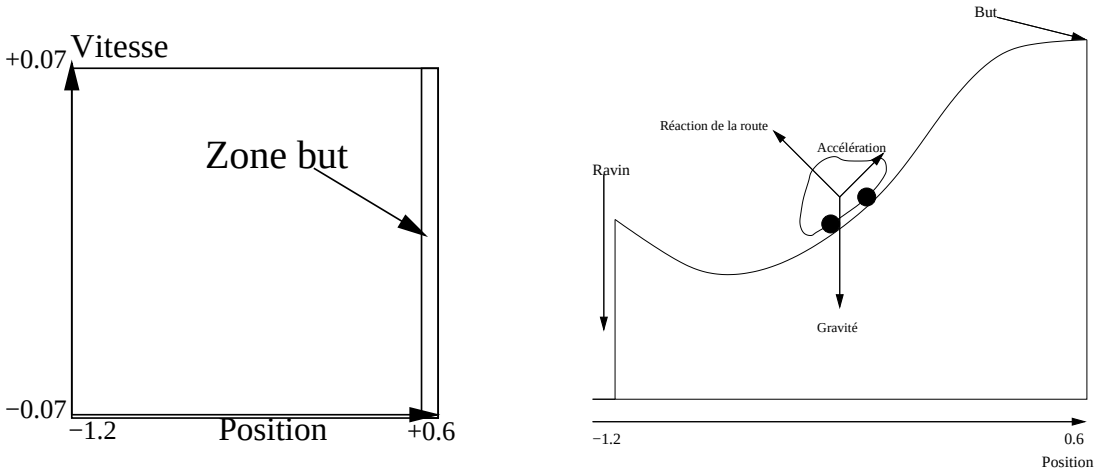

FIG. 3.4: **La voiture sur la colline.** Une voiture peu puissante doit atteindre le haut d'une colline. Pour ce faire, elle doit osciller dans le bassin et utiliser la gravité afin de gagner assez d'énergie cinétique pour gravir la colline. Elle doit faire attention à ne pas prendre trop de vitesse sinon elle risque de tomber dans le ravin. La dynamique de ce problème peut être décrite dans l'espace des phases (position,vitesse). L'objet de ce problème de contrôle est de trouver un chemin jusqu'à la zone but dans cet espace des phases en contrôlant l'accélération du véhicule.

1998] pour plus de détails). Nous ajoutons une contrainte supplémentaire à cette dynamique : lorsque la voiture est tombée dans le ravin (si x_t se retrouve dans $[-1.2, -1.15]$), elle ne peut plus en sortir (x_{t+1} y est aussi). Pour motiver la voiture à gravir la colline et à s'arrêter en haut, on donne une récompense positive et linéairement décroissante selon la vitesse lorsque la position est dans l'intervalle $[0.55, 0.6]$: la récompense vaut en effet $0.07 - |v|$. Pour dissuader la voiture de tomber dans le ravin, on donne une récompense négative (-1) pour tous les $x \in [-1.2, -1.15]$. Pour tous les autres états, on donne une récompense nulle.

Comme précédemment, nous approximons ce problème à l'aide d'un modèle agrégé $\widehat{\mathcal{M}}$. L'espace des macro-états $\widehat{S}$ contient 32×32 états qui découpent régulièrement l'espace S continu. Nous estimons ici encore les paramètres $\widehat{R}$ et $\widehat{T}$ à l'aide de nombreux échantillons de trajectoires.

Nous pouvons en déduire l'erreur d'interpolation ainsi que l'erreur d'approximation. Nous présentons ces erreurs à la figure 3.6. Dans ce deuxième exemple, il est intéressant de faire l'observation suivante : contrairement au problème de "navigation continue" où l'erreur d'interpolation se propageait de manière isotrope, on voit ici que la façon avec laquelle l'erreur se propage dépend intrinsèquement du problème considéré. Ici, l'erreur d'interpolation se propage selon un mouvement en spirale qui correspond aux lois de la dynamique du problème (équation 3.10).

3.2 Dépendances non locales : la notion d'influence

Nous venons de voir un moyen de calculer l'erreur d'approximation en chaque état d'un PDM approximatif. Nous avons vu en particulier que l'erreur d'approximation est le résultat d'une "propagation" de l'erreur d'interpolation à travers l'espace $\widehat{S}$ tout entier. Ainsi l'erreur d'approximation en un macro-état dépend de l'erreur d'interpolation dans tous les macro-états. Comme nous le verrons par la suite, la mesure des dépendances à distance joue un rôle très important pour le problème que nous abordons dans cette partie : le calcul automatique d'une

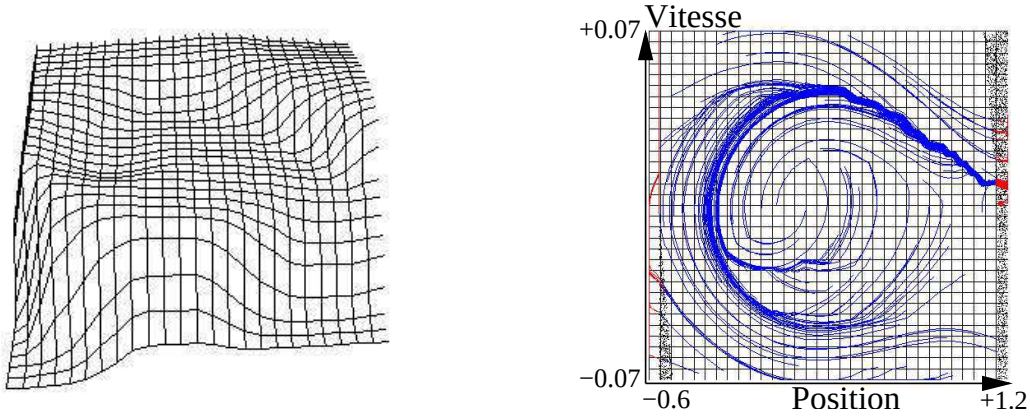

FIG. 3.5: **Approximation par agrégation du problème "voiture sur la colline".** Une approximation par agrégation permet de calculer une fonction de valeur optimale approximative (à gauche). A partir de celle-ci, nous pouvons en déduire un plan et observer des exemples de trajectoires que la voiture emprunterait dans l'espace des phases (à droite). Décrivons une trajectoire typique qui permet d'atteindre le haut de la colline : la voiture commence par s'éloigner du but, puis elle utilise la gravité pour gagner de la vitesse et gravir la colline. On peut également faire la remarque suivante : un certain nombre de conditions initiales (vitesse négative d'amplitude trop élevée et départ trop près du ravin) amènent inéluctablement la voiture dans le ravin.

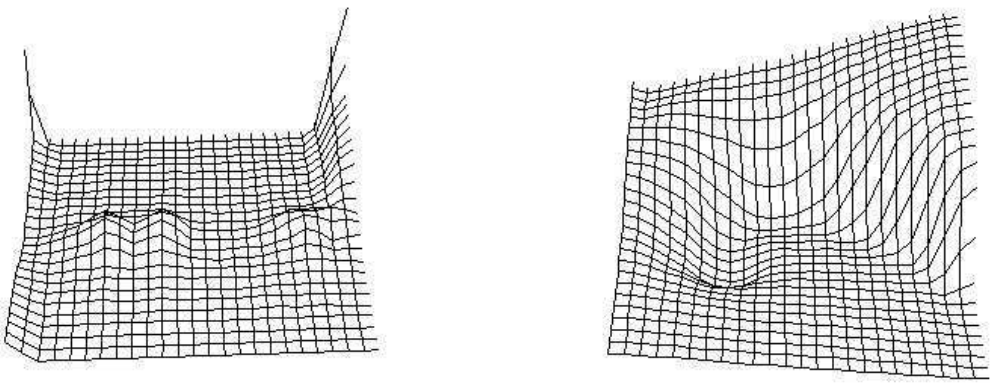

FIG. 3.6: **Bornes supérieures de l'erreur d'interpolation et de l'erreur d'approximation pour le problème "voiture sur la colline".** La propagation de l'erreur d'interpolation n'est pas toujours isotrope ; elle dépend de la dynamique du problème.

agrégation de l'espace d'états. La mesure des dépendances non-locales impose l'introduction d'un nouveau concept : l'influence.

3.2.1 Influence sur un état

Définition

La notion d'influence, originellement introduite dans [Munos et Moore, 2000], permet de mesurer des corrélations non locales entre états pour le calcul d'une quantité vérifiant une équation de type Bellman. Considérons une chaîne de Markov sur un espace S, de probabilité de transition $p(s_{t+1} = s'|s_t = s) = T(s, s')$. Etant donnée une fonction $y : S \to \mathbb{R}$, considérons la fonction $x : S \to \mathbb{R}$ implicitement définie par l'équation de Bellman suivante :

$$x(s) = y(s) + \gamma . \sum_{s'} T(s, s').x(s') \tag{3.11}$$

Définition 5 (Influence sur un état)
L'influence de l'état s sur l'état s' est définie comme étant la contribution de $y(s)$ au calcul de $x(s')$:

$$I_T(s|s') = \frac{\partial x(s')}{\partial y(s)} \tag{3.12}$$

Trois propriétés

Pour bien comprendre la notion d'influence, nous allons énoncer trois de ses propriétés caractéristiques.

1. L'influence $I_T(s|s')$ d'un état s sur un état s' est la somme pondérée des probabilités de présence dans l'état s sachant que la chaîne de Markov est dans l'état s' à $t = 0$. Si on note $(s_t)_{(t\in\mathbb{N})}$ cette chaîne de Markov, on a :

$$I_T(s|s') = \sum_{t=0}^{\infty} \gamma^t . p(s_t = s|s_0 = s')$$

2. La notation $I_T(s|s')$ est cohérente car l'influence dépend uniquement de la probabilité de transition T (et non de y). En particulier elle vérifie l'équation récursive suivante :

$$\forall (s, s') \in S^2,\ I_T(s|s') = \delta_{s,s'} + \gamma . \sum_{s''} T(s'', s).I_T(s''|s')$$

avec

$$\forall (s, s') \in S^2,\ \delta_{s,s'} = \begin{cases} 1 & \text{si } s = s' \\ 0 & \text{si } s \neq s' \end{cases}$$

3. Soit $I^{(0)}$ une fonction quelconque sur S. Alors l'influence sur s' $I(.|s')$ est l'unique limite de la suite $(I^{(n)})_{(n\in\mathbb{N})}$ définie par la relation de récurrence suivante :

$$\forall s \in S,\ I^{(n+1)}(s) = \delta_{s,s'} + \gamma . \sum_{s''} T(s'', s).I^{(n)}(s'') \tag{3.15}$$

L'annexe A reproduit les démonstrations de ces propriétés (initialement publiées dans [Munos et Moore, 2002]).

Illustration

Nous illustrons la notion d'influence sur les deux problèmes que nous avons introduits un peu plus tôt : la navigation continue et la voiture sur la colline. Pour chacun de ces deux problèmes, nous pouvons calculer la fonction de valeur optimale $\widehat{V}^{\widehat{\pi}^*}$ du modèle approché et en déduire la politique correspondante $\widehat{\pi}^*$. On a alors la relation :

$$\widehat{V}^{\widehat{\pi}^*}(\widehat{s}) = \widehat{R}(s, \widehat{\pi}^*(\widehat{s})) + \gamma . \sum_{\widehat{s'}} \widehat{T}(\widehat{s}, \widehat{\pi}^*(\widehat{s}), \widehat{s'}) . \widehat{V}^{\widehat{\pi}^*}(\widehat{s'})$$

On peut s'intéresser à l'influence définie à partir de cette équation. D'après la définition que nous avons donnée (équation 3.12), l'influence sur un macro-état $\widehat{s'}$ permet de calculer la contribution de la récompense en tout macro-état $\widehat{s}$ sur la fonction de valeur en $\widehat{s'}$:

$$I_T(\widehat{s}|\widehat{s'}) = \frac{\partial \widehat{V}^{\widehat{\pi}^*}(\widehat{s'})}{\partial \widehat{R}(\widehat{s}, \widehat{\pi}^*(\widehat{s}))}$$

De manière équivalente (cf. propriétés caractéristiques de l'influence), l'influence est aussi la somme pondérée des probabilités si le système suit la politique optimale en partant de ce même macro-état $\widehat{s'}$.

La figure 3.7 montre l'influence sur un macro-état s'. Pour le problème de navigation, s' est le macro-état qui contient le vrai état $(x, y) = (2, 3)$. Pour la voiture sur la colline, $\widehat{s'}$ est le macro-état qui contient le vrai état $(x, v) = (0, 0)$. Dans les deux cas, on voit que la zone de l'espace d'états qui a le plus d'influence sur le point de départ s' correspond à la zone but. C'est assez logique : dans ces deux problèmes, l'exécution du plan optimal va amener le système à atteindre la zone but et à y rester. Si on changeait la valeur du but, cela changerait la valeur le long de la trajectoire optimale. L'interprétation de l'influence en terme de probabilité de présence dans le futur lorsque le système part de s' est aussi assez claire : On distingue sous forme de trace la trajectoire de la politique π^* qui amène le système jusqu'à la zone but.

3.2.2 Influence pondérée

Définition

La notion d'influence permet de considérer la dépendance entre un macro-état et tous les autres (relation $1 \leftrightarrow n$). Les auteurs de [Munos et Moore, 2000] proposent d'étendre la notion d'influence sur un état à celle d'influence sur un ensemble d'états. Cette notion permet de considérer les dépendances entre plusieurs macro-états et tous les autres (relation $n \leftrightarrow n$) :

Définition 6 (Influence sur un ensemble)
Soit $A \subset S$ un ensemble d'états. L'influence de l'état $s \in S$ sur A est :

$$I_T(s|A) = \sum_{s' \in A} \frac{\partial x(s')}{\partial y(s)} \tag{3.16}$$

Nous proposons d'aller un peu plus loin en définissant d'une manière plus générale la notion d'influence pondérée :

Définition 7 (Influence pondérée)
Soit une fonction de pondération sur l'espace d'états $w : S \to \mathbb{R}^+$ telle que $\sum_s w(s) < \infty$. Alors l'influence d'un état s pondérée par w est :

$$I_T(s|w) = \sum_{s' \in S} \frac{\partial x(s')}{\partial y(s)} . w(s') = \sum_{s' \in S} I_T(s|s') . w(s')$$

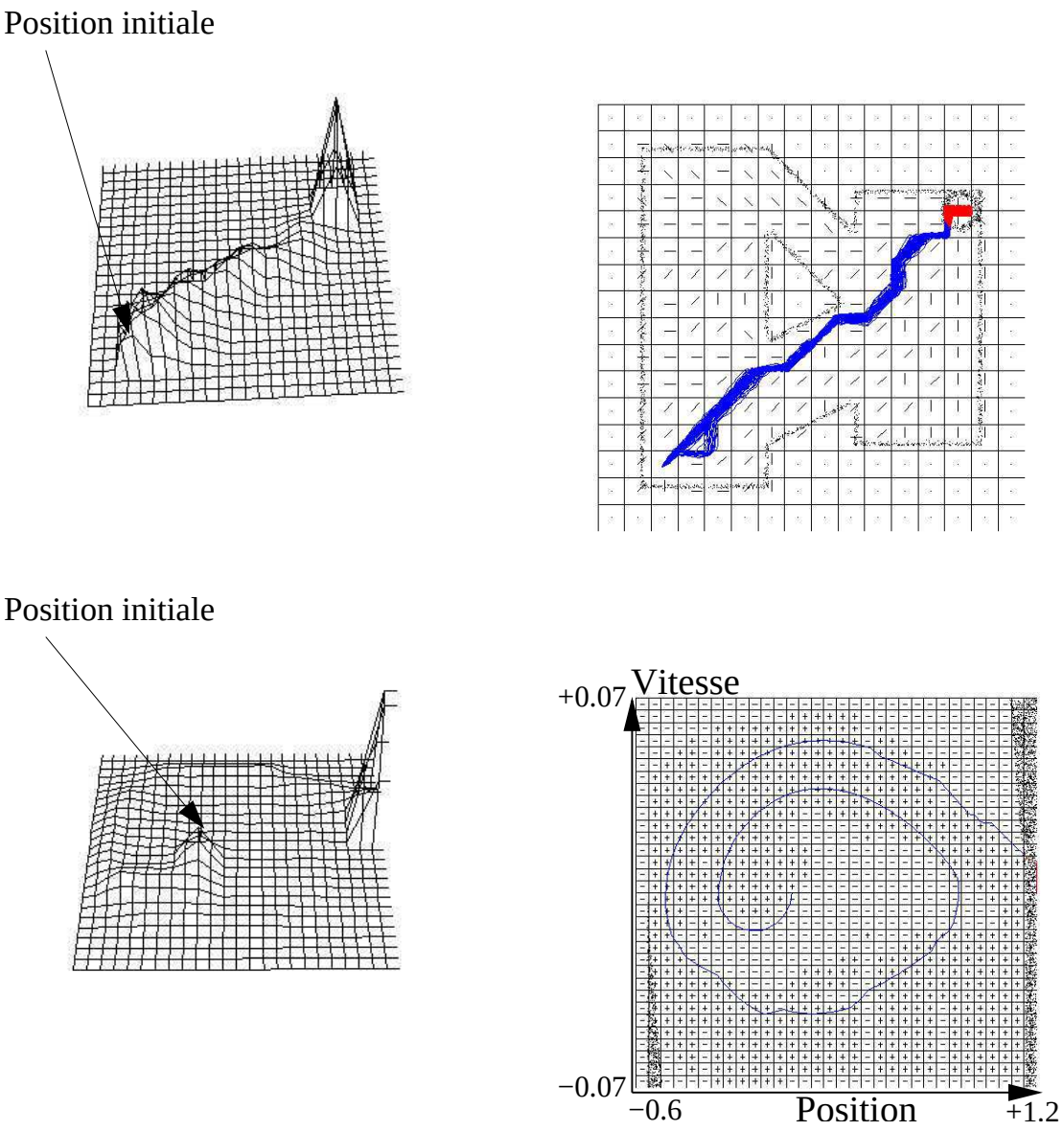

FIG. 3.7: **Influence sur la position initiale.** L'influence dit combien la récompense en tout état contribue au calcul de la fonction de valeur en un état donné ; elle mesure également la probabilité de présence dans le futur sachant que le système est dans cet état à $t = 0$. Le haut de cette figure correspond au problème "navigation continue". Le bas correspond au problème "voiture sur la colline". A gauche nous avons tracé l'influence sur l'état "position initiale". A droite, nous avons lancé des trajectoires partant de cet état. On observe le lien entre le faisceau des trajectoires et la répartition de l'influence.

Il s'agit bien d'une notion plus générale : l'influence sur un ensemble $A \subset S$ est l'influence pondérée où le poids $w : S \to \mathbb{R}^+$ est la fonction indicatrice de l'ensemble A ($w(s) = 1 \Leftrightarrow s \in A$ et $w(s) = 0$ sinon). Nous utiliserons cette notion dans la prochaine section.

Propriétés

Les propriétés de l'influence sur un état s'étendent ainsi à l'influence selon une pondération :

1. Notons $(s_t)_{(t\in\mathbb{N})}$ la chaîne de Markov définie à partir de T. Alors on a :

$$I_T(s|w) = \sum_{s'} \sum_{t=0}^{\infty} \gamma^t . p(s_t = s | s_0 = s') . w(s') \tag{3.17}$$

2. L'influence selon une pondération vérifie l'équation récursive suivante :

$$\forall s \in S,\ I_T(s|w) = w(s) + \gamma . \sum_{s''} T(s'', s) . I_T(s''|w)$$

3. Soit $I^{(0)}$ une fonction quelconque sur S. Alors l'influence selon la pondération w est l'unique limite de la suite $(I^{(n)})_{(n\in\mathbb{N})}$ définie par la relation de récurrence suivante :

$$\forall s \in S,\ I^{(n+1)}(s) = w(s) + \gamma . \sum_{s''} T(s'', s) . I^{(n)}(s'') \tag{3.19}$$

Preuves : Les deux premières propriétés découlent des résultats concernant l'influence sur un état et le fait que :

$$I_T(s|w) = \sum_{s' \in S} I_T(s|s') . w(s')$$

1. Pour tout $s \in S$:

$$I_T(s|w) = \sum_{s' \in S} I_T(s|s') . w(s') = \sum_{s' \in S} I_T(s|s') . w(s') = \sum_{s' \in S} \left(\sum_{t=0}^{\infty} \gamma^t . p(s_t = s | s_0 = s') \right) . w(s') \ \Diamond$$

2. Pour tout $s \in S$:

$$\begin{aligned}
I_T(s|w) &= \sum_{s' \in S} I_T(s|s') . w(s') && (3.20)\\
&= \sum_{s' \in S} \left(\delta_{s,s'} + \gamma . \sum_{s''} T(s'', s) . I_T(s''|s') \right) . w(s') && (3.21)\\
&= \sum_{s' \in S} \delta_{s,s'} . w(s') + \gamma . \sum_{s''} T(s'', s) . \left(\sum_{s' \in S} I_T(s''|s') . w(s') \right) && (3.22)\\
&= w(s) + \gamma . \sum_{s''} T(s'', s) . I_T(s''|w) \ \Diamond && (3.23)
\end{aligned}$$

3. La démonstration de la dernière propriété est analogue à celle concernant l'influence sur un état (voir annexe A). L'équation 3.19 est une contraction si on utilise la norme $\|f\|_1 = \sum_{s\in S} |f(s)|$. En effet :

$$\|I^{(n+1)} - I_T(.|w)\|_1 = \sum_{s\in S} \left| \gamma . \sum_{s''} T(s'',s).(I^{(n)}(s'') - I_T(s''|w)) \right| \quad (3.24)$$

$$= \gamma . \sum_{s,s''} T(s'',s). \left| I^{(n)}(s'') - I_T(s''|w) \right| \quad (3.25)$$

$$= \gamma . \|I^{(n)} - I_T(.|s')\|_1 \quad (3.26)$$

La suite $(I^{(n)})_{(n\in\mathbb{N})}$ a donc une unique solution qui est $I_T(.|w)$. $\Diamond$

Illustration

Considérons les deux problèmes avec lesquels nous avons illustré la notion d'influence sur un point : la navigation continue et la voiture sur la colline. Si nous considérons cette fois-ci que le système a une probabilité de présence initiale homogène sur l'espace d'états S, l'influence pondérée par cette distribution permet de mesurer :

- La contribution de la récompense en un macro-état à la fonction de valeur toute entière. En effet, l'influence pondérée par une distribution homogène est par définition proportionnelle à $\frac{\partial \sum_{\widehat{s'}} \widehat{V}^{\widehat{\pi}^*}(\widehat{s'})}{\partial \widehat{R}(\widehat{s},\widehat{\pi}^*(\widehat{s}))}$
- La somme pondérée des probabilités de présence en chaque macro-état lorsqu'on a une probabilité de présence initiale homogène dans S.

La figure 3.8 illustre le calcul de l'influence pondérée par une distribution homogène pour les deux PDM que considérons dans cette partie.

3.3 Réduction de l'erreur d'approximation

Une bonne agrégation est une agrégation qui induit une petite erreur d'approximation. Pour diminuer l'erreur d'approximation initiale d'un PDM approché, la relation entre erreur d'approximation et erreur d'interpolation locale (équation 3.9 page 65) suggère de diminuer l'erreur d'interpolation locale. Diminuer l'erreur d'interpolation locale requiert souvent d'augmenter le nombre d'états du PDM. Comme le nombre d'états est un facteur crucial (pour conserver en particulier les propriétés présentées dans la partie précédente), une approche consiste à augmenter la précision de manière variable, c'est-à-dire seulement là ou cela permet de diminuer le plus l'erreur initiale.

Déterminer les macro-états pour lesquels on a le plus intérêt à diminuer l'erreur d'interpolation afin de globalement diminuer l'erreur d'approximation nous pousse à nous intéresser à la quantité suivante :

$$\frac{\partial \overline{E_{app}^{(0)}}}{\partial \overline{E_{int}}(\widehat{s})} = \sum_{\widehat{s'}\in S} \frac{\partial \overline{E_{app}}(\widehat{s'})}{\partial \overline{E_{int}}(\widehat{s})} . p(s_0 \in \widehat{s'})$$

En effet, si on diminue l'erreur d'interpolation en $\widehat{s}$ de la quantité $\Delta\overline{E_{int}}(\widehat{s})$, l'erreur d'approxi-

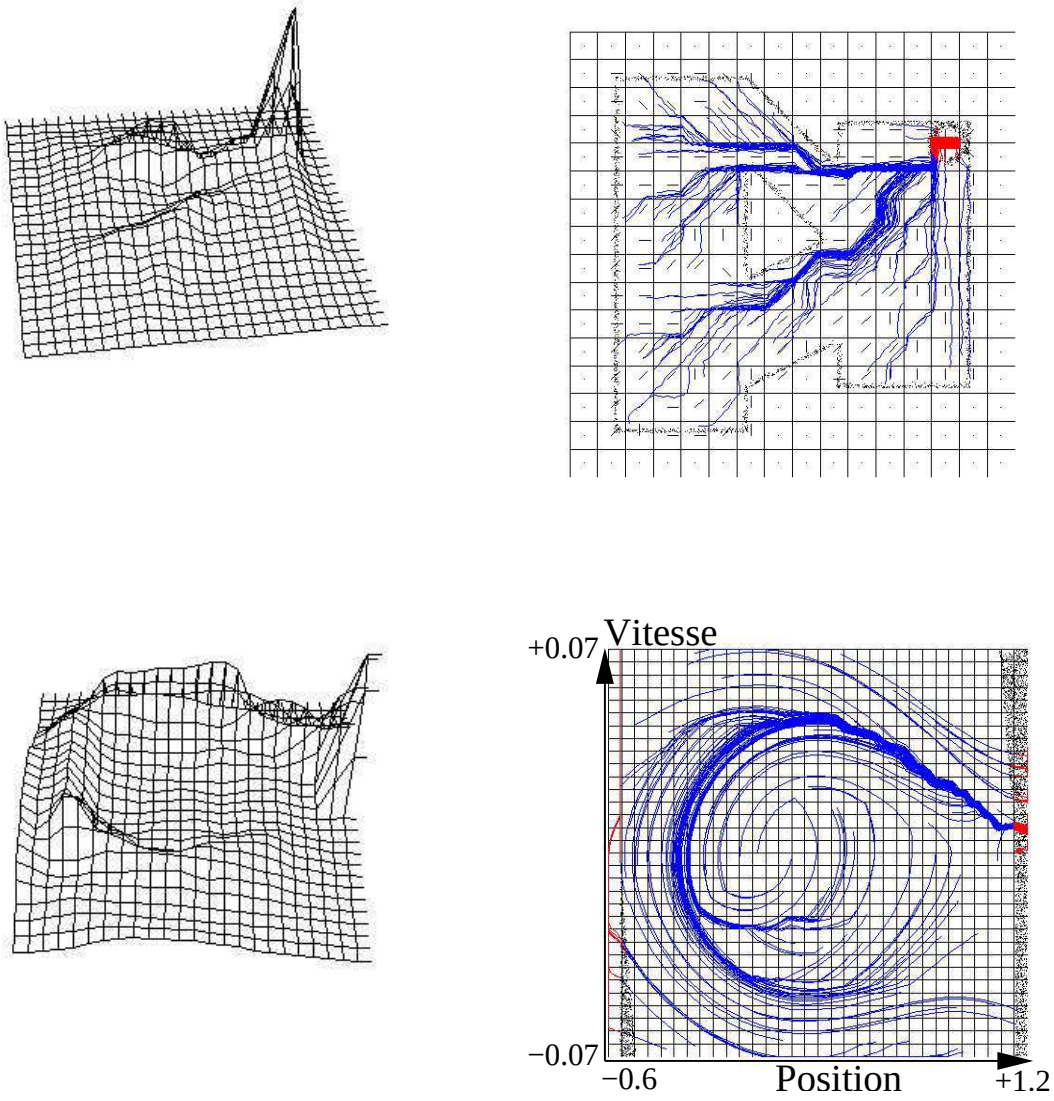

FIG. 3.8: **Influence pondérée sur une distribution homogène.** L'influence pondérée par une distribution homogène dit combien la récompense en tout état contribue au calcul de la fonction de valeur pour l'ensemble des états ; elle mesure également la probabilité de présence dans le futur sachant que le système a une position initiale aléatoire. Le haut de cette figure correspond au problème "navigation continue". Le bas correspond au problème "voiture sur la colline". A gauche nous avons tracé l'influence. A droite, nous avons lancé des trajectoires partant de positions initiales aléatoires. On observe le lien entre le faisceau des trajectoires et la répartition de l'influence.

mation initiale devrait diminuer ainsi :

$$\Delta\overline{E_{app}^{(0)}} \simeq \frac{\partial\overline{E_{app}^{(0)}}}{\partial\overline{E_{int}}(\widehat{s})}.\Delta\overline{E_{int}}(\widehat{s})$$

Le concept d'influence pondérée que nous avons présenté dans la section précédente permet de calculer la dérivée partielle de cette équation. C'est ce que nous détaillons dans ce qui suit.

