UNIVERSITÉ SULTAN MOULAY SLIMANE
Faculté des Sciences et Techniques
Béni Mellal

Centre d'Etudes Doctorales : Sciences et Techniques
Formation doctorale : Mathématiques et Physique Appliquées

THESE
Présentée par

Abdelhadi Larach

Pour l'obtention du grade de
Docteur
Spécialité : Informatique – Electronique

Contribution à la Résolution des Processus Décisionnels de Markov : Applications à la Robotique

---

JURY

Président :	Pr. Belaid BOUIKHALENE,	FP - Beni Mellal.
Rapporteurs :	Pr. Hamid El MAROUFY,	FST - Beni Mellal.
	Pr. Abdelmoutalib METRANE,	ENSA - Khouribga.
	Pr. Khalid RAHHALI,	FSR - Rabat.
Examinateurs :	Pr. Abdessamad MALAOUI,	FP - Beni Mellal.
	Pr. Mohammed FAKIR,	FST - Beni Mellal.
	Pr. Mohammed BASLAM,	FST - Beni Mellal.
Directeur de thèse :	Pr. Cherki DAOUI,	FST - Beni Mellal.

Soutenue le 10 janvier 2020

Remerciements

*Les travaux présentés dans cette thèse ont été effectués au sein du Laboratoire Traitement d'Information et Aide à la Décision (**TIAD**) affilié au Centre d'Etudes Doctorales de la Faculté des Sciences et Techniques (FST) à l'Université Sultan Moulay Slimane de Beni-Mellal.*

En premier lieu, je remercie le Bon Dieu de m'avoir donné la force et le courage pour accomplir ce travail et qui m'a procuré ce succès.

Mes remerciements s'adressent spécialement, à mon directeur de thèse, le Professeur Cherki DAOUI, pour le temps et la patience qui m'a accordé tout au long de ces années, et qui s'est toujours montré disponible et à l'écoute pendant toutes les étapes de l'élaboration de ce travail. Je tiens à lui témoigner ma gratitude et ma reconnaissance les plus sincères.

Je présente mes sincères remerciements aux rapporteurs et examinateurs de cette thèse qui ont mis à ma disposition leurs compétences et ont contribué par leurs remarques à l'amélioration de ce travail de recherche.

Aussi, j'adresse mes remerciements à tous les membres du laboratoire TIAD et à tous les enseignants des départements d'informatique et de physique, de la Faculté des Sciences et Techniques.

Enfin, il m'est agréable d'exprimer ma reconnaissance auprès de toutes les personnes qui ont participé de près ou de loin au bon déroulement de cette thèse.

Résumé

Cette thèse porte sur les Processus Décisionnels de Markov (PDM) qui offrent un modèle mathématique simple et un ensemble d'algorithmes permettant de résoudre des problèmes de décisions séquentielles dans l'incertain. La complexité de ces algorithmes croît exponentiellement en fonction de la taille de l'espace d'états, ce qu'on appelle en littérature la malédiction de la dimension, cela limite grandement l'utilisation des PDM dans la plupart des problèmes réels. En outre, pour le problème du plus court chemin stochastique avec des impasses, qui est modélisé sous forme d'un PDM, la convergence de ses algorithmes de résolution ne peut être assurée.

En littérature, les techniques les plus utilisées pour remédier au problème des PDM de grandes dimensions, sont les méthodes hiérarchiques (ou topologiques) qui consistent à : (i) décomposer l'espace d'états en Composantes Fortement Connexes (CFC), (ii) résoudre les PDM restreints correspondants à chaque CFC et combiner leurs solutions pour obtenir une solution globale.

Néanmoins, la procédure de décomposition risque lui aussi de croître exponentiellement en fonction de la taille de l'espace d'états. À travers nos contributions, nous proposons une nouvelle technique de décomposition optimisée basée sur l'algorithme de Tarjan, qui nous a permis de réduire l'espace d'états de chaque PDM restreint et de construire de nouveaux algorithmes hiérarchiques pour les PDM actualisés à espaces d'états et d'actions finis.

En outre, nous considérons la classe des PDM orientés but, à savoir le problème du plus court chemin stochastique avec des impasses. Nous présentons, d'une part, une transformation de son modèle PDM permettant de résoudre le problème de divergence des algorithmes de résolution et de répondre à la question d'insuffisance de ressources pour atteindre le but. D'autre part, nous proposons une nouvelle méthode de partition de l'espace d'états en niveaux d'accessibilité aux états buts, à chaque niveau correspondra un ou plusieurs PDM restreints dont les solutions seront combinées pour fournir une heuristique admissible du problème initial. Enfin, dans l'application de la couverture de zone par un robot démineur ou robot nettoyeur, etc., nous présentons un modèle PDM et un algorithme de résolution en ligne orienté but, offrant la possibilité de choisir le mode de balayage adéquat.

Par ailleurs, nous considérons les PDM inconnus. Nous proposons, tout d'abord, un nouvel algorithme d'apprentissage par renforcement (AR) guidé pour le problème de la navigation robotique dans un environnement inconnu. Ensuite, nous présentons deux nouveaux algorithmes d'AR pour la classe des PDM déterministes ayant une fonction de

récompenses inconnue. Ces algorithmes peuvent être utilisés dans différentes applications en robotique, telle que l'apprentissage à la marche robotique. Enfin, nous proposons deux nouveaux modèles d'AR, le premier appliqué au robot suiveur de ligne et le deuxième modèle appliqué au robot auto-balancé.

Finalement, des simulations et des expérimentations seront présentées pour montrer les performances des algorithmes proposés et leurs compétitivités avec les algorithmes de résolution classiques.

Mots clés : Processus Décisionnel de Markov, Apprentissage par Renforcement, Plus Court Chemin Stochastique, Décomposition, Méthodes Hiérarchiques, Théorie des Graphes, Robotique.

Abstract

This thesis focuses on Markov Decision Processes (MDP) that provides a simple mathematical model and a set of algorithms that solve sequential decision problems with uncertainty. The complexity of these algorithms increases exponentially with the size of the state space, known, in the literature, as the curse of dimensionality, which greatly limits the use of MDP in most real problems. In addition, in some classes of MDP, such as the stochastic shortest path problem with dead ends, the convergence of these algorithms cannot be guaranteed.

The most commonly used techniques to solve the problem of large dimensions are the hierarchical (or topological) methods based on the partition of the state space into strongly connected components (SCC) that can be classified into some levels. In each level, smaller problems named restricted MDPs are solved, and then these partial solutions are combined to obtain the global solution.

Through our contributions, we propose a novel algorithm, which is a variant of Tarjan's algorithm, which simultaneously finds the SCC and their belonging levels, and a new definition of the restricted MDP that permits to build new hierarchical algorithms for undiscounted MDP with finite state and action spaces.

In addition, we consider the goal-oriented MDP class, namely the Stochastic Shortest Path (SSP) problem with dead ends. Firstly, we propose a new Transformed SSP MDP that guarantees the convergence of the classical iterative algorithms and addresses the problem of energy-reachability. Secondly, we propose a topological algorithm based on a decomposition of the state space into goal state accessibility levels. In each level, will correspond one or more restricted PDM, whose solutions will be combined to provide a suitable heuristic solution used as an initial point in the Gauss-Seidel Value Iteration (GSVI) algorithm. Finally, an example of robot navigation in hexagonal grid's environment will be presented to show the advantages of the TSSP MDP and the performance of the proposed topological algorithm.

Moreover, we consider unknown MDP models. Firstly, we propose a new Guided Reinforcement Learning (RL) algorithm for robot navigation in an unknown environment. Secondly, we present two novels RL algorithms for the deterministic MDP class with an unknown reward function. These algorithms can be used in different applications in robotics,

such as learning to robotic walking. Finally, we exposed two RL models, the first applied to the follower robot and the second model to the self-balanced robot.

Finally, simulations and experiments results are presented to show the performances of the proposed algorithms and their competitiveness compared to classical algorithms.

Keywords: Markov Decision Problem, Reinforcement Learning, Stochastic Shortest Path, Decomposition, Hierarchical Methods, Graph Theory, Robotics.

Table des matières

Liste des figures

Chapitre 1

Chapitre 2

Chapitre 3

Chapitre 4

Chapitre 5

Liste des algorithmes

Liste des tableaux

Liste des abréviations

AR	: Apprentissage par Renforcement
BFD	: Breadth-First Search
CC	: Classe Communicante
CCD	: Charge Coupled Device
CFC	: Composante Fortement Connexe
CPU	: Central Processing Unit
DFS	: Depth First Search
IA	: Intelligence Artificielle
IP	: Itération de la Politique
IPM	: Itération de la Politique Modifié
IV	: Itération de la Valeur
IVA	: Itération de la Valeur Accéléré
IVGS	: Itération de la Valeur de Gauss-Seidel
IVGSD	: Itération de la valeur de Gauss-Seidel Déterministe
IVGST	: Itération de la valeur de Gauss-Seidel Topologique
IVHA	: Itération de la Valeur Hiérarchique Accéléré
IVR	: Itération de la Valeur Relative
LDC	: Liste Doublement Chainée
LRTDP	: Labelled Real-Time Dynamic Programming
MDP	: Markov Decision Processes
MPI	: Message Passing Interface
NPDMR	: Nouvel PDM Restreint
PDM	: Processus Décisionnel de Markov
PDMPO	: PDM Partiellement Observable
PL	: Programmation Linéaire
QLG	: Q-Learning Guidé
RAM	: Random Access Memory
ROM	: Read Only Memory
RTDP	: Real-Time Dynamic Programming
SSP	: Stochastic Shortest Path
SSPADE	: Stochastic Shortest Path with Avoidable Dead Ends
SSPUDE	: Stochastic Shortest Path with Unavoidable Dead Ends

Introduction Générale

1. Contexte et motivation

Nous nous intéressons, dans ce travail, à la conception d'intelligences agissantes aux prises avec leur monde extérieur. On parle alors d'agent rationnel en interaction avec son environnement, c'est-à-dire capable de le percevoir et d'agir sur celui-ci. Plus précisément, nous considérons des cadres dynamiques, dans lesquels l'agent doit prendre une séquence de décisions pour contrôler l'état du système qu'il constitue avec l'environnement. On parle alors de prise de décision séquentielle. Cela recouvre à la fois la planification d'actions dans un environnement dont on connaît la dynamique (les règles de fonctionnement), mais aussi l'apprentissage par renforcement (AR), c'est-à-dire l'apprentissage du comportement à adopter par essai-erreur.

Souvent, il faut tenir compte d'incertitudes pour traiter de tels problèmes. Il peut s'agir d'incertitudes inhérentes à la dynamique du système (à supposer qu'elle est par nature stochastique et non déterministe). Il peut s'agir d'incertitudes liées aux capacités perceptives limitées, l'agent n'ayant à chaque instant qu'une vue partielle et bruitée de l'état réel de l'environnement. Il peut s'agir d'incertitudes introduites dans le modèle pour en réduire la complexité. On parle alors de prise de décision séquentielle dans l'incertain.

Les Processus Décisionnels de Markov (PDM), sur lesquels reposent les travaux de ce mémoire, offrent un formalisme mathématique qui permet de modéliser et résoudre un grand nombre de problèmes de prise de décision séquentielle dans l'incertain sous la condition que l'état du système et le choix d'une meilleure décision ne dépendent que des situations coutantes (propriété de Markov). La démarche des PDM consiste à contrôler séquentiellement des systèmes dont la dynamique est incertaine, ce qui amène naturellement à voir ces systèmes comme des chaînes de Markov dont la dynamique dépend du choix d'une action. En outre, pour spécifier un objectif, une utilité est associée aux états visités et aux actions effectuées. Les solutions d'un PDM sont donc des décisions ou des séquences de décisions, appelées politiques, ou stratégies, qui spécifient l'action à entreprendre en chacune des étapes pour toutes les situations futures possibles d'un agent selon un critère d'optimalité donné. Ce critère a pour but de déterminer les politiques qui permettront de générer des séquences de récompenses les plus importantes possibles. Il existe trois algorithmes classiques de résolution des PDM : la Programmation Linéaire (PL), l'Itération de la Valeur (IV) et l'Itération de la Politique (IP).

Ainsi, les PDM se situent à l'intersection de trois principales disciplines, à savoir : la théorie du contrôle, la recherche opérationnelle et l'intelligence artificielle. La Théorie du Contrôle, qui est une discipline des mathématiques appliquées, a pour objectif l'étude des systèmes et la conception des composants de contrôle. La Recherche Opérationnelle est une branche interdisciplinaire des mathématiques et des sciences formelles qui fait appel aux méthodes de modélisation mathématique, aux statistiques et à l'algorithmique, afin d'aboutir à des solutions optimales ou presque optimales pour des problèmes complexes. Enfin, l'Intelligence Artificielle (IA) qui est un ensemble de méthodes permettant la conception d'entités intelligentes agissant de manière rationnelle appelées agents. Par exemple, un robot est un agent particulier qui possède des capteurs avec lesquels perçoit son environnement et des effecteurs avec lesquels agit sur son environnement, et qui possède un certain degré d'intelligence à partir du moment où il est capable de résoudre des problèmes réels.

Les domaines d'application des PDM sont très nombreux, ils sont utilisés en économie, dans les systèmes de maintenance, dans les systèmes de file d'attente, dans le domaine médical, [BOG99, BUO04, LEW02, LOZ13, PAV06] etc. En intelligence artificielle et plus précisément en robotique, les modèles des PDM sont très répandus, de la navigation ordinaire à la robotique dite développementale qui tente de donner aux robots la capacité de s'adapter à des environnements variés et inconnus de leurs concepteurs.

En outre, le problème de plus court chemin stochastique, défini par WHITE dans [WHI93] est un PDM ayant des coûts positifs (gains négatifs) et un état ou ensemble d'états absorbants (états buts). Il appartient à une classe particulière des PDM, appelée PDM orienté but, qui consiste à trouver une stratégie optimale atteignant le but ou le plus proche but, depuis l'état initial, avec une probabilité égale à 1 et une espérance de coût minimale. Cette condition est très forte et empêche de résoudre de nombreux problèmes intéressants, par exemple le cas où toutes les politiques possibles atteignent des états impasses "culs-de-sac" avec une probabilité strictement positive, l'agent ne peut toujours atteindre le but avec une probabilité de 100%. Ainsi, le problème devient multicritère : maximiser la probabilité d'atteindre le but et minimiser l'espérance du coût. Le problème du plus court chemin stochastique est largement utilisé dans le transfert d'informations dans un réseau, dans différents domaines du transport routier, etc.

2. Problématiques

La programmation dynamique, sur laquelle sont basés les algorithmes de résolution (solveurs) des PDM, est l'une des techniques les plus connues de planification, c'est une

méthode d'optimisation mathématique qui construit une solution optimale ou presque optimale à partir de sous solutions optimales afin de garantir l'optimalité de la solution globale, ce qu'on appelle en littérature le principe d'optimalité de Bellman. La programmation dynamique souffre néanmoins du problème de la malédiction de la dimension, identifié par Bellman en personne [BEL57]. En effet, dans la plus part des problèmes réels, l'espace d'état est de grande taille, les algorithmes de résolution classiques ne peuvent être appliqués. Il s'agit d'un phénomène selon lequel la complexité de la résolution d'un problème d'optimisation croît avec leur nombre de paramètres et la taille de leurs domaines respectifs. C'est ainsi que plusieurs recherches s'intéressent à résoudre ce problème en proposant différentes approches : les méthodes de décomposition qui consiste à subdiviser un problème en sous-problèmes plus faciles à résoudre ; les techniques d'agrégation qui réduisent l'espace d'états ; la programmation dynamique approchée ; les techniques d'élimination d'actions qui réduisent l'espace d'actions ; et le parallélisme.

La technique de décomposition du PDM initial en petits problèmes appelés PDM restreints, inspirée de la méthode algorithmique « diviser pour régner » et des concepts de la théorie des graphes, a été introduite par Ross et Varadarajan [ROS91] pour la classe des PDM avec contraintes. Elle a était réutilisée et améliorée par Abbad et Boustique [ABD03a]; Abbad et Daoui [ABD03b]; Daoui et Abbad [DAO07, DAO10]; Dai et Goldsmith [DAI11] pour différentes catégories de PDM et pour différents critères d'optimalité. Cette technique consiste à : (1) diviser l'espace d'états en petits sous-espaces ; (2) résoudre les problèmes restreints correspondant à chaque sous-espace ; (3) combiner ces solutions locales pour obtenir une solution du problème global. Cependant, cette technique ne peut pas s'appliquer pour tout modèle PDM, notamment lorsque le problème global se décompose en un petit nombre de problèmes restreints. Elle souffre aussi de la complexité temporelle de la procédure de décomposition.

D'ailleurs, la programmation dynamique ne peut être appliquée aux modèles inconnus ou partiellement connus, qui nécessitent des algorithmes d'apprentissage par renforcement et qui souffrent eux aussi de la malédiction de la dimension. Un robot, par exemple, doit être capable d'organiser ses plans afin de répondre au mieux aux rétroactions imprévues de l'environnement. Souvent, il aura à construire un plan sur la base d'informations incomplètes et corriger par la suite son comportement lors de l'exécution de ce plan. Il procède à une technique d'apprentissage par essais et erreurs afin de converger vers une séquence de décisions optimales.

Dans cette thèse, nous avons aussi considéré la classe des PDM orientés but, plus précisément le problème du plus court chemin stochastique avec des impasses, où se pose les problématiques : de la convergence des algorithmes de résolution, de l'insuffisance énergétique pour atteindre le but, et du problème multicritères (maximiser la probabilité d'atteindre le but et minimiser l'espérance du coût).

3. Contributions de la thèse

Nos premières contributions à la résolution des PDM actualisés, sont fondées sur la méthode hiérarchique proposée par Abbad et Daoui [ABD03a, ABD03b], et qui consiste à : (1) décomposer l'espace d'états en Composantes Fortement Connexes (CFC) ; (2) trouver les niveaux d'appartenance de chaque CFC ; (3) résoudre les PDM restreints correspondant à chaque CFC, puis combiner leurs solutions pour obtenir une solution globale. Chercher un algorithme efficace de décomposition est d'une importance majeure, c'est ainsi que nous proposons, en premier temps, une nouvelle variante de l'algorithme de Tarjan [TAR72] permettant de déterminer les CFC et un nouvel algorithme qui permet de trouver simultanément les CFC et leurs niveaux d'appartenance. Ensuite, nous exposons une nouvelle définition des PDM restreints qui permet de réduire l'espace d'états et d'accélérer l'algorithme IV hiérarchique. Enfin, pour optimiser plus le temps de calcul, nous présentons des versions parallèles de l'algorithme IV et l'algorithme IV hiérarchique. Cette technique de décomposition des PDM est intéressante pour les deux raisons suivantes :

- elle permet d'alléger la tâche de résolution des PDM actualisés de grande taille.
- elle accepte la recherche d'une action optimale, pour un état fixé, en ne résolvant que quelques PDM restreints (sans utiliser tout le PDM initial).

En outre, nous proposons une nouvelle transformation du modèle PDM pour le problème du plus court chemin stochastique avec des états impasses, permettant de résoudre le problème de convergence de ses algorithmes de résolution et de répondre à la question d'insuffisance des ressources (énergie, temps,..) pour atteindre le but. En plus, pour accélérer les solveurs de ce problème, nous proposons une nouvelle technique de décomposition du problème original basée sur la partition de l'espace d'états en niveaux d'accessibilité aux états buts. Nous proposons également un modèle PDM dédié à la couverture de zone d'un robot démineur qui est un problème classique du plus court chemin stochastique. Enfin, nous présentons deux versions d'un algorithme de résolution en ligne offrant la possibilité de choisir le mode de balayage adéquat.

Dans nos dernières contributions en AR, nous proposons, tout d'abord, un nouvel algorithme d'apprentissage par renforcement guidé appliqué à la navigation robotique dans un environnement inconnu. Cet algorithme permet de réduire la zone de l'environnement à explorer et d'accélérer ainsi le processus d'apprentissage. Ensuite, nous présentons des méthodes d'AR pour la classe des PDM déterministes ayant une fonction de récompenses inconnue. En fait, nous proposons deux nouveaux algorithmes d'AR appliquées à la marche robotique, le premier algorithme direct basé sur la couverture de zone et le deuxième indirect basé sur l'élimination des actions non optimales. Nous présentons également deux nouveaux modèles d'AR appliqués au robot suiveur de ligne et au robot auto-balancé. Enfin, pour faire une évaluation expérimentale des modèles proposés, nous avons réalisé, au sein de notre laboratoire (TIAD), trois petits robots : robot escargot-mécanique, robot suiveur de ligne et robot auto-balancé.

Finalement, les simulations réalisées et discutées des méthodes proposées, dans ce mémoire, montrent qu'elles ont donné une très bonne satisfaction comparativement à de nombreuses techniques de la littérature.

4. Plan du Mémoire

Le contenu principal de la thèse est divisé en cinq chapitres et une conclusion générale. Même s'ils peuvent être lus indépendamment, leur lecture dans l'ordre présenté est recommandée, car la terminologie et la notation définie dans les premiers chapitres peuvent être omises dans les chapitres suivants. Le **premier chapitre** présente une introduction aux bases et aux outils de la théorie des PDM complètement observables, ses critères d'optimalité les plus utilisés et ses différents algorithmes de résolution classiques. Par la suite, il expose quelques techniques pour remédier au problème des grandes dimensions : la technique d'élimination des actions, la réduction de l'espace d'états, la méthode de décomposition et le parallélisme. Ensuite, il présente, brièvement, les PDM Partiellement Observables (PDMPO) et enfin les algorithmes d'apprentissage par renforcement appliqués aux modèles des PDM inconnus.

Dans le **deuxième chapitre**, l'agent robot sera présenté avec ses différents constituants, les types de représentations de l'environnement, ainsi que certains types de microcontrôleurs pouvant être utilisés pour le contrôler. Ensuite, nous présentons quelques exemples de modèles PDM appliqués en robotique.

Dans le **troisième chapitre**, nous exposons nos contributions pour la résolution des PDM sous le critère d'actualisation. Nous commençons par présenter une nouvelle variante de l'algorithme de Tarjan [TAR72] permettant de déterminer les CFC, ainsi qu'un nouvel algorithme qui permet de trouver simultanément les CFC et leurs niveaux d'appartenance. Ensuite, nous proposons une nouvelle définition des PDM restreints et des versions séquentielles et parallèles des solveurs hiérarchiques.

Le **quatrième chapitre** présente, d'abord, une nouvelle transformation du modèle PDM pour le problème du plus court chemin stochastique avec des impasses. Nous proposons, également, des solveurs accélérés de ce problème qui sont basés sur une nouvelle technique de décomposition du problème original en sous problèmes. Enfin, un modèle PDM appliqué à la couverture de zone d'un robot démineur sera présenté.

Dans le **cinquième chapitre**, nous proposons, tout d'abord, un nouvel algorithme d'AR guidé pour le problème de la navigation robotique dans un environnement inconnu. Ensuite, nous proposons deux nouveaux algorithmes d'AR appliquées à la marche robotique. Enfin, nous exposons deux nouveaux modèles d'AR, le premier appliqué au robot suiveur de ligne et le deuxième modèle appliqué au robot auto-balancé.

À la fin de ce document, nous présentons une **conclusion générale** dédiée à la synthèse des travaux effectués, ainsi que les principaux résultats obtenus. Nous finissons notre mémoire par une présentation des perspectives sur les futurs travaux en soulignant les points importants.

Liste des Publications

- Larach, A. Chafik, S. & Daoui, C. (2017) : Accelerated decomposition techniques for large discounted Markov decision processes, Journal of Industrial Engineering International, ISSN 2251-712X, Springer, Heidelberg, Vol. 13, Iss. 4, pp. 417-426, http://dx.doi.org/10.1007/s40092-017-0197-7

- Larach, A. Daoui, C. & Baslam, M. (2017). A Markov Decision Model for Area Coverage in Autonomous Demining Robot. International Journal of Informatics and Communication Technology (IJ-ICT) Vol. 6, No. 2, pp. 105~116 ISSN: 2252-8776, DOI: 10.11591/ijict.v6i2.pp105-116.

- Chafik, S., Larach, A. & Daoui, C. (2018). Parallel Hierarchical Pre-Gauss-Seidel Value Iteration Algorithm. International Journal of Decision Support System Technology, Volume 10, Issue 2. DOI: 10.4018/IJDSST.2018040101.

- Larach, A. & Daoui, C. (2019). A New Transformed Stochastic Shortest Path for Dead Ends and Energy Constraint. International Journal of Advanced Science and Technology, Vol. 129, pp. 43-58, http://dx.doi.org/10.33832/ijast.2019.129.04.

- Larach, A. & Daoui, C. A Topological Guided Reinforcement Learning Algorithm for Robot Navigation in an Unknown environment. Submitted to International Journal of Advanced in Applied Science.

- Larach, A. & Daoui, C. A Reinforcement Learning Algorithms for Deterministic Markov Decision Process with Unknown Reward Function. Submitted to Robotics and Biomimetic.

Liste des Communications

- Modified Tarajn's Algorithms to Solve Large Markov Decision Processes. Numerical Analysis and Optimization Days. JANO'16. April 27-29, 2016, Beni Mellal.

- A Markov Decision Model for Area Coverage in Autonomous Demining Robot. Third International Conference on Business Intelligence. CBI'17, March 31, 2017, Beni Mellal.

- Topological Reinforcement Learning Algorithm for Robot Navigation in an Initially Unknown Environment. 4th International Conference on Business Intelligence. CBI'18, April 26, 2018, Beni Mellal.

- Application of the Fixed-Point Method for Solving Stochastic Shortest Path for Dead Ends and Energy Constraint. Workshop international sur la théorie des points fixes et applications. WIFPTA 19, April 05-06, 2019, Beni Mellal.

- Solving Markov Decision Processes with Decomposition Technique. 5th International Conference on Business Intelligence. CBI'19, April 19, 2019, Beni Mellal.

Chapitre 1

Problèmes de Décision Markoviens et Algorithmes de Résolution

1.1. Introduction

Les Problèmes de Décision Markoviens (PDM) couplent à la fois les deux problématiques de la décision séquentielle et de la décision dans l'incertain. La décision séquentielle s'inscrit dans la durée et la décision dans l'incertain est liée à l'incertitude des conséquences mêmes de chacune des décisions possibles. Ainsi, l'agent ne sait que des lois de probabilités des effets des décisions qu'il prend. Les PDM intègrent les notions d'état qui définit la situation de l'agent à chaque instant, d'action (ou décision) qui contrôle la dynamique de l'état, de récompense qui est associée à chacune des transitions d'état.

Résoudre un PDM, c'est contrôler l'agent pour qu'il se comporte de manière optimale, de tel sorte qu'il optimise son revenu ou son coût. Les solutions d'un PDM sont donc des décisions ou séquences de décisions, appelées politiques (ou stratégies), qui spécifient l'action à entreprendre en chacune des étapes pour toutes les situations futures possibles de l'agent.

Dans ce chapitre, nous exposons d'abord le cadre théorique des PDM [BEL57, BER87, PUT94] en se limitant aux problèmes mono agent de prise de décision séquentielle à temps discrets T = {0, l,…N} et à horizon fini (N<∞) ou infini (N=∞). Puis, nous présentons les critères d'optimisation et les algorithmes classiques de résolution (solveurs) des PDM sous l'hypothèse d'une connaissance complète et exacte des lois qui régissent la dynamique du système (PDM complètement observable). Les PDM Partiellement Observables (PDMPO) où l'état du système est partiellement connu seront ensuite brièvement présentés. Enfin, pour les modèles initialement inconnus où les fonctions de transition et de récompense ne sont pas a priori connues ou partiellement connues, nous exposons les méthodes d'apprentissage par renforcement qui ont pour objectif l'estimation d'une stratégie optimale sans nécessairement connaitre le modèle complet.

1.2. Cadre théorique d'un PDM

Considérons un système dynamique qui est observé aux instants $t = 0, 1, 2,..., N$. A chaque instant t, l'état du processus est noté par S_t où S_t est une variable aléatoire à valeurs dans un ensemble d'états S**.** Si le processus est dans l'état i, un contrôleur choisit une action $a \in A(i)$ où $A(i)$ est l'ensemble des actions possible dans l'état i. L'action choisie à l'instant t peut être considérée comme une variable aléatoire A_t. Si le système est dans l'état i et une action $a \in A(i)$ est choisie, alors indépendamment de l'histoire du processus, il en résulte deux choses**:**

1. Une récompense $r(i , a)$ est acquise immédiatement**.**
2. Le système passe à un autre état j avec une probabilité de transition P_{iaj} et le processus continu de la même façon à partir de l'état j.

Un PDM est donc définit par un 5-uplet (S, A, T, P, R) où :

- S est l'espace d'états qui peut être continu ou discret, fini ou infini.
  - $S=\{S_0, S_1,\ldots, S_t,\ldots\}$ dénote l'ensemble des états possibles du système.
- A est l'espace d'actions contrôlant la dynamique du système et qui peut aussi être continu ou discret, fini ou infini.
  - $A=\{a_0, a_1,\ldots, a_t,..\}$ représente l'ensemble des actions.
- $T=\{0,1,2,\ldots,N\}$ est l'horizon du système qui est généralement discret et qui représente les temps de décision, on parle ainsi d'horizon fini ($N<\infty$) ou infini ($N=\infty$) .
- $P: S \times A \times S \rightarrow [0,1]$ est la loi de transition d'un état à un autre définie par :

$$P_{sas'} = P(S_t = s' \mid s_0, a_0, s_1, a_1, \ldots S_{t-1} = s, A_{t-1} = a) = P(s' \mid s, a) \quad (1.1)$$

En tenant compte de la propriété de Markov qui suppose que l'état du système dans le futur ne dépend que de son état courant et de l'action à exécuter, l'équation (1.1) devient :

$$P_{sas'} = P(S_t = s' \mid S_{t-1} = s, A_{t-1} = a) \quad (1.2)$$

- R est la fonction de récompense $R: S \times A \rightarrow \mathbb{R}$ qui associe à tout paire état-action (i, a) une valeur numérique réelle $r(i, a)$ quantifiant l'utilité immédiate d'exécuter l'action a dans l'état i. La fonction de récompense peut également dépendre de l'état futur j ($r(i, a, j)$) avec R: $S \times A \times S \rightarrow \mathbb{R}$.

La **figure 1.1** représente un PDM sous la forme d'un diagramme d'influence. A chaque instant t, l'agent choisie une action A_t qui est appliquée dans l'état courant S_t et qui transite le processus vers l'état S_{t+1}. La récompense R_t est émise au cours de cette transition.

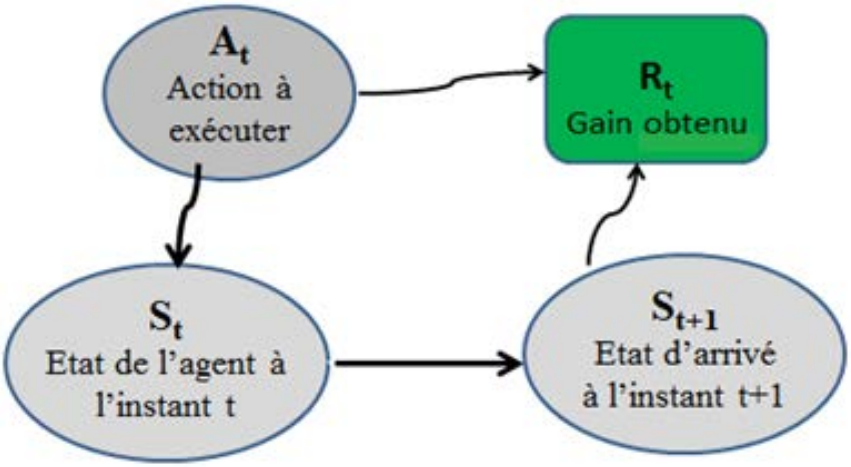


Figure 1.1: PDM sous forme d'un diagramme d'influence.

1.2.1. Différents types de politiques

Une politique, ou encore une stratégie, est une application qui spécifie aux agents l'action à exécuter pour tout état ou historique du système. Il existe différentes classes de politiques allant des politiques déterministes aux politiques aléatoires dépendantes de l'historique. On note par $H_t = (S \times A)^{t-1} \times S$, l'ensemble des histoires possibles du système jusqu'à l'instant t. On définit par $\Omega = (S \times A)^{\infty}$ l'espace échantillonnal.

Soit $\Psi = \left\{ (q_1, q_2, \ldots, q_{|A|}) \in I\mathrm{R}^{|A|} : \sum_{a=1}^{|A|} q_a = 1, q_a \geq 0, 1 \leq a \leq |A| \right\}$ l'ensemble des distributions de probabilité sur A.

Définition 1.1 : Une stratégie π est une suite $\pi = (\pi^1, \pi^2, \dots)$ où π^t est une règle de décision c'est à dire une fonction $\pi^t : H_t \to \psi$.

Etant donné une stratégie π, un état initial $i \in S$, il existe une et une seule mesure de probabilité P_π^i [PUT94]:

- $P_\pi^i(S_0 = i) = 1$.
- $P_\pi^i(A_m = a | S_0 = i_0, A_0 = a_0, \dots, S_m = i_m) = \pi_a^m(i_0, a_0, \dots, i_m)$.
- $P_\pi^i(X_{m+1} = z | S_0 = i_0, A_0 = a_0, \dots, S_m = y, A_m = a) = P_{yaz}$.

Où $\pi_a^m(h_t)$ désigne la a^{ème} composante de $\pi^m(h_t)$.

Définition 1.2 (stratégie markovienne) : Une stratégie $\pi = (\pi^1, \pi^2, \dots)$ est dite markovienne si sa règle de décision, $\pi^t : S \times A \to [0,1]$, dépend du temps et indépendante de l'historique.

Définition 1.3 (stratégie stationnaire) : Une stratégie $\pi = (\pi^1, \pi^2, \dots)$ est dite stationnaire s'il est markovienne et si sa règle de décision π^t ne dépend pas du temps ($\forall\, t, \pi^t = \pi$).

Définition 1.4 (stratégie déterministe) : Une stratégie stationnaire π est dite déterministe (ou pure) si sa règle de décision $\pi : S \to A$ est déterministe, indiquant pour chaque état quelle action à effectuer.

Si on note par :

F : l'ensemble de toutes les stratégies.

F_M : l'ensemble des stratégies markoviennes.

F_S : l'ensemble des stratégies stationnaires.

F_D : l'ensemble des stratégies déterministes.

On a alors les inclusions suivantes: $F_D \subset F_S \subset F_M \subset F$.

A chaque stratégie stationnaire $f \in F_S$, on associe une chaine de Markov d'espace d'état S, de matrice de transition P_f et de vecteur récompense R_f où :

$$[P_f]_{xy} = \sum_{a \in A(x)} P_{xay} \times P_f(A_t = a | S_t = x) \;\; \forall x, y \in S \tag{1.3}$$

$$[R_f]_x = \sum_{a \in A(x)} r(x,a) \times P_f(A_t = a | S_t = x) \;\; \forall x \in S \tag{1.4}$$

La récompense espérée à l'instant t, sachant que l'état initial est i et la stratégie π est utilisée, notée par $E_i^t(\pi)$, est donnée par :

$$E_i^t(\pi) = \sum_{j \in S} \sum_{a \in A(j)} P_\pi(S_t = j, A_t = a \mid S_0 = i) \times r(j,a) \tag{1.5}$$

1.2.2. Graphe associé à un PDM

A chaque PDM, on associe le graphe orienté $G = (S, U)$, où l'espace d'états S représente l'ensemble des sommets et $U = \{ (i,j) \in S^2 : \exists\, a \in A(i), P_{iaj} > 0 \}$ représente l'ensemble des arcs.

Considérons l'exemple suivant (**Figure 1.2**), du PDM à deux états: S={1,2}, A={1,2}, r(1,1)=2 ; r(1,2)=4; r(2,1)=3; r(2,2)=3; $P_{111} = 0$; $P_{112} = 0.5$; $P_{121} = 0.4$; $P_{122} = 0.6$; $P_{211} = 1$; $P_{212} = 0$; $P_{221} = 0$; $P_{222} = 1$.

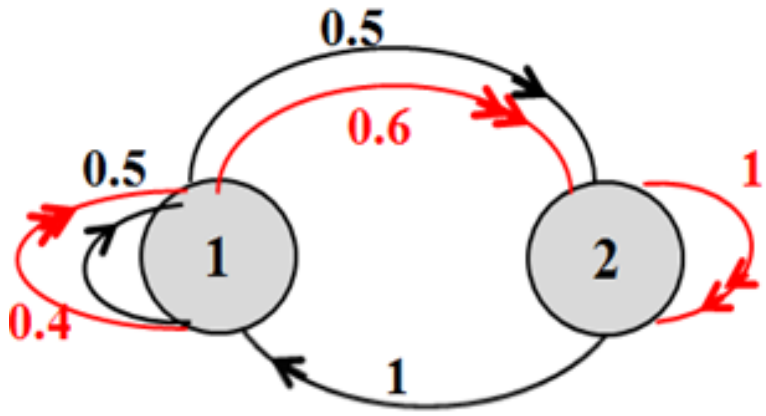


Figure 1.2: Exemple d'un graphe associé à un PDM.

Considérons la relation : i ***R*** j ⇔ il existe un chemin de *i* à *j* et de *j* à *i*, ou *i* = *j*.
R est une relation d'équivalence, ses classes d'équivalence C_1, C_2, ...C_q forment une partition de *S*.

Définition 1.5 : On appelle Composante Fortement Connexe (CFC), toute classe d'équivalence associée à ***R***.

Définition 1.6 : Soit G=(S, U) le graphe associé à un PDM.

- Un sous ensemble *E* de *S* est dit communicant si et seulement si *E* est une CFC dans *G*.
- Un sous ensemble E de S est dit fermé si et seulement si $P_{iaj} = 0 \; \forall i \in E, \forall a \in A(i) \; et \; \forall j \notin E$.

Définition 1.7 : Un graphe est dit fortement connexe si étant donné deux sommets quelconques *i* et *j*, il existe un chemin d'extrémité initiale i et d'extrémité terminale j.

1.3. Critères d'optimalité

La résolution d'un PDM consiste à chercher, parmi une famille de politiques celles qui optimisent un critère de performance donné. Ce critère a pour but de caractériser les politiques qui permettront de générer des séquences de récompenses les plus importantes possibles. Les critères les plus étudiés au sein de la théorie des PDM sont :

- **Critère fini** : $E[R_0 + R_1 + \cdots + R_{N-1} | \, S_0]$
- **Critère α-pondéré** : $E[R_0 + \alpha R_1 + \alpha^2 R_2 + \cdots + \alpha^t R_t + \cdots | S_0]$
- **Critère total** : $\mathrm{E}[\mathrm{R}_0 + \mathrm{R}_1 + \cdots + \mathrm{R}_t + \cdots | \mathrm{S}_0]$
- **Critère moyen** : $\lim_{t\to\infty} \frac{1}{t} E[R_0 + R_1 + \cdots + R_{t-1} | S_0]$

Où R_t représente la variable aléatoire récompense à l' instant *t*.

Il faut noter que les deux caractéristiques communes à ces quatre critères sont d'une part, leur formule additive en R_t, qui est une manière simple de résumer l'ensemble des

récompenses reçues le long d'une trajectoire et d'autre part, l'espérance $E[..]$ qui est retenue pour résumer la distribution des récompenses pouvant être reçues le long des trajectoires, pour une même politique et un même état de départ. En outre, chacun de ces critères permet de définir une mesure d'utilité appelée fonction de valeur V^{π} associée à chaque politique π avec $V^{\pi} : S \rightarrow \mathbb{R}$.

Une politique π^* est dite optimale si elle maximise l'espérance de l'agent en tout état $s \in S : V^{\pi}(s) \leq V^{\pi^*}(s), \forall s \in S$.

1.3.1. Critère fini

Le critère fini suppose que l'agent doit contrôler le système en *N* étapes, avec *N* fini. Ce qui conduit à définir la fonction de valeur $V_N^{\pi}(\boldsymbol{s})$ qui associe à tout état $\boldsymbol{s}$ l'espérance de la somme des *N* prochaines récompenses en suivant la politique π à partir de s.

$$\forall s \in S, \qquad V_N^{\pi}(\boldsymbol{s}) = E^{\pi}\left[\sum_{t=0}^{N-1} R_t | S_0 = \boldsymbol{s}\right] \tag{1.6}$$

Si l'agent se trouve dans l'état $\boldsymbol{s}$ lors de la dernière étape de décision, il est clair que la meilleure décision à prendre est celle qui maximise la récompense instantanée à venir et qui viendra s'ajouter à celles qu'il a déjà perçues. On a ainsi :

$$V_1^*(\boldsymbol{s}) = \max_{a \in A(s)} R_{N-1}(s, a) \tag{1.7}$$

$$\pi_{N-1}^*(\boldsymbol{s}) \in \underset{a \in A(s)}{argmax}\, R_{N-1}(s, a) \tag{1.8}$$

Où $\pi_{N-1}^*(s)$ est une politique optimale à suivre à l'étape *N*-1 et $V_1^*(s)$ la fonction de valeur optimale pour un horizon de longueur 1, obtenue en suivant cette politique optimale.

Supposons maintenant que l'agent est dans l'état **s** à l'étape *N*-2. Le choix d'une action optimale qui va lui rapporter la récompense $R_{N-2}(s, a)$ et l'amènera de manière aléatoire vers un nouvel état $\boldsymbol{s'}$ à l'étape *N*–1 où il sait qu'en suivant la politique optimale $\pi_1^*(s')$, il pourra récupérer une récompense moyenne $V_1^*(s')$**.** Ainsi, le problème de l'agent à l'étape N - 2 se ramène simplement à rechercher l'action qui maximise la somme suivante :

$$R_{N-2}(s, a) + \sum_{s' \in S} P_{N-2}(s'|s, a) V_1^*(s') \tag{1.9}$$

Où $P_{N-2}(s'|s, a)$ désigne la probabilité que le système transite vers l'état s' dans l'étape $N - 2$ sachant qu'il est dans l'état s et que l'action exécutée est a.

La politique optimale $\pi_{N-2}^*(s)$ à suivre à l'étape $N - 2$ et la fonction de valeur optimale $V_2^*(s)$**,** pour un horizon de longueur 2, sont donc données par les deux équations suivantes :

$$V_2^*(s) = \max_{a \in A(s)} \left(R_{N-2}(s, a) + \sum_{s' \in S} P_{N-2}(s'|s, a) V_1^*(s') \right), s \in S \tag{1.10}$$

$$\pi_{N-2}^*(s) = \underset{a \in A(s)}{argmax} \left(R_{N-2}(s, a) + \sum_{s' \in S} P_{N-2}(s'|s, a) V_1^*(s') \right), s \in S \tag{1.11}$$

En suivant le même raisonnement jusqu'à la première étape de décision, une politique optimale π_0^* à suivre à l'étape 0 et la fonction de valeur optimale V_N^*, pour un horizon de longueur N, sont données par les deux équations suivantes :

$$V_N^*(s) = \max_{a \in A(s)} \left(R_0(s,a) + \sum_{s' \in S} P_0(s'|s,a) V_{N-1}^*(s') \right), s \in S \quad (1.12)$$

$$\pi_0^*(s) = \operatorname*{argmax}_{a \in A(s)} \left(R_0(s,a) + \sum_{s' \in S} P_0(s'|s,a) V_{N-1}^*(s') \right), s \in S \quad (1.13)$$

Ces derniers résultats conduisent à l'énoncé du théorème suivant, qui montre l'existence d'une stratégie optimale pour un PDM stationnaire où les récompenses et les probabilités de transition ne dépendent pas du temps de décision.

Théorème 1.1 : Soient $N < \infty$ et π une politique déterministe. Alors la fonction de valeur optimale V_π^* est solution du système d'équations linéaires suivant :

$$V_{n+1}(s) = \max_{a \in A(s)} \left(r(s,a) + \sum_{s' \in S} P(s'|s,a) V_n(s') \right), s \in S \quad (1.14)$$

pour n = 0, . . ., $N - 1$ et $V_0 = 0$.
Une politique optimale est ainsi donnée par l'équation :

$$\pi^*(s) = \operatorname*{argmax}_{a \in A(s)} \left(r(s,a) + \sum_{s' \in S} P(s'|s,a) V_n(s') \right), s \in S \quad (1.15)$$

Preuve : Voir [BEL57, GRO08].

1.3.2. Critère α-pondéré

Le critère **α**-pondéré (ou d'actualisation) est le critère à horizon infini le plus classique. Sa fonction de valeur $V_\alpha^\pi(s)$ associe à tout état s, la limite lorsque N tend vers l'infini de l'espérance, en suivant la politique π à partir de s, de la somme des N futures prochaines récompenses pondérées par un facteur d'actualisation.

$$\forall s \in S, \qquad V_\alpha^\pi(s) = E^\pi \left[\sum_{t=0}^{\infty} \alpha^t R_t | S_0 = s \right] \quad (1.16)$$

Soit π une politique stationnaire, on définit l'opérateur L_π de $\mathcal{V}$ dans $\mathcal{V}$, un espace vectoriel (muni de la norme max : $\forall V \in \mathcal{V}, \|V\| = \max_{s \in S} |V(s)|$) par :

$$\forall V \in \mathcal{V},\ L_\pi V = R_\pi + \alpha P_\pi V \quad (1.17)$$

Où R_π (P_π) sont respectivement le vecteur récompense (matrice de transition) associés à la politique π.

Les théorèmes suivants montrent l'existence d'une stratégie déterministe optimale [BEL57, GRO08].

Théorème 1.2 : Soient $\alpha < 1$ et π une politique stationnaire. Alors V_α^π est l'unique solution de l'équation $V = L_\pi V$.

$$\forall s \in S, \quad V(s) = R_\pi(s) + \alpha \sum_{s' \in S} P_\pi(s'|s) V(s') \quad (1.18)$$

La résolution d'un PDM revient à résoudre le problème d'optimisation $V^* = max_\pi V_\alpha^\pi$. Ainsi, on définit l'opérateur L de l'ensemble des fonctions de valeur $\mathcal{V}$ dans lui-même, nommé opérateur de programmation dynamique par :

$$\forall V \in \mathcal{V} \; et \; \forall s \in S \qquad LV(s) = \max_{a \in A(s)} \left\{ r(s,a) + \alpha \sum_{s' \in S} P(s'|s,a) V(s') \right\} \tag{1.19}$$

Soit encore en notation vectorielle par :

$$\forall V \in \mathcal{V} \qquad LV = \max_{\pi} \{ R_\pi + \alpha P_\pi V \} \tag{1.20}$$

Théorème 1.3 (Equation de Bellman)

Soit $\alpha < 1$. Alors V_α^* est l'unique solution de l'équation $LV = V$.

$$\forall s \in S \qquad V(s) = \max_{a \in A(s)} \left\{ r(s,a) + \alpha \sum_{s' \in S} P(s'|s,a) V(s') \right\} \tag{1.21}$$

De plus toute politique stationnaire π^* définie par :

$$\forall s \in S \qquad \pi^*(s) \in argmax_{a \in A} \left\{ r(s,a) + \alpha \sum_{s' \in S} P(s'|s,a) V_\alpha^*(s') \right\} \tag{1.22}$$

est une politique optimale.

1.3.3. Critère total

Le critère total a pour fonction de valeur :

$$\forall s \in S, \qquad V^\pi(s) = E^\pi \left[\sum_{t=0}^{\infty} R_t | X_0 = s \right] \tag{1.23}$$

L'existence de la limite (ou d'une stratégie optimale) sous le critère total ne peut pas être assurée pour tous les PDM. Deux classes de problèmes pour lesquelles cette limite (stratégie) existe nécessairement, et est finie pour au moins une politique : les modèles positifs et les modèles négatifs [BEL57, BER87, PUT94]. Notamment, cette limite existe pour les PDM avec un état absorbant tel que le problème du plus court chemin stochastique. On définit le même operateur de Bellman et on montre que le vecteur optimal est aussi l'unique solution de l'équation de Bellman.

La **figure 1.3** montre des exemples de PDM où la somme à l'infinie peut ne pas exister comme elle peut être infinie.

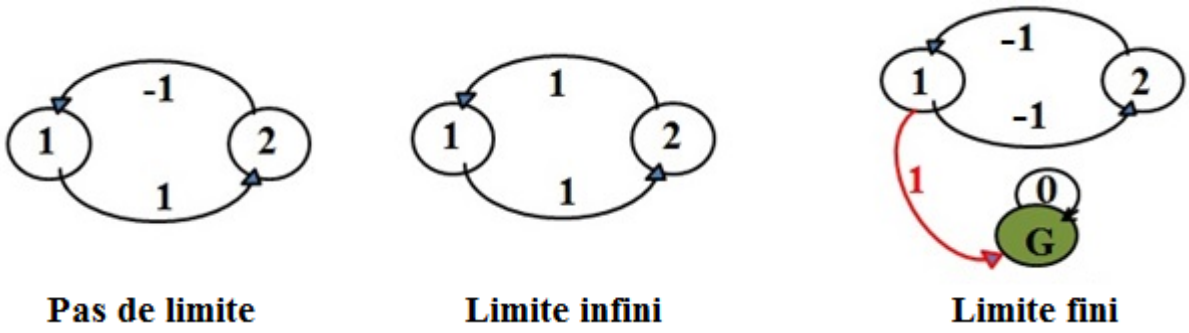


Figure 1.3 : Exemples de PDM où la limite est finie, infinie ou n'existe pas.

1.3.4. Critère moyen

Le critère moyen est utilisé lorsque la fréquence des décisions est importante, avec un facteur d'actualisation proche de 1, ou lorsqu'il n'est pas possible de donner une valeur économique aux récompenses. On préfère considérer ce critère qui représente la moyenne des récompenses le long d'une trajectoire et non plus leur somme pondérée. Il est particulièrement

utilisé dans des applications de type gestion de file d'attente, de réseaux de télécommunication, de stock, etc. On associe d'abord à une politique π, l'espérance de la récompense moyenne par étape. Ensuite, on définit la fonction de valeur $\rho^{\pi}(s)$ associé à une politique π et à un état s par :

$$\forall s \in S, \rho^{\pi}(s) = \lim_{N\to\infty} E^{\pi}\left[\frac{1}{N}\left(\sum_{t=0}^{N-1} R_{\pi}(s_t)|S_0 = s\right)\right] \quad (1.24)$$

L'analyse théorique du critère moyen est le plus complexe des critères. Nous nous limitons ici à présenter brièvement le cas particulier des PDM uni-chaines où pour toute politique markovienne déterministe, l'espace d'états est décomposé en une seule classe communicante. Le théorème suivant montre l'existence d'une stratégie déterministe optimale [PUT94].

Théorème 1.4 (Equations d'optimalité uni-chaine [PUT94])
Il existe une solution (ρ, U) au système d'équations définies pour tout s ∈ S par :

$$U(s) + \rho = \max_{a\in A(s)}\left(r(s,a) + \sum_{s'\in S} P(s'|s,a)\,U(s')\right) \quad (1.25)$$

Une stratégie optimale est donnée par :

$$\pi^*(s) = \operatorname*{argmax}_{a\in A(s)}\left(r(s,a) + \sum_{s'\in S} P(s'|s,a)\,U(s')\right) \quad (1.26)$$

1.4. Algorithmes de résolution des PDM complètement observables

Nous présentons dans cette section les algorithmes de résolution des PDM complétement observables qui supposent une connaissance parfaite du 5-tuple (*S*, *A*, *T*, *P*, *R*).

1.4.1. Critère fini

Pour le critère à horizon fini, les équations d'optimalité (1.14) et (1.15) permettent de calculer récursivement, à partir de la dernière étape, les fonctions de valeur optimales selon l'algorithme de programmation dynamique (**Algorithme 1.1**). Ce qui permet encore de déterminer une stratégie optimale π^*.

Algorithme 1.1 : Algorithme de programmation dynamique pour le critère fini.

PD (S, A, P, R)
1: Pour tout s ∈ S Faire $V_0(s) \leftarrow 0$
2: Pour n←0 jusqu'à N-1 Faire
3: Pour tout s ∈ S Faire

$$V_{n+1}(s) \leftarrow \max_{a\in A(s)}\left(r(s,a) + \sum_{s'\in S} P(s'|s,a)V_n(s')\right)$$

4: Pour tout $s \in S$ faire

$$\pi^*(s) \in \operatorname*{argmax}_{a\in A(s)}\left(r(s,a) + \sum_{s'\in S} P(s'|s,a)V_N(s')\right)$$

5: Retourner V_N, π^*

La complexité temporelle et spatiale de l'**algorithme 1.1** est en $\mathcal{O}(N|S|^2|A|)$, ce qui le rend couteux pour des espaces d'états et d'actions assez grands [BEL57, PUT94].

1.4.2. Critère α-pondéré

Il existe, dans la littérature, trois algorithmes classiques de résolution des PDM sous le critère α-pondéré : la Programmation Linéaire (PL), l'Itération de la Valeur (IV) et l'Itération de la Politique (IP).

A. Algorithme de programmation linéaire

Une stratégie optimale peut être obtenue seulement par la résolution d'un programme linéaire qu'on va présenter dans ce paragraphe. En effet si $V \in \mathcal{V}$ minimise la fonction $\sum_{s\in S} V(s)$ sous la contrainte $V \geq LV$, alors $V = V_\alpha^*$. Le vecteur gain optimal V_α^* est donc une solution du programme linéaire (P) :

$$(P)\begin{cases} \underset{V\in\mathcal{V}}{\text{Min}} \sum_{i\in S} V(i) \\ V(s) \geq r(i,a) + \alpha \sum_{j\in S} P(j|i,a)V(j)\,, \forall\, i \in S, \forall\, a \in A(i) \end{cases}$$

Ceci nous permet de déterminer une stratégie optimale π^* (Théorème 1.3) par :

$$\pi^*(i) = \underset{a\in A(i)}{argmax}\left(r(i,a) + \alpha \sum_{j\in S} P(j|i,a)\, V_\alpha^*(j) \right), i \in S \qquad (1.27)$$

C'est l'idée principale de l'algorithme de programmation linéaire (**Algorithme 1.2**) qui a été proposée initialement par [D'E63]. Par ailleurs, LITTMAN [LIT95] a montré que si *n* et *m* sont les tailles respectives de *S* et *A*, avec *P*() et *R*() codées sur *b* bits, la complexité de l'algorithme de programmation linéaire sur les rationnels est polynomiale en |*S*|, |*A*|, *b*, avec des temps de résolution assez lents.

Algorithme 1.2 : Algorithme de Programmation Linéaire (PL).

PL (*S*, *A*, *P*, *R*, α)

1: Résoudre: $\underset{V\in\mathcal{V}}{\text{Min}} \sum_{s\in S} V(s)$

Avec: $V(s) \geq r(s,a) + \alpha \sum_{s'\in S} P(s'|s,a)V(s')\,, \forall\, s \in S, \forall\, a \in A(s)$

2: $V_\alpha^* \leftarrow V$

3: Pour tout $s \in S$ faire

$\pi^*(s) \in argmax_{a\in A}\left\{ r(s,a) + \alpha \sum_{s'\in S} P(s'|s,a)V_\alpha^*(s') \right\}$

4: Retourner V_α^*, π^*

B. Algorithme d'Itération de la Valeur (IV)

L'approche la plus classique, se base aussi sur la résolution directe de l'équation d'optimalité de Bellman $V = LV$, en utilisant pour cela une méthode itérative de type point fixe (**Algorithme 1.3**), d'où son nom : Itération de la Valeur [BEL57, BER87, PUT94].

Algorithme 1.3: Algorithme IV.

IV (S, A, P, R, α, ε)

1: Initialiser $V_0 \in \mathcal{V}$
2: n←0

3 : Répéter

4 : Pour tout $s \in S$ faire

$$V_{n+1}(s) \leftarrow max_{a \in A}\left\{r(s,a) + \alpha \sum_{s' \in S} P(s'|s,a)V_n(s')\right\}$$

5: n←n+1

Jusqu'à $\|V_{n+1} - V_n\| < \varepsilon$

6 : Pour tout $s \in S$ faire

$$\pi^*(s) \in argmax_{a \in A}\left\{r(s,a) + \alpha \sum_{s' \in S} P(s'\backslash s,a)V_n(s')\right\}$$

7: Retourner V_n, π^*

A partir d'une valeur initiale arbitraire de chaque état, l'algorithme améliore itérativement l'évaluation des états par approximations successives en utilisant l'équation de Bellman (1.21). La valeur d'un état, à l'itération *t*, est calculée à partir de sa valeur à l'itération *t-1*. Le processus itératif s'arrête lorsque la différence entre les valeurs successives de tous les états est inférieure à un paramètre ε assez petit.

D'ailleurs, un nombre maximum d'itérations polynomial en |S|, |*A*|, *b*, $1/(1-\alpha)$ $\log(1/(1-\alpha))$ est nécessaire pour atteindre une stratégie ε-optimale π^*, chaque itération étant de complexité $\mathcal{O}(|A||S|^2)$ [PAP87]. Au-delà de ce nombre d'itérations, la suite V_n est de plus en plus proche du vecteur optimal V^*, mais la politique correspondante π_n ne change plus.

Il est possible d'améliorer la vitesse de convergence de l'algorithme IV en modifiant légèrement le calcul de V_{n+1}. L'idée consiste à utiliser $V_{n+1}(s)$ à la place de $V_n(s)$ lorsque cette valeur a déjà été calculée. On définit ainsi l'algorithme de Gauss-Seidel (**Algorithme 1.4**), une variante de l'algorithme IV qui permet de réduire le nombre d'itérations nécessaire avant la convergence.

Algorithme 1.4 : Algorithme IV Gauss-Seidel (IVGS).

IVGS (S, A, P, R, α, ε)
1: Initialiser $V^* \in \mathcal{V}$
2: Convergence ← Faux
3: Tant que (Convergence = Faux) Faire
4: Convergence ← Vrai
5: Pour tout s ∈ S faire
$V_{tmp} \leftarrow max_{a \in A}\left\{r(s,a) + \alpha \sum_{s' \in S} P(s'|s,a)V^*(s')\right\}$
Si $\left(\left|V_{tmp} - V^*(s)\right| \geq \varepsilon \right)$Alors Convergence ← Faux
$V^*(s) = V_{tmp}$
6: Pour tout $s \in S$ faire
$\pi^*(s) \in argmax_{a \in A}\left\{r(s,a) + \alpha \sum_{s' \in S} P(s'|s,a)V^*(s')\right\}$
7: Retourner V^*, π^*

C. Algorithme d'itération de la politique (IP)

L'algorithme d'itération de la politique (IP) (**Algorithme 1.5)** est également un algorithme itératif utilisant le principe d'optimalité de Bellman. L'algorithme part d'une politique initiale arbitraire π_0, à chaque itération, la politique courante est évaluée puis améliorée. L'évaluation utilise l'équation de Bellman et revient à résoudre un système d'équations linéaires. La phase d'amélioration consiste à mettre à jour la politique courante en utilisant les valeurs calculées lors de la phase d'évaluation.

Algorithme 1.5 : Algorithme d'itération de la politique (IP)

IP (S, A, P, R, α, ε)
1 : Initialiser $\pi_0 \in F_D$
2 : n←0
3 : Répéter
4 : Résoudre le système :
$\forall s \in S, \quad V(s) \leftarrow r\left(s, \pi_n(s)\right) + \alpha \sum_{s' \in S} P(s'|s, \pi_n(s))V(s')$
5 : Pour tout $s \in S$ faire
$\pi_{n+1}(s) \in argmax_{a \in A}\left\{r(s,a) + \alpha \sum_{s' \in S} P(s'|s,a)V(s')\right\}$
6: n←n+1
7: jusqu'à $\pi_n = \pi_{n-1}$
8: Retourner V, π_n

L'algorithme IP converge lorsqu'aucune modification de la politique n'est plus possible. La complexité en temps de cet algorithme dépend de la complexité d'une itération et du nombre d'itérations. La complexité de la phase d'évaluation d'une politique est en $\mathcal{O}(|S|^3)$ et celle de l'amélioration est en $\mathcal{O}(|A||S|^2)$ [BEL57, PUT94]. Le nombre d'itérations de l'algorithme dépend fortement de la politique initiale π_0. Il existe $|A|^{|S|}$ politiques possibles.

Étant donné qu'à chaque itération la politique mise à jour domine strictement la politique précédente, le nombre d'itérations sera, au pire des cas, exponentiel en nombre d'états.

Certes, il est possible d'améliorer l'efficacité l'algorithme IP en simplifiant la phase d'évaluation de la politique courante π_n en résolvant l'équation $V_n = L_{\pi_n} V_n$ de manière itérative, comme pour l'algorithme IV. L'utilisation de ce principe conduit ainsi à construire l'algorithme IP modifié (**Algorithme 1.6**) qui est d'une complexité de l'ordre $\mathcal{O}(|A||S|^2)$ par itération [BEL57, PUT94].

Algorithme 1.6 : Algorithme IP modifié.

IPM (S, A, P, R, α, ε)

1: Initialiser $\pi_0 \in F_D$

2 : $n \leftarrow 0$

3: Répéter

4: Initialiser $V^* \in \mathcal{V}$

5 : Convergence ← Faux

6 : Tant que (Convergence = Faux) Faire

7 : Convergence ← Vrai

8 : Pour tout $s \in S$ faire

$V_{tmp} \leftarrow r(s, \pi_n(s)) + \alpha \sum_{s'} P(s'|s, \pi_n(s)) V^*(s')$

Si $\left(|V_{tmp} - V^*(s)| \geq \varepsilon \right)$ Alors Convergence ← Faux

$V^*(s) = V_{tmp}$

9: Pour tout $s \in S$ faire

$\pi_{n+1}(s) \in \operatorname{argmax}_{a \in A} \left\{ r(s,a) + \alpha \sum_{s' \in S} P(s'|s,a) V^*(s') \right\}$

10: $n \leftarrow n + 1$

11: Jusqu'à $\pi_n = \pi_{n-1}$

12: Retourner V^*, π_n

1.4.3. Critère total

Sous le critère total ($\alpha = 1$), l'algorithme IV et ses variantes convergent de manière monotone vers V^* pour les modèles bornés positivement (ou négativement) sous l'hypothèse que V_0 vérifie la condition : $V_0 \in [0, V^*]$ (ou $V_0 \in [V^*, 0]$).

De même, sous les hypothèses $LV_0 \leq V_0$ et $V_0 \leq V^*$, on montre que l'algorithme IP modifié, converge vers une politique optimale (en pratique, $V_0 = 0$ est une condition suffisante).

1.4.4. Critère moyen

De nombreux algorithmes de programmation dynamique peuvent être utilisés dans le cas du critère de la moyenne. Nous nous limitons ici à présenter l'algorithme d'itération de la valeur relative (**Algorithme 1.7**) dans le cas simplifié d'un PDM uni chaîne, qui est caractérisé par un gain moyen constant pour tous les états.

Algorithme 1.7 : Algorithme IV Relative.

IVR $(S, A, P, R, \alpha, \varepsilon)$

1: Initialiser $U_0 \in \mathcal{V}$

2: Choisir $s_0 \in S$

3: n←0

4: Répéter

$$\rho \leftarrow \underset{a \in A(s_0)}{\text{Max}} \left(r(s_0, a) + \sum_{s' \in S} P(s'|s_0, a)\, U_n(s') \right)$$

Pour tout $s \in S$ Faire

$$U_{n+1}(s) \leftarrow \underset{a \in A(s)}{\text{Max}} \left(r(s, a) + \sum_{s' \in S} P(s'|s, a)\, U_n(s') \right) - \rho$$

n←n+1

Jusqu'à $(span(U_{n+1} - U_n) < \varepsilon)$

5: Pour tout $s \in S$ faire

$$\pi^*(s) \in \underset{a \in A(s)}{\max} \left\{ r(s, a) + \sum_{s' \in S} P(s'|s, a) U_n(s') \right\}$$

6: Retourner π^*, U_n, ρ

Le test d'arrêt est basé sur l'emploi de la semi-norme *span* sur $\mathcal{V}$ définie par $span(V) = max_{s \in S}(V(s)) - min_{s \in S}(V(s))$ et qui mesure l'écart de V à un vecteur constant.

L'**algorithme 1.7** s'exécute en $\mathcal{O}(|A||S|^2)$ opérations arithmétiques par itération [BEL57, PUT94], il est aussi coûteux en temps d'exécution pour un espace d'états de grande taille.

1.5. Techniques de résolution des problèmes de grande taille

Les algorithmes de résolution des PDM, présentés dans la section précédente, sont d'une complexité polynomiale en taille d'espace d'états, ils sont donc impraticables pour des problèmes de grande taille. En littérature, plusieurs techniques ont étés proposées pour remédier à ce problème, parmi ceux, la technique d'élimination des actions non optimales, la réduction de l'espace d'états en utilisant le graphe d'accessibilité à partir d'un état initial donné, les méthodes de décomposition (diviser pour régner) et enfin le parallélisme.

1.5.1. Réduction de l'espace d'actions

La technique d'élimination des actions a été introduite par MacQueen [MAC67], il a proposé une borne inférieure V_n^I et une borne supérieure V_n^S du vecteur gain optimal V_α^*, en fonction du facteur de pondération α, définis par l'équation suivante :

$$V_n^I = V_n + \frac{\Delta_{n+1}^{min}}{(1-\alpha)} \leq V^* \leq V_n + \frac{\Delta_{n+1}^{max}}{(1-\alpha)} = V_n^S \quad (1.28)$$

Où :

$$\Delta_n^{min} = \min_{i \in S}\{V_n(i) - V_{n-1}(i)\} \; et \; \Delta_n^{max} = \max_{i \in S}\{V_n(i) - V_{n-1}(i)\} \quad (1.29)$$

Durant chaque itération n, si la borne supérieure de la fonction valeur du couple état/action (s, a'), notée $V_n^S(s, a')$, est strictement inférieure à la borne inférieure du couple état/action (s, a), notée $V_n^I(s, a)$, alors l'action a' ne peut pas être une action optimale dans l'état s, cette action sera éliminée dans la suite des itérations. Par conséquent, une action a' non optimale est identifiée par l'équation suivante.

$$r(i, a') + \alpha \sum_{j \in S} P(j|i, a')V_n(j) < V_{n+1}(i) - \alpha \frac{(\Delta_{n+1}^{max} - \Delta_{n+1}^{min})}{(1-\alpha)} \quad (1.30)$$

Enfin, il faut noter que Porteus [POR71], Hastings et Mello [HAS73] et Hubner [HUB77] ont proposé d'autres bornes et d'autres équations d'élimination des actions [JAB08].

1.5.2. Réduction de l'espace d'états

La technique de réduction de l'espace d'états est basée sur la recherche du sous-espace d'états accessible à partir d'un état initial donné S_0. En effet, considérons l'exemple du graphe associé au PDM de la **figure 1.4**, l'espace d'états comporte six états et tenant compte de l'équation de Bellman (1.20) on a :

$$V(s_0) = \max_{a \in A(S_0)} \left\{ r(s_0, a) + \alpha \sum_{s' \in \{s_0, s_1, s_2\}} P(s'|s_0, a)V(s') \right\}$$

Ainsi, le calcul de la fonction de valeur optimale au point s_0, se limite à considérer seulement l'espace d'états accessibles à partir de s_0 , c'est-à-dire $\Gamma^+(s_0) = \{s_0, s_1, s_2\}$ l'ensemble des successeurs de s_0.

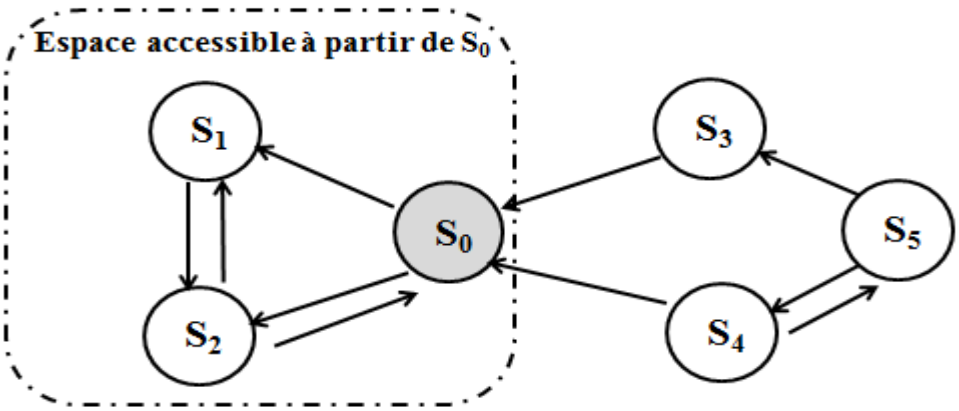


Figure 1.4: Exemple d'un PDM montrant les états accessibles à partir d'un état initial s_0.

1.5.3. Décomposition en petits problèmes

L'approche de la décomposition en petits problèmes (diviser pour régner) est répartie en deux catégories différentes suivant le type du PDM étudié. Les méthodes de décomposition en série [DEA95, ROS91, PAR98, DAO07, DAO10] dont la tâche globale du décideur est vue comme un ensemble de tâches séquentielles. Les méthodes de décomposition en parallèle [BEY04, GUE02, MEU98] dont la tâche globale est vue comme un ensemble de tâches compétitrices.

Les méthodes sérielles sont utilisées lorsqu'on peut subdiviser l'espace d'états en union de sous-espaces: $S = S_1 \cup S_2 \cup \ldots \cup S_p$, ces sous-espaces doivent être exécutés l'un après l'autre. Les méthodes parallèles sont utilisées lorsque l'espace d'états du modèle peut être vu comme un produit cartésien de sous-espaces: $S = S_1 \times S_2 \times \ldots \times S_p$. Dans ce cas, on est devant plusieurs décideurs qui cherchent, simultanément, un même objectif ou qui ont des objectifs distincts et on doit prendre en considération leurs interactions.

Nous considérons, dans ce travail, le type séquentiel de décomposition des PDM qui a été introduit par Ross et Varadarajan [ROS91] pour le critère de la moyenne sous contraintes. Il a été utilisé par Abbad [ABD03a, ABD03b]) et Daoui [DAO07, DAO10] pour différents critères. Ils ont proposé des algorithmes hiérarchiques basés sur la décomposition de l'espace d'états en union de classes communicantes, classifiées eux aussi en niveaux. Pour chaque niveau et chaque classe, ils définissent un PDM restreint, ces derniers sont résolus séparément et leurs solutions sont combinées pour obtenir une solution du problème globale.

Nous présenterons, en détail, dans le chapitre 3, cette technique de décomposition qui est la base de la plupart de nos contributions.

1.5.4. Parallélisme

Le parallélisme ou le multithreading repose sur une architecture multiprocesseur des ordinateurs à mémoire partagée ou à mémoires distribuées, c'est une méthode très connue d'optimisation basée sur le principe de « diviser pour régner ». Le concept du parallélisme appliqué au niveau des PDM est divisé en deux catégories : la première consiste au calcul parallèle de la fonction de valeur dans chaque itération ; la deuxième est une combinaison du parallélisme avec des méthodes inventées pour accélérer ce modèle, telle que la méthode de décomposition séquentielle.

Les algorithmes de résolution des PDM sont conçus de façon naturelle d'itérations, où chaque itération est de complexité polynomiale en temps. Pour cela, plusieurs chercheurs ont pensé à appliquer le parallélisme [ARC93, ZHA09, CHE13], qui revient à paralléliser le calcul au niveau de chaque itération en subdivisant l'ensemble des états S en P groupes : $S = S_1 \cup S_2 \cup \ldots \cup S_p$. Chaque groupe est attribué à un processeur. À chaque itération donnée t, tous les processeurs maintiennent la valeur optimale et la politique optimale obtenues dans l'itération t-1 pour tous les états des autres groupes, soit en utilisant une communication entre processeurs dans le cas d'une architecture à mémoires distribuées, soit en faisant référence à une mémoire partagée.

Dans la méthode de décomposition séquentielle, le parallélisme est utilisé au niveau des sous-ensembles où les calculs de la fonction de valeur et de la stratégie optimale sont indépendants. En effet, l'espace d'états S est divisé en p niveaux de sous-ensembles : $S = \{S_{11} \cup S_{12} \cup \ldots \cup S_{1N_1}\} \cup \{S_{21} \cup S_{22} \cup \ldots \cup S_{2N_2}\} \cup \ldots \cup \{S_{p1} \cup S_{p2} \cup \ldots \cup S_{pN_n}\}$. Dans

chaque niveau n, les sous-ensembles $S_{n1}, S_{n2}, \dots, S_{nN_n}$ sont indépendants vis-à-vis du calcul de la stratégie optimale. Chaque processeur se charge d'un sous-ensemble ou d'une partie des sous-ensembles.

1.6. Processus Décisionnels de Markov Partiellement Observables (PDMPO)

Les algorithmes de résolution des PDM, présentés dans la section 1.3, supposent une connaissance complète du tuple (S, A, T, P, R). Mais, en réalité, un agent chargé du contrôle a rarement accès à une information suffisante pour connaître l'état du processus, il a uniquement une distribution de probabilité sur l'état du système. Le formalisme des PDM Partiellement Observables (PDMPO) permet de modéliser ce genre de situation où un agent n'a accès qu'à des informations souvent partielles sur le processus à contrôler.

Un PDMPO est un PDM dans lequel l'agent ne connaît pas l'état réel du processus, l'agent n'a accès qu'à une observation partielle de cet état. Il est définit par le tuple (S, A, Ω, T, P, O, R, B_0). Les seules nouveautés par rapport à la définition d'un PDM sont donc la distribution initiale des états du processus B_0, l'espace des observations Ω et la fonction d'observation associée O(). A chaque instant t ∈ T, l'agent ne connaît pas l'état courant s_t mais peut seulement le percevoir partiellement sous forme d'une observation $o_t \in \Omega$. Cette observation est donnée par la fonction d'observation O. Quand il applique une action $a_t \in A$ sur le processus, cela modifie alors aléatoirement l'état du processus selon P() pour l'amener dans l'état s_{t+1} alors que l'agent ne perçoit que l'observation o_{t+1} qui dépend de la fonction O(). Finalement, comme dans un PDM, l'agent reçoit une récompense R ∈ IR et les mêmes critères d'optimalité utilisés dans les PDM complétement observables sont aussi applicables pour la résolution des PDMPO.

1.6.1. Etat d'information

L'agent décideur, qui ne connait pas son état, doit choisir ses actions en fonction des informations qui lui sont disponibles, dans ce cas on parle d'état d'information. Soit I_t l'ensemble des informations accessibles à un agent, à un instant *t*. Après avoir effectué l'action a_t, l'agent perçoit le processus sous la forme de l'observation o_{t+1}. L'ensemble $I = \{I_t, t \in T\}$ forme un ensemble d'états d'information si la séquence $(I_t)_{t \in T}$ définit une chaine de Markov contrôlée par les actions, c'est-à-dire si :

$$\forall t, P(I_{t+1} \mid I_0, I_1, \dots I_t, o_{t+1}, a_t) = P(I_{t+1} \mid I_t, o_{t+1}, a_t). \quad (1.31)$$

Soit I_t^C l'état d'information complet constitué de la distribution de probabilité initiale sur les états B_0, de l'historique des observations passées (o_0, …, o_{t-1}) et présente o_t, et de l'historique des actions passées (a_0, ..., a_{t-1}). Les états d'information complets satisfont à la propriété de Markov, mais leurs principal problème est que la taille d'un état d'information grossit à chaque pas de temps ce qui n'est pas facile à traiter.

Une séquence d'états d'information $(I)_t$ définit un état d'information suffisant si on a :

$$P(s_t|I_t) = P(s_t|I_t^C) \;\; et \;\; P(o_t|I_{t-1}, a_{t-1}) = P(s_t|I_{t-1}^C, a_{t-1}) \quad (1.32)$$

Ces méta-informations sur le processus satisfont la propriété de Markov et préservent les informations contenues dans les états d'information complets qui sont suffisantes pour contrôler le processus.

1.6.2. Etat de croyance

Les états de croyance sont des états d'informations suffisants couramment utilisés dans les PDMPO. Un état de croyance B_t , à l'instant *t*, est défini par : $B_t(s) = P(S_t = s|I_t^C)$. C'est une distribution de probabilité sur l'ensemble des états, et on peut exprimer l'état de croyance B_o^a après une transition du processus, c'est-à-dire après que l'agent a effectué une action *a* et a reçu une observation *o* par :

$$B_{t,o}^a(s') = P(s'|B_{t-1}, a, o) \tag{1.33}$$

Les auteurs de [GRO08] montrent que :

$$B_{t,o}^a(s') = \frac{O(o|s')\sum_{s\in S} P(s'|s,a)B_{t-1}(s)}{\sum_{s\in S}\sum_{s''\in S} O(o|s'')P(s''|s,a)B_{t-1}(s)} \tag{1.34}$$

Où $O(o|s')$ est la probabilité que l'agent reçoit une observation *o* sachant que l'état du système est s'.

Nous limitons ce cadre théorique des PDMPO, à cette distribution de probabilité d'états de croyance qui est suffisante pour résoudre certains problèmes de mobilité robotique où la position du robot n'est a priori pas connue. En effet, cette distribution permet au robot de se localiser après une suite d'observations. Nous présenterons, au chapitre 2, un exemple d'application qui illustrera ce principe de localisation.

D'ailleurs, les politiques sont généralement définies sur l'espace des observations ou l'historique des observations puisque les états ou les historiques d'états sont inaccessibles à l'agent et l'optimalité n'est généralement pas garantie [GRO08]. Les mêmes solveurs des PDM complétement observables sont aussi utilisés dans les PDMPO, mais avec d'autres types d'équation d'optimalité utilisant les états d'information et l'ensemble d'observations [GRO08].

1.7. Apprentissage par Renforcement (AR)

Les méthodes d'AR permettent de traiter les situations dans lesquelles les fonctions de transition et de récompense ne sont pas apriori connues ou partiellement connues. La plupart des méthodes algorithmiques de l'apprentissage par renforcement reposent sur des principes simples issus de l'étude de la cognition humaine ou animale [SIG04], comme par exemple le fait de renforcer la tendance à exécuter une action si ses conséquences sont jugées positives. Les premiers travaux qui entrent dans le cadre de l'AR datent approximativement de 1960, en 1961, Michie [MIC61] décrit un système capable d'apprendre à jouer au morpion par essais et erreurs. Puis, Michie et Chambers [MIC68] écrivent en 1968 un programme capable d'apprendre à maintenir un pendule inversé à l'équilibre. Parallèlement, Samuel [SAM69] réalise un logiciel qui apprend à jouer aux dames en utilisant une notion de différence temporelle. Les deux composantes, exploration par essais-erreurs et gestion de séquences d'actions par différences temporelles vont être au cœur des modèles ultérieurs. La

formalisation mathématique des algorithmes d'AR telle que nous la connaissons aujourd'hui s'est développée à partir de 1988 lorsque Sutton [SUT88], puis Watkins [WAT89] ont fait le lien entre ces travaux et le cadre théorique de la commande optimale proposée par Bellman en 1957 avec la notion de PDM [BEL57].

L'AR est basé sur une interaction itérée du système apprenant avec l'environnement, sous la forme de l'exécution à chaque instant t d'une action a_t depuis l'état courant s_t, qui conduit au nouvel état s_{t+1} et qui fournit la récompense R_t. La politique est petit à petit améliorée, sur la base de cette interaction. La plupart des algorithmes d'AR ne travaillent pas directement sur la politique, mais passent par l'approximation itérative de la fonction de valeur.

Les méthodes d'AR sont classifiées en deux catégories : (1) les méthodes indirectes qui estiment les paramètres inconnus au cours d'une série d'expériences et utilisent par la suite les algorithmes classiques de résolution des PDM; (2) les méthodes directes ne passent pas par l'estimation du modèle inconnu mais visent à approximer à partir d'expériences une fonction de valeur optimale des états (V) ou une fonction de valeur optimale du couple état/action (Q).

1.7.1. Méthodes directes et indirectes

Les méthodes indirectes, appelées en anglais «model-based», identifient en ligne et par la méthode de maximum de vraisemblance le modèle PDM inconnu. Ensuite, les algorithmes classiques des PDM seront appliqués pour résoudre le problème. L'**algorithme 1.8** présente un pseudo code de la méthode directe.

Algorithme 1.8: Méthode indirecte de résolution d'un PDM inconnu

1 : $\forall(s,a) \in (A,S), N(s,a) \leftarrow 0 \; et \;\; \forall s' \in S, N(s'|s,a) \leftarrow 0$
2: Observer l'état initial s
3: Tant que (non fin d'expérience) Faire
  a. Choisir une action $a \in A(s)$
  b. Exécuter l'action a
  c. Observer la récompense $r(s,a)$ et l'état suivant s'
  d. $N(s,a) \leftarrow N(s,a) + 1$; *//Nombre d'occurrences du couple* (s,a)
  e. $N(s'|s,a) \leftarrow N(s'|s,a) + 1$; *//Nombre d'occurrences de la transition* $(s,a) \rightarrow s'$
  f. $s \leftarrow s'$
4: Pour tout $(s,a) \in (S,A)$ Faire *//Estimation de la fonction de transition*
  $P(s'|s,a) \leftarrow \frac{N(s'|\,s,a)}{N(s,a)}$
5: Résoudre le PDM (A,S,R,P) *//Résolution du PDM estimé*

L'inconvénient de cette méthode est la taille gigantesque de la place mémoire nécessaire pour l'estimation. En effet, l'algorithme nécessite de charger en mémoire, en plus des données inconnus du PDM, une matrice de dimension trois pour compter le nombre d'occurrences de chaque transition $(s,a) \rightarrow s'$ et une matrice de dimension deux pour compter le nombre d'occurrences de chaque couple (s,a). De plus, une meilleure estimation nécessite, en générale, un très grand nombre de visites de chaque couple état/action.

Les méthodes directes, appelées en anglais «model-free», ne passent pas par l'identification ou l'estimation du modèle inconnu du système. En effet, les fonctions de transitions et de récompenses ne sont pas estimés et seule la fonction de valeur est mise à jour itérativement au cours du temps. Le principal avantage est ici en terme de place mémoire nécessaire, et historiquement, l'apprentissage par renforcement revendiquait d'être une méthode directe. Ainsi, les algorithmes les plus classiques de l'AR que nous présentons par la suite, partagent tous, chacun à sa manière, le même principe général. Ils consistent à approximer à partir d'expériences une fonction de valeur optimale *V* sans nécessiter la connaissance a priori d'un modèle du processus, et sans chercher à estimer ce modèle à travers les expériences accumulées. Plutôt qu'une fonction V qui associe une valeur à chaque état, nous verrons que l'on peut aussi chercher à approximer une fonction *Q* qui associe une valeur à chaque couple état/action, ce qui permet d'estimer une politique optimale.

1.7.2. Modèles d'apprentissage par renforcement

Il existe trois principales modèles d'apprentissage par renforcement :

- Le modèle d'apprentissage en ligne (**Figure 1.5**), où l'agent suit l'évolution du système jusqu'à la fin de l'expérience, en choisissant à chaque instant une action selon une règle d'apprentissage. Ce modèle est très appliqué à la marche robotique où on cherche une séquence d'actions qui permettent de maintenir le robot en équilibre horizontal et de le déplacer en avant avec une vitesse donnée.

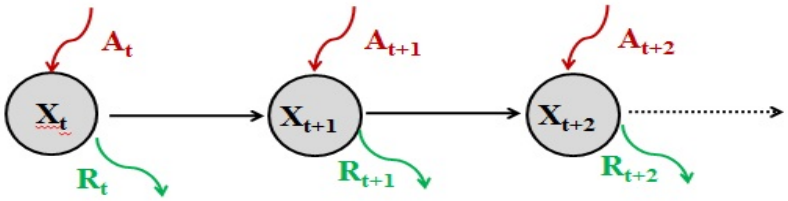


Figure 1.5 : Modèle d'apprentissage en ligne.

- Le modèle d'apprentissage épisodique (**Figure 1.6**) est constitué d'épisodes, chaque épisode se termine lorsque l'agent atteint un état terminal. Ce modèle est utilisé dans la navigation robotique, dans les jeux d'ordinateur, etc. Bref dans tout processus où l'agent doit atteindre un état but terminal.

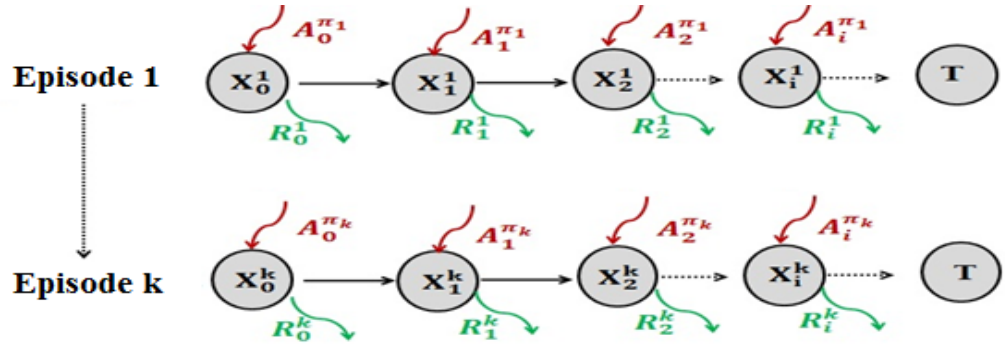


Figure 1.6 : Modèle d'apprentissage épisodique.

- Le modèle d'apprentissage génératif (**Figure 1.7**) où l'agent peut à chaque instant choisir l'état et l'action à exécuter. Ce modèle est très utilisé dans les jeux d'ordinateur où le programmeur peut à chaque instant choisir l'état voulu ou générer un état aléatoire.

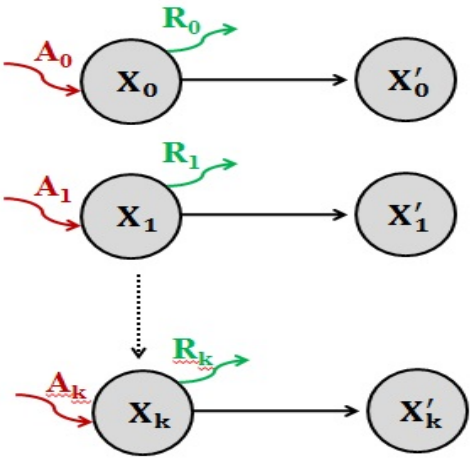


Figure 1.7 : Modèle d'apprentissage génératif.

1.8. Algorithmes d'apprentissage par renforcement

Dans cette section, nous présentons d'abord des méthodes pour estimer la fonction de valeur V^{π} d'une politique stationnaire π, puis nous exposons quelques algorithmes classiques d'AR basés sur ces méthodes. Nous nous limitons au critère actualisé qui est souvent le plus utilisé pour ce type d'approches. Cependant, des formules similaires peuvent être énoncées pour les autres critères.

1.8.1. Méthode de Monte Carlo

La fonction de valeur V^{π} associée à une politique déterministe π, donnée par l'équation 1.35, peut être estimée à l'aide de simulations qui génère des trajectoires indépendantes se terminant par un état absorbant durant lesquelles la politique π est suivie (**Figure 1.8**).

$$V^{\pi}(s) = \mathbb{E}\left(\sum_{t=1}^{T-1} R^{\pi}(s_t) | s_0 = s\right), s \in S \tag{1.35}$$

Où s_t, $R^{\pi}(s_t)$ sont respectivement l'état et le gain obtenu à l'instant t.

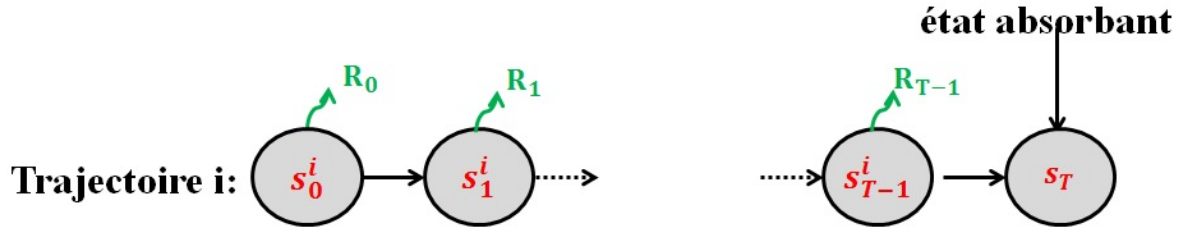


Figure 1.8 : Trajectoire suivie, sous une politique déterministe π, pour atteindre l'état absorbant S_T.

Maintenant, considérons les trajectoires suivies par une politique déterministe π, après n trajectoires $(s_0^i = s, s_1^i, \dots, s_{t_i}^i = T)$, i ≤ n, la moyenne de la fonction de valeur V^{π} est donnée par :

$$V_n(s) = \frac{1}{n}\sum_{i=1}^{n}\left[R^{\pi}(s_0^i) + \cdots + R^{\pi}(s_{T-1}^i)\right] = \frac{1}{n}\sum_{i=1}^{n} R_i^{\pi}(s) \tag{1.36}$$

Ce qui implique que $V_{n+1}(s)$ et $V_n(s)$ sont liées par :

$$V_{n+1}(s) = V_n(s) + \frac{1}{n+1}(R^{\pi}_{n+1}(s) - V_n(s)) \tag{1.37}$$

L'équation (1.37) peut s'écrire sous la forme :

$$\begin{aligned} V_{n+1}(s) &= V_n(s) + \boldsymbol{\alpha}_n\big(R^{\pi}_{n+1}(s) - V_n(s)\big) \\ &= V_n(s) + \alpha_n(R_n + R_{n+1} + \cdots + R_T - V_n(s)) \end{aligned} \tag{1.38}$$

Où $\boldsymbol{\alpha}_n$ est un taux d'apprentissage qui tend vers 0 au cours des itérations. On a $V_n(s)$ est un estimateur non biaisé si la valeur de $V_n(s)$ est mise à jour une seule fois lors de sa première rencontre le long de la trajectoire (méthode « first-visit » [SIN96], $V_n(s)$ tend presque sûrement vers $V^{\pi}(s)$. L'inconvénient de cette méthode est que la mise à jour s'effectue à la fin de chaque trajectoire observée, ce qui nécessite un nombre important d'épisodes pour avoir un estimateur non biaisé. De plus, une possibilité d'amélioration est de mettre à jour de la fonction de valeur à la suite de chaque transition du système, c'est la méthode de Différence Temporelle (TD).

1.8.2. Méthode de Différence Temporelle (DT)

L'algorithme d'apprentissage par Différence Temporelle (en anglais Temporal Différence) est un apprentissage par renforcement utilisant un mécanisme d'estimation temporelle pour la prédiction du temps d'arrivée d'une récompense. Il s'inspire d'études d'apprentissage chez les animaux. La règle de mise à jour de l'équation (1.38) peut être réécrite d'une autre manière (Equation 1.39) en introduisant le terme γ et en utilisant le fait que $V(S_T) = 0$.

$$\begin{aligned} V(s_t) = V(s_t) + \alpha(s_t)\ (\quad & R_t + \gamma V(S_{t+1}) - V(s_t) + \\ & R_{t+1} + \gamma V(S_{t+2}) - V(s_{t+1}) + \\ & R_{t+2} + \gamma V(S_{t+3}) - V(s_{t+2}) + \\ & \ldots\ldots\ldots\ldots \\ & \qquad \ldots\ldots\ldots\ldots \\ & R_T + \gamma V(S_T) - V(s_{T-1})\ ) \end{aligned} \tag{1.39}$$

Soit encore

$$V(s_t) = V(s_t) + \alpha(s_t)\ (\delta_t + \delta_{t+1} + \cdots + \delta_{T-1}), \tag{1.40}$$

En définissant le terme δ_t, appelé erreur de différence temporelle, par :

$$\delta_t = R_t + \gamma V(s_{t+1}) - V(s_t),\ \ t = 0, .., T-1 \tag{1.41}$$

Cette erreur peut être interprétée, en chaque état, comme une mesure de la différence entre l'estimation courante $V(s_t)$ et l'estimation corrigée à un coup $R_t + V(s_{t+1})$. Son calcul est possible dès que la transition (s_t, s_{t+1}, R_t) a été observée, et cela conduit donc à une version « on-line » de la règle de mise à jour (1.40) (**Algorithme 1.9**) où il n'est plus nécessaire d'attendre la fin de la trajectoire pour commencer à modifier les valeurs de V.

Ainsi, la méthode DT repose sur deux processus de convergences couplées, un processus qui estime de plus en plus, précisément, la récompense immédiate reçue dans

chacun des états et un autre qui approche de mieux en mieux la fonction de valeur résultant de ces estimations en les propageant de proche en proche. La convergence de cet algorithme a été prouvée par Dayan et Sejnowski [DAY94].

Algorithme 1.9: Algorithme DT

DT (S, A, π, V)
1: Initialiser V arbitrairement
2: Pour chaque épisode Faire
3: Initialiser l'état initial s
4 : Répéter
5: $a \leftarrow \pi$(s)
6: $(r, s') \leftarrow$ Simuler (s, a)
7: $V(s) \leftarrow V(s) + \alpha_t\,(r + \gamma V(s') - V(s))$
8: $s \leftarrow s'$
9: Jusqu'à atteindre l'état terminal

1.8.3. Méthode SARSA

La méthode DT permet d'estimer la fonction de valeur d'une politique π, mais l'agent est incapable d'en déduire quelle politique suivre, car il ne peut réaliser un pas de regard en avant pour déterminer quelle est l'action qui lui permettra de rejoindre l'état suivant de plus grande valeur. Une solution consiste à mettre à jour la fonction de valeur associée aux couples (état, action). Reprenons l'équation de Bellman qui peut s'écrire, en utilisant la fonction de valeur $Q(s,a)$ introduit par Watkins [WAT89] et associée aux couples (état, action) comme suit :

$$\forall s \in S, \forall a \in A, Q^*(s,a) = r(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) \underset{b \in A}{Max}(Q^*(s',b)) \quad (1.42)$$

Une politique optimale est donnée par l'équation :

$$\pi^*(s) = \underset{a \in A}{argmax}(Q^*(s,a)) \quad (1.43)$$

L'algorithme SARSA est similaire à l'algorithme TD sauf qu'il travaille sur la valeur des couples $(\boldsymbol{s}, \boldsymbol{a})$ plutôt que sur la valeur des états. Son équation de mise à jour est identique à celle de TD en remplaçant la fonction de valeur par la fonction de valeur du couple état/action. C'est ainsi que l'algorithme SARSA (**Algorithme 1.10**) utilise l'équation de mise à jour suivante :

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t\big(R_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\big) \quad (1.44)$$

En outre, l'information nécessaire pour réaliser une telle mise à jour, alors que l'agent réalise une transition, est le quintuple $(s_t, a_t, R_t, s_{t+1}, a_{t+1})$, d'où découle le nom de l'algorithme. L'inconvénient de l'algorithme SARSA est qu'il nécessite un pas de regard en avant et un temps de convergence trop long. C'est pour cela que l'algorithme Q-learning est proposé par le fait qu'il n'est plus nécessaire de déterminer un pas de temps à l'avance quelle sera l'action réalisée au pas de temps suivant.

Algorithme 1.10: Algorithme SARSA

SARSA (S, A, π^*)

1. Pour tout $s \in S$, $a \in A(s)$ Faire
   - $Q(s,a) \leftarrow 0$
2. Pour chaque épisode Faire
   - $t \leftarrow 0$; $s_t \leftarrow s_0$
   - $a_t \leftarrow$ choisir une action
   - Tant que (l'état terminal n'est pas atteint) Faire
     - $(s_{t+1}, r_t) \leftarrow$Simuler (s_t, a_t)
     - $a_{t+1} \leftarrow$ choisir une action
     - $Q(s_t, a_t) = Q(s_t, a_t) + \alpha_t\big(r_t + \gamma\, Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\big)$
     - $t \leftarrow t + 1$
3. Pour tout $s \in S$ Faire
   - $\pi^*(s) = argmax_{a \in A}(Q(s,a))$

1.8.4. Méthode Q-learning

L'équation de mise à jour utilisée dans la méthode Q-learning est la suivante :

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t \left(R_t + \gamma \max_{b \in A(s_{t+1})} Q(s_{t+1}, b) - Q(s_t, a_t) \right) \quad (1.45)$$

Il est clair que la différence essentielle entre SARSA et Q-learning se situe au niveau de la définition du terme d'erreur. Le terme $Q(s_{t+1}, a_{t+1})$ apparaissant dans l'équation (1.44) a été remplacé par le terme $\max_{b \in A(s_{t+1})} Q(s_{t+1}, b)$ dans l'équation (1.45). L'algorithme SARSA effectue ainsi les mises à jour en fonction des actions choisies effectivement alors que l'algorithme Q-learning (**Algorithme 1.11**) effectue les mises à jour en fonction des actions optimales mêmes si elles ne sont pas les actions optimales qui réalise.

Algorithme 1.11: Algorithme Q-learning

QL (S, A, π^*)

1. Pour tout s∈S, a∈ A(s) Faire
   - $Q(s,a) \leftarrow 0$
2. Pour chaque épisode Faire
   - $t \leftarrow 0$; $s_t \leftarrow s_0$
   - Tant que (l'état terminal n'est pas atteint) Faire
     - $a_t \leftarrow$ choisir une ε-gloutonne action
     - $(s_{t+1}, r_t) \leftarrow$Simule$(s_t, a_t)$
     - $Q(s_t, a_t) = Q(s_t, a_t) + \alpha_t \left(r_t + \gamma \max_{b \in A(s_{t+1})} Q(s_{t+1}, b) - Q(s_t, a_t) \right)$
     - $t \leftarrow t + 1$
3. Pour tout s∈S Faire
   - $\pi^*(s) = \text{argmax}_{a \in A}(Q(s,a))$

En outre, WATKINS [WAT89] a montré que l'algorithme Q-learning converge presque sûrement sous les conditions suivantes :

1. S et A sont finis.
2. Chaque paire (s, a) est visitée un nombre infini de fois.
3. $\sum_{t\geq 0} \alpha_t = \infty$ et $\sum_{t\geq 0} {\alpha_t}^2 < \infty$; $(\alpha_t = \frac{1}{n(s,a)})$.
4. $\gamma \leq 1$.
5. Pour toute politique il existe un état absorbant.

En pratique, la suite α_t est souvent définie par $\frac{1}{n(s,a)}$.

1.8.5. Dilemme Exploration/Exploitation

La balance entre l'exploration qui permet d'exploiter de nouvelles connaissances et l'exploitation qui exploite les connaissances acquises jusqu'ici est un problème de challenge, l'approche classique est la politique gloutonne ε-greedy ($\varepsilon \in [0,1]$) bien choisie explore avec la probabilité ε en choisissant une action au hasard et exploite avec la probabilité 1-ε en choisissant l'action gloutonne et la valeur ε peut décrémenter durant l'expérience [SCH12].

Une autre approche appelée Softmax [SUT98] permet de sélectionner l'action à exécuter a dans l'état s avec la probabilité définie par:

$$P(a,s) = \frac{e^{\frac{Q(s,a)}{\mathcal{T}}}}{e^{\sum_{a'}\frac{Q(s,a')}{\mathcal{T}}}} \Rightarrow \begin{cases} \mathcal{T} \to \infty \Rightarrow P = \frac{1}{|A|} \\ \mathcal{T} \to 0 \Rightarrow Greedy \end{cases} \quad (1.46)$$

Où $\mathcal{T}$ est un paramètre qui permet d'avoir une politique gloutonne s'il tend vers 0 et une exploration équiprobable des couples (a,s) s'il tend vers l'infini.

D'ailleurs, Ronen [RON02] propose une autre approche appelée Rmax qui consiste à explorer si le nombre d'occurrences du couple d'état-action est inférieur à un nombre prédéfini m et à exploiter sinon.

$$Q(a,s) = \begin{cases} Q(a,s)\ if\ n(s,a) \geq m \\ Rmax\ \ if\ n(s,a) < m \end{cases} \quad (1.47)$$

Enfin, l'approche proposée par Kaelbling [KAE96] basée sur la distribution de Boltzmann et qui tient compte des valeurs relatives de la fonction de valeurs Q(s, a) de chaque couple état/action. La probabilité de choisir une action dépend de la façon dont elle est comparée avec les autres. Une action est choisie avec la probabilité définie par l'équation suivante :

$$P(s,a) = \frac{e^{\frac{Q(s,a)-\max_b Q(s,b)}{\mathcal{T}}}}{\sum_a e^{\frac{Q(s,a)-\max_b Q(s,b)}{\mathcal{T}}}}, \quad (1.48)$$

Où $\mathcal{T}$ est la température dans la distribution de Boltzmann, elle permet de favoriser l'exploration s'il augmente et inversement.

Cependant, ces différents algorithmes d'AR convergent lentement pour des espaces d'états et d'actions assez grands [GUU97]. Ainsi, plusieurs approches ont étés proposées pour accélérer la convergence, entre autres, les modèles basés sur des options [HWA12, MOO93],

l'utilisation du principe de la trace d'éligibilité [WIE98, PEN96], les méthodes hiérarchiques ou topologiques [LIN93, HAF15], et l'élimination des états et des actions en ligne [SAR14].

1.8.6. Trace d'éligibilité et algorithmes TD(λ), SARSA(λ) et Q(λ)

Les algorithmes TD, SARSA et Q-learning présentent l'inconvénient de la mise à jour d'une seule valeur par pas de temps, celle de la valeur de l'état que l'agent est en train de visiter. Cette procédure de mise à jour est particulièrement lente. En effet, pour un agent ne disposant a priori d'aucune information sur la structure de la fonction de valeur, il faut au moins n expériences successives pour que la récompense immédiate reçue dans un état donné soit propagée jusqu'à un état distant du premier de n transitions. Une amélioration consiste à doter l'algorithme d'une mémoire des transitions appelée une trace d'éligibilité. C'est ainsi que Sutton et Barto [SUT98] ont proposé une classe d'algorithmes appelés « TD(λ) » qui généralisent l'algorithme TD au cas où l'agent dispose d'une mémoire de transitions. Egalement, les algorithmes SARSA et Q-learning ont été généralisés en SARSA (λ) et Q(λ) par WATKINS [WAT92] et Peng [PEN96]. Nous nous limitons, par la suite, à présenter l'algorithme Q(λ) (**Algorithme 1.12**) avec Z_t est la trace d'éligibilité accumulative [GRO08].

Algorithme 1.12: Algorithme Q (λ)

1. Pour tout s∈S, a∈ A(s) Faire
   - $Q_0(s,a) \leftarrow 0$; *//Initialisation de la fonction valeur* $Q(s,a)$
   - $Z_0(s,a) \leftarrow 0$; *//Initialisation de la trace d'éligibilité*

2. Pour chaque épisode Faire
   - $t \leftarrow 0$; $s_t \leftarrow s_0$
   - Tant que (l'état terminal n'est pas atteint) Faire
     - $a_t \leftarrow$ Choisir une action selon le mode d'exploration/exploitation utilisé
     - $(s_{t+1}, r_t) \leftarrow$ Simuler(s_t, a_t)
     - $Z_t(s_t,a_t) \leftarrow Z_t(s_t,a_t) + 1$
     - $\delta_t \leftarrow r_t + \gamma \max_{a' \in A(s_{t+1})} Q_t(s_{t+1},a') - Q_t(s_t,a_t)$
     - Pour tout s∈S, a∈ A(s) Faire
       - $Q_{t+1}(s,a) \leftarrow Q_t(s,a) + \alpha Z_t(s,a)\delta_t$
       - $Z_{t+1}(s,a) \leftarrow \gamma\lambda Z_t(s,a)$
     - $t \leftarrow t+1$

3. Pour tout s∈S Faire
   - $\pi^*(s) = \text{argmax}_{a\in A}(Q(s,a))$

La trace d'éligibilité Z_{t+1} augmente à chaque nouveau passage dans l'état associé, puis décroît exponentiellement au cours des itérations suivantes, jusqu'à un nouveau passage dans cet état.

1.9. Conclusion

Dans ce chapitre, nous avons présenté, tout d'abord, le cadre théorique des PDM complètement observables, leurs critères d'optimalité et leurs algorithmes de résolution. Ensuite, nous avons présenté brièvement les principes du PDM partiellement observables.

En outre, les algorithmes de résolution des PDM sont d'une complexité temporelle polynomiale en fonction de la dimension de l'espace d'états, ce qui les rend coûteux pour des espaces d'états de grande taille. Ainsi, pour y remédier à ce problème, nous avons présenté brièvement, quelques techniques : la technique d'élimination des actions, la réduction de l'espace d'états, la méthode de décomposition et le parallélisme.

Enfin, nous avons considéré les problèmes où l'agent ne connait pas a priori les fonctions de transition qui définissent la dynamique du système, ainsi que les récompenses qui évaluent ses performances. Nous avons exposé différentes méthodes d'apprentissage par renforcement qui sont très utilisées en robotique.

Le passage du cadre théorique des PDM à son application en robotique, nécessite une étape de modélisation, une connaissance de l'agent robot, ses différents types de plateformes, ainsi que les types de représentation de l'environnement dans lequel évolue l'agent robot. De la sorte, le chapitre suivant présente l'agent robot, ses différents constituants et des modèles PDM appliqués à la robotique.

Chapitre 2

Processus Décisionnels de Markov appliqués à la Robotique

2.1. Introduction

Les PDM deviennent de plus en plus utilisés en robotique, que ce soit dans la robotique classique axée sur la planification de trajectoires à l'aide d'un modèle géométrique et explicite du monde ou dans la robotique développementale considérée comme cognitive. En effet, les modèles des PDM s'adaptent aux incertitudes liés aux perceptions des robots et offrent des méthodes d'apprentissage afin de doter les robots de capacité à apprendre et à améliorer leur comportement en s'adaptant aux différents changements de l'environnement avec lequel ils interagissent.

Nous commençons ce chapitre, par une brève présentation de l'agent robot, ses différents constituants (plateformes, capteurs de perception, actionneurs et microcontrôleurs), ainsi que les différents types de représentation de l'environnement dans lequel il peut évoluer. Nous présentons, ensuite, les différents systèmes de localisation utilisés en robotique. Enfin, nous exposons quelques modèles de décision markoviens utilisés en robotique que ce soit dans un environnement complètement observable ou dans un environnement partiellement observable ou inconnu.

2.2. Agent robot

Selon Ferber **[**FER95**]**, un agent est une entité physique ou virtuelle qui est capable d'agir dans un environnement et qui peut communiquer directement avec d'autres agents. Par perception de son environnement, un agent récolte des informations qu'il utilisera ensuite afin de décider de façon autonome comment agir.

Un robot est un cas particulier d'agent qui est généralement équipé de capacités de perceptions, de décisions et d'actions qui lui permettent d'agir de manière autonome dans son environnement en fonction des perceptions qu'il en a. C'est un agent qui est généralement en boucle de perception-action (**Figure 2.1**).

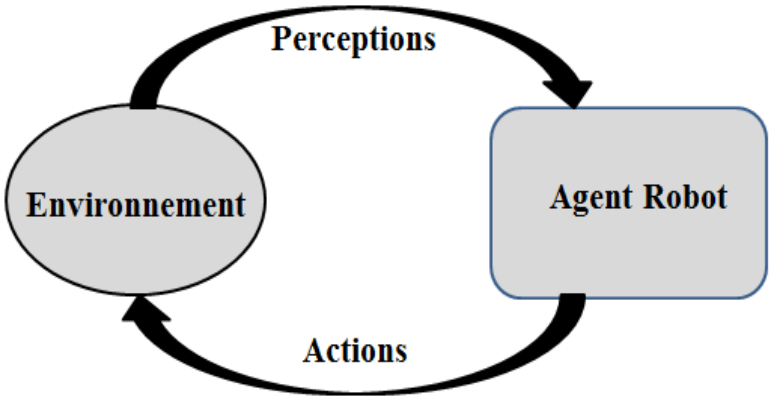


Figure 2.1 : L'agent robot en boucle de perception-action.

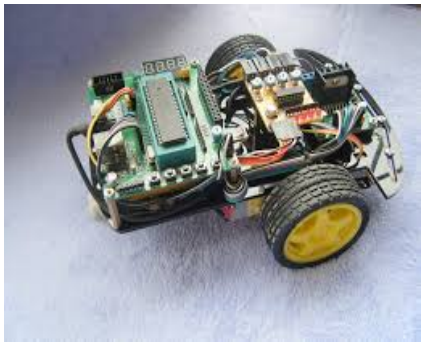
Les perceptions sont assurées par des capteurs tels que les cameras vidéos, les capteurs laser ou infra-rouge, les capteurs ultrasons, etc. Les actions se font par des bras, des roues, des pinces, etc. Le processus complet qui permet à un robot de mémoriser son environnement, puis de s'y déplacer pour rejoindre un but, peut être découpé en trois phases : la cartographie, la localisation, la modélisation et la planification.

La cartographie est la phase qui permet la construction d'une carte reflétant la structure spatiale de l'environnement à partir des différentes informations recueillies par le robot. Une telle carte étant disponible, la localisation permet alors de déterminer la position du robot dans la carte qui correspond à sa position dans son environnement réel. La modélisation et la planification, enfin, est la phase qui permet, connaissant la carte de l'environnement et la position actuelle du robot, de décider des mouvements à effectuer afin de rejoindre un but fixé dans l'environnement.

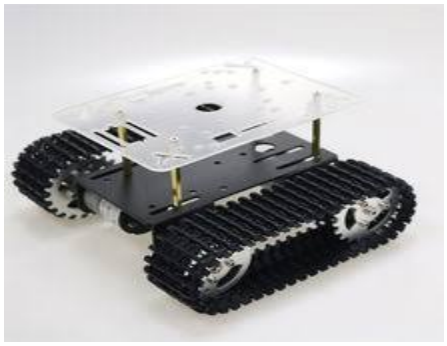
2.2.1. Plates-formes mobiles

Le choix d'une plate-forme robotique dépend du milieu d'évolution et des fonctionnalités mises en œuvre dans un robot. Différentes types de plates-formes mobiles peuvent être utilisées en robotique.

Plate-forme différentielle : Ce type de plate-forme comporte deux roues commandées indépendamment et une ou plusieurs roues folles sont ajoutées à l'avant ou à l'arrière du robot pour assurer sa stabilité (**Figure 2.2-a**). Cette plate-forme est très simple à commander, puisqu'il suffit de spécifier les vitesses des deux roues, et permet de plus au robot de tourner sur place, ce qui va simplifier la planification de déplacement et la commande du robot. Ce type de plate-forme peut également être utilisé avec des chenilles ce qui fournit une capacité de franchissement de petits obstacles intéressante (**Figure 2.2-b**).



a



b

Figure 2.2 : Exemples de plate-forme différentielle (s.d. Images libres de droits).

Plate-forme omnidirectionnelle: Ce type de plate-forme holonome rend le contrôle de la rotation et de la translation d'un robot plus simple. Deux catégories sont utilisées, le premier utilise trois ou quatre roues qui tournent à la même vitesse pour fournir une translation et un mécanisme qui permet d'orienter simultanément ces roues dans la direction du déplacement souhaitée (**Figure 2.3-a**), le deuxième utilise des roues dites "suédoises", qui n'offrent pas de résistance au déplacement latéral (**Figure 2.3-b**). Les déplacements dans toutes les directions et en rotation sont obtenus en faisant varier individuellement les vitesses des roues.

a

b

Figure 2.3 : Exemples de plate-forme omnidirectionnelle (s.d. Images libres de droits).

Plate-forme non holonome : Telles que les voitures, ces plates-formes sont également utilisées en robotique mobile (**Figure 2.4**). Ils ne peuvent pas tourner sur place et doivent manœuvrer, ce qui peut être difficile dans des environnements encombrés.

Figure 2.4 : Exemple de plate-forme non holonome (s.d. Image libre de droits).

Plate-forme à pattes : Des plates-formes à deux, quatre ou six (voir plus) pattes peuvent également être utilisées (**Figures 2.5**). Elles peuvent se déplacer sur des terrains assez complexes. Ces plates-formes sont plus complexes à commander et le simple contrôle de la stabilité et d'une allure de marche correcte reste aujourd'hui difficile, ce qui les rend en général relativement lentes. L'AR offre des méthodes permettant d'apprendre à ces types de robot de marcher correctement.

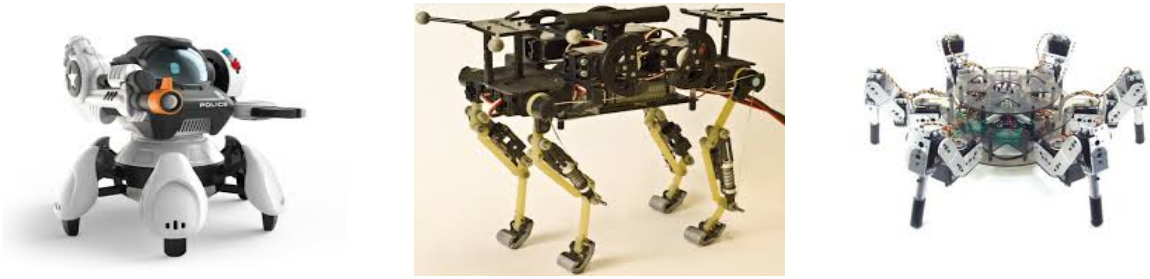


Figure 2.5 : Exemples de plates-formes à pattes (s.d. Images libres de droits).

2.2.2. Capteurs

Le choix d'un système de perception dépend souvent du milieu d'évolution du robot mobile et des fonctionnalités mises en œuvre sur le robot pour qu'il puisse remplir sa mission. La précision désirée et la fréquence d'acquisition influencent sur le choix des capteurs. Il s'agit donc, d'une contrainte qui pèsera inévitablement sur le choix d'un système de perception. On distingue deux grandes familles de capteurs:

- **Les capteurs proprioceptifs** qui effectuent leurs mesures par rapport à ce qu'ils perçoivent localement du déplacement du robot.
- **Les capteurs extéroceptifs** qui fournissent des informations sur le monde extérieur au robot en se basant sur des mesures prises par rapport à l'environnement.

Capteurs proprioceptifs : Deux familles de capteurs proprioceptifs peuvent être utilisés. Les capteurs de déplacement qui comprennent les odomètres, les accéléromètres, les radars Doppler, les mesureurs optiques permettant de mesurer des déplacements élémentaires, des variations de vitesse ou d'accélération sur des trajectoires rectilignes ou curvilignes. Les capteurs d'attitude qui sont principalement constitués par les gyroscopes, les gyromètres, les capteurs inertiels composites, les inclinomètres et les magnétomètres.

- Les odomètres permettent de fournir une quantification des déplacements curvilignes du robot en mesurant la rotation de ses roues.

- Le radar Doppler fournit une estimation instantanée de la vitesse linéaire d'une plate-forme mobile par rapport à un objet de la scène en se basant sur l'effet Doppler-Fizeau.
- Le gyroscope et le gyromètre permettent de mesurer une variation et une vitesse angulaire.
- Le gyrocompas est un capteur qui permet de mesurer le cap. Il est composé d'un gyroscope et d'un compas magnétique. Le gyrocompas conserve le nord magnétique durant tout le déplacement du véhicule, après l'avoir initialement déterminé de façon autonome.
- Le magnétomètre qui est aussi appelé compas magnétique mesure la direction du champ magnétique terrestre pour déduire l'orientation du robot.

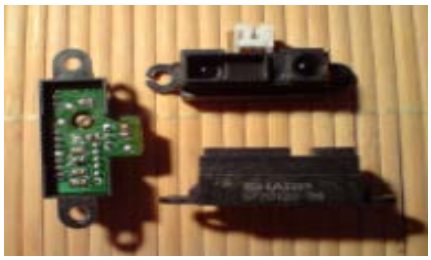
Capteurs extéroceptifs : Les capteurs extéroceptifs permettent de percevoir le milieu d'évolution du robot. Ils sont généralement le complément indispensable aux capteurs présentés précédemment. Deux familles de capteurs extéroceptifs embarqués peuvent être identifiées:

- Les capteurs à ultrasons utilisent l'air comme milieu de propagation. La méthode de mesure consiste à exciter une céramique piézo-électrique à l'aide de quelques impulsions de fréquence égale à la fréquence de la pastille. Connaissant la vitesse de propagation de l'onde acoustique dans un milieu donné, la mesure du temps de vol de l'onde permet d'obtenir la distance d'un objet par rapport au capteur. Le HC-SR04 (**Figure 2.6**) est un exemple de capteurs Ultrason très connu permettant de mesurer des distances jusqu'à 4 m.

Figure 2.6 : Exemple de capteur Ultrason (HC-SR04) (s.d. Image libre de droits).

- Les capteurs optiques (lumière visible ou infrarouge) sont basés sur l'émission et la réception d'une onde lumineuse et peuvent être utilisés pour détecter des obstacles, suivre une ligne noire, mesurer une distance par rapport à un obstacle, etc. Parmi ces capteurs, on distingue les capteurs à laser qui sont basés sur l'émission d'une onde électromagnétique structurée, ce qui permet d'obtenir un faisceau d'ondes très concentré, contrairement aux capteurs ultrasonores. Le module TCRT5000 (**Figure 2.7**, droite) est composé d'un émetteur-récepteur TCRT5000, d'un amplificateur et de deux sorties numérique et analogique et peut être utilisé pour détecter des obstacles à proximité ou suivre une ligne de couleur noire entourée du blanc. La famille GP2D de Sharp (**Figure 2.7**, gauche) est connue en robotique pour servir de capteur de présence ou de mesurer la distance par rapport à un objet réflecteur, la distance est évaluée par une bande de récepteurs à l'intérieur du capteur.

Module TCRT5000



Sharp GP2D

Figure 2.7 : Exemples de capteurs optiques (s.d. Images libres de droits).

Les systèmes de vision : Les systèmes de vision en robotique sont basés sur l'utilisation d'une caméra CCD. L'arrivée des capteurs CCD (Charge Coupled Device), en 1975, a été déterminante dans l'évolution de la vision : la rapidité d'acquisition, la robustesse et la miniaturisation sont autant d'avantages qui ont facilité leur intégration. Les systèmes de vision sont très performants en termes de portée, précision et quantité d'informations exploitables. Ils sont de plus les seuls capables de restituer une image sensorielle de l'environnement la plus proche de celle perçue par l'être humain. En revanche, l'inconvénient majeur de tels systèmes de perception se situe au niveau de la gestion du flux important de données exploitables : traiter une image demeure une opération délicate et surtout coûteuse en temps de calcul. Actuellement les systèmes de vision omnidirectionnels sont de plus en plus utilisés du fait qu'ils permettent de couvrir un angle de 360 degrés.

D'autres capteurs peuvent être utilisés tels que les capteurs tactiles ou de touche qui permettent de stopper le robot en cas de touche d'obstacle, les capteurs de localisation par satellite (GPS, en anglais : Global Positioning System), les balises (signal radio, codes couleurs ou des codes-barres) utilisées comme repères artificielles, les capteurs de température, de force, d'humidité, etc. Généralement, tout type de capteur utile à une application déterminée peut être utilisé.

2.2.3. Représentation de l'environnement

Deux principaux types de représentation de l'environnement sont utilisés, les cartes topologiques et les cartes métriques [FIL11].

Cartes topologiques : Ce type de cartes permettent de représenter l'environnement du robot sous forme de graphe [DED99, KUN99] (**Figure 2.8**). Les nœuds du graphe correspondent à des lieux que le robot peut atteindre. Les arêtes liant les nœuds, marquent la possibilité pour le robot de passer directement d'un lieu à un autre et mémorisent en général la manière de réaliser ce passage.

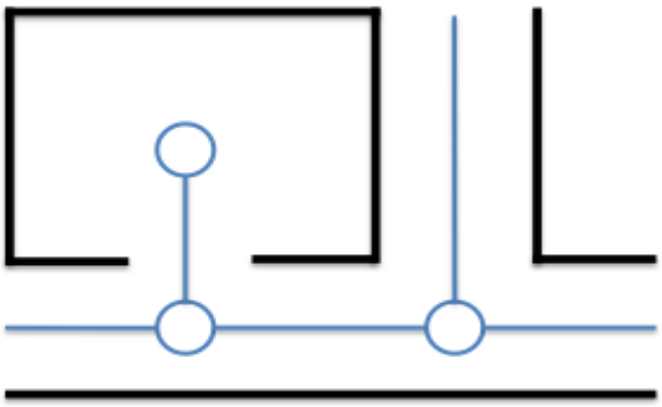


Figure 2.8 : Exemple de carte topologique (s.d. Image libre de droits).

Cartes métriques : Dans ce type de carte [LEV90, FED99] (**Figure 2.9**), l'environnement est représenté par un ensemble d'objets auxquels sont associées des positions dans un espace métrique, généralement en deux dimensions. Les perceptions permettent, en utilisant un modèle métrique des capteurs, de détecter ces objets et d'estimer leur position par rapport au robot. Ces objets peuvent être des obstacles que le robot pourra rencontrer dans son environnement.

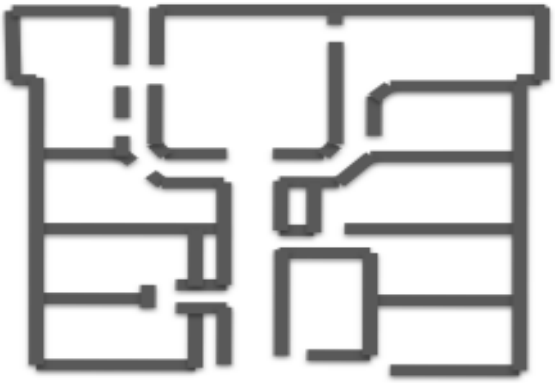


Figure 2.9 : Exemple de carte métrique (s.d. Image libre de droits).

Cartes hybrides : C'est une représentation hybride [MOZ07] mélangeant ces deux types : les représentations topo-métriques qui sont des cartes topologiques contenant des informations métriques sur les arêtes du graphe (**Figure 2.10**). Ce type de représentation est par exemple bien adapté pour construire des cartes à partir de la vision : chaque nœud du graphe peut être associé à une image, relié à ses voisins par des informations obtenues par l'odométrie du robot ou par odométrie visuelle.

Figure 2.10 : Exemple de carte hybride (s.d. Image libre de droits).

2.2.4. Localisation

La localisation d'un robot évoluant dans un environnement est très importante du faite qu'elle permette de savoir la position du robot ainsi que la correction après un déplacement qui n'atteint pas la position désirée. On peut distinguer deux types de méthodes de localisation [FIL11] : les méthodes de localisation relatives, basées sur l'utilisation des capteurs proprioceptifs et les méthodes de localisation absolues, basées sur l'utilisation de capteurs extéroceptifs.

Localisation relative : Appelée aussi navigation à l'estime, consiste à évaluer la position, l'orientation, et éventuellement la vitesse du robot mobile par intégration des informations fournies par des capteurs dits proprioceptifs. L'intégration se fait par rapport au point de départ du robot. Ces données peuvent être des informations de déplacement (odomètre), de vitesse (vélocimétrie) ou d'accélération (accéléromètre). Ces systèmes permettent d'obtenir un flux relativement important au niveau de l'obtention des estimations de position. Cette caractéristique intéressante a favorisé l'utilisation de ces systèmes de localisation en robotique mobile.

Localisation absolue : C'est une technique qui permet à un robot de se repérer directement dans son milieu d'évolution, que ce soit en environnement extérieur (mer, espace, terre), ou en environnement intérieur (ateliers, immeubles, centrales nucléaires...). Ces méthodes de

localisation sont basées sur l'utilisation de capteurs extéroceptifs. Ce type de localisation absolue nécessite toujours une représentation de l'environnement. Le robot possède donc une banque de données regroupant les caractéristiques des références externes en utilisant des repères naturels ou artificiels.

2.2.5. Actionneurs

Les actionneurs sont à l'origine d'une action du robot (un moteur pour se déplacer, une lampe ou un haut-parleur pour avertir, etc.). Les principaux actionneurs d'un robot sont les moteurs avec les quelles, il peut se déplacer, lever sa main pour prendre un objet, etc. On distingue trois principaux types de moteurs : Les moteurs à courant continu généralement utilisés pour commander des roues, les moteurs pas à pas utilisés pour contrôler des pattes ou des mains et les servomoteurs qui sont des moteurs à courant continue équipés d'une interface de commande permettant une rotation paramétrable dans un angle de 180° ou 360°.

2.2.6. Microcontrôleurs

Le microcontrôleur est un système programmable qui ressemble à un ordinateur ayant des performances réduites, mais de faible taille et consomment peu d'énergie. Il est très utilisé dans l'électronique embarqué (voiture, porte de garage, robots, …). C'est le cerveau du robot qui traite les données reçus par ces capteurs et ordonne ces actionneurs, c'est un circuit intégré qui contient en interne : (1) une unité centrale ou CPU (Central Processing Unit) qui exécute le programme et pilote ainsi tous les autres éléments ; (2) une mémoire morte ou ROM (Read Only Memory) dont le contenu est conservé même en cas de coupure de courant et qui contient le programme que va exécuter l'unité centrale ; (3) une mémoire vive ou RAM (Random Access Memory) dans laquelle l'unité centrale peut lire et écrire à tout instant et utilisée dans les phases de calcul du programme, pour stocker des résultats intermédiaires et les variables d'une application ; (4) des entrées/sorties permettent au microcontrôleur de communiquer avec ces capteurs, ces actionneurs et le monde extérieur.

Il existe différents types de microcontrôleurs utilisés dans les robots selon leur puissance de calcul, capacité de stockage d'informations, nombre et types d'entrées/sorties ainsi que les interfaces intégrés dédiés à des fonctions plus précises. Nous citons par exemple :

- Le microcontrôleur PIC (en anglais : Programmable Interface Controller) est un circuit fabriqué par la société américaine « Arizona MICROCHIP Technology ».
- Le microcontrôleur Atmel AVR (ATmega, ATtiny, etc.) intégré dans les cartes Arduino et qui est un microcontrôleur open source avec une interface de programmation sous le langage C. Il a été adopté par la communauté des «Makers» permettant toute sortes de réalisations diverses et rendant facilement accessible ce qui nécessitait avant de l'électronique compliqué.

2.3. Quelques modèles de décision markoviens appliqués à la Robotique

Le passage du cadre théorique du PDM à son application en robotique nécessite la définition des paramètres du modèle qui sont l'espace d'états, l'espace d'actions, la fonction de récompenses, la fonction de transitions et l'espace du temps. Nous présentons, dans cette section, quelques modèles PDM appliquées à la navigation robotique et en apprentissage à la marche robotique.

2.3.1. Modèle de décision markovien appliqué à la navigation Robotique

Un PDM est un modèle qui peut être paramétré de diverses manières. Pour passer à son application à la robotique mobile le choix du type de représentation de l'environnement, l'ensemble des états, l'ensemble des actions sont les premiers paramètres à définir. Mais, un

PDM doit également être paramétré au niveau de ces éléments fondamentaux comme les fonctions de récompenses et de transitions. Nous présentons un modèle PDM appliqué à la robotique mobile où l'environnement est discrétisé sous forme de grilles d'occupation [MOR85, THR98], carrées (**Figure 2.11**-gauche) ou hexagones (**Figure 2.11**-droite). Les états seront définis comme étant des positions géographiques alors que les actions, des déplacements d'une grille à une autre. Des récompenses seront données à chaque déplacement en fonction de l'état de transition (état libre, occupé ou but). Les probabilités de transitions sont des données du problème qui dépendent de l'environnement et des caractéristiques du robot.

L'inconvénient des grilles d'occupation est qu'ils utilisent une quantité de mémoire importante, qui croît proportionnellement avec la surface de l'environnement. Pour s'affranchir de ce problème, certains modèles font appel à des discrétisations irrégulières de l'espace ou à des discrétisations hiérarchiques. De telles discrétisations permettent de s'adapter à la complexité de l'environnement, en représentant de manière grossière les grands espaces libres.

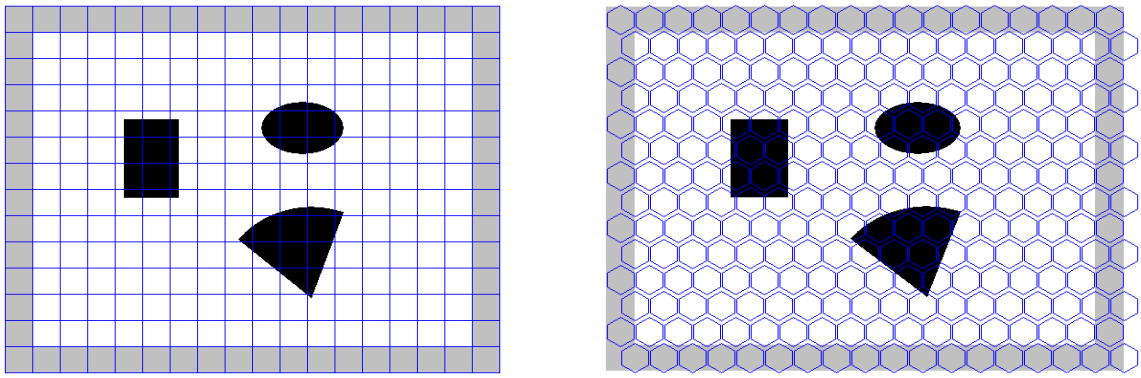
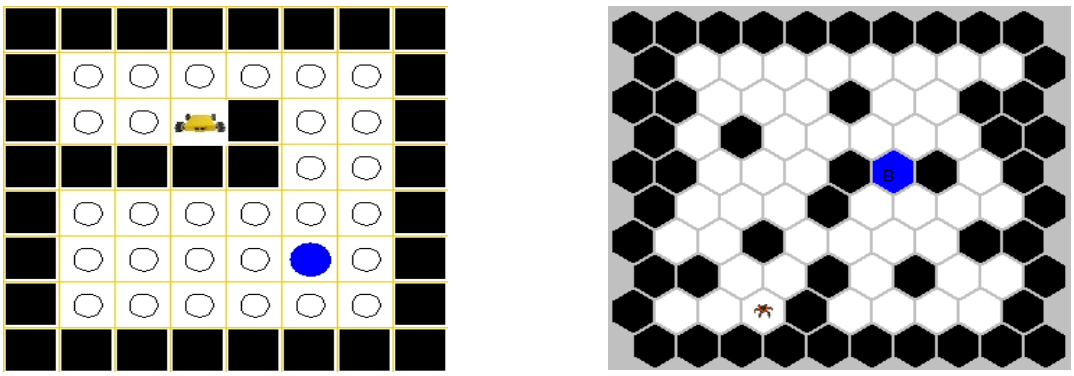


Figure 2.11 : Exemple d'un environnement discrétisé sous forme de grilles carrées et hexagonales.

Définition des états : Les états définissent donc des positions géographiques, ainsi dans cette représentation chaque case correspond à un état qui est libre ou occupé. La **figure 2.12** représente deux exemples de petit environnement, le premier est discrétisé en grilles carrées, le deuxième en grilles hexagones, une case définit donc, un état, pour un obstacle la case est entièrement noire, pour un but un cercle plein en couleur bleu ou case en bleu.



Grilles carrées Grilles hexagones

Figure 2.12 : Représentation des grilles dans un environnement.

Définition des actions : Pour les grilles carrées, nous pouvons contrôler le robot à travers quatre actions : Avancer Haut d'une case, Avancer Bas d'une case, Avancer Gauche d'une case et Avancer Droite d'une case (**Figure 2.13-a**). Quant aux grilles hexagonales, nous pouvons contrôler le robot à travers six actions (**Figure 2.13-b**), les six positions qui entourent une position donnée. Ces actions sont simples et peuvent être facilement

programmées comme commandes de base du robot. Il est possible d'ajouter une action « Rester sur place » qui peut être très utile dans certaines applications. Cette action est aussi utile dans le cas où le robot n'a pas de chemin pour se déplacer vers une position donnée.

a) Quatre actions de déplacement b) Six actions de déplacement

Figure 2.13 : Actions de déplacement dans les deux types de grilles.

Fonction de récompense : La fonction de récompense permet de désigner le (ou les) but(s) à atteindre par le système. Dean et al [DEA93] ont utilisé une fonction très simple définie par : r(*s*)=0 si *s* est l'état but et r(*s*)=-1 sinon. Dans le cas d'un environnement contenant des zones dangereuses (mines ou obstacles), nous proposons une fonction de récompense comme suit :

r(s, a) =	$+N$	Si	Transition vers un état but
r(s, a) =	-1	Si	Transition vers un état libre
r(s, a) =	0	Si	Action = « rester sur place »
r(s, a) =	$-M$	Si	Transition vers un état obstacle

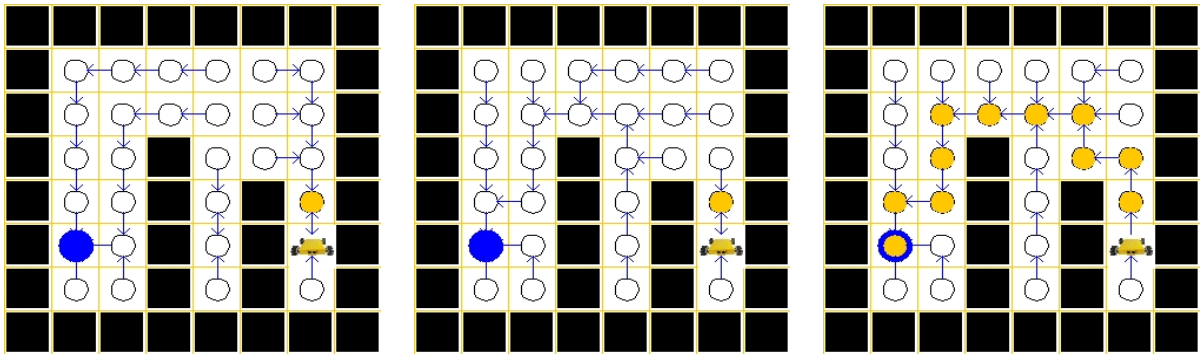
où N>>1 est la récompense acquise lorsqu'une transition vers un état but est observée (l'état but se voit allouer une récompense très grande), -*M* (*M*>>1) est la récompense acquise si il y a une possibilité de transition vers un état dangereux (l'état constituant une zone dangereuse se voit allouer une punition très grande). Le déplacement du robot, d'une case libre à une autre, entraine une perte d'énergie, on donne donc, une récompense négative de l'ordre d'une unité (-1). La récompense pour l'action « rester sur place » est égale à 0. Les états buts et les états dangereux seront définies comme des états absorbants, impossible de transiter de ces états, ainsi un état but est un état très attirant alors qu'un état dangereux est un état très repoussant.

Fonction de transition : La fonction de transition est un élément fondamental dans l'application des PDM. En effet, c'est grâce à cette fonction que nous prenons en compte l'incertitude quant aux effets des actions. Cette fonction est considérée comme une donnée du problème ou est générée expérimentalement. La **figure 2.14** représente les valeurs de probabilités de transitions vers les états proches de l'action spécifiée par la flèche pour les deux types de grilles. Il en est de même pour les autres actions. Ces valeurs sont choisies arbitrairement pour faire un test de simulation, elles dépendent du type des roues utilisées par le robot et de la structure du sol de l'environnement et peuvent être déterminées expérimentalement par un apprentissage.

Figure 2.14 : Probabilités de transition vers les états proches dans les deux types de grilles.

Le paramètre de pondération **α** permet d'accorder plus ou moins d'importance aux récompenses futures. Pour un même environnement, la valeur de ce paramètre influe sur les différents politiques obtenus.

Exemple de simulation : La **figure 2.15** représente des politiques calculées pour un problème de navigation robotique vers un état but donné utilisant l'algorithme IV. Pour un même environnement discrétisé en grilles carrées et en utilisant des valeurs différentes de α, la politique est déterminée sans utiliser l'action « rester sur place », la flèche indique l'action optimale que le robot doit prendre s'il se trouve dans une telle case.

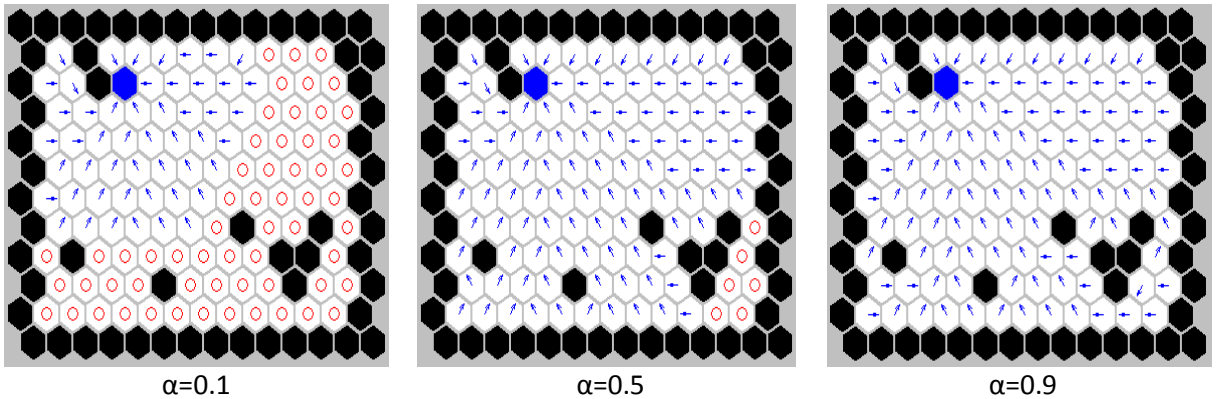


α=0.1 α=0.5 α=0.9

Figure 2.15 : Politiques calculées, pour différents valeurs de α, dans un exemple d'environnement discrétisé sous forme de grilles carrées.

Nous remarquons que ces trois politiques diffèrent et que c'est la politique (α=0.9) qui a permis d'accomplir la mission d'atteindre le but, donc plus la valeur de α est proche de 1 plus les politiques sont efficaces.

La **figure 2.16** représente des politiques calculées pour un environnement discrétisé en grilles hexagones, en utilisant des valeurs différentes de α. L'action « Rester sur place » est utilisée, la direction de la flèche indique l'action optimale que le robot doit prendre s'il se trouve dans une telle case, un petit cercle dans une case indique que l'action optimale est « Rester sur place ».



α=0.1 α=0.5 α=0.9

Figure 2.16 : Politiques calculées, pour différents valeurs de α, dans un exemple d'environnement discrétisé sous forme de grilles hexagonales.

Il est clair l'influence de la valeur de α, par exemple pour α très petit (Figure 2.6, α=0.1), seuls les états proches du but qui mènent vers ce dernier, alors que dans les états loin du but, le robot préfère rester sur place. Alors que pour α assez grand, tous les états mènent vers le but.

2.3.2. Modèle de décision markovien appliqué à un robot transporteur d'objets

Nous proposons, dans cette section, un modèle de décision markovien appliqué à un agent robot transporteur d'objets ou de marchandises d'une position B_c à une autre position B_d. En cas de décharge de sa batterie, il a la possibilité de se charger d'énergie dans la position la plus proche parmi les trois positions possibles B_{e1}, B_{e2} et B_{e3} (**Figure 2.17**). En plus, le robot est équipé d'un capteur de force qui lui indique la masse *M* des objets déposés.

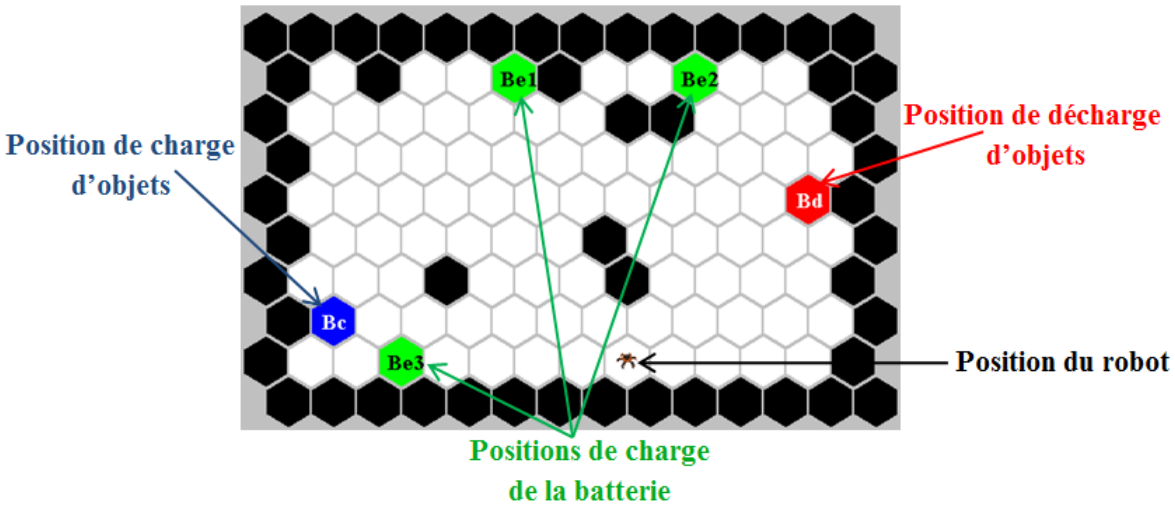


Figure 2.17 : Exemple d'environnement pour un robot transporteur d'objets.

Le robot doit se rendre à la position B_c et rester sur place jusqu'à charger une masse M d'objets supérieure à une masse minimum prédéfini M_{min}, puis s'orienter vers une autre position B_d pour décharger ses objets. Il reste dans la position B_d jusqu'à ce que la masse M soit égale à zéro. Il reprend ainsi son chemin vers la position B_c pour charger d'autres objets et ainsi de suite dans une boucle infinie entre les positions B_c et B_d. De plus, le robot peut se rendre à une position parmi trois autres positions possibles où il peut se charger d'énergie en cas de décharge de sa batterie, les trois positions de charge d'énergie sont Be_1, Be_2 et Be_3. Lorsqu'aucune masse n'est déposée sur le robot, il doit chercher à atteindre la position du but B_c, et lorsque la masse déposée est supérieure ou égale à un seuil M_{min}, le robot doit atteindre la position du but B_d. Dans tous les cas, lorsque la charge de la batterie est inférieure ou égale à un seuil Vmin prédéfini, le robot doit se rendre à la position la plus proche, parmi les trois positions Be_1, Be_2 et Be_3 (Figure 2.17).

Nous supposons que l'environnement est connu et nous définissons alors notre modèle PDM comme suit :

- **Espace d'états :** On définit l'état du robot sous forme d'un couple (position_robot, état_robot) où la variable état_robot est soit égale à 0 (batterie déchargé), soit égale à 1 (batterie chargé et robot déchargé d'objets), soit égale à 2 (batterie chargé et robot chargé d'objets). Ainsi, on a 3|S| états possibles du système où |S| est le nombre de cases. Les états buts sont donc : (Bc, 1), (Bd,2) et (Bei,0) ayant une même récompense lorsqu'une transition vers ces états est observée et aussi lorsque l'action « Rester sur place » est choisie.
- **Espace d'actions :** Les actions sont les six directions $d_i, i = 1,..,6$ dans l'hexagone ; l'action θ « rester sur place » ; l'action $C_m = \theta$ « charger les marchandises » qui ne s'applique que dans l'état (Bc,1) et transite le système vers l'état (Bc,2) ; l'action $D_m = \theta$ « Décharger les marchandises » qui ne s'applique que dans l'état (Bd,2) et transite le système vers l'état (Bd,1) ; l'action C_b « Charger batterie » qui ne s'applique que dans l'état (Bei,0) et transite le système vers l'état (Bei,1) ou (Bei,2);

- **Fonction de transition :** Nous utilisons la fonction de transition définie dans la figure 2.14, mais il faut noter que certains transitions sont impossibles : $(S_i,2)\rightarrow(Bc,1)$; $(S_i,0)\rightarrow(Bc,1)$; $(S_i,1)\rightarrow(Bd,2)$; $(S_i,0)\rightarrow(Bd,2)$; $(S_i,2)\rightarrow(Bc,1)$; $(S_i,0)\rightarrow(Bc,1)$;
- **Fonction de récompense :** On définit le coût (récompense négative) de la transition d'état vers un autre sous forme d'énergie ***e*** consommée lors d'un déplacement d'une case à une autre.

Finalement, une stratégie optimale est calculée (offline) par un algorithme de résolution tel que IV ou IVGS et mémorisée sous forme de matrice (position_robot, état_robot, action_optimale).

2.3.3. Agent robot de position inconnue orienté vers un but

Comme nous l'avons montré au du chapitre précédent, une politique optimale donnée par la résolution d'un PDM est un ensemble de couples (état, action), spécifiant dans chaque état une action optimale à exécuter. Lors de l'exécution d'une politique, la position courante du robot est très rarement connue avec certitude, à cause des incertitudes liées aux actions. Pour pallier à ce problème, un robot est muni de capteurs pour percevoir ou observer son environnement et s'y repérer. Cependant, ces capteurs renvoient des données bruitées, ce qui rend la localisation très incertaine. De plus, dans chaque environnement, nous aurons toujours des positions différentes pour lesquelles les observations du robot seront identiques.

Observation du robot : Considérons un environnement à grilles carrées, nous supposons que le robot est équipé de capteurs capables de détecter la présence d'un obstacle dans les huit états qui l'entoure. Nous utilisons la distribution de probabilité (Eq 1.34) sur les états de croyance pour déterminer la probabilité que le système soit dans l'état ***s',*** après avoir effectué l'action *a* et observer ***o***, les valeurs des probabilités de chaque état sont mises à jour en fonction des positions probables précédentes, de l'observation du robot et de la fonction de transition. Au pas initial, nous prenons les observations du robot, puis nous calculons la distribution de probabilité initiale. Les états possibles sont, au début, équiprobables. Nous exécutons une action aléatoire et nous reprenons de nouveau les observations faites par le robot. A chaque pas, nous exécutons l'action correspondante à l'état le plus probable. Nous répétons cette phase jusqu'à ce que le robot arrive au but qui peut être reconnu par un indice quelconque (**Algorithme 2.1**).

Algorithme 2.1: Algorithme de localisation et d'orientation vers un but.

Données : Environnement connu, position du but connue.
1: Calculer offline une politique optimale à l'aide de l'**algorithme 1.3**
2: Observer l'environnement proche
3: Déterminer les états équiprobables à l'instant t=0, $B_0(s)$
4: $t \leftarrow 0$
4: **Répéter**
5: | Choisir une action qui correspond à l'état le plus probable.
6: | Déplacer le robot selon l'action choisie.
4: | $t \leftarrow t+1$
7: | Observer l'environnement proche
8: | Calculer la distribution de probabilité $B_t(s)$ à l'instant t en fonction $B_{t-1}(s)$ (Eq 1.34)
9: **Jusqu'à** atteindre le but

a. Exemple de simulation

La **figure 2.18** représente les étapes de localisation et d'orientation vers un but dans un petit environnement.

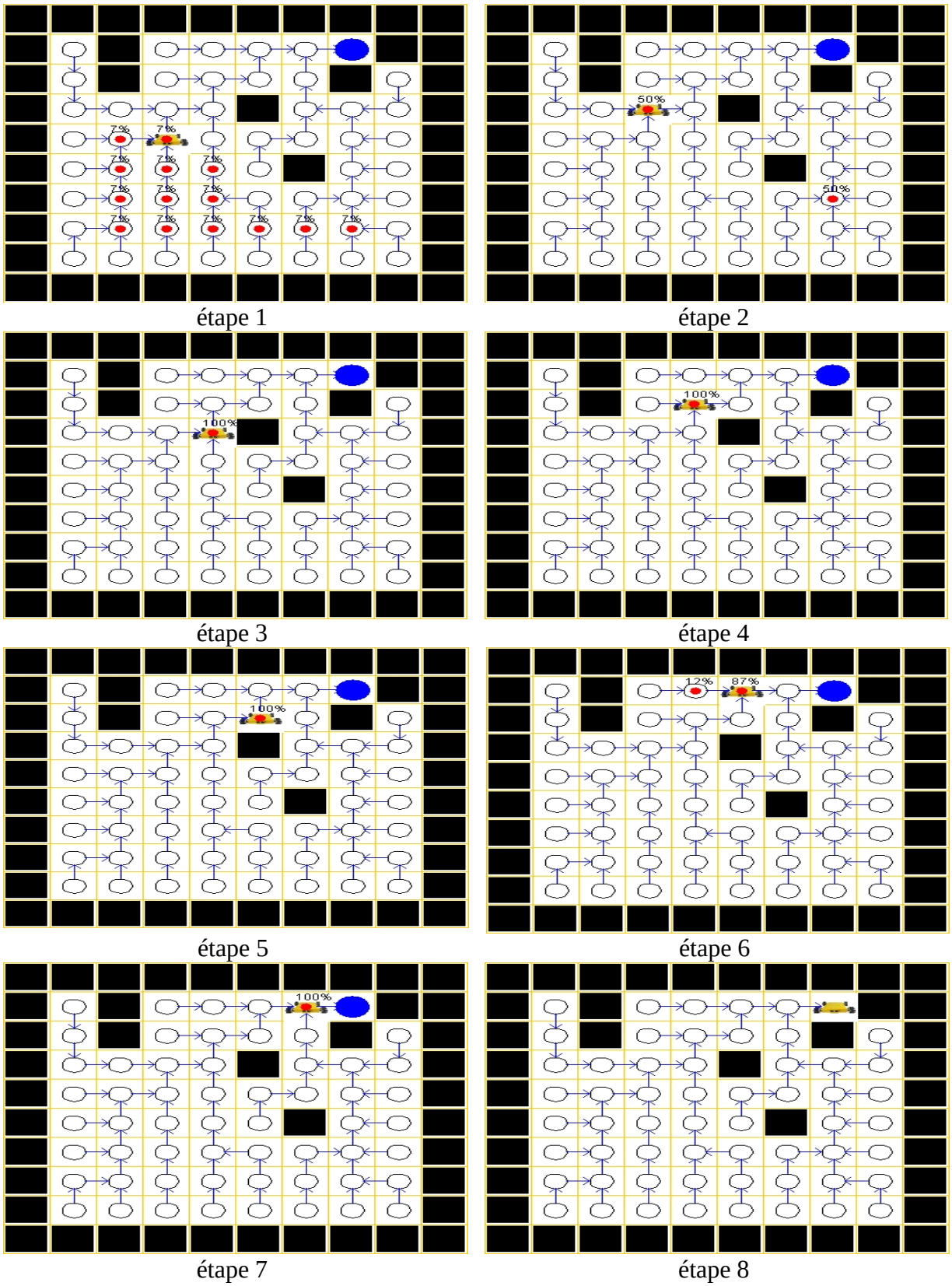


Figure 2.18 : Exemple des étapes de localisation par la distribution de probabilité d'état de croyance.

À la première étape, le robot observe son environnement (libre, libre, libre, libre, libre, libre, libre, libre), il y a donc 14 positions équiprobables (7%), le programme choisit une position aléatoire. Le robot se déplace vers le haut selon l'action optimale de l'état choisie. À l'étape 2, le robot observe (libre, libre, libre, libre, libre, libre, libre, occupe), il y a donc deux positions probables, le programme calcule la distribution de probabilité. Puisque ces deux positions sont équiprobables (50% chacune), le programme choisit un état au hasard et se déplace vers la droite selon l'action optimale de l'état choisie. À l'étape 3, le robot observe (libre, libre, occupé, libre, libre, libre, libre, libre), il y a donc deux positions probables, mais le calcul de la probabilité donne, à une position, une probabilité de 100%. Le robot est donc localisé et se déplace selon l'action optimale de l'état le plus probable et se dirige vers le haut. A l'étape 4, le robot observe (libre, libre, libre, occupé, libre, libre, libre, libre), une seule position probable (100%), le robot se dirige vers la droite. L'étape 5 ressemble à l'étape 4, le robot observe (libre, libre, libre, libre, occupé, libre, libre, libre), encore une seule position probable (100%), le robot se dirige vers le haut. À l'étape 6, le robot observe (occupé, occupé, libre, libre, libre, libre, libre, occupé), deux positions probables de probabilité respectivement 12% et 87%, le programme choisit la plus probable et se dirige vers la droite. Enfin, dans l'étape 7, le robot observe (occupé, occupé, but, occupé, libre, libre, libre, occupé), une seule position probable et le robot se déplace vers la droite pour atteindre le but.

b. Expérience réelle

Nous avons réalisé, au sein de notre laboratoire TIAD, un petit robot (**Figure 2.19**) afin de valider expérimentalement l'application du robot transporteur d'objets, ainsi que l'application du robot de position inconnue orienté vers un but. Le robot est équipé de quatre roues commandées chacune par un servomoteur.

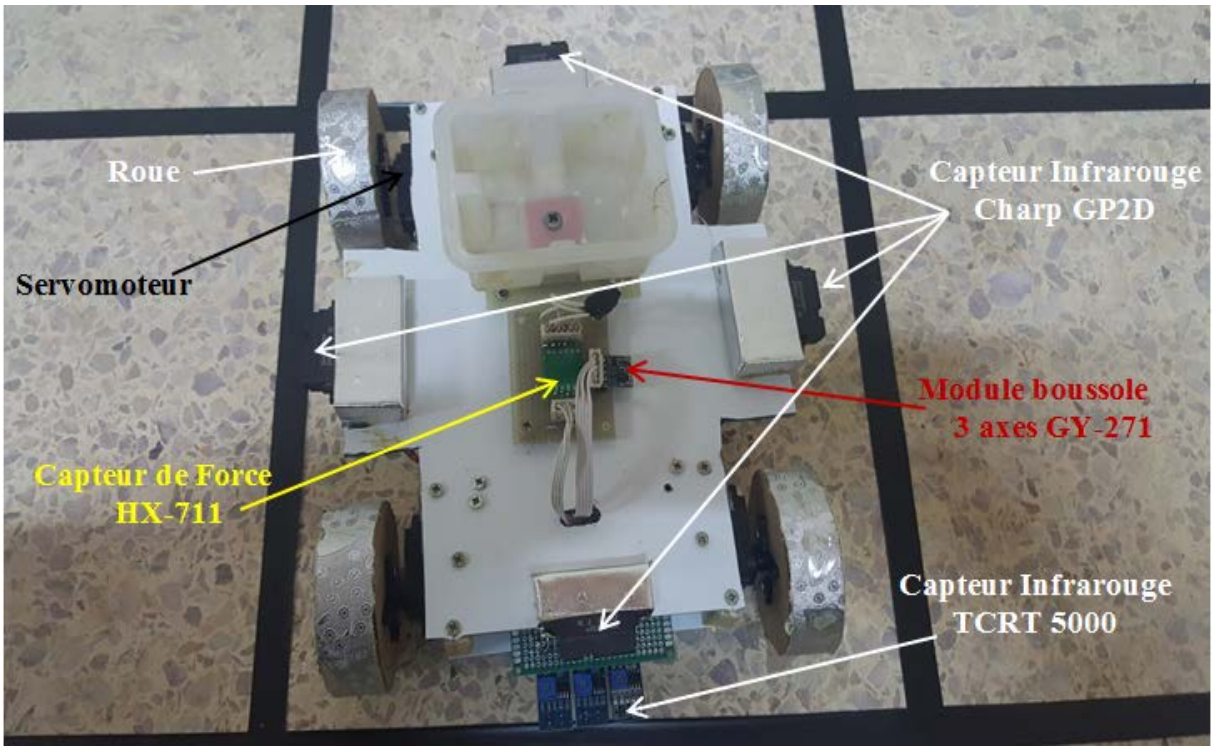


Figure 2.19 : Robot autonome transporteur d'objets.

Nous avons utilisé une carte de type Arduino (**Figure 2.20**) facilement programmable à l'aide d'un logiciel gratuit (Logiciel Arduino) comportant un ensemble de bibliothèques pour une gestion facile des différents matériels qu'on peut lui y connecter. Le robot comporte en outre, quatre capteurs infrarouges de type Sharp GP2D (**Figure 2.7**), trois capteurs infrarouges de type TCRT5000, un capteur de force de type HX711 (**Figure 2.21**) et un capteur d'orientation de type Module boussole trois axes GY-271 qui fournit l'orientation du robot avec une précision de 2° (**Figure 2.22**).

Figure 2.20 : Carte Arduino.

Le déplacement d'une case à une autre est ainsi assuré par le module boussole qui indique l'orientation du robot, les trois capteurs infrarouges et par commande des quatre servomoteurs en utilisant une bibliothèque « servo » disponible gratuitement.

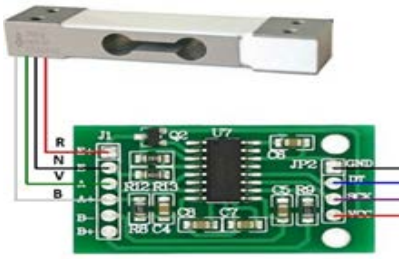


Figure 2.21: Capteur de force HX711.

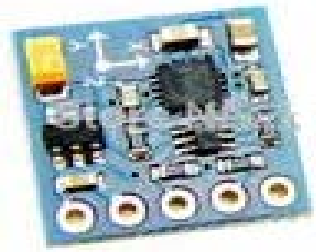


Figure 2.22 : Module boussole 3 axes GY-271.

Nous avons testé l'application du modèle proposé dans un environnement de 7×9 grilles carrées (**Figure 2.23**).

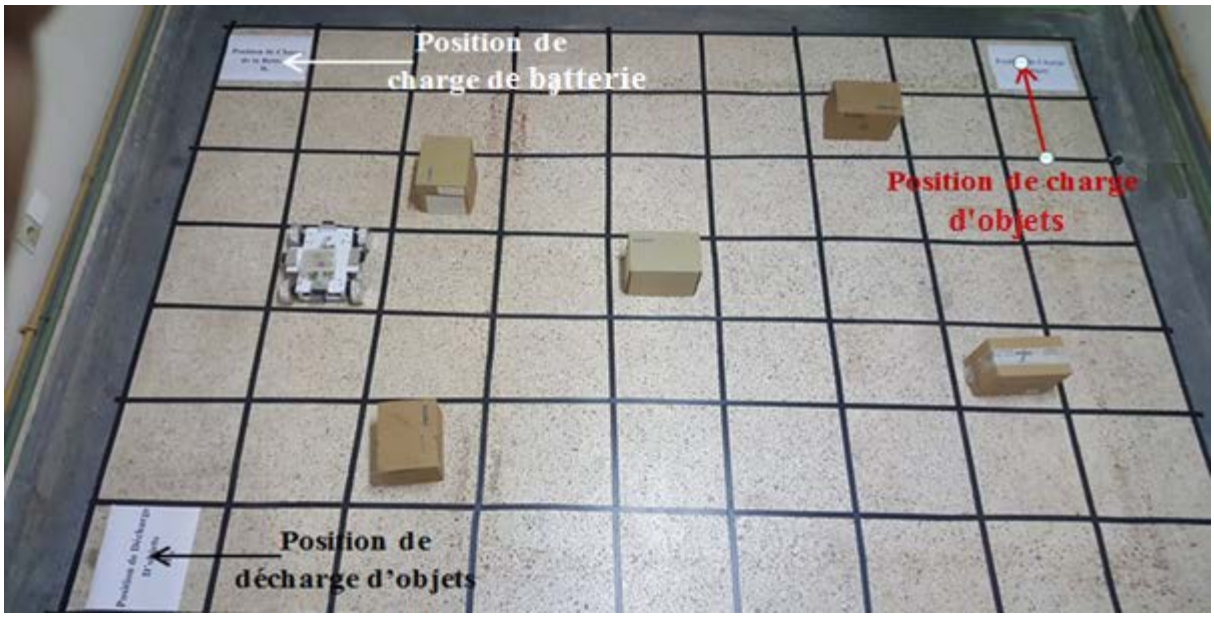


Figure 2.23 : Environnement expérimental de l'agent robot transporteur de marchandises.

Après avoir mémorisée une stratégie optimale dans le cas d'une position connue, nous avons appliqué l'**algorithme 2.1**. L'expérience s'est bien déroulée pour toute position de départ et même si on change, à un moment donné, la position du robot.

2.3.4. Modèle de décision markovien pour l'apprentissage à la marche Robotique

Les modèles de décision markoviens inconnus sont très utilisés, notamment en robotique. Apprendre au robot à marcher avec différents types de pattes est un exemple très connu. Les pattes sont généralement commandées par des moteurs pas à pas et des capteurs sont utilisés pour détecter le pas de déplacement et le maintien en équilibre. Nous présentons un exemple simple de modèle d'apprentissage utilisé pour apprendre à un robot (Escargot mécanique) de marcher avec une seule patte comportant deux moteurs pas à pas (ou deux servomoteurs) ainsi qu'un capteur ultrason lui permettant de mesurer la différence de distance entre une position à l'instant t et une autre à l'instant t+1 (**Figure 2.24**).

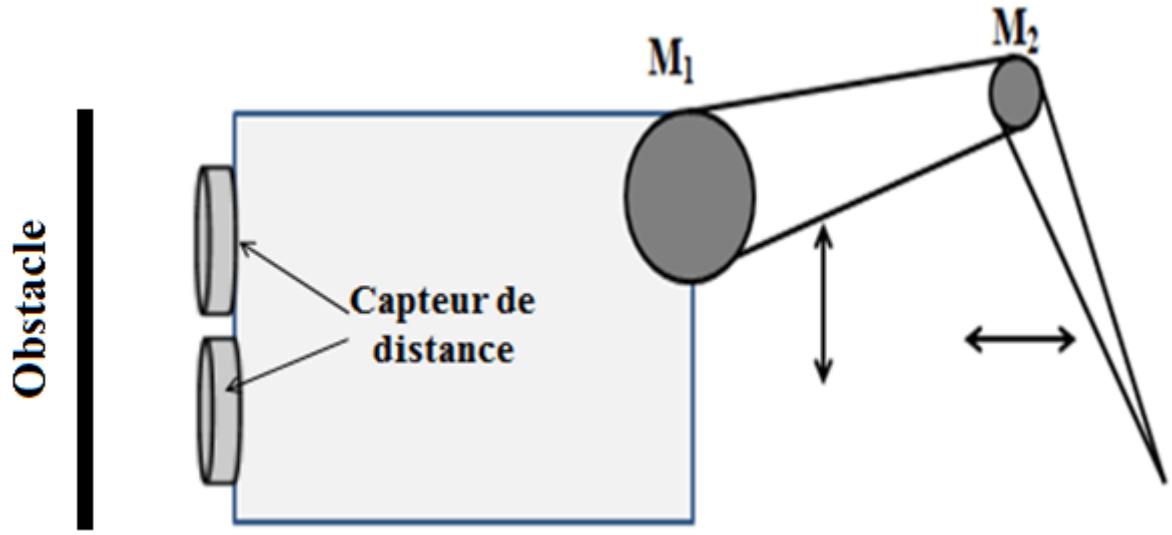


Figure 2.24 : L'escargot mécanique ayant un bras à deux moteurs et un capteur de distance.

Nous définissons les paramètres du modèle comme suit :

- **Espace d'états :** Chaque moteur peut être commandé de tel sorte qu'il tourne dans un sens ou dans l'autre avec un pas angulaire $d_\theta \in [-90^\circ, +90^\circ]$. La position possible de chaque moteur est ainsi un angle $\theta_i \in [-90^\circ, +90^\circ]$. Nous discrétisons cet intervalle avec un pas de $d_\theta = 30^\circ$. La position possible de chaque moteur M_i est ainsi $\theta_i \in E = \{-90^\circ, -60^\circ, -30^\circ, 0^\circ, +30^\circ, +60^\circ, +90^\circ\}$. Soit θ_1 (θ_2) la position du moteur M_1 (M_2) à l'instant t. Nous définissons l'état du système par le couple (θ_1, θ_2) et l'espace d'états est donc l'ensemble $S = E \times E$.

- **Espace d'actions :** Chaque servomoteur peut être actionné de tel sorte qu'il tourne de $+d_\theta$ ou $-d_\theta$ selon le sens voulu ou bien il reste sur place. On définit ainsi l'action du système dans un état donné par le couple (d_θ^1, d_θ^2), où d_θ^1 (d_θ^2) désigne l'action choisit pour le moteur M_1 (M_2). Chaque action $d_\theta^i \in \{-d_\theta, 0, +d_\theta\}$.

- **Fonction de récompense :** La récompense est le déplacement du robot après une action choisie, cette récompense est déterminée par le capteur ultrason qui mesure la différence de distance entre l'instant t et l'instant t+1. Cette récompense est inconnue a priori et peut dépendre du type de sol.

- **Fonction de transition :** Nous supposons que $P_{iaj} = 0\ ou\ 1, \forall i, j \in S, a \in A(i)$ (PDM déterministe, un seul successeur par couple état/action).

- **Espace du temps :** L'espace du temps est discrétisé en périodes de même longueur T suffisante pour laisser le temps aux moteurs de tourner du pas prédéfini égal à $d_\theta = 30^\circ$.

Ce PDM déterministe peut être représenté sous forme d'un graphe matriciel (**Figure 2.25**).

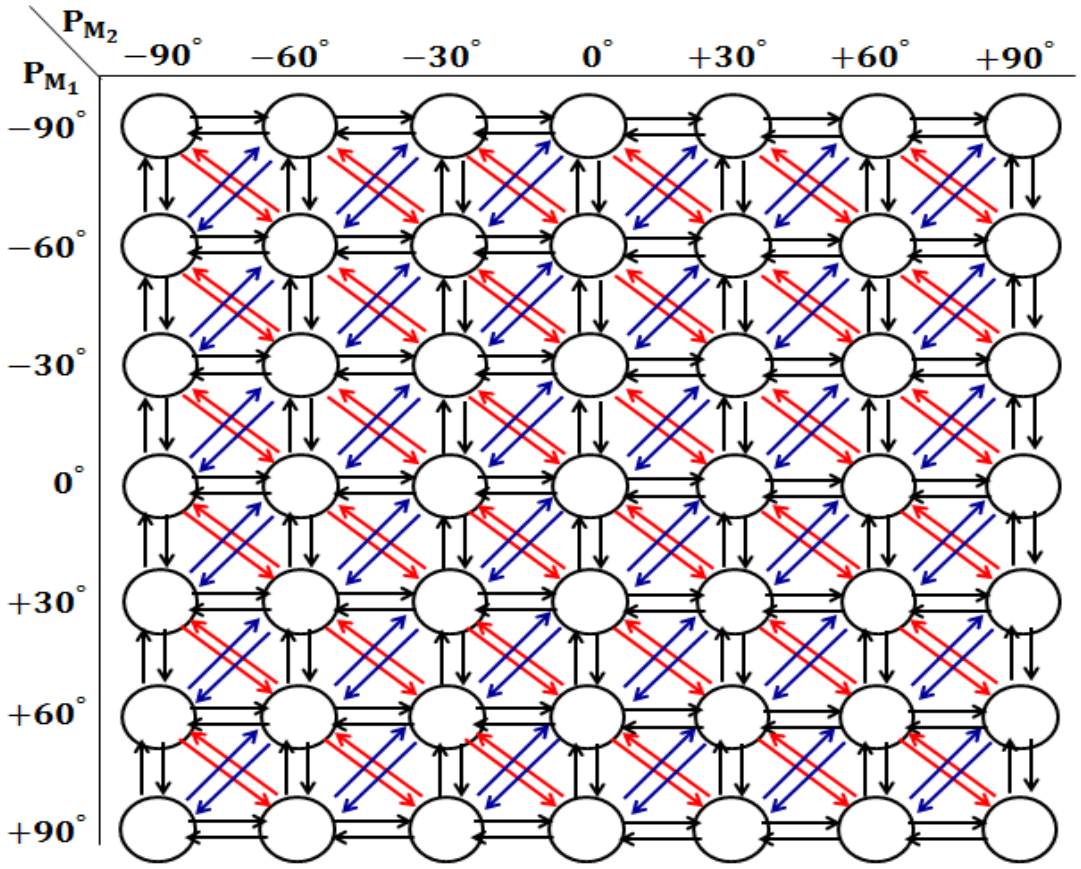


Figure 2.25 : Représentation des espaces d'états et d'actions de l'escargot mécanique sous forme de grilles.

D'ailleurs, l'objectif est de trouver la séquence d'actions qui permettent au robot de se déplacer efficacement en avant. Si la fonction de récompense est connue, un algorithme de résolution sous le critère de la moyenne ou le critère actualisé peut déterminer une stratégie optimale. Cependant, la fonction de récompense est a priori inconnue, nous procédons ainsi à un algorithme d'apprentissage par renforcement.

Le fait que, dans un exemple de marche robotique, un état terminal n'existe pas, la convergence de l'algorithme Q-learning ne peut pas être assurée. Pour cela, nous choisissons une méthode introduite par ToKic [TOK09] et qui est, en quelque sorte, une combinaison de l'algorithme Q-learning et l'algorithme IV (**Algorithme 2.2**). L'algorithme commence par une initialisation arbitraire du vecteur d'états V_0 (Ligne 1) et une initialisation à zéro des récompenses inconnues (Ligne 2). Une action ε-gloutonne est choisie et exécutée dans chaque itération (Ligne 6 et 7). La récompense observée est mise à jour (Lignes 8 et 9) et la fonction de valeur est mise à jour pour chaque état selon l'équation de Bellman (Ligne 10).

Algorithme 2.2: Algorithme de ToKic d'AR appliqué à l'escargot mécanique.

1: Initialiser V_0
2: Pour tout $s \in S, a \in A(s)$ Faire
    $r(s,a) \leftarrow 0$
3: $s \leftarrow s_0$ *//état initial*
4: **Tant que** (non fin d'apprentissage) **Faire**
5: $a \leftarrow \epsilon$-gloutonne(s) *//Choisir une action ϵ-gloutonne*
6: Exécuter l'action a *//Exécuter l'action choisie*
7: Observer l'état suivant s' et le gain $r(s,a)$
8: Mettre à jours le gain observé
9: Pour tout $s \in S$ faire

$$V(s) \leftarrow \max_{a \in A(s)} \left\{ r(s,a) + \alpha \sum_{s'} V(s') \right\}$$

$$\pi(s) \in \underset{a \in A(s)}{argmax} \left\{ r(s,a) + \alpha \sum_{s'} V(s') \right\}$$

10: $s \leftarrow s'$

11: Retourner V, π

Exemple de simulation : Pour tester la convergence de cet algorithme, nous avons généré aléatoirement des graphes sous forme de matrices. La **figure 2.26** montre un exemple avec les valeurs des récompenses (valeurs en bleu) des différentes actions (flèches) pour chaque état.

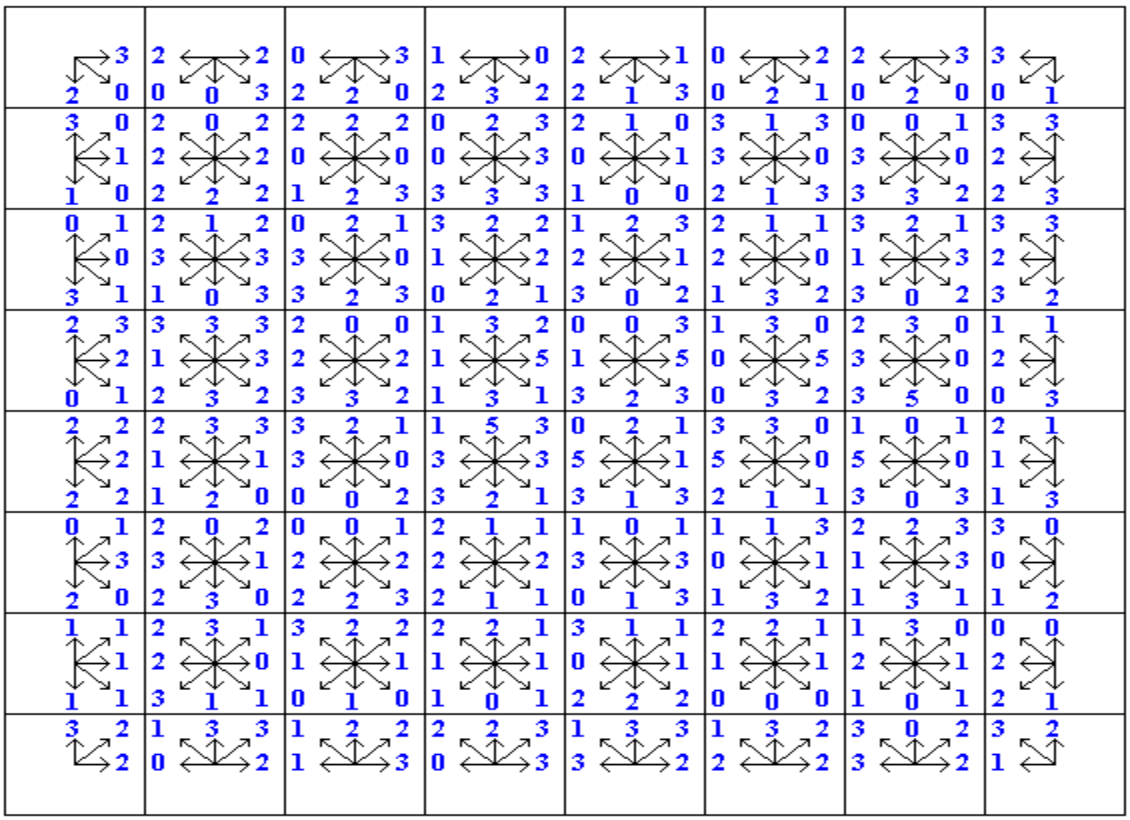


Figure 2.26 : Exemple d'espace d'actions et de fonction de récompenses pour le modèle de l'escargot mécanique.

La **figure 2.27** montre qu'après convergence de l'**algorithme 2.2**, toutes les actions optimales permettent d'atteindre une boucle (flèches en gras) qui représente la séquence optimale permettant un déplacement du robot.

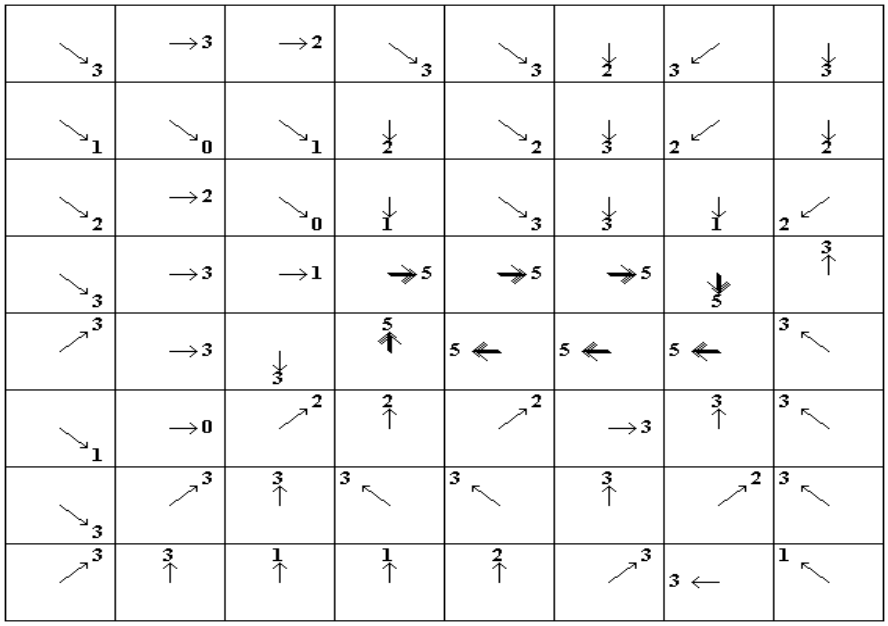


Figure 2.27 : Stratégie optimale obtenue pour l'exemple simulé de l'escargot mécanique.

Pour avoir une idée sur le temps de convergence de l'**algorithme 2.2**, nous avons calculé la différence $\Delta V = \sum_{i \in S} |V^*(i) - V_t(i)|$ entre les valeurs optimales et les valeurs calculés dans chaque itération de l'algorithme. Nous avons choisi une stratégie d'exploration ϵ-gloutonne avec des valeurs de ϵ variant de 1 à 0 par un pas de $\frac{1}{N}$ ou N est le nombre d'itérations. La **figure 2.28** montre les résultats obtenus.

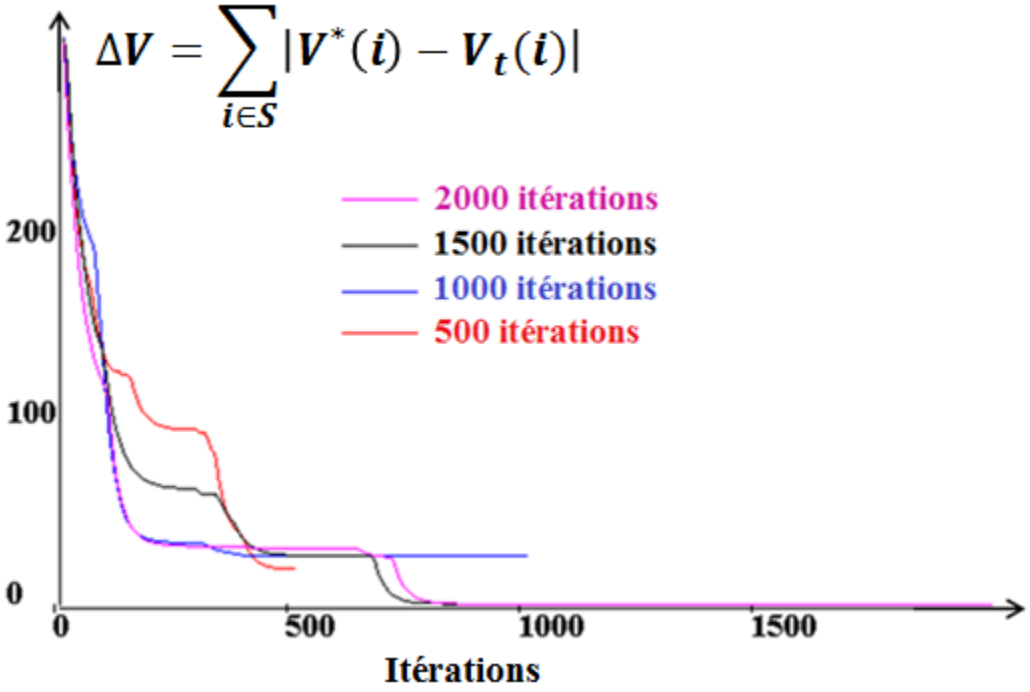


Figure 2.28 : Processus de convergence de l'algorithme d'AR appliqué à l'escargot mécanique.

Nous remarquons que la convergence nécessite plus que 1000 itérations, ce qui est très grand pour un petit modèle. Nous présenterons aussi, dans le chapitre 5, des techniques pour minimiser le nombre d'itérations et ainsi le temps de convergence. Ces techniques sont basées sur l'élimination en ligne des actions non-optimales et sur une technique d'exploration guidée.

2.4. Conclusion

Nous avons présenté, dans ce chapitre, l'agent robot qui est un système constitué de différents composants, sa conception nécessite un choix du système de perception et d'une plateforme qui dépendent généralement du milieu dans lequel il évolue et du champ d'application. Une représentation de l'environnement est aussi une étape nécessaire et qui est généralement mémorisé par le robot où reconstruit à chaque fois dans le cas d'un environnement dynamique par les techniques de cartographie. La localisation, qui est la base de la cartographie, est très importante, de ce fait un bon choix des capteurs rend la localisation plus probable.

Nous avons aussi présenté des exemples de modèle PDM appliqués à la robotique de navigation dans un environnement représenté sous forme de grilles et un modèle PDM d'apprentissage par renforcement appliqué à la marche robotique. En fait, les exemples sont très nombreux dans le domaine robotique du fait les PDM s'adaptent aux incertitudes liées aux capteurs de perceptions et aux actionneurs et du fait qu'ils offrent des méthodes d'AR permettant aux robots de s'adapter au changement de l'environnement et d'apprendre à manœuvrer dans différents situations. Nous présentons dans le chapitre 5 d'autres modèles PDM appliqués au robot suiveur de ligne et au robot auto-balancé.

Toutefois, les méthodes de résolution des PDM, que ce soit dans un modèle complètement connu, partiellement connu ou inconnu, risquent de ne pas être praticables pour des grands espaces d'états. Ce qui nécessite des techniques pour remédier à ce problème de la dimension, telles que les méthodes hiérarchiques, l'élimination d'actions non optimales, et l'exploration guidée que nous exposerons dans les chapitres suivants.

Chapitre 3

Méthodes Hiérarchiques de Résolution des PDM Actualisés

3.1. Introduction

Généralement, la plupart des problèmes réels, modélisés sous forme de PDM, ont des espaces d'état et d'action de très grande taille. En outre, la complexité temporelle des algorithmes classiques de résolution des PDM est polynomiale en fonction du nombre d'états [LIT95]. Ainsi, ces algorithmes sont impraticables pour des PDM de grandes dimensions. Dans la littérature, plusieurs travaux se sont focalisés sur l'optimisation de ces algorithmes, telle que la technique de décomposition [DAO10] introduite par Ross et Varadarajan [ROS91] pour des PDM avec contraintes, qui consiste à : (1) décomposer l'espace d'états en sous petits espaces ; (2) résoudre le problème restreint à chaque sous espace ; (3) combiner ces solutions locales pour obtenir une solution globale. Cette technique a été réutilisée par Abbad et Boustique [ABD03a], Daoui et Abbad [ABD03b, DAO07] et Dai et Goldsmith [DAI11] pour différents catégories de PDM et sous différents critères. Chafik et Daoui [CHA15] ont combiné cette technique avec le parallélisme afin d'obtenir plus d'optimisation. Cependant, l'algorithme de décomposition, dans ces travaux, est un algorithme polynomial en temps, ce qui rajoute un grand coût de décomposition.

Dans ce chapitre, nous considérons les PDM sous le critère d'actualisation, nous commençons par présenter un algorithme d'itération de la valeur accéléré, basé sur la liste des successeurs du couple état/action. Nous présentons également le principe de décomposition de l'espace d'états en Composantes Fortement Connexes (CFC) qui sont classifiées eux aussi en niveaux. Ensuite, un nouvel algorithme de décomposition basé sur l'algorithme de Tarjan [TAR72], sera proposé. Nous exposons aussi, une nouvelle définition des PDM restreints, qui nous a permis de proposer des variantes d'algorithmes de résolution hiérarchiques. Finalement, nous présentons un exemple de simulation en navigation robotique qui illustre l'apport de ces contributions.

3.2. Algorithme d'Itération de la Valeur Accéléré

L'algorithme IV (**Algorithme 1.2**) est l'un des algorithmes standards les plus utilisés pour trouver une stratégie optimale ou approximativement optimale d'un PDM α-pondéré (actualisé). En définissant la liste des successeurs du chaque couple état/action : $\Gamma_a^+(i) = \{j \in S : P_{iaj} \neq 0\}$ et la probabilité pondérée P^α: $P^\alpha(j|i,a) = \alpha P(j|i,a)$, nous proposons une version accélérée de l'algorithme IV (**Algorithme 3.1**) pour des espaces d'actions de petite taille. La complexité temporelle de l'algorithme proposé est réduite à $\mathcal{O}(|\Gamma_a^+||A||S|)$ par itération, où $|\Gamma_a^+|$ désigne le nombre moyen des successeurs par couple état/action. De plus, la probabilité pondérée permet de réduire un nombre important de multiplications dans chaque itération. En outre, si le nombre d'actions est très petit, sa complexité est linéaire par itération.

Algorithme 3.1: Algorithme IV accéléré.

IVA (**Entrée**: MDP: S, P, A, R ; Γ_a^+; α ; ε; **Sortie**: V^*; π^*)

1: t←0;

2: Pour tout i ∈ S Faire

$V^t(i) \leftarrow 0;$

3: **Répéter**

4: Pour tout i ∈ S Faire

$$V^{t+1}(i) \leftarrow \max_{a \in A(i)} \left\{ r(i,a) + \sum_{j \in \Gamma_a^+(i)} P^{\alpha}(j|i,a) V^t(j) \right\}$$

t← t +1;

Jusqu'à $(MAX|V^{t+1}(i) - V^t(i)| < \varepsilon)$

5: Pour tout i ∈ S Faire

$$\pi^*(i) \leftarrow \underset{a \in A(i)}{argmax} \left\{ r(i,a) + \sum_{j \in \Gamma_a^+(i)} P^{\alpha}(j|i,a) V^*(j) \right\}$$

6: Retourner V^*, π^*

Les résultats présentés dans la **figure 3.1** montrent le gain en temps d'exécution de l'algorithme IV accéléré comparé avec la version standard, pour l'exemple de navigation robotique, présenté au Chapitre 2.

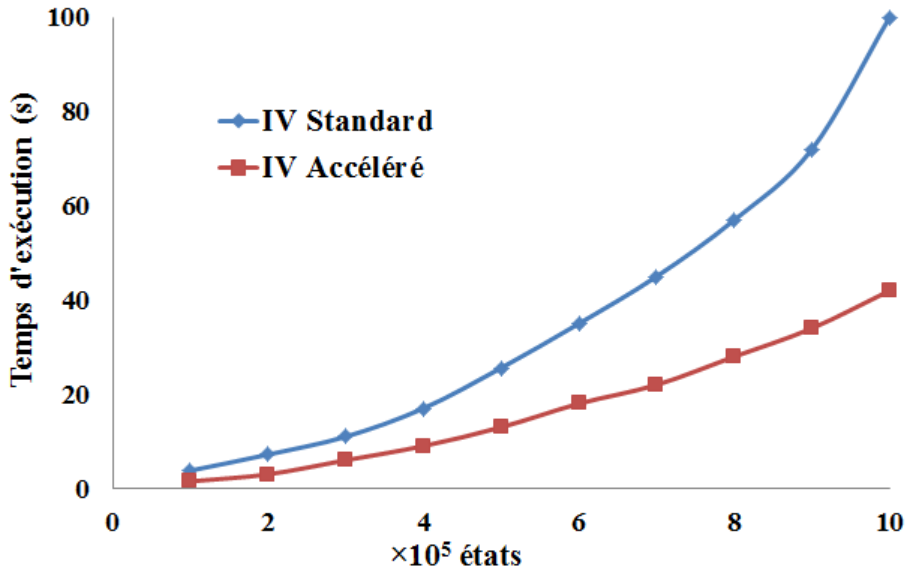


Figure 3.1: Comparaison entre l'algorithme IV et l'algorithme IV accéléré.

En outre, nous signalons que la même technique peut être utilisée dans les différentes variantes de l'algorithme IV, telle que la variante de gauss-Seidel (IVGS) ou la variante modifiée de l'algorithme IP présentée au chapitre 1. Nous nous limitons ici à présenter la variante IVGS accélérée (**Algorithme 3.2**).

Algorithme 3.2: Algorithme IVGS accéléré.

IVGSA (**Entrée**: MDP: *S, P, A, R ;* Γ_a^+; α ; ε; **Sortie**: V^*; π^*)

1 : Initialiser $V^* \in \mathcal{V}$

2 : Convergence ← Faux

3 : **Tant que** (Convergence = Faux) **Faire**

4 : Convergence ← Vrai

5 : Pour tout i ∈ S faire

$$V_{tmp} \leftarrow max_{a \in A}\left\{ r(i,a) + \sum_{j \in \Gamma_a^+(i)} P^{\alpha}(j|i,a)V^*(j) \right\}$$

Si $\left(\|V_{tmp} - V^*(s)\| \geq \varepsilon \right)$ Alors Convergence ← Faux

$V^*(s) = V_{tmp}$

6 : Pour tout $i \in S$ faire

$$\pi^*(i) \in argmax_{a \in A}\left\{ r(i,a) + \sum_{j \in \Gamma_a^+(i)} P^{\alpha}(j|i,a)V^*(j) \right\}$$

7 : Retourner V^*, π^*

.

3.3.Technique hiérarchique de résolution des PDM actualisés

En utilisant la liste des successeurs $\Gamma_a^+(s)$ de chaque couple (s, a), l'équation de Bellman (1.21) peut s'écrire sous la forme suivante :

$$\forall\, s \in S \qquad V(s) = \max_{a \in A(s)} \left\{ r(s,a) + \alpha \sum_{s' \in \Gamma_a^+(s)} P(s'|s,a)V(s') \right\} \qquad (3.1)$$

D'après cette équation, nous remarquons que la fonction de valeur V d'un état s, ne dépend que de ceux de l'ensemble des successeurs de chaque couple (s, a), $a \in A(s)$. Ainsi, à chaque classe communicante fermée de S, on peut associer un PDM restreint qui peut être résolu indépendamment des autres classes. Ces classes fermées seront considérées comme étant les classes du niveau 0. Les classes communicantes dont les états ne peuvent transiter que vers les états d'une classe du niveau 0, seront considérées comme étant des classes de niveau 1. Leurs correspondants PDM restreints sont résolus indépendamment des autres classes, et ainsi de suite pour les classes de niveaux 2, 3,…, p. C'est le principe général de la méthode de décomposition de l'espace d'états que nous allons considérer dans la suite.

3.3.1. Décomposition en niveaux

Soit $G = (S, U)$ le graphe associé au PDM original, où l'espace d'états S représente les sommets et $U = \{ (i,j) \in S^2 : \exists\, a \in A(i), P_{iaj} > 0\}$ représente l'ensemble des arcs. Il existe une partition unique $T = \{C_1, C_2, \dots, C_P\}$ de l'espace d'états en Composantes Fortement Connexes (CFC), qui correspondent aux Classes Communicantes dans la théorie des PDM [ROS91, DAO10]. Ces classes peuvent être classifiées en niveaux, le niveau L_0 est formé par l'ensemble des classes fermées C_i ($\forall\, i \in C_i, a \in A(i): \forall\, j \notin C_i, P_{iaj} = 0$). Le niveau L_p est

formé par l'ensemble des classes C_i tel que la fin de tout arc émanant de C_i est dans un niveau $L_p,\ L_{p-1}, \dots, L_0$ (voir l'exemple de la **figure 3.2**).

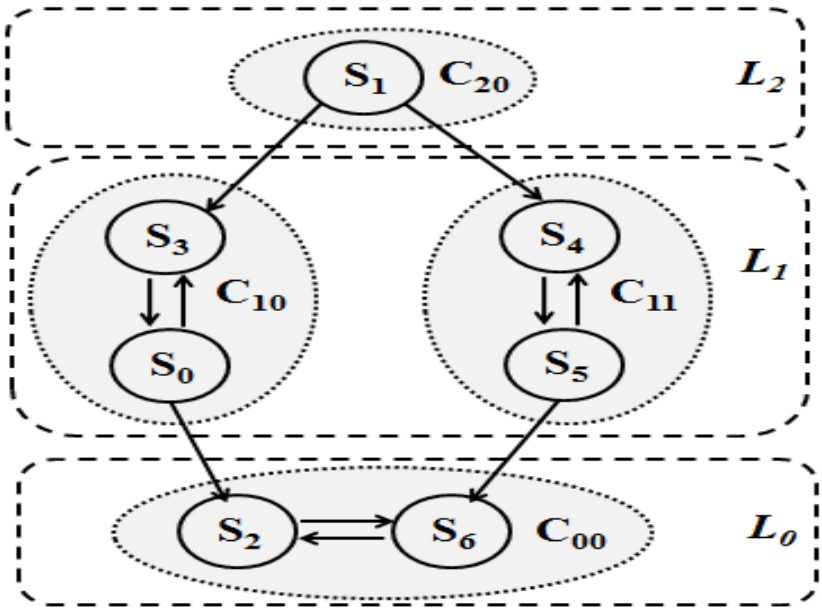


Figure 3.2 : Exemple d'un PDM décomposé en CFC ($\mathbf{C_{ij}}$) et classifié en trois niveaux.

3.3.2. PDM restreints

Pour les PDM sous le critère α-pondéré, Abbad et Daoui [ABD03b] définissent et construisent, par induction, les PDM restreints (PDM_{pk}) correspondant à chaque CFC (C_{pk}) du niveau L_p, $p = 0, \dots, L$; $k = 0, \dots, K(p)$ comme suit:

Espace d'états:
$S_{pk} = C_{pk} \cup \bar{C}_{pk}$,
où $\bar{C}_{pk} = \{j \in S \backslash C_{pk} : \exists\ i \in\ C_{pk}, \exists\ a \in A(i), P_{iaj} > 0\}$

Espace d'actions:
$$\begin{cases} A_{pk}(i) = A(i)\ si\ i \in C_{pk} \\ A_{pk}(i) = \theta\ si\ \ i \in \bar{C}_{pk} \end{cases}$$
où θ est une action imaginaire tel que $P_{i\theta i} = 1\ si\ \ i \in \bar{C}_{pk}$

Fonction de transition:
$$\begin{cases} P_{pk}(j|i,a) = P_{iaj}\ \forall\ i \in C_{pk}\ , a \in A(i) \\ P_{pk}(j|i,\theta) = 1\ \forall\ i = j \in \bar{C}_{pk} \end{cases}$$

Fonction de récompense:
$$\begin{cases} R_{pk}(i,a) = r(i,a)\ \forall\ i \in C_{pk}\ , \forall\ a \in A(i) \\ R_{pk}(i,a) = (1-\alpha)V^*(i)\ \ \forall\ i \in \bar{C}_{pk} \end{cases}$$

Où $V^*(i)$ est la valeur optimale de i calculée dans les niveaux précédents (classes $\bar{C}_{pk}$). Notons que, pour le niveau 0, on a : $\bar{C}_{0k} = \emptyset$.

Ensuite, Abbad et Daoui [ABD03b] proposent l'algorithme hiérarchique (**Algorithme 3.3**) et montrent qu'il est correct et permet de trouver une stratégie optimale, dans un nombre fini d'itérations.

Algorithme 3.3: Algorithme hiérarchique proposé par Abbad et Daoui.

```
1 : Déterminer les CFC Ci
2 : Déterminer les niveaux Lp, p=0,…,L et leurs classes C_pk, k=1,…,K(p)
3 : Pour chaque niveau Lp, p=0,…,L Faire
4 :     Pour chaque classe C_pk, k=1,…,K(p) Faire
5 :         Construire le PDM restreint MDP_pk
6 :         Résoudre le PDM restreint MDP_pk
```

L'inconvénient de cet algorithme, est qu'il nécessite une décomposition en CFC (ligne 1) et une décomposition en niveaux (ligne 2) dont la complexité temporelle est polynomiale, ce qui rend la phase de décomposition très coûteuse pour un espace d'états de grande taille. Nous proposons, par la suite, un nouvel algorithme basé sur celui de Tarjan [TAR72] et qui permet de trouver simultanément les CFC et leurs niveaux.

Remarque 3.1 : Nous indiquons qu'il est possible de ne pas se déterminer les niveaux, du fait qu'on peut utiliser certains algorithmes de la recherche en profondeur dans un graphe qui permettent d'assurer l'ordre de résolution des PDM restreints dans la méthode hiérarchique.

3.3.3. Nouvelle variante de l'algorithme de Tarjan

Comme la méthode hiérarchique est basée sur la décomposition de l'espace d'états en CFC, il est ainsi nécessaire de choisir un algorithme efficace permettant de faire cette décomposition. En théorie des graphes, les algorithmes de décomposition les plus connus sont : Tarjan [TAR72], Kosaraju-Sharir [SHA81], Dijkstra [DIJ82] et Lowe [LOW14].

L'algorithme de Tarjan [TAR72] est basé sur la recherche en profondeur dans le graphe. Il utilise une pile où sera stocké chaque sommet S_0visité lors du parcourt et qui sera associé avec deux paramètres, l'ordre où le numéro de visite $S_0.idx$ et le numéro minimum $S_0.low$ du sommet déjà visité vers le quel il y a un arc de retour. Une CFC est détectée lorsque $(S_0.low = S_0.idx)$, ainsi l'algorithme dépile les sommets jusqu'au sommet S_0. Les sommets dépilés forment une CFC. L'algorithme de Tarjan est exécuté en 0(n+m) opérations arithmétiques, où n désigne le nombre de sommets et m désigne le nombre d'arcs.

Kosaraju-Sharir [SHA81] propose un algorithme basé aussi sur la recherche en profondeur dans le graphe, mais il utilise deux parcours en profondeur. Un premier utilisant la liste des successeurs, et un deuxième utilisant la liste des prédécesseurs, ce qui est couteux en comparaison avec l'algorithme de Tarjan[TAR72] qui n'utilise qu'un seul parcourt.

Dijkstra [DIJ82] propose une variante de l'algorithme de Tarjan, dans laquelle, il maintient dans la pile un sommet S_0 candidat d'une CFC, au lieu d'utiliser la variable tableau $S_0.low$. Les CFC sont ainsi détectées par un deuxième parcourt en profondeur commençant à partir de S_0. Couvreur [COU99] propose une variante de l'algorithme de Dijkstra dans le but de trouver les cycles dans le graphe et qui peut être translaté à la détection des CFC. Lowe [LOW14] propose une version itérative de l'algorithme de Tarjan pour un calcul multithread appelé l'algorithme concurrent.

Comme première contribution, nous proposons, dans ce paragraphe, une variante de l'algorithme de Tarjan (**Algorithme 3.4**) en utilisant une table d'adresses des sommets (Addrs()) stockée dans une liste doublement chainée (LDC), cette liste est initialisée arbitrairement par un sommet et elle est construite simultanément avec le graphe associé au

PDM, représenté par Γ^+ : la liste des successeurs. Chaque sommet S_i dans la liste est directement accessible par son adresse Addrs(S_i).

Dans l'étape d'initialisation, la valeur de l'ordre de visite de chaque sommet est mise à zéro (Ligne 3) afin d'indiquer qu'il n'est pas encore visité. Quand une CFC est détectée, cette valeur est remise à 1 pour indiquer qu'il est déjà visité et qu'il est dans une CFC, ce qui évite d'ajouter un tableau logique comme dans le cas de l'algorithme classique de Tarjan. La valeur du paramètre index est ainsi initialisée par 2 (Ligne 5) pour éviter la valeur 1.

Algorithme 3.4: Variante de l'algorithme de Tarjan

```
1: VTarjan (Entrée: LDC, Γ+, Sortie: Liste_CFC)
2:     Pour chaque sommet s_i ∈ S Faire
3:         s_i.idx← 0; //Sommet non visité
4:     Tant que la liste LDC est non vide Faire
5:         index←2 ;
6:         s_0←LDC.fin; //choisir le dernier élément de la liste
7:         VTarjan_DFS (s_0); //Appel de la fonction récursif pour la recherche en profondeur

8: VTarjan_DFS (s_0)
9:     s_0.low← index;
10:    s_0.idx← index;
11:    index ← index +1;
12:    Pour tout s_i ∈ Γ+(s_0) Faire
13:        Si (s_i n'est pas visité) Alors // Si (s_i.idx = 0)
14:            Déplace_sommet(s_i) //Procédure 3.1
15:            VTarjan_DFS (s_i); //Appel récursif
16:            s_0.low ← Min (s_0.low , s_i.low)
17:        Sinon Si (s_i n'est pas dans une CFC ) Alors // Si (s_i.idx > 1)
18:            s_0.low ← Min (s_0.low , s_i.idx)
19:    SI (s_0.low =s_0.idx) Alors // Si détection d'une CFC
20:        CFC =Deplace_classe( Adrs(s_0) ); //Procédure 3.2
21:        pour tout sommet s_k dans la classe trouvée Faire
22:            s_k.idx ←1; //sommet dans une CFC
```

La recherche des CFC commence à partir du dernier élément de la LDC (Ligne 6), chaque sommet visité est déplacé à la fin de la liste (Ligne 14, procédure 3.1).

Procédure 3.1: Déplacement d'un sommet à la fin de la liste doublement chainée.

```
1: Déplace_sommet(s_0)
2: Addrs[s_0].prcedent.suivant←Addrs[s_0].suivant;
3: Addrs[s_0].suivant.prcedent ←Addrs[s_0].prcedent;
4: Addrs[s_0].suivant←LDC.fin
5: LDC.fin←Addrs[s_0]
```

Quand une CFC est détectée avec la racine s_0, la sous-liste identifiée par l'adresse Addrs(S0) est déplacée vers un tableau de liste (Ligne 20, procédure 3.2). Ce tableau est en fait une liste simplement chainée, où chaque élément représente une CFC.

Procédure 3.2: Déplacement de la CFC détectée vers la liste des CFCs

1: Deplace_classe(Adrs(s_0))
2: CFC.debut←Addrs[s_0];
3: CFC.fin← LDC.fin
4: LDC.fin←Addrs[s_0].precedent
5: Liste_des_CFCs.fin.suivant← CFC
6: Liste_des_CFCs.fin. ← CFC

La **figure 3.3** illustre les étapes de détection des CFC de l'exemple de la **figure 3.2**.

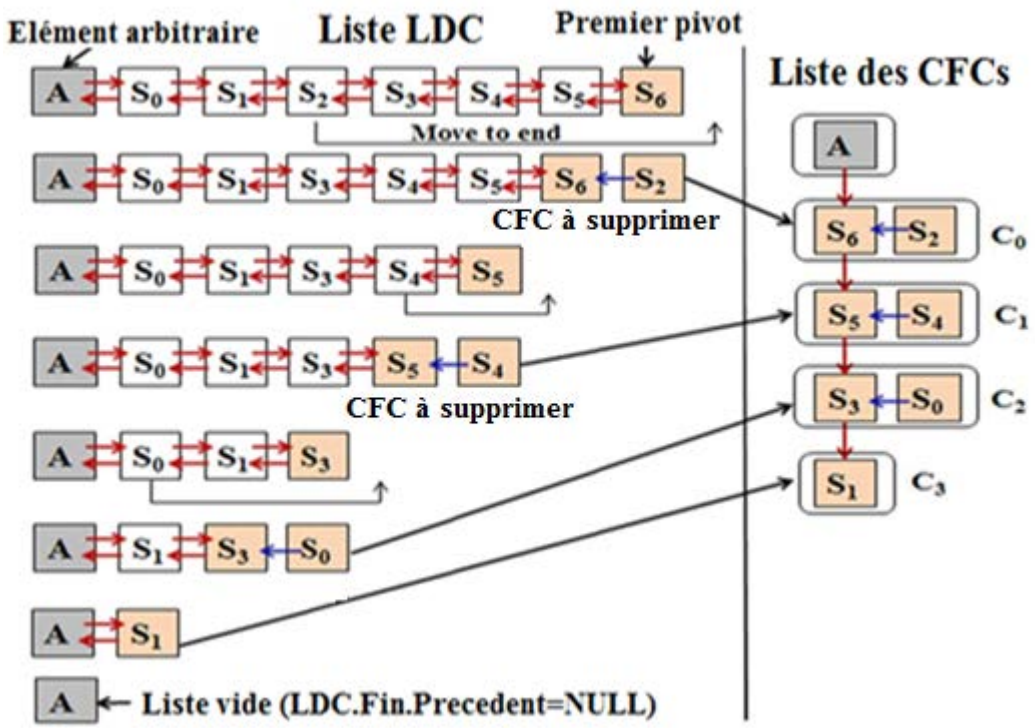


Figure 3.3 : Etapes de détection des CFC (exemple de la figure 3.2) à partir de la liste doublement chainée.

Remarque 3.2 : Notons que le premier élément ajouté dans la liste permet d'éviter le test d'existence d'un sommet au début de la liste, ce qui réduit le nombre de tests lors du déplacement d'un sommet vers la fin. De plus, dans l'algorithme de Tarjan, l'étape de la ligne 4 nécessite $\mathcal{O}(n)$ opérations arithmétiques, alors que dans cette variante, il ne nécessite que $\mathcal{O}(k)$ opérations arithmétiques, où k est généralement très petit que n et dépend du nombre des CFC. La complexité temporelle de l'algorithme reste similaire à celle de l'algorithme de Tarjan, mais le temps d'exécution sera réduit si $k \ll n$.

La **figure 3.4** montre la comparaison de cette variante de Tarjan avec quelques algorithmes classiques pour des modèles générés aléatoirement. Nous remarquons un gain du temps d'exécution avec la variante proposée.

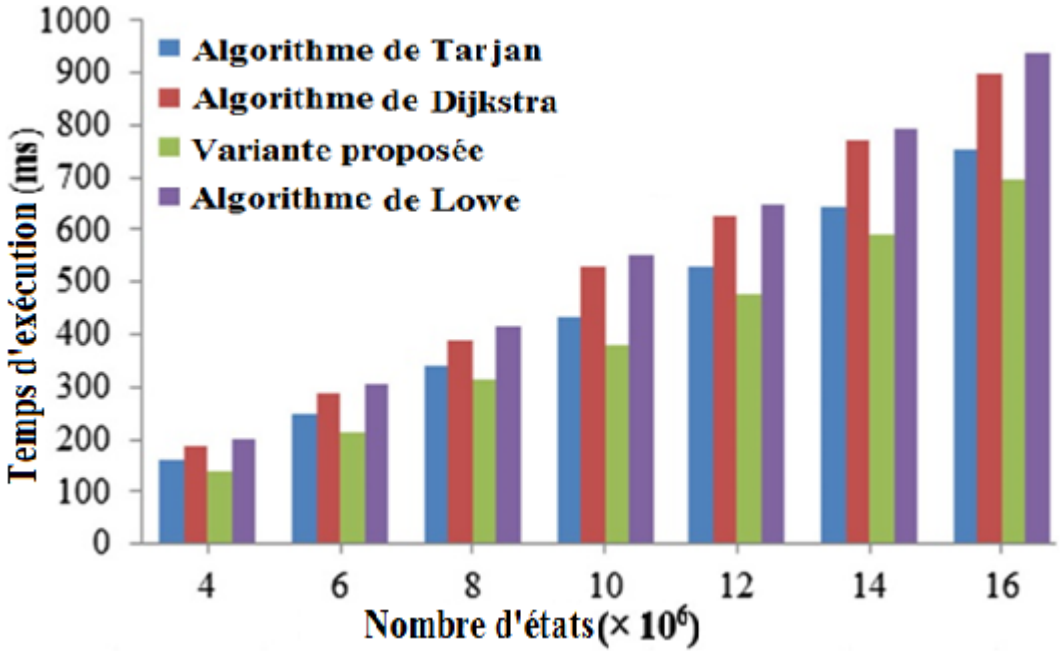


Figure 3.4 : Comparaison du temps d'exécution de la variante proposée avec les algorithmes de Tarjan, de Dijkstra et de Lowe.

3.3.4. Nouvel algorithme de recherche simultanée des CFC et leurs niveaux.

Après la décomposition de l'espace d'états en CFC, Abbad et Boustique [ABD03a] utilisent l'algorithme suivant (**Algorithme 3.5**) pour classifier ces CFC en niveaux.

Algorithme 3.5: Classification des CFC en niveaux

1: **Classifier_CFCs_en_niveaux** (**Entrée** : CFC, **Sortie** : niveaux L_i)
2: $\Omega \leftarrow S$;
3: $n \leftarrow 0$;
4: $L_n \leftarrow \{C_i : C_i$ est fermée dans $\Omega\}$;
5: $\Omega \leftarrow \Omega - L_n$;
6: **Tant que** ($\Omega \neq \emptyset$) **Faire**
7: $n \leftarrow n + 1$;
8: $L_n \leftarrow \{ C_i : C_i$ est fermée dans $\Omega\}$;
9: $\Omega \leftarrow \Omega - L_n$;

Cet algorithme nécessite $0(n^2)$ opérations arithmétiques, où n est le nombre d'états. Pour réduire cette complexité polynomiale, nous proposons un nouvel algorithme (**Algorithme 3.6**) basé sur l'algorithme de Tarjan et qui permet de trouver simultanément, les CFC et leurs nivaux en un temps d'exécution linéaire.

Avant d'énoncer cet algorithme, nous présenterons tout d'abord les résultats suivants.

Définition 3.1 : On définit la fonction niveau $\mathbb{L}$, qui pour toute classe $\boldsymbol{C}$ ou tout état $\mathbf{s}$ associe son niveau. Si $\mathbb{L}(\boldsymbol{C})$ désigne le niveau de la classe $\mathbf{C}$ alors $\forall \boldsymbol{s} \in \boldsymbol{C}$, $\mathbb{L}(\mathbf{s}) = \mathbb{L}(\mathbf{C})$.

Lemme 3.1: Soit $(x, y) \in S^2$. Si $y \in \Gamma^+(x)$ alors $\mathbb{L}(x) \geq \mathbb{L}(y)$, où $\Gamma^+(x) = \{y \in S: \exists a \in A(x), P_{xay} > 0\}$.
Preuve : Supposons que $\mathbb{L}(x) < \mathbb{L}(y)$, d'après la définition des niveaux, le fait que $\mathbb{L}(x) < \mathbb{L}(y)$ implique que $y \notin \Gamma^+(x)$, d'où la contradiction.

Lemme 3.2: Soient C une CFC et $(x, y) \in S^2$ tel que $x \in C$. Si $y \in \Gamma^+(x)$ et $y \notin C$ alors $\mathbb{L}(x) > \mathbb{L}(y)$.
Preuve : La propriété 3.1 implique que $\mathbb{L}(x) \geq \mathbb{L}(y)$. Supposons maintenant que $\mathbb{L}(x) = \mathbb{L}(y)$, ce qui implique que $y \in C$, d'où la contradiction.

Lemme 3.3: Soient C une CFC transitoire (i.e. non fermée) et $\{C_1, C_2, .. C_p\}$ l'ensemble des classes successeurs de C. Alors $\mathbb{L}(C) = \left(\max_{i=1,..,p} \mathbb{L}(C_i)\right) + 1$.
Preuve : $\{C_1, C_2, .. C_p\}$ l'ensemble des classes successeurs de C implique que $\exists\, y \in C_i, \exists\, x \in C : y \in \Gamma^+(x)$. D'après la lemme 3.2, on a : $\mathbb{L}(C) > (\max_i \mathbb{L}(C_i))$ et d'après la définition des niveaux, $\mathbb{L}(C) = \left(\max_{i=1,..,p} \mathbb{L}(C_i)\right) + 1$.

Algorithme 3.6: Nouvel algorithme de recherche simultanée des CFC et leurs niveaux

```
1: VTarjan_niveaux (Entrée: LDC, Γ+, Sortie: Liste_CFC)
2:   Pour chaque sommet s_i ∈ S Faire
3:     s_i.idx← 0; //Sommet non encore visité
4:     s_i.niveau← -1; //Initialisation du niveau et aussi Sommet non encore dans une CFC
5:   Tant que la liste LDC est non vide Faire
6:     index←1 ;
7:     s_0←LDC.Fin; //choisir le dernier élément de la liste
8:     VTarjan_DFS2 (s_0); //Appel de la fonction récursif pour la recherche en profondeur
9: VTarjan_DFS 2(s_0)
10:  s_0.low← index;
11:  s_0.idx← index;
12:  index ← index +1;
13:  N_0 ← 0; //Initialisation du niveau
14:  Pour tout s_i ∈ Γ+(s_0) Faire
15:    Si (s_i n'est pas visité) Alors // Si (s_i.idx = 0)
16:      Déplace_sommet(s_i) //Procédure 3.1
17:      N_i ←VTarjan_DFS2 (s_i); //Appel récursif
18:      s_0.low ← Min (s_0.low , s_i.low)
19:      N_0 ← Max(N_0, N_i); // lemme 3.1
20:    Sinon Si (s_i n'est pas une CFC ) Alors // Si (s_i.niveau = −1)
21:      s_0.low ← Min (s_0.low , s_i.idx)
22:    Sinon //Successeur externe
23:      N_0 ← Max(N_0, s_i.niveau + 1); // lemme 3.2
24:  SI (s_0.low =s_0.idx) Alors // Si détection d'une CFC
25:    CFC =Deplace_classe( Adrs(s_0) ); //Procédure 3.2
26:    pour tout sommet s_k dans la classe trouvee Faire
27:      s_k.niveau ← N_0; // lemmes 3.1, 3.2 et 3.3
28:    Retourner N_0 + 1; // lemme 3.2
29:  Retourner N_0; // lemme 3.1
```

Théorème 3.1: L'**algorithme 3.6** est correct et s'exécute en $\mathcal{O}$ (n + m) opérations arithmétiques.

Preuve : La preuve provient des lemmes 3.1, 3.2 et 3.3. En effet, les instructions des lignes 19 et 23 transmettent le niveau de la CFC à son racine. Après chaque appel récursif, la fonction VTarjan_DFS2() retourne le minimum niveau possible au prédécesseur appelant. Dans ce cas, le niveau est mis à jour en utilisant le lemme 3.1 (ligne 19). Si un successeur externe est détecté, le niveau est mis à jour en utilisant le lemme 3.2 (ligne 23). Quand une CFC est détectée, son niveau est déterminé par le niveau transmis à son racine (lemme 3.3), dans ce cas la fonction récursive doit retourner ce niveau incrémenté par 1 (lemme 3.2).

L'**algorithme 3.6** a la même structure que celui de Tarjan, il est donc de même complexité linéaire $\mathcal{O}$(n + m).

Remarque 3.3: Ces propriétés peuvent être facilement appliquées à l'algorithme standard de Tarjan ou à certaines de ses variantes [DIJ82, COU99, LOW14] pour détecter simultanément les CFC et leurs niveaux.

La **figure 3.5** montre la comparaison entre la variante proposée (**algorithme 3.6**) et l'algorithme de décomposition classique (**algorithme 3.5**) avec des modèles générés aléatoirement. Nous remarquons que l'algorithme proposé est linéaire et il est plus rapide que l'algorithme classique.

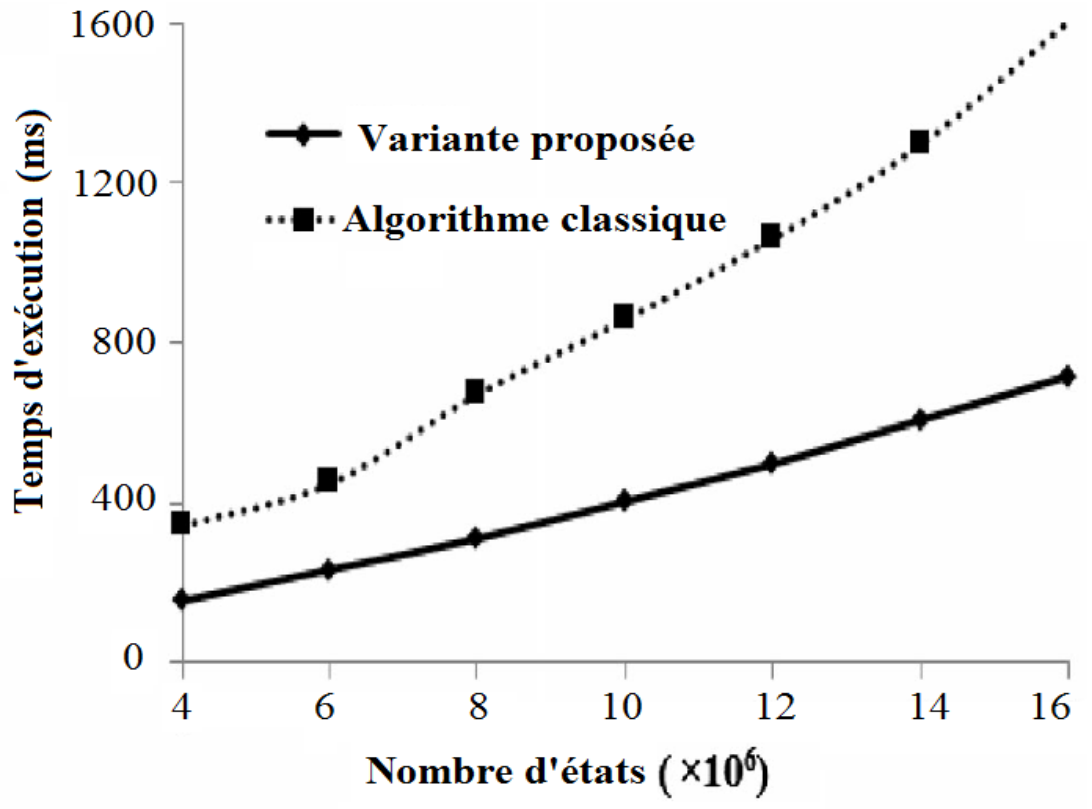


Figure 3.5 : Comparaison entre les deux algorithmes de décomposition en CFC : 3.4 et 3.5.

3.3.5. Algorithmes hiérarchiques

En se basant sur l'**algorithme 3.6**, nous présentons, ci-dessous, une première version de l'algorithme IV hiérarchique séquentiel (**Algorithme 3.7**) comme suit.

Algorithme 3.7: Algorithme IVGS hiérarchique séquentiel (version 1)

1: IVGSHS_V1 (**Entrée** : PDM ; **Sortie** : V^*, π^*)
2: Déterminer les CFC et leurs niveaux en utilisant l'**algorithme 3.6**;
3: Pour chaque niveau L_p, $p=0,\ldots,L$ Faire *// L : nombre de niveaux détectés*
4: Pour chaque CFC $C_{pk}, k = 0,\ldots,K(p)$ dans le niveau L_p Faire *//$K(p)$: nombre de classes*
5: Construire le PDM restreint PDM_{pk};
6: (V^*_{pk}, π^*_{pk}) ←IVGSA (PDM_{pk}); *//Résoudre le PDM restreint PDM_{pk}*
7: $\pi^* \leftarrow \bigcup_{p,k} \pi^*_{pk}$ et $V^* \leftarrow \bigcup_{p,k} V^*_{pk}$

Nous remarquons que les PDM restreints PDM_{pk}, k=0,..K(p) dans un même niveau L_p sont indépendants et peuvent être résolus en parallèle, d'où la version de l'algorithme hiérarchique parallèle suivant (**Algorithme 3.8**).

Algorithme 3.8: Algorithme IVGS hiérarchique parallèle (version 2)

IVGSHP _V2 (Entrée : PDM ; Sortie : V^*, π^*)
1. Déterminer les CFC et leurs niveaux en utilisant l'**algorithme 3.6**;
2. **Pour** chaque niveau L_p, p=0,…,L **Faire en parallèle** *// L : nombre de niveaux détectés*
3. Construire les PDM restreints : PDM_{pk}**,** $k = 0,\ldots,K(p)$; *//$K(p)$: nombre de classes*
4. (V^*_{pk}, π^*_{pk}) ←**IVGSA (**PDM_{pk}**)**; //Résoudre les PDM restreints PDM_{pk}**,**k=1,…N_p ;
5. $\pi^* \leftarrow \bigcup_{p,k} \pi^*_{pk}$ et $V^* \leftarrow \bigcup_{p,k} V^*_{pk}$

D'ailleurs, la technique de recherche en profondeur, utilisée par l'algorithme de Tarjan, assure l'ordre de dépendance des CFC dans la résolution hiérarchique du PDM. Dans un calcul séquentiel (non parallèle), il est inutile d'utiliser les niveaux. En effet, il suffit de résoudre les PDM restreints correspondant aux CFC par ordre de leur détection. Ainsi, nous proposons une autre version (**Algorithme 3.9**) de l'algorithme IV hiérarchique séquentiel.

Algorithme 3.9: Algorithme IVGS hiérarchique séquentiel (version 3)

IVGSHS_V3 (Entrée : PDM ; Sortie : V^*, π^*)
1. Déterminer les CFC C_i (**Algorithme 3.4**) et maintenir l'ordre de détection i=0,…N ;
2. **Pour** chaque CFC C_i, i=0,…,N **Faire** *// N : nombre de niveaux détectés*
3. Construire le PDM restreint PDM_i**;**
4. (V^*_i, π^*_i) ←**IVGSA (**PDM_i**)**; //Résoudre le PDM restreint PDM_i
5. $\pi^* \leftarrow \bigcup_{i} \pi^*_i$ et $V^* \leftarrow \bigcup_{i} V^*_i$

Remarque 3.4: La technique de recherche en profondeur utilisée par l'algorithme de Tarjan, évite de trouver les CFC non accessibles à partir d'un état initial S_0. Par exemple, dans le graphe agrégé en CFC de la **figure 3.6**, l'état initial S_0 se trouve dans la classe C_{20}, alors seulement les PDM restreints correspondants aux CFC C_{00}, C_{10}, C_{11} et C_{20} seront résolus.

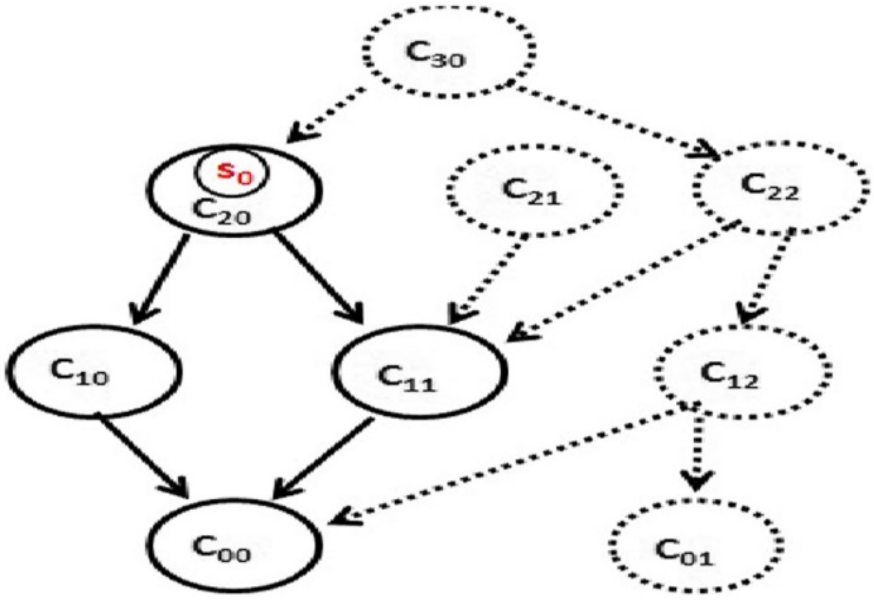


Figure 3.6 : Exemple d'un graphe agrégé en CFC montrant les classes accessibles à partir d'un état initial.

3.3.6. Nouvelle définition des PDM restreints

Afin d'accélérer les algorithmes de résolution hiérarchiques, nous proposons, dans ce paragraphe, une nouvelle définition des PDM restreints. Reprenons l'équation de Bellman relative à chaque PDM restreint $\text{PDM}_{pk}, p = 0, .., L; k = 0, .., K(p)$, et qui peut s'écrire sous la forme :

$$V_{pk}(i) = \max_{a \in A(i)} \left\{ r(i,a) + \alpha \sum_{j \in \bar{\Gamma}_a^+(i)} P(j|i,a) V^*(j) + \alpha \sum_{j \in \hat{\Gamma}_a^+(i)} P(j|i,a) V_{pk}(j) \right\} \qquad (3.2)$$

Où $\bar{\Gamma}_a^+(i)$ désigne l'ensemble des successeurs externes de i relatifs à l'action a et $\hat{\Gamma}_a^+(i)$ l'ensemble des successeurs internes de i relatifs à l'action a (notons que, pour le niveau L_0, $\bar{\Gamma}_a^+(s) = \emptyset$). $V^*(j)$ désigne la valeur optimale de l'état j calculé dans les niveaux précédents.

Puisque le terme $\alpha \sum_{j \in \bar{\Gamma}_a^+(i)} P(j|i,a) V^*(j)$ est une constante, il ne faut pas le recalculer, à chaque itération, dans les niveaux supérieurs. Ainsi, nous présentons la nouvelle définition des PDM restreints $\text{PDM}_{pk}, p = 0, \ldots L; k = 0, \ldots, K(p)$ comme suit :

- **Espace d'états:** $S_{pk} = C_{pk}$
- **Espace d'actions:** $A_{pk}(i) = A(i), i \in C_{pk}$
- **Fonction de transition:** $P_{pk}(j|i,a) = P_{iaj}\ , i \in C_{pk}, a \in A(i)$
- **Fonction de gain:** $R_{pk}(i,a) = r(i,a) + \alpha \sum_{j \in \bar{\Gamma}_a^+(i)} P_{iaj} V^*(j)\ , i \in C_{pk},\ a \in A_{pk}(i)$

Par la suite, le théorème suivant montre qu'en suivant l'ordre ascendant des niveaux, une solution du PDM original est la réunion (combinaison) des solutions des PDM restreints.

Théorème 3.2: Si V_{pk}^{*} désigne le vecteur gain optimal du PDM restreint PDM_{pk}, p=0,..,L ; k=0,..,K(p), alors $V_{pk}^{*}(i) = V^{*}(i), \forall\, i \in S_{pk}$.

Preuve : La preuve est par induction, pour p=0 et pour chaque classe C_{0k}, le vecteur gain optimal V_{0k}^{*} est l'unique solution de l'équation :

$$V_{0k}(i) = \max_{a \in A_{0k}(i)} \left\{ R_{0k}(i,a) + \alpha \sum_{j \in S_{0k}} P_{pk}(j|i,a) V_{0k}(j) \right\}, i \in S_{0k} \quad (3.3)$$

Le vecteur gain optimal V^{*} est l'unique solution de l'équation :

$$V(i) = \max_{a \in A(i)} \left\{ r(i,a) + \alpha \sum_{j \in S_{0k}} P_{iaj} V(j) \right\}, i \in S_{0k} \quad (3.4)$$

Le fait que $R_{0k}(i,a) = r(i,a)$, $A_{0k}(i) = A(i)$ et $P_{pk}(j|i,a) = P_{iaj}$ implique que $V_{0k}^{*}(i) = V^{*}(i)\ \forall\, i \in C_{0k}$. Soit p>0, supposons que le résultat est vraie pour tous les niveaux précédents et montrons qu'il est vrai pour le niveau p.

Si V_{pk}^{*} est le vecteur gain optimal du problème restreint MDP_{pk}, alors V_{pk}^{*} est l'unique solution de l'équation :

$$V_{pk}(i) = \max_{a \in A_{pk}(i)} \left\{ R_{pk}(i,a) + \alpha \sum_{j \in S_{pk}} P_{pk}(j|i,a) V_{pk}(j) \right\}, i \in S_{pk} \quad (3.5)$$

Puisque $R_{pk}(i,a) = r(i,a) + \alpha \sum_{j \in \bar{\Gamma}_a^{+}(i)} P_{iaj} V^{*}(j)$ et $P_{pk}(j|i,a) = P_{iaj}$, V_{pk}^{*} est l'unique solution de l'équation :

$$V_{pk}(i) = \max_{a \in A(i)} \left\{ r(i,a) + \alpha \sum_{j \in \bar{\Gamma}_a^{+}(i)} P_{iaj} V^{*}(j) + \alpha \sum_{j \in S_{pk}} P_{iaj} V_{pk}(j) V_{pk}(j) \right\}, i \in S_{pk} \quad (3.6)$$

$V^{*}(j)$ est la valeur optimale du PDM original où j est un état appartenant à un niveau précèdent. D'après l'hypothèse de récurrence, on a $V_{pk}^{*}(j) = V^{*}(j), \forall j \in \bar{\Gamma}_a^{+}(i)$, alors $\forall i \in S_{pk}$, $V_{pk}^{*}(i) = V^{*}(i)$.

Corollaire 3.1 : Si $\pi_{pk}^{*}, p = 0, \dots L; k = 0, \dots, K(p)$ est une stratégie optimale pour le PDM restreint PDM_{pk}, alors $\pi_{pk}^{*}(i)$ est une action optimale du PDM original.

Preuve : La preuve est évidente puisque $V_{pk}^{*}(i) = V^{*}(i), \forall\, i \in S_{pk}$.

Remarque 3.5: Au cours du calcul de la fonction valeur d'un état *i*, la nouvelle définition de l'espace d'états du PDM restreint ne considère pas les successeurs externes *i*, ce qui permet de réduire l'espace d'états et ainsi le temps d'exécution.

La procédure 3.3 permet de construire le nouveau PDM restreint.

Procédure 3.3: Construction du nouvel PDM restreint

NPDMR($C_i, P, A, \bar{\Gamma}_a^+, \Gamma_a^+$**)** //$\bar{\Gamma}_a^+$: *successeurs externes ;* Γ_a^+: *successeurs externes et nternes ;*

1. Pour tout $i \in C_i$ Faire
2. Pour tout $a \in A(i)$ Faire
3. Pour tout $j \in \bar{\Gamma}_a^+(i)$ Faire *//parcourt des successeurs externes du couple (i, a).*
4. $R_{ia} \leftarrow R_{ia} + \alpha P_{iaj} V^*(j)$ *//Ajout du gain résultant du niveau inférieur.*
5. $\Gamma_a^+(i) \leftarrow \Gamma_a^+(i) \backslash \{j\}$ *//Elimination du successeur externe de l'ensemble des successeurs.*

Notons que pour chaque successeur externe du couple état/action (*i*,*a*), le gain résultant du niveau précèdent est ajouté (Ligne 4, procédure 3.3) et le successeur externe est éliminé (Ligne 5, procédure 3.3). Nous proposons, par la suite, une quatrième version de l'algorithme IV hiérarchique séquentiel (**Algorithme 3.10**) comme suit.

Algorithme 3.10: Algorithme IVGS hiérarchique séquentiel (version 4)

IVGSHS_V4 (Entrée : PDM ; Sortie : V^*, π^*)

1. Déterminer les CFC C_i et maintenir l'ordre de détection *i*=0,…*N* ;
2. Pour chaque CFC C_i, *i*=0,…,*N* Faire
3. Construire le nouveau PDM restreint PDM_i en utilisant la procédure 3.3;
4. $(V_i^*, \pi_i^*) \leftarrow$IVGSA ($PDM_i$); //Résoudre le PDM restreint PDM_i
5. $\pi^* \leftarrow \bigcup_i \pi_i^* \quad et \quad V^* \leftarrow \bigcup_i V_i^*$

Remarque 3.6: Il est plus efficace de construire le nouveau PDM restreint durant la première itération de l'algorithme IV. En effet, l'étape 3 de l'**algorithme 3.10** peut être exécutée dans la première itération de l'étape 4. Une autre version peut ainsi être conçue. De plus, un successeur externe peut être détecté par un numéro de classe ou de niveau diffèrent.

La **figure 3.7** montre l'apport de la nouvelle définition des PDM restreints en comparant les deux versions 3 et 4 de l'algorithme IV hiérarchique séquentiel, pour des modèles générés aléatoirement.

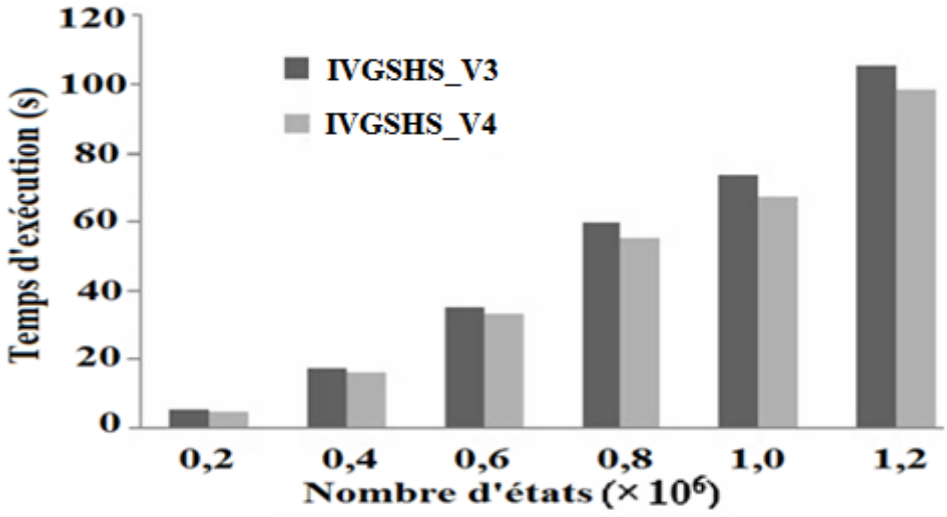


Figure 3.7 : Comparaison entre l'algorithme IVGS hiérarchique séquentiel avec l'ancienne définition des PDM restreints (IVGSHS_V3) et celui avec la nouvelle définition (IVGSHS_V4).

3.4. Algorithmes parallèles

L'architecture multiprocesseur des ordinateurs à mémoire partagée ou à mémoires distribuées offre la possibilité d'exécuter plusieurs programmes en même temps s'ils sont indépendants. Le modèle à mémoire partagée [DAG98, SAT98] propose un espace mémoire commun accessible pour tous les processeurs, alors que le modèle à mémoire distribuée est constituée de plusieurs processeurs ayant chacun sa propre mémoire centrale isolée des autres. Ils sont reliés entre eux à travers un réseau d'interconnexion. Ainsi, les communications doivent se faire explicitement par le programmeur en partageant des messages entre processeurs, entrainant un effort de développement supplémentaire, alors on est devant une communication de type « passage de messages ». L'interface MPI (**M**essage **P**assing **I**nterface) [ELN11, GRO96, GRO99, GRO02] est l'environnement de programmation le plus connu pour les architectures à mémoires distribuées, inventée en 1992, elle comporte une librairie de gestion de la programmation parallèle et distribuée sous différents langages de programmation (C, C++ et JAVA). Le programme parallèle utilisant le MPI, est exécuté en lançant plusieurs processus où chaque processus gère sa propre mémoire sans connaissance de celle des autres. La communication entre ces processus, soit pour transférer des données ou bien de se synchroniser; se fait via un mécanisme d'échange de messages.

Commençant par le modèle à mémoire partagée qui offre la possibilité d'exécuter simultanément plusieurs sous-programmes appelés « threads ». Reprenons l'algorithme IV (**Algorithme 1.3**), les calculs des $V_{n+1}(i), i \in S$, dans chaque itération, sont indépendants. Si p est le nombre de processeurs, l'espace d'états de dimension $|S|$ est divisé en p sous-espaces $S_i, i = 1,..,p$ de même taille égale à $|S|/p$. Ainsi, deux « threads » seront créés, un premier pour le calcul de V_{n+1} dans chaque itération (Procédure 3.4) et un deuxième pour le calcul d'une stratégie optimale (Procédure 3.5). Chaque thread est paramétré par un numéro et le sous-espace auquel il sera attribué.

Procédure 3.4 : Thread pour le calcul de V_{n+1}.

Thread1 (num_thread, S_i)

Pour tout $s \in S_i$ faire

$$V_{n+1}(s) \leftarrow \max_{a \in A(s)} \left\{ r(s,a) + \alpha \sum_{s' \in S} P(s'|s,a) V_n(s') \right\}$$

Procédure 3.5: Thread pour le calcul de π^*.

Thread2 (num_thread, S_i)

Pour tout $s \in S_i$ faire

$$\pi^*(s) \leftarrow \underset{a \in A(s)}{argmax} \left\{ r(s,a) + \alpha \sum_{s' \in S} P(s'|s,a) V_n(s') \right\}$$

D'ailleurs, les calculs des V_{n+1} relatives aux sous espaces $S_i, i = 1,..,p$ se fait en parallèle dans chaque itération, ainsi le temps de calcul sera réduit. L'**algorithme 3.11** présente une autre version de l'algorithme IV parallèle.

Algorithme 3.11: Algorithme IV Parallèle

IVP (PDM)
1: Initialiser $V_0 \in \mathcal{V}$;
2: n←0 ;
3: Diviser S en p sous-espace S_i ;
4: Répéter
5: Pour tout i←1 jusqu'à p Faire *//Exécution en parallèle*
6: Thread1 (i, S_i) *//Appel du Thread1*
7: attendre la fin de tous les Threads
8: Jusqu'à convergence à ε-pres
9: Pour tout i←1 jusqu'à p Faire en parallèle
10: Thread2 (i, S_i) *//Appel du Thread2*
11: Attendre la fin de tous les Threads
12: Retourner V^*, π^*

La **figure 3.8** montre une comparaison entre l'algorithme IV séquentiel et l'algorithme IV parallèle avec une architecture à trois processeurs. Nous constatons une réduction importante du temps d'exécution.

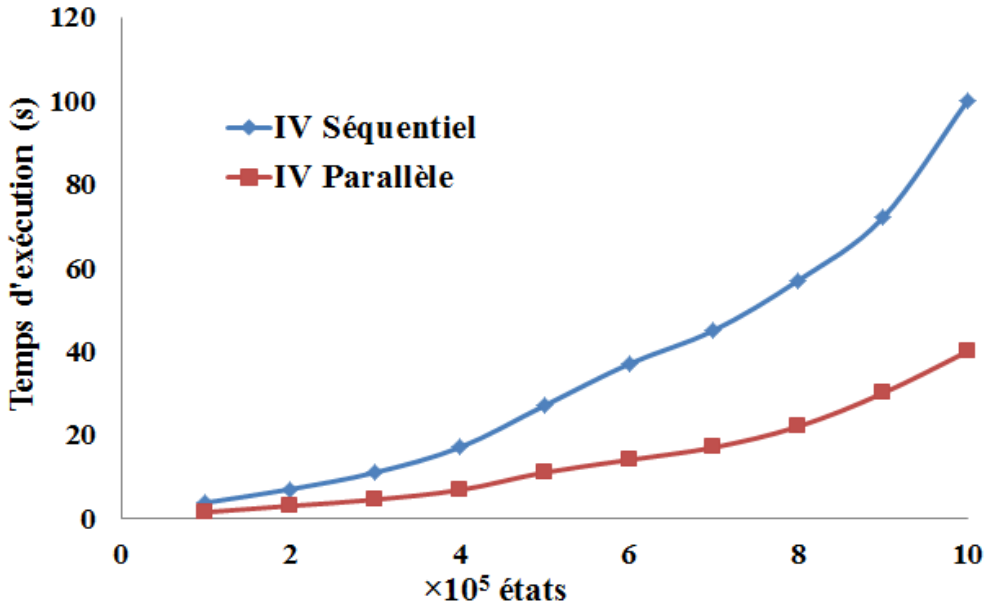


Figure 3.8 : Comparaison du temps d'exécution des deux algorithmes IV séquentiel et parallèle.

Notons que le parallélisme ne peut être appliqué directement à la variante IVGS, qu'après la décomposition du PDM original en PDM restreints. Nous proposons ainsi l'algorithme IVGS hiérarchique parallèle (**Algorithme 3.12**). Sa première étape consiste à décomposer l'espace d'états en niveaux $L_p, p = 0, .., N-1$ (Ligne 1, **Algorithme 3.12**). Après avoir initialisée et configurée l'interface de programmation MPI (Ligne 2, **Algorithme 3.12**), il calcule le nombre β_m^p de classes à attribuer au processeur μ_m dans le niveau L_p (Ligne 3, **Algorithme 3.12**). Ensuite, il transmet les données nécessaires via le protocole MPI

à chaque processeur (Ligne 4, **Algorithme 3.12**) qui se chargera d'initialiser la fonction de valeur des classes attribuées (Ligne 5, **Algorithme 3.12**). La résolution des PDM restreints commence à partir du niveau 0 jusqu'au dernier niveau (Ligne 6, **Algorithme 3.12**) où chaque processeur construit et résolve les PDM restreints PMD_{pk} (Ligne 7, **Algorithme 3.12**). Vu que nous considérons la nouvelle définition des PDM restreints qui nécessite la connaissance des données des niveaux précédents, un échange de ces données via le protocole MPI est ordonné (Ligne 8, **Algorithme 3.12**).

Algorithme 3.12: Algorithme IVGS Hiérarchique Parallèle

IVGSHP(PDM)

1: Décomposer l'espace d'états en niveaux L_p en utilisant l'**algorithme 3.6**
2: Initialiser et Configurer l'interface MPI. //MPI.init()
3: Calculer le nombre β_m^p de CFC à attribuées à chaque processeur μ_m dans chaque niveau L_p
4: Transmettre les données via le protocole MPI
5: Ordonner chaque processeur μ_m d'exécuter les tâches suivantes
    Pour chaque $i \in S_{pk}$, $p = 0, .., N-1$ et $k = 1,2, \ldots \beta_m^p$ Faire
        $V_{pk}^0(i) \leftarrow 0$
6: Pour chaque niveau $L_p, p = 0, \ldots N-1$ Faire
7: Ordonner chaque processeur μ_m d'exécuter les tâches suivantes
8: Pour chaque classe S_{pk} , $k = 1,2, \ldots \beta_m^p$ Faire
            Construire le PDM restreint PMD_{pk} selon la nouvelle définition
            IVGSA (PMD_{pk}) *//Résoudre le PDM restreint* PMD_{pk}
9: Echanger les données via le protocole MPI

La **figure 3.9** présente une comparaison du temps d'exécution entre l'algorithme IVGS séquentiel et l'algorithme IVGS hiérarchique parallèle en fonction du nombre de processeurs utilisés.

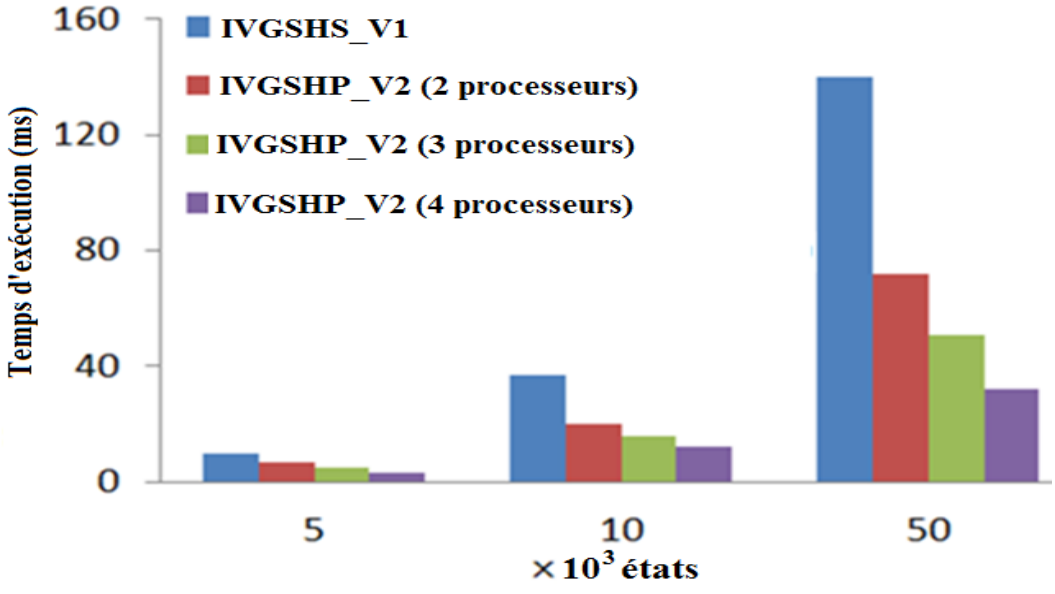


Figure 3.9 : Comparaison du temps d'exécution de l'algorithme IVGS séquentiel avec l'algorithme IVGS hiérarchique parallèle.

3.5. Exemple de simulation en navigation Robotique.

Dans ce paragraphe, nous considérons le modèle de décision markovien, présenté au chapitre 2, pour la navigation robotique. Nous choisissons l'exemple du problème, présenté dans la **figure 3.10**, où un robot est chargé d'une mission dans un couloir. Une fois qu'il a besoin de charger sa batterie, il doit décider de s'orienter vers la position de charge (état but) la plus proche qui se trouve dans les bureaux. Tenant compte de l'espace d'actions. Ce PDM est communicant (une seule classe communicante), mais dans certains états (entrées des bureaux), le robot ne peut exécuter qu'une seule action (actions indiquées par flèches dans la **figure 3.10**), une fois il atteint l'un de ces états, il ne peut pas revenir en arrière.

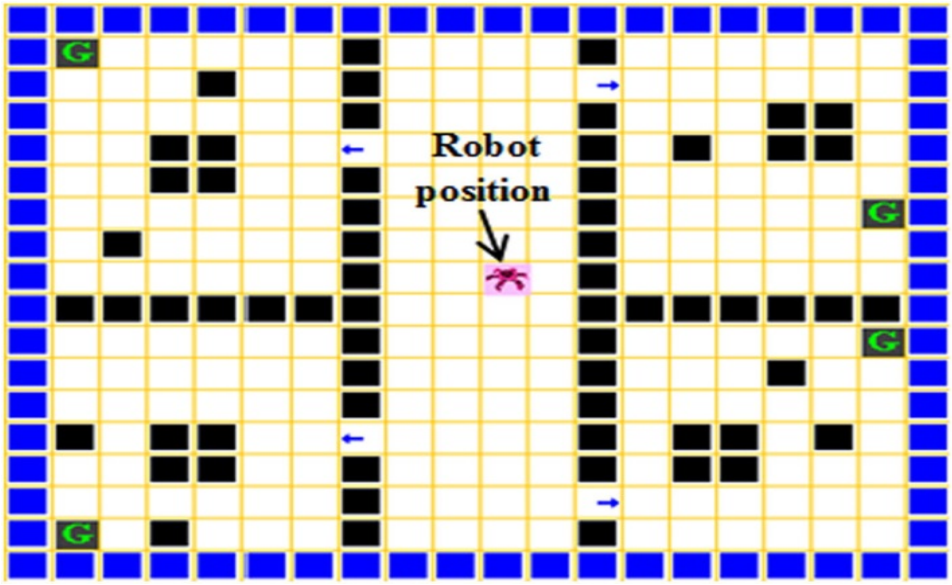


Figure 3.10 : Exemple d'environnement contenant un couloir et quatre bureaux.

En utilisant l'algorithme modifié de Tarjan (**Algorithme 3.6**), l'espace d'états est décomposé en cinq régions qui correspondent aux CFC (**Figure 3.11**). A chaque région R_{pk} correspond un PDM restreint.

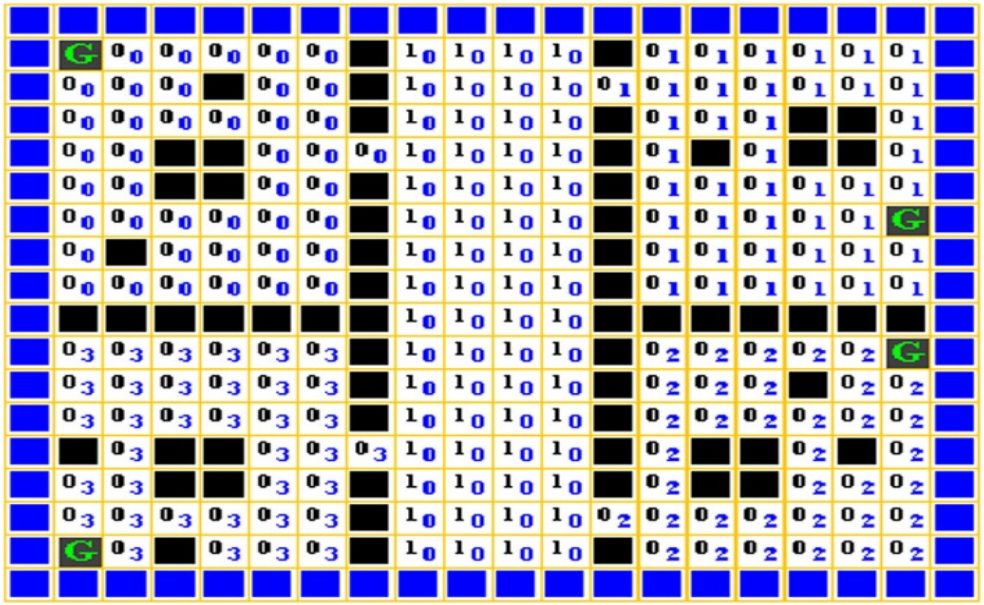


Figure 3.11 : Environnement décomposé en niveaux (régions).

Les PDM restreints du niveau 0, $PDM_{0k,}$ k=0,1,2 et 3 correspondants aux quatre régions (bureaux) sont résolus en parallèle. La **figure 3.12** montre une stratégie optimale pour chacune de ces quatre régions.

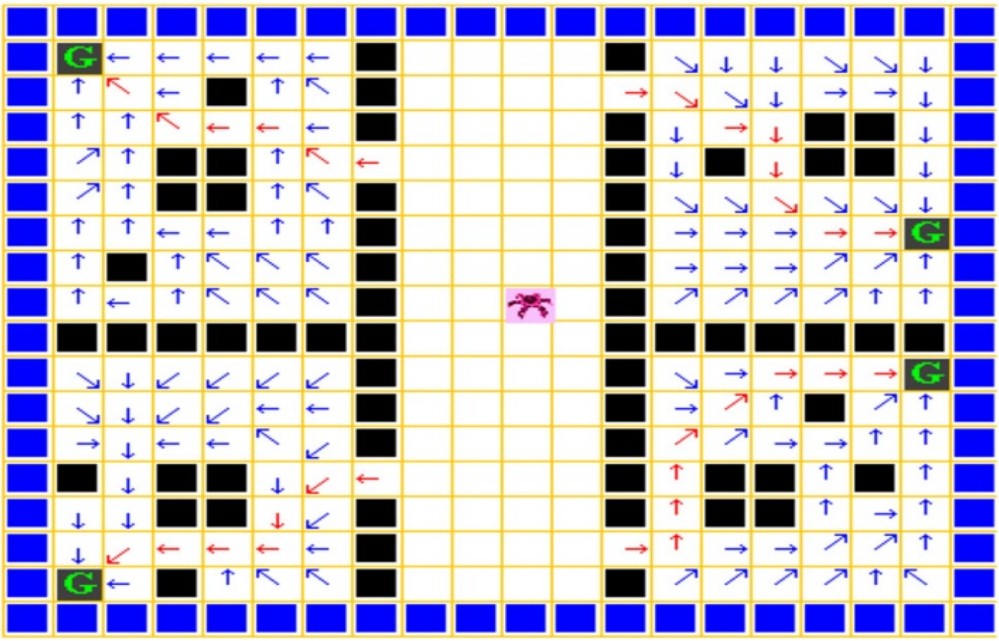


Figure 3.12 : Solutions obtenues pour les régions du niveau 0.

Après avoir résolu les quatre PDM restreints du niveau L_0, on passe à la résolution du PDM restreint PDM_{10} du niveau 1. La **figure 3.13** montre la solution obtenue, les états buts indiqués par des flèches représentent des successeurs externes éliminés par la construction du PDM restreint, ces états sont considérés comme étant des états buts dans l'ancienne définition des PDM restreints [ABD03b].

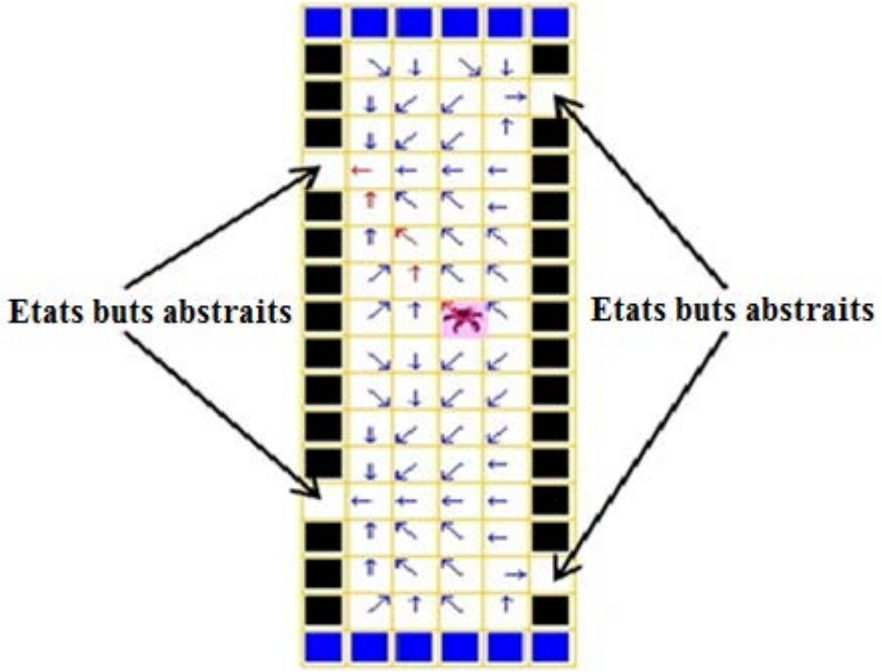


Figure 3.13 : Solution obtenue pour la région de niveau 1.

Finalement, en combinant les cinq solutions obtenues pour ces PDM retreints, nous obtenons ainsi une stratégie optimale indiquée par des flèches rouges (**Figure 3.14**).

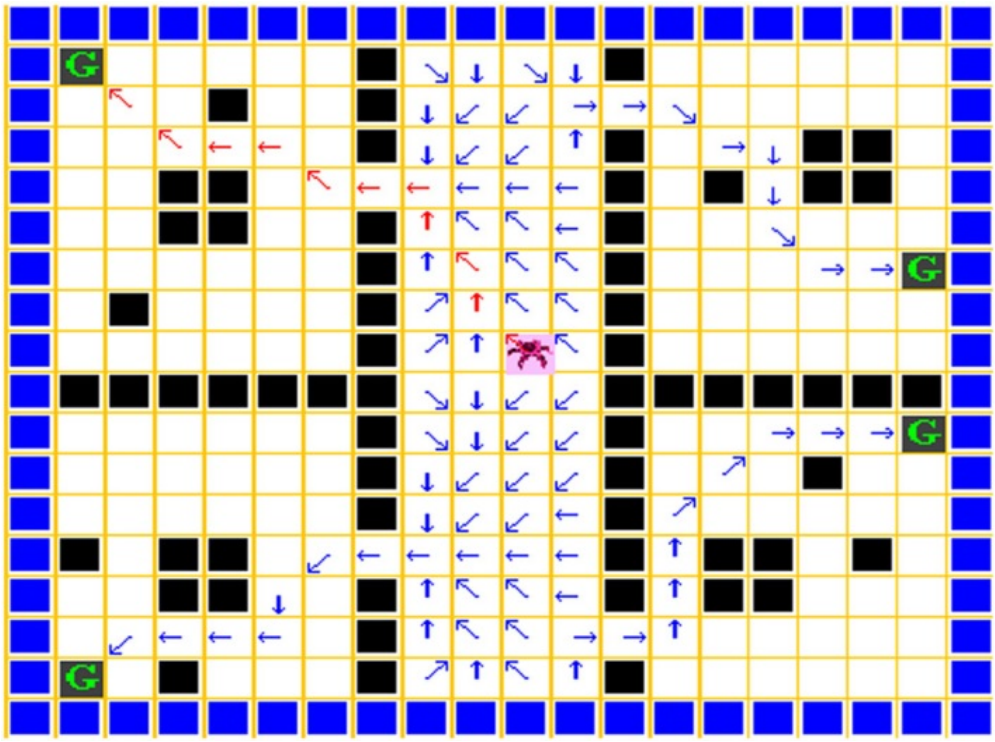


Figure 3.14 : Solution optimale pour l'ensemble d'environnement contenant les cinq régions.

3.6. Conclusion

Dans ce chapitre, nous avons mis l'accent sur les méthodes hiérarchiques de résolution des PDM basées sur la décomposition de l'espace d'états en CFC, qui sont eux aussi classifiées en niveaux. Chercher un algorithme efficace de décomposition est d'une importance majeure, c'est ainsi que nous avons présenté deux contributions, à savoir une variante optimisée de l'algorithme de Tarjan permettant de trouver les CFC et un nouvel algorithme qui détermine simultanément les CFC et leurs niveaux d'appartenance. Ceci nous a permis d'optimiser le temps d'exécution de la décomposition.

Nous avons également exposé une nouvelle définition des PDM restreints qui permet de réduire l'espace d'états et d'accélérer l'algorithme de résolution IV hiérarchique.

En outre, pour plus d'optimisation, nous avons présenté l'algorithme IV parallèle, IV hiérarchique parallèle et l'algorithme IVGS hiérarchique parallèle.

Cependant, ces méthodes hiérarchiques ne peuvent pas être bénéfiques dans le cas d'un PDM communicant ne comportant qu'une seule classe communicante, et aussi dans le cas du critère total où le problème d'existence de solution s'impose. Nous présentons d'abord, dans le chapitre suivant, une autre technique de décomposition adéquate pour un PDM communicant orienté but, tel que le problème du plus court chemin stochastique avec des impasses qui est modélisé sous le critère total. Ensuite, nous proposons une nouvelle transformation du modèle qui assure l'existence d'une solution, la convergence des algorithmes itératifs, et résout le problème d'accessibilité énergétique.

Chapitre 4

Contributions au Problème du Plus Court Chemin Stochastique avec des Impasses

4.1. Introduction

Actuellement, de nombreux auteurs se sont attelés à raffiner les modèles de plus court chemin, en proposant des algorithmes de plus en plus efficaces ou en leur apportant des sophistications (du statique vers le dynamique, du déterministe au stochastique). Le problème de plus court chemin stochastique (en anglais Stochastic Shortest Path problem SSP) défini par WHITE [WHI93] comme un PDM avec des coûts positifs (gains négatifs) et un état ou ensemble d'états absorbants (états buts). Il consiste à trouver une stratégie optimale atteignant le but ou le plus proche but avec une espérance de coût minimal. Si l'espace d'états contient des impasses, l'agent ne peut toujours atteindre le but avec une probabilité égale à 1, dans ce cas le problème sera multicritères : maximiser la probabilité d'atteindre le but et minimiser l'espérance du coût.

Vu son importance et ses diverses applications, plusieurs contributions ont été proposées pour résoudre le problème du plus court chemin stochastique. Bonet et Geffner [BON02, BON03a, BON03b] proposent quelques algorithmes de résolution en utilisant le critère total : 1) l'algorithme RTDP [BON02] (en anglais Real-Time Dynamic Programming en anglais), qui est un algorithme heuristique utilisant une série d'épisodes qui se termine par l'atteint de l'état but, il ressemble en quelque sorte à un algorithme d'apprentissage; 2) l'algorithme LRTDP [BON03a] (en anglais Labeled RTDP) qui converge plus vite que RTDP car chaque état qui converge dans une épisode est marqué comme résolu, ce qui minimise la mise à jour des états dans les épisodes suivantes; (3) l'algorithme hiérarchique (en anglais FIND-and-REVISE heuristic search framework) [BON03b] qui ressemble en quelque sorte aux méthodes hiérarchiques présentées dans le chapitre précédent.

Néanmoins, ces approches nécessitent que l'état but soit accessible à partir de l'état initial avec une probabilité égale à 1, ce qui ne garantit ni une solution du problème des impasses, ni le traitement de la contrainte des ressources disponibles (énergie par exemple). Kolobov [KOL11] propose l'approche dite MAX-PROB qui maximise uniquement la probabilité d'atteindre le but. Subséquemment, Weld [WEL12] présente quelques approches pour trois différentes classes du problème: i) une première classe, où il suppose que les impasses peuvent être évités à partir de l'état initial (en anglais SSP with Avoidable Dead Ends SSPADE) ; ii) deux autres classes où il assigne une pénalité finie ou infinie si un état impasse est visité (en anglais Finite and infinite SSP for a visited Unavoidable Dead Ends (fSSPUDE, iSSPUDE)). Ensuite, Teichteil-Konigsbuch [TEI12] et Kolobov [KOL12] proposent d'autres approches dans lesquelles les impasses sont éliminées et la probabilité d'atteindre le but est maximisée. Trevizan [TRE17] propose un nouveau critère d'optimisation qui minimise le coût étant donnée une probabilité maximum d'atteindre le but, ce critère est connu sous le nom anglais « Min-Cost given Max-Prob (MCMP criterion) », la

résolution du problème est basée sur l'algorithme de programmation linéaire, qui est, cependant, très coûteux pour des espaces d'états de grande taille.

Nous proposons, dans ce chapitre, une nouvelle transformation du problème du plus court chemin stochastique qui permet de résoudre le problème des impasses et de répondre à la question de suffisance des ressources (énergie, temps,..) pour atteindre le but. Nous commençons, dans la section 2, par définir le cadre théorique du problème du plus court chemin stochastique avec les impasses, puis nous présentons, dans les deux sections suivantes, la nouvelle transformation suivie d'une étude théorique montrant sa validité. Ensuite, pour accélérer la résolution du problème, un algorithme topologique sera présenté dans la section 5 et des exemples de simulation montrant les avantages de cette transformation, ainsi que la performance de l'algorithme proposé. Finalement, nous présentons, dans la section 6, une application au problème de couverture de zone, dans laquelle nous proposons un modèle de décision markovien et un algorithme de couverture de zone appliqué au robot démineur.

4.2. Plus court chemin stochastique avec des impasses

Le problème de plus court chemin stochastique, défini par WHITE [WHI93], est un PDM avec des coûts positifs (gains négatifs) et un état ou ensemble d'états absorbants (états buts), il est défini par le tuple (*S, A, T, P, C, G, S_0*), où:

- (*S, A, T, P, C)* est le tuple PDM.
- *G* est l'ensemble des états absorbants ou états buts, $\forall\, g \in G, \forall\, a \in A, P_{gag}=1$ et $C_{ga}= 0$.
- S_0 est l'état initial qui peut être un ensemble d'états.

Ce problème est généralement modélisé par un PDM sous le critère total, la fonction de valeur, qui est l'espérance du coût quand le processus démarre de l'état initial *i* en suivant la stratégie π est définie par :

$$V_\pi(i) = E_\pi\left(\sum_{t=1}^{\infty} C_{x_t a_t}\right), i \in S, \tag{4.1}$$

où $C_{x_t a_t}$ désigne le coût obtenu quand le processus est dans l'état x_t après t décisions et l'action exécutée est a_t. Lorsque le coût dépend de l'état suivant, il est défini par :

$$C_{x_t a_t} = \sum_{x_{t+1} \in S} P_{x_t a_t x_{t+1}} C_{x_t a_t x_{t+1}}, \tag{4.2}$$

où $C_{x_t a_t x_{t+1}}$ désigne le coût obtenu lorsque l'action a_t est choisi dans l'état x_t et le processus transite vers l'état x_{t+1}.

L'objectif est de trouver V^*, l'espérance du vecteur coût minimal à horizon infini défini par :

$$V^*(i) = \inf_{\pi \in F_D} V_\pi(i), i \in S \tag{4.3}$$

Le vecteur coût minimal V^*, s'il existe, est l'unique point fixe de l'équation de Bellman définie par :

$$V(i) = \min_{a \in A(i)} \left\{ \sum_{j \in S} P_{iaj}\left(C_{iaj} + V(j)\right) \right\}, i \in S \quad (4.4)$$

En cas d'existence du vecteur coût minimal V^*, la stratégie optimale est donnée par :

$$\pi^*(i) \in \underset{a \in A(i)}{argmin} \left\{ \sum_{j \in S} P_{iaj}\left(C_{iaj} + V^*(j)\right) \right\}, i \in S \quad (4.5)$$

Reprenons le modèle de la navigation robotique dans un environnement représenté sous forme de grilles hexagonales. L'agent robot doit naviguer vers un état but en évitant les obstacles et les mines cachées, considérées comme étant des impasses, avec un minimum coût d'énergie. Nous formulons ce problème sous forme d'un problème de plus court chemin stochastique. Chaque grille est libre, obstacle, mine cachée ou but. L'espace d'états est donc l'ensemble des grilles et peut être représenté sous forme de matrice, chaque grille est associée avec une valeur d'état, libre (0), obstacle (1), but (2) ou mine (3). Nous supposons que le robot est contrôlé par six actions (les six directions dans la grille hexagonale).

La transition vers un état libre ou un état but est caractérisée par un coût d'énergie proportionnel à la distance parcourue, nous supposons qu'il est constant et égal à $\boldsymbol{e}$.

La fonction de transition, qui définit l'incertitude liée aux effets des actions, est difficile à modéliser et peut être déterminée par un algorithme d'apprentissage. Pour une raison de simplicité, nous utilisons la fonction de transition définie dans la **figure 4.1**. L'agent transite vers l'état désiré avec une probabilité de 0.8 et vers l'un des deux états voisins avec 0.1 de probabilité. Si la probabilité de transition vers un état obstacle est diffèrent de zéro, la position du robot reste la même et le coût vaut $C_{iai} = \boldsymbol{e}$.

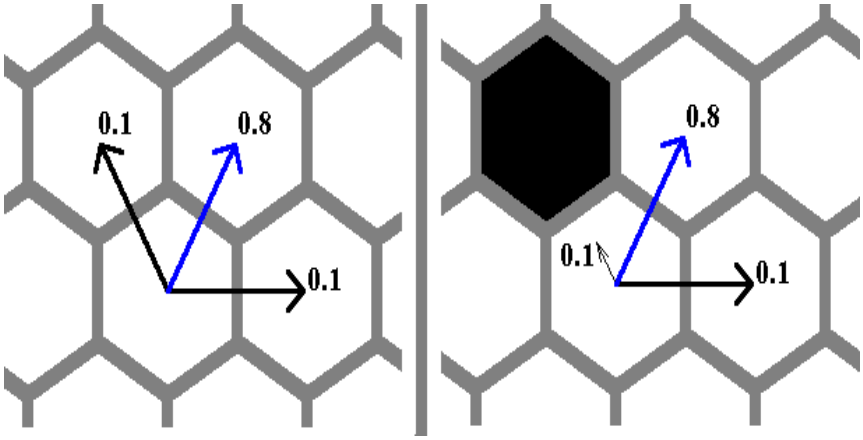


Figure 4.1 : Exemple de fonction de transition d'un robot se déplaçant dans un environnement représenté sous forme de grilles hexagonales.

La convergence des algorithmes de résolutions itératifs, tels que l'algorithme IV et ses variantes, ne peut être garantie pour les modèles comportant des impasses ou lorsque l'état but ne peut être accessible à partir d'un état initial. La **figure 4.2** montre un exemple d'environnement, où nous simulons l'algorithme IV après 100 et 200 itérations. Nous remarquons que, pour les états à partir desquels l'agent ne peut pas atteindre le but, la fonction de valeur incrémente infiniment, alors que pour les autres états, une stratégie optimale est obtenue en un petit nombre d'itérations.

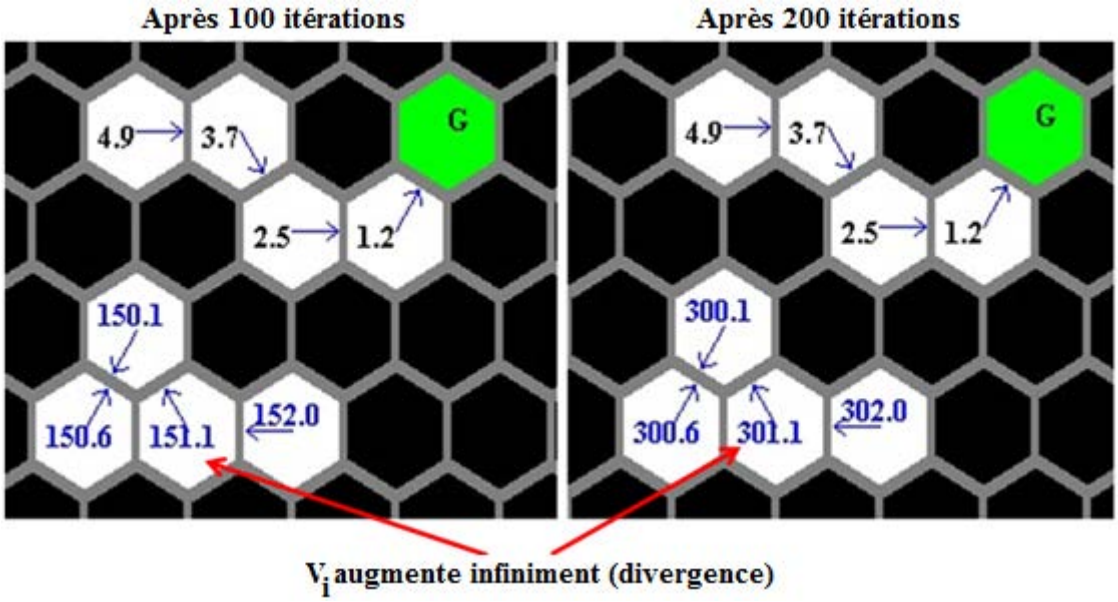


Figure 4.2 : Exemple de stratégies et de fonction de valeurs calculées après 100 et 200 itérations.

La divergence de l'algorithme IV est due à une somme infinie de coûts strictement positifs. C'est ainsi que la convergence des algorithmes itératifs ne peut être assurée avec l'existence des impasses. Pour contourner ce problème et répondre aussi à la question de suffisance des ressources (énergie, temps, etc.), nous proposons, par la suite, une nouvelle transformation du cadre théorique du problème du plus court chemin stochastique.

4.3. Nouvelle transformation du modèle

La nouvelle transformation proposée est définie par le tuple (S', A', T, P', C', G', S_0, E, C_d, C_a, C_g) où:

- $S' = (S \backslash S_D) \cup \{s_d\}$, où S_D est l'ensemble des états impasses et s_d un état représentant l'ensemble S_D agrégé.
- $G' = G \cup \{g^*\}$ où g^* est un état but abstrait.
- A'=$A \cup \{\theta\}$ où θ est une action imaginaire qui déplace le processus vers l'état but abstrait g*, pour tout $s_t \in S' \backslash \{s_d\}$, $A'(s_t) = A(s_t) \cup \{\theta\}$.
- $P'_{s_t a_t s_{t+1}} = P_{s_t a_t s_{t+1}}$, $\forall\, a \in A'(s_t) \backslash \{\theta\}$ *et* $P'_{s_t \theta g^*} = 1$
- $C'_{s_t a_t s_{t+1}} = C_{s_t a_t s_{t+1}} si\ s_{t+1} \in S' \backslash \{s_d\}$
- $C'_{s_t \theta g^*} = C_a$, où $C_a \geq 0$ désigne un coût positif abstrait obtenu lorsque l'état but abstrait est atteint.
- $C'_{s_t a_t g} = C_{s_t a_t g} - C_g$, où $C_g \geq 0$ désigne un coût ajouté lorsque l'état but est atteint.
- $C'_{s_t a_t s_d} = C_{s_t a_t s_d} + C_d$, où $C_d \geq 0$ désigne un coût obtenue lorsque l'état impasse s_d *agrégé* est atteint.
- S_0 est l'état initial ou l'ensemble d'états initiaux donnés.
- E désigne le coût maximal supporté par l'agent avant d'atteindre le but (énergie initiale, temps maximal, etc.).

La **figure 4.3** présente un exemple du modèle transformé représenté par un graphe avec une seule action par état, les valeurs en noir désignent les probabilités de transition et les valeurs en rouge désignent les coûts obtenus. Les états impasses d_1 et d_2 sont agrégés en un unique état S_d et l'action θ (flèche en rouge) qui déplace le processus vers l'état but abstrait g*, est ajoutée à l'espace d'actions avec un coût égal à C_a; le coût C_d est ajouté lorsqu'il y a transition vers l'état impasse s_d agrégé. Le gain $-C_g$ est aussi ajouté lorsqu'il y a transition vers l'état but.

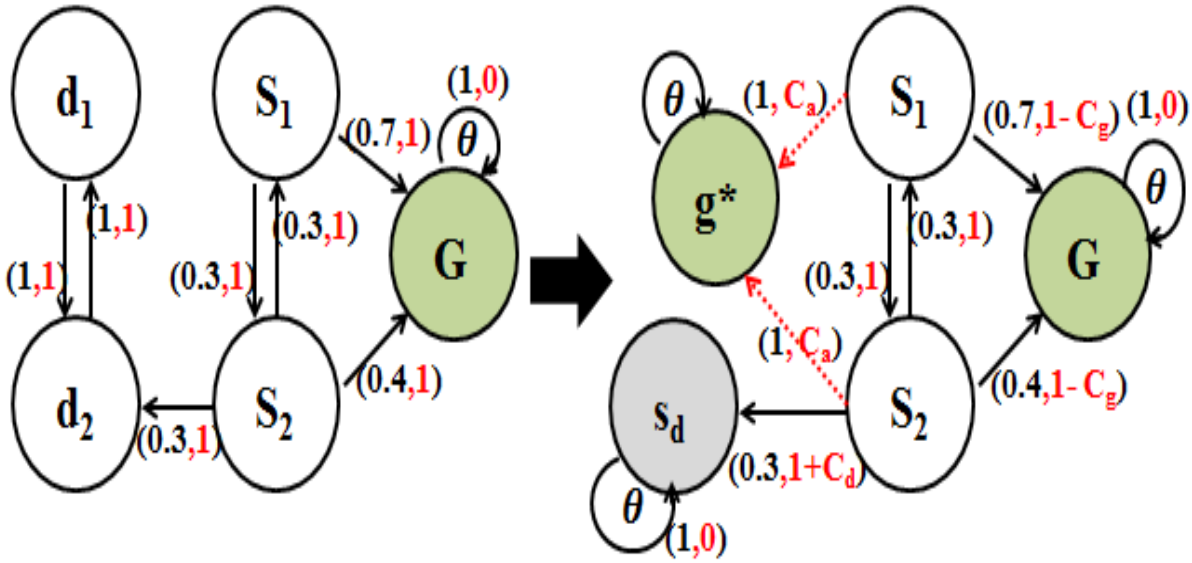


Figure 4.3 : Exemple d'un modèle transformé.

4.4. Etude théorique du modèle transformé

Nous présentons, dans cette section, une étude théorique du modèle transformé afin de montrer la convergence des algorithmes de résolution itératifs et présenter ses avantages.

Proposition 4.1: Le vecteur coût optimal V^* existe, fini et $V^*(i) \in [C_g, C_a], i \in S'$.

Preuve: Supposons que $V^* > C_a$ et soit $\pi_\theta \in F_D$ définie par $\pi_\theta(s_t) = \theta$ pour tout $s_t \in S' \backslash \{s_d\}$. A partir de la définition du modèle transformé on a : $C'_{s_t\theta} = C_a$ et $P_{s_t\theta g^*} = 1$ pour tout $s_t \in S' \backslash \{s_d\}$, ce qui implique que $V_{\pi_\theta} = C_a$ et $V_{\pi_\theta} < V^*$. Ce qui contredit le fait que V^* est le vecteur coût optimal. Dans le modèle transformé, un coût négatif ne peut être obtenu que si l'état but terminal est atteint et le fait que tout autre coût est positif implique que $-C_g$ est la borne inférieure de V^*.

Proposition 4.2: Si l'espace d'états ne contient aucun état but et aucun état impasse, alors le vecteur coût minimal est fini ($V^* = C_a$) et $\pi^* = \pi_\theta$ est une stratégie ε-optimale.

Preuve: Le fait qu'il n'existe aucun état but et aucun état impasse implique que $\forall \pi \in F_D$, $\pi \neq \pi_\theta$, $s_t \in S'$, $a_t \in A(s_t)$, on a $C'_{s_t a_t s_{t+1}} > 0$, le vecteur coût optimal obtenu quand le processus part de l'état s_0 en utilisant la stratégie π est une somme de valeurs strictement positifs qui tend vers l'infini, donc ($V_\pi(s_0) > C_a$), ce qui contredit la proposition 4.1.

Proposition 4.3: Soit s_0 un état initial, si aucune politique ne peut atteindre l'état but à partir de s_0, alors le vecteur coût minimal est fini $(V^*(s_0) = C_a)$ et $\pi^*(s_0) = \pi_\theta(s_0) = \theta$.
Preuve: La preuve est similaire à celle de la proposition 4.2.

La **figure 4.4** montre un exemple de simulation dans un environnement généré aléatoirement avec les paramètres indiqués en bas de la figure. Nous remarquons que l'action optimale est $\pi_\theta(s_0) = \theta$ si le processus part de l'état s_0 où il n'est pas possible d'atteindre le but. Pour les autres états, l'action optimale est indiquée par une flèche. Il est clair que la fonction de valeur de chaque état est compris entre -C_g et C_a pour les deux cas. Ce qui montre la convergence des algorithmes de résolution dans pour le cas d'inaccessibilité vers l'état but.

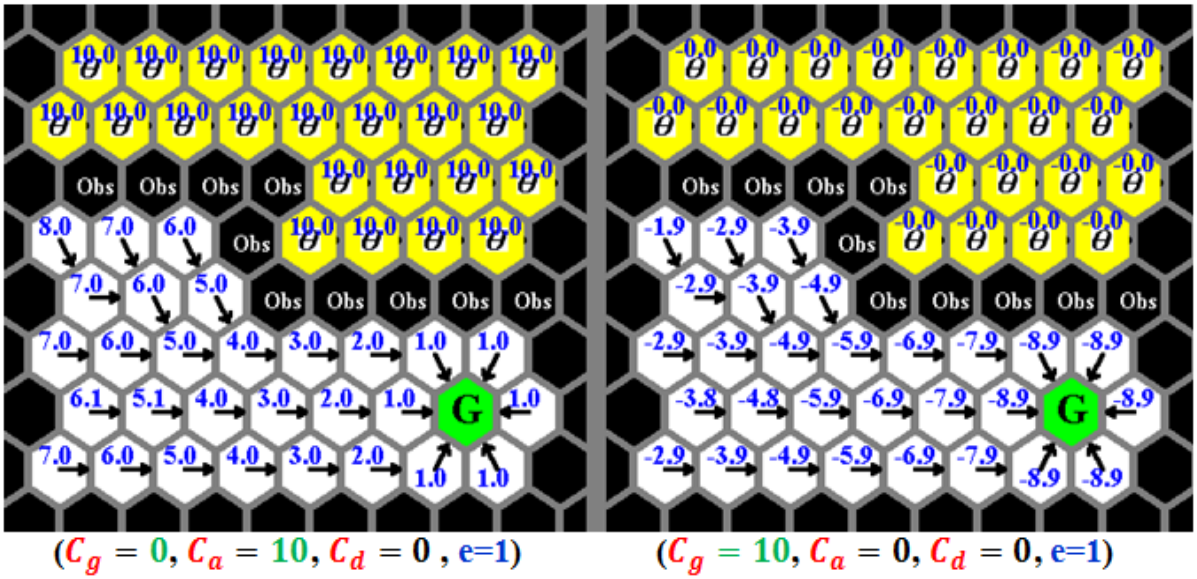


Figure 4.4 : Exemples de convergence, avec l'inaccessibilité vers l'état but, pour deux types de paramètres.

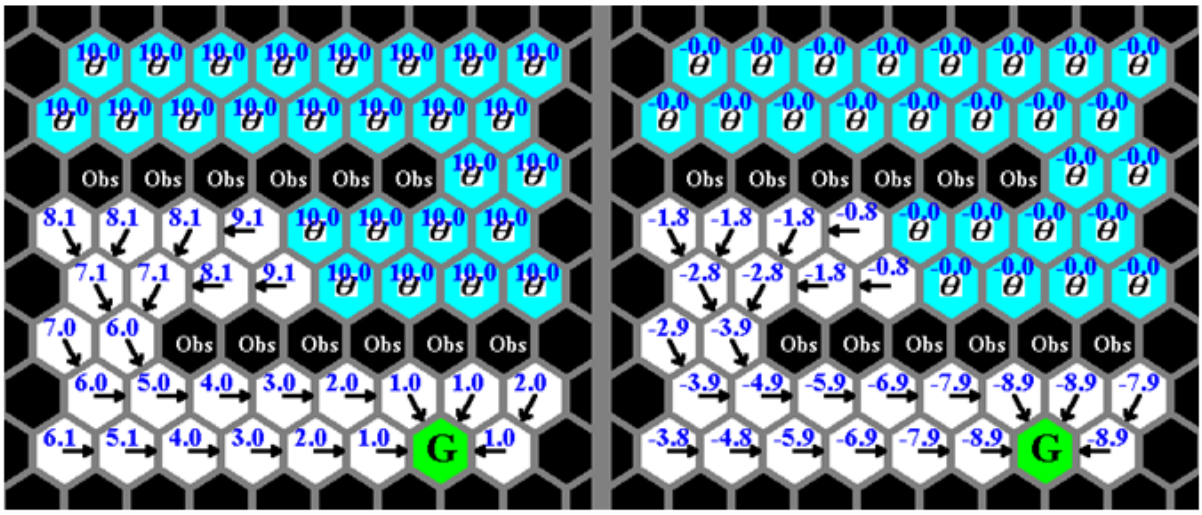
Proposition 4.4: Pour $C_g = 0$, $C_a = E$ et $C_d = 0$, si le vecteur coût minimal pour le modèle non transformé est supérieur à E, alors une stratégie ε-optimale pour le modèle transformé est $\pi^* = \pi_\theta$.

Preuve: La preuve provient du fait que $V^*_{\pi_\theta} = C_a = E$ et aucun vecteur coût optimal ne peut être supérieur à E (Proposition 4.1).

Proposition 4.5: Pour $C_g = E$, $C_a = 0$ et $C_d = 0$, si le vecteur coût optimal pour le modèle initial est supérieur à E, alors une stratégie ε-optimale pour le modèle transformé est $\pi^* = \pi_\theta$.

Preuve: Soient V_1^* (V_2^*) le vecteur coût optimal du modèle initial (modèle transformé), il est clair que $V_2^* = V_1^* - E$, donc si $V_1^* > C_g = E$, alors $V_2^* > 0$ et le fait que $V^*_{\pi_\theta} = C_a = 0$ implique que $\pi^* = \pi_\theta$.

La **figure 4.5** montre le problème d'insuffisance d'énergie pour l'atteint des buts, dans un environnement généré aléatoirement avec les paramètres indiqués en bas de la figure.



$(C_g = 0, C_a = E, C_d = 0$, e=1) $(C_g = E = 10, C_a = 0, C_d = 0$, e=1)

Figure 4.5 : Exemples de solutions optimales, pour deux types de paramètres, avec insuffisance d'énergie.

Proposition 4.6: Une stratégie ε-optimale du modèle transformé évite sûrement d'atteindre un état impasse si : $C_a = 0$, $C_g > 0$ et $C_d = \frac{C_g}{P_{s_t a_t s_d}}$.

Preuve: Soient π^* une stratégie ε-optimale et V_{π^*} le vecteur coût optimal. Supposons qu'à l'instant *t*, le processus est dans l'état s_t, $P_{s_t a_t s_d} > 0$ et qu'il atteint l'état impasse agrégé s_d à l'instant *t*+1. L'espérance du coût minimal quand le processus part de l'état s_t est donnée par :

$$V_{\pi^*}(s_t) = E_{\pi^*}^{s_t}\left(C'_{s_t a_t}\right) + \left(\sum_{t'=t+1}^{\infty} C'_{s_d \theta}\right) \tag{4.6}$$

L'espérance du coût minimal au temps t est donnée par:

$$E_{\pi^*}^{s_t}\left(C'_{s_t a_t}\right) = \sum_{s_{t+1} \in S' \cup G'} P_{s_t a_t s_{t+1}} \times C'_{s_t a_t s_{t+1}} = P_{s_t a_t s_d} \times C'_{s_t a_t s_d} + X + Y \tag{4.7}$$

Où les valeurs de *X* et *Y* sont donnés par :

$$X = \sum_{s_{t+1} \in S' \setminus S_d} P_{s_t a_t s_{t+1}} \times C'_{s_t a_t s_{t+1}} \geq 0 \tag{4.8}$$

$$Y = \sum_{g \in G'} P_{s_t a_t g} \times C'_{s_t a_t g} \tag{4.9}$$

Puisque, nous supposons que le processus atteint l'état impasse agrégé s_d à l'instant t+1, $a_t \neq \theta$, l'équation 4.9 s'écrit comme suit:

$$Y = \sum_{g \in G} P_{s_t a_t g} \times (C_{s_t a_t g} - C_g) \tag{4.10}$$

Le fait que $P_{s_t a_t g} < 1$(le processus n'atteint pas l'état but) et $C_{s_t a_t g} \geq 0$ pour tout $g \in G$ implique que $Y > -C_g$ et on a: $C'_{s_t a_t s_d} = C_{s_t a_t s_d} + C_d = C_{s_t a_t s_d} + \frac{C_g}{P_{s_t a_t s_d}}$, ce qui implique que:

$$P_{s_t a_t s_d} \times C'_{s_t a_t s_d} = P_{s_t a_t s_d} \times C_{s_t a_t s_d} + C_g \tag{4.11}$$

L'équation 4.7 peut donc s'écrire sous la forme :

$$E_{\pi^*}^{S_t}\left(C'_{s_t a_t}\right) = P_{s_t a_t s_d} \times C_{s_t a_t s_d} + C_g + X + Y \tag{4.12}$$

Le fait que $X \geq 0$ et $Y > -C_g$ implique que $C_g + X + Y > 0$ et donc: $E_{\pi^*}^{S_t}\left(C'_{s_t a_t}\right) > C_a = 0$, ce qui contredit la proposition 1.

La **figure 4.6** montre deux exemples de simulation, avec deux types de paramètres, montrant l'atteint de l'état but avec une probabilité égale à 1 ou l'agent reste sur place.

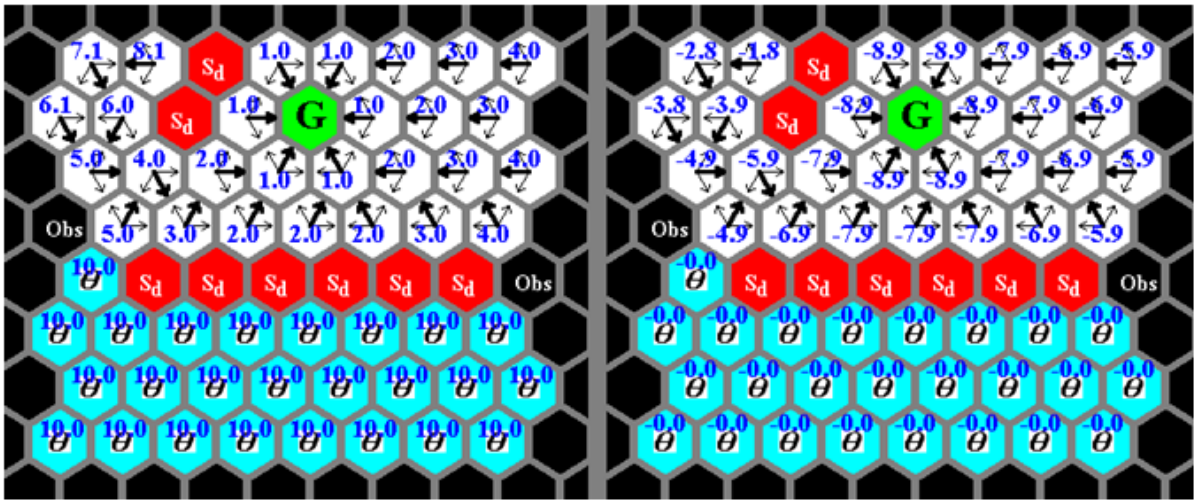


Figure 4.6 : Exemple de stratégies qui atteignent le but avec probabilité égale à 1 ou ordonnent l'agent à rester sur place.

Proposition 4.7: Si ($C_a = \infty, C_d \simeq \infty$ et $C_g \simeq \infty$) alors une stratégie ε-optimale maximise généralement la probabilité d'atteindre le but puis minimise l'espérance du coût.

Preuve: Soient π_1 (π_2) une stratégie qui permet d'atteindre le but avec une probabilité P_1 (P_2) tel que $P_2 > P_1$ et V_{π_1} (V_{π_2}) le vecteur coût espéré avec π_1 (π_2). Pour $C_d \simeq \infty$ et $C_g \simeq \infty$, on a: $V_{\pi_1} \simeq -P_1 \times C_g + (1 - P_1) \times C_d$ et $V_{\pi_2} \simeq -P_2 \times C_g + (1 - P_2) \times C_d$. Le fait que $P_2 > P_1$ implique que $-P_1 \times C_g > -P_2 \times C_g$ et $(1 - P_1) \times C_d > (1 - P_2) \times C_d$, ce qui implique que $V_{\pi_1} > V_{\pi_2}$.

Notons que $\boldsymbol{C_d \simeq \infty}$ et $\boldsymbol{C_g \simeq \infty}$ désignent que $\boldsymbol{C_d}$ et $\boldsymbol{C_g}$ sont considérablement très grands.

En utilisant la proposition 4.7, nous proposons une nouvelle approche qui maximise la probabilité d'atteindre le but puis minimise le coût avec une seule procédure d'itérations, ce qui est plus efficace par rapport à la méthode MAXPROB-MINCOST qui nécessite deux

procédures d'itérations : une première qui détermine l'espace d'actions qui maximise la probabilité d'atteindre le but et une deuxième qui minimise le coût dans cet espace d'actions. La **figure 4.7** montre la stratégie obtenue avec la méthode MAXPROB-MINCOST proposée par [**WEL12**] (**Figure 4.7, gauche**) et la stratégie obtenue avec l'approche proposée (**Figure 4.7, droite**). Nous remarquons que les deux stratégies sont identiques.

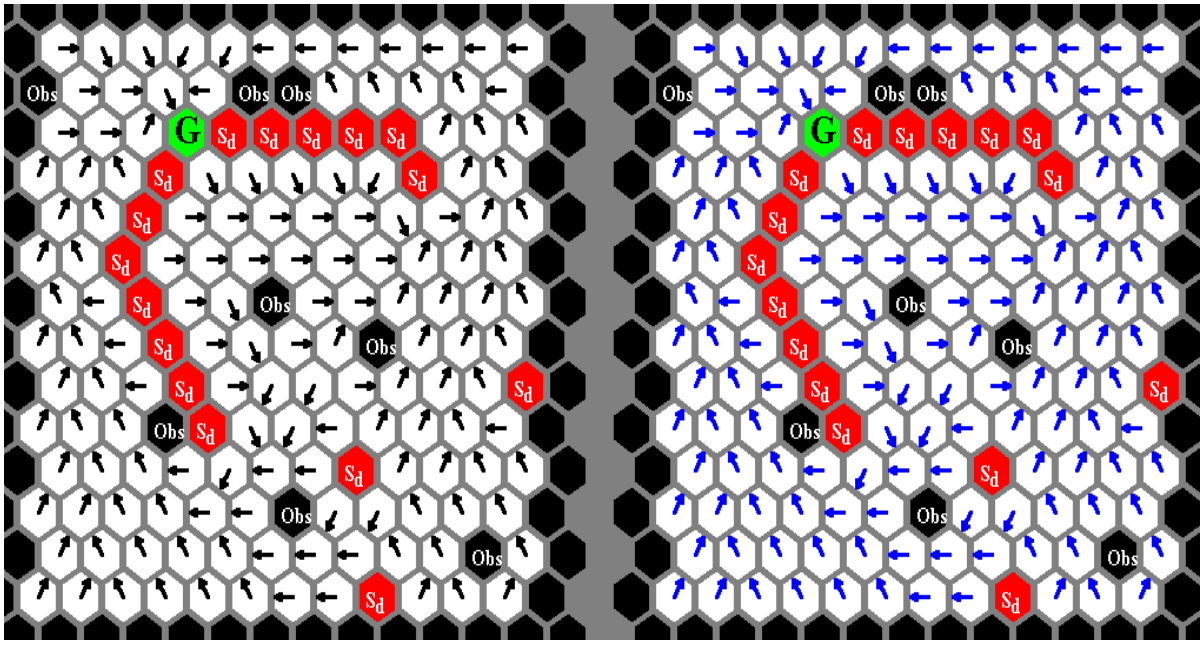


Figure 4.7 : Exemples de stratégies maximisant la probabilité d'atteindre le but et minimisant le coût.

Lemme 4.1 : Pour tout vecteur initial $V^0 \in [C_g, C_a]$, l'algorithme IV et ses variantes convergent vers une stratégie ε-optimale dans un nombre fini d'itérations.

Preuve: La preuve provient des propositions précédentes et de l'existence des états absorbants (état but et/ou état but abstrait) [BEL57, PUT94].

4.5. Algorithme topologique

Les méthodes hiérarchiques de résolution des PDM ne peuvent être intéressantes pour un PDM décomposé un petit nombre de CFC, puisqu'on peut avoir des PDM restreints de tailles assez grands. Pour cela nous proposons, dans cette section, une autre technique de décomposition et un autre algorithme hiérarchique (ou topologique) qui est efficace pour certains modèles de décision markoviens orientés but, particulièrement pour le problème du plus court chemin stochastique.

4.5.1. Technique de décomposition

Soit G_r=(S, U) le graphe associé au PDM original, où l'espace d'états S représente l'ensemble des sommets et U={ $(i,j) \in S^2 : \exists\, a \in A(i), P_{iaj} > 0$} représente l'ensemble des arcs et soit G_g l'ensemble des états buts. Nous décomposons l'espace d'états en niveaux d'accessibilité aux états buts comme suit: Le niveau 0 contient l'ensemble des états buts G_g, (L_0= G_g), le niveau 1 comporte l'ensemble des états à partir desquels le processus peut

atteindre l'un des états de L_0 en une seule transition, $L_1 = \{ i \in S \backslash L_0 : \exists\, a \in A(i), \exists\, j \in L_0, P_{iaj} > 0 \}$, et ainsi de suite, le niveau k > 1 est défini par :

$$L_k = \{ i \in \{S \backslash \{L_0 \cup L_1 \cup L_2 \cup .. \cup L_{k-1}\}\} : \exists\, a \in A(i), \exists\, j \in L_{k-1}, P_{iaj} > 0 \} \quad (4.13)$$

La **figure 4.8** montre un exemple de graphe associé à un PDM décomposé en niveaux d'accessibilité aux buts.

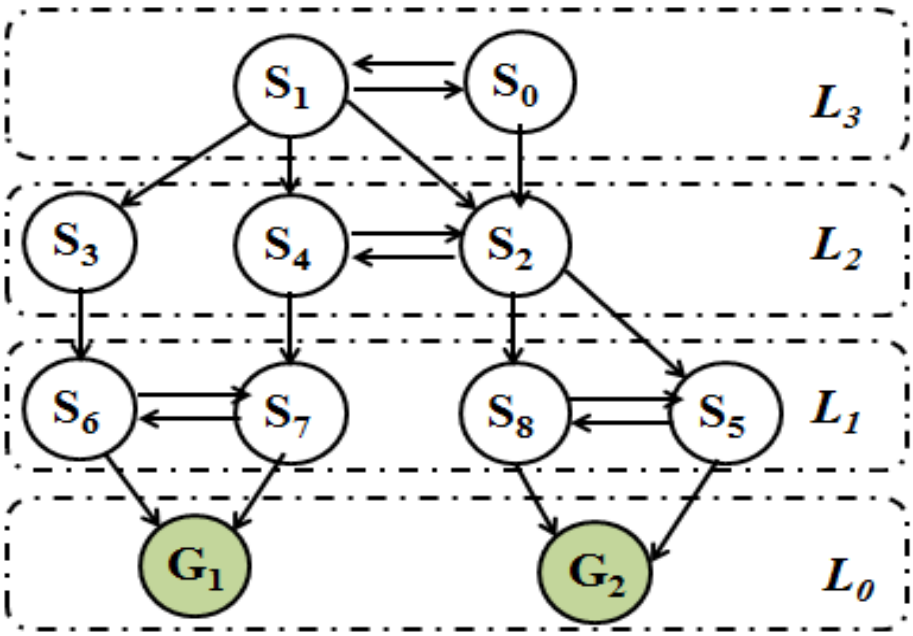


Figure 4.8 : Exemple de graphe décomposé en niveaux d'accessibilité aux états buts.

4.5.2. Algorithme de décomposition

Dans ce paragraphe, nous allons utiliser l'algorithme de recherche en largeur dans un graphe (Breadth-First Search en anglais (BFD)) pour décomposer l'espace d'états en niveaux d'accessibilité. La recherche commence à partir de l'ensemble des états buts en cherchant les états non visités contenus dans le niveau suivant jusqu'à ce que tous les états qui accèdent au moins à l'un des états buts soient visités.

Nous présentons l'algorithme de recherche ci-après (**Algorithme 4.1**) où $\Gamma^-(i) = \{ j \in S : \exists\, a \in A(j), P_{jai} > 0 \}$ désigne l'ensemble des prédécesseurs de l'état i. Après l'initialisation du paramètre "visiter" de chaque état (Faux pour tous les états et Vraie pour les états buts) (lignes 1 et 2, **algorithme 4.1**), la recherche des niveaux commence à partir des états buts (lignes 3 et 4, **algorithme 4.1**). La recherche se termine après avoir visité tous les états pouvant accéder au moins à l'un des états buts (ligne 5, **algorithme 4.1**). L'ensemble du niveau suivant est initialisé par ∅ (ligne 6, **algorithme 4.1**) et chaque prédécesseur non visité (lignes 7-9, **algorithme 4.1**) sera ajouté au niveau suivant (ligne 10, **algorithme 4.1**) et marqué comme étant visité (ligne 11, **algorithme 4.1**). L'**algorithme 4.1** s'exécute en $\mathcal{O}(|S| \times \|\Gamma^-\|)$ opérations arithmétiques où $|S|$ est le nombre d'états et $\|\Gamma^-\|$ le nombre moyen des prédécesseurs par état.

Algorithme 4.1: Algorithme de décomposition en niveaux d'accessibilité

Recherche_niveaux_accessibilite (**Entrée:** S, G, Γ^-; **Sortie**: L)

1: **pour tout** i ∈ S **Faire**
    i. visiter ←Faux;

2: **pour tout** i ∈ G **Faire**
    i. visited ←Vraie;

3: $\mathbf{L_0} \leftarrow$ G *// Le niveau 0 comporte l'ensemble des états buts*
4: k← 0; *//identificateur des niveaux*

//Recherche des niveaux jusqu'à visiter tous les états pouvant accéder au moins à l'un des états buts
5: **Tant que(** $\mathbf{L_k} \neq \emptyset$**) Faire**
6: $\mathbf{L_{k+1}} \leftarrow \emptyset$ *// Initialisation de l'ensemble des états du niveau k*
7: **pour tout** i ∈ $\mathbf{L_k}$ **Faire**
8: **pour tout** j ∈ Γ^-**(i) Faire**
9: **Si** (j.visiter = Faux) **Alors**
10: $\mathbf{L_{k+1}} \leftarrow \mathbf{L_{k+1}} \cup \{j\}$ *//Ajouter l'état j à l'ensemble* $\boldsymbol{L_{k+1}}$
11: j.visiter = Vraie *// l'état j est visité*

12: k← k+1 *//Niveau suivant*

13: **retourner** L

Remarque 4.1: i) L'**algorithme 4.1** permet d'éliminer les états impasses. Ce qui facilite l'agrégation de ces états dans le modèle transformé.

ii) La recherche du sous ensemble d'états, noté par S_1, à partir duquel au moins un état but peut être accessible avec une probabilité égale à 1 est d'une importance majeure pour les modèles où l'accès à un état impasse est une situation dramatique. Une solution consiste à utiliser l'approche MAXPROB [WEL12] pour trouver le vecteur optimal P* et dans ce cas, $S_1=\{i \in S: P^*(i) = 1\}$. Une autre solution consiste à utiliser l'algorithme de recherche en largeur (**Algorithme 5.1**) en commençant l'analyse à partir des états impasses, dans ce cas $S_1=S\setminus\{ L_0 \cup L_1 \cup ... \cup L_k\}$.

4.5.3. PDM restreints

En utilisant la décomposition en niveaux d'accessibilité, présentée ci-dessous, nous proposons une nouvelle définition des PDM restreints PDM_k correspondant à chaque niveau L_k (k=1,…, p) comme suit :

- **Espace d'états :** $S^k = L_k$
- **Espace d'actions :** $A^k(i) = A(i)$, pour tout $i \in S^k$.
- **Fonction de transition :** $\forall\, i \in S^k,\ P^k{}_{iaj} = P_{iaj}$ si $j \in \{S^{k-1} \cup S^k\}$ et $P^k{}_{iaj} = 0$ sinon.

- **Fonction coût :** $\forall\, i \in S^k$, $C^k{}_{iaj} = C'_{iaj} + P_{iaj}V^*(j)$ si $j \in S^{k-1}$ et $C^k{}_{iaj} = C'_{iaj}$ sinon, où $V^*(j)$ est la valeur optimale de j calculée dans le niveau précèdent L_{k-1}.

Les PDM restreints PDM_k correspondant à chaque niveau L_k sont résolus dans l'ordre ascendant des niveaux. Si V_k^H désigne la solution obtenue pour PDM_k, la solution combinée $V^H = \bigcup_{k=1}^{p} V_k^H$ est une heuristique admissible du problème original.

4.5.4. Enoncé de l'algorithme topologique

Après la partition de l'espace d'états en niveaux d'accessibilité et la construction des PDM restreints correspondant à chaque niveau, nous appliquons l'algorithme IVGS à chaque problème restreint, en suivant l'ordre ascendant des niveaux (**Algorithme 4.2**). La solution heuristique obtenue (V^H) est généralement très proche de la solution optimale et peut ainsi être utilisée comme vecteur initial dans l'algorithme IVGS.

Algorithme 4.2: Algorithme IVGS Topologique

IVGST(**Entrée:** PDM: S, P, A, C,G, ε; *E, C_g, C_d, C_a;* **Sortie**: V^*, π^*)

1: L ← **Recherche_niveaux_accessibilite**(S,G) *//Construction des niveaux (**algorithme 4.1**)*
2: **Pour chaque** niveau L_k, k=1,…N_l **Faire** *//Construction des PDM restreints*
   PDM_k ← PDM_restreint(PDM, L_k)

// Résolution des PDM restreints PDM_k, k=1,…N_l
3: **Pour chaque** niveau L_k, k=1,…N_l **Faire**
   (V_k^H, π_k^H) ← **IVGS**(PDM_k ; $V^0 = 0$)

4: $V^H = \bigcup_k V_k^H$ *//Combiner les solutions obtenues*
5: (V^*, π^*) ←**IVGS**(PDM, $V^0 = V^H$) *//Utiliser V^H comme vecteur initial*
6. **Return** V^*, π^*

Le tableau 4.1 montre le nombre moyen d'itérations nécessaires pour la convergence de l'algorithme IVGS qui utilise l'heuristique comme vecteur initial. Nous remarquons que la convergence est atteinte après un nombre très petit d'itérations.

Tableau 4.1 : Nombre d'itérations pour la convergence avec l'utilisation de la solution heuristique dans l'algorithme IVGS.

Nombre d'états	Nombre d'itérations
10^5	4
4×10^5	8
8×10^5	11
12×10^5	14

Théorème 4.1: L'**algorithme 4.2** est correcte et converge de façon monotone vers une solution ε-optimale dans un nombre fini d'itérations.

Preuve: Le fait que chaque PDM restreint PDM_k fournit une heuristique $V_k^H \in [C_g, C_a]$, alors la lemme 4.1 prouve la convergence de l'algorithme.

La **figure 4.9** montre la comparaison entre le temps d'exécution de l'algorithme topologique (IVGST) et celui des deux algorithmes classiques : l'itération de la valeur (IV) et sa variante de Gauss-Seidel (IVGS). Cette comparaison a été faite avec le modèle de navigation robotique dans un environnement représenté sous forme de grilles hexagonales, avec 5% d'états impasses et 5% d'obstacles. La première remarque est la rapidité de convergence de la variante de Gauss-Seidel (IVGS) par rapport à l'algorithme IV. La deuxième remarque est que l'algorithme topologique proposé est plus rapide que les deux derniers.

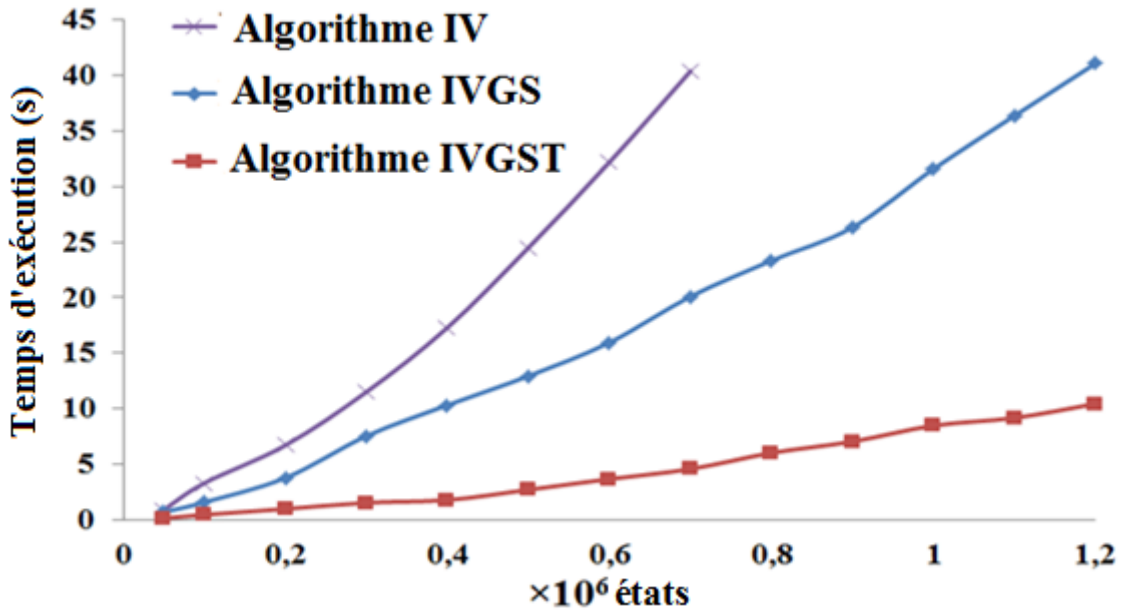


Figure 4.9 : Comparaison du temps d'exécution de l'algorithme topologique (IVGST) avec l'algorithme IV et sa variante IVGS.

Remarque 4.2: Il est possible d'appliquer la méthode hiérarchique, présentée au chapitre 3, à chaque PDM restreint. Ainsi, on peut combiner les deux types de décomposition pour accélérer la résolution du problème. Par exemple, $L_1=\{S_6,S_7\}\cup\{S_8,S_5\}$ dans la figure 4.8, alors le PDM restreint PDM1 est décomposée, lui aussi, en deux PDM restreints : PDM_{11} ($\{S_6,S_7\}$) et PDM_{12} ($\{S_8,S_5\}$).

4.6. Application à la planification de couverture de zone

La planification de couverture de zone est utilisée dans un large domaine d'applications en robotique, tels que le robot démineur [NAJ00], le robot de peinture [ATK05], le robot nettoyeur [OH04], etc. Plusieurs recherches dans ce domaine ont étés effectuées [TOK14, KAN07, GAL13]. Les algorithmes de planification de couverture de zone sont classifiés en deux catégories : les algorithmes hors ligne utilisés dans des environnements connus et les algorithmes en ligne principalement utilisés dans des environnements inconnus où la politique à suivre est mise à jour après chaque nouvelle observation de l'environnement. La planification de couverture de zone est toujours un sujet d'actualité, spécialement dans un environnement inconnu. Dans le cas du robot démineur, l'objectif est de détecter les positions des mines cachées, éviter les obstacles et chercher le plus court chemin éventuellement stochastique sans répéter des chemins. Réaliser ces objectifs n'est pas toujours facile ni possible, surtout dans un environnement inconnu. En effet, l'algorithme en ligne doit à chaque

fois trouver le plus court chemin en fonction des nouvelles données perçues de l'environnement.

Dans la suite de cette section, nous proposons un modèle de décision markovien appliqué à la couverture de zone et un algorithme en ligne appelée « couverture de zone but vers but ». Nous considérons un environnement inconnu comme étant un ensemble d'états buts, dont le coût ou le gain varie d'une petite région à une autre selon une stratégie ou un mode de balayage, tels que les modes définies par Huang [HUA01] et Ryu [RYU11]. Notre approche est inspirée du modèle transformé du plus court chemin stochastique présenté dans les sections 3 et 4 de ce chapitre.

4.6.1. Modèle de décision markovien pour la couverture de zone

Nous choisissons le modèle de décision markovien conçu pour la navigation robotique défini au chapitre 2, où l'environnement est entièrement discrétisée sous forme de grilles carrées. La taille de grille peut être choisie en fonction de la structure du robot et la taille de la zone couverte par les capteurs ou détecteurs de mines. Dans un état libre, le robot peut être contrôlé par les huit directions, plus l'action θ « rester sur place ». Dans l'état but la seule action possible est θ. Les actions qui mènent vers un obstacle peuvent être éliminées, la **figure 4.10** présente un exemple d'environnement où les actions possibles, dans chaque état, sont désignées par des flèches.

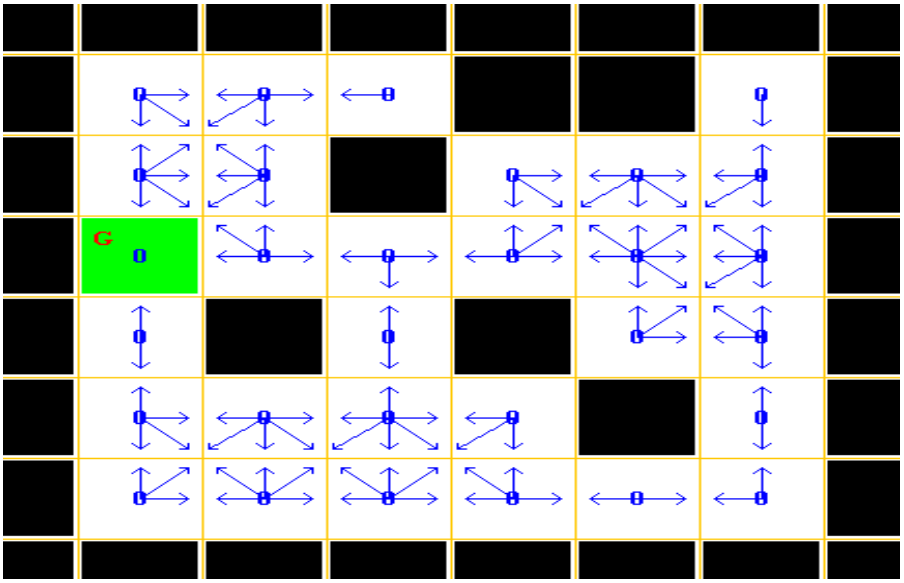


Figure 4.10 : Exemple d'environnement sous forme de grilles carrées contenant des obstacles avec les actions possibles dans chaque état libre.

Fonction de transition: Nous supposons que les transitions sont déterministes, $P_{iaj} = 1\ ou\ 0$, $\forall\, i,j \in S, \forall a \in A(i)$, le robot transite vers l'état désiré avec une probabilité égale à 1.

Fonction coût: Nous définissons le coût sous forme d'énergie consommée durant chaque transition et que nous supposons proportionnel à la distance parcourue. Ainsi, le coût est égal à une constante x ou $\frac{x}{\sqrt{2}}$ selon que l'action est horizontale ou diagonale. Le coût défini lorsqu'une transition mène vers un état but est égal à une constance notée R_b.

Capteurs de détection: Plusieurs technologies peuvent être utilisées, parmi ceux, les méthodes basées sur la détection des métaux à l'aide des oscillateurs LC et les méthodes électromagnétiques [GOO04]. Nous supposons que le robot est capable de détecter les mines cachées dans les états voisins, comme il est indiqué dans la **figure 4.11**. Un mouvement du détecteur par des moteurs ou plusieurs détecteurs peuvent couvrir la zone désirée.

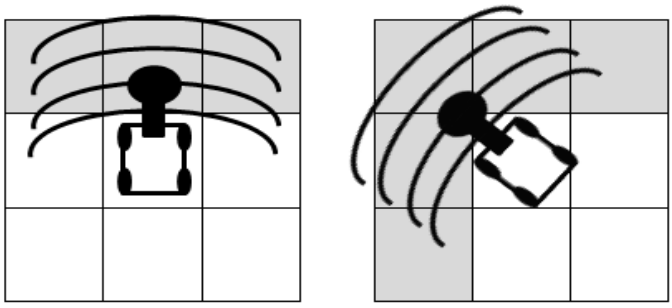


Figure 4.11 : Exemple d'états couverts par le détecteur des mines.

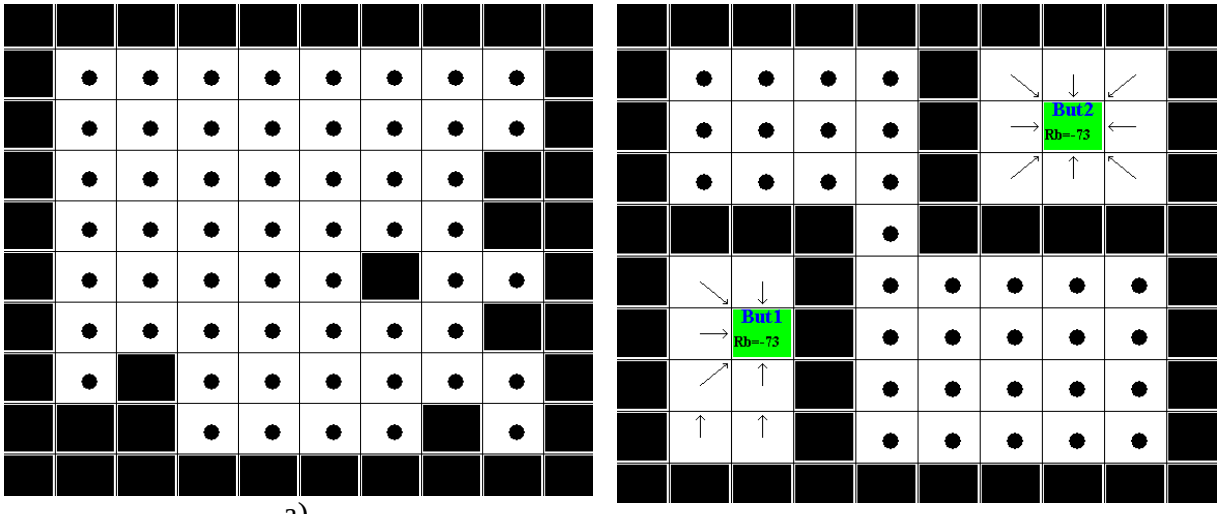
Structure du robot: Différents structures mécaniques sont utilisés dans le cas du robot démineur, telles que les structures unidirectionnels ou multidirectionnels [SUR15], etc.

4.6.2. Etude théorique du modèle

Proposition 4.8: Soit s_0 un état initial, si $x>0$ et si l'espace d'états ne contient aucun état but ou aucun chemin ne peut mener vers un état but alors $\pi^*(s_0) = \theta$

Preuve: la preuve est évidente puisque tous les coûts sont strictement positifs si $\pi(s_0) \neq \pi^*(s_0)$ alors que pour $\pi^*(s_0) = \theta$, le vecteur coût est nul.

La **figure 4.12** montre deux exemples de simulation. Dans l'environnement à gauche qui ne comporte aucun état but, une action optimale est donc $\pi^*(s_0) = \theta, \forall\, s_0 \in S$. Dans l'environnement à droite, pour les états s_0 qui ne peuvent pas mener vers un état but, une action optimale est donc $\pi^*(s_0) = \theta$ et pour les autres états, une action optimale est déterminée par le plus court chemin déterministe.



a) b)

Figure 4.12 : Exemple de stratégies optimales dans le cas de non existence d'un état but et de l'inaccessibilité à l'état but.

Proposition 4.9: Soit G un état but avec R_b=0 et *x*>0, une action optimale pour tout état initial libre s_0 est $\pi^*(s_0) = \theta$.

Preuve : Puisque pour toute stratégie $\pi^* \neq \pi_\theta$, on a : $V^*(s_0) > 0$, alors que pour $\pi^* = \pi_\theta$ on a : $V^*(s_0) = 0$.

La **figure 4.13** montre un exemple de simulation avec les paramètres *x=1 et* R_b=0. Comme on peut le constater, une action optimale pour tout état initial s_0 est $\pi^*(s_0) = \theta$.

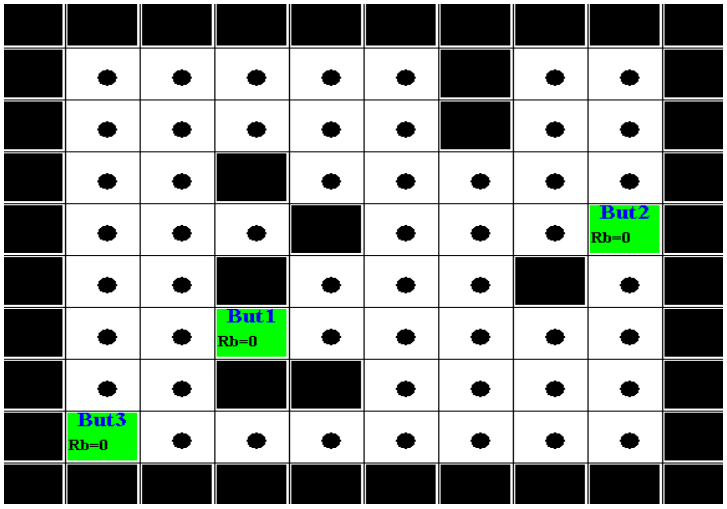


Figure 4.13 : Exemple de stratégie obtenue dans le cas d'existence d'état buts avec $\boldsymbol{R_b = 0}$.

Proposition 4.10: Soit s_0 un état initial libre et G un état but de coût $R_b < 0$. Soit S_c la somme des coûts obtenus en suivant le plus court chemin déterministe qui mène vers l'état but G. Si $S_c > -R_b$ alors une action optimale est $\pi^*(s_0) = \theta$.

Preuve : soit $\pi \neq \pi_\theta$ une stratégie qui suit le plus court chemin déterministe qui mène vers l'état but G, on a $V^*(s_0) = S_c + R_b$, le fait que $S_c > -R_b$ implique que $V^*(s_0) > 0$. Ce qui montre que $\pi(s_0)$ n'est pas une action optimale si on la compare avec $\pi^*(s_0) = \theta$ qui donne un coût minimal $V^*(s_0) = 0$.

La **figure 4.14** montre un exemple de simulation avec les paramètres x=1 et $R_b = -4$, nous remarquons que pour les chemins qui atteignent le but avec un coût supérieur à 4, une stratégie optimale est $\pi^* = \pi_\theta$.

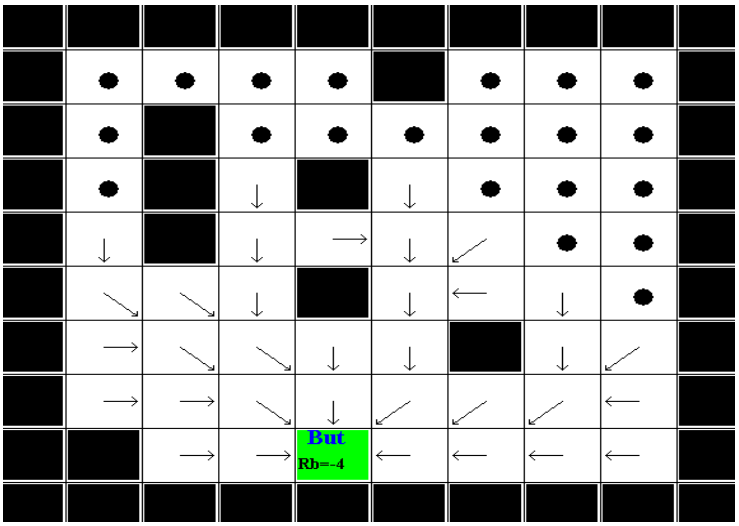


Figure 4.14 : Exemple de stratégie obtenue avec les paramètres x=1 et $\mathbf{R_b = -4}$.

Proposition 4.11: Soit n le nombre d'états libres et G un état but avec un coût $R_b = -n \times x$. Pour tout état initial libre s_0, s'il existe au moins un chemin qui mène vers l'état but, alors une stratégie optimale π^* est différente de π_θ ($\pi^*(s_0) \neq \theta$).

Preuve : Soit l la longueur du plus court chemin déterministe, tenant compte du nombre d'états libres n, on a $l < n$ et soit π la stratégie déterministe qui suit ce chemin. On a $V^{\pi}(s_0) < n \times x + R_b$, ce qui implique que $V^{\pi}(s_0) < 0$ et donc $\pi^*(s_0) \neq \theta$ puisque $V^{\pi_\theta}(s_0) = 0$.

La **figure 4.15** montre un exemple de simulation avec les paramètres x=1 et $R_b = -73$. Nous remarquons que pour tout état initial, la stratégie optimale mène vers l'état but.

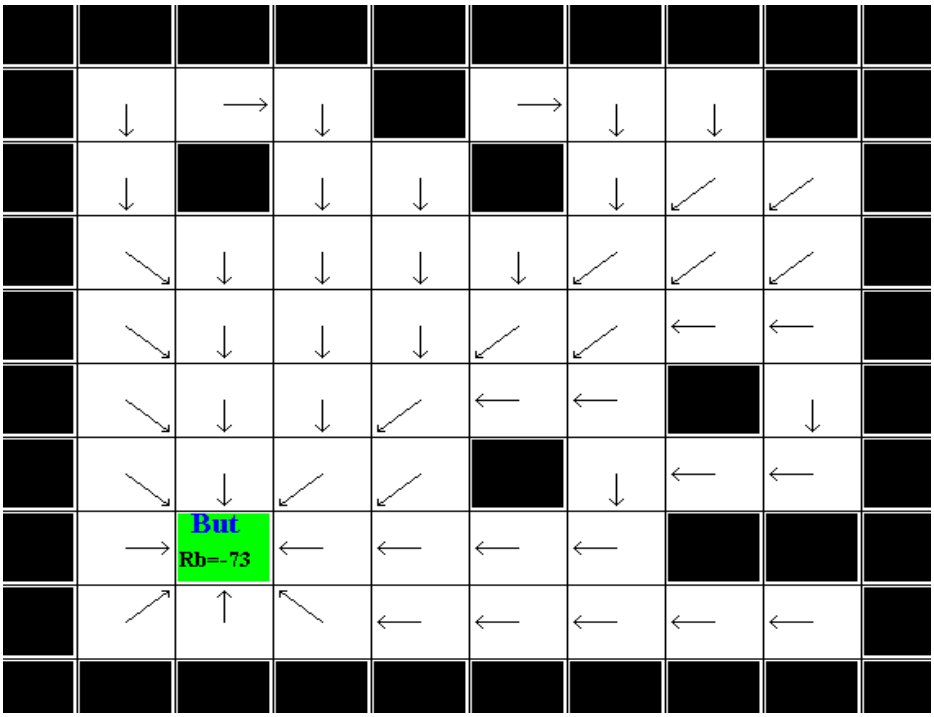


Figure 4.15 : Exemple de stratégie obtenue avec les paramètres x=1 et $\mathbf{R_b = -73}$.

Proposition 4.12 : Soient n le nombre d'états libres et G_1 (G_2) un état but de coût égal à $R_b^1 = -n \times x$ ($R_b^2 = -2 \times n \times x$), et soit s_0 un état initial libre. S'il existe un chemin qui mène vers l'état but R_b^1 et un chemin qui mène vers l'état but R_b^2, alors quel que soit la longueur de ces chemins, une stratégie optimale mène vers l'état but R_b^2.

Preuve: La preuve vient du fait que le plus long chemin contient un nombre de transitions strictement inférieur à n, donc même si l'état initial est proche de G_1, la transition vers l'état but G_2 donne le plus petit coût.

La **figure 4.16** montre un exemple de simulation dans un environnement contenant trois états buts de coûts respectifs -73, -146 et -219 avec x=1 et le nombre d'états libres est égal à 73. Nous constatons que tous les stratégies mènent vers l'état but numéro 3 ayant le coût optimal. Cette proposition permet de diriger l'agent vers un but particulier et d'assurer un déplacement but vers but, ce qui est la base de l'algorithme de couverture de zone proposé.

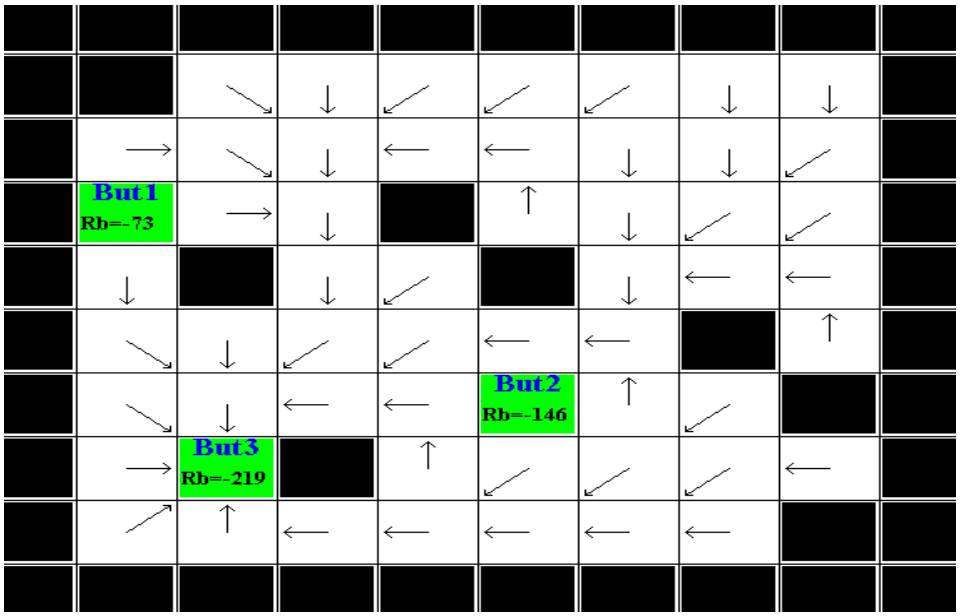


Figure 4.16 : Exemple de stratégie obtenue dans un environnement contenant trois buts et avec les paramètres x=1 et $\mathbf{R_{b1}}=-73$, $\mathbf{R_{b2}}=-146$ et $\mathbf{R_{b3}}=-219$.

4.6.3. Algorithme en ligne de couverture de zone but vers but

Une première version de l'algorithme en ligne (**Algorithme 4.3**) de couverture de zone consiste à supposer que tous les états de l'environnement comme étant des états buts de même gain $R_b = -n \times x$, où n est le nombre d'états libres. Ensuite, explorer ou observer le petit environnement qui entoure l'état courant (Ligne 2.1). Chaque zone explorée est mise à jour (ces états ne sont plus des buts) (Ligne 2.2), l'agent calcule la stratégie optimale (Ligne 2.3) et exécute l'action correspondante à sa position courante. L'algorithme se termine une fois l'action optimale détectée est égale à θ, ce qui implique que tout l'environnement est balayé sauf la zone inaccessible à partir de l'état initial.

Algorithme 4.3: Algorithme en ligne de couverture de zone but vers but (mode aléatoire)

Données: $S, P, A, C, \varepsilon; x > 0, R_b = -n \times |S|$; s_0: état initial

1. Supposer que tous les états sont des buts avec un gain égal à R_b.
2. **Répéter**
   - 2.1. Observer les états voisins avec les capteurs
   - 2.2. Mettre à jour les états observés
   - 2.3. Calculer une stratégie optimale à l'aide de l'algorithme IVGST
   - 2.4. Déplacer le robot selon une action optimale choisie

Jusqu'à ce que (l'action optimale soit égale à θ)

Théorème 4.2 : L'**algorithme 4.3** est correct et termine après avoir couvert tout l'environnement sauf la zone inaccessible à partir de l'état initial s_0.

Preuve: La preuve provient des propositions 4.8 et 4.11. En effet, au début de l'algorithme, tous les états sont supposés comme étant des buts, le robot explore ces états voisins, les mis à jour et détermine une stratégie optimale à l'aide d'un algorithme itératif (IV, IVGS ou IVGST). La proposition 4.11 assure une action optimale différent de θ tant qu'il existe un état accessible à partir de s_0 et non exploré. Une fois tous les états accessibles sont explorés, l'action optimale est égale à θ (Propositions 4.8) et l'algorithme en ligne se termine.

La **figure 4.17** montre un exemple de simulation dans un environnement généré aléatoirement. Nous remarquons que l'environnement tout entier est exploré, mais avec ce mode de recherche aléatoire, on risque d'avoir un nombre très important de répétitions des chemins, ce qui pourrait être évité en utilisant d'autres modes de recherche.

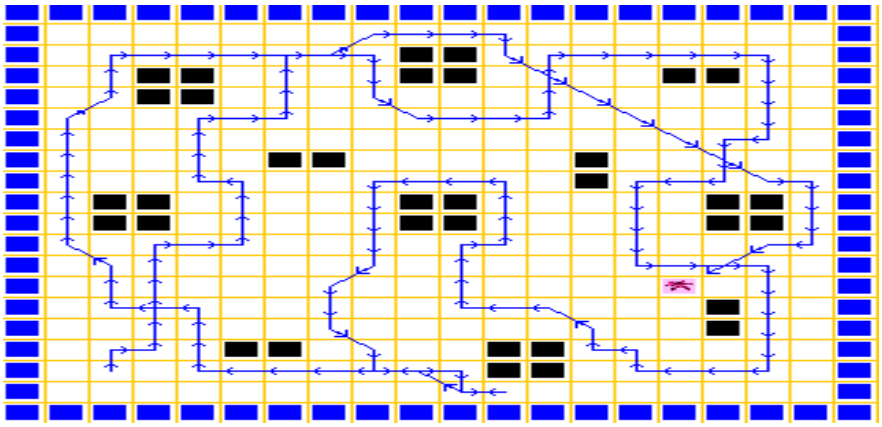


Figure 4.17 : Exemple de chemin généré en utilisant l'algorithme de couverture de zone but vers but avec un mode de balayage aléatoire.

Pour éviter la répétition des chemins, nous proposons une autre version de l'algorithme en ligne (**Algorithme 4.4**) basée sur des états buts dont les valeurs des coûts se décrémentent d'une petite région à une autre selon le mode de recherche choisi, par exemple le mode de balayage de ligne décrit dans [HUA01] ou le mode de diffusion cellulaire spatiale présenté dans [RYU11]. Dans chaque petite région (par exemple, dans la **figure 4.18**, une petite région contient 9 états), tous les états sont supposés comme étant des buts avec la même valeur du coût et le mode de recherche est assuré par la proposition 4.12.

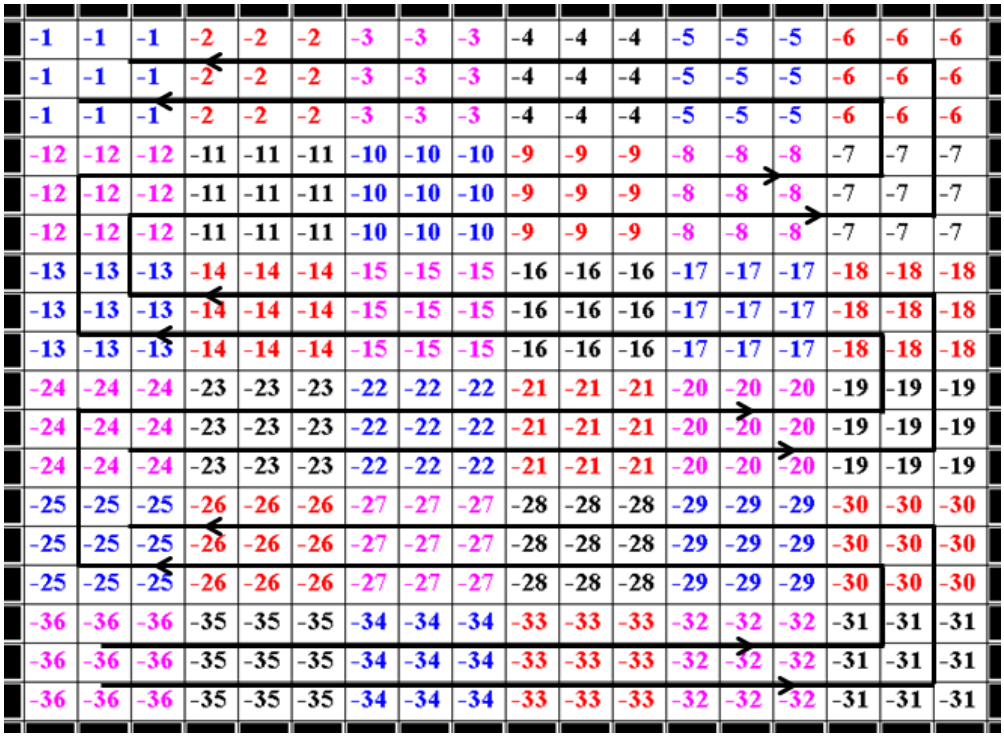


Figure 4.18 : Exemple d'environnement décomposé en états buts dont les gains décroit sous forme de balayage en lignes horizontaux.

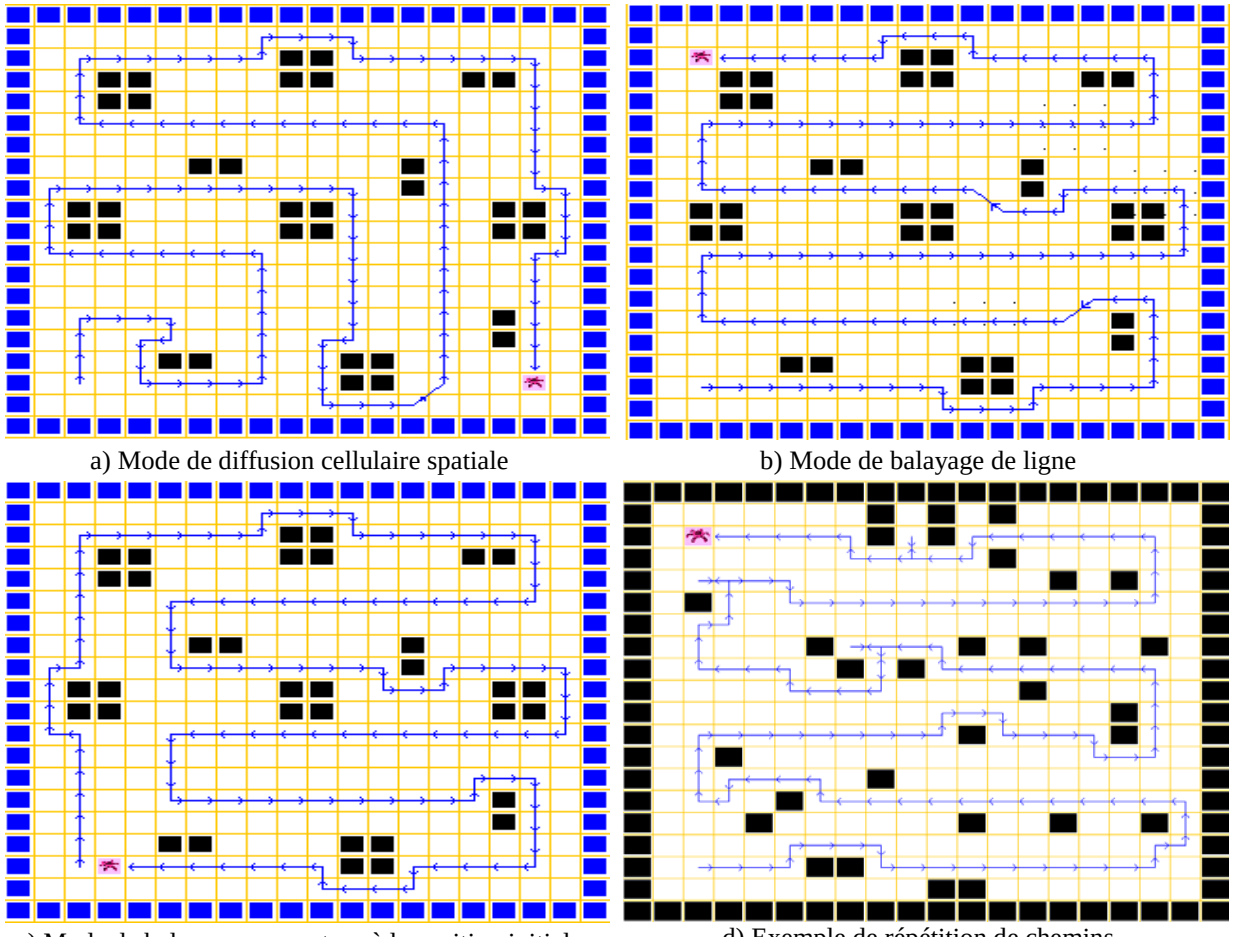
Algorithme 4.4: Algorithme en ligne de couverture de zone but vers but (avec mode de balayage).

Données: S, P, A, C, ε; $x > 0$, $R_b = -n \times |S|$, s_0: état initial

1. Décomposer l'espace d'états en p petites régions : $S_i, i = 1,..,p$.
2. Définir l'ordre d'exploration (1,2,… p) des petites régions selon le mode de recherche désiré
3. **Pour chaque** petite région $i = 1,…., p$ **Faire**
   Supposer la petite région S_i comme étant des buts avec un gain $R_b^i = -(p + 1 - i)|S|x$
4. **Répéter**
   4.1. Observer les états voisins avec les capteurs
   4.2. Mettre à jour les états observés
   4.3. Calculer une stratégie optimale à l'aide de l'algorithme IVGST
   4.4. Déplacer le robot selon une action optimale choisie

Jusqu'à ce que (l'action optimale soit égale à θ)

La **figure 4.19** montre des exemples de simulation de stratégies en ligne générés avec l'**algorithme 4.4** pour différents modes de balayage.



a) Mode de diffusion cellulaire spatiale

b) Mode de balayage de ligne

c) Mode de balayage avec retour à la position initiale

d) Exemple de répétition de chemins

Figure 4.19 : Exemples de stratégies en ligne générés avec l'algorithme 4.4 pour différents modes de balayage.

La couverture d'une zone sans répétition de chemins n'est pas toujours facile ni possible surtout dans un environnement initialement inconnu. La figure 4.19-d montre des exemples de répétition de chemins.

Remarque 4.3 : Pour un environnement réel avec un espace d'états très grand, une décomposition de l'espace d'états en petites régions peut être utilisée et dans chaque région, un mode de balayage adéquat sera choisi. Le robot explore les régions une par une, chaque région couverte sera considérée comme étant des buts avec un gain d'accès égal à 0 afin qu'il n'y aura aucun retour -vers cette région- par la suite (proposition 4.9). Dans ce cas, le temps nécessaire à chaque décision sera réduit à un temps réel puisque la complexité de l'algorithme IVGS pour un modèle déterministe est proportionnelle au nombre d'états libres.

La **figure 4.20** de gauche montre un exemple d'environnement divisé en deux parties, après avoir exploré la première région de gauche, ces états sont transformés en buts de coût d'accès égal à 0. La **figure 4.20** de droite montre le chemin généré. Notons que la valeur indiquée dans une case désigne la valeur absolue du coût R_b d'accès à l'état but.

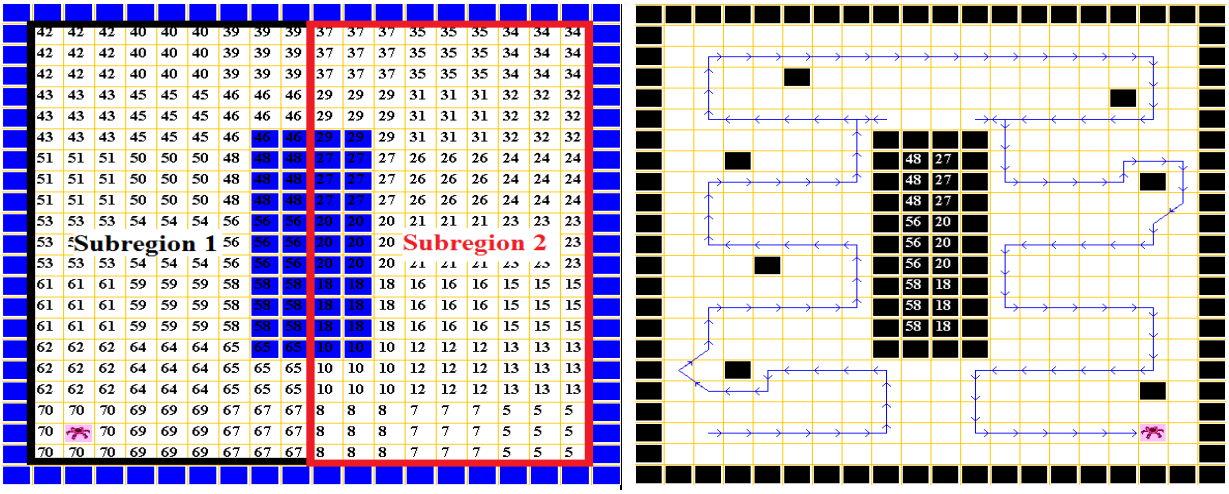


Figure 4.20 : Exemple d'environnement décomposé en deux régions et de stratégie en ligne obtenu pour une couverture de zone région par région.

Remarque 4.4: Pour minimiser la consommation d'espace mémoire dans le cas d'un espace d'état très grand, chaque sous-région peut être agrégée en un seul état avant et après son exploration, uniquement la sous-région active qui est désagrégée.

4.7. Conclusion

Dans ce chapitre, nous avons présenté, d'une part, une nouvelle transformation du modèle PDM du plus court chemin stochastique avec des impasses qui assure la convergence des algorithmes itératifs et répond au problème d'accessibilité énergétique. De plus, cette transformation permette de : 1) d'éviter, sûrement, d'atteindre les états impasses; 2) de résoudre efficacement le problème multicritères en maximisant la probabilité d'atteindre le but et minimisant l'espérance du coût ; 3) à l'agent, dans un environnement dynamique, de rester sur place jusqu'à ce sa probabilité d'atteindre le but soit égale à 1. D'autre part, nous avons présenté un nouvel algorithme de résolution (IVGST) basé sur la méthode de décomposition de l'espace d'états en niveaux d'accessibilité à l'état but.

En outre, dans l'application de planification de couverture de zone, nous avons proposé un modèle de décision markovien et un nouvel algorithme en ligne orienté but vers but, offrant la possibilité de choisir le mode de balayage adéquat. Enfin, si l'environnement est complexe, une décomposition de l'environnement en sous-régions sera nécessaire pour optimiser le chemin de couverture.

Néanmoins, ces dernières contributions sont seulement valables pour des modèles parfaitement connus. Pour traiter le problème du plus court chemin dans un environnement inconnu, le chapitre suivant présentera d'autres techniques d'apprentissage par renforcement et qui a de très nombreuses applications en robotique.

Chapitre 5

Techniques d'Apprentissage par Renforcement Appliquées à la Robotique

5.1. Introduction

Dans ce chapitre, nous considérons, tout d'abord, le problème de la navigation robotique dans un environnement inconnu et nous proposons un algorithme d'Apprentissage par Renforcement (AR) guidé, permettant de réduire le nombre d'états à explorer et ainsi d'accélérer le processus d'apprentissage. Ensuite, nous proposons quelques méthodes d'AR appliquées à la classe des PDM déterministes, de fonction de récompenses inconnue. Ce type de modèles est utilisé dans différentes applications robotique, telle que l'AR appliqué à la marche robotique. Enfin, nous exposons deux modèles d'AR, appliqué au robot suiveur de ligne et au robot auto-balancé.

5.2. Apprentissage par renforcement guidé appliqué à la navigation Robotique

Nous considérons le problème de la navigation robotique dans un environnement inconnu et représenté sous forme de grilles carrées. Nous supposons que l'état initial et l'état terminal sont connus, et que les fonctions de récompense et de transition sont inconnues. Les algorithmes d'AR, présentés au chapitre 1, convergent lentement et explorent des régions inutiles. Pour remédier à ce problème, nous proposons, dans cette section, un nouvel algorithme d'AR basé sur l'algorithme Q-learning et guidé par un modèle de décision markovien déterministe afin d'accélérer la convergence et minimiser la taille de la région à explorer. Cet algorithme utilise le même principe que le problème de couverture de zone présenté dans le chapitre précédent. Bref, une décomposition de l'espace d'états en niveaux d'accessibilité au but sera utilisée et l'environnement initialement inconnu sera supposé comme étant des buts dont les récompenses décroissent en fonction du niveau auquel appartient chaque état. L'algorithme Q-learning guidé est divisé en deux phases, une première phase d'exploration basée sur un modèle PDM déterministe connu et une deuxième phase qui utilise l'algorithme Q-learning avec une stratégie ε-gloutonne de valeur très petite ($\varepsilon < 0.1$).

5.2.1. Modèle d'exploration et décomposition en niveaux d'accessibilité

Dans cette section, nous présentons le modèle de décision markovien utilisé comme un guide d'exploration. L'environnement est entièrement discrétisé sous forme de grilles carrées régulières. Chaque état, caractérisé par sa position géographique, est associé à une valeur indiquant s'il est libre (0), obstacle (1), but (2) ou inconnu (3). L'environnement est donc représenté sous forme de matrice dont les numéros de ligne et de colonne indiquent la position géographique. Nous supposons que le robot est contrôlé par les huit directions indiquées dans la **figure 5.1**. La transition vers un état libre ou but est caractérisée par un gain proportionnelle à la distance parcourue et égal à -1 ou $-\sqrt{2}$ (**Figure 5.1**). Nous supposons

aussi que la fonction de transition est déterministe et que le robot est équipé de capteurs capables de détecter les obstacles dans les huit états qui l'entourent.

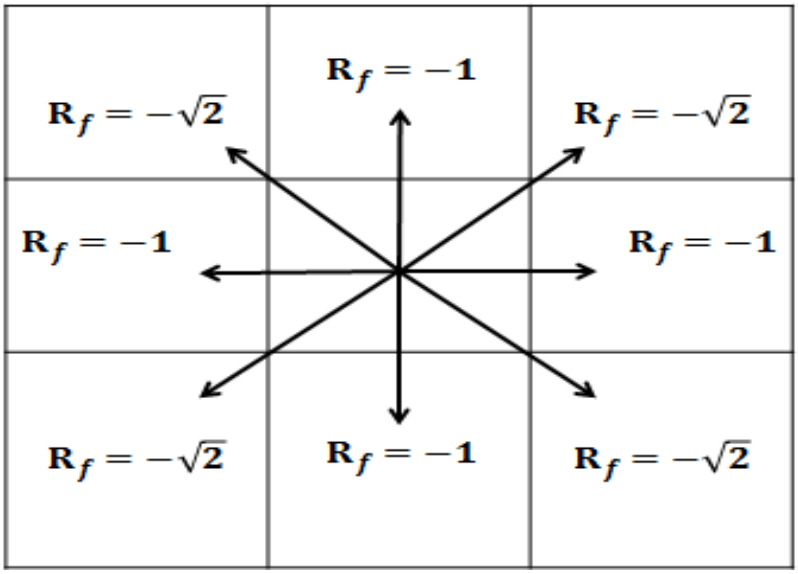


Figure 5.1: Actions et récompenses utilisées pour le modèle d'exploration.

Ensuite, l'environnement, initialement inconnu, est décomposé en niveaux d'accessibilité à l'état but, le niveau L_0 contient l'état but, le niveau L_1 contient les premiers états voisins de L_0, le niveau L_2 contient les deuxièmes états voisins de L_1, etc. Le niveau de chaque état est donné par l'équation suivante :

$$L(i,j) = \text{Max}\left(|i - n_g|, |j - m_g|\right) \quad (5.1)$$

Où (n_g, m_g) sont les coordonnées de la position de l'état but et (i,j) les cordonnées de la position de l'état inconnu. L'**algorithme 5.1** permet de définir le niveau de chaque état inconnu de l'environnement.

Algorithme 5.1: Algorithme de décomposition en niveaux d'accessibilité à l'état but.

Decomposition_en_niveaux (**Entree:** A, n, m, G(n_g, m_g) ; **Sortie**: L, matrice des niveaux)

1: **pour i** ← 0 jusqu'à **n-1 Faire**

2: **Pour j** ← 0 jusqu'à **m-1 Faire**

3: **L(i,j)=Max(|i-n_g |,|j-m_g|)**

La **figure 5.2** montre un exemple d'environnement inconnu, décomposé en niveaux d'accessibilité à l'état but.

Notons que pour faire une exploration efficace, le gain acquis lorsqu'il y a transition vers un état but de niveau *l* sera multiplié par une constante M, qui peut dépendre du type de l'environnement (M=-1 dans la **figure 5.2**). La stratégie d'exploration est donc un algorithme en ligne orienté but qui cherche à s'orienter vers le but ayant le gain maximal.

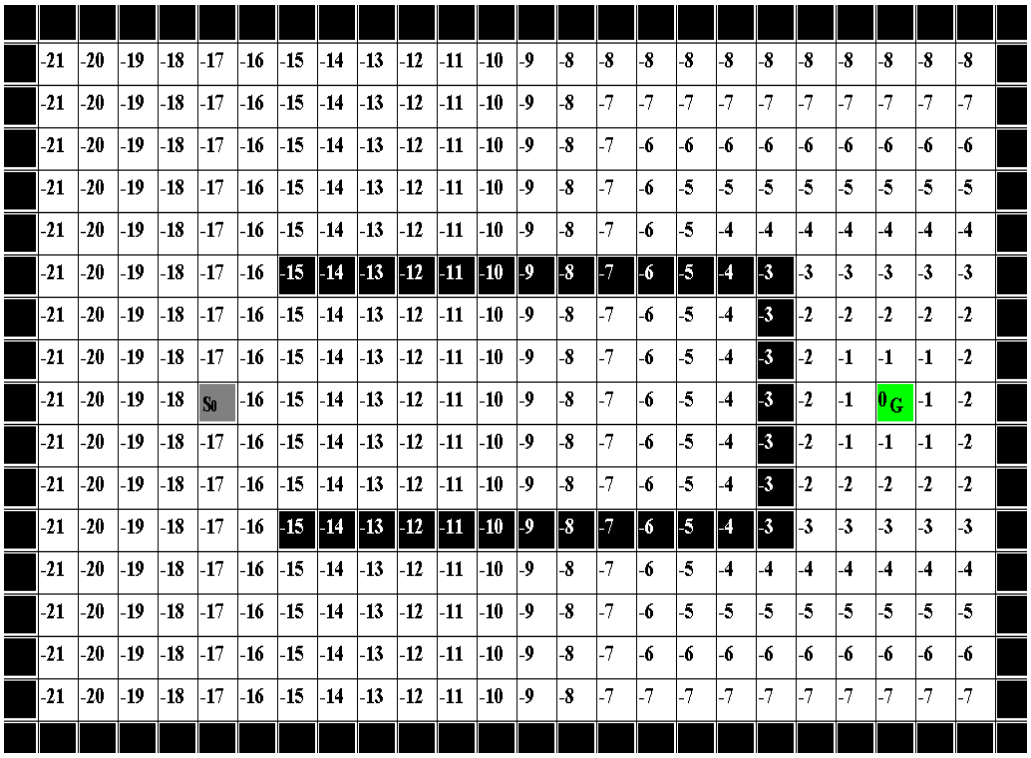


Figure 5.2 : Exemple d'environnement, initialement inconnu, décomposé en niveaux d'accessibilité à l'état but (G).

5.2.2. Algorithme d'exploration

Dans l'étape d'exploration, nous utilisons l'algorithme IVGS Déterministe (IVGSD), et la liste des successeurs de chaque couple état/action $\mathbf{\Gamma_a^+}(i) = \{ j \in S : P_{iaj} > 0\}$. L'équation de mise à jour de Bellman est appliquée uniquement aux états libres (S \G). Les transitions sont déterministes, chaque couple état/action a un seul successeur noté par $\mathbf{\Gamma_{a,i}^+}$ et le nombre d'actions par état ne peut pas dépasser huit actions. Ainsi, l'algorithme IVGSD est linéaire par itération et ne dépend que du nombre d'états libres, ce qui permet d'accélérer la décision du robot à choisir tel ou tel direction.

Algorithme 5.2: Algorithme IVGSD.

IVGSD (Entrée: S,G,A,R, ε; Sortie: π^*, V^*)

1: **Pour chaque** i ∈ S **Faire** $V^*(i) \leftarrow 0$;
2: Convergence← *Faux*;*//paramètre de convergence*
3: **Tant que (**Convergence= *Faux* **) Faire**
4: Convergence ←*Vraie*;
5: **Pour chaque** i ∈ S \G **Faire**
6: $V_{tmp} \leftarrow \max_{a \in A(i)} \{ r(i,a) + V^*(\mathbf{\Gamma_{a,i}^+}) \}$
7: **Si** $(V_{tmp} - V^*(i) \geq \varepsilon$ **)** Convergence ← *Faux*
8: $V^*(i) \leftarrow V_{tmp}$
9: **Pour chaque** i ∈ S \G **Faire**
10: $\pi^*(i) \leftarrow \underset{a \in A(S_0)}{argmax} \{ r(i,a) + V^*(\mathbf{\Gamma_{a,i}^+}) \}$
11: **Retourner** (π^*, V^*)

5.2.3. L'algorithme Q-learning guidé

L'algorithme Q-learning guidé (**Algorithme 5.3**) est divisé en deux phases : (1) une phase d'exploration composée d'un nombre très petit d'épisodes, l'action d'exploration est choisie en fonction de la stratégie optimale calculée par l'algorithme en ligne : **Algorithme 5.2**. (2) une deuxième phase d'exploitation/exploration avec une stratégie ε-gloutonne ($\varepsilon < 0.1$).

