View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by OpenGrey Repository

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par **l'Institut Supérieur de l'Aéronautique et de l'Espace**
Spécialité : Informatique

Présentée et soutenue par **Pascal SCHMIDT**
le **24 septembre 2012**

Planification multi-niveaux avec expertise humaine

JURY

M. Jean-François Gabard, président
M. Alain Dutech, rapporteur
M. Humbert Fiorino
M. Malik Ghallab
M. Éric Jacopin, rapporteur
M. Abdell-Illah Mouaddib
M. Florent Teichteil-Königsbuch, co-directeur de thèse
M. Gérard Verfaillie

École doctorale : **Systèmes**

Unité de recherche : **Équipe d'accueil ISAE-ONERA MOIS**

Directeur de thèse : **M. Patrick Fabiani**
Co-directeur de thèse : **M. Florent Teichteil-Königsbuch**

Remerciements

C'est une page qui se tourne pour moi que de finaliser ce manuscrit. Mais bien que ce soit le fruit de mon travail qui soit présenté ici, je n'aurais pas pu le réaliser seul, et bien des personnes méritent d'y être associées.

Je tiens tout d'abord à remercier vivement Christophe Garion, professeur à Supaéro, qui a su me pousser sur le chemin de la thèse et me donner les bons contacts aux bons moment pour trouver ce sujet et cette équipe de travail. Christophe, j'ai beaucoup apprécié ton engagement auprès des élèves, que ce soit au niveau des cours comme au niveau des projets professionnels. Je garderai longtemps en mémoire une course effrénée et perdue d'avance pour trouver un financement qui m'aurait permis de faire un stage aux États-Unis mais dans laquelle tu as refusé de baisser les bras tant que toutes les solutions envisageables, même les plus farfelues n'avaient pas été testées.

Comment ne pas ensuite penser à Florent Teichteil, celui qui a défini le sujet de cette thèse ? Je t'en ai donné du fil à retordre avec mes difficultés à me mettre à la formalisation et mes idées toujours très arrêtées. Mais lequel est le plus têtu d'entre nous ? Ce serait difficile à déterminer, et le jugement serait certainement contesté par le non-lauréat. Après tout, n'est-ce pas justement cette confrontation qui a permis à cette thèse d'avancer et d'atteindre son but ? Quoi qu'il en soit, je te dois un très grand merci pour ton encadrement, pour m'avoir poussé à avancer et pour avoir été à mon écoute quand le moral n'était pas au plus fort.

Patrick Fabiani m'a été présenté au début comme quelqu'un qui serait plus un nom sur un papier qu'un directeur de thèse, car trop occupé par ses obligations de directeur de département. Je tiens donc à contredire ces idées reçues ! Il n'a certes pas toujours été facile d'obtenir quelques minutes de ton temps, Patrick, et ça s'est souvent traduit par des heures de sorties du travail peu recommandables, mais ça en a systématiquement valu le coup. Ton point de vue toujours détaché et libéré du terre à terre m'a permis de prendre du recul sur mes travaux (une fois le choc du "Mais qu'est-ce qu'il raconte ?" passé) et de bien progresser.

Outre les conseils et l'encadrement officiel, j'ai tout de même pu me rendre compte que l'encadrement d'une thèse, ce n'est pas seulement les deux personnes indiquées sur le papier. Non, ce n'est vraiment pas pour rien que le laboratoire d'accueil aussi est mentionné. Je ne citerai aucun nom de peur d'en oublier et que ceux-là se vexent, mais je peux dire que travailler dans un laboratoire avec une telle dimension humaine, où tout le monde est heureux de faire son travail et le montre, où tout le monde est ouvert et volontaire pour donner des conseils et de l'aide, c'est vraiment une expérience magnifique et enrichissante aussi bien sur le plan moral qu'intellectuel. Merci à vous tous et pourvu que ça dure !

Une thèse, c'est bien sûr un début et un développement, mais c'est aussi une fin (sans pour autant être une fin en soi). On a beau croire que le plus dur est passé une fois le manuscrit remis entre les mains des rapporteurs, il n'en est rien. Je remercie très chaleureusement Alain Dutech et Éric Jacopin, les deux rapporteurs de

cette thèse, pour le temps qu'ils y ont consacré, les retours extrêmement pertinents qu'ils m'ont faits, les conseils qu'ils ont pu me donner, et les travaux vers lesquels ils m'ont redirigé. Alain, j'ai tout particulièrement apprécié les échanges que nous avons eus et qui ont grandement contribué à l'amélioration de mon manuscrit. Je tiens aussi à remercier les membres du jury, pour m'avoir fait l'honneur d'être là pour ma soutenance et pour leurs questions pertinentes et constructives. Jean-François, je te remercie tout particulièrement pour ton rôle de président du jury que tu as tenu avec brio et pour ta relecture attentive de mon manuscrit.

Il ne faut tout de même pas oublier que ce qui fait un docteur, ce n'est pas seulement ses travaux de thèse, mais aussi toute la vie autour.

C'est pourquoi quand je pense à cette thèse, je pense aussi à Charles Lesire et Magali Barbier, avec qui j'ai beaucoup aimé travailler pendant mon stage de M2, et qui ont presque réussi à me débaucher de la thèse prévue de longue date avec Florent pour la faire sur leur sujet. Je pense que vous n'avez pas perdu au change avec Thibault.

Je tiens aussi à dire un grand merci à Pierre Siron, professeur à Supaéro, pour les trois années que nous avons passées ensemble à encadrer des élèves en C, et pour m'avoir offert la possibilité de faire aussi de l'enseignement. J'espère avoir la possibilité de continuer à travailler avec toi par la suite.

Ces petits plus qui font un docteur, qui aident à garder le moral, ne sont pas forcément à chercher très loin non plus. Ils sont souvent autour d'un repas, d'une pause café ou d'une soirée. Oui, je pense bien sûr à vous tous, doctorants du DCSD, à toutes ces pauses café passées à résoudre des énigmes ou à refaire le monde. Glop, Nicos, Gregs, Julien, Stéphane et Patrice, nos aînés qui nous ont prouvé qu'on pouvait s'en sortir ; Alex avec qui j'ai partagé les moindres détails de mes études supérieures (je maintiens, c'est toi le copieur), Quentin, mon co-bureau des premières heures avec qui on a refait le monde un nombre incalculable de fois, Julie, et ton inoubliable cuisine arménienne, Mario et tes spécialités mexicaines qui nous décapent la plomberie, et Jack, ce sacré Romain (ou l'inverse), toujours avec le sourire et le premier à rire à mes blagues, même celles dont j'avais honte. Caro, rassure-toi, mon principal souvenir que je garderai ne sera pas les innombrables fois où je suis venu t'aider à déboguer ton code, mais ta gaieté permanente qui apporte le bonheur partout où tu passes. Simon, beaucoup pensaient à tort qu'il était difficile de faire plus geek que moi, tu apportes la preuve du contraire me surpassant de loin dans beaucoup de domaines, mais je garde une longueur d'avance à SupCom. Quand est-ce qu'on s'en refait un ? Gaëtan, Sergio, Clément, Pierre, Sylvain et Damien, je pense aussi très fort à vous en écrivant ces lignes. À tous ceux qui sont encore dans la thèse, que vous soyez en pleine écriture ou juste à vos débuts, accrochez-vous, le jeu en vaut la chandelle !

Thibault, mon co-bureau de trois longues années, que certains qualifieraient de mon jumeau tellement on était toujours dans les mêmes délires, non, je ne t'ai pas oublié. Je n'ai pas oublié non plus tous les projets dans lesquels on s'est engagés tous les deux et l'hélico qu'il va falloir qu'on finisse un jour. Je pense qu'il me sera

difficile de trouver un autre co-bureau de la même qualité. Merci à toi pour toutes ces années passées en ta compagnie, et promis, si on se retrouve à travailler ensemble dans le futur, je ne ferai plus joujou avec la pince à agrafes. Bon courage pour la fin de ta thèse, tu y es presque !

Plus loin du bureau mais d'une importance à ne pas sous-estimer, je dois un grand merci à tous mes amis. Les mauvaises langues diront que ce n'est pas si loin que ça du bureau, à peine trois cent mètres par le chemin le plus court. Parce que oui, il faut bien l'avouer, le Simu présente une très forte intersection avec mes amis qui ne sont pas déjà cité plus haut dans ces remerciements. À vous tous, amis rôlistes, amis joueurs, je vous dis merci pour toutes les soirées que l'on a passées ensemble dans ce sous-sol et qui m'ont permis de bien décompresser. Je déliverai une petite mention spéciale à Valérian et Olivier pour avoir eu la difficile tâche de m'avoir supporté à la fois comme prof et comme ami. Ce seront les seuls noms que je citerai, mais ne pensez pas, tous les autres, que c'est parce que je ne vous aime pas, bien au contraire. Mais si je commençais à vous faire un petit mot pour chacun, les remerciements deviendraient vite plus longs que le reste du manuscrit. J'espère que vous me le pardonnerez.

Cyril, ô toi mon parrain, ô toi qui a eu le courage de faire une thèse en physique fondamentale à la sortie d'une école d'aéronautique, ô toi qui m'a servi de modèle et qui m'a montré qu'on pouvait surmonter les pires difficultés d'une thèse pour finir par obtenir le sacro-saint diplôme, je te remercie. Je pense encore à tous ces moments où nous nous sommes remontés le moral l'un l'autre, à tous ces bons conseils que tu as pu me donner. Mais je pense encore plus à tous ces bons moments que l'on a pu passer tous les deux ou avec la râliste dans laquelle tu m'as introduit. J'espère par dessus tout avoir l'occasion de te revoir prochainement sur les planches.

Mes pensées vont maintenant à ma famille, à mes parents, Odile et Jean-Christophe, et à ma fratrie, Nicolas, Carole et Ludovic, qui ont toujours été à mes côtés (Carole, je te pardonne de n'avoir pas été là le jour J, je sais ce que c'est que la prépa). J'ai été très heureux de vous voir et d'avoir pu être avec vous pour fêter ça juste après la soutenance. Merci aussi à tout le reste de ma famille, que je porte bien au chaud dans mon cœur.

Angela, je ne peux conclure autrement ces remerciements que par toi. Toi qui m'a soutenu de façon inconditionnelle, toi qui a toujours cru en moi et qui, avec amour m'a redonné confiance les fois où elle faisait défaut et m'a remonté le moral pendant les moments difficiles. L'angoisse de cette thèse m'a parfois rendu ronchon, mais il en aurait fallu bien plus pour t'arrêter. Maintenant les rôles s'inversent, et je vais pouvoir te rendre la pareille.

Table des matières

Introduction **1**
1 Planification . . . 2
1.1 Historique . . . 2
1.2 Terminologie . . . 3
1.3 Applications . . . 3
2 Motivation . . . 4
2.1 La planification hiérarchique . . . 4
2.2 La planification hiérarchique avec expertise . . . 6
2.3 Préparer hiérarchiquement une stratégie . . . 7
3 Plan du manuscrit . . . 10

1 Contexte théorique et formel **11**
1.1 La planification classique . . . 11
1.1.1 Formalisation et PDDL . . . 11
1.1.2 Exemple de formalisation de problème . . . 13
1.1.3 Principaux algorithmes . . . 14
1.2 Autres formes de planification . . . 18
1.2.1 Planification temporelle . . . 19
1.2.2 Planification probabiliste . . . 19
1.2.3 Planification multi-agent probabiliste . . . 21
1.2.4 Expertise humaine . . . 22
1.3 Les structures hiérarchiques automatiques . . . 23
1.3.1 En planification de trajectoire . . . 23
1.3.2 En planification classique . . . 24
1.4 Formalismes de connaissance procédurale non retenus . . . 25
1.4.1 Nonlin et O-Plan . . . 25
1.4.2 High Level Actions . . . 26
1.5 Les HTN, un cadre privilégié pour la planification hiérarchique . . . 27
1.5.1 Intuition . . . 27
1.5.2 Formalisme des HTN . . . 29
1.5.3 Exemple . . . 31
1.5.4 Algorithme SHOP2 . . . 33
1.5.5 Les HTN probabilistes . . . 34
1.6 Synthèse . . . 36

2 Formalisation et contributions **39**
2.1 Expressivité choisie . . . 39
2.2 Comment lier des tâches de haut niveau ? . . . 40
2.2.1 Motivations . . . 40
2.2.2 Les méta-effets . . . 42

2.2.3 Notion d'optimisme . . . 43
2.2.4 Exemple . . . 44
2.2.5 Intégration des heuristiques . . . 46
2.3 Résoudre les problèmes de hiérarchie . . . 47
2.3.1 Motivations . . . 47
2.3.2 Les macro-tâches . . . 49
2.3.3 Utilisation . . . 49
2.3.4 Exemple . . . 50
2.4 Formalisation générale d'un couple domaine-problème . . . 52

3 Algorithmes 55
3.1 Planifier au sein d'un niveau . . . 55
3.1.1 Idée générale . . . 56
3.1.2 Planification par niveau avec A* . . . 57
3.1.3 Ajout de la notion de parallélisme de tâche . . . 64
3.2 Gérer les interactions entre les niveaux . . . 65
3.2.1 Idée générale . . . 65
3.2.2 Communication vers les niveaux inférieurs . . . 67
3.2.3 Backtracks . . . 68
3.2.4 Détection précoce des sur-coûts et communication entre les niveaux . . . 69
3.3 Propriétés . . . 71
3.3.1 Cadre d'application . . . 71
3.3.2 Propriétés dans les situations favorables . . . 72
3.4 Déroulement sur un exemple . . . 75

4 Expérimentations et résultats 77
4.1 Protocole de test . . . 77
4.1.1 Choix d'implémentation . . . 77
4.1.2 Domaines de test . . . 78
4.2 Performances . . . 86
4.2.1 Performance des différents apports . . . 86
4.2.2 Comparaison avec SHOP2 . . . 89
4.2.3 Expérimentations sur un domaine réel . . . 93

Conclusion et perspectives 95
1 Synthèse des contributions . . . 95
2 Travaux complémentaires . . . 97
2.1 Voies de recherche sur le planificateur de niveau . . . 97
2.2 Travaux sur l'interaction entre les niveaux . . . 98
2.3 Extensions de langage . . . 98

Bibliographie 101

A Fichiers de domaines **107**
A.1 Blocks World - PDDL 107
A.1.1 Fichier de domaine 107
A.1.2 Fichier de problème 108
A.2 Labyrinth - Formalisme HDS 109
A.2.1 Fichier de domaine 109
A.2.2 Fichier de problème 110
A.3 Blocks World - Formalisme HDS 111
A.4 Zeno Traveler - Formalisme HDS 114
A.5 Freecell - Formalisme HDS 120
A.6 Explore and Guide - Formalisme HDS 135
A.7 Déminage - Formalisme HDS 138

Introduction

Sommaire

1 Planification 2
1.1 Historique 2
1.2 Terminologie 3
1.3 Applications 3
2 Motivation 4
2.1 La planification hiérarchique 4
2.2 La planification hiérarchique avec expertise 6
2.3 Préparer hiérarchiquement une stratégie 7
3 Plan du manuscrit 10

1956, Dartmouth, États-Unis. Alors que la Guerre qui a vu naître les premiers ordinateurs est finie depuis une dizaine d'années et que ceux-ci ont commencé à faire leurs preuves dans les domaines du jeu de dames et des preuves de théorème, John McCarthy réunit quelques chercheurs du domaine (parmi lesquels Marvin Minsky, Nathaniel Rochester et Claude Shannon) à l'université de Dartmouth afin de définir les grands problèmes prioritaires de l'Intelligence Artificielle (et au passage inventer le terme).

De ce mois de réflexion est sortie une liste de problématiques pour lesquelles il suffirait, prétendument, de se pencher à plusieurs pendant quelques mois pour y apporter une solution et poser les fondements de l'Intelligence Artificielle. Au sein de cette liste, on trouve la simulation de capacités intrinsèquement humaines telles que le traitement du langage naturel, l'apprentissage, les capacités d'abstraction et de créativité.

Les vingt années suivantes ont vu exploser le nombre de travaux dans le domaine, et on a pu voir de grandes avancées dans les domaines du traitement du langage naturel et de la planification. Certains, malgré le fait que les problèmes précédents étaient toujours considérés comme difficiles continuaient à faire preuve d'optimisme. Ainsi on a pu entendre Herbert Simon dire, en 1965 : «Les machines seront capables, d'ici une vingtaine d'années, de faire n'importe quel travail qu'un humain sait faire», ou alors Marvin Minsky, en 1967 : «D'ici une génération [...] le problème de créer une "intelligence artificielle" sera globalement résolu» [Crevier 1993].

Mais les investisseurs eurent vite fait de déchanter. En effet, les résultats n'atteignaient pas les promesses des plus optimistes, les algorithmes atteignaient les limites de la puissance de calcul des machines, très modestes à l'époque, et la théorie de la complexité par Richard Karp [Karp 1972] a montré que les problèmes auxquels

on s'attaquait étaient bien trop difficiles pour la puissance de calcul disponible. En effet, la catégorie de problèmes auxquels on s'attaque appartient majoritairement aux problèmes de type NP-complet ou NP-difficiles, et ces problèmes font face au fait qu'une petite augmentation de la taille du problème engendre une forte augmentation du temps de calcul, ce qu'on appelle l'explosion combinatoire. Dès le milieu des années soixante dix, le domaine de l'IA s'est retrouvé privé de ses fonds.

Il s'avère effectivement que les difficultés ne sont pas les mêmes pour un ordinateur que pour un humain. Il est en effet plus facile pour une machine de faire un raisonnement logique que de reconnaître un objet dans une image. Une des raisons est que contrairement à l'ordinateur, l'humain a grandi en explorant le monde qui l'environne et a accumulé une quantité incroyable de connaissances à son sujet qu'il ne viendrait à l'idée de personne de remettre en cause, mais qui ne sont pas évidentes pour un ordinateur sans senseur dans le monde réel et sans apprentissage de ce même monde.

L'intelligence artificielle a finalement récupéré ses lettres de noblesses dans le courant des années quatre-vingt avec une évolution des attentes envers l'intelligence artificielle, l'augmentation de la puissance de calcul des ordinateurs, l'utilisation des réseaux de neurones et l'apparition des systèmes experts. Un système expert est un système qui, pour contrer l'explosion combinatoire et limiter l'espace de recherche, utilise des règles précises, dictées par des experts humains. Ces règles, restreignant l'espace de recherche, rendent le solveur spécifique à un domaine, mais extrêmement performant dans celui-ci.

1 Planification

1.1 Historique

La planification est une branche de l'intelligence artificielle où l'ordinateur cherche à trouver une combinaison d'actions permettant d'atteindre un but à partir d'une situation initiale connue. Elle a aussi connu un premier départ après la conférence de Dartmouth, avec des algorithmes comme Dijkstra (1959) ou A* (1968). L'algorithme STRIPS ainsi que le formalisme associé naissent aussi à cette période (1971). Il est en effet le premier à proposer un formalisme concis pour la planification, décrivant l'état du monde à l'aide uniquement des prédicats vrais, considérant que tout ce qui est non exprimé est faux, permettant ainsi d'économiser de l'espace mémoire, c'est ce qu'on appelle l'hypothèse du monde clos. C'est pourtant ce manque d'espace mémoire qui a fini par avoir raison de la planification au milieu des années soixante-dix.

Il a ensuite fallu attendre que la loi de Moore fasse son effet et que les ordinateurs gagnent en RAM pour que les algorithmes de planification soient en mesure de résoudre des problèmes intéressants et que le domaine reparte. Les premiers planificateurs raisonnaient dans l'espace des plans et restaient limités en performances. L'arrivée de GraphPlan en 1995 a changé la donne, poussant la communauté à se pencher sur la planification dans l'espace d'états et atteindre ainsi de bonnes performances. L'année 1998 a aussi vu naître le langage PDDL [McDermott 1998],

langage standardisé pour la planification basé sur une évolution de STRIPS, qui a permis d'organiser des compétitions de planification (IPC) dès l'année 2000 et ainsi motiver la communauté à poursuivre ses travaux.

Longtemps confinée aux cas déterministes, totalement observables et à variables discrètes, la planification s'est aussi ouverte aux situations plus complexes mais plus réalistes :

- les MDP, où le résultat des actions est soumis à une loi stochastique ;
- les POMDP, extension des MDP où l'état ne peut être observé avec une précision absolue ou dans son ensemble ;
- à la planification non déterministe (sans pour autant avoir de loi stochastique sous-jacente) ;
- à la planification temporelle, avec des actions durative et/ou concurrentes ;
- à la planification multi-agents ;
- à la planification en milieux continus ;
- ...

1.2 Terminologie

Plus formellement, la planification automatique vise à produire des *plans* (séquences d'actions) ou des *politiques* (association état courant - action à effectuer, utilisées dans le cadre de la planification non déterministe) pour la résolution d'un problème. Le planificateur part d'un *état initial*, plus ou moins bien connu suivant les cas et tente de trouver une stratégie (plan ou politique) permettant d'atteindre un *but* ou de minimiser un coût sur le long terme.
Un *état* est la situation du monde à un instant donné et est représenté par un ensemble de variables nommées *variables d'état*.
Un *but* est représenté par une formule booléenne reposant sur les variables d'état.
Une *action* est une primitive directement réalisable modifiant l'état du monde, ayant des préconditions (sous forme de formule logique) et des effets, représentés par les modifications des variables d'état.

1.3 Applications

On retrouve la planification dans un grand nombre de domaines à l'heure actuelle. L'application la plus évidente est la planification de trajectoire, au sein des assistants de conduite équipés d'un GPS, ou dans les jeux vidéo par exemple. On retrouve aussi la planification dans des applications industrielles de gestion de projets et de gestion de stocks. Enfin, à titre plus expérimental, elle est utilisée pour des missions robotiques dans le cas où les engins ont besoin d'une grande autonomie due à une difficulté de communication, dans le cas des missions spatiales ou sous-marines. Enfin, on peut trouver des essais pour appliquer la planification en gestion de crise.

Au DCSD (Département de Commande des Système et Dynamique du vol), département de l'ONERA dans lequel j'ai effectué ma thèse, de nombreuses problématiques sont étudiées, pour lesquelles la planification est une brique essentielle. Nous

pouvons citer par exemple le projet AGATA [AGATA 2005], visant à augmenter l'autonomie des engins spatiaux via une architecture adaptée et de la planification embarquée, ou le PEA Action [PEA Action 2007] qui vise à faire de la localisation automatique d'intrusion, de mines et de nappes de pollution à partir de flottilles d'agents autonomes hétérogènes (hélicoptères, véhicules terrestres, bâtiments de surface et sous-marins) dans des environnements mal connus et sans GPS.

2 Motivation

Nos travaux portent sur le domaine de la planification, et sont inspirés des systèmes experts. En effet, les planificateurs actuels, bien que performants sur des benchmarks [1], restent limités quand il s'agit de résoudre un problème réel, trop complexe en nombre de variables d'état et d'actions possibles. En effet, l'algorithme de planification n'ayant aucun sens pratique ni aucune expérience de la vie réelle contrairement aux êtres humains, celui-ci ne sait pas mettre de côté les solutions évidemment contre-productives. Ceci fait que le planificateur doit explorer un nombre très conséquent de combinaisons d'actions éventuellement inutiles et d'états éventuellement non pertinents avant d'atteindre le but.

On pourrait comparer cette façon d'explorer les stratégies à un robot, qui cherchant à réaliser une mission complexe, commencerait par essayer les différentes actions qui s'offrent à lui, telles que le dépliement d'un bras, un changement de cap, ou de vitesse, sans avoir au préalable étudié de stratégie globale pour sa mission. Ces modifications trop locales de son état ne permettant pas de mesurer s'il s'est rapproché ou éloigné du but en les effectuant, il se retrouve avec un espace de recherche bien trop grand et mal structuré pour pouvoir être convenablement exploré.

C'est en se basant sur ce constat, et en s'inspirant des solutions existantes à l'heure actuelle pour tenter de pallier à ce problème que nous avons développé les diverses idées fondatrices de cette thèse. Notre objectif principal est de mettre au point un formalisme et un algorithme de planification qui permettent une bonne aggrégation de la connaissance experte afin d'accélérer la résolution des problèmes actuels et de rendre accessible le travail sur des problèmes réels.

Bien que nos travaux soient prévus pour s'appliquer dans le cadre général de la planification d'actions de type STRIPS, nous nous appuierons tout du long de ce manuscrit sur un exemple simple de planification de trajectoire présenté en figure 1 : un robot au point `A` doit rejoindre le point `L` par le chemin le plus court en ayant une parfaite connaissance au préalable du terrain. Un obstacle (entre `G` et `H`) oblige à choisir un contournement par `E, G, I` ou par `F, H, J, K`.

2.1 La planification hiérarchique

La planification hiérarchique est un des outils permettant de forcer le planificateur à restreindre la recherche aux actions pertinentes en ne l'autorisant à appliquer qu'une

1. Les benchmarks sont des problèmes standardisés créés dans le but de tester et comparer les différents planificateurs. Ils sont rarement associés à des problèmes concrets.

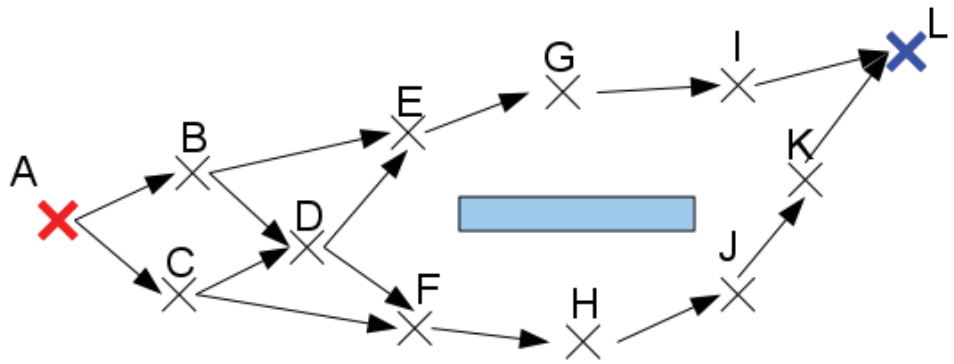


FIGURE 1 – Exemple de graphe. L'objectif est de rejoindre `L` depuis `A`.

sous-partie limitée des actions dans chaque situation. Pour ce faire, la planification hiérarchique permet au planificateur de planifier avec des tâches de haut niveau, abstraites. Ces tâches sont des abstractions d'actions permettant de générer des effets intéressants à faible coût combinatoire en sautant des étapes de bas niveau. Le planificateur peut ensuite les détailler en sous-tâches plus précises, tout en sachant ce qu'il est en train d'accomplir, et ainsi de suite jusqu'à tomber sur les actions élémentaires. Ces tâches peuvent être produites par des algorithmes d'inférence automatique, ou données par un expert.

Dans l'exemple présenté en figure 2, le planificateur peut utiliser des sauts, ou déplacements de haut niveau, afin de faire un choix quant à passer au Nord ou au Sud de l'obstacle. Il faudra ensuite au planificateur, une fois qu'il aura choisi d'insérer un saut, `A-G` par exemple, détailler le chemin nécessaire pour effectuer ce saut (`A-B-E-G`).

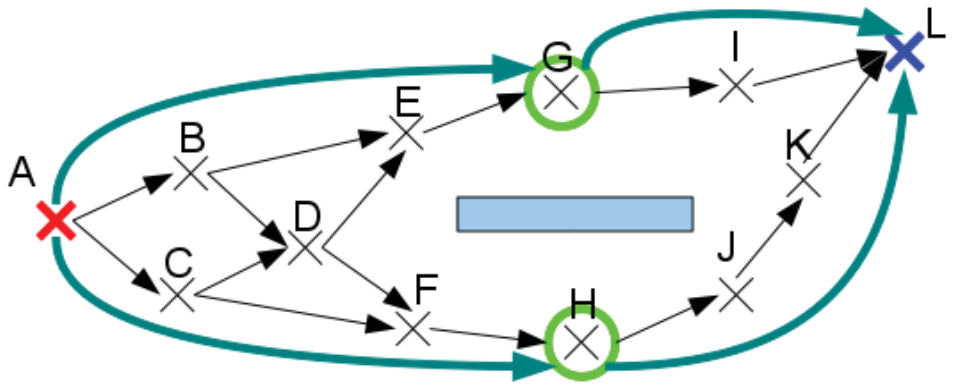


FIGURE 2 – Exemple de graphe avec déplacements haut niveau (en vert).

L'intérêt d'une telle démarche est d'éviter d'explorer les détails du plan dès le départ sans savoir à l'avance de quel côté on compte passer. En effet, une fois cette décision prise, planifier un chemin est très facile, le nombre de décisions restant à

prendre est minime. On peut ainsi vite voir si le choix n'a pas été le bon et reprendre la planification à haut niveau en en tenant compte. Alors que si on n'a aucune idée de la suite du chemin, il faut explorer tous les détails de chacune des directions de départ possibles avant de se rendre compte que la plupart d'entre elles ne mènent à rien.

Une analogie peut être faite avec le calcul de trajet en voiture. On cherche généralement d'abord à savoir par quelle(s) autoroute(s) on va passer avant de décider si en sortant de chez soi on va prendre à droite ou à gauche à la première intersection. En effet, si on commençait à planifier d'abord les détails du départ, on pourrait vouloir partir dans la route la plus axée vers le but final, alors qu'il suffirait de faire cinq cent mètres dans la direction opposée pour arriver à l'entrée de l'autoroute qui nous amènera quasi-directement à l'endroit souhaité. C'est ce type d'algorithme, basé sur des grands axes, qui est implanté dans les assistants de navigation automobiles utilisant le GPS [Flinsenberg 2004].

2.2 La planification hiérarchique avec expertise

Alors que des systèmes automatiques sont en mesure de construire une hiérarchie de bonne qualité dans des cas simples, notamment pour la planification de trajectoire [Botea 2004], le cas général s'avère bien plus difficile. Des systèmes parviennent à construire quelques tâches de niveau intermédiaire de façon automatique, mais sans atteindre l'efficacité d'une bonne hiérarchie. En effet, les experts humains ont une bonne vue d'ensemble du problème et une bonne capacité d'abstraction. Il est même souvent assez facile pour un humain de trouver des stratégies intéressantes face à un type de problème donné, ou de pouvoir déterminer une classe d'actions pertinentes dans une situation donnée. Cette information peut alors être fournie au planificateur afin de retreindre fortement le facteur de branchement.

Dans le cadre de la planification hiérarchique, l'expertise humaine permet généralement d'écrire les tâches abstraites pertinentes pour le domaine, ainsi que leurs décompositions possibles en sous-tâches. Cette décomposition (appelée *méthode*), souvent bien plus pertinente, complète et générique que celle qui pourrait être faite automatiquement par un ordinateur, permet ainsi au planificateur d'être très efficace dans la résolution des problèmes. C'est ce que l'on peut rencontrer par exemple dans le formalisme HTN qui sera présenté par la suite.

Si on garde pour exemple notre problème de planification de trajectoire, un opérateur humain pourra très facilement créer une tâche de haut niveau représentant l'intégralité de la mission, nommée `moveTo`, puis indiquer que cette tâche se fait à l'aide de sauts de longueur moyenne vers des points de passage obligatoires (`G` ou `H` dans notre cas), appelés `jumpTo` et enfin préciser que ces sauts se raffinent en une série de déplacements élémentaires dans le graphe, baptisés `goTo`. Cette décomposition est visible en figure 3.

Cette décomposition présente l'avantage d'être générique, c'est à dire adaptée à tout parcourt de graphe où l'on peut extraire des points de passage obligatoires.

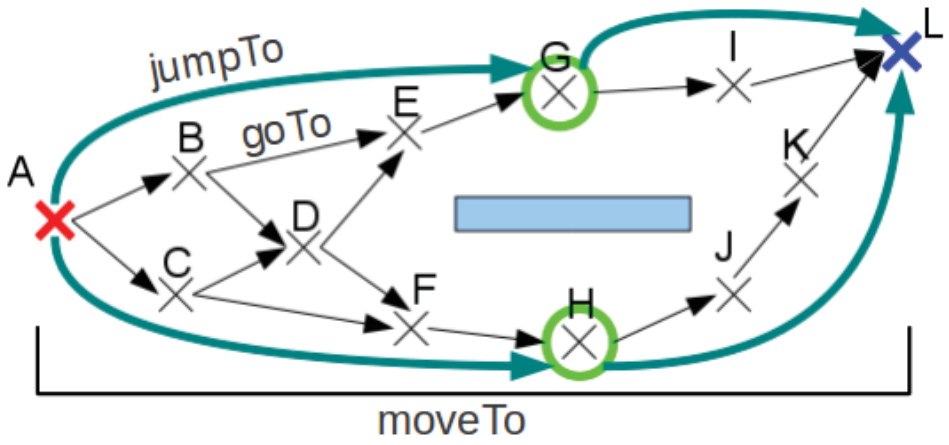


FIGURE 3 – Exemple de référence : décomposition hiérarchique d'un graphe.

2.3 Préparer hiérarchiquement une stratégie

Nos travaux se basent sur le constat que actuellement (et à notre connaissance), les planificateurs hiérarchiques ne sont pas en mesure de conserver une certaine indépendance entre les niveaux de hiérarchie, c'est à dire ne sont pas capable de mener une planification à haut niveau sans devoir raffiner les tâches. En effet, leurs méthodes de résolution consistent à développer une tâche en sous-tâches dès qu'elle est insérée dans le plan, et non de continuer la résolution à haut niveau hiérarchique avant de raffiner le plan. Cette particularité leur permet de garder un formalisme concis, mais peut se payer en termes de performance. En effet, si la tâche de haut niveau insérée à un instant s'avère être un cul-de-sac, le planificateur explore tout de même les plus bas niveaux de cette tâche avant de remonter et se rendre compte qu'il ne peut insérer d'autre tâche à la suite de celle-ci. Une analyse à haut niveau aurait pourtant peut-être permi de se rendre compte que l'insertion de cette tâche ne permettait pas d'atteindre le but. On aurait ainsi pu épargner au planificateur l'exploration d'une voie non prometteuse.

2.3.1 Cadres actuels

En effet, les formalismes actuels de modélisation hiérarchique modélisent des tâches comme étant un ensemble d'une précondition et de plusieurs recettes pour l'accomplir, sans pour autant exprimer l'état atteint par cette tâche. Ce qui fait que le planificateur doit la raffiner pour obtenir cette information et insérer d'autres tâches dans le plan. De plus, les recettes sont exprimées grâce à une combinaison d'opérateurs de séquence et de parallélisme, qui ne permettent pas d'écrire des décompositions avancées sans avoir fortement recours à la récursivité. Dans un tel cadre, la planification par niveaux hiérarchiques distincts est non seulement rendue impossible par le fait qu'on ne puisse prévoir l'état futur à haut niveau, mais aussi non productive par le fait que, à cause de la récursivité nécessaire à la construction

d'un modèle, les plans de granularité intermédiaire se retrouvent contenir des tâches de niveaux d'abstraction très disparates.

Si l'on reprend notre exemple précédent, dans les formalismes actuels, il est possible d'exprimer que l'on choisit la tâche `jumpTo G`, sans pour autant savoir qu'appliquer cette tâche nous amènera au point `G`. On ne peut donc continuer à développer le plan avec des tâches sans avoir entièrement précisé le développement de `jumpTo G` et obtenu qu'elle nous fait arriver en `G`.

De plus, en utilisant la récursivité, on exprime un `moveTo ?l` comme un `jumpTo ?l`$_1$ suivi d'un `moveTo ?l` pour terminer le mouvement, et un `jumpTo ?l` comme un `goTo ?l`$_1$ suivi d'un `jumpTo ?l`. On obtient alors les plans suivants pour les différents niveaux hiérarchiques :

- niveau 0 : `moveTo L`
- niveau 1 : `jumpTo G ; moveTo L`
- niveau 2 : `goTo B ; jumpTo G ; moveTo L`
- niveau 3 : `goTo B ; goTo E ; jumpTo G ; moveTo L`
- niveau 4 : `goTo B ; goTo E ; goTo G ; moveTo L`
- niveau 5 : `goTo B ; goTo E ; goTo G ; jumpTo L`
- niveau 6 : `goTo B ; goTo E ; goTo G ; goTo I ; jumpTo L`
- niveau 7 : `goTo B ; goTo E ; goTo G ; goTo I ; goTo L`

on se rend alors bien compte qu'au sein de presque chaque niveau, on retrouve des actions de différents niveaux d'abstraction, ce qui n'est pas souhaitable, non seulement pour une question de lisibilité de chacun des plans mais aussi pour l'intégration de ces plans hiérarchiques dans un superviseur de même type.

À notre connaissance, peu d'algorithmes dérogent à cette règle.
Les premiers cadres de planification hiérarchique, tels que Nonlin [Tate 1976], résolvaient en effet les problèmes par niveaux successifs, mais étaient basés sur des processus d'ordonnancement, et donc ne pouvaient effectuer que des optimisations temporelles au sein d'un niveau. De plus, aucun mécanisme de backtrack n'était mis en place afin de tenir compte des difficultés rencontrées à bas niveau pour reprendre une planification de haut niveau.
Dans les formalismes actuels, le cadre des HLA (High Level Actions) [Marthi 2008] apporte lui aussi un début de réponse en effectuant une planification complète au plus haut niveau avant d'employer un algorithme en profondeur d'abord pour raffiner ce plan. Ce planificateur ne peut donc pas gérer plusieurs niveaux de hiérarchie distincts, chacun avec un fort niveau de cohérence. De plus, ses effets de haut niveau, bien que très informants, utilisent un formalisme et une expressivité qui leur est propre et qui ne leur permet pas d'être utilisés par un planificateur classique comme nous souhaitons le faire (voir la section 2.2 pour plus de détails).

2.3.2 Propositions

Nous proposons donc d'étendre les formalismes actuels de planification hiérarchique afin de permettre à l'expert de donner des informations supplémentaires quant à l'effet escompté d'une tâche haut niveau d'une part, et en s'inspirant de la

programmation structurée [2] pour permettre une plus grande liberté dans l'expression des méthodes d'autre part.

Il serait alors possible, pour le planificateur, de connaître l'état d'arrivée d'une tâche et donc de continuer à planifier à un niveau d'abstraction donné, en testant la validité des préconditions des tâches suivantes. Il serait aussi possible de détecter l'avancement de l'état et l'arrivée dans un état but ainsi que de calculer une heuristique.

De plus, l'expression des méthodes par programmation structurée permettra de définir l'équivalent de boucles, évitant ainsi la récursivité pour les sous-tâches répétitives. En effet, pour définir une méthode qui décompose une tâche en une répétition d'une sous routine donnée, la méthode actuelle est de dire qu'elle s'effectue en combinant une fois la sous-routine, puis une fois la tâche (ici, `jumpTo` se décompose en `goTo` puis à nouveau `jumpTo`). L'introduction de la programmation structurée permettra de dire qu'une tâche s'effectue effectivement en répétant la routine (donc pour nous, un `jumpTo` se décompose en un nombre inconnu (que le planificateur doit déterminer) de `goTo`).

À partir de ce formalisme, nous allons proposer un nouvel algorithme de planification hiérarchique. S'appuyant sur les acquis précédents, il permettra une résolution plus rapide des problèmes avec des choix de tâches motivés et un faible facteur de branchement à bas niveau. Grâce à la connaissance de l'état d'arrivée d'une tâche, nous allons planifier directement à haut niveau de hiérarchie et ne descendre en finesse qu'une fois la stratégie de haut niveau bien établie. La possibilité d'écrire les méthodes de façon structurée aidera à conserver une cohérence hiérarchique au sein d'un plan d'un niveau donné.

Ainsi, dans notre exemple, le planificateur pourra calculer comme stratégie de passer par le Nord, et donc d'effectuer le chemin `A-G-L`. Une fois cette stratégie déterminée, il pourra la raffiner en planifiant les sous-chemins, donc `A-B-E-G` et `G-I-L`.

La planification de haut niveau se basant sur des effets escomptés et non sur des effets réels qui ne sont pas connus au préalable, il se peut que la phase de raffinement ne donne pas les résultats prévus. Si par exemple le point `I` se trouvait très décalé et que l'arc `G-I-L`, seul chemin possible pour faire le trajet `G-L`, se trouve bien plus long que sa longueur à vol d'oiseau, la stratégie de passer par `G` ne serait plus la meilleure, bien que ça ne soit pas prévisible quand on ne regarde le problème qu'à haut niveau (sauf à planifier entièrement à bas niveau au préalable pour avoir les effets exacts).

Le planificateur sera donc en mesure de faire remonter les difficultés à plus haut niveau de façon à recalculer une stratégie en tenant compte. Ici, planifier de passer par `H` au lieu de `G`. Le planificateur calculera ensuite la stratégie à bas niveau correspondant, et si aucune nouvelle difficulté n'est apparue, déclarera le plan comme valide.

2. programmation structurée : paradigme de programmation actuel consistant à utiliser des boucles plutôt que des goto et de la récursivité.

Cet algorithme est nommé HDS pour "Hierarchical Deepening Search".

3 Plan du manuscrit

Dans ce manuscrit, nous commencerons par présenter dans le chapitre 1 le cadre de la planification classique ainsi que les algorithmes associés qui nous serviront de base par la suite. Nous présenterons aussi diverses branches de la planification qui se relient à nos travaux et le cadre HTN ainsi que l'algorithme SHOP2 qui seront le point de départ et de comparaison de nos extensions et apports.

Nous détaillerons ensuite dans le chapitre 2 les apports que nous avons fait au formalisme HTN. Apports qui nous permettront d'une part de mettre en œuvre notre idée, mais aussi de faciliter l'écriture des domaines de planification et la compilation d'expertise humaine.

Une troisième partie (chapitre 3) sera consacrée à l'algorithme que nous avons développé dans cette thèse, d'une part pour pouvoir produire un plan à une granularité d'expression donnée, et d'autre part pour pouvoir faire interagir ces différents plans pour construire et optimiser le plan hiérarchique final.

Nous présenterons enfin dans le chapitre 4 nos choix d'implémentation, les gains de performances liés aux différents apports, ainsi qu'une comparaison avec le planificateur de référence en planification optimale, hiérarchique avec expertise humaine, SHOP2.

Chapitre 1

Contexte théorique et formel

Sommaire

1.1 La planification classique 11
1.1.1 Formalisation et PDDL 11
1.1.2 Exemple de formalisation de problème 13
1.1.3 Principaux algorithmes 14
1.2 Autres formes de planification 18
1.2.1 Planification temporelle 19
1.2.2 Planification probabiliste 19
1.2.3 Planification multi-agent probabiliste 21
1.2.4 Expertise humaine 22
1.3 Les structures hiérarchiques automatiques 23
1.3.1 En planification de trajectoire 23
1.3.2 En planification classique 24
1.4 Formalismes de connaissance procédurale non retenus . . . 25
1.4.1 Nonlin et O-Plan 25
1.4.2 High Level Actions 26
1.5 Les HTN, un cadre privilégié pour la planification hiérarchique 27
1.5.1 Intuition 27
1.5.2 Formalisme des HTN 29
1.5.3 Exemple 31
1.5.4 Algorithme SHOP2 33
1.5.5 Les HTN probabilistes 34
1.6 Synthèse 36

1.1 La planification classique

1.1.1 Formalisation et PDDL

Le but de la planification dite classique est de calculer une stratégie, appelée *plan* pour atteindre un *but* avec la connaissance exacte des *actions* applicables et de leurs *effets* dans un monde parfaitement connu.

Un problème de planification classique est un triplet :

$$P = (s_0, g, A)$$

où s_0 est l'état initial du monde ; g le but à atteindre, défini comme une formule logique ; et A un ensemble d'actions.
L'état initial ainsi que tous les autres états du monde sont représentés par un ensemble de littéraux L décrivant le monde. Ces littéraux sont des fonctions de la logique L1 [Stolyar 1970] où l'on peut définir des objets et des fonctions sur ces objets dont la valeur sera `vrai` ou `faux`. La représentation utilisée dans les langages de planification actuels comme PDDL est de ne stocker que les prédicats et fonctions dont la valeur est `vrai`, et considérer tous ceux non exprimés comme étant faux.
Le but s'exprime généralement sous la forme d'une formule logique quantifiée. On peut ainsi utiliser les opérateur `pour tout` et `il existe`. Certaines fois, le but pourra être écrit plus simplement comme une liste de prédicats qui devront être vérifiés dans l'état final.

Chaque action a est un quadruplet :

$$a = (\text{name}(a), \text{precond}(a), \text{effects}(a), \text{cost}(a))$$

où $\text{name}(a)$ est le nom de l'action ; $\text{precond}(a)$ la précondition requise sur l'état courant pour pouvoir appliquer a ; $\text{effects}(a)$ les modifications faites sur l'état du monde lorsqu'on applique a ; et $\text{cost}(a)$ (paramètre optionnel) le coût d'exécution de l'action.
Une précondition, de la même façon qu'un but, est décrite par une formule de logique quantifiée, qui doit prendre la valeur `vrai` dans l'état courant pour que l'action puisse être appliquée.
Un effet, dans son expression la plus simple, est l'ensemble d'une liste des prédicats qui deviennent vrais (ajouts) et d'une liste des prédicats qui deviennent faux (retraits). Mais de plus en plus, les planificateurs acceptent des formulations conditionnelles qui permettent de gagner en expressivité des effets.

Un plan π est une séquence d'actions, dont le coût est la somme des coûts de ses actions, ou le nombre d'actions utilisées si la fonction de coût n'est pas définie. π est une solution au problème si en appliquant toutes ses actions depuis s_0, l'état final obtenu vérifie g. π est une solution optimale au problème s'il est plan solution et s'il n'existe pas d'autre plan π' solution dont le coût soit inférieur à celui de π.

Pour décrire les problèmes de planification, le langage PDDL [McDermott 1998] est couramment utilisé. Ce langage a été développé pour la compétition de planification IPC, maintenant attaché à la conférence ICAPS. Il est basé sur le formalisme de STRIPS [Fikes 1971] et s'exprime à l'aide de la logique propositionnelle quantifiée. Un problème de planification se décompose en deux parties, un fichier *domaine*, indépendant de l'instance du problème, contenant l'ensemble des actions A permises dans le monde, ainsi que leurs effets, et un fichier de *problème*, spécifique à chaque instance, contenant la liste des objets, l'état initial du monde s_0 (c'est à dire la liste des prédicats vrais) et la définition du but g.

Pour distinguer un objet (constante du problème) d'une variable, ces dernières sont précédées d'un '`?`'. Ainsi, une variable `?v` peut représenter un véhicule, et prendre la valeur `veh1` s'il fait partie des objets du problème.

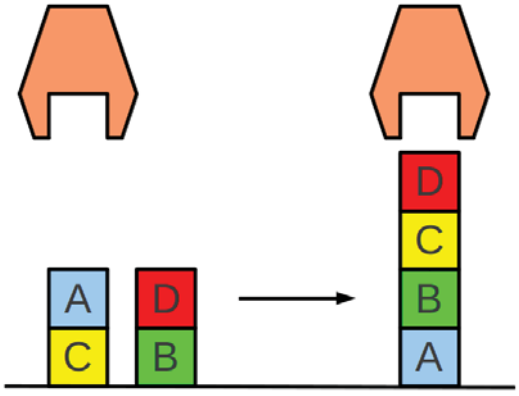


FIGURE 1.1 – Problème de Blocks World, avec l'état initial à gauche et le but à droite.

La notation utilisée est la notation préfixe, c'est à dire avec le nom du prédicat suivi de ses paramètres, le tout entre parenthèses. Par exemple, le `et` logique entre trois prédicats sera noté : `(and (isReady ?v1) (isReady ?v2) (isReady ?v3))`. Un prédicat est dit instancié si ses variables sont remplacées par des objets du problème.

On trouve de nombreuses extensions à PDDL, dont des extensions pour la définition des problèmes temporels [Fox 2003], ou pour l'expression de problèmes probabilistes [Younes 2004].
Ce formalisme est le plus utilisé dans la communauté académique de planification, notamment avec la compétition de planification ICAPS. Il existe cependant de nombreux autres langages de planification, présentés dans [Ghallab 2004].

1.1.2 Exemple de formalisation de problème

Nous allons présenter ici un exemple de couple domaine/problème formalisé en PDDL. Nous prendrons pour cela l'exemple bien connu du domaine Blocks World. Dans ce domaine, qui est un benchmark de la compétition de planification IPC, le but est de déplacer des cubes à l'aide d'une pince robotique, en minimisant le nombre d'actions.

Dans le domaine de Blocks World, il y a deux types d'objets. Les pinces (`crane`), et les caisses (`blocks`). Cinq types de prédicats sont utilisés :

- `on ?b1 ?b2 - blocks`, qui définit que la caisse `?b2` est sur la caisse `?b1` ;
- `on_table ?b - block`, qui définit que la caisse `?b` est posée directement sur la table ;
- `clear ?b - block`, qui définit que la caisse `?b` est en sommet de pile ;
- `holds ?b - block ?c - crane`, qui définit que la pince `?c` tient la caisse `?b` ;
- `empty ?c - crane`, qui définit que la pince `?c` est vide.

Quatre actions sont alors possibles :

- `grab_on_table`, qui permet à une pince d'attraper une caisse qui est sur la table et en dessus de pile avec une pince inutilisée ;

- `grab_on_stack`, qui permet d'attraper une caisse qui est en dessus de pile et posée sur une autre ;
- `put_on_table`, qui permet de poser sur la table une caisse tenue par une pince ;
- `put_on_stack`, qui permet de poser sur une caisse libre une caisse tenue par une pince.

La figure 1.1 montre la situation initiale ainsi que le but du problème, qui sont décrits formellement dans le fichier de problème. La totalité du domaine et un exemple de problème écrits en PDDL sont présentés en annexe A.1.

1.1.3 Principaux algorithmes

Nous présenterons ici les principaux algorithmes de planification classique. Ces algorithmes sont les bases sur lesquelles est construite la grande majorité des algorithmes de planification actuels.

1.1.3.1 Dijkstra

L'algorithme Dijkstra (cf. algorithme 1) a été publié en 1959 [Dijkstra 1959], et est destiné à la base à la planification de trajectoire. Il présente le fondement même de la planification *en avant*. Le principe consiste à stocker pour chacun des états le coût minimum nécessaire pour les atteindre depuis l'état initial, jusqu'à ce que le but soit atteint. On appellera un nœud l'ensemble constitué de l'état et du coût minimum constaté. Cet algorithme nécessite de travailler sur un problème dit "monotone", c'est à dire où il y a soit des coûts, soit des gains, mais pas les deux mélangés.

Pour ce faire, on utilise deux ensembles, l'ensemble des ouverts, c'est à dire l'ensemble des nœuds atteints mais non encore développés et l'ensemble des nœuds développés (ou l'ensemble des fermés). Au début, l'ensemble des états développés est vide, et l'ensemble des ouverts ne contient que l'état initial (lignes 2 à 5).

À chaque étape, l'algorithme sélectionne parmi les ouverts le nœud qui a le coût minimum pour l'atteindre (ligne 8). Il est démontrable, dans le cas où le problème est monotone, que le nœud ouvert choisi a comme coût le coût optimal pour l'atteindre depuis l'état initial (et il en est de même pour tous les nœuds fermés). Il étend ensuite ce nœud en lui appliquant les différentes actions autorisées. Les états obtenus sont traités de la manière suivante :

- *l'état obtenu appartient à l'ensemble des fermés :* le coût pour atteindre cet état en passant pas le nœud en cours de développement est nécessairement plus grand que celui déjà trouvé, on ne fait rien (ligne 15),
- *l'état obtenu appartient à l'ensemble des ouverts :* si le nouveau coût est inférieur au meilleur coût déjà trouvé, on le met à jour, sinon on ne fait rien (ligne 17),
- *l'état obtenu n'a pas encore été atteint :* on assigne comme meilleur coût à cet état le coût nécessaire pour atteindre le nœud en cours plus le coût de la dernière action, et on place cet état dans l'ensemble des ouverts (lignes 19 à 21).

On retire ensuite le nœud tout juste développé de l'ensemble des ouverts pour le mettre dans l'ensemble des nœuds développés.

On répète ensuite cette opération jusqu'à ce que le nœud choisi pour être développé soit le nœud but, auquel cas on a trouvé une solution, ou jusqu'à ce que l'ensemble des ouverts soit vide, auquel cas, le problème n'a pas de solution et le but n'est pas atteignable.

Algorithm 1: Dijkstra planner:

```
1  begin  dijkstra(init_st, goal)
2      n_0.st ← init_st;
3      n_0.cost ← 0;
4      O ← {n_0};
5      C ← ∅;
6      plan ← ∅;
7      while O ≠ ∅ do
8          n ← arg min_{n'∈O} n'.cost;
9          if n.st = goal then
10             plan ← extractPlan(n);
11             break;
12         forall the a ∈ A_st do
13             next ← apply(a,n.st);
14             if ∃n'∈ C\(n'.st = next) then
15                 NOP;
16             else if ∃n'∈ O\(n'.st = next) then
17                 n'.cost ← min(n'.cost, n.cost + a.cost);
18             else
19                 n'.st ← next;
20                 n'.cost ← n.cost + a.cost;
21                 O ← O ∪ {n'};
22         C ← C ∪ {n};
23         O ← O\n;
24     return plan;
```

Il suffit alors, à partir du but, de revenir vers l'état initial en prenant à chaque fois le lien vers l'état de plus faible coût, pour extraire le plan optimal qui a servi à l'atteindre.

Définition 1.1 (*Correction*)

*La propriété de correction (*correctness *en anglais) pour un planificateur traduit le fait que le plan retourné par ce planificateur, lorsque celui-ci en retourne un, mène bien au but, sans violer de précondition.*

Définition 1.2 (*Complétion*)

La propriété de complétion pour un planificateur définit le fait que s'il existe un plan solution au problème, alors le planificateur pourra le trouver.

Définition 1.3 (*Optimalité*)

La propriété d'optimalité définit le fait qu'aucun plan solution ne peut être strictement moins coûteux que le plan retourné.

Définition 1.4 (*Terminaison*)

La propriété de terminaison définit que le planificateur doit déterminer en un temps fini si le problème a une solution ou pas.

Théorème 1.1

L'algorithme de Dijkstra est correct et termine en un temps fini dans tous les cas. Dans le cas où l'algorithme est appliqué sur un problème monotone, l'algorithme est de plus prouvé complet et optimal.

Bien que cet algorithme soit initialement destiné à la planification de trajectoire, il est utilisable tel quel pour faire de la planification classique, en remplaçant la position par un état et les distances entre les points par des coûts. Il est par contre très lent par le fait qu'aucun guide ne lui permet de savoir dans quelle direction explorer pour se rapprocher du but et doit donc parcourir un grand nombre d'états avant de le trouver.

1.1.3.2 A*

L'algorithme A* (cf. algorithme 2) a été inventé en 1968 [Hart 1968]. Il est lui aussi initialement destiné à la planification de trajectoire, mais s'utilise facilement aussi en planification classique. Cet algorithme s'appuie sur une connaissance de l'état but, et sur une estimation de la "distance" au but. Ainsi, un état avec un coût élevé, mais estimé très proche du but sera développé prioritairement à un état de coût faible, mais estimé très loin du but.

Cette estimation de la distance est appelée heuristique. Par rapport à l'algorithme de Dijkstra, on rajoute au nœud un attribut d'heuristique, `h` (ligne 4), et on calcule cette valeur à chaque fois qu'on ouvre un nouveau nœud (ligne 22). Cette fois-ci, plutôt que de développer le nœud avec le plus petit coût dans l'ensemble des ouverts, on développe le nœud le plus prometteur, c'est à dire celui qui a la somme {coût depuis l'origine} plus {coût estimé vers le but} la plus faible (ligne 9).

Définition 1.5

Une heuristique est dite admissible *si et seulement si, dans tous les cas, son estimation du coût restant pour atteindre le but est inférieure ou égale au coût réel. Dans le langage courant, on dira que cette heuristique est optimiste.*

Algorithm 2: A* planner:

```
1  begin  astar(init_st, goal)
2      n_0.st ← init_st;
3      n_0.cost ← 0;
4      n_0.h ← heurist(init_st);
5      O ← {n_0};
6      C ← ∅;
7      plan ← ∅;
8      while O ≠ ∅ do
9          n ← arg min_{n'∈O} n'.cost + n'.h;
10         if n.st = goal then
11             plan ← extractPlan(n);
12             break;
13         forall the a ∈ A_st do
14             next ← apply(a,n.st);
15             if ∃n'∈ C\(n'.st = next) then
16                 NOP;
17             else if ∃n'∈ O\(n'.st = next) then
18                 n'.cost ← min(n'.cost, n.cost + a.cost);
19             else
20                 n'.st ← next;
21                 n'.cost ← n.cost + a.cost;
22                 n'.h ← heurist(n'.st);
23                 O ← O ∪ {n'};
24         C ← C ∪ {n};
25         O ← O\n;
26     return plan;
```

Théorème 1.2

Si l'heuristique est admissible, A a les mêmes propriétés que Dijkstra, c'est à dire les propriétés d'exactitude, de complétion, d'optimalité et de terminaison. Si l'heuristique n'est pas admissible, les propriétés de complétion et d'optimalité sont perdues.*

Si l'heuristique est suffisamment précise, le planificateur pourra gagner quelques ordres de grandeur en terme de performances. Il faut cependant faire attention à la complexité calculatoire de celle-ci, vu que ce calcul est effectué très fréquemment et ce qui peut donc annuler le gain de temps dû au moindre nombre de nœuds développés. Il y a donc généralement un compromis à trouver entre la précision de l'heuristique et sa difficulté calculatoire.

Dans le cas où l'expert ne parvient pas à déterminer une heuristique admissible, il

reste possible de la fixer à zéro, ce qui définit tout de même une heuristique admissible bien que non informante, et ramène cet algorithme à celui de Dijkstra.

L'heuristique la plus fréquemment utilisée en planification de trajectoire est la distance à vol d'oiseau entre le point courant et le but. Dans le cas de la planification classique, les heuristiques sont plus difficiles à déterminer car le domaine est plus complexe et structuré, et demandent généralement une expertise humaine, d'autant plus qu'il faut qu'elles soient *admissibles* pour que l'algorithme ait de bonnes propriétés. Cependant, des algorithmes ont été développés pour générer automatiquement des heuristiques admissibles de façon générique [Haslum 2000, Richter 2008], mais celles-ci sont à utiliser en complément des heuristiques générées par des experts, qui sont souvent plus adaptées tout en étant moins coûteuses à calculer.

1.1.3.3 Graphplan

Le planificateur Graphplan [Blum 1997], spécialisé dans la résolution STRIPS, prend un parti totalement différent. A partir d'un état initial, il construit un graphe des prédicats atteignables en un nombre fixé de coups. Pour ce faire, à chaque étape, il prend l'ensemble des prédicats vrais et lui rajoute les prédicats rendus vrais par les actions applicables avec cet ensemble de prédicats, sans effectuer les retraits de prédicats imposés par ces actions.
Un ensemble de *mutex* est ajouté pour signaler deux prédicats ne pouvant être vrais en même temps à cette étape, ou deux actions ne pouvant être appliquées en même temps à cette étape, pour cause de préconditions incompatibles, par exemple. Ces *mutex* permettent de repousser la date de première apparition d'un prédicat ou de première "applicabilité" d'une action à une date ultérieure plus réaliste. La production de ce graphe jusqu'à l'état but s'effectue en temps polynomial. Il est ensuite utilisé pour extraire le plan des actions qui ont permis d'atteindre le but ; cette extraction se fait en temps exponentiel, mais sur un domaine plus petit que les autres méthodes, permettant ainsi de gagner du temps. Graphplan est un algorithme optimal, complet et exact.

L'algorithme de Graphplan (avant l'extraction du plan) définit d'ailleurs une heuristique admissible pour A* dans le cadre de la planification STRIPS, appelée Hmax [Haslum 2000]. L'idée est de prendre la profondeur de première apparition du but dans le graphe comme heuristique (calculée en temps polynomial), sans faire l'extraction du plan.

1.2 Autres formes de planification

La planification classique, bien que très utilisée et très étudiée suppose des conditions d'utilisation peu adaptées à certaines situations réelles. En effet, il n'est pas rare que les missions que l'on cherche à planifier se réalisent avec plusieurs agents, réalisant donc des actions en parallèle, actions pouvant être duratives et avec des variables continues. Il est, de plus, peu fréquent dans les applications réelles que l'on sache prédire exactement l'évolution du monde, via les actions de l'agent, ou via le

déroulement du temps (incluant les actions d'autres agents non contrôlables). Il est même souvent impossible de connaître avec exactitude l'ensemble du monde dans lequel on évolue. Enfin, que ce soit dans le but de réduire l'espace de recherche ou de contraindre la solution, il peut être intéressant d'utiliser une expertise humaine dans le processus de planification. C'est pourquoi d'autres formalismes et formes de planifications sont étudiées. Nous allons ici en présenter quelques-uns.

1.2.1 Planification temporelle

Alors que l'on pouvait représenter les séquences d'actions en planification classique ainsi que des dates de début et de fin d'actions (et les contraintes associées) grâce à des variables numériques, il n'était pas possible de représenter finement les contraintes de parallélisation entre plusieurs actions. C'est ce manque qu'est venu combler la planification temporelle, introduite de façon formelle avec PDDL 2.1 [Fox 2003] en 2003.

Ce formalisme permet de définir des durées d'action, des pré-conditions, ou expressions devant être vraies au début de l'action, des conditions globales, qui doivent être vérifiées pendant toute la durée de l'action, ainsi que des post-conditions, devant être vérifiées à la fin de l'action. Il permet de plus d'exprimer des effets initiaux en plus des effets finaux classiques, c'est à dire des effets qui sont appliqués lors du commencement de l'action et non lors de se terminaison. Avec ces ajouts, il est alors possible pour le planificateur de savoir si deux actions peuvent être exécutées en parallèle ou pas, et si oui les contraintes de positionnement relatif entre elles.

Les algorithmes de planifications basés sur ce formalisme doivent alors être en mesure de gérer une variable de décision continue : la date de début de chacune des actions. Différentes techniques sont alors possibles. Ces planificateurs sont donc des planificateurs hybrides (gérant des variables discrètes comme continues). LPG [Gerevini 2003], l'un des plus performants d'entre eux, résout ce problème grâce à des techniques de recherche locale. Le planificateur YAHSP [Vidal 2011], basé sur des heuristiques non admissibles permet une résolution nettement plus performante que LPG, mais n'est pas optimal.

1.2.2 Planification probabiliste

1.2.2.1 MDP

Alors qu'en planification classique, l'effet des actions est considéré comme entièrement déterministe et connu, dans le cadre des MDP (ou Markov Decision Process)[Puterman 1994], l'effet des actions est supposé suivre une distribution de probabilité connue. On peut alors définir l'ensemble des états atteignables à partir d'un état donné et en exécutant une action donnée, ainsi que leurs probabilités et coûts (ou récompenses) associés.

Sur de tels problèmes, l'objectif n'est plus de calculer un plan, séquence d'actions à exécuter quoi qu'il arrive, mais plutôt une politique, qui donne dans chaque état atteignable l'action à effectuer. En effet, cela permet de prendre en compte le fait que

le résultat d'une action est incertain, et qu'il faut donc avoir une stratégie capable de s'adapter à cette incertitude. Ces stratégies sont soit calculées sur un horizon fini, auquel cas on cherche à maximiser les gains (ou minimiser les pertes) sur l'ensemble de l'intervalle considéré, soit sur un horizon infini, auquel cas, on cherche à maximiser les gains en favorisant les gains à moyen et court terme par rapport au très long terme à l'aide d'une fonction d'évaporation des coûts et gains (on parle ici de critère γ pondéré), qui évite aussi de faire diverger la fonction de coût.

En planification MDP classique, on travaille généralement avec les hypothèses de stationnarité, ce qui veut dire que les probabilités des transitions ne dépendent pas des actions faites auparavant, et d'homogénéité, c'est à dire que les probabilités de transition sont invariantes en fonction du temps. On travaille aussi avec des formalismes de temps et de variables discrets. Il existe toute fois des formalismes et des algorithmes de résolution de MDP instationnaires, ou inhomogènes, voire à temps continu, mais nous ne nous attarderons pas sur ces points ici.

La résolution d'un MDP repose sur l'équation de Bellman qui exprime que le coût à long terme d'une action dans un état donné est égal à l'espérance mathématique des coûts à long terme des états dans lesquels on peut arriver directement en appliquant cette action plus l'espérance de coût immédiat de cette action. Ainsi, pour calculer une politique optimale, on définit la valeur d'une action donnée comme étant celle obtenue en appliquant l'action la plus prometteuse au regard de cette espérance.

Les algorithmes de base de résolution d'un problème de MDP sont *value iteration* et *policy iteration*.
Value iteration [Bellman 1957] part de l'hypothèse que tous les états sont atteignables. Pour chacun des états, il regarde la meilleure action à effectuer à un coup (pour chaque action, on fait la moyenne pondérée de la valeur des états atteignables ajoutée au coût de l'action), et attribue une nouvelle valeur à l'état en fonction de cette action. Le processus est réitéré jusqu'à obtenir une stabilité dans la valeur des actions. On extrait ensuite la politique grâce à l'équation de Bellman. Cet algorithme, bien qu'étant la base de la planification MDP, est très lent de par la nécessité de développer tous les états.
Policy iteration [Howard 1960], lui, essaye de calculer la valeur de chaque état à partir d'une politique donnée (action à faire déjà fixée pour un état donné), et ensuite applique la fonction de Bellman pour calculer la nouvelle politique. Il s'arrête quand lors d'une mise à jour de la politique, celle-ci reste inchangée. De même que value iteration, cet algorithme nécessite de développer tous les états, mais gagne en performances en ayant (statistiquement) moins d'itérations à effectuer pour atteindre la convergence.

Pour essayer de réduire l'espace de recherche, l'algorithme LAO* [Hansen 2001] utilise une recherche de type A*, transposée aux MDP. Il permet, grâce à une recherche en avant à partir de l'état initial, de restreindre l'espace de recherche à un fuseau autour de la politique optimale et ne développe que les états atteignables par les différentes politiques étudiées. Il a par contre les mêmes contraintes que A* en terme d'heuristiques : elles doivent être admissibles pour garantir les propriétés de complétion et d'optimalité de l'algorithme, et ne sont pas évidentes à construire

dans le cas général.

Pour gagner encore en performances, certains algorithmes, tels que RFF [Teichteil-Königsbuch 2010], simplifient le problème en le déterminisant, utilisant ainsi les avancées de la planification classique pour obtenir une pré-solution, qui servira ensuite de guide pour repasser le problème dans le cadre probabiliste.

1.2.2.2 POMDP

Les POMDP [Kaelbling 1998] poussent l'incertitude et le réalisme un peu plus loin en supposant que l'agent n'a pas une connaissance exacte du monde qui l'entoure. Il n'a accès à l'état que via des variables dites observables. Ces variables peuvent être un sous-ensemble des variables d'état, créant ainsi des variables d'état cachées, ou une imprécision sur la mesure d'une variable, ou enfin une agrégation sous une forme quelconque de plusieurs variables d'état. L'agent ne raisonne alors généralement plus sur un état donné, mais sur un état de croyance (*belief state*) qui consiste en une distribution continue de probabilités sur les états, mis à jour au fur et à mesure avec les règles de Bayes. Celle-ci s'effectue en connaissant l'état de croyance de l'état précédent, l'action choisie, la répartition probabiliste des états de sortie de l'action et la probabilité de l'observation reçue en fonction de l'état.

L'état de croyance étant une distribution de probabilité continue sur l'ensemble des états, les techniques de planification MDP discrètes ne s'appliquent plus, et des méthodes tirées de l'optimisation linéaire entrent en jeu dans le processus de planification.

L'algorithme de base est l'algorithme Witness [Cassandra 1994], qui calcule les points d'optimalité des différentes stratégies en fonction du belief state grâce à la programmation linéaire et l'algorithme du simplex, et utilise un "value iteration" sur ces différents points. Très lourd à cause d'une prise en compte exacte de l'état de croyance, PBVI [Pineau 2003] l'améliore en sélectionnant des points représentatifs de l'état de croyance, ce qui permet de travailler sur un ensemble réduit de valeurs, de ne pas avoir à faire appel à la programmation linéaire, mais avec une perte sur l'optimalité.

1.2.3 Planification multi-agent probabiliste

Les formalismes dec-MDP [Becker 2003] et dec-POMDP [Bernstein 2002] ont pour vocation d'étendre les MDP et POMDP au cas multi-agent. Ainsi, les modifications de l'état du monde ne dépendent plus uniquement d'une action, mais de l'ensemble de toutes les actions effectuées par tous les agents au pas de temps étudié. L'augmentation de la complexité est alors exponentielle avec le nombre d'agents.

Dans les dec-MDP, on considère que chaque agent a sa propre politique mais que celle-ci peut dépendre des états des autres agents qui sont connus, tandis que dans les dec-POMDP, l'état des autres agents n'est pas forcément connu dans son intégralité. Le but du planificateur dans ces cas là est de mettre au point la politique de chacun des agents de façon à maximiser la récompense globale.

Une des techniques envisageables pour résoudre ces problèmes est de les découper en sous-problèmes : des petits dec-MDP (ou dec-POMDP) aux endroits où il y a réellement interaction entre les agents et un MDP (ou POMDP) par agent dans les situations où les agents n'interagissent pas entre eux [Canu 2011].

1.2.4 Expertise humaine

Dans de nombreux problèmes, la planification classique ne suffit plus. Soit parce que le problème est trop grand pour pouvoir être résolu en un temps raisonnable par un ordinateur, soit à cause de contraintes opérationnelles imposées mais difficiles à traduire en PDDL. C'est dans ces situations que l'expertise humaine, basée sur plusieurs années d'apprentissage (processus qui lui-même découle de plusieurs années d'évolution), intervient pour aider ou guider le planificateur.

1.2.4.1 Logique temporelle

Une façon de guider le planificateur est de lui donner un certain nombre de propriétés sur le plan. Ces propriétés, aussi appelées invariants, permettent d'exprimer, globalement, ce qui fait un plan valide, ou un plan performant. Elles peuvent aussi être utilisées pour exprimer une forme d'heuristique, pas nécessairement admissible. Ces contraintes, exprimées en logique temporelle, peuvent porter sur l'intégralité du plan, sur une sous partie du plan, ou sur une action.

Les deux principaux planificateurs utilisant ce principe sont TLplan [Bacchus 2000], se basant sur la logique temporelle LTL pour exprimer ces invariants, et TALplan [Kvanström 2001], se basant sur la logique temporelle TAL. Ces logiques, issues de la logique modale permettent d'exprimer des formules de la forme "cette formule sera vraie au moins une fois dans le futur" ou "cette formule sera vérifiée jusqu'à ce que tel événement se produise".

1.2.4.2 Connaissance de procédure

Une autre façon d'exploiter l'expertise humaine, et celle que l'on utilisera par la suite, est la connaissance de procédure. Il s'agit du cas où un expert connaissant le domaine et le type de problèmes associés est en mesure d'exprimer une stratégie à employer pour le résoudre. Ces stratégies s'expriment comme des procédures ou séquences d'actions à effectuer pour réaliser une tâche, une tâche étant une action abstraite validant un sous-but. Suivant la complexité hiérarchique du problème, ces stratégies peuvent être exprimées sur plusieurs niveaux hiérarchiques, des procédures pouvant être elles-mêmes composées de tâches. Chaque tâche peut avoir plusieurs méthodes de décomposition possibles, à la discrétion du planificateur, ensuite, de choisir la plus pertinente dans la situation en question. L'exemple le plus connu dans le cadre de la connaissance de procédure est les HTN (Hierarchical Task Network) [Erol 1994a], mais on retrouve aussi des idées similaires dans les cadres AHP (Angelic Hierarchical Planning) [Marthi 2008] et BDI (Belief Desire Intention) [de Silva 2009].

1.3 Les structures hiérarchiques automatiques

Les domaines de connaissance de procédure utilisent une représentation hiérarchique pour représenter leurs tâches. En effet les tâches modélisent une vision du problème suivant différents niveaux d'abstraction, du niveau mission pour la tâche de plus haut niveau, au niveau opérationnel quand on se concentre uniquement sur les actions élémentaires. Mais en plus de ces cadres avec expertise, de nombreux algorithmes utilisent des constructions hiérarchiques dans leur résolution de problèmes, hiérarchies construites de façon automatique.

1.3.1 En planification de trajectoire

Un des premiers domaines où a été appliquée la notion de hiérarchie est, encore une fois, la planification de trajectoire. En effet, le domaine du jeu vidéo et le guidage par GPS ont très rapidement nécessité des fonctions de ce type, efficaces, peu gourmandes en ressources, et de bonne qualité.

HPA* L'algorithme HPA* [Botea 2004], pour Hierarchical Path-Finding A*, est un algorithme très largement utilisé par l'industrie du jeu vidéo pour faire de la planification de trajectoire sous-optimale mais efficace. L'idée consiste à grouper automatiquement les positions en petits clusters, puis grouper ceux-ci en clusters de plus en plus grand, et planifier ensuite la trajectoire d'abord sur les régions et affiner cette planification jusqu'à retomber sur les déplacements élémentaires.

La décomposition étant faite à la volée et sans expertise humaine, il n'y a aucune garantie d'optimalité dans la solution trouvée, mais l'erreur se situe, expérimentalement, autour de 1%. L'utilisation de cet algorithme permettant de diviser par dix le temps de calcul, pour des applications réelles, cet algorithme est parfaitement valable.

Pour les quelques cas où une meilleure précision est nécessaire, d'autres méthodes sont mises en œuvre afin d'améliorer la construction de la hiérarchie.

Multi-hiérarchies [Fernandez-Madrigal 2002] La première consiste à se dire que les cas où la construction des clusters n'est pas bonne sont rares, et ne sont dus qu'à des petits écarts par rapport au choix optimal. L'algorithme construit donc à chaque étape, en plus des clusters de HPA*, plusieurs hiérarchies différentes, avec des petites variations autour de la première. Pour chacune, il effectue ensuite les calculs de trajectoire, et lors des fusions de hiérarchies à chaque étape, il ne garde que les meilleurs résultats. Ceci permet, sans trop perdre en performances, de réduire considérablement le nombre de cas où la solution trouvée s'écarte de l'optimal.

Apprentissage par renforcement [Lee 2004, Aguirre 2009] Une seconde stratégie consiste à essayer de faire apprendre au planificateur ce qui fait une bonne décomposition. Cet apprentissage s'effectue avec des techniques issues du domaine

de l'apprentissage par renforcement. Il s'agit ici de construire des hiérarchies, au début de faible qualité, puis de les améliorer au fur et à mesure des planifications les utilisant, en détectant quelles décompositions donnent les meilleurs résultats.

Bien que ces recherches soient intéressantes et nous montrent les avantages de la hiérarchie ainsi que la manière de l'utiliser, elles sont difficiles à mettre en œuvre dans le cadre de la planification classique. En effet, alors qu'il est facile de définir des clusters de positions voisines géographiquement, il est nettement plus difficile de déterminer automatiquement ce qu'est un groupe d'états voisins et de mettre en valeur les variables d'état qui vont avoir une faible influence sur la planification (et donc qui sont intéressantes à regrouper) des autres.

1.3.2 En planification classique

La solution transverse, pour la planification classique, consiste à faire des groupes d'actions au lieu de groupes d'états. C'est le parti pris entre autres par le planificateur Macro-FF [Botea 2005]. L'idée générale est d'essayer de faire des séquences d'actions et d'étudier les préconditions et effets de la macro-action composée de ce groupe. Si la séquence semble avoir un effet intéressant (validation d'un prédicat avec peu de préconditions, par exemple), alors le planificateur crée une *macro-action*, séquence de ces actions, et la rajoute au "pool" des actions disponibles. Des techniques similaires sont employées dans divers planificateurs afin de générer des macro-actions [Coles 2007, Newton 2007].

Dans certains cas, on peut au contraire subdiviser le problème en sous-problèmes plus simples avec un faible niveau de dépendance. Cette technique est appelée technique des sous-buts [Eldracher 1997]. Ces sous-buts peuvent être, suivant les cas, appris avec un réseau de neurones [Eldracher 1993] ou bien trouvés par des techniques d'essais-erreurs [Surynek 2009].

D'autres planificateurs prennent le parti d'essayer de générer automatiquement des "landmarks", c'est à dire des points de passage intéressants. Le planificateur tentera donc de planifier des plans passant par ces points, et des systèmes d'apprentissage et ou de sélection permettent d'affiner ces points de passage jusqu'à obtenir un plan valide [Richter 2008].

Enfin, certains travaux se basent sur une collection de plans valides pour en inférer des structures. Soit des structures hiérarchiques de tâches comme dans HTN-MAKER [Hogg 2008], soit des abstractions sur l'instanciation des variables comme dans λ-graphplan [Martin 2011], permettant ainsi d'inférer une structure condensée d'un plan. Ces structures permettent ensuite une planification bien plus efficace avec les techniques de planification hiérarchique.

Ces recherches en abstraction automatique ouvrent de bonnes perspectives dans la planification et les capacités d'abstraction des machines, mais, à l'heure actuelle, n'atteignent pas les performances d'abstraction et de synthèse d'un être humain. C'est pourquoi nous nous intéresserons par la suite au cas où la structure hiérarchique est construite par un expert humain, même si nos méthodes restent valables pour le

cas général, où l'on disposerait d'une structure, qu'elle soit inférée automatiquement ou construite à la main.

1.4 Formalismes de connaissance procédurale non retenus

Dans le domaine de la connaissance procédurale, de nombreux travaux ont été réalisés. Nous en présenterons ici quelques-uns, importants dans leurs idées, mais non retenus pour nos travaux. Le cadre des HTN sera, lui, présenté en détails dans le chapitre 1.5.

1.4.1 Nonlin et O-Plan

Nonlin [Tate 1976], créé en 1976, est un des cadres précurseurs à la planification hiérarchique avec expertise humaine. Il introduit la notion de tâches, ou actions de haut niveau, qui se raffinent ensuite en sous-tâches. De façon similaire à ce que nous souhaitons faire, Nonlin effectue une planification par niveaux séparés. Une première planification est effectuée au niveau le plus abstrait, puis le plan est raffiné jusqu'à arriver au niveau de granularité le plus fin du modèle.

Nonlin ainsi que son successeur O-Plan [Tate 1995] se basent de plus sur un système de planification non linéaire (d'où le nom du premier). C'est à dire qu'ils ne considèrent pas un plan comme une séquence ordonnée d'actions, mais comme une séquence partiellement ordonnée d'actions, avec des branches pouvant s'effectuer en parallèle les unes des autres, le tout contraint avec des dates au plus tôt et dates au plus tard de démarrage des différentes actions. Les plans produits sont sous la forme d'un diagramme de Gantt [Gantt 1910].

Pour effectuer une telle planification, le formalisme utilisé est le formalisme TF pour *Task Formalism* [Tate 1995]. Il permet de modéliser des actions et des tâches avec des contraintes de précédence, indiquant quelles tâches ou actions doivent être réalisées en amont ; des préconditions (comme en planification classique), sous formes de prédicats devant être vrais ; ainsi que des effets d'actions ou de tâches, indiquant sous forme d'ajouts et de retraits les effets d'une action ou d'une tâche sur l'état courant. Chaque tâche contient de plus une ou plusieurs décompositions en sous-tâches pour la raffiner. Les effets de tâches sont considérés comme exacts, et une méthode de raffinement de tâche ne sera considérée comme correctement déroulée que si les effets produits à bas niveau correspondent à ceux de la tâche développée.

Étant basé sur un algorithme d'ordonnancement et non sur un algorithme de planification classique, l'optimisation s'effectue sur la date au plus tôt de la tâche finale, et ne peut se faire sur le coût total du plan final, contrairement à ce que nous souhaitons faire.

Le processus de raffinement s'effectue en essayant les différentes décompositions possibles des différentes tâches pour trouver un plan non linéaire valide, c'est à dire vérifiant les différentes préconditions des tâches et actions présentes, et où les dates

de début des actions sont bien postérieurs aux dates de fin des actions avec une relation de précédence. Seule une impossibilité de construire un plan ou de réaliser les effets d'une tâche peuvent contraindre le planificateur à backtracker et produire un plan sans la tâche incriminée. Nous pensons au contraire que pour pouvoir effectuer une planification de bonne qualité, il est nécessaire que le planificateur fasse des allers-retours entre les niveaux plus fréquents, afin de détecter non seulement les impossibilités, mais surtout des possibles surcoûts et optimiser le coût final. En effet, Nonlin et O-Plan ne peuvent se rendre compte qu'un plan développé à bas niveau prend plus longtemps que prévu et qu'un autre plan à haut niveau pourrait alors être plus intéressant que celui actuellement utilisé.

Le langage utilisé dans Nonlin, TF, est aujourd'hui dépassé par des langages plus récents basés sur PDDL. En effet, on peut aujourd'hui écrire aisément des coûts sous forme de formules mathématiques ; les effets conditionnels et numériques ont fait leur apparition ; et le langage PDDL et ses dérivés sont aisément extensibles et très largement utilisés dans la communauté de planification. C'est pour cela que, nous préférerons pour la suite nous baser sur un langage de étendu de PDDL (le formalisme HTN), quitte à y réintégrer par la suite certaines des idées présentes dans TF.

1.4.2 High Level Actions

Le cadre des High Level Actions (HLA) [Marthi 2008] est lui assez proche des HTN qui seront présentés par la suite. Il fonctionne sur des tâches de haut niveau se raffinant en séquences de sous-tâches, chacune de ces sous-tâches pouvant être soit une tâche à raffiner, soit une action élémentaire.

Le mécanisme de planification se situe à mi-chemin entre la planification par niveaux que nous souhaitons réaliser et la planification en profondeur d'abord telle qu'elle est faite avec la planification basée sur les HTN. En effet, de même que Nonlin, le planificateur effectue au préalable une planification à haut niveau n'utilisant que les tâches de plus haut niveau. Une fois ce plan obtenu, il le raffine en profondeur d'abord, tâche après tâche, comme cela est fait pour les HTN classiques.

Afin de pouvoir effectuer cette planification de haut niveau, leurs tâches sont affectées d'effets. Ces effets sont formalisés assez particulièrement. Ils contiennent trois sous-effets, le meilleur cas, le pire cas et le cas attendu, ce afin que leur planification de haut niveau ne rende pas le raffinement impossible. Ce cadre ne permettant pas l'utilisation d'algorithmes classiques pour une planification à un niveau donné de hiérarchie pour cause d'effets trop écartés du cadre de la planification classique, nous n'utiliserons pas ce formalisme comme point de départ de notre algorithme.

1.5 Les HTN, un cadre privilégié pour la planification hiérarchique

1.5.1 Intuition

Les réseaux de tâches hiérarchiques ou HTN (pour Hierarchical Task Network) [Erol 1994b], sont un formalisme de modélisation de connaissance procédurale. Basés sur un formalisme proche de STRIPS, on voit apparaître, en plus des actions, la notion de tâches abstraites. Une tâche peut être vue comme une action abstraite, qu'on ne sait pas effectuer directement, mais qui accomplit un but ou un sous-but. Une tâche abstraite contient un ensemble de préconditions nécessaires pour qu'elle puisse être exécutée, ainsi qu'un ensemble de méthodes valides pour la décomposer en sous-tâches. Chaque méthode, soumise elle aussi à précondition, propose une décomposition sous forme d'ordonnancement de sous-tâchesS (pouvant être des actions élémentaires). Nous allons voir par la suite que ce cadre définit un modèle simple pour la planification hiérarchique, et peut donc servir de base pour toute planification de ce type, que ce soit pour exprimer l'expertise humaine (domaine pour lequel il a été conçu), mais aussi pour recevoir les résultats d'algorithmes conçus pour construire une hiérarchie automatiquement, comme ceux que l'on a pu voir précédemment.

On peut en voir un exemple sur la figure 1.2. Cet exemple présente le cas où une personne souhaite se rendre à une conférence (nous ne traiterons pas la partie des événements une fois sur place). Le cadre HTN standard (sans ses extensions, que nous présenterons par la suite) ne permettant pas de gérer le cadre probabiliste, nous ferons l'hypothèse que cette conférence n'a pas de comité de relecture, et qu'un article soumis est automatiquement accepté.

Le HTN de l'exemple présente deux tâches abstraites, représentées dans des rectangles :

- `aller_à_une_conférence` : cette tâche est la tâche de plus haut niveau du problème. Elle symbolise la réalisation de la mission dans sa globalité. Elle prend comme paramètres une conférence `?c` et une personne `?p`, et a deux méthodes, que nous expliciterons par la suite.
- `déplacer_vers` : cette tâche représente un déplacement vers un lieu `?l` d'une personne `?p`, et comporte plusieurs méthodes dont seules quelques unes sont représentées ici.

Il comporte aussi plusieurs actions élémentaires, qui sont représentées de la même façon que les tâches, dans des rectangles :

- `écrire_article` : cette action, consiste, pour une personne `?p`, à rédiger un article. Du point de vue où on se place, on considérera que la personne est rompue à ce genre d'exercice, et cela peut être considéré comme une action élémentaire. Si l'on voulait un plan plus détaillé en sortie, il faudrait considérer cette tâche comme une tâche abstraite ;
- `s'inscrire` ;
- `préparer_planches` ;

- `prendre_avion` : cette action, de même que les suivantes, est une action de déplacement, permettant à une personne `?p` de se rendre à un endroit, ici en prenant l'avion de l'aéroport (considéré comme un lieu) `?l1` à l'aéroport `?l2`.
- `prendre_train` ;
- `conduire` ;
- `marcher`.

Chaque tâche abstraite présentée ci-dessus peut être réalisée par différentes méthodes (le choix des méthodes est représenté par une ligne horizontale sous la tâche associée), et chaque méthode a ses propres préconditions (représentées dans des hexagones allongés). Ainsi, une personne voulant aller à une conférence peut vouloir y aller pour présenter ses travaux, dans ce cas il lui faudra effectuer en séquence (l'opération de séquence est représentée par la flèche horizontale) les sous-tâches (représentées par les "feuilles" de la précondition) d'écriture d'article, d'inscription, de préparation de présentation et de déplacement vers le lieu de la conférence `?l` (déterminé par la précondition `at ?l ?c`, qui doit être vérifiée, et force donc le planificateur à assigner à `?l` la bonne valeur). Elle peut aussi vouloir s'y rendre pour seulement assister aux présentations, auquel cas elle aura juste à s'inscrire et à se rendre sur le lieu de la conférence.

La tâche de déplacement est exprimée avec une écriture récursive. Une personne souhaitant effectuer un déplacement vers un lieu `?l` devra effectuer un ou plusieurs sous-déplacements, notamment pour se rendre sur les lieux de départ ou repartir des lieux d'arrivée des transports de masse (avion, train, ...). Ainsi, s'il veut faire un trajet en avion, il lui faudra s'assurer d'avoir le budget nécessaire, puis il lui faudra choisir un aéroport de départ (`?l1`), un aéroport d'arrivée (`?l2`), puis se rendre à l'aéroport de départ de la façon de son choix (avec un autre avion, en voiture, en train, à pied, ou avec d'autres moyens de transports non listés dans ce HTN par souci de lisibilité), puis prendre son vol, et enfin, se rendre de l'aéroport à sa destination finale `?l` [1] par le moyen de transport de son choix. La procédure est similaire pour un déplacement (qui peut n'être qu'une portion du trajet global) en train.

Si on utilise ce HTN pour planifier notre présentation à JFPDA (qui s'est tenu à Rouen en 2011), la tâche mission est alors `"aller _à_une_conférence JFPDA11 Pascal"`, pour laquelle nous sélectionnons la méthode qui correspond à une soumission d'article. Cette tâche se décompose donc en une écriture d'article ([Schmidt 2011a]), l'inscription à la conférence, la préparation de la présentation, et du déplacement vers Rouen. Pour effectuer ce déplacement, nous avons choisi le train. Pour ce faire, nous avons donc effectué un déplacement vers la gare de Toulouse, nous avons pris le train de Toulouse à Paris Montparnasse, puis avons terminé notre trajet vers Rouen. Pour aller à Matabiau (la gare de Toulouse), nous nous y sommes rendus à pied. Pour faire le trajet de Paris Montparnasse jusqu'à Rouen, nous avons dû faire un déplacement jusqu'à la gare Paris Saint-Lazare, puis nous avons pris le train jusqu'à Rouen, et enfin nous nous sommes déplacés jusqu'au lieu de la conférence. Le changement de

1. Est considérée comme destination finale le lieu `?l` du `déplacer_vers` en cours, indépendamment qu'il soit lui même fils d'un autre `déplacer_vers` ou pas.

gare à Paris s'est effectué en métro (non présent sur le graphique). Une fois arrivé à Rouen, nous avions suffisamment de temps devant nous pour faire la fin du trajet à pied.

Pour résoudre un problème HTN, un planificateur commence avec la tâche de plus haut niveau (cette tâche, instanciée, est donnée dans le fichier de problème), puis choisit une décomposition pour cette tâche, et continue ainsi pour chacune des sous-tâches de la méthode sélectionnée. Et ainsi de suite jusqu'à obtenir un plan avec uniquement des actions élémentaires. L'algorithme du planificateur définit la sélection des méthodes (et les instanciations des variables qui vont avec) ainsi que le processus de backtracks dans ces choix de méthode et d'instanciation.

Les HTN présentent un grand intérêt dans des problèmes avec une structure hiérarchique évidente, quand il est possible pour un expert humain de savoir quels choix sont judicieux, quels choix sont contre-productifs et quelles sont les bonnes stratégies à adopter. Ils sont aussi intéressants lorsque le problème présente de fortes contraintes opérationnelles, dans le cadre d'une mission de type militaire ou critique, par exemple, où le suivi d'une procédure est indispensable. En effet, la décomposition en tâches et sous-tâches se prête très bien à l'écriture de procédures, alors qu'elles seraient bien moins évidentes à modéliser dans les préconditions de chacune des actions pour la planification classique. Il est par contre fort peu recommandé d'utiliser des HTN lorsqu'aucune information n'est disponible quant aux procédures à appliquer ou sur les bonnes stratégies. En effet, l'écriture d'un HTN contraint le planificateur dans la forme de sa solution, qui doit suivre les méthodes imposées par le HTN. Ces contraintes pourraient alors occulter certaines solutions (voire la solution optimale) au planificateur.

1.5.2 Formalisme des HTN

Formellement, un problème HTN est un tuple :

$$hP = (s_0, t_0, g, A, T)$$

où s_0 est l'état initial ; t_0 la tâche de plus haut niveau, instanciée ; g le but ; A l'ensemble des actions élémentaires (comme en planification classique) ; et T l'ensemble des tâches.
Une tâche $t \in T$ est un ensemble composé d'une précondition et de plusieurs méthodes :

$$t = (\text{precond}(t), M(t))$$

où precond(t) est une formule littérale qui représente l'ensemble des états dans lesquels la tâche peut être appliquée, à la manière des préconditions en planification classique ; et $M(t)$ l'ensemble des méthodes $m(t)$.

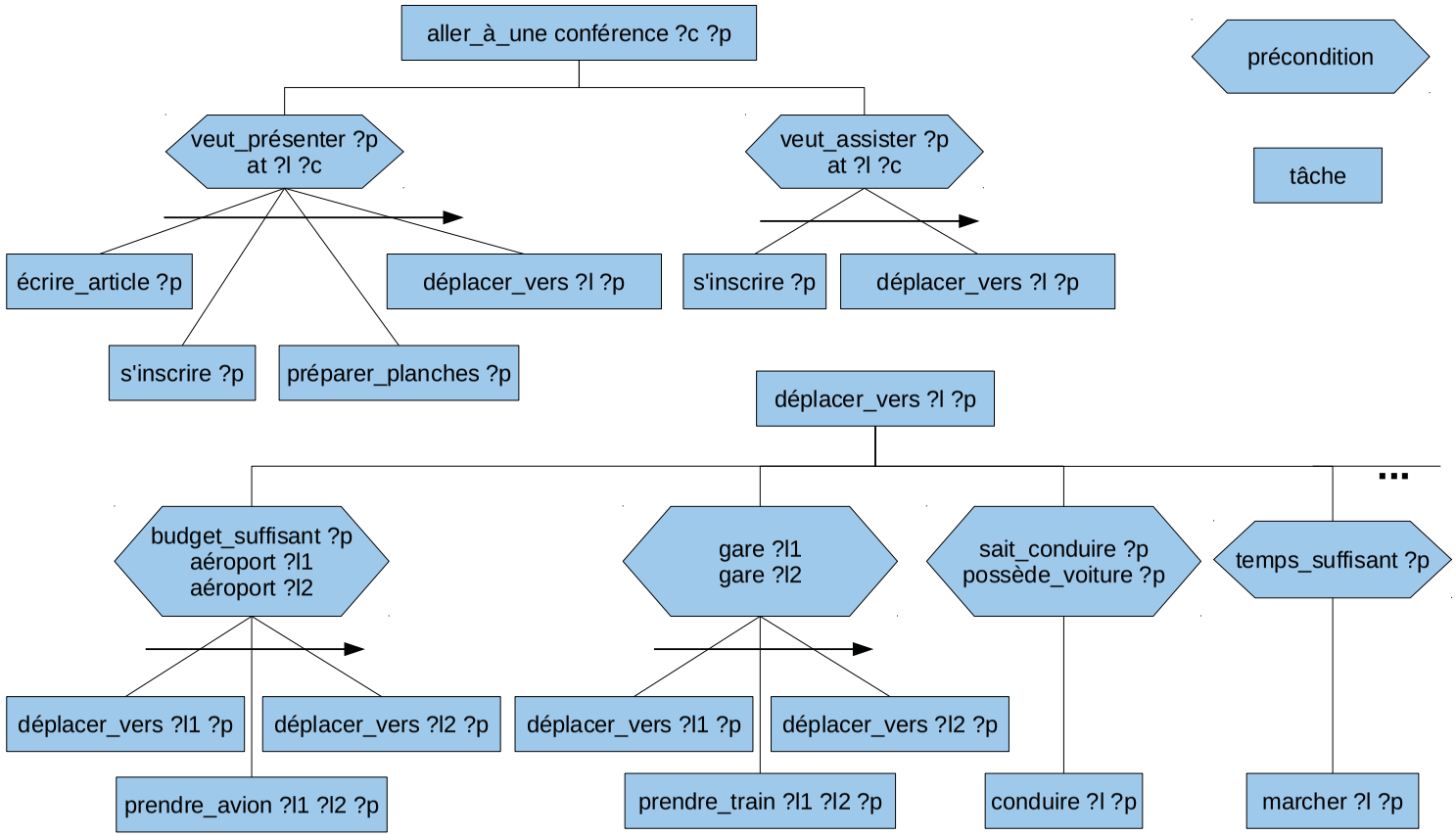


FIGURE 1.2 – Exemple graphique d'un HTN. Ici, une personne `?p` veut se rendre à une conférence `?c`.

Cette tâche s'écrit de la manière suivante :

```
<task> ::=
  ":task" <name>
  ":parameters" <typedVarLst>
  ":precondition" <cond>
  <method>+
```

Où `<typedVarLst>` est une liste de variables typées et `<cond>` une formule booléenne.

Chaque méthode $m(t)$ définit une des décompositions possibles de la tâche en sous-tâches ou actions élémentaires. Une méthode est donc une paire :

$$m(t) = (\text{precond}(m(t)), \text{subtasks}(m(t)))$$

où $\text{precond}(m(t))$ est la précondition à vérifier pour que la méthode soit applicable ; et $\text{subtasks}(m(t))$ la décomposition en sous tâches.
Une méthode s'écrit donc :

```
<method> ::=
  ":method" <name>
  ":precondition" <cond>
  ":subtasks" <subtasks>
```

Et une décomposition en sous-tâches :

```
<subtasks> ::= (<naryTask>)
  | (ordered <subtasks>* )
  | (unordered <subtasks>* )
  | (parallel <subtasks>* )
```

où `<naryTask>` est une tâche avec ses arguments, lesquels proviennent soit des arguments de la tâche t, soit d'une instanciation valide des différentes préconditions. Les mots clés `ordered`, `unordered` et `parallel` indiquent l'agencement des sous-tâches. Dans le cas `ordered`, les sous-tâches sont à exécuter en séquence pure, dans le cas `unordered`, il s'agit de toutes les exécuter en séquence, mais l'ordre d'exécution n'est pas imposé. Le cas `parallel`, permet au planificateur d'exécuter les sous-tâches en parallèle si nécessaire. Le format HTN standard ne permettant pas de modéliser le parallélisme fort entre les actions élémentaires, cette notation autorise en fait n'importe quel entrelacement des sous-tâches jusqu'au bas de la hiérarchie.

Le fichier de domaine contient l'ensemble des tâches et des actions, tandis que le fichier de problème contient l'état initial, le but et la tâche de plus haut niveau, instanciée.

1.5.3 Exemple

Une version textuelle de cet exemple est présentée plus loin.

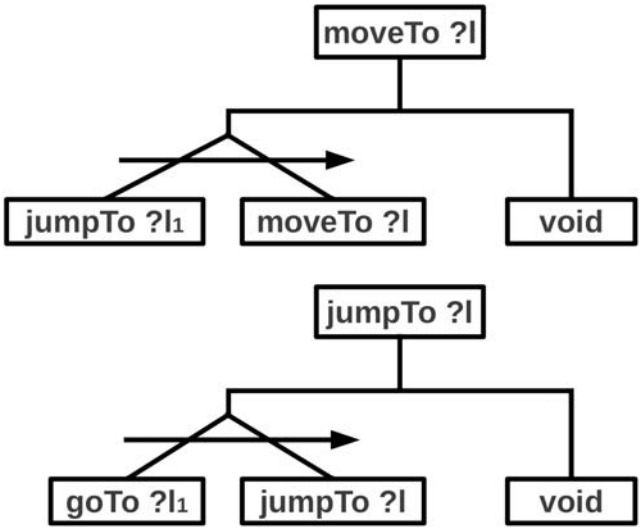


FIGURE 1.3 – Décomposition HTN d'un parcours hiérarchique de graphe.

Si l'on reprend l'exemple du parcours de graphe vu en introduction, on a donc une tâche de haut niveau, `moveTo`, qui va se raffiner en tâches de plus bas niveau, `jumpTo`, qui elles mêmes se raffinent en actions élémentaires, `goTo`.

Pour décrire ceci dans le formalisme HTN, on va utiliser la récursivité :

- il y a deux méthodes pour raffiner un `moveTo` :
  - si la position actuelle correspond à la position d'arrivée du `moveTo` (?l), on ne fait rien,
  - sinon, on fait un `jumpTo` vers une position atteignable en un saut ($?l_1$) puis on continue le `moveTo`,
- il y a deux méthodes pour raffiner un `jumpTo` :
  - si la position actuelle correspond à la position d'arrivée du `jumpTo` (?l), on ne fait rien,
  - sinon, on fait un `goTo` vers une position atteignable en un déplacement élémentaire ($?l_1$) puis on continue le `jumpTo`.

Cette décomposition est illustrée en figure 1.3.

Le code suivant illustre l'écriture formelle en HTN de la tâche `jumpTo`. Elle prend en entrée la position d'arrivée souhaitée ; via les préconditions, elle affecte à $?l_0$ la position courante (`at` $?l_0$), et vérifie qu'un saut est bien faisable (`exists_path` $?l_0$ `?l`) (on supposera qu'un certain nombre d'informations est déjà donné dans le fichier de problème).

La première méthode correspond au cas où il n'y a plus rien à faire, c'est le cas terminal de la fonction récursive.

La seconde méthode est le cas général. On affecte via les préconditions de celle-ci une position voisine de la position courante à la variable $?l_1$ (`elem` $?l_0$ $?l_1$) (il s'agit en fait d'un point de choix déguisé pour le planificateur), puis on fait un déplacement

élémentaire vers ce point (goTo ?l_1) avant de continuer le processus du jumpTo (jumpTo ?l).

```
(:task jumpTo
    :parameters
        (?l - loc)
    :precondition
        (and
            (at ?l_0)
            (exists_path ?l_0 ?l)
        )
    :method terminé
        :precondition
            (= ?l_0 ?l)
        :subtasks
            ()
    :method continuer
        :precondition
            (and
                (not (= ?l_0 ?l)
                (elem ?l_0 ?l_1)
            )
        :subtasks
            (ordered
                (goTo ?l_1)
                (jumpTo ?l)
            )
)
```

1.5.4 Algorithme SHOP2

SHOP2 [Nau 2003] est un planificateur développé par l'Université du Maryland (USA-MD) pour résoudre les problèmes HTN dans leur version définie dans [Erol 1994b]. Développé en Lisp, il en existe aussi une version en Java (JSHOP2), plus facile d'utilisation, mais moins performante. Bien que d'autres planificateurs soient basés sur un formalisme inspiré de HTN, SHOP2 est, à notre connaissance, le seul planificateur HTN orienté STRIPS.

Son algorithme est fortement récursif et consiste en un parcours de l'arbre HTN en profondeur d'abord (cf Algorithme 3). Le planificateur est initialisé avec la tâche instanciée de plus haut niveau t_0 et l'état initial s_0 (ligne 1). Il lance ensuite la procédure récursive openTask avec l'état initial, la tâche correspondant à t_0 (t_0.task), ses paramètres (t_0.args), ainsi que des ensembles vides correspondants au plan construit actuellement et aux tâches en attente (ligne 2).

Pour chaque développement de tâche, le planificateur teste si les préconditions sont valides, puis sélectionne une instance valide de ses variables "libres" (présentes

dans les préconditions, mais non encore instanciées, comme $?l_0$ ou $?l_1$ dans l'exemple précédent)(ligne 8). Nous noterons $\mathcal{V}$ un ensemble de variables instanciées.
Deux cas se présentent alors :

- la tâche à développer est une action élémentaire : l'algorithme applique cette action à l'état courant pour obtenir l'état suivant (ligne 11). Il teste ensuite s'il reste des tâches à développer (`nextTasks` $\neq \emptyset$), auquel cas il récupère le premier élément de la liste des tâches en liste d'attente (`nextTasks.head`) ainsi que la suite de la liste (`nextTasks.tail`) (lignes 13 et 14), et ré-appelle `openTask` pour continuer la planification (ligne 16), avec le nouvel état, la prochaine tâche à développer, les tâches en file d'attente, et le nouveau plan partiel, qui correspons au plan partiel courant (`partPlan`) concaténé (`::`) avec l'action que l'on vient d'appliquer (`t`). Si, au contraire, il ne reste plus de tâches à développer, le plan est terminé. Il faut tester si le but est atteint, auquel cas on le note dans la variable `plan'`.
- la tâche à développer est une tâche abstraite : le planificateur sélectionne alors une méthode valide et instancie ses variables "libres" (ligne 24). Il développe ensuite, par récursivité, la suite du plan, avec l'état courant, la première des sous-tâches de la méthode choisie (`m.head`), et une concaténation de la suite des sous-tâches de la méthode (`m.tail`) et des tâches en attente (`nextTasks`) (lignes 25 à 28). La boucle sur les instanciations libres permet d'optimiser le plan.

Si à un moment donné les préconditions d'une tâche ne sont pas valides, ou aucune méthode n'a ses préconditions valides, l'algorithme effectue un "backtrack" sur son dernier choix de variable pour faire une instanciation différente. De même quand toutes les instanciations ont été testées et qu'aucune ne donne de résultat valide. Ce backtrack s'effectue naturellement par la structure récursive de l'algorithme, en retournant un booléen faux (lignes 38 et 40).

Lorsqu'un plan est trouvé, si l'option d'optimisation n'est pas choisie, le plan est retourné à l'utilisateur (ligne 4). Sinon, le plan est stocké en mémoire, la planification continue, et une fois toutes les possibilités explorées, le planificateur retourne celle avec le coût le plus faible (boucles sur les instanciations libres). L'algorithme 3 présente le cas où l'algorithme est optimal.

1.5.5 Les HTN probabilistes

Des travaux ont été menés pour appliquer les HTN au cadre des MDP [Kuter 2005]. Le modèle ne change pas, sauf au niveau des actions élémentaires qui ont alors des effets probabilistes. L'algorithme de résolution, lui, est complètement différent. Il est présenté en algorithme 4, dans sa version générique pouvant utiliser n'importe quel planificateur MDP. `s` représente l'état courant, `G` le but, π le plan courant et `x` la position courante dans le HTN `D`. Le HTN n'est plus parcouru branche par branche comme dans SHOP2, mais sert de guide d'exploration à un algorithme de planification MDP en avant classique. En effet, pour chaque état, il stocke, à la manière d'une pile d'exécution, sa position dans le HTN. Le suivi de la position dans

Algorithm 3: SHOP2:

```
1  begin  initPlanner(s0,t0)
2      (plan, valide) ← openTask(s0, t0.task, t0.args, ∅, ∅);
3      if valide = true then
4          return plan
5  begin  openTask(st, t, vars, partPlan, nextTasks)
6      if (t.precond = true) then
7          valide ← false;
8          forall the V\t.precond(st,vars∪V)=true do
9              valide' ← false;
10             if isElemAct t then
11                 st' ← apply(a,n.st);
12                 if nextTasks ≠ ∅ then
13                     t' ← nextTasks.head;
14                     nextTasks' ← nextTasks.tail;
15                     (plan', valide') ← openTask(st', t', vars∪V,
16                                                 partPlan::t, nextTasks');
17                 else
18                     if st' = goal then
19                         plan' ← partPlan+t;
20                         valide' ← true;
21             else
22                 forall the m in t.methods do
23                     valide" ← false;
24                     forall the V'\t.precond(st,vars∪V ∪ V')=true do
25                         t' ← m.head;
26                         nextTasks' ← m.tail::nextTasks;
27                         (plan", valide") ← openTask(st, t', vars∪V ∪ V',
28                                                     partPlan, nextTasks');
29                     if valide" and cost(plan") < cost(plan') then
30                         valide' ← true;
31                         plan' ← plan";
32             if valide' and cost(plan') < cost(plan) then
33                 valide ← true;
34                 plan ← plan';
35         if valide then
36             return (plan, true)
37         else
38             return (∅, false)
39     else
40         return (∅, false)
```

le HTN après l'exécution de chaque action s'effectue en ligne 10. L'algorithme de planification en avant choisi (RTDP ou LAO*, par exemple) explore ensuite l'espace d'états comme à l'accoutumée (ligne 7), sauf que la liste des actions (élémentaires) possibles est restreinte par la liste des actions permises par le HTN dans l'état courant (ligne 4, la fonction `applicable(a, s)` teste si les préconditions de `a` sont valides en `s`, tandis que `acceptable(s, a, x, D)` détermine si `a` est autorisé dans l'état `s` au regard de la décomposition `D` et de la position `x` dans celle-ci). Ceci permet de restreindre grandement le facteur de branchement [2] dans chaque état par rapport à un MDP classique, et donc l'espace de recherche, et ainsi d'obtenir de très bonnes performances.

Algorithm 4: Controlled-Plan:

```
1  begin  Controlled-Plan(s, G, π, x, D)
2      if s ∈ G then
3          return π
4      actions ← {a | applicable(a, s) ∧ acceptable(s, a, x, D)};
5      if actions = ∅ then
6          return (failure)
7      nondeterministically choose a ∈ actions;
8      s ← result(s, a);
9      π ← append(π, a);
10     x ← progress(s, a, x, D);
11     return Controlled-Plan(s, G, π, x, D);
```

1.6 Synthèse

Nous avons pu voir dans ce chapitre le formalisme PDDL ainsi que les techniques de résolution d'un problème de planification classique. Ce formalisme ainsi que les algorithmes serviront de base à la suite de nos développements, notamment pour la construction de l'algorithme de résolution d'un problème à un niveau hiérarchique fixé.

Nous avons de plus pu faire un rapide tour des différents cadres et techniques de résolution rencontrés dans le domaine de la planification de nos jours. Notamment les techniques liées à l'expertise humaine ainsi qu'à la résolution hiérarchique.

En utilisant comme point de départ le formalisme HTN et l'idée d'une résolution hiérarchique, nous proposerons un nouvel algorithme qui permettra d'effectuer une résolution de type hiérarchique, réconciliant d'une part les approches des premières heures comme Nonlin avec les approches récentes telles que les HTN. Nous voulons en effet combiner les avantages d'une approche de planification par niveaux séparés,

2. Facteur de branchement : ici, le nombre d'actions possibles dans chaque état.

permettant une détection au plus tôt des difficultés et ceux des approches de planification PDDL, qui, par des mécanismes de backtrack peuvent faire une optimisation globale de la solution en fonction des coûts sur les tâches. Nous nous attacherons de plus à relier tout cela au formalisme PDDL afin de rassembler les algorithmes et apports dans un cadre moderne et fortement utilisé par la communauté.

Pour cela, nous étendrons le formalisme HTN de façon à rendre la planification possible à des niveaux abstraits ne contenant que des tâches. Nous nous inspirerons aussi de l'algorithme des HTN probabilistes qui permet d'étendre les algorithmes de planification actuels pour leur rajouter les contraintes des HTN et faire cette résolution par niveaux. Enfin, nous proposerons une structure de contrôle novatrice pour faire communiquer des planificateurs opérant à chacun des niveaux hiérarchiques distincts.

Chapitre 2

Formalisation et contributions

Sommaire

2.1 Expressivité choisie 39
2.2 Comment lier des tâches de haut niveau ? 40
2.2.1 Motivations 40
2.2.2 Les méta-effets 42
2.2.3 Notion d'optimisme 43
2.2.4 Exemple 44
2.2.5 Intégration des heuristiques 46
2.3 Résoudre les problèmes de hiérarchie 47
2.3.1 Motivations 47
2.3.2 Les macro-tâches 49
2.3.3 Utilisation 49
2.3.4 Exemple 50
2.4 Formalisation générale d'un couple domaine-problème . . . 52

Après avoir présenté quelques solutions retenues par les différents acteurs de la planification dans les domaines de la modélisation et de la résolution des problèmes, nous allons présenter nos choix et propositions de formalisme pour l'application de notre algorithme introduit en section 2.3.2 de l'introduction. Nous illustrerons au fur et à mesure ce formalisme sur l'exemple de référence présenté en introduction.

2.1 Expressivité choisie

Nous avons fait le choix de nous baser sur le formalisme HTN proposant des tâches de divers niveaux hiérarchiques, et des méthodes pour les étendre en sous-tâches. Nous voyons là un point de départ idéal pour pouvoir construire des plans de divers niveaux de précision.

Cependant, nous souhaitons que notre planificateur puisse être utilisable dans de nombreux domaines, notamment celui de la robotique.
Pour cela, nous jugeons nécessaire que l'expressivité de notre planificateur ne se limite pas à la logique L1 [Stolyar 1970]. En effet, une telle logique se révèle vite contraignante quand il s'agit d'écrire des problèmes concrets. Les prédicats ne pouvant prendre que les valeurs *vrai* et *faux*, on ne peut par exemple pas écrire que la position d'un robot (`at ROBOT`) vaut une position `P`, mais lister que pour chacune des positions (`P'`) différentes de P, le prédicat `at ROBOT P'` vaut *faux* et que `at ROBOT P` vaut

vrai. Une telle représentation ne reste concise que grâce à l'hypothèse du monde clos, mais peut atteindre une complexité en $O(n)$ avec n le nombre de positions possibles sur le temps de recherche, notamment dans le cas de préconditions négatives.

Nous proposons donc une représentation avec une logique L2 [Shapiro 1991] qui permettra d'exprimer des fonctions d'objet, c'est à dire des prédicats pouvant prendre des valeurs autre que booléennes : des objets typés déclarés dans le problème, et des valeurs numériques. Ainsi, on pourra exprimer que `(at ROBOT)` vaut `P`, et trouver la position du robot avec une complexité en $O(1)$, et avoir une représentation compacte. Cette représentation utilise des parties de la version 3.1 de PDDL [Geffner 2000].

Ce que nous appellerons maintenant une *fonction* est un mot clef permettant d'associer plusieurs objets appelés *paramètres* à un objet appelé *valeur*. On pourra bien sûr utiliser des combinaisons de fonctions valuées pour obtenir une valeur, tel que :

```
(angle (brasGauche ROBOT))
```

avec la fonction `brasGauche` qui associe un robot et son bras gauche, et la fonction `angle` qui renvoie l'angle d'un objet dans un repère donné.

Nous autorisons de plus les opérations numériques standards (+, -, *, /) et quelques primitives un peu plus évoluées et non linéaires telles que *sqrt* pour le calcul de racine carrée, très utile pour calculer une distance, ou *pow* pour calculer une puissance. Nous nous sommes limités à ces opérations pour notre version du planificateur, mais le formalisme est tel qu'il ne sera pas difficile de développer un planificateur capable de comprendre plus de fonctions, comme *log*, *max*, *argmin*... Ces opérations numériques permettent de faire un grand pas en avant sur la facilité de modélisation par rapport à la logique L1, qui force à définir les objets `ZERO`, `UN`, `DEUX`... et les prédicats `suivant` et `précédent` pour définir des compteurs par exemple. Ces variables numériques sont supportées depuis la version 2.1 de PDDL [Fox 2003], mais nous n'avons pas souhaité garder la partie temporelle de PDDL2.1.

C'est à partir de ce formalisme que nous effectuerons des extensions nous permettant de mettre en œuvre nos algorithmes.

2.2 Comment lier des tâches de haut niveau ?

Notre première problématique a porté sur la construction d'un plan à un niveau hiérarchique donné. Nous souhaitons en effet pouvoir construire un plan abstrait, indépendamment du choix des méthodes qui pourrait être fait à plus bas niveau, et donc sans avoir à effectuer cette décomposition. Nous verrons par la suite que comme c'est cette décomposition qui permet aux planificateurs HTN d'inférer l'état courant pour pouvoir faire les calculs de précondition, il va nous falloir introduire une nouvelle extension permettant d'estimer cet état.

2.2.1 Motivations

Comme on l'a vu dans la section 1.5, une tâche HTN comprend une précondition et des méthodes, mais rien pour indiquer au planificateur ce que fait effectivement

cette tâche. Ceci pose un certain nombre de problèmes techniques pour la mise en place de notre algorithme.
En effet, un tel manque rend impossible la construction d'un plan composé uniquement de tâches tel que nous souhaiterions le faire. On ne peut pas connaître l'état dans lequel on va se trouver après avoir appliqué une tâche (ni même une estimation de cet état), il est donc impossible de savoir par exemple quel est l'ensemble des actions applicables dans cet état car aucun calcul de précondition ne peut être fait. De plus, en l'absence de connaissance de l'état courant, il est impossible de savoir si le but est atteint ou pas ni même le coût d'un plan donné.
Enfin, ce manque ne permettant pas de connaître l'effet d'une action, il rend totalement impossible le calcul d'une quelconque heuristique à un niveau donné et fait que le choix d'une action se fait sur un critère arbitraire, et non motivé.

Sur notre exemple de référence, une tâche `jumpTo X` ne contient pas l'information qu'après son utilisation, on se retrouve dans un état où le prédicat `(at X)` est vrai (ou plutôt, exprimé en logique L2, la fonction `(at)` prend la valeur `X`). Nous ne pouvons donc détecter quand la succession de `jumpTo` nous amène dans l'état but, `L`, ni même lister après un saut quelles sont les destinations possibles pour le saut suivant.

De plus, lorsqu'un expert écrit une tâche, il le fait en sachant ce à quoi elle sert, ce qu'elle est censée faire, au moins dans les grandes lignes. Il ne lui sera donc pas nécessairement plus coûteux de l'indiquer au planificateur.

Nous proposons donc, afin d'adapter le formalisme HTN à nos exigences de résolution, de rajouter au formalisme un moyen pour exprimer des effets d'une tâche qui seraient exploitables par le planificateur. Ils pourront servir à lier les tâches entre elles à un niveau donné et suivre l'évolution de l'état à ce niveau là. On peut les voir comme une sorte d'heuristique experte.
Ces effets seront ajoutés à la main et non inférés automatiquement en fonction des méthodes comme on pourrait le faire avec la logique de Hoare [Hoare 1969], car nous pensons que d'une part les effets principaux de la tâche risquent de ne pas être inférés car il existe des choix de méthodes qui n'aboutiront pas au but et d'autre part que les effets et préconditions inférés se retrouvent en contradiction avec le niveau d'abstraction : si l'algorithme infère un besoin en carburant pour une tâche mais n'infère pas que la tâche précédente pourra recharger le véhicule en carburant, la planification sera rendue impossible.

Nous verrons de plus ici que ces effets, contrairement à ce qui est fait dans Nonlin, ne traduisent pas l'exactitude et la multitude des effets qui seront ou pourront être rencontrés lors du développement d'une tâche (une discussion sur l'expressivité et l'exactitude de ces effets a été menée dans [Schmidt 2009]). Nous apporterons donc un certain nombre de règles et contraintes ainsi que des technique d'utilisation de ces effets pour garantir le bon déroulement de la planification.

2.2.2 Les méta-effets

Nous appellerons ces effets *méta-effets* dans le sens où ils s'appliquent à des tâches, parfois appelées *méta-actions* et ne correspondent pas à des effets normaux car ils ne donnent pas l'effet exact d'une tâche, mais un effet attendu. Leur syntaxe, quant à elle, respectera la syntaxe des effets dans les actions élémentaires, de façon à être utilisable par un planificateur classique.

Il s'agit donc, dans le cadre de la planification STRIPS d'une liste d'ajouts et de retraits, mais pouvant être composée de formules plus complexes, comme des effets conditionnels, d'affectation de valeurs ou d'objets à des fonctions par exemple.

L'expression en BNF (Backus-Naur Form) de ces méta-effets est la suivante :

```
<metaEffect> ::= ":metaEffect" <effect>

<effect> ::= <pEffect>
  |"(forall" "(" <typedVarLst> ")" <effect> ")"
  |"(when" <cond> <effect> ")"
  |"(and" <effect>+ ")"

<pEffect> ::=
  |"(assign" <fHead> <fExp> ")"
  |"(increase" <fHead> <fExp> ")"
  |"(decrease" <fHead> <fExp> ")"
  |"(" <fHead> ")"
  |"(" not <fHead> ")"
```

où `<metaEffect>` représente le méta effet en lui même ; `<effect>` la construction d'un effet classique ; et `<pEffect>` un effet primaire, sur un prédicat ou une fonction `<fHead>`. Ces méta-effets sont introduits par le mot clef `:metaEffect` dans une tâche, et permettent d'appliquer un effet sur toutes les occurrences d'une variable typée (`<typedVarLst>`) grâce à `forall`, ou lorsqu'une condition booléenne (`<cond>` est vérifiée sur l'état et les variables locales grâce au mot clef `when` ou d'appliquer plusieurs effets distincts (le tout étant cumulable).
L'effet primaire, dans notre cas peut concerner un prédicat `<fHead>` que l'on rajoute ou que l'on retire à l'ensemble des prédicats vrais, ou à une modification de la valeur d'une fonction `<fHead>`, via le mot clef `assign` qui assigne le résultat d'une expression à une fonction, ou les mots clef `increase` ou `decrease` qui augmentent ou diminuent la valeur numérique d'une fonction de la valeur d'une expression `<fExp>`. Chaque prédicat ou fonction, que ce soit dans les expressions ou dans les fonctions modifiées, peut prendre comme argument des objets présents en variable locale à la tâche ou d'autres fonctions (qui retourneront une valeur, un booléen ou un objet).

Ces méta-effets ont donc la même expressivité que les effets des actions. Dans notre cas, toute formule de la logique L2 peut être exprimée, mais si le formalisme de base se restreint à la logique des prédicats, les méta-effets subiront la même restriction. De même, ils bénéficieront de toute extension dans l'ouverture du formalisme à d'autres formes d'expressivité pour l'état et les effets.

2.2.3 Notion d'optimisme

Comme nous l'avons signalé en introduction de cette section, les effets d'une tâche ne sont pas connus avec certitude lors de l'écriture du domaine. En effet, une tâche pouvant avoir plusieurs méthodes, chaque méthode plusieurs instanciations possibles, et de même sur les sous-méthodes, connaître l'effet exact d'une tâche reviendrait à l'avoir déjà planifiée dans tous les contextes possibles.
Prenons l'exemple simpliste de la tâche `jumpTo`. Si l'on sait que pour que la tâche soit applicable il faut qu'il existe un chemin entre le point de départ du saut et le point d'arrivée, on ne connaît pas la longueur du chemin optimal, ni même celle du chemin choisi par le planificateur s'il doit optimiser autre chose que la distance parcourue. Si on peut affirmer avec certitude que la fonction `at` prendra la position d'arrivée comme valeur après application de cette tâche, il est par contre impossible de donner le temps de parcours exact qui sera nécessaire pour l'appliquer.

Nous proposons donc un compromis, qui consiste à ne donner que des effets partiels voire potentiellement erronés.
En ce qui concerne les effets partiels, il s'agit ici de volontairement laisser de côté un certain nombre de variables d'état et de fonctions qui sont jugées non pertinentes à ce niveau de granularité. Par exemple, dans une planification de trajectoire pour véhicule, on peut, à haut niveau, ne se préoccuper que des variables de position clés du trajet qui donneront une valeur approximative de la distance et donc du temps de trajet, mais laisser de côté la fonction de niveau de carburant, et ne se soucier qu'à des niveaux plus fins de la planification des détours pour aller faire le plein.
Les effets potentiellement erronés visent, eux, à permettre de donner une information sur l'évolution d'une fonction, sans pour autant en connaître le comportement exact. Toujours sur l'exemple précédent, on peut donner une information sur le temps d'une portion de trajet, qui sera indicative et non exacte, car on ne sait pas quels détours le planificateur va choisir de prendre, mais qui permettront au planificateur d'avoir un minimum d'information, surtout si c'est la variable qu'on cherche à minimiser sur le problème.

Nos méta-effets n'étant pas exacts mais devant respecter la syntaxe des effets normaux, il nous faut définir un certain nombre de contraintes pour que la planification se passe correctement.

La première contrainte est directement une contrainte de cohérence. Afin que le planificateur soit en mesure d'effectuer une planification en utilisant toutes les tâches d'un niveau hiérarchique donné, il est nécessaire que l'on retrouve une cohérence au sein de leurs effets et préconditions. Ainsi, il faudra que l'expert humain en charge de la modélisation évite soigneusement d'exprimer certains prédicats dans les effets et préconditions de tâches de haut niveau s'ils ne sont pas présents dans l'abstraction des autres tâches de niveau équivalent. On évite ainsi de se retrouver avec des préconditions d'une tâche non réalisables parce que la tâche précédente n'exprime pas qu'un certain prédicat apparaît dans ses post-conditions.

La seconde contrainte concerne les fonctions dont nous avions qualifié les valeurs de potentiellement erronées, comme la distance d'un saut. Pour contraindre ces écarts,

nous nous sommes inspirés des heuristiques admissibles de A*. De même que ces heuristiques, nos méta-effets ne représentent pas exactement l'effet d'une tâche, mais une indication de cet effet. Et comme au paragraphe précédent, nous souhaitons que ces approximations ne brisent pas le processus de planification.

Définition 2.1 (*Admissibilité des méta-effets*)

Nous définissons un méta-effet comme admissible si et seulement si il ne sur-contraint pas le problème et définit une heuristique admissible au sens de A.*

Pour ne pas sur-contraindre un problème, il faut que le méta-effet ne fasse pas apparaître d'effets contraignants alors qu'ils pourraient ne pas apparaître dans certaines décompositions. Le fait de volontairement ignorer ces contraintes dans l'écriture des méta-effets revient à avoir un vision optimiste de la tâche.

Dans notre exemple de référence, attribuer comme méta-effet d'une tâche `jumpTo ?l` le fait que la fonction (`at`) prenne la valeur `?l` est admissible. Mais mettre comme méta-effet que (`at`) prenne `?l0` (où `?l0` est le point de départ du saut, et non son point d'arrivée) en pensant aux branches qui ne permettent pas d'atteindre `?l` depuis `?l0` sur-contraint le problème et est donc non-admissible.

Définition 2.2 (*Optimisme des méta-effets*)

Nous définissons un effet comme optimiste si et seulement si le coût sur le long terme de la tâche associée, et ce quelle que soit la décomposition choisie, est supérieur au coût à long terme associé à l'effet.

La propriété d'optimisme, bien que non indispensable au bon déroulement du processus de planification permet, comme on le verra plus tard, de garantir les propriétés de complétion et d'optimalité du planificateur.

Dans notre exemple de référence, attribuer comme coût dans le méta-effet d'un `jumpTo` la distance à vol d'oiseau depuis le point de départ du saut est optimiste, car aucun chemin ne peut être plus court que celui-là, par contre mettre une valeur plus élevée est nécessairement non-optimiste car il se peut qu'il existe un chemin en ligne droite, et cela risquerait d'occulter une solution (à la manière de la résolution heuristique de A*) à cause de ce méta-effet.

Bien entendu, il se peut que toutes précautions prises, le plan solution ne soit pas trouvable directement par le planificateur haut-niveau du fait d'effets trop éloignés des méta-effets prévus. Si par exemple les méta-effets d'une tâche donnent une branche comme nettement mois coûteuse qu'elle ne l'est normalement et la font passer comme meilleure que la branche optimale, qui elle, est bien estimée, le plan solution trouvé localement, peut ne pas être celui recherché. Nous verrons plus loin comment garantir que le plan solution soit quand même trouvé.

2.2.4 Exemple

Nous allons maintenant reprendre l'exemple de recherche de trajectoire dans un graphe et voir comment lui appliquer les méta-effets.

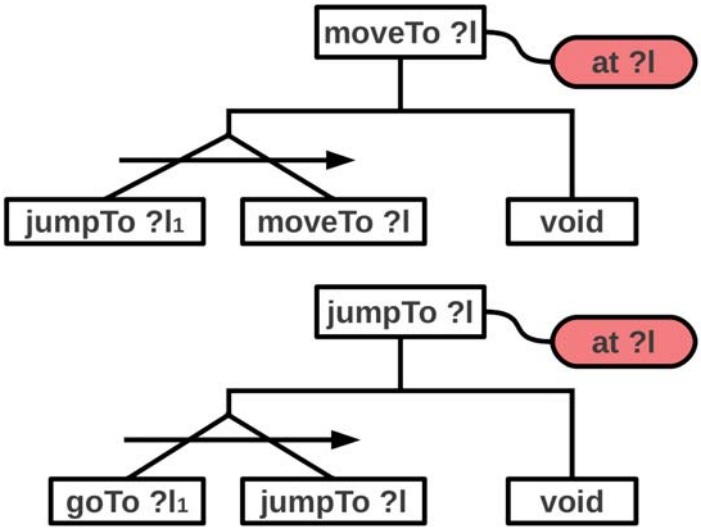


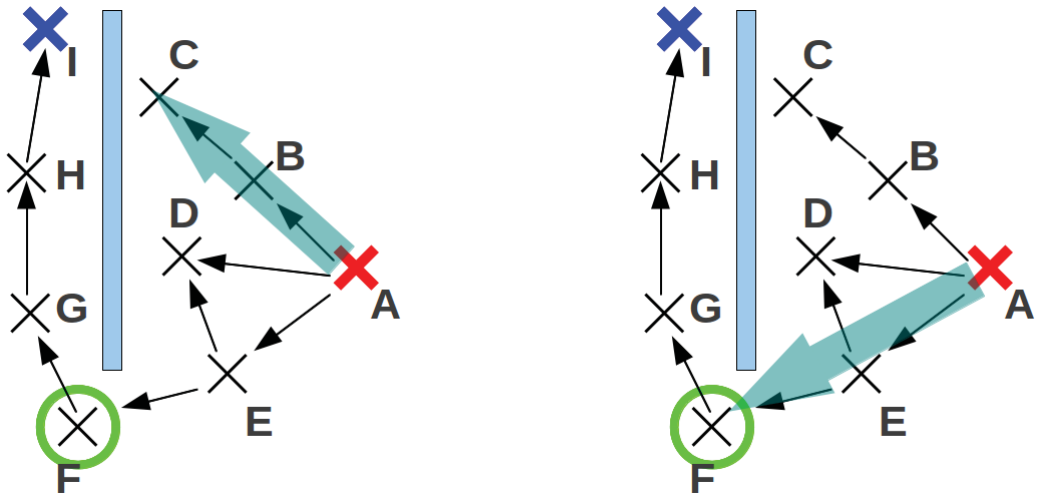
FIGURE 2.1 – Exemple de méta-effet sur les méta-actions de déplacement. Ceux-ci sont exprimés dans les cartouches accolés aux tâches.