L'équation 3.9 page 65 montre que le calcul d'une borne supérieure de l'erreur d'approximation correspond à la propagation maximale de l'erreur d'interpolation. Pour tout macro-état $\widehat{s}$, notons $\pi_{err}(\widehat{s})$ l'action qui propage cette erreur au maximum :

$$\pi_{err}(\widehat{s}) = \arg\max_a \left(\gamma. \sum_{\widehat{s'}} \widehat{T}(\widehat{s}, a, \widehat{s'}).\overline{E_{app}}(\widehat{s'}) \right)$$

Alors, l'équation 3.9 devient :

$$\overline{E_{app}}(\widehat{s}) = \gamma. \sum_{\widehat{s'}} \widehat{T}(\widehat{s}, \pi_{err}(\widehat{s}), \widehat{s'}).\overline{E_{app}}(\widehat{s'}) + \overline{E_{int}}(\widehat{s})$$

Finalement, l'équation 3.3 et la définition de l'influence pondérée (équation 3.16) nous permettent de conclure :

$$\frac{\partial\overline{E_{app}^{(0)}}}{\partial\overline{E_{int}}(\widehat{s})} = I_{\widehat{T}_{\pi_{err}}}(\widehat{s}|p^{(0)})$$

avec $\widehat{T}_{\pi_{err}}(\widehat{s}, \widehat{s'}) = \widehat{T}(\widehat{s}, \pi_{err}(s), \widehat{s'})$ et $p^{(0)}(\widehat{s}) = p(s_0 \in \widehat{s})$.

L'intérêt de diminuer l'erreur d'interpolation en un macro-état $\widehat{s}$ se mesure donc à l'aide de l'influence correspondant à l'équation de Bellman du flux d'erreur pondérée par les conditions initiales. Plus précisément, le gain escompté sur l'erreur d'approximation initiale lorsqu'on fait diminuer l'erreur d'interpolation d'un point de $\Delta\overline{E_{int}}(\widehat{s})$ est :

$$\Delta\overline{E_{app}^{(0)}} \simeq I_{\widehat{T}_{\pi_{err}}}(\widehat{s}|p^{(0)}).\Delta\overline{E_{int}}(\widehat{s}) \tag{3.27}$$

Cette dernière quantité peut être utilisée pour décider comment faire évoluer l'agrégation d'un PDM. Si on décrit plus finement certaines zones de l'espace, ou au contraire si on les rend plus grossières, on peut estimer l'impact que cela va avoir sur l'erreur d'interpolation et donc aussi sur l'erreur d'approximation initiale. Nous aurons donc intérêt à découper les zones de l'espace qui font le plus diminuer l'erreur d'approximation. De manière symétrique, nous pourrons nous permettre de rendre grossières des zones qui augmentent le moins l'erreur d'approximation.

Illustration

Considérons de nouveau les deux PDM modélisant les problèmes "navigation continue" et "voiture sur la colline". Les figures 3.9 et 3.10 illustrent la dérivée de l'erreur d'approximation par rapport à l'erreur d'interpolation et la diminution de l'erreur d'approximation si l'erreur d'interpolation était diminuée de moitié (c'est-à-dire pour $\Delta\overline{E_{int}}(\widehat{s}) = -\frac{\overline{E_{int}}(\widehat{s})}{2}$). Les zones de l'espace d'états pour lesquelles on observe des valeurs élevées de diminution potentielle sont des zones où il faudrait réduire en priorité l'erreur d'interpolation. Ceci pourrait par exemple se faire en discrétisant de manière plus précise ces zones.

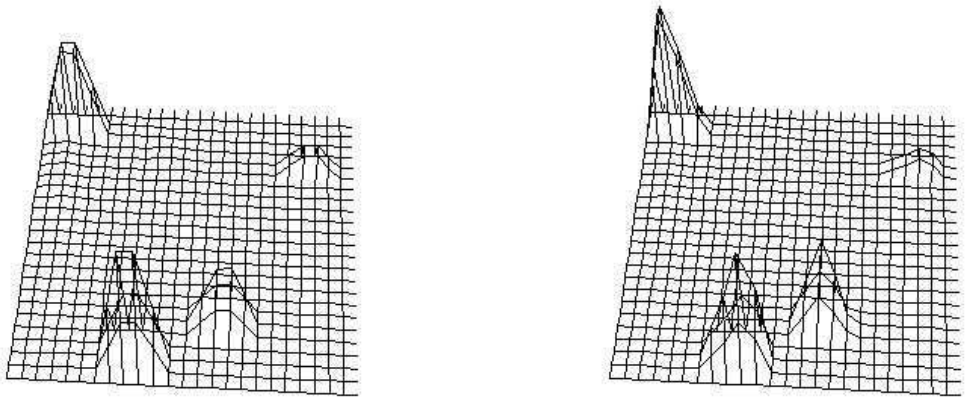

FIG. 3.9: **Zones dont l'erreur d'interpolation influence le plus l'erreur d'approximation et zones à "désagréger en priorité" pour le problème "navigation continue".** A gauche, les zones ayant une valeur élevée sont celles où l'erreur d'interpolation contribue le plus à l'erreur d'approximation. A droite, les zones ayant une valeur élevée sont celles qui pourraient le plus faire diminuer l'erreur d'approximation ; ce sont donc des zones pour lesquelles il serait très intéressant de rendre l'agrégation plus détaillée.

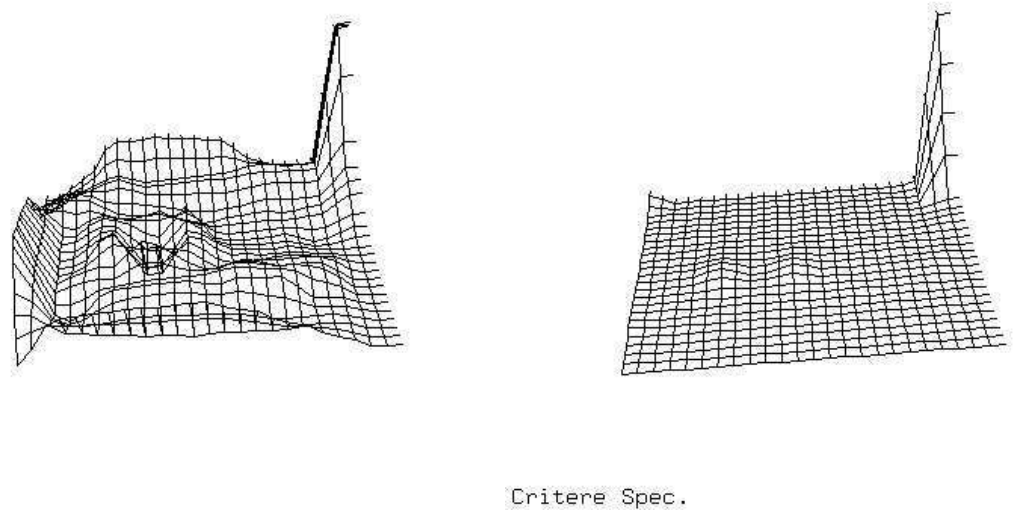

FIG. 3.10: **Zones dont l'erreur d'interpolation influence le plus l'erreur d'approximation et zones à "désagréger en priorité" pour le problème "voiture sur la colline".** Voir les commentaires de la figure 3.9.

Résumé du chapitre

Ce chapitre nous a permis d'étudier de manière théorique l'approximation d'un PDM à l'aide d'une méthode de type agrégation d'états.

Dans un premier temps, nous avons montré comment calculer une borne supérieure de l'erreur commise sur la fonction de valeur optimale lorsqu'on utilise une approximation par agrégation d'états. Cette borne est caractérisée par une équation de Bellman analogue à l'équation qui caractérise la fonction de valeur optimale dans le PDM agrégé. Ensuite, nous avons introduit les notions d'influence et d'influence pondérée. Elles permettent de mesurer des relations de dépendance entre états pour des calculs du type équation de Bellman. Finalement, nous avons montré comment la notion d'influence pondérée permet de déterminer les zones critiques de l'espace d'états, c'est-à-dire les zones agrégées qui engendrent le plus d'erreur. Dans l'optique où l'on souhaiterait diminuer l'erreur d'approximation, ce sont ces zones qu'il faudrait "désagréger" en priorité. Cette idée sera illustrée en détail dans le prochain chapitre.

La méthodologie que nous avons présentée est fortement inspirée de celle décrite dans [Munos et Moore, 2000]. Les contributions nouvelles sont les suivantes :

- Nous avons adapté la démarche générale d'analyse de l'erreur d'approximation au cas particulier de l'agrégation d'états ; en particulier, nous avons montré que le calcul d'une borne supérieure de l'erreur d'approximation est du même ordre de complexité que le calcul de la fonction de valeur optimale dans le modèle approché.
- Nous avons introduit le concept d'influence pondérée qui est une généralisation de l'influence sur un ensemble (initialement introduit dans [Munos et Moore, 2002]). L'influence pondérée donne un peu plus de flexibitilité pour mesurer les relations de dépendance dans un PDM.

Chapitre 4

Amélioration itérative d'une approximation par agrégation : expériences

Le chapitre précédent nous a montré comment faire le lien entre l'erreur de description locale d'un PDM agrégé et l'erreur sur le calcul de la fonction de valeur optimale. Nous avons en particulier montré comment déterminer les zones critiques d'une agrégation, c'est-à-dire les zones qui génèrent le plus d'erreur dans le calcul de la fonction de valeur optimale.

Ce chapitre propose d'exploiter pratiquement ce genre d'informations pour améliorer itérativement l'agrégation d'un PDM. Nous allons étudier expérimentalement trois cas pratiques d'amélioration que nous appelons respectivement spécialisation, généralisation et apprentissage. Ces trois cas correspondent aux modifications suivantes :

- Spécialisation : on complexifie l'agrégation (on augmente le nombre des états) afin que la qualité augmente le plus vite possible ;
- Généralisation : on simplifie l'agrégation (on diminue le nombre d'états) de sorte à minimiser la perte de qualité du modèle ;
- Apprentissage : on modifie l'agrégation en maintenant sa complexité constante ; l'objectif est ici d'améliorer la qualité de l'approximation tout en gardant constant le nombre d'états.

Le dernier procédé, "apprentissage", est particulièrement intéressant : il montre comment faire évoluer une agrégation à ressources et complexité constantes.

Nous commençons par présenter le cadre général dans lequel nous allons expérimenter l'amélioration de l'agrégation : nous décrivons une technique d'agrégation d'espaces continus et détaillons le protocole expérimental que nous utilisons. Nous étudions expérimentalement la spécialisation, la généralisation et l'apprentissage sur les deux problèmes simples que nous avons considérés dans le chapitre précédent : la "navigation continue" et la "voiture sur la colline". Nous proposons ensuite trois heuristiques qui permettent d'augmenter significativement les performances. Finalement, nous montrons que notre approche et les heuristiques s'appliquent efficacement à des problèmes d'A/R complexes : nous l'illustrons sur deux problèmes de contrôle continu et un problème d'observabilité partielle.

4.1 Description générale du protocole expérimental

Nous allons introduire le protocole expérimental utilisé pour tester l'amélioration d'une agrégation. Nous commençons par présenter la technique pour agréger des espaces d'états de très

grande taille. Nous montrons ensuite comment nous effectuons l'apprentissage des paramètres dans la version agrégée du PDM. Enfin, nous détaillons l'algorithme général suivi ainsi que la manière dont nous mesurons les performances.

4.1.1 Une technique d'agrégation

La plupart des illustrations que nous allons faire dans ce chapitre considèrent que l'espace d'états du vrai PDM s'inscrit dans un pavé de l'espace vectoriel $\mathbb{R}^d$. Autrement dit, nous nous plaçons dans le cas où tout point du vrai espace d'états S a d coordonnées numériques $(x_1, .. , x_d)$ avec $x_i \in [A_i, B_i]$. Afin de réaliser une agrégation dans un tel espace, nous proposons d'utiliser un découpage à l'aide d'un arbre, structure connue sous le nom de *kd-tree* [Friedman *et al.*, 1977]. L'emploi d'une telle structure est usuel pour agréger des PDM inscrits dans des pavés vectoriels. On en trouve plusieurs exemples dans la littérature[18] : [Moore, 1994] [Munos et Moore, 2002] [Reynolds, 2001] [Nakamura, 1998]. Le principe de l'agrégation est alors le suivant : la racine couvre l'espace-pavé tout entier; elle a deux branches qui coupent l'espace en deux selon une dimension choisie. De manière récursive, tout nœud couvre une partie de l'espace et ses deux fils en couvrent deux moitiés. Les feuilles d'un tel arbre constituent alors une partition à résolution variable de l'espace S. La figure 4.1 illustre sur un cas simple ce procédé.

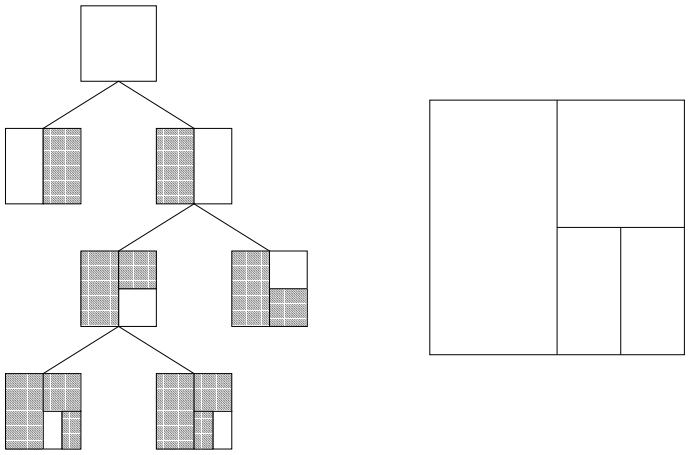

FIG. 4.1: **Exemple de structure en arbre pour discrétiser un espace à 2 dimensions.**

4.1.2 Méthode d'estimation des paramètres

Avoir une agrégation en macro-états ne suffit pas pour décrire complètement le modèle approché $\widehat{\mathcal{M}}$. Il faut aussi connaître ses paramètres $\widehat{T}$ et $\widehat{R}$. Pour faire concrètement l'apprentissage des paramètres, nous nous appuyons sur ce qu'on appelle un modèle générateur [Kearns *et al.*, 2000]. Ce dernier est un simulateur du problème qui nous intéresse : pour tout état $s \in S$ et toute action $a \in A$ du vrai PDM, le modèle générateur renvoie un échantillon correspondant à une récompense immédiate $r \in \mathbb{R}$ et un nouvel état $s' \in S$.

Comme chaque macro-état du modèle agrégé $\widehat{\mathcal{M}}$ est un pavé de $\mathbb{R}^d$, nous proposons d'estimer $\widehat{T}$ et $\widehat{R}$ à partir de n^d points répartis de façon homogène dans chacun de ces pavés (n positions

[18] L'intérêt essentiel de l'organisation en arbre tient au temps relativement rapide (en log du nombre de feuilles de l'arbre) pour déterminer dans quelle partie se situe un point quelconque de $\mathbb{R}^d$.

équiréparties selon chaque dimension). A partir de ces points, nous utilisons le modèle générateur pour évaluer l'effet de chacune des actions. Pour tenir compte du caractère potentiellement stochastique du problème, nous proposons de lancer ces expérimentations k fois. Nous avons donc, pour chaque état et chaque action, un ensemble de $k.n^d$ échantillons donnant chacun un état suivant, et la récompense reçue. Nous utilisons ces informations pour calculer des estimations de $\widehat{T}$ et $\widehat{R}$. Nous utilisons le même nombre d'échantillons pour chaque état afin de "connaître" tous les états du PDM avec la même précision. La figure 4.2 illustre l'estimation des paramètres pour un état d'un PDM agrégé.

Ce procédé nous permet également de mesurer les variations de R et de T dans chacun des macro-états. Rappelons en effet que cette information est nécessaire pour le calcul de l'erreur d'interpolation locale (voir section 3.1.2).

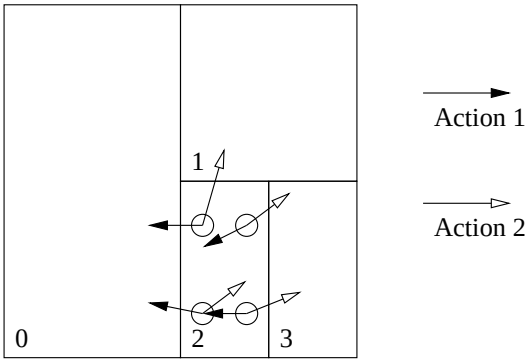

FIG. 4.2: **Estimation des paramètres pour un PDM à 2 actions.** Il s'agit d'un espace à $d = 2$ dimensions. Dans chaque état, on prend $n \times n$ points (ici $n = 2$) répartis de manière homogène selon les d dimensions. Pour chacun de ces points, on simule les actions et on obtient des échantillons (état, action, état suivant, récompense). On utilise ces échantillons pour avoir des estimations de $\widehat{T}$, $\widehat{R}$, ΔR et ΔT (voir section 3.1.2 page 65). Dans l'exemple présenté ici, nous avons numéroté les états de 0 à 3, et nous avons les échantillons suivants : (s_2, a_1, s_0), (s_2, a_1, s_0), (s_2, a_1, s_2), (s_2, a_1, s_2)... et on en déduit par exemple $\widehat{T}(s_2, a_1, s_0) = 0.5$ et $\widehat{T}(s_2, a_2, s_1) = 0.25$

4.1.3 Etude de l'évolution de l'agrégation

Nous évaluons expérimentalement la mise à jour de l'agrégation sur plusieurs problèmes. Nous considérons les deux exemples de PDM que nous avons présentés au début de ce chapitre :

- la navigation continue,
- la voiture sur la colline.

Pour chacun de ces deux problèmes, nous considérons deux types de conditions initiales :

- départ unique : une position initiale particulière,
- départ aléatoire : une distribution homogène de points de départ sur l'espace d'états.

Nous avons donc au total quatre problèmes. Pour chacun d'entre eux l'étude des modifications d'agrégation sont étudiées à l'aide de l'algorithme itératif et générique 4.1. A chaque itération de cet algorithme, nous évaluons les plans optimaux approximatifs à l'aide de 500 trajectoires de 500 pas de temps simulées à l'aide du modèle générateur. Sur chacune de ces trajectoires, on mesure les performances en faisant la somme pondérée des récompenses : $\sum_{t=0}^{499} \gamma^t.r_t$

Algorithme 4.1 Algorithme générique pour la mise à jour d'une agrégation

Initialisation

On définit une agrégation initiale régulière découpant l'espace d'états en zones de tailles égales.

Itérations

Estimer $\widehat{T}$, $\widehat{R}$, ΔR et ΔT en utilisant le modèle générateur (paramètres $k = 3$ et $n = 5$).

Calculer la politique optimale à l'aide de l'algorithme Value Iteration. (paramètres $\beta = 0.99$ et $\varepsilon = 10^{-4}$)

Calculer la borne supérieure de l'erreur d'interpolation

Calculer la borne supérieure de l'erreur d'approximation (cf. section 3.1.2)

Calculer la dérivée de l'erreur d'approximation par rapport à l'erreur d'interpolation (cf. section 3.3).

En déduire une nouvelle agrégation (selon le cas "spécialisation" "généralisation" ou "apprentissage", voir section 4.2.1 ou 4.2.2 ou 4.2.3)

4.2 Premiers résultats empiriques

Nous détaillons à présent les trois processus que nous avons testés. Nous considérons successivement la spécialisation (on ajoute des états), la généralisation (on enlève des états) et l'apprentissage (on ajoute et on enlève des états).

4.2.1 Procédé de spécialisation

Pour augmenter la précision du modèle, nous commençons par nous intéresser au cas où nous augmentons le nombre d'états. Itération après itération, nous choisissons une proportion des états du modèle courant et les divisons en deux nouveaux états. Si l'on regarde l'arbre qui définit l'agrégation, diviser un état consiste à lui ajouter deux fils. Nous supposons, pour simplifier, que la division d'un état permet de diminuer sa borne d'erreur d'interpolation d'une quantité proportionnelle à la borne courante :

$$\Delta\overline{E_{int}}(\widehat{s}) \propto -\overline{E_{int}}(\widehat{s})$$

Alors l'équation 3.27 nous permet de décider quels états nous avons le plus intérêt à diviser : ce sont ceux pour qui $I_{\widehat{T}_{\pi_{err}}}(s|p^{(0)}).\overline{E_{int}}(s)$ a les plus grandes valeurs. Dans les expériences que nous avons menées, nous avons divisé à chaque itération les 10% des états qui ont le plus grand score.

La figure 4.3 montre les premières étapes de la spécialisation dans le PDM navigation continue. La figure 4.4 montre des agrégations obtenues pour les deux PDM avec départ aléatoire.

Pour juger de l'intérêt du procédé de spécialisation, nous avons comparé les performances avec celles obtenues pour des grilles homogènes régulières de plusieurs tailles : $2^{n.d}$ pour $n \geq 2$. La figure 4.5 illustre, pour chacun des quatre problèmes que nous avons considérés les performances en fonction du nombre d'états. Dans trois cas sur quatre, les performances que nous obtenons atteignent nettement plus rapidement ce qui apparaît comme la performance optimale. Le seul cas pour lequel l'utilisation du critère n'est pas concluante est le PDM navigation continue avec départ connu.

4.2.2 Procédé de généralisation

La relation entre erreur d'interpolation et erreur d'approximation permet également de s'intéresser au problème de diminuer le nombre d'états d'un PDM en minimisant la perte des perfor-

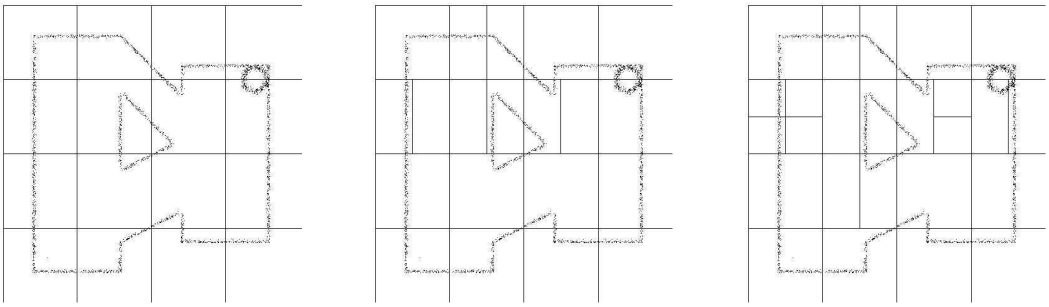

FIG. 4.3: **Evolution de l'agrégation du PDM navigation continue par spécialisations successives.** Les trois schémas ci-dessus représentent les agrégations de l'espace d'états aux trois premières itérations du processus de spécialisation. A chaque itérations le nombre d'états augmente de 10%. Le choix des zones où l'on rend plus précise l'agrégation est motivé par l'amélioration globale des performances.

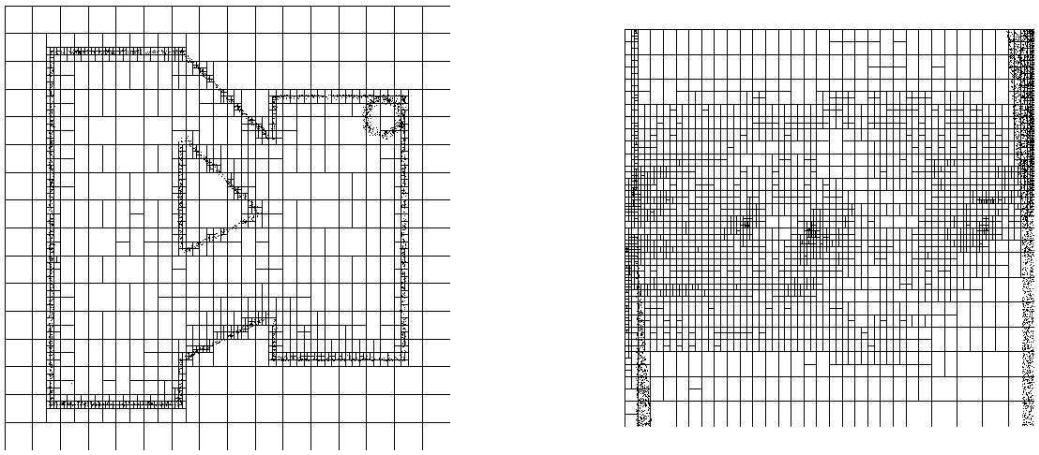

FIG. 4.4: **Agrégations obtenues après spécialisation pour les problèmes "Navigation continue" et "Voiture sur la colline".** Ces figures représentent les agrégations obtenues après un grand nombre d'itérations du processus de spécialisation. Dans les deux cas, on observe que la répartition des ressources n'est pas homogène mais adaptée à la dynamique du problème. Sur le problème de "navigation continue" (à gauche) qui est le plus intuitivement interprétable, on peut voir que les ressources se concentrent dans les zones où la dynamique varie le plus (près des murs).

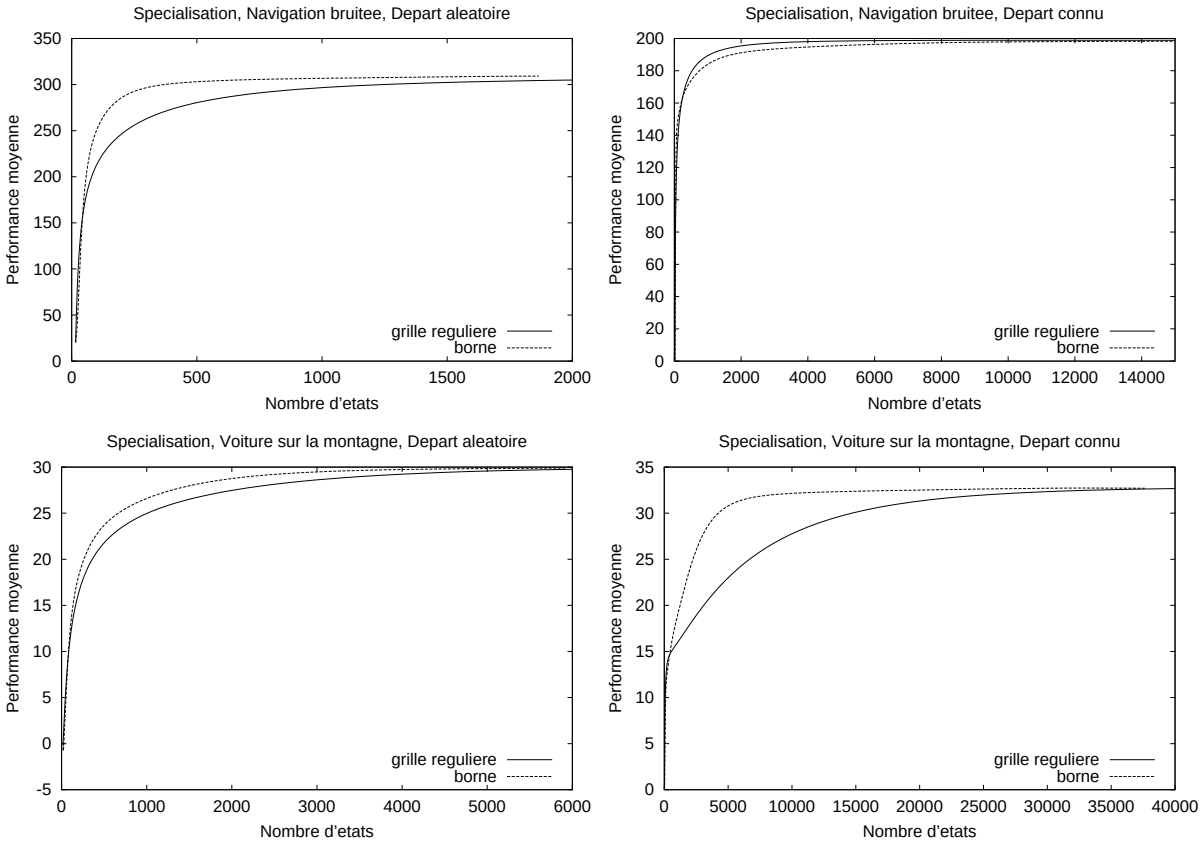

FIG. 4.5: **Evolution des performances pendant le processus de spécialisation.** Ces figures comparent, pour les quatre problèmes considérés, l'évolution des performances quand on augmente la précision de l'agrégation aux endroits déterminés par le processus de spécialisation (courbe notée "borne"), et quand on augmente la précision de manière homogène (dans tous les états à chaque itération, courbe notée "grille régulière"). L'ordonnée représente les performances et l'abscisse le nombre d'états. On observe que dans trois cas sur quatre, les agrégations (non-régulières) successives obtenues par le processus de spécialisation amènent plus rapidement de bonnes performances. L'utilisation d'une agrégation non-régulière adaptée est alors profitable.

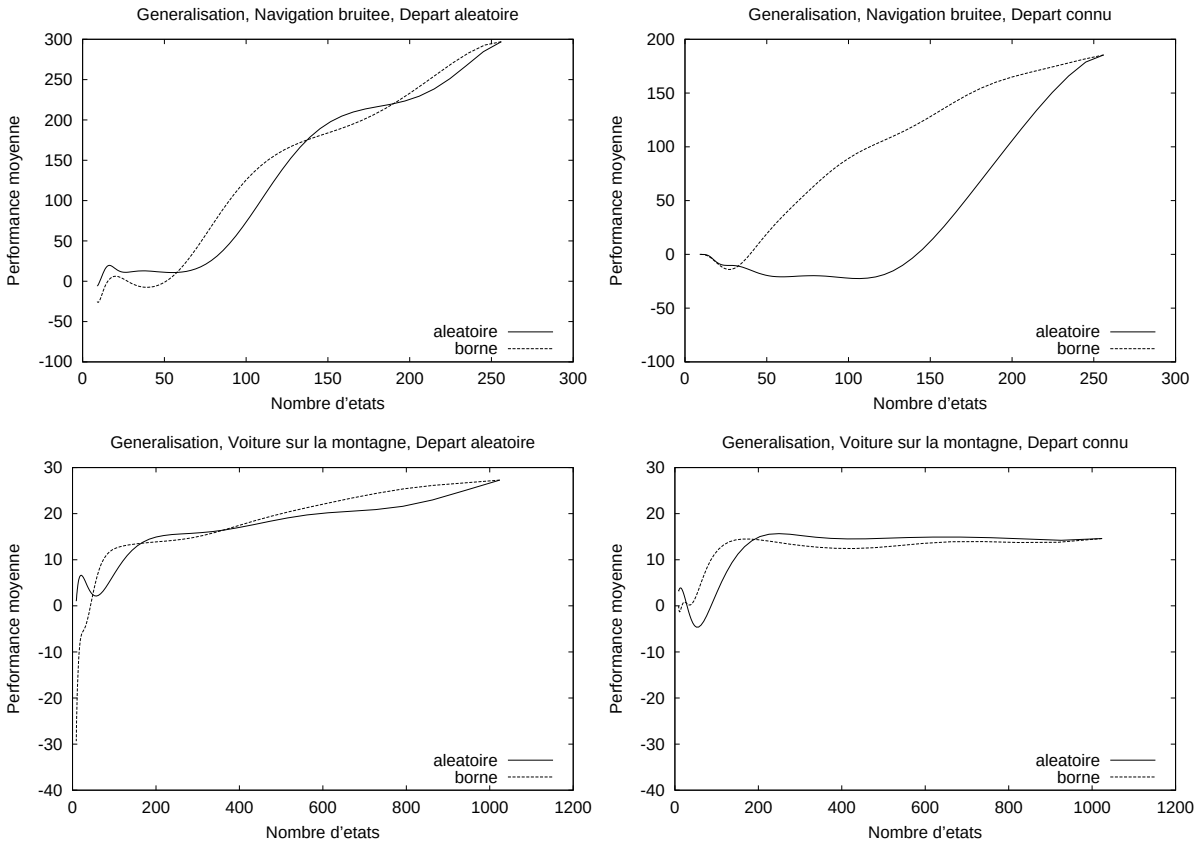

FIG. 4.6: **Evolution des performances pendant le processus de généralisation.** Ces figures comparent, pour les quatre problèmes considérés, l'évolution des performances quand on diminue la précision de l'agrégation aux endroits déterminés par le processus de généralisation (courbe notée "borne"), et quand on diminue cette précision de manière aléatoire (courbe notée "aléatoire"). L'ordonnée représente les performances et l'abscisse le nombre d'états. Au début du processus le nombre d'états est maximal. Petit à petit, le nombre d'états diminue et les performances également (l'évolution du processus se fait de droite à gauche). On observe que dans l'un des cas ("navigation continue" avec départ connu) le choix des états est pertinent : les performances se maintiennent significativement plus longtemps.

mances alors qu'on supprime des ressources. Ceci a un intérêt fonctionnel particulier : diminuer la complexité de calcul de la politique optimale pour ce modèle.

Avec l'agrégation de l'espace représentée sous forme d'arbre, nous proposons de diminuer le nombre des états en "coupant deux feuilles" ayant un même père, ou encore (cela revient au même) en fusionnant des états. Comme pour la spécialisation, nous faisons une hypothèse sur l'effet que cela va avoir sur l'erreur d'interpolation. Nous supposons que le fait de fusionner les feuilles $\widehat{s_1}$ et $\widehat{s_2}$ aura pour effet d'augmenter leurs erreurs d'interpolation respectives d'une quantité qui est proportionnelle à cette erreur :

$$\forall i \in \{1,2\}, \Delta\overline{E_{int}}(s_i) \propto \overline{E_{int}}(s_i)$$

L'équation 3.27 nous dit alors que l'intérêt a priori de fusionner deux feuilles s_1 et s_2 est inversement proportionnel à la quantité $I_{\widehat{T}_{\pi_{err}}}(s|p^{(0)}).\overline{E_{int}}(s)$.

Nous avons utilisé cette quantité pour faire progressivement diminuer le nombre d'états (jusqu'à un état). Nous comparons ceci à une fusion aléatoire des feuilles. La figure 4.6 illustre

l'évolution des performances au fur et à mesure qu'on enlève des états. On observe que dans trois cas sur quatre, l'utilisation du critère de choix n'amène pas à une baisse significativement plus lente des performances qu'une fusion aléatoire.

4.2.3 Procédé d'apprentissage

Augmenter par spécialisation le nombre d'états permet d'avoir une meilleure précision. Diminuer par généralisation le nombre d'états permet de diminuer la complexité. La dernière forme d'expérimentation que nous appelons apprentissage propose d'utiliser conjointement les deux mises à jour de l'agrégation que nous venons de présenter. En alternant une phase de spécialisation avec une phase de généralisation, nous pouvons faire varier l'agrégation sans changer le nombre d'états du modèle. Sous contrainte de la taille du PDM approché et donc de sa complexité, nous faisons varier l'agrégation en vue d'obtenir une meilleure approximation.

La figure 4.7 illustre l'évolution des performances pour les quatre problèmes. Le nombre d'états ne variant pas, c'est une donnée qu'il faut définir a priori. Les deux PDM de navigation continue sont testés avec 64 et 256 états. Les deux PDM de voiture sur la colline sont eux testés avec 256 et 1024 états. A chaque étape de l'apprentissage, nous avons mis à jour 10% des états. Contrairement aux attentes, on observe que dans la plupart des cas, les performances stagnent voire même diminuent.

4.2.4 Discussion

Les résultats expérimentaux que nous venons de présenter ne sont pas complètement satisfaisants. Il existe plusieurs raisons pour expliquer le manque de performances. En voici deux :

- Discontinuité des fonctions T et R : L'erreur d'interpolation dans un macro-état dépend de la variation des paramètres R et T. Dans les exemples que nous avons illustrés, il y a plusieurs zones de l'espace d'états où ces fonctions admettent des discontinuités. L'existence de ces discontinuités fait qu'il est difficile de diminuer l'erreur d'interpolation locale des états qui les représente. Le processus de diminution de l'erreur fait que ces zones de discontinuité font perdre beaucoup de ressources sans forcément diminuer l'erreur d'approximation. Une façon de remédier à ce problème serait d'utiliser un cadre d'approximation qui gère mieux les problèmes de discontinuité que celui que nous avons utilisé (arbre).
- Mesure pessimiste de l'erreur d'approximation : nous utilisons des bornes supérieures de toutes les erreurs que nous considérons, c'est-à-dire des mesures *pessimistes*. A cause de cette surestimation de l'erreur, certaines zones du vrai espace d'états recquièrent via nos processus d'adaptation de l'agrégation beaucoup de ressources, ressources qui ne sont finalement pas toujours nécessaires pour approcher la vraie fonction de valeur. La section suivante propose des solutions pour remédier à ce deuxième problème.

4.3 Proposition d'heuristiques générales optimistes

Cette section va introduire des modifications légères pour pallier les problèmes liés à l'estimation pessimiste de l'erreur d'approximation. Il est important de noter que ces modifications sont générales, dans le sens où elles ne sont spécifiques ni à un problème particulier, ni à un procédé d'agrégation. Nous allons voir que ces légères modifications permettent d'améliorer significativement les performances. Ceci validera l'idée que le caractère pessimiste de l'estimation jouait un rôle négatif important dans le procédé d'apprentissage que nous venons de présenter.

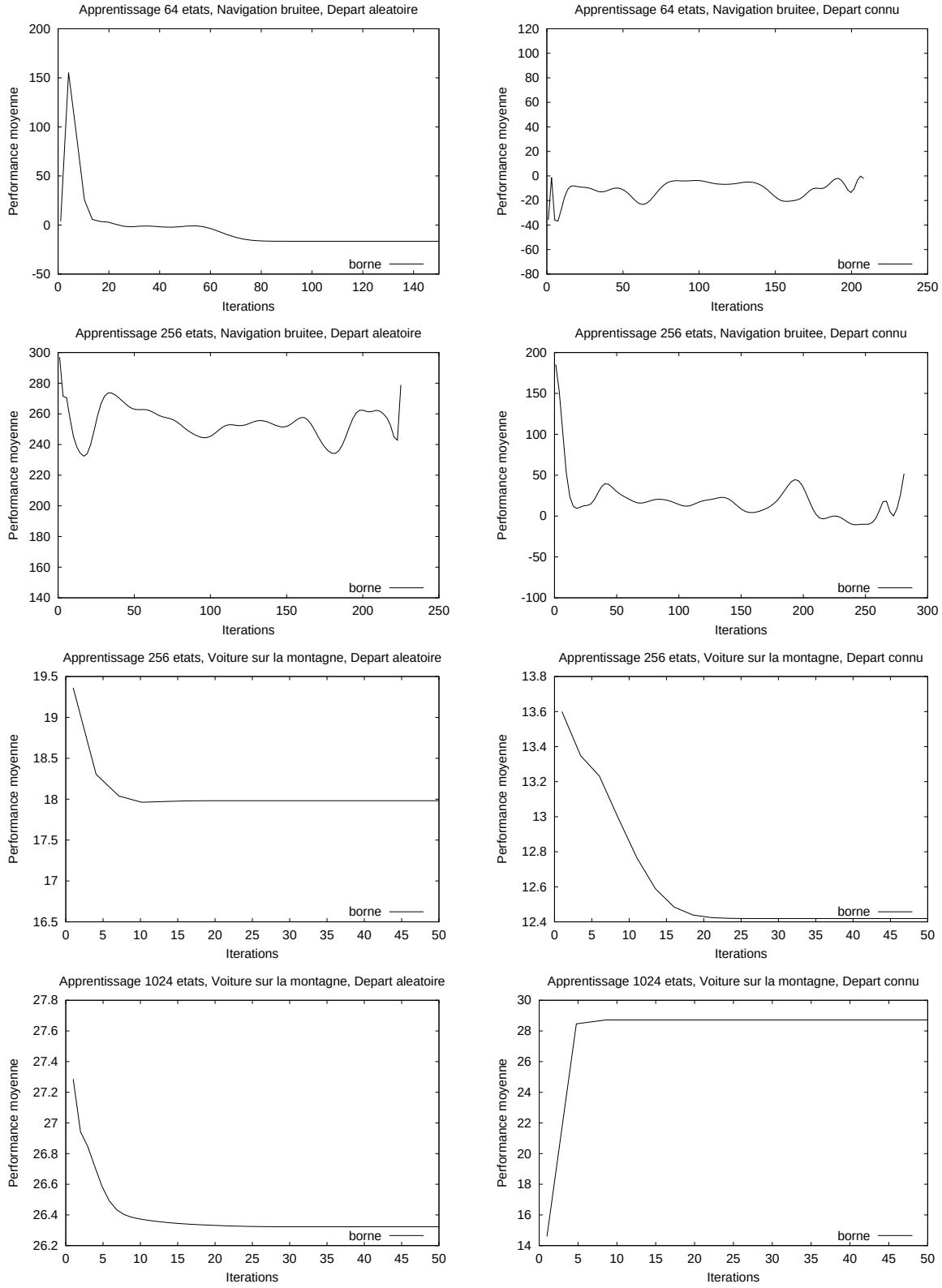

FIG. 4.7: **Evolution des performances pendant le processus d'apprentissage.** Ces graphiques représentent l'évolution des performances quand on utilise successivement les processus de spécialisation et de généralisation décrits précédemment. A chaque itération, on diminue la précision de certaines zones de l'espace d'états et on augmente la précision d'autres zones. Itération après itération, la forme de l'agrégation évolue bien que le nombre d'états reste constant. L'évolution des performances que l'on peut observer globalement est contraire à ce que l'on pouvait espérer (voir le texte et en particulier le paragraphe discussion pour une analyse de ce manque de performances).

En particulier, ceci nous permettra d'obtenir de bonnes performances sans changer de procédé d'agrégation (agrégation sous forme d'arbre).

Nous proposons d'introduire des heuristiques *optimistes* originales qui ont pour objectif de pallier pratiquement le manque de performances relevé précédemment. Nous disons de ces heuristiques qu'elles sont *optimistes* dans la mesure où elles exploitent la politique calculée à partir de l'agrégation courante $\widehat{\pi}^*$ pour dériver des approximations de bornes plus étroites de l'erreur d'approximation. Les nombreux résultats expérimentaux que nous présentons plus loin seront leur principale justification.

Nous allons décrire trois heuristiques. La première concerne la mesure de l'erreur d'interpolation. La deuxième propose une nouvelle manière de définir l'erreur d'approximation globale à partir des erreurs d'approximation locales. La dernière propose une autre mesure de l'erreur d'approximation locale. Ces heuristiques se fondent toutes sur l'idée suivante : améliorer la précision de la politique courante approchée $\widehat{\pi^*}$ est un premier pas pour s'approcher de la vraie politique optimale π^*. Nous mesurons expérimentalement l'intérêt de ces heuristiques en reprenant les expériences que nous avons faites précédemment : spécialisation, généralisation et apprentissage pour les quatre PDM exemples.

4.3.1 Heuristique 1 : Réinjection de l'approximation dans le calcul de l'erreur d'interpolation pour une agrégation

La borne de l'erreur d'interpolation que nous avons calculée précédemment est relativement grossière. En particulier, nous avons utilisé la majoration grossière suivante $\left|V^{\pi^*}\right| < \frac{R_{max}}{1-\gamma}$. Ceci nous donnait :

$$\overline{E_{int}}(\widehat{s_1}) = \overline{\Delta R}(\widehat{s_1}) + \frac{\gamma.R_{max}}{1-\gamma}.\sum_{\widehat{s_2}\in\widehat{S}} \overline{\Delta T}(\widehat{s_1},\widehat{s_2})$$

La première heuristique propose d'utiliser l'estimation courante $\widehat{V}^{\widehat{\pi}^*}$ pour estimer une borne supérieure de l'erreur d'interpolation. L'erreur d'interpolation devient alors :

$$\overline{E_{int}}(\widehat{s_1}) = \overline{\Delta R}(\widehat{s_1}) + \gamma.\sum_{\widehat{s_2}\in\widehat{S}} \overline{\Delta T}(\widehat{s_1},\widehat{s_2}).\widehat{V}^{\widehat{\pi}^*}(\widehat{s_2})$$

4.3.2 Heuristique 2 : Biais de la zone de diminution de l'erreur

Dans la définition de l'erreur d'approximation initiale à partir de l'erreur d'approximation locale (voir équation 3.3 page 62), nous utilisons uniquement les conditions initiales pour pondérer les différentes zones de l'espace d'états. Dans la mesure où le modèle approché nous permet de calculer un plan approximatif $\widehat{\pi^*}$, il nous permet également d'avoir une idée sur la probabilité de présence en chaque état pour $t = 0, 1, 2, etc....$ L'heuristique que nous proposons ci-dessous consiste à donner à chaque zone de l'espace d'états un poids proportionnel à la probabilité de présence sur le long terme et pas uniquement à $t = 0$. Nous proposons de substituer une erreur d'approximation que nous qualifions d'erreur à long terme :

Définition 8 (Erreur d'approximation à long terme)
L'erreur d'approximation à long terme tient compte de la probabilité de présence dans les macro-états dans le futur si on suit la politique optimale approximative $\widehat{\pi^*}$ *:*

$$E_{app}^{(\widehat{\pi}^*)} = \sum_s E_{app}(\widehat{s}).p^{\widehat{\pi}^*}(\widehat{s})$$

où

$$p^{\widehat{\pi}^*}(\widehat{s}) = (1-\gamma).\sum_{t=0}^{\infty}\gamma^t.p(\widehat{s}_t = \widehat{s}|\pi = \widehat{\pi}^*)$$

est la probabilité moyenne pondérée de présence en chaque macro-état $\widehat{s}$ si on suit la politique $\widehat{\pi}^*$.

Une autre façon de présenter cette heuristique est de dire qu'elle consiste à biaiser les zones dans lesquelles on diminue l'erreur d'approximation en utilisant la politique approchée courante.

La probabilité pondérée $p^{\widehat{\pi}^*}$ de présence en chaque état s dans le futur se calcule à l'aide de la notion d'influence pondérée introduite section 3.2.2. Si on note $T_{\widehat{\pi}^*}(\widehat{s},\widehat{s'}) = T(s,\widehat{\pi}^*(\widehat{s}),\widehat{s'})$, les caractéristiques de l'influence pondérée (équation 3.17 page 74) nous permettent de dire :

$$p^{\widehat{\pi}^*}(\widehat{s}) = (1-\gamma).I_{T_{\widehat{\pi}^*}}(\widehat{s}|p^{(0)})$$

4.3.3 Heuristique 3 : Biais du flux d'erreur

Au lieu de proposer de diminuer l'erreur entre la vraie fonction de valeur et la fonction de valeur approchée, la dernière heuristique que nous décrivons propose de commencer par augmenter la précision de la fonction de valeur approchée de la politique optimale approchée courante. Nous nous intéressons en effet à une autre définition de l'erreur d'approximation locale :

Définition 9 (Erreur d'approximation locale optimiste)
L'erreur d'approximation locale optimiste est l'erreur entre la vraie *fonction de valeur de la politique optimale* approximative *et la fonction de valeur* approchée *de la politique optimale* approximative *:*

$$E_{app}(\widehat{s}) = |\widehat{V}^{\widehat{\pi}^*}(\widehat{s}) - V^{\widehat{\pi}^*}(\widehat{s})|$$

Dans ce cas, la relation qui lie l'erreur d'approximation à l'erreur d'interpolation est la suivante :

$$\overline{E_{app}}(\widehat{s}) = \gamma.\sum_{\widehat{s'}}\widehat{T}(\widehat{s},\widehat{\pi^*}(s),\widehat{s'}).\overline{E_{app}}(\widehat{s'}) + \overline{E_{int}}(\widehat{s}) \tag{4.1}$$

L'erreur d'approximation se déduit donc de l'erreur d'interpolation comme la fonction de valeur se déduit à partir de sa politique : par propagation selon la politique. Dans le calcul rigoureux, nous avions la définition suivante (voir équation 3.3 page 77) :

$$\overline{E_{app}}(\widehat{s}) = \gamma.\sum_{\widehat{s'}}\widehat{T}(\widehat{s},\pi_{err}(\widehat{s}),\widehat{s'}).\overline{E_{app}}(\widehat{s'}) + \overline{E_{int}}(\widehat{s})$$

L'erreur d'approximation se déduisait par propagation pessimiste de l'erreur d'interpolation. Cette propagation pessimiste est caractérisée par la politique π_{err}. L'équation 4.1 consiste pour sa part à propager l'erreur d'interpolation selon la politique approximative courante $\widehat{\pi^*}$. Ainsi, une manière équivalente de présenter cette troisième heuristique est de la voir comme un biais du flux d'erreur pour le calcul de l'erreur d'approximation : cela revient en effet à prendre $\pi_{err} = \widehat{\pi^*}$.

4.3.4 Evaluation expérimentale des heuristiques

Dans les pages qui suivent, nous présentons les résultats expérimentaux obtenus pour ces trois heuristiques que nous nommons respectivement $h1$, $h2$ et $h3$. Nous présentons également les résultats que nous avons obtenus avec leur utilisation simultanée : $h1h2h3$.

La figure 4.8 correspond aux expériences de spécialisation. La figure 4.9 présente les résultats

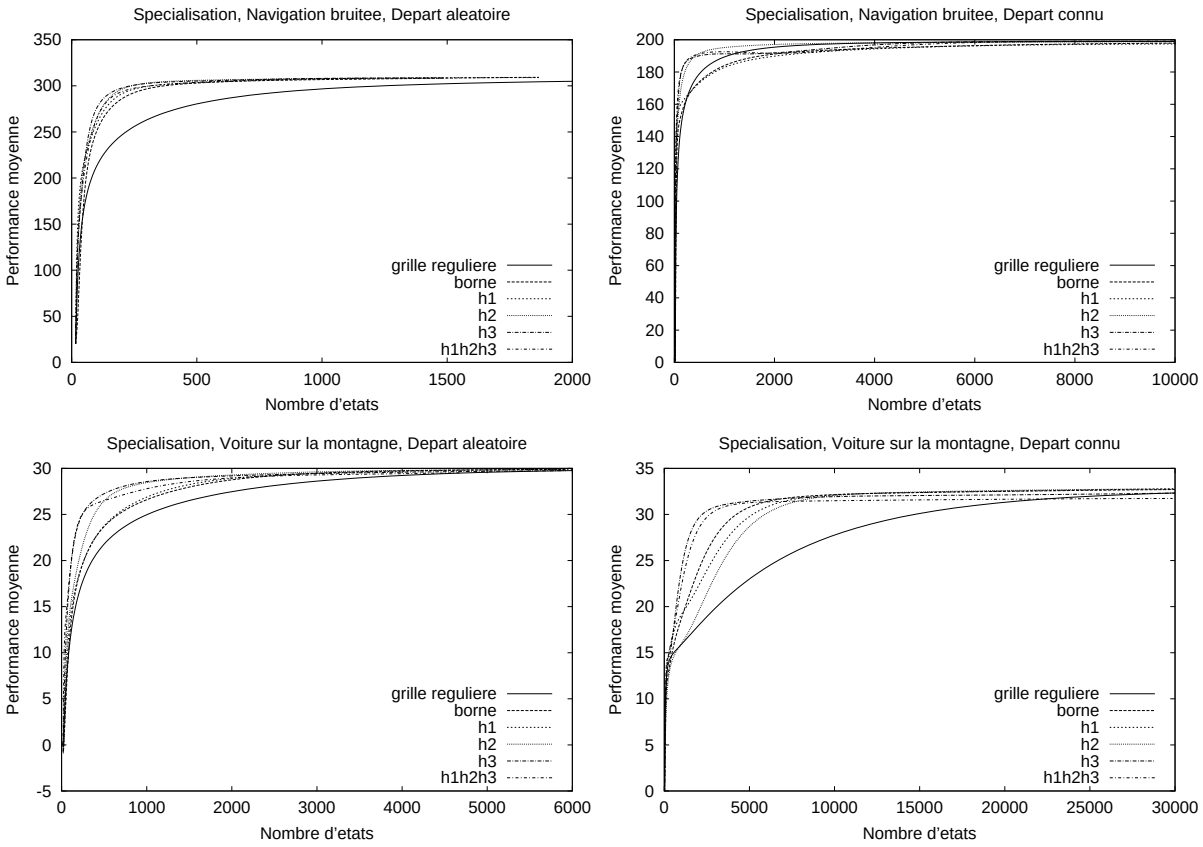

FIG. 4.8: **Evaluation des heuristiques pour le processus de spécialisation.** Voir les commentaires concernant ces courbes dans le texte.

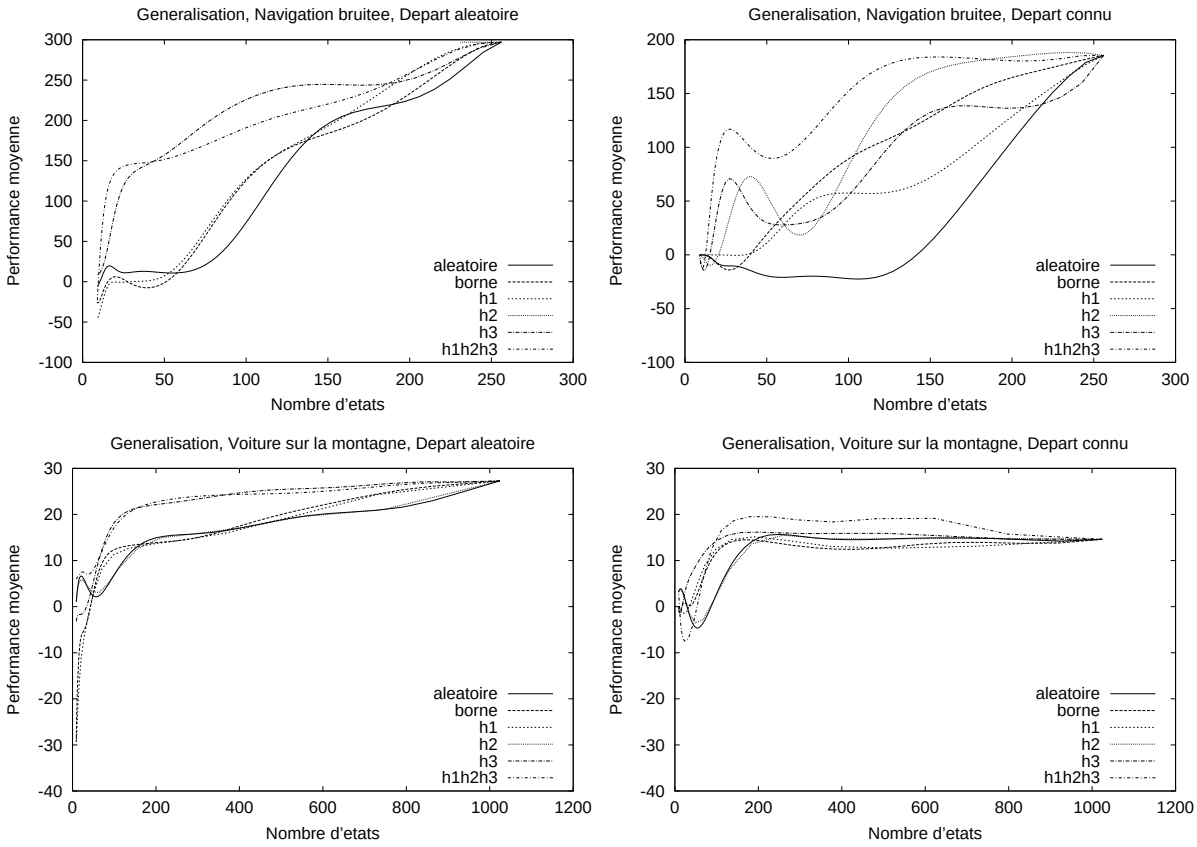

FIG. 4.9: **Evaluation des heuristiques pour le processus de généralisation.** Voir les commentaires concernant ces courbes dans le texte.

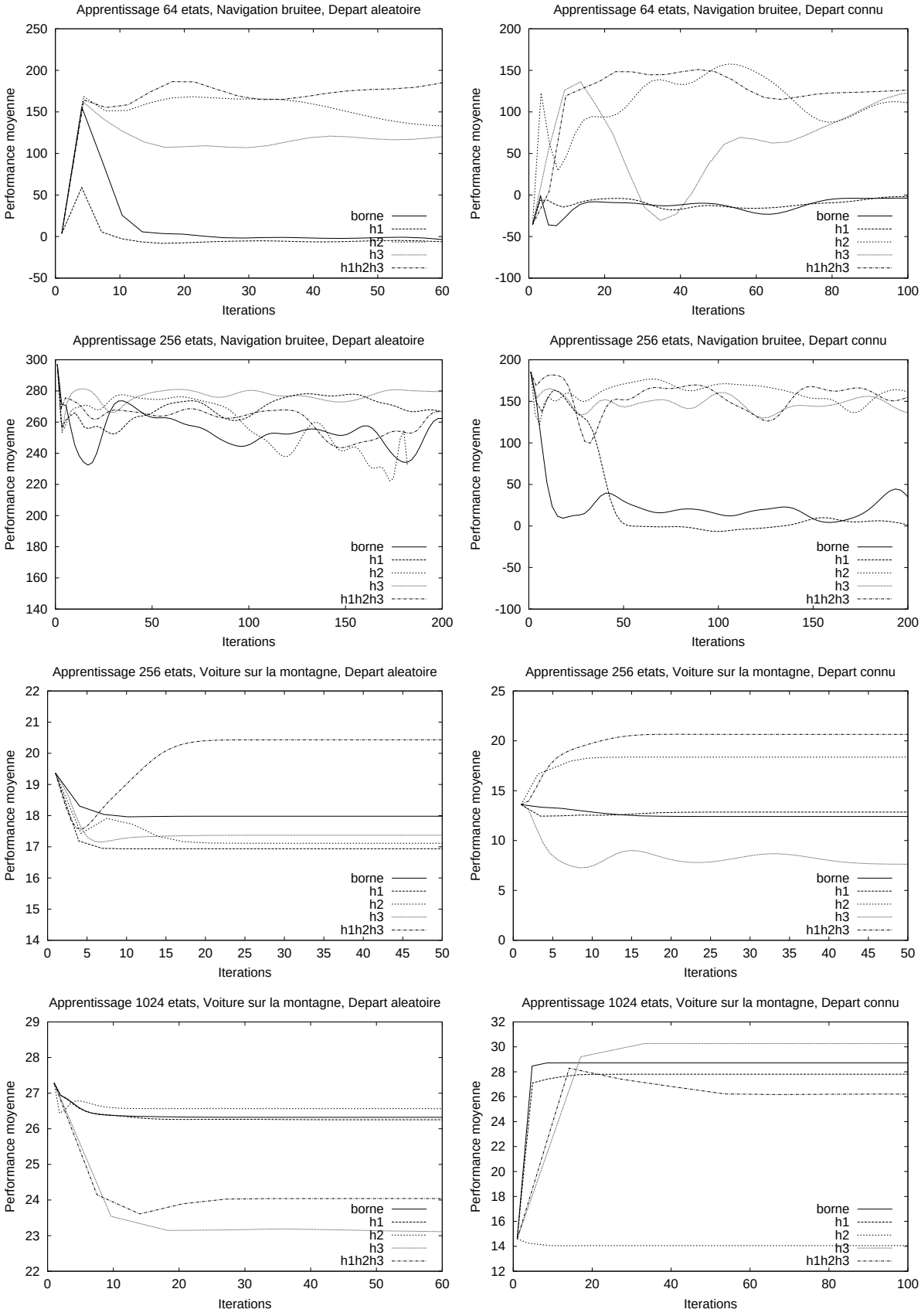

FIG. 4.10: **Evaluation des heuristiques pour le processus d'apprentissage.** Voir les commentaires concernant ces courbes dans le texte.

des expériences de généralisation. Finalement la figure 4.10 montre l'influence des heuristiques pour l'apprentissage. D'une manière générale, les heuristiques amènent une amélioration significative des performances. Afin de comparer leurs capacités respectives, nous proposons de résumer les résultats sous la forme de tableaux.