Algorithme 5.3: Première phase de l'algorithme Q-learning guidé

Premieres_episodes (S, S_0, G, Ne, F_d)
1: L ← **Decomposition_en_niveaux** (S, n, m, G(n_g, m_g))
2: **Pour** n←1 jusqu'à *Ne* **Faire** *// Ne : nombre d'épisodes de la première phase*
3: t ← 0; $S_t \leftarrow S_0$
5: **Tant que** ($S_t \neq G$) **Faire** *//Tant que l'état but n'est pas atteint*
6: Observer les états voisins
7: Mettre à jour les états observés
8: Eliminer les actions impossibles *//Eliminer les actions qui mènent vers un obstacle*
9: (π^*, V^*) ← **IVGSD (**S, G, A, R, ε) *//calculer une stratégie optimale (**Algorithme 5.2**)*
10: Exécuter une action optimale $\pi^*(S_i)$
11: Observer la récompense r_t
12: S_{t+1}= état suivant observé
13: Mettre à jour les Q-valeurs en utilisant l'équation (1.45)
14: t ← t + 1;
15: **Eliminer les états non explorés**
16: **Eliminer_etats** (S, s_0, V^*, F_d) //Procedure 5.1

Dans la première phase (**Algorithme 5.3**), un nombre très petit d'épisodes (Ne<10) est choisi. Avant chaque décision, le robot met à jour les états observés (Lignes 6-7) puis élimine les actions impossibles (Ligne 8). Ensuite, l'**algorithme 5.2** est utilisé pour déterminer l'action d'exploration (Ligne 9), le robot exécute l'action choisie (Ligne 10) puis observe la récompense et l'état suivant (Lignes 11-12) et met à jour le Q-valeur de l'état courant (Ligne 13). Chaque épisode est terminée une fois l'état but est atteinte. Une procédure d'élimination des états non explorés est utilisée (Lignes 15). En outre, en tenant compte des valeurs optimales calculées par le modèle connu, chaque état ayant une valeur optimale inférieure à celle de l'état initial à une différence par rapport à un facteur F_d, choisi rigoureusement, est aussi éliminé (Ligne 16, Procédure 5.1). Notons que pour une raison de simplicité, un état éliminé sera considéré comme un obstacle.

Procedure 5.1: Elimination des états explorés ne pouvant pas être dans le chemin optimal

Eliminer_etats (S, s_0, V^*, F_d)
1: **Pour chaque** état visité $S_i \in S$ **Faire**
2: **Si** ($V^*(S_i) < V^*(S_0) - Fd$) **Alors**
3: Elimine l'état s_i *//S(s_i)=1, état considéré comme un obstacle*

La **figure 5.3** montre un exemple de simulation de la première phase de l'algorithme proposé. Nous avons choisi un environnement contenant une barrière rectangulaire. Durant une première phase de quatre épisodes, les états non explorés (cases avec point d'interrogation ?) sont éliminés et les états explorés avec une valeur de gain ($V^*(S_i) < -24.9$) inférieur à celui de l'état initial S_0 ($V^*(S_0) = -22.9$, $F_d=2$) sont aussi éliminés. Ainsi, la taille de l'espace d'état à exploiter/explorer est réduit durant la deuxième phase.

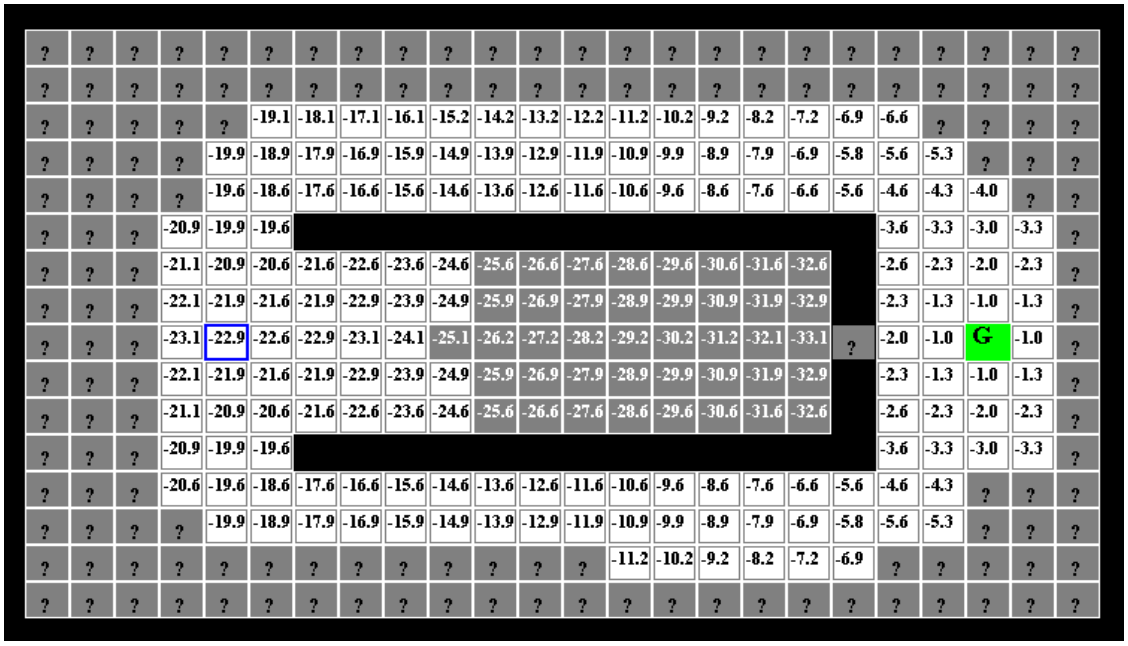


Figure 5.3 : Exemple d'états explorés (cellules en noire et en bleu ciel) et éliminés durant la première phase de l'algorithme Q-learning guidé, dans un environnement contenant une barrière rectangulaire.

Après la phase d'exploration, une deuxième phase d'exploitation et d'exploration avec une politique ε-gloutonne ($\varepsilon < 0.1$) est ensuite utilisée. L'algorithme Q-learning guidé sera présenté comme suit:

Algorithme 5.4: Algorithme Q-learning Guidé (QLG)

QLG (**S**, S_0, g, n)
1: Décomposer S en niveaux selon l'**algorithme 5.1**;
2: **Premieres_episodes** (S, S_0, G, Ne, F_d) *//Première phase d'exploration*
3: **Pour chaque** épisode **Faire**
4: $t \leftarrow 0$; $S_t \leftarrow S_0$;
5: **Tant que** ($S_t \neq G$) **Faire**
6: Observer les états voisins
7: Mettre à jour les états observés
8: Eliminer les actions impossibles
9: Choisir une action ε-gloutonne : $\pi^*(S_t)$
10: Exécuter l'action optimale $\pi^*(S_t)$
11: Observer la récompense r_t
12: $S_{t+1} \leftarrow$ état suivant observé
13: Mettre à jour les Q-valeurs en utilisant l'équation (1.45)
14: $t \leftarrow t + 1$;

Après la décomposition de l'espace d'états en niveaux d'accessibilité à l'état but, nous supposons que les états inconnus sont des buts avec des gains dépendants des valeurs de ces niveaux (ligne 1). Ensuite, la première phase d'exploration est appliquée (ligne 2), le modèle PDM déterministe guidera le robot pour explorer la zone nécessaire à la convergence et ainsi éliminer la zone non explorée et une partie de celle explorée qui ne peut pas être dans le chemin optimale. Enfin, l'algorithme Q-learning est appliqué avec une politique ε-gloutonne ($\varepsilon < 0.1$) (lignes 3-13).

La **figure 5.4** montre le chemin optimal généré pour l'environnement de la **figure 5.3**, nous avons choisi quatre épisodes dans la première phase et six épisodes dans la deuxième phase.

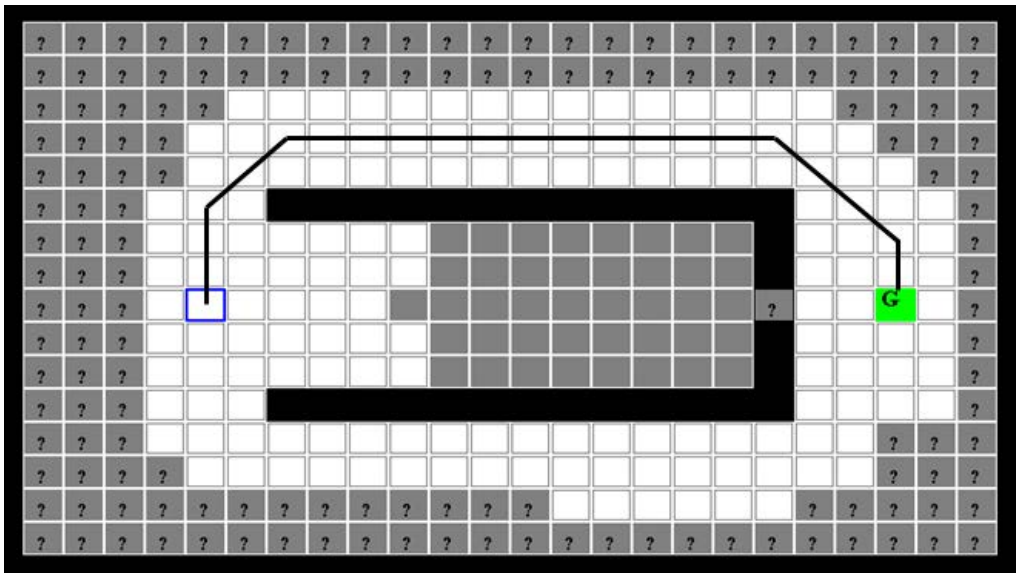


Figure 5.4 : Chemin généré dans l'environnement de la figure 5.3.

5.2.4. Résultats de simulation

L'algorithme proposé est simulé dans différents types d'environnement en utilisant le langage java-applet, Intel(R) Core(TM)2 Duo process (2.6 GHz). La **figure 5.5** montre les quatre environnements que nous avons utilisé pour étudier la performance de l'algorithme proposé : simple barrière (E1) ; environnement complexe (maze) (E2) ; environnement aléatoire (E3) et environnement avec multiples barrières (E4).

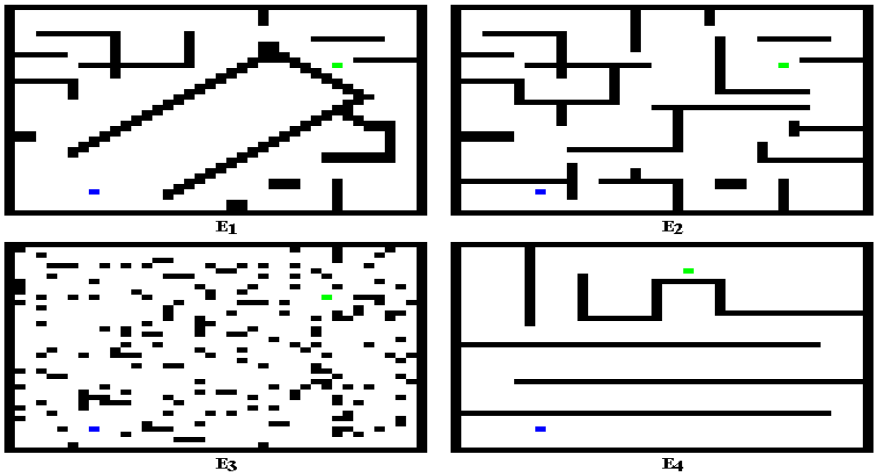


Figure 5.5 : Les quatre environnements utilisés pour étudier la performance de l'algorithme QLG.

La **figure 5.6** montre les zones éliminées durant les quatre premières épisodes de la première phase d'exploration (la zone en rouge est explorée mais éliminée et la zone en gris est non explorée et éliminée), uniquement la zone en blanc qui sera utilisée dans la deuxième phase d'apprentissage. Nous remarquons que, dans la plupart des cas (Environnements E_1, E_2 et E_3), la procédure d'élimination des états utilisée dans l'algorithme Q-learning guidé est très bénéfique puisqu'elle réduit énormément l'espace d'états.

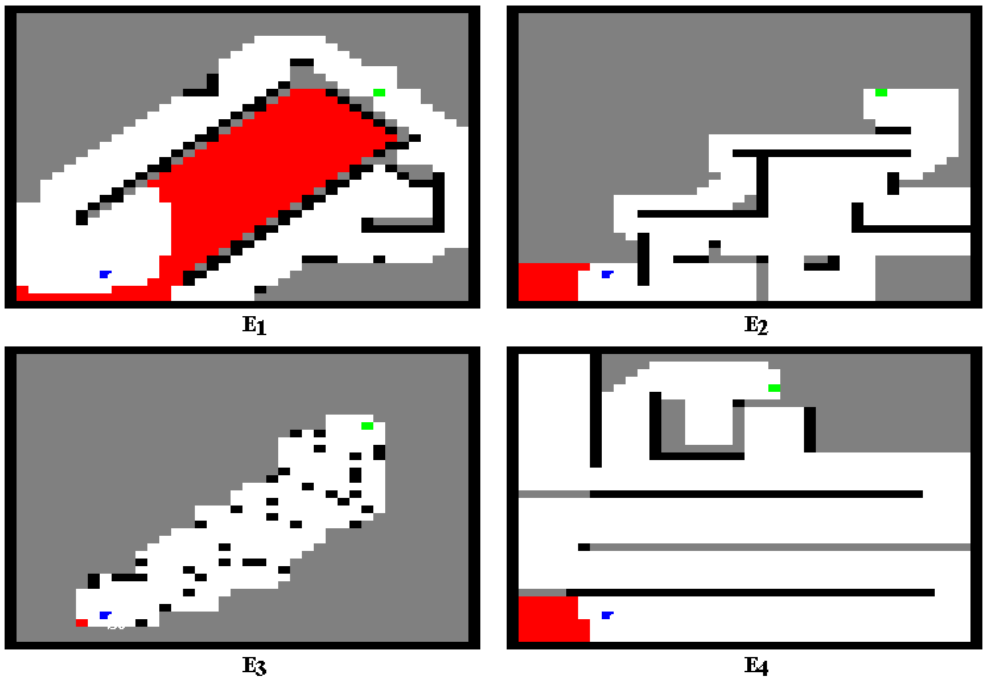


Figure 5.6 : Zones éliminées (explorées en rouge, non explorées en gris) après quatre épisodes de la première phase d'exploration.

La **figure 5.7** montre les chemins optimaux générés par l'algorithme Q-learning guidé pour les différents environnements.

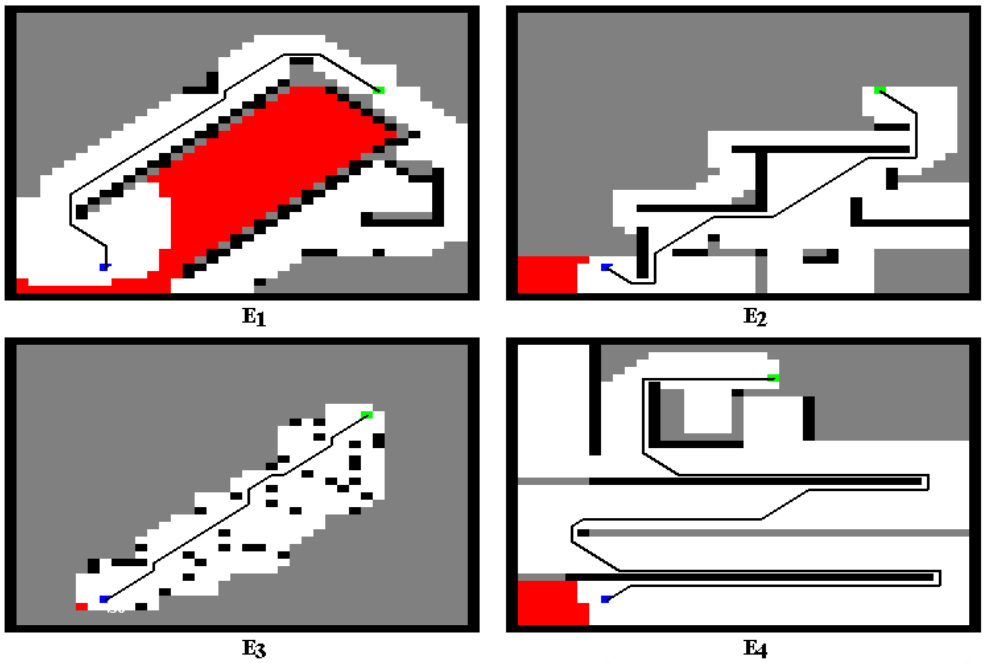


Figure 5.7 : Chemins générés après quatre épisodes de la 1ère phase et six épisodes de la 2ème phase.

La **figure 5.8** montre l'évolution du processus d'apprentissage appliqué aux quatre environnements présentés précédemment en comparant l'algorithme Q-learning guidé avec les deux algorithmes Q-learning et Q(λ).

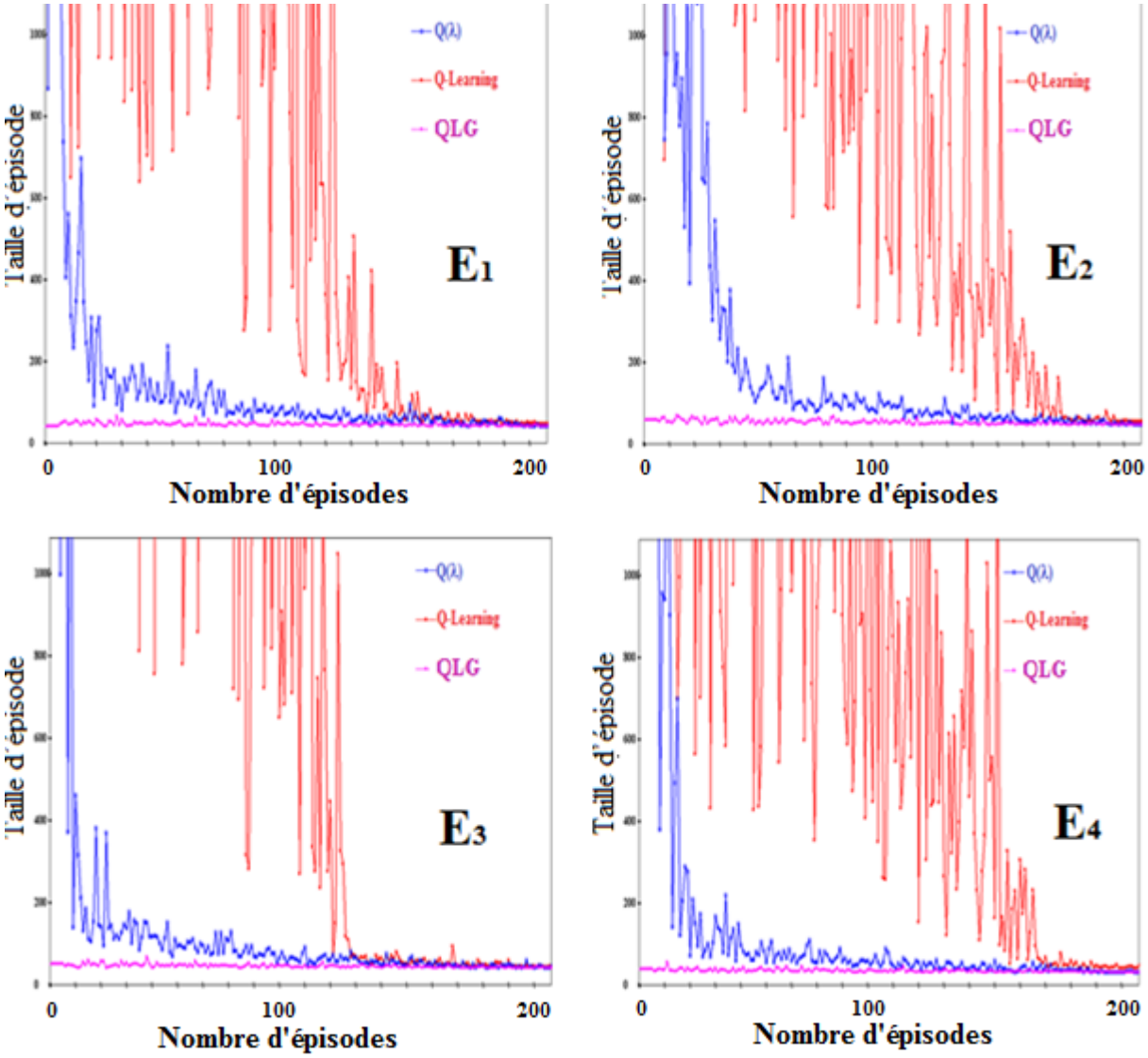


Figure 5.8 : Evolution du processus d'apprentissage dans les quatre environnements pour les algorithmes Q-learning, Q(λ) et Q-learning guidé (QLG).

Nous constatons que l'algorithme Q(λ) converge plus vite que Q-learning en terme du nombre d'épisodes nécessaires avant la convergence, ce qui est évident puisque l'algorithme Q(λ) assure une propagation rapide des récompenses et met à jour tous les couples état/action durant chaque itération. Nous remarquons aussi que l'algorithme **QLG** proposé est plus rapide que ces deux approches pour tous les environnements (E_1, E_2, E_3 et E_4). Si on considère la taille de chaque épisode, nous remarquons aussi que l'algorithme proposé est le plus performant puisque le nombre de décisions dans chaque épisode est le plus petit par rapport aux deux autres algorithmes.

D'ailleurs, pour montrer l'avantage de cette procédure d'élimination d'états lors de la première phase de l'algorithme proposé, nous avons comparé le processus d'apprentissage avec et sans élimination d'états. La **figure 5.9** montre les résultats obtenus, nous remarquons

que le processus d'élimination des états réduit clairement le nombre de décisions dans chaque épisode à l'exception de l'environnement E_4 où la zone éliminée est très petite. En effet, l'élimination en ligne des états permet d'éviter l'exploration des zones inutiles, ce qui réduit la taille des épisodes et accélère le processus de convergence.

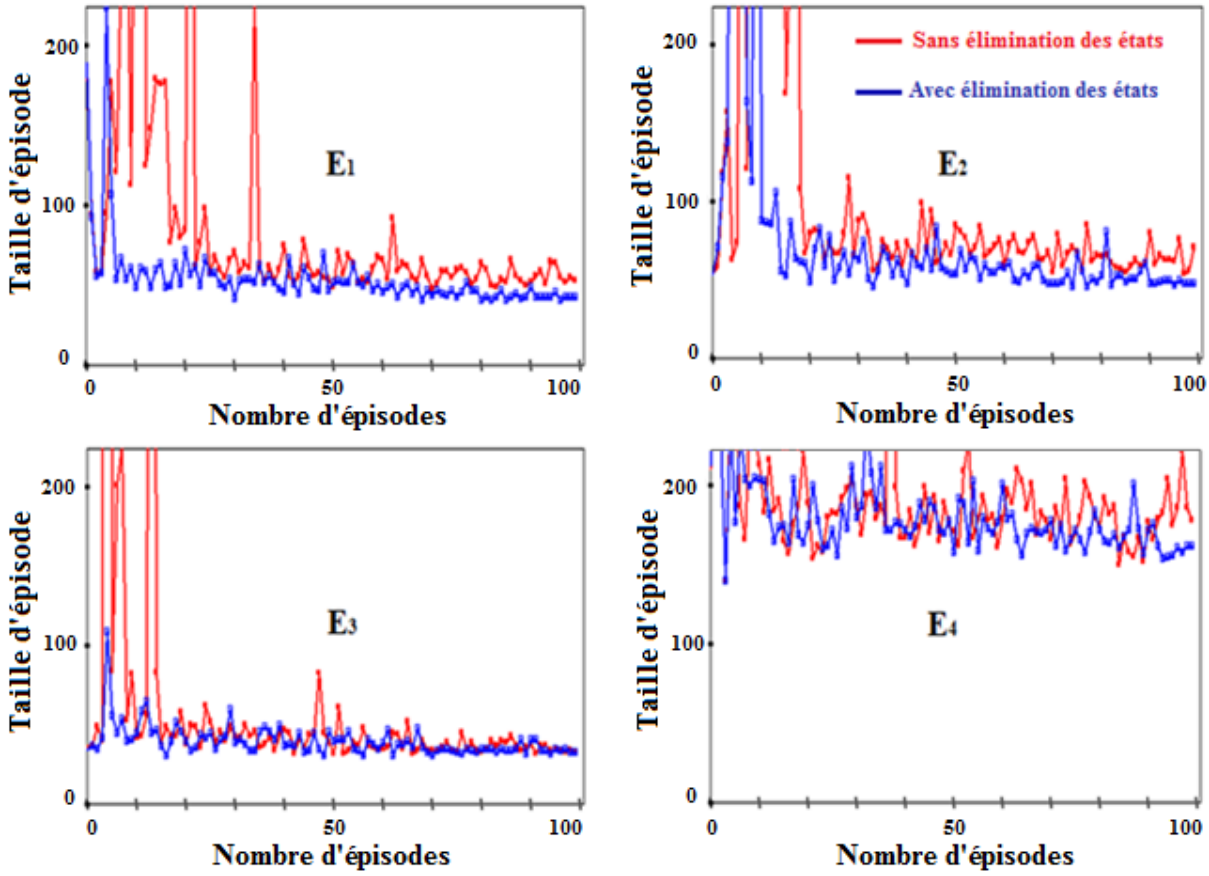


Figure 5.9 : Comparaison du processus d'apprentissage avec et sans technique de réduction de l'espace d'états.

5.3. Méthodes d'AR appliqué à la marche Robotique

Considérons un robot de plateforme à pattes, dans l'exemple le plus simple de l'escargot mécanique, qui doit marcher en avant et en équilibre, en fonction des actionneurs « généralement des moteurs pas à pas » qui commandent ces pattes. Il doit tenir son équilibre et avancer d'une distance positive, des capteurs lui permettent de mesurer cette distance ainsi que son équilibre. Les techniques d'AR lui permettent d'apprendre à marcher par expérience, par essais et erreurs, jusqu'à atteindre la séquence optimale, cela peut dépendre du type de l'environnement. Cette séquence optimale doit le tenir en équilibre et lui permettre d'avancer sans reculer, ainsi une action qui le recule ou qui le fait tomber par terre ne peut appartenir à cette séquence optimale. En se basant sur cette idée, nous considérons la classe des PDM définie par le tuple (S, A, T, P, R) sous les conditions suivantes :

- Les deux espaces S et A sont finis et discrets
- La fonction de transition P est déterministe et connue.
- La fonction de récompense R est a priori inconnue.

Notre objectif est ainsi de trouver une séquence d'actions qui maximise la somme des gains à long terme sous les deux contraintes suivantes : $r(i,\pi^*(i)) \geq R_{inf}$ et $f(i,\pi^*(i)) \in [c,d]$, où R_{inf} est une constante donnée et f une fonction quelconque qui doit appartenir à l'intervalle $[c,d]$ donné.

5.3.1. Méthode indirecte d'AR par la couverture de zone

Dans cette section, nous proposons une première méthode indirecte qui consiste à déterminer par expérience la fonction de récompenses inconnue en utilisant la méthode de couverture de zone proposée au chapitre 4. En effet, il faut explorer tous les couples état/action (i,a) afin de déduire la fonction R. Pour cela, nous définissons le modèle de couverture de zone comme suit :

- **Espace d'états :** $S' = S \times A = \{(i,a): i \in S \; et \; a \in A(i)\}$.
- **Espace d'actions :** A' défini par : $\forall (i,a) \in S', A'\big((i,a)\big) = \{b \in A(j): P(j|i,a) = 1\}$.
- **Fonction de transition :** P' définie par : $P'\big((j,b)\big|(i,a),b\big) = 1$ si $P(j|i,a) = 1$.
- **Fonction de récompenses :** R' définie par : $R' \; \big((i,a),b\big) = 1, \forall (i,a) \in S' \, , \forall \; b \in A'\big((i,a)\big)$.

Partant d'un état initial (i_0,a_0), il faut visiter tous les états $(i,a) \in S'$. Nous appliquons l'un des algorithmes de couverture de zone proposés au chapitre 4 (**Algorithme 4.3** ou **Algorithme 4.4**). L'**algorithme 5.5** présente la méthode directe d'AR par la couverture de zone.

Algorithme 5.5: Méthode Indirecte d'Apprentissage par Couverture de Zone (MIACV).

MIACV (S,A,P,R)
1: Construire le modèle PDM (S',A',P',R') de couverture de zone
2: Appliquer l'**algorithme 4.4** ou **4.5** au modèle (S',A',P',R') en observant les récompenses inconnues $r(i,a), \forall \; i \in S, \forall \; a \in A(i)$.
3: Appliquer l'algorithme IVGSA (**Algorithme 3.2**) au modèle (S,A,P,R).

Certes, l'avantage de cette méthode est qu'elle permet d'accélérer le processus d'exploration mais elle présente l'inconvénient du grand espace mémoire nécessaire pour définir le nouveau modèle de couverture de zone. Elle présente aussi l'inconvénient des deux contraintes à considérer lors de la résolution du modèle PDM à l'étape 3. Cette méthode est plus efficace pour un espace d'états et d'actions très petit. En effet, pour l'exemple de l'apprentissage au robot escargot présenté au chapitre 2, section 3, cette méthode directe nécessite 426 itérations sur 420 états possibles alors que l'algorithme indirect nécessite plus de 1000 itérations pour converger.

5.3.2. Méthode d'AR par élimination en ligne des actions non optimales

En littérature, parmi les techniques utilisées pour accélérer le processus de convergence des algorithmes d'AR, la méthode d'élimination en ligne des actions non optimales. Ainsi, en tenant compte de la contrainte qu'aucune action optimale ne peut générer un gain strictement inférieur à R_{inf}, une action a est éliminée si $r(i,a) < R_{inf}$ ou $f(i,a) \notin [c,d]$. Mais, le

processus n'est pas aussi simple du fait qu'il se peut, à un instant t, qu'on se retrouve dans un état où toutes ses actions possibles sont éliminées. De plus, une action qui mène vers cet état doit être aussi éliminée. Ce qui rend le processus plus compliqué que prévu. Comme solution, nous proposons l'**algorithme 5.6** qui utilise un espace d'actions temporaire A' égal à A au début du processus d'apprentissage et l'élimination des actions se fait dans A', on aura donc la possibilité de choisir une action de l'état i dans l'ensemble $A(i)$ si $A'(i)$ est vide.

Algorithme 5.6: Algorithme d'AR par élimination en ligne des actions.

1: Initialiser V_0
2: $A' \leftarrow A$; *//Espace d'actions temporaire*
3: Pour tout $s \in S, a \in A(s)$ **Faire**
    $R(s,a) \leftarrow \mathbf{0}$ **;** *//Initialisation des récompenses inconnues*
4: $\boldsymbol{s} \leftarrow s_0$
6 : Tant que (non fin d'apprentissage) **Faire**
7: **Si** $(A'(s) \neq \emptyset)$ **Alors** $a \leftarrow$ **ϵ-gloutonne**$\big(A'(s)\big)$
8: **Si** $(A'(s) = \emptyset)$ **Alors** *choisir* $\boldsymbol{a} \in A(s)$ qui mène vers s' tel que $A'(s') \neq \emptyset$
9: Exécuter l'action a choisie
10: Observer l'état suivant $\mathbf{s}'$ et le gain $r(s,a)$
11: Mettre à jour le gain observé
12: **Si** ($(r(s,a) < R_{\text{inf}})$ ou $f(s,a) \notin [c,d]$ ou $(A'(s') = \emptyset)$) **Alors**
        $A'(s) = A'(s) \backslash \{a\}$ *//Eliminer l'action a de A'*
13: **Pour** tout $s \in S$ **faire**
        $$V(s) \leftarrow \max_{a \in A(s)} \left\{ r(s,a) + \gamma \sum_{s'} V(s') \right\}$$
        $$\pi(s) \in \operatorname{argmax}_{a \in A(s)} \left\{ r(s,a) + \gamma \sum_{s'} V(s') \right\}$$
14: $\boldsymbol{s} \leftarrow \mathbf{s}'$
15: Retourner V, π

Finalement, le choix d'une action est ϵ-gloutonne si $A'(s) \neq \emptyset$ (Ligne 7, **Algorithme 5.6**), sinon on choisit une action qui mène vers un état s' tel que $A'(s') \neq \emptyset$ (Ligne 8, **Algorithme 5.6**). Si on observe un gain $r(s, \boldsymbol{a}) < R_{\text{inf}}$ ou si l'action mène vers un état s' telle que $A'(s') = \emptyset$ (Ligne 11, **Algorithme 5.6**) alors on élimine l'action $\boldsymbol{a}$ (Ligne 12, **Algorithme 5.6**).

5.3.3. Exemple de simulation

Nous avons procédé à une simulation de l'**algorithme 5.6** et nous l'avons comparé avec l'**algorithme 2.2** sans technique d'élimination des actions. La simulation a été effectuée avec des modèles générés aléatoirement sous forme de grilles carrées. Nous avons choisi des gains positifs et d'autres négatifs avec R_{inf}=0. Nous avons tracé la différence $\Delta V = \sum_{i \in S} |V^*(i) - V_t(i)|$ en fonction de *t*, où *t* désigne le numéro d'itération, $V^*(i)$ la valeur optimale de i et

$V_t(i)$ la valeur estimée à l'itération *t*. La **figure 5.10** montre la différence du processus d'apprentissage entre ces deux méthodes pour une stratégie ϵ-gloutonne fixe (ϵ=0.5).

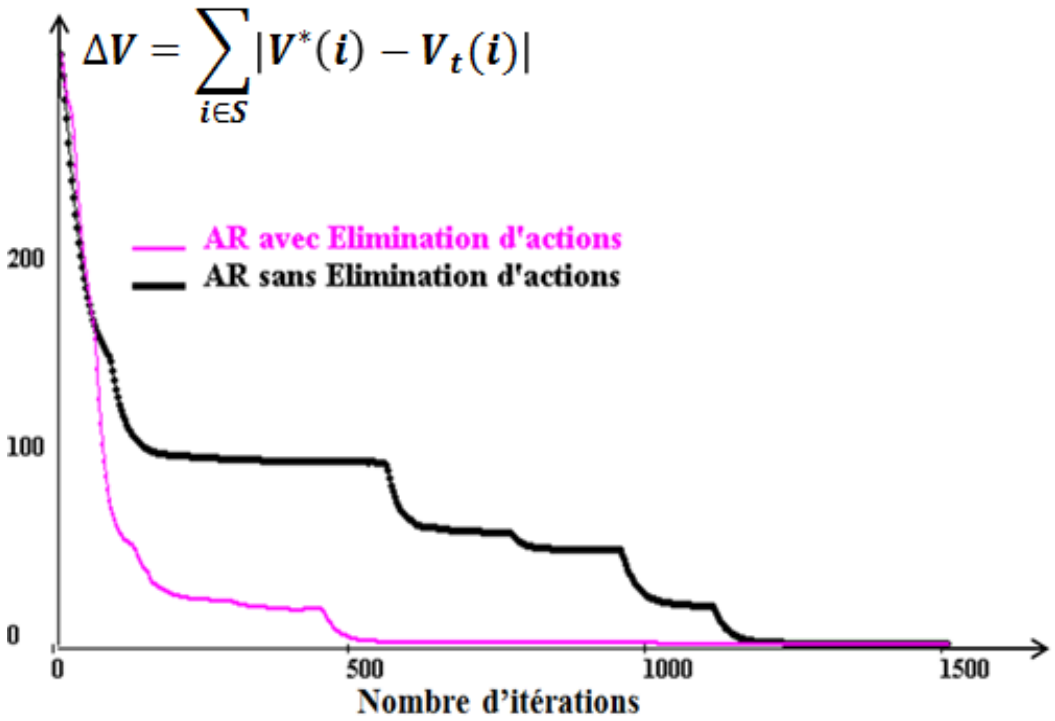


Figure 5.10 : Processus d'apprentissage des deux algorithmes avec et sans élimination des actions pour l'exemple simulé d'escargot mécanique (stratégie $\boldsymbol{\epsilon}$-gloutonne fixe $\boldsymbol{\epsilon}$=0.5).

La **figure 5.11** montrent la différence du processus d'apprentissage entre ces deux méthodes pour une stratégie ϵ-gloutonne qui tend vers 0 au cours des itérations ($\epsilon = 1 \rightarrow 0$).

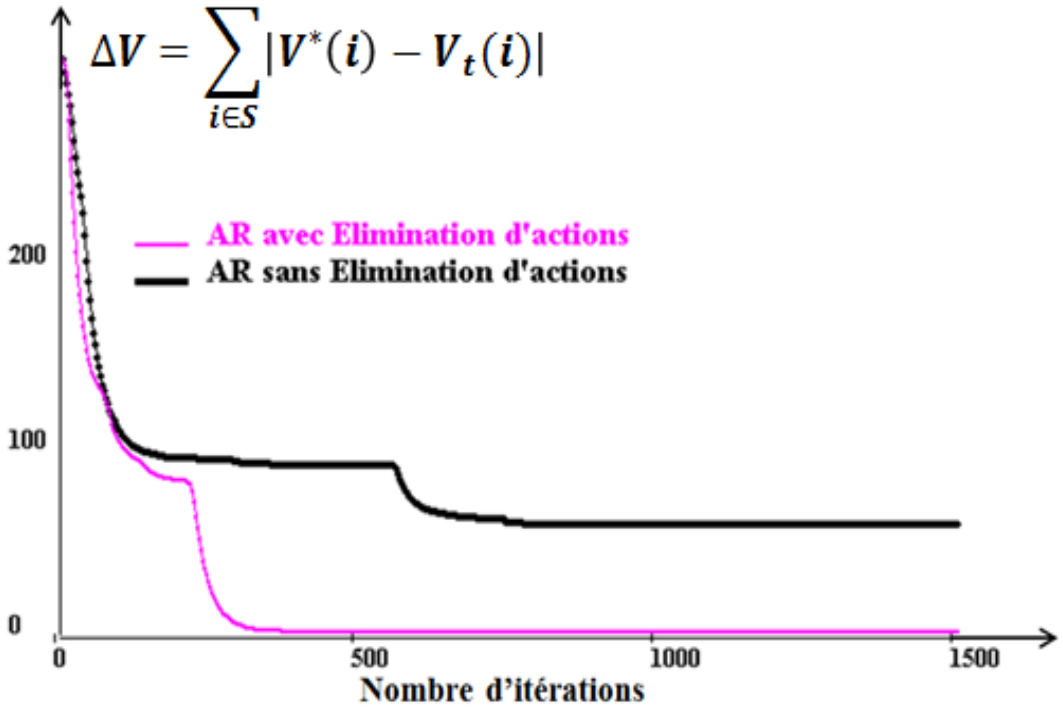


Figure 5.11 : Processus d'apprentissage des deux algorithmes avec et sans élimination des actions (stratégie $\boldsymbol{\epsilon}$-gloutonne qui tend vers 0 au cours des itérations).

Nous remarquons que le processus d'apprentissage converge plus vite avec cette nouvelle technique d'élimination des actions, ce qui est évident puisqu'on évite d'explorer à nouveau, des couples état/action qui n'entrent pas dans une stratégie optimale et le processus d'apprentissage converge plus vite que sans technique d'élimination en ligne des actions non optimales. En effet, pour une stratégie ϵ-gloutonne fixe (ϵ=0.5), l'**algorithme 5.6** converge après 500 itérations alors que l'**algorithme 2.2** a nécessité plus que 1000 itérations pour converger vers une solution optimale. Nous remarquons aussi que, pour une stratégie ϵ-gloutonne qui tend vers 0 ($\epsilon = 1 \rightarrow 0$) avec un nombre d'itérations fixe égal à 1500, l'algorithme proposé a nécessité moins de 500 itérations pour converger, alors que celui sans technique d'élimination a convergé vers une solution non optimale.

5.3.4. Expérimentation réelle

Afin de tester et comparer ces deux algorithmes d'apprentissage, nous avons réalisé un escargot mécanique (**Figure 5.12**) composé de deux roues libres et de deux bras qui sont commandés chacun par un servomoteur à rotation de 180°. L'escargot mécanique est composé aussi d'un capteur ultrason qui mesure la distance de déplacement relative à la position courante et d'un gyroscope qui mesure l'angle de la plateforme avec l'horizontale. La première contrainte est qu'une action optimale doit déplacer le robot d'une distance supérieure ou égale à zéro, ainsi $R_{inf} = 0$. La deuxième contrainte est que l'angle que fait la plateforme avec l'horizontale ne doit pas dépasser 5°, ainsi $f(i, \pi^*(i)) \in [0°, 5°]$.

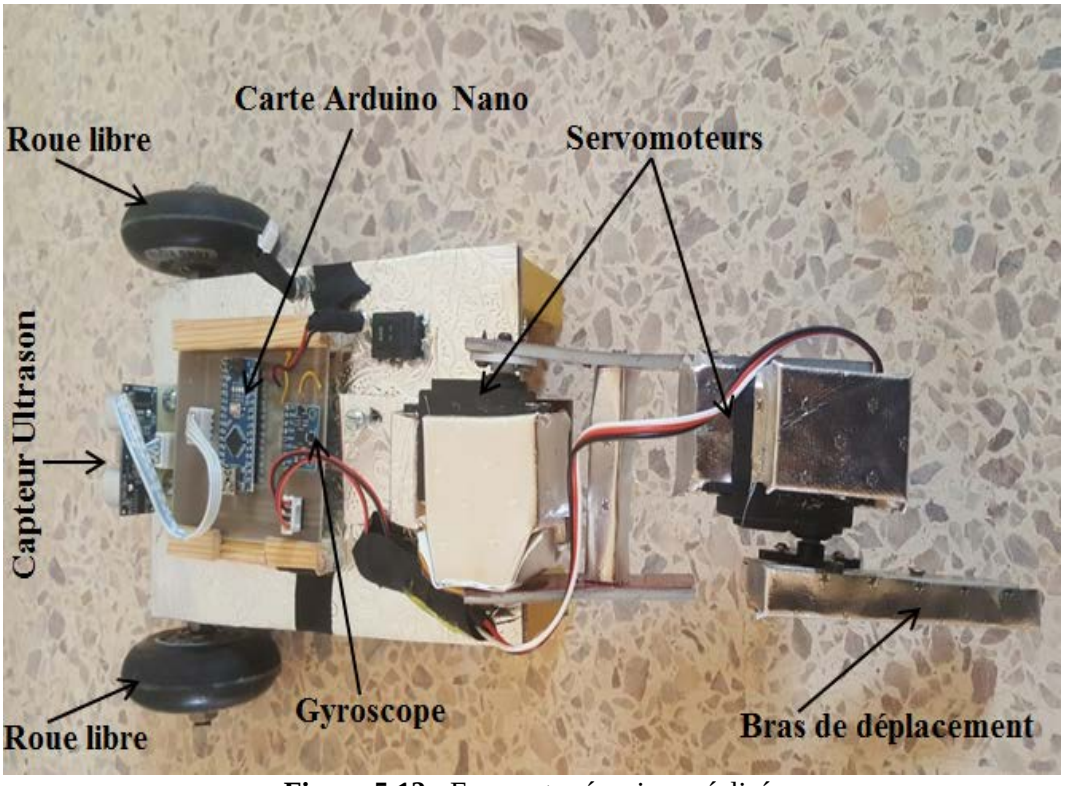


Figure 5.12 : Escargot mécanique réalisé.

Carte de commande : La carte de commande (**Figure 5.14**) est basée sur un microcontrôleur programmable de type ATMega328 intégré dans la carte Arduino Nano (**Figure 5.13**) dont les caractéristiques sont présentées dans le tableau 5.1. Elle comporte en outre, un module gyroscope trois axes de type MPU6050.

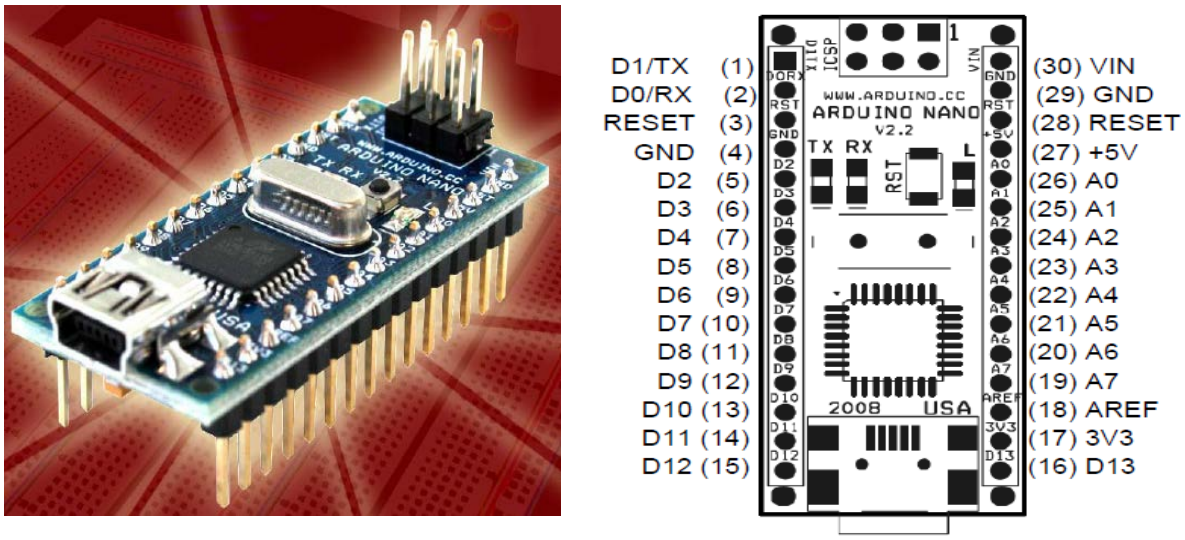


Figure 5.13: Carte de type Arduino Nano.

Tableau 5. 1: Caractéristiques de la carte Arduino Nano.

Pine	**Nom**	**Type**	**Description**
1-2, 5-16	D0-D13	E/S	Entrée/Sortie digitale (40mA)
3, 28	RESET	Initialisation	Entrée d'Initialisation
4, 29	GND	GND	Masse
17	3V3	Sortie	Sortie +3.3V
18	AREF	Entrée	Entrée de référence pour la conversion A/N
19-26	A7-A0	Entrée	Entrée analogique
27	+5V	E/S	Entrée +5V externe ou sortie +5V interne
30	VIN	PWR	Alimentation externe (6-20V)
Mémoire Programme		16KOctets	
EEPROM		512 Octets	
SRAM		1KOctets	
Fréquence d'Horloge		16MHz	

Figure 5.14: Carte de commande du robot « crawler ».

La carte de commande est reliée au capteur Ultrason et au deux servomoteurs selon le montage de la figure 5.15. Le servomoteur contient un petit circuit électronique qui permet de contrôler un moteur à courant continu en fonction de la position d'un potentiomètre intégré au servomoteur. La sortie du moteur à courant continu est reliée mécaniquement à une série d'engrenages qui augmente son couple de force en réduisant sa vitesse de rotation. Le servomoteur est commandé par des impulsions d'une longueur comprise entre 1 et 2 millisecondes. Une bibliothèque « Servo-1.1.6.zip[1] » est disponible gratuitement permettant de commander facilement ces servomoteurs.

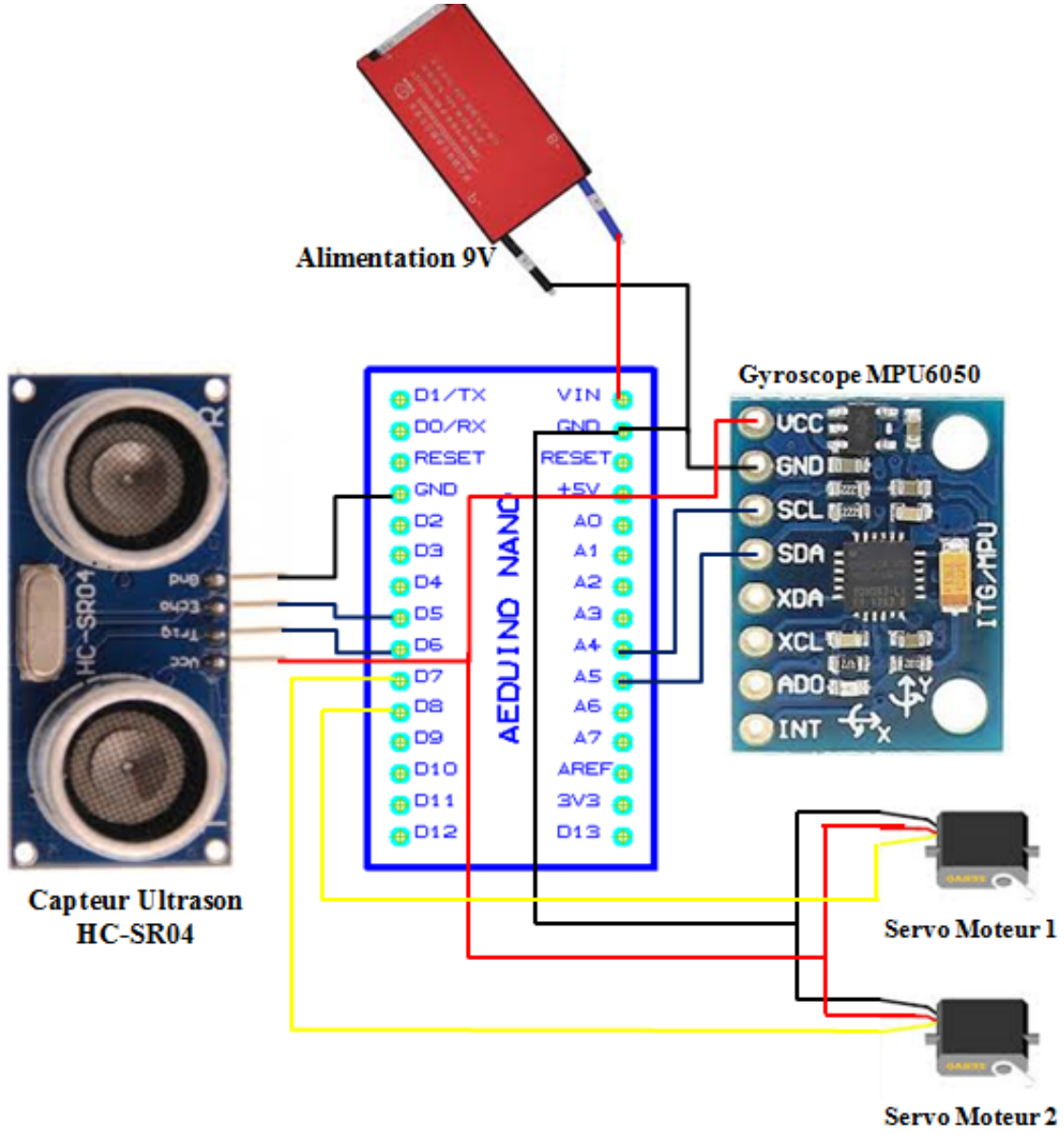


Figure 5.15 : Montage de commande du robot crawler.

Nous avons comparé les deux algorithmes d'AR avec et sans élimination d'actions du point de vue temps de convergence. Dans l'**algorithme 5.6**, les actions qui déplacent le robot vers l'arrière sont éliminées, ainsi que celles qui ne le maintiennent pas horizontale à 5° près. Les résultats obtenues avec cinq essaies sont regroupées dans le tableau ci-dessous.

[1] https://github.com/arduino-libraries/Servo

Tableau 5. 2: Temps de convergence des deux algorithmes d'AR pour l'escargot mécanique réalisé.

	Temps de convergence (s)	
	sans élimination des actions	avec élimination des actions
Essai 1	140	90
Essai 2	130	85
Essai 3	135	70
Essai 4	120	80
Essai 5	129	76

Nous remarquons une convergence plus vite avec cette technique d'élimination en ligne des actions non optimales. En effet, cette technique évite de ré-explorer les couples état/action qui n'entrent pas dans une séquence optimale de déplacement. Ceci permet, évidement, d'accélérer le processus de convergence.

5.4. Modèle d'AR appliqué à un robot suiveur de ligne

L'application du robot suiveur de ligne est très connue en robotique, il consiste à concevoir un robot capable de suivre une ligne noire entourée du blanc. Le suivi est généralement assuré par des capteurs optiques capables de différencier entre une zone noire et une autre blanche. Chaque fois que le robot dévie de la trajectoire noire, des actionneurs (moteurs de commande) le redirigent vers la piste. Nous présentons, dans cette section, un modèle d'AR appliqué au robot suiveur de ligne. L'objectif est de trouver une stratégie optimale dépendante du type de la ligne et de la vitesse de commande des roues, et qui permet un suivi avec un minimum de zigzag possible.

5.4.1. Modèle d'AR pour le suivi d'une ligne droite

Nous considérons l'exemple le plus simple, celui d'une ligne droite en noire (**Figure 5.16**) et d'un robot composé de trois capteurs infrarouges, d'une roue libre en avant et de deux roues en arrière commandées, chacune, par un moteur à courant continue. Le robot est contrôlé à l'aide de l'état de chacun de ces trois capteurs infrarouges qui indiquent s'ils sont au-dessus d'une zone noire ou blanche, tant que le capteur du milieu est au-dessus de la piste noire les deux roues tournent dans le même sens. Si, à un instant t, le capteur à droite est au-dessus de la piste, on arrête le moteur (ou on change le sens) qui commande la roue de droite afin de rediriger le robot vers la piste. Si le capteur à gauche est au-dessus du noire, à un instant t, on arrête le moteur (ou on change le sens) qui commande la roue de gauche.

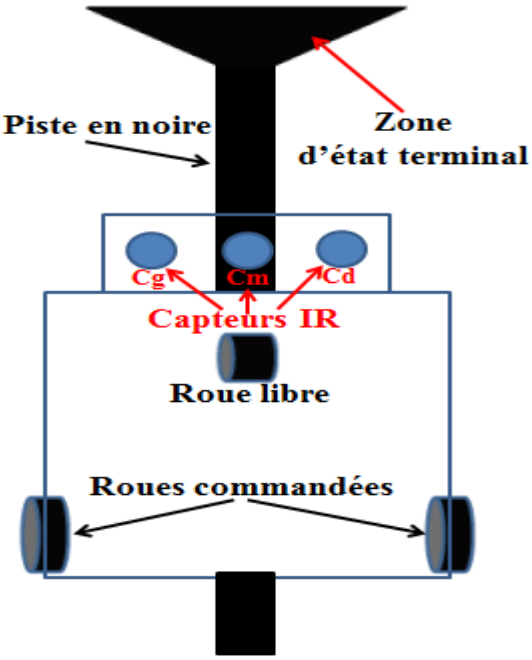


Figure 5.16 : Schéma formel d'un Robot suiveur de ligne droite avec trois capteurs optiques.

Le parallélisme des deux roues n'est pas généralement assuré, surtout lorsqu'on construit au laboratoire un petit robot avec du matériel simple. En outre, la vitesse des deux moteurs n'est pas toujours le même. Pour cela le robot ne peut pas suivre la ligne noire infiniment. L'objectif est donc de trouver une stratégie optimale en fonction de la vitesse de chacun des deux moteurs de commande qui permet de maintenir le robot tout droit tout le long de cette piste noire. On procède ainsi, à un apprentissage par renforcement après avoir définit le modèle PDM adéquat que nous proposons comme suit.

Espace d'états : On définit l'état du système par le triplet (E_g, E_m, E_d), les états des trois capteurs, avec $E_g, E_m, E_d \in \{0,1\}$ où 1 indique que le capteur est au-dessus de la piste noire et 0 sinon. L'espace d'états est donc fini et discret.

Espace d'actions : L'espace d'actions est défini par le couple (V_g, V_d), où V_g (V_d) est la vitesse de rotation du moteur à gauche (à droite). Pour avoir un espace d'actions fini et discret, les valeurs possibles de chaque $V_i, i = g, d$ sont : $\{-V_5, -V_4, -V_3, -V_2, -V_1, V_0, +V_1, +V_2, +V_3, +V_4, +V_5\}$, avec $V_0 = 0$ et V_5 la vitesse maximale du moteur subdivisée en cinq, ce qui implique que $V_i = \frac{\pm i}{5} \times V_5$. Notons que les signes + et – désignent le sens de rotation.

Espace du temps : l'espace du temps est discrétisé en périodes, où le temps entre deux décisions peut varier en fonction du temps de réponse des capteurs. Tout de même, le programme de commande observe ou lit à chaque pas du temps constant les données des trois capteurs.

Fonction coût: Le coût dépend de l'état observé après chaque action et avant de le définir, notons qu'il existe deux états terminaux définis par les triplets $(0,0,0)$ pour le débordement complet de la piste et $(1,1,1)$ pour l'atteint du but (voir figure 5.13). Ainsi, la fonction coût est définie comme suit :

- $C\left[(0,0,0)|(E_g, E_m, E_d), (V_g, V_d)\right] = +\infty$; //Débordement complet
- $C\left[(0,1,0)|(E_g, E_m, E_d), (V_g, V_d)\right] = 0$; //Robot sur la bonne piste
- $C\left[(1,0,0)|(E_g, E_m, E_d), (V_g, V_d)\right] = 1$; // Robot dévié à droite
- $C\left[(0,0,1)|(E_g, E_m, E_d), (V_g, V_d)\right] = 1$; // Robot dévié à gauche
- $C\left[(1,1,1)|(E_g, E_m, E_d), (V_g, V_d)\right] = 0$; // Robot atteint un état terminal

Notons que nous considérons que les états $(1,0,1)$, $(0,1,1)$ et $(1,1,0)$ sont impossibles.

Fonction de transition: Cette fonction est inconnue.

Le modèle PDM est ainsi défini, l'objectif est de minimiser la somme pondérée des coûts à long terme $\sum_{t=0}^{\infty} \gamma^t C_{x_t a_t}$, et vu que la fonction de transition est inconnue, nous utiliserons l'algorithme Q-lerning ou Q(λ) pour estimer une stratégie optimale. Le choix de l'un ou de l'autre dépend du système embarqué (carte de commande) utilisé dans le robot puisque la contrainte d'espace mémoire sera imposée.

5.4.2. Modèle d'AR pour le suivi d'une tournée de 90°

Nous considérons un robot suiveur de ligne composé de cinq capteurs infrarouges (**Figure 5.17**), l'objectif est d'apprendre au robot le suivi d'une tournée 90° en supposant que l'apprentissage du suivi d'une ligne droite a été déjà réalisé.

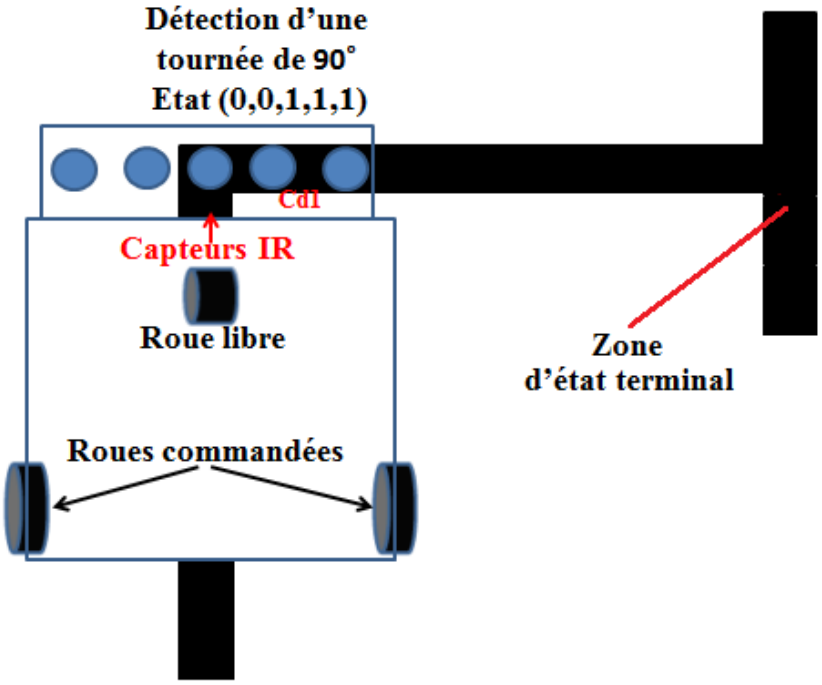


Figure 5.17 : Schéma formel d'un suiveur de ligne avec tournée de 90°et cinq capteurs optiques.

Nous modélisons ce problème avec un PDM comme suit :

Espace d'états : Nous définissons l'état du système par le tuple $(E_{g_2}, E_{g_1}, E_m, E_{d_1}, E_{d_2})$, les états des cinq capteurs infrarouges, avec $E_{g_i},\ E_m,\ E_{d_i} \in \{0,1\}$ où 1 indique que le capteur est au-dessus de la piste noire et 0 sinon. L'espace d'états est donc fini et discret.

Espace d'actions : L'espace d'actions est défini par le couple (V_g, V_d), où V_g (V_d) est la vitesse de rotation du moteur à gauche (à droite). Pour avoir un espace d'actions fini et discret, les valeurs possibles de chaque $V_i, i = g\ ou\ d$ sont : $\{-V_5, -V_4, -V_3, -V_2, -V_1, V_0, +V_1, +V_2, +V_3, +V_4, +V_5\}$, avec $V_0 = 0$ et V_5 la vitesse maximale du moteur subdivisée en cinq, ce qui implique que $V_i = \frac{\pm i}{5} \times V_5$. Notons que les signes + et – désignent le sens de rotation.

Espace du temps : l'espace du temps est discrétisé en périodes, dont le temps entre deux décisions peut varier en fonction de l'observation des capteurs. Tout de même, le programme de commande observe ou lit à chaque pas du temps constant les données des trois capteurs.

Fonction coût: Le coût dépend de l'état observé après chaque action et avant de le définir, notons que l'état (0,0,1,1,1) de détection d'une tournée 90° est un état initial et l'état (1,1,1,1,1) est un état terminal (Figure 5.14). Ainsi, la fonction coût est définie comme suit :

- $C[(0,0,0,0,0)] = +\infty$; //Débordement complet
- $C[(0,0,1,0,0)] = 1$; //Robot sur la bonne piste
- $C[(0,1,0,0,0)] = 2$; // Robot est proche de la bonne piste
- $C[(0,0,0,1,0)] = 2$; // Robot est proche de la bonne piste
- $C[(1,0,0,0,0)] = 3$; // Robot est plus au moins proche de la bonne piste
- $C[(0,0,0,0,1)] = 3$; // Robot est plus au moins proche de la bonne piste
- $C[(1,1,1,1,1)] = 0$; // Etat terminal

Fonction de transition: Cette fonction est inconnue.

L'objectif est de minimiser la somme pondérée des coûts à long terme $\sum_{t=0}^{\infty} \boldsymbol{\gamma^t C_{x_t a_t}}$, Cette minimisation est assurée si le robot est sur la bonne piste, puisque le coût $\boldsymbol{C[(0,0,1,0,0)] = 1}$ est le plus petit. L'algorithme Q-lerning ou Q(λ) est utilisé pour estimer la stratégie optimale.

5.4.3. Expérimentation

Afin de tester expérimentalement le modèle d'apprentissage de suivi de piste, nous avons réalisé un petit robot suiveur de ligne (**Figure 5.18**) composé de deux roues commandées chacune par un moteur à courant continue, d'une roue libre en avant, de cinq capteurs infrarouges.

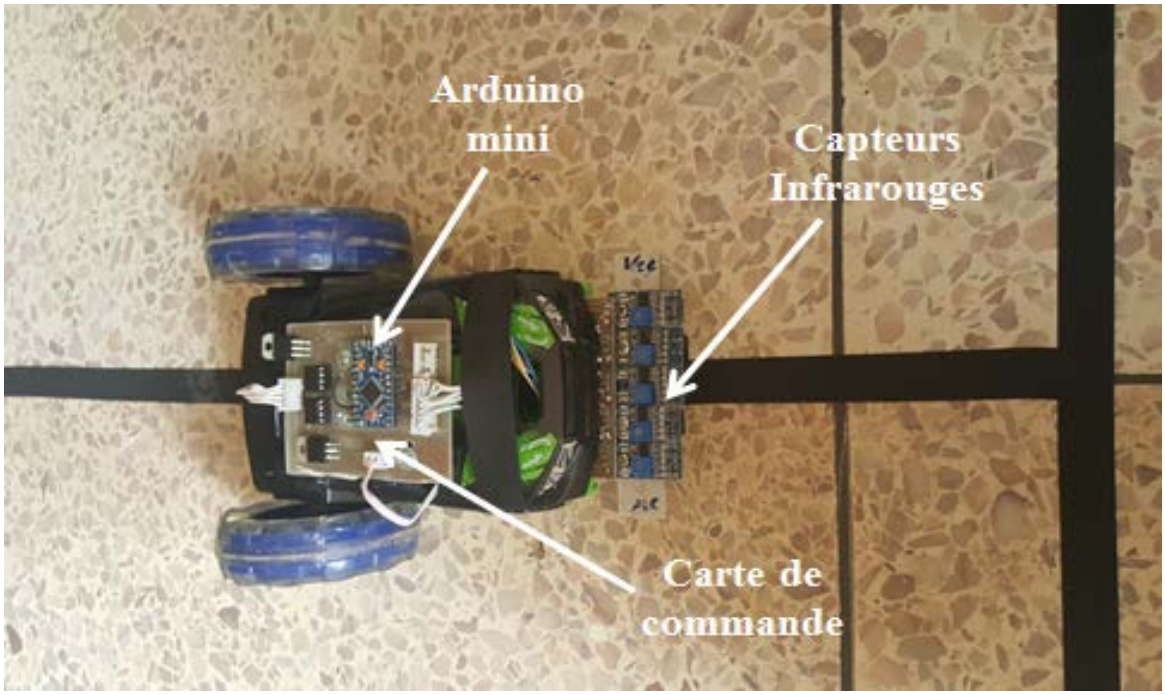


Figure 5.18 : Robot suiveur de ligne.

Carte de commande : L'élément de base de la carte de commande est un microcontrôleur de type Arduino Mini (**figure 5.19**) dont les caractéristiques sont présentées dans le tableau 5.3.

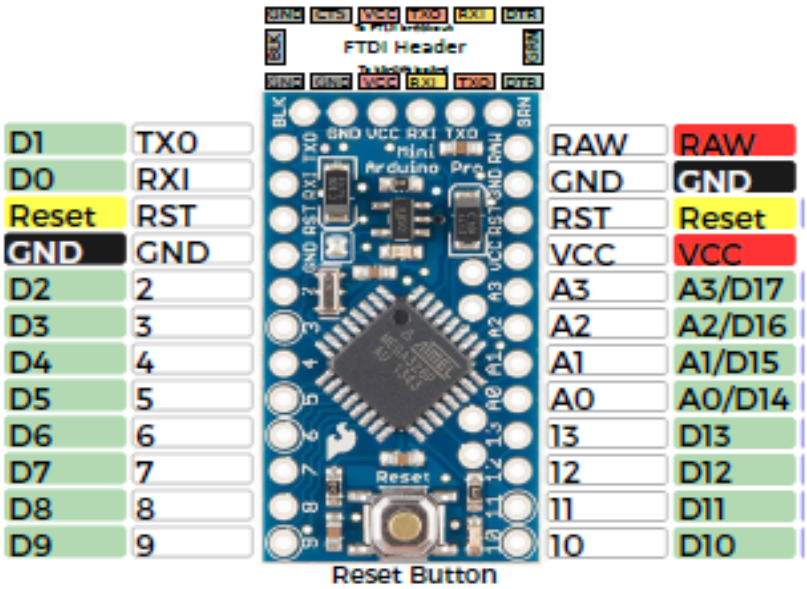


Figure 5.19: Carte Arduino Mini.

Tableau 5.3 : Caractéristiques de la carte Arduino Mini.

Microcontrôleur	ATMega328
Mémoire flash	32KOctets
EEPROM	1KOctets
RAM	2KOctets
Fréquence d'Horloge	16MHz

La carte comporte, en outre, deux circuits « pont H » pour commander les moteurs et de deux régulateurs de tension 5V, le premier pour alimenter la carte Arduino Mini et le deuxième pour séparer l'alimentation des moteurs. La carte est reliée aux deux moteurs et aux capteurs infrarouges de suivi de lignes selon le montage de la **figure 5.20**. Les broches Ci_j, j=1,..,2 désignent les sorties numériques des capteurs infrarouges.

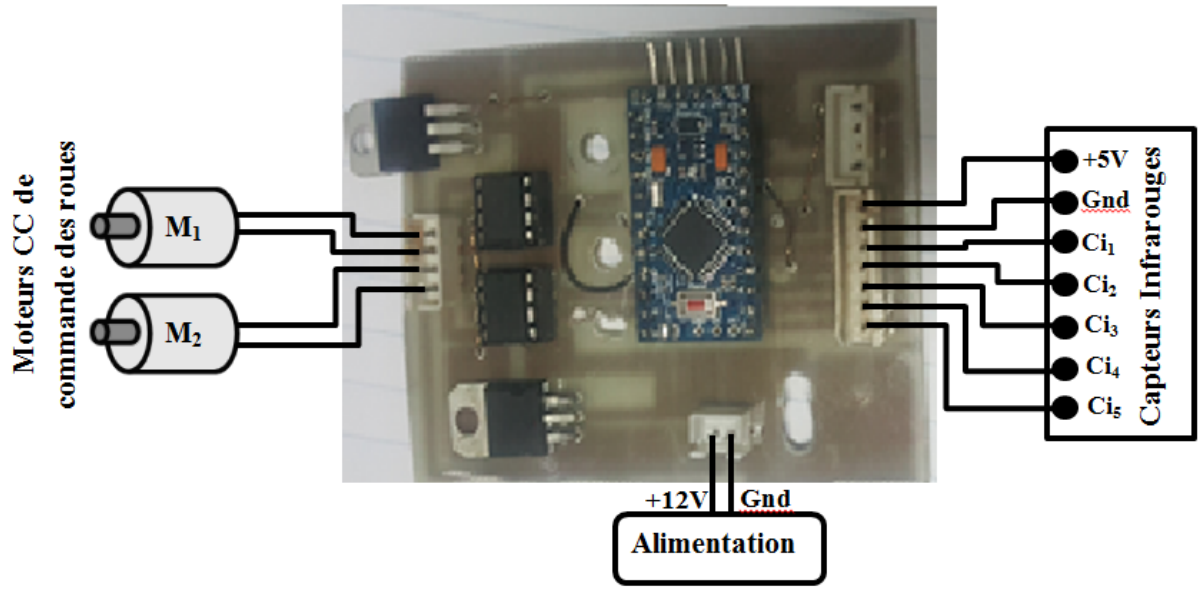


Figure 5.20 : Montage de commande du robot suiveur de ligne.

La réalisation du circuit imprimé est la première étape de réalisation de la carte de commande, différents méthodes de réalisation sont possibles, allant des méthodes classiques

aux méthodes automatiques sophistiquées. Dans un premier temps, nous avons utilisé une méthode classique basée sur le logiciel TCI (Tracé du Circuit Imprimé) qui permet de tracer manuellement le circuit (**Figure 5.21**) puis l'imprimer sur un transparent, ce dernier est collé sur la face d'une plaque en cuivre recouverte de résine et qui sera attaquée par de l'ultra-violet (Insolation). Un révélateur permet de retirer la résine de la surface non attaquée par l'ultra-violet (Révélation). Ensuite un liquide d'Hyperchlorurie de Fer, par exemple, permet d'arracher le cuivre (Rinçage). Enfin une phase de perçage et de soudage des composants est effectuée.

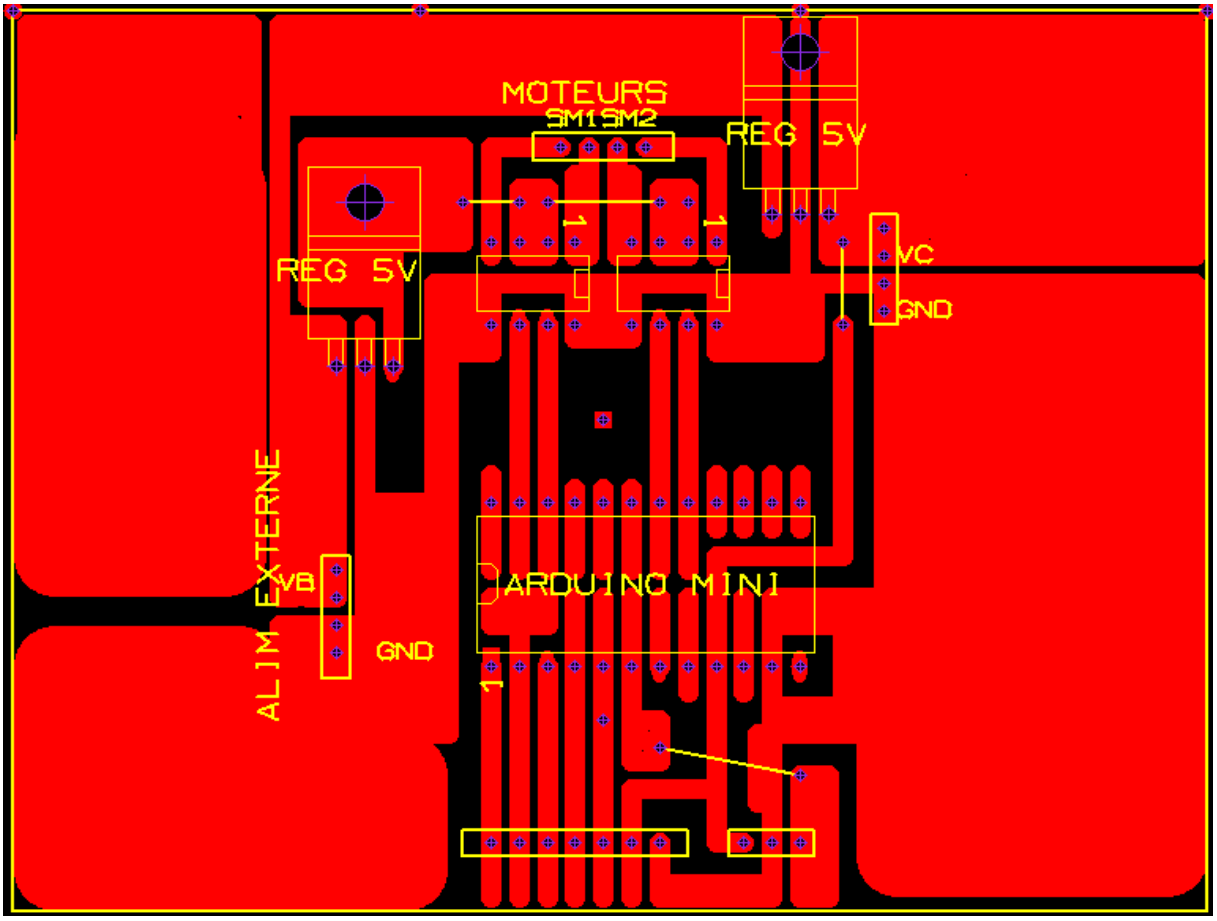


Figure 5.21: Circuit imprimé de la carte de commande du robot suiveur de ligne.

Nous avons implémenté l'algorithme Q-learning. Au début de l'apprentissage, le robot zigzag dans le suivi et chaque fois il est remis sur piste mais après convergence, le suivi devient rectiligne. Le tableau 5.4 présente le temps mis par le robot pour suivre une ligne rectiligne de 10 mètres de longueur durant les dix premières phases d'apprentissage.

Tableau 5. 4: Temps de suivi d'une ligne rectiligne de 10 mètres durant les dix premières phases d'apprentissage.

Episodes	1	2	3	4	5	6	7	8	9	10
Temps (ms)	9400	9300	9000	8000	7500	7050	7030	7090	7050	7040

Nous remarquons que le temps mis, après convergence, pour suivre la ligne rectiligne est réduit par rapport au suivi des premières épisodes, ce qui permet d'économiser le temps et l'énergie.

Remarque 5.1 : Pour permettre un suivi de ligne de différentes formes, un modèle d'apprentissage avec une matrice de capteurs infrarouges peut être utilisées afin de détecter les différents types de tournées d'une piste.

5.5. Modèle d'AR appliqué au robot auto-balancé

Le robot auto-balancé (**Figure 5.22**) est constitué d'une plate-forme solidaire de deux roues. L'axe de rotation des roues est perpendiculaire à l'axe de déplacement du robot. Le principe du contrôle est basé sur le pendule inversé. L'inclinaison de la plate-forme d'un angle α provoquera le déplacement du robot, de telle sorte que l'angle α redevient nul. L'objectif est donc de maintenir la plate-forme horizontale. Un déplacement du centre de gravité vers l'avant ou vers l'arrière obligera le robot à se déplacer dans le même sens pour maintenir son équilibre. Le contrôle du robot nécessite donc la connaissance de l'angle d'inclinaison α ainsi que la vitesse de variation de cet angle. Un accéléromètre-Gyroscope permet de déterminer ces deux paramètres.

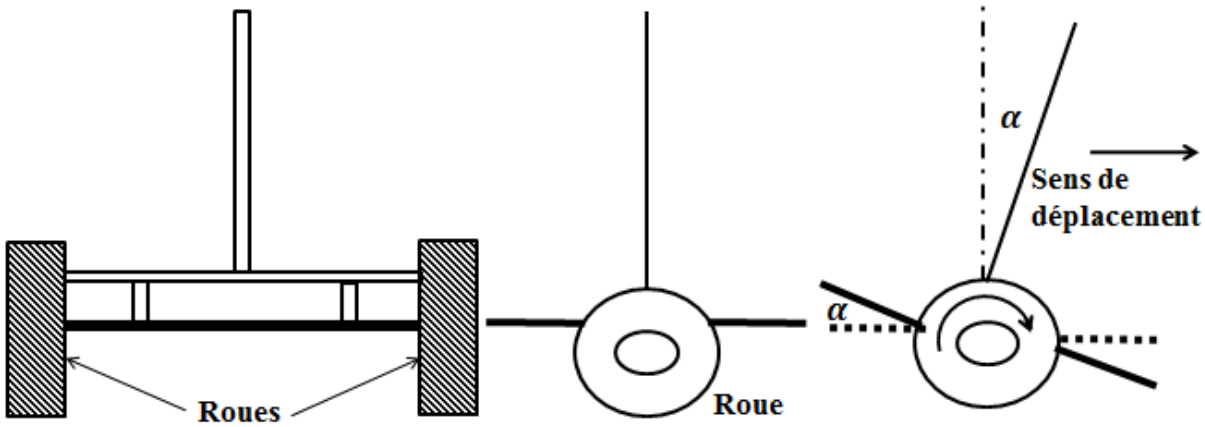


Figure 5.22 : Schéma formel du robot auto-balancé.

5.5.1. Modèle d'apprentissage

Nous présentons, dans cette section, un exemple simple de modèle de décision markovien appliqué au robot auto-balancé où l'état du système est l'angle d'inclinaison. L'espace d'états est donc continu mais une discrétisation de 2° sera utilisée. Les actions sont des déplacements en avant ou en arrière avec une vitesse appartenant à un intervalle prédéfinie et le coût dépend de l'angle d'inclinaison. Nous définissons le modèle comme suit :

Espace d'états : E={ α_i } avec $\alpha_i \in \{-12°, -10°, .., 0°, 2°, \dots, 12°\}$. Notons que l'état $(\alpha_i = 0°)$ est un état terminal avec un coût nul.

Espace d'action : A=$\{-v_5, -v_4, -v_3, -v_2, -v_1, v_0 = 0, +v_1, +v_2, +v_3, +v_4, +v_5\}$.

Espace de temps: l'espace de décision est discrétisé selon le temps de traitement des données du capteur et le temps nécessaire à un pas de déplacement des deux moteurs.

Fonction coût: Le coût dépend de l'état suivant observé, il est égal à zéro si l'angle est nul et il est égal à la valeur absolue de l'angle s'il est non nul, ainsi $C_{\alpha_t V_t \alpha_{t+1}} = |\alpha_{t+1}|$.

Fonction de transition: Cette fonction est inconnue.

L'objectif est de minimiser la somme pondérée des coûts à long terme $\sum_{t=0}^{\infty} \gamma^t C_{x_t a_t}$, et puisque le coût est proportionnel à l'angle d'inclinaison de la plate-forme, une stratégie optimale oriente le robot vers l'équilibre le plus vite possible. L'algorithme Q-lerning ou Q(λ) est utilisé pour estimer une stratégie optimale. Une stratégie d'exploration guidée par un programme de contrôle PID[2] et une élimination en ligne des actions non optimales accélère le processus de convergence.

5.5.2. Expérimentation réelle

Nous avons réalisé un robot auto-balancé (**Figure 5.23**) composé de deux roues commandées chacune par un moteur pas à pas et d'un gyroscope qui permet de calculer l'angle d'inclinaison.

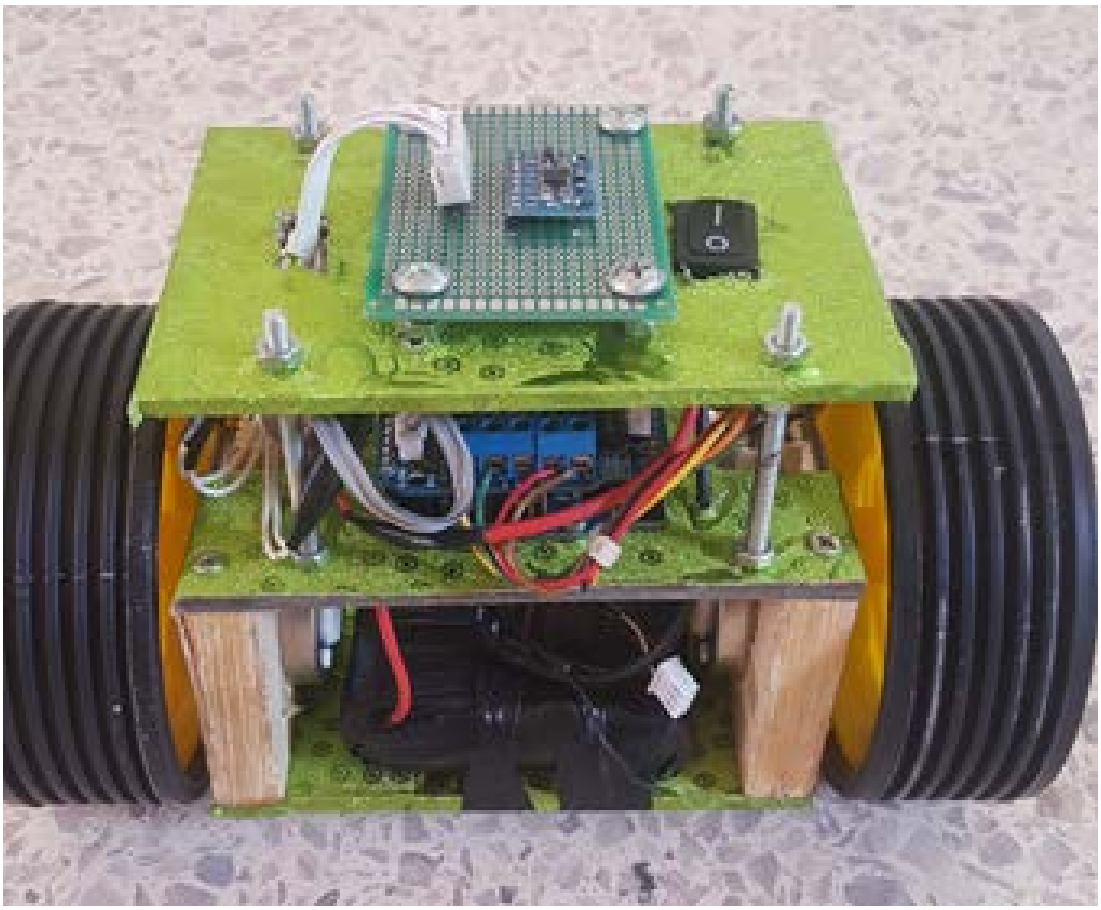


Figure 5.23 : Robot auto-balancé.

Carte de commande : La carte de commande est basée sur une carte shield Moteur (**Figure 5.24**) comportant les circuits nécessaires qui permettent de commander facilement deux servos moteurs, quatre moteurs continu et/ou deux moteurs pas-à-pas. Elle est conçue de

[2] (Proportionnel, Intégral, Dérivé) : Régulateur ou contrôleur, très connue dans les systèmes de commande, permettant d'améliorer les performances d'un système en boucle fermée.

telle sorte qu'elle se connecte au-dessus de la carte arduino. Une bibliothèque « Adafruit Motor Shiel library-1.0.1.zip [3] » est disponible gratuitement pour gérer facilement ces différents types de moteurs.

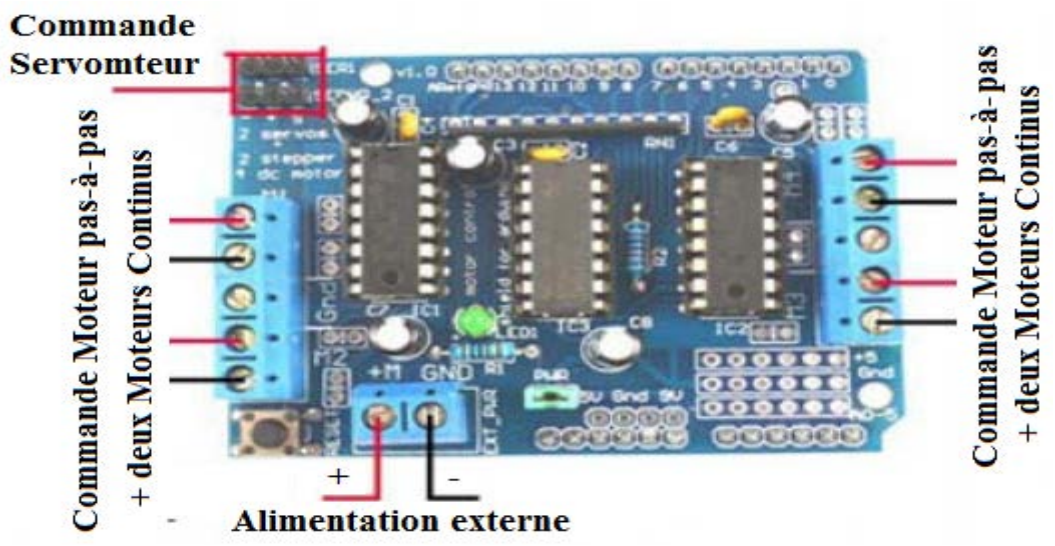


Figure 5.24: Carte shield Moteur.

La carte de commande est connectée aux deux moteurs pas-à-pas et au module gyroscope selon le montage de la **figure 5.25**. Une bibliothèque « MPU6050.zip [4] » est disponible gratuitement permettant d'utiliser facilement le module gyroscope MPU6050 et de déterminer, à l'aide d'une fonction disponible, l'angle que fait le robot avec l'axe verticale.

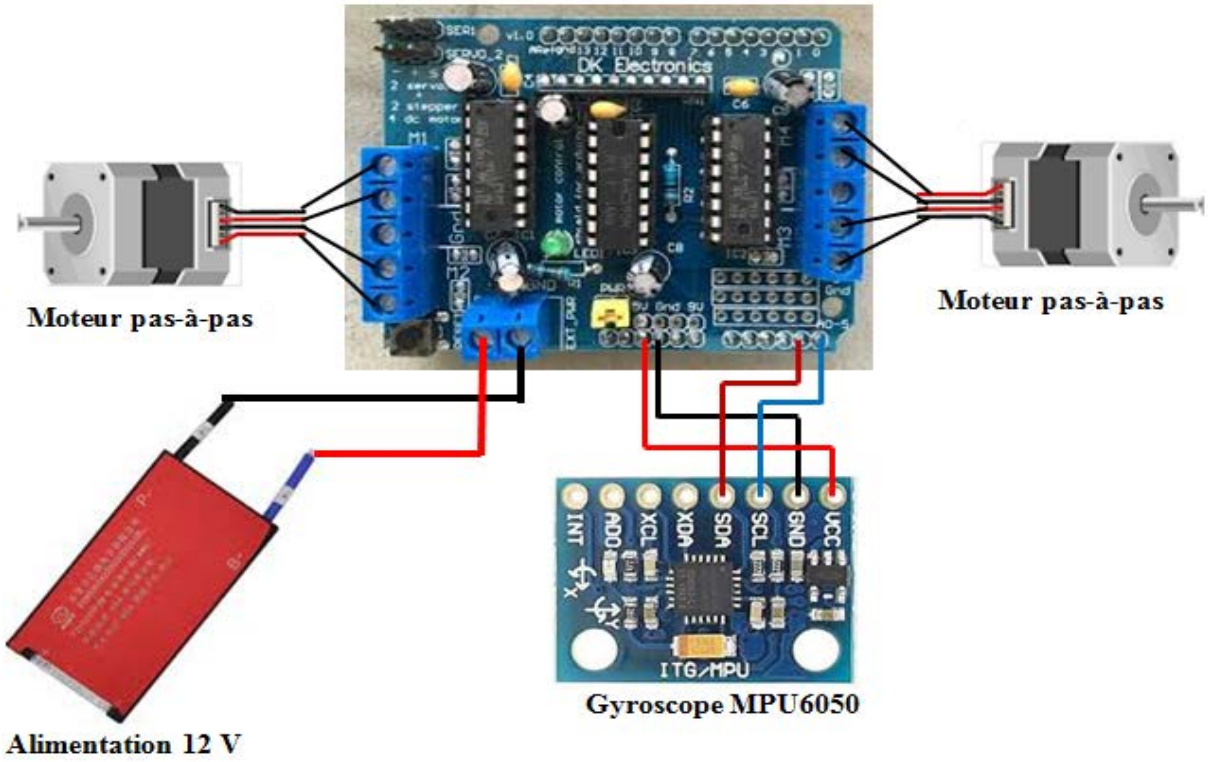


Figure 5.25 : Montage de commande du robot auto-balancé.

[3] https://github.com/adafruit/Adafruit-Motor-Shiel-library
[4] Https://github.com/electroniccats/mpu6050

Enfin, nous avons implémenté l'algorithme Q-learning, après un certain nombre d'essaies, l'expérience c'est bien déroulée et le robot converge vers son équilibre après chaque déviation excitée.

5.6. Conclusion

Dans ce chapitre, nous avons, tout d'abord, présenté un algorithme d'apprentissage par renforcement guidé appliqué à la navigation robotique dans un environnement inconnu. Cet algorithme permet de réduire la zone de l'environnement à explorer et ainsi d'accélérer le processus d'apprentissage.

Ensuite, nous avons proposé deux nouvelles méthodes d'AR appliquées à la marche robotique, la première méthode directe basée sur la couverture de zone et une deuxième méthode indirecte basée sur l'élimination des actions non optimales.

Nous avons également présenté deux nouveaux modèles d'AR appliqués au robot suiveur de ligne et au robot auto-balancé.

Finalement, afin de tester expérimentalement les modèles proposés, nous avons réalisé, au sein de notre laboratoire, trois petits robots : robot escargot-mécanique, robot suiveur de ligne et robot auto-balancé.

Conclusions Générales & Perspectives

Dans ce travail, on s'est préoccupé aux problèmes décisionnels de Markov qui sont actuellement parmi les approches les plus étudiées pour la planification et l'apprentissage par renforcement. Dans la plupart de ses applications réelles, l'espace d'état est de grande taille, ainsi notre objectif principal est d'alléger la tâche de résolution numérique de ces PDM en proposant des algorithmes plus rapides.

Nous avons commencé par une introduction aux bases théoriques des PDM connus, partiellement connus et inconnus, leurs critères d'optimalité et leurs différents algorithmes de résolution classiques. Nous avons brièvement présenté, quelques techniques pour remédier au problème des grandes dimensions : la technique d'élimination des actions, la réduction de l'espace d'états, la méthode de décomposition et le parallélisme. Nous avons aussi présenté un état de l'art sur l'agent robot avec ses différents constituants et les types de représentations de l'environnement. Ensuite, nous avons exposé quelques exemples de modèles PDM appliqués en robotique, à savoir la navigation robotique, robot transporteur d'objets, robot de position inconnue orienté vers un but, et un modèle d'AR appliqué l'escargot mécanique.

Ultérieurement, nous avons présenté des solveurs accélérés des PDM sous le critère d'actualisation. Ces solveurs sont basés sur un nouvel algorithme permettant de trouver simultanément les CFC avec leurs niveaux d'appartenance et sur une nouvelle définition des PDM restreints. Cette technique de décomposition contribue sûrement à accélérer le processus de partition de l'espace d'états et de réduire l'espace d'états de chaque sous problème.

Par ailleurs, nous avons considéré le problème du plus court chemin stochastique avec des impasses, modélisé par un PDM sous le critère total, dans lequel s'impose le problème de convergence des algorithmes de résolution et la complication d'optimisation multicritères. Nous avons, d'une part, proposé une nouvelle transformation de son modèle PDM qui permette de résoudre ces problèmes et de répondre à la question d'accessibilité énergétique. D'autre part, nous avons présenté une nouvelle méthode topologique de résolution du problème du plus court chemin stochastique avec des impasses. Enfin, dans l'application de planification de couverture de zone, nous avons proposé un modèle de décision markovien et un algorithme de résolution permettant un balayage optimal de la zone à explorer.

En outre, nous avons exposé un nouvel algorithme d'AR guidé pour le problème de la navigation robotique dans un environnement inconnu. Cet algorithme permet de réduire la zone de l'environnement à explorer et ainsi d'accélérer le processus d'apprentissage. Ensuite,

nous avons proposé deux nouvelles méthodes d'AR appliquées à la marche robotique, la première méthode directe basée sur la couverture de zone et une deuxième méthode indirecte basée sur l'élimination des actions non optimales. Nous avons également présenté deux nouveaux modèles d'AR appliqués au robot suiveur de ligne et au robot auto-balancé.

Il faut signaler que les simulations de ces différents algorithmes sont développées avec le langage JAVA et les différents environnements considérés sont générées aléatoirement. Les résultats obtenus sont très satisfaisants et montrent la performance et l'apport de nos contributions.

D'ailleurs, les expérimentations réelles réalisées avec les trois petits robots (escargot-mécanique, robot suiveur de ligne et robot auto-balancé) construits au sein de notre laboratoire à l'aide du simple matériel à faible coût, ont montré la validité et les performances des modèles et des algorithmes proposés en AR.

Nous pouvons conclure que les PDM sont bien adaptés à la prise de décision séquentielle sous incertitude et permettent de résoudre de nombreux problèmes dans le domaine robotique.

Certes, les contributions proposées sont loin d'être parfais. Nous présenterons ci-dessous un ensemble de perspectives qui vont permettre l'amélioration de nos contributions en enrichissant leurs fonctionnalités.

Perspectives

Le travail réalisé dans le cadre de cette thèse nous ouvre plusieurs perspectives de recherche intéressantes que nous comptons développer :

- Dans une première orientation, il s'agit d'appliquer nos approches dans d'autres applications récentes en robotique, surtout en AR où la conception de robots dotés d'une intelligence capable de s'adapter au changement de l'environnement dans lequel ils évoluent est un sujet d'actualité.
- Nous comptons aussi appliquer nos approches dans d'autres formes de représentation d'environnement, telles que les cartes métriques et les cartes topologiques. Dans lesquelles interviennent les problèmes complexes de cartographie et de localisation avec les tel-mètres à laser et les systèmes de vision.
- Dans une deuxième direction, nous comptons appliquer nos approches dans d'autres domaines d'IA, notamment en finance (gestion de risque), en médecine, en industrie, etc.
- Dans une troisième direction, nous allons étudier les PDM multi agents où la planification multi-robots intervient avec les contraintes d'interactions et de communications.

Bibliographies

[ABD03a] Abbad, M and Boustique, H. (2003). A decomposition algorithm for limiting average Markov Decision Problems. Operations Research Letters, vol. 31, no 6, p. 473-476. https://doi.org/10.1016/S0167-6377(03)00055-5.

[ABD03b] Abbad, M. & Daoui, C. (2003). Hierarchical algorithms for discounted and weighted Markov Decision Processes. Mathematical Methods of Operations Research 58(2):237–245. https://doi.org/10.1007/s001860300290.

[ARC93] Archibald, T. W., McKinnon, K. I. M., & Thomas, L. C. (1993). Serial and parallel value iteration algorithms for discounted Markov decision processes. European journal of operational research, 67(2), 188-203. https://doi.org/10.1016/0377-2217(93)90061-Q.

[ATK05] Atkar, P. N., Greenfield, A., Conner, D. C., Choset, H., & Rizzi, A. A. (2005). Uniform coverage of automotive surface patches. The International Journal of Robotics Research, 24(11), 883-898. https://doi.org/10.1177/0278364905059058.

[BEL57] BELLMAN R, E. (1957). Dynamic Programming, Princeton University Press.

[BER87] BERTSEKAS, D. P. (1987). Dynamic programming: deterministic and stochastic models. Prentice-Hall, Inc. Upper Saddle River, NJ, USA ©1987.

[BEY04] Beynier, A., et Mouaddib, A. I. (2010). Solving efficiently Decentralized MDPs with temporal and resource constraints. Autonomous Agents and Multi-Agent Systems. Volume 23, Issue 3, pp 486–539. https://doi.org/10.1007/s10458-010-9145-2.

[BOG99] Boger, J., Poupart, P., Hoey, J., Boutilier, C., Fernie, G., & Mihailidis, A. (2005, July). A decision-theoretic approach to task assistance for persons with dementia. In IJCAI (pp. 1293-1299). https://www.ijcai.org/Proceedings/05/Papers/1186.pdf.

[BON02] Bonet, B., & Geffner, H. (2002). Solving stochastic shortest-path problems with RTDP. Universidad Simon Bolivar, Tech. Rep. https://bonetblai.github.io/reports/rtdp.pdf.

[BON03a] Bonet, B., & Geffner, H. (2003, June). Labeled RTDP: Improving the Convergence of Real-Time Dynamic Programming. In ICAPS (Vol. 3, pp. 12-21). https://ftp.cs.ucla.edu/pub/stat_ser/R319.pdf.

[BON03b] Bonet, B., & Geffner, H. (2003, August). Faster heuristic search algorithms for planning with uncertainty and full feedback. In IJCAI (pp. 1233-1238). https://ldc.usb.ve/~bonet/reports/hdp.pdf.

[BUO04] Buongiorno, J., & Zhou, M. (2004). The use of Markov optimization models in the economic and ecological management of forest landscapes under risk. University of Wisconsin-Madison. Technical Series, (35). https://www.ipef.br/publicacoes/stecnica/nr35/cap06.pdf.

[CHA15] Chafik S, Daoui C (2015) A modified value iteration algorithm for discounted Markov decision process. J Electron Commerce Organ 13(3):47–57. DOI: 10.5120/ijca2016908033.

[CHE13] Chen, P., & Lu, L. (2013, October). Markov decision process parallel value iteration algorithm on GPU. In 2013 International Conference on Information Science and Computer Applications (ISCA 2013). Atlantis Press. doi:10.2991/isca-13.2013.51.

[COU99] Couvreur J-M (1999) On-the-fly verification of linear temporal logic. In: Wing JM, Woodcock J, Davies J (eds) FM'99—formal methods. FM 1999, in computer science, vol 1708. Springer, Berlin. https://www.lrde.epita.fr/dload/spot/bib/couvreur.99.fm.pdf .

[D'E63] d'Epenoux, F. (1963). A probabilistic production and inventory problem. Management Science, 10(1), 98-108. https://doi.org/10.1287/mnsc.10.1.98.

[DAG98] Dagum, L., & Enon, R. (1998). OpenMP: an industry standard API for shared-memory programming. Computational Science and Engineering, IEEE, 5(1), 46-55. doi: 10.1109/99.660313.

[DAI11] Dai, P., Weld, D. S., & Goldsmith, J. (2011). Topological value iteration algorithms. Journal of Artificial Intelligence Research, 42, 181-209. https://doi.org/10.1613/jair.3390.

[DAO07] Daoui, C., & Abbad, M. (2007). On some algorithms for Limiting Average Markov Decision Processes. Operations Research Letters, 35(2), 261-266. https://doi.org/10.1016/j.orl.2006.03.006.

[DAO10] Daoui, C., Abbad, M., & Tkiouat, M. (2010). Exact decomposition approaches for Markov decision processes: A survey. Advances in Operations Research, 2010. http://dx.doi.org/10.1155/2010/659432.

[DAY94] Dayan, P., & Sejnowski, T. J. (1994). TD (λ) converges with probability 1. Machine Learning, 14(3), 295-301. https://doi.org/10.1007/BF00993978.

[DEA93] Dean, T. L., Kaelbling, L. P., Kirman, J., & Nicholson, A. E. (1993, July). Planning With Deadlines in Stochastic Domains. In AAAI (Vol. 93, pp. 574-579). https://pdfs.semanticscholar.org/5c9a/43b521309666f555c5b22f84ac5943abc0fd.pdf.

[DEA95] Dean, T., and Lin, S. H. (1995, August). Decomposition techniques for planning in stochastic domains. In IJCAI (Vol. 2, p. 3). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.80.4538&rep=rep1&type=pdf.

[DED99] Dedeoglu, G., Mataric, M.J., & Sukhatme, G.S. (1999). Incremental online topological map building with a mobile robot. Mobile Robots. 10.1117/12.369248.

[DIJ82] Dijkstra E (1982) Finding the maximum strong components in a directed graph. Selected writings on computing: a personal perspective, texts and monographs in computer science. Springer, New York, pp 22–30. //doi.org/10.1007/978-1-4612-5695-3_3.

[ELN11] Elnashar, A. I. (2011). Parallel performance of MPI sorting algorithms on dual-core processor windows-based systems. arXiv preprint arXiv:1105.6040. 10.5121/ijdps.2011.2301.

[FED99] Feder, H. J. S., Leonard, J. J., & Smith, C. M. (1999). Adaptive mobile robot navigation and mapping. The International Journal of Robotics Research, 18(7), 650-668. https://doi.org/10.1177/02783649922066484.

[FER95] Ferber, J. (1995). Livre : Les systèmes multi-agents vers une intelligence collective. InterEdition. www.lirmm.fr/~ferber/publications/LesSMA_Ferber.pdf .

[FIL11] Filiat, David. (2011) Robotique Mobile. These de doctorat. EDX. https://www.theses.fr/070072337.

[GAL13] Galceran, E., & Carreras, M. (2013). A survey on coverage path planning for robotics. Robotics and Autonomous systems, 61(12), 1258-1276. https://doi.org/10.1016/j.robot.2013.09.004.

[GOO04] Gooneratne, C. P., Mukhopahyay, S. C., & Gupta, G. S. (2004, December). A review of sensing technologies for landmine detection: Unmanned vehicle based approach. In Proceedings of the 2nd International Conference on Autonomous Robots and Agents, Palmerston North, New Zealand (pp. 401-407). http://www-ist.massey.ac.nz/conferences/ICARA2004/files/Papers/Paper70_ICARA2004_401_407.pdf.

[GRO02] Gropp, W. (2002, September). MPICH2: A new start for MPI implementations. In European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting (pp. 7-7). Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-45825-5_5.

[GRO08] Groupe PDMIA (2008). Livre : Tome 2, Processus Décisionnels de Markov en Intelligence Artificielle. http://researchers.lille.inria.fr/~munos/papers/files/bouquinPDMIA.pdf.

[GRO96] Gropp, W., Lusk, E., Doss, N., & Skjellum, A. (1996). A high-performance, portable implementation of the MPI message passing interface standard. Parallel computing, 22(6), 789-828. https://doi.org/10.1016/0167-8191(96)00024-5.

[GRO99] Gropp, W., Lusk, E., et Skjellum, A (1999). Using MPI: portable parallel programming with the message-passing interface (Vol. 1). MIT press. https://www.amazon.com/Using-MPI-Programming-Engineering-Computation/dp/0262571323.

[GUE02] Guestrin, C., & Gordon, G. (2002, August). Distributed planning in hierarchical factored MDPs. In Proceedings of the Eighteenth conference on Uncertainty in artificial intelligence (pp. 197-206). Morgan Kaufmann Publishers Inc. http://www.cs.cmu.edu/~ggordon/guestrin-gordon.distributed-planning-UAI02.pdf.

[GUU97] Gaussier, P., Revel, A., Joulain, C., & Zrehen, S. (1997). Living in a partially structured environment: How to bypass the limitations of classical reinforcement techniques. Robotics and Autonomous Systems, 20(2-4), 225-250. https://doi.org/10.1016/S0921-8890(97)80708-7.

[HAF15] Hafez, M. B., & Loo, C. K. (2015). Topological Q-learning with internally guided exploration for mobile robot navigation. Neural Computing and Applications, 26(8), 1939-1954. https://doi.org/10.1007/s00521-015-1861-8.

[HAS73] Hastings, N. A. J., & Mello, J. M. C. (1973). Tests for suboptimal actions in discounted Markov programming. Management Science,19(9), 1019-1022. https://doi.org/10.1287/mnsc.19.9.1019.

[HUA01] Huang, W. H. (2001). Optimal line-sweep-based decompositions for coverage algorithms. In Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No. 01CH37164) (Vol. 1, pp. 27-32). IEEE. 10.1109/ROBOT.2001.932525.

[HUB77] Hübner, G. (1977). Improved procedures for eliminating suboptimal actions in Markov programming by the use of contraction properties. In Transactions of the Seventh Prague Conference on Information Theory, Statistical Decision Functions, Random Processes and of the 1974 European Meeting of Statisticians (pp. 257-263). Springer, Dordrecht. https://doi.org/10.1007/978-94-010-9910-3_27.

[HWA12] Hwang, K. S., Jiang, W. C., & Chen, Y. J. (2012, July). Tree-based Dyna-Q agent. In 2012 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM) (pp. 1077-1080). IEEE. DOI: 10.1109/AIM.2012.6266001.

[JAB08] Jaber, N. (2008). Accelerating successive approximation algorithm via action elimination (Doctoral dissertation). http://hdl.handle.net/1807/16794.

[KAE96] Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. Journal of artificial intelligence research, 4, 237-285. https://doi.org/10.1613/jair.301.

[KAN07] J Kang, J. W., Kim, S. J., Chung, M. J., Myung, H., Park, J. H., & Bang, S. W. (2007, August). Path planning for complete and efficient coverage operation of mobile robots. In 2007 International Conference on Mechatronics and Automation (pp. 2126-2131). IEEE. DOI: 10.1109/ICMA.2007.430388.

[KOL11] Kolobov, A., Mausam, M., Weld, D. S., & Geffner, H. (2011, March). Heuristic search for generalized stochastic shortest path MDPs. In Twenty-First International Conference on Automated Planning and Scheduling. https://www.aaai.org/ocs/index.php/ICAPS/ICAPS11/paper/viewPaper/2682.

[KOL12] Kolobov, A., & Weld, D. (2012). A theory of goal-oriented MDPs with dead ends. arXiv preprintarXiv:1210.4875. https://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/viewPaper/5102.

[KUN99] Kunz, C., Willeke, T., & Nourbakhsh, I. R. (1999). Automatic mapping of dynamic office environments. Autonomous Robots,7(2), 131-142. https://doi.org/10.1023/A:1008958016898.

[LEV90] Levitt, T. S., & Lawton, D. T. (1990). Qualitative navigation for mobile robots. Artificial Intelligence, 44(3), 305-360. https://doi.org/10.1016/0004-3702(90)90027-W.

[LEW02] Lewis, M. E., Ayhan, H., & Foley, R. D. (2002). Bias optimal admission control policies for a multiclass nonstationary queueing system. Journal of applied probability, 39(1), 20-37. https://doi.org/10.1239/jap/1019737985.

[LIN93] Lin, L. J. (1993). Reinforcement learning for robots using neural networks (No. CMU-CS-93-103). Carnegie-Mellon Univ Pittsburgh PA School of Computer Science. https://apps.dtic.mil/dtic/tr/fulltext/u2/a261434.pdf.

[LIT95] Littman, M. L., Dean, T. L., & Kaelbling, L. P. (1995, August). On the complexity of solving Markov decision problems. In Proceedings of the Eleventh conference on Uncertainty in artificial intelligence (pp. 394-402). Morgan Kaufmann Publishers Inc. https://dl.acm.org/citation.cfm?id=2074203.

[LOW14] Lowe G (2014) Concurrent depth-first search algorithms. tools and algorithms for the construction and analysis of systems. In: Lecture notes in computer science, vol. 8413, p 202. https://doi.org/10.1007/978-3-642-54862-8_14.

[LOZ13] Lozenguez, G., Adouane, L., Beynier, A., Martinet, P., & Mouaddib, A. I. (2013, July). Résolution approchée par décomposition de processus décisionnels de Markov appliquée à l'exploration en robotique mobile. In 8èmes Journées Francophones sur la Planification, la Décision et l'Apprentissage pour la conduite de systèmes, JFPDA 2013. https://hal.archives-ouvertes.fr/hal-01714848/.

[MAC67] MacQueen, J. (1967). Letter to the Editor—A Test for Suboptimal Actions in Markovian Decision Problems. Operations Research,15(3), 559-561. https://doi.org/10.1287/opre.15.3.559.

[MOZ07] Mozos, O. M., Triebel, R., Jensfelt, P., Rottmann, A., & Burgard, W. (2007). Supervised semantic labeling of places using information extracted from sensor data. Robotics and Autonomous Systems, 55(5), 391-402. https://doi.org/10.1016/j.robot.2006.12.003.

[MEU98] Meuleau, N., Hauskrecht, M., Kim, K. E., Peshkin, L., Kaelbling, L. P., Dean, T. L., et Boutilier, C. (1998, July). Solving very large weakly coupled Markov decision processes. In AAAI/IAAI (pp. 165-172). https://www.aaai.org/Papers/AAAI/1998/AAAI98-023.pdf.

[MIC61] MICHIE D., « Trial and Error », Science Survey, vol. 2, p. 129–145, 1961.

[MIC68] Michie, D., & Chambers, R. A. (1968). BOXES: An experiment in adaptive control. Machine intelligence, 2(2), 137-152. https://pdfs.semanticscholar.org/2f02/7193fb703d0af58ec382bd1438daff9417d7.pdf.

[MOO93] Moore, A.W. & Atkeson, C.G. Prioritized sweeping: reinforcement learning with less data and less time. Mach Learn (1993) 13: 103. https://doi.org/10.1007/BF00993104.

[MOR85] Moravec, H., & Elfes, A. (1985, March). High resolution maps from wide angle sonar. In Proceedings. 1985 IEEE international conference on robotics and automation (Vol. 2, pp. 116-121). IEEE. DOI: 10.1109/ROBOT.1985.1087316.

[NAJ00] Najjaran, H. and Kircanski, N. (2000). Path planning for a terrain scanner robot. In Proc. 31st Int. Symp. Robotics, Montereal, QC, Canada 2000, pp. 132-137. DOI: 10.1109/ROBOT.2001.933198.

[OH04] Oh, J. S., Choi, Y. H., Park, J. B., & Zheng, Y. F. (2004). Complete coverage navigation of cleaning robots using triangular-cell-based map. IEEE Transactions on Industrial Electronics, 51(3), 718-726. DOI: 10.1109/TIE.2004.825197.

[PAP87] Papadimitriou, C. H., & Tsitsiklis, J. N. (1987). The complexity of Markov decision processes. Mathematics of operations research, 12(3), 441-450. https://doi.org/10.1287/moor.12.3.441.

[PAR98] Parr, R. (1998, July). Flexible decomposition algorithms for weakly coupled Markov decision problems. In Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence (pp. 422-430). Morgan Kaufmann Publishers Inc. https://dl.acm.org/citation.cfm?id=2074144.

[PAV06] Pavitsos, A., & Kyriakidis, E. G. (2009). Markov decision models for the optimal maintenance of a production unit with an upstream buffer. Computers & Operations Research, 36(6), 1993-2006. https://doi.org/10.1016/j.cor.2008.06.014.

[PEN96] Peng, J., & Williams, R. J. (1996). Incremental Multi-Step Q-Learning. Machine Learning, 1(22), 283-290. DOI 10.1023/A:1018076709321.

[POR71] Porteus, E. L. (1971). Some bounds for discounted sequential decision processes. Management Science, 18(1), 7-11. https://doi.org/10.1287/mnsc.18.1.7.

[PUT94] PUTERMAN M., Markov Decision Processes : Discrete Stochastic Dynamic Programming, John Wiley & Sons, Inc., New York, USA, 1994. https://dl.acm.org/citation.cfm?id=528623.

[RON02] Brafman, R. I., & Tennenholtz, M. (2002). R-max-a general polynomial time algorithm for near-optimal reinforcement learning. Journal of Machine Learning Research, 3(Oct), 213-231. http://www.jmlr.org/papers/volume3/brafman02a/brafman02a.pdf.

[ROS91] Ross, K. W., & Varadarajan, R. (1991). Multichain Markov decision processes with a sample path constraint: A decomposition approach. Mathematics of Operations Research, 16(1), 195-207. https://doi.org/10.1287/moor.16.1.195.

[RYU11] Ryu, S. W., Lee, Y. H., Kuc, T. Y., Ji, S. H., & Moon, Y. S. (2011, November). A search and coverage algorithm for mobile robot. In 2011 8Th international conference on ubiquitous robots and ambient intelligence (URAI) (pp. 815-821). IEEE. DOI: 10.1109/URAI.2011.6146029.

[SAM69] Samuel, A. L. (1969). Some studies in machine learning using the game of checkers. II—Recent progress. Annual Review in Automatic Programming, 6, 1-36.

https://doi.org/10.1016/0066-4138(69)90004-4.

[SAR14] Sari, S. C., Prihatmanto, A. S., & Adiprawita, W. (2014). Online State Elimination in Accelerated reinforcement Learning. International Journal on Electrical Engineering and Informatics, 6(4), 665. http://ijeei.org/docs-136261624954bcc86058df3.pdf.

[SAT98] Sato, M. (2002, October). OpenMP: parallel programming API for shared memory multiprocessors and on-chip multiprocessors. In 15th International Symposium on System Synthesis, 2002.(pp. 109-111). IEEE. DOI: 10.1145/581199.581224.

[SCH12] Schuitema, E. (2012). Reinforcement learning on autonomous humanoid robots. https://doi.org/10.4233/uuid:986ea1c5-9e30-4aac-ab66-4f3b6b6ca002.

[SHA81] Sharir, M. (1981). A Strong-connectivity Algorithm and its Applications in Data Flow analysis. Computers & Mathematics with Applications, 7(1):67-72. https://doi.org/10.1016/0898-1221(81)90008-0.

[SIG04] SIGAUD O. (2004) Comportements adaptatifs pour les agents dans des environnements informatiques complexes, Mémoire d'Habilitation à Diriger des Recherches de l'Université PARIS VI http://www.isir.upmc.fr/files/2004THDR473.pdf.

[SIN96] Singh, S. P., & Sutton, R. S. (1996). Reinforcement learning with replacing eligibility traces. Machine learning, 22(1-3), 123-158. https://doi.org/10.1023/A:1018012322525.

[SUR15] Suresh, K., Vidyasagar, K., & Basha, A. F. (2015). Multi Directional Conductive Metal Detection Robot Control. International Journal of Computer Applications, 109(4). https://pdfs.semanticscholar.org/be85/d89cac6ae3ec5d8ed18431c67c18b9c7ea0b.pdf.

[SUT88] Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. Machine learning, 3(1), 9-44. https://doi.org/10.1007/BF00115009.

[SUT98] Sutton, R. S., & Barto, A. G. (1998). Introduction to reinforcement learning (Vol. 135). Cambridge: MIT press. https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf.

[TAR72] Tarjan, R. (1972). Depth-first search and linear graph algorithms. SIAM journal on computing, 1(2), 146-160. https://doi.org/10.1137/0201010.

[TEI12] Teichteil-Königsbuch, F. (2012, July). Stochastic safest and shortest path problems. In Twenty-Sixth AAAI Conference on Artificial Intelligence. https://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/viewPaper/5102.

[THR98] Thrun, S. (1998). Learning metric-topological maps for indoor mobile robot navigation. Artificial Intelligence, 99(1), 21-71. https://doi.org/10.1016/S0004-3702(97)00078-7.

[TOK09] Tokic. M., Ertel. W. & Fesseler. J. (2009). The crawler. A class Room Demonstrator for Reinforcement Learning. In Proceeding of the Twenty-Second International FLAIRS Conference. https://pdfs.semanticscholar.org/338e/cba7ec419ea67324dfbd1911b6fc10234511.pdf

[TOK14] Tokekar, P., Karnad, N., & Isler, V. (2014). Energy-optimal trajectory planning for car-like robots. Autonomous Robots, 37(3), 279-300. https://doi.org/10.1007/s10514-014-9390-3.

[TRE17] Trevizan, F., Teichteil-Konigsbuch, F. and Thiebaux, S. (2017). Efficient solutions for stochastic shortest path problems with dead ends. In: Proceedings of the Thirty-Third

Conference on Uncertainty in Artificial Intelligence, UAI. 2017. p. 11-15. https://felipe.trevizan.org/papers/trevizan17:mcmp.pdf

[WAT89] WATKINS C. (1989). Learning from Delayed Rewards, PhD thesis, Cambridge University, Cambridge, UK. https://www.cs.rhul.ac.uk/home/chrisw/thesis.html.

[WAT92] Watkins, C. J., & Dayan, P. (1992). Q-learning. Machine learning, 8(3-4), 279-292. https://doi.org/10.1007/BF00992698.

[WEL12] Weld, A. K. M. D. S. (2012). Stochastic Shortest Path MDPs with Dead Ends. HSDIP 2012, 78. https://pdfs.semanticscholar.org/cf4b/20f9ed99e805185861d3f88e24754eef5d68.pdf.

[WHI93] White, D. J. (1993). A survey of applications of Markov decision processes. Journal of the operational research society, 44(11), 1073-1096. https://doi.org/10.1057/jors.1993.181.

[WIE98] Wiering, M., & Schmidhuber, J. (1998). Fast online Q (λ). Machine Learning, 33(1), 105-115. https://doi.org/10.1023/A:1007562800292.

[ZHA09] Zhang, Q., Sun, G., & Xu, Y. (2009, June). Parallel Algorithms for Solving Markov Decision Process. In International Conference on Algorithms and Architectures for Parallel Processing (pp. 466-477). Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-03095-6_45.