UNIVERSITÉ DU QUÉBEC À MONTRÉAL

RÉSOLUTION EFFICACE DE PROCESSUS DÉCISIONNELS DE MARKOV PAR L'EXPLOITATION D'APPROCHES STRUCTURELLES ET ALGORITHMIQUES TIRANT PARTI DE L'ARCHITECTURE MODERNE DES ORDINATEURS

THÈSE

PRÉSENTÉE

COMME EXIGENCE PARTIELLE

DU DOCTORAT EN INFORMATIQUE

PAR

JAËL CHAMPAGNE GAREAU

AVRIL 2024

REMERCIEMENTS

J’aimerais commencer par remercier mes directeurs de recherche, les professeurs Éric Beaudry et Vladimir Makarenkov, qui m’ont recruté puis soutenu, tant académiquement que personnellement, tout au long de mon doctorat et de ma maîtrise. Chacun d’eux a, à sa manière, contribué à ma réussite en me poussant à me dépasser, par exemple en m’encourageant à viser des publications dans des conférences de haut niveau. Pour cela et pour leur grande disponibilité tout au long de mon parcours, je leur en suis très reconnaissant.

Je remercie tous les collègues universitaires de mon laboratoire de recherche, anciens et actuels. Entre autres, Éric, Mathieu, Jonathan, Jean, Benoît et Gabrielle. Une mention particulière à Guillaume, qui a d’abord été un stagiaire durant ma maîtrise, et qui est maintenant un collègue et un ami. Guillaume est coauteur de certains articles avec moi et sa contribution a été essentielle, entre autres pour l’implémentation de l’algorithme LAO*. Il m’a aidé de nombreuses fois pour toutes sortes d’aventures (avec Bertha et Red, entre autres), et pour cela je lui en suis très reconnaissant. Je tiens également à remercier Marc-André, qui m’a laissé poursuivre la recherche qu’il avait entamée durant sa maîtrise sur la planification d’initinéraires multiagents pour véhicules électriques (qui est elle-même une suite de ma propre recherche de maîtrise), ce qui a mené à un article à la conférence AAMAS 2024.

Je remercie également le *Conseil de recherches en sciences naturelles et en génie du Canada* (CRSNG) et le *Fonds de recherche du Québec – Nature et Technologie* (FRQNT) pour m’avoir soutenu financièrement durant mon doctorat via, respectivement, la *bourse d’études supérieures du Canada Alexandre-Graham-Bell pour le doctorat* (Programme BESC-D, N° de la demande : BESCD3 - 534817 - 2019) et la *bourse de doctorat en recherche* (Programme B2X, N° de dossier : 284673). Grâce à ce soutien, j’ai pu me concentrer à temps plein sur mon doctorat malgré les hausses incessantes du coût de la vie.

Sur un plan plus personnel, je tiens finalement à exprimer ma profonde gratitude envers ma grand-mère, mes parents, mon frère, ma soeur et le reste de la famille pour leur soutien inébranlable tout au long de mon (long !) voyage académique. Leurs encouragements constants, leur compréhension et leur amour ont été les piliers sur lesquels j’ai pu construire ma carrière académique. Leur foi en moi a été une source d’inspiration inestimable, et je suis reconnaissant pour leur soutien qui a rendu cette réalisation possible.

DÉDICACE

À Bertrand,
où que tu sois.

À Katia,
qui me supporte.

À Blocksworld,
sans lequel la planification
ne serait pas la même.

AVANT-PROPOS

J'ai découvert le domaine de la planification automatique durant ma maîtrise, d'abord grâce à mon projet de recherche de l'époque, la planification d'itinéraires pour véhicules électriques (Champagne Gareau, 2019), et ensuite grâce au cours *DIC938K : Planification en intelligence artificielle* donné par mon directeur de recherche. C'est durant cette période que j'ai commencé à m'intéresser à la planification automatique sous incertitude, et plus précisément aux processus décisionnels de Markov.

Durant la même période, j'ai été souvent auxiliaire d'enseignement — et chargé de cours dans le premier cas — pour le cours *INF3105 : Structures de données et algorithmes* et le cours *INF5130 : Analyse et conception d'algorithmes*. Cela m'a permis de développer un intérêt, voire une obsession, pour la conception de programmes efficaces, autant algorithmiquement qu'au niveau de l'implémentation. En raison de ce nouvel intérêt, j'ai suivi certains cours en ligne, dont le cours *LAFF-On Programming for High Performance* de l'Université du Texas à Austin [1], où j'ai pu approfondir ma compréhension de concepts tels que la hiérarchie de mémoire ou encore les différents niveaux de parallélisme des ordinateurs.

En 2019, lorsque vint le moment de m'inscrire au doctorat et de choisir mon sujet de recherche, j'ai consulté quelques articles scientifiques récents en ligne. J'ai alors découvert plusieurs contributions récentes d'apprentissage automatique qui considèrent des éléments tels que les instructions et les types de données de bas niveau ou encore la mémoire cache dans la conception et l'implémentation des algorithmes du domaine. La considération de ces éléments menait à des algorithmes étant plusieurs ordres de grandeur plus rapides qu'auparavant. Cependant, ces éléments semblaient avoir peu été considérés dans l'autre grande branche de l'intelligence artificielle, la planification.

C'est ainsi que l'idée m'est venue de combiner mon double intérêt pour les processus décisionnels de Markov et pour la programmation haute-performance ; mon projet de recherche était né : *la considération d'éléments tels que la hiérarchie de mémoire et le parallélisme dans la conception et l'implémentation d'algorithmes pour trouver une politique optimale dans un processus décisionnel de Markov*.

1. https://www.cs.utexas.edu/users/flame/laff/pfhp/

TABLE DES MATIÈRES

REMERCIEMENTS ii

DÉDICACE iii

AVANT-PROPOS iv

LISTE DES FIGURES vii

LISTE DES TABLEAUX ix

LISTE DES ALGORITHMES x

RÉSUMÉ xi

ABSTRACT xiii

INTRODUCTION 1

CHAPITRE 1 PROCESSUS DÉCISIONNELS DE MARKOV 7

1.1 Types de PDM 7

1.2 Algorithmes de base 14

1.2.1 *Value Iteration* 14

1.2.2 *Value Iteration* Asynchrone 15

1.2.3 *Policy Iteration* 16

CHAPITRE 2 APPROCHES EXISTANTES VISANT À AMÉLIORER LES ALGORITHMES CLASSIQUES 18

2.1 Méthodes par priorisation 18

2.2 Méthodes heuristiques 22

CHAPITRE 3 ARCHITECTURE MODERNE DES ORDINATEURS 27

3.1 Hiérarchie de mémoire 27

3.2 Différents niveaux de parallélisme 30

3.2.1 Parallélisme d'instructions 30

3.2.2 Parallélisme de données 33

3.2.3 Parallélisme de fils d'exécution 34

CHAPITRE 4 REPRÉSENTATION EFFICACE DES PROCESSUS DÉCISIONNELS DE MARKOV 36

4.1 Analyse des implémentations existantes 37

4.2 Représentation CSR-MDP 38

4.3 Évaluation 41

CHAPITRE 5 PLANIFICATEURS PARALLÈLES DE PDM 48

5.1 Algorithmes parallèles existants 48

5.2 *Parallel-Chained* TVI (pcTVI) 49

5.3 Évaluation 52

CHAPITRE 6 RÉORDONNANCEMENT D'ÉTATS 55

6.1 Méthodes existantes 55

6.2 Algorithme eTVI 56

6.3 Algorithme eiTVI 59

6.4 Évaluation 61

CHAPITRE 7 GÉNÉRATION SYNTHÉTIQUE DE PDM 71

7.1 Caractéristiques topologiques des PDM 72

7.2 Algorithme de génération d'instances synthétiques 74

CONCLUSION 78

ANNEXE A PROGRAMME MDPTK 80

A.1 Implémentation 80

A.2 Utilisation 81

RÉFÉRENCES 84

LISTE DES FIGURES

Figure 0.1 Exemple d'une instance du problème de planification de chemin couvrant dans une grille . 2

Figure 0.2 Représentation visuelle d'une configuration du jeu Tetris 3

Figure 1.1 Exemple d'un PDM-PCCS 9

Figure 2.1 Illustration d'un ordre de balayage sous-optimal 19

Figure 2.2 Exemple de décomposition d'un PDM-PCCS par l'algorithme TVI. 22

Figure 2.3 Illustration du concept d'élagage d'espace d'états. 23

Figure 3.1 Hiérarchie de mémoire des ordinateurs modernes. 28

Figure 3.2 Représentation schématique de l'architecture Skylake d'Intel. 31

Figure 4.1 Représentation visuelle de la structure CSR-MDP proposée. 39

Figure 4.2 Exemple d'une instance de PDM et la représentation CSR-MDP correspondante. 40

Figure 4.3 Évaluation de CSR-MDP sur le domaine *Layered* avec 10 couches. 45

Figure 4.4 Évaluation de CSR-MDP sur le domaine *Layered* avec 1M d'états. 45

Figure 4.5 Évaluation de CSR-MDP sur le domaine *SAP*. 46

Figure 4.6 Évaluation de CSR-MDP sur le domaine *Wetfloor*. 46

Figure 5.1 Exemple de l'exécution de l'algorithme pcTVI 51

Figure 5.2 Instance du domaine de planification *Chained-MDP*. 52

Figure 5.3 Évaluation de pcTVI sur le domaine *Chained-MDP* avec 32 chaînes. 53

Figure 5.4 Évaluation de pcTVI sur le domaine *Chained-MDP* avec 1M états. 54

Figure 6.1 Comparaison visuelle entre TVI, eTVI et eiTVI. 60

Figure 6.2 Illustration de divers ordres de balayage dans une CFC. 60

Figure 6.3 Évaluation de eTVI et eiTVI sur le domaine *Layered* avec 10 couches. 63

Figure 6.4 Évaluation de eTVI et eiTVI sur le domaine *Layered* avec 1M d'états. 63

Figure 6.5 Évaluation de eTVI et eiTVI sur le domaine *SAP*. 64

Figure 6.6 Évaluation de eTVI et eiTVI sur le domaine *Wetfloor*. 64

Figure 7.1 Exemple d'une instance de PDM et de sa déterminisation *all-outcomes*. 73

Figure 7.2 Exemple d'un graphe synthétique généré par le modèle d'Erdös-Rényi. 77

Figure 7.3 Exemple d'un PDM synthétique généré à partir du graphe de la figure 7.2. 77

Figure A.1 Options du planificateur `MDPtk` 81

Figure A.2 Fichier d'entrée représentant une instance de PDM 82

Figure A.3 Format de sortie de la fonction de valeur et de la politique optimale (V^* et π^*) 83

Figure A.4 Affichage détaillé des évènements de journalisation lors de l'exécution de `MDPtk` 83

LISTE DES TABLEAUX

Table 3.1 Vitesse, taille et latence de différents types de mémoire 29

Table 4.1 Facteurs d'accélération moyens obtenus par la représentation mémoire CSR-MDP. 43

Table 4.2 Résultats détaillés de l'évaluation de la représentation CSR-MDP. 44

Table 6.1 Comparaison de la performance de VI, LRTDP, ILAO*, TVI, eTVI et eiTVI. 65

Table 6.2 Évaluation détaillée de la performance de TVI, eTVI et eiTVI. 67

Table 6.3 Facteurs d'accélération moyens obtenus entre VI, TVI, eTVI et eiTVI. 68

Table 6.4 Comparaison des accès en mémoire cache de TVI, eTVI et eiTVI. 68

Table 7.1 Méthodes de génération de graphes synthétiques. 75

LISTE DES ALGORITHMES

Algorithme 1.1 *Value Iteration* (synchrone). 15

Algorithme 1.2 *Value Iteration* Asynchrone. 16

Algorithme 1.3 *Policy Iteration*. 17

Algorithme 2.1 *Prioritized Value Iteration*. 19

Algorithme 2.2 *Partitioned Value Iteration*. 21

Algorithme 2.3 *Topological Value Iteration*. 21

Algorithme 2.4 LAO*. 24

Algorithme 2.5 *Real-Time Dynamic Programming*. 25

Algorithme 5.1 *Parallel-Chained Topological Value Iteration*. 50

Algorithme 6.1 *Cache-Efficient Topological Value Iteration*. 58

Algorithme 7.1 Génération d'une PDM-PCCS synthétique. 76

RÉSUMÉ

Cette thèse présente des contributions en planification automatique sous incertitude, un domaine de l'intelligence artificielle. Ce domaine s'intéresse principalement au calcul de politiques optimales permettant à un agent d'atteindre un objectif lorsque de l'incertitude, due à l'agent lui-même ou à l'environnement, est présente. Les travaux présentés dans cette thèse portent plus précisément sur les processus décisionnels de Markov (PDM) qui permettent de modéliser les problèmes de planification probabiliste, c'est-à-dire les problèmes de planification pour lesquels de l'incertitude est présente, mais que cette dernière peut être exprimée par un modèle probabiliste connu.

Plusieurs algorithmes de planification, ayant des stratégies variées, permettent de calculer une politique optimale d'un PDM. Cependant, vu les instances de problèmes de planification de plus en plus grandes que l'on souhaite pouvoir résoudre, les méthodes actuelles ne suffisent pas : elles ne sont pas assez rapides. Une façon d'accélérer les calculs passe par une meilleure exploitation de l'architecture des processeurs modernes lors de la conception et de l'implémentation des algorithmes. En effet, tirer profit d'éléments tels que les différents niveaux de parallélisme accessibles (comme le parallélisme d'instructions et le parallélisme de fils d'exécution) ou encore la hiérarchie de mémoire a mené à des gains de performance significatifs dans plusieurs branches de l'informatique. Par exemple, les processeurs de type GPU, autrefois spécifiquement optimisés pour des tâches hautement parallélisables pour des rendus graphiques, sont maintenant largement utilisés pour réduire significativement le temps d'apprentissage des réseaux de neurones artificiels.

Cette stratégie reste cependant quasi inédite en planification. Cette thèse présente des contributions visant à combler ce manque dans la littérature en améliorant l'implémentation des algorithmes de planification de MDPs de sorte à exploiter les caractéristiques des processeurs modernes.

La première contribution est une représentation mémoire, appelée CSR-MDP, indépendante du choix de l'algorithme de résolution utilisé pour trouver la politique optimale du PDM. Cette structure vise à minimiser la consommation en mémoire et le nombre de défauts de cache lors du calcul d'une politique optimale. Cette méthode a permis des gains d'un facteur allant de 2.8 à 8.6, selon l'algorithme utilisé.

La seconde contribution est un algorithme basé sur l'algorithme *Topological Value Iteration* (TVI), que nous appelons pcTVI (pour *parallel-chained* TVI). Cet algorithme permet d'exploiter plusieurs cœurs d'un CPU en parallélisant les calculs lors de la recherche d'une politique optimale. Les planificateurs parallèles existants ont pour désavantage de soit nécessiter une décomposition manuelle de l'espace d'états dépendante de chaque domaine de planification, ou de nécessiter beaucoup de communications inter fils d'exécution, ce qui réduit le gain de vitesse possible. L'algorithme proposé, pcTVI, a comme principal atout de n'avoir aucun de ces deux désavantages. Il a permis un facteur 20 d'accélération sur un ordinateur ayant 32 cœurs de calcul sur le domaine de planification testé.

La troisième contribution consiste en deux algorithmes, eTVI et eiTVI, permettant d'exploiter au maximum la représentation mémoire CSR-MDP en réordonnant les états du PDM de sorte que les états qui doivent souvent être accédés consécutivement soient le plus près possible les uns des autres en mémoire. Ce faisant, le pourcentage de données utiles aux calculs actuels lors du chargement d'une ligne de cache augmente et le nombre de défauts de cache diminue, ce qui réduit également le temps de calcul. Ces algorithmes seuls ont permis des gains de vitesses d'un facteur 2.1 en moyenne sur les domaines testés.

Ces trois contributions peuvent être combinées. Le planificateur en résultant a permis de diminuer le temps de calcul de plusieurs ordres de grandeur sur les instances des domaines de planification testés.

Les domaines de planifications probabilistes standardisés — par exemple, ceux utilisés lors de la compétition internationale de planification, l'IPC, ayant lieu à la conférence ICAPS — ne couvrent pas toutes les propriétés topologiques possibles des PDM, ce qui peut rendre difficile la comparaison des algorithmes existants dans des contextes variés. Pour pallier ce problème, une quatrième contribution est proposée : un algorithme de génération de PDM synthétiques, pouvant générer des instances couvrant un vaste éventail de propriétés topologiques.

Mots clés : intelligence artificielle, planification automatique, incertitude, calcul haute-performance, hiérarchie de mémoire, mémoire cache, optimisation, parallélisme, partitionnement, priorisation, opérations vectorielles, SIMD, MDP, Processus décisionnels de Markov, structures de données.

ABSTRACT

This thesis presents contributions in automated planning under uncertainty, a field of artificial intelligence. This field is primarily concerned with the calculation of optimal policies that allow an agent to achieve a goal when uncertainty, due to the agent itself or the environment, is present. The work presented in this thesis focuses specifically on Markov decision processes (MDPs) that allow for the modeling of probabilistic planning problems, i.e., planning problems where uncertainty is present, but where it can be expressed by a known probabilistic model.

Several planning algorithms, with sometimes very distinct strategies, allow for the calculation of an optimal policy of an MDP. However, given the increasingly large planning problem instances that we wish to solve, current methods are not sufficient : they are not fast enough. One way to speed up calculations is to exploit the architecture of modern processors when designing and implementing algorithms. Taking advantage of elements such as the different levels of accessible parallelism (for example, instruction-level parallelism and thread-level parallelism) or the memory hierarchy has led to significant speedups in several branches of computer science such as machine learning. For example, GPUs, once specifically optimized for highly parallelizable tasks for graphical rendering, are now widely used to significantly reduce the learning time of artificial neural networks.

This strategy remains, however, almost unheard of in planning. This thesis presents contributions aimed at filling this gap in the literature by improving the implementation of MDP planning algorithms to exploit the characteristics of modern processors.

The first contribution is a memory representation, called CSR-MDP, independent of the choice of MDP resolution algorithms used to find an optimal policy. This structure aims to minimize memory consumption and the number of cache misses during the calculation of an optimal policy. This method has allowed for speedup factors ranging from 2.8 to 8.6, depending on the algorithm used.

The second contribution is an algorithm based on the *Topological Value Iteration* (TVI) algorithm, which we call pcTVI (for *parallel-chained* TVI). This algorithm allows for the exploitation of multiple CPU cores by parallelizing calculations during the computation of an optimal policy. Existing parallel planners have the disadvantage of either requiring a manual decomposition of the state space dependent on each planning domain, or requiring a lot of interprocess communication, which reduces the possible speed gain. The proposed algorithm, pcTVI, has the main advantage of having neither of these two disadvantages. It allowed for a 20-fold acceleration on a computer with 32 computing cores on the planning domain tested.

The third contribution consists of two algorithms, eTVI and eiTVI, which maximize the use of the CSR-MDP memory representation by reordering the states of the MDP such that states that often need to be accessed consecutively are as close as possible to each other in memory. By doing so, the percentage of data useful for current calculations when loading a cache line increases and the number of cache misses decreases, which also reduces the calculation time. These algorithms alone achieved an average speedup of 2.1 times accross the tested domains.

These three contributions can be combined. The resulting planner has allowed for a decrease in the calculation time by several orders of magnitude on instances of the planning domains considered.

Standardized probabilistic planning domains — e.g., those used in the International Planning Competition, IPC, held at the ICAPS conference — do not cover all possible topological properties of MDPs, which can make it difficult to compare existing algorithms in various contexts. To address this issue, a fourth contribution is proposed : a synthetic MDP generation algorithm that can generate instances covering a wide range of topological properties.

Keywords : artificial intelligence, automated planning, uncertainty, high-performance computing, memory hierarchy, cache memory, optimization, parallelism, partitioning, prioritization, vector operations, SIMD, MDP, Markov decision processes, data structures.

INTRODUCTION

L'**intelligence artificielle** (IA) s'intéresse à concevoir des **agents intelligents**, c'est-à-dire des systèmes autonomes quelconques capables de percevoir leur environnement via des capteurs et d'effectuer des actions via des actuateurs. On peut répartir les différentes approches d'IA en quatre catégories, selon l'objectif qu'on souhaite atteindre (Russell et Norvig, 2020). Historiquement, l'objectif de l'IA était d'avoir un agent qui *agit humainement*, ce qu'on peut établir avec par exemple le test de Turing (Turing, 1950), qui évalue si des participants sont capables de différencier un humain d'un agent artificiel en communiquant textuellement. En informatique cognitive, les chercheurs s'intéressent davantage aux agents qui *réfléchissent humainement*. Puisque les humains sont faillibles et ne réfléchissent ou n'agissent pas toujours de façon rationnelle, on peut plutôt chercher à avoir des agents qui *réfléchissent rationnellement* ou qui *agissent rationnellement*. Dans le premier cas, l'agent raisonne (par exemple en faisant des déductions à l'aide de syllogismes logiques) pour découvrir de nouveaux faits. Le dernier cas (agir rationnellement) est le paradigme qui prévaut le plus de nos jours en IA, de même qu'en recherche opérationnelle, en économie, et dans plusieurs autres domaines.

On ne sait pas encore comment concevoir une machine intelligente. Toutefois, il existe plusieurs approches variées qui s'intéressent à reproduire certaines capacités associées à l'intelligence. L'**apprentissage machine** s'intéresse aux agents qui peuvent *apprendre*. Cette branche permet de réaliser des tâches telles que la classification, le partitionnement de données (*clustering*) et la détection d'anomalies (Goodfellow *et al.*, 2016). Le **traitement automatique du langage naturel** est une branche qui s'intéresse quant à elle aux différentes façons de communiquer, et vise à résoudre des tâches telles que la synthèse vocale, la reconnaissance vocale, la traduction automatique ou encore la génération automatique de textes (Jurafsky et Martin, 2009).

Une autre des branches de recherche en IA se nomme **planification automatique** (*automated planning & scheduling*). Dans celle-ci, on s'intéresse généralement à trouver des plans (c.-à-d., des séquences d'actions) optimaux permettant à un agent d'atteindre un certain objectif (Ghallab *et al.*, 2016; LaValle, 2006). On peut diviser la planification automatique en deux grandes branches : la planification déterministe et la planification probabiliste.

Dans le premier cas, on considère que les actions qui peuvent être exécutées par l'agent sont déterministes, c'est-à-dire qu'elles ont un effet préalablement connu qui est toujours le même. Le problème de planification d'itinéraires pour véhicules électriques (Champagne Gareau, 2019) est un exemple de problème où l'on

peut supposer que les actions (par exemple, se déplacer, ou se recharger) sont déterministes, bien qu'on puisse imaginer des variantes du problème où de l'incertitude est présente. Le problème de planification de chemin couvrant (Champagne Gareau *et al.*, 2023a) est un autre exemple de problème de planification déterministe. Dans celui-ci, on cherche à trouver un chemin qui couvre entièrement une région donnée en utilisant le moins de ressources possible, par exemple de l'énergie ou du temps. La figure 0.1 montre une instance d'un tel problème dans une grille.

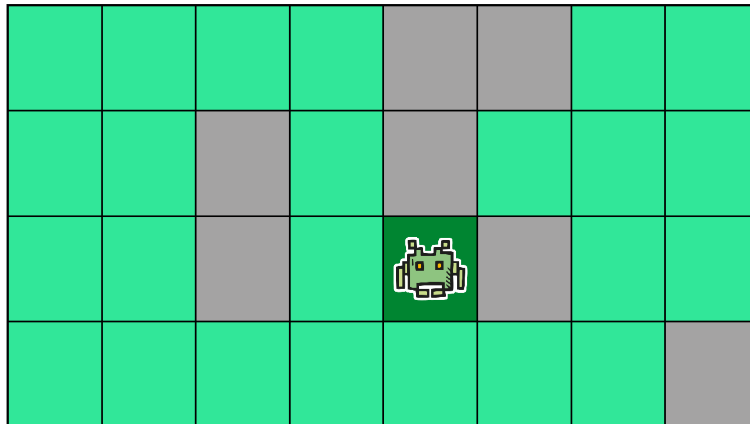


Figure 0.1 – Exemple d'une instance du problème de planification de chemin couvrant dans une grille. Les cases grises représentent des endroits inaccessibles, tandis que les cases vert pâle sont les cases que l'agent doit couvrir.

Lorsque l'effet de certaines actions est incertain, mais qu'on peut quantifier cette incertitude, on parle plutôt de problèmes de planification probabiliste (Mausam et Kolobov, 2012). On peut modéliser ces problèmes à l'aide d'un **processus décisionnel de Markov** (PDM) [1]. Ceux-ci permettent de modéliser des situations dans des domaines aussi variés que la finance (Bäuerle et Rieder, 2011), la neuroscience (Suchow et Griffiths, 1965), les jeux de hasard (Dubins *et al.*, 2002), le marketing (Howard, 2002), la théorie du contrôle (Marcus *et al.*, 1997) et le déplacement de robots (Bernstein *et al.*, 2001). Un exemple de problème de planification probabiliste est une variante du problème de planification de chemin couvrant mentionné précédemment, où certaines des cases de la grille sont glacées. Lorsque l'agent tente de se déplacer sur une case glacée, il y a une probabilité qu'il dérape et se retrouve sur une case adjacente.

Plusieurs des problèmes de planification — qu'ils soient déterministes ou probabilistes — sont représentés par des millions, des milliards, voire encore plus d'états, provenant parfois d'une discrétisation d'un environnement continue, ou simplement d'une explosion combinatoire inhérente au problème. À titre d'exemple,

1. Aussi appelé en français *processus de décision markovien*. Nous utiliserons la première nomenclature dans cette thèse. Le nom anglais de ce concept, très répandu, est *Markov Decision Process* (MDP).

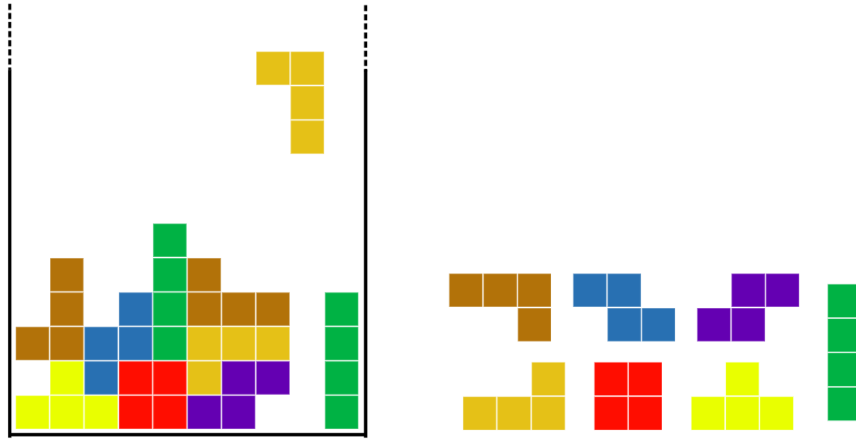


Figure 0.2 – Représentation visuelle d'une configuration possible d'une partie du jeu Tetris (gauche) et des différentes pièces (tétraminos) possibles (droite).

le jeu Tetris, créé par Alekseï Pajitnov en 1984 et représenté à la figure 0.2, consiste en général en une grille de 20 rangées et 10 colonnes dans laquelle des pièces (appelées *tétraminos*) tombent. L'objectif est de déplacer latéralement les tétraminos et de leur faire faire des rotations pendant leur chute de sorte à remplir des rangées dans le bas de l'écran. Lorsqu'une rangée est pleine, elle disparait et les rangées au-dessus sont déplacées d'une rangée vers le bas. Le jeu termine lorsqu'il n'y a plus assez d'espace dans le haut de l'écran pour qu'un nouveau tétramino puisse apparaître. Puisqu'on ne connait pas à l'avance le prochain tétramino qui apparaîtra au sommet, le problème de trouver une stratégie optimale pour survivre le plus longtemps possible peut être vu comme un problème de planification probabiliste — où les actions probabilistes sont celles qui font atterrir un tétramino et, donc, qui causent la génération d'un nouveau tétramino aléatoire — modélisable à l'aide d'un PDM (Scherrer *et al.*, 2015). On peut définir un état comme étant la configuration de la grille, la pièce actuelle, et la pièce suivante (on connait en général un tétramino d'avance, dépendamment des variantes du jeu). Comme chaque case dans la grille est occupée ou non, qu'il y a 7 types de tétraminos et qu'il y a $10 \times 20 = 200$ cases, on peut donc dire qu'il y a au total $7 \times 7 \times 2^{200} \approx 10^{62}$ états.

Au vu de l'explosion du nombre d'états dans un problème relativement simple tel que Tetris, on peut facilement imaginer que la majorité des problèmes réels qu'on souhaite résoudre possèdent également une quantité considérable d'états. La complexité spatiale est principalement un problème, puisque stocker explicitement 10^{62} états, même en supposant un coût minime d'un octet par état, nécessiterait plus de mémoire que ce que contient n'importe quel ordinateur. En supposant qu'on puisse élaguer l'espace d'états des PDM d'intérêt (ce qui est possible, par exemple en utilisant des algorithmes de recherche heuristiques), et qu'on puisse donc stocker celui-ci, la complexité temporelle demeurera un enjeu. En effet, nous sommes parfois

soumis à des contraintes de temps souvent rigides lorsqu'on souhaite trouver une politique optimale ou quasi optimale (ex. : un robot extraplanétaire qui doit prendre rapidement une décision sur sa prochaine action avant l'arrivée d'une tempête), il va sans dire que la rapidité du temps de calcul d'une solution optimale d'un PDM est donc une caractéristique critique d'un bon planificateur.

Malgré que plusieurs algorithmes récents aient été proposés pour résoudre des PDM plus rapidement, le problème reste difficile. On peut classifier les algorithmes proposés en deux grandes catégories : les méthodes heuristiques et les méthodes par priorisation. Or, une approche orthogonale, presque jamais tentée en planification, est également possible : l'exploitation d'éléments d'architecture des processeurs modernes lors de la conception et de l'implémentation d'algorithmes. En effet, tirer profit au maximum d'éléments tels que la hiérarchie de mémoire des ordinateurs ou encore les différents niveaux de parallélisme a permis à la communauté de chercheurs dans plusieurs domaines une grande amélioration de la performance de certains algorithmes, que ce soit les algorithmes de tri (LaMarca et Ladner, 1999), les algorithmes dans les graphes (Park *et al.*, 2004) ou encore les algorithmes utilisant des matrices (Goto et Van de Geijn, 2002). Dans ce dernier cas, l'utilisation intelligente des différentes mémoires caches combinée à l'utilisation des opérations vectorielles a permis des gains allant jusqu'à plusieurs ordres de grandeur (Goto et Van De Geijn, 2008). Plusieurs autres exemples de tâches informatiques où ces éléments ont été exploités existent, tels que le décodage de tableaux d'entiers (Lemire et Boytsov, 2015) ou l'analyse syntaxique d'URLs (Nizipli et Lemire, 2023) et de fichiers JSON (Langdale et Lemire, 2019).

Au cours des dernières années, ces éléments ont été considérés dans d'autres branches de l'IA et ont permis des vitesses de calcul étant de plusieurs ordres de grandeur plus rapides. Par exemple, l'utilisation de nombres à virgule flottante spécialisés (*bfloat*) et l'utilisation d'instructions SIMD adaptées à ces nombres ont permis des gains substantiels dans plusieurs applications d'apprentissage machine (Henry *et al.*, 2019; Burgess *et al.*, 2019). De plus, le parallélisme, obtenu à la fois sur CPU ou GPU, et des stratégies de stockage en mémoire tel que le *tiling* ont permis d'entraîner des réseaux de neurones artificiels plus rapidement permettant de résoudre des problèmes de plus grande taille (Raina *et al.*, 2009; Guo *et al.*, 2020).

Au vu de l'amélioration considérable des performances obtenue en apprentissage machine, il est raisonnable d'espérer des gains similaires en exploitant ces mêmes éléments dans l'implémentation d'algorithmes de planification automatique. Cela peut mener à des planificateurs de PDM capables de résoudre des instances de problème beaucoup plus grandes que ce qui était possible. L'objectif principal de la recherche

doctorale qui a mené à cette thèse était donc d'explorer la faisabilité de développer de nouvelles méthodes générales de résolution de PDM exploitant l'architecture des ordinateurs, en vue de pouvoir résoudre des problèmes de planification probabiliste de plus grande taille dans un temps raisonnable.

Le **chapitre 1** introduit les différents types de processus décisionnels de Markov, présente quelques résultats théoriques, et présente les algorithmes classiques permettant de trouver une politique optimale d'un PDM.

Plusieurs stratégies ont été proposées pour permettre de résoudre des PDM plus rapidement. Le **chapitre 2** fait un survol de l'état de l'art sur le sujet et décrit plus en détail les deux principales catégories de méthodes : les méthodes heuristiques et les méthodes par priorisation.

Le **chapitre 3** présente quant à lui des éléments de l'architecture des processeurs modernes, en passant par la hiérarchie de mémoire ainsi que par les différents types de parallélisme disponibles, permettant ainsi d'aborder les gains de vitesses théoriquement atteignables en exploitant au maximum ces éléments.

Bien que les articles scientifiques parlant des PDM décrivent souvent très bien les algorithmes proposés, les détails d'implémentations sont rarement abordés, et pire, leur code source est souvent non publié ou indisponible. En effet, les structures de données utilisées sont rarement mentionnées, alors que le choix de celles-ci peut avoir un impact considérable sur la performance finale. Le **chapitre 4** présente et compare les structures de données utilisées par les quelques implémentations disponibles en ligne — ou ayant pu être obtenues en contactant les auteurs — pour représenter en mémoire les PDM. Il décrit également la première contribution originale de cette thèse (Champagne Gareau *et al.*, 2022), une structure de données appelée CSR-MDP et permettant le stockage d'un PDM en mémoire de façon plus économe en termes de consommation spatiale et plus efficace au niveau de la mémoire cache, ce qui permet aux PDM stockés sous cette forme d'atteindre une vitesse d'un ordre de grandeur plus grande, en moyenne, par rapport aux structures existantes.

Quelques rares algorithmes de PDM utilisant le parallélisme de fils d'exécution sont présents dans la littérature. Or, aucun d'eux n'est à la fois utilisable automatiquement indépendamment du domaine et sans communication inter fils d'exécution. Dans le **chapitre 5**, une seconde contribution originale est présentée, un algorithme appelé pcTVI qui est le premier dans la littérature à avoir à la fois les deux propriétés susmentionnées (Champagne Gareau *et al.*, 2023b).

La représentation en mémoire CSR-MDP proposée au chapitre 4 peut fonctionner avec n'importe lequel des algorithmes existants de calcul d'une politique optimale d'un PDM. Or, on peut se demander s'il est possible de concevoir un algorithme spécifiquement pensé pour tirer parti de cette dernière au maximum. Le **chapitre 6** présente notre troisième et principale contribution originale à la recherche, qui a été présentée à la *European Conference of Artificial Intelligence* (ECAI) (Champagne Gareau *et al.*, 2023c). Cette contribution consiste en deux algorithmes, eTVI et eiTVI. Le premier vise à exploiter la représentation CSR-MDP en augmentant la quantité d'information utile dans chaque ligne de cache chargée — diminuant ainsi le nombre de défauts de cache, ce qui augmente la vitesse globale du calcul —, tandis que le second vise à améliorer la propagation des valeurs dans l'espace d'états, tout en maintenant une bonne performance en mémoire cache.

Lorsqu'un nouvel algorithme de planification pour PDM est proposé, il peut être difficile de comparer sa performance aux algorithmes déjà présents dans la littérature. En effet, la performance relative d'un algorithme par rapport aux autres sur une instance de PDM peut dépendre de plusieurs facteurs, tels que les caractéristiques topologiques du domaine de planification d'intérêt. Le **chapitre 7** propose une quatrième contribution originale à la recherche (Champagne Gareau *et al.*, 2024) visant à combler un manque de variété topologique dans les domaines existants. Il identifie d'abord certaines caractéristiques topologiques importantes des PDM, et présente une méthode générative d'instances synthétiques ayant des caractéristiques topologiques contrôlables et diverses, permettant de mieux comparer les performances des algorithmes de planification dans des scénarios variés.

Les contributions théoriques présentées dans les divers chapitres, de même que plusieurs algorithmes de l'état de l'art, ont été implémentés dans une bibliothèque C++, appelée `MDPTk`. Le code source est disponible en ligne [2]. Certains détails d'implémentation et d'utilisation de cette bibliothèque sont présentés à l'annexe A. Les articles, diapositives, affiches, et autres productions ayant découlé de la recherche présentée dans cette thèse sont également disponibles en ligne [3].

2. https://gitlab.info.uqam.ca/champagne_gareau.jael/mdptk

3. https://www.jaelgareau.com/fr/project/mdp

CHAPITRE 1
PROCESSUS DÉCISIONNELS DE MARKOV

> *My first task in dynamic programming was to put it on a rigorous basis. I found that I was using the same technique over and over again to derive a functional equation. I decided to call this technique 'The principle of optimality'.*
>
> — Richard Bellman (Dreyfus, 2002)

Ce chapitre présente les définitions et résultats fondamentaux nécessaires pour aborder la théorie des processus décisionnels de Markov (PDM). Il présente entre autres les trois principaux types de PDM, ainsi que le principe d'optimalité, un résultat fondamental sur lequel sont basées les méthodes de programmation dynamiques permettant de trouver une solution. Il introduit finalement deux algorithmes classiques de résolution d'un PDM, *Value Iteration* (VI) et *Policy Iteration* (PI). Les définitions et les résultats présentés dans ce chapitre sont autosuffisants, mais le lecteur souhaitant une présentation plus détaillée est invité à lire le livre d'introduction aux PDM de Mausam et Kolobov (2012), principalement les chapitres 2 et 3.