Spécialisation Le premier tableau correspond aux expériences de spécialisation. Il propose un palmarès des trois meilleures heuristiques pour la spécialisation. Pour effectivement classer les heuristiques, nous disons que la qualité d'une heuristique est d'autant plus grande qu'elle permet d'atteindre 90% des performances optimales rapidement (c'est-à-dire avec le plus petit nombre d'états).

	Navig. départ al.	Navig. départ connu	Voiture départ al.	Voiture départ connu
1^{ere}	h3	h1h2h3	h3	h3
2^{eme}	h2	h3	h2	h1h2h3
3^{eme}	h1h2h3	h2	h1h2h3	-

Il apparaît que c'est l'heuristique $h3$ qui présente globalement les meilleures performances pour la spécialisation. Pour les quatre PDM elle permet d'obtenir les performances optimales significativement plus rapidement que la référence (grille régulière). L'utilisation simultanée des 3 heuristiques donne également de bons résultats.

La figure 4.11 illustre des agrégations obtenues à l'aide de l'heuristique $h3$

Généralisation Considérons à présent les résultats obtenus pour les expériences de généralisation. Nous proposons de nouveau un palmarès des heuristiques : plus une heuristique permet de maintenir longtemps les performances au-dessus des 50% de l'optimal et plus elle est de bonne qualité.

	Navig. départ al.	Navig. départ connu	Voiture départ al.	Voiture départ connu
1^{ere}	h3	h1h2h3	h1h2h3	h3
2^{eme}	h1h2h3	h2	h3	h1h2h3
3^{eme}	h1	h3	h1	h1

Il semble ici que les heuristiques $h1h2h3$ et $h3$ se valent globalement. Si l'on regarde de plus près les graphiques de la figure 4.9, on remarque que ces deux heuristiques sont significativement meilleures que toutes les autres : elles maintiennent les performances beaucoup plus longtemps.

Apprentissage Pour l'apprentissage enfin, l'analyse est un peu plus complexe. On constate globalement que selon les problèmes et le nombre d'états, il y a deux grandes formes de résultats :

- Pour les problèmes { Navigation départ aléatoire 64 états ; Navigation départ connu 64 états ; Voiture sur la colline départ aléatoire 256 états ; Voiture sur la colline départ connu 256 états ; Voiture sur la colline départ connu 1024 états } on observe une ou plusieurs augmentations des performances. Voici un classement des conditions expérimentales selon l'ampleur de l'augmentation :

	Navig. al. 64	Navig. c. 64	Voiture al. 256	Voiture c. 256	Voiture c. 1024
1^{ere}	h1h2h3	h1h2h3	h1h2h3	h1h2h3	h3
2^{eme}	h2	h2	-	h2	$\emptyset$
3^{eme}	h3	h3	-	-	h1

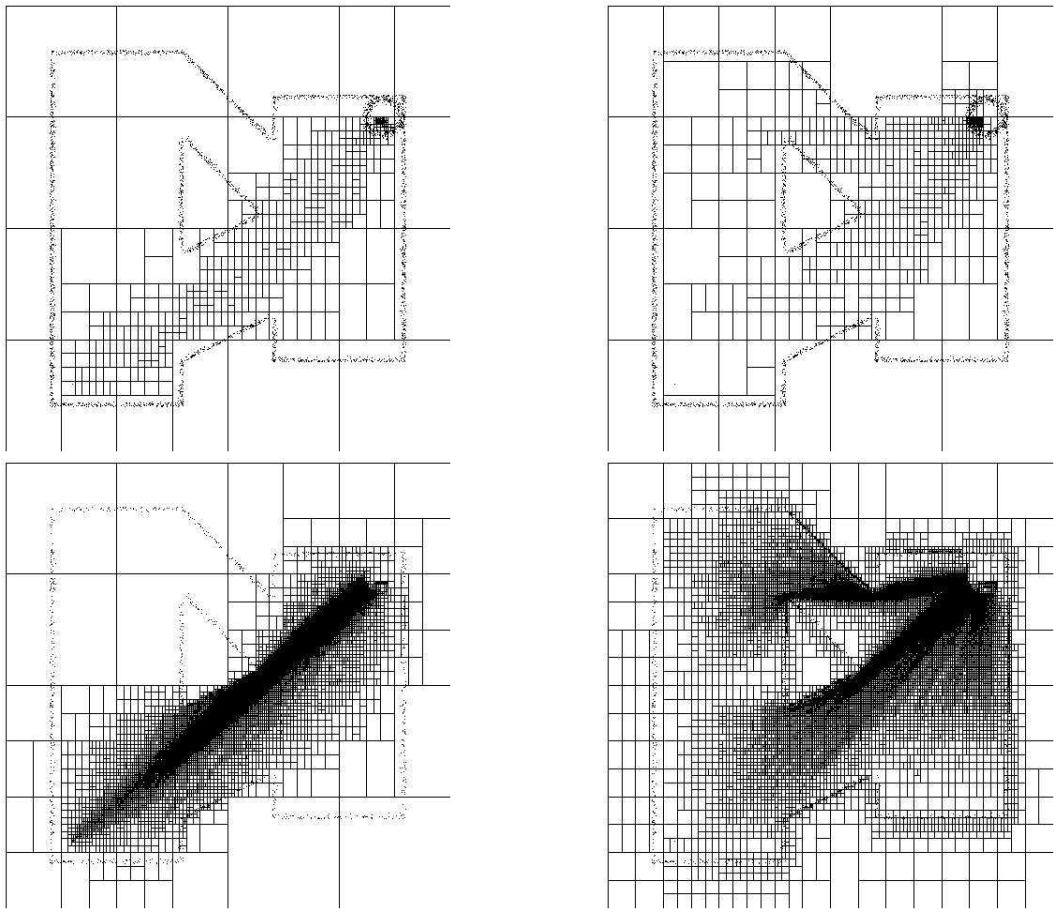

FIG. 4.11: **Exemples d'agrégations obtenues après spécialisation pour le problème "navigation continue".** Nous avons reproduit les agrégations à résolution variable obtenues à l'aide du processus de spécialisation quand on utilise l'heuristique h_3. Les dessins de gauche correspondent à un départ unique (dans le coin bas-gauche), les dessins de droite à un départ aléatoire. Les dessins du haut comporte environ 200 états tandis que ceux du bas en ont un millier. On observe qualitativement que cette spécialisation détaille les zones de l'environnement qui seront effectivement rencontrées par l'agent pendant l'utilisation de la politique optimale. Il est particulièrement intéressant de voir dans le dessin en bas à droite que les zones cruciales (près des murs et des coins) sont très détaillées.

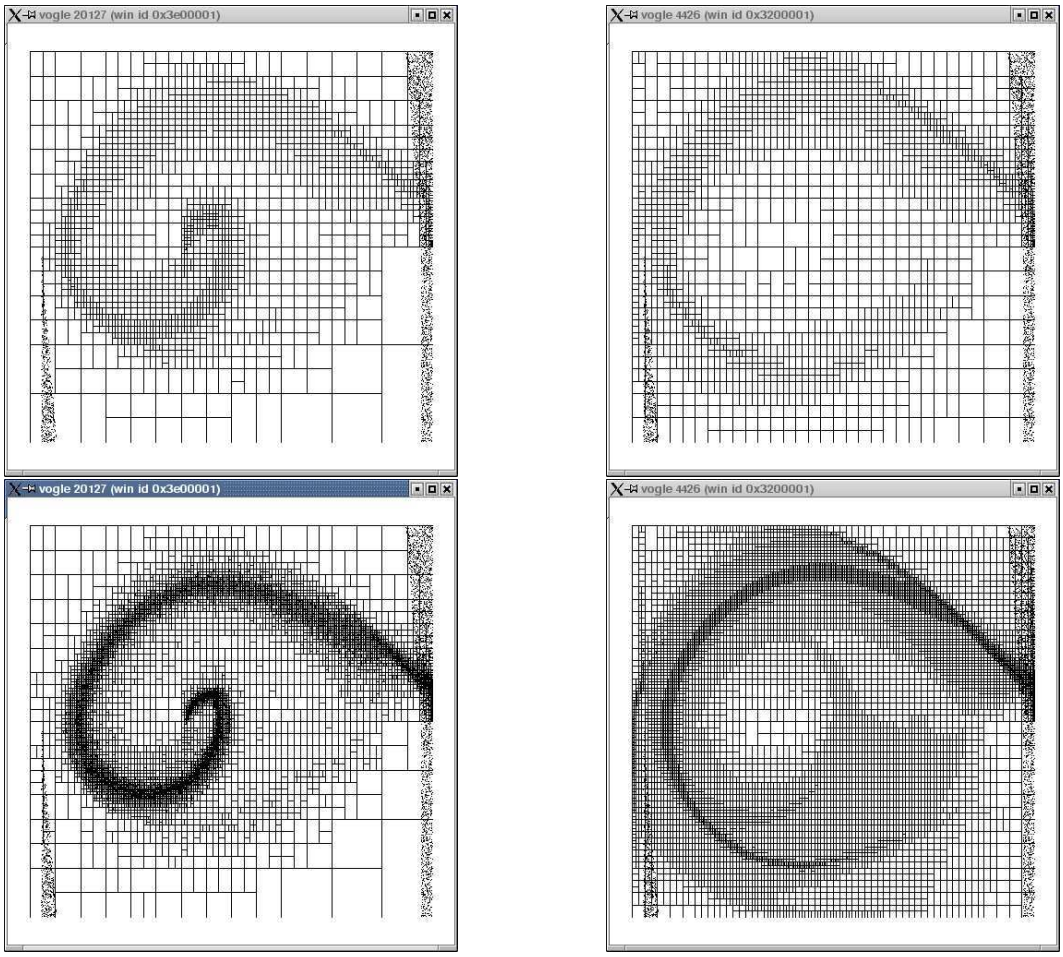

FIG. 4.12: **Exemples d'agrégations obtenues après spécialisation pour le problème "voiture sur la colline".** Ces agrégations à résolution variable sont obtenues à l'aide du processus de spécialisation quand on utilise l'heuristique h_3. Les dessins de gauche correspondent à un départ unique (vitesse nulle et position centrale), les dessins de droite à un départ aléatoire. On observe dans les agrégations à départ unique que la fonction de valeur optimale est particulièrement bien décrite autour de la trajectoire optimale.

– Pour les trois autres problèmes { Navigation départ aléatoire 256 états ; Navigation départ connu 256 états ; Voiture sur la colline départ aléatoire 1024 états }, on n'a pas d'augmentation significative des performances. Ceci s'explique par le fait que l'agrégation initiale est déjà de bonne qualité (les performances commencent près de l'optimal et ne peuvent par conséquent pas augmenter). Sans les heuristiques, nous avons observé que les performances pouvaient chuter. Certaines heuristiques que nous avons introduites permettent de les maintenir. Voici un tableau qui montre les heuristiques qui assurent au mieux ce maintien :

	Navig. al. 256	Navig. c.	Voiture al. 1024
1^{ere}	h3	h2	h2
2^{eme}	h1	h1h2h3	h3
3^{eme}	h1h2h3	h3	h1h2h3

D'une manière générale, les heuristiques $h2$, $h3$ et $h1h2h3$ donnent de bonnes performances pour l'apprentissage. On note néanmoins que c'est l'utilisation simultanée des trois heuristiques $h1h2h3$ qui amène le plus souvent les meilleurs résultats. Les figures 4.13 et 4.14 illustrent des agrégations typiques obtenues par le procédé d'apprentissage en utilisant la combinaison des heuristiques $h1h2h3$.

L'utilisation des heuristiques a permis d'améliorer significativement les performances. La suite de ce chapitre va appronfondir leur étude en considérant des problèmes d'A/R plus complexes.

4.4 Application à des problèmes d'A/R complexes

Pour confirmer et départager l'utilité des heuristiques que nous avons introduites pour le procédé d'apprentissage, nous les avons testées sur deux problèmes plus difficiles. L'ensemble des états du premier des PDM que nous décrivons ci-après s'inscrit dans un espace de dimension 4 (contre 2 pour les précédents). Le deuxième PDM est un problème ayant une dynamique discontinue qui met en jeu un problème d'adaptation de la mémoire des événements passés. Finalement, nous présentons une application de notre approche à un problème de contrôle réel (tiré de la littérature "robotique mobile") concernant la conduite d'une voiture devant atteindre une zone but en contournant un obstacle.

4.4.1 Un problème de navigation par accélération

Considérons le problème de navigation continue que nous avons introduit plus tôt et supposons que les actions ne sont pas des déplacements mais des accélérations. A chaque instant, le robot a une vitesse. Ses actions ne peuvent que moduler cette vitesse afin de le diriger vers la zone but. L'ensemble des états de ce système s'inscrit dans un espace à 4 dimensions : $(x, y, \dot{x}, \dot{y})$ tels que $(x, y) \in [0; 10]^2$ et $(\dot{x}, \dot{y}) \in [-0.05; 0.05]^2$. L'amplitude maximale des accélérations selon les axes x et y est de 0.001. La dynamique de ce problème peut donc être décrite par les équations suivantes :

$$\begin{cases} x \leftarrow x + \dot{x} \\ y \leftarrow y + \dot{y} \\ \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} \leftarrow \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} + 0.001.\overrightarrow{u_{a_t}} + 0.0001.\overrightarrow{b_t} \end{cases}$$

où b_t est un léger bruit sur la commande. Les récompenses sont définies exactement comme dans le problème précédent : on donne au système une récompense 1 si on atteint la zone but,

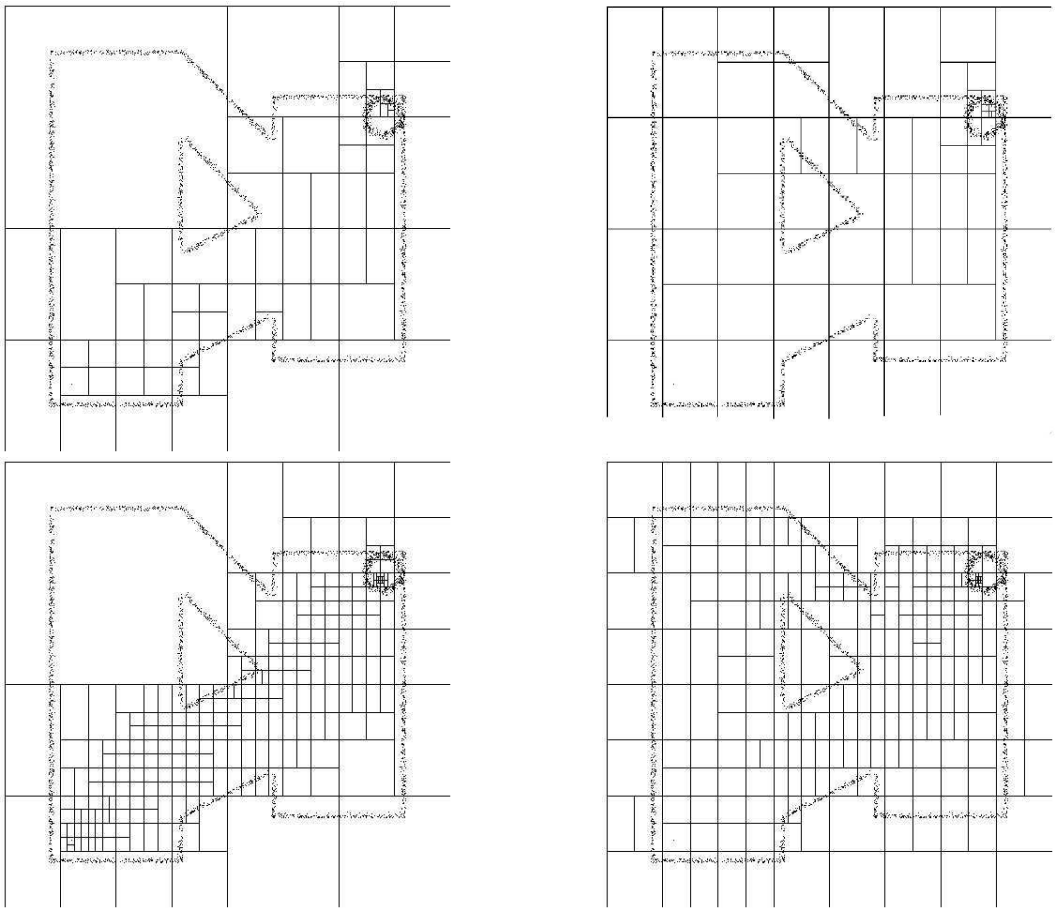

FIG. 4.13: **Exemples d'agrégations obtenues par apprentissage pour le problème "Navigation continue".** Ces dessins représentent les agrégations obtenues lorsqu'on contraint le nombre d'états (64 états en haut et 256 en bas). Les dessins de gauche correspondent à un départ unique, ceux de droite à un départ aléatoire.

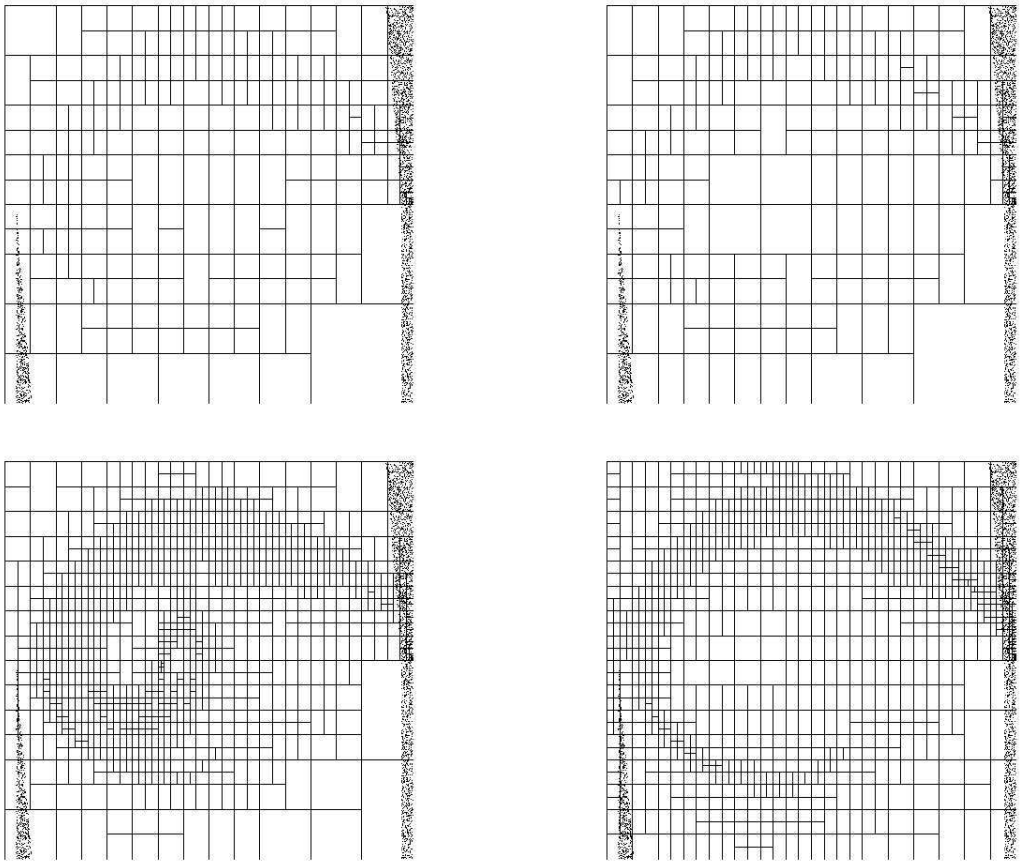

FIG. 4.14: **Exemples d'agrégations obtenues après apprentissage pour le problème "Voiture sur la colline".** Ces dessins représentent les agrégations obtenues lorsqu'on contraint le nombre d'états (256 états en haut et 1024 en bas). Les dessins de gauche correspondent à un départ unique, ceux de droite à un départ aléatoire.

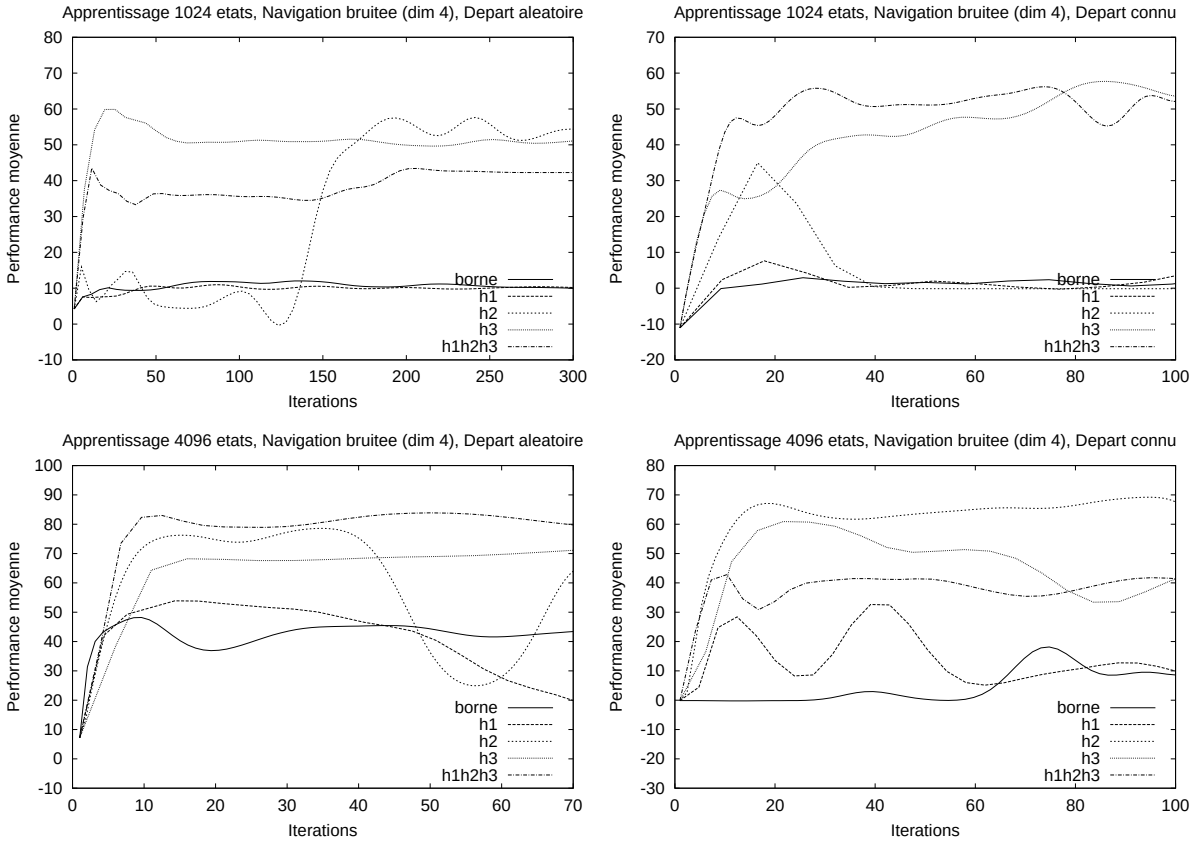

FIG. 4.15: **Evolution des performances pour le procédé d'apprentissage dans le problème "navigation par accélération".**

une récompense -1 lorsque l'on se heurte à un mur et une récompense nulle dans tous les autres cas.

La figure 4.15 présente les résultats de l'utilisation des heuristiques introduites précédemment pour le procédé d'apprentissage :

- pour des approximations ayant 1024 et 4096 états,
- pour un départ aléatoire et pour un départ connu

On remarque ici que l'heuristique qui donne les meilleurs résultats est l'heuristique $h3$: c'est celle qui globalement, donne les meilleurs résultats le plus rapidement et qui les maintient le mieux.

4.4.2 Un problème d'observabilité partielle

Considérons à présent un problème de décision où l'agent ne voit pas directement l'état dans lequel il est. Il fait des observations $o \in \mathcal{O}$ qui peuvent être ambiguës[19]. L'agent est dans un labyrinthe et il ne voit que les murs qui l'entourent. Il doit atteindre une zone but particulière. On donne au système des récompenses qui dépendent de son véritable état qu'il ne peut pas observer. Un exemple simple est illustré à la figure 4.16. Bien qu'il diffère significativement des problèmes de contrôle continu que nous avons abordés jusqu'ici, ce problème peut être vu comme un PDM. Soit $o_t \in O$ l'observation perçue par l'agent à l'instant t. Si on définit l'espace d'états

[19]Ce problème est généralement appelé PDM partiellement observable (PDMPO).

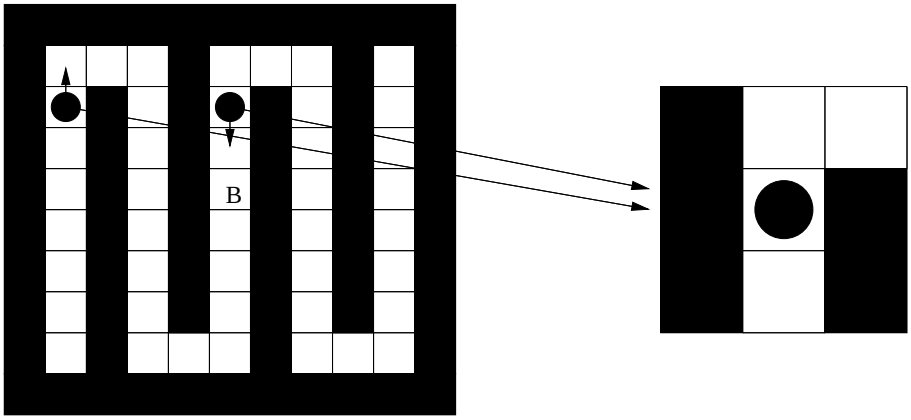

FIG. 4.16: **Un problème d'observabilité partielle.** Un agent est dans un labyrinthe simple. Il peut se déplacer dans les 4 directions cardinales. Il ne perçoit que les 8 cases adjacentes. Il doit se rendre jusqu'à une position but particulière (case B). Certaines positions ambiguës compliquent la tâche de l'agent : selon la vraie position de l'agent, les actions optimales peuvent être différentes.

comme étant l'ensemble des historiques observations/actions :

$$S = \{(o_0, a_0, .., o_n); n \geq 0\}$$

alors, pour toute action a, ces états vérifient la propriété de Markov :

$$P\left((o_0, a_0, .., o_n, a, o_{n+1}|(o_0), (o_0, a_0, o_1), .., (o_0, a_0, .., o_n), a\right) = P((o_0, a_0, .., o_n, a, o_{n+1}|(o_0, a_0, .., o_n), a)$$

Ainsi, le problème qui consiste à agir en se basant sur un ensemble fini d'observations $\mathcal{O}$ est dans le cas général un PDM sur l'ensemble (infini et dénombrable) des historiques observations/actions.

Nous proposons d'appliquer au problème de navigation illustré à la figure 4.16 l'approche de l'agrégation en considérant que les macro-états sont des mémoires finies des dernières observations/actions. Par exemple on note $(.., a, o)$ le macro-état qui rassemble tous les historiques qui ont donné l'observation o après que l'action a a été effectuée. On représente l'ensemble des macro-états d'une telle approximation sous la forme d'arbre $n-aires$. En haut de l'arbre se situe une racine. Sous la racine se situent les dernières observations faites. Les fils des observations sont les actions qui ont eu lieu juste avant. De même, les fils des actions sont les observations faites précédemment. La figure 4.17 donne des exemples simples d'arbres dans un cas où il n'y a que 2 observations et 2 actions. Plus on descend dans l'arbre et plus les informations ont eu lieu il y a longtemps. On remarque qu'il y a bijection entre les feuilles de cet arbre et les macro-états du PDM approximatif : si on part d'une feuille et si on remonte jusqu'à la racine, on peut lire les historiques observations/actions qui servent de base au PDM approximatif. La figure 4.17 illustre également comment se transposent les procédés de spécialisation et de généralisation introduits dans les problèmes de contrôle précédents. Faire une spécialisation consiste à développer les deux fils d'une feuille de l'arbre. Faire une généralisation consiste à supprimer deux feuilles de cet arbre.

Le problème de l'apprentissage est le suivant : étant donné un nombre de ressources fixé a priori pour décrire les historiques (un nombre de feuilles contraint pour l'arbre), faire varier l'ensemble des historiques (l'arbre) sur lequel se base l'agent de sorte à améliorer les performances.