Nous cherchons ici à trouver le chemin le plus court en distance entre deux points, connaissant des points de passage obligatoires, et sachant s'il existe un chemin (non absurde [1]) les reliant entre eux. Dans notre problème, nous ne nous préoccupons pas du temps de parcours (et donc d'un éventuel facteur de vitesse) ni d'une quelconque consommation de ressources, comme le carburant. Les deux seules variables du problème sont donc la position et la distance parcourue.

Étant donné que nous avons la connaissance du fait qu'un déplacement, quelque soit son niveau d'abstraction, n'est possible que s'il existe un chemin entre le point de départ et le point d'arrivée (donnée entrée par le modélisateur humain), nous pouvons affirmer avec certitude que l'application d'une tâche de déplacement (`moveTo` ou `jumpTo`) amènera le véhicule au point souhaité. On peut donc dire que, parmi les effets de ces tâches, `(assign (at) ?l)` sera présent (`?l` étant le paramètre passé à ces tâches et correspondant à leur destination). C'est ce qui est exprimé dans la figure 2.1 par les cartouches associés aux tâches.

En ce qui concerne la distance parcourue pour effectuer une tâche de déplacement, l'évaluation est plus difficile. En effet, sans avoir fait l'action de planification, on ne peut pas connaître la longueur du plus court chemin. Par contre, pour avoir un méta-effet optimiste, il faut et il suffit que cette longueur soit un minorant de la vraie longueur. En supposant que le problème se passe dans un espace euclidien, nous pouvons affirmer que la longueur d'un chemin reliant deux points est nécessairement supérieure ou égale à la distance à vol d'oiseau entre ces deux points. En écrivant cette distance-là dans les méta-effets de la tâche, nous obtenons un méta-effet admissible.

1. Nous qualifions un chemin de non absurde entre deux points de passage s'il ne nécessite pas de passer par un autre point de passage obligatoire.



(a) Heuristique classique

(b) Heuristique de tâche

FIGURE 2.2 – Direction privilégiée par différentes heuristiques.

Dans notre formalisme, ce méta effet s'écrit alors :

```
:metaEffect
  (and
    (assign (at) ?l)
    (increase (cost)
      (sqrt
        (+
          (pow (- (x (at)) (x ?l)) 2)
          (pow (- (y (at)) (y ?l)) 2)
        )
      )
    )
  )
```

2.2.5 Intégration des heuristiques

Nous avons de plus affirmé, au début de cette section, que les méta-effets permettaient de calculer des heuristiques pour motiver le choix des tâches. Nous pensons qu'il n'est pas judicieux de calculer une heuristique globale sur chaque état, c'est à dire une heuristique qui en chaque état définit une distance au but. En effet, lors du développement d'une tâche, l'objectif primaire est de se rapprocher du but de la tâche (cf. figure 2.2(b)) et non de se rapprocher du but global (cf. figure 2.2(a)), plus particulièrement quand la tâche fait faire un détour par rapport au chemin menant au but pour éviter un obstacle prévu à plus haut niveau.

Notre méthode de calcul des heuristiques est donc un peu plus complexe. Afin

que le planificateur soit guidé vers la réalisation de la tâche, mais que la proximité du but joue un rôle important sur l'ordre de développement des états, notre heuristique est de la forme :

$$h = h_s^t + h_t$$

avec h la valeur de l'heuristique ; h_s^t l'estimation heuristique du coût restant à partir de l'état courant s pour terminer la tâche t en cours ; et h_t la distance de la fin de la tâche t jusqu'au but, trouvée en planifiant à plus haut niveau.
La formule de l'heuristique pour atteindre la fin de la tâche est donnée pour chacune des tâches, et peut donc différer d'une tâche à l'autre, afin d'être plus précis dans sa formulation.

Si l'heuristique formulée dans la tâche est admissible, alors h_s^t est bien inférieur au coût réel pour atteindre la fin de la tâche. Si les méta-effets formulés au niveau supérieurs sont bien admissibles, le coût du plan calculé pour passer de la fin de la tâche au but h_t est inférieur au coût réel nécessaire. Le plan de niveau courant devant passer par le point indiqué par la fin de la tâche, la valeur de h étant la somme de deux parties sous-estimant chacune le chemin à parcourir, elle est inférieure au coût réel nécessaire pour atteindre le but depuis l'état courant et est bien admissible.

Nous avons ici défini la première extension faite au formalisme HTN : les méta-effets. Ils forment le point de base indispensable à la planification par niveaux distincts en permettant le bon déroulement d'une planification en avant grâce au suivi de l'évolution de l'état. Ils permettent de plus une motivation dans le choix des tâches de haut niveau grâce à une notion d'heuristique.

2.3 Résoudre les problèmes de hiérarchie

Une fois ce prérequis défini, nous allons nous attaquer à une seconde extension qui, bien qu'elle ne modifie pas l'expressivité générale du formalisme, permet une résolution par niveaux cohérents et facilite le travail de modélisation.

2.3.1 Motivations

Notre objectif est de pouvoir résoudre des problèmes hiérarchiques niveau par niveau, avec chacun de ces niveaux cohérents entre eux au niveau de l'abstraction représentée. Or, une telle chose n'est pas toujours possible avec le formalisme des HTN.

Le cas le plus évident où l'on ne peut forcer le planificateur à utiliser des tâches d'abstraction équivalente au sein d'un même niveau est le cas d'une arité inconnue a priori. Par exemple, nous souhaitons décomposer un `jumpTo` en une série de déplacements élémentaires `goTo`. Si l'on pouvait savoir à l'avance qu'un `jumpTo` se décompose toujours en trois `goTo`, il serait facile d'écrire qu'il s'agit d'une séquence de trois déplacements élémentaires. Mais dans notre cas, le nombre de `goTo` est totalement imprévisible, et il faudrait écrire une infinité de méthodes différentes, chacune donnant la décomposition pour un nombre de `goTo` donné.

L'autre possibilité pour décomposer cette tâche en HTN est celle que nous avons vue jusqu'à présent, et qui consiste à utiliser la récurrence. Un `jumpTo` se décompose soit en rien du tout si on est déjà à destination, soit en une séquence d'un `goTo` vers une position voisine, et d'un `jumpTo` vers la position finale du `jumpTo` en cours de décomposition.
Cette structure hiérarchique, bien que représentant correctement la décomposition souhaitée d'un `jumpTo` dans le plan de plus bas niveau, présente des incohérences dans les niveaux intermédiaires. Ainsi, dans notre exemple présenté en figure 3, si l'on est en position `A` et que l'on fait un `jumpTo` vers `G`, les plans des différents niveaux hiérarchiques seront :

- niveau 0 : `jumpTo G`
- niveau 1 : `goTo B` ; `jumpTo G`
- niveau 2 : `goTo B` ; `goTo E` ; `jumpTo G`
- niveau 3 : `goTo B` ; `goTo E` ; `goTo G` ; `jumpTo G`
- niveau 4 : `goTo B` ; `goTo E` ; `goTo G`.

On voit bien, sur les niveaux 1 à 3 une cohabitation entre des `goTo` et des `jumpTo` qui forment donc des plans incohérents sur le niveau d'abstraction contenu (ces combinaisons seraient encore plus aberrantes si on avait gardé la hiérarchie complète avec les `moveTo`).

De plus, nous pensons que l'usage de la récursivité n'est pas très intuitif pour la plupart des utilisateurs potentiels du planificateur, qui ne sont pas tous des experts en programmation. En effet, les modèles récursifs présentent plusieurs difficultés ou pièges qui ne sont pas facile à surmonter pour qui n'a pas l'habitude de travailler avec :

- **le cas terminal :** servant à détecter la fin de la récursivité et à faire l'action (ou l'absence d'action) nécessaire pour en sortir, pas toujours évident à concevoir, et nuisant à la lisibilité du modèle en diluant l'information utile ;
- **récursion droite ou récursion gauche ?** Choix nécessaire pour éviter que l'algorithme ne fasse des descentes récursives infinies en développant le modèle. Dans la plupart des cas, une récursion gauche, qui place la répétition de la fonction récursive avant l'action "utile" conduit à une non-terminaison de la résolution, il faut donc choisir la récursion droite (comme dans la figure 1.3). Cependant, certains mécanismes de résolution, effectuant un branchement à partir du but par exemple, nécessitent une récursion gauche. Il est donc nécessaire d'avoir une bonne habitude des formalismes récursifs et une bonne connaissance de l'algorithme pour modéliser des problèmes avec ce formalisme ;
- **la compréhension :** il est souvent difficile de visualiser rapidement le concept sous-jacent d'un algorithme (ou modèle) récursif. En effet, il faut généralement essayer de dérouler soi même les premières étapes de l'algorithme, et étudier en détail le cas terminal pour se rendre compte de l'effet.

2.3.2 Les macro-tâches

C'est pour cela que nous avons mis en place notre seconde extension, les macro-tâches. Cette extension, initialement inspirée des expressions régulières, propose de nouveaux opérateurs pour grouper les sous-tâches dans une méthode. Nous appellerons ces groupements de tâches des macro-tâches, et elles viendront compléter les macros existantes (`ordered`, `unordered`, `parallel`).

Le besoin initial a été pour les méthodes dont la décomposition s'effectuait en un nombre inconnu *a priori* d'exécution d'une sous-tâche (ou d'un groupe de sous-tâches) donnée. En nous inspirant de la notation étoilée des expressions régulières, nous avons mis en place un mot clé pour les gérer : `while`. L'implémentation de ceci nous a éloigné des expressions régulières pour nous rapprocher de la programmation structurée, avec la notion de boucle pour remplacer la plupart des récursivités, d'où l'appellation.
La définition de ce mot clé, `while`, nous permet de forcer le planificateur à effectuer un groupe de sous-tâches de façon répétée tant qu'une condition est valide.

Cet ajout nous a amené à enrichir le formalisme d'un certain nombre d'autres extensions pratiques :

- `forpar`, qui, pour tous les objets vérifiant une certaine propriété, effectue un groupe d'actions en parallèle ;
- `optional`, qui permet de laisser au planificateur la possibilité d'insérer ou non un groupe d'actions dans le plan ;
- `pickone`, qui permet de définir un point de choix et une portée de définition d'une variable locale.

Ces blocs, que nous appellerons macro-actions peuvent s'imbriquer les uns dans les autres pour faire l'équivalent d'une procédure pour chacune des méthodes.

L'expression en BNF d'une méthode s'écrit maintenant :

```
<method> ::= ":method" <name> ":precondition" <cond> ":subtasks" <macro>
<macro> ::= "(" <task> <fExp>* ")"
  | "(:ordered" <macro>+ ")"
  | "(:unordered" <macro>+ ")"
  | "(:parallel" <macro>+ ")"
  | "(:while" <cond> <macro> ")"
  | "(:forpar" "(" <typedVarLst> ")" "with" <cond> <macro> ")"
  | "(:optional" <macro> ")"
  | "(:pickone" "(" <typedVarLst> ")" "with" <cond> <macro> ")"
```

où `<task>` est le nom d'une tâche ; `<fexp>` représente une expression à base de fonctions, en logique L2, et définissant les paramètres de la tâche en fonction des variables et de l'état courants et `<cond>` est une condition booléenne.

2.3.3 Utilisation

La macro `while` servira donc à gérer les répétitions multiples d'une sous-macro ou d'une tâche. La condition de continuation est définie en premier. Il n'est pas

nécessaire, contrairement à la programmation structurée, de garantir que cette condition deviendra nécessairement fausse après un nombre fini d'itérations. En effet, le planificateur, contrairement à un programme, explore plusieurs branches plus ou moins simultanément (avec une stratégie dépendant de l'algorithme) ce qui lui permettra de trouver une branche dans laquelle la boucle se termine (ou qui esquivera la boucle si cela est plus judicieux).

La macro `pickone`, souvent couplée avec la macro `while`, permet de faire un choix de variable à un instant donné, avec une portée de définition interne à la macro. Elle permet de donner des stratégies de choix d'une variable et de garantir que celle-ci vérifie bien certaines propriétés au moment du choix, et qu'elle ne soit pas modifiée jusqu'à la fin de la macro.

La macro `optional` permet de définir qu'une sous-macro est optionnelle, et peut donc ne pas être employée par le planificateur s'il le juge opportun. Il faut tout de même faire attention à ne pas encapsuler un `optional` directement dans un `while`, sans autre action nécessaire, car suivant le planificateur utilisé à un niveau, il pourrait considérer que l'exécution d'une boucle vide amène à un état différent (de par sa nouvelle position dans l'arbre HTN), sans pour autant que le coût ou l'heuristique associés à l'état n'aient changé, et continuer à considérer celui-ci comme un état prioritaire à développer, et finir ainsi dans une boucle infinie. Dans notre algorithme (que nous présenterons plus loin), nous avons implémenté une sécurité forçant une exécution d'une boucle d'un `while` à effectuer au moins une action, sous peine de considérer la branche comme invalide.

La macro `forpar` (contraction de "for parallèle") s'utilise pour faire exécuter une macro en simultané à un groupe d'objets. Ces objets sont l'ensemble des objets d'un type donné vérifiant une condition (qui pourra être l'appartenance à un groupe par exemple).

2.3.4 Exemple

Reprenons maintenant notre exemple de parcours de graphe avec ces macro-tâches. Comme nous l'avons vu, un `moveTo` se décompose en une séquence de `jumpTo` vers des points d'arrivée de sauts connus comme atteignables (avec le prédicat `exists_path`), et chaque `jumpTo` se décompose en une séquence de `goTo` vers des points voisins. Une représentation graphique en est faite en figure 2.3. Le `while` est noté avec une flèche étoilée, notation inspirée des expressions régulières, tandis que les indices i et j représentent les variables de boucle, définis par les `pickone` (les domaines d'évolution de ceux-ci ne sont pas représentés pour des questions de lisibilité). Une branche optionnelle serait notée avec un point d'interrogation, tandis que un `for` parallèle utiliserait un symbole parallèle suivi d'une étoile.
Ici, on exprime que la tâche `moveTo ?l` (qui a comme effet de se retrouver en `?l`) se décompose comme une séquence d'un nombre inconnu de `jumpTo` $?l_i$ (les points $?l_i$ sont définis pour chaque `jumpTo`). Et de même, chaque tâche `jumpTo` $?l_i$ se décompose comme une séquence de `goTo` $?l_{ij}$.

On se rend rapidement compte du gain en compréhension et en facilité de

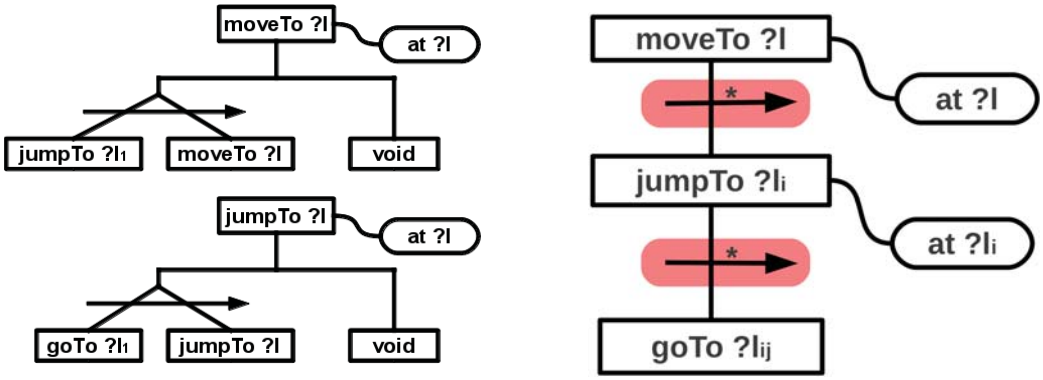


FIGURE 2.3 – Évolution de la représentation HTN du parcours de graphe avec les macro-tâches. La figure de gauche présente la représentation sans macro-tâches, tandis que la figure de droite force la répétition des tâches `jumpTo` et `goTo`. Les indices i et j indiquent les variables de boucles, en pratiques choisies avec un `pickone`.

modélisation de cette extension. D'une part, on exprime directement la notion voulue, sans avoir à la transposer en récursif et gérer le cas terminal, et d'autre part, en voyant le schéma, on comprend tout de suite de quoi il est question.

Le même gain en facilité se retrouve lors de l'écriture et la lecture du domaine dans notre formalisme. Comme on peut le voir ci-dessous pour la décomposition d'une tâche `jumpTo`, nous n'avons plus besoin que d'une seule méthode, applicable dans tous les cas, et qui définit que tant qu'on n'est pas à la position d'arrivée du `jumpTo` (`?l`), on répète une sélection d'un point (`?l1`) voisin à la position courante (`at`), et un `goTo` vers ce point.

```
:method raffiner
  :precondition
    ()
  :subtasks
    (:while (not (= (at) ?l))
      (:pickone (?l1 - loc) with (elem ?l ?l1)
        (goTo ?l1)
      )
    )
```

Comparée à la modélisation HTN (rappelée ci-dessous), nous n'avons plus besoin de gérer le cas terminal et n'avons donc plus besoin que d'une seule méthode. Cette méthode permet de rassembler aisément le concept de répétition (seulement implicite dans la version récursive des HTN), le lien d'enchaînement entre les tâches avec la fonction `pickone`, alors qu'il fallait aller chercher ce lien dans les préconditions, ainsi que la condition de sortie. De plus, aucun piège (tel que le sens de la récursion) n'est posé à l'expert qui peut modéliser son domaine sans se poser trop de questions

parasites.

```
:method terminé
  :precondition
    (= ?l_0 ?l)
  :subtasks
    ()
:method continuer
  :precondition
    (and
      (not (= ?l_0 ?l)
      (elem ?l_0 ?l_1)
    )
  :subtasks
    (ordered
      (goTo ?l_1)
      (jumpTo ?l)
    )
```

2.4 Formalisation générale d'un couple domaine-problème

Après avoir précisé les ajouts que nous effectuons sur le formalisme, nous allons voir comment ils s'agencent pour former les fichiers de domaines et de problèmes qui permettront au planificateur de fonctionner.

Un fichier de domaine est défini tel que suit :

```
<domain> ::=
  "(define (domain" <name> ")"
  "(:requirements" <req>+ ")"
  "(:types" <type>+ ")"
  "(:functions" <funct>+ ")"
  <task>+
  ")"
```

Il contient donc un nom ; une liste de fonctionnalités requises (liste de mots clés définissant un certain nombre de fonctionnalités que le planificateur doit posséder pour être en mesure de planifier sur le domaine qui va suivre. Ces fonctionnalités peuvent être la gestion des variables numériques, la temporalité...) ; la liste des types, qui peuvent être des sous-types d'autres types (voir plus bas pour la formalisation) ; la liste des fonctions disponibles pour le planificateur, prenant des paramètres typés et ayant un type de retour (booléen si omis) ; et enfin la liste des tâches (actions ou méta-actions) disponibles.

```
<type> ::= <name>
  | <name>+ "-" <name>
<funct> ::= "((" <name> <typedVarLst> ") -" <name> ")"
  | "(" <name> <typedVarLst> ")
```

```
<task> ::= <action>
  | <metaAction>
<action> ::=
  "(:action" <name>
  ":parameters (" <typedVarLst> ")"
  ":precondition" <cond>
  ":effect" <effect>
  ")"
<metaAction> ::=
  "(:metaAction" <name>
  ":parameters (" <typedVarLst> ")"
  ":precondition" <cond>
  <method>+
  ":metaEffect" <effect>
  ":heuristic" <fExp>
  ")"
<method>
  ":method" <name>
  ":precondition" <cond>
  ":subtasks" <macro>
```

Une tâche est soit une action, soit une méta-action. Les deux ont une liste de paramètres (typés) et une précondition sous forme de formule booléenne. Une action possède en plus un effet, sous la forme définie précédemment. Une méta-action possède une liste de méthodes, elles aussi soumises à précondition et avec la décomposition sous forme de macro (voir précédemment pour la définition) ainsi qu'un méta-effet et une fonction heuristique propre.

Le fichier de problème, qui contient les informations spécifiques à l'instance se définit comme suit :

```
<problem> ::=
  "(define (problem" <name> ")"
  "(:domain" <name> ")"
  "(:objects" <obj>+ ")"
  "(:init" <valFunct>+ ")"
  "(:goal" <cond> ")"
  "(:metric" <fExp> ")"
  "(:heuristic" <fExp> ")"
  "(:run (" <task> <fExp>* "))"
  ")"
```

Le fichier de problème commence par une référence au domaine associé ainsi que le nom du problème. Il contient ensuite la liste des objets typés utilisés dans le problème et l'état initial, sous forme d'une liste de fonctions instanciées et de prédicats (tous vrais, selon l'hypothèse du monde clos). De plus, il contient le but du problème, écrit sous forme de condition booléenne et une métrique pour indiquer au planificateur

quelle fonction maximiser ou minimiser. Enfin, le fichier contient une heuristique sous forme d'expression numérique à évaluer, et la tâche (instanciée) à exécuter en premier lieu (elle constituera le plan de plus haut niveau).

Un exemple de couple domaine-problème est donné en annexe A.2 pour le domaine de parcours de graphe présenté précédemment.

Conclusion

Nous avons proposé dans cette partie deux extensions au formalisme HTN. Celles-ci permettent d'une part pour un modélisateur humain d'entrer beaucoup plus d'information dans le modèle pour un coût qui nous semble assez faible devant le coût nécessaire pour concevoir le HTN en lui même. D'autre part, elles simplifient le processus d'écriture de ce HTN en le rendant, à notre avis, bien plus cohérent d'un point de vue structure hiérarchique et bien plus lisible. Nous avons de plus exposé le formalisme dans son ensemble, tel qu'il sera utilisé par l'algorithme présenté dans la partie suivante. Ce formalisme a été présenté à la communauté scientifique aux Journées Françaises de Planification, Décision et Apprentissage [Schmidt 2010].

Chapitre 3

Algorithmes

Sommaire

3.1 Planifier au sein d'un niveau 55
3.1.1 Idée générale 56
3.1.2 Planification par niveau avec A* 57
3.1.3 Ajout de la notion de parallélisme de tâche 64
3.2 Gérer les interactions entre les niveaux 65
3.2.1 Idée générale 65
3.2.2 Communication vers les niveaux inférieurs 67
3.2.3 Backtracks 68
3.2.4 Détection précoce des sur-coûts et communication entre les niveaux 69
3.3 Propriétés 71
3.3.1 Cadre d'application 71
3.3.2 Propriétés dans les situations favorables 72
3.4 Déroulement sur un exemple 75

Après avoir présenté les extensions aux HTN qui définissent notre formalisme, nous allons nous attarder sur les algorithmes que nous avons conçus pour mettre en place nos concepts de planification par niveaux.

Nous décrirons ces algorithmes en deux temps. Dans un premier temps (cf. section 3.1), nous verrons comment, à partir d'un plan de haut niveau hiérarchique, nous pouvons calculer un plan de plus haute précision. Dans un second temps (cf. section 3.2), nous verrons comment nous faisons interagir les différents niveaux hiérarchiques afin d'obtenir le plan final.

Une fois réunis, ces deux algorithmes forment l'algorithme HDS (Hierarchical Deepening Search).

3.1 Planifier au sein d'un niveau

Nous supposerons dans cette section que le planificateur de niveau n, niveau étudié, a accès à un plan de niveau supérieur déjà construit pour le guider dans sa planification. Nous appellerons ce plan $\mathcal{P}_{n-1}$.

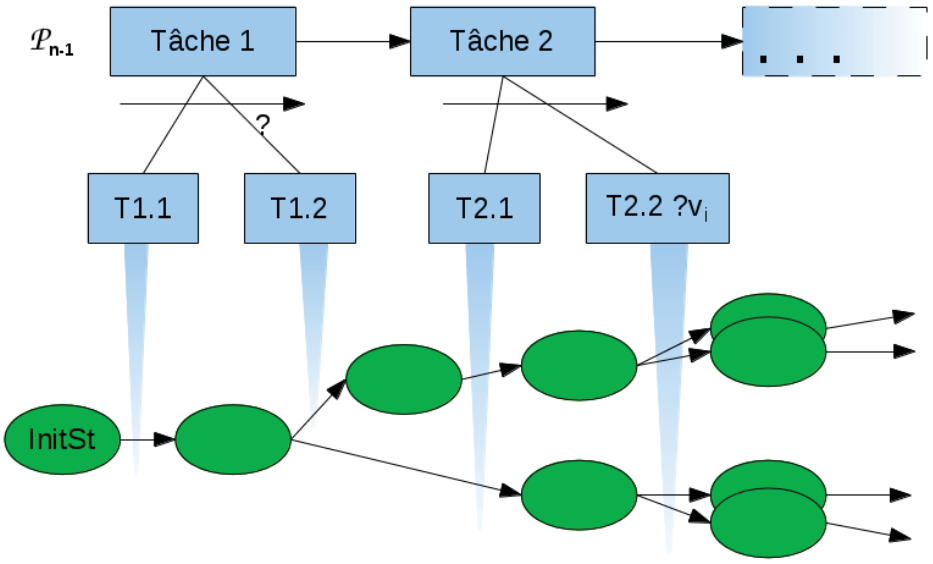


FIGURE 3.1 – Planification au sein d'un niveau. Les cadres représentent les tâches et actions, qu'elles soient du plan supérieur $\mathcal{P}_{n-1}$ pour la première ligne ou appliquées par le planificateur pour la seconde. Les cercles représentent les états atteints par le planificateur en appliquant lesdites tâches ou actions.

3.1.1 Idée générale

L'idée générale de cet algorithme est d'utiliser un algorithme de planification *en avant* classique, mais de lui faire considérer les tâches du problème comme étant des actions élémentaires. De plus, nous allons restreindre son facteur de branchement en utilisant le plan $\mathcal{P}_{n-1}$ et en limitant les séquences d'actions possibles dans les différentes situations aux décompositions possibles des tâches en cours de développement.

Pour ce faire, nous allons modifier un algorithme de planification classique de façon à ce que les états sur lesquels il travaille contiennent d'une part l'état courant, mais aussi une pile d'exécution qui correspond à la tâche du plan $\mathcal{P}_{n-1}$ en train d'être déroulée. Cette pile d'exécution donnera au planificateur la liste des actions qui sont autorisées dans l'état courant. Il lui faudra donc, lors du développement d'un état, sélectionner parmi les actions permises par cette pile d'exécution celles dont les préconditions sont valides, puis les exécuter pour ouvrir les nouveaux nœuds. Lors de l'ouverture de chacun des nouveaux nœuds, l'algorithme effectuera un suivi dans la pile d'exécution pour refléter l'avancement dans la méthode en cours. C'est une idée similaire qui a été utilisée pour effectuer la planification HTN probabiliste [Kuter 2005] et nous nous en sommes inspirés pour mettre au point notre algorithme.

La figure 3.1 illustre ce principe. L'état initial (`initSt`) se développe en son état suivant en utilisant la seule sous-tâche possible pour commencer la tâche 1 du niveau supérieur, L'état obtenu se développe de deux façons suivant que l'on applique ou pas la sous-tâche optionnelle (`T1.2`). La tâche `T2.2` est une tâche avec une variable

à instancier (`?v`$_i$). A chaque fois qu'un état doit utiliser cette tâche pour développer ses fils, il utilise toutes les instanciations autorisées de cette variable et créé les états correspondants, pour peu que les préconditions de `T2.2` soient vérifiées.
Nous n'avons pas représenté dans cet exemple le cas d'un choix d'une méthode ou d'un `while`, mais la gestion est similaire, avec un branchement effectué à ces endroits là.

Le choix de l'ordre dans lequel les états sont développés incombe uniquement à l'algorithme de planification choisi comme base. Par la suite, nous utiliserons A* (présenté dans la partie 1.1.3.2 de l'état de l'art) comme algorithme de planification en-avant de base pour la planification au sein d'un niveau, et montrerons comment nos idées s'appliquent dans ce cas. Les explications que nous donnerons seront volontairement le plus possible détachées de A* de façon à pouvoir le plus facilement possible les extrapoler pour tout autre algorithme de planification en avant.

3.1.2 Planification par niveau avec A*

3.1.2.1 Prérequis en terme de données

Pour pouvoir effectuer cette planification, il nous faudra comme pré-requis le plan de niveau supérieur, $\mathcal{P}_{n-1}$, composé d'actions et de tâches instanciées, ainsi que les coûts associés aux différentes tâches et actions dudit plan et le coût total de celui-ci. Ces coûts serviront à effectuer le calcul de la part h_t du calcul d'heuristique présenté en section 2.2.5.

D'autre part, le planificateur dispose, via les données du domaine, des fonctions heuristiques associées à chacune des tâches du plan $\mathcal{P}_{n-1}$, qui lui permettront de calculer la part h_s^t de l'heuristique.

En ce qui concerne le données permettant de construire le plan en suivant le modèle HTN, le planificateur dispose, toujours via le domaine, de la liste des méthodes de chacune des tâches de $\mathcal{P}_{n-1}$ et de leur décomposition respectives en sous-tâches. Nous avons choisi de stocker les décompositions en sous-tâches de chacune des méthodes suivant un arbre lexicographique, ce qui permettra de les exécuter facilement.

Le planificateur dispose de plus de la liste des actions du domaine. Cette liste d'actions est complétée par la liste des méta-actions, qui seront utilisées comme des actions standards dans cette phase. Pour ce faire, on mettra de côté leurs attributs de méthode et d'heuristique et on interprétera les méta-effets comme des effets standards.

Enfin, comme pour tout problème de planification, nous disposons de l'état initial et du but à atteindre.

Pour notre exemple de référence, le planificateur de niveau 1, planifiant les sauts a comme données d'entrée :

- le plan $\mathcal{P}_0$, donné non pas par le planificateur de niveau supérieur mais par le fichier de domaine. Ce plan ne contient qu'une tâche instanciée : `moveTo L` ;
- la fonction heuristique de la tâche `moveTo` qui est la distance à vol d'oiseau jusqu'à `L` ;

– la méthode de `moveTo` : une séquence de `jumpTo` ;
– les tâches `moveTo`, `jumpTo` et `goTo` modifiées pour être utilisables en tant qu'actions ;
– l'état initial (`(at) = A` et `(cost)` = 0) et le but à atteindre : `(at) = L`.

3.1.2.2 Utilisation de la pile d'exécution

Là où l'algorithme que nous présentons diffère de l'algorithme choisi comme base (ici A*) est le développement des états suivants lors de la fermeture d'un nœud. A* récupère la liste des actions possibles avec toutes leurs instanciations possibles, enlève de cette liste les actions instanciées dont les préconditions ne sont pas vérifiées dans l'état courant. Il applique ensuite les actions de cette liste à l'état, et place les états obtenus dans l'ensemble des ouverts s'ils n'ont pas déjà été explorés. Notre fonctionnement se rapproche bien plus de l'exécution d'un programme classique.

En effet, chaque nœud σ se compose non seulement de l'état courant, $\sigma.s$, mais aussi d'une pile d'exécution qui représente la position de l'état dans l'arbre HTN. Nous appellerons cette pile $\sigma.p$. Le planificateur, après avoir sélectionné le nœud à développer, analyse la pile d'exécution et l'arbre lexicographique donnant la recette de la méthode en cours de développement. Cette analyse lui permet de savoir quelles actions sont disponibles dans l'état courant.

Dans le formalisme des HTN, et de manière renforcée dans notre formalisme étendu, les choix, et donc le facteur de branchement, ne s'effectue plus au niveau des actions (dans un état donné, le planificateur peut directement trouver les actions applicables et développer les nœuds correspondants) mais au niveau des macro-actions : choix de la méthode à développer, choix de l'instanciation de variables à sélectionner dans un `pickone`, choix d'effectuer ou non une macro optionnelle... L'état courant n'étant pas modifié par ces choix, qui ouvrent à chaque fois de nouveaux nœuds de planification, il est illusoire de vouloir les ajouter dans l'ensemble des ouverts et sélectionner à chaque fois le nœud ouvert le plus intéressant. Nous avons donc choisi d'utiliser un algorithme de type profondeur d'abord pour le développement de ces macros. L'algorithme, lorsqu'il développe une macro à partir d'un nœud, créant ainsi plusieurs nœuds fils, va directement développer ces fils, sans repasser par l'ensemble des ouverts. Ce développement s'arrête, pour chacune des branches, quand le planificateur rencontre une action ou une tâche. Le calcul des effets s'effectue alors, et le nœud résultant est ajouté à l'ensemble des ouverts s'il n'y est pas déjà. Une fois toutes les branches terminées, les nœuds intermédiaires sont supprimés.

Le développement d'un nœud en fonction de la macro à effectuer se fait de la façon suivante :

– *Tâche de $\mathcal{P}_{n-1}$:* lorsque le planificateur, en développant sa pile d'exécution entre dans une tâche du plan supérieur, il teste les préconditions de cette tâche, puis, si elles sont valides, ouvre autant de nœuds qu'il y a de méthodes pour développer cette tâche et les empile sur la pile d'exécution. Chaque nœud ainsi créé est immédiatement développé.
– *Action de $\mathcal{P}_{n-1}$:* lorsque le planificateur entre dans une action élémentaire

incluse dans le plan supérieur, il en teste les préconditions, puis, si elles sont valides, applique ses effets sur l'état courant, et ajoute le nœud ainsi créé dans l'ensemble des ouverts.
- *Méthode :* lors de l'entrée dans une méthode, le planificateur teste les préconditions de celle-ci. Si elles sont valides, il crée un nœud dans lequel l'entrée dans la macro est ajouté sur la pile et développe la macro associée à la méthode.
- *Ordered :* lors de l'entrée dans une macro ordonnée, le planificateur crée un nœud dans lequel il ajoute sur la pile d'exécution la liste des sous-macros à développer, puis développe dans ce nœud la première des macros de la liste.
- *Unordered :* si la macro est non ordonnée, le planificateur crée un bag [1] contenant l'ensemble des sous-macros à exécuter, puis pour chacune d'elles, crée et développe un nouveau nœud avec la sous-macro extraite et le bag privé de ladite sous-macro ajouté sur la pile.
- *While :* lors de l'entrée dans un while, le planificateur teste si la condition d'entrée est bien vérifiée. Si oui, il ajoute la sous-macro sur la pile d'exécution et la développe immédiatement, sinon il développe la macro suivante de la pile d'exécution.
- *Optional :* lors de l'entrée dans une macro optionnelle, le planificateur ouvre deux branches, une où il sort directement de la macro et développe la suite de la pile d'exécution, une autre où il développe la sous-macro associée.
- *Pickone :* lors de l'entrée dans un pickone, le planificateur crée autant de nœuds que d'instanciations valides des variables du pickone. Pour chacun de ces nœuds, il rajoute sur la pile la sous-macro du pickone et étend la liste des variables courantes des variables instanciées par le pickone. Chacun des nœuds ainsi créé est immédiatement développé.
- *Méta-action :* si la macro à développer correspond à une méta-action (n'appartenant pas au plan supérieur), le planificateur en teste les préconditions, et si elles sont valides, applique les méta-effets à l'état courant avant d'ajouter le nœud ainsi créé à l'ensemble des ouverts.
- *Action :* si la macro à développer correspond à une action élémentaire, le planificateur en teste les préconditions, et si elles sont valides, applique les effets à l'état courant avant d'ajouter le nœud ainsi créé à l'ensemble des ouverts.

Dans le cas d'une macro *ordered* ou *unordered*, après avoir développé une de ses sous-macro, le planificateur développe la macro suivante, soit en prenant (et retirant) la première de la liste des macros à exécuter, soit en ouvrant autant de nœuds que ce qui est autorisé par le bag contenant les macros suivantes, et pour chacun de ses nœuds en prenant la macro en question.

Une fois qu'une macro est terminée, le planificateur doit remonter dans la pile d'exécution pour connaître l'action suivante à exécuter. Certains cas sont cependant à étudier dans le détail :

1. bag : il s'agit d'un conteneur d'éléments non ordonné pouvant contenir plusieurs occurrences d'un même élément.

- *While :* dans ce cas, le planificateur doit, avant de remonter à la macro parente, vérifier la condition de boucle. Si elle est vérifiée, le planificateur doit ouvrir à nouveau le while et exécuter ses sous-macros. Si elle n'est pas vérifiée, le planificateur remonte à la macro précédente dans la pile d'exécution. Afin d'éviter les boucles infinies, nous avons rajouté une garde supplémentaire à cette macro, qui est que lors de la sortie de cette macro, que la condition de boucle soit vérifiée ou non, si aucune action concrète (action élémentaire ou tâche) n'a été développée dans la dernière itération de la boucle, la branche de l'arbre en cours de développement est considérée invalide et la suite n'est pas développée.
- *La macro n'a pas de parent dans la pile :* ce qui veut dire qu'on arrive à la fin du plan de niveau supérieur. On teste alors la condition de but. Si celle-ci est vérifiée, on place le nœud courant dans l'ensemble des ouverts, et on déclarera le plan valide lorsqu'il sera fermé. Sinon on déclare la branche comme invalide.

Dans notre exemple de référence, la pile initiale du planificateur de niveau 1 est la première tâche du niveau précédent : la tâche instanciée `moveTo L`. Elle se développe immédiatement dans la seule méthode disponible, qui elle-même se développe en l'ouverture du `while`. Cette pile s'ouvre ensuite sur un choix à effectuer dans le `pickone`. Deux états sont alors développés, l'un en ayant effectué le saut vers `G` et l'autre pour le saut vers `H`. La pile d'exécution est alors :

- `task : moveTo L`
- `method : m0`
- `macro : while (not (= (at) L))`
- `macro : pickone ?l1 ← G`
- `action : jumpTo G`

On continue ensuite le développement de cette pile jusqu'à terminer le développement de la racine : `task : moveTo ?L`. Si le but est atteint en ce point, le plan sera considéré valide.

3.1.2.3 Choix du nœud à développer

Maintenant que l'on a défini le mécanisme de fermeture d'un nœud, nous allons définir comment est sélectionné, dans l'ensemble des ouverts, le nœud à développer.

Ici, nous revenons à l'algorithme A*. Le choix du nœud à fermer s'effectue toujours en prenant le nœud ouvert le plus prometteur, c'est à dire ayant le coût espéré $f = h + g$ le plus faible. Le calcul du paramètre g, représentant le coût minimum pour atteindre l'état courant depuis l'état initial, reste inchangé, mais la valeur heuristique est modifiée afin de prendre en compte le plan de plus haut niveau.

Comme expliqué précédemment, nous définissons h comme $h = h_s^t + h_t$, avec h_s^t la valeur heuristique, définie pour chaque tâche, nécessaire pour terminer la tâche, et h_t l'estimation du chemin restant à parcourir une fois la tâche terminée, pour atteindre le but. h_t est donné directement par le plan $\mathcal{P}_{n-1}$ en regardant le coût nécessaire avec les tâches de haut niveau pour atteindre le but depuis la fin de la

tâche courante.

Dans l'exemple précédent, lorsque le choix se pose entre le développement de `G` ou de `H`, le coût pour atteindre chacun des points est égal (environ 3 pour chacun si l'on considère que chacun des déplacements élémentaires est de longueur 1 sur le dessin), et l'heuristique de tâche nous donne :

- pour `G` : un coût d'environ 2 pour atteindre `L`, plus un coût nul pour terminer le plan une fois la tâche `moveTo` terminée (c'est la dernière tâche du plan) ;
- pour `H` : un coût un peu inférieur à 3 pour atteindre `L` plus un coût de zero.

Le nœud `G` est donc le nœud choisi pour être développé dans notre cas.

3.1.2.4 Gestion des nœuds dupliqués et recherche du meilleur ouvert

Dans un algorithme de recherche arborescente classique, tel que A*, l'ajout d'un nœud dans l'ensemble des ouverts ne se fait que si un nœud correspondant au même état n'est pas déjà présent dans l'ensemble des ouverts ou des fermés. En effet, comme le plan optimal pour atteindre le but à partir d'un état ne dépend que de celui-ci et non du chemin qui a été choisi pour atteindre cet état (hypothèse markovienne), dupliquer un nœud revient à faire les calculs en double à partir de ce nœud.

Malheureusement, dans le cadre HTN, ceci n'est plus vrai.
D'une part, un même état rencontré à deux positions différentes dans l'arbre HTN n'auyant pas les mêmes chemins autorisés pour les développements futurs, il est indispensable de ne pas fusionner deux nœuds comprenant le même état et se trouvant dans deux positions distinctes de l'arbre HTN.
D'autre part, même si à un niveau donné le chemin choisi pour atteindre un état donné (à la position dans le HTN fixée) n'influe pas sur les chemins possibles à partir de cet état, ce n'est plus vrai pour les niveaux plus fins. En effet, des chemins différents peuvent faire varier les effets dans leurs détails, et deux états qui étaient considérés identiques à faible niveau de granularité se retrouvent différents, et donc avec des chemins possibles différents, une fois étudiés plus en détails. Les fusionner poserait donc problème lorsque l'on chercherait à effectuer un backtrack d'un niveau très détaillé vers un niveau plus grossier.

C'est pourquoi le concept de nœuds dupliqués n'a plus de valeur dans notre situation, et que nous ne testerons pas si un état a déjà été développé ou non avant de l'ajouter dans la liste des ouverts. Notre liste d'ouverts devenant alors beaucoup plus grande, la recherche du nœud le plus prometteur devient bien plus longue que dans le cas classique. Nous avons alors modifié cette partie de l'algorithme afin d'accélérer cette recherche.

La recherche du meilleur ouvert s'effectue maintenant de manière récursive, en parcourant le plan partiel déjà produit. Chacun de nos nœuds contient la liste de ses nœuds fils (i.e. ceux atteignables en une action à partir de celui-ci). Pour trouver le nœud à développer, on part du nœud racine, correspondant à l'état initial, et on sélectionne son nœud fils le plus prometteur (coût pour arriver jusqu'à celui-ci plus estimation du coût pour atteindre le but), et ainsi de suite jusqu'à atteindre un nœud

non encore développé. Ensuite, l'algorithme récursif, sur un nœud donné, après avoir développé son fils le plus prometteur, met à jour sa propre valeur heuristique. Pour ce faire, il sélectionne son nouveau fils le plus prometteur, récupère son estimation du coût du meilleur plan possible, et en effectuant la différence avec son propre coût depuis l'état initial, met à jour son estimation :

$$h_n \leftarrow (h_{best} + g_{best}) - g_n$$

Cette mise à jour de l'estimation permettra au planificateur, lors de la passe suivante de trouver le nouveau meilleur plan courant.