1.1 Types de PDM

Il existe plusieurs types de PDM. Certains sont plus utilisés en apprentissage par renforcement (par exemple, les PDM à horizon infini avec escomptes) tandis que d'autres, tel que les PDM de plus court chemin stochastique, sont plus utilisés en planification automatique. Bien que les deux branches de recherches utilisent des PDM, les problématiques techniques et les divers enjeux sont généralement très différents. Le livre de Sutton et Barto (2018) donne un bon aperçu des PDM sous l'angle de l'apprentissage par renforcement. Dans cette thèse, nous considérons toutefois les PDM sous l'angle de la planification automatique. Les trois classes de PDM les plus communes sont présentées aux définitions 1.1, 1.2 et 1.3.

Définition 1.1. *Un* ***PDM à horizon fini*** *(PDM-HF) est un quintuplet* (S, A, D, T, R) *où :*

- S *est l'ensemble fini des états (aussi appelé l'*espace d'états*) ;*
- A *est l'ensemble fini des actions que l'agent peut exécuter ;*
- $D = (0, 1, \dots, |D|)$ *est la suite fini des pas de temps (time steps) dans le système, où* $|D|$ *est appelé l'horizon du PDM ;*

— $T\colon S \times A \times S \to [0,1]$ *est la fonction de transition, où* $T(s,a,s')$ *donne la probabilité que l'agent atteigne l'état* s' *s'il exécute l'action* a *à l'état* s *;*

— $R\colon S \times A \to \mathbb{R}$ *donne la récompense obtenue en exécutant l'action* a *à l'état* s *;*

Définition 1.2. *Un* ***PDM à horizon infini avec escompte*** *(PDM-HIE) est un quintuplet* (S, A, T, R, γ) *où les éléments du quadruplet* (S, A, T, R) *ont la même signification que pour les PDM-HF introduits à la définition 1.1, mais sans limite* $|D|$ *du nombre de transitions permises, et* $\gamma \in [0, 1[$ *est un nombre appelé taux d'amortissement ou taux d'escompte (discount factor) qui permet de pondérer l'importance donnée aux récompenses à long terme par rapport aux récompenses à court terme.*

Définition 1.3. *Un* ***PDM de plus court chemin stochastique*** *(PDM-PCCS)* [1] *est un uplet* (S, A, T, C, G) *où :*

— S*,* A *et* T *ont la même signification qu'à la définition 1.1;*

— $C\colon S \times A \to \mathbb{R}^+$ *est similaire à* R*, mais formulé sous forme de coûts plutôt que de récompenses;*

— $G \subseteq S$ *est l'ensemble des états buts.*

Dans le reste de ce document, nous supposons que les PDM-PCCS n'ont qu'un seul état but, noté s_g. Dans le cas contraire, on peut créer sans perte de généralité un nouvel état but et une action déterministe, de coût nul et applicable à tous les anciens états buts, se rendant à l'état but synthétique nouvellement créé. On peut aussi supposer que s_g est un état absorbant, c'est-à-dire que pour tout $a \in A$ et $s' \in S \setminus \{s_g\}$, on a : $C(s_g, a) = 0$, $T(s_g, a, s_g) = 1$ et $T(s_g, a, s') = 0$.

Les PDM-HF permettent de modéliser les problèmes pour lesquels on souhaite maximiser une récompense dans un intervalle de temps prédéfini. Les PDM-HIE sont quant à eux utilisés dans des domaines où l'on souhaite maximiser une récompense à long terme. Par exemple, en finance, ils peuvent modéliser des stratégies d'investissements (le facteur d'escompte γ peut représenter par exemple l'inflation). Intuitivement, les PDM-PCCS modélisent quant à eux les problèmes où l'on souhaite atteindre un objectif sous le plus faible coût possible. C'est donc une généralisation du problème de plus court chemin dans un graphe, mais où les actions sont probabilistes. Topologiquement, un PDM-PCCS peut être vu comme un graphe ET/OU ou comme un hypergraphe. La figure 1.1 montre un exemple de PDM-PCCS.

Bien que certains algorithmes soient optimisés pour une de ces classes de PDM en particulier, il est possible de montrer que les algorithmes applicables aux PDM-PCCS le sont également aux PDM-HF et aux PDM-HIE. Ces deux représentations sont donc assimilables à des cas particuliers de PDM-PCCS (Bertsekas, 1995).

1. Plus connu sous son nom anglais, *stochastic shortest-path MDP* (SSP-MDP).

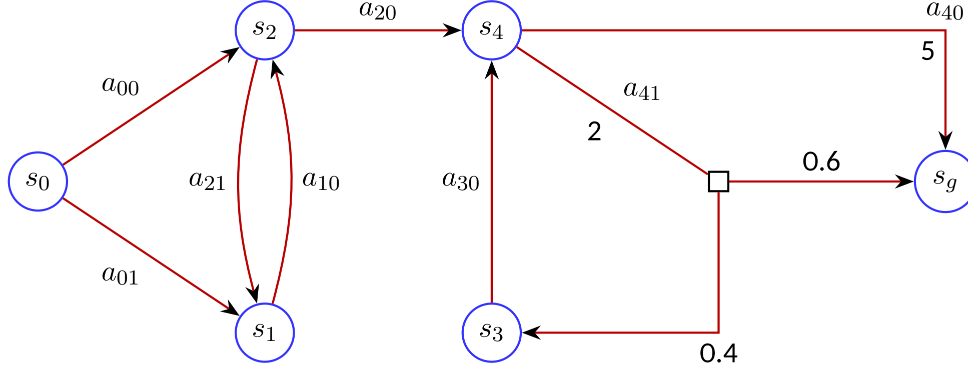


Figure 1.1 – Exemple d'un PDM-PCCS. Il n'y a qu'un seul état but, s_g. La seule action non déterministe est a_{41}. Elle mène l'agent à l'état but $60\,\%$ du temps, et à l'état s_3 $40\,\%$ du temps. Le coût est de 2 dans les deux cas. Les actions dont le coût n'est pas indiqué ont un coût de 1.

Proposition 1.4. *Il est possible de transformer les PDM-HF et les PDM-HIE en PDM-PCCS, mais pas vice-versa. Autrement dit, la modélisation sous forme de PDM-PCCS est* strictement *plus générale.*

Preuve. (1) Soit (S, A, T, R, D) un PDM-HF. Alors (S', A, T', C, G) est un PDM-PCCS équivalente, où :

— $S' = S \times \{0, 1, \ldots, |D|\}$;

— T' est tel que $\forall t \in \{0, 1, \ldots, |D| - 1\}, T'((s, t), a, (s', t + 1)) = p$ si $T(s, a, s') = p$;

— $C = -R'$ où R' est définie de la même manière que T' ;

— $G = \{(s, |D|) \mid s \in S\}$.

(2) Soit (S, A, T, R, γ) un PDM-HIE. Alors $(S \cup \{s_g\}, A', T', -R, \{s_g\})$ est un PDM-PCCS, où :

— T' est une modification de T où on ajoute une possibilité de transition à l'état s_g (avec probabilité γ) à partir de n'importe quel état en faisant n'importe quelle action (les autres probabilités de transition sont ensuite normalisées pour qu'elles totalisent $1 - \gamma$) ;

— $\forall s \in S, \forall a \in A, C'(s, a, s_g) = 0$.

(3) Soit (S, A, T, C, G), un PDM-PCCS. Celui-ci ne peut pas se convertir en PDM-HF car le nombre de transitions nécessaires pour atteindre un but $g \in G$ est en général indéfini. Ainsi, on ne peut pas fixer à priori un horizon fini. De plus, celui-ci ne peut pas se convertir en PDM-HIE, car il faudrait fixer $\gamma = 1$ pour créer une équivalence. Or, la valeur 1 est exclue de l'ensemble des valeurs possibles pour γ, par définition. ■

En raison de la proposition 1.4, nous n'allons nous intéresser explicitement qu'aux PDM-PCCS (qu'on appellera parfois simplement PDM dans le reste de ce document). Nous noterons parfois $A(s)$ l'ensemble des **actions applicables** à l'état s, c'est-à-dire les actions $a \in A$ pour lesquelles il existe $s' \in S$ où $T(s, a, s') > 0$.

En planification déterministe, une solution est un plan, c'est-à-dire une suite d'actions (Ghallab *et al.*, 2016). Cependant, en planification probabiliste, puisque les actions ne sont pas déterministes, on ne peut prévoir avec certitude que l'état qui suivra un état quelconque s_i sera l'état s_{i+1}. Ainsi, une solution ne peut pas être simplement une suite d'actions. On définit donc plutôt la solution d'un PDM par une politique (définition 1.5). On introduit aussi le concept de politique propre (définition 1.6).

Définition 1.5. *Une **politique** (markovienne et stationnaire) est une application $\pi : S \to A$ qui, pour chaque état, indique à l'agent une action qu'il devrait choisir. À moins d'avis contraire, toutes les politiques mentionnées dans cette thèse seront markoviennes et stationnaires, ce qui veut respectivement dire que le choix de la prochaine action ne dépend pas du chemin qui a mené à l'état actuel, et ne dépend pas du temps d'arrivée (la politique ne varie pas dans le temps).*

Définition 1.6. *Une **politique propre** (appelée parfois proper policy ou safe policy en anglais) est une politique pour laquelle l'agent est certain de pouvoir atteindre un but après un certain temps, peu importe son état initial (il ne restera pas coincé dans un cul-de-sac ou dans un cycle). Autrement dit, c'est une politique π pour laquelle pour tout état de départ $s_0 \in S$, il existe un nombre $m_\pi \in \mathbb{N}$ tel que*

$$\max_{s \in S} P(s_{m_\pi} \neq s_g | s_0 = s, \pi) < 1,$$

où s_{m_π} représente l'état actuel après l'exécution de m_π actions données par la politique π.

Pour comparer la qualité de deux politiques, on introduit le concept de fonction de valeur (définition 1.7). La notion de politique optimale est définie en fonction de ce concept (définition 1.8). Il est possible de montrer que dans un PDM-PCCS, une telle politique existe toujours (proposition 1.9).

Définition 1.7. *La **fonction de valeur** (value function) associée à une politique π est une fonction $V^\pi : S \to \mathbb{R}$ qui associe à chaque état s l'espérance du coût total futur si l'agent exécute indéfiniment les actions données par π à partir de s. Autrement dit, si s_0 représente un état de départ et s_i représente l'état où l'agent se trouve après l'exécution de i actions données par la politique π, alors :*

$$V^\pi(s) = \lim_{N \to \infty} \mathbb{E}\left[\sum_{i=0}^{N-1} C(s_i, \pi(s_i))\right].$$

Définition 1.8. *Une **politique optimale** d'un PDM est une politique $\pi^\star$ pour laquelle la fonction de valeur associée $V^{\pi^\star}$ (qu'on notera simplement $V^\star$) est telle que $\forall \pi, \forall s \in S, V^\star(s) \leq V^\pi(s)$, c'est-à-dire que $\pi^\star$ est une politique qui minimise l'espérance du coût total futur à partir de n'importe quel état du PDM.*

Proposition 1.9 (Principe d'optimalité dans les PDM-PCCS). *Si un PDM-PCCS n'admet que des politiques propres, alors il existe toujours une politique optimale $V^\star$ qui soit markovienne et stationnaire. De plus, la politique générée en appliquant itérativement les* ***équations de Bellman*** *:*

$$V_{i+1}(s) = \begin{cases} 0 & \text{si } s \in G, \\ \min_{a \in A} \left[C(s,a) + \sum_{s' \in S} T(s,a,s') V_i(s') \right] & \text{sinon,} \end{cases} \tag{1.1}$$

où V_0 est défini de manière arbitraire, sauf pour $V_0(s_g) = 0$, converge vers la politique optimale.

La partie entre crochets dans l'équation précédente est souvent notée $Q(s,a)$ et appelée la *Q-Valeur* d'une paire état-action. Avant de démontrer la proposition 1.9, nous prouvons d'abord un résultat intermédiaire.

Lemme 1.10. *Il existe un nombre $\rho \in \mathbb{R}^+$ tel que pour toute politique propre π et pour tout $s_0 \in S$,*

$$V^\pi(s_0) \leq \frac{m_\pi}{1-\rho} \max_{\substack{s \in S \\ a \in A(s)}} C(s,a),$$

où m_π est le nombre défini à la définition 1.6 correspondant à la politique π considérée.

Preuve. Nous noterons s_i l'état de l'agent après l'exécution de i actions données par la politique. Soit π une politique propre. Alors, par définition, il existe $m_\pi \in \mathbb{N}$ tel que

$$\rho_\pi := \max_{s \in S} P(s_{m_\pi} \neq s_g | s_0 = s, \pi) < 1. \tag{1.2}$$

Ainsi,

$$\begin{aligned} & P(s_{2m_\pi} \neq s_g | s_0 = s, \pi) \\ &= P(s_{2m_\pi} \neq s_g | s_{m_\pi} \neq s_g, s_0 = s, \pi) P(s_{m_\pi} \neq s_g | s_0 = s, \pi) && \text{(déf. probabilité conditionnelle)} \\ &= P(s_{m_\pi} \neq s_g | s_0 = s, \pi)^2 && \text{(propriété markovienne)} \\ &\leq \rho_\pi^2. && \text{(Équation 1.2)} \end{aligned}$$

On peut facilement montrer par induction que pour tout $k \in \mathbb{N}$,

$$P(s_{km_\pi} \neq s_g | s_0 = s, \pi) \leq \rho_\pi^k. \tag{1.3}$$

L'espérance du coût encourue par les m_π actions ayant eu lieu entre la km_π ième et la $((k+1)m_\pi - 1)$ ième étape est par conséquent borné par

$$\rho_\pi^k \left(m_\pi \max_{\substack{s \in S \\ a \in A(s)}} C(s,a) \right).$$

On a donc finalement que

$$\begin{aligned} V^\pi(s) &\leq \sum_{k=0}^{\infty} \rho_\pi^k \left(m_\pi \max_{\substack{s \in S \\ a \in A(s)}} C(s,a) \right) && \text{(Somme sur chaque période } k \text{ de } m_\pi \text{ actions)} \\ &= \frac{m_\pi}{1-\rho_\pi} \max_{\substack{s \in S \\ a \in A(s)}} C(s,a). && \text{(Série géométrique de raison } \rho_\pi < 1\text{)} \end{aligned}$$

Comme un PDM-PCCS possède un nombre fini d'états, le nombre de politiques propres possible est également fini. Par conséquent, le nombre $\rho := \max_\pi \rho_\pi$ satisfait l'énoncé. ■

Preuve (de la Proposition 1.9). L'idée générale est de décomposer l'espérance en deux parties : une partie finie, et une partie infinie qui tend vers zéro. Tout comme dans la preuve du lemme 1.10, on peut décomposer la suite d'états parcourus en groupes de m_π états. Par le lemme 1.10, on sait qu'un groupe de m_π états parcourus, en supposant qu'on n'atteigne pas l'état but durant ces m_π transitions, a au plus un coût de

$$M := m_\pi \max_{\substack{s \in S \\ a \in A(s)}} C(s,a).$$

On sait par le lemme 1.10 que l'espérance du coût encouru lors du K ième groupe de m_π transitions d'états (de l'état Km_π à l'état $(K+1)m_\pi - 1$) en exécutant une politique propre π est d'au plus $M\rho^k$. Ainsi,

$$\begin{aligned} \left| \lim_{N\to\infty} \mathbb{E}\left[\sum_{k=m_\pi K}^{N-1} C(s_k, \pi(s_k)) \right] \right| &\leq \sum_{k=K}^{\infty} M\rho^k && \text{(Somme sur chaque période } k \text{ de } m_\pi \text{ actions)} \\ &= M\left(\sum_{k=0}^{\infty}(\rho^k) - \sum_{k=0}^{K-1}(\rho^k) \right) \\ &= M\left(\frac{1}{1-\rho} - \frac{1-\rho^K}{1-\rho} \right) && \text{(Série géométrique)} \\ &= \frac{M\rho^k}{1-\rho}. && (1.4) \end{aligned}$$

Par conséquent,

$$\begin{aligned} V^{\pi}(s) &= \lim_{N\to\infty} \mathbb{E}\left[\sum_{k=0}^{N-1} C(s_k,\pi(s_k))\right] && \text{(définition 1.7)} \\ &= \mathbb{E}\left[\sum_{k=0}^{m_\pi K-1} C(s_k,\pi(s_k))\right] + \lim_{N\to\infty} \mathbb{E}\left[\sum_{k=m_\pi K}^{N-1} C(s_k,\pi(s_k))\right] \\ &\le \mathbb{E}\left[\sum_{k=0}^{m_\pi K-1} C(s_k,\pi(s_k))\right] + \frac{M\rho^k}{1-\rho}. && \text{(Inéquation 1.4)} \qquad (1.5) \end{aligned}$$

De plus, on peut affirmer que l'espérance de la fonction de valeur V_0 après mK transitions est :

$$\begin{aligned} |\mathbb{E}(V_0(s_{m_\pi K}))| &= \sum_{s\in S} P(s_{m_\pi K} = s|x_0,\pi)V_o(s) && \text{(déf. espérance mathématique)} \\ &\le \left(\sum_{s\in S} P(s_{m_\pi K} = s|x_0,\pi)\right) \max_{s\in S} V_0(s) \\ &\le \rho^k \max_{s\in S} V_0(s). && \text{(Inéquation 1.3)} \qquad (1.6) \end{aligned}$$

En combinant les inégalités 1.5 et 1.6, on obtient donc

$$V^{\pi}(s) - \tfrac{M\rho^k}{1-\rho} - \rho^K \max_{s\in S} V_0(s) \le \mathbb{E}\left[\sum_{k=0}^{m_\pi K-1} C(s_k,\pi(s_k)) + V_0(s_{m_\pi K})\right] \le V^{\pi}(s) + \tfrac{M\rho^k}{1-\rho} + \rho^K \max_{s\in S} V_0(s).$$

La double inégalité vient du fait que les inégalités 1.5 et 1.6 sont vraies en valeurs absolues. L'espérance au centre de cette inégalité correspond au coût espéré de la politique π, partant de l'état initial s_0, lorsque $m_\pi K$ actions sont exécutés (la somme représente l'espérance de coût pour se rendre à l'état $s_{m_\pi K}$, et $V_0(s_{m_\pi K})$ représente le coût prédit pour se rendre au but à partir de $s_{m_\pi K}$. En prenant le minimum de ce coût sur l'ensemble des politiques π, on obtient $V_{m_\pi K}(s_0)$, le coût donné par la fonction de valeur généré par $m_\pi K$ itérations de l'équation de Bellman. Ainsi, en prenant le minimum sur π, on obtient

$$V^{\star}(s) - \frac{M\rho^k}{1-\rho} - \rho^K \max_{s\in S} V_0(s) \le V_{m_\pi K}(s_0) \le V^{\star}(s) + \frac{M\rho^k}{1-\rho} + \rho^K \max_{s\in S} V_0(s).$$

En prenant la limite lorsque K tend vers l'infini, les termes contenant ρ^K disparaissent et on obtient donc

$$\lim_{K\to\infty} V_{m_\pi K}(s_0) = V^{\star}(s_0).$$

Ce résultat étant vrai pour n'importe quel état initial $s \in S$, on en conclut donc que $\lim_{i\to\infty} V_i = V^{\star}$. ■

Plusieurs preuves de la version de l'équation de Bellman s'appliquant aux PDM-HIE et aux PDM-HF existent dans la littérature scientifique et sur le web. Or, la version pour PDM-PCCS présentée ci-dessus ne semble être disponible que dans le livre de Bertsekas (1995), un ouvrage de plusieurs milliers de pages et assez technique où les résultats intermédiaires qui sont utilisés dans la preuve sont répartis à différents endroits. La preuve précédente constitue une réécriture synthétisée de celle proposée par Bertsekas, écrite de sorte qu'elle soit plus claire et détaillée. Cette réécriture constitue, à notre connaissance, la première présentation de cette preuve à la fois complète et autosuffisante.

Il est possible de montrer que le principe d'optimalité s'applique également sous des hypothèses moins contraignantes que dans la proposition 1.9. En l'occurrence, plutôt que de devoir admettre *uniquement* des politiques propres, il suffit qu'il existe *au moins une* politique propre et que chaque politique impropre mène à une espérance de coût infinie à partir d'*au moins un* état initial (Bertsekas, 1995). Or, la preuve de ce résultat plus fort est beaucoup plus complexe et dépasse l'objectif de ce document.

1.2 Algorithmes de base

Le principe d'optimalité nous indique qu'il existe toujours une politique optimale qui soit markovienne et stationnaire dans un PDM-PCCS (à condition qu'une politique propre existe, chose que nous supposerons toujours dans ce texte). On voudrait maintenant trouver la meilleure stratégie pour trouver cette politique.

1.2.1 *Value Iteration*

La proposition 1.9 indique qu'une méthode possible est d'initialiser une fonction de valeur V_0 arbitrairement (avec $V_0(s_g) = 0$), et de mettre à jour itérativement cette fonction de valeur en appliquant l'équation de Bellman à chaque itération. Cette stratégie est celle utilisée par l'algorithme ***Value Iteration*** (Bellman, 1957).

Puisqu'il est possible de ne pas converger numériquement vers un point fixe en un nombre fini d'itérations, l'algorithme prend en entrée un paramètre d'erreur, noté ϵ, qui contrôle la précision souhaitée. À chaque itération sur l'ensemble de l'espace d'états, chaque état est mis à jour en utilisant l'équation de Bellman avec les anciennes valeurs $V(s')$ des états voisins. La mise à jour d'un seul état est appelée une **mise à jour de Bellman** (*Bellman backup*, en anglais), tandis qu'une mise à jour de Bellman effectuée sur chaque état de l'espace d'états est appelée un **balayage** (*sweep*) de l'espace d'états. La variation (en valeur absolue) de la valeur $V(s)$ d'un état s suite à une mise à jour de Bellman est appelée le **résidu** de s, et noté $Res(s)$.

Lorsque la fonction de valeur a convergé à la précision souhaitée, la politique est simplement déterminée de manière gloutonne. Pour chaque état s, elle retourne l'action a pour laquelle le nombre $Q(s, a)$ est le moins grand, c'est-à-dire l'action qui minimise l'espérance de coût futur. L'ensemble des étapes de l'algorithme *Value Iteration* sont présentées à l'algorithme 1.1.

Algorithme 1.1 *Value Iteration* (synchrone).

1: Initialiser arbitrairement une fonction de valeur V_0 ▷ avec $V_0(s_g) = 0$
2: $n \leftarrow 0$
3: **faire**
4: $\quad n \leftarrow n + 1$
5: $\quad$ **pour tout** $s \in S$ **faire**
6: $\quad\quad V_n(s) \leftarrow \min_{a \in A(s)} \left[C(s, a) + \sum_{s' \in S} T(s, a, s')V_{n-1}(s')\right]$
7: $\quad\quad Res(s) \leftarrow |V_n(s) - V_{n-1}(s)|$
8: **tant que** $\max_{s \in S} Res(s) \geq \epsilon$
9: **retourne** π^{V_n} ▷ $\pi^{V_n}(s) \leftarrow \arg\min_{a \in A} Q_n(s, a)$

En pratique, dans l'implémentation de l'algorithme on ne garde pas en mémoire toutes les fonctions de valeurs V_n. Il suffit d'en garder deux, celle du dernier balayage et celle du balayage courant. Chaque mise à jour de Bellman (ligne 6 dans l'algorithme) prend un temps $\mathcal{O}(|S||A|)$. Étant donné que chaque balayage (chaque itération de la boucle *tant que*) effectue $|S|$ mises à jour de Bellman, alors chaque balayage de VI prend un temps $\mathcal{O}(|S|^2|A|)$. Dans les PDM-PCCS généraux, aucune borne sur le nombre de balayages nécessaires avant que le critère d'arrêt (convergence de la fonction de valeur à une précision ϵ) ne soit atteint n'est connue (Mausam et Kolobov, 2012).

1.2.2 *Value Iteration* Asynchrone

La version synchrone de l'algorithme *Value Iteration* (VI) a quelques désavantages. Premièrement, il faut garder à chaque instant en mémoire deux valeurs pour chaque état, $V_n(s)$ et $V_{n-1}(s)$, ce qui consomme plus de mémoire que nécessaire. Deuxièmement, lors d'une itération, les calculs se font toujours en fonction des valeurs de l'itération précédente. Or, la convergence serait plus rapide si l'on pouvait utiliser les valeurs des états déjà mis à jour durant le balayage courant.

Plusieurs algorithmes se basent sur l'observation suivante pour être plus efficace : *l'ordre de considération des états n'est pas important, et le nombre de fois que la valeur de chaque état est mise à jour n'a pas à être identique*. En effet, la seule condition à respecter pour s'assurer de converger vers la solution optimale est que le résidu $Res(s)$ de chaque état diminue jusqu'à être plus petit que ϵ (Bertsekas et Tsitsiklis, 2015).

L'algorithme ***Value Iteration* Asynchrone** est une modification de l'algorithme 1.1 qui met à jour la fonction de valeur en place directement. Autrement dit, une seule fonction de valeur est stockée en mémoire et les valeurs des états sont remplacées à mesure que les mises à jour de Bellman sont effectuées. Le terme « *Value Iteration* Asynchrone » englobe parfois n'importe quel algorithme qui met à jour les valeurs des états dans un ordre basé sur un critère prédéfini. Le pseudo-code général est présenté à l'algorithme 1.2.

Algorithme 1.2 *Value Iteration* Asynchrone.

1: Initialiser arbitrairement une fonction de valeur V
2: **faire**
3: Choisir un état s
4: $v_{ancien} \leftarrow V(s)$
5: $V(s) \leftarrow \min_{a \in A(s)} \left[C(s,a) + \sum_{s' \in S} T(s,a,s') V(s') \right]$
6: $Res(s) \leftarrow |v_{ancien} - V(s)|$
7: **tant que** $\max_{s \in S} Res(s) \geq \epsilon$
8: **retourne** π^V ▷ $\pi^V(s) \leftarrow \arg\min_{a \in A} Q(s,a)$

Value Iteration Asynchrone ne spécifie pas d'ordre ou de méthode spécifique pour choisir le prochain état considéré à chaque nouvelle itération. Une variante souvent utilisée est la variante *Gauss-Seidel*, qui consiste à choisir chaque état dans un ordre prédéterminé de façon cyclique (façon « *round-robin* »). Il est important de remarquer que cette variante n'est pas équivalente à VI synchrone car, par exemple, la valeur qui vient d'être mise à jour pour un état s_i peut être utilisée lorsque vient le temps de faire le calcul de la nouvelle valeur de l'état s_{i+1}. Dans le reste de ce document, lorsque nous ferons référence à « l'algorithme VI », cela désignera spécifiquement la variante Gauss-Seidel de *Value Iteration* Asynchrone.

1.2.3 *Policy Iteration*

Value Iteration consiste à se déplacer d'une fonction de valeur à l'autre dans l'espace des fonctions de valeur. La politique est trouvée gloutonnement à la toute fin, lorsque la fonction de valeur courante a atteint un point fixe. Plutôt que de se déplacer dans l'espace des fonctions de valeur, il est aussi possible de se déplacer d'une politique à une autre dans l'espace des politiques (propres). C'est la stratégie utilisée par l'autre algorithme classique de résolution des PDM, ***Policy Iteration*** (Howard, 1960). Il nécessite de connaître initialement une politique propre. Ensuite, l'algorithme alterne entre deux étapes : (1) l'évaluation de la politique, qui consiste à trouver la fonction de valeur correspondante à la politique actuelle, et (2) l'amélioration de la politique, qui consiste à trouver gloutonnement la politique qui correspond à la fonction de valeur actuelle. L'alternance entre ces deux étapes permet de trouver une suite de politiques et de fonctions de valeur qui s'améliorent de façon monotone : $\pi_0 \xrightarrow{E} V^{\pi_0} \xrightarrow{A} \pi_1 \xrightarrow{E} V^{\pi_1} \xrightarrow{A} \pi_2 \xrightarrow{E} \cdots \xrightarrow{A} \pi^\star \xrightarrow{E} V^\star$. où

« E » représente l'étape d'évaluation, et « A » représente l'étape d'amélioration de la politique. Les détails sont présentés à l'algorithme 1.3.

Algorithme 1.3 *Policy Iteration*.

1: Trouver une politique propre π quelconque
2: **répéter**
3: ▷ Évaluation de la politique
4: **faire**
5: **pour tout** $s \in S$ **faire**
6: $v_{ancien} \leftarrow V^{\pi}(s)$
7: $V^{\pi}(s) \leftarrow C(s,a) + \sum_{s' \in S} T(s,a,s')V^{\pi}(s')$
8: $Res(s) \leftarrow |v_{ancien} - V^{\pi}(s)|$
9: **tant que** $\max_{s \in S} Res(s) \geq \epsilon$
10:
11: ▷ Amélioration de la politique
12: estStable ← Vrai
13: **pour tout** $s \in S$ **faire**
14: $a_{ancien} \leftarrow \pi(s)$
15: $\pi(s) \leftarrow \arg\min_{a \in A} \left[C(s,a) + \sum_{s' \in S} T(s,a,s')V(s')\right]$
16: **si** $a_{ancien} \neq \pi(s)$ **alors**
17: estStable ← Faux
18:
19: **si** estStable **alors**
20: **retourne** π

Il est important de noter que lors de l'évaluation de la politique, la mise à jour de la valeur de chaque état (ligne 7) est différente de celle faite par *Value Iteration*. En effet, puisqu'on évalue une politique connue, le minimum sur chaque action qui était présent dans l'équation de Bellman disparaît, ce qui rend l'équation linéaire. Plutôt que de faire l'évaluation de la politique itérativement comme à l'algorithme 1.3, les implémentations de *Policy Iteration* utilisent parfois un algorithme pour résoudre le système d'équations linéaires équivalent (une équation par état).

Policy Iteration est parfois plus rapide que la version classique de *Value Iteration*. Il a aussi pour avantage de ne pas nécessiter de paramètre de précision ϵ. Cependant, il est de nos jours moins utilisé que VI pour deux raisons. Premièrement, contrairement à VI où l'on peut initialiser la fonction de valeur arbitrairement, *Policy Iteration* ne converge vers la politique optimale que si on lui donne en entrée une politique propre, ce qui peut être difficile à trouver dans le cas général. Deuxièmement, il est moins flexible et est donc moins facilement adaptable à diverses optimisations permettant d'accélérer le calcul, dont plusieurs seront présentées aux chapitres suivants.

CHAPITRE 2
APPROCHES EXISTANTES VISANT À AMÉLIORER LES ALGORITHMES CLASSIQUES

The performance of value and policy iteration can be dramatically improved by eliminating useless backups, and by backing up states in the right order.

— Wingate et Seppi (2005)

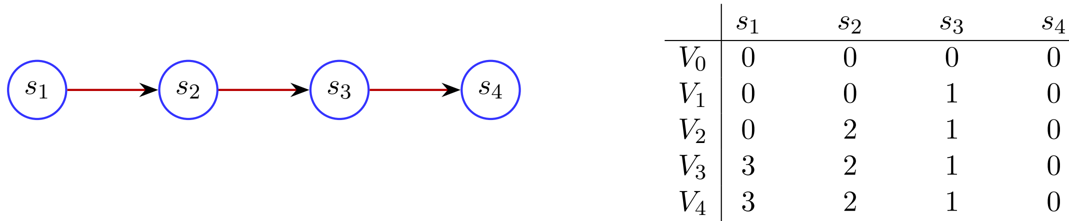
For problems with large state spaces, heuristic search has an advantage over dynamic programming because it can find an optimal solution for a start state without evaluating the entire state space.

— Hansen et Zilberstein (2001)

Depuis que les algorithmes *Value Iteration* et *Policy Iteration* ont été proposés, il y a de cela plus d'un demi-siècle, plusieurs méthodes permettant de calculer une politique optimale plus rapidement ont été proposées. Dans ce chapitre, nous détaillons les deux principales approches possibles : celles basées sur une priorisation des états (section 2.1), et celles basées sur l'utilisation de fonctions heuristiques (section 2.2).

2.1 Méthodes par priorisation

Comme nous l'avons vu au chapitre précédent, *Value Iteration* Asynchrone ne spécifie pas d'ordre en particulier pour faire les mises à jour de Bellman lors d'un balayage de l'espace d'états. Or, celui-ci peut avoir une influence significative sur le temps nécessaire pour que l'algorithme converge vers une fonction de valeur à la précision ϵ souhaitée. Par exemple, dans un PDM sans cycles, si les états sont choisis en ordre topologique inverse, alors une seule mise à jour de Bellman par état est nécessaire. Si l'ordre inverse avait été choisi, l'algorithme pourrait devoir faire $\mathcal{O}(|S|)$ mises à jour de Bellman par état (Bertsekas, 1995). L'exemple ci-dessous, où on a un PDM sans cycle qui contient une chaîne de quatre états liés deux à deux par des actions déterministes de coût unitaire, illustre l'inefficience de la propagation des valeurs dans le cas où les balayages sont faits dans l'ordre $\langle s_1, s_2, s_3, s_4 \rangle$. En effet, quatre balayages sont alors nécessaires, alors qu'un seul ne l'aurait été si les balayages étaient effectués dans l'autre sens.



	s_1	s_2	s_3	s_4
V_0	0	0	0	0
V_1	0	0	1	0
V_2	0	2	1	0
V_3	3	2	1	0
V_4	3	2	1	0

Figure 2.1 – Exemple d'un PDM de quatre états et trois actions où l'ordre des balayages est sous-optimal. Chaque rangée contient la valeur $V_i(s)$ de chacun des états après i balayages de l'espace d'états en ordre topologique. Quatre balayages sont nécessaires avant de pouvoir constater la convergence, alors qu'un seul ne l'aurait été si ceux-ci avaient plutôt lieu en ordre topologique inverse.

L'une des façons de contrôler l'ordre dans lequel les états sont choisis lors d'un balayage est de leur assigner une priorité. L'algorithme 2.1 présente ***Prioritized Value Iteration*** (PVI), une variante de *Value Iteration* Asynchrone qui utilise cette stratégie (Wingate et Seppi, 2005). Il assigne une priorité aux prédécesseurs de l'état en cours de traitement (c'est-à-dire les états ayant au moins une action pouvant mener directement à l'état en cours). En effet, les prédécesseurs sont les états pour lesquels la valeur $V(s)$ peut avoir changé suite à la mise à jour de Bellman de l'état en cours. Ceux-ci sont alors insérés dans une file prioritaire pour être considérés prochainement. Les états traités sont alors choisis en fonction de leur priorité dans la file.

Algorithme 2.1 *Prioritized Value Iteration*.

1: Initialiser arbitrairement une fonction de valeur V
2: $q \leftarrow$ file prioritaire vide
3: q.empiler$(s_g, 0)$
4: **faire**
5: $\quad s \leftarrow q$.défiler$()$
6: $\quad V(s) \leftarrow \min_{a \in A(s)} \left[C(s,a) + \sum_{s' \in S} T(s,a,s')V(s')\right]$
7: $\quad$ **pour tout** $s_p \in \{s_p \mid \exists a \in A, T(s_p, a, s) > 0\}$ **faire** $\qquad \triangleright$ pour tour prédécesseur de s
8: $\qquad q$.empiler$(s_p,$ priorité$(s_p))$
9: **tant que** $\max_{s \in S} Res(s) \geq \epsilon$
10: **retourne** π^V $\qquad \triangleright \pi^V(s) \leftarrow \arg\min_{a \in A} Q(s,a)$

Plusieurs méthodes de priorisation ont été proposées, tels que *Prioritized Sweeping* (PS) (Moore et Atkeson, 1993) et *Generalized PS* (Andre *et al.*, 1998). Certaines formules de priorités tentent d'éviter des cas tels que celui présenté à la figure 2.1 en donnant une plus grande priorité aux états les plus proches d'un but, ce qui favorise une meilleure rétropropagation des valeurs de ce dernier vers les autres états. Une autre stratégie est de prioriser les états s dont le résidu $Res(s)$ est le plus élevé, car leurs prochaines mises à jour de Bellman auront le plus d'impact localement. Certaines fonctions de priorisation qui combinent ces deux idées ont été proposées. Par exemple, McMahan et Gordon (2005) proposent d'utiliser une métrique de priorité appelée

Improved Prioritized Sweeping (IPS) :
$$IPS(s) = \frac{Res(s)}{V(s)}.$$
Le numérateur de cette métrique augmente la priorité des états les plus éloignés de la convergence, tandis que le dénominateur augmente la priorité des états dont le coût pour se rendre au but est le plus faible (ce qui correspond généralement aux états topologiquement les plus proches du but).