Nous avons utilisé les trois heuristiques qui donnaient les meilleurs résultats dans les expériences précédentes ($h2$, $h3$, $h2h3$) et les avons testées pour le problème d'apprentissage avec

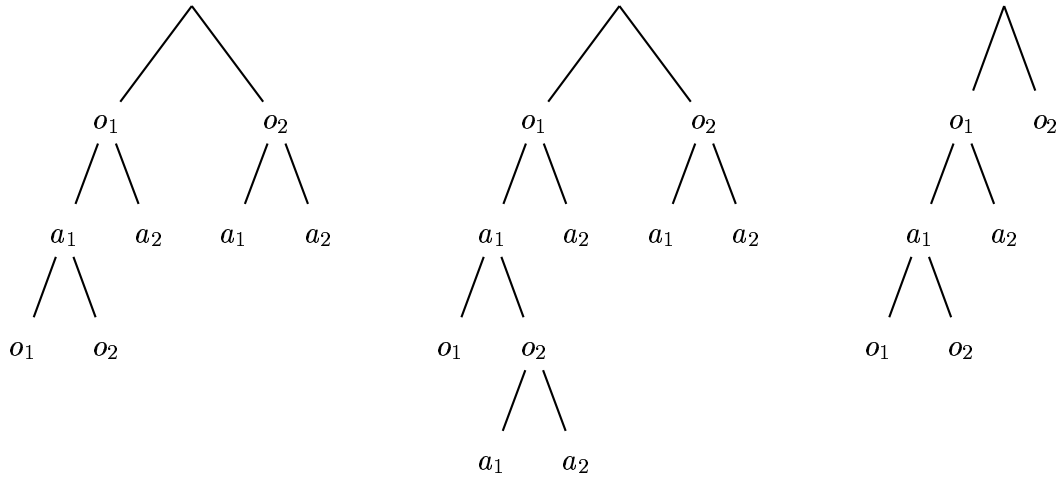

FIG. 4.17: **Représentation d'un ensemble d'historiques observations/actions par des arbres.** L'arbre de gauche décrit les 5 macro-états d'un PDM approximatif : $(..., o_1, a_1, o_1)$, $(..., o_2, a_1, o_1)$, $(..., a_2, o_1)$, $(..., a_1, o_2)$ et $(..., a_2, o_2)$. On passe de cet arbre à l'arbre du centre en faisant l'équivalent d'une spécialisation : on différencie les deux historiques $(..., a_1, o_2, a_1, o_1)$ et $(..., a_2, o_2, a_1, o_1)$ qui étaient auparavant assimilés à un seul historique $(..., o_2, a_1, o_1)$. On passe de l'arbre de gauche à l'arbre de droite par généralisation : on identifie alors ces deux historiques $(..., a_1, o_2)$ et $(..., a_2, o_2)$ qui deviennent $(..., o_2)$.

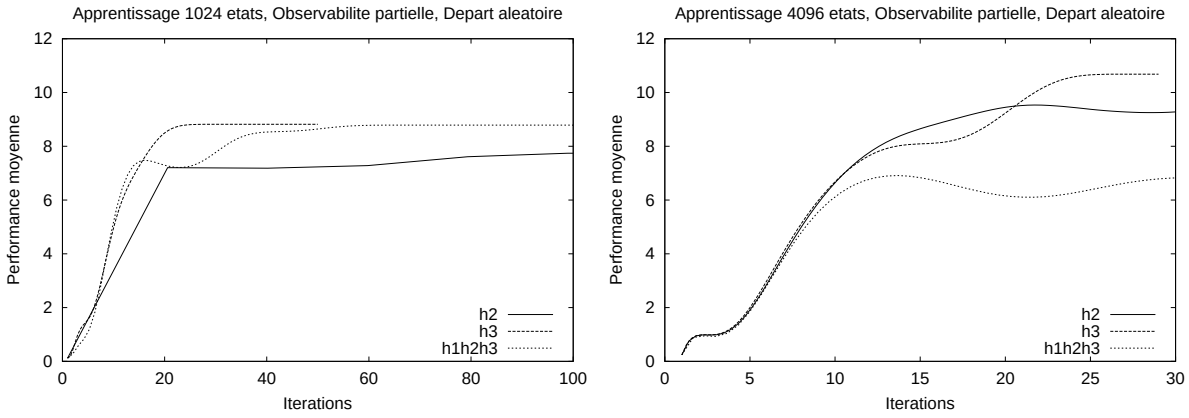

FIG. 4.18: **Evolution des performances pour le procédé d'apprentissage dans le problème "observabilité partielle".**

1024 et 4096 macro-états. La figure 4.18 présente l'évolution des performances. On observe que l'heuristique $h3$ donne ici encore les meilleures performances.

4.4.3 Conduite d'un engin de type voiture

Afin de valider l'efficacité de notre approche, et en particulier celle de l'heuristique h_3, nous présentons une dernière application sur un problème de contrôle. Contrairement aux exemples précédents, que l'on peut considérer comme des problèmes "jouets", ce dernier est tiré de la littérature concernant la robotique mobile autonome. Il s'agit d'un problème de planification de chemins pour un robot mobile de type voiture. Le modèle que nous allons détailler est particulièrement étudié dans [Scheuer, 1998].

Nous commençons par décrire les caractéristiques cinématiques du véhicule ainsi modélisé. Ensuite, nous l'intégrons dans un problème de navigation nécessitant le contournement d'un obstacle.

Description des contraintes cinématiques

La présentation que nous faisons ici est pour une grande part tirée de [Scheuer, 1998]. Considérons un robot mobile de type voiture se déplaçant sur un terrain plat horizontal. L'état du robot est caractérisé les paramètres suivants $(x, y, \theta, \kappa, v, a)$ (voir la figure 4.19) :

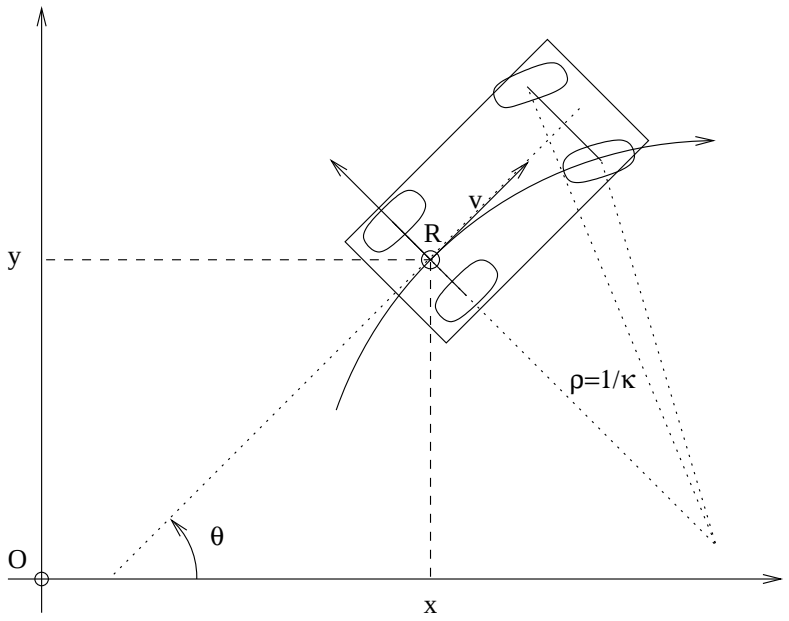

FIG. 4.19: **Un robot mobile de type voiture.** Voir la description de la dynamique dans le texte

- x et y sont les coordonnées d'un point de référence R du véhicule (le milieu de l'axe des roues arrières)
- θ est l'orientation du véhicule, c'est-à-dire l'angle par rapport à l'axe orienté $[Ox)$
- κ est la courbure instantanée (c'est-à-dire l'inverse de son rayon de braquage instantané)
- v est la vitesse du véhicule ;

– a est l'accélération du véhicule[20] ;

Si on suppose que ce véhicule ne dérape pas, on peut dériver les lois de la cinématique suivantes (voir [Scheuer, 1998]) :

$$\begin{cases} \frac{\partial x}{\partial t} &= v.\cos\theta \\ \frac{\partial y}{\partial t} &= v.\sin\theta \\ \frac{\partial v}{\partial t} &= a \\ \frac{\partial \theta}{\partial t} &= v.\kappa \end{cases} \tag{4.2}$$

avec les contraintes suivantes

$$\begin{cases} |\kappa| \leq \kappa_{max} \\ \frac{\partial \kappa}{\partial t} \leq \sigma_{max} \\ v \in [-2;5] \\ a \in [-1;2.5] \end{cases} \tag{4.3}$$

$\kappa_{max} = 0.2$ est une contrainte sur la courbure : elle empêche que les roues de la voiture tournent d'une manière irréaliste. $\sigma_{max} = 0.2$ est une borne supérieure de l'orientation des roues : elle empêche que la voiture ne braque trop brutalement.

Mise sous forme PDM

Nous utilisons le cadre que nous venons de décrire pour proposer un problème d'A/R. Considérons que le véhicule que nous venons de décrire s'inscrit dans un environnement de taille 100×100 contenant une zone "but", quatre murs latéraux et un mur à contourner (voir figure 4.20). Nous définissons l'espace d'états comme étant l'ensemble des valeurs possibles $(x, y, \theta, \kappa, v)$ avec $(x, y) \in [0; 100]$, $\theta \in [0; 2\pi]$, $\kappa \in [-0.2; 0.2]$ et $v \in [-2; 5]$. Le contrôle s'applique sur l'accélération ($a \in \{-1; 0; +2.5\}$) et sur la rotation des roues ($\Delta\kappa \in \{-0.2; 0; +0.2\}$. Ceci définit $3 \times 3 = 9$ actions. Nous approximons la dynamique décrite plus haut (équations 4.2) en l'intégrant sur un instant de temps ($dt = 1$) :

$$\begin{cases} x_{t+1} &\leftarrow x_t + v_t.\cos\theta_t \\ y_{t+1} &\leftarrow y_t + v_t.\sin\theta_t \\ v_{t+1} &\leftarrow v_t + a \\ \theta_{t+1} &\leftarrow v.\kappa_t \\ \kappa_{t+1} &\leftarrow \kappa_t + \Delta\kappa \end{cases} \tag{4.4}$$

Finalement, nous définissons la fonction récompense comme suit : $R(s, a)$ vaut $+1$ si s appartient à la zone but. $R(s, a)$ vaut -1 si l'action a amène la voiture à se cogner contre l'un des murs. Pour toutes les autres situations, $R(s, a)$ vaut 0.

Nous avons appliqué la technique d'apprentissage décrite dans ce chapitre en considérant deux types de conditions initiales :

– une position initiale connue (voir figure 4.20)

– une position initiale aléatoire

Nous avons contraint le nombre d'états de ces deux PDM à 4096 états. Nous sommes partis d'un découpage régulier de l'espace d'états. A chaque itération, l'apprentissage avec l'utilisation de l'heuristique h_3 nous a permis de faire évoluer l'agrégation. Nous avons mesuré les performances pour chaque itération comme précédemment : en lançant 500 trajectoires. L'évolution des performances est illustrée à la figure 4.21. On y observe que bien que le nombre d'états du PDM reste constant, la mise à jour de l'agrégation permet d'améliorer les performances. La figure 4.22 montre l'évolution des trajectoires pour la planification avec position initiale aléatoire. On observe l'amélioration des courbes suivies par le véhicule à partir de 10 positions initiales.

[20] L'accélération ne faisait pas partie du modèle décrit dans [Scheuer, 1998].

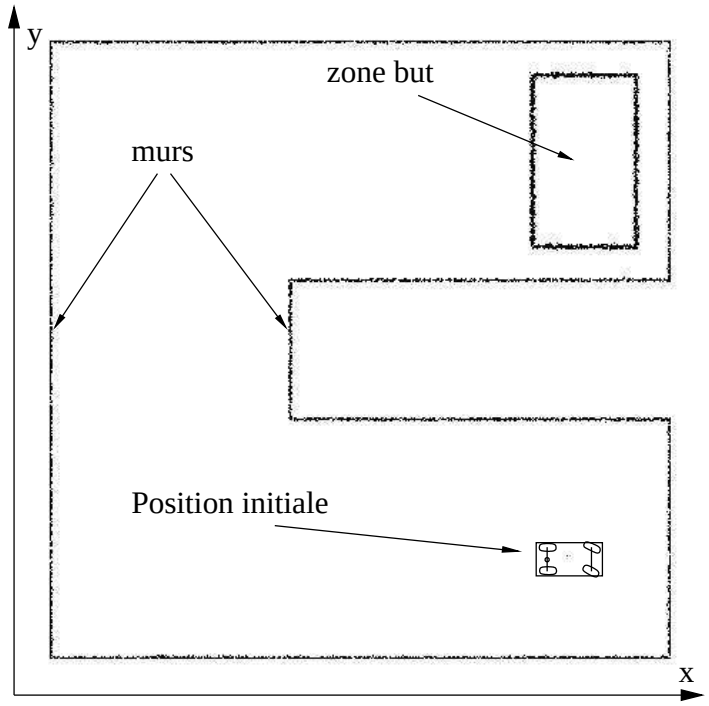

FIG. 4.20: **Environnement du problème de conduite d'une voiture.** Un engin de type voiture doit atteindre une zone but en évitant de heurter les murs et en contournant un obstacle.

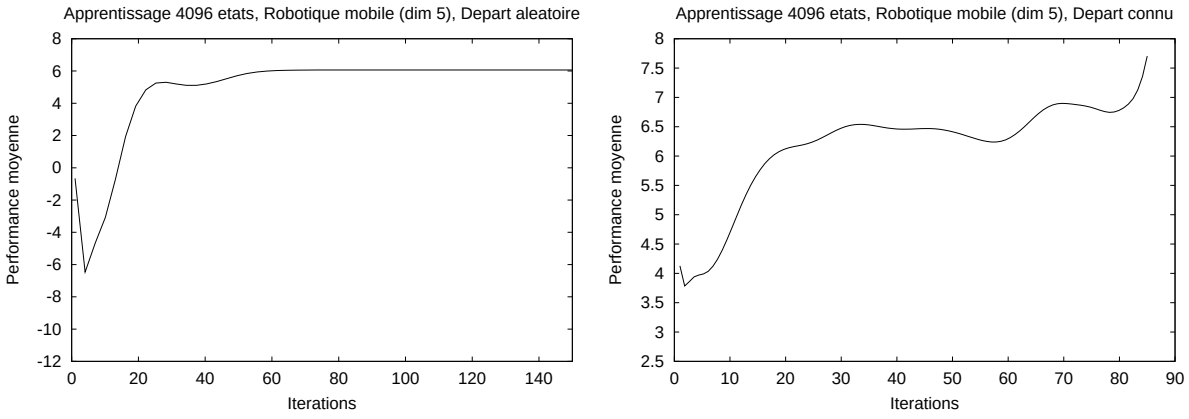

FIG. 4.21: **Evolution des performances pour le procédé d'apprentissage dans le problème "conduite d'une voiture".**

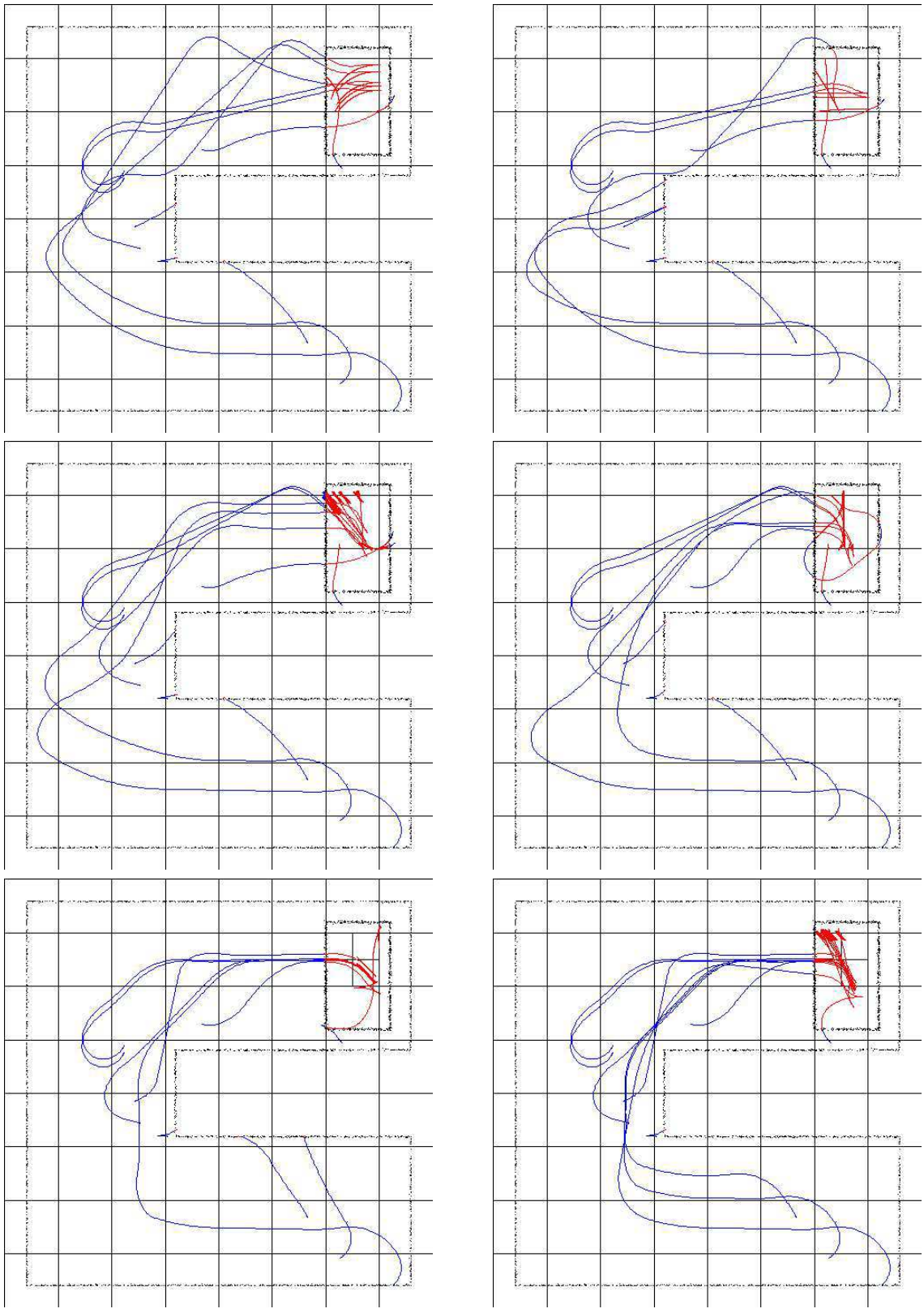

FIG. 4.22: **Evolution de dix trajectoires calculées pendant le procédé d'apprentissage.** Nous avons choisi aléatoirement dix conditions initiales (position, orientations) ayant une vitesse nulle. Pour chacune de ces positions, et pour des itérations successives du procédé d'apprentissage, nous représentons une trajectoire induite par le suivi de la politique optimale courante. On observe visuellement que la planification s'améliore : les trajectoires sont de plus en plus courtes et le véhicule se cogne de moins en moins dans les murs.

La méthode d'approximation itérative que nous avons proposée, et en particulier l'utilisation de l'heuristique h_3, est par conséquent applicable à un problème difficile comme le contrôle d'un véhicule de type voiture. Notre objectif n'était pas tant ici de proposer une alternative aux approches propres à la robotique mobile, mais plutôt de montrer que notre approche, au demeurant très générale, avait un potentiel intéressant pour des problèmes pointus comme la robotique mobile.

Résumé du chapitre

Nous avons mis en œuvre la démarche introduite dans le chapitre précédent pour des problèmes simulés relativement complexes (à notre connaissance il n'existe pas de mise en œuvre de la méthodologie introduite par [Munos et Moore, 2000] publiée dans la littérature).

Les premiers résultats nous ont amené à reconsidérer l'utilisation d'erreurs pessimistes de l'approximation. Nous avons alors proposé plusieurs heuristiques optimistes qui améliorent pratiquement les performances. L'une d'entre elles (h_3), équivalente à un biais de la propagation d'erreur d'interpolation pour le calcul de l'erreur d'approximation, a produit de bons résultats pour l'ensemble des problèmes que nous avons considérés.

L'utilisation d'une approche unique pour résoudre les multiples problèmes que nous avons présentés suggère que notre approche est générique. Elle s'applique à des problèmes d'A/R variés, allant de problèmes de navigation simplifiée à des problèmes plus réalistes comme la planification de trajectoires pour un véhicule de type voiture, en passant par un problème d'observabilité partielle mettant en jeu la gestion d'une mémoire des événements passés.

Conclusion

Nous avons présenté une méthodologie générale pour mesurer l'erreur liée à l'utilisation d'une approximation par agrégation pour un problème d'A/R. Elle implique de différencier deux sortes d'erreur : l'erreur d'interpolation mesure l'approximation des paramètres du PDM agrégé tandis que l'erreur d'approximation quantifie l'erreur effectivement commise sur la fonction de valeur optimale. Dans le cas agrégation, le calcul de l'erreur d'approximation est d'une complexité équivalente à celle du calcul de la fonction de valeur optimale dans le modèle agrégé. L'erreur d'approximation se déduit par propagation de l'erreur d'interpolation.

Nous avons introduit le concept d'influence pondérée. Il permet de déterminer l'importance relative des zones de l'espace d'états pour le calcul de la fonction de valeur optimale. Ceci permet en théorie de prédire comment il faudrait faire évoluer l'agrégation afin d'améliorer l'approximation. Nous avons exploité pratiquement ces informations et proposé plusieurs méthodes pour améliorer l'agrégation d'un PDM complexe.

Dans les exemples que nous avons présentés, nous avons utilisé une représentation d'agrégation sous forme d'arbre ; les processus qui permettaient de faire évoluer l'agrégation étaient l'ajout ou le retrait de feuilles dans l'arbre. Il est important de souligner que la méthode utilisée pour diminuer l'erreur d'approximation n'est pas spécifique à l'utilisation d'arbres. Pour la mettre en pratique, la seule chose dont on ait besoin est un processus de modification progressive de l'agrégation d'un modèle. L'étude théorique que nous avons présentée permet de faire le lien entre la modification de l'agrégation et la variation attendue des performances.

La technique générale consistant à faire évoluer l'agrégation d'un PDM approximatif est applicable dans de nombreux problèmes. Nous l'avons illustrée sur plusieurs problèmes de contrôle (en particulier un problème de conduite d'une voiture inspiré de la robotique mobile) et un problème d'observabilité partielle. Les premiers résultats expérimentaux nous ont poussés à proposer trois heuristiques originales. Si elles ont toutes amené une augmentation des performances, l'une d'elle, qui consiste en un biais de la propagation de l'erreur d'interpolation pour le calcul de l'erreur d'approximation, a donné de bons résultats sur l'ensemble des problèmes testés.

Pour finir, remettons cette partie en perspective par rapport à la précédente. Quand un problème d'A/R s'inscrit dans un espace d'états de très grande taille, nous pouvons appliquer la méthode d'agrégation automatique décrite dans cette partie pour en avoir une approximation de taille et de qualité raisonnables. En particulier, si nous imposons a priori le nombre d'états (ce qui revient à la fois à contraindre la quantité des ressources et la complexité temporelle du modèle), le procédé d'évolution de l'agrégation que nous avons appelé "apprentissage" permet d'itérativement améliorer les performances.

Dans la partie précédente, nous avons insisté sur la possibilité d'aborder l'A/R de manière connexionniste. Les calculs impliqués dans le processus de mise à jour de l'agrégation s'appuient sur des équations de type Bellman identiques (ou analogues pour l'influence) à celle qui permet de calculer la fonction de valeur d'un PDM fini. Les calculs pour l'erreur d'approximation et pour l'influence sont des processus itératifs qui impliquent une contraction. Par conséquent,

ils sont tout autant parallélisables [Bertsekas et Tsitsiklis, 1989] et tolérants aux fautes que l'algorithme *Value Iteration*. La fonctionnalité qui consiste à mettre à jour la représentation peut donc s'intégrer dans l'architecture que nous avons décrite en détail dans la première partie. Un exemple d'intégration d'un processus de mise à jour de l'agrégation dans une architecture connexionniste comme celle décrite dans la première partie peut être consulté dans un article publié récemment [Scherrer, 2002].

Troisième partie

Auto-organisation modulaire

Introduction

> « Selon la tâche à laquelle l'individu sera confronté, c'est l'hémisphère le plus compétent qui prendra momentanément le contrôle des opérations et sera donc temporairement "dominant", mais que change la nature de la tâche, et la dominance pourra échoir à l'autre hémisphère. »
>
> *Deux hémisphères, un cerveau, Jean-Louis Juan de Mendoza*

Lorsqu'on est confronté à un problème d'A/R de grande taille, l'utilisation d'une approximation par agrégation permet de diminuer significativement la complexité de résolution. La partie précédente nous a montré comment nous pouvions choisir l'agrégation de sorte à ne pas trop pénaliser la qualité de l'approximation. Le processus itératif que nous avons introduit nous permet de calculer une représentation adaptée au problème considéré. Il peut être pratiquement envisageable de supposer qu'une architecture apprenant par renforcement n'a pas seulement *un* problème d'A/R à résoudre mais *plusieurs*. Par exemple, considérons un véhicule de type voiture comme celui que nous avons présenté dans la partie précédente. Selon les circonstances, un tel véhicule pourrait avoir à accomplir différentes tâches : rallier un lieu de rendez-vous, aller à la pompe à essence, aller au garage, etc...

Nous avons vu dans la partie précédente qu'un bon moyen pour être efficace dans une tâche consiste à construire une représentation qui lui est adaptée. Une première solution pour aborder la multiplicité des problèmes serait de construire autant de représentations qu'il y a de problèmes. Néanmoins une telle approche deviendrait impraticable si le nombre de problèmes à résoudre est grand. Par exemple la mémoire nécessaire pour stocker toutes les représentations pourrait poser problème. L'idée que nous proposons dans cette dernière partie est la suivante. Si plusieurs problèmes d'A/R partagent le même espace d'états S, *une* agrégation de cet espace peut être candidate à la résolution de *plusieurs* problèmes. Si la pompe à essence et le garage sont dans des lieux géographiques proches, la voiture pourrait utiliser une seule carte pour se rendre dans chacun d'eux.

Dans la première partie de ce mémoire, nous exploitions déjà cette idée en utilisant un même espace d'états pour calculer les politiques optimales associées à plusieurs fonctions récompenses. Nous utilisions une représentation pour répondre à trois tâches (voir section 1.4). Nous allons dans cette partie généraliser cette approche : nous allons décrire une méthode pour approximer

n problèmes d'A/R à l'aide de $m < n$ approximations par agrégation. Ces m approximations peuvent être interprétées comme des modules spécialisés et indépendants. Ces m approximations sont *différentes représentations du même espace d'états* S. Lorsque l'un des n problèmes est donné au système, tous les modules peuvent essayer de le résoudre en parallèle. L'un d'entre eux aura la représentation la plus adaptée (comme précédemment, nous supposerons que la représentation est d'autant plus adaptée qu'elle induit une petite erreur sur le calcul de la fonction de valeur optimale). C'est ce module qui sera finalement chargé de la tâche.

La problématique que nous abordons précisément dans cette partie concerne l'organisation automatique des modules, ce que nous appelons encore l'*auto-organisation modulaire*. Si le nombre n de tâches est très grand par rapport au nombre m de modules, il paraît évident que la répartition des n tâches sur les m modules sera cruciale. En particulier, il est naturel d'imaginer que le fonctionnement global du système sera d'autant plus efficace que la répartition des tâches sur les modules est adéquate. La répartition automatique des n problèmes sur les m modules est le problème que nous formalisons et traitons dans cette partie.

Il est assez facile de formaliser le problème de cette partie, et de montrer comment il généralise celui abordé dans la partie précédente. Soit un PDM $\mathcal{M} = \langle S, A, T, R \rangle$ où S est de grande taille. Dans la partie précédente, nous avons présenté une méthode qui permettait de trouver un PDM approché par agrégation. Nous nous intéressions alors au problème de trouver l'agrégation $\widehat{S}^*$ qui impliquait la plus petite erreur d'approximation :

$$\widehat{S}^* = \arg\min_{\widehat{S}} E_{app}^{\mathcal{M}}(\widehat{S})$$

Soient maintenant n PDM $(\mathcal{M} = \langle S, A, T_i, R_i \rangle)_{(1 \leq i \leq n)}$ définis sur le même espace d'états S de grande taille. Nous cherchons à trouver m agrégations de l'espace d'états S qui minimisent l'erreur commise sur l'ensemble des n PDM :

$$(\widehat{S}_1^*, .., \widehat{S}_m^*) = \arg\min_{(\widehat{S}_1, .., \widehat{S}_m)} \left(\sum_{i=1}^{n} \min_{1 \leq j \leq m} E_{app}^{\mathcal{M}_i}(\widehat{S}_j) \right)$$

Si on avait $m = n$, le problème serait facile, il suffirait d'associer le i^{eme} PDM approché $\widehat{\mathcal{M}_i}$ au i^{eme} PDM $\mathcal{M}_i$. Nous considérons donc le cas $m < n$.

La contribution essentielle est ici de montrer que le problème de l'auto-organisation modulaire peut se formaliser dans le cadre de la *catégorisation par noyaux*. L'auto-organisation modulaire est alors un processus de classification de PDM qui utilise les techniques d'adaptation de la représentation introduites dans la partie précédente. Ainsi, cette troisième et dernière partie est une application de la partie précédente à un problème de catégorisation. N'apportant pas de développements techniques nouveaux, cette troisième partie sera plus courte que les précédentes. Néanmoins, l'angle original sous lequel elle présente un problème d'A/R a priori difficile nous a poussé à en faire une partie indépendante des autres.

Notre propos s'articule en deux chapitres :

- Nous introduisons le thème de la catégorisation, en commençant par son cas le plus simple : la quantification vectorielle. Nous abordons ensuite le cas plus général de la catégorisation par noyaux. En analogie avec la quantification vectorielle, nous introduisons une version adaptative originale de l'algorithme des nuées dynamiques.
- Nous formalisons ensuite le problème qui nous intéresse, l'auto-organisation modulaire, dans le cadre de la catégorisation par noyaux. Nous montrons que sa résolution peut se faire en réutilisant la méthode d'apprentissage de représentation que nous avons introduite dans la partie précédente. Nous illustrons l'auto-organisation modulaire sur une adaptation du problème de navigation continue.

Chapitre 5

Catégorisation par quantificateurs et noyaux

L'objet de ce chapitre est d'introduire quelques éléments concernant la problématique de la catégorisation. Le lecteur qui serait déjà familier avec la catégorisation par noyaux et l'algorithme des nuées dynamiques pourra directement se rendre à la section 5.2.3 qui constitue (à notre connaissance) le point essentiel de ce chapitre : nous y introduisons une version adaptative de l'algorithme des nuées dynamiques qui sera utilisée pour résoudre le problème de l'auto-organisation modulaire.

Pour aborder le problème de la catégorisation, il est intéressant de commencer par en présenter un cas particulier simple et intuitif : la catégorisation sous forme de quantification vectorielle. Nous détaillons une solution algorithmique : l'algorithme des *k-means*. Nous nous intéressons ensuite à sa version *adaptative* et aux avantages particuliers qu'elle procure.

Nous considérons ensuite l'approche par noyaux qui permet d'aborder le problème de la catégorisation d'une manière plus générale. Nous décrivons la méthode des nuées dynamique qui généralise l'algorithme des *k-means*. En analogie avec la quantification vectorielle, nous finissons par dériver une version *adaptative* des nuées dynamiques. Celle-ci constitue à notre connaissance une contribution originale.

5.1 Quantification vectorielle

Face à une grande quantité d'informations (bases de données, banques d'images, etc.), il est souvent indispensable d'ordonner, de ranger, de dresser des catégories. Le procédé de catégorisation (appelé aussi partitionnement, classification) consiste à répartir un grand nombre d'objets dans des groupes homogènes et distincts. La notion d'homogénéité, en particulier sa définition formelle, ont un rôle central dans les problèmes de catégorisation.

5.1.1 Définitions

La quantification vectorielle, cas particulier de la catégorisation, est souvent vue comme une méthode de compression de données avec pertes. Elle discrétise un espace continu et infini en un nombre fini de parties. Elle repose sur l'utilisation d'un objet mathématique, dit quantificateur vectoriel [Gersho et Gray, 1992] :

Définition 10 (Quantificateur vectoriel)
Un quantificateur vectoriel Q de dimension k et de taille N est une application de l'espace vec-

toriel $\mathbb{R}^k$ vers un ensemble fini $\mathcal{C}$ contenant N éléments. L'ensemble $\mathcal{C}$ est généralement appelé dictionnaire[21]*, ses éléments des* mots. *Formellement on a :*

$$Q : \mathbb{R}^k \rightarrow \mathcal{C} = \{w_1, .., w_N\} \tag{5.1}$$

La notion de quantificateur vectoriel est souvent décrite comme une cartographie [22] de l'espace vectoriel $\mathbb{R}^k$ en N zones. On parle de "zones" dans la mesure où les images inverses des N mots $(Q^{-1}(w_i)$, i variant de 1 à $N)$ sont souvent des parties connexes de $\mathbb{R}^k$.

Bien que ce ne soit pas toujours le cas, une méthode courante de quantification vectorielle consiste à identifier les mots du dictionnaire $\mathcal{C}$ à des points de $\mathbb{R}^k$; ces points sont alors appelés *prototypes*. Dans ce cas, la fonction Q est définie à l'aide d'une distance sur $\mathbb{R}^k$, par exemple la distance euclidienne :

$$\forall x \in \mathbb{R}^k, Q(x) = w_{i^*} \text{ avec } i^* = \arg\min_i \|x - w_i\| = \|x - Q(x)\| \tag{5.2}$$

Tout point de $\mathbb{R}^k$ est ainsi implicitement projeté sur l'un des prototypes $\{w_1, .., w_N\}$. Lorsque x est projeté sur w_{i^*}, on dit que "le prototype associé à x est w_{i^*}" ou encore que "w_{i^*} représente x". La cartographie ainsi induite correspond à ce qu'on appelle le diagramme de Voronoï induit par les prototypes. La figure 5.1 présente plusieurs prototypes dans un espace à 2 dimensions et montre le diagramme de Voronoï correspondant.

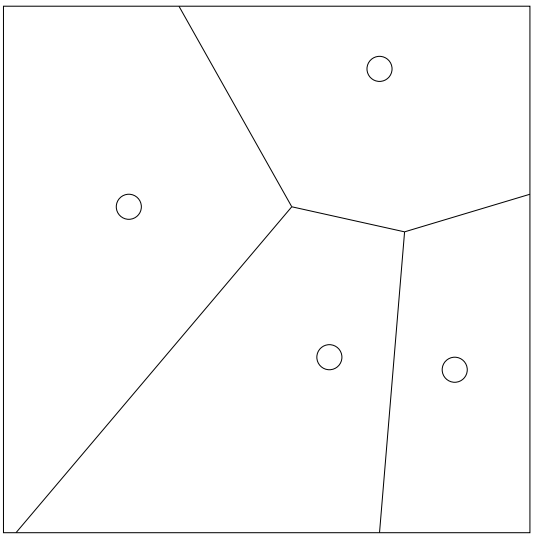

FIG. 5.1: **Quatre prototypes et le diagramme de Voronoï associé.** Quatre points d'un espace borné définissent implicitement un découpage en quatre parties.

5.1.2 Formalisation du problème de catégorisation

La notion de quantificateur vectoriel permet de formaliser le problème de la catégorisation dans un espace vectoriel. Soit une variable aléatoire X sur $\mathbb{R}^k$, de fonction de répartition f_X quelconque. Le problème de la quantification vectorielle consiste à chercher un quantificateur

[21] *Codebook* en anglais.
[22] *mapping* en anglais.

vectoriel Q qui *représente* au mieux la variable X. Formellement, il s'agit de minimiser une mesure, généralement appelée *distortion moyenne*, de l'erreur commise en considérant que Q *représente* X :

$$D = \mathrm{E}_X[d(x, Q)] = \int_{\mathbb{R}^k} \|x - Q(x)\|^2 . f_X(x) . dx \tag{5.3}$$

Considérons une réalisation $x \in \mathbb{R}^k$ de X. On peut mesurer l'erreur commise en considérant que le prototype $Q(x)$ représente le point x ; ici cette erreur vaut $\|x - Q(x)\|^2$. Alors, la distortion moyenne D est la moyenne de ces erreurs selon la distribution X.

En "découpant" l'intégrale à l'aide des N mots du dictionnaire, on obtient une autre écriture de la distortion moyenne :

$$D = \sum_{i=1}^{N} \int_{R_i} d(x, w_i) . f_X(x) . dx \tag{5.4}$$

où $R_i = Q^{-1}(w_i)$ est la "région couverte" par w_i.

En pratique, on ne connaît pas la fonction de répartition f_X de la variable aléatoire qu'on veut représenter à l'aide de Q. Par contre, on dispose d'un corpus $\mathcal{X}$ d'échantillons qu'on suppose tirés au hasard selon cette distribution. Il est alors naturel de considérer la version discrète de la distortion par rapport à ce corpus :

$$D = \sum_{x \in \mathcal{X}} d(x, Q(x)) = \sum_{i=1}^{N} \sum_{x \in X_i} d(x, w_i) \tag{5.5}$$

en notant $(X_1, .., X_N)$ la partition du corpus $\mathcal{X}$ induite par le quantificateur vectoriel (on a pour tout i, $X_i \subset Q^{-1}(w_i)$).

Le problème de la quantification vectorielle s'énonce souvent ainsi : étant donnés un corpus $\mathcal{X} \subset \mathbb{R}^k$, et la distance euclidienne $\|.\|$ dans l'espace $\mathbb{R}^k$, trouver un ensemble de prototypes $\{w_1, .., w_N\}$ qui induit un quantificateur Q minimisant la distortion au sens de l'équation 5.5.

5.1.3 L'algorithme des *k-means*

Il existe de nombreux travaux pour répondre au problème de la catégorisation. Nous présentons ici l'une des approches qui est à la fois la plus connue et l'une des plus simples. L'algorithme que nous allons décrire ci-après porte plusieurs appellations selon le domaine auquel il est appliqué. On parle de l'algorithme de LLoyd généralisé [Gersho et Gray, 1992] en quantification vectorielle ; on trouve le nom d'algorithme LBG[23] [Linde *et al.*, 1980] dans la littérature concernant la compression de données au sens large ; dans le domaine de la classification, on parle de l'algorithme *k-means* [Queen, 1967] (algorithme 5.1). Nous utilisons cette dernière dénomination dans ce chapitre.

Il s'agit d'un algorithme qui diminue la distortion de manière itérative. Chaque itération exploite consécutivement les deux constats suivants :

- Si on impose une partition du corpus en N classes, il existe une façon optimale de choisir les N prototypes correspondants. On peut montrer que la distortion d'une classe est minimale si le prototype correspondant est le centre de gravité des éléments de cette classe.
- Si on fixe les N prototypes caractérisant le quantificateur vectoriel Q, on peut partitionner de manière naturelle le corpus en N classes : deux éléments x et x' appartiennent à la même classe si et seulement si $Q(x) = Q(x')$.

[23] Acronyme du nom de ses auteurs *Linde-Buzo-Gray*.

Algorithme 5.1 Algorithme des *k-means*

Données : Un corpus de données $\mathcal{X}$
But : Trouver une catégorisation de $\mathcal{X}$ en N classes
1. Initialisation
  Soit $\mathcal{P} = \{P_1, .., P_N\}$ une partition quelconque du corpus $\mathcal{X}$.
  pour i variant de 1 à N **faire**
    $w_i \leftarrow$ barycentre de P_i
  fin pour
2. Itérations
  répéter
    pour tous les éléments x du corpus $\mathcal{X}$ **faire**
      Soit i tel que $x \in P_i$.
      Soit j tel que $Q(x) = w_j$ où Q est le quantificateur induit par $(w_1, .., w_n)$.
      si $i \neq j$ **alors**
        On transfère x de la partie P_i vers la partie P_j.
        $w_i \leftarrow$ barycentre de P_i
        $w_j \leftarrow$ barycentre de P_j
      fin si
    fin pour
  jusqu'à ce qu'il n'y ait plus de changements dans la partition

L'algorithme *k-means* est souvent considéré comme une version adaptative d'un autre algorithme, appelé *centres mobiles* [Forgy, 1965] (algorithme 5.2). Ce dernier met à jour le centre de gravité après le passage de tous les éléments du corpus alors que l'algorithme *k-means* fait cette mise à jour dès qu'un élément change de classe. S'ils ne donnent pas nécessairement le même résultat, il est facile de voir qu'une solution de l'un est solution de l'autre. Ces deux algorithmes sont assurés de trouver un minimum local de la fonction distortion. Malheureusement, il se peut que cette fonction en ait plusieurs. Notons de plus que la partition finale n'est pas toujours la même : celle-ci dépend des conditions d'initialisation et de l'ordre de présentation des données [Queen, 1967] [Diday, 1979].

Relativement récemment [Bottou et Bengio, 1995], il a été prouvé que l'algorithme *k-means* correspondait exactement à la méthode de Newton-Raphson appliquée à la minimisation de la distortion. Autrement dit, cet algorithme est équivalent à une descente de gradient déterministe du critère de distortion. La convergence vers des minima locaux, problème typique des méthodes de gradient déterministes, se comprend alors simplement.

5.1.4 Une version adaptative des *k-means*

En introduisant un peu de hasard à chaque itération d'une descente de gradient, il est possible de s'évader des minima locaux. C'est l'idée que l'on retrouve dans une famille de techniques appelées relaxation stochastique, dont les méthodes de gradient stochastiques. L'idée est de perturber (de faire bouger légèrement) l'état du système à chaque itération de manière plus ou moins aléatoire. Ce procédé peut faire augmenter le critère de coût lors d'une itération ; c'est en particulier ce qui permet de s'évader des minima locaux.

On peut aisément vérifier que la distortion (équation 5.5) est une fonction continue et déri-

Algorithme 5.2 Algorithme des centres mobiles

Données : Un corpus de données $\mathcal{X}$
But : Trouver une catégorisation de $\mathcal{X}$ en N classes
1. Initialisation
Soit $\mathcal{P} = \{P_1, .., P_N\}$ une partition quelconque du corpus $\mathcal{X}$.
2. Itérations
répéter
pour i variant de 1 à N **faire**
$w_i \leftarrow$ barycentre de P_i
fin pour
pour tous les éléments x du corpus $\mathcal{X}$ **faire**
Soit i tel que $x \in P_i$.
Soit j tel que $Q(x) = w_j$ où Q est le quantificateur induit par $(w_1, .., w_n)$.
si $i \neq j$ **alors**
On transfère x de la partie P_i vers la partie P_j.
fin si
fin pour
jusqu'à ce qu'il n'y ait plus de changements dans la partition

vable. Le gradient local (c'est-à-dire calculé par rapport à un seul individu x du corpus) vaut :

$$\frac{\partial D}{\partial w_i} = \begin{cases} 2.(x - w_i) & \text{si } Q(x) = w_i \\ 0 & \text{sinon} \end{cases} \tag{5.6}$$

L'algorithme en-ligne (c'est-à-dire qu'il met à jour les prototypes à chaque individu présenté) que l'on peut en déduire est bien souvent lui aussi désigné par le nom *k-means* [Fritzke, 1997]. Nous l'appellerons *k-means adaptatif* (voir l'algorithme 5.3 pour le détail).

Algorithme 5.3 Algorithme des *k-means* adaptatif

Données : Un corpus de données $\mathcal{X}$
But : Trouver une catégorisation de $\mathcal{X}$ en N classes
1. Initialisation
Soient $(w_1, .., w_n)$ des prototypes quelconques dans $\mathbb{R}^k$.
2. Itérations
répéter indéfiniment
Tirer aléatoirement un élément x dans le corpus $\mathcal{X}$
$i^* \leftarrow \arg\min_i \|x - w_i\|^2$
$w_{i^*} \leftarrow w_{i^*} + \varepsilon_t.(x - w_{i^*})$
fin répéter

Les itérations de cet algorithme sont plutôt intuitives : à chaque élément x présenté, on regarde le prototype qui est le plus proche et on le rapproche légèrement de x. On parle alors souvent de compétition entre les prototypes et le prototype le plus proche est alors désigné comme *vainqueur* ou *gagnant*. L'ampleur de ce déplacement est régie par le réel ε, qu'on appelle naturellement taux d'apprentissage. Les choix possibles pour ce taux d'apprentissage sont identiques à ceux que nous avons décrits dans la première partie du mémoire pour l'estimation des paramètres R et T d'un PDM (voir section 2.2.2) :

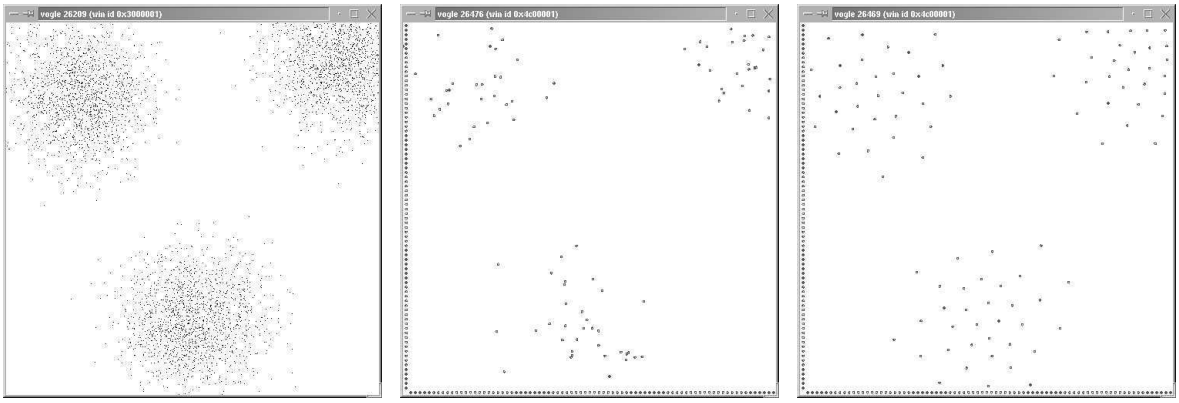

FIG. 5.2: **Illustration de l'algorithme *k-means* adaptatif.** De gauche à droite : un corpus créé à partir de 3 gaussiennes ; 100 prototypes initiaux (égaux à 100 points du corpus) ; les 100 prototypes après convergence de l'algorithme.

- on peut choisir un taux constant $\varepsilon \in]0;1[$: alors on peut montrer [Fritzke, 1997] qu'à chaque instant t, tout prototype i effectue une moyenne exponentiellement décroissante des entrées $(x_i^{(0)}, .., x_i^{(t)})$ pour lesquelles il a été gagnant et de sa position initiale $w_i^{(0)}$:

$$w_i = \varepsilon . \sum_{k=0}^{t} (1-\varepsilon)^k . x_i^{(k)} + (1-\varepsilon)^{t+1} . w_i^{(0)}$$

  Ainsi, un prototype reflète l'ensemble de points pour lesquels il a été gagnant tout en les "oubliant" progressivement au cours du temps. Ce processus ne s'arrête jamais : chaque entrée x peut provoquer des changements significatifs pour le prototype gagnant. Cela permet en théorie à ce genre de systèmes de suivre une distribution f_X non stationnaire. Dans le cas où elle est stationnaire, les prototypes finissent par devenir quasi-stationnaires autour d'un minimum local de la distortion. Le choix du taux d'apprentissage relève d'un compromis : plus il est grand, plus cette position de quasi-équilibre est atteinte rapidement mais plus on oscille autour de ce minimum.
- Un autre choix consiste à faire décroître ε lentement vers 0. Dans ce cas, les prototypes finissent par se stabiliser. L'analyse de la convergence et des bons choix de (ε_t) peuvent être consultés dans [Bottou, 1991]. On rentre alors dans le cadre très général de l'approximation stochastique.

La figure 5.2 illustre cet algorithme pour un corpus simple et stationnaire (nous avons donc utilisé un taux d'apprentissage variable).

5.1.5 Discussion

Entre la première version de *k-means* que nous avons présentée et la version *adaptative*, laquelle choisir ? La version adaptative possède un certain nombre d'avantages. Tout d'abord, l'aspect stochastique donne de meilleures chances de s'échapper des minima locaux. Ensuite, on peut remarquer que chaque itération est un calcul extrêmement simple. Enfin, le fait de traiter les données en-ligne donne plus de souplesse pour la gestion du corpus. A contrario, l'inconvénient essentiel de la version adaptative est la suivante : elle nécessite de gérer un taux d'apprentissage. Dans le cas ou le taux d'apprentissage décroît, il faut un nombre d'itérations théoriquement infini pour atteindre le point de convergence. Ceci est à comparer avec le fait que

l'algorithme des *k-means* (non adaptatif) converge généralement en un nombre fini (et en pratique souvent petit) d'itérations. Au final, le choix entre les deux approches relève d'un compromis entre qualité/souplesse d'une part et rapidité d'autre part.

5.2 La catégorisation par noyaux

La quantification vectorielle formalise le problème de la catégorisation dans un espace vectoriel. Il n'est pas bien difficile de l'étendre à des espaces bien plus complexes et génériques. C'est précisément ce que propose le principe de la catégorisation par noyaux, qui peut être vue comme une version abstraite de la quantification vectorielle.

5.2.1 Définitions

L'essence de l'approche par noyaux est le constat suivant : pour formaliser un problème de catégorisation, il suffit d'avoir deux éléments :

- Des représentants de classes appelés noyaux et notés $\{L_1, .., L_n\}$ pris dans un ensemble quelconque $\mathcal{L}$ dit espace de représentation (potentiellement différent de l'espace des données).
- Une mesure de proximité $d(x, L)$ entre un individu x du corpus et tout représentant possible $L \in \mathcal{L}$. Disons arbitrairement que plus cette mesure est petite, mieux l'individu est représenté.

Lorsqu'on a un ensemble de noyaux, il est naturel de projeter toute donnée x du corpus sur le noyau $Q(x)$ qui la représente au mieux, ce qui est défini comme suit :

$$Q(x) = \arg \min_{L \in \{L_1,..,L_N\}} d(x, L) \tag{5.7}$$

De façon analogue à la section précédente, le problème de la catégorisation consiste à chercher les noyaux $\{L_1, .., L_N\}$ qui minimisent l'équivalent de la distortion :

$$D = \sum_{x \in X} d(x, Q(x)) = \sum_{i=1}^{n} \sum x \in X_i \tag{5.8}$$

où $X_i = \{x \in \mathcal{X}; Q(x) = L_i\}$ est la classe induite par le noyau L_i :

Il est ici facile de voir que l'approche par noyaux est une généralisation de la quantification vectorielle. Si on identifie chaque noyau à un prototype et si on prend $d(x, L) = \|x - L\|^2$ on retombe exactement sur le problème posé à la section précédente. La différence fondamentale entre l'approche par noyaux et la quantification vectorielle est que pour la première, les représentants de classe ne font pas nécessairement partie de l'espace des données. Le fait de relâcher cette contrainte permet d'aborder des problèmes de catégorisation d'une grande richesse [Diday, 1971].

5.2.2 L'algorithme des nuées dynamiques

A la manière des centres mobiles (page 119), l'algorithme des nuées dynamique [Diday, 1971] [Diday, 1979] (algorithme 5.4) est une méthode itérative qui consiste à faire progressivement décroître la quantité D. On est ainsi assuré de converger vers un minimum local du critère.

A l'instar des méthodes des *k-means* et centres mobiles, les nuées dynamiques reposent sur les deux calculs complémentaires suivants :

Algorithme 5.4 Algorithme des nuées dynamiques

Données : Un corpus de données $\mathcal{X}$
But : Trouver une catégorisation de $\mathcal{X}$ en N classes
1. Initialisation
Soit $\mathcal{P} = \{P_1, .., P_N\}$ une partition quelconque du corpus $\mathcal{X}$.
2. Itérations
répéter
pour i variant de 1 à N **faire**
$L_i = \arg\min_{L \in \mathcal{L}} \sum_{x \in P_i} d(x, L)$
fin pour
pour tous les éléments x du corpus $\mathcal{X}$ **faire**
Soit i tel que $x \in P_i$.
Soit j tel que $Q(x) = L_j$ où Q est le quantificateur induit par $(L_1, .., L_n)$.
si $i \neq j$ **alors**
On transfère x de la partie P_i vers la partie P_j.
fin si
fin pour
jusqu'à ce qu'il n'y ait plus de changements dans la partition

- Etant donné une partition du corpus en N classes $\{P_1, .., P_n\}$, choisir les N noyaux optimaux au sens de D.
- Etant donnés N noyaux, partitionner le corpus en N classes.

Si le deuxième calcul est immédiat (il suffit d'utiliser la projection Q induite par les noyaux et la distance $d(.,.)$), il n'en va pas de même pour le premier. Afin d'utiliser pratiquement les nuées dynamiques, il est en effet nécessaire d'avoir une méthode qui permette de déterminer les noyaux optimaux à partir d'une partition. Dans le cas particulier de la quantification vectorielle, les prototypes optimaux sont les barycentres des données qu'ils représentent. Dans le cas général, ce calcul peut être complexe, et ce d'autant plus que les noyaux et les données sont issus d'espaces très différents.

5.2.3 Une version adaptative des nuées dynamiques

Pour rendre plus pratique l'utilisation des nuées dynamiques dans le cadre de l'approche par noyaux, et pour tirer parti des avantages de l'approximation stochastique, nous proposons d'en dériver une version adaptative (algorithme 5.5).

Algorithme 5.5 Algorithme des nuées dynamiques (version adaptative)

Données : Un corpus de données $\mathcal{X}$
But : Trouver une catégorisation de $\mathcal{X}$ en N classes
1. Initialisation
Soient $(L_1, .., L_n)$ des noyaux quelconques dans $\mathcal{L}$.
2. Itérations
répéter indéfiniment
Tirer aléatoirement un élément x dans le corpus $\mathcal{X}$
$i^* \leftarrow \arg\min_i d(x, L_i)$
Modifier légèrement L_{i^*} afin de faire diminuer $d(x, L_{i^*})$
fin répéter

Pour chaque individu x présenté, on effectue une compétition pour déterminer le noyau qui le représente au mieux, puis on effectue une mise à jour légère de ce noyau afin qu'il le représente encore mieux. En itérant ce processus un grand nombre de fois, nous conjecturons que nous pouvons obtenir un minimum local de D c'est-à-dire une catégorisation raisonnable. Dans cette version adaptative, il suffit d'avoir un processus qui permette de légèrement modifier un noyau afin qu'il représente un peu mieux une entrée. C'est une hypothèse bien moins contraignante que celle de pouvoir calculer les noyaux optimaux étant donnée une partition.

La version adaptative des nuées dynamiques que nous venons d'introduire est au centre du processus d'auto-organisation modulaire que nous décrivons dans le prochain chapitre. Son illustration est par conséquent reportée au prochain chapitre.

Résumé du chapitre

Ce chapitre a survolé la problématique de la catégorisation. Nous nous sommes tout d'abord intéressé au problème de la quantification vectorielle. Nous avons décrit l'une de ses réponses algorithmiques, les *k-means*, sous deux formes : une forme "traditionnelle" et une forme adaptative. Nous avons ensuite considéré le cadre plus général de la catégorisation en formalisant l'approche par noyaux. L'algorithme central de cette approche, les nuées dynamiques, est fortement analogue à la version "traditionnelle" des *k-means*. C'est donc naturellement que nous en avons dérivé une version adaptative.

La formulation adaptative de l'algorithme des nuées dynamiques présente un certain nombre d'avantages pratiques : entre autres elle fait moins d'hypothèses sur les processus computationnels nécessaires à son application que l'approche "traditionnelle". Le prochain chapitre va montrer qu'elle peut, de plus, jouer un rôle central dans le problème qui nous motive dans la fin du mémoire : l'auto-organisation modulaire. C'est par conséquent dans les pages qui restent que sera illustré l'algorithme que nous venons d'introduire.

Chapitre 6

L'auto-organisation modulaire ou la catégorisation de PDM

La problématique centrale de ce mémoire est l'A/R. Aussi pourrait-on penser que le précédent chapitre, traitant du problème de la catégorisation, est plutôt hors-sujet. L'un des objectifs des quelques pages qui suivent est, au contraire, de montrer que l'approche générale de la catégorisation par noyaux permet d'appréhender simplement et rapidement un problème a priori difficile de l'A/R : l'auto-organisation d'un centre de décision en modules spécialisés.

Considérons un agent apprenant, par renforcement, à réaliser un certain nombre de tâches. Par exemple, prenons le cas d'un robot ou d'un véhicule autonome devant apprendre à naviguer dans un environnement vaste. Pour aller d'un point x à un point y, il a besoin d'une carte qui lui permette, étape par étape, de planifier adéquatement chacun de ses mouvements. Cet agent pourrait utiliser une *grande* carte qui rassemble toutes les informations nécessaires à l'ensemble des déplacements qu'il doit effectuer. Une telle approche présente néanmoins un inconvénient majeur : la planification s'avèrera d'autant plus dure à réaliser que la carte est complexe.

Une solution pratique pour contrer cette complexité consiste à utiliser plusieurs cartes réduites : selon le déplacement à effectuer, l'agent déterminera la carte qui est la mieux adaptée. Une fois la bonne carte choisie, la planification sera rapide. Si nous avons simplifié une partie du problème (la planification), nous venons d'introduire une difficulté : choisir la bonne carte réduite. Pour limiter le temps potentiellement perdu pour faire ce choix, il pourrait être judicieux d'utiliser une approche de type "diviser pour régner". Des agents spécialisés vont examiner, en parallèle, chacune des cartes, et donner un avis sur leur adéquation avec la tâche courante. Le robot utilisera ces avis pour choisir la carte à utiliser.

L'idée de concevoir un agent autonome de manière modulaire relève exactement de la même philosophie. Les modules, parties constituantes de l'agent, peuvent analyser de leurs points de vue la tâche courante et évaluer leur compétence pour s'en charger. L'agent n'aura plus qu'à laisser s'exprimer le module qui est le plus adapté. Si l'approche modulaire est en théorie séduisante, elle amène dans le cas général de nombreuses questions : comment définir les limites de compétences de chacun des modules ? Lorsque plusieurs modules proposent des solutions, comment choisir la meilleure ?

La théorie de l'A/R, l'apprentissage de représentation auquel nous avons consacré la deuxième partie de ce mémoire, et la catégorisation permettent de proposer une réponse à toutes ces questions. Ce chapitre est organisé en deux sections : dans la première, nous formalisons l'approche modulaire et les problèmes qu'elle pose dans le cadre de l'A/R. La deuxième partie en fera une illustration sur un problème de navigation continue.

6.1 Modularité et A/R

Nous commençons par formaliser les notions de "modules" et de "module le plus compétent pour une tâche donnée". Ensuite nous présentons le problème de l'auto-organisation modulaire comme un cas particulier de catégorisation par noyaux. Enfin, nous montrons comment utiliser la méthode adaptative des nuées dynamiques pour trouver une organisation modulaire adéquate.

6.1.1 Définitions

Plusieurs tâches

L'intérêt d'avoir une approche modulaire tient à l'hypothèse qu'il puisse y avoir plusieurs tâches à résoudre. Cela revient à considérer qu'il y a n PDM $(\mathcal{M}_i = \langle S, A, T_i, R_i \rangle)_{(1 \leq i \leq n)}$. Implicitement, nous venons de considérer que ces PDM sont définis sur le même espace d'états S et sur le même ensemble d'actions A. Ceci n'est pas une limitation mais simplement une commodité de notation. Si les n PDM étaient définis sur des espaces différents (S_i, A_i), on pourrait considérer qu'ils sont alors implicitement définis dans $S = S_1 \times .. \times S_n$ et $A = A_1 \times .. \times A_n$.