Cet algorithme reste équivalent à celui de A*, car il garantit que ce sera bien l'ouvert le plus prometteur qui sera sélectionné.

3.1.2.5 Algorithme final

L'algorithme résultant est décrit dans les algorithmes 5, 6 et 7.

Algorithm 5: initPlanner(s_0, $\mathcal{P}_{n-1}$)

```
1 begin
2     init_node.σ.s ← s0;
3     init_node.σ.p ← ∅;
4     init_node.succs ← ∅;
5     goalReached ← false;
6     while root.cost< ∞ ∧ ¬goalReached do
7         goalReached ← aStarRecPlanner(init_node, P_{n-1});
8     return extractPlan(root);
```

L'algorithme 5 correspond à la phase d'initialisation, où l'on crée le nœud initial avec l'état initial (s_0) (lignes 2 à 4), puis on lance l'algorithme récursif de développement de l'arbre A*, jusqu'à ce qu'une solution soit trouvée, ou que l'estimation de coût restant associée à l'état initial atteigne $+\infty$, c'est à dire que l'algorithme décide qu'aucun plan n'est possible (lignes 6 et 7). Le plan final est alors extrait à partir de l'état initial, en utilisant le même type de parcours que celui utilisé pour trouver le meilleur ouvert, et est retourné (ligne 8). Retourner un plan (non solution) même si aucun plan n'a été trouvé peut être utile dans certains cas pour aider à comprendre ce qui, dans le modèle, n'a pas permis au planificateur de trouver une solution.

L'algorithme 6 gère la phase de recherche et d'extension du meilleur ouvert. Si un nœud est un ouvert, il appelle l'algorithme d'extension pour récupérer les nœuds fils (ligne 3), sinon, il cherche le nœud fils le plus prometteur et recommence le processus avec ce nœud (ligne 10). Il cherche ensuite parmi les nœuds fils le coût total du nouveau plus prometteur, et met à jour son estimation de distance au but avec cette donnée (ligne 14).

Enfin, l'algorithme 7 gère la phase d'extension de l'ouvert sélectionné. Pour ce faire, via la pile d'exécution, il récupère les nœuds suivants (ligne 3). S'ils correspondent

Algorithm 6: recPlanner(node, $\mathcal{P}_{n-1}$)

```
1  begin
2      if node.succs = ∅ then
3          node.succs ← extend(node,P_{n-1});
4          goalReached ← false;
5      else
6          if satisfies_goal(node.σ.s) then
7              goalReached ← true;
8          else
9              node' ← argmin_{n'∈node.succs}(n'.cost+n'.estim);
10             goalReached ← recPlanner(node',P_{n-1});
11     if node.succs = ∅ then
12         node.estim = ∞
13     else
14         node.estim ← min_{n'∈node.succs}(n'.cost+n'.estim) - node.cost;
15     return goalReached;
```

Algorithm 7: extend(node, $\mathcal{P}_{n-1}$)

```
1  begin
2      succs ← ∅;
3      M ← next_from_stack(node.σ.p);
4      forall the m ∈ M do
5          if is_action(m) then
6              if valid(m.precond,node) then
7                  node'.σ.s ← apply(a.effects, node.σ.s);
8                  node'.σ.p ← extend_stack(node.σ.p, m, P_{n+1});
9                  node'.cost ← node.cost + cost(node.σ.s, a, node'.σ.s);
10                 node'.estim ← heurist(node'.σ,P_{n-1});
11                 node'.succs ← ∅;
12                 succs ← succs ∩ {node'};
13         else
14             node'.σ.s ← node'.σ.s;
15             node'.σ.p ← extend_stack(node.σ.p, m, P_{n+1});
16             node'.succs ← ∅;
17             succs ← succs ∩ extend(node',P_{n-1});
18     return succs;
```

à des actions (action élémentaire ou méta-action à développer comme une action), il calcule les nœuds résultants et les renvoie (lignes 6 à 12). Si les nœuds suivants

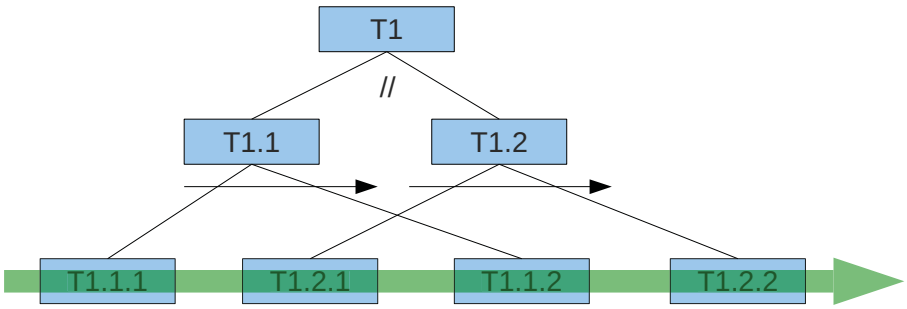


FIGURE 3.2 – Entrelacement de sous-tâches dans le cas de tâches parallèles.

correspondent à des macros autres, il les développe en se rappelant de manière récursive, mettant à jour la pile d'exécution à chaque étape (lignes 14 à 17).

3.1.3 Ajout de la notion de parallélisme de tâche

Le planificateur, tel que défini précédemment ne permet pas de gérer les mots clef `parallel` et `forpar`. En effet, notre algorithme se base principalement sur A*, qui ne permet pas de gérer les actions parallèles.

Cependant, le parallélisme de tâche tel que défini par ces macros est moins contraignant que le parallélisme d'actions au sens classique du terme. Il veut juste dire que les tâches sont développées en parallèle, et que donc leurs sous-tâches peuvent être entrelacées dans le plan final. La figure 3.2 montre une exemple d'un tel entrelacement.

Notre idée est donc de définir qu'une macro `parallel` va se développer de la même façon qu'un `unordered` dans le plan au niveau de sa première apparition, et que lors de l'extraction du plan, ces tâches seront re-parallélisées (c'est à dire considérées comme s'exécutant en parallèle). Cette re-parallélisation correspond plus à un relâchement des contraintes de séquentialité de ces tâches qu'à une obligation de les exécuter en parallèle. C'est pour cette raison que nous nous permettons de le faire de façon systématique sans vérifier que les pré- et post- conditions ne permettent vraiment de faire du parallélisme.

Au niveau inférieur, si le planificateur rencontre plusieurs tâches en parallèle dans plan $\mathcal{P}_{n-1}$, il ouvre la pile d'exécution en autant de sous-branches qu'il y a de branches parallèles (elle devient alors un arbre d'exécution, que nous continuerons à appeler pile d'exécution par abus de langage). Lors du développement d'un nœud ouvert ayant plusieurs branches parallèles ouvertes dans la pile, le planificateur peut exécuter toutes les macros disponibles dans chacune des branches parallèles, permettant ainsi de réaliser tous les entrelacements possibles des sous-tâches de tâches parallèles. Lors de la fermeture d'un nœud parallèle de la pile d'exécution, le planificateur s'assure que toutes les branches parallèles sont bien terminées. La fermeture ne peut s'exécuter sinon.

Nous avons vu ici comment nous avons adapté A* de façon à ce qu'il puisse, à partir d'un arbre HTN, raffiner un plan composé de tâches en un plan plus précis composé de sous-tâches, qui peuvent, éventuellement, être des actions élémentaires. Nous avons, de plus, décrit la construction de cet algorithme de façon indépendante de l'algorithme support, afin que l'on puisse réutiliser cette démarche et l'adapter à tout autre planificateur de type *en avant*. Contrairement à Nonlin et O-Plan, nous basons notre planification de niveau sur une restriction du domaine des actions possibles à chaque étape pour le planificateur plutôt qu'une insertion de la décomposition d'une tâche dans le plan, ce qui nous permet d'être compatible avec les planificateurs en avant de la planification classique, contrairement à Nonlin qui cherche à avoir une entrée compatible avec un ordonnanceur. Dans la suite, nous montrerons comment coupler plusieurs planificateurs de niveau entre eux afin de générer un plan hiérarchique.

3.2 Gérer les interactions entre les niveaux

Ayant montré comment générer des plans de niveau hiérarchique donné, nous allons montrer comment mettre en place la communication entre les différents niveaux afin de pouvoir planifier hiérarchiquement. Cette planification hiérarchique, qui constitue le cœur de notre algorithme général, s'effectue en deux phases. La première consiste à communiquer le plan trouvé et les informations associées au niveau inférieur, afin que le planificateur de niveau puisse planifier, utilisant ces informations comme un guide. La seconde, quand à elle, remonte les informations concernant les difficultés rencontrées aux niveaux de granularité plus fins pour un possible backtrack.

3.2.1 Idée générale

L'idée générale de cette planification est d'adapter un algorithme de recherche arborescente à la recherche dans la hiérarchie. On transforme une recherche basée sur des états et des actions en une recherche basée sur des plans et des raffinements (chaque raffinement utilisant le plan précédent comme guide). L'état initial consiste alors en un plan constitué de la tâche de plus haut niveau, et le facteur de branchement de l'arbre est défini par les sorties possibles du planificateur décrit en section 3.1 avec le plan de niveau en entrée. Bien entendu, il serait illusoire de stocker tous les plans évalués par les différents planificateurs de niveau ainsi que les structures de données utilisées pour reprendre une planification en cours. Nous verrons donc quelles stratégies et structures de stockages nous avons utilisées afin de pouvoir quand même procéder à une recherche arborescente.

La figure 3.3 présente un cheminement de planification hiérarchique arborescente. Les choix du planificateur sont ici symbolisés par le choix de la méthode à appliquer dans le HTN, et sont marqués en gris les plans qui se sont révélés trop coûteux et qui ont poussés le planificateur à explorer d'autres branches.

Nous avons donc un planificateur que nous appellerons global, qui gère le lancement des planificateurs de niveau ainsi que la communication entre eux, et c'est le

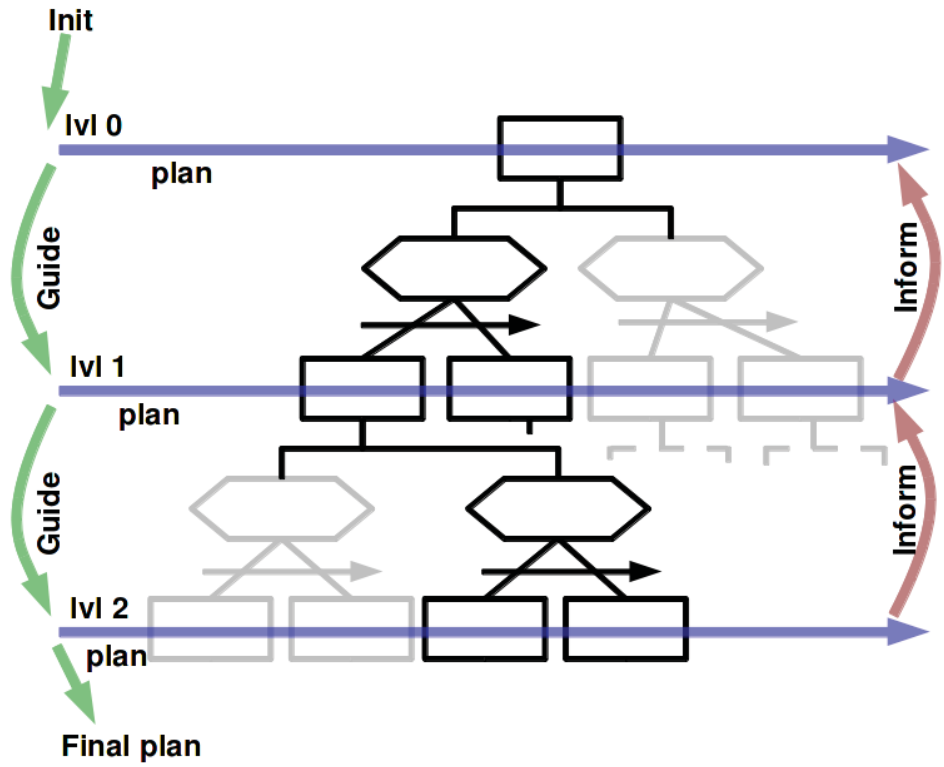


FIGURE 3.3 – Interaction entre les niveaux.

Algorithm 8: Global planner

```
1  begin
2      P_0 ← run_task;
3      n← 0;
4      while (cost(P_0) < ∞) ∧ (¬is_elem(P_n)) do
5          P_{n+1} ← initPlanner(s_0, P_n);
6          max_cost ← cost(P_{n+1});
7          last_change←n+1;
8          for i ← n to 1 step −1 do
9              if cost(P_i) ≥ max_cost then
10                 break;
11             old_P_i ← P_i;
12             reeval(P_i);
13             if P_i ≠ old_P_i then
14                 last_change ← i;
15             max_cost ← cost(P_i);
16         n← last_change;
17     if cost(P_0) = ∞ then
18         return ∅;
19     else
20         return [P_0 : P_n];
```

fonctionnement de celui-ci, formalisé dans l'algorithme 8 que nous allons décrire en détails par la suite.

3.2.2 Communication vers les niveaux inférieurs

La première partie de cet algorithme est donc le raffinement d'un plan. Nous avons vu dans la partie précédente comment celui-ci s'effectuait, nous allons maintenant nous concentrer sur son intégration dans le processus hiérarchique.

L'initialisation de ce processus utilise l'attribut `run` du problème définissant la tâche de plus haut niveau à réaliser pour effectuer la mission. Cette tâche est alors considérée comme un plan à une seule action, et transmise au planificateur de niveau inférieur (ligne 2).

Lors de la transmission d'un plan à un niveau inférieur, le planificateur global lance un nouveau planificateur de niveau avec ce plan comme guide (ligne 5). Une fois la planification de ce niveau terminée, le planificateur lance une procédure de vérification par backtrack. Cette procédure, expliquée en détails un peu plus loin, permet de vérifier que le plan hiérarchique courant est bien toujours le meilleur au vu des informations actuelles. Si ce backtrack valide le plan actuel, le planificateur global lance un raffinement de plan s'il contient encore des méta-actions, ou renvoie

la plan validé si celui-ci ne contient que des actions élémentaires. Si un planificateur de niveau ne trouve pas de plan solution, seul le processus de backtrack est lancé.

3.2.3 Backtracks

Le processus de backtrack (des lignes 8 à 15) est un processus permettant de faire remonter aux planificateurs de haut niveau l'information découverte par les planificateurs de bas niveau. L'idée de ce backtrack est de permettre au planificateur global de trouver le plan de meilleur coût. Pour ce faire, à la manière de A*, il utilise le coût des plans de haut niveau comme estimation optimiste du coût total. Cette estimation optimiste lui permet de développer systématiquement la branche de plus bas coût estimé, et de garantir qu'il ne passera pas à côté de la branche optimale. Afin de ne pas consommer trop de mémoire, les branches écartées momentanément car trop coûteuses sont supprimées de la mémoire, et entièrement recalculées par la suite si nécessaire.

A la fin de chaque processus de planification, le planificateur global ajoute à la dernière tâche du plan trouvé par le planificateur de niveau immédiatement supérieur $(n-1)$ la différence de coût entre le plan de niveau n et celui de niveau $n-1$ (ou une valeur infinie si aucun plan n'a été trouvé). Le planificateur de niveau $n-1$ reprend alors son activité pour tenter de trouver le nouveau meilleur plan sachant ce nouveau coût. Ce report n'a lieu que s'il est vraiment nécessaire, c'est à dire si le coût trouvé au niveau n est strictement supérieur à celui trouvé au niveau $n-1$.

Pour ce faire, le nœud terminal du plan découvert au niveau $n-1$ est remis dans l'ensemble des ouverts avec son nouveau coût, et la planification reprend comme si elle n'avait pas fermé le nœud. Le planificateur peut alors soit refermer à nouveau ce nœud et considérer le plan inchangé (mis à part son coût), soit finir par fermer un autre nœud but et changer de plan solution (cet algorithme, non écrit ici, est appelé en ligne 12).

Ce nouveau coût, que le plan ait changé ou pas, est ensuite propagé au planificateur $n-2$ qui effectue la même manipulation, et ainsi de suite jusqu'au sommet de la pile ou jusqu'à ce que le coût que l'on cherche à remonter soit inférieur [2] ou égal à celui déjà trouvé par le planificateur supérieur (ce qui garantit qu'il n'y aura pas de remise en cause du plan à plus haut niveau).

Si aucun des planificateurs n'a remis en cause son plan précédent, le plan courant est validé et on tente de le raffiner.

Dès lors qu'un des planificateurs remet en cause son plan, le plan courant est considéré invalide. Tous les planificateurs en dessous de lui (donc plus fins) sont alors arrêtés et supprimés, car ils travaillent sur un plan en entrée qui n'est plus d'actualité.

Le processus de planification reprend alors à partir du dernier planificateur à avoir remis le plan en cause, c'est à dire celui de plus haut niveau parmi tous ceux

2. Cas possible si la valeur d'un nœud a été revue à la hausse précédemment, mais que la "raison" en a été momentanément oubliée à cause de l'exploration d'une autre branche entre temps.

ayant changé de plan. On utilise alors son nouveau plan solution pour le processus de raffinage de plans.

Le processus s'arrête soit quand un plan solution ne contenant que des actions élémentaires est trouvé, soit lorsque le planificateur de plus haut niveau ne trouve plus de solution, c'est à dire quand le problème en lui même n'a pas de solution.

3.2.4 Détection précoce des sur-coûts et communication entre les niveaux

Nous avons de plus créé un algorithme plus complexe permettant de remonter les informations au plus tôt afin d'isoler au mieux les tâches dont les effets ne sont pas les effets attendus. Cette version de l'algorithme permet aussi d'utiliser le multi-cœur des calculateurs pour effectuer le processus de backtrack en parallèle, c'est pourquoi nous l'avons appelé P-HDS pour "Parallel Hierarchical Deepening Search". Malheureusement, nous n'avons développé cet algorithme que très tardivement, et ne pouvons apporter le même niveau de formalisation que pour l'algorithme précédent, par manque de temps.

L'idée générale de cet algorithme est de conserver la planification par niveau vue dans le chapitre précédent, mais de rajouter des étapes de communication entre les niveaux au fur et à mesure de l'avancée de la planification du niveau courant.

Chaque planificateur de niveau devient alors un processus à part entière, ne se terminant pas à la fin d'une planification, mais se mettant en attente de nouvelles informations. Par la suite, à chaque fois qu'un planificateur de niveau (appelons le planificateur de niveau n) termine le développement d'une des tâches du plan de niveau supérieur $(n-1)$, celui-ci communique au planificateur de niveau $n-1$ l'effet exact obtenu (sous la forme d'un état).

Deux cas se présentent alors :

- si l'état communiqué au planificateur de niveau $n-1$ est égal à l'état obtenu lors de sa planification de haut niveau, l'ancien nœud est considéré comme confirmé, et rien ne se passe ;
- si l'état communiqué au planificateur de niveau $n-1$ est différent de l'état obtenu lors de sa planification de haut niveau :
  - un nouveau nœud est créé, comprenant ce nouvel état ;
  - il est inséré dans l'arbre de planification à côté de l'ancien nœud, créant ainsi une transition artificielle avec l'état précédent, car ne respectant pas les effets de la tâche utilisée ;
  - il est ensuite rajouté dans l'ensemble des ouverts ;
  - si l'ancien nœud n'est considéré comme confirmé, son coût est arbitrairement augmenté pour coller à celui du nouveau nœud, afin de montrer que d'après les planificateurs de bas niveau, ce nœud n'est pas atteignable en un coût inférieur, mais peut toujours l'être avec un coût plus élevé. Ce sur-coût est aussi propagé à toute la suite du plan ;
  - le processus de planification reprend, s'il n'est pas déjà en cours (suite à une autre modification précédente).

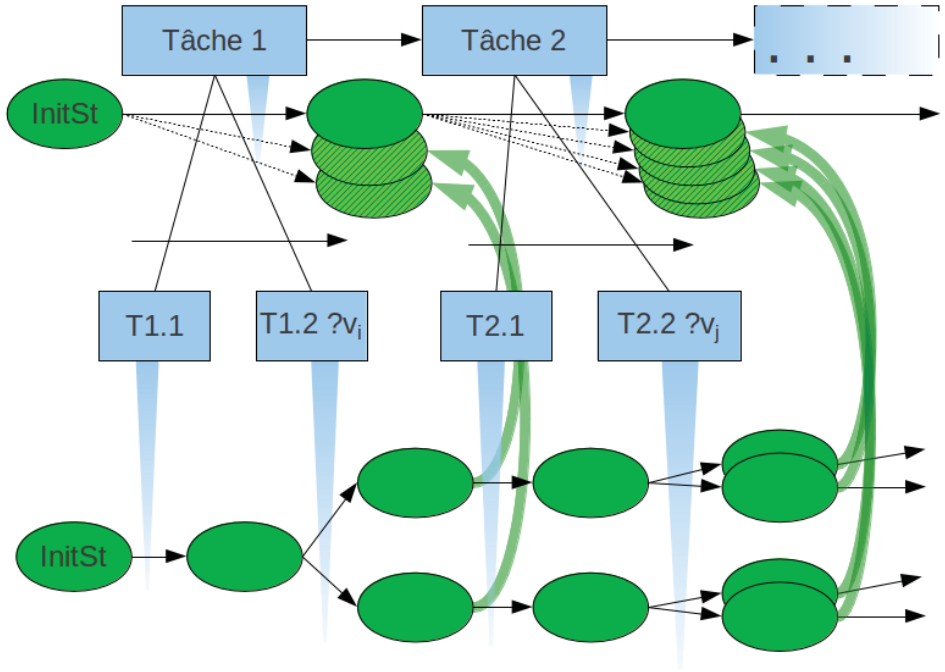


FIGURE 3.4 – Insertion d'états réellement atteints dans le planificateur de niveau supérieur à des fins de détection précoce de surcoûts.

Lors de la reprise de planification avec les nouveaux états ajoutés dans l'ensemble des ouverts, le planificateur se retrouve avec le choix de développer soit l'état qui a été dérivé des méta-effets fournis (avec un éventuel surcoût), soit de reprendre la planification à partir de l'état fourni par le plan de plus bas niveau, vu qu'on sait maintenant que cet état (plus précis) est atteignable. Ceci lui permet de développer une autre suite de plan avec d'autres tâches (toutes les préconditions valides dans le plan précédent ne le sont plus nécessairement, l'inverse étant faux avec l'hypothèse de méta-effets optimistes). D'autres plans peuvent alors être considérés comme étant plus prometteurs que l'ancien, étant donné le surcoût de l'ancien plan.

Nous illustrons ces insertions de nœuds en figure 3.4. Les nœuds représentés par les ovales hachurés sont des nœuds ajoutés à partir des états rencontrés dans le plan de bas niveau. Nous avons représenté par des flèches en pointillés les relations entre ces nœuds et leurs nœuds précédents, pour signifier que même s'ils sont considérés comme états suivants, leurs effets ne sont pas directement dérivés de la tâche.

De même que précédemment, dès lors qu'un planificateur de haut niveau trouve un nouveau plan, il arrête tous les planificateurs de niveau inférieurs, et relance un planificateur avec son nouveau plan.

Lors d'un backtrack dû à une impossibilité de trouver un plan valide, tous les nœuds du plan supérieur qui n'ont pas été soit confirmés soit modifiés par le planificateur de niveau inférieur sont considérés inatteignables. On leur attribue alors

un coût rédhibitoire pour qu'ils ne soient plus visités par le planificateur.

Le processus de planification se termine lorsqu'un planificateur a renvoyé un plan constitué uniquement d'actions élémentaires, et que tous les autres threads de planification ont arrêté leur processus de replanification en considérant qu'ils ne trouvaient pas mieux avec les nouvelles informations.

Chaque processus de planification/replanification est alors exécuté sur un thread séparé, ne communiquant que rarement. Ceci permet aux planificateurs de haut niveau d'effectuer des vérifications au fur et à mesure de la planification de plus bas niveau. Ces vérifications permettent d'une part de stopper le planificateur de plus bas niveau au plus tôt s'il est détecté qu'il travaille de façon inutile, et d'autre part détecter le point où les estimations des effets sont trop optimistes pour replanifier à partir de là, plutôt que de devoir considérer un plan dans son ensemble et ne pas savoir quel est le point qui pose problème. En effet, si c'est sur la première tâche du plan qu'il faut faire un changement, dans la version précédente de l'algorithme, il faudra attendre que tous les plans utilisant cette première tâche se révèlent, les uns après les autres, trop coûteux avant que cette première tâche ne soit remise en cause, ce qui n'est plus le cas ici.

De plus, si le nombre de cœurs de la machine est au moins égal au nombre de niveaux du HTN, chacun des planificateurs peut s'exécuter de façon parallèle, et donc ne pas ralentir le planificateur de niveau inférieur, alors que précédemment, nous étions obligés d'attendre que les replanifications soient terminées avant de commencer une planification de niveau inférieur. Nous évitions alors des calculs inutiles, mais étant donné qu'avec le parallélisme, les calculs inutiles ne pénalisent pas le reste du processus, ils ne posent pas de problème.

Comme dit précédemment, nous ne présenterons pas d'algorithmes écrits précisément, de preuves ou résultats formels pour cet algorithme, mais nous en présenterons les résultats par la suite.

3.3 Propriétés

Nous allons ici présenter les principales propriétés de notre algorithme [3]. Celles-ci dépendant de plusieurs critères, tels que l'écriture du HTN ou l'algorithme choisi pour la planification par niveau, nous ne pourrons pas en donner des preuves formelles, mais seulement les idées sous-jacentes.

3.3.1 Cadre d'application

Influence du HTN fourni

Les propriétés de cet algorithme dépendent en premier lieu de la façon dont le HTN a été écrit. Il est tout d'abord évident que si en suivant l'arbre HTN il n'est

3. Il s'agit ici des résultats pour la version sans les calculs des différents niveaux en parallèle, car nous n'avons pas eu le temps de faire les études nécessaires, mais il est probable que les propriétés soient similaires.

pas possible de trouver une solution au problème, alors le planificateur ne pourra pas trouver de solution. De même, si l'arbre HTN occulte les solutions optimales, le planificateur ne pourra trouver que des solutions dégradées au problème.

Définition 3.1 (*Décomposition complète*)

Une décomposition HTN d'un domaine est dite complète si pour tout problème `P` associé à ce domaine, si `P` admet une solution, alors il existe au moins une solution à ce problème atteignable par cette décomposition.

Définition 3.2 (*Décomposition optimale*)

Une décomposition HTN d'un domaine est dite optimale si pour tout problème `P` associé à ce domaine, si `P` admet une solution, alors il existe au moins une solution `S` à ce problème atteignable par cette décomposition telle que pour toute solution `S'` au problème `P` (non nécessairement atteignable par la décomposition HTN) le coût de `S'` est supérieur ou égal à celui de `S`.

De plus, comme on l'a vu précédemment, l'écriture des méta-effets joue un rôle important dans le parcours de l'arbre de raffinement. Si les méta-effets ne sont pas de type optimiste, c'est à dire sous-évaluant le coût à long terme, le planificateur peut considérer la branche contenant le plan optimal comme trop coûteuse et ne pas l'explorer, et ainsi perdre ses garanties.

Influence du planificateur de niveau

De façon tout aussi évidente, le planificateur est contraint par l'algorithme de planification choisi pour le planificateur de niveau. En effet, aucune propriété ne peut être vérifiée pour le planificateur global si elle n'est pas respectée par le planificateur de niveau.

3.3.2 Propriétés dans les situations favorables

Plaçons nous maintenant dans le cas favorable où le HTN fourni permet l'accès à la solution optimale si elle existe, et où le planificateur de niveau est A*, qui possède donc les propriétés de correction, complétion, optimalité et terminaison.

Théorème 3.1 (*Correction*)

L'algorithme de HDS est correct si le planificateur de niveau est A.*

Ceci repose sur le fait que le planificateur de niveau est A*, qui est correct. Le plan fourni passe en dernier lieu par l'algorithme A* qui planifie le niveau le plus précis. Sachant que A* travaille alors sur les actions élémentaires avec pour seules restrictions des restrictions sur le nombre d'actions à sa disposition dans chaque état, toutes les préconditions de toutes les actions insérées dans le plan sont vérifiées au cours de ce processus de planification. Le plan final produit par HDS est donc un plan qui est applicable car produit par A*. HDS utilisant A* comme planificateur de niveau est correct.

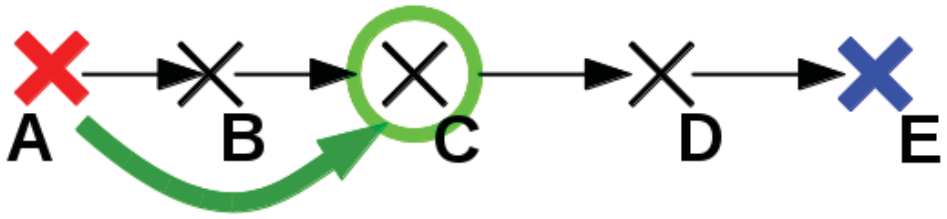


FIGURE 3.5 – Exemple de problème pour lequel la décomposition de l'exemple de référence est non complète. Ici, le planificateur n'a pas l'information qu'un saut est réalisable de `C` vers `E`, et ne sera donc pas capable de trouver de plan au niveau des `jumpTo`. Il conclura que le problème n'a pas de solution, alors qu'il en a effectivement une, trouvable par un planificateur classique.

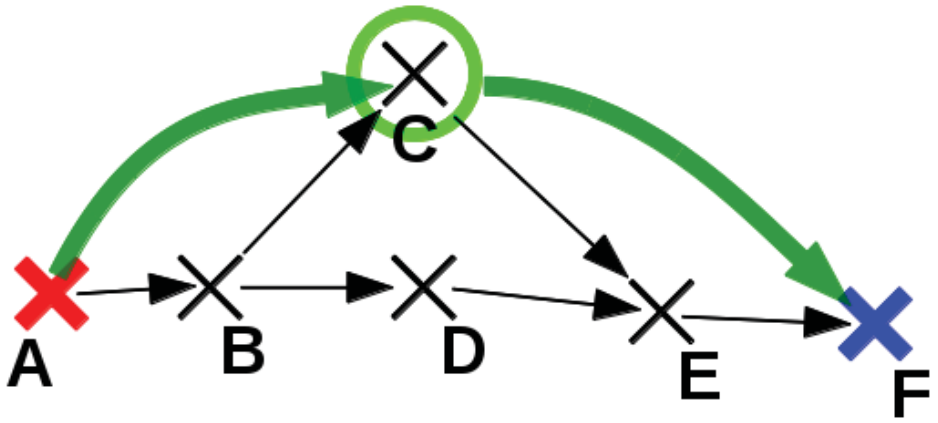


FIGURE 3.6 – Exemple de problème pour lequel la décomposition de l'exemple de référence est non optimale. Le planificateur de haut niveau fera passer son plan constitué de `jumpTo` par `C`, et le planificateur de bas niveau, contraint par le plan précédent fera de même, alors que le chemin serait plus court en passant par `D`.

Cette démonstration peut s'étendre à tout autre planificateur de niveau de type en avant qui serait correct.

Théorème 3.2 (*Complétion et optimalité*)

L'algorithme HDS est complet (respectivement optimal) si la décomposition HTN est complète (respectivement optimale) et les heuristiques utilisées ainsi que les méta-effets admissibles.

Cette propriété repose sur le fait que A* est complet et optimal, que les méta-effets sont optimistes et que le plan optimal est atteignable avec le HTN donné. Les méta-effets étant optimistes, le plan trouvé à chaque étape a forcément un coût

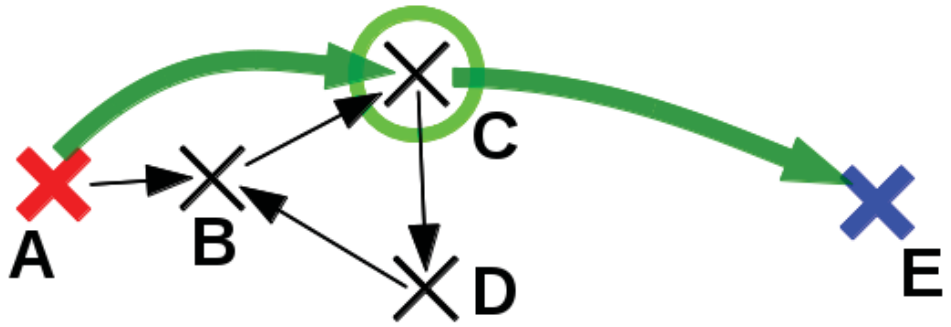


FIGURE 3.7 – Exemple de problème sans solution à bas niveau pour lequel le planificateur, sans détection des états dupliqués, tourne sans cesse dans la boucle `B, C, D` sans se rendre compte qu'il ne visite que des états déjà rencontrés, et donc ne termine pas.

inférieur ou égal à celui du plan optimal. Si à un moment donné un plan dépasse le coût optimal, le processus de backtrack forcera le planificateur de niveau ayant trouvé le mauvais plan de haut niveau à reprendre les calculs pour trouver une autre branche de plus bas coût. De même, si le planificateur conclut qu'il n'y a pas de solution (ce qui peut être vu comme un plan de coût infini), il cherchera à remettre en question les autres planificateurs de niveau afin de trouver un plan de coût plus faible, jusqu'à ce que le planificateur de plus haut niveau trouve que le plan solution a un coût infini, ce qui n'est possible, avec les meta-effets optimistes que si le problème n'a pas de solution. Les figures 3.5 et 3.6 présentent des problèmes sur lesquels notre décomposition de l'exemple de référence est non complète ou non optimale, et où le planificateur est incapable de trouver une solution, ou la solution optimale, alors que chacun de ces problèmes admet une solution trouvable par un planificateur classique.

Théorème 3.3 (*Terminaison*)

L'algorithme HDS n'a pas la propriété de terminaison.

Ici, même si le planificateur de niveau présente la garantie de terminaison, notre algorithme ne peut prétendre terminer en un temps fini dans le cas où un problème n'aurait pas de solution. Ceci vient du fait que nous ne considérons pas deux états identiques mais avec des plans différents comme un doublon, vu que le processus de raffinement pourrait produire des effets différents. Notre algorithme pourrait alors, dans certains cas, tenter de rallonger le chemin à l'infini pour essayer de trouver une solution, en visitant plusieurs fois le même état. La figure 3.7 présente un cas de problème où notre algorithme ne termine pas.

Nous conjecturons que sur un problème exempt de boucles, HDS terminerait en un temps fini, qu'il y ait une solution ou pas.

Nous conjecturons de plus que dans le cas d'un planificateur de niveau quelconque et non basé sur A*, HDS hérite toujours des propriétés de correction, de complétude

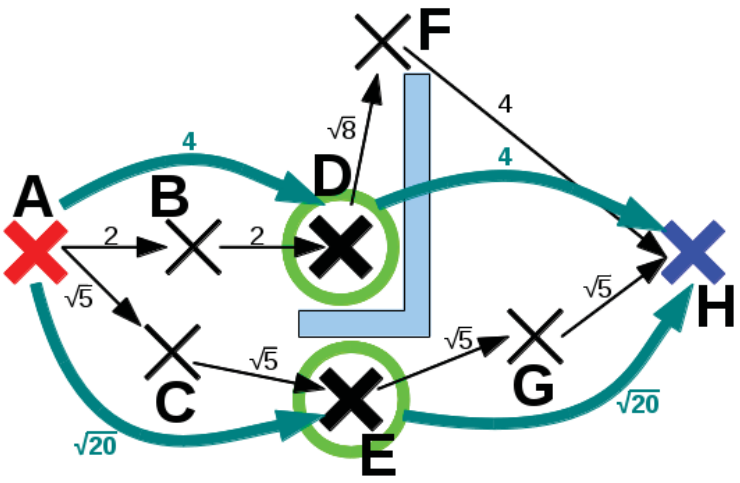


FIGURE 3.8 – Labyrinthe simplifié.

et d'optimalité de celui-ci, mais pas de la propriété de terminaison.

3.4 Déroulement sur un exemple

Nous allons maintenant dérouler cet algorithme sur un exemple simple pour mieux comprendre ce qu'il se passe. L'exemple sera toujours le problème de planification de trajectoire, dont le graphe aura été simplifié pour l'occasion. Une représentation graphique en est donnée sur la figure 3.8, et le code correspondant est donné en annexe A.2. Cette fois-ci, un gros obstacle rallonge le trajet entre `D` et `H`, obstacle qui ne peut être vu au niveau des `jumpTo` et dont les effets ne seront observés qu'au niveau inférieur.

Dans l'état initial, on a alors l'agent en position `A` avec comme tâche de haut niveau `moveTo H`.

Un planificateur de niveau 1 est alors lancé, avec `at` valant `A` (que nous noterons `(at A)` par la suite pour plus de commodité) comme état initial, et `{moveTo H}` comme plan consigne. La décomposition HTN lui impose donc de choisir des positions atteignables avec `exists_path` à partir de `A` et de leur appliquer la tâche `jumpTo`. Les états suivants sont donc `(at D)` et `(at E)`. L'état `(at D)` a un coût de 4 et un coût estimé au but de 4 (h+g = 8), tandis que `(at E)` a un coût de $\sqrt{20}$ (> 4) et un coût estimé au but de $\sqrt{20}$ (h+g = $2 * \sqrt{20}$).

L'état `(at D)` est alors choisi comme nœud à développer. En suivant les macros, le développement de ce nœud correspond à la sortie d'un `while`. La condition du while étant toujours vérifiée (`D` $\neq$ `H`), on rentre à nouveau dans le `while`, et on ouvre le seul état suivant possible, `(at H)` (g = 8, h = 0, h+g = 8). La remontée récursive ne change pas la valeur d'estimation de distance au but (h) de `(at D)`.

A la boucle suivante, `(at D)` est toujours le meilleur fils de `(at A)`, `(at H)` celui de `(at D)`, donc on le développe, ce qui nous fait sortir du `while`, atteindre la fin

du plan précédent, et comme `(at H)` vérifie le but, on obtient le plan solution : `{jumpTo D, jumpTo H}`, d'un coût total de 8.

Étant déjà au premier niveau, aucun backtrack n'est effectué, et on crée un nouveau planificateur, de niveau 2, avec ce plan en entrée. De la même façon que précédemment, le planificateur ouvre tous les choix qui lui sont permis par l'arbre HTN, c'est à dire tous les états obtenus en effectuant un `goTo` depuis `A` vers un état voisin (`elem`). Les états `(at B)` et `(at C)` sont donc ouverts, `(at B)` avec un coût de 2, une valeur heuristique de 2 pour arriver en `D` (fonction heuristique de la tâche `jumpTo`) et un chemin restant de longueur estimé de 4, donc un h+g de 8. `(at C)`, quand à lui, a un coût de $\sqrt{5}$ et une valeur heuristique de $\sqrt{5}+4$, et donc un h+g supérieur à 8. C'est donc l'état `at B` qui sera développé par la suite.

En déroulant ainsi la planification à bas niveau, on finit par trouver le plan `{goTo B, goto D, goTo F, goto H}`, d'une longueur de $8+\sqrt{8}$. La valeur de $8+\sqrt{8}$ est alors reportée en bout de branche `{jumpTo D, jumpTo H}` du planificateur de niveau 1, qui change alors l'estimation au but de l'état `(at D)` à $4+\sqrt{8}$ (au lieu de 4). Lors du développement suivant de `(at A)` dans le planificateur de niveau 1, le nœud le plus prometteur devient alors `E`, et en continuant ce développement, on termine sur le plan `{jumpTo E, jumpTo H}`, d'un coût total de $2*\sqrt{20}$ ($\approx$ 8,9).

Le plan ayant changé, on supprime le planificateur de niveau 2 déjà existant (et tous les suivants s'il en existe), et on recommence une planification avec un nouveau guide. A la première étape, on développe donc toujours les nœuds `(at B)` et `(at C)`, sauf que la valeur de `(at B)` est maintenant $2+\sqrt{8}+\sqrt{20}$ ($\approx$ 9,3) et celle de `(at C)` est de $\sqrt{5}+\sqrt{5}+\sqrt{20}$($\approx$ 8,9). C'est donc le nœud `(at C)` qui se trouve développé. Et en continuant ce développement, on obtient le plan `{goTo C, goto E, goTo G, goto H}`, de coût $4*\sqrt{5}$. Ce coût étant identique à celui du planificateur juste supérieur, on n'effectue pas de backtrack. Étant donné qu'il est de plus composé uniquement d'actions élémentaires, le planificateur global considère la planification terminée et renvoie le plan trouvé.

Conclusion

Nous avons pu voir dans ce chapitre un algorithme exploitant le formalisme proposé précédemment. Cet algorithme permet de planifier à différents niveaux de hiérarchie pour ne pas avoir à traiter avec les petits niveaux de détail tant que les grandes lignes du plan ne sont pas tracées. Il est de plus en mesure de remettre en question un plan de haut niveau dont le développement ne se passerait pas de manière nominale, et de trouver un plan optimal alors qu'il n'était pas dans les premiers choix pour un planificateur abstrait.

Nous avons de plus proposé une variante de cet algorithme permettant de tirer parti des processeurs multi-cœurs présents actuellement dans quasiment tous les calculateurs. Cette variante n'est à l'heure actuelle qu'une preuve de faisabilité, et demandera des travaux supplémentaires afin de pouvoir la formaliser d'avantage et déterminer ses propriétés.

Chapitre 4

Expérimentations et résultats

Sommaire

4.1 Protocole de test 77
4.1.1 Choix d'implémentation 77
4.1.2 Domaines de test 78
4.2 Performances 86
4.2.1 Performance des différents apports 86
4.2.2 Comparaison avec SHOP2 89
4.2.3 Expérimentations sur un domaine réel 93

Après avoir expliqué en détail nos contributions en matière de formalisme et d'algorithme, nous allons voir comment on peut les mettre en place en pratique, et quels sont les résultats que nous avons obtenus.

Nous présenterons dans un premier temps les choix que nous avons effectués pour l'implémentation de notre planificateur, en présentant le langage utilisé, OCaml.

Nous effectuerons ensuite des tests de notre planificateur sur l'exemple de référence pour montrer les performances des différents ajouts ainsi que les limitations de notre planificateur. Dans un deuxième temps, nous effectuerons des tests comparatifs entre SHOP2 et HDS sur plusieurs domaines connus et un domaine maison. Nous montrerons enfin l'aptitude de notre planificateur à traiter un problème réel issu d'un projet actuellement en cours à l'ONERA.

Nous nous limiterons aux comparaisons avec SHOP2, car bien que d'autres planificateurs avec expertise hiérarchique existent, ils ne sont pas prévus pour fonctionner sur les mêmes domaines que le nôtre ou ne peuvent résoudre les problèmes en garantissant de trouver la solution optimale.