Malheureusement, bien qu'une priorisation des états puisse à priori sembler très bénéfique, quelques problèmes surviennent en pratique. En effet, bien que l'ordre de sélection des états lors d'un balayage soit amélioré, le gain de vitesse obtenu est tempéré par le coût de maintenance de la file prioritaire. En effet, celle-ci est habituellement implémentée à l'aide d'un monceau. Or, les monceaux binaires ont une complexité (insertion et retrait de l'élément prioritaire) de $\mathcal{O}(\log n)$ tandis que les monceaux asymptotiquement plus rapides, tel que le monceau de Fibonacci, sont généralement plus lent pour la quantité de données requises lors de la résolution d'exemples réels. Wingate et Seppi (2005) donnent l'exemple d'un domaine de planification où PVI atteint une vitesse 5 à 6 fois plus faible que celle de l'algorithme VI, bien qu'il nécessite 2 fois moins de mises à jour de Bellman pour converger.

Une façon de réduire le coût de maintenance de la file prioritaire est de partitionner l'espace d'états, et d'assigner des priorités aux partitions plutôt qu'aux états. L'avantage est double : (1) l'impact de la file prioritaire est moindre, puisqu'on n'a à gérer qu'une priorité par partition plutôt qu'une par état (on suppose qu'il y a beaucoup moins de partitions que d'états) et (2) on peut faire converger une partition entière avant de passer à la suivante. Ce dernier point est important car, par exemple, s'il n'y a pas de cycles entre les partitions, on n'a alors jamais à revenir sur une partition déjà traitée, ce qui peut accélérer substantiellement les calculs. ***Partitioned Value Iteration*** est un algorithme général illustrant l'idée de priorisation d'un partitionnement d'états. Celui-ci ne spécifie cependant pas la manière d'effectuer le partitionnement ni la métrique de priorité à utiliser. L'algorithme 2.2 présente les étapes communes à tout choix de partitionnement et de priorisation. De manière générale, la performance de *Partitioned Value Iteration* dépend grandement du choix du partitionnement et de la métrique de priorisation. Deux principaux problèmes surviennent avec cet algorithme : (1) un bon partitionnement doit en général être trouvé manuellement, par analyse du domaine de planification considéré, et (2) bien que le fait de prioriser les partitions plutôt que les états réduise le coût de gestion de la file prioritaire, celui-ci reste, en général, non négligeable.

L'algorithme ***Topological Value Iteration*** (TVI) répond à ces deux problématiques en proposant un partition-

Algorithme 2.2 *Partitioned Value Iteration*.

1: Initialiser arbitrairement une fonction de valeur V
2: $\mathcal{P} \leftarrow$ Partitionnement(S) ▷ méthode de partitionnement au choix
3: Initialiser les priorités pour chaque $p \in \mathcal{P}$ ▷ métrique de priorité au choix
4: **faire**
5: $p \leftarrow$ la partition de $\mathcal{P}$ la plus prioritaire
6: Faire un (ou plusieurs) balayage(s) sur les états de p
7: Mettre à jour la priorité des partitions de $\mathcal{P}$ ayant une transition vers la partition p
8: **tant que** $\max_{s \in S} Res(s) \geq \epsilon$
9: **retourne** π^V ▷ $\pi^V(s) \leftarrow \arg\min_{a \in A} Q(s, a)$

nement basé sur des caractéristiques topologiques du PDM (Dai *et al.*, 2011). Plus précisément, il calcule les composantes fortements connexes (CFC) [1] du graphe orienté obtenu en *déterminisant* le PDM, c'est-à-dire le graphe où chaque sommet représente un état du PDM, et où un arc $s \rightarrow s'$ est présent si et seulement s'il existe une action $a \in A$ telle que $T(s, a, s') > 0$. TVI considère les CFC en ordre topologique inverse et effectue des balayages sur chacune d'elles jusqu'à convergence avant de passer à la suivante. Puisque les CFC sont calculées automatiquement par un algorithme (en l'occurence, l'algorithme de Kosaraju ou l'algorithme de Tarjan [2] qui s'exécutent tous les deux en temps $\mathcal{O}(|S| + |A|)$) et que l'ordre de considération de celles-ci est calculé une seule fois avant le lancement des balayages, alors aucune file prioritaire n'est nécessaire pour prioriser les composantes, résolvant ainsi les deux désavantages de *Partitioned Value Iteration*. Le pseudo-code de TVI est présenté à l'algorithme 2.3. La figure 2.2 représente visuellement la décomposition d'un PDM en composantes fortement connexes.

Algorithme 2.3 *Topological Value Iteration*.

1: **procédure** TVI(M : PDM)
2: ▷ Les CFC sont retournées par Kosaraju ou Tarjan directement en ordre topologique inverse
3: $CFC \leftarrow$ Kosaraju(M) ▷ ou l'algorithme de Tarjan
4: **pour tout** $cfc \in CFC$ **faire**
5: VIPartiel(M, cfc) ▷ fait converger la CFC courante à une précision ϵ
6: **retourne** π^V ▷ $\pi^V(s) \leftarrow \arg\min_{a \in A} Q(s, a)$
7:
8: **procédure** VIPartiel(M : PDM, $\mathcal{S}$: un sous-ensemble d'états)
9: **faire**
10: **pour tout** $s \in \mathcal{S}$ **faire**
11: $V(s) \leftarrow \min_{a \in A(s)} \left[C(s, a) + \sum_{s' \in S} T(s, a, s') V(s') \right]$
12: **tant que** $\max_{s \in \mathcal{S}} Res(s) \geq \epsilon$

1. L'acronyme anglais SCC (pour *Strongly Connected Component*) est plus utilisé dans la littérature, mais nous utilisons volontairement l'acronyme français dans cette thèse.

2. La version proposée par les auteurs de TVI utilise l'algorithme de Kosaraju, mais la plupart des personnes qui l'implémente utilisent l'algorithme de Tarjan, qui est un peu plus rapide (Tarjan, 1972).

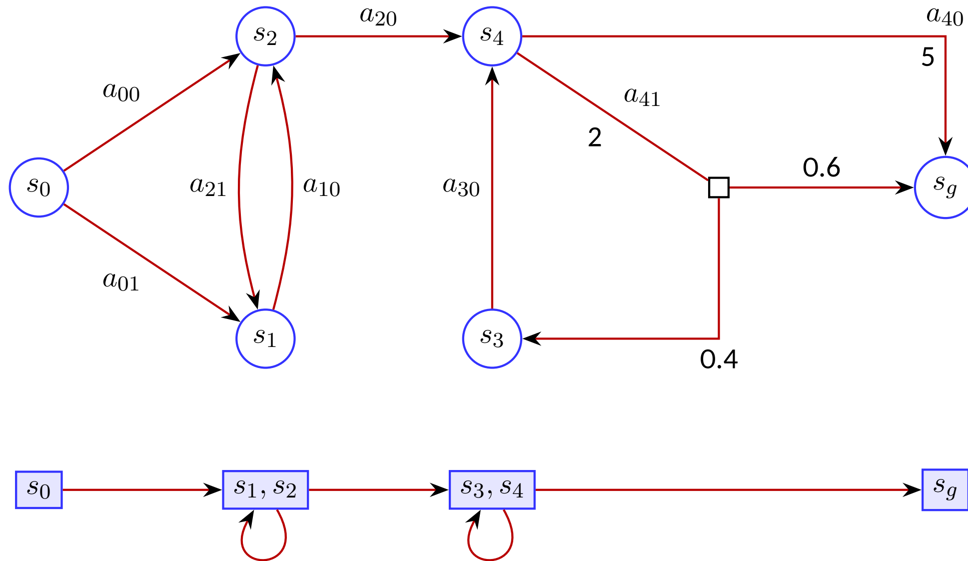


Figure 2.2 – Exemple de décomposition d'un PDM-PCCS en composantes fortement connexes (CFC). La condensation du PDM (le graphe de ses CFC) est illustrée sous le PDM. TVI fait converger les CFC une à une en ordre topologique inverse.

L'algorithme TVI est particulièrement efficace dans les PDM qui contiennent beaucoup de CFC de tailles semblables. Dans le meilleur cas, le PDM est acyclique, et alors une seule mise à jour de Bellman est nécessaire par état. Dans le pire des cas, le PDM ne contient qu'une seule CFC et l'algorithme devient alors quasi identique à *Value Iteration* Asynchrone [3].

2.2 Méthodes heuristiques

Les algorithmes mentionnés à la section précédente sont tous basés sur *Value Iteration* Asynchrone et améliorent la performance en jouant sur l'ordre de considération des états lors des balayages. Une autre manière de réduire le temps de calcul est d'élaguer certains éléments de l'ensemble des états considérés pendant celui-ci. En effet, s'il est possible de prouver que certains d'entre eux ne seront jamais atteints lors de l'exécution de la politique optimale, alors ils n'ont pas à être considéré lors des calculs. De plus, la politique n'a pas à être définie pour ces états. Ainsi, plutôt que de chercher une politique complète $\pi : S \rightarrow A$, comme précédemment, on cherche désormais une **politique partielle** $\pi : S' \rightarrow A$ où $S' \subseteq S$.

Il est clair que l'ensemble des états non atteignables par une politique dépend de l'état initial. Le calcul d'une politique partielle nécessite donc de connaître à priori un état initial du PDM. Nous noterons donc π_{s_0} une

3. À une différence importante près, que nous expliciterons au chapitre 6.

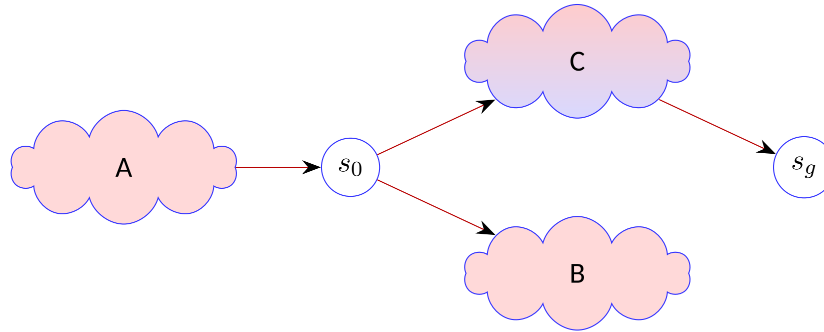


Figure 2.3 – Illustration de certaines situations rendant l'élagage d'états possible : (A) les états ne pouvant pas être atteints à partir de s_0 ; (B) les états ne pouvant pas atteindre s_g ; (C) certains états atteignables par s_0 et pouvant atteindre s_g, mais n'étant jamais visités en exécutant $\pi^\star$ en partant de s_0.

politique partielle ayant s_0 comme état de départ. Puisque la définition générale d'un PDM-PCCS ne contient pas d'état initial, il est d'usage de noter PDM-PCCS$_{s_0}$ la variante avec état initial s_0 du problème PDM-PCCS. Connaissant cet état, il devient possible d'élaguer de l'espace d'états tous les éléments non atteignables à partir de celui-ci. Par analogie avec le calcul de chemins dans un graphe, on peut voir les problèmes PDM-PCCS et PDM-PCCS$_{s_0}$ respectivement comme la recherche d'un chemin optimal entre toutes les paires de sommets, et celui de trouver un chemin optimal d'un sommet vers tous les autres.

Avec une information supplémentaire connue à priori — une **fonction heuristique** $h\colon S \to \mathbb{R}$ qui fournit une estimation du coût nécessaire pour atteindre le but à partir de chaque état — il devient possible d'élaguer davantage d'états. Les algorithmes combinant la connaissance d'un état initial s_0 et d'une fonction heuristique h sont appelés **algorithmes de recherche heuristique**. Lorsque h est **admissible**, c'est-à-dire que $\forall s \in S$, $h(s) \leq V^\star(s)$, ces algorithmes ont la garantie de ne pas élaguer d'états en trop, et donc de retourner une politique optimale. La figure 2.3 illustre certains cas où des états peuvent être élagués.

L'**algorithme LAO*** (*Loop-AO**) est un des algorithmes de recherche heuristique les plus populaires pour les PDM (Hansen et Zilberstein, 2001). Celui-ci est une extension gérant les cycles (*loops*) de l'algorithme AO* (*AND/OR**). Ce dernier s'applique aux graphes « ET/OU » sans cycles (Nilsson, 1982). Il est basé sur A*, l'algorithme de recherche heuristique permettant de trouver un plus court chemin dans un graphe (Hart *et al.*, 1968). Tout comme pour A*, l'idée est d'étendre la frontière des états à explorer autour de s_0 dans la direction du but s_g (cette direction est donnée implicitement par la fonction heuristique h). Dans la terminologie liée à LAO*, le graphe $\mathcal{G}$ contenant les états atteignables en partant de s_0 et en suivant les actions données par la politique obtenue gloutonnement par la fonction de valeur actuelle est appelé le *graphe solution*. Il

représente la connectivité des états explorés par rapport à la politique actuelle. Les états de celui-ci forment un sous-ensemble de l'*enveloppe*, qui est l'ensemble des états ayant à un moment ou à un autre fait partie du graphe solution. Tout comme pour A*, l'ensemble des états de l'enveloppe ayant au moins un successeur hors de l'enveloppe est appelée la *frontière*, tandis que l'ensemble des autres états de l'enveloppe est appelée l'*intérieur*. Le fonctionnement de LAO* est décrit à l'algorithme 2.4.

Algorithme 2.4 LAO*.

1: $V \leftarrow h$ ▷ la fonction de valeur est initialisée à l'aide de l'heuristique
2: $F \leftarrow \{s_0\}$ ▷ ensemble frontière ; correspond à la file prioritaire dans A*
3: $I \leftarrow \{\}$ ▷ ensemble intérieur
4: $\mathcal{G} \leftarrow \{$sommets : $\{s_0\},$ actions : $\{\}\}$ ▷ graphe solution
5: **tant que** $F \cap \mathcal{G}$ contient des états autre que s_g **faire**
6: $s \leftarrow$ un état de $F \cap \mathcal{G}$ autre que s_g
7: $F \leftarrow F \setminus \{s\}$
8: $F \leftarrow F \cup \{s' \notin I \mid \exists a \in A, T(s,a,s') > 0\}$ ▷ les voisins de s pas dans I se joignent à la frontière
9: $I \leftarrow I \cup \{s\}$
10: $\mathcal{G} \leftarrow \{$sommets : $I \cup F,$ actions : toutes les actions de tous les états de $I\}$
11: $Z \leftarrow \{s$ et tous les autres états de $\mathcal{G}$ pouvant être atteint par la politique gloutonne actuelle $\pi^V_{s_0}\}$
12: VIPartiel(M, Z) ▷ fonction définie à l'algorithme 2.3 ; cela met à jour V
13: Reconstruire $\mathcal{G}$ pour le rendre cohérent avec la nouvelle politique implicite $\pi^V_{s_0}$
14: **retourne** $\pi^V_{s_0}$ ▷ $\forall s \in I, \pi^V_{s_0}(s) \leftarrow \arg\min_{a \in A} Q(s,a)$

L'étape la plus difficile à implémenter est celle de la ligne 11. Elle oblige à stocker une structure de données contenant les prédécesseurs pouvant atteindre chaque état en suivant la politique gloutonne actuelle, et à la mettre à jour à chaque fois que la fonction de valeur (et donc, la politique gloutonne associée) change. De plus, le fait de devoir faire converger la fonction de valeur à l'aide de la fonction *VIPartiel* à chaque fois qu'un nouvel état est ajouté à l'ensemble intérieur ralentit considérablement chaque itération de LAO*.

L'**algorithme ILAO*** (*Improved LAO**) améliore LAO* de trois façons : (1) à chaque itération, il traite l'ensemble des éléments dans $F \cap \mathcal{G}$ plutôt que de n'en choisir qu'un seul, (2) il effectue un unique balayage de Z plutôt que d'en faire plusieurs jusqu'à convergence, et (3) les mises à jour de Bellman lors du balayage sont faites dans l'ordre donnée par un parcours en profondeur postordre inversé partant de s_0. Ainsi, les valeurs sont mieux propagés du but vers l'état initial. Il existe également d'autres variantes, comme RLAO* (*Reverse LAO**) et BLAO* (*Bidirectional LAO**) qui explorent respectivement les états en suivant les actions en sens inverse (de s_g vers s_0), ou simultanément dans les deux directions (Dai et Goldsmith, 2006).

L'autre algorithme de recherche heuristique très utilisé est l'**algorithme RTDP**, l'acronyme signifiant *Real-Time Dynamic Programming* (Barto et Bradtke, 1995). Celui-ci a été conçu dans le contexte de l'apprentissage

par renforcement, où l'on souhaite à la fois explorer l'environnement et apprendre une politique optimale. Il combine une approche basée sur des essais par échantillonage d'actions (*trial-based sampling algorithm*) et l'utilisation d'une fonction heuristique pour guider cet échantillonage. Le pseudo-code est présenté à l'algorithme 2.5. RTDP simule des *trajectoires* partant de s_0 et terminant à s_g en effectuant toujours l'action actuellement optimale d'après la fonction de valeur courante. Puisque les actions sont probabilistes, chaque trajectoire a le potentiel d'explorer des états encore jamais explorés. À chaque état traversé lors d'une trajectoire, la fonction de valeur est mise à jour — si s a déjà été visité précédemment — avec la valeur $Q(s, a)$, qui peut avoir changée depuis la dernière visite à s.

Algorithme 2.5 *Real-Time Dynamic Programming*.

```
1:  procédure RTDP(s_0)
2:      tant que temps restant > 0 faire
3:          TrajectoireRTDP(s_0)
4:      retourne π^V_{s_0}          ▷ pour chaque état s visité, π^V_{s_0}(s) ← arg min_{a∈A} Q(s, a)
5:
6:  procédure TrajectoireRTDP(s)
7:      tant que s ≠ s_g faire
8:          a ← arg min_{a∈A(s)} Q(s, a)
9:          V(s) ← Q(s, a)
10:         s ← état atteint après l'exécution de l'action probabiliste a
```

Puisqu'en apprentissage par renforcement, l'environnement est exploré à mesure que la planification est effectuée (les algorithmes de planification sont utilisés en temps réel, d'où le nom de RTDP), le critère d'arrêt utilisé est généralement une limite de temps. En revanche, lorsqu'on cherche une politique dans le contexte de la planification automatique, une garantie d'optimalité (ou une borne limitant la sous-optimalité) est généralement recherchée. Une limite de temps ne constitue donc pas un critère d'arrêt satisfaisant. Un autre désavantage de RTDP est qu'il visitera souvent les mêmes états, même ceux pour lesquels $V(s)$ a déjà convergé, puisqu'il n'a aucun moyen de quantifier le niveau de convergence à chaque état.

L'**algorithme LRTDP** (*Labeled RTDP*) offre une solution à ces deux désavantages (Bonet et Geffner, 2003). Cet algorithme modifie RTDP en ajoutant une étiquette à chaque état (calculée à partir des résidus de Bellman, comme pour VI), indiquant si celui-ci a convergé à une précision ϵ ou non. Cet ajout permet de savoir lorsque la politique (partielle) a globalement convergé, et permet donc d'utiliser le même critère d'arrêt que celui des algorithmes basés sur VI : $\max_{s \in S} Res(s) \leq \epsilon$. Cela permet également de concentrer les trajectoires sur les états qui n'ont pas encore convergé, augmentant ainsi la vitesse de convergence globale.

Plusieurs variantes de LRTDP existent. Si, en plus de connaitre une fonction heuristique h qui sous-estime les coûts réels, on connait une autre fonction heuristique h_u qui surestime les coûts réels, alors il est possible de guider l'exploration lors des trajectoires encore plus précisément qu'avec LRTDP. Par exemple, BRTDP (*Bounded RTDP*) (McMahan *et al.*, 2005) et FRTDP (*Focused RTDP*) (Smith et Simmons, 2006) sont des algorithmes qui supposent une connaissance de h_u. Lors des calculs, ils maintiennent tous les deux des fonctions de valeurs V_l et V_u, qui représentent respectivement une borne inférieure et supérieure de $V^\star$. BRTDP concentre ses trajectoires sur les états où l'écart $V_u(s) - V_l(s)$ est le plus grand. Quant à lui, FRTDP utilise l'écart entre les bornes tout comme BRTDP, mais considère également une mesure de l'espérance de la fraction de temps que la politique gloutonne actuelle s'attend à passer à l'état s choisi.

Certains algorithmes proposés dans la littérature combinent les deux approches présentées ci-dessus, c'est-à-dire les méthodes heuristiques et les méthodes par priorisation. Par exemple, l'**algorithme FTVI** (*Focused TVI*), qui nécessite deux heuristiques, h_l et h_u, consiste en deux phases : (1) une recherche heuristique, similaire à LAO*, permettant d'éliminer des actions sous-optimales du PDM, et (2) une décomposition du PDM résultant en CFC et une résolution de celles-ci une par une en ordre topologique inverse, de manière analogue à TVI (Dai *et al.*, 2011). L'élimination d'actions effectuée lors de la phase 1 de FTVI est possible grâce au théorème 2.1, appelé théorème de l'élimination d'actions (Bertsekas, 1995).

Théorème 2.1. *Soit deux fonctions de valeurs V_l et V_u telles que pour tout $s \in S$, $V_l(s) \leq V^\star(s) \leq V_u(s)$. De plus, soit un état $s \in S$ et deux actions $a \in A, a' \in A$. Si $Q^{V_u}(s,a) < Q^{V_l}(s,a')$, alors $\pi^\star(s) \neq a'$ et l'action peut donc être éliminée du PDM.*

Le chapitre précédent a introduit Value Iteration (VI) et Policy Iteration (PI), deux méthodes classiques pour résoudre les PDM-PCCS. Le chapitre actuel a quant-à-lui présenté les deux principales approches proposées dans la littérature pour améliorer les algorithmes classiques : les méthodes par priorisation et les méthodes heuristiques. Le prochain chapitre fait un survol de l'architecture moderne des ordinateurs, ce qui viendra compléter l'ensemble des notions prealables nécessaires à la compréhension des contributions proposées dans cette thèse.

CHAPITRE 3
ARCHITECTURE MODERNE DES ORDINATEURS

We believe that more research on cache efficiency of MDP algorithms is desirable and could lead to substantial payoffs — similar to the literature in the algorithms community, one may gain huge speedups due to better cache performance of the algorithms.

— Mausam et Kolobov (2012)

Les diverses approches présentées au chapitre 2 ont permis d'obtenir des gains significatifs de performance (plusieurs ordres de grandeur) lors du calcul de politiques optimales pour des PDM-PCCS. Cependant, comme le sous-entend la citation en épigraphe de ce chapitre, une autre approche — encore très peu considérée en planification automatique — est possible : *la considération de la mémoire cache lors de la conception et de l'implémentation des algorithmes*. Ce chapitre vise à introduire différents concepts d'architecture moderne des ordinateurs, tels que la hiérarchie de mémoire à la section 3.1, et les différents niveaux de parallélisme existant à la section 3.2, sur lesquels reposent les nouvelles contributions scientifiques présentées à partir du chapitre 4. Nous analysons également les gains maximaux théoriques pouvant être obtenus par la considération de tels éléments.

3.1 Hiérarchie de mémoire

Les premiers processeurs (*Central Processing Unit*, CPU) transféraient directement les données requises de la mémoire vive aux registres pour réaliser des calculs. Toutefois, alors que la vitesse de calcul des processeurs augmentait, la rapidité d'accès à la mémoire n'évoluait pas au même rythme. Face à ce décalage, les fabricants de processeurs ont choisi d'intégrer des niveaux de mémoires intermédiaires, appelés **mémoires cache**, dont la vitesse et la taille se situent entre celles des registres et celles de la mémoire vive. Les processeurs modernes contiennent généralement trois de ces mémoires intermédiaires, qui sont nommées respectivement « cache L1, L2 et L3 ». La figure 3.1 présente la hiérarchie de mémoire ayant cours dans la plupart des ordinateurs modernes. Les mémoires les plus près d'un cœur d'un processeur sont plus rapides, mais ont aussi moins d'espace de stockage, en raison de leur coût plus élevé et de l'espace physique limitée sur le CPU. Généralement, les CPU modernes sont constitués de plusieurs *cœurs*, où chacun possède sa

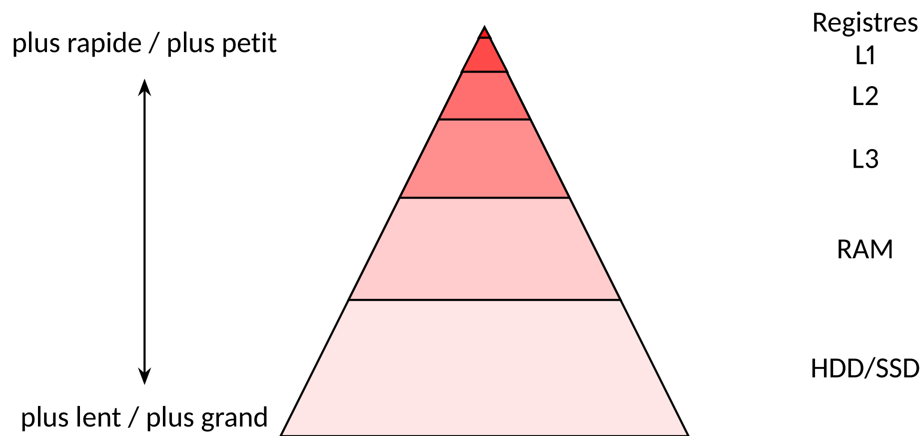


Figure 3.1 – Hiérarchie de mémoire des ordinateurs modernes.

propre mémoire cache L1 et L2, tandis que la mémoire cache L3 est partagée entre tous les cœurs. Le tableau 3.1 indique la taille, la vitesse en lecture, en écriture, et la latence type des différentes mémoires dans la hiérarchie [1] [2] [3]. Les registres d'un processeur ont généralement une vitesse de lecture et d'écriture environ trois fois supérieure à celle de la mémoire cache L1, et une latence environ trois fois moindre [4]. Comme l'indique la table, le chargement d'une donnée présente en mémoire cache L1 est près de deux ordres de grandeur plus rapide qu'un chargement à partir de la mémoire vive. De plus, le coût d'un **défaut de cache** L3 — tentative d'accès à une adresse absente du cache L3, nécessitant donc une lecture en RAM — est de deux à trois ordres de grandeur plus grand qu'une opération arithmétique. Par exemple, sur les processeurs Intel Core Skylake (2015), un défaut de cache mène à une pénalité allant de 50 à 70 cycles [5].

On appelle **principe de localité** l'observation selon laquelle les programmes accèdent généralement à des données stockées près de celles accédées dernièrement (*localité spatiale*) et les données réutilisées le sont généralement fréquemment lors d'un même intervalle de temps (*localité temporelle*). Pour éviter d'être ralenti par la latence et la vitesse beaucoup moins élevées de la mémoire vive par rapport à celles des mémoires cache, un programme se doit de maximiser le plus possible la localité temporelle et spatiale de ses accès mémoires. Ainsi, les données seront chargées en mémoire cache uniquement lors de leur première utilisation, et y resteront au moins aussi longtemps que la localité d'accès mémoire sera maintenue.

1. https://www.overclockers.com/amd-ryzen-9-7900x-and-7700x-review/
2. https://www.storagereview.com/review/samsung-990-pro-ssd-review-2tb
3. https://www.seagate.com/ca/en/products/nas-drives/ironwolf-pro-hard-drive/
4. https://arstechnica.com/gadgets/2002/07/caching/
5. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

Type	Modèle	Taille	Lecture Go/s	Écriture Go/s	Latence ns
Cache L1	AMD Ryzen 9 7900X	$12 \cdot 64$ ko	4000	2042	0.7
Cache L2		$12 \cdot 1$ Mo	2000	1955	2.5
Cache L3		64 Mo	1480	1381	9.5
Mémoire vive DDR5	Trident Z5 neo 6000Mhz	8-128 Go	75.7	76.9	62.7
Stockage de masse SSD	Samsung Evo 990 Pro	0.25-2 To	5.6	5.9	356 400
Stockage de masse HDD	Seagate IronWolf Pro	2-24 To	0.285	0.285	4 160 000

Table 3.1 – Vitesse, latence et taille des mémoires cache sur un CPU AMD Ryzen 9 7900X et comparaison avec des vitesses et des tailles typiques pour la mémoire vive (DDR5) et la mémoire secondaire (SSD/HDD).

À titre d'exemple, les algorithmes de calcul du produit matriciel implémentés dans les bibliothèques d'algèbre linéaire — telles que `Eigen` [6] ou `Armadillo` [7] — exploitent la hiérarchie de mémoire, leur permettant d'obtenir des gains substantiels de performance, allant jusqu'à trois ordres de grandeur par rapport à une implémentation naïve. Par exemple, pour calculer le produit de deux grandes matrices, ils décomposent généralement celles-ci en blocs de tailles adaptées à la mémoire cache, puis ils les multiplient ensuite bloc par bloc. Puisqu'un bloc peut être multiplié par plusieurs autres, il peut rester en mémoire cache relativement longtemps ce qui augmente la localité temporelle (Goto et Van De Geijn, 2008). De plus, le simple fait de réordonner les traditionnelles boucles $i \to j \to k$ d'un produit matriciel — par exemple, en utilisant l'ordre $i \to k \to j$ dans le cas de matrices stockées de façon contiguë ligne par ligne (*row-major order*) — augmente la localité spatiale des accès mémoires, et ainsi, la performance.

Au-delà de la localité, un autre aspect important est le concept de **ligne de cache**, une segmentation en sous-blocs des mémoires cache. Les transferts de données entre différents niveaux de mémoire cache doivent s'effectuer une ligne de cache à la fois. Sur les processeurs modernes, une ligne a généralement une taille de 64 octets. Supposons que, dans un programme, on ait le tableau de 1 GiB suivant :

```
int t[262144] // int: 4 octets; total: 262144*4 = 1 GiB.
```

Dans ce cas, la boucle

```
for(int i = 0; i < 262144; i += 16)
  s += t[i]; // 4 octets utiles par ligne de 64 octets.
```

nécessite le transfert de $\frac{262\,144 \cdot 4}{64} \frac{\text{octets}}{\text{octets par ligne}} = 16\,384$ lignes de cache, tandis que la boucle

6. https://eigen.tuxfamily.org/index.php?title=Main_Page

7. https://arma.sourceforge.net/

```
for(int i = 0; i < 16384; ++i)
  s += t[i]; // les 64 octets de chaque ligne sont utiles.
```

n'en nécessite que $\frac{16\,384 \cdot 4}{64} \frac{\text{octets}}{\text{octets par ligne}} = 1024$, soit 16 fois moins, bien que les deux boucles font la somme de la même quantité de nombres : $16\,384$. Cet exemple illustre qu'en accédant aux données de manière contiguë, on réduit le nombre de lignes à charger au travers des trois niveaux de mémoire cache — car chacune d'elle contient plus de données « utiles » —, ce qui libère de l'espace en cache et réduit le nombre de lectures à partir de la mémoire vive, économisant du temps de calcul car ces dernières sont lentes.

3.2 Différents niveaux de parallélisme

Plusieurs niveaux de parallélisme existent sur les processeurs modernes, tels que le **parallélisme d'instructions** (*Instruction Level Parallelism*, ILP), le **parallélisme de données** (*Data Level Parallelism*, DLP) et le **parallélisme de fils d'exécution** (*Thread Level Parallelism*, TLP).

3.2.1 Parallélisme d'instructions

Les processeurs modernes, dits *superscalaires*, contiennent plusieurs **unités de calcul** (Abel et Reineke, 2019). Chacune d'entre elles peut exécuter un sous-ensemble d'instructions-machine. Par exemple, certaines unités de calcul ne peuvent exécuter que des opérations arithmétiques, tandis que d'autres ne peuvent exécuter que des opérations logiques. En général, il existe plusieurs unités de calcul pouvant exécuter un même type d'opérations. Le parallélisme d'instruction fait référence au fait que les instructions-machine peuvent être exécutées en parallèle par différentes unités de calcul.

La figure 3.2 présente un schéma [8] de l'architecture Skylake d'Intel, dans lequel on peut voir les différentes unités de calcul dans le *back-end* (appelé *execution engine* dans le schéma) du CPU. Le *front-end* est quant à lui la partie dédiée au décodage des instructions. L'architecture Skylake est très proche de *Kaby Lake*, l'architecture du processeur principal utilisé dans les évaluations empiriques présentées dans cette thèse.

Lors de l'exécution d'un programme, le processeur fait de l'**exécution dans le désordre** (*out-of-order execution*) permettant de réordonner les opérations de sorte à en maximiser le nombre qui peuvent être exécutées simultanément sans altérer le résultat obtenu. En revanche, certaines instructions étant interdépen-

8. https://chipsandcheese.com/p/skylake-intels-longest-serving-architecture

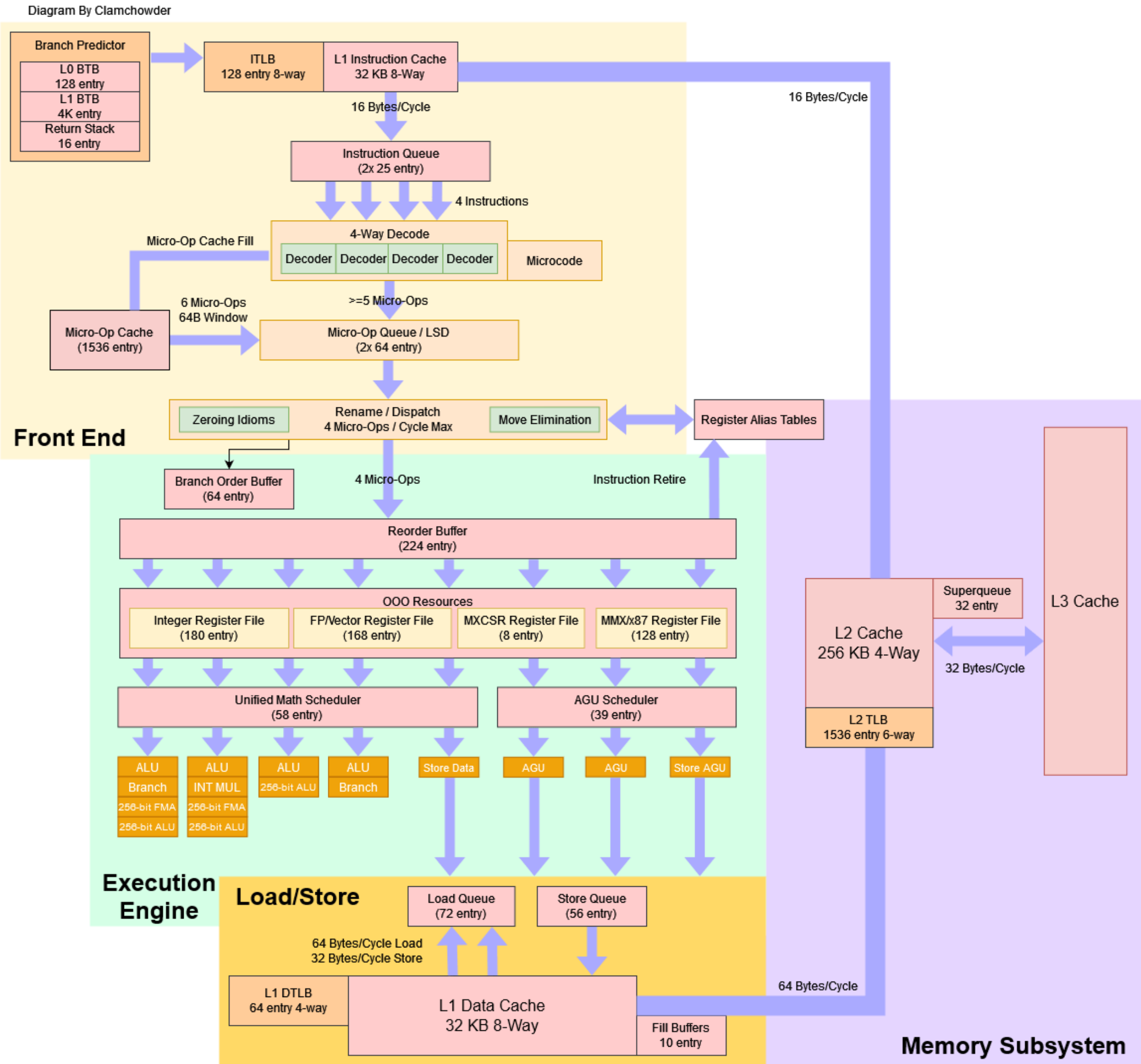


Figure 3.2 – Représentation schématique de l’architecture Skylake d’Intel.

dantes, toutes ne peuvent pas être exécutées en parallèle. Par exemple, considérons le code ci-dessous :

```
x = t[++i]; // (1) moins rapide
x = t[i++]; // (2) plus rapide
```

Dans la première version, on doit attendre que l'incrément soit fait avant de pouvoir commencer l'accès au tableau, tandis que dans la seconde, le CPU peut faire les deux opérations en parallèle dans des unités de calcul distinctes. De même, dans l'exemple :

```
x = a + b + c + d;      // (1) moins rapide; ((a + b) + c) + d
x = (a + b) + (c + d); // (2) plus rapide
```

la seconde version permet à l'ordonnanceur d'instructions dans le CPU de lancer le calcul des deux parenthèses simultanément, tandis que la première version ne le permet pas. Finalement, dans l'exemple :

```
for(int i = 0; i < 100; ++i) // (1) moins rapide
  s += t[i];
for(int i = 0; i < 100; i += 2) { // (2) plus rapide
  s1 += t[i];
  s2 += t[i + 1];
}
s = s1 + s2;
```

la première boucle contient une chaîne de dépendances, c'est-à-dire que le résultat de chaque itération dépend de celui de l'itération précédente. Le CPU ne pourra donc pas maximiser l'utilisation des unités de calcul. Dans la seconde boucle, deux unités arithmétiques peuvent être utilisées simultanément. En supposant qu'une addition prend 5 cycles, de T à T+5, la seconde instruction peut s'exécuter en parallèle après avoir été décodée, de T+1 à T+6, et la boucle prend donc $(\frac{100 \cdot 5}{2} + 1) + 5 = 256$ cycles au lieu de 500.

Les architectures modernes ont souvent beaucoup plus que deux unités de calcul. Par exemple, les processeurs utilisant l'architecture *Haswell* d'Intel, sortie en 2013, peut décoder et répartir jusqu'à 8 opérations par cycle sur les unités adaptées. En général, il y a beaucoup plus de 8 unités au total, de sorte que, même si certaines opérations ont un temps d'exécution de plus d'un cycle, de nouvelles opérations peuvent continuer à être réparties sur les unités libres pendant que les opérations longues sont toujours en cours [9]. En minimisant les chaînes de dépendances, et donc, en maximisant l'utilisation des unités de calcul, les programmes peuvent s'exécuter au moins un ordre de grandeur plus rapidement (Fog, 2023).

9. https://www.realworldtech.com/haswell-cpu/4/

3.2.2 Parallélisme de données

Comme mentionné à la section précédente, les processeurs modernes possèdent plusieurs unités de calcul pouvant effectuer les mêmes opérations, telle que des additions. Or, si on souhaite additionner 8 nombres contigus en mémoire avec 8 autres nombres contigus, le CPU devra décoder les 8 opérations l'une après l'autre. Ce décodage prend du temps et diminue la vitesse d'exécution maximale théorique. Pour pallier ce désavantage, les fabricants de processeurs, tels qu'Intel et AMD, ont donc introduit des instructions, appelées *instructions vectorielles*, permettant de faire des opérations sur plusieurs données contiguës simultanément. Le parallélisme obtenu en utilisant ces instructions est appelé parallélisme de données. On dit que ces instructions sont des instructions *SIMD*, pour *Single Instruction*, *Multiple Data*.

Sur l'architecture x86, les principaux jeux d'instructions vectorielles sont, en ordre historique, MMX (64 bits), SSE (128 bits), AVX (256 bits) et AVX-512 (512 bits). Ces instructions utilisent des registres spéciaux, de plus grandes tailles que les registres utilisés par les instructions scalaires. On peut générer ces instructions à l'aide d'appel aux fonctions intrinsèques dans des langages tels que C ou C++ [10]. Par exemple, si on souhaite additionner 16 nombres flottants de 4 octets d'un tableau vers un autre, la boucle :

```
for(int i = 0; i < 16; ++i)
  v1[i] += v2[i];
```

nécessite l'exécution de plusieurs dizaines d'instructions, et est équivalente à l'exécution des quatre instructions intrinsèques (correspondant chacune à une instruction SIMD) suivantes :

```
__m512 v1_values = _mm512_load_ps(v1);
__m512 v2_values = _mm512_load_ps(v2);
v1_values = _mm512_add_ps(v1_values, v2_values);
_mm512_store_ps(v1, v1_values);
```

Les compilateurs modernes peuvent vectoriser automatiquement certaines boucles. Par exemple, ils peuvent en général générer l'équivalent assembleur des instructions ci-dessus pour vectoriser la boucle de l'exemple précédent. En utilisant des entiers et des nombres flottants de 32 bits, on peut obtenir un exécutable nécessitant jusqu'à 16 fois moins d'instructions en vectorisant un code à l'aide d'un jeu d'instructions tel qu'AVX-512.

10. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

3.2.3 Parallélisme de fils d'exécution

La loi de Moore (Moore, 1965) prédisait que le nombre de transistors dans les CPU doublerait à chaque 2 ans. Historiquement, la fréquence des processeurs a également augmenté très rapidement d'année en année. Par exemple, en 1973, la fréquence moyenne était de 1 MHz, tandis qu'elle était de 1 GHz en 2003. Or, peu après 2005, la fréquence moyenne des processeurs a presque cessé de croître et se maintient autour de 3 à 4 Ghz. L'un des facteurs prédominants expliquant ce plafonnement est le fait que la consommation énergétique (et donc, la génération de chaleur) augmente à mesure que la fréquence augmente (Bentley, 2018). Une fréquence plus élevée nécessiterait donc des systèmes de refroidissement trop gros, bruyants ou complexes, un phénomène appelé en anglais le *power wall* (González, 2011).

Pour contourner ce problème, les fabricants de processeurs ont décidé d'ajouter plusieurs cœurs d'exécution à l'intérieur d'un même CPU. En effet, deux CPU de 2 GHz consomment moins d'énergie qu'un CPU de 4 GHz et génèrent ainsi moins de chaleur. Chacun des cœurs contient sa propre mémoire cache (généralement L1 et L2), ses propres unités de calcul, sa propre unité de contrôle, etc. En général, seule la mémoire cache L3 est partagée entre les différents cœurs. Bien que les fréquences n'aient pas beaucoup augmenté depuis 2005, le nombre de cœurs par processeur ne cesse, lui, d'augmenter. Se limitant en moyenne à deux cœurs par processeur en 2005, on peut de nos jours trouver des CPU, tel que le *AMD Threadripper Pro 7995WX*, pouvant avoir 96 cœurs [11]. La tendance de l'augmentation du nombre de cœurs semble donc se poursuivre.

Pour tirer profit de ce nombre de cœurs grandissant, les algorithmes doivent être repensés pour éviter qu'un trop grand nombre d'opérations doivent être exécutées de façon sérielle. En effet, la **loi de Amdahl** (Amdahl, 2013) indique que le facteur d'accélération maximal S pouvant être obtenu en passant d'un processeur monocœur vers un processeur à N cœurs, en supposant que s est la proportion du temps total d'exécution qui ne peut pas être parallélisé, est :

$$S = \frac{1}{s + \frac{1-s}{N}}. \tag{3.1}$$

Par exemple, sur un processeur de 96 cœurs, si $1\,\%$ d'un algorithme n'est pas parallélisable, alors le facteur d'accélération théorique maximal sera d'environ 49, tandis que si $10\,\%$ ne l'est pas, alors le facteur ne sera plus que de 9 environ.

11. https://www.amd.com/en/newsroom/press-releases/2023-10-19-amd-introduces-new-amd-ryzen-threadripper-7000-ser.html

Un autre élément à prendre en compte est le partage des données entre les différents cœurs. En effet, si les fils d'exécution doivent s'échanger beaucoup d'information, le temps de transfert de ces données d'un cœur à l'autre peut vite dépasser le potentiel gain de temps atteignable par l'exécution en parallèle. De plus, même lorsque les différents fils d'exécution parallèle n'ont pas d'information à s'échanger, il faut faire attention au problème de *faux partage*. Celui-ci se produit lorsque deux fils d'exécution tentent d'accéder à des éléments d'une même ligne de cache, avec au moins l'un des deux qui effectue une écriture. Lorsque cela survient, les deux fils n'ont plus la même version de la ligne de cache. Le prochain fil souhaitant lire une donnée dans cette ligne doit donc la resynchroniser à partir du cache L3 (qui est commun entre les fils) ou de la mémoire vive, nécessitant de la refaire transiter par le cache L2 puis L1.

À titre d'exemple, considérons le cas d'un programme où les éléments pairs d'un tableau d'entiers sont modifiés par un fil d'exécution, et les éléments impairs le sont par un autre. Une invalidation de cache pourrait alors survenir à presque chaque itération de chacun des deux fils, ce qui ralentira le programme bien plus que l'accélération permise par la parallélisation.

Les chapitres suivants montrent comment les éléments mentionnés dans ce chapitre peuvent être exploités lors du design et de l'implémentation d'algorithmes de résolution de processus décisionnels de Markov.

CHAPITRE 4

REPRÉSENTATION EFFICACE DES PROCESSUS DÉCISIONNELS DE MARKOV

> *When the speed of memory accesses rivalled that of the CPU, the perception that memory accesses are "for free" was a valid one. However, today, CPU speeds greatly exceed memory bandwidths on most platforms, to the point where computation is almost for free, and the real cost of execution, both in terms of speed and power consumption, is in accessing memory.*
>
> — Franco *et al.* (2017)

La recherche dans le domaine des PDM visant à accélérer les calculs d'une politique optimale se concentre généralement sur les avancées algorithmiques tel que la recherche heuristique (LRTDP, LAO*, etc.). En revanche, les détails d'implémentation reçoivent beaucoup moins d'attention. Nous soutenons que le choix de la représentation utilisée pour stocker un PDM en mémoire peut avoir un impact significatif sur les performances d'un solveur de PDM. Celui-ci peut parfois même être plus important que celui découlant du choix du solveur en soi. Cette différence de performance est principalement due à la quantité et à la localité des accès mémoires qui varient grandement selon les structures de données utilisées.

Dans ce chapitre, nous démontrons que l'exploitation de la hiérarchie de mémoire des ordinateurs modernes (décrite au chapitre 3) lors du chargement d'une instance de PDM peut permettre aux solveurs de fonctionner un ordre de grandeur plus rapidement. Ce chapitre présente des contributions de notre article présenté à la conférence *Canadian AI 2022* (Champagne Gareau *et al.*, 2022) et ayant été nommé parmi les *best papers* de la conférence [1]. La section 4.1 fait un survol d'implémentations de PDM existantes et analyse leur représentation en mémoire, la section 4.2 présente la nouvelle représentation en mémoire proposée, appelée CSR-MDP. Enfin, la section 4.3 présente les résultats empiriques obtenus par cette structure en les comparant à ceux obtenus par une implémentation de référence sur plusieurs instances de trois domaines de planification.

1. https://www.ai-crv.org/2022/award-winners

4.1 Analyse des implémentations existantes

La recherche en planification s'étant largement concentré sur les avancés des algorithmes et des heuristiques, la littérature présente rarement les détails d'implémentation, comme les structures de données utilisées pour stocker les PDM. De plus, le code source est lui aussi rarement disponible. Par conséquent, nous avons analysé le code source de différentes implémentations d'algorithmes de PDM disponibles publiquement (ou obtenu suite à une demande aux auteurs), pour avoir une meilleure idée des structures les plus couramment utilisées.

AI Toolbox est une bibliothèque C++ regroupant plusieurs algorithmes de PDM et de PDMOP [2] (Bargiacchi *et al.*, 2020). Elle permet à l'utilisateur de choisir entre des matrices denses ou creuses pour stocker les PDM. Chaque PDM est stocké à l'aide de deux matrices : une matrice 3D de transitions et une matrice 2D pour les coûts (ou récompenses). Les matrices denses ont comme désavantage de prendre une quantité de mémoire déraisonnable, même sur de petites instances, lorsque les PDM ne sont pas eux-mêmes denses [3]. D'autre part, les matrices creuses sont implémentées de telle manière qu'une moins grande quantité de mémoire est gaspillée, mais au prix d'une certaine diminution de la vitesse de calcul.

L'implémentation des auteurs de **TVI/FTVI** stocke les états des PDM à l'aide d'une liste chaînée. Chacune des entrées de cette liste d'états contient à son tour une liste chaînée d'actions applicables, elles-même composées de listes chaînées de transitions possibles (chaque transition étant une paire contenant un pointeur vers l'état voisin et la probabilité que cette transition survienne). Les éléments dans ces listes étant liés par des pointeurs, il y a donc un grand nombre d'indirections.

D'autres implémentations, comme la bibliothèque **MDP-Engine** de Blai Bonet [4] ou l'implémentation **G-Pack** [5] de l'algorithme Gourmand (Kolobov *et al.*, 2012), utilisent des tables de hachage stockant les états d'un PDM sous forme de structures. Ces implémentations, bien que nécessitant moins d'indirections que celles utilisant des listes chaînées, souffrent d'une pénalité calculatoire découlant de l'évaluation de la fonction de hachage qui doit être faite pour chaque état auquel on souhaite accéder.

2. Les processus décisionnels de Markov à observabilité partielle (*Partially Observable MDP*, POMDP) sont les PDM où on ne connait pas exactement l'état dans lequel l'agent se trouve. On ne les considère pas dans cette thèse.

3. Similaire à la notion de graphe dense, un PDM dense est un PDM où le nombre de transitions à chaque état est dans le même ordre de grandeur que le nombre d'états, ce qui est relativement rare dans les domaines de planification considérés en pratique.

4. `https://github.com/bonetblai/mdp-engine`

5. `https://aiweb.cs.washington.edu/ai/planning/gourmand.html`

Aucune d'entres elles n'exploite la hiérarchie de mémoire lors du stockage d'un PDM. De plus, elles se focalisent toutes sur une représentation en mémoire dite « *tableau de structures* » (*Array of Structures*, **AoS**), plutôt que « *structure de tableaux* » (*Structure of Arrays*, **SoA**), bien que cette représentation pourrait présenter certaines propriétés avantageuses [6]. Le code C++ suivant montre un exemple simple de représentations AoS et SoA, et d'accès à un attribut d'un élément dans ces structures :

```
// AoS
struct Point3D {
  float x, y, z;
};
Point3D points[N];
points[i].y = 3.0f;
```

```
// SoA
struct TableauPoints3D {
  float x[N], y[N], z[N];
};
TableauPoints3D points;
points.y[i] = 3.0f;
```

4.2 Représentation CSR-MDP

Il existe deux principales façons d'exploiter la mémoire cache et ainsi de réduire le temps d'accès à la mémoire : (1) les données qui sont souvent utilisées ensemble peuvent être regroupées de façon contiguë pour garantir que le plus de mémoire possible à l'intérieur des lignes de cache chargées soit utile pour le calcul en cours, et (2) les algorithmes peuvent être modifiés pour minimiser le nombre d'accès à la mémoire, par exemple, en travaillant plus longtemps avec des données déjà chargées avant d'en nécessiter de nouvelles. Ce chapitre se concentre sur la première de ces façons, tandis que le chapitre 6 détaille la seconde.

Dans cette section, nous présentons une nouvelle représentation en mémoire des PDM. Celle-ci est inspirée par la représentation *Compressed Sparse Row* (CSR) des graphes orientés (Wheatman et Xu, 2018), connue pour offrir d'excellentes performances de cache et une faible consommation mémoire (Qian *et al.*, 2018). La représentation proposée peut être classée comme faisant partie des approches à disposition de mémoire « *structure de tableaux* » (SoA).

La figure 4.1 illustre la représentation en mémoire proposée, que nous appelons **CSR-MDP**. Cette représentation inclut cinq tableaux, $(\mathcal{S}, \mathcal{C}, \mathcal{A}, \mathcal{N}, \mathcal{P})$, où :

6. https://www.intel.com/content/www/us/en/developer/articles/technical/memory-layout-transformations.html

— $[\mathcal{S}[i],\ \mathcal{S}[i+1])$ est un intervalle donnant les indices j des actions possibles à l'état i ;

— $\mathcal{C}[j]$ donne le coût de l'exécution de l'action j ;

— $[\mathcal{A}[j],\ \mathcal{A}[j+1])$ est un intervalle donnant les indices k des effets probabilistes pouvant survenir suite à l'exécution de l'action j ;

— $\mathcal{N}[k]$ donne l'indice de l'état atteint lorsque l'effet probabiliste k survient ;

— $\mathcal{P}[k]$ donne la probabilité que l'effet probabiliste k survienne.

Dans la figure 4.1, les lignes rouges sous $\mathcal{S}$, $\mathcal{A}$ et $\mathcal{C}$ symbolisent les données associées à l'état i. Les deux nombres dans la région rouge de $\mathcal{S}$ représentent l'intervalle semi-ouvert des indices dans $\mathcal{A}$ et $\mathcal{C}$ qui correspondent, respectivement, aux actions applicables à l'état i et au coût de chacune d'entres-elles. De même, les lignes bleues sous $\mathcal{A}$, $\mathcal{N}$ et $\mathcal{P}$ symbolisent les données associées à l'action j_1. Les deux nombres dans la région bleue de $\mathcal{A}$ représentent l'intervalle semi-ouvert des indices dans $\mathcal{N}$ et $\mathcal{P}$ qui correspondent aux résultats possibles de l'action j_1 (voisins possibles et probabilités de transition correspondantes).

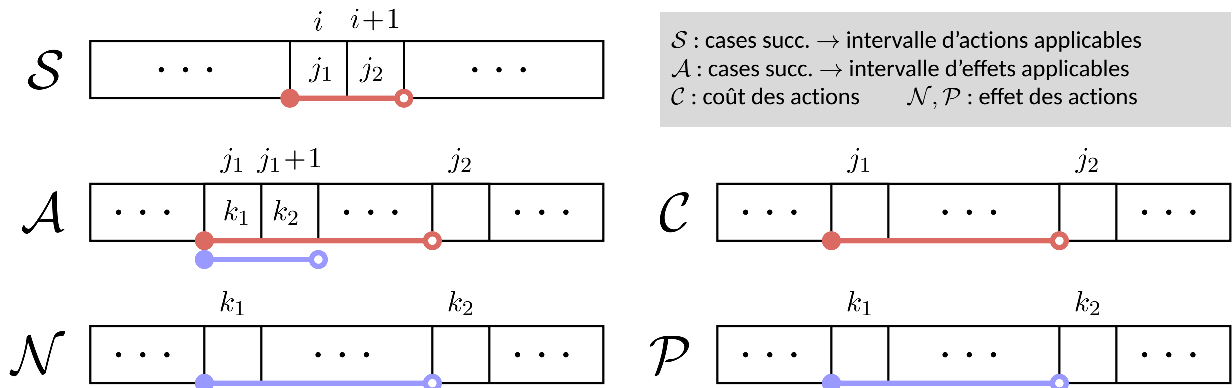
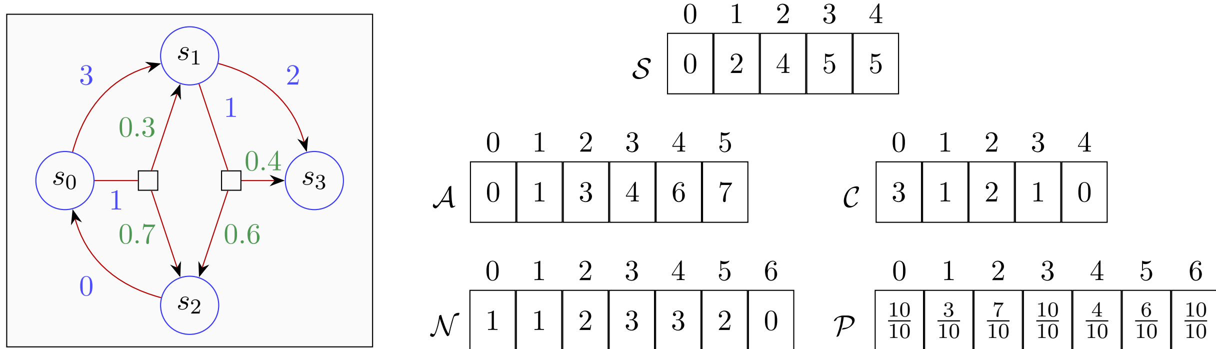


Figure 4.1 – Représentation visuelle de la structure CSR-MDP proposée.

Les tableaux $\mathcal{A}$ et $\mathcal{C}$ (resp. $\mathcal{N}$ et $\mathcal{P}$), peuvent être fusionnés en un seul tableau de paires de variables, mais nous avons choisi de ne pas le faire dans notre implémentation pour les raisons suivantes : (1) nous n'avons pas toujours besoin d'accéder aux deux variables en même temps (et dans un tel cas, nous pourrions doubler la quantité d'informations utiles en mémoire cache en gardant la séparation), et (2) l'utilisation éventuelle d'instructions SIMD dans un solveur optimisé nécessiterait (ou, à tout le moins, bénéficierait) d'accéder à des données de coût et d'effets probabilistes d'actions qui sont contiguës en mémoire.

La figure 4.2 montre un exemple de PDM-PCCS et sa représentation CSR-MDP associée. Les nombres en bleu représentent le coût de chaque action. Seules deux actions sont probabilistes, une à l'état 0 et l'autre à l'état 1. Les nombres en vert représentent la probabilité de chacun des effets probabilistes possibles.



	0	1	2	3	4
$\mathcal{S}$	0	2	4	5	5

	0	1	2	3	4	5
$\mathcal{A}$	0	1	3	4	6	7

	0	1	2	3	4
$\mathcal{C}$	3	1	2	1	0

	0	1	2	3	4	5	6
$\mathcal{N}$	1	1	2	3	3	2	0

	0	1	2	3	4	5	6
$\mathcal{P}$	$\frac{10}{10}$	$\frac{3}{10}$	$\frac{7}{10}$	$\frac{10}{10}$	$\frac{4}{10}$	$\frac{6}{10}$	$\frac{10}{10}$

Figure 4.2 – Exemple d'une instance de PDM et la représentation CSR-MDP correspondante.

Le principal avantage de CSR-MDP par rapport à d'autres représentations de PDM est que toutes les données sont regroupées, maximisant ainsi l'utilité de la mémoire présente en cache. Un autre avantage est que les données sont stockées de façon homogène dans des tableaux séparés, ce qui facilite la vectorisation du code par le programmeur (ou le compilateur) en utilisant des instructions SIMD. En terme de mémoire, CSR-MDP a une consommation plus faible que les représentations existantes. Par exemple, la représentation par listes chaînées utilise une importante quantité de mémoire pour stocker les pointeurs entre les cellules, tandis que la représentation par tables de hachage doit maintenir en mémoire certaines cases vides.

Dans notre implémentation de CSR-MDP, les cases de chaque tableau prennent quatre octets (32 bits). En effet, nous utilisons des entiers de quatre octets dans les tableaux $\mathcal{S}, \mathcal{A}$ et $\mathcal{N}$ et des nombres à virgule flottante à précision simple (*float*) de quatre octets dans $\mathcal{C}$ et $\mathcal{P}$. Supposons que nous ayons un PDM M contenant n états, où le nombre moyen d'actions applicables par état est m et le nombre moyen d'effets probabilistes par action est k. Alors, la mémoire totale (en octets) nécessaire pour stocker entièrement M dans la représentation CSR-MDP est :

$$
\begin{aligned}
Taille(M) &= 4(n+1) && \text{(Tableau } \mathcal{S}\text{)} \\
&+ 4(nm) && \text{(Tableau } \mathcal{C}\text{)} \\
&+ 4(nm+1) && \text{(Tableau } \mathcal{A}\text{)} \\
&+ 4(nmk) && \text{(Tableau } \mathcal{N}\text{)} \\
&+ 4(nmk) && \text{(Tableau } \mathcal{P}\text{)} \\
&= 8nm(k+1) + 4n + 8. && \text{(en octets)}
\end{aligned}
$$

Par exemple, stocker une instance du problème *Single-Armed Pendulum* (avec $m = 2$ et $k = 3$, tel que décrit en détail dans la section suivante) contenant n états requiert $68n + 8$ octets.

4.3 Évaluation

Dans cette section, nous évaluons les performances de la représentation mémoire CSR-MDP en comparant notre implémentation, exploitant cette structure, à une implémentation de référence. Cette dernière est celle utilisée dans l'article original présentant TVI (Dai *et al.*, 2011). Bien que le code ne soit pas public, les auteurs nous l'ont gracieusement fourni. La structure utilisée dans cette dernière est de type « tableau de structures » (AoS), ce qui est représentatif des implémentations disponibles actuellement.

Nous comparons les deux représentations mémoire en évaluant leur impact sur la performance des trois algorithmes suivants : (1) *Value Iteration* (algorithme 1.2, variante asynchrone Gauss-Seidel), (2) LRTDP (variante de l'algorithme 2.5), et (3) TVI (algorithme 2.3). Pour LRTDP, nous avons utilisé l'heuristique admissible et indépendante du domaine $h_{\min}$, décrite pour la première fois dans l'article original introduisant LRTDP (Bonet et Geffner, 2003) :

$$h_{\min}(s) = \begin{cases} 0, & \text{si } s \in G, \\ \min_{a \in A_s} \left[C(s,a) + \min_{s' \in succ_a(s)} V(s') \right], & \text{sinon,} \end{cases}$$

où A_s désigne l'ensemble des actions applicables à l'état s, et $succ_a(s)$ est l'ensemble des successeurs lors de l'application de l'action a à l'état s.

Nous avons implémenté les trois algorithmes testés (VI, LRTDP et TVI) en C++ et compilé ces implémentations à l'aide du compilateur GNU g++ (version 11.2, avec des optimisations de niveau 3). Nous n'avons pas tenté de vectoriser le code manuellement à l'aide d'instructions SIMD, mais le compilateur a auto-vectorisé certaines parties de celui-ci. Tous les tests ont été effectués sur un ordinateur équipé de 16 GiB de mémoire vive DDR4, et d'un processeur Intel Core i5-7600K (4,2 GHz ; 32KiB de cache L1 et 256KiB de cache L2 par cœur ; 6MiB de cache L3 partagé). Pour chaque domaine de test, nous avons mesuré le temps d'exécution jusqu'à la convergence vers une fonction de valeur ϵ-optimale pour chacun des trois algorithmes comparés (la valeur de ϵ a été fixée à 10^{-6} pour tous les tests). Chaque taille de domaine a été testée 15 fois avec des instances de PDM générées aléatoirement. Pour minimiser les facteurs aléatoires, nous rapportons les valeurs médianes obtenues sur ces 15 instances.

Nous avons évalué les performances des algorithmes VI, LRTDP et TVI sur trois différents domaines de planification. Le premier d'entre eux est le domaine générique ***Layered*** proposé dans l'article de TVI (Dai *et al.*, 2011). Ce domaine est paramétré par quatre valeurs différentes : n, n_l, n_a et n_s, décrivant respectivement le nombre d'*états*, le nombre de *couches*, le nombre d'*actions applicables par état*, et le nombre maximum d'*états successeurs par action* (c'est-à-dire, chaque action a peut mener à k_a états différents, où k_a est tiré d'une distribution uniforme dans $\{1, 2, \ldots, n_s\}$). Les états résultant de chaque transition sont uniformément échantillonnés parmi les successeurs possibles. Les états de ce domaine sont uniformément répartis en n_l couches, $\{1, 2, \ldots, n_l\}$. Un état dans la couche i ne peut avoir que des états successeurs dans les couches $\{i, i+1, \ldots, n_l\}$, ce qui signifie que les PDM dans ce domaine ont au moins n_l composantes fortement connexes (CFC).

Le deuxième domaine que nous avons considéré est le domaine du Pendule à un bras (*Single-Armed Pendulum*, ***SAP***) (Wingate et Seppi, 2005). Ce domaine représente un problème de contrôle optimal en temps minimum à deux dimensions dans lequel un agent a toujours deux actions possibles : appliquer un couple positif ou négatif à un pendule en rotation. L'objectif de l'agent est d'équilibrer le pendule vers le haut. L'espace d'état est défini par deux variables : l'angle θ et la vitesse angulaire ω.

Enfin, le dernier domaine utilisé dans notre évaluation est une variante du domaine ***Wetfloor*** (Bonet et Geffner, 2006), qui est un domaine bien établi dans la communauté pour l'évaluation d'algorithmes de planification. Dans celui-ci, l'espace d'état est une grille de navigation carrée dans laquelle les cellules peuvent être à l'un des trois niveaux d'humidité suivants : (1) sèche, (2) légèrement humide ou (3) très humide. Dans la grille, les cellules sont indépendamment choisies comme humides avec une probabilité p. Parmi les cellules humides, un deuxième paramètre q contrôle la probabilité d'être légèrement humide (q) ou très humide ($1 - q$). L'agent commence à une certaine position et l'objectif est d'atteindre une autre position avec un nombre minimal d'actions. Les actions sont Haut, Bas, Gauche, Droite. Elles sont déterministes sur les cellules sèches. Sur les cellules humides, le résultat des actions est probabiliste et dépend des paramètres r_l et r_t, qui donnent respectivement la probabilité de glisser lorsqu'une cellule est légèrement ou très humide. Dans notre évaluation, nous avons utilisé une version généralisée du domaine *Wetfloor* où, au lieu d'avoir une seule grille carrée, nous avons de nombreuses grilles de ce type reliées entre elles (cela peut être analogue à de nombreuses pièces avec planchers potentiellement humides dans une maison).

Le tableau 4.1 rapporte les facteurs d'accélération moyens fournis par CSR-MDP par rapport à l'implémenta-

Domaine	VI	LRTDP	TVI
Layered (var. états)	5.87	7.92	4.47
Layered (var. couches)	6.77	> 4.08	> 3.88
SAP	4.36	> 1.05	5.34
Wetfloor	> 15.33	> 13.81	12.36
Moyenne	> 8.70	> 2.86	> 6.65

Table 4.1 – Facteurs d'accélération moyens obtenus par chaque solveur sur chaque domaine en utilisant la représentation mémoire proposée, CSR-MDP, lorsque comparée à l'implémentation de base. Les nombres avec un symbole "$>$" sont des bornes inférieures du vrai facteur d'accélération.

tion de base. Nous comparons les accélérations obtenues pour chacun des domaines et des solveurs testés, pour déterminer si un solveur spécifique bénéficie davantage de la représentation CSR-MDP. Le tableau 4.2 présente les résultats détaillés obtenus sur différentes instances des domaines *Layered*, *SAP* et *Wetfloor*. Les quatre premières colonnes rapportent les caractéristiques des instances considérées : le nom du domaine, le nombre d'états des instances générées, le nombre de CFC (excluant la composante fortement connexe contenant uniquement l'état but s_g) et la taille de la plus grande CFC. Les six colonnes suivantes présentent le temps d'exécution médian (en ms) obtenu par les trois algorithmes testés (VI, LRTDP et TVI) lorsqu'ils sont exécutés avec l'implémentation de base et avec notre implémentation CSR-MDP. Les figures 4.3, 4.4, 4.5 et 4.6 illustrent graphiquement les résultats obtenus. Les sous-indices "b" et "csr" dans les figures désignent respectivement les implémentations de base et CSR-MDP.

La figure 4.3 présente les résultats obtenus pour le domaine *Layered* en fixant le nombre de couches à 10 et en faisant varier le nombre d'états de 100k à 1M, tandis que la figure 4.4 montre les résultats pour ce domaine en fixant le nombre d'états à 1M et en faisant varier le nombre de couches de 1 à 16 384. Dans le premier cas (variation du nombre d'états), nous pouvons voir que TVI est capable d'exploiter les 10 couches, ce qui lui permet de battre clairement VI et LRDTP. L'implémentation de base est beaucoup plus lente que l'implémentation CSR-MDP dans ce domaine. Dans le second cas (variation du nombre de couches), nous pouvons en outre observer que LRTDP et TVI deviennent plus lents au niveau de 128 couches, ce qui s'explique par le fait qu'avec ce nombre de couches, le nombre d'états par couche (et donc la mémoire nécessaire pour stocker l'information des états dans la couche) dépasse la taille de l'un des trois niveaux de cache dans le CPU. L'algorithme VI est moins affecté par cet inconvénient puisqu'il ne considère pas les couches. Cependant, LRTDP est affecté par cela, même s'il ne considère pas explicitement les couches, car le nombre de couches a un impact sur la profondeur de recherche atteinte par LRTDP avant qu'il n'atteigne un objectif.

Charactéristiques des instances de PDM				Base			CSR-MDP		
Domaine	$\|S\|$	CFC	$\|CFC_{\max}\|$	VI	LRTDP	TVI	VI	LRTDP	TVI
Layered	100 000	10	10 000	6 253	5 829	2 355	1 051	905	**400**
Layered	200 000	10	20 000	17 499	16 301	5 828	3 742	3 018	**1 323**
Layered	300 000	10	30 000	26 655	28 131	9 001	3 243	3 006	**1 633**
Layered	400 000	10	40 000	40 489	40 160	13 142	6 154	4 491	**2 543**
Layered	500 000	10	50 000	72 815	68 544	19 343	8 262	7 614	**3 447**
Layered	600 000	10	60 000	58 511	77 312	20 602	10 135	12 103	**4 575**
Layered	700 000	10	70 000	68 574	85 359	24 532	14 606	12 093	**5 928**
Layered	800 000	10	80 000	93 054	122 189	29 821	14 392	14 601	**6 466**
Layered	900 000	10	90 000	115 157	142 215	35 335	21 385	17 718	**8 591**
Layered	1 000 000	10	100 000	144 708	158 977	40 554	26 602	18 546	**9 997**
Layered	1 000 000	1	1 000 000	273 731	-	-	**41 788**	232 539	91 612
Layered	1 000 000	2	500 000	207 160	-	134 678	**32 291**	93 500	37 682
Layered	1 000 000	4	250 000	120 241	230 598	66 380	28 539	57 884	**19 941**
Layered	1 000 000	8	125 000	143 112	206 107	45 595	29 266	24 954	**12 210**
Layered	1 000 000	16	62 500	117 847	150 479	31 583	19 123	12 806	**6 898**
Layered	1 000 000	32	31 250	136 257	107 789	25 469	15 946	6 814	**4 128**
Layered	1 000 000	64	15 625	95 319	80 321	18 271	14 543	4 592	**2 562**
Layered	1 000 000	128	7 813	77 791	84 933	40 361	13 413	11 500	**9 637**
Layered	1 000 000	256	3 907	63 078	67 585	22 438	8 595	4 391	**4 214**
Layered	1 000 000	512	1 954	83 238	65 363	19 015	6 417	2 676	**2 353**
Layered	1 000 000	1 024	977	70 765	63 297	18 698	7 009	3 732	**2 630**
Layered	1 000 000	2 048	489	54 964	63 966	15 267	6 288	2 559	**2 141**
Layered	1 000 000	4 096	245	60 675	52 289	15 141	7 824	2 995	**2 184**
Layered	1 000 000	8 192	123	71 687	59 868	16 050	5 981	2 035	**1 997**
Layered	1 000 000	16 384	62	74 595	63 480	15 198	6 756	2 043	**1 992**
SAP	10 000	1	10 000	117	343	121	32	227	**12**
SAP	40 000	1	40 000	1 155	8 552	1 170	234	4 962	**109**
SAP	90 000	1	90 000	3 673	42 597	3 735	857	16 184	**476**
SAP	160 000	1	160 000	7 663	142 882	7 702	1 903	78 565	**1 538**
SAP	250 000	1	250 000	15 387	-	15 665	3 919	292 439	**3 124**
SAP	360 000	1	360 000	25 262	-	25 292	6 055	-	**4 852**
SAP	490 000	1	490 000	39 694	-	39 914	9 591	-	**7 425**
SAP	640 000	1	640 000	54 786	-	55 188	13 832	-	**11 772**
SAP	810 000	1	810 000	81 573	-	81 939	18 756	-	**15 707**
SAP	1 000 000	1	1 000 000	111 414	-	111 861	22 945	-	**19 136**
Wetfloor	500 000	1	500 000	65 185	212 101	65 968	**7 721**	10 997	21 756
Wetfloor	500 000	2	250 000	109 979	-	72 191	**10 297**	9 326	14 988
Wetfloor	500 000	3	166 667	-	-	138 184	**12 414**	17 502	13 514
Wetfloor	500 000	4	125 000	-	-	144 481	15 582	22 166	**12 711**
Wetfloor	500 000	5	100 000	-	-	184 969	15 985	20 536	**12 116**
Wetfloor	500 000	6	83 334	-	-	161 854	19 680	28 703	**12 075**
Wetfloor	500 000	7	71 429	-	-	186 367	19 972	24 696	**11 400**
Wetfloor	500 000	8	62 500	-	-	206 437	19 524	22 522	**10 845**
Wetfloor	500 000	9	55 556	-	-	218 317	23 127	33 287	**10 790**
Wetfloor	500 000	10	50 000	-	-	226 503	23 634	27 467	**9 676**

Table 4.2 – Temps d'exécution médian (ms) mesuré pour chaque domaine testé. Le temps le plus rapide pour chaque instance est mis en gras. Le symbole "-" est présent pour indiquer qu'un solveur n'a pas réussi à trouver une solution dans le temps imparti de 5 minutes.