Un agent modulaire

Nous avons vu dans la partie précédente le rôle central que pouvait prendre la représentation simplifiée d'un problème d'A/R, c'est-à-dire d'une manière plus précise sa représentation agrégée. Nous allons par conséquent considérer qu'un module est une agrégation particulière de l'espace d'états. Les m modules qui constituent l'agent sont donc identifiés à des agrégations :

$$(\widehat{S}_1, .., \widehat{S}_m)$$

Le module le plus compétent

Considérons l'une des tâches à résoudre, c'est-à-dire l'un des PDM $\mathcal{M}_i$. On peut construire les m PDM $(\widehat{\mathcal{M}}_1, .., \widehat{\mathcal{M}}_m)$ approximatifs déduits à partir des m agrégations. En effet, il suffit d'estimer leurs m fonctions de récompenses et de transitions. La partie précédente nous a également montré que nous pouvions estimer l'erreur d'approximation que chacun d'eux commet en tant qu'approximation de $\mathcal{M}_i$. Le PDM approché $\widehat{\mathcal{M}}^*_{k(i)}$ qui est meilleur que tous les autres est tel que :

$$k(i) = \arg \min_{1 \leq j \leq m} E^{\mathcal{M}_i}_{app}(\widehat{S}^*_j)$$

L'erreur d'approximation que son agrégation implique est en effet la plus petite. Cette loi nous permet donc de désigner sans ambiguité le module $k(i)$ qui est le plus compétent pour la tâche i.

Un critère de coût pour évaluer l'organisation modulaire

On peut définir m modules, c'est-à-dire m agrégations de l'espace d'états de nombreuses manières. Lorsqu'on a ces m modules, on peut estimer l'erreur $E(\widehat{S}_1, .., \widehat{S}_m)$ qu'ils commettent globalement pour résoudre les n tâches :

$$E(\widehat{S}_1, .., \widehat{S}_m) = \sum_{i=1}^{n} E^{\mathcal{M}_i}_{app}(\widehat{S}_{k(i)}) = \sum_{i=1}^{n} \min_{1 \leq j \leq m} E^{\mathcal{M}_i}_{app}(\widehat{S}_j)$$

Cette équation permet donc d'évaluer numériquement l'organisation des modules. Une bonne organisation en modules spécialisés est donc une organisation qui minimise ce coût. Le problème de l'auto-organisation consiste par conséquent à chercher les m agrégations $(\widehat{S}_1^*, .., \widehat{S}_m^*)$ qui vérifient :

$$(\widehat{S}_1^*, .., \widehat{S}_m^*) = \arg \min_{(\widehat{S}_1, .., \widehat{S}_m)} E(\widehat{S}_1, .., \widehat{S}_m) = \arg \min_{(\widehat{S}_1, .., \widehat{S}_m)} \left(\sum_{i=1}^{n} \min_{1 \leq j \leq m} E_{app}^{\mathcal{M}_i}(\widehat{S}_j) \right) \quad (6.1)$$

6.1.2 Auto-organisation modulaire et catégorisation

Nous allons à présent montrer que le problème que nous venons de poser est un problème de catégorisation par noyaux. La catégorisation par noyaux requiert de définir l'espace des *données*, l'*espace de représentation* dans lequel on trouve les noyaux, et une mesure de représentativité/distance entre tout noyau et toute donnée :

- L'espace des *données* à catégoriser est l'ensemble des PDM définis sur l'espace d'états S. Le corpus des données à catégoriser est l'ensemble des PDM $(\mathcal{M} = \langle S, A, T_i, R_i \rangle)_{(1 \leq i \leq n)}$ pour lesquels on cherche des politiques optimales.
- L'*espace de représentation* des noyaux est l'ensemble de toutes les agrégations possibles de l'espace d'états. Un noyau est donc ici synonyme de module.
- Finalement, la mesure de représentativité/distance entre une *donnée* et un *noyau*, c'est-à-dire entre un PDM $\mathcal{M}$ et une agrégation de son espace d'états, est l'erreur d'approximation induite par l'agrégation :

$$d(\mathcal{M}, \widehat{S}) = E_{app}^{\mathcal{M}}(\widehat{S})$$

Le problème formalisé par l'équation 6.1 correspond alors exactement à la minimisation de la distortion (cf. équation 5.8) dans le problème de catégorisation que nous venons de décrire.

6.1.3 Application des nuées dynamiques adaptatives

Pour répondre pratiquement à ce problème de catégorisation par noyaux, nous utilisons la version adaptative de l'algorithme des nuées dynamiques. Afin d'utiliser cet algorithme, nous avons besoin d'un processus qui permette d'adapter un noyau pour faire diminuer la distance $d(\mathcal{M}, \widehat{S})$ qui le sépare d'une donnée. Autrement dit, nous avons besoin d'un procédé pour faire évoluer une agrégation de l'espace d'états de sorte à diminuer l'erreur d'approximation. Ce processus est exactement le processus d'apprentissage de représentation que nous avons présenté dans la partie précédente.

L'algorithme 6.1 montre concrètement ce que devient l'algorithme des nuées dynamiques lorsqu'on l'adapte au cadre de notre problématique d'auto-organisation.

Décrivons le processus d'auto-organisation d'une manière schématique. On présente l'un des n problèmes à résoudre. Appelons-le $\mathcal{M}$. Chaque module estime ses paramètres et en déduit l'erreur qu'il commet en approximant $\mathcal{M}$. Une compétition entre les modules s'ensuit : le module qui fait la plus petite erreur d'approximation est désigné comme vainqueur. On adapte l'agrégation caractéristique du module de sorte qu'il donne la prochaine fois une erreur d'approximation moindre. Et on recommence avec un autre des n problèmes.

Il est important de noter que la boucle centrale **pour** de cet algorithme est potentiellement parallèle : l'estimation des paramètres des m approximations peut se faire en parallèle à l'aide des mêmes échantillons du vrai modèle $\mathcal{M}$ et les calculs d'erreur d'approximation sont indépendants deux à deux.

Algorithme 6.1 Algorithme d'auto-organisation modulaire

1. Initialisation
   Soit $(\widehat{S}_1, .., \widehat{S}_m)$ des agrégations quelconques de S.
2. Itérations
   répéter
   Soit $\mathcal{M}$ choisi aléatoirement dans $\{\mathcal{M}_1, .., \mathcal{M}_n\}$
   pour tout j variant de 1 à m **faire**
   Utiliser des échantillons du PDM $\mathcal{M}$ pour estimer les paramètres $\widehat{R}_j$ et $\widehat{T}_j$, $\overline{\Delta R}_j$ et $\overline{\Delta T}_j$ du PDM approximatif défini par l'agrégation $\widehat{S}_j$
   En déduire l'erreur d'approximation $E^{\mathcal{M}}_{app}(\widehat{S}_j)$ (cf. section 3.1)
   fin pour
   $j^* \leftarrow \arg\min_j E^{\mathcal{M}}_{app}(\widehat{S}_j)$
   Mettre à jour l'agrégation $\widehat{S}_{j^*}$ de sorte à diminuer l'erreur d'approximation $E^{\mathcal{M}}_{app}(\widehat{S}_{j^*})$ (cf. section 3.3 et chapitre 4)
   jusqu'à condition d'arrêt

6.2 Une mise en œuvre expérimentale

Nous proposons d'illustrer le potentiel de l'algorithme que nous venons de présenter sur un problème de contrôle simple. Le problème que nous considérons est une variation du problème de navigation continue que nous avons considéré dans la partie précédente (page 67). Nous considérons cette fois-ci 6 tâches et montrons comment nous pouvons calculer 3 modules ou agrégations de l'espace d'états qui permettent d'y répondre efficacement.

6.2.1 Description de l'exemple traité

Comme précédemment, un agent de type robot peut se déplacer dans un environnement qui comporte deux pièces et deux couloirs. Le mouvement de cet agent est légèrement bruité. Nous allons définir 6 PDM pour décrire six tâches différentes auxquelles cet agent est confronté. Dans l'exemple que nous donnons, les 6 problèmes à résoudre partagent la même fonction de transition T. Seule la fonction de récompense et les conditions initiales varient.

Rappelons quelques caractéristiques de ce problème de navigation. L'espace d'états S est l'ensemble des positions dans l'environnement $s = (x, y)$ avec $0 \leq x \leq 10$ et $0 \leq y \leq 10$. Les actions sont des mouvements d'amplitude 0.1 dans 8 directions cardinales plus l'action "ne pas bouger". Chaque déplacement est légèrement bruité (bruit d'amplitude 0.03). La dynamique est identique à celle que nous avons présentée dans la partie précédente (page 66).

Dans cet espace d'états, on définit 6 zones particulières que l'on numérote de 1 à 6 (voir figure 6.1). Les 6 tâches de l'agent, définies par 6 PDM, sont d'aller d'une zone à une autre. Le tableau ci-dessous les précise :

Problème	Condition initiale	Objectif
$\mathcal{M}_1$	centre de la zone 2	atteindre la zone 1
$\mathcal{M}_2$	centre de la zone 3	atteindre la zone 2
$\mathcal{M}_3$	centre de la zone 4	atteindre la zone 3
$\mathcal{M}_4$	centre de la zone 5	atteindre la zone 4
$\mathcal{M}_5$	centre de la zone 6	atteindre la zone 5
$\mathcal{M}_6$	centre de la zone 1	atteindre la zone 6

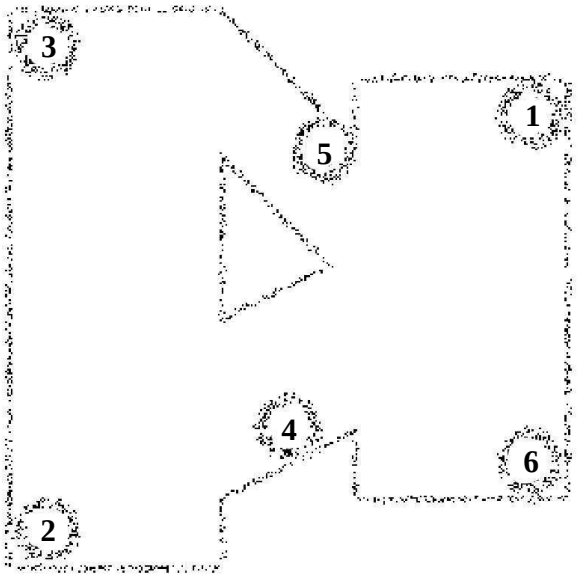

FIG. 6.1: **Environnement du problème de navigation continue multi-objectifs.** Dans cet environnement, nous avons distingué 6 zones qui correspondent chacune à un objectif et à une position intiale particuliers.

Pour formaliser les objectifs, nous utilisons comme d'habitude la fonction récompense. La fonction récompense du i^{eme} PDM vaut 0 partout sauf dans la zone i ou elle vaut 1. Nous donnons, comme dans les modèles précédents, une récompense négative -1 lorsque ses actions l'amènent à se cogner contre un mur.

Nous avons expérimenté le problème d'auto-organisation avec les paramètres suivants : nous allons approximer les 6 problèmes définis ci-dessus à l'aide de 3 approximations par agrégations $(\widehat{S}_1, \widehat{S}_2, \widehat{S}_3)$ qui discrétisent chacune l'espace d'états S en 64 macro-états. Nous proposons donc de calculer 3 modules pour résoudre les 6 PDM que nous venons de décrire.

6.2.2 Résultats de la classification

Nous avons utilisé l'algorithme d'auto-organisation 6.1. A chaque itération, pour mettre à jour l'agrégation de l'approximation qui gagne la compétition, nous avons utilisé le procédé d"'apprentissage" décrit au chapitre 4 conjointement avec l'heuristique qui s'est avérée la plus performante : h_3.

Nous avons mesuré à l'aide de simulations l'évolution des performances du système pour chacune des 6 tâches pendant le processus d'auto-organisation. De même, nous avons enregistré la répartition des 6 PDM sur les 3 noyaux/modules. Ces résultats sont illustrés à la figure 6.2. Premièrement, on observe sur le graphique de gauche que les performances pour chacun des n PDM tend à augmenter au cours du processus. Il en va logiquement de même pour la performance moyenne. Le graphique de droite nous montre l'évolution de la répartition des 6 problèmes sur les 3 noyaux. Cette répartition évolue significativement durant les 30 premières itérations puis elle se stabilise. Au final, le système converge vers trois classes rassemblant chacune deux des PDM à résoudre. Ces classes sont $C_1 = \{\mathcal{M}_2, \mathcal{M}_3\}$, $C_2 = \{\mathcal{M}_4, \mathcal{M}_5\}$ et $C_3 = \{\mathcal{M}_1, \mathcal{M}_6\}$

La figure 6.3 donne une vision duale de cette classification. Elle représente les agrégations correspondant à chacune des trois classes. Il est facile de faire visuellement le lien entre une

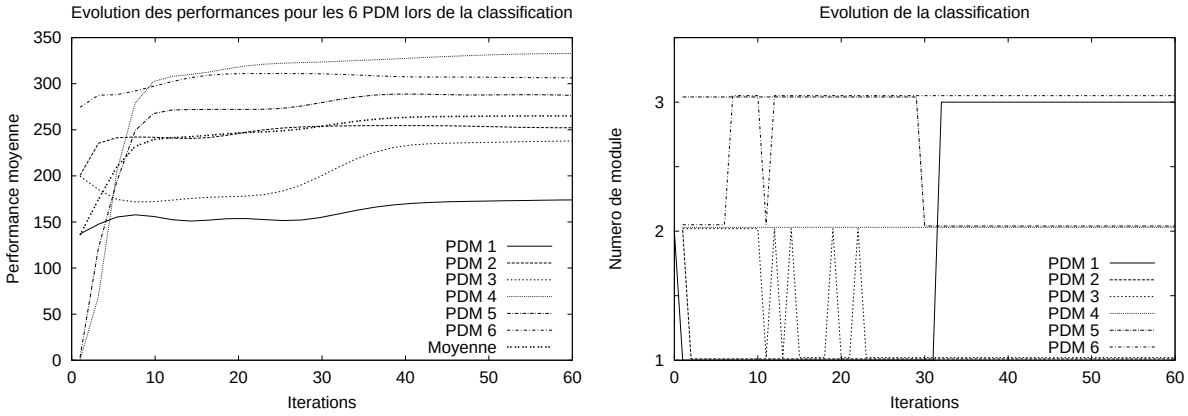

FIG. 6.2: **Evolution de la classification des 6 tâches par 3 noyaux.** A gauche nous avons tracé le niveau des perfomances pour les 6 tâches. A droite : nous montrons l'évolution de la classification

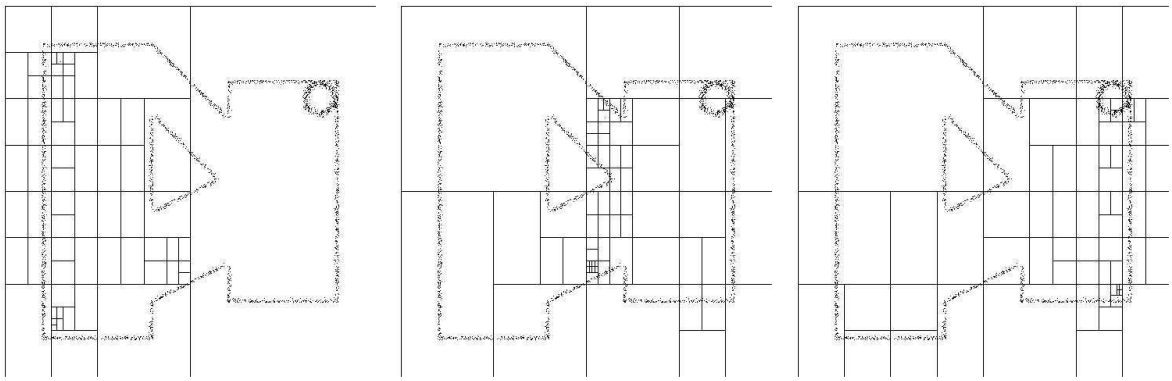

FIG. 6.3: **Représentations obtenues après classification de 6 tâches en 3 modules.** De gauche à droite, ces représentations se sont respectivement spécialisées sur les 3 couples de tâches $\{\mathcal{M}_2, \mathcal{M}_3\}$, $\{\mathcal{M}_4, \mathcal{M}_5\}$ et $\{\mathcal{M}_1, \mathcal{M}_6\}$.

représentation et les problèmes auxquels elle répond efficacement : on voit que la représentation qui est adaptée à une tâche représente précisément sa zone but.

6.2.3 Commentaires

Un exemple d'utilisation des modules

Supposons que l'agent veuille suivre le parcours $6 \rightarrow 5 \rightarrow 4$.... Pour rendre simple l'illustration qui suit, nous allons supposer que la fonction de transition est apprise une fois pour toutes par chacun des modules et que la fonction récompense est directement accessible à l'agent. Initialement, l'agent se trouve au centre de la zone 6. Son premier objectif est d'atteindre la zone 5. Il met en compétition ses trois modules et, d'après les résultats que nous avons obtenus pour la classification, c'est le module correspondant à la classe $C_2 = \{\mathcal{M}_4, \mathcal{M}_5\}$ qui fera la plus petite erreur d'approximation. C'est par conséquent ce module qui va se charger du déplacement. L'agent est à présent au centre de la zone 5, il veut rallier la zone 4. Les modules sont remis en compétition. C'est de nouveau le module correspondant à la classe C_2 qui est gagnant et qui se charge de la tâche. La prochaine tâche, aller de la zone 4 à la zone 3, sera traitée par le module correspondant à la classe $C_1 = \{\mathcal{M}_2, \mathcal{M}_3\}$. Et ainsi de suite.

Traitement de la nouveauté

Supposons maintenant que ce même agent veuille effectuer une tâche sur laquelle il ne s'est jamais entraîné : par exemple aller de la zone 3 à la zone 5. Chaque module, en tant que PDM, peut effectuer la planification correspondant à cette nouvelle tâche. L'un d'entre eux fera moins d'erreurs que les autres. Dans la mesure où c'est celui qui décrit le mieux la zone but, il est probable que le module correspondant à la classe $C_2 = \{\mathcal{M}_4, \mathcal{M}_5\}$ fera la meilleure approximation. Alors un coup d'œil à son agrégation (figure 6.3) suggère que le plan amènera l'agent à faire le trajet $3 \rightarrow 4 \rightarrow 5$. Bien que cela ait peu de chance d'amener à des politiques optimales, le fait que les modules soient des PDM permet d'envisager l'occurrence de nouvelles tâches.

Pour une auto-organisation perpétuelle

Si une tâche nouvelle peut s'appuyer sur l'une des représentations caractérisant les modules, elle pourra être traitée par cette architecture modulaire. De plus, si cette tâche revient souvent et si on n'arrête pas le processus d'auto-organisation modulaire, les modules finiront par se réorganiser pour en tenir compte. Un autre événement pourrait également avoir cet effet de réorganisation. Imaginons qu'un des modules arrête de fonctionner. Si les modules qui restent continuent le processus d'auto-organisation, ils finiront par petit à petit se réorganiser pour globalement intégrer les tâches dont se chargeait le module qui vient de défaillir. Pour eux, c'est exactement comme si ces tâches étaient nouvelles. Si le processus d'auto-organisation ne s'arrête jamais, on peut penser que l'architecture gagnera en adaptation et en tolérance aux pannes.

Résumé du chapitre

Ce chapitre a été l'occasion de montrer que l'A/R, l'apprentissage de représentation et la catégorisation par noyaux permettent de proposer des mécanismes de fonctionnement et d'organisation modulaire.

Nous avons argumenté qu'il était naturel d'identifier la notion de module à celle d'agrégation de l'espace d'états. Pour une tâche donnée, la compétence d'un module se mesure à l'aide de l'erreur d'approximation que son agrégation implique. Le module le plus compétent est par conséquent celui qui induit la plus petite erreur. Enfin, l'auto-organisation peut être vue comme un processus qui vise à diminuer globalement les erreurs d'approximation des modules les plus compétents pour l'ensemble des tâches à résoudre.

Ce dernier problème, l'auto-organisation modulaire, s'énonce naturellement dans le cadre de la catégorisation par noyaux. Nous avons montré que la version adaptative des nuées dynamiques introduite dans le chapitre précédent permettait d'y répondre en pratique. Nous avons illustré ce procédé sur un problème de navigation continue impliquant six tâches à répartir sur trois modules.

Conclusion

La partie précédente a montré comment déterminer une représentation adéquate pour un problème d'A/R, c'est-à-dire une bonne agrégation de l'espace d'états. Ici, nous avons généralisé ce problème à celui de construire plusieurs représentations pour plusieurs problèmes. Nous avons en particulier traité le cas où le nombre de problèmes d'A/R à résoudre est strictement plus grand que le nombre de représentations que l'on peut construire. Ceci implique, entre autres choses, de répartir adéquatement les tâches sur les représentations.

Dans un premier temps, nous avons introduit la problématique de la catégorisation de données. Nous avons considéré le cas particulier de la quantification vectorielle et nous avons détaillé l'algorithme *k-means*, dans sa version traditionnelle puis dans sa version adaptative. Nous avons ensuite abordé le cas général de la catégorisation par noyaux. A notre connaissance, il n'existe pas de réponse algorithmique adaptative pour ce problème. Nous en avons donc dérivé une à partir de l'algorithme des nuées dynamiques.

Ce détour par la problématique de la catégorisation nous a permis de formaliser rapidement et simplement le problème central de cette partie : l'auto-organisation modulaire. Etant donnés un ensemble de n problèmes d'A/R à résoudre et de m modules, l'utilisation conjointe de 1) la version adaptative des nuées dynamiques et de 2) l'apprentissage de représentation permet de formaliser ce problème et d'y répondre algorithmiquement. Nous avons illustré cette approche sur un problème simple de navigation : un agent ayant six tâches à accomplir. L'algorithme que nous avons proposé lui a permis de répartir automatiquement ces six tâches sur trois modules, répartition qui est uniquement guidée par la nécessité pour l'agent d'être performant.

Un agent, apprenant par renforcement, et constitué du mécanisme d'auto-organisation modulaire, présente des qualités intéressantes. Lorqu'on lui propose une tâche (connue ou inconnue), l'ensemble de ses modules peuvent essayer de la résoudre. Ces résolutions sont complètement indépendantes. Avoir plusieurs petits modules spécialisés permet de répondre plus rapidement que si on avait un seul gros module. Et pour chaque tâche, l'approche que nous proposons fait que le choix du module est guidé par la qualité des performances : c'est le module qui fait le moins d'erreur sur la fonction de valeur optimale qui est choisi pour agir. Finalement, l'idée de distribuer la résolution de problèmes d'A/R sur plusieurs représentations indépendantes et adaptatives augmente la tolérance aux pannes : si un module meurt, les autres modules peuvent se réorganiser afin de maintenir les performances de l'agent.

Une fois encore, remettons-nous en perspective de la première partie, où nous montrions que les problèmes d'A/R ayant un petit espace d'états admettent une réponse complètement connexionniste. Lorsqu'on considère n problèmes d'A/R ayant un grand espace d'états, le procédé d'auto-organisation modulaire permet de les transformer en $m < n$ problèmes ayant un petit espace d'états. Ainsi, on transforme n problèmes difficiles en m problèmes approchés plus faciles et par bien des aspects plus pratiques à résoudre. Cette transformation simplificatrice, amène inéluctablement des approximations, mais il faut souligner le fait que la répartition est guidée par la minimisation de l'erreur d'approximation sur la fonction de valeur, c'est-à-dire par le désir

de performances de l'architecture.

Nous disions, dans la conclusion de la partie précédente, que le processus d'apprentissage d'une représentation peut s'intégrer dans une architecture complètement connexionniste. Les traitements impliqués pour le calcul d'une erreur d'approximation sont analogues à ceux, complètement connexionnistes, utilisés par l'algorithme *Value Iteration* pour calculer une fonction de valeur optimale. Les procédures algorithmiques que nous avons introduites dans cette dernière partie accentuent le parallélisme obtenu jusqu'ici : les calculs effectués par les modules sont complètement indépendants, et donc naturellement parallèles. Seul le processus le plus simple, c'est-à-dire la compétition des modules (pour désigner celui qui prend en charge une tâche ou celui qui se spécialise) nécessite une coordination. La planification, l'estimation de l'erreur d'approximation effectuée par chaque module peut se faire de manière indépendante des autres, et donc en complet parallélisme.

Conclusion générale

D'une manière conventionnelle, nous allons conclure ce manuscrit en présentant un synopsis du travail que nous avons effectué pendant ces trois dernières années, puis en proposant une liste des perspectives que nous pourrions envisager.

Synopsis de la thèse

Le travail présenté dans ce mémoire tente de transposer un certain nombre de qualités intéressantes que l'on observe dans le cerveau des êtres vivants, qualités qui constituent à notre avis des caractéristiques essentielles de ce qu'on appelle "l'intelligence". Nous avons ainsi travaillé en plaçant comme horizon la construction de programmes d'IA qui sont *autonomes*, *robustes*, *dynamiques*, *multi-tâches*, *anytimes*, *tolérants aux pannes* et *massivement parallèles*.

Nous avons considéré le paradigme de l'A/R qui permet de modéliser de manière générique un agent devant *apprendre par l'interaction à atteindre des objectifs*. Nous avons argumenté que cette approche permettait d'aborder de front les problèmes généraux impliqués pour un *apprentissage autonome et robuste*. Il constitue à notre avis un condensé théorique puissant et crédible des enjeux de l'apprentissage automatique.

La littérature de ce domaine montre qu'il admet de nombreux domaines d'application : la théorie des jeux [Tesauro, 1995], le contrôle d'un bras mécanique [Sutton, 1996], l'optimisation d'un réseau d'ascenseurs [Crites et Barto, 1996], ou l'allocation dynamique de canaux téléphoniques [Singh et Bertsekas, 1997]. Le présent manuscrit, en tendant vers des problèmes réalistes comme la conduite automatique d'une voiture, espère contribuer à prouver que l'A/R est un domaine qui, en plus de ses enjeux théoriques, devrait amener à des applications réelles conséquentes.

Le manuscrit s'est articulé en trois parties. Les tâches d'A/R qui y sont traitées ont une complexité croissante au long du manuscrit. Deux caractéristiques jouent un rôle dans cette complexité : la taille de l'espace d'états sur lequel est défini le problème et, dans une moindre mesure, le nombre de tâches à résoudre. La première partie s'est intéressée au cas *multi-tâche dans un petit espace d'états*. La deuxième partie a étudié le cas *mono-tâche dans un grand espace d'états*. Finalement, la dernière partie a proposé une réponse pour le cas *multi-tâches dans un grand espace d'états*.

Dans la première partie, nous avons posé les bases théoriques de l'A/R. Nous avons décrit en détail le formalisme des PDM qui constitue son cœur et l'algorithme *Value Iteration* qui a d'intéressantes qualités ; ce dernier est en effet *dynamique* et *multi-tâche*. Lorsque l'espace d'états est petit, l'utilisation simultanée 1) d'une approche indirecte de l'A/R et 2) de *Value Iteration*

peut être vue comme une architecture connexionniste. Les *unités* de cette architecture sont les états du PDM. Les *connexions* sont le support de la fonction décrivant la dynamique du problème T. La récompense R est estimée de manière répartie par l'ensemble des unités. Enfin, la fonction de valeur optimale est une activité distribuée dans le réseau et l'algorithme *Value Iteration* est une loi de propagation de cette activité.

Dans la deuxième partie, nous avons présenté et analysé théoriquement une méthode pour approximer un problème d'A/R ayant un grand espace d'états. Cette méthode, en effectuant une agrégation des états, c'est-à-dire en construisant automatiquement une représentation simplifiée, permet de transformer un problème a priori difficile en un problème approché qu'on peut aborder avec les techniques connexionnistes décrites dans la première partie. En nous appuyant sur des résultats récents, nous avons analysé théoriquement cette approche approximative et montré que nous pouvions, tout en maîtrisant la complexité du modèle approché, améliorer itérativement ses performances. Nous avons expérimenté plusieurs voies pour améliorer ce genre d'approximations, processus que nous avons rassemblés sous l'appellation "apprentissage de représentation" : diminution de la complexité sans faire chuter ses performances (généralisation), augmentation de la qualité en augmentant au minimum la complexité (spécialisation) voire en la maintenant constante (apprentissage). Nous avons testé ce mécanisme sur de multiples problèmes. D'un point de vue algorithmique, les calculs impliqués pour cette amélioration des performances sont complètement analogues à ceux impliqués par Value Iteration. Ils pourraient ainsi s'intégrer dans l'architecture complètement connexionniste décrite dans la première partie.

Dans la troisième partie, nous avons voulu étendre les résultats obtenus pour l'apprentissage de représentation à un cadre encore plus complexe : celui où une architecture doit résoudre non plus *un* mais *plusieurs* problèmes d'A/R ayant un grand espace d'états. Nous avons replacé ce problème dans le cadre général de la catégorisation automatique. En particulier nous avons montré que le processus qui consiste à construire un nombre réduit de représentations pour résoudre un grand nombre de problèmes d'A/R peut se formaliser comme un problème de catégorisation par noyaux. Pour répondre pratiquement à ce problème, nous avons dérivé une version adaptative de l'algorithme des nuées dynamiques. Nous avons argumenté que ce problème pouvait s'interpréter comme celui de l'auto-organisation en modules spécialisés : le processus que nous avons décrit permet de répartir intelligemment un ensemble de compétences sur un nombre prédéfini de modules, qui maintiennent chacun une représentation simplifiée de l'espace d'états. La construction d'une architecture modulaire présente de nombreux avantages. Parmi ceux que nous avons décrits, soulignons qu'en augmentant la distribution des traitements, on augmente potentiellement la tolérance au pannes et les possibilités de parallélisation.