4.1 Protocole de test

4.1.1 Choix d'implémentation

Nous avons choisi de développer notre planificateur dans le langage OCaml [1]. Ce langage, développé par l'INRIA, est un langage de type fonctionnel, paradigme très utilisé dans l'intelligence artificielle. En effet, les langages fonctionnels sont particulièrement adaptés à l'écriture d'algorithmes récursifs, tels qu'une grande majorité d'algorithmes du domaine, et tel que le nôtre. De plus, le langage OCaml

1. http://caml.inria.fr/ocaml/index.fr.html

présente l'avantage d'être compilable en code natif, ce qui n'est pas le cas de Common Lisp qui est interprété, ce qui lui confère de très bonnes performances.

Ce langage fut développé à l'origine pour des applications mathématiques telles que la preuve de théorèmes (l'assistant de preuves Coq est codé en OCaml). Il a ensuite rapidement évolué pour devenir particulièrement adapté pour toute application de type calcul symbolique : compilation, analyse de programmes et planification. Bien que peu connu du grand public, il est néanmoins largement utilisé pour l'enseignement de la programmation fonctionnelle. Certains grands groupes industriels tels que Dassault Systèmes, Airbus, Microsoft, IBM et le CEA ont aussi été conquis par sa sûreté de programmation liée au typage fort et à l'inférence de types.

OCaml s'accompagne de puissants outils de lexing et de parsing, ocamllex et ocamlyacc (basés sur Lex et Yacc) que nous avons utilisés pour analyser les fichiers de domaine et problème.

Enfin, OCaml utilise un ramasse miettes pour la gestion de la mémoire, que nous avons paramétré afin d'optimiser l'exécution de notre code. Ce ramasse miettes classe les utilisations mémoire en deux groupes, un pour les utilisations à court terme, ou les données amenées à avoir une faible durée de vie devant celle du programme, et un autre pour les utilisations à long terme où les données ont une durée de vie nettement plus longue. Il peut se déclencher selon trois modes, l'un pour purger la mémoire rapide, un second pour purger les deux mémoires, et un troisième qui permet de forcer OCaml à réorganiser entièrement la mémoire pour optimiser son fonctionnement. Pour des raisons évidentes de performances, seuls les deux premiers modes sont déclenchés automatiquement par le programme. Le troisième, dont le temps d'exécution peut prendre plusieurs secondes, ne peut être déclenché que manuellement. Malgré son temps d'exécution particulièrement élevé, un déclenchement manuel toutes les 20 secondes permet de gagner presque trente pourcents sur le temps total d'exécution de notre planificateur.

La gestion des threads en OCaml est assez particulière. En effet, les threads à mémoire partagés ne peuvent s'exécuter que de façon entrelacée, et donc sur un seul cœur, tandis que les threads exécutables en parallèles sont forcément en mémoire non partagée. Ce fonctionnement est rendu nécessaire afin de ne pas dégrader les performances du ramasse miettes. Pour l'implémentation de notre planificateur multi-niveaux parallélisé, nous avons donc dû utiliser des threads à mémoire séparée, et donc mettre en place un dialogue par chaînes de caractères via des pipes.

4.1.2 Domaines de test

4.1.2.1 Parcours hiérarchique de graphe

Ce domaine est le domaine qui a été présenté en fil rouge de cette thèse. Il s'agit de planifier une trajectoire dans un graphe dont certaines particularités sont connues, comme un certain nombre de points importants, et s'il existe ou pas un chemin non absurde les reliant. On pourrait relier ce cas au problème du GPS, où la carte est connue et pré-analysée une bonne fois pour toute, avec les grandes villes comme

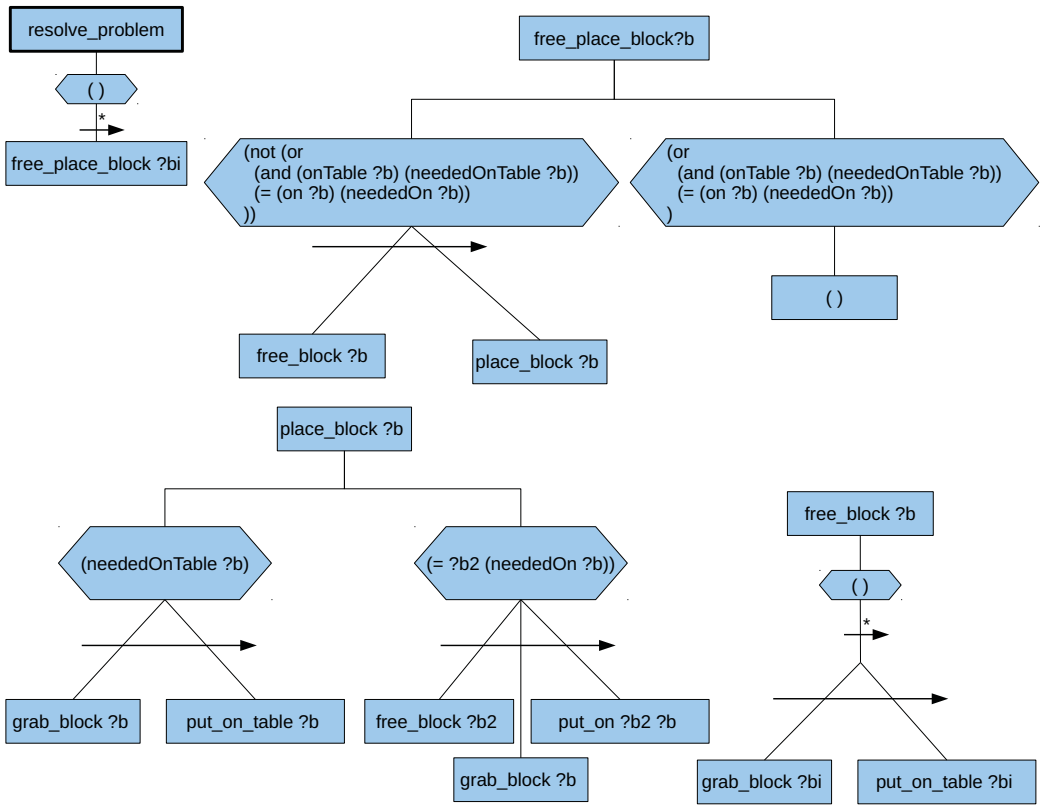


FIGURE 4.1 – Représentation graphique du HTN utilisé pour le domaine Blocks World.

points importants, et où une instance du problème consiste à trouver un chemin avec un point de départ et un point d'arrivée donnés.

Le chapitre 2 s'appuie sur ce domaine et explique sa décomposition. Le code du domaine et un exemple de problème sont donnés en annexe A.2.

4.1.2.2 Blocks World

Il s'agit d'un domaine bien connu de la communauté de planification classique. Le principe est qu'une pince robotisée doit, à partir d'un arrangement donné de piles de blocs, faire la pile finale demandée. La pince peut prendre un cube en sommet d'une pile, prendre un cube sur la table, poser un cube sur la table, ou poser un cube sur un autre qui est lui-même en sommet de pile. Chacune de ces actions a un coût de un, et le but est de minimiser le coût total. Le nombre d'espaces libres sur la table est illimité.

Dans le cadre de la compétition de planification IPC, la solution demandée ne contient qu'une seule pile. Il est donc facile d'en déduire la solution optimale, qui consiste à prendre les blocs de la tour finale du bas vers le haut, et s'ils ne sont pas déjà à leur place, poser sur la table tout bloc les recouvrant avant de les mettre à

leur place [Gupta 1992]. Cette stratégie s'écrit très facilement en HTN, et donne de très bons résultats. La représentation graphique de cette décomposition est donnée en figure 4.1. Le domaine tel qu'écrit pour HDS est présenté en annexe A.3.

4.1.2.3 Zeno Traveler

Zeno Traveler est un domaine provenant lui aussi de la compétition IPC consistant à déplacer plusieurs personnes de leur point de départ à leur point d'arrivée, à l'aide de plusieurs avions. Chaque avion peut embarquer autant de passager que souhaité.

Dans la version choisie, chaque déplacement a un coût temporel de un, de même pour chaque embarquement et débarquement. Chaque déplacement coûte aussi en carburant de façon proportionnelle à la distance à parcourir. L'objectif est de minimiser une combinaison linéaire donnée de ces deux paramètres de coût.

La décomposition HTN que nous proposons se limite à forcer les avions à n'aller qu'à des endroits où il y a un intérêt (une personne présente et pas déjà à sa destination ou une destination d'une personne déjà à bord) et à forcer les passagers à descendre s'ils sont dans un avion à leur destination ou à ne pas monter dans un avion s'ils sont déjà à destination. Le domaine tel qu'écrit pour HDS est présenté en annexe A.4.

4.1.2.4 Freecell

Ce domaine est tiré du très célèbre solitaire informatique du même nom. Le but du jeu consiste à faire des piles de cartes de même couleur allant de l'as au roi, sachant que le plateau de départ a un nombre de cases fixé, que sur le plateau, on ne peut poser une carte donnée que sur une carte de valeur immédiatement supérieure et de couleur inversée. Le plateau comporte de plus quatre cases ne pouvant contenir qu'une seule carte à la fois (et non des piles), utilisées pour faire du stockage temporaire. Contrairement à bon nombre de solitaires, il est interdit de déplacer des piles de cartes, les mouvements ne se font qu'une carte à la fois.

La décomposition HTN que nous proposons pour HDS est très simple. En effet, nous avons écrit le problème comme une boucle consistant :

1. à monter dans les piles finales toute carte dont on est certain qu'elle ne peut se révéler utile à la suite du jeu (un as, par exemple, ou toute carte dont toutes les cartes de valeur directement inférieure et de couleur opposée ont déjà été montées) ;
2. à effectuer une opération consistant soit à déplacer une carte (que ce soit pour la mettre sur une autre case, sur une case libre, ou la monter sur une des piles finales), ce qui est une action élémentaire, soit à déplacer une pile, si on a pu voir que le nombre de cases vides est suffisant pour effectuer ce déplacement.

Cette boucle s'effectue jusqu'à ce que toutes les cartes soient montées.

Nous avons de plus rajouté une heuristique qui est le nombre de cartes restant à monter. Cette heuristique est bien évidemment admissible.

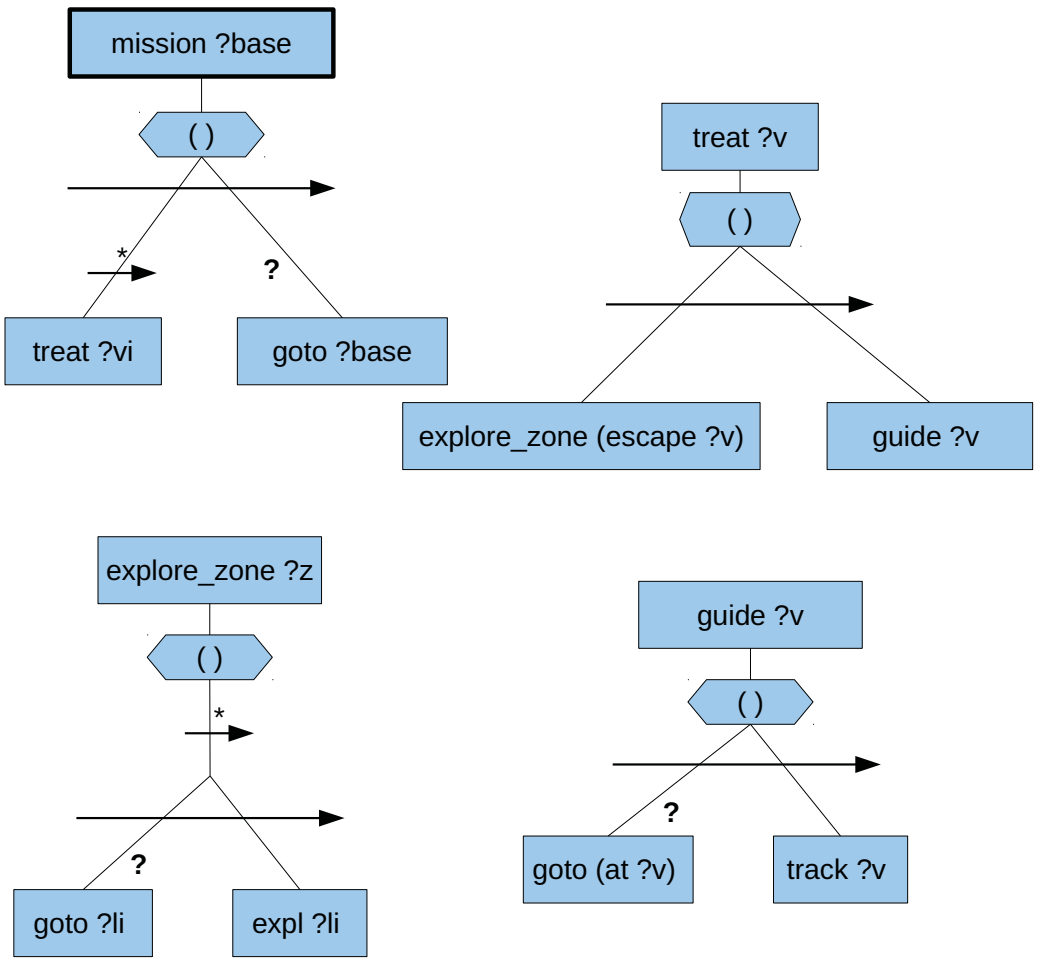


FIGURE 4.2 – Représentation graphique du HTN utilisé pour le domaine Explore and Guide.

4.1.2.5 Explore and Guide

Ce domaine est un domaine maison dérivé du PEA Action [PEA Action 2007] où un hélicoptère doit raccompagner plusieurs agents à l'extérieur de la zone contrôlée. Il connaît à l'avance la position de chacun des agents, coopératifs lors de la reconduction et immobiles le reste du temps, et il connaît aussi les itinéraires qu'ils veulent emprunter pour sortir. Il doit par contre parcourir au préalable ces itinéraires pour déceler les éventuels pièges qui seraient placés (mines par exemple). Le but de la mission est de raccompagner la totalité des agents en un temps minimum.

L'espace est discrétisé, et le temps nécessaire pour se rendre d'une case à l'autre est proportionnel à la distance euclidienne. La durée nécessaire pour explorer et sécuriser une case de la grille est constante, et la vitesse des agents est constante et connue.

Pour celui-ci, la modélisation HTN avec méta-effets a servi à découpler le choix

de l'ordre dans lequel l'hélicoptère raccompagnait les agents du choix de l'ordre dans lequel les cases de sortie étaient explorées. La représentation graphique de cette décomposition est donnée en figure 4.2. Le domaine tel qu'écrit pour HDS est présenté en annexe A.6.

4.1.2.6 Déminage de chenal

Ce domaine est directement inspiré du scénario II du PEA Action. Il s'agit d'une problématique de déminage de chenal pour préparer la sortie d'un sous-marin nucléaire (SNLE ou SNA) par exemple. Sur zone, deux véhicules autonomes sont présents, un sous-marin ayant pour but de trouver les éventuelles mines et d'en communiquer la position avec le sol pour que des plongeurs démineurs soient envoyés, et un hélicoptère dont le but est de trouver et faire sortir de la zone les bateaux présents afin d'éviter qu'ils ne posent de nouvelles mines dans la zone une fois celle-ci nettoyée. Les bateaux sont supposés coopératifs et sortent de la zone dès la première injonction sans qu'on ait besoin de les raccompagner. Un schéma explicatif est présenté en figure 4.3.

Dans la pratique, le sous-marin est soumis à une dérive de son estimation de position, et doit refaire surface et communiquer avec l'hélicoptère afin que celui-ci le repositionne, et que le géoréférencement des mines trouvées soit précis. Le sous-marin ne peut pas non plus communiquer directement avec le sol, et doit se servir de l'hélicoptère pour faire relais de communication. Pour toute communication avec l'hélicoptère, le sous-marin doit être à la surface. La distance de communication entre les deux engins est limitée, et l'hélicoptère doit se situer à la verticale du sous-marin pour pouvoir le repositionner. Un repositionnement (ou recalage) devra avoir leur au plus toutes les dix minutes.

Afin de s'assurer que le sous-marin n'est pas entrain de déminer une zone où des bateaux sont encore présents, le problème est découpé en plusieurs sous-zones. L'hélicoptère s'occupera de vider la zone n tandis que le sous-marin déminera la zone $n-1$.

Ce problème comporte de nombreuses incertitudes, notamment sur le nombre de mines ainsi que leurs positions, et sur la localisation des véhicules pour cause de dérive. Il est donc plus adapté à une résolution probabiliste. Nous en avons néanmoins fait une formulation déterministe. Chaque zone est discrétisée selon une grille, et la consigne est d'explorer chaque point de la grille ; nous considérons que l'action d'exploration d'une case de la grille permet au passage de détecter une mine s'il y en a une ou de demander le départ d'un bateau s'il y en a un, le tout sans changement de la durée de l'action. L'action de recalage du sous-marin via l'hélicoptère est une action symbolique et obligatoire au maximum tous les n pas de temps, et que les éventuelles communications de mines trouvées se font à ce moment là, sans surcoût temporel.

Les principales variables d'état de ce problème sont donc les positions des véhicules (exprimées en terme de cellule dans laquelle ils sont), ainsi que leur horloge interne

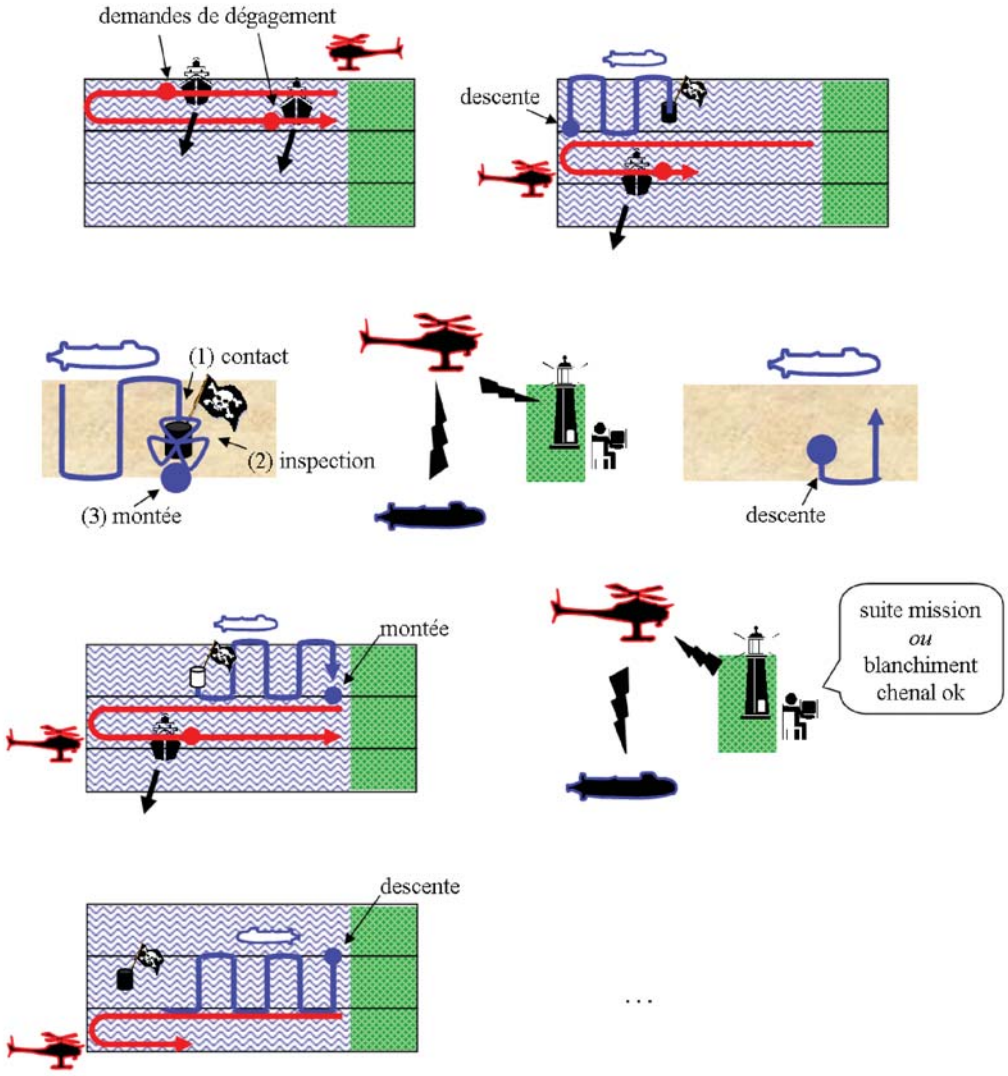


FIGURE 4.3 – Présentation du scénario II du PEA Action. Un sous-marin et un hélicoptère autonomes doivent déminer un chenal pour préparer la sortie d'un SNLE. Le sous-marin cherche les éventuelles mines et communique leur position à un opérateur terrestre, tandis que l'hélicoptère, avec une zone d'avance sur le sous-marin, demande aux bateaux sur site de sortir.

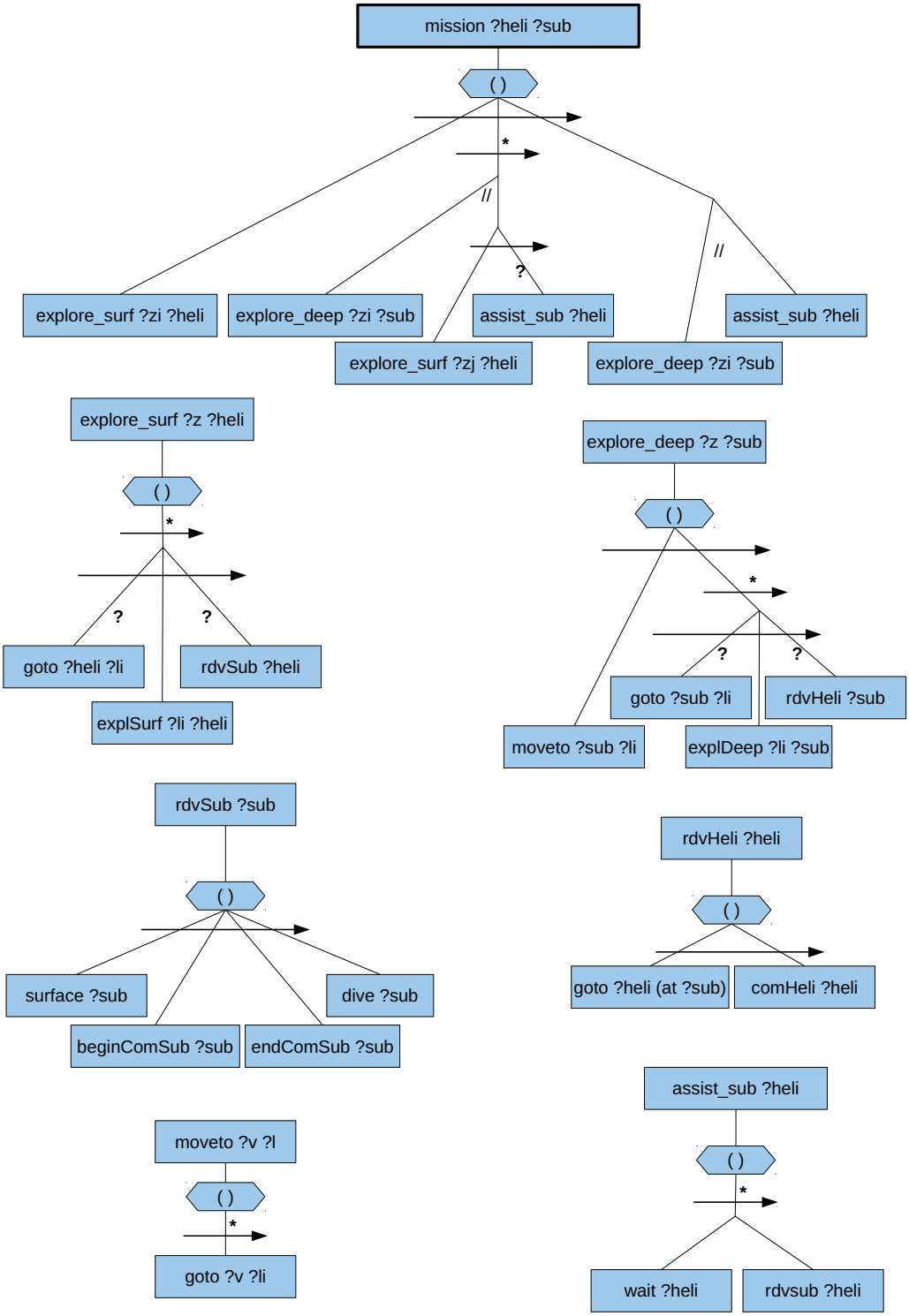


FIGURE 4.4 – Représentation graphique du HTN utilisé pour le domaine de déminage.

et l'état à la surface ou en profondeur du sous-marin. Chaque cellule de la grille comprend deux marqueurs, un pour indiquer que la surface de cette cellule a été nettoyée, et un autre pour le exprimer la même notion pour le fond.

Notre modélisation ajoute de plus, pour pouvoir fonctionner, un état "En communication" pour le sous-marin, ce qui nous permet de synchroniser les véhicules. Nous ajoutons aussi des variables pour connaître la date de la dernière communication et un compteur de cases restantes à explorer dans la zone. Ces variables ne sont pas inhérentes au problème mais dépendent de nos choix de modélisation et des restrictions causées par les HTN.

Les actions élémentaires du problèmes sont les suivantes :

- pour le sous-marin :
  - se déplacer d'une case ;
  - faire surface/plonger ;
  - explorer une case en profondeur pour vérifier l'absence de mine ;
  - engager une communication radio
- pour l'hélicoptère :
  - se déplacer vers une case quelconque ;
  - explorer une case pour en vérifier la présence ou non de bateaux et le cas échéant leur demander de partir ;
  - répondre à une communication engagée par le sous-marin pour transmettre les coordonnées d'une mine si nécessaire et recaler ses senseurs.

Dans notre modèle HTN, nous avons fait évoluer en parallèle la mission du sous-marin et celle de l'hélicoptère.

- La mission du sous-marin consiste en l'exécution en séquence des sous-missions de zone avec un temps d'attente au début, et pour chaque zone, en un nombre limité de couples déplacement/exploration, suivi d'une communication. Une communication se décompose en une action pour faire surface, une action de communication qui nécessite l'hélicoptère au même endroit, et une action de plongée.
- La mission de l'hélicoptère consiste aussi en une boucle sur les zones, puis en une boucle durant laquelle l'hélicoptère se déplace et explore des cases à la recherche de bateaux, un nombre limité de fois avant de se déplacer à la position courante du sous-marin et communiquer avec lui.

Les HTN permettent dans cette mission de grandement réduire le facteur de branchement. En effet, un agent ne tentera jamais d'effectuer plusieurs déplacements d'affilé, sans faire une exploration entre chacun. La séquence d'actions nécessaire pour communiquer est entièrement donnée, et le planificateur n'a pas à la retrouver. Et surtout, ils permettent d'exprimer facilement les contraintes de zones séparées et de temps maximum entre les communications. Le domaine écrit pour HDS est disponible en annexe A.7, et une illustration de ce domaine est présentée en figure 4.4.

4.2 Performances

Nous présenterons dans ce chapitre les résultats obtenus avec nos algorithmes. Ces résultats seront sous forme de performances temporelles sur des problèmes de planification tirés des compétitions de planification IPC, ainsi que des problèmes construits à des fins utiles pour nos travaux. Dans une première partie, nous présenterons les résultats obtenus uniquement par notre planificateur, mais avec divers niveaux de fonctionnalités, puis nous comparerons notre planificateur dans sa forme complète avec le meilleur planificateur du domaine à l'heure actuelle, SHOP2.

Chacun des tests présentés par la suite a été réalisé sur une machine linux 64 bits, avec un processeur quadri-cœurs cadencés à 3GHz. Chaque planificateur peut utiliser au plus 1Go de RAM, et est déclaré hors temps après une durée de 10 minutes.

4.2.1 Performance des différents apports

Afin de bien cerner quelles sont les gains dus aux différents apports que nous avons pu faire au cours de cette thèse, nous comparerons ici les performances de notre planificateur dans diverses situations, chacune ajoutant une fonctionnalité à notre planificateur. Avec une formulation de problème n'exploitant pas les macro-tâches tout d'abord, puis en les rajoutant, puis en ajoutant des heuristiques, et enfin en utilisant l'algorithme P-HDS. Nous ne montrerons pas ici les apports des meta-effets car l'algorithme ne peut fonctionner sans, sous peine de ne pas pouvoir enchaîner les tâches au sein d'un niveau.

Nous avons effectué ce test sur le domaine de la planification de trajectoire dans un graphe présenté jusqu'ici. La taille de l'instance représente le nombre de nœuds dans le graphe, et les graphes sont générés aléatoirement sur une grille hexagonale, avec une distance entre les points voisins de 1 et une distance à vol d'oiseau entre les points de saut de 5. Lorsqu'une heuristique est utilisée, il s'agira de l'heuristique classique du parcours de graphe, c'est à dire la distance à vol d'oiseau du point courant vers le point cible.

4.2.1.1 Les Macro-Tâches

Nous connaissons déjà les apports des macro-tâches en terme de facilité de modélisation d'une part et en ce qui concerne la cohérence des plans hiérarchiques d'autre part. Nous allons ici tenter de mettre en valeur les apports en matière de performance de cet ajout. Pour ce faire, nous avons modélisé le domaine du parcours hiérarchique de graphe sans utiliser d'opérateur de boucle, mais une représentation récursive comme lorsque modélisé pour SHOP2.

Il est évident que la résolution par notre planificateur HDS sera nettement plus longue en utilisant ce modèle qu'avec la version du domaine utilisant des macro-actions. En effet, chaque développement récursif du plan, selon notre algorithme, génère un nouveau planificateur, et chaque backtrack dans le plan en cours de génération doit se faire non pas au sein d'un planificateur, mais de façon hiérarchique, en remontant l'information à un autre planificateur. Ces processus (création d'un

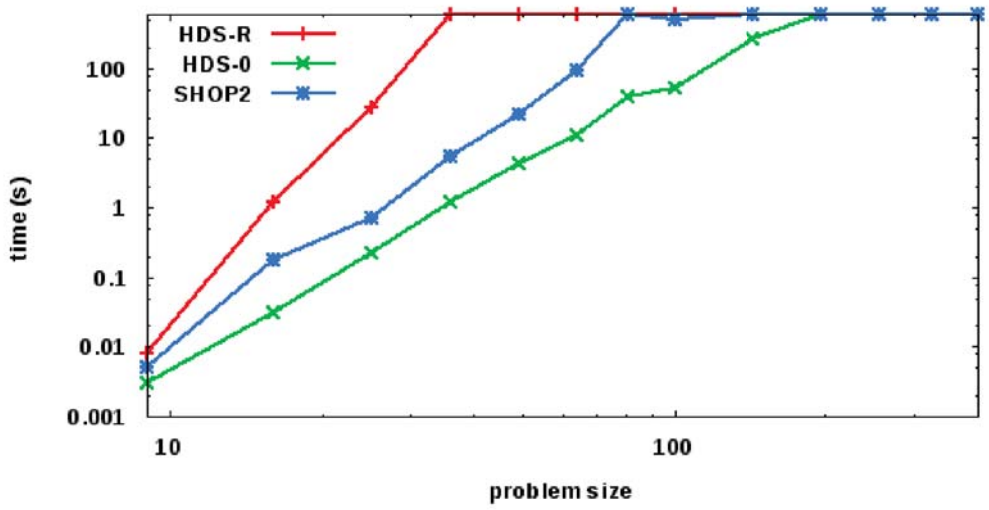


FIGURE 4.5 – Comparatif de performances sur le domaine de parcours de graphe entre HDS sans les macro-tâches (HDS-R), HDS avec macro-tâches (HDS-0), et SHOP2. Aucun des planificateurs présents dans ce test n'utilise d'heuristique.

planificateur et backtrack hiérarchique) étant plusieurs ordres de grandeur plus lents que les processus d'avance et backtrack dans un planificateur classique, nous pouvons nous attendre à des résultats très mauvais.

Les résultats sont présentés sur la figure 4.5. Ils y montrent la comparaison des performances de HDS avec un domaine écrit de façon récursive (HDS-R), HDS avec le domaine écrit comme il le faut mais sans utilisation d'une quelconque heuristique (HDS-0), et SHOP2, avec une version récursive du problème.

Ce test ne permet pas vraiment de mettre en valeur l'apport des macro-tâches, mais montre surtout que HDS est totalement inadapté à des représentations récursives des domaines, et que lors de l'utilisation de ce planificateur, il faut s'assurer que le domaine est correctement écrit en fonction des capacités de traitement de HDS, qui sont différentes de celles de SHOP2, conçu pour les domaines hiérarchiques.

4.2.1.2 Gain de performances avec les heuristiques

Les heuristiques sont un facteur bien connu de gain. Guidant le planificateur dans une direction privilégiée qui pointe vers le but plutôt que de le laisser explorer de façon anisotropique le problème, on peut généralement obtenir un gain de l'ordre d'un ordre de grandeur sur la rapidité d'obtention de la solution.

Pour évaluer l'apport des heuristiques dans HDS, nous avons comparé notre algorithme avec trois entrées différentes. La première, sans heuristique, soit exactement la deuxième version du test précédent. La seconde intègre uniquement l'heuristique d'objectif final, comme A*, c'est à dire la distance au but à vol d'oiseau. La troisième intègre les heuristiques présentées en section 2.2.5 du chapitre sur les formalismes.

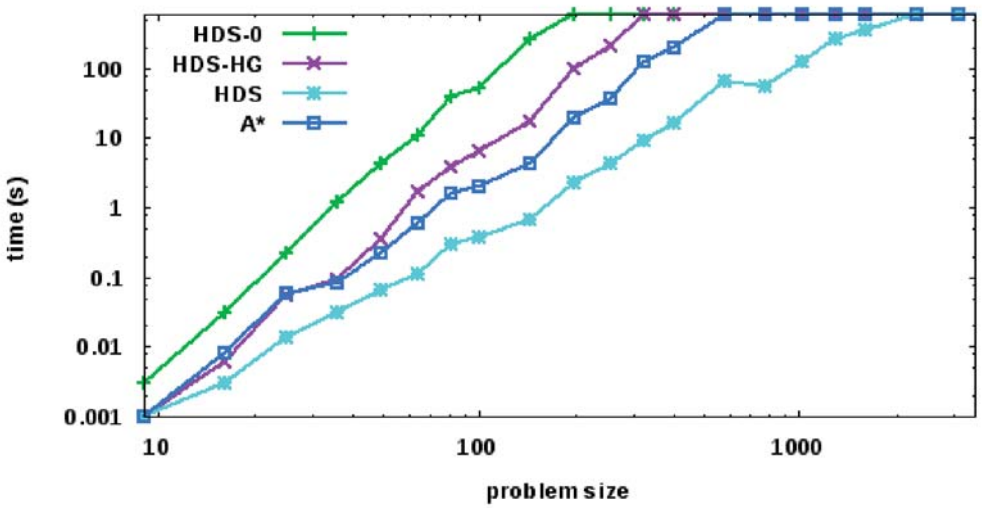


FIGURE 4.6 – Comparatif de performances sur le domaine de parcours de graphe entre HDS sans heuristique (HDS-0), HDS avec heuristiques globales (HDS-HG), HDS dans sa version complète, c'est à dire avec heuristiques de tâches (HDS), et A*.

Les performances de A* ont été rajoutées pour référence.

La figure 4.6 présente ces résultats. Notre planificateur est étiqueté HDS-0 lorsqu'il est utilisé sans heuristique ; HDS-HG (pour Heuristique Globale) quand l'heuristique ne fait que pointer vers le but global ; et HDS pour le planificateur dans sa forme complète, utilisant une heuristique le guidant vers la fin de tâche sommée au coût trouvé au niveau supérieur pour atteindre le but à partir de la fin de la tâche. Les résultats de A* sur le même problème sont donnés à titre de comparaison.

On y voit clairement l'intérêt des heuristiques, le planificateur utilisant les heuristiques globales étant nettement plus performant que lorsqu'il n'en utilise pas.
On note aussi que HDS-HG est significativement plus lent que A*. Cet écart se justifie par le fait que dans ce problème, la décomposition HTN ne restreint pas la liste des actions possibles à plus bas niveau par rapport à la planification classique. Le choix des actions se fait en effet dans les deux cas parmi les `goTo` vers un point voisin. A* a de plus comme avantage par rapport à HDS qu'il élimine les revisites, ce que nous ne pouvons faire à l'heure actuelle pour les raisons expliquées en section 3.1.2.4 du chapitre Algorithmes.

L'ajout des heuristiques de tâche vient compenser ce défaut, en guidant localement la planification de trajectoire vers le point de passage suivant. Nous notons alors un fort gain de performances par rapport à A* (notamment sur les gros problèmes) et par rapport à HDS-HG.

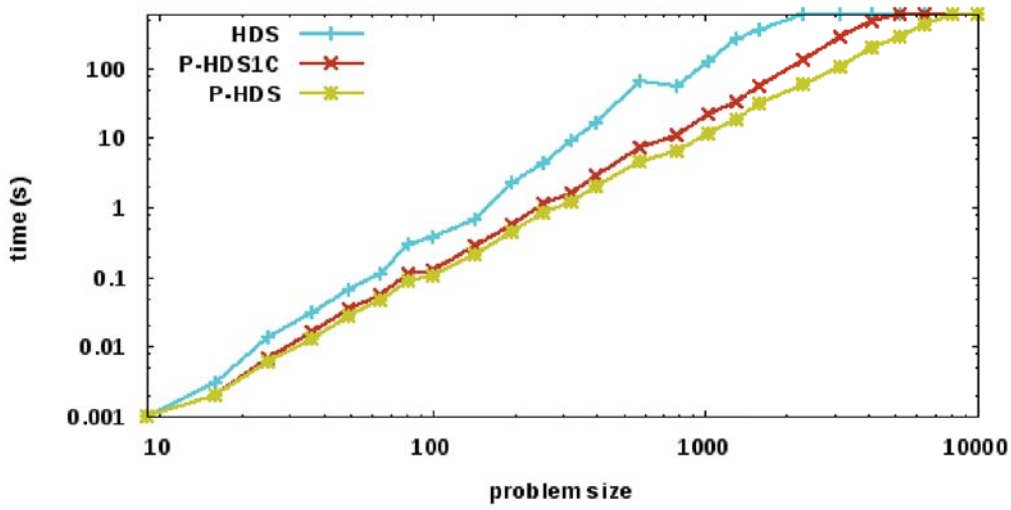


FIGURE 4.7 – Comparatif de performances sur le domaine de parcours de graphe entre HDS avec heuristiques de tâche (HDS), et P-HDS qui détecte au plus tôt les surcoûts, qu'il soit sur un seul thread (P-HDS1C) ou parallélisé (P-HDS).

4.2.1.3 La détection précoce aide, le parallélisme apporte un petit plus

Le plan de test suivant cherche à quantifier le gain obtenu lors de l'utilisation de notre algorithme utilisant le parallélisme entre les différents planificateurs ainsi que la détection précoce de surcoûts.

La figure 4.7 présente les résultats de P-HDS, puis de P-HDS bridé sur un seul cœur afin de voir les gains du parallélisme (P-HDS-1C), et de la version sans détection précoce et sans parallélisme (HDS). L'ensemble de ces résolutions se fait avec l'heuristique de tâche.

On remarque que les résultats sont clairement en faveur de la détection précoce, comme on pouvait s'y attendre. Nous nous rendons de plus bien compte de la nécessité de paralléliser cet algorithme. En effet, à chaque fois que le planificateur de bas niveau finit une tâche de haut niveau (et pour chaque chemin emprunté permettant de finir cette tâche), le planificateur de haut niveau est relancé. Et ceci avec interruption complète du planificateur de bas niveau si le parallélisme n'est pas activé. Le temps de calcul est alors parfois presque doublé par rapport au temps de calcul de la version parallèle.

4.2.2 Comparaison avec SHOP2

Nous avons ensuite comparé notre planificateur avec le planificateur de référence dans le domaine des HTN, SHOP2 [Nau 2003].

Pour les domaines issus de la compétition (Blocks World, Zeno Traveler et Freecell), nous avons fourni en entrée à SHOP2 les domaines créés par les concepteurs du planificateur lors de la compétition. Pour Explore and Guide, nous avons donné à

SHOP2 la même décomposition que celle que nous avons fournie à HDS, méta-effets et macro-tâches en moins (les boucles étaient modélisées sous forme récursive, et les `optional` avec plusieurs choix de méthodes). De façon à ce que la comparaison ait un sens, nous avons de plus configuré SHOP2 pour qu'il cherche la solution optimale aux problèmes. Le problème de Déminage de chenal étant trop complexe à modéliser dans le formalisme HTN, nous ne présenterons que les résultats de HDS sur celui-ci.

Pour la comparaison, nous lui avons opposé trois versions de notre planificateur : HDS-0 sans heuristiques, afin d'avoir une comparaison non biaisée, car SHOP2 n'utilise pas les heuristiques non plus, HDS, avec les heuristiques de tâche, et P-HDS, avec la détection précoce et le parallélisme.

4.2.2.1 Blocks World

Les résultats du test comparatif sur le domaine de Blocks World sont présentés sur la figure 4.8. Comme expliqué précédemment, le domaine fourni à HDS pour résoudre les problèmes de Blocks World donne une méthode optimale, sans choix à effectuer et donc sans backtrack. Le calcul d'heuristique ne permet donc pas d'accélérer la résolution, de même pour la détection précoce de surcoûts. Les trois courbes sont donc quasiment confondues (HDS et P-HDS sont très légèrement plus lents à cause du calcul d'heuristiques à faire tout de même).

On remarque que la résolution par notre planificateur est entre un et deux ordres de grandeur plus rapide que celle de SHOP2.

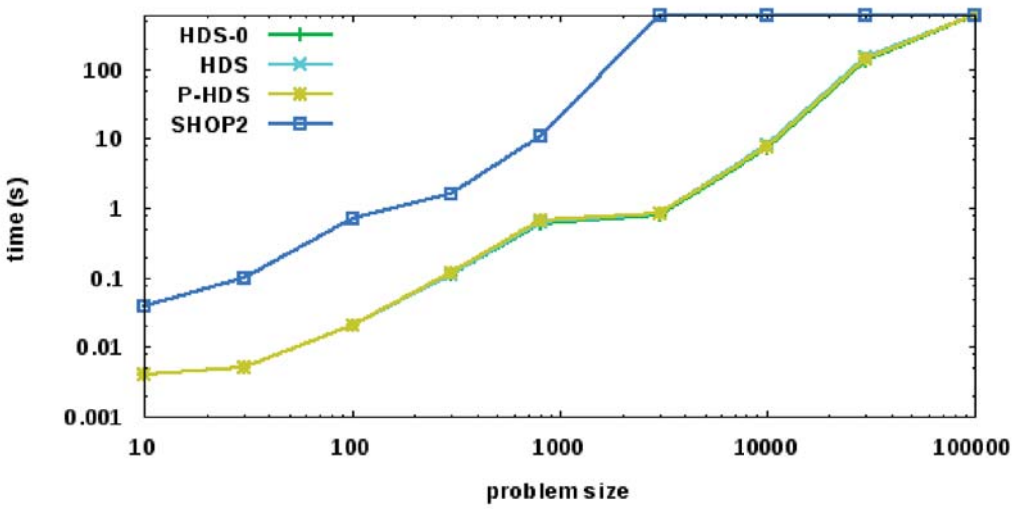


FIGURE 4.8 – Comparatif de performances sur le domaine Blocks World entre HDS-0, sans heuristique, HDS, P-HDS et SHOP2.

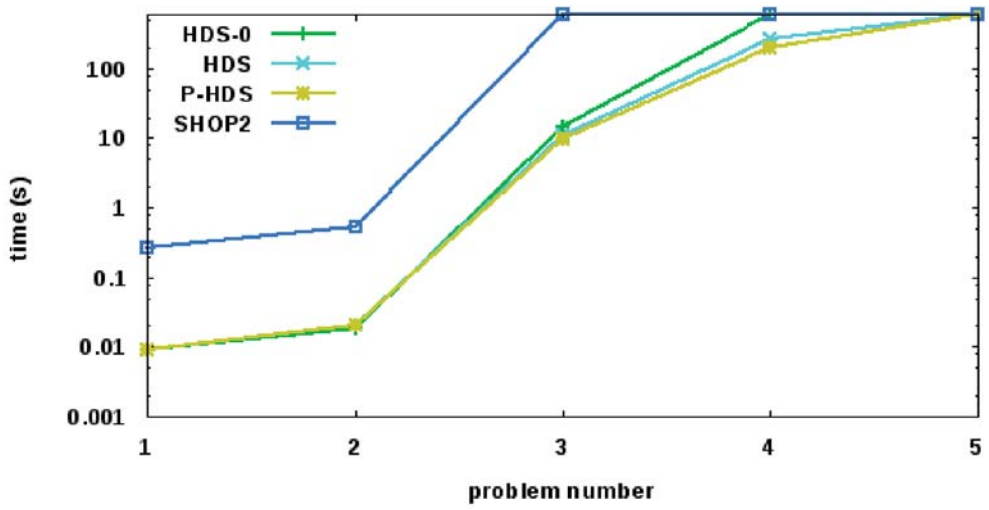


FIGURE 4.9 – Comparatif de performances sur le domaine Zeno Traveler entre HDS-0, sans heuristique, HDS, P-HDS et SHOP2.

4.2.2.2 Zeno Traveler

Les résultats du test comparatif sur le domaine de Zeno Traveler sont présentés en figure 4.9. Les résultats sont très mauvais, à la fois pour SHOP2 et pour HDS. En effet, les petits problèmes (jusqu'au numéro quatre) associés à ce domaine sont suffisamment simples pour qu'un humain puisse en donner la solution optimale du premier coup d'œil, alors qu'ici les planificateurs s'y cassent les dents. Ceci est dû au fait qu'aucun des deux algorithmes n'est conçu pour travailler sur du parallélisme, présent ici entre les différents avions.

HDS parvient tout de même à faire un peu mieux que SHOP2, résolvant le problème numéro 3 sans nécessiter d'heuristique, et le numéro 4 avec une heuristique de mauvaise qualité. L'heuristique aurait pu être bien meilleure en écrivant à la main une heuristique inspirée Hmax [Haslum 2000], telle que "le coût nécessaire pour acheminer la personne qui coûte la plus cher à acheminer dans le cas le plus favorable" (un avion est déjà présent sur place). Cependant le langage que nous avons développé ne comprenant pas la fonction `max`, il nous a été impossible d'écrire cette heuristique. Nous nous sommes contentés d'un coût lié au plus petit des déplacements possibles (donnée extraite lors de la reformalisation du problème pour notre formalisme et donnée en entrée du planificateur comme constante du problème), multiplié par le nombre minimum de voyages nécessaires, c'est à dire le nombre de personnes ayant des trajets distincts divisé par le nombre d'avions.

4.2.2.3 Freecell

Les résultats du test comparatif sur le domaine de Freecell sont présentés sur la figure 4.10. On remarque encore une fois l'avantage de HDS sur SHOP2. Cependant,

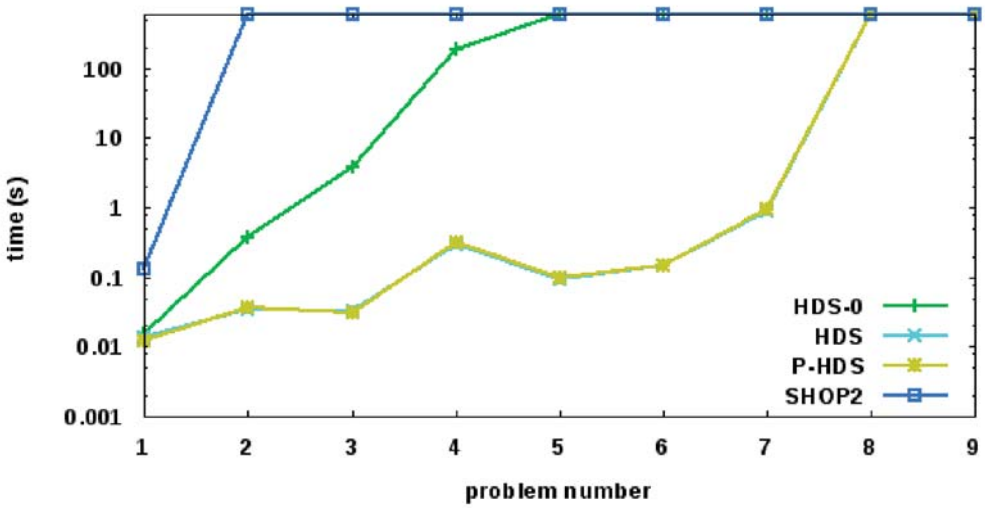


FIGURE 4.10 – Comparatif de performances sur le domaine Freecell entre HDS-0, sans heuristique, HDS, P-HDS et SHOP2.

les deux planificateurs font ici aussi de piètres scores (les problèmes à résoudre sont encore une fois considérés comme triviaux pour des humains), SHOP2 n'étant en mesure de ne résoudre que le premier problème avec 8 cartes, tandis que HDS parvient péniblement à résoudre le problème numéro 4 qui en comprend 20. La raison semble ici être la non détection des revisites.

En ajoutant comme heuristique le nombre de cartes restant à monter, on parvient à de bien meilleurs résultats, puisque le problème 7, dernier à être résolu avec HDS, comporte 32 cartes. L'apport de la détection précoce est ici nul, car le domaine n'utilise la hiérarchie que pour déplacer des piles de cartes, dont on connaît de façon exacte le coût nécessaire à la réalisation. Le reste des techniques est formalisé grâce aux macro-tâches sur un seul niveau hiérarchique.

4.2.2.4 Explore and Guide

Les résultats du test comparatif sur le domaine de Explore and Guide sont présentés sur la figure 4.11. Ces tests sont clairement en faveur de HDS, où la version sans heuristique est parfois deux ordres de grandeur plus rapide que SHOP2. Ceci est dû au fait que ce problème est clairement découpé en niveaux hiérarchiques avec assez peu de backtracks (comme on peut le voir avec le faible écart avec la version avec détection précoce). En effet, sauf dans de rares cas, les estimations de coûts pour le raffinement sont assez fiables, bien que non exactes. Les deux problèmes qui sont d'une part le choix de l'ordre de traitement des cibles et d'autre part l'ordre d'exploration du chemin de sortie sont alors traités de façon complètement découplée, ce qui favorise amplement notre découpage par niveaux hiérarchiques.

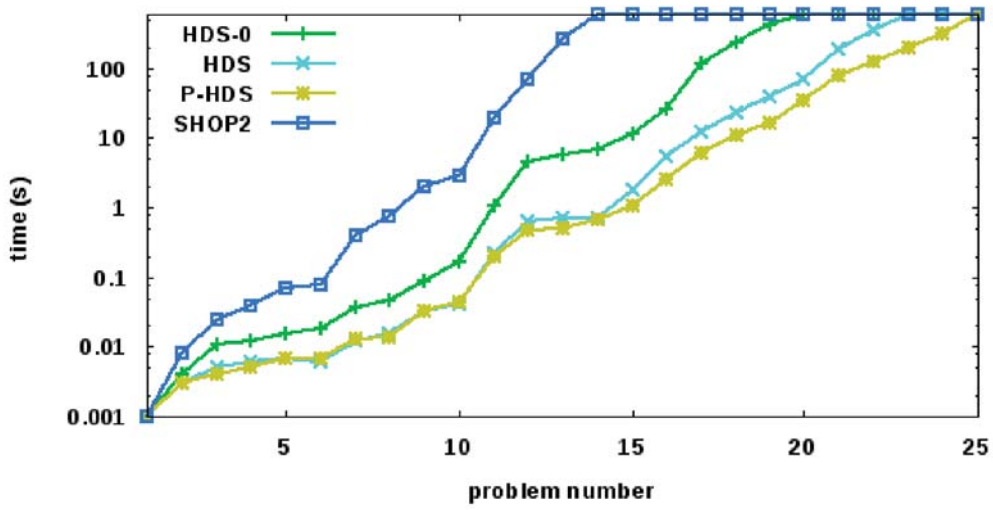


FIGURE 4.11 – Comparatif de performances sur le domaine Explore and Guide entre HDS-0, sans heuristique, HDS, P-HDS et SHOP2.

4.2.3 Expérimentations sur un domaine réel

Après avoir montré précédemment que notre formalisme s'adapte assez facilement à la modélisation de problèmes réels comme le problème de déminage de chenal, nous avons cherché à voir comment se comporte le planificateur sur ce domaine. Nous avons pour cela repris les caractéristiques du problème tel que décrit dans le PEA Action, avec trois zones distinctes, et avons fait varier le niveau de détail dans la description de chaque zone en changeant le nombre de cases par zone (chaque zone étant rectangulaire).

Les résultats sont présentés sur la figure 4.12. On voit qu'en utilisant les heuristiques, notre planificateur s'en sort en un temps raisonnable pour un problème allant jusqu'à soixante cases par zones. La détection précoce des surcoûts est ici inutile, car seul un niveau nécessite de la planification : celui déterminant dans quel ordre les cases seront explorées et quand faire les communications. Le niveau précédant ainsi que le suivant sont entièrement déterministes dans leurs choix. Il n'y a donc aucun backtrack hiérarchique.

Conclusion

Nous avons pu voir, au travers des résultat présentés dans ce chapitre, les différents apports de nos algorithmes et les gains de performance associés : les macro-tâches, indispensables au bon fonctionnement de notre planificateur pour limiter les backtracks hiérarchiques, les heuristiques de tâches pour valoriser le plan hiérarchique obtenu aux niveaux précédents, et la détection précoce de surcoûts avec des planificateurs tournant en parallèle, qui aide dans les problèmes où l'estimation des méta-effets

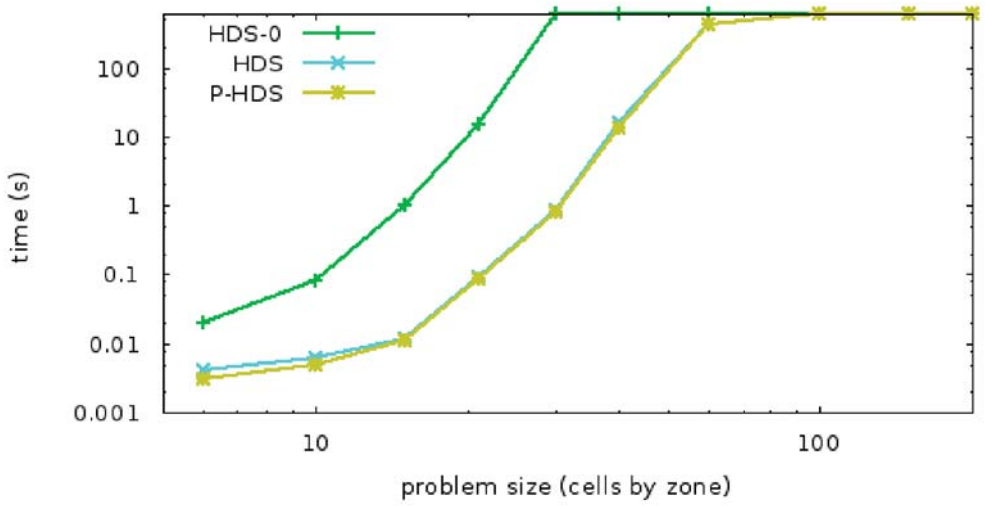


FIGURE 4.12 – Comparatif de performances sur le domaine de déminage de chenal entre HDS-0, sans heuristique, HDS et P-HDS.

n'est pas suffisamment précise.

Nous avons de plus comparé notre algorithme à A* sur la planification de trajectoire, et à SHOP2, planificateur de référence dans le domaine de la planification HTN. Ces tests ont montré que HDS, dans sa version comparable à SHOP2, c'est-à-dire sans utilisation d'heuristiques, était nettement plus performant, ce qui valide notre approche. Nous avons néanmoins montré les limites de la planification optimale sans détection des états dupliqués (que ce soit pour HDS comme pour SHOP2) sur les domaines Zeno Traveler et Freecell.

Enfin, nous avons montré que notre langage de modélisation permettait l'apport d'expertise sur des problèmes réels, et que notre planificateur était en mesure de les résoudre en un temps raisonnable.

Conclusion et perspectives

Sommaire

1 Synthèse des contributions 95
2 Travaux complémentaires 97
2.1 Voies de recherche sur le planificateur de niveau 97
2.2 Travaux sur l'interaction entre les niveaux 98
2.3 Extensions de langage 98

1 Synthèse des contributions

Dans cette thèse, nous avons mis au point un nouveau paradigme de planification basé sur l'expertise humaine. Ce paradigme s'inspire très fortement de la planification hiérarchique avec expertise telle qu'on peut la rencontrer avec les HTN et SHOP2. Il reprend les idées de *tâches*, qui sont des actions abstraites, non réalisables directement par l'agent à qui le plan est destiné. Il reprend aussi le concept des *méthodes*, qui sont des procédures permettant de décomposer les tâches en sous-tâches, voire actions élémentaires, permettant au planificateur de construire un plan réalisable par l'exécutant à partir de tâches abstraites. Ces travaux ont été acceptés par la communauté de planification au travers des papiers [Schmidt 2011a, Schmidt 2011b, Schmidt 2011c].

Par rapport à ce formalisme initial, nous avons ajouté et formalisé les méta-effets, qui permettent au planificateur de faire, dans un premier temps, abstraction des tâches de précision trop élevée dans son processus de planification, et donc de planifier un plan donnant les grandes lignes de la réalisation de la mission. Ce plan sera ensuite raffiné de façon itérative, jusqu'à atteindre un plan de bas niveau, réalisable par l'exécutant.

Nous avons de plus renforcé le formalisme d'écriture des méthodes avec des systèmes de macro-tâches, très fortement inspirées de la programmation structurée utilisée par les langages de programmation classiques. Ce renforcement permet, grâce à des opérateurs de boucles, d'obtenir des plans cohérents en ce qui concerne le niveau d'abstraction de son contenu. Nous pensons de plus qu'un tel formalisme est nettement plus accessible à un utilisateur qui ne serait pas spécialiste en informatique ni habitué aux contraintes et pièges de la programmation récursive.

Une telle ouverture dans le champ d'utilisation nous fait penser que ce formalisme de planification pourra être utilisé dans divers champs, notamment dans le cadre de la robotique militaire. En effet, notre formalisme permet de contraindre le planificateur à trouver des plans respectant certaines contraintes opérationnelles telles que les

protocoles de mission. Mais nous pensons de plus que les opérationnels militaires pourront, grâce à ce formalisme, intégrer des contraintes d'ordre éthique dans les robots, ce qui facilitera leur intégration sur le terrain.

Sur la base de ce formalisme, nous avons développé un algorithme de planification. Cet algorithme exploite le côté hiérarchique du modèle, et s'appuie sur les extensions des méta-effets et macro-tâches pour créer des plans à différents niveaux hiérarchiques. Chaque plan présente une unité en ce qui concerne le niveau d'abstraction des tâches présentes. Ils découlent de plus chacun du plan de niveau directement supérieur qui sert alors de guide, et servent à leur tour de guide pour le plan immédiatement inférieur.

Conscients que les plans de niveau abstraits, construits à partir d'effets optimistes, peuvent soit ne pas être réalisables, soit ne pas se dérouler comme prévu et avoir des surcoûts qui seront détectés en les raffinant, nous avons mis en place un système de retour d'information. Celui-ci permet de faire ce que nous appelons un backtrack hiérarchique, c'est à dire une reprise de la planification à haut niveau lorsqu'on détecte que le déroulement d'un plan peut être trop coûteux par rapport aux prévisions.

L'algorithme ainsi trouvé permet de construire un plan optimal pour un problème donné, à partir d'une décomposition HTN, du moment que cette dernière le permet. L'algorithme a aussi été prouvé complet et correct.

Nous avons de plus proposé une amélioration de cet algorithme, augmentant la communication entre les planificateurs pour détecter au plus tôt les nœuds d'un plan abstrait dont la décomposition serait plus coûteuse ou dont les effets seraient moins bons qu'attendus, et qui rendraient le plan solution moins prometteur qu'une autre solution de haut niveau. Cette amélioration permet donc de réduire le nombre de replanifications nécessaires à l'obtention du plan optimal.

L'algorithme est de plus adapté aux processeurs multi-cœurs de plus en plus présents dans les ordinateurs actuels, en attribuant un process séparé à chaque planificateur de niveau.

Nous avons enfin montré au travers de tests comparatifs entre différentes versions de notre planificateur les gains en matière de performance de nos différents apports, et l'intérêt fondamental des macro-tâches dans notre algorithme.

Nous avons de plus mis en valeur les performances de notre planificateur comparé au planificateur de référence du domaine, SHOP2. Cette mise en valeur s'est effectuée sur des domaines connus et couramment utilisés, tirés de la compétition de planification IPC, et des domaines maison qui tirent pleinement parti des avantages de notre planificateur.

Notre proposition répond donc bien aux demandes initiales. Un gain significatif en performances et en taille de problèmes résolubles est constaté par rapport à SHOP2, et la forme des plans hiérarchiques en sortie est plus cohérente dans sa structure, facilitant son utilisation avec un superviseur hiérarchique.

Cependant, une question reste ouverte pour savoir si l'utilisation de HDS est systématiquement préférable à celle de SHOP2, celle des méta-effets. En effet, ils

requièrent un travail supplémentaire de la part de l'expert, qui doit maintenant formaliser des effets pour chacune des tâches, ces effets étant de plus contraints par une exigence d'optimisme et d'admissibilité. Il est donc légitime de se demander si cette nouvelle exigence ne vient pas compenser les apports en performance au niveau de l'utilisabilité de nos travaux.

2 Travaux complémentaires

Bien que cette thèse touche à sa fin, la voie que nous avons ouverte durant celle-ci n'en est qu'à ses débuts, et pour bien la mettre en valeur, de nombreux travaux complémentaires sont à envisager.

2.1 Voies de recherche sur le planificateur de niveau

Dans nos travaux, nous avons utilisé la base de A* (ou Dijkstra lorsque les valeurs heuristiques étaient assignées à 0) pour mettre au point notre planificateur de niveau. Bien que nos idées soient très générales et devraient, à priori, pouvoir s'appliquer à tout autre planificateur en avant, il serait intéressant de confronter ceci à la pratique. En effet, il n'est pas garanti que les planificateurs élaborant leurs propres heuristiques ne risquent pas d'avoir des problèmes pour les construire étant donné les fortes restrictions imposées par l'arbre HTN, par exemple.
Utiliser un planificateur sous-optimal comme planificateur par niveau pourrait être une voie d'amélioration intéressante. En effet, dans la pratique, l'écriture d'un HTN masque certaines solutions, et son utilisation est plus souvent motivée par des contraintes d'ordre opérationnel ou éthique. Nous pensons donc que les utilisateurs de planificateur HTN peuvent, le plus souvent, être plus intéressés par un planificateur capable de résoudre des problèmes de très grande taille quitte à ce que la solution soit approchée, plutôt que par un planificateur garantissant l'optimalité de sa solution, mais capable de ne travailler que sur des instances plus réduites.

Nous avons de plus fait le choix de ne pas détecter les états dupliqués. Ce choix est réellement nécessaire lorsque ces états sont à des positions différentes dans l'arbre HTN et mènent vers des suites de plans différentes. Mais nous avons de plus choisi de ne pas détecter la duplication des états même si la position dans l'arbre HTN est la même. La motivation de ce choix est que si deux plans peuvent sembler donner les mêmes effets à haut niveau, il n'en est pas de même à plus bas niveau, dû à l'inexactitude des meta-effets. Nous pensons qu'il doit être possible d'effectuer un compromis, permettant de détecter les états dupliqués, et d'interrompre la planification pour les branches menant à ces duplications, mais seulement de façon momentanée. Le développement de ces branches ne reprendrait que si la planification à bas niveau détecte que la branche développée a de mauvaises estimations sur ses effets, et que donc les autres branches peuvent mener à un état voisin et moins coûteux. Cette amélioration permettrait de gagner beaucoup sur la consommation en mémoire et les temps d'exécution.

Enfin, nous pensons qu'il pourrait être intéressant de voir si les travaux sur les HTN probabilistes [Kuter 2005] pourraient être compatibles avec nos travaux de résolution par niveaux distincts. En effet, nous avons noté une forte similitude entre les algorithmes utilisés pour la résolution globale des MDP probabilistes (utilisation d'algorithmes en avant avec suivi de la position dans le HTN) et l'algorithme que nous avons employé pour la planification au sein d'un niveau.

2.2 Travaux sur l'interaction entre les niveaux

Nous avons, lors de cette thèse, proposé une version de l'algorithme avec une interactivité accrue entre les niveaux. Malheureusement, le temps a manqué pour pouvoir étudier dans les détails ses propriétés. Des travaux ultérieurs sont donc à réaliser afin de pouvoir en connaître les caractéristiques de façon plus précise.

Un lecteur averti aura aisément fait le lien entre notre processus de backtracks et les processus de replanification utilisés dans la robotique. Un travail de bibliographie sur cette replanification est certainement à prévoir afin d'améliorer l'algorithme que nous utilisons actuellement.

Dans le cadre de la planification probabiliste par HTN, nous pensons que cet algorithme peut-être utilisé pour effectuer de la planification partielle en préparation de mission. Comme nous l'avons vu, notre processus de backtracks est proche de celui de replanification en robotique. Nous pensons donc qu'il devrait être possible de planifier grâce à des algorithmes de planification probabiliste anytime tels que RFF, en planifiant entièrement les premiers niveaux de la hiérarchie, puis en ne planifiant plus que le début des niveaux inférieurs de façon précise, et en attendant que la situation se précise avec l'avancement de la mission pour planifier les détails de la suite de celle-ci. Une telle méthode de planification permettrait d'avoir une bonne estimation du caractère réalisable de la mission dans sa globalité, tout en ne faisant pas exploser la complexité calculatoire en évitant de planifier les moindres détails de situations très peu probables.

2.3 Extensions de langage

D'un autre côté, nous pensons que les HTN présentent une forte limitation, qui est que tout plan qui n'est pas permis par le HTN ne sera pas atteignable par le planificateur. Dans un grand nombre de cas, cette limitation est voulue, afin de respecter des contraintes opérationnelles ou des protocoles par exemple, mais nous estimons que dans certains cas, l'utilisateur est en droit de rechercher un comportement différent. Nous pensons que certaines tâches, comme des tâches de recharge de batteries par exemple, ne doivent pas être placées de façon précise dans le HTN, sous peine soit de sur-contraindre le problème, soit d'en complexifier l'écriture. Nous pensons donc qu'il faudrait réfléchir à une extension du formalisme, qui permettrait d'effectuer des insertions opportunistes de tâches. Ces tâches seraient alors placées dans un cadre à part, et non dans l'arbre HTN de la mission, et le planificateur pourrait les insérer où bon lui semble (pour peu que ses préconditions

soient vérifiées).

Conclusion

Nous avons donc pu voir dans cette thèse un cadre de modélisation et un algorithme qui réconcilient planification et systèmes experts. Ce cadre rend accessible la résolution de problèmes aujourd'hui trop complexes pour la planification classique car présentant une combinatoire trop élevée, tout en n'exigeant qu'un travail expert modeste.

Bien qu'ayant déjà prouvé leur utilité, ces travaux restent encore à améliorer, en poursuivant les pistes de recherche ouvertes lors de cette thèse, mais aussi en explorant d'autres voies qui se sont présentées à nous lors des développements, tels que la détection précoce d'une nécessité de replanification, ou la possibilité pour le planificateur de s'écarter du cadre strict imposé par la décomposition HTN pour saisir une opportunité d'amélioration ou de solution non prévue par l'expert.

Ces travaux pourront de plus se révéler utiles lorsque les technologies d'extraction automatique de structure hiérarchique seront plus performantes, pour accueillir et traiter les données générées.

Bibliographie

[AGATA 2005] AGATA, 2005. http://agata.cnes.fr. (Cité en page 4.)

[Aguirre 2009] Hernán Aguirre et Kiyoshi Tanaka. *Space partitioning with adaptive ε-ranking and substitute distance assignments: a comparative study on many-objective mnk-landscapes*. In GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation, pages 547–554, New York, NY, USA, 2009. ACM. (Cité en page 23.)

[Bacchus 2000] F. Bacchus et F. Kabanza. *Using Temporal Logics to Express search control Knowledge for Planning*. Artificial Intelligence, 2000. (Cité en page 22.)

[Becker 2003] Raphen Becker, Shlomo Zilberstein, Victor Lesser et Claudia V. Goldman. *Transition-independent decentralized markov decision processes*. In AAMAS'03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems, pages 41–48, New York, NY, USA, 2003. ACM. (Cité en page 21.)

[Bellman 1957] Richard Ernest Bellman. Dynamic programming. Dover Publications, Incorporated, 1957. (Cité en page 20.)

[Bernstein 2002] Daniel S. Bernstein, Robert Givan, Neil Immerman et Shlomo Zilberstein. *The Complexity of Decentralized Control of Markov Decision Process*. In Mathematics of Operations Research, 27(4):819–840, 2002. (Cité en page 21.)

[Blum 1997] A. L. Blum et M. L. Furst. *Fast planning through planning graph analysis*. Artificial Intelligence, vol. 90, no. 1-2, pages 279 – 298, 1997. (Cité en page 18.)

[Botea 2004] Adi Botea, Martin Müller et Jonathan Schaeffer. *Near optimal hierarchical path-finding*. Journal of Game Development, vol. vol.1 - No.1, pages 7–28, 2004. (Cité en pages 6 et 23.)

[Botea 2005] A. Botea, M. Enzenberger, M. Müller et J. Schaeffer. *Macro-FF: Improving AI Planning with Automatically Learned Macro-Operators*. Journal of Artificial Intelligence Research, vol. 24, pages 581–621, 2005. (Cité en page 24.)

[Canu 2011] Arnaud Canu et Abdel-Illah Mouaddib. *A new approach for solving large instances of DEC-POMDPs: Vector-Valued DEC-POMDPs.* In MSDM , Toronto, Canada, 2011. (Cité en page 22.)

[Cassandra 1994] Anthony R. Cassandra, Leslie Pack Kaelbling et Michael L. Littman. *Acting Optimally in Partially Observable Stochastic Domains*, 1994. (Cité en page 21.)

[Coles 2007] A. Coles, M. Fox et A. J. Smith. *Online identification of useful macro-actions for planning*. In Proc. ICAPS, 2007. (Cité en page 24.)

[Crevier 1993] Daniel Crevier. Ai: The tumultuous search for artificial intelligence. New York, NY: BasicBooks, 1993. (Cité en page 1.)

[de Silva 2009] Lavindra de Silva, Sebastian Sardina et Lin Padgham. *First Principles Planning in BDI Systems*. In Autonomous Agents and Multiagent Systems (AAMAS-09), 2009. (Cité en page 22.)

[Dijkstra 1959] Edsger W. Dijkstra. *The shortest spanning subtree of a graph*. In A short introduction to the art of programming, pages 67–73, 1959. (Cité en page 14.)

[Eldracher 1993] M. Eldracher et B. Baginski. *Hierarchical Planning Using Neural Subgoal Generation*. In Systems, Man and Cybernetics, 1993. (Cité en page 24.)

[Eldracher 1997] Martin Eldracher et Thomas Pic. *Path planning using a subgoal graph*. In Foundations of Computer Science, volume 1337/1997, 1997. (Cité en page 24.)

[Erol 1994a] Kutluhan Erol, James Hendler et Dana S. Nau. *HTN planning: complexity and expressivity*. In AAAI'94: Proceedings of the twelfth national conference on Artificial intelligence (vol. 2), pages 1123–1128, 1994. (Cité en page 22.)

[Erol 1994b] Kutluhan Erol, James Hendler et Dana S. Nau. *Semantics for hierarchical task-network planning*. Rapport technique, University of Maryland at College Park, College Park, MD, USA, 1994. (Cité en pages 27 et 33.)

[Fernandez-Madrigal 2002] Juan-Antonio Fernandez-Madrigal et Javier Gonzalez. *Multihierarchical Graph Search*. IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. vol.24, pages 103–113, 2002. (Cité en page 23.)

[Fikes 1971] R. Fikes et N. J. Nilsson. *STRIPS : A new approach to the application of theorem proving toproblem solving*. Artificial Intelligence, vol. 2, no. 3-4, pages 61 – 124, 1971. (Cité en page 12.)

[Flinsenberg 2004] Ingrid C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. PhD thesis, Université technique d'Eindhoven, Pays Bas, 2004. (Cité en page 6.)

[Fox 2003] Maria Fox et Derek Long. *PDDL2.1: An extension to PDDL for expressing temporal planning domains*. Journal of Artificial Intelligence Research, vol. 20, pages 61–124, 2003. (Cité en pages 13, 19 et 40.)

[Gantt 1910] H.L. Gantt. Work, wages and profit. The Engineering Magazine, New York, 1910. (Cité en page 25.)

[Geffner 2000] Héctor Geffner. *Functional Strips: a more flexible language for planning and problem solving*. Logic-based Artificial Intelligence, pages 188–209, 2000. (Cité en page 40.)

[Gerevini 2003] Alfonso Gerevini, Ivan Serina, Alessandro Saetti et Sergio Spinoni. *Local Search Techniques for Temporal Planning in LPG*. In 13th International

Conference on Automated Planning and Scheduling (ICAPS-03), AAAI Press, Trento, Italy, 2003. (Cité en page 19.)

[Ghallab 2004] M. Ghallab, D. Nau et P. Traverso. Automated planning. Morgan Kaufmann, San Francisco, CA, USA, 2004. (Cité en page 13.)

[Gupta 1992] Naresh Gupta et Dana Nau. *On the Complexity of Blocks-World Planning.* Artificial Intelligence, vol. 56, no. 2-3, pages 223 – 254, 1992. (Cité en page 80.)

[Hansen 2001] Eric A. Hansen et Shlomo Zilberstein. *LAO*: A heuristic search algorithm that finds solutions with loops.* Artificial Intelligence, vol. 129, no. 1-2, pages 35 – 62, 2001. (Cité en page 20.)

[Hart 1968] Peter E. Hart, Nils J. Nilson et Bertram Raphael. *A formal basis for the heuristic determination of minimum cost path.* Transactions of systems science and cybernetics, vol. 2, pages 100–107, 1968. (Cité en page 16.)

[Haslum 2000] Patrik Haslum et Héctor Geffner. *Admissible Heuristics for Optimal Planning.* In AAAI Press, pages 140–149, 2000. (Cité en pages 18 et 91.)

[Hoare 1969] C.A.R Hoare. *An axiomatic basis for computer programming.* Communication of the ACM, vol. 12, no. 10, pages 576 – 585, 1969. (Cité en page 41.)

[Hogg 2008] Chad Hogg, Héctor Muñoz-Avila et Ugur Kuter. *HTN-MAKER: Learning HTNs with Minimal Additional Knowledge Engineering Required.* In Association for the Advancement of Artificial Intelligence (AAAI-08), 2008. (Cité en page 24.)

[Howard 1960] Ronald A. Howard. Dynamic programming and markov processes. The M.I.T. Press, 1960. (Cité en page 20.)

[Kaelbling 1998] L. P. Kaelbling, M. L. Littman et A. R. Cassandra. *Planning and acting in partially observable stochastic domains.* Artificial Intelligence Journal, vol. 101, pages 99–134, 1998. (Cité en page 21.)

[Karp 1972] Richard M. Karp. *Reducibility Among Combinatorial Problems.* In Complexity of Computer Computations, pages 85–103, 1972. (Cité en page 1.)

[Kuter 2005] Ugur Kuter et Dana S. Nau. *Using Domain-Configurable Search Control for Probabilistic Planning.* In AAAI, Pittsburgh, Pennsylvania, USA, July 2005. (Cité en pages 34, 56 et 98.)

[Kvanström 2001] Jonas Kvanström et Patrick Doherty. *TALplanner: A Temporal Logic Based Forward Chaining Planner.* Annals of Mathematics and Artificial Intelligence, 2001. (Cité en page 22.)

[Lee 2004] Ivan S.K. Lee et Henry Y.K. Lau. *Adaptive state space partitioning for reinforcement learning.* Engineering Applications of Artificial Intelligence, vol. vol.17 - No.6, pages 577–588, 2004. (Cité en page 23.)

[Marthi 2008] Bhaskara Marthi, Stuart Russel et Jason Wolfe. *Angelic Hierarchical Planning: Optimal and Online Algorithms.* In International Conference on

Automated Planning and Scheduling (ICAPS-08), 2008. (Cité en pages 8, 22 et 26.)

[Martin 2011] Cyrille Martin, Humbert Fiorino et Gaëlle Calvary. *Flexibilité dans les plans pour une planification centrée humain*. In JFPDA, 2011. (Cité en page 24.)

[McDermott 1998] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld et D. Wilkins. *PDDL – The Planning Domain Definition Language*. Rapport technique, Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, New Haven, CI, USA, 1998. (Cité en pages 2 et 12.)

[Nau 2003] Dana Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu et Fusun Yaman. *SHOP2: An HTN planning system*. Journal of Artificial Intelligence Research, vol. 20, pages 379–404, 2003. (Cité en pages 33 et 89.)

[Newton 2007] M. Newton, J. Levine, M. Fox et D. Long. *Learning macro-actions for arbitrary planners and domains*. In Proc. ICAPS, 2007. (Cité en page 24.)

[PEA Action 2007] PEA Action, 2007. `http://action.onera.fr`. (Cité en pages 4 et 81.)

[Pineau 2003] Joelle Pineau, Geoff Gordon et Sebastian Thurn. *Point-based value iteration : An anytime algorithm for POMDPs*. In IJCAI, Acapulco, Mexico, August 2003. (Cité en page 21.)

[Puterman 1994] Martin L. Puterman. Markov decision processes: Discrete stochastic dynamic programming. John Wiley & Sons, Inc., New York, NY, USA, 1994. (Cité en page 19.)

[Richter 2008] Silvia Richter, Malte Helmert et Matthias Westphal. *Landmarks Revisited*. In 23rd AAAI Conference on Artificial Intelligence (AAAI-08), 2008. (Cité en pages 18 et 24.)

[Schmidt 2009] Pascal Schmidt, Florent Teichteil-Königsbuch, Patrick Fabiani et Guillamue Infantes. *Planification probabiliste hiérarchique : méta-effets et coopération multi-engins*. In JFPDA, Paris, France, 2009. (Cité en page 41.)

[Schmidt 2010] Pascal Schmidt, Florent Teichteil-Königsbuch, Patrick Fabiani et Guillamue Infantes. *Sauts dans l'espace d'états et raffinement de plan en planification classique*. In JFPDA, Besançon, France, 2010. (Cité en page 54.)

[Schmidt 2011a] Pascal Schmidt, Florent Teichteil et Patrick Fabiani. *Tirer parti de l'expertise humaine en planification hiérarchique optimale*. In JFPDA, Rouen, France, 2011. (Cité en pages 28 et 95.)

[Schmidt 2011b] Pascal Schmidt, Florent Teichteil-Königsbuch et Patrick Fabiani. *Taking Advantage of Domain Knowledge in Optimal Hierarchical Deepening Search Planning*. In KEPS, Freiburg, Allemagne, 2011. (Cité en page 95.)

[Schmidt 2011c] Pascal Schmidt, Florent Teichteil-Königsbuch et Patrick Fabiani. *Taking Advantage of Domain Knowledge in Optimal Hierarchical Deepening Search Planning*. In RCRA, Barcelone, Espagne, 2011. (Cité en page 95.)

[Shapiro 1991] Stewart Shapiro. Foundations without foundationalism: A case for second-order logic. Oxford University Press, 1991. (Cité en page 40.)

[Stolyar 1970] Abram Aronovic Stolyar. Introduction to elementary mathematical logic. Dover Publications, 1970. (Cité en pages 12 et 39.)

[Surynek 2009] Pavel Surynek. *On Pebble Motion on Graphs and Abstract Multi-robot Path Planning*. In Proceedings of the ICAPS 2009 Workshop on Generalized Planning, pages 2–9, 2009. (Cité en page 24.)

[Tate 1976] Austin Tate. *Project Planning Using a Hierarchic Non-linear Planner*. Rapport technique, D.A.I. Research Report No. 25, Department of Artificial Intelligence, University of Edinburgh, 1976. (Cité en pages 8 et 25.)

[Tate 1995] Austin Tate. *O-Plan, Task Formalism manual*. Rapport technique, Department of Artificial Intelligence, University of Edinburgh, 1995. (Cité en page 25.)

[Teichteil-Königsbuch 2010] Florent Teichteil-Königsbuch, Guillaume Infantes et Ugur Kuter. *Incremental plan aggregation for generating policies in MDPs*. In Proc. AAMAS, page 1231–1238, 2010. (Cité en page 21.)

[Vidal 2011] Vincent Vidal. *YAHSP2: Keep It Simple, Stupid*. In 7th International Planning Competition (IPC'11), Freiburg, Germany, 2011. (Cité en page 19.)

[Younes 2004] Håkan L. S. Younes et Michael L. Littman. *PPDDL1.0: An extension to PDDL for expressing planning domains with probabilistic effects*. In Tech rep. CMU-CS-04-167. Carnegie Mellon University, Pittsburgh, PA, 2004. (Cité en page 13.)