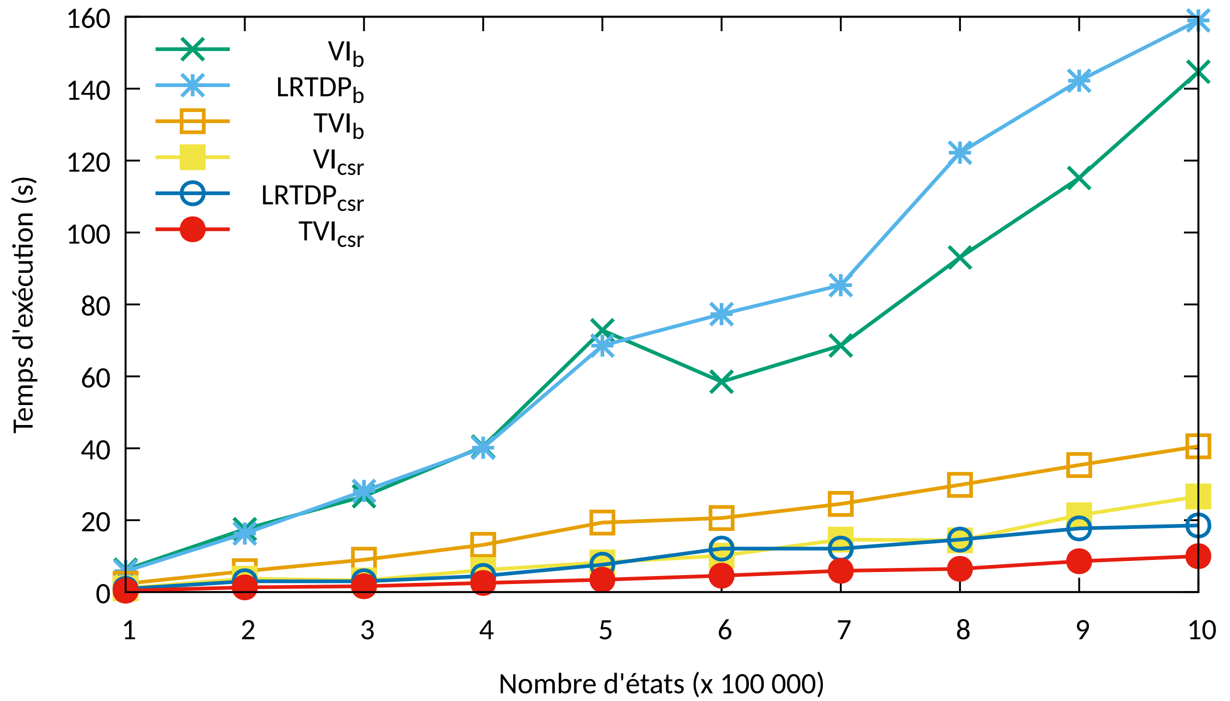


Figure 4.3 – Temps d'exécution (en secondes) pour le domaine *Layered* en fixant le nombre de couches à 10 et en faisant varier le nombre d'états.

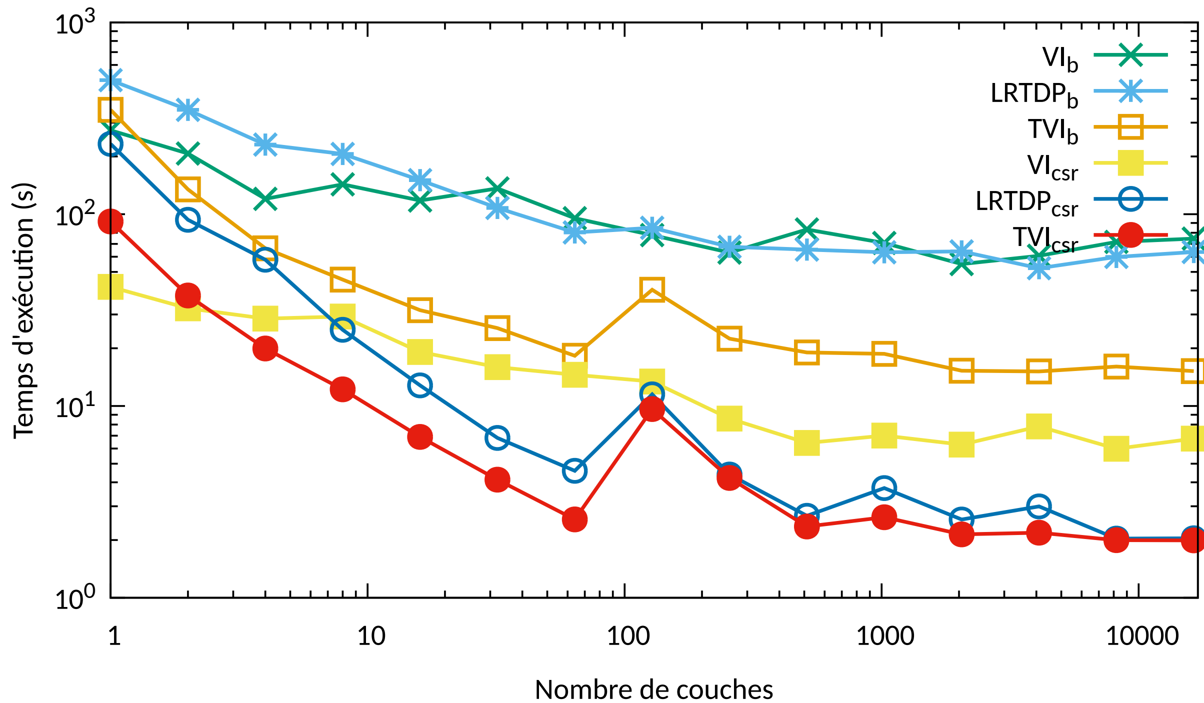


Figure 4.4 – Temps d'exécution (en secondes) pour le domaine *Layered* en fixant le nombre d'états à 1M et en faisant varier le nombre de couches.

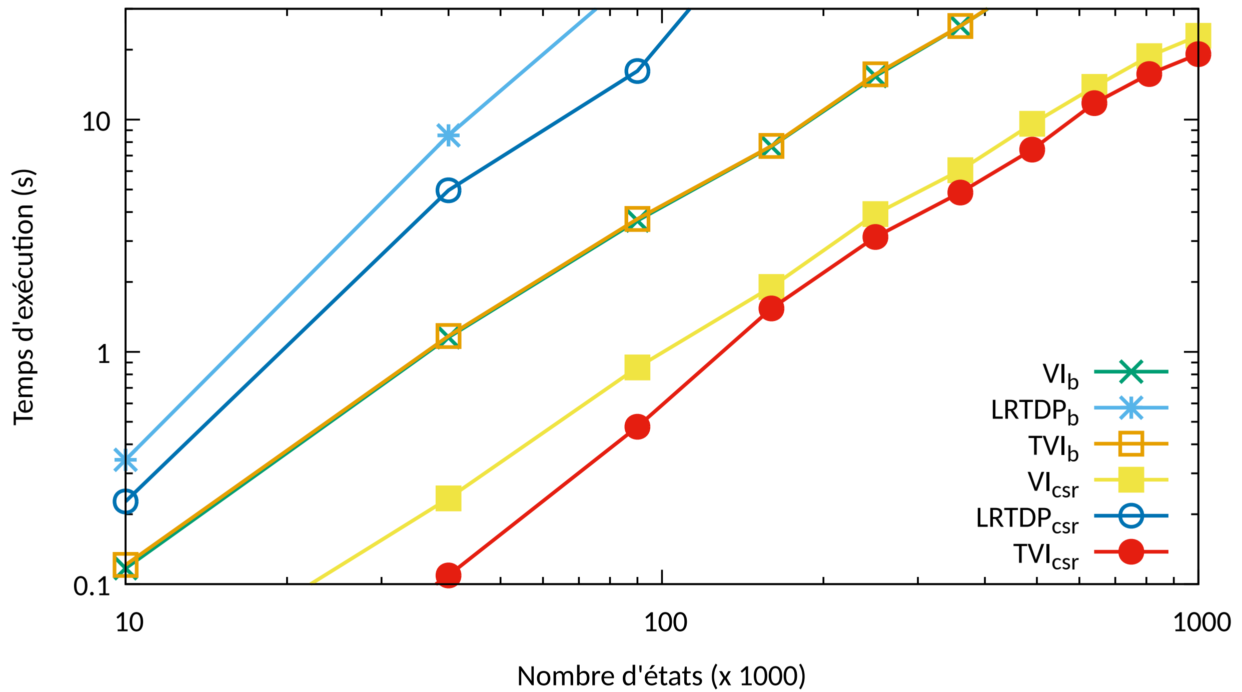


Figure 4.5 – Temps d'exécution (en secondes) pour le domaine *SAP* en faisant varier le nombre d'états.

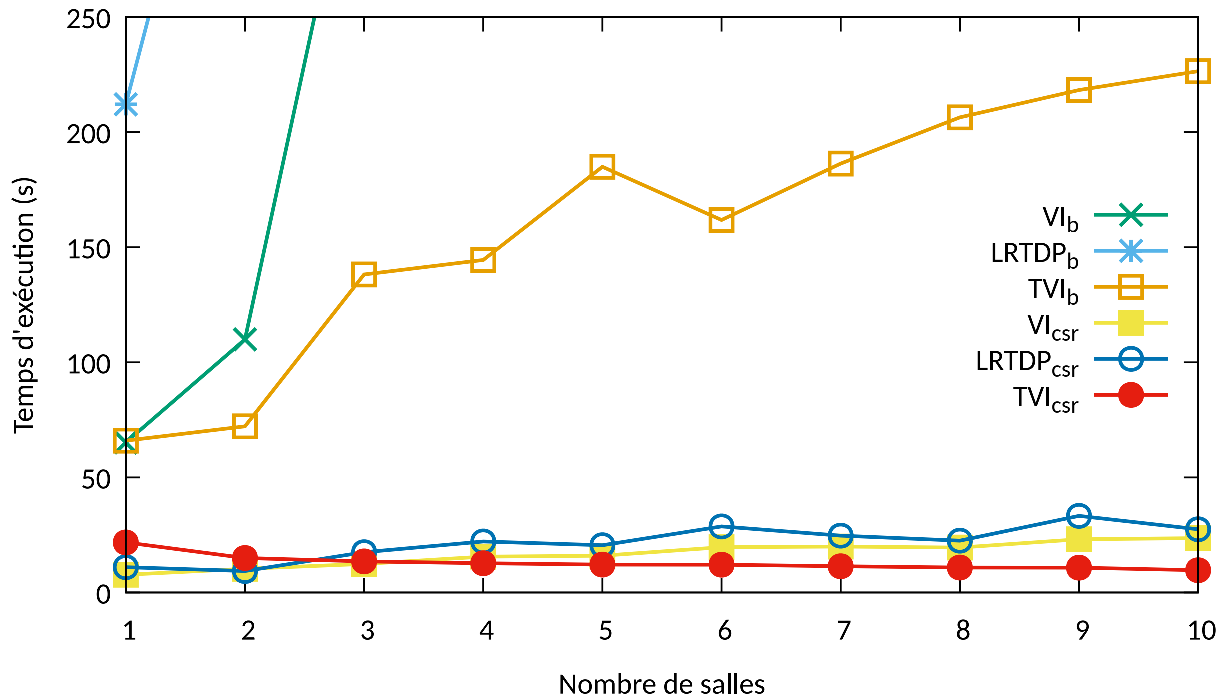


Figure 4.6 – Temps d'exécution (en secondes) pour le domaine *Wetfloor* en fixant le nombre d'états à 500k et en faisant varier le nombre de salles.

Pour le domaine *SAP* (figure 4.5), LRTDP a beaucoup de difficulté à trouver une solution en un temps raisonnable (les auteurs de TVI ont observés le même phénomène). Deux raisons peuvent expliquer ce fait : (1) l'heuristique $h_{\min}$ est relativement peu informative pour ce domaine et (2) le nombre minimum d'actions nécessaires pour atteindre un objectif dans les instances de *SAP* est relativement élevé, et il est connu que LRTDP offre de meilleures performances lorsque le nombre d'actions nécessaires pour atteindre un objectif est faible (Dai *et al.*, 2011). Dans l'implémentation de base, VI et TVI ont fourni des performances équivalentes, ce qui était attendu puisque le domaine *SAP* n'a qu'une seule CFC (toutes les actions sont réversibles), et TVI devient essentiellement VI lorsqu'il n'y a qu'une seule CFC dans le domaine. Étonnamment, l'implémentation CSR-MDP de TVI était plus rapide que l'implémentation CSR-MDP de VI, même si le calcul de la seule CFC (en utilisant l'algorithme de Tarjan) aurait dû causer un surcoût inutile. En effet, en moyenne, environ $50\,\%$ de balayages à travers l'espace d'état en moins ont été nécessaires avant que la fonction de valeur ne converge vers une précision ϵ. Ce phénomène sera expliqué plus précisément au chapitre 6.

En ce qui concerne le domaine *Wetfloor*, nous pouvons voir que, comme pour le domaine *Layered*, TVI a pu tirer parti du nombre de CFC (le nombre de salles) dans le domaine par rapport à VI et LRTDP. Cependant, l'implémentation de base a eu beaucoup de difficulté à résoudre les instances de ce domaine, ce qui a impacté VI, LRTDP, et même TVI. La perte de performance lorsque le nombre de pièces augmente est due à la manière de stocker les CFC en mémoire dans cette implémentation. Plus précisément, comme les états contenus dans une CFC ne sont pas stockés de manière contigus, l'augmentation du nombre de CFC provoque une augmentation considérable du nombre de défauts de cache. Pour ce domaine, les performances de VI et de LRTDP diminuent à mesure que le nombre de salles augmente. Dans le cas de VI, cela s'explique par le fait que lorsque le nombre de salles est élevé, la plupart des mises à jour de Bellman effectuées par VI sont inutiles (elles propagent des valeurs d'état qui n'ont pas encore convergé). Dans le cas de LRTDP, cela s'explique par le fait qu'un nombre plus élevé de salles correspond généralement à une plus grande profondeur de recherche pour se rendre de l'état initial s_0 à l'état but s_g dans ce domaine.

Somme toute, l'expérimentation menée sur divers domaines de planification probabilistes démontre que la représentation CSR-MDP proposée dans ce chapitre a permis aux algorithmes évalués (VI, LRTDP et TVI) de trouver une politique optimale respectivement 8.6, 2.8 et 6.6 fois plus rapidement, en moyenne, sur les trois domaines testés. Le prochain chapitre présente un nouvel algorithme de planification — pouvant être utilisé en combinaison avec la représentation CSR-MDP — permettant d'exploiter le parallélisme de fils d'exécution (décrite au chapitre 3) lorsque le domaine de planification le permet.

CHAPITRE 5
PLANIFICATEURS PARALLÈLES DE PDM

> *Unparallelized applications leave significant performance on the table for current processors. Furthermore, such serial applications will not improve in performance over time. Efficiently parallelized applications, in contrast, will make good use of current processors and should be able to scale automatically to even better performance on future processors. Over time, this will lead to large and decisive differences in performance.*
>
> — McCool *et al.* (2012)

Le chapitre 4 s'est concentré sur la représentation mémoire des PDM de sorte à accélérer les calculs d'une politique optimale en exploitant la hiérarchie de mémoire. Dans ce chapitre, nous continuons notre cheminement vers un planificateur de PDM plus rapide en exploitant le parallélisme de fils d'exécution (*threads*) mentionné à la section 3.2.3. Les principales contributions du chapitre sont (1) une version parallélisée de l'algorithme TVI, appelé *pcTVI*, et (2) un nouveau domaine de planification, *chained-MDP*. Les contributions de ce chapitre ont fait l'objet d'une publication présentée à la conférence de l'*International Federation of Classification Societies* (IFCS 2022) qui a eu lieu à Porto, au Portugal (Champagne Gareau *et al.*, 2023b). La section 5.1 présente les planificateurs de PDM existants qui exploitent le parallélisme de fils d'exécution ; la section 5.2 présente l'algorithme proposé, pcTVI, tandis que la section 5.3 présente le nouveau domaine de planification proposé et compare la performance de pcTVI avec celle de VI, LRTDP et TVI sur celui-ci.

5.1 Algorithmes parallèles existants

L'algorithme *Value Iteration* (VI, algorithme 1.1) peut se paralléliser de manière relativement directe en effectuant plusieurs mises à jour de Bellman en parallèle, mais stratégie n'est pas idéale. D'une part, l'ordre de propagation des états sera moins optimal que celui que fournirait Prioritized VI (algorithme 2.1) ou TVI (algorithme 2.3). D'autre part, les mises à jour de Bellman calculées dans un fil d'exécution peuvent nécessiter de connaître les valeurs à jour des états calculées dans un autre fil, ce qui peut causer d'autres difficultés nuisant à la performance (p. ex., dû à la synchronisation nécessaire et à des problèmes de faux partages).

À notre connaissance, le seul algorithme explicitement parallèle qui a été proposé dans la littérature pour résoudre les PDM-PCCS est l'algorithme **P3VI** (Wingate et Seppi, 2004). Cet algorithme, dont le nom signifie *Partitioned*, *Prioritized*, *Parallel Value Iterator*, est une version parallèle de l'algorithme *Partitioned VI* (algorithme 2.2). Il décompose l'espace d'états en partitions (définies manuellement par un expert du domaine), maintient une priorité associée à chacune d'elles dans une file prioritaire, et résout les k partitions les plus prioritaires en parallèle. Dès qu'un fil d'exécution a fait converger les valeurs des états d'une partition, il choisit parmi les partitions restantes la plus prioritaire, et la résout. L'algorithme termine lorsque toutes les partitions ont convergé, ce qui, contrairement au cas des CFC utilisées par l'algorithme TVI, peut nécessiter qu'une même partition soit considérée plus d'une fois.

L'algorithme P3VI a malheureusement deux principaux désavantages. Premièrement, tout comme Partitioned VI, le partitionnement doit être choisi manuellement, au cas par cas, selon le domaine de planification. Deuxièmement, les valeurs des états d'une partition peuvent partiellement dépendre des valeurs des états d'une autre partition, ce qui force à avoir beaucoup de communication inter fils d'exécution et nuit à la performance.

5.2 *Parallel-Chained* TVI (pcTVI)

L'algorithme TVI mentionné précédemment (algorithme 2.3) a plusieurs avantages comparativement à P3VI. D'une part, puisqu'il effectue le partitionnement de l'espace d'états en se basant sur la structure topologique du graphe correspondant à la déterminisation du PDM considéré, il est plus général que P3VI, car il élimine le besoin d'un partitionnement manuel dépendant du domaine de planification. D'autre part, il ne nécessite pas de maintenir explicitement une file prioritaire, ce qui améliore les performances par rapport à P3VI lorsque le domaine de planification possède plusieurs composantes fortement connexes (CFC) et que, donc, la décomposition obtenue par TVI est non triviale. Cependant, aucune version parallèle de TVI n'existe dans la littérature. Dans cette section, nous présentons un nouvel algorithme basé sur TVI, nommé **pcTVI** (*Parallel-Chained Topological Value Iteration*), capable de résoudre un PDM en parallèle (comme P3VI) mais en utilisant la décomposition de l'espace d'états en CFC utilisée par TVI.

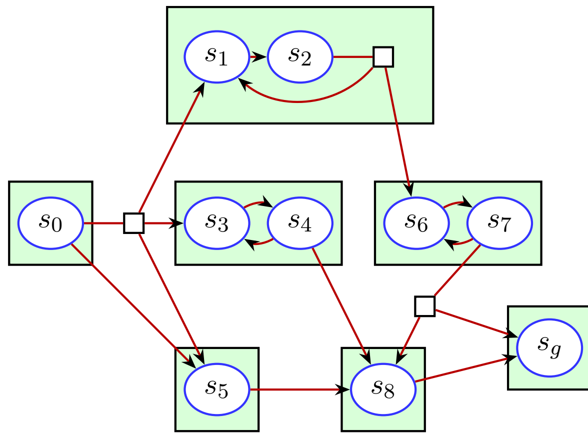
Étant donné que les ordinateurs modernes disposent de nombreux cœurs qui peuvent travailler en parallèle, nous pouvons théoriquement résoudre de nombreuses CFC en parallèle pour réduire considérablement le temps de calcul. Naïvement, nous pourrions choisir les CFC à résoudre en parallèle en utilisant une métrique de priorité (comme le fait P3VI). Or, le partage des valeurs entre les fils d'exécution entraînerait un surcoût.

De plus, on n'aurait pas de garantie à priori de ne devoir considérer qu'une seule fois chaque CFC. Au lieu de cela, pcTVI considère l'ordre topologique inverse des CFC (comme le fait TVI) pour minimiser les calculs redondants ou inutiles. Une façon de partager le travail entre fils d'exécution est de trouver des chaînes indépendantes de CFC qui peuvent être résolues en parallèle. L'avantage des chaînes indépendantes est qu'aucune coordination ou communication n'est nécessaire entre les fils d'exécution, ce qui élimine une grande partie du surcoût en temps d'exécution et simplifie la mise en œuvre.

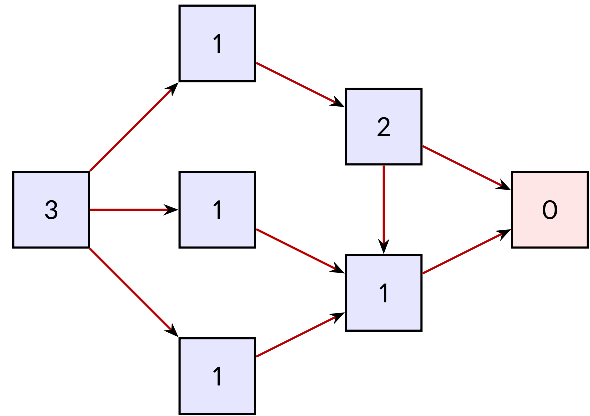
Algorithme 5.1 *Parallel-Chained Topological Value Iteration*.

```
1:  procédure pcTVI(M : PDM, t : Nombre de fils d'exécution)
2:      ▷ Trouver les CFC de M
3:      G ← Graphe(M)              ▷ G partage implicitement les mêmes structures de données que M
4:      CFC ← Tarjan(G)                          ▷ Les CFC sont trouvées en ordre topologique inverse
5:
6:      ▷ Construire le graphe des CFC de G
7:      G_c ← CondensationGraphe(G, CFC)
8:
9:      ▷ Résoudre en parallèle les chaînes indépendantes de CFC
10:     Bassin ← CréerBassinFils(t)                                              ▷ Créer t fils
11:     V ← NouvelleFonctionValeur()          ▷ Arbitrairement initialisée ; Partagée par tous les fils
12:     Q ← CréerFile()                                                ▷ Partagée par tous les fils
13:     Insérer(Q, Tête(CFC))                    ▷ La CFC contenant le but est insérée dans la file
14:     tant que NonVide(Q) faire                                ▷ Un seul fil exécute cette boucle
15:         C ← ExtraireProchain(Q)
16:         pour tout voisin ∈ Prédécesseurs(G_c, C) faire
17:             Décrémenter NombreSuccesseursRestants(voisin) de 1
18:             si NombreSuccesseursRestants(voisin) = 0 alors
19:                 AssignerTâcheÀFilDisponible(Bassin, VIPartiel(M, voisin, V))
20:                 Enfiler(Q, voisin)             ▷ Les voisins de la CFC C sont prêts à être considérés
21:
22:     ▷ Calculer et retourner la politique correspondant à la fonction de valeur actuelle
23:     retourne π^V                                      ▷ π^V(s) ← arg min_{a∈A} Q(s, a)
```

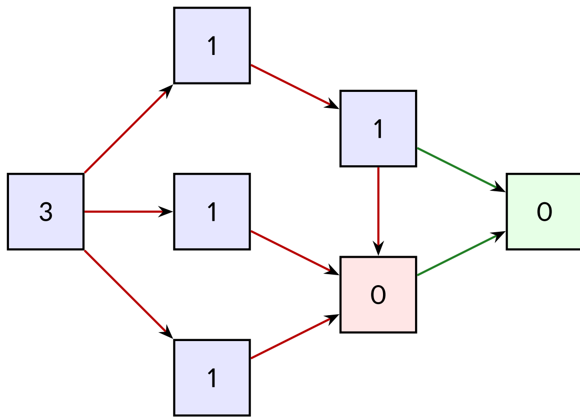
L'algorithme *Parallel-Chained TVI* (algorithme 5.1) fonctionne comme suit. Tout d'abord, nous trouvons le graphe G correspondant à la structure topologique du PDM, le décomposons en CFC et trouvons l'ordre topologique inverse de ces dernières (comme pour TVI). Nous construisons ensuite la condensation du graphe G, c'est-à-dire le graphe G_c dont chaque sommet représente une CFC de G, et où un arc est présent entre deux sommets C_1 et C_2 s'il existe un arc dans G entre un état $s_1 \in C_1$ et un état $s_2 \in C_2$. Nous stockons également dans chaque sommet C_i de G_c l'ensemble des prédécesseurs (les composantes ayant une transition directe vers la composante considérée) et un compteur $C_i^{\#}$ qui indique combien de successeurs n'ont pas encore convergé à une précision ϵ. Nous utilisons ce graphe G_c (généralement petit) pour détec-



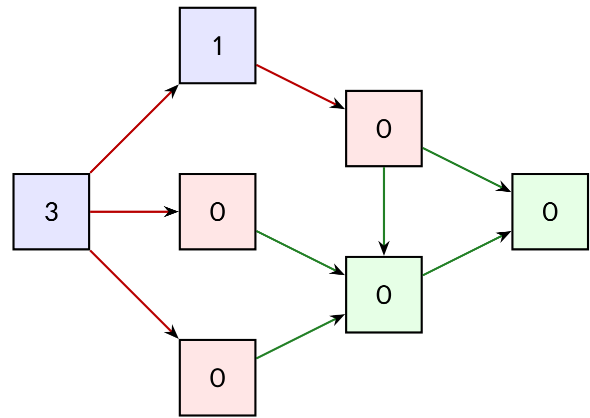
(a) PDM avec CFC représentées.



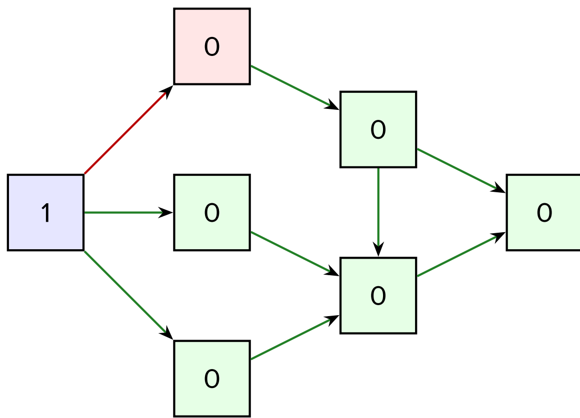
(b) Étape 1.



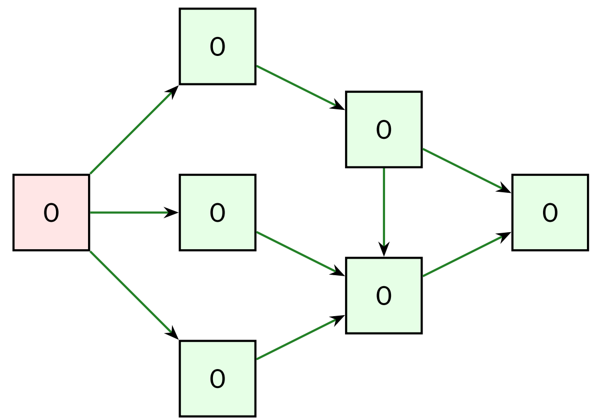
(c) Étape 2.



(d) Étape 3.



(e) Étape 4.



(f) Étape 5.

Figure 5.1 – Exemple de l'exécution de l'algorithme pcTVI. Chaque rectangle représente une CFC. À chaque étape, les éléments en bleu n'ont pas encore été traités, ceux en rouge sont en cours de traitement, et ceux en vert ont convergé. Le nombre à l'intérieur de chaque CFC représente le compteur du nombre de successeurs qui n'ont pas encore convergé à une précision ϵ. On suppose pour simplifier l'exemple que chaque composante prend le même temps à résoudre.

ter quelles CFC sont prêtes à être considérées, c'est-à-dire celles dont dont tous les successeurs ont déjà convergé (leur compteur associé $C_i^{\#}$ est 0). Lorsqu'une nouvelle CFC est prête, elle est insérée dans une file d'attente de travail à partir de laquelle les fils d'exécution en attente acquièrent leur prochaine tâche. La figure 5.1 montre un exemple de l'exécution de pcTVI.

5.3 Évaluation

Dans cette section, nous comparons la performance de pcTVI à celle des trois mêmes algorithmes utilisés pour l'évaluation de CSR-MDP : VI, LRTDP et TVI. Tous les tests ont été exécutés sur un ordinateur équipé de quatre processeurs Intel Xeon E5-2620V4 (chacun d'entre eux ayant 8 cœurs à 2.1 GHz, pour un total de 32 cœurs). Les autres paramètres de l'évaluation (par exemple, le nombre de répétitions par test, le langage ou le compilateur utilisé) sont les mêmes que ceux mentionnés à la section 4.3.

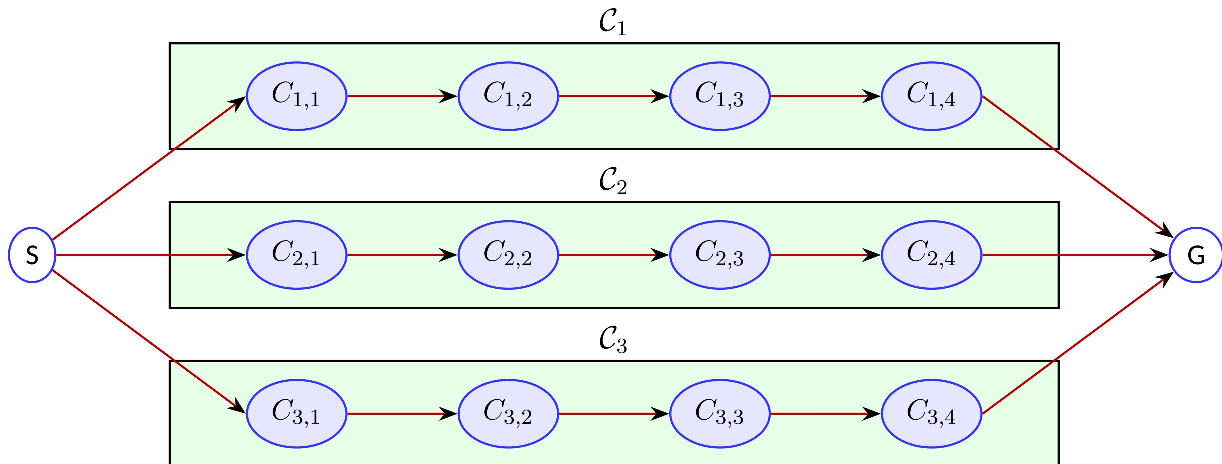


Figure 5.2 – Une instance du domaine de planification *Chained-MDP* où $k = 3$ et $n_C = 4$. Chaque ellipse représente une composante fortement connexe.

Comme il n'existe pas de domaine de planification standard dans la littérature scientifique adapté à l'évaluation du potentiel d'un algorithme de planification parallèle pour PDM, nous proposons un nouveau domaine paramétrique général que nous utilisons pour comparer empiriquement les algorithmes. Ce domaine, que nous appelons *Chained-MDP*, utilise cinq paramètres : (1) k, le nombre de chaînes indépendantes $\{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k\}$ dans le PDM ; (2) n_C, le nombre de CFC $\{C_{i,1}, C_{i,2}, \ldots, C_{i,n_C}\}$ dans chaque chaîne $\mathcal{C}_i$; (3) n_{epc}, le nombre d'états par CFC ; (4) n_a le nombre d'actions applicables par état, et (5) n_e le nombre d'effets probabilistes par action. Les successeurs possibles d'un état s dans $C_{i,j}$, noté $succ(s)$, peuvent être dans cette même composante ou en sortir, auquel cas ils seront soit dans la composante suivante de la chaîne, $C_{i,j+1}$, si elle existe, ou l'état but s_g sinon. Lors de la génération de la fonction de transition d'une paire

état-action (s, a), nous avons échantillonné n_e états uniformément à partir de $succ(s)$ avec des probabilités aléatoires. Une représentation d'une instance de *Chained-MDP* est montrée à la figure 5.2. On peut voir les instances de ce domaine comme modélisant des situations où différentes stratégies (correspondant à une chaîne) peuvent être utilisées pour atteindre un objectif, mais où, une fois engagé dans une stratégie, il n'est pas possible de passer à une autre. Dans chacun de nos tests, nous avons fixé les paramètres $n_C = 2$, $n_a = 5$ et $n_e = 5$.

La figure 5.3 présente les résultats obtenus pour le domaine *Chained-MDP* en faisant varier le nombre d'états et en fixant le nombre de chaînes (32). Nous pouvons observer que lorsque le nombre d'états est faible, pcTVI n'offre pas un avantage important par rapport aux algorithmes existants, car le surcoût de création et de gestion des fils d'exécution compense la plupart des gains possibles. Cependant, à mesure que le nombre d'états augmente, l'écart dans le temps d'exécution entre pcTVI et les trois autres algorithmes augmente. Cela indique que pcTVI est particulièrement utile sur les PDM de grande taille, qui sont fréquents dans les domaines de planification plus réalistes.

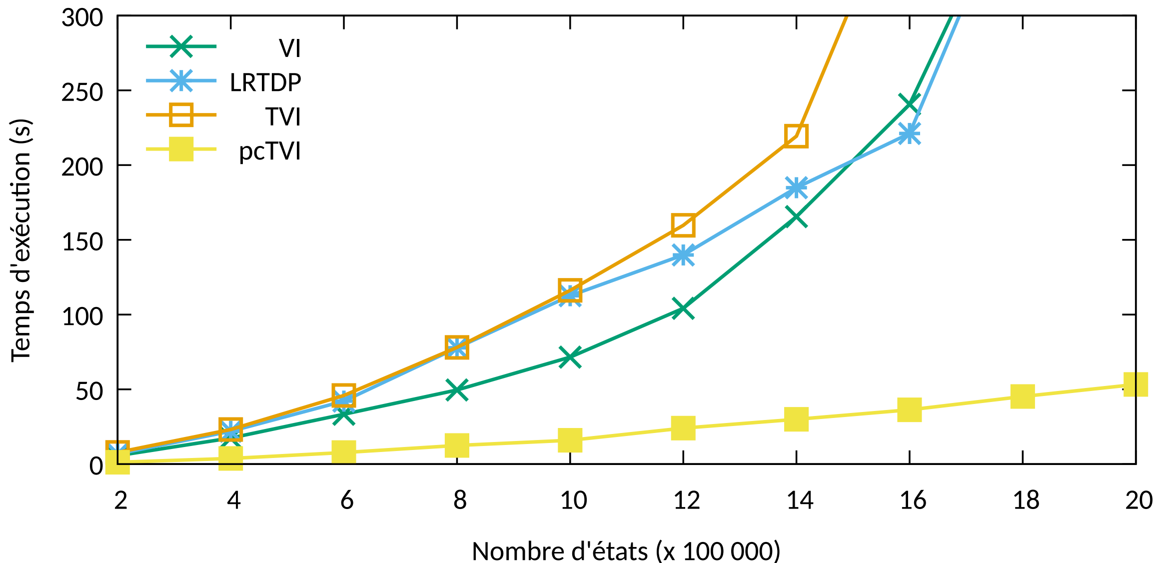


Figure 5.3 – Temps d'exécution moyen (s) pour le domaine *Chained-MDP* avec un nombre d'états variables et un nombre de chaînes fixe (32).

La figure 5.4 présente quant à elle les résultats obtenus pour ce même domaine *Chained-MDP* en variant le nombre de chaînes et en fixant le nombre d'états (1M). Lorsque le nombre de chaînes augmente, le nombre total de CFC augmente implicitement, ce qui implique que le nombre d'états par CFC diminue. Cela explique pourquoi chaque algorithme testé devient plus rapide : TVI devient plus rapide par conception, car il résout

les CFC une par une sans faire de mises à jour de Bellman inutiles, et VI et LRTDP deviennent plus rapides en raison d'une plus grande localité en mémoire des états considérés. En plus des raisons ci-dessus, pcTVI bénéficie d'un gain supplémentaire en raison du nombre plus important d'opportunités de parallélisation permises par l'augmentation du nombre de chaînes. Nous pouvons également observer que pour les domaines avec seulement quatre chaînes, pcTVI surpasse clairement les autres méthodes. Ainsi, le sucroît de performance permis par pcTVI peut également être obtenu sans devoir mobiliser une infrastructure massivement parallèle. Par exemple, un processeur grand public n'ayant que quatre cœurs suffit pour en bénéficier.

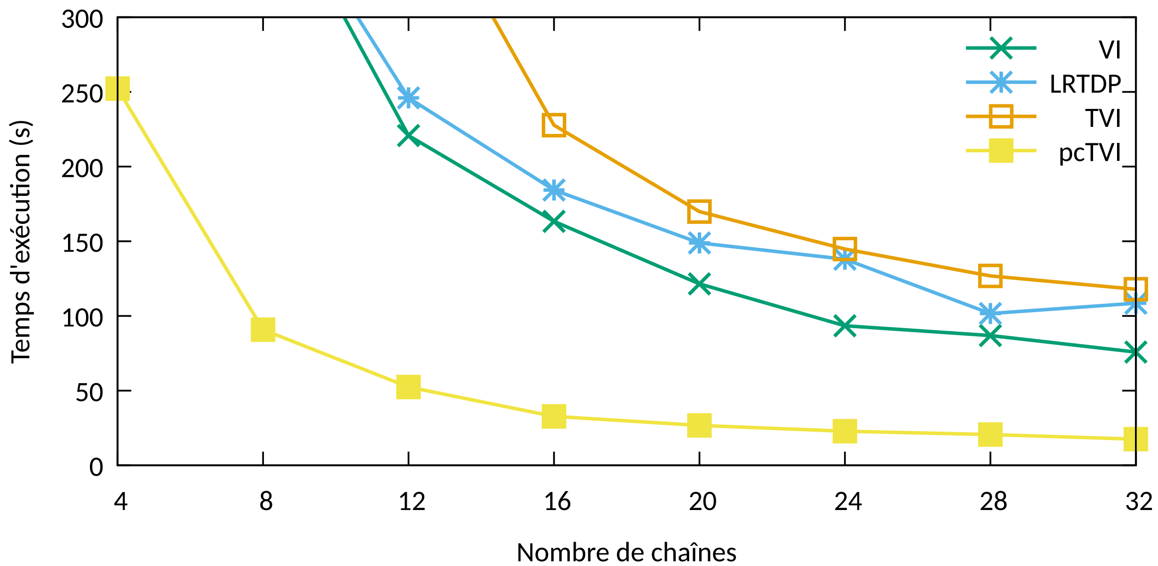


Figure 5.4 – Temps d'exécution moyen (s) pour le domaine *Chained-MDP* avec un nombre de chaînes variables et un nombre fixe d'états (1M).

Un fait peut-être surprenant est que VI est plus rapide que TVI dans les résultats. TVI est connu pour surpasser VI lorsque le domaine de planification contient de nombreuses CFC, ce qui est pourtant le cas dans le domaine *Chained-MDP*. En mesurant quelques métriques de performance à l'aide de l'outil `perf`, nous avons trouvé que TVI avait plus de défauts de cache que VI sur ce domaine. Le surcoût mémoire impliqué par ces défauts de cache explique que VI ait pu surpasser TVI dans les résultats. Le prochain chapitre présente une façon d'éviter une partie de ces défauts de cache. En définitive, ce chapitre a introduit *Chained-MDP*, un nouveau domaine de planification paramétrique, et pcTVI, un algorithme basé sur TVI exploitant le parallélisme de fils d'exécution. L'algorithme pcTVI est particulièrement efficace lorsque le PDM considéré a de nombreuses chaînes indépendantes de CFC — offrant des opportunités de parallélisation accrues — et de grande taille, permettant une réduction du surcoût d'attribution des tâches aux fils d'exécution. Celui-ci a permis d'obtenir un facteur d'accélération de 20, en moyenne, sur un ordinateur de 32 cœurs au total.

CHAPITRE 6
RÉORDONNANCEMENT D'ÉTATS

> *Cache misses are very expensive. A variable can be fetched from the cache in just a few clock cycles, but it can take more than a hundred clock cycles to fetch the variable from RAM memory if it is not in the cache. The cache works most efficiently if pieces of data that are used together are stored near each other in memory.*
>
> — Fog (2023)

Les deux précédents chapitres ont proposé respectivement (1) une représentation en mémoire, CSR-MDP, permettant de regrouper de façon contiguë les données fréquemment utilisées ensemble de sorte à maximiser la quantité d'information utile en mémoire cache, et (2) un algorithme, pcTVI, permettant d'exploiter le parallélisme de fils d'exécution. Or, même si tous les algorithmes évalués bénéficiaient de la représentation CSR-MDP, aucun n'a été proposé pour tenir compte explicitement de cette représentation et en tirer parti au maximum.

Ce chapitre présente deux nouveaux algorithmes, eTVI et eiTVI, qui permettent d'exploiter davantage la hiérarchie de mémoire. L'aspect novateur de ceux-ci repose sur un réordonnancement des états en mémoire suivant les caractéristiques de la représentation CSR-MDP. Ce chapitre présente des contributions de notre article présenté à la *26th European Conference on Artificial Intelligence* (ECAI 2023) qui a eu lieu à Cracovie, en Pologne (Champagne Gareau *et al.*, 2023c). La section 6.1 présente les algorithmes de planification existants qui considérent la mémoire cache ou des stratégies de réordonnancement d'états, les sections 6.2 et 6.3 présentent respectivement les deux nouveaux algorithmes proposés, eTVI et eiTVI, tandis que la section 6.4 présente une analyse de la performance en cache et une comparaison des performances d'exécution des algorithmes VI, LRTDP, LAO*, TVI, eTVI et eiTVI sur trois domaines de planification différents.

6.1 Méthodes existantes

Lors de l'utilisation d'algorithmes basés sur Partitioned VI (algorithme 2.2), tels que P3VI, une méthode de réordonnancement d'états peut être utilisée pour améliorer la propagation des valeurs d'état $V(s)$ à l'intérieur de chacune des partitions (Wingate et Seppi, 2005). Cette méthode associe à chaque état s son

degré entrant $d^{-}(s) = |\{s'|\exists a, T(s,a,s') > 0\}|$, qui correspond au nombre d'états ayant une transition directe pouvant mener à s. Les partitions sont ensuite résolues une à une — possiblement en revenant plusieurs fois à la même partition, étant donné les cycles qu'il peut y avoir entre celles-ci — en utilisant l'algorithme Value Iteration (VI) asynchrone (algorithme 1.2). Cependant, au lieu d'itérer sur les états de chaque partition en utilisant un ordre *round-robin*, l'ordre décroissant des $d^{-}(s)$ est utilisé. Ce faisant, on s'assure de calculer en premier les états dont la valeur influence le plus celle des autres.

Un algorithme explicitement conçu pour améliorer la performance d'un planificateur de PDM en exploitant la mémoire cache a été récemment proposé (Jain et Sahni, 2020). L'algorithme, appelé *Cache-Efficient with Clustering* (CEC), subdivise les composantes fortement connexes (CFC) trouvées par l'algorithme FTVI en groupes d'états d'une taille correspondant à celle de la mémoire cache L3 du CPU. L'étape de FTVI consistant à résoudre les CFC une à une à l'aide de VI est remplacée par une procédure qui résout cycliquement chaque sous-groupe dans la CFC jusqu'à ce que la composante ait intégralement convergée. Les auteurs de CEC indiquent que leur algorithme a permis d'atteindre un facteur d'accélération variant entre 2 et 8, comparativement à FTVI.

D'autres travaux ont considéré la mémoire cache des disques durs lorsque les instances de PDM à résoudre ne peuvent pas être totalement chargées en mémoire principale (Wingate et Seppi, 2004), mais nous n'en discutons pas ici, car ce problème est orthogonal à notre projet de recherche.

6.2 Algorithme eTVI

Puisque les algorithmes présentés dans cette section et dans la suivante — eTVI et eiTVI — sont basés sur TVI, nous commençons par analyser certaines caractéristiques de TVI plus en détail qu'à la section 2.1, en particulier en ce qui concerne (1) les accès mémoire et (2) l'ordre de propagation des états, deux éléments qui sous-tendent les optimisations présentées dans ce chapitre.

Premièrement, rappelons que TVI divise l'espace d'états en CFC et résout celles-ci une par une en ordre topologique inverse. Cet ordre maximise l'utilité des mises à jour de Bellman, car il assure que seulement les valeurs d'état $V(s)$ qui ont convergé à une précision ϵ soient propagées d'une CFC à une autre. Un second avantage de TVI, moins évident à priori, est qu'étant donnée qu'un balayage se fait sur un sous-ensemble des états, la proportion des états considérés à chaque instant qui pourra rentrer en cache sera plus importante. De plus, même lorsqu'une CFC est trop grande pour être stockée entièrement en mémoire cache, le nombre

de défauts de cache sera tout de même réduit comparativement au nombre d'entre eux qui auraient lieu lors d'un balayage complet de l'espace d'états. Finalement, selon la méthode d'implémentation de TVI, un troisième facteur d'amélioration des performances peut également entrer en ligne de compte. En effet, si un PDM ne contient qu'une seule CFC, TVI peut se comporter différemment de VI, car l'algorithme de Kosaraju (ou l'algorithme de Tarjan) découvrira les états de l'unique CFC à l'aide d'une recherche profondeur (DFS) postordre. Or, si le balayage à l'intérieur de la CFC s'effectue en utilisant cet ordre, la propagation des valeurs sera probablement améliorée par rapport à l'ordre implicite qui aurait été utilisé par la variante Gauss-Seidel de l'algorithme VI asynchrone.

Bien que la performance vis-à-vis de la mémoire cache de TVI soit meilleure que celle de VI, cela n'est qu'une heureuse coïncidence. En effet, d'après les auteurs de TVI, cela ne faisait pas parti des éléments qui ont été considérés lors de la conception de l'algorithme. Ces gains sont donc accidentels, et pourraient être améliorés avec des travaux centrés sur cette piste. Une façon d'améliorer davantage les performances en cache est de subdiviser chaque CFC en sous-ensembles dont les informations (transitions, valeurs d'état, etc.) peuvent être entièrement contenues dans le cache L3. Cette stratégie a été utilisée par l'algorithme CEC mentionné à la section 6.1. Une autre stratégie, qui est l'objet de ce chapitre, consiste à réorganiser le contenu dans la ou les structures de données contenant le PDM de sorte que les données des CFC soient stockées de manière contiguë en mémoire, assurant des schémas d'accès facilement prévisibles et maximisant la quantité de contenu utile dans les lignes de cache chargées. Par exemple, si un PDM a deux CFC contenant respectivement les états $(0, 2, 4, 6)$ et les états $(1, 3, 5, 7)$, nous pouvons les réorganiser de sorte que les données des états du PDM soient stockées dans l'ordre $(0, 2, 4, 6, 1, 3, 5, 7)$ au lieu de l'ordre original $(0, 1, 2, 3, 4, 5, 6, 7)$. Cet exemple n'est pas le pire cas. En effet, si chaque état d'une CFC de quatre états occupe 16 octets contigus en mémoire et est stocké sur une ligne de cache différente des trois autres états, alors la résolution de la CFC nécessiterait le chargement de quatre lignes de cache de la mémoire vive à travers tous les niveaux de cache. En rendant les états contigus, un seul chargement de ligne de cache (de 64 octets) sera nécessaire.

L'algorithme eTVI que nous proposons (algorithme 6.1) exploite cette stratégie de réorganisation pour réduire la quantité de transferts en mémoire et ainsi améliorer la vitesse de calcul. Cependant, réorganiser les états en mémoire ne sera pas plus avantageuse si le PDM est stocké en mémoire dans une représentation inefficace pour le cache (par exemple, en utilisant des listes chaînées). Par conséquent, notre algorithme suppose que le PDM est stocké en utilisant la représentation mémoire CSR-MDP mentionnée dans au chapitre 4, ou une autre représentation présentant des propriétés analogues.

Algorithme 6.1 *Cache-Efficient Topological Value Iteration.*

1: **procédure** eTVI(M : PDM (stocké avec la représentation CSR-MDP))
2: ▷ Les CFC sont trouvées en ordre topologique inverse
3: $CFC \leftarrow$ Tarjan(M) ▷ Tableau de tableaux
4: $ID_{CFC} \leftarrow$ Réordonner(M, CFC)
5: **pour** $i \leftarrow 0$ à Taille$(CFC) - 1$ **faire**
6: $debutID \leftarrow ID_{CFC}[i]$
7: $finID \leftarrow ID_{CFC}[i+1]$ ▷ Le dernier est exclut
8: VIPartiel$(M, debutID, finID)$ ▷ Même fonction que pour TVI (algorithme 2.3)
9:
10: **procédure** Réordonner(M, CFC)
11: ▷ Étape 1 : Assigner un nouvel id à chaque état
12: $(\mathcal{S}, \mathcal{C}, \mathcal{A}, \mathcal{N}, \mathcal{P}) \leftarrow M$ ▷ Dépaquetage des tableaux de la représentation CSR-MDP
13: $n \leftarrow$ NumÉtats(M)
14: $k \leftarrow$ Taille(CFC) ▷ Nombre de CFC
15: $ID_{CFC} \leftarrow$ Tableau de capacité $k+1$
16: Insérer$(ID_{CFC}, 0)$
17: $anciensID, nouveauxID \leftarrow$ Tableau de taille n
18: $actuel \leftarrow 0$
19: **pour tout** $C \in CFC$ **faire**
20: **pour tout** $etatID \in C$ **faire**
21: $anciensID[actuel] = etatID$
22: $nouveauxID[etatID] = actuel$
23: $actuel \leftarrow actuel + 1$
24: Insérer$(ID_{CFC}, actuel)$
25:
26: ▷ Étape 2 : Reconstruire la représentation mémoire avec les nouveaux ids
27: ReconstruireCSRMDP$(M, anciensID, nouveauxID)$
28: **retourne** ID_{CFC}
29:
30: **procédure** ReconstruireCSRMDP($M, anciensID, nouveauxID$)
31: $(\mathcal{S}', \mathcal{C}', \mathcal{A}', \mathcal{N}', \mathcal{P}') \leftarrow$ Nouveaux tableaux de la même taille
32: **pour** $s_{id}^{ancien} \leftarrow 0$ à $n-1$ **faire** ▷ itérer sur tous les états
33: **pour tout** $a_{id}^{ancien} \in \mathcal{S}_s$ **faire** ▷ itérer sur toutes les actions de s
34: **pour tout** $e_{id}^{ancien} \in \mathcal{A}_a$ **faire** ▷ itérer sur tous les effets de a
35: mettre à jour $\mathcal{N}'$ et $\mathcal{P}'$
36: mettre à jour $\mathcal{A}'$ et $\mathcal{C}'$
37: mettre à jour $\mathcal{S}'$
38: $M \leftarrow (\mathcal{S}', \mathcal{C}', \mathcal{A}', \mathcal{N}', \mathcal{P}')$

L'algorithme eTVI détermine d'abord les k CFC de manière analogue à TVI. Il attribue ensuite un nouvel identifiant à chaque état du PDM, de sorte qu'il existe des identifiants $i_0 < \cdots < i_k$ (donnés par le tableau ID_{CFC} dans le pseudocode) qui garantissent que tous les états dans la $j^{\text{ème}}$ CFC ont un identifiant i, où $i_j \leq i < i_{j+1}$. Une fois les nouveaux identifiants calculés, eTVI itère sur chaque état, action et effet d'action probabiliste pour reconstruire une représentation mémoire CSR-MDP du PDM en utilisant les nouveaux indices. Les cinq tableaux de la représentation seront alors implicitement divisés en k régions contiguës de sorte que la $i^{\text{ème}}$ région d'un tableau ne contienne que des données correspondant à la $i^{\text{ème}}$ CFC. Il est à noter que nous aurions pu générer un quintuplet de tableaux différent pour chaque CFC, mais nous avons préféré diviser implicitement les mêmes cinq tableaux en différentes régions pour améliorer la localité de la mémoire en éliminant un niveau d'indirection. Lorsque la reconstruction de la représentation mémoire du PDM est terminée, chaque CFC est résolue successivement en ordre topologique inverse (comme le fait TVI) en considérant les régions des cinq tableaux une par une. Les CFC étant stockées de manière contiguë dans chacun des tableaux, le nombre d'octets gaspillés dans chaque ligne de cache chargée est minimal.

6.3 Algorithme eiTVI

Jusqu'à présent, nous avons uniquement considéré l'ordre des états en mémoire par rapport à la CFC dont ils font partie, ce que nous appellerons l'ordre des états *extra-CFC*. Qu'en est-il de l'ordre des états *à l'intérieur* d'une CFC (que nous appelerons l'ordre des états *intra-CFC*) ? Comme mentionné précédemment, TVI (et pour la même raison, eTVI) peut parfois être plus rapide que VI même sur des PDM n'ayant qu'une seule CFC, car il effectue des balayages en utilisant un ordre d'états différent : celui utilisé lors du calcul des CFC (par exemple, l'ordre donné par la recherche profondeur postordre dans l'algorithme de Tarjan). On peut se demander s'il est possible de trouver un meilleur ordre que ce dernier.

Étant donné qu'une CFC contient, par définition, des cycles, l'ordre optimal peut changer après chaque balayage. Les algorithmes qui utilisent un ordre d'états dynamique, comme Prioritized VI (algorithme 2.1), ont un surcoût important, car ils doivent maintenir une file prioritaire. Au lieu de cela, nous proposons un ordre de balayage des états qui est *statique* pour chaque CFC, donné par une recherche en largeur (BFS) inversée où, au départ, la file utilisée par la recherche contient tous les *états frontières vers l'extérieur* de la CFC, que nous définissons dans la définition 6.1.

Définition 6.1. *Soit $M = (S, A, T, C, G)$ un PDM et K une CFC de M. Un **état frontière vers l'extérieur** de K est un état $s \in K$ pour lequel il existe une action $a \in A$ et un état $s' \in S \setminus K$ tels que $T(s, a, s') > 0$.*

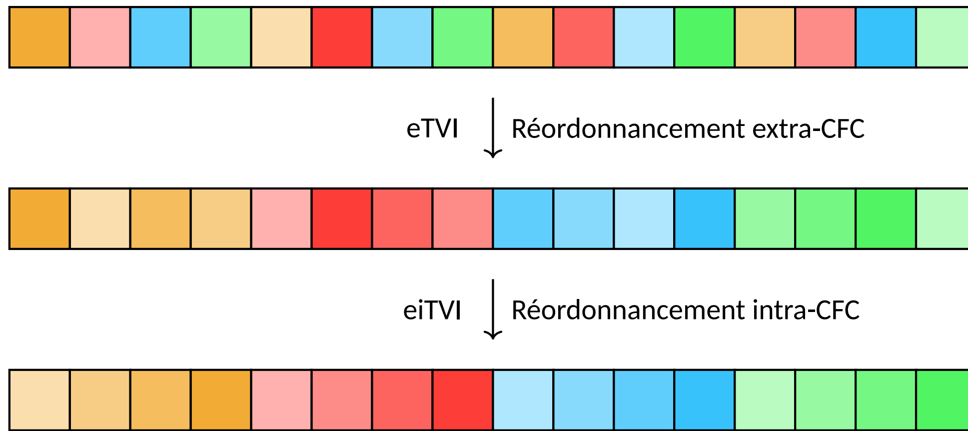


Figure 6.1 – Comparaison visuelle de la différence d'agencement mémoire entre TVI, eTVI et eiTVI. Les couleurs (orange, rouge, bleu, vert) représentent chacune une CFC, et les nuances d'une même couleur représentent les états d'une même CFC.

En faisant une recherche en largeur inversée partant des états frontières vers l'extérieur de la CFC courante, les premières mises à jour de Bellman à l'intérieur de la CFC propageront dans celle-ci les valeurs déjà convergées provenant des CFC voisines. L'ordre garantit alors que les valeurs d'état seront propagées de sorte que chaque mise à jour de Bellman ait de nouvelles valeurs d'état voisins à considérer (aucune mise à jour de Bellman n'est inutile). Comme la CFC contenant l'état but n'a pas d'état frontière vers l'extérieur, la recherche largeur inversée est commencée dans ce cas en partant de l'état but.

L'algorithme qui combine eTVI avec cet ordre intra-CFC est appelé eiTVI. Celui-ci est implémenté en remplaçant les lignes 20–23 dans l'algorithme 6.1 par une détection des états frontières vers l'extérieur et l'exécution de la recherche en largeur inversée partant de ces états. Les noms eTVI et eiTVI signifient respectivement *extra-TVI* et *extra-intra-TVI*, car le premier réorganise les états selon un ordre extra-CFC, tandis que le dernier réorganise les états selon à la fois les ordres extra-CFC et intra-CFC. La figure 6.1 illustre visuellement la différence d'agencement en mémoire entre TVI, eTVI et eiTVI, tandis que la figure 6.2 illustre la différence de l'ordre de balayage interne dans une CFC entre VI, TVI et eiTVI.

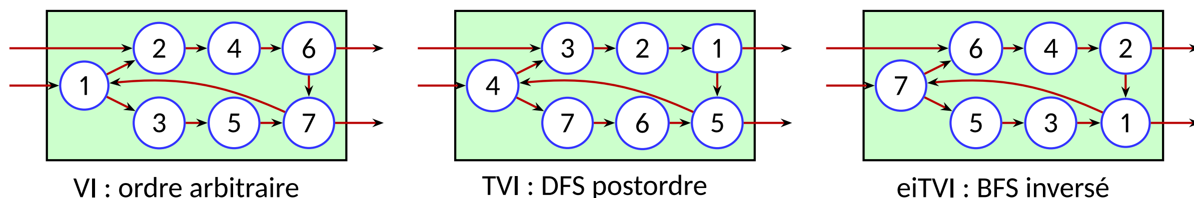


Figure 6.2 – Illustration de divers ordres de balayage à l'intérieur d'une CFC. Les deux états les plus à droite sont les états frontières vers l'extérieur de cette CFC.

6.4 Évaluation

Dans cette section, nous comparons la performance des algorithmes suivants :

- *Value Iteration* (VI) asynchrone, variante Gauss-Seidel (algorithme 1.2) ;
- LRTDP (modification de RTDP décrite à la section 2.2) ;
- ILAO* (version améliorée de l'algorithme 2.4) ;
- TVI (algorithme 2.3) ;
- eTVI (algorithme 6.1) ;
- eiTVI (décrit à la section 6.3).

Dans le cas des planificateurs utilisant une heuristique, LRTDP et ILAO*, l'heuristique utilisée est $h_{\min}$, décrite à la section 4.3. L'ordinateur et le compilateur ayant été utilisés pour l'évaluation sont les mêmes que pour l'évaluation de CSR-MDP et sont décrits à la section 4.3.

Bien que l'algorithme FTVI (mentionné à la section 2.1) offre de meilleures performances que TVI sur certains domaines, nous avons choisi de ne pas l'inclure dans notre évaluation. En effet, l'étape initiale ajoutée par FTVI comparativement à TVI — la découverte d'actions sous-optimales qui peuvent être élaguées de l'espace d'états pour permettre de trouver un plus grand nombre de CFC — dépend d'une recherche heuristique. Au mieux de notre connaissance, personne n'a encore étudié la performance en cache de tels algorithmes. De plus, rien n'empêche d'ajouter cette étape à eTVI ou à eiTVI. Le gain de performance permis par l'étape additionnelle de FTVI est donc orthogonal à ce que nous souhaitons améliorer dans ce chapitre, et son inclusion dans notre évaluation serait peu pertinente.

Nous avons aussi choisi de ne pas inclure l'algorithme CEC mentionné à la section 6.1 dans notre évaluation. En effet, nous n'avons malheureusement pas pu reproduire les résultats rapportés dans l'article de recherche le proposant (Jain et Sahni, 2020). Lors de la mise en œuvre de ces algorithmes, nous avons constaté que la surcharge calculatoire nécessaire pour trouver les « états externes » et le sous-partitionnement des CFC s'adaptant au cache L3 éclipsait l'amélioration obtenue due à la réduction du nombre de défauts de cache. Nous soupçonnons que la différence entre ce que nous avons observé et ce que les auteurs de l'algorithme CEC prétendent est due à la mauvaise performance de leur représentation mémoire des PDM, qui biaise leurs résultats. En effet, dans leur évaluation, tous les algorithmes comparés (y compris leur référence de base, FTVI) sont mis en œuvre avec des PDM stockés en utilisant une liste chaînée de listes chaînées, ce qui conduit à de nombreuses opportunités d'amélioration du cache qui ne peuvent pas être reproduites lorsque le PDM est stocké de manière efficace pour les accès mémoires, comme c'est le cas avec la repré-

sentation CSR-MDP. Par conséquent, l'implémentation utilisée par les auteurs de CEC semble surestimer l'amélioration potentielle du cache de leurs méthodes (par exemple, notre implémentation naïve de VI avec la représentation CSR-MDP était plus rapide que leur implémentation C++ de CEC).

Puisque TVI et, par extension, eTVI et eiTVI également, sont conçus pour résoudre des PDM-PCCS énumératifs — des PDM où les états peuvent être listés explicitement, en comparaison aux problèmes décrits sous une forme factorisée (par exemple, sous forme PPDDL ou RDDL) —, nous évaluons la performance des algorithmes sur trois domaines énumératifs. Les domaines choisis sont les mêmes qu'utilisés lors de l'évaluation de CSR-MDP présentée à la section 4.3, c'est-à-dire les domaines *Layered*, *SAP* et *Wetfloor*. L'évaluation a été faite sur deux autres domaines, *Double-Armed Pendulum* (DAP) et *Mountain Car* (MCar), mais les résultats obtenus sur ces deux domaines sont similaires à ceux obtenus pour les trois domaines mentionnés plus haut, et nous ne les incluons donc pas pour ne pas obfusquer les éléments importants de notre évaluation avec des domaines qui n'apportent pas de nouveaux éléments d'analyse.

Pour chacun des trois domaines inclus dans notre évaluation, nous comparons, pour les six algorithmes évalués, le temps d'exécution de chacun d'eux jusqu'à convergence de la fonction de valeur V à une précision $\epsilon = 10^{-6}$. Chaque taille d'instance de domaine a été résolue 15 fois et chaque algorithme est évalué sur les mêmes 15 instances générées aléatoirement.

Les figures 6.3, 6.4, 6.5, et 6.6 présentent les temps moyens de résolution obtenus par les six algorithmes sur les instances des trois domaines évalués. Sur chacune des figures, l'intervalle de confiance $95\ \%$ est présenté pour chaque algorithme. Puisque le générateur du domaine *SAP* est déterministe, les 15 instances générées pour chacune des tailles testées étaient les mêmes. La faible variation de temps (à peine visible dans la figure 6.5 de *SAP*) est donc due à des biais environnementaux (ex. : tâches d'arrière-plan qui s'exécutent sur l'ordinateur) et non pas à une différence entre les instances générées (contrairement à la variation observée pour les deux autres domaines).

La table 6.1 présente les temps d'exécutions présents dans les figures mentionnées plus haut, de même que le nombre de mises à jour de Bellman qui ont été nécessaires avant que chacun des algorithmes comparés ne trouve une fonction de valeur V qui soit ϵ-optimale [1].

---

1. Dû à des contraintes de mises en page (par exemple, la taille des marges), les données des intervalles de confiance ne sont pas présentes dans les tables du chapitre, mais le sont dans une version plus détaillée des tables, disponible en ligne : `https://www.jaelgareau.com/fr/publication/gareau-ecai23/supp.pdf`

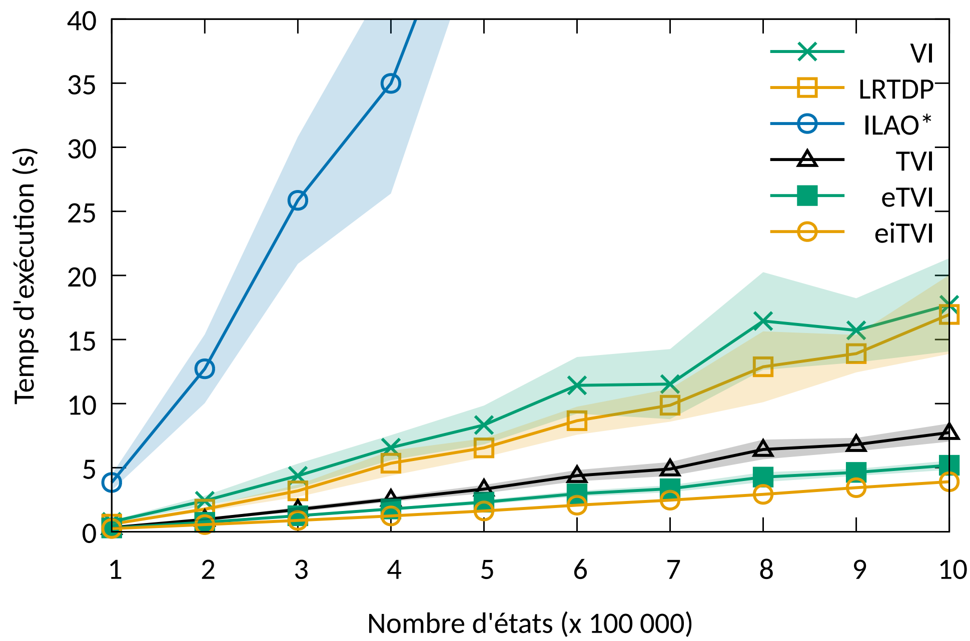


Figure 6.3 – Temps d'exécution (en secondes) pour le domaine *Layered* en fixant le nombre de couches (10) et en faisant varier le nombre d'états.

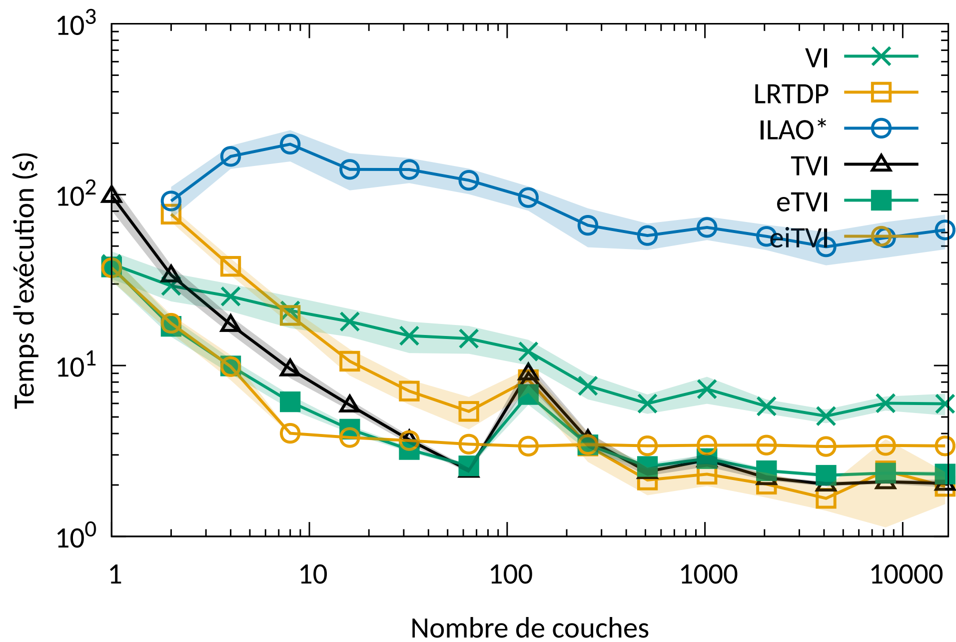


Figure 6.4 – Temps d'exécution (en secondes) pour le domaine *Layered* en fixant le nombre d'états (1M) et en faisant varier le nombre de couches.

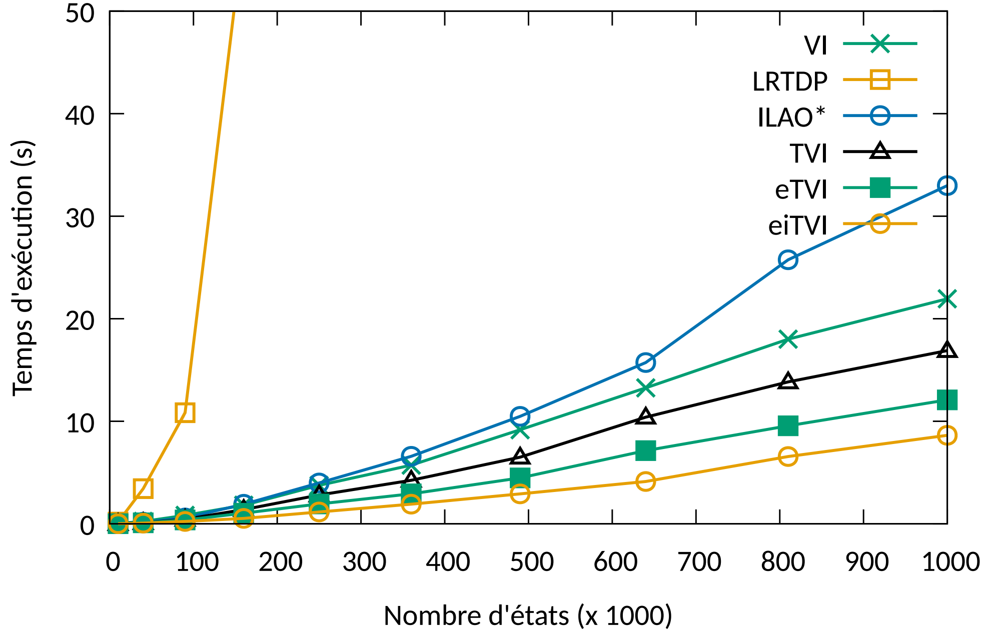


Figure 6.5 – Temps d’exécution (en secondes) pour le domaine *SAP* en faisant varier le nombre d’états.

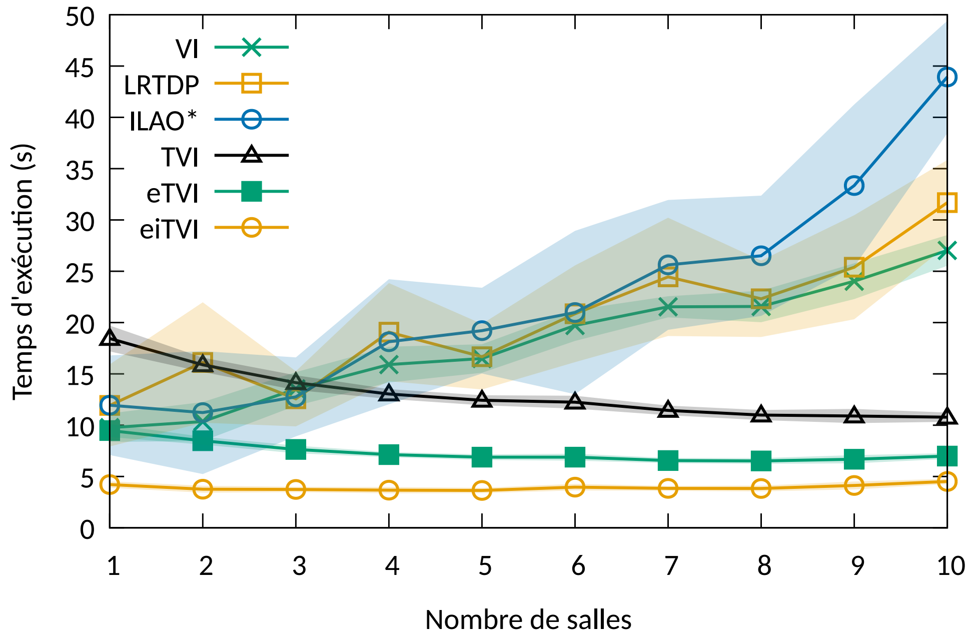


Figure 6.6 – Temps d’exécution (en secondes) pour le domaine *Wetfloor* en fixant le nombre d’états (500k) et en faisant varier le nombre de salles.

Caract. instances				VI		LRTDP		ILAO*		TVI		eTVI		eiTVI	
D	$\|S\|$ (k)	$\|K\|$	$\|k_{\max}\|$ (k)	B (M)	T_{tot} (s)	B (M)	T_{tot} (s)	B (M)	T_{tot} (s)	B (M)	T_{tot} (s)	B (M)	T_{tot} (s)	B (M)	T_{tot} (s)
Layered	100	10	10	5.46	0.777	1.39	0.595	6.46	3.85	1.43	0.328	1.43	0.298	0.552	**0.253**
	200	10	20	15.7	2.42	3.41	1.76	19	12.7	3.53	0.964	3.53	0.74	1.14	**0.561**
	300	10	30	27.1	4.37	5.61	3.19	36.4	25.9	5.88	1.74	5.88	1.25	1.73	**0.886**
	400	10	40	39.6	6.57	8.78	5.35	47	35	8.18	2.54	8.18	1.78	2.32	**1.24**
	500	10	50	49.8	8.33	10.3	6.54	71	53.7	10.4	3.33	10.4	2.31	2.89	**1.62**
	600	10	60	67.3	11.4	13.4	8.66	106	82	13.3	4.38	13.3	2.97	3.52	**2.08**
	700	10	70	66.7	11.5	14.7	9.87	133	106	14.1	4.89	14.1	3.36	4.09	**2.47**
	800	10	80	93.7	16.4	18.6	12.9	–	–	18.4	6.42	18.4	4.27	4.68	**2.92**
	900	10	90	89	15.7	19.8	13.9	–	–	18.6	6.8	18.6	4.63	5.29	**3.44**
	1000	10	100	96.7	17.7	23.7	17	–	–	20.4	7.74	20.4	5.19	5.89	**3.9**
Layered	1000	2^0	1000	201	39.3	–	–	–	–	197	98.8	197	37.7	191	**37.3**
	1000	2^1	500	143	29.3	88	76.9	87.6	91.8	83.2	33.9	83.2	**17**	82	17.7
	1000	2^2	250	132	25.4	48.4	38.1	179	167	46.2	17.4	46.2	9.94	39.3	**9.86**
	1000	2^3	125	112	21	27	19.6	221	197	25.8	9.54	25.8	6.13	6.16	**4.01**
	1000	2^4	62.5	100	18.1	15.5	10.6	178	141	15.9	5.87	15.9	4.24	5.5	**3.79**
	1000	2^5	31.3	84.6	14.9	10.7	7.08	185	140	9.52	3.67	9.52	**3.22**	4.17	3.64
	1000	2^6	15.63	82.2	14.4	7.97	5.39	165	121	5.06	**2.44**	5.06	2.59	2.3	3.47
	1000	2^7	7.81	69.7	12.1	15.5	8.26	130	96.1	31.1	9.04	31.1	6.77	2.81	**3.37**
	1000	2^8	3.91	43.9	7.61	6.08	**3.38**	91.4	66.2	10	3.69	10	3.43	1.44	3.45
	1000	2^9	1.96	34.7	5.99	3.43	**2.14**	79.8	57.6	4.33	2.39	4.33	2.57	0.782	3.39
	1000	2^{10}	0.98	41.8	7.28	3.65	**2.31**	88.6	64.3	6.13	2.8	6.13	2.85	0.878	3.42
	1000	2^{11}	0.49	33.3	5.78	3.02	**2.02**	78.5	57	3.37	2.21	3.37	2.42	0.602	3.42
	1000	2^{12}	0.25	29.3	5.06	2.43	**1.67**	68.9	49.5	2.48	2.02	2.48	2.28	0.54	3.36
	1000	2^{13}	0.12	34.7	6.02	3.51	2.42	77.6	56	2.77	**2.09**	2.77	2.35	0.474	3.41
	1000	2^{14}	0.06	34.6	5.97	2.77	**1.96**	85.6	62.1	2.57	2.04	2.57	2.32	0.447	3.39
SAP	10	1	10	3.12	0.032	4.13	0.165	0.484	0.016	1.04	0.012	1.04	0.011	0.81	**0.009**
	40	1	40	22.2	0.232	86.6	3.43	4.42	0.159	9.08	0.105	9.08	0.098	5.6	**0.065**
	90	1	90	71.6	0.822	258	10.8	12.8	0.57	32.2	0.438	32.2	0.375	18.6	**0.23**
	160	1	160	155	1.81	1220	57.8	35.8	1.91	88.2	1.39	88.2	1.04	42.7	**0.526**
	250	1	250	314	3.75	3820	207	66.8	3.98	165	2.81	165	1.94	95.2	**1.15**
	360	1	360	482	5.72	–	–	107	6.58	250	4.24	250	2.92	160	**1.92**
	490	1	490	764	9.17	–	–	166	10.5	377	6.5	377	4.48	241	**2.92**
	640	1	640	1110	13.3	–	–	241	15.7	600	10.4	600	7.14	339	**4.12**
	810	1	810	1510	18	–	–	380	25.8	813	13.8	813	9.56	549	**6.56**
	1000	1	1000	1850	21.9	–	–	467	33	1020	16.9	1020	12.1	718	**8.63**
Wetfloor	500	1	500	433	9.76	19.1	11.9	73.4	11.9	413	18.4	413	9.48	180	**4.22**
	500	2	250	459	10.4	90	16.1	72.3	11.2	372	15.9	372	8.49	161	**3.76**
	500	3	166	601	13.5	13.3	12.6	83.2	12.8	336	14.1	336	7.64	160	**3.74**
	500	4	125	706	15.9	43.3	19.1	117	18.1	314	13	314	7.13	157	**3.67**
	500	5	100	732	16.5	18.4	16.7	125	19.2	303	12.4	303	6.88	157	**3.64**
	500	6	83.5	875	19.7	6.2	20.9	135	21	304	12.2	304	6.88	171	**3.96**
	500	7	71.3	957	21.5	24.2	24.5	169	25.6	289	11.4	289	6.55	166	**3.84**
	500	8	62.5	959	21.6	16.4	22.3	174	26.5	289	11	289	6.52	166	**3.83**
	500	9	55.7	1070	24	4.22	25.4	220	33.4	299	10.9	299	6.68	181	**4.13**
	500	10	50.2	1200	27	6.99	31.7	289	43.9	316	10.8	316	6.99	201	**4.52**

Table 6.1 – Temps d'exécution moyen (s) et nombre de mises à jour de Bellman (en millions) pour tous les algorithmes et sur chacun des domaines évalués. Le temps (T_{tot}) de l'algorithme le plus rapide pour chaque instance est en gras. Le symbole "-" indique que l'algorithme n'a pas résolu l'instance en moins de 5 minutes.

La table 6.2 rapporte quant-à-elle les temps d'exécutions de chacune des principales étapes de TVI, eTVI et eiTVI, plutôt que seulement les temps totaux, obtenus sur les différentes instances des différents domaines considérées. Les quatre premières colonnes rendent compte des caractéristiques des instances de test : le nom du domaine (D), le nombre d'états dans chacune des 15 instances générées ($|S|$), le nombre de CFC ($|K|$) (nous ne comptons pas les CFC contenant un seul état, telle que celle de l'état but) et la taille de la plus grande CFC ($|k_{max}|$). La colonne suivante rend compte du temps d'exécution de l'algorithme de Tarjan utilisé pour trouver les composantes. Enfin, nous rapportons pour chacun des trois algorithmes le temps nécessaire pour calculer la réorganisation des états et pour construire la nouvelle représentation mémoire CSR-MDP (T_r), le temps nécessaire pour résoudre séquentiellement chaque CFC en ordre topologique inverse (T_s), le temps total d'exécution de l'algorithme (T_{tot}) et, enfin, le nombre de mises à jour de Bellman (B) nécessaires pour atteindre la ϵ-convergence de V.

Les facteurs d'accélération obtenus en moyenne pour TVI, eTVI et eiTVI par rapport à VI sur l'ensemble des tailles d'instances pour chaque domaine sont présentés dans la table 6.3.

Pour confirmer que les gains de vitesse observés proviennent bien d'une meilleure exploitation de la hiérarchie de mémoire, nous évaluons la performance en mémoire cache des algorithmes de deux façons : l'une directe, l'autre indirecte. L'évaluation directe a été faite en mesurant le nombre d'accès en mémoire cache L3 et le nombre de défauts de cache L3 pour le domaine *Layered*[2]. Le résultat de cette approche directe est présenté à la figure 6.4. On peut y voir qu'à la fois le nombre d'accès au cache et le nombre de défauts de cache diminuent lorsqu'on passe de l'algorithme TVI à l'algorithme eTVI, puis lorsqu'on passe de l'algorithme eTVI à l'algorithme eiTVI. En revanche, le pourcentage de défauts de cache par rapport au nombre d'accès décroit lorsqu'on passe de TVI à eTVI, mais croit lorsqu'on passe de eTVI à eiTVI. Cela s'explique par le fait que la recherche en largeur inversée nécessite des accès mémoire supplémentaires, causant des défauts de cache additionnels. Le meilleur ordre de balayage permis suite à cette recherche en largeur permet cependant de diminuer le nombre d'accès mémoire qui suit.

Cette évaluation directe est utile, mais les métriques mesurées ont comme désavantage de potentiellement dépendre de plusieurs facteurs, comme la prélecture matérielle du cache (*prefetching*) ou les accès en cache L1 et L2, qui peuvent causer un biais sur les métriques du cache L3 mesurées. Par conséquent, nous analysons

2. Les métriques de cache ont été mesurées à l'aide du programme `perforator` (https://github.com/zyedidia/perforator), qui est basé sur la commande Linux `perf` mais qui permet d'évaluer les métriques pour des régions spécifiques plutôt que pour l'ensemble du programme.

D	Caract. instances $\lvert S \rvert$ (k)	$\lvert K \rvert$	$\lvert k_{\max} \rvert$ (k)	Tarjan	TVI T_s (s)	T_{tot} (s)	B (M)	eTVI T_r (s)	T_s (s)	T_{tot} (s)	B (M)	eiTVI T_r (s)	T_s (s)	T_{tot} (s)	B (M)
Layered	100	10	10	0.049	0.276	0.328	1.43	0.052	0.194	0.298	1.43	0.125	0.077	**0.253**	0.552
	200	10	20	0.115	0.844	0.964	3.53	0.111	0.509	0.74	3.53	0.266	0.172	**0.561**	1.14
	300	10	30	0.196	1.53	1.74	5.88	0.176	0.87	1.25	5.88	0.410	0.273	**0.886**	1.73
	400	10	40	0.288	2.24	2.54	8.18	0.246	1.23	1.78	8.18	0.559	0.376	**1.24**	2.32
	500	10	50	0.391	2.92	3.33	10.4	0.320	1.59	2.31	10.4	0.739	0.479	**1.62**	2.89
	600	10	60	0.509	3.85	4.38	13.3	0.399	2.04	2.97	13.3	0.945	0.592	**2.08**	3.52
	700	10	70	0.632	4.25	4.89	14.1	0.483	2.22	3.36	14.1	1.129	0.689	**2.47**	4.09
	800	10	80	0.760	5.64	6.42	18.4	0.562	2.92	4.27	18.4	1.335	0.804	**2.92**	4.68
	900	10	90	0.887	5.89	6.8	18.6	0.652	3.06	4.63	18.6	1.602	0.923	**3.44**	5.29
	1000	10	100	1.026	6.69	7.74	20.4	0.736	3.41	5.19	20.4	1.811	1.05	**3.9**	5.89
Layered	1000	2^0	1000	1.309	97.5	98.8	197	0.919	35.5	37.7	197	1.537	34.5	**37.3**	191
	1000	2^1	500	1.332	32.5	33.9	83.2	0.853	14.7	**17**	83.2	1.759	14.6	17.7	82
	1000	2^2	250	1.201	16.1	17.4	46.2	0.786	7.93	9.94	46.2	1.819	6.81	**9.86**	39.3
	1000	2^3	125	1.062	8.46	9.54	25.8	0.750	4.3	6.13	25.8	1.824	1.11	**4.01**	6.16
	1000	2^4	62.5	0.955	4.89	5.87	15.9	0.716	2.54	4.24	15.9	1.835	0.965	**3.79**	5.5
	1000	2^5	31.3	0.899	2.7	3.67	9.52	0.691	1.56	**3.22**	9.52	1.906	0.756	3.64	4.17
	1000	2^6	15.63	0.836	1.4	**2.44**	5.06	0.687	0.875	2.59	5.06	1.963	0.485	3.47	2.3
	1000	2^7	7.81	0.773	8.04	9.04	31.1	0.675	5.11	6.77	31.1	1.822	0.58	**3.37**	2.81
	1000	2^8	3.91	0.831	2.57	3.69	10	0.670	1.67	**3.43**	10	1.955	0.37	3.45	1.44
	1000	2^9	1.96	0.821	1.25	**2.39**	4.33	0.680	0.763	2.57	4.33	2.015	0.254	3.39	0.782
	1000	2^{10}	0.98	0.837	1.64	**2.8**	6.13	0.691	1.02	2.85	6.13	2.001	0.286	3.42	0.878
	1000	2^{11}	0.49	0.823	1.05	**2.21**	3.37	0.684	0.606	2.42	3.37	2.048	0.233	3.42	0.602
	1000	2^{12}	0.25	0.818	0.867	**2.02**	2.48	0.682	0.476	2.28	2.48	2.025	0.211	3.36	0.54
	1000	2^{13}	0.12	0.826	0.92	**2.09**	2.77	0.686	0.529	2.35	2.77	2.045	0.207	3.41	0.474
	1000	2^{14}	0.06	0.824	0.879	**2.04**	2.57	0.684	0.497	2.32	2.57	2.037	0.204	3.39	0.447
SAP	10	1	10	0.001	0.011	0.012	1.04	0.001	0.01	0.011	1.04	0.001	0.008	**0.009**	0.81
	40	1	40	0.002	0.102	0.105	9.08	0.002	0.092	0.098	9.08	0.005	0.057	**0.065**	5.6
	90	1	90	0.004	0.431	0.438	32.2	0.005	0.363	0.375	32.2	0.012	0.211	**0.23**	18.6
	160	1	160	0.008	1.37	1.39	88.2	0.009	1.01	1.04	88.2	0.022	0.491	**0.526**	42.7
	250	1	250	0.013	2.79	2.81	165	0.015	1.9	1.94	165	0.036	1.1	**1.15**	95.2
	360	1	360	0.018	4.21	4.24	250	0.021	2.87	2.92	250	0.054	1.83	**1.92**	160
	490	1	490	0.027	6.46	6.5	377	0.029	4.41	4.48	377	0.075	2.8	**2.92**	241
	640	1	640	0.037	10.3	10.4	600	0.038	7.04	7.14	600	0.100	3.97	**4.12**	339
	810	1	810	0.045	13.8	13.8	813	0.050	9.44	9.56	813	0.130	6.37	**6.56**	549
	1000	1	1000	0.055	16.8	16.9	1020	0.061	11.9	12.1	1020	0.162	8.39	**8.63**	718
Wetfloor	500	1	500	0.053	18.4	18.4	413	0.054	9.36	9.48	413	0.104	4.04	**4.22**	180
	500	2	250	0.051	15.8	15.9	372	0.053	8.37	8.49	372	0.097	3.6	**3.76**	161
	500	3	166	0.050	14.1	14.1	336	0.053	7.52	7.64	336	0.095	3.58	**3.74**	160
	500	4	125	0.049	13	13	314	0.053	7.02	7.13	314	0.097	3.51	**3.67**	157
	500	5	100	0.048	12.4	12.4	303	0.053	6.77	6.88	303	0.093	3.49	**3.64**	157
	500	6	83.5	0.048	12.2	12.2	304	0.053	6.77	6.88	304	0.094	3.81	**3.96**	171
	500	7	71.3	0.046	11.4	11.4	289	0.052	6.44	6.55	289	0.093	3.69	**3.84**	166
	500	8	62.5	0.046	10.9	11	289	0.053	6.4	6.52	289	0.092	3.68	**3.83**	166
	500	9	55.7	0.046	10.8	10.9	299	0.052	6.57	6.68	299	0.092	3.97	**4.13**	181
	500	10	50.2	0.046	10.7	10.8	316	0.052	6.88	6.99	316	0.093	4.36	**4.52**	201

Table 6.2 – Temps d'exécution moyen (s) des différentes étapes des algorithmes TVI, eTVI et eiTVI, et nombre de mises à jour de Bellman (en millions) pour chacun des algorithmes et sur tous les domaines évalués. Le temps total (T_{tot}) de l'algorithme le plus rapide pour chaque instance de domaine est en gras.

Domaine	TVI vs VI	eTVI vs TVI	eiTVI vs eTVI	eiTVI vs TVI
Layered (var. états)	2.50	1.43	1.40	2.00
Layered (var. couches)	1.80	1.45	0.98	1.42
SAP	1.40	1.37	1.74	2.39
Wetfloor	1.38	1.78	1.86	3.31
Moyenne	1.63	1.60	1.31	2.10

Table 6.3 – Facteurs d'accélération moyens obtenus entre VI, TVI, eTVI et eiTVI.

Algorithme	Accès au cache	Défauts de cache	Accès / Défauts
TVI	2.87 G	0.860 G	29.96 %
eTVI	2.39 G	0.413 G	17.28 %
eiTVI	1.59 G	0.328 G	20.62 %

Table 6.4 – Métriques d'accès et de défauts de cache obtenus sur un CPU Intel Core i5-7600k (6 MiO de cache L3) sur le domaine *Layered* (instance avec 1M d'états et 10 couches).

aussi indirectement la performance des accès mémoire en mesurant le nombre de mises à jour de Bellman, qui permettent à la fois de faire des observations sur les accès mémoire, mais également sur la qualité de la propagation des valeurs dans l'espace d'états. Par exemple, si on compare les résultats de TVI et eTVI en regardant les colonnes (B) de la table 6.2, on peut voir qu'elles sont identiques, ce qui est normal, puisque le seul changement entre TVI et eTVI est l'ordre de stockage des états en mémoire (l'ordre de balayage reste identique, et le nombre de mises à jour de Bellman nécessaire pour converger reste donc identique également). Puisque le nombre de mises à jour de Bellman est identique, mais que le temps d'exécution de eTVI est plus petit, on peut en déduire indirectement que le réordonnancement a mené à une meilleure performance en cache. Puisque le nombre de mises à jour de Bellman a diminué entre eTVI et eiTVI, on peut en déduire que le nouvel ordre de balayage améliore la propagation des valeurs d'état, en plus d'améliorer la performance en cache.

Dans le domaine *Layered*, nous pouvons voir que les algorithmes TVI, eTVI et eiTVI ont largement surpassé VI, LRTDP et ILAO*. De plus, nous pouvons observer que l'avantage de eTVI et eiTVI sur TVI semble augmenter à mesure que le nombre d'états augmente. En faisant varier le nombre de couches, nous pouvons voir que eTVI et eiTVI sont plus rapides que les autres algorithmes jusqu'à un certain point (dans notre cas, 128 couches), après quoi le nombre de couches devient si élevé que le nombre d'états par couche devient suffisamment petit pour tenir en cache. Lorsque cela se produit, des défauts de cache ne se produiront que lors du premier balayage dans la CFC. Le réordonnancement des états en mémoire minimisera ce petit nombre

de défauts de cache, mais, dans cette situation (petites CFC), TVI est déjà très rapide, et l'augmentation de performance due au plus petit nombre de défauts de cache est négligeable par rapport au coût du réordonnancement (ce coût est d'environ $1\,\%$ du temps total d'exécution lorsque les CFC sont grandes, mais peut être jusqu'à $30\,\%$ du temps total lorsque la résolution des CFC est déjà extrêmement rapide comme c'est le cas lorsqu'elles sont petites).

Sur le domaine *SAP*, la différence entre eiTVI et eTVI est similaire à la différence entre eTVI et TVI, qui est également similaire à la différence entre TVI et VI. Ce constat confirme que eTVI et eiTVI peuvent fournir une accélération importante même pour les domaines ne contenant qu'une seule CFC. Dans le cas de eTVI, l'accélération dans les domaines à CFC unique, tels que *SAP*, est due aux différences d'organisation de la mémoire. En fait, même si TVI et eTVI utilisent le même ordre de considération des états lors d'un balayage, l'ordre dans lequel les états sont stockés en mémoire lors de l'utilisation de TVI ne correspond pas nécessairement à l'ordre dans lequel les états sont considérés par les mises à jour de Bellman successives. En revanche, eTVI reconstruit le PDM en mémoire de manière à ce que ces deux ordres correspondent, ce qui améliore les performances du cache. Dans le cas de eiTVI, l'accélération dans les domaines à CFC unique est due au fait que eiTVI réorganise les états dans l'ordre donné par la recherche en largeur inversée. Par conséquent, l'ordre de propagation des valeurs dans l'espace d'états est également amélioré.

Sur le domaine *Wetfloor*, nous pouvons voir sans surprise que les performances de TVI et eTVI augmentent à mesure que le nombre de salles augmente. Plus surprenant est le fait que la performance de eiTVI semble assez constante même lorsque le nombre de salles change. Ceci s'explique par le fait que le flux d'information avec eiTVI est initialement bien meilleur et donc, l'ajout de nouvelles CFC conduit à des opportunités minimales d'amélioration supplémentaire. Par exemple, en comparant la colonne (B) de eiTVI et eTVI, nous voyons que eiTVI a besoin de moins de la moitié des mises à jour de Bellman avant la ϵ-convergence. En comparaison, la performance de VI diminue à mesure que le nombre de salles augmente, puisqu'une augmentation du nombre de salles fait en sorte que la plupart des mises à jour de Bellman effectuées par VI sont inutiles (elles propagent des valeurs d'état qui n'ont pas encore convergé). Dans le cas de LRTDP et ILAO*, la faible performance lorsque le nombre de salles augmente peut s'expliquer par le fait qu'un plus grand nombre de salles correspond à une plus grande profondeur de recherche de l'état initial à l'état but dans le domaine *Wetfloor*, rendant la fonction heuristique moins informative.

Nous avons également évalué les performances des algorithmes comparés sur un processeur Intel Core i5-

13500 (qui a 24 MiO de cache L3) pour évaluer l'impact d'une architecture de cache plus moderne. Pour les domaines *SAP* et *Layered*, chacun des algorithmes évalués était environ deux à trois fois plus rapide sur le CPU i5-13500, mais l'écart entre les algorithmes était presque le même. Pour le domaine *Wetfloor*, l'écart entre TVI et eTVI était environ $50\,\%$ plus petit sur le CPU i5-13500 que sur le CPU i5-7600k, mais l'écart entre eTVI et eiTVI était similaire.

Dans l'ensemble, les algorithmes eTVI et eiTVI proposés surpassent clairement leurs homologues VI, TVI, ILAO* et LRTDP sur presque chaque instance de PDM. Ils sont particulièrement efficaces lorsqu'un domaine de planification a de nombreuses CFC de grandes tailles, mais peuvent également surpasser les autres algorithmes lorsqu'il n'y a qu'une seule CFC (comme c'est le cas pour le domaine *SAP*). Le seul cas où TVI est plus rapide que eTVI et eiTVI survient lorsque le nombre de CFC est grand, mais chacune d'elle est petite.

Dans ce chapitre ainsi que dans le chapitre 4, l'évaluation empirique a été réalisée sur les domaines de planification Layered, SAP et Wetfloor. Ces domaines permettent d'illustrer plusieurs situations variées, mais ne couvrent évidemment pas l'ensemble des propriétés topologiques que l'on pourrait observer dans d'autres domaines de planification. Le prochain chapitre propose un algorithme de génération de domaines de planification synthétiques, offrant ainsi une plus grande diversité de propriétés topologiques que celles présentes dans les environnements réels accessibles publiquement. Celui-ci facilitera les évaluations empiriques futures en élargissant le champ des scénarios testables.

CHAPITRE 7
GÉNÉRATION SYNTHÉTIQUE DE PDM

> *[M]ore theory is needed to guide the development and selection of such enhancements. The most useful would be problem features and optimality definitions that would indicate which metric, reordering method and partitionning scheme are maximally effective, and which would guide the development of new enhancements. These may include distributional properties of the reward functions, distributional properties of transition matrices, strongly/weakly connected component analyses, etc.*
>
> — Wingate et Seppi (2005)

Tel qu'abordé aux chapitres précédents, plusieurs nouveaux algorithmes de planification pour les PDM ont été proposés durant les dernières décennies, chacun ayant ses forces et ses faiblesses. En général, lorsque de nouveaux algorithmes de planification sont proposés, ils sont évalués sur un petit nombre de domaines soigneusement conçus pour démontrer leur efficacité. Or, l'absence d'un nombre varié de domaines d'évaluation standardisés rend difficile la comparaison ou la prédiction de la performance de ces algorithmes dans différents contextes. Par exemple, certains algorithmes, tels que TVI, sont particulièrement efficaces lorsqu'un PDM contient un grand nombre de CFC, tandis que d'autres, tels que LRTDP, le sont plus particulièrement lorsqu'un PDM contient un grand nombre d'états buts atteignables suite à un nombre relativement petit d'actions (Dai *et al.*, 2011). Cependant, puisqu'il n'existe pas de domaines standardisés qui contiennent à la fois un grand nombre de CFC et un grand nombre d'états buts, il est difficile de savoir à priori lequel de ces algorithmes est le plus efficace pour résoudre des PDM ayant simultanément ces deux propriétés.

En planification classique, il y a eu un effort de standardisation des domaines d'évaluation (Haslum, 2024). Or, lorsqu'on tombe dans des branches plus pointues de la planification, notamment pour les PDM statiques, il y a moins de domaines établis. Les domaines qui se rapprochent le plus de ce qu'on pourrait considérer comme des domaines standardisés de planification probabiliste sont ceux utilisés dans la *International Planning Competition* (IPC), qui est organisée — environ une fois aux trois ans — dans le cadre de la conférence ICAPS (Vallati *et al.*, 2015). Même si quelques domaines de planification ont été ajoutés lors des dernières

compétitions de l'IPC, leur nombre total est relativement restreint et ne couvre pas l'ensemble des combinaisons de propriétés topologiques qui pourraient avoir un impact sur la performance relative des algorithmes. En effet, les domaines de l'IPC visent en général à modéliser des situations plus ou moins réalistes, plutôt qu'à couvrir l'entièreté des topologies de PDM imaginables. De plus, les domaines utilisés dans l'IPC sont principalement conçus pour évaluer des PDM à horizon fini (PDM-HF) et des PDM à horizon infini avec escompte (PDM-HIE) plutôt que des PDM-PCCS.

Ce chapitre est basé sur notre article présenté à la conférence de l'*International Federation of Classification Societies* (IFCS 2024), qui a eu lieu à San José, au Costa-Rica (Champagne Gareau *et al.*, 2024). Il vise à combler les lacunes mentionnées ci-dessus en proposant d'abord à la section 7.1 une liste de propriétés topologiques pouvant possiblement influencer la performance des algorithmes de planification probabiliste sur les PDM-PCCS, puis en proposant à la section 7.2 une approche permettant de générer des instances synthétiques de PDM-PCCS qui peuvent couvrir différentes combinaisons de propriétés topologiques d'intérêt.

7.1 Caractéristiques topologiques des PDM

Cette section présente certaines propriétés topologiques des PDM. Quelques-unes d'entre elles correspondent à des propriétés qui existent également en théorie des graphes, tandis que d'autres sont spécifiques aux PDM. La liste de propriétés que nous proposons est la suivante :

- Le **nombre d'états** $|S|$ dans le PDM.
- Le **nombre d'actions** $|A|$ dans le PDM.
- Le **nombre d'états buts** $|G|$ dans le PDM.
- Le **nombre de composantes fortement connexes (CFC)** $|\mathfrak{S}|$ dans le PDM.
- Le **nombre d'états dans la plus grande CFC** $\max_{\mathcal{S}\in\mathfrak{S}} |\mathcal{S}|$.
- La **distribution des actions** : $\forall k, P^a_k :=$ proportion des états qui ont k actions applicables.
- La **distribution des transitions** : $\forall k, P^t_k :=$ proportion des actions qui ont k transitions probabilistes.
- Le **coefficient de clustering** : $\mathfrak{C} := \frac{1}{|S|}\sum_{s\in S}\frac{e_s}{k_s(k_s-1)}$, où e_s est le nombre de paires d'états directement atteignables à partir de s qui sont également directement atteignables l'un de l'autre, et k_s est le nombre d'états directement atteignables à partir de s. De plus, $\mathfrak{C}$ est fixé à 0 lorsque $k_s < 2$.
- L'**excentricité des états buts** du MDP : $\mathcal{G} := \min_{g\in G}\max_{s\in S}\bar{d}(s,g)$, où $\bar{d}(s,g)$ est le nombre minimum d'actions (le coût de chaque action n'est pas considéré) qui doivent être exécutées pour atteindre g à partir de s.

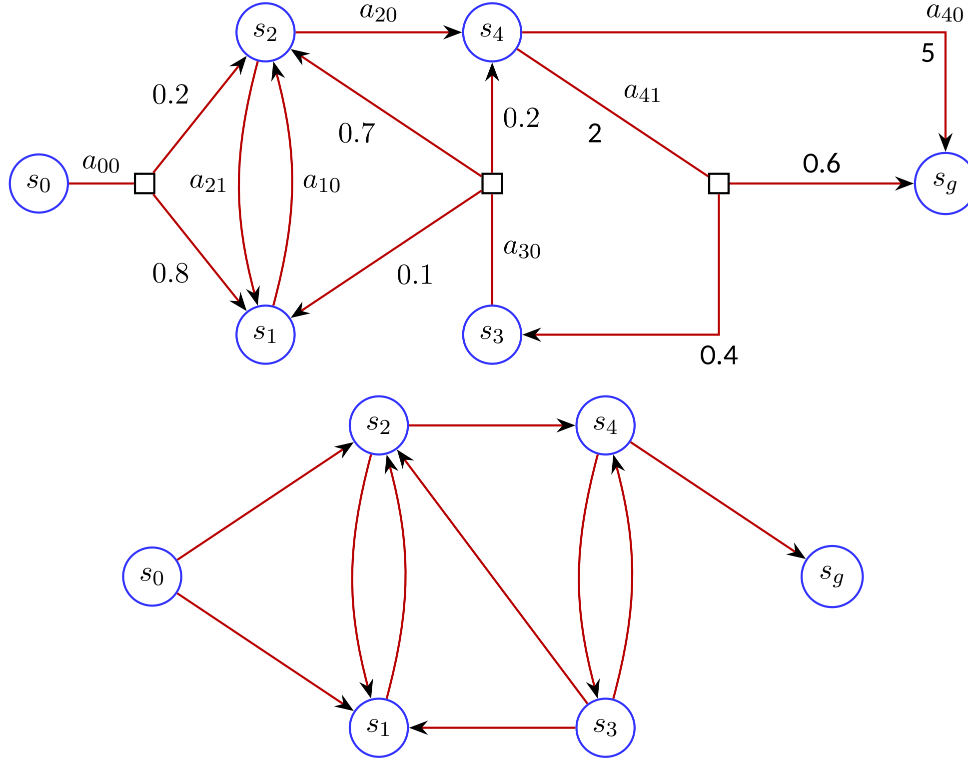


Figure 7.1 – Exemple d'une instance de PDM (haut) et du graphe correspondant à sa déterminisation *all-outcomes* (bas).

Certaines de ces propriétés sont déjà connues pour avoir un impact sur la performance de certains algorithmes de planification. Par exemple, le nombre de CFC $\mathfrak{S}$ et le nombre d'états buts $|G|$ dans un PDM affectent respectivement la performance de TVI et de LRTDP (Dai *et al.*, 2011). Comme l'indique la citation en épigraphe de ce chapitre, d'autres des propriétés mentionnées ci-dessus n'ont pas encore été étudiées en détail dans la littérature, mais sont soupçonnées d'avoir un impact sur la performance des algorithmes de planification.

Nous illustrons les propriétés topologiques mentionnées ci-dessus à l'aide de l'exemple de la figure 7.1. Le PDM en haut de la figure contient 6 états, 7 actions, 1 état but (s_g) et 3 CFC, $\{\{s_0\}, \{s_1, s_2, s_3, s_4\}, \{s_g\}\}$. La plus grande CFC contient 4 états. De plus, la distribution des actions est donnée par $\mathbf{P^a} = [\frac{1}{6}, \frac{3}{6}, \frac{2}{6}]$ et la distribution des transitions probabilistes est donnée par $\mathbf{P^t} = [0, \frac{4}{7}, \frac{2}{7}, \frac{1}{7}]$. Le coefficient de clustering est $\mathfrak{C} = \frac{1}{6}(\frac{2}{2 \cdot 1} + 0 + \frac{0}{2 \cdot 1} + \frac{3}{3 \cdot 2} + 0 + 0) = \frac{1}{4}$. Finalement, l'excentricité des états buts est $\mathcal{G} = 3$, puisqu'il faut exécuter au moins 3 actions pour atteindre s_g à partir de s_0.

7.2 Algorithme de génération d'instances synthétiques

Certains domaines de planification probabilistes sont synthétiques, dans le sens où ils ont été conçus non pas pour correspondre directement à une situation réelle, mais plutôt pour mesurer l'impact d'une caractéristique topologique particulière sur la performance relative des algorithmes de planification de PDM. Par exemple, les domaines *Layered* (Dai *et al.*, 2011) et *Chained* (Champagne Gareau *et al.*, 2023b) ont été conçus respectivement pour mesurer l'impact sur la performance d'algorithmes de planification (1) du nombre de CFC et (2) de leur placement relatif dans des "chaînes indépendantes". Cependant, ces domaines sont limités dans le sens où ils ne couvrent qu'un sous-ensemble des combinaisons possibles de propriétés topologiques d'intérêt. De plus, la conception de domaines synthétiques est chronophage. Par conséquent, dans cette section, nous proposons un algorithme de génération de PDM synthétiques paramétrique pouvant couvrir un large éventail de propriétés topologiques dans les instances de PDM générées.

L'approche de génération synthétique de PDM proposée dans cette section est inspirée du concept de déterminisation, qui consiste à générer un graphe à partir d'un PDM, ce qui revient à enlever (*déterminiser*) l'incertitude sur les transitions. La déterminisation a été initialement proposée comme une façon de résoudre les PDM en utilisant des algorithmes existants de planification déterministe (Yoon *et al.*, 2007). Plusieurs types de déterminisation ont été proposés. Les plus courantes sont la déterminisation *max-outcome* — où pour chaque action, seul le résultat le plus probable est conservé et devient un arc dans le graphe — et la déterminisation *all-outcomes* — où pour chaque action, chacun des résultats probabilistes possibles devient un arc dans le graphe. La figure 7.1 montre un exemple de déterminisation *all-outcomes*.

Le graphe obtenu suite à cette déterminisation peut être utilisé pour trouver et analyser des propriétés topologiques du PDM original. Par exemple, c'est sur le graphe produit suite à une déterminisation *all-outcomes* que l'algorithme TVI trouve les composantes fortement connexes d'un PDM. Le graphe résultant d'une déterminisation partage également d'autres propriétés avec le PDM original, telles que le coefficient de clustering. Cependant, certaines propriétés, telles que la distribution des transitions probabilistes, n'ont pas d'équivalence en théorie des graphes et doivent être calculées directement à partir du PDM.

L'idée derrière l'approche proposée est d'inverser le processus de déterminisation, c'est-à-dire de générer un PDM à partir d'un graphe. Cela nous permet d'utiliser des méthodes de génération de graphes existantes, qui servent ensuite de bases aux PDM synthétiques générés. Les propriétés des graphes synthétiques peuvent être utilisées pour contrôler certaines des propriétés topologiques des PDM correspondants.

Plusieurs algorithmes de génération de graphes synthétiques ont été proposés dans la littérature. Parmi les plus connus, on retrouve les modèles d'Erdös-Rényi (Erdös et Rényi, 1959), de Watts-Strogatz (Watts et Strogatz, 1998), de Barabási-Albert (Barabási et Albert, 1999) et de Kronecker (Leskovec *et al.*, 2010). Chacun de ces modèles génère des graphes ayant des propriétés topologiques distinctives. La table 7.1 montre quelques-unes des propriétés de ces méthodes de génération de graphes synthétiques.

Modèle	**Distribution degrés**	**Coefficient de clustering**	**Diamètre**
Erdös-Rényi	binomiale	petit ($\bar{k}/n$)	petit : $\mathcal{O}(\log n)$
Watts-Strogatz	quasi constante	grand	petit
Barabási–Albert	invariant d'échelle ($\bar{k}^{-3}$)	grand ($\bar{k}^{-1}$)	petit : $\mathcal{O}(\frac{\log n}{\log(\log n)})$
Kronecker	multinomiale	flexible	flexible

Table 7.1 – Méthodes de génération de graphes synthétiques et liste de certaines des propriétés topologiques leur étant associées, où $\bar{k}$ est le degré moyen des sommets dans le graphe, et n est le nombre de sommets.

Notre approche génère d'abord un graphe ayant les propriétés topologiques les plus proches de celles désirées. Elle utilise ensuite ce graphe comme base pour générer le PDM. Pour chaque état s dans le graphe, nous générons a_s actions, où a_s est un nombre aléatoire compris entre 1 et le degré k_s du sommet s dans le graphe. Nous générons ensuite un tableau qui consiste en a_s nombres aléatoires tels que leur somme est égale à k_s. Par exemple, si un sommet donné a un degré de 8, et que le nombre aléatoire d'actions est 3, un tableau possible pourrait être $[4, 1, 3]$. Ce tableau représente le nombre d'états qui peuvent être atteints en appliquant chacune des actions générées. L'étape suivante consiste à générer (1) un coût pour chaque action (suivant n'importe quelle distribution souhaitée), (2) une probabilité pour chaque transition possible (normalisée à 1) et (3) un état correspondant à chaque transition probabiliste possible de chaque action (parmi tous les voisins du sommet dans le graphe). Enfin, les états buts sont choisis au hasard parmi l'ensemble des états. L'algorithme 7.1 montre les étapes principales de l'approche proposée.

L'algorithme 7.1 et les quatre méthodes de génération de graphes présentées dans la table 7.1 font partie d'une bibliothèque de génération et d'analyse de graphes appelée `graph-toolkit`, que nous avons implémentée en C++ et qui est disponible en ligne [1]. Cette bibliothèque peut également mesurer et afficher toutes les propriétés topologiques de la section 7.1 Les figures 7.2 et 7.3 montrent respectivement un exemple de graphe synthétique généré par `graph-toolkit` en utilisant le modèle d'Erdös-Rényi $G(n, M)$ (où $n = 10$ et $M = 15$) et un PDM synthétique généré à partir de ce graphe à l'aide de l'algorithme 7.1.

1. `https://gitlab.info.uqam.ca/champagne_gareau.jael/graph-toolkit`

Algorithme 7.1 Génération d'une PDM-PCCS synthétique.

Entrée: Une liste de propriétés topologiques désirées (n : nombre d'états ; k : nombre de buts, etc.)
Sortie: Un PDM-PCCS (S, A, T, C, G)

1: ▷ Utiliser la méthode de génération de graphes la plus appropriée par rapport aux propriétés voulues
2: $\Gamma \leftarrow$ GénérerGrapheSynthétique(n) ▷ p. ex., une des méthodes de la table 7.1
3: $S \leftarrow \Gamma$.obtenirNœuds$()$ ▷ $|S| = n$
4:
5: **pour tout** $s \in S$ **faire**
6: $a_s \leftarrow$ EntierAléatoireUniformeEntre$(1, k_s)$ ▷ Générer le nombre d'actions ; k_s est le degré de s
7: $A_s \leftarrow$ DécomposerEnSomme(k_s, a_s) ▷ A_s est un tableau de a_s éléments où $\sum_{n_a \in A_s} n_a = k_s$
8: **pour tout** $n_a \in A_s$ **faire** ▷ n_a est le nombre de transitions possibles de l'action courante
9: $a \leftarrow$ nouvel identifiant d'action
10: $A \leftarrow A \cup \{a\}$
11: $C(s, a) \leftarrow$ CoûtAléatoire$()$ ▷ Distribution uniforme ou autre, selon le besoin
12: $P_a \leftarrow$ GénérerProbabilitées(n_a) ▷ P_a est un tableau où $\sum_{p \in P_a} p = 1.0$ et $|P_a| = n_a$
13: **pour tout** $i \in [1..n_a]$ **faire**
14: $s' \leftarrow$ VoisinAléatoire(Γ, s) ▷ Voisin aléatoire de s dans le graphe Γ
15: $T(s, a, s') \leftarrow P_a[i]$
16: $G \leftarrow$ SousEnsembleAléatoire(S, k) ▷ k est un paramètre contrôlant le nombre de buts souhaités
17: **retourne** (S, A, T, C, G)

L'approche proposée permet de générer des PDM synthétiques couvrant une variété de propriétés topologiques, ce qui facilite l'évaluation des algorithmes de planification probabiliste dans des scénarios où ces propriétés sont connues à l'avance. L'algorithme se distingue par sa simplicité, sa rapidité et sa flexibilité, mais il présente une limitation : le choix de la méthode de génération de graphe doit être fait manuellement. Pour surmonter cette faiblesse, il est prévu de développer une méthode permettant de sélectionner automatiquement la méthode de génération de graphe la plus appropriée en fonction des caractéristiques topologiques recherchées.

Comme travaux futurs, nous prévoyons de générer une grande quantité de PDM synthétiques en utilisant cette approche, de mesurer sur chacun de ces PDM plusieurs caractéristiques, telles que celles listées à la section 7.1, et d'évaluer la performance des algorithmes de planification probabiliste existants sur chacun de ces PDM. Avec les données recueillies, nous pourrons alors vraisemblablement entraîner un modèle de classification, où les caractéristiques topologiques des PDM sont les variables indépendantes et l'algorithme prédit comme étant le plus performant est la variable dépendante. À l'aide de ce classificateur, il sera alors possible de prédire quel algorithme est le plus susceptible d'être le plus performant sur un PDM donné, en fonction de ses caractéristiques topologiques.

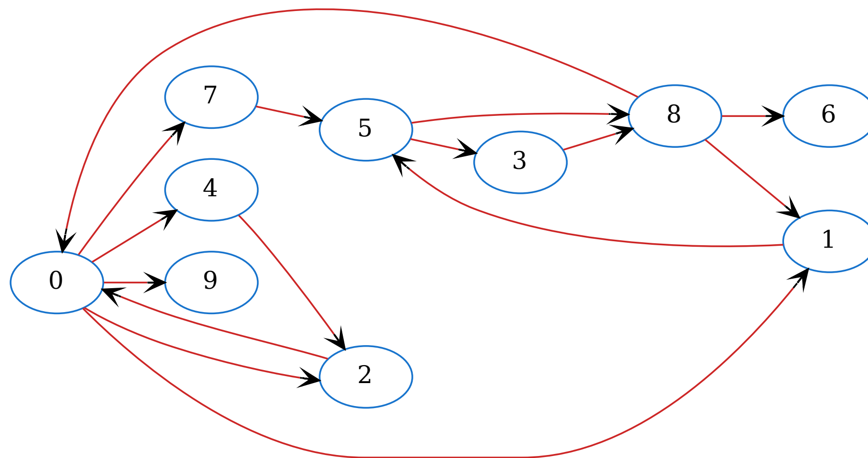


Figure 7.2 – Exemple d'un graphe synthétique généré par `graph-toolkit` en utilisant le modèle d'Erdös-Rényi $G(n, M)$, où $n = 10$ et $M = 15$.

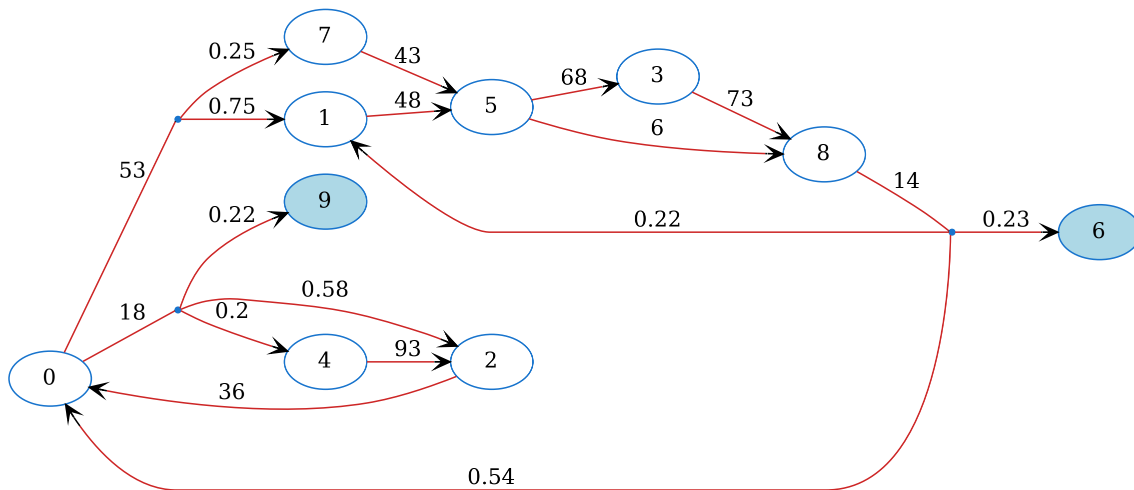


Figure 7.3 – Exemple d'un PDM synthétique généré à partir du graphe de la figure 7.2 à l'aide de l'algorithme 7.1. Les distributions probabilistes utilisées donnant le coût des actions et les probabilités de transitions sont toutes deux uniformes, respectivement $U(0, 100)$ et $U(0, 1)$. De plus, 2 états buts ont étés générés : 6 et 9.

CONCLUSION

La recherche doctorale présentée dans cette thèse avait pour objectif d'accélérer de manière significative la résolution des processus décisionnels de Markov (PDM) en exploitant des caractéristiques d'architecture bas niveau, tels que la hiérarchie de mémoire et le parallélisme des ordinateurs modernes, lors de la conception des structures de données pour représenter les PDM et des algorithmes permettant de les résoudre.

Pour ce faire, une étude de la littérature récente a d'abord été présentée, ce qui a permis de confirmer qu'encore très peu d'efforts ont été mis pour exploiter l'architecture des ordinateurs dans l'implémentation des planificateurs. Les contributions scientifiques de la thèse ont ensuite été présentées successivement.

— Une représentation mémoire des PDM : CSR-MDP, stockant les données internes du PDM de manière homogène et contiguë, permettant d'accélérer les planificateurs de PDM existants et servant de base aux nouveaux algorithmes proposés (Champagne Gareau *et al.*, 2022).
— Un algorithme : pcTVI, permettant de paralléliser le calcul d'une politique optimale pour les domaines de planification qui contiennent plusieurs chaînes indépendantes de composantes fortement connexes (Champagne Gareau *et al.*, 2023b).
— Deux algorithmes : eTVI et eiTVI, réordonnant les états d'un PDM en mémoire de sorte à minimiser le nombre de défauts de cache et à maximiser l'utilisation de chaque ligne de cache chargée, respectivement en réordonnant selon l'ordre extra composantes (les états dans une même composante sont contigus) et l'ordre intra composante (les états sont stockés dans l'ordre qui maximise la propagation des valeurs dans l'espace d'états) (Champagne Gareau *et al.*, 2023c).
— Un algorithme de génération de PDM-PCCS synthétiques, permettant d'avoir des instances topologiquement variées, et ainsi d'évaluer plus facilement les contextes où un algorithme est plus performant qu'un autre (Champagne Gareau *et al.*, 2024).

Les éléments d'architecture des ordinateurs mentionnés au chapitre 3 ont été pour la plupart exploités par au moins une des contributions présentées dans cette thèse. Par exemple, la hiérarchie de mémoire est exploitée par la structure CSR-MDP et par les algorithmes eTVI et eiTVI. L'algorithme pcTVI exploite quant à lui le parallélisme de fils d'exécutions (exécution sur plusieurs cœurs en parallèle), et permet d'obtenir des améliorations de vitesse très près du maximum théorique. Le parallélisme d'instructions et le parallélisme de données (SIMD) définis au chapitre 3 n'ont pas été considérés explicitement, mais ils le sont de

manière implicite logiciellement (par le compilateur) et matériellement (par l'ordonnanceur d'instructions dans le processeur). La structure CSR-MDP et l'algorithme eTVI, en rendant les accès mémoires plus prévisibles et contigus, rendent la considération automatique de ces niveaux de parallélisme plus efficace. Par exemple, on peut observer que le code généré par le compilateur contient plus d'instructions SIMD avec nos contributions que sans elles. Il serait toutefois possible d'envisager dans des travaux futurs de concevoir un algorithme exploitant explicitement ces deux formes de parallélismes.

Les contributions présentées ont une bonne synergie entre elles et les gains de performance obtenus sont donc cumulables. Par exemple, il est possible d'implémenter une instance de PDM sous format CSR-MDP, d'utiliser pcTVI pour décomposer l'espace d'états en chaînes de composantes fortement connexes, et de résoudre chacune de ces chaînes en parallèle en utilisant l'algorithme eTVI ou eiTVI. Les contributions combinées ont permis une accélération allant jusqu'à un facteur de 344.

Une limite des contributions proposées est qu'elles supposent que le PDM soit connu à l'avance et que l'espace d'états puisse être entièrement stocké en mémoire. Or, il arrive parfois que l'espace d'états soit trop grand, ou encore que celui-ci ne soit pas connu à l'avance (ce qui est le cas dans le contexte de l'apprentissage par renforcement). Une piste de recherche est donc de voir s'il est possible d'adapter les contributions présentées dans cette thèse à des PDM où l'espace d'états est découvert ou considéré dynamiquement.

Une autre limite des algorithmes eTVI, eiTVI et pcTVI est héritée de TVI : les domaines de planification ne possédant qu'un faible nombre de composantes fortement connexes ne peuvent pas bénéficier grandement de la décomposition de l'espace d'états. Par exemple, pcTVI et eTVI n'ont aucun impact par rapport à TVI dans un PDM ne possédant qu'une seule composante. Or, dans plusieurs domaines de planification réalistes, il arrive que toutes les actions soient réversibles — l'agent peut toujours retourner d'où il vient —, ce qui implique qu'il n'y a qu'une seule composante fortement connexe dans l'espace d'états. Pour pallier ce problème, il est prévu comme travail futur d'étudier des moyens de trouver des actions sous-optimales du PDM — c'est-à-dire qui ne peuvent faire partie d'une politique optimale — qui peuvent donc être élaguées, ce qui permettrait de trouver plus de composantes fortement connexes, rendant ainsi les algorithmes proposés applicables de manière efficace sur un nombre de domaines encore plus grand. L'une des stratégies prévues est de déterminer les *ponts orientés* du PDM (Italiano *et al.*, 2012) — les actions qui, si elles étaient retirées, feraient que le nombre de composantes fortement connexes du PDM augmenterait — et d'identifier lesquels, le cas échéant, peuvent être élagués du PDM sans perte d'optimalité.

ANNEXE A

PROGRAMME MDPTK

Afin de permettre la reproductivité des résultats pour que la communauté puisse comparer les contributions proposées et notre implémentation aux leurs, l'ensemble du code ayant découlé de cette thèse est disponible en ligne dans un projet appelé `MDPtk`[1]. En plus de l'implémentation mentionnée, le dépôt contient également les scripts utilisés au cours du projet, notamment pour générer et lancer les évaluations ayant été utilisées pour générer les différents résultats empiriques présentés dans cette thèse, générer les graphiques, etc. Cette annexe présente des détails d'implémentation et d'utilisation de `MDPtk`.

A.1 Implémentation

Le projet `MDPtk` a été écrit en C++ (standard 2020) et compilé avec g++ (version 13.2), du projet *GNU Compiler Collection*. Hormis la bibliothèque standard de C++, les seules dépendances sont :

- `glogs`[2] : une bibliothèque pour simplifier la journalisation des temps de calcul aux différentes étapes d'exécution ;
- `gflags`[3] : une bibliothèque permettant de simplifier la gestion des options (*flags*) données en entrée lors du lancement du programme ;
- `boost`[4] : utilisée uniquement pour son allocateur de mémoire dynamique alignée — malheureusement pas disponible dans la bibliothèque standard — qui permet d'utiliser les conteneurs, tel que `std::vector`, en précisant l'alignement des données allouées sur le tas par ceux-ci ;
- `cereal`[5] : une bibliothèque pour sérialiser/désérialiser les PDM en format binaire, ce qui permet de charger les tests beaucoup plus rapidement qu'en les chargeant à partir des fichiers textes décrivant les instances de tests produits par les générateurs.

La classe `MDP` dans le projet `MDPtk` est implémentée avec la structure CSR-MDP décrite au chapitre 4. La méthode principale de cette classe, `MDP::solve`, permet de calculer une politique optimale du PDM à la précision ϵ souhaitée. Cette méthode prend simplement en paramètre le nom de l'algorithme à utiliser.

1. https://gitlab.info.uqam.ca/champagne_gareau.jael/mdptk
2. https://github.com/google/glog
3. https://gflags.github.io/gflags/
4. https://www.boost.org/doc/libs/1_65_0/boost/align/aligned_allocator.hpp
5. https://uscilab.github.io/cereal/

A.2 Utilisation

Le projet `MDPtk` contient à la fois une bibliothèque, pouvant être utilisée dans n'importe quel projet, et un utilitaire en ligne de commande permettant de lancer les algorithmes de planification de PDM. La description des options permises lors du lancement du programme s'affiche en utilisant l'option `-helpshort` et est présentée à la figure A.1.

```
This program finds an optimal value function V* and policy Pi*
for a MDP specified in the 'inputFile' parameter, and output them in
the 'outputFile' parameter.

The solver to be used is specified in the 'solver' parameter.
Possible values: VI (Gauss-Seidel asynchronous variant)
                 LAOstar ILAOstar
                 RTDP    LRTDP   BRTDP
                 TVI     FTVI
                 eTVI    eiTVI   pcTVI

The heuristic to be used is specified in the 'heuristic' parameter.
Possible values: NONE:  No heuristic (h = 0)
                 H_MIN: h_min heuristic

The upper bound to be used is specified in the 'upperbound' parameter.
Possible values: NONE:  No upper bound (V_u = infty)
                 FTVI:  upper bound described in FTVI paper
                 BRTDP: upper bound described in BRTDP paper (DS-MPI)
                 BRTDP_FTVI: both previous upper bounds combined

-inputFile   path to a file containing the MDP to solve               type: string  default: "cin"
-outputFile  path to a file where to output results                   type: string  default: "cout"
-output      output the MDP in specified format (graph[,viz])         type: string  default: ""
-start       id of the start state (needed for some solvers)          type: int32   default: 0
-goal        id of the goal state [last state loaded if not specified] type: int32   default: 2147483647
-solver      name of the solver to use                                type: string  default: "VI"
-policy      print the obtained policy after the solver has completed type: bool    default: false
-graphviz    output graphviz representation of the loaded MDP to stdout type: bool  default: false
-heuristic   name of the heuristic to use                             type: string  default: "H_MIN"
-reach       do a reachability analysis before starting solver        type: bool    default: true
-upperbound  name of the initial upper bound for FTVI/BRTDP           type: string  default: "BRTDP"
-benchmark   print the info used by the benchmark script to stdout    type: bool    default: false
```

Figure A.1 – Message d'aide affiché lorsqu'on lance le programme `MDPtk` avec l'option `-helpshort` qui décrit les différentes options (*flags*) pouvant être spécifiées lors du lancement du programme.

Le projet `MDPtk` utilise son propre format de fichier. La première ligne contient le nombre d'états du PDM. Pour chacun d'eux, une ligne contient son identifiant et le nombre d'actions applicables. Pour chacune de ces actions, il y a une ligne contenant, en ordre, son coût, le nombre d'états atteignables par cette action, et pour chacun d'eux, l'identifiant de l'état d'arrivée et la probabilité de transition vers celui-ci. Un exemple de ce format de fichier, représentant le PDM de la figure 1.1 (où s_g est l'état 5), est présenté à la figure A.2.

```
6
0 2
1.00 1 1 1.00
1.00 1 2 1.00
1 1
1.00 1 2 1.00
2 2
1.00 1 1 1.00
1.00 1 4 1.00
3 1
1.00 1 4 1.00
4 2
2.00 2 3 0.40 5 0.60
5.00 1 5 1.00
5 0
```

Figure A.2 – Fichier d'entrée représentant une instance de PDM

Le projet ne supporte pas les deux formats les plus utilisés pour modéliser les domaines de planification probabilistes : PPDDL (*Probabilistic Planning Domain Description Language*) (Younes *et al.*, 2005) et RDDL (*Relational Dynamic influence Diagram Language*) (Sanner, 2010). Ceux-ci ont été, respectivement, les langages officiels de la compétition internationale de planification (*International Planning Competition*, IPC) d'avant (pour PPDDL) et après (pour RDDL) la compétition IPC 2011. Or, ces formats sont davantage adaptés aux PDM à horizon infini (définition 1.2) qu'aux PDM-PCCS. De plus, ils sont plus utiles lorsque l'environnement est découvert dynamiquement. Par exemple, RDDL est utilisé dans l'IPC 2023 pour construire un simulateur d'environnement de style *boîte noire* à l'aide d'une bibliothèque nommée `pyRDDLGym` (Taitler *et al.*, 2023). Or, les méthodes présentées dans cette thèse nécessitent de connaître l'ensemble du PDM considéré à l'avance, ce qui explique le choix d'un nouveau format dans `MDPtk`.

Le projet contient plusieurs générateurs d'instances de domaines, incluant les domaines : (1) *Layered*, (2) *Chained*, (3) *Wetfloor*, (4) *SAP*, (5) *MountainCar* et (6) *SysAdmin*. Les générateurs des trois premiers domaines ont été implémentés manuellement, tandis que les générateurs des trois derniers nous ont été gracieusement fournis par David Wingate, professeur à l'université BYU, qui les avaient implémentés pour son article sur P3VI (Wingate et Seppi, 2004) (nous les avons légèrement modifiés pour adapter le format de sortie). Tous les générateurs produisent des instances de PDM sous le format décrit ci-haut.

La figure A.3 présente le résultat affiché par le planificateur sur *stdout* lorsqu'on le lance avec la commande

```
./mdptk -inputFile exemple.mdp -solver TVI -policy
```

où `exemple.mdp` est le fichier d'entrée présenté à la figure A.2.

```
s_id    a_id    V(s_id)
0       1       6
1       2       6
2       4       5
3       5       5
4       6       4
5       Goal    0
```

Figure A.3 – Format de sortie de la fonction de valeur et de la politique optimale ($V^\star$ et $\pi^\star$)

Puisque les algorithmes de planification présentés dans cette thèse calculent tous une politique optimale, nous nous sommes plutôt intéressés au temps de calcul des diverses étapes de ceux-ci. En utilisant l'option `-logtostderr`, il est possible d'afficher tous les évènements journalisés par le programme sur la sortie d'erreur du terminal. La figure A.4 présente un exemple d'affichage obtenu en lançant le planificateur avec l'algorithme TVI sur une instance du domaine Layered de $500\,000$ états.

```
I20240126 19:30:00.183571 365554 solver.cpp:59] Creating the output and input stream
I20240126 19:30:00.183626 365554 solver.cpp:70] Reading the MDP from the input stream
I20240126 19:30:14.820829 365554 mdp.cpp:1130] size(states)        : 500001
I20240126 19:30:14.820852 365554 mdp.cpp:1131] size(actions)       : 4999991
I20240126 19:30:14.820858 365554 mdp.cpp:1132] size(costs)         : 4999990
I20240126 19:30:14.820861 365554 mdp.cpp:1133] size(neighbors)     : 27502673
I20240126 19:30:14.820866 365554 mdp.cpp:1134] size(probabilities) : 27502673
I20240126 19:30:14.820870 365554 mdp.cpp:1135] sizeof(mdp)         : 680
I20240126 19:30:14.820875 365554 solver.cpp:84] Initializing V_0 with the specified heuristic
I20240126 19:30:14.820880 365554 mdp.cpp:63] Computing the h_min heuristic
I20240126 19:30:15.003407 365554 solver.cpp:91] Detecting which states are reachable from the start
I20240126 19:30:15.117276 365554 mdp.cpp:101] Number of reachable states: 499775
I20240126 19:30:15.117547 365554 solver.cpp:96] Solving the MDP with the specified solver
I20240126 19:30:15.117553 365554 mdp.cpp:17] Using epsilon = 0.0001
I20240126 19:30:15.117568 365554 mdp.cpp:18] Solving MDP using TVI
I20240126 19:30:15.117571 365554 mdp.cpp:494] TVI Phase 1: compute the SCCs of the MDP
I20240126 19:30:15.388531 365554 mdp.cpp:500] TVI Phase 2: solve each SCC in reverse topological order
I20240126 19:30:18.753773 365554 mdp.cpp:505] Running time of TVI (Tarjan): 270 ms
I20240126 19:30:18.753798 365554 mdp.cpp:506] Running time of TVI (VI on SCCs): 3365 ms
I20240126 19:30:18.754405 365554 mdp.cpp:907] --- Partitions Info ---
I20240126 19:30:18.769241 365554 mdp.cpp:923] Number of SCC: 694
I20240126 19:30:18.769255 365554 mdp.cpp:924] Size of largest SCC: 50000
I20240126 19:30:18.769259 365554 mdp.cpp:925] -----------------------
I20240126 19:30:18.769264 365554 mdp.cpp:110] Number of bellman backups: 12692862
I20240126 19:30:18.769268 365554 solver.cpp:108] Max memory used: 1512 MB
I20240126 19:30:18.769275 365554 solver.cpp:109] Time to load MDP from file: 14637 ms
I20240126 19:30:18.769280 365554 solver.cpp:110] Time to compute the heuristic: 182 ms
I20240126 19:30:18.769284 365554 solver.cpp:111] Time to mark reachable states (DFS): 114 ms
I20240126 19:30:18.769289 365554 solver.cpp:112] Time to solve the MDP: 3651 ms
I20240126 19:30:18.769294 365554 solver.cpp:113] Time to find and evaluate the policy: 0 ms
```

Figure A.4 – Affichage détaillé des évènements de journalisation lors de l'exécution de `MDPtk`

RÉFÉRENCES

Abel, A. et Reineke, J. (2019). uops.info : Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. Dans *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, 673–686., Providence, RI, USA. ACM. http://dx.doi.org/10.1145/3297858.3304062

Amdahl, G. M. (2013). Computer Architecture and Amdahl's Law. *Computer*, *46*(12), 38–46. http://dx.doi.org/10.1109/MC.2013.418

Andre, D., Friedman, N. et Parr, R. (1998). Generalized prioritized sweeping. Dans *Proceedings of the Tenth International Conference on Neural Information Processing Systems (NIPS'97)*, 1001–1007., Denver, CO. MIT Press.

Barabási, A.-L. et Albert, R. (1999). Emergence of Scaling in Random Networks. *Science*, *286*(5439), 509–512. http://dx.doi.org/10.1126/science.286.5439.509

Bargiacchi, E., Roijers, D. M. et Nowé, A. (2020). AI-Toolbox : A C++ library for Reinforcement Learning and Planning (with Python Bindings). *Journal of Machine Learning Research*, *21*(102), 1–12. Récupéré le 2021-07-11 de http://jmlr.org/papers/v21/18-402.html

Barto, A. G. et Bradtke, S. J. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, *72*, 81–138. http://dx.doi.org/10.1016/0004-3702(94)00011-O

Bäuerle, N. et Rieder, U. (2011). *Markov Decision Processes with Applications to Finance*. Springer.

Bellman, R. (1957). *Dynamic Programming*. Prentice Hall.

Bentley, P. (2018). The end of Moore's Law : what happens next ? - BBC Science Focus Magazine. Récupéré le 2020-10-23 de https://www.sciencefocus.com/future-technology/when-the-chips-are-down/

Bernstein, D., Zilberstein, S., Washington, R. et Bresina, J. (2001). Planetary Robot Control as a Markov Decision Process. *Sixth International Symposium on Artificial Intelligence*.

Bertsekas, D. et Tsitsiklis, J. (2015). *Parallel and Distributed Computation : Numerical Methods*. Athena Scientific.

Bertsekas, D. P. (1995). *Dynamic Programming and Optimal Control*, volume 1. Athena scientific Belmont, MA.

Bonet, B. et Geffner, H. (2003). Labeled RTDP : Improving the Convergence of Real-Time Dynamic Programming. Dans *Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS 2003)*, volume 3, 12–21., Trento.

Bonet, B. et Geffner, H. (2006). Learning Depth-First Search : A Unified Approach to Heuristic Search in Deterministic and Non-Deterministic Settings, and its application to MDPs. Dans *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 142–151.

Burgess, N., Milanovic, J., Stephens, N., Monachopoulos, K. et Mansell, D. (2019). Bfloat16 processing for neural networks. Dans *Proceedings of the Symposium on Computer Arithmetic*, 88–91. Institute of Electrical and Electronics Engineers Inc. http://dx.doi.org/10.1109/ARITH.2019.00022

Champagne Gareau, J. (2019). *Planification d'itinéraires pour véhicule électrique avec disponibilité incertaine des bornes de recharge*. (Mémoire de maîtrise). Université du Québec à Montréal, Montréal. Récupéré le 2023-09-26 de https://archipel.uqam.ca/id/eprint/13780

Champagne Gareau, J., Beaudry, É. et Makarenkov, V. (2022). Cache-Efficient Memory Representation of Markov Decision Processes. *Proceedings of the Canadian Conference on Artificial Intelligence*. http://dx.doi.org/10.21428/594757db.0e910d58

Champagne Gareau, J., Beaudry, É. et Makarenkov, V. (2023a). Fast and optimal branch-and-bound planner for the grid-based coverage path planning problem based on an admissible heuristic function. *Frontiers in Robotics and AI*, *9*, 1076897. http://dx.doi.org/10.3389/frobt.2022.1076897

Champagne Gareau, J., Beaudry, É. et Makarenkov, V. (2023b). pcTVI : Parallel MDP solver using a decomposition into independent chains. Dans P. Brito, J. G. Dias, B. Lausen, A. Montanari, et R. Nugent (dir.). *Classification and Data Science in the Digital Age – IFCS 2022*, Studies in Classification, Data Analysis, and Knowledge Organization, 101–109., Cham. Springer International Publishing. http://dx.doi.org/10.1007/978-3-031-09034-9_12

Champagne Gareau, J., Beaudry, É. et Makarenkov, V. (2024). Towards topologically diverse probabilistic planning benchmarks : Synthetic domain generation for markov decision processes. Dans *Classification and Data Science in the Digital Age – IFCS 2024*, Studies in Classification, Data Analysis, and Knowledge Organization, Cham. Springer International Publishing. Soumis, en évaluation.

Champagne Gareau, J., Gosset, G., Beaudry, É. et Makarenkov, V. (2023c). Cache-Efficient Dynamic Programming MDP Solver. Dans *Proceedings of the 26th European Conference on Artificial Intelligence (ECAI 2023)*, volume 372 de *Frontiers in Artificial Intelligence and Applications*, 373–380., Krakow. IOS Press. http://dx.doi.org/10.3233/FAIA230293

Dai, P. et Goldsmith, J. (2006). LAO*, RLAO*, or BLAO*. Dans *AAAI Workshop on Heuristic Search*. AAAI Press.

Dai, P., Mausam, Weld, D. S. et Goldsmith, J. (2011). Topological value iteration algorithms. *Journal of Artificial Intelligence Research*, *42*, 181–209.

Dreyfus, S. (2002). Richard Bellman on the Birth of Dynamic Programming. *Operations Research*, *50*(1), 48–51. http://dx.doi.org/10.1287/opre.50.1.48.17791

Dubins, L. E., Maitra, A. P. et Sudderth, W. D. (2002). Invariant Gambling Problems and Markov Decision Processes. In *Handbook of Markov Decision Processes* 409–428. Springer, Boston, MA.

Erdös, P. et Rényi, A. (1959). On random graphs I. *Publicationes Mathematicae*, *6*, 290–297.

Fog, A. (2023). *Optimizing Software in C++*. Rapport technique, Technical University of Denmark.

Franco, J., Hagelin, M., Wrigstad, T., Drossopoulou, S. et Eisenbach, S. (2017). You can have it all : Abstraction and good cache performance. Dans *Onward ! 2017 : Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Co-Located with SPLASH 2017*, volume 17, 148–167. http://dx.doi.org/10.1145/3133850.3133861

Ghallab, M., Nau, D. S. et Traverso, P. (2016). *Automated Planning and Acting*. Cambridge University Press.

González, A. (2011). Moore's law implications on energy reduction. Dans *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC'11)*, New York, USA. Association for Computing Machinery (ACM).

Goodfellow, I., Bengio, Y. et Courville, A. (2016). *Deep Learning*, volume 22. MIT Press. http://dx.doi.org/10.1038/nature14539

Goto, K. et Van de Geijn, R. (2002). *On reducing TLB misses in matrix multiplication (TR02-55)*. Rapport technique, Department of Computer Sciences, U. of Texas at Austin.

Goto, K. et Van De Geijn, R. (2008). Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, *34*(3).

Guo, C., Hsueh, B. Y., Leng, J., Qiu, Y., Guan, Y., Wang, Z., Jia, X., Li, X., Guo, M. et Zhu, Y. (2020). Accelerating sparse DNN models without hardware-support via tile-wise sparsity. Dans *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC*. ACM. http://dx.doi.org/10.1109/SC41405.2020.00020

Hansen, E. A. et Zilberstein, S. (2001). LAO* : A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, *129*(1-2), 35–62.

Hart, P., Nilsson, N. et Bertram, R. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems, Man and Cybernetics, Part A : Systems and Humans - TSMCA ,*, *4*(2), 100–107. http://dx.doi.org/10.1109/tssc.1968.300136. Récupéré le 2018-05-28 de http://ieeexplore.ieee.org/document/4082128/

Haslum, P. (2024). Writing Planning Domains and Problems in PDDL. Récupéré le 2024-10-13 de https://users.cecs.anu.edu.au/~patrik/pddlman/writing.html

Henry, G., Tang, P. T. P. et Heinecke, A. (2019). Leveraging the bfloat16 artificial intelligence datatype for higher-precision computations. Dans *Proceedings of the Symposium on Computer Arithmetic*, 69–76. Institute of Electrical and Electronics Engineers Inc. http://dx.doi.org/10.1109/ARITH.2019.00019

Howard, R. A. (1960). *Dynamic Programming and Markov Processes*. John Wiley.

Howard, R. A. (2002). Comments on the origin and application of Markov decision processes. *Operations Research*, *50*(1), 100–102.

Italiano, G. F., Laura, L. et Santaroni, F. (2012). Finding strong bridges and strong articulation points in linear time. Dans *Theoretical Computer Science*, volume 447, 74–84. http://dx.doi.org/10.1016/j.tcs.2011.11.011

Jain, A. et Sahni, S. (2020). Cache efficient Value Iteration using clustering and annealing. *Computer Communications*, *159*, 186–197. http://dx.doi.org/10.1016/j.comcom.2020.04.058

Jurafsky, D. et Martin, J. H. (2009). *Speech and Language Processing : An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Pearson Prentice Hall.

Kolobov, A., Mausam et Weld, D. S. (2012). LRTDP versus UCT for online probabilistic planning. Dans *Proceedings of the National Conference on Artificial Intelligence*, 1786–1792.

LaMarca, A. et Ladner, R. E. (1999). The Influence of Caches on the Performance of Sorting. *Journal of Algorithms*, *31*(1), 66–104.

Langdale, G. et Lemire, D. (2019). Parsing gigabytes of JSON per second. *VLDB Journal*, *28*(6), 941–960. http://dx.doi.org/10.1007/s00778-019-00578-5

LaValle, S. M. (2006). *Planning algorithms*. Cambridge University Press.

Lemire, D. et Boytsov, L. (2015). Decoding billions of integers per second through vectorization. *Software : Practice and Experience*, *45*(1), 1–29. http://dx.doi.org/10.1002/spe.2203

Leskovec, J., Chakrabarti, D., Kleinberg, J., Faloutsos, C. et Ghahramani, Z. (2010). Kronecker Graphs : An Approach to Modeling Networks. *Journal of Machine Learning Research*, *11*(2).

Marcus, S. I., Fernández-Gaucherand, E., Hernández-Hernández, D., Coraluppi, S. et Fard, P. (1997). Risk Sensitive Markov Decision Processes. In *Systems and Control in the Twenty-First Century* 263–279. Birkhäuser Boston.

Mausam et Kolobov, A. (2012). *Planning with Markov Decision Processes : an AI perspective*. Morgan & Claypool Publishers.

McCool, M., Reinders, J. et Robison, A. (2012). *Structured Parallel Programming : Patterns for Efficient Computation*. Elsevier.

McMahan, H. B. et Gordon, G. J. (2005). Fast Exact Planning in Markov Decision Processes. Dans *Proceedings of the Third International Conference on Automated Planning and Scheduling*, 151–160.

McMahan, H. B., Likhachev, M. et Gordon, G. J. (2005). Bounded real-time dynamic programming : RTDP with monotone upper bounds and performance guarantees. Dans *ICML 2005 - Proceedings of the 22nd International Conference on Machine Learning*, 569–576.

Moore, A. W. et Atkeson, C. G. (1993). Prioritized sweeping : Reinforcement learning with less data and less time. *Machine Learning*, *13*(1), 103–130.

Moore, G. (1965). Moore's Law. *Electronics Magazine*, *38*(8), 114.

Nilsson, N. J. (1982). *Principles of Artificial Intelligence*. Springer Science & Business Media.

Nizipli, Y. et Lemire, D. (2023). Parsing millions of URLs per second. *Software : Practice and Experience*, 1–15. http://dx.doi.org/10.1002/spe.3296

Park, J. S., Penner, M. et Prasanna, V. K. (2004). Optimizing graph algorithms for improved cache performance. *IEEE Transactions on Parallel and Distributed Systems*, *15*(9), 769–782.

Qian, C., Childers, B., Huang, L., Guo, H. et Wang, Z. (2018). CGAcc : A compressed sparse row representation-based BFS graph traversal accelerator on hybrid memory cube. *Electronics (Switzerland)*, *7*(11). http://dx.doi.org/10.3390/electronics7110307

Raina, R., Madhavan, A. et Ng, A. Y. (2009). Large-scale deep unsupervised learning using graphics processors. Dans *Proceedings of the 26th International Conference On Machine Learning, ICML 2009*, 873–880.

Russell, S. et Norvig, P. (2020). *Artificial Intelligence : A Modern Approach* (4th edition éd.). Hoboken : Pearson.

Sanner, S. (2010). Relational Dynamic Influence Diagram Language (RDDL) : Language Description. Récupéré le 2024-04-20 de https://users.cecs.anu.edu.au/~ssanner/IPPC_2011/RDDL.pdf

Scherrer, B., Ghavamzadeh, M., Gabillon, V., Lesner, B. et Geist, M. (2015). Approximate Modified Policy Iteration and its Application to the Game of Tetris. *Journal of Machine Learning Research*, *16*(49), 1629–1676.

Smith, T. et Simmons, R. G. (2006). Focused Real-Time Dynamic Programming for MDPs : Squeezing More Out of a Heuristic. Dans *Proceedings of the Twenty-first National Conference on Artificial Intelligence*.

Suchow, J. W. et Griffiths, T. L. (1965). Deciding to Remember : Memory Maintenance as a Markov Decision Process. *CogSci*, 2063–2068.

Sutton, R. S. et Barto, A. G. (2018). *Reinforcement learning : an introduction*. Cambridge : MIT Press.

Taitler, A., Gimelfarb, M., Jeong, J., Gopalakrishnan, S., Mladenov, M., Liu, X. et Sanner, S. (2023). pyRDDLGym : From RDDL to Gym Environments. Dans *Bridging the Gap Between AI Planning and Reinforcement Learning - PRL Workshop @ ICAPS 2023*, Prague, Czech Republic. AAAI Press.

Tarjan, R. (1972). Depth-First Search and Linear Graph Algorithms. *SIAM journal on computing*, *1*(2), 15. http://dx.doi.org/10.1137/0201010

Turing, A. M. (1950). Computing Machinery and Intelligence. *Mind*, *49*, 433–460.

Vallati, M., Chrpa, L., Grześ, M., McCluskey, T. L., Roberts, M. et Sanner, S. (2015). The 2014 International Planning Competition : Progress and Trends. *AI Magazine*, *36*(3), 90–98. http://dx.doi.org/10.1609/aimag.v36i3.2571

Watts, D. J. et Strogatz, S. H. (1998). Collective dynamics of 'small-world' networks. *Nature*, *393*(6684), 440–442. http://dx.doi.org/10.1038/30918

Wheatman, B. et Xu, H. (2018). Packed Compressed Sparse Row : A Dynamic Graph Representation. Dans *IEEE High Performance Extreme Computing Conference, HPEC 2018*. Institute of Electrical and Electronics Engineers Inc. http://dx.doi.org/10.1109/HPEC.2018.8547566

Wingate, D. et Seppi, K. D. (2004). P3VI : A partitioned, prioritized, parallel value iterator. Dans *Proceedings of the Twenty-First International Conference on Machine Learning, ICML 2004*, 863–870.

Wingate, D. et Seppi, K. D. (2005). Prioritization methods for accelerating MDP solvers. *Journal of Machine Learning Research*, *6*, 851–881.

Yoon, S., Fern, A. et Givan, R. (2007). FF-Replan : A baseline for probabilistic planning. Dans *ICAPS 2007, 17th International Conference on Automated Planning and Scheduling*, 352–359.

Younes, H. L. S., Littman, M. L., Weissman, D. et Asmuth, J. (2005). The First Probabilistic Track of the International Planning Competition. *Journal of Artificial Intelligence Research*, *24*, 851–887. http://dx.doi.org/10.1613/jair.1880