Quelques perspectives

Il apparaît de nombreuses voies de recherches pour approfondir ce travail. En voici quelques unes :

- Dans cette thèse, nous avons fait le choix de nous concentrer sur un point précis : la construction d'une ou plusieurs représentations du vrai espace d'états. Pour ce faire, nous avons éludé certaines des caractéristiques de l'A/R comme la nécessité d'acquérir des connaissances en ligne et le dilemme exploration/exploitation que cela implique. L'incertitude liée au manque d'exploration (c'est-à-dire à une trop petite quantité d'échantillons pour estimer R et T) peut être vue comme une composante de ce que nous avons appelé l'erreur d'interpolation locale. Cette idée pourrait être creusée dans deux directions : 1) on pourrait examiner le lien qui existe entre la vision du manque d'exploration comme

une composante de l'erreur d'interpolation et les réponses de la littérature au dilemme exploration/exploitation. 2) Si l'apprentissage de représentation et l'exploration sont alors deux moyens de faire diminuer l'erreur d'approximation, comment peut-on les combiner efficacement ?

- Lorsqu'on utilise une approximation par agrégation, l'ajout de ressources permet, en théorie, de diminuer l'erreur d'approximation autant qu'on le souhaite. On pourrait fixer un seuil d'erreur à atteindre. On ajouterait et supprimerait autant de ressources que nécessaires pour rester sous ce seuil d'erreur. Ainsi, on adapterait la complexité de représentation à la difficulté du problème considéré.
- Nous avons montré comment construire plusieurs modules spécialisés pour un ensemble de tâches. Ces modules font des calculs de planification indépendants deux à deux. Il pourrait être intéressant de creuser l'idée de faire une planification qui implique la coopération de plusieurs modules. Des travaux de la littérature utilisant un réseau CMAC [Sutton, 1996] [Wiering *et al.*, 1998] exploitent déja un peu cette idée : plusieurs grilles découpent différemment un même espace d'états et font un calcul commun.
- Nous avons mis en valeur le caractère fortement parallélisable des mécanismes que nous proposons dans cette thèse. Il serait intéressant de les implanter sur des machines parallèles. En particulier, leur intégration dans une unique architecture serait l'occasion de présenter notre approche d'une manière plus unifiée.
- En tant qu'informaticiens, nous avons construit des mécanismes connexionnistes pour résoudre un problème qui nous paraît motivant : l'A/R. L'évolution a manifestement fait la même chose : les animaux et les hommes sont des réponses connexionnistes au problème réel de l'A/R. Il serait donc naturel de creuser les liens entre nos travaux et des études concernant les sciences du vivant. Nous voyons déjà plusieurs pistes. La propagation d'un signal de récompense pour calculer un plan semble commune à nos travaux et des travaux directement inspirés du cortex humain [Frezza-Buet, 1999]. Le principe d'agrégation d'états semble fortement lié à un mécanisme connu sous le nom de catégorisation perceptivo-motrice [Munos, 1997b]). Enfin, l'idée d'une organisation modulaire n'est pas sans rappeler le constat que le cerveau est organisé en aires fonctionnelles relativement spécialisées...

Annexe

Annexe A

Erreur d'approximation et influence

Cette annexe donne les détails calculatoires qui permettent de démontrer
- le lien entre l'erreur d'interpolation et l'erreur d'approximation ;
- les propriétés caractéristiques de l'influence.

Les calculs présentés ici sont tirés de [Munos et Moore, 2000] et [Munos et Moore, 2002]

A.1 Relation entre l'erreur d'interpolation et l'erreur d'approximation

A.1.1 Enoncé

Notations et définitions

Soit un PDM $\mathcal{M} = \langle S, A, T, R \rangle$ que nous appelons le vrai modèle. Soit un deuxième PDM $\widehat{\mathcal{M}} = \langle S, A, \widehat{T}, \widehat{R} \rangle$ qui approxime le PDM $\mathcal{M}$. Soit B^* l'opérateur de Bellman dans le *vrai* modèle. Pour toute fonction $W : S \to \mathbb{R}$, $B^*.W$ est une nouvelle fonction de $S \to \mathbb{R}$ qui est telle que :

$$[B^*.W](s) = \max_a \left(R(s,a) + \gamma . \sum_{s'} T(s,a,s').W(s') \right)$$

De façon analogue, soit $\widehat{B}^*$ l'opérateur de Bellman dans le modèle approché :

$$[\widehat{B}^*.W](s) = \max_a \left(\widehat{R}(s,a) + \gamma . \sum_{s'} \widehat{T}(s,a,s').W(s') \right)$$

Notons respectivement V^{π^*} et $\widehat{V}^{\widehat{\pi}^*}$ les fonctions de valeur optimale du vrai modèle $\mathcal{M}$ et du modèle approché $\widehat{\mathcal{M}}$. Par définition, ces fonctions sont les uniques points fixes des opérateurs que nous venons de définir : $B^*.V^{\pi^*} = V^{\pi^*}$ et $\widehat{B}^*.\widehat{V}^{\widehat{\pi}^*} = \widehat{V}^{\widehat{\pi}^*}$.

L'erreur induite en un point par l'utilisation du modèle approché est définie par :

Définition 11 (Erreur d'approximation locale)
L'erreur d'approximation locale est la différence entre la fonction de valeur optimale approchée et la vraie fonction de valeur optimale :

$$E_{app}(s) = |V^{\pi^*}(s) - \widehat{V}^{\widehat{\pi}^*}(s)|$$

L'erreur d'interpolation locale :

Définition 12 (Erreur d'interpolation locale)
L'erreur d'interpolation locale est l'erreur induite si on utilise l'opérateur de Bellman approché $\widehat{B}^*$ *au lieu du vrai opérateur de Bellman* B^* *sur la vraie fonction de valeur* V^{π^*} *:*

$$E_{int}(s) = |\widehat{B}^*.V^{\pi^*}(s) - B^*.V^{\pi^*}(s)|$$

Enoncé du résultat

Si $\overline{E_{int}}(s)$ est une borne supérieure de $E_{int}(s)$, alors une borne supérieure $\overline{E_{app}}(s)$ de $E_{app}(s)$ est solution de l'équation de Bellman suivante :

$$\overline{E_{app}}(s) = \max_a \left(\gamma . \sum_{s'} \widehat{T}(s,a,s').\overline{E_{app}}(s') \right) + \overline{E_{int}}(s)$$

A.1.2 Démonstration

Un lemme préliminaire sur l'opérateur max

Soient deux fonctions f et g de X dans $\mathbb{R}$. On a :

$$|\max_{x\in X} f(x) - \max_{x\in X} g(x)| \leq \max_{x\in X} |f(x) - g(x)|$$

En effet, si on note $x_f = \arg\max_{x\in X} f(x)$ et $x_g = \arg\max_{x\in X} g(x)$, on a :

$$\begin{aligned} -\max_{x\in X} |f(x) - g(x)| &\leq -|f(x_g) - g(x_g)| \\ &\leq f(x_g) - g(x_g) \\ &\leq f(x_f) - g(x_g) \\ &\leq f(x_f) - g(x_f) \\ &\leq |f(x_f) - g(x_f)| \end{aligned}$$

Corps de la démonstration

La définition des fonctions de valeur optimales et l'inégalité triangulaire nous permettent d'écrire :

$$\begin{aligned} E_{app}(s) &= |\widehat{V}^{\widehat{\pi}^*}(s) - V^{\pi^*}(s)| \\ &= |\widehat{B}^*.\widehat{V}^{\widehat{\pi}^*}(s) - B^*.V^{\pi^*}(s)| \\ &\leq |\widehat{B}^*.\widehat{V}^{\widehat{\pi}^*}(s) - \widehat{B}^*.V^{\pi^*}(s)| + |\widehat{B}^*.V^{\pi^*}(s) - B^*.V^{\pi^*}(s)| \\ &= |\widehat{B}^*.\widehat{V}^{\widehat{\pi}^*}(s) - \widehat{B}^*.V^{\pi^*}(s)| + E_{int}(s) \end{aligned}$$

A l'aide du lemme nous développons le terme de gauche :

$$\begin{aligned} |\widehat{B}^*.\widehat{V}^{\widehat{\pi}^*}(s) - \widehat{B}^*.V^{\pi^*}(s)| &\leq \max_a \left| \gamma . \sum_{s'} \widehat{T}(s,a,s').(\widehat{V}^{\widehat{\pi}^*}(s') - V^{\pi^*}(s')) \right| \\ &= \max_a \left(\gamma . \sum_{s'} \widehat{T}(s,a,s'). \left| \widehat{V}^{\widehat{\pi}^*}(s') - V^{\pi^*}(s') \right| \right) \end{aligned}$$

On obtient alors :

$$E_{app}(s) \leq \max_a \left(\gamma . \sum_{s'} \widehat{T}(s,a,s').E_{app}(s') \right) + E_{int}(s)$$

Soit $\overline{E_{app}}$ une solution de l'équation ci-dessous :

$$\overline{E_{app}}(s) = \max_a \left(\gamma . \sum_{s'} \widehat{T}(s,a,s').\overline{E_{app}}(s') \right) + \overline{E_{int}}(s)$$

Alors on a clairement :

$$\forall s \in S, E_{app}(s) \leq \overline{E_{app}}(s) \ \Diamond$$

A.2 Propriétés de l'influence

A.2.1 Enoncé

Considérons une chaîne de Markov sur un espace S, de probabilité de transition $p(s_{t+1} = s'|s_t = s) = T(s,s')$. Etant donnée une fonction $y : S \to \mathbb{R}$, considérons la fonction $x : S \to \mathbb{R}$ implicitement définie par l'équation de Bellman suivante :

$$x(s) = y(s) + \gamma . \sum_{s'} T(s,s').x(s') \tag{A.1}$$

Définition 13 (Influence sur un état)
L'influence de l'état s sur l'état s' est définie comme étant la contribution de $y(s)$ au calcul de $x(s')$:

$$I_T(s|s') = \frac{\partial x(s')}{\partial y(s)}$$

L'influence vérifie les trois propriétés suivantes :

1. L'influence $I_T(s|s')$ d'un état s sur un état s' est la somme pondérée des probabilités de présence dans l'état s sachant que le processus est dans l'état s' à $t = 0$. Si on note $(s_t)_{(t \in \mathbb{N})}$ le processus, on a :
$$I_T(s|s') = \sum_{t=0}^{\infty} \gamma^t . p(s_t = s|s_0 = s')$$
2. La notation $I_T(s|s')$ est cohérente car l'influence dépend uniquement de la probabilité de transition T (et non de y). En particulier elle vérifie l'équation récursive suivante :
$$\forall (s,s') \in S^2, \ I_T(s|s') = \delta_{s,s'} + \gamma . \sum_{s''} T(s'',s).I_t(s''|s')$$
avec
$$\forall (s,s') \in S^2, \ \delta_{s,s'} = \begin{cases} 1 & \text{si } s = s' \\ 0 & \text{si } s \neq s' \end{cases}$$
3. Soit $I^{(0)}$ une fonction quelconque sur S. Alors l'influence sur s' $I(.|s')$ est l'unique limite de la suite $(I^{(n)})_{(n \in \mathbb{N})}$ définie par la relation de récurrence suivante :
$$\forall s \in S, \ I^{(n+1)}(s) = \delta_{s,s'} + \gamma . \sum_{s''} T(s'',s).I^{(t)}(s'') \tag{A.2}$$

A.2.2 Démonstrations

Voici les démonstrations des trois propriétés :

1. En développant l'équation A.1, on a pour toute séquence $(s^{(0)}, s^{(1)}, ...) \in S^{\mathbb{N}}$:

$$\begin{aligned} x(s^{(0)}) &= y(s^{(0)}) + \gamma . \sum_{s^{(1)}} T(s^{(0)}, s^{(1)}) . \left(y(s^{(1)}) + \gamma . \sum_{s^{(2)}} T(s^{(1)}, s^{(2)}) . \left[y(s^{(2)}) + ... \right] \right) \\ &= \sum_{t=0}^{\infty} \sum_{(s^{(1)},..,s^{(t)})} \gamma^t . \left[T(s^{(0)}, s^{(1)}) . T(s^{(1)}, s^{(2)}) ... T(s^{(t-1)}, s^{(t)}) \right] . y(s_t) \\ &= \sum_{t=0}^{\infty} \gamma^t . p(s_t = s^{(t)} | s_0 = s^{(0)}) . y(s_t) \end{aligned}$$

   Nous pouvons donc en déduire que :

$$I_T(s|s^{(0)}) = \frac{\partial x(s^{(0)})}{\partial y(s)} = \sum_{t=0}^{\infty} \gamma^t . p(s_t = s | s_0 = s^{(0)}) \ \Diamond$$

2. En utilisant cette première propriété de l'influence et le caractère stationnaire de la chaîne de Markov, on démontre facilement la deuxième propriété :

$$\begin{aligned} I_T(s|s') &= \sum_{t=0}^{\infty} \gamma^t . p(s_t = s | s_0 = s') \\ &= p(s_0 = s | s_0 = s') + \sum_{t=1}^{\infty} \gamma^t . p(s_t = s | s_0 = s') \\ &= \delta_{s,s'} + \sum_{t=0}^{\infty} \gamma^{t+1} . p(s_{t+1} = s | s_0 = s') \\ &= \delta_{s,s'} + \gamma . \sum_{s''} p(s_{t+1} = s | s_t = s'') . \left(\sum_{t=0}^{\infty} \gamma^t . p(s_t = s'' | s_0 = s') \right) \\ &= \delta_{s,s'} + \gamma . \sum_{s''} T(s'', s) . I_T(s''|s') \ \Diamond \end{aligned}$$

3. L'équation A.2 est une contraction si on utilise la norme $\|f\|_1 = \sum_{s \in S} |f(s)|$. En effet :

$$\begin{aligned} \|I^{(n+1)} - I_T(.|s')\|_1 &= \sum_{s \in S} \left| \gamma . \sum_{s''} T(s'', s) . (I^{(n)}(s'') - I_T(s''|s')) \right| \\ &= \gamma . \sum_{s,s''} T(s'', s) . |I^{(n)}(s'') - I_T(s''|s')| \\ &= \gamma . \|I^{(n)} - I_T(.|s')\|_1 \end{aligned}$$

   La suite $(I^{(n)})_{(n \in \mathbb{N})}$ a donc une unique solution qui est $I_T(.|s')$. $\Diamond$

Bibliographie

[Barnden, 1995] J.A. Barnden. Artificial intelligence and neural networks. Dans *The Handbook of Brain Theory and Neural Networks*, rédacteur M.A. Arbib, pages 98–102. Cambridge, Mass. : Bradford Books/MIT Press, 1995.

[Baxter *et al.*, 1999] J. Baxter, L. Weaver et P.L. Bartlett. Direct gradient-based reinforcement learning : II. gradient descent algorithms and experiments. Rapport technique, Research School of Information Sciences and Engineering, Australian National University, 1999.

[Bellman, 1957] R. E. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[Bellman, 1958] R. Bellman. On a routing problem. *Quarterly of Applied Mathematic*, 16(1) :87–90, 1958.

[Bertsekas et Tsitsiklis, 1989] D.P. Bertsekas et J. N. Tsitsiklis. *Parallel and Distributed Computation : Numerical Methods.* Prentice hall, 1989.

[Bertsekas et Tsitsiklis, 1996] D.P. Bertsekas et J.N. Tsitsiklis. *Neurodynamic Programming.* Athena Scientific, 1996.

[Bertsekas, 1983] D.P. Bertsekas. Distributed asynchronous computation of fixed points. *Math. Programming*, 27 :107–120, 1983.

[Boniface, 2000] Y. Boniface. *Etude et développement d'une bibliothèque d'adaptation du parallélisme neuromimétique au parallélisme MIMD.* PhD thesis, Université Henri Poincaré - Nancy 1, 2000.

[Bottou et Bengio, 1995] L. Bottou et Y. Bengio. Convergence properties of the K-means algorithms. Dans *Advances in Neural Information Processing Systems*, rédacteurs G. Tesauro, D. Touretzky et T. Leen, volume 7, pages 585–592. The MIT Press, 1995.

[Bottou, 1991] L. Bottou. *Une approche Théorique de l'Apprentissage Connectioniste : Applications à la Reconnaissance de la Parole.* PhD thesis, Université de Paris XI, 1991.

[Coulom, 2002] R. Coulom. *Apprentissage par renforcement utilisant des réseaux de neurones, avec des applications au contrôle moteur.* PhD thesis, Ecole doctorale Ingénierie pour le vivant : santé, cognition, environnement, 2002.

[Crabbe *et al.*, 1997] R. Crabbe, A. Dubey et M. Kantrowitz. Artificial Intelligence FAQ : General questions and answers. URL : http ://www.faqs.org/faqs/ai-faq/general/part1/, 1997.

[Crites et Barto, 1996] Robert H. Crites et Andrew G. Barto. Improving elevator performance using reinforcement learning. Dans *Advances in Neural Information Processing Systems*, rédacteurs David S. Touretzky, Michael C. Mozer et Michael E. Hasselmo, volume 8, pages 1017–1023. The MIT Press, 1996.

[Diday, 1971] E. Diday. Une nouvelle méthode en classification automatique et reconnaissance des formes : la méthode des nuées dynamiques. *Revue de Statistique Appliquée*, 2(18), 1971.

[Diday, 1979] E. Diday. *Optimisation en classification automatique.* INRIA, 1979.

[Dutech, 1999] A. Dutech. *Apprentissage d'environnements : approches cognitives et comportementales.* PhD thesis, Ecole Nationale Superieure de l'Aeronautique et de l'Espace, Toulouse, France, 1999.

[Fabiani *et al.*, 2001] P. Fabiani, J. L. Farges et F. Garcia. Décision dans l'incertain, polycopié supaero, 2001.

[Finton, 2002] D.J. Finton. *Cognitive economy and the role of representation in on-line learning.* PhD thesis, University of Wisconsin, 2002.

[Forgy, 1965] E. Forgy. Cluster analysis of multivariate data : efficiency versus interpretability of classifications. Dans *Biometrics*, volume 21, page 768, 1965.

[Frezza-Buet, 1999] H. Frezza-Buet. *Un modèle de cortex pour le comportement motivé d'un agent neuromimétique autonome.* PhD thesis, Université Henri Poincaré - Nancy 1, 1999.

[Friedman *et al.*, 1977] J. H. Friedman, J. L. Bentley et R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *TOMS*, 3(3) :209–226, 1977.

[Fritzke, 1997] B. Fritzke. Some competitive learning methods, 1997.

[Gersho et Gray, 1992] A. Gersho et R. M. Gray. *Vector Quantization and Signal Compression.* Kluwer Academic Publishers, Boston, 1992.

[Gordon, 1999] G. Gordon. Approximate solutions to markov decision processes, 1999.

[Haykin, 1994] S. Haykin. *Neural Networks : A Comprehensive Foundation.* NY : Macmillan, 1994.

[Hebb, 1949] D.O. Hebb. *The organization of behavior : a neuropsychological theory.* John Wiley & Sons, New York, 1949.

[Hertz *et al.*, 1991] J. Hertz, A. Krogh et R. Palmer. *Introduction to the theory of neural computation.* Addison-Wesley, Reedwood City, 1991.

[Jaakkola *et al.*, 1994] T. Jaakkola, M. I. Jordan et S. P. Singh. Convergence of stochastic iterative dynamic programming algorithms. Dans *Advances in Neural Information Processing Systems*, rédacteurs Jack D. Cowan, Gerald Tesauro et Joshua Alspector, volume 6, pages 703–710. Morgan Kaufmann Publishers, Inc., 1994.

[Kaelbling *et al.*, 1996] L. P. Kaelbling, M. L. Littman et A. P. Moore. Reinforcement learning : A survey. *Journal of Artificial Intelligence Research*, 4 :237–285, 1996.

[Kaelbling, 1993] L. P. Kaelbling. *Learning in Embedded Systems.* MIT Press, 1993.

[Kearns *et al.*, 2000] M. Kearns, Y. Mansour et A. Y. Ng. Approximate planning in large POMDPs via reusable trajectories. Dans *Advances in Neural Information Processing Systems 12.* MIT Press, 2000.

[Kohonen, 1988] T. Kohonen. *Self-Organization and Associative Memory.* Springer-Verlag, 1988.

[Kumar, 1985] P. R. Kumar. A survey of some results in stochastic adaptive control. *SIAM Journal of Control and Optimization*, 23 :329–38, 1985.

[Laroche, 2000] P. Laroche. *Processus Décisionnels de Markov appliqués à la planification sous incertitudes.* PhD thesis, Université Henri Poincaré - Nancy 1, 2000.

[Linde *et al.*, 1980] Y. Linde, A. Buzo et R.M. Gray. An algorithm for vector quantizer design. Dans *IEEE Trans. Communications*, pages 84–95, 1980.

[Littman *et al.*, 1995] M. L. Littman, T. L. Dean et L. P. Kaelbling. On the complexity of solving Markov decision problems. Dans *Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence (UAI–95)*, pages 394–402, Montreal, Québec, Canada, 1995.

[Littman, 1996] M. L. Littman. *Algorithms for Sequential Decision Making.* PhD thesis, Department of Computer Science, Brown University, 1996. Also Technical Report CS-96-09.

[McCallum, 1995] A. K. McCallum. *Reinforcement Learning with Selective Perception and Hidden State.* PhD thesis, Department of Computer Science University of Rochester Rochester, NY, 14627, USA, 1995.

[Meuleau et Bourgine, 1999] N. Meuleau et P. Bourgine. Exploration of multi-state environments : Local measures and back-propagation of uncertainty. *Machine Learning*, 35(2) :117–154, 1999.

[Moore, 1994] A. W. Moore. The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. Dans *Advances in Neural Information Processing Systems*, rédacteurs Jack D. Cowan, Gerald Tesauro et Joshua Alspector, volume 6, pages 711–718. Morgan Kaufmann Publishers, Inc., 1994.

[Munos et Moore, 2000] R. Munos et A. Moore. Rates of convergence for variable resolution schemes in optimal control. Dans *International Conference on Machine Learning*, 2000.

[Munos et Moore, 2002] R. Munos et A. Moore. Variable resolution discretization in optimal control. *Machine Learning Journal*, 49 :291–323, 2002.

[Munos, 1997a] R. Munos. *Apprentissage par renforcement, étude du cas continu.* PhD thesis, Ecole des Hautes Etudes en Sciences Sociales, 1997.

[Munos, 1997b] R. Munos. Catégorisation adaptative de données sensori-motrices pour un système d'apprentissage par renforcement. Dans *Journées de Rochebrune 1997, Rencontres interdisciplinaires sur les systèmes complexes naturels et artificiels. Invariance, Interaction, Référence : L'identité en questions.*, 1997.

[Nakamura, 1998] T. Nakamura. Self-organizing internal representation for behavior acquisition of vision-based mobile robot. Dans *From animals to animats 5 : The fifth conference on the Simulation of Adaptive Behavior*, Zurich, Switzerland, August 1998.

[Nigrin, 1993] A. Nigrin. *Neural Networks for Pattern Recognition.* Cambridge MA : The MIT Press, 1993.

[Peng et Williams, 1993] J. Peng et R. J. Williams. Efficient learning and planning within the DYNA framework. Dans *Proceedings of the 2nd International Conference on Simulation of Adaptive Behavior, Hawaii*, 1993.

[Queen, 1967] J. Mac Queen. Some methods for classifications and analysis of multivariate observations. Dans *Fifth Berkeley Symposium on Mathematical statistics and probability*, volume I, pages 281–297. Berkeley, University of California Press, 1967.

[Reiss et Taylor, 1991] M. Reiss et J.G. Taylor. Storing temporal sequences. *Neural Networks*, 4 :773–778, 1991.

[Reynolds, 2001] Stuart I. Reynolds. Adaptive representation methods for reinforcement learning. *Lecture Notes in Artificial Intelligence Series*, 2056 :345–348, 2001.

[Rougier, 2000] N. Rougier. *Modèles de mémoire pour la navigation autonome.* PhD thesis, Université Henri Poincaré - Nancy 1, 2000.

[Rumelhart *et al.*, 1986] D. E. Rumelhart, G. E. Hinton et R. J. Williams. Learning internal representations by error propagation. Dans *Parallel Distributed Processing : Explorations in the Microstructure of Cognition*, rédacteurs D. E. Rumelhart et J. L. McClelland, volume 1. MIT Press, 1986.

[Sarle, 1997] W.S. Sarle. Neural Network FAQ, part 1 of 7 : Introduction, periodic posting to the usenet newsgroup comp.ai.neural-nets. URL : ftp ://ftp.sas.com/pub/neural/FAQ.html, 1997.

[Scherrer, 2002] B. Scherrer. A connectionist architecture that adapts its representation to complex tasks. Dans *International Joint Conference on Neural Networks*, Mai 2002.

[Scheuer, 1998] A. Scheuer. *Planification de chemins à courbure continue.* PhD thesis, Institut National Polytechnique de Grenoble, 1998.

[Singh et Bertsekas, 1997] Satinder Singh et Dimitri Bertsekas. Reinforcement learning for dynamic channel allocation in cellular telephone systems. Dans *Advances in Neural Information Processing Systems*, rédacteurs Michael C. Mozer, Michael I. Jordan et Thomas Petsche, volume 9, page 974. The MIT Press, 1997.

[Smart, 2002] William D. Smart. *Making Reinforcement Learning Work on Real Robots.* PhD thesis, Department of Computer Science, Brown University, May 2002.

[Sutton et Barto, 1998] R.S. Sutton et A.G. Barto. *Reinforcement Learning, An introduction.* BradFord Book. The MIT Press, 1998.

[Sutton, 1990] R. Sutton. First results with DYNA, an intergrated architecture for learning, planning, and reacting. Dans *AAAI Spring Symposium on Planning in Uncertain, Unpredictable, or Changing Environments*, 1990.

[Sutton, 1991] R. S. Sutton. DYNA, an Integrated Architecture for Learning, Planning and Reacting. Dans *Working Notes of the AAAI Spring Symposium on Integrated Intelligent Architectures*, 1991.

[Sutton, 1996] R. S. Sutton. Generalization in Reinforcement Learning : Successful Examples Using Sparse Coarse Coding. Dans *Advances in Neural Information Processing Systems*, rédacteurs David S. Touretzky, Michael C. Mozer et Michael E. Hasselmo, volume 8, pages 1038–1044. The MIT Press, 1996.

[Tesauro, 1995] G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3) :58–68, 1995.

[Touzet, 1992] C. Touzet. *Les réseaux de neurones artificiels, Introduction au connexionnisme.* EC2 éditeur, Paris, 1992.

[Watkins, 1989] C. Watkins. *Learning with delayed rewards.* PhD thesis, Cambridge University, 1989.

[Wiering *et al.*, 1998] M. A. Wiering, R. P. Salustowicz et J. Schmidhuber. Cmac models learn to play soccer. Dans *Proceedings of the 8th International Conference on Artificial Neural Networks*, rédacteurs L. Niklasson, M. Boden et T. Ziemke, volume 1, pages 443–448. Springer-Verlag, 1998.

[Williams et Baird, III, 1990] R. J. Williams et L. C. Baird, III. A mathematical analysis of actor-critic architectures for learning optimal controls through incremental dynamic programming. Dans *Proc. 6th Yale Workshop on Adaptive and Learning Systems, August 1990, New Haven, CT*, pages 96–101, 1990.

[Williams et Baird, 1993] R. Williams et L. Baird. Tight performance bounds on greedy policies based on imperfect value functions. Northeastern University Technical Report NU-CCS-93-14, November, 1993.

Résumé

Cette thèse étudie l'utilisation d'algorithmes connexionnistes pour résoudre des problèmes d'apprentissage par renforcement. Les algorithmes connexionnistes sont inspirés de la manière dont le cerveau traite l'information : ils impliquent un grand nombre d'unités simples fortement interconnectées, manipulant des informations numériques de manière distribuée et massivement parallèle. L'apprentissage par renforcement est une théorie computationnelle qui permet de décrire l'interaction entre un agent et un environnement : elle permet de formaliser précisément le problème consistant à atteindre un certain nombre de buts via l'interaction.

Nous avons considéré trois problèmes de complexité croissante et montré qu'ils admettaient des solutions algorithmiques connexionnistes : 1) L'apprentissage par renforcement dans un petit espace d'états : nous nous appuyons sur un algorithme de la littérature pour construire un réseau connexionniste ; les paramètres du problème sont stockés par les poids des unités et des connexions et le calcul du plan est le résultat d'une activité distribuée dans le réseau. 2) L'apprentissage d'une représentation pour approximer un problème d'apprentissage par renforcement ayant un grand espace d'états : nous automatisons le procédé consistant à construire une partition de l'espace d'états pour approximer un problème de grande taille. 3) L'auto-organisation en modules spécialisés pour approximer plusieurs problèmes d'apprentissage par renforcement ayant un grand espace d'états : nous proposons d'exploiter le principe "diviser pour régner" et montrons comment plusieurs tâches peuvent être réparties efficacement sur un petit nombre de modules fonctionnels spécialisés.

Mots-clés: intelligence artificielle, apprentissage par renforcement, connexionnisme, processus décisionnels de Markov

Abstract

This thesis studies the use of connectionist algorithms for solving reinforcement learning problems. Connectionist algorithms are inspired by the way information is processed by the brain : they rely on a large network of highly interconnected simple units, which process numerical information in a distributed and massively parallel way. Reinforcement learning is a computational theory that describes the interaction between an agent and an environment : it enables to precisely formalize goal-directed learning from interaction.

We have considered three problems, with increasing complexity, and shown that they can be solved with connectionist algorithms : 1) Reinforcement learning in a small state space : we exploit a well-known algorithm in order to build a connectionist network : problem paramaters are stored into weighted units and connections and planning is the result of a distributed activity in the network. 2) Learning a representation for approximating a reinforcement learning problem with a large state space : we provide an algorithm for automatically building a state space partition in order to approximate a large problem. 3) Self-organization of specialized modules for approximating various reinforcement problems with a large state space : we exploit a "divide and conquer" approach and show that various tasks can efficiently be spread over a little number of specialized functional modules.

Keywords: artificial intelligence, reinforcement learning, connexionism, Markov decision processes