Annexe A

Fichiers de domaines

A.1 Blocks World - PDDL

A.1.1 Fichier de domaine

```
(define (domain blocksWorld)
  (:types
    crane
    block
  )
  (:predicates
    (on ?b1 ?b2 - block)
    (on_table ?b - block)
    (clear ?b - block)
    (holds ?b - block ?c - crane)
    (empty ?c - crane)
  )
  (:action grab_from_table
    :parameters
      (?b - block ?c - crane)
    :precondition
      (and
        (empty ?c)
        (clear ?b)
        (on_table ?b)
      )
    :effect
      (and
        (not (empty ?c))
        (not (clear ?b))
        (not (onTable ?b))
        (holds ?b ?c)
      )
  )
  (:action grab_from_stack
    :parameters
      (?b1 ?b2 - block ?c - crane)
    :precondition
      (and
        (empty ?c)
        (clear ?b1)
        (on ?b2 ?b1)
      )
    :effect
      (and
        (not (empty ?c))
        (not (clear ?b1))
        (not (on ?b2 ?b1))
        (holds ?b1 ?c)
        (clear ?b2)
      )
  )
  (:action put_on_table
    :parameters
      (?b - block ?c - crane)
    :precondition
      (and
        (not (empty ?c))
        (holds ?b ?c)
      )
    :effect
      (and
        (empty ?c)
        (clear ?b)
        (on_table ?b)
        (not (holds ?b ?c))
      )
  )
```

```
  (:action put_on_stack
    :parameters
      (?b1 ?b2 - block ?c - crane)
    :precondition
      (and
        (not (empty ?c))
        (holds ?b1 ?c)
        (clear ?b2)
      )
    :effect
      (and
        (empty ?c)
        (clear ?b1)
        (on ?b2 ?b1)
        (not (clear ?b2))
        (not (holds ?b1 ?c))
      )
  )
)
```

A.1.2 Fichier de problème

```
(define (problem blocks_world_01)
  (:domain blocksWorld)
  (:objects
    A B C D - block
    CRANE - crane
  )
  (:init
    (on_table B)
    (on B D)
    (clear D)
    (on_table C)
    (on C A)
    (clear A)
    (empty CRANE)
  )
  (:goal
    (and
      (on_table A)
      (on A B)
      (on B C)
      (on C D)
      (clear D)
    )
  )
)
```

A.2 Labyrinth - Formalisme HDS

A.2.1 Fichier de domaine

```
   (define (domain labyrinth)

   (:requirements
     :strips
     :typing
     :hierarchical
     :macros
     :metaeffect
   )

   (:types
     loc
   )
(:functions
  ((at) - loc)

  (elem ?l1 ?l2 - loc)
  (exists_path ?l1 ?l2 - loc)

  ((x ?l - loc) - continuous)
  ((y ?l - loc) - continuous)

  ((cost) - continuous)
)
```

```
(:metaAction moveTo
  :parameters
    (?l - loc)
  :precondition
    ()
  :method m0
    :precondition
      ()
    :subtasks
      (:while (not (= (at) ?l))
        (:pickone (?l1 - loc) with
            (exists_path (at) ?l1)
          (jumpTo ?l1)
        )
      )
  :metaEffect
    (and
      (assign (at) ?l)
      (increase (cost)
        (sqrt (+
          (pow (- (x (at)) (x ?l)) 2)
          (pow (- (y (at)) (y ?l)) 2)
        ))
      )
    )
  :heuristic
    (sqrt (+
      (pow (- (x (at)) (x ?l)) 2)
      (pow (- (y (at)) (y ?l)) 2)
    ))
)
```

```
(:metaAction jumpTo
  :parameters
    (?l - loc)
  :precondition
    (exists_path (at) ?l)
  :method m0
    :precondition
      ()
    :subtasks
      (:while (not (= (at) ?l))
        (:pickone (?l1 - loc) with
            (elem (at) ?l1)
          (goTo ?l1)
        )
      )
  :metaEffect
    (and
      (assign (at) ?l)
      (increase (cost)
        (sqrt
          (+
            (pow (- (x (at)) (x ?l)) 2)
            (pow (- (y (at)) (y ?l)) 2)
          )
        )
      )
    )
  :heuristic
    (sqrt
      (+
        (pow (- (x (at)) (x ?l)) 2)
        (pow (- (y (at)) (y ?l)) 2)
      )
    )
)
(:action goTo
  :parameters
    (?l - loc)
  :precondition
    (elem (at) ?l)
  :effect
    (and
      (assign (at) ?l)
      (increase (cost)
        (sqrt (+
          (pow (- (x (at)) (x ?l)) 2)
          (pow (- (y (at)) (y ?l)) 2)
        ))
      )
    )
)
)
```

A.2.2 Fichier de problème

```
   (define (problem lab0)
   (:domain labyrinth)
   (:objects
     A B C D E F G H - loc
   )
(:init
  (assign (at) A)
  (elem A B)
  (elem A C)
  (elem B D)
  (elem C E)
  (elem D F)
  (elem E G)
  (elem F H)
  (elem G H)
  (exists_path A D)
  (exists_path A E)
  (exists_path E H)
  (exists_path D H)
  (assign (x A) 0.)
  (assign (y A) 0.)
  (assign (x B) 2.)
  (assign (y B) 0.)
  (assign (x C) 2.)
  (assign (y C) -1.)
  (assign (x D) 4.)
  (assign (y D) 0.)
  (assign (x E) 4.)
  (assign (y E) -2.)
  (assign (x F) 4.)
  (assign (y F) 4.)
  (assign (x G) 6.)
  (assign (y G) -1.)
  (assign (x H) 8.)
  (assign (y H) 0.)
  (assign (cost) 0.)
)
(:goal
  (= (at) H)
)
(:metric
  (minimize
    (cost)
  )
)
(:heuristic
  (sqrt
    (+
      (pow (- (x (at)) (x H)) 2)
      (pow (- (y (at)) (y H)) 2)
    )
  )
)
(:run
  (moveTo H)
)
)
```

A.3 Blocks World - Formalisme HDS

```
(define (domain blocks_world)

(:requirements
  :strips
  :typing
  :hierarchical
  :macros
  :metaeffect
)

(:types
  block
)
(:functions
  (on_table ?b - block)
  ((on ?b - block) - block)
  (clear ?b - block)
  ((stack_base ?b - block) - block)
  (empty)
  ((holds) - block)

  (needed_on_table ?b - block)
  ((needed_on ?b) ?b2)
  (placed ?b)
  ((cost) - continuous)
)
```

```
(:metaAction resolve_problem
  :parameters
    ()
  :precondition
    ()
  :method m0
    :precondition
      ()
    :subtasks
      (:while (exists (?b - block)
                 (not (placed ?b)))
        (:pickone (?b - block) with
            (and
              (not (placed ?b))
              (or
                (needed_on_table ?b)
                (placed (needed_on ?b))
              )
            )
          (free_place_block ?b)
        )
      )
  :metaEffect
    (forall (?b - block)
      (and
      (when (needed_on_table ?b)
        (on_table ?b)
        )
        (when (not (needed_on_table ?b))
        (assign (on ?b) (needed_on ?b))
        )
      )
    )
  :heuristic
    (0.)
)
```

```
(:metaAction free_place_block
  :parameters
    (?b - block)
  :precondition
    ()
  :method m1
    :precondition
      (or
        (and (onTable ?b) (needed_on_table ?b))
        (= (on ?b) (neededOn ?b))
      )
    :subtasks
      ()
  :method m2
    :precondition
      (not (or
        (and (onTable ?b) (needed_on_table ?b))
        (= (on ?b) (neededOn ?b))
      ))
    :subtasks
      (:ordered
        (free_block ?b)
        (place_block ?b)
      )

  :metaEffect
    (placed ?b)
  :heuristic
    (0.)
)
(:metaAction place_block
  :parameters
    (?b - block)
  :precondition
    ()
  :method m1
    :precondition
      (needed_on_table ?b)
    :subtasks
      (:ordered
        (grab_block ?b)
        (put_on_table ?b)
      )
  :method m2
    :precondition
      (not (needed_on_table ?b))
    :subtasks
      (:ordered
        (free_block (needed_on ?b))
        (grab_block ?b)
        (put_on (needed_on ?b) ?b)
      )

  :metaEffect
    (placed ?b)
  :heuristic
    (0.)
)
```

```
(:metaAction free_block
  :parameters
    (?b - block)
  :precondition
    ()
  :method m1
    :precondition
      ()
):subtasks
      (:while (not (clear ?b))
        (:pickone (?b2 - block) with
            (and
              (clear ?b2)
              (=
                (stack_base ?b2)
                (stack_base ?b)
              )
            )
          (:ordered
            (grab_block ?b2)
            (put_on_table ?b2)
          )
        )
      )
  :metaEffect
    (clear ?b)
  :heuristic
    (0.)
)
(:action grab_block
  :parameters
    (?b - block)
  :precondition
    (and
      (empty)
      (clear ?b)
    )
  :effect
    (and
      (when (not (on_table ?b))
        (clear (on ?b))
      )
      (when (on_table ?b)
        (not (on_table ?b))
      )
      (not (empty))
      (not (clear ?b))
      (holds ?b)
    )
)
```

```
(:action put_on_table
  :parameters
    (?b - block)
  :precondition
    (and
      (not (empty))
      (holds ?b)
    )
  :effect
    (and
      (empty)
      (clear ?b)
      (on_table ?b)
      (assign (stack_base ?b) ?b)
    )
)
```

```
(:action put_on
  :parameters
    (?b1 ?b2 - block)
  :precondition
    (and
      (not (empty))
      (holds ?b2)
      (clear ?b1)
    )
  :effect
    (and
      (empty)
      (clear ?b2)
      (assign (on ?b2) ?b1)
      (assign (stack_base ?b2) (stack_base ?b1))
      (not (clear ?b1))
      (not (holds ?b2))
    )
)
)
```

A.4 Zeno Traveler - Formalisme HDS

```
(define (domain zeno)
(:requirements
  :strips
  :typing
  :hierarchical
  :macros
  :metaeffect
)
(:types
  place
  object - place
  aircraft - object
  person - object
  city - place
)
(:functions
  ((at ?o - object) - place)
  ((fuel ?a - aircraft) - continuous)
  ((distance ?c1 - city ?c2 - city) - continuous)
  ((slow-burn ?a - aircraft) - continuous)
  ((fast-burn ?a - aircraft) - continuous)
  ((capacity ?a - aircraft) - continuous)
  ((onboard ?a - aircraft) - continuous)
  ((zoom-limit ?a - aircraft) - continuous)
  ((total-fuel-used) - continuous)
  ((total-time) - continuous)
  ((hasMoved ?o - object) - bool)
  ((finished) - bool)
  ((inGoal ?o - object) - bool)
  ((goalAt ?o - object) - place)
)
```

```
(:metaAction mission
  :parameters
    ()
  :precondition
    ()
  :method m
    :precondition
      ()
    :subtasks
      (:while (or
            (exists (?a - aircraft)
              (imply (inGoal ?a)
                (not (= (at ?a) (goalAt ?a)))
              )
            )
            (exists (?p - person)
              (imply (inGoal ?p)
                (not (= (at ?p) (goalAt ?p)))
              )
            )
          )
        (:ordered
          (:while (not (finished))
            ;;prendre un avion qui n'a encore rien fait
            (:pickone (?p - aircraft) with (not (hasMoved ?p))
              (:ordered
                (set_unfinished)
                (:optional
                  (:ordered
                    (refuel ?p)
                    (set_finished)
                  )
                )
                (:optional
                  (:ordered
                    (:pickone (?p2 - person) with
                      (and
                        (not (finished))
                        (not (hasMoved ?p2))
                        (= (at ?p2) (at ?p))
                        (inGoal ?p2)
                        (not (= (at ?p2) (goalAt ?p2)))
                      )
                      (board ?p2 ?p)
                    )
                    (set_finished)
                  )
                )
```

```
(:optional
  (:ordered
    (:pickone (?p2 - person) with
      (and
        (not (finished))
        (not (hasMoved ?p2))
        (= (at ?p2) ?p)
        (inGoal ?p2)
        (= (at ?p) (goalAt ?p2))
      )
      (debark ?p2 ?p)
    )
    (set_finished)
  )
)
(:optional
  (:ordered
    (:pickone (?c - city) with
      (and
        (not (finished))
        (not (= (at ?p) ?c))
        (or
          ;;ya des personnes dans l'avion qui veulent y aller
          (exists (?p2 - person)
            (and
              (= (at ?p2) ?p)
              (inGoal ?p2)
              (= (goalAt ?p2) ?c)
            )
          )
          ;;ya des personnes à ladite destination qui veulent bouger
          (exists (?p2 - person)
            (and
              (= (at ?p2) ?c)
              (inGoal ?p2)
              (not (= (goalAt ?p2) ?c))
            )
          )
          ;;c'est la destination de l'avion
          (and (inGoal ?p) (= (goalAt ?p) ?c))
        )
      )
      (fly ?p ?c)
    )
    (set_finished)
  )
)
```

```
                (force)
                (set_finished)
                (:optional
                  (set_unfinished)
                )
              )
            )
          )
          (steptime)
        )
      )
  :metaEffect
    (forall (?o - object)
      (when (inGoal ?o)
        (assign (at ?o) (goalAt ?o))
      )
    )
  :heuristic
    (0)
)
(:action board
  :parameters
    (?p - person ?a - aircraft)
  :precondition
    (= (at ?p) (at ?a))
  :effect
    (and
      (assign (at ?p) ?a)
      (increase (onboard ?a) 1)
      (hasMoved ?a)
      (hasMoved ?p)
    )
)
(:action debark
  :parameters
    (?p - person ?a - aircraft)
  :precondition
    (= (at ?p) ?a)

  :effect
    (and
      (assign (at ?p) (at ?a))
      (decrease (onboard ?a) 1)
      (hasMoved ?a)
      (hasMoved ?p)
    )
)
```

```
(:action fly
  :parameters
    (?a - aircraft ?c - city)
  :precondition
    (>= (fuel ?a) (* (distance (at ?a) ?c) (slow-burn ?a)))

  :effect
    (and
      (increase (total-fuel-used) (* (distance (at ?a) ?c) (slow-burn ?a)))
      (decrease (fuel ?a) (* (distance (at ?a) ?c) (slow-burn ?a)))
      (assign (at ?a) ?c)
      (hasMoved ?a)
    )
)
(:action zoom
  :parameters
    (?a - aircraft ?c - city)
  :precondition
    (and
      (>= (fuel ?a) (* (distance (at ?a) ?c) (fast-burn ?a)))
      (<= (onboard ?a) (zoom-limit ?a))
    )
  :effect
    (and
      (increase (total-fuel-used) (* (distance (at ?a) ?c) (fast-burn ?a)))
      (decrease (fuel ?a) (* (distance (at ?a) ?c) (fast-burn ?a)))
      (assign (at ?a) ?c)
      (hasMoved ?a)
    )
)
(:action refuel
  :parameters
    (?a - aircraft)
  :precondition
    (> (capacity ?a) (fuel ?a))

  :effect
    (and
      (assign (fuel ?a) (capacity ?a))
      (hasMoved ?a)
    )
)
(:action set_finished
  :parameters
    ()
  :precondition
    ()
  :effect
    (finished)
)
```

```
(:action set_unfinished
  :parameters
    ()
  :precondition
    ()
  :effect
    (not (finished))
)
(:action force
  :parameters
    ()
  :precondition
    ()
  :effect
    (when (not (finished))
      (increase (total-time) 100000)
    )
)
(:action steptime
  :parameters
    ()
  :precondition
    ()
  :effect
    (and
      (increase (total-time) 1)
      (forall (?a - aircraft)
        (not (hasMoved ?a))
      )
      (forall (?p - person)
        (not (hasMoved ?p))
      )
    )
)
)
```

A.5 Freecell - Formalisme HDS

```
(define (domain freecell)

(:requirements
  :strips
  :typing
)
(:types
  card
  suitsort
  color
  denomination
  column
)
(:functions
  ((on ?c - card) - card)
  ((infreecell ?c - card) - bool)
  ((home ?c - card) - bool)
  ((bottom ?c - card) - bool)
  ((clear ?c - card) - bool)
  ((colname ?c - card) - card)
  ((suit ?c - card) - suitsort)
  ((value ?c - card) - continuous)
  ((col ?s - suitsort) - color)
  ((cardRank ?c - card) - continuous)
  ((freeCol) - continuous)
  ((freeUpCol) - continuous)
  ((cost) - continuous)
  ((remaining_cards) - continuous)
  ((finished) - bool)
)
```

```
(:metaAction mission
  :parameters
    ()
  :precondition
    ()
  :method m
    :precondition
      ()
    :subtasks
      (:while (not (= (remaining_cards) 0))
        (:ordered
          (set_unfinished)
          ;;monter toutes les cartes qui doivent l'être
          (:optional
            (:while (not (finished))
              (:ordered
                (:pickone (?c - card) with ((clear ?c))
                  (moveHome ?c)
                )
                (:optional
                  (set_finished)
                )
              )
            )
          )
          (force2)
```

```
;;proposer un déplacement de colonne vers une freecase
(:optional
  (:pickone () with (and  (not (finished)))
    (:ordered
      (:pickone (?c1 - card) with
        (and
          (clear ?c1)
          (not (bottom ?c1))
          (not (infreecell ?c1))
          (> (cardRank ?c1) 2)
        )
        (:pickone (?c2 - card) with
          (and
            (not (clear ?c2))
            (not (home ?c2))
            (= (colname ?c2) (colname ?c1))
            (not (bottom ?c2))
            (not (infreecell ?c2))
            (< (- (cardRank ?c1) (cardRank ?c2)) (+ (freeCol) (freeUpCol)))
            (= (- (cardRank ?c1) (cardRank ?c2)) (- (value ?c2) (value ?c1)))
            (forall (?c3 - card)
              (imply
                (and
                  (not (home ?c3))
                  (not (bottom ?c3))
                  (not (infreecell ?c3))
                  (= (colname ?c3) (colname ?c1))
                  (> (cardRank ?c3) (cardRank ?c2))
                )
                (and
                  (= (value ?c3) (- (value (on ?c3)) 1))
                  (not (= (col (suit ?c3)) (col (suit (on ?c3)))))
                )
              )
            )
          )
          (moveBlockToFree ?c1 ?c2)
        )
      )
      (set_finished)
    )
  )
)
```

```
;;proposer un déplacement de colonne vers une carte
(:optional
  (:pickone () with (and  (not (finished)))
    (:ordered
      (:pickone (?c1 - card) with
        (and
          (clear ?c1)
          (not (bottom ?c1))
          (not (infreecell ?c1))
        )
        (:pickone (?c2 - card) with
          (and
            (not (clear ?c2))
            (not (home ?c2))
            (= (colname ?c2) (colname ?c1))
            (not (infreecell ?c2))
            (< (- (cardRank ?c1) (cardRank ?c2)) (+ (freeCol) (freeUpCol)))
            (= (- (cardRank ?c1) (cardRank ?c2)) (- (value ?c2) (value ?c1)))
            (forall (?c3 - card)
              (imply
                (and
                  (not (home ?c3))
                  (not (bottom ?c3))
                  (not (infreecell ?c3))
                  (= (colname ?c3) (colname ?c1))
                  (> (cardRank ?c3) (cardRank ?c2))
                )
                (and
                  (= (value ?c3) (- (value (on ?c3)) 1))
                  (not (= (col (suit ?c3)) (col (suit (on ?c3)))))
                )
              )
            )
          )
          (:pickone (?c3 - card) with
            (and
              (clear ?c3)
              (not (infreecell ?c3))
              (not (= (colname ?c2) (colname ?c3)))
              (= (value ?c2) (- (value ?c3) 1))
              (not (= (col (suit ?c2)) (col (suit ?c3))))
            )
            (moveBlockToCard ?c1 ?c2 ?c3)
          )
        )
      )
      (set_finished)
    )
  )
)
```

```
;;proposer un déplacement d'une carte vers une freecell
(:optional
  (:pickone () with (not (finished))
    (:ordered
      (:pickone (?c - card) with
          (and
            (not (home ?c))
            (not (infreecell ?c))
            (clear ?c)
          )
        (moveUpCase ?c)
      )
      (set_finished)
    )
  )
)
;;mettre une carte dans une colonne libre
(:optional
  (:pickone () with (not (finished))
    (:ordered
      (:pickone (?c - card) with
          (and
            (not (home ?c))
            (not (bottom ?c))
            (clear ?c)
          )
        (moveFreeColumn ?c)
      )
      (set_finished)
    )
  )
)
```

```
          ;;mettre une carte sur une autre
          (:optional
            (:pickone () with (not (finished))
              (:ordered
                (:pickone (?c1 - card) with (clear ?c1)
                  (:pickone (?c2 - card) with
                    (and
                      (not (infreecell ?c2))
                      (clear ?c2)
                      (not (= ?c1 ?c2))
                      (= (value ?c1) (- (value ?c2) 1))
                      (not (= (col (suit ?c1)) (col (suit ?c2))))
                    )
                    (moveToCard ?c1 ?c2)
                  )
                )
                (set_finished)
              )
            )
          )
          (force)
        )
      until
        (= (remaining_cards) 0)
      )
  :metaEffect
    (forall (?c - card)
      (home ?c)
    )
  :heuristic
    (remaining_cards)
)
```

```
(:action moveBlockToFree
  :parameters
    (?c1 ?c2 - card)
  :precondition
    (and
      (clear ?c1)
      (not (bottom ?c1))
      (not (infreecell ?c1))
      (> (cardRank ?c1) 2)

      (not (clear ?c2))
      (not (home ?c2))
      (= (colname ?c2) (colname ?c1))
      (not (bottom ?c2))
      (not (infreecell ?c2))
      (< (- (cardRank ?c1) (cardRank ?c2)) (+ (freeCol) (freeUpCol)))
      (= (- (cardRank ?c1) (cardRank ?c2)) (- (value ?c2) (value ?c1)))
      (forall (?c3 - card)
        (imply
          (and
            (not (home ?c3))
            (not (bottom ?c3))
            (not (infreecell ?c3))
            (= (colname ?c3) (colname ?c1))
            (> (cardRank ?c3) (cardRank ?c2))
          )
          (and
            (= (value ?c3) (- (value (on ?c3)) 1))
            (not (= (col (suit ?c3)) (col (suit (on ?c3)))))
          )
        )
      )
    )
```

```
  :effect
    (and
      (increase (cost) (- (cardRank ?c) (cardRank ?c2))
      (forall (?c - card)
        (when
          (and
            (not (= ?c ?c2))
            (not (home ?c))
            (= (colname ?c2) (colname ?c))
            (not (bottom ?c))
            (not (infreecell ?c))
            (> (cardRank ?c) (cardRank ?c2))
          )
          (and
            (assign (colname ?c) ?c2)
            (decrease (cardRank ?c) (cardRank ?c2))
            (increase (cardRank ?c) 1)
          )
        )
      )
      (clear (on ?c2))
      (assign (colname ?c2) ?c2)
      (assign (cardRank ?c2) 1)
      (bottom ?c2)
      (decrease (freeCol) 1)
    )
)
```

```
(:action moveBlockToCard
  :parameters
    (?c1 ?c2 ?c3 - card)
  :precondition
    (and
      (clear ?c1)
      (not (bottom ?c1))
      (not (infreecell ?c1))
      (not (clear ?c2))
      (not (home ?c2))
      (= (colname ?c2) (colname ?c1))
      (not (infreecell ?c2))
      (< (- (cardRank ?c1) (cardRank ?c2)) (+ (freeCol) (freeUpCol)))
      (= (- (cardRank ?c1) (cardRank ?c2)) (- (value ?c2) (value ?c1)))
      (forall (?c - card)
        (imply
          (and
            (not (home ?c))
            (not (bottom ?c))
            (not (infreecell ?c))
            (= (colname ?c) (colname ?c1))
            (> (cardRank ?c) (cardRank ?c2))
          )
          (and
            (= (value ?c) (- (value (on ?c)) 1))
            (not (= (col (suit ?c)) (col (suit (on ?c)))))
          )
        )
      )
      (clear ?c3)
      (not (infreecell ?c3))
      (not (= (colname ?c2) (colname ?c3)))
      (= (value ?c2) (- (value ?c3) 1))
      (not (= (col (suit ?c2)) (col (suit ?c3))))
    )
```

```
  :effect
    (and
      (increase (cost) (- (cardRank ?c2) (cardRank ?c3))
      (forall (?c - card)
        (when
          (and
            (not (= ?c ?c2))

            (not (home ?c))
            (= (colname ?c2) (colname ?c))
            (not (bottom ?c))
            (not (infreecell ?c))
            (> (cardRank ?c) (cardRank ?c2))
          )
          (and
            (assign (colname ?c) ?c3)
            (decrease (cardRank ?c) (cardRank ?c2))
            (increase (cardRank ?c) (cardRank ?c3))
            (increase (cardRank ?c) 1)
          )
        )
      )
      (when (not (bottom ?c2))
        (clear (on ?c2))
      )
      (when (bottom ?c2)
        (and
          (not (bottom ?c2))
          (increase (freeCol) 1)
        )
      )
      (assign (colname ?c2) ?c3)
      (assign (cardRank ?c2) (cardRank ?c3))
      (increase (cardRank ?c2) 1)
    )
)
```

```
(:action moveToCard
  :parameters (?c1 ?c2 - card)
  :precondition
    (and
      (not (home ?c1))
      (not (infreecell ?c2))
      (clear ?c1)
      (clear ?c2)
      (not (= ?c1 ?c2))
      (= (value ?c1) (- (value ?c2) 1))
      (not (= (col (suit ?c1)) (col (suit ?c2))))
    )
  :effect
    (and
      (when (and (not (bottom ?c1)) (not (infreecell ?c1)))
        (clear (on ?c1))
      )
      (when (infreecell ?c1)
        (and
          (not (infreecell ?c1))
          (increase (freeUpCol) 1)
        )
      )
      (when (bottom ?c1)
        (and
          (not (bottom ?c1))
          (increase (freeCol) 1)
        )
      )
      (not (clear ?c2))
      (assign (colname ?c1) (colname ?c2))
      (assign (cardRank ?c1) (+ (cardRank ?c2) 1))
      (assign (on ?c1) ?c2)
      (increase (cost) 1)
    )
)
```

```
(:action moveFreeColumn
  :parameters
    (?c - card)
  :precondition
    (and
      (not (home ?c))
      (not (bottom ?c))
      (clear ?c)
      (> (freeCol) 0)
    )
  :effect
    (and

      (when (not (infreecell ?c))
        (clear (on ?c))
      )
      (when (infreecell ?c)
        (and
          (increase (freeUpCol) 1)
          (not (infreecell ?c))
        )
      )
      (bottom ?c)
      (assign (colname ?c) ?c)
      (assign (cardRank ?c) 1)
      (decrease (freeCol) 1)
      (increase (cost) 1)
    )
)
(:action moveUpCase
  :parameters
    (?c - card)
  :precondition
    (and
      (not (home ?c))
      (not (infreecell ?c))
      (clear ?c)
      (> (freeUpCol) 0)
    )
  :effect
    (and
      (when (not (bottom ?c))
        (clear (on ?c))
      )
      (when (bottom ?c)
        (and
          (increase (freeCol) 1)
          (not (bottom ?c))
        )
      )
      (infreecell ?c)
      (decrease (freeUpCol) 1)
      (increase (cost) 1)
    )
)
```

```
(:action moveHome
  :parameters
    (?c - card)
  :precondition
    (and
      (clear ?c)
      (imply (not (= (value ?c) 1))
        (exists (?c1 - card)
          (and
            (= (suit ?c) (suit ?c1))
            (= (value ?c) (+ (value ?c1) 1))
            (home ?c1)
          )
        )
      )
    )
  :effect
    (and
      (when (and (not (bottom ?c)) (not (infreecell ?c)) )
        (clear (on ?c))
      )
      (when (infreecell ?c)
        (and
          (increase (freeUpCol) 1)
          (not (infreecell ?c))
        )
      )
      (when (bottom ?c)
        (and
          (increase (freeCol) 1)
          (not (bottom ?c))
        )
      )
      (home ?c)
      (not (clear ?c))
      (increase (cost) 1)
      (decrease (remaining_cards) 1)
    )
)
(:action set_finished
  :parameters
    ()
  :precondition
    ()
  :effect
    (finished)
)
(:action set_unfinished
  :parameters
    ()
  :precondition
    ()
  :effect
    (not (finished))
)
```

```
(:action force
  :parameters
    ()
  :precondition
    ()
  :effect
    (when (not (finished))
      (increase (cost) 1000)
    )
)
```

```
(:action force2
  :parameters
    ()
  :precondition
    ()
  :effect
    (when
      (exists (?c - card)
        (or
          (and
            (clear ?c)
            (> (value ?c) 2)
            (exists (?c2 - card)
              (and
                (= (value ?c) (+ (value ?c2) 1))
                (= (suit ?c) (suit ?c2))
                (home ?c2)
              )
            )
            (forall (?c2 - card)
              (imply
                (and
                  (= (value ?c) (+ (value ?c2) 1))
                  (not (= (col (suit ?c)) (col (suit ?c2))))
                )
                (home ?c2)
              )
            )
          )
          (and
            (clear ?c)
            (= (value ?c) 2)
            (exists (?c2 - card)
              (and
                (= (value ?c) (+ (value ?c2) 1))
                (= (suit ?c) (suit ?c2))
                (home ?c2)
              )
            )
          )
          (and
            (clear ?c)
            (= (value ?c) 1)
          )
        )
      )
      (increase (cost) 1000)
    )
)
)
```

A.6 Explore and Guide - Formalisme HDS

```
(define (domain emptyZone)

(:requirements
  :strips
  :typing
  :hierarchical
  :macros
  :metaeffect
)
(:types
  loc
  veh
  zone
)
(:functions
  ((pos) - loc)
  ((at ?v - veh) - loc)
  ((escape ?v - veh) - zone)
  ((output ?v - veh) - loc)
  ((treated ?v - veh) - bool)
  ((x ?l - loc) - continuous)
  ((y ?l - loc) - continuous)
  ((inZone ?l - loc ?z - zone) - bool)
  ((explored ?l - loc) - bool)
  ((cost) - continuous)
)
```

```
(:metaAction mission
  :parameters
    (?base - loc)
  :precondition
    ()
  :method miss
    :precondition
      ()
    :subtasks
      (:ordered
        (:while (exists (?v - veh)
                  (not (treated ?v)))
          (:pickone (?v - veh) with
                  (not (treated ?v))
            (treatVeh ?v)
          )
        )
        (:optional
          (goto ?base)
        )
      )
  :metaEffect
    (and
      (forall (?v - veh)
        (and
          (forall (?l - loc)
            (when (inZone ?l (escape ?v))
              (increase (cost) 2.)
            )
          )
          (treated ?v)
          (increase (cost) 5.)
        )
      )
    )
  :heuristic
    (0.)
)
```

```
(:metaAction treatVeh
  :parameters
    (?v - veh)
  :precondition
    (and
      (not (treated ?v))
    )
  :method tr
    :precondition
      ()
    :subtasks
      (:ordered
        (explZone (escape ?v))
        (guide ?v)
      )
  :metaEffect
    (and
      (increase (cost)
        (sqrt (+
          (pow (- (x (pos)) (x (at ?v))) 2)
          (pow (- (y (pos)) (y (at ?v))) 2)
        ))
      )
      (forall (?l - loc)
        (when (and (inZone ?l (escape ?v))
                   (not (explored ?l))
              )
          (and
            (explored ?l)
            (increase (cost) 2.)
          )
        )
      )
      (treated ?v)
      (assign (pos) (output ?v))
      (increase (cost) 5.)
    )
  :heuristic
    (0.)
)
```

```
(:metaAction explZone
  :parameters
    (?z - zone)
  :precondition
    ()
  :method explore
    :precondition
      ()
    :subtasks
      (:while
          (exists (?l1 - loc)
            (imply (inZone ?l1 ?z)
              (not (explored ?l1))
            )
          )
        (:pickone (?l - loc) with
            (and (inZone ?l ?z)
                 (not (explored ?l))
            )
            (:ordered
              (:optional
                (goto ?l)
              )
              (expl ?l)
            )
          )
        )
  :metaEffect
    (forall (?l - loc)
      (when (and (inZone ?l ?z)
                 (not (explored ?l))
            )
        (and
          (explored ?l)
          (increase (cost) 2.)
        )
      )

    )
  :heuristic
    (0.)
)
```

```
(:metaAction guide
  :parameters
    (?v - veh)
  :precondition
    (and
      (not (treated ?v))
      (forall (?l - loc)
        (imply (inZone ?l (escape ?v))
          (explored ?l)
        )
      )
    )
  :method explore
    :precondition
      ()
    :subtasks
      (:ordered
        (:optional
          (goto (at ?v))
        )
        (track ?v)
      )
  :metaEffect
    (and
      (treated ?v)
      (assign (pos) (output ?v))
      (increase (cost) 5.)
    )
  :heuristic
    (0.)
)
(:action goto
  :parameters
    (?l - loc)
  :precondition
    ()
  :effect
    (and
      (increase (cost)
        (sqrt (+
          (pow (- (x (pos)) (x ?l)) 2)
          (pow (- (y (pos)) (y ?l)) 2)
        ))
      )
      (assign (pos) ?l)
    )
)
(:action expl
  :parameters
    (?l - loc)
  :precondition
    (and
      (= (pos) ?l)
      (not (explored ?l))
    )
  :effect
    (and
      (increase (cost) 1.)
      (explored ?l)
    )
)
(:action track
  :parameters
    (?v - veh)
  :precondition
    (and
      (= (pos) (at ?v))
      (forall (?l - loc)
        (imply (inZone ?l (escape ?v))
          (explored ?l)
        )
      )
    )
  :effect
    (and
      (increase (cost) 5.)
      (assign (pos) (output ?v))
      (treated ?v)
    )
))
```

A.7 Déminage - Formalisme HDS

```
(define (domain mines)

(:requirements
  :strips
  :typing
)
(:types
  loc
  zone
  veh
  heli - veh
  sub - veh
)
(:functions
  ((at ?v - veh) - loc)
  ((surf ?s - sub) - bool)
  ((inCom ?s - sub) - bool)
  ((x ?l - loc) - continuous)
  ((y ?l - loc) - continuous)
  ((inZone ?l - loc ?z - zone) - bool)
  ((exploredSurf ?l - loc) - bool)
  ((clearedSurf ?z - zone) - bool)
  ((exploredDeep ?l - loc) - bool)
  ((clearedDeep ?z - zone) - bool)
  ((zoneNumber ?z - zone) - continuous)
  ((time ?v - veh) - continuous)
  ((lastCom) - continuous)
  ((maxWait) - continuous)
  ((remCasesSurf) - continuous)
  ((remCasesDeep) - continuous)
)
```

```
(:metaAction mission
  :parameters
    (?heli - uav ?sub - auv)
  :precondition
    ()
  :method m0
    :precondition
      ()
    :subtasks
      (:ordered
        (:pickone (?z - zone) with
            (and
              (not (clearedSurf ?z))
              (forall (?z2 - zone)
                (>=
                  (zoneNumber ?z2)
                  (zoneNumber ?z)
                )
              )
            )
          (:ordered
            (exploreSurf ?z ?heli)
            (sync ?sub)
          )
        )
```

```
        (:while (not (forall (?z - zone)
                     (clearedSurf ?z)))
          (:pickone (?z - zone) with
              (and
                (not (clearedSurf ?z))
                (forall (?z2 - zone)
                  (imply
                    (not (clearedSurf ?z2))
                    (>=
                      (zoneNumber ?z2)
                      (zoneNumber ?z)
                    )
                  )
                )
              )
            (:pickone (?z1 - zone) with
                (and
                  (not (clearedDeep ?z))
                  (forall (?z2 - zone)
                    (imply (not (clearedDeep ?z2))
                      (>=
                        (zoneNumber ?z2)
                        (zoneNumber ?z1)
                      )
                    )
                  )
                )
              (:parallel
                (exploreDeep ?z1 ?sub)
                (:ordered
                  (exploreSurf ?z ?heli)
                  (assistSub ?z ?heli)
                )
              )
            )
          )
        )
        (:pickone (?z - zone) with
            (not (clearedDeep ?z))
          (:parallel
            (exploreDeep ?z ?sub)
            (assistSub ?z ?heli)
          )
        )
      )
  :metaEffect
    (and
      (forall (?l - loc)
        (exploredSurf ?l)
        (exploredDeep ?l)
      )
    )
  :heuristic
    (0.)
)

  (:metaAction exploreSurf
    :parameters
      (?h - heli ?z - zone)
    :precondition
      ()
    :method expl
      :precondition
        ()
      :subtasks
        (:ordered
          (initZoneSurf ?z)
          (:while (exists (?l - loc)
                    (and
                      (not (exploredSurf ?l))
                      (inZone ?l ?z)
                    )
                  )
            (:pickone (?l - loc) with
                (and
                  (not (exploredSurf ?l))
                  (inZone ?l ?z)
                )
              (:ordered
                (:optional
                  (goto ?h ?l)
                )
                (explSurf ?h ?l)
                (:optional
                  (rdvHeli ?h)
                )
              )
            )
          )
        )
      :metaEffect
        (and
          (clearedSurf ?z)
          (forall (?l - loc)
            (:when (inZone ?l ?z)
                (exploredSurf ?l)
            )
          )
        )
      :heuristic
        (* (remCasesSurf) 2)
  )
```

```
(:metaAction exploreDeep
  :parameters
    (?s - sub ?z - zone)
  :precondition
    ()
  :method expl
    :precondition
      ()
    :subtasks
      (:ordered
        (initZoneDeep ?z)
        (:optional
          (:pickone (?l - loc) with
              (inZone ?l ?z)
            (moveto ?s ?l)
          )
        )
        (:while (exists (?l - loc)
                  (and
                    (not (exploredDeep ?l)))
                    (inZone ?l ?z)
                  ))
          (:pickone (?l - loc) with
              (and
                (not (exploredDeep ?l))
                (inZone ?l ?z)
              )
            (:ordered
              (:optional
                (goto ?s ?l)
              )
              (explDeep ?s ?l)
              (:optional
                (rdvSub ?s)
              )
            )
          )
        )
      )
  :metaEffect
    (and
      (clearedDeep ?z)
      (forall (?l - loc)
        (:when (inZone ?l ?z)
          (exploredDeep ?l)
        )
      )
    )
  :heuristic
    (* (remCasesDeep) 2)
)
```

```
(:metaAction assistSub
  :parameters
    (?z - zone ?v - veh)
  :precondition
    ()
  :method assist
    :precondition
      ()
    :subtasks
      (:while (not (clearedDeep ?z))
        (:ordered
          (sync ?v)
          (rdvHeli ?v)
        )
      )
  :metaEffect
    ()
  :heuristic
    (0)
)
(:metaAction rdvSub
  :parameters
    (?v - veh)
  :precondition
    ()
  :method wait
    :precondition
      ()
    :subtasks
      (:ordered
        (surface ?v)
        (beginComSub ?v)
        (endComSub ?v)
        (dive ?v)
      )
  :metaEffect
    (increase (time ?v) 3)
  :heuristic
    (0.)
)
```

```
(:metaAction rdvHeli
  :parameters
    (?v - veh)
  :precondition
    ()
  :method wait
    :precondition
      ()
    :subtasks
      (:pickone (?s - sub)
        (:ordered
          (goto ?heli (at ?s))
          (comHeli ?heli)
        )
      )
  :metaEffect
    (and
      (increase (time ?v) 2)
      (assign (lastCom) (time ?v))
    )
  :heuristic
    (0.)
)
```

```
(:metaAction moveto
  :parameters
    (?v - veh ?l - loc)
  :precondition
    (isSubmarine ?v)
  :method move
    :precondition
      ()
    :subtasks
      (:while (not (= (at ?v) ?l))
        (:pickone (?l1 - loc) with
            (=
              (+
                (abs (- (x ?l1)(x (at ?v))))
                (abs (- (y ?l1)(y (at ?v))))
              )
              1
            )
          (goto ?v ?l1)
        )
      )
  :metaEffect
    (and
      (increase (time ?v)
        (+
          (abs (- (x ?l)(x (at ?v))))
          (abs (- (y ?l)(y (at ?v))))
        )
      )
      (assign (at ?v) ?l)
    )
  :heuristic
    (+
      (abs (- (x ?l)(x (at ?v))))
      (abs (- (y ?l)(y (at ?v))))
    )
)
(:action comHeli
  :parameters
    (?v - veh)
  :precondition
    (exists (?s - sub)
      (and
        (= (at ?s) (at ?v))
        (= (time ?s) (time ?v))
        (inCom ?s)
      )
    )
  :effect
    (and
      (increase (time ?v) 1)
      (assign (lastCom) (time ?v))
    )
)
```

```
(:action beginComSub
  :parameters
    (?v - veh)
  :precondition
    (and
      (surf ?v)
      (not (inCom ?v))
    )
  :effect
    (inCom ?v)
)
(:action endComSub
  :parameters
    (?v - veh)
  :precondition
    (and
      (surf ?v)
      (inCom ?v)
    )
  :effect
    (and
      (not (inCom ?v))
      (increase (time ?v) 1)
    )
)
(:action dive
  :parameters
    (?v - veh)
  :precondition
    (and
      (surf ?v)
      (not (inCom ?v))
    )
  :effect
    (and
      (not (surf ?v))
      (increase (time ?v) 1)
    )
)
(:action surface
  :parameters
    (?v - veh)
  :precondition(:action surface
  :parameters
    (?v - veh)
  :precondition
    (not (surf ?v))
  :effect
    (and
      (surf ?v)
      (increase (time ?v) 1)
    )
)
```

```
(:action gotomarine ?v)
      (=
        (+
          (abs (- (x ?l)(x (at ?v))))
          (abs (- (y ?l)(y (at ?v))))
        )
        1
      )
    )
  :effect
    (increase (time ?h) 1)
)
(:action explSurf
  :parameters
    (?h - heli ?l - loc)
  :precondition
    (and
      (= (at ?h) ?l)
      (not (exploredSurf ?l))
    )
  :effect
    (and
      (exploredSurf ?l)
      (decrease (remCasesSurf) 1)
      (increase (time ?h) 1)
    )
)
(:action explDeep
  :parameters
    (?s - sub ?l - loc)
  :precondition
    (and
      (< (time ?s) (+ (lastCom) (maxWait)))
      (= (at ?s) ?l)
      (not (exploredDeep ?l))
    )
  :effect
    (and
      (exploredDeep ?l)
      (decrease (remCasesDeep) 1)
      (increase (time ?s) 1)
    )
)
(:action sync
  :parameters
    (?v - veh)
  :precondition
    ()
  :effect
    (forall (?v1 - veh)
      (when (> (time ?v1) (time ?v))
        (assign (time ?v) (time ?v1))
      )
    )
)
```

```
(:action initZoneDeep
  :parameters
    (?z - zone)
  :precondition
    ()
  :effect
    (and
      (assign (remCasesDeep) 0)
      (forall (?l - loc)
        (when (inZone ?l ?z)
          (increase (remCasesDeep) 1)
        )
      )
    )
)
```

```
(:action initZoneSurf
  :parameters
    (?z - zone)
  :precondition
    ()
  :effect
    (and
      (assign (remCasesSurf) 0)
      (forall (?l - loc)
        (when (inZone ?l ?z)
          (increase (remCasesSurf) 1)
        )
      )
    )
)
)
```

Planification multi-niveaux avec expertise humaine

La planification automatique est un champ de recherche de l'Intelligence Artificielle qui vise à calculer automatiquement une séquence d'actions menant d'un état initial donné à un but souhaité. De nombreux sous-domaines de la planification ont été explorés, certains faisant l'hypothèse d'actions à effets déterministes. Même sous cette hypothèse, résoudre des problèmes réalistes est un problème difficile car trouver un chemin solution peut demander d'explorer un nombre d'états croissant exponentiellement avec le nombre de variables d'état. Pour faire face à cette explosion combinatoire, les algorithmes performants ont recours aux heuristiques, qui guident la recherche grâce à des solutions optimistes ou approximatives. Les méthodes hiérarchiques, elles, décomposent itérativement le problème en sous-problèmes plus petits et plus simples.
Dans une grande majorité des cas, le planificateur doit prendre en compte un certain nombre de contraintes telles que des phases d'actions prédéfinies ou des protocoles. Ces contraintes aident généralement à résoudre le problème de planification car elles élaguent un grand nombre de branches de l'arbre de recherche, où ces contraintes ne sont pas vérifiées. Elles peuvent être données par un spécialiste du domaine (ce qui est souvent le cas dans les applications réelles, comme les missions militaires) ou extraites automatiquement du modèle au préalable. Dans cet article, nous supposons que ces contraintes sont connues et données au planificateur. Nous proposons alors une nouvelle méthode pour modéliser et résoudre des problèmes de planification déterministes basée sur une approche hiérarchique et heuristique utilisant ces contraintes à son avantage.
Nous nous sommes inspirés des formalismes de programmation structurée afin de fournir à l'utilisateur un cadre de travail plus intuitif pour la modélisation des domaines de planification hiérarchique. D'autre part, nous avons proposé un algorithme de planification capable d'exploiter ce formalisme et composer des stratégies à différents niveaux de granularité, ce qui lui permet de planifier rapidement une stratégie globale, tout en étant en mesure de pallier aux difficultés rencontrées à plus bas niveau. Cet algorithme a fait ses preuves face au principal planificateur HTN, SHOP2, sur des problèmes de planification classique.

Mots-clés : Décomposition hiérarchique, planification classique, expertise humaine, optimisation globale

Multi-level Planning and Human Expertise

Automated planning is a field of Artificial Intelligence which aims at automatically computing a sequence of actions that lead to some goals from a given initial state. Many subareas have been explored, some assuming that effects of actions are deterministic. Even in this case, solving realistic problems is challenging because finding a solution path may require to explore an exponential number of states with regard to the number of state variables. To cope with this combinatorial explosion, efficient algorithms use heuristics, which guide the search towards optimistic or approximate solutions. Remarkably, hierarchical methods iteratively decompose the planning problem into smaller and much simpler ones.
In a vast majority of problems, the planner must deal with constraints, such as multiple predefined phases or protocols. Such constraints generally help solving the planning problem, because they prune lots of search paths where these constraints do not hold. They can be given by an expert of the problem to solve --- which is often the case in many realistic applications such as military missions --- or beforehand automatically deduced from the model. In this paper, we assume that these constraints are known and given to the planner. We thus propose a new method to model and solve a deterministic planning problem, based on a hierarchical and heuristic approach and taking advantage of these constraints.
We inspired ourselves from structured programming formalisms in order to offer a more intuitive modeling framework in the domain of hierarchical planning to the user. We also proposed a planning algorithm able to exploit this formalism and make strategies at various levels of granularity, thus allowing to plan quickly a global strategy, while still being able to overcome the difficulties at lower level. This algorithm showed its performances compared with the main HTN planner, SHOP2, on classical planning problems.

Keywords: Hierarchical breakdown, classical planning, human expertise, global optimization