UNIVERSITÉ TOULOUSE III − PAUL SABATIER
U.F.R. Mathématiques − Informatique − Gestion

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ TOULOUSE III

Discipline : Informatique

présentée et soutenue

par

Vincent VIDAL

le 10 juillet 2001

Recherche dans les graphes de planification, satisfiabilité et stratégies de moindre engagement

Les systèmes LCGP et LCDPP

Directeur de thèse :
Michel CAYROL

JURY

M. Michel CAYROL	Professeur d'université IRIT − Université Paul Sabatier	(examinateur)
Mme Marie-Odile CORDIER	Professeur d'université IRISA − Université de Rennes	(président)
M. Malik GHALLAB	Directeur de recherche LAAS − CNRS	(examinateur)
M. Éric JACOPIN	Enseignant-chercheur, HDR CREC Saint-Cyr	(rapporteur)
M. Philippe LABORIE	Ingénieur ILOG − Gentilly	(examinateur)
M. Pierre RÉGNIER	Maître de conférences IRIT − Université Paul Sabatier	(examinateur)

M. Héctor GEFFNER	Professeur d'université Universidad Simón Bolívar Caracas, Venezuela	(rapporteur, absent)

Remerciements

Je remercie tout d'abord Héctor Geffner (Professeur à l'université Simón Bolívar, Caracas) et Éric Jacopin (Enseignant-chercheur du ministère de la défense, CREC Saint-Cyr) pour l'intérêt qu'ils ont porté à mon travail en acceptant d'en être les rapporteurs.

Je remercie également Marie-Odile Cordier (Professeur à l'université de Rennes), Malik Ghallab (directeur de recherche au LAAS−CNRS) et Philippe Laborie (Industriel, ILOG) pour avoir accepté de participer à mon jury de thèse.

Mon intérêt pour l'Intelligence Artificielle, en dehors de mes lectures de science-fiction, est né des cours de Michel Cayrol, puis par la suite de ceux de Claudette Cayrol. Je les remercie donc tous les deux pour m'avoir fait découvrir cette discipline, qui ne m'a apporté que des satisfactions. Je remercie particulièrement Michel pour m'avoir accepté dans son équipe, pour m'avoir laissé une très grande liberté dans mes recherches tout en sachant les recentrer quand c'était nécessaire, pour m'avoir appris la rigueur scientifique (ou du moins m'avoir mis sur la voie), pour son dynamisme, sa disponibilité et sa bonne humeur (parfois un peu altérée par mon opiniâtreté lors de certaines discussions assez animées)...

Je remercie bien sûr Pierre Régnier pour avoir encadré mon stage de DEA et co-encadré ma thèse, pour son aide et son intérêt constants tant dans mon travail que dans la rédaction des articles et de la thèse, et enfin tout simplement pour avoir été parfaitement complémentaire avec Michel.

Je remercie tous les membres actuels de l'équipe RPDMP pour leur gentillesse et leur bonne humeur, avec une mention spéciale pour Salem qui par son optimisme et son sens de l'humour m'a bien souvent remonté le moral lors de moments de doute (même s'il ne s'en est pas forcément aperçu...). Évidemment je n'oublie pas ceux qui ne font plus partie de cette équipe, et dont l'accueil a fait que j'ai eu envie d'y rester : Laurent, Fred, Régis... et bien sûr Daniel qui a laissé un grand vide en partant à l'autre bout de la planète. Je remercie également toutes les personnes de l'IRIT et des laboratoires voisins qui ont croisé ma route.

Merci aussi à mes collègues du bureau 301, et à tous ceux avec qui je partage les repas au RU, les pauses cafés, et autres moments de détente (pots de thèse...) : Céline, Christophe, Hélène, Jean-François, Jérôme, Nathalie, Nicolas, Olivier, Philippe, Pierre, Rami, Souhila, Sylvain, Sylvie, Thomas, Vincent, Yannick... pardon si j'en oublie.

Merci à tous ceux qui me rappellent qu'il y a une vie en dehors de l'IRIT, qui vaut la peine d'être vécue : Thomas et Céline, Gilles et la bande à Blagnac, Nikolaï, Fredo et Lydie, Fred, Véro, Guillaume, Wilfried, Dada et Caro, Jérôme, Virgo et Éric, Julien, Vincent et Nathalie, Damien et Nathalie... et tous les autres.

Enfin, un grand merci à mes parents et à ma soeur qui m'ont apporté leur soutien constant et ont supporté mes humeurs, ainsi qu'à Lisa, mon rayon de soleil personnel.

Sommaire

I. Introduction

1. Introduction au domaine

L'objectif de la *planification* est de fournir à un système (robotique, informatique...) la capacité de raisonner pour interagir avec son environnement de façon autonome, afin d'atteindre les objectifs qui lui ont été assignés. On peut distinguer plusieurs types de planification, suivant le niveau d'abstraction auquel on se situe. Par exemple, *la planification de mouvements* doit permettre à un système robotique (un robot autonome, comme le robot Pathfinder envoyé sur Mars, ou bien un robot employé dans une chaîne de production en usine, etc.) d'effectuer des déplacements dans l'espace. Il s'agit ici d'une planification de bas niveau, par rapport à une action globale qui serait par exemple "se déplacer du point A au point B puis prélever un échantillon du sol". Afin d'effectuer le trajet du point A au point B, le robot doit exploiter sa capacité de vision pour planifier son déplacement, en fonction des propriétés topologiques du terrain : il doit tenir compte de la nature du sol et des obstacles qui se dressent sur son chemin. Il doit aussi prendre en compte les erreurs d'estimation qu'il va commettre sur sa position, à moins de posséder un système de positionnement global très précis. Il s'agit ici d'un raisonnement de bas niveau, dans le sens où il participe d'une tâche plus globale qui consiste à se déplacer d'un endroit vers un autre dans le but précis d'effectuer un prélèvement. Ces deux actions, le déplacement et le prélèvement, font certainement partie d'un plan plus complexe : sortir d'une navette, effectuer une série de prélèvements en plusieurs endroits, et ramener les échantillons à la navette.

Suivant le niveau d'abstraction auquel on se situe, on voit apparaître une différence fondamentale dans la nature de l'information traitée. En effet, pour la planification de mouvements, il faut prendre en compte les propriétés physiques de l'environnement : l'information sur laquelle on travaille est un ensemble de données numériques représentant l'espace dans lequel se trouve le robot. La planification de plus haut niveau, la *planification de tâches* ou *planification d'actions*, traite des données qui sont essentiellement de nature *symbolique* : le point A, le point B et la navette sont des endroits de l'espace, une action possible (c'est-à-dire que l'on sait pouvoir effectuer dans certaines conditions) est de se déplacer de A vers B, une autre action possible est d'effectuer un prélèvement... Peu importent les propriétés physiques des informations. La préoccupation principale est le but global assigné au robot : ramener un certain nombre d'échantillons à la navette. L'objectif étant de rendre le robot capable d'agir de façon autonome dans son environnement, c'est-à-dire sans intervention humaine, la planification doit répondre à la question : quelles actions l'agent doit-il effectuer et dans quel ordre ?

La difficulté de ce processus apparaît suivant le degré de prise en compte des réalités de l'univers. En effet, si l'agent évolue dans le monde réel, il faut tenir compte de la nature dynamique et plus ou moins imprévisible de ce dernier. Les deux extrêmes sont les suivants :

- Une première technique consiste à calculer un plan d'actions sans tenir compte de la nature dynamique du monde et des effets imprévus des actions de l'agent. Un problème survenant lors de l'exécution du plan sera géré au moment où il se présente, et plusieurs techniques peuvent alors être utilisées : replanifier complètement, ou bien tenter de réparer le plan suivant les critères considérés. On suppose en outre que l'agent a une connaissance parfaite de l'univers et des effets de ses actions ; un défaut dans ces connaissances sera géré comme un événement imprévu.
- Une seconde technique consiste à calculer un plan d'actions en tenant compte de tout événement pouvant se produire et en tenant compte du fait que la connaissance de l'univers est imparfaite. Il va alors falloir que le plan d'actions de l'agent soit capable de gérer tout événement imprévu, que celui-ci soit lié à l'état

de l'univers où aux effets des actions. Un tel plan d'actions est par nature beaucoup plus complexe que pour la première technique : l'agent doit par exemple effectuer des observations, afin d'orienter l'exécution du plan. Mais il est en revanche plus robuste, c'est-à-dire qu'il permettra de gérer un plus grand nombre de situations.

D'autres possibilités existent entre ces deux extrêmes, comme par exemple prendre en compte la connaissance imparfaite de l'univers, tout en considérant que les actions ont des effets parfaitement connus. La difficulté de la planification augmente évidemment en fonction du degré de prise en compte de la réalité. Il faudra trouver un compromis acceptable entre la puissance de représentation et la rapidité de la planification : le second extrême, qui consiste à fournir un plan capable de gérer toutes les situations possibles et imaginables, va demander un travail considérablement plus important de la part du module de planification.

Nous avons parlé de la nature essentiellement symbolique de l'information traitée, du fait du haut niveau d'abstraction dans lequel on se situe ; mais certaines propriétés physiques peuvent quand même avoir de l'importance, comme le temps : le robot préleveur d'échantillons peut se voir assigner une durée maximale pour effectuer tous ses relevés. De même, il peut se voir assigner une quantité limitée de carburant. Une représentation uniquement symbolique de l'information ne paraît donc pas tout à fait adaptée à la gestion du temps et des ressources.

Mais même une représentation symbolique peut présenter plusieurs niveaux de complexité. On peut en effet vouloir représenter un effet de l'action "se déplacer du point A vers le point B" qui serait : toutes les éprouvettes qui se trouvent au point A et qui sont portées par le robot se trouvent au point B lorsque le robot s'y est rendu. Comme on ne sait pas a priori combien le robot transporte d'éprouvettes et quelles sont ces éprouvettes, on ne peut pas coder "en dur" dans l'action de se déplacer le changement de lieu des éprouvettes ; il faut que cette action possède une forme suffisamment générique pour pouvoir être utilisée dans tous les cas.

2. Cadre de notre étude de la planification

2.1. Les hypothèses simplificatrices

Ce préambule nous amène à formuler les hypothèses de travail sur lesquelles est basée la suite de cette thèse. Nous venons en effet de voir que plusieurs voies sont possibles, concernant la complexité de la prise en compte de l'univers et la richesse du langage de représentation utilisé. Afin de mieux comprendre les options choisies, décrivons un domaine classique en planification de tâches, le Monde des cubes. Il s'agit d'un domaine artificiel très simple à modéliser, mais n'offrant pas vraiment d'applications directes. Néanmoins, dans sa simplicité même, il a beaucoup à nous apprendre.

Il existe deux types d'objets dans le Monde des cubes : une table, et des cubes tous différents identifiés par une lettre de l'alphabet. Dans sa version la plus classique, cet univers possède les propriétés suivantes :

- la table peut supporter un nombre quelconque de cubes,
- un cube est posé soit sur la table, soit sur un autre cube,
- un cube ne peut avoir qu'un seul cube posé directement au-dessus de lui,
- la hauteur d'une pile de cubes n'est limitée que par le nombre de cubes présents dans l'univers.

Il n'y a qu'un seul agent dans cet univers, et la seule action possible qu'il peut effectuer est (en un seul mouvement) de prendre un cube non recouvert par un autre (situé au sommet d'une pile) et de le poser ailleurs, c'est-à-dire soit sur une autre pile, soit sur la table. Cet univers est parfaitement connu, et aucun événement impromptu ne peut survenir. Il s'agit d'un univers très simple en comparaison avec celui dans lequel évoluerait un robot préleveur d'échantillons.

Un petit problème que l'on peut se poser dans cet univers (cf. Figure 1) consiste par exemple à considérer au départ trois cubes A, B et C posés sur la table, l'objectif étant de constituer un empilement de ces trois cubes.

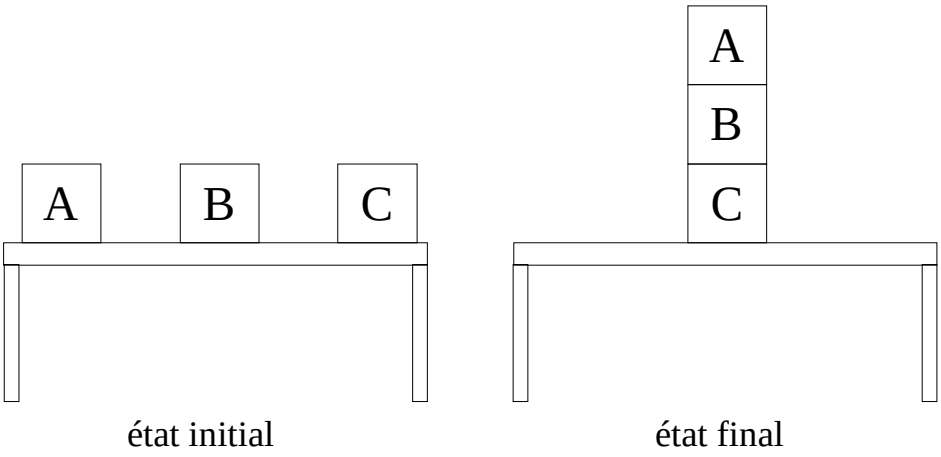

Figure 1 : Un problème du Monde des cubes

Comment est-on passé de l'état initial à l'état final, sachant que la seule action possible consiste à déplacer un cube d'une pile (ou de la table) vers une autre pile (ou sur la table) ? Étant donné que l'on est dans un monde idéal dont on a une connaissance parfaite, qu'aucun événement imprévu ne peut survenir, et que l'on connaît exactement les effets des actions exécutables, il suffit dans un premier temps de déplacer le cube B de la table vers le cube C, et dans un second temps de déplacer le cube A de la table vers le cube B. L'enchaînement de ces deux actions est un plan-solution, dans la mesure où l'exécution séquentielle des deux actions permet de faire évoluer l'univers de son état initial vers son état final. Le raisonnement d'un être humain pour résoudre ce problème pourrait être basé sur :

- Des capacités d'abstraction, permettant de généraliser une situation : comme tous les cubes sont posés sur la table, on peut déplacer n'importe lequel d'entre eux.
- Une analyse des lois de cet univers particulier : pour construire une pile de cubes, on doit procéder de bas en haut à partir de sa base.
- Une stratégie particulière de résolution : réduire la différence entre l'état initial et l'état final, en résolvant successivement chacun des sous-buts composant l'objectif global.

Réfléchissons à la manière dont une machine résout ce problème. De la façon dont on a spécifié le problème, elle ne peut pas se servir de connaissances sur l'univers du problème : tout ce qu'elle connaît est l'état initial, l'état but, les actions possibles et leurs effets. Si l'on applique la même stratégie qu'un être humain, un problème se pose : rien n'indiquant qu'une pile de cubes devant être construite à partir de sa base, il y a une chance sur deux pour que la machine tente de déplacer le cube A depuis la table vers le cube B avant d'avoir déplacé le cube B depuis la table vers le cube C. En faisant ceci, on s'éloigne de la solution puisqu'il faudra ensuite défaire ce qui vient d'être fait pour pouvoir recommencer autrement. La machine doit donc faire ce que l'humain a évité, c'est-à-dire des essais, menant peut-être à des situations d'échecs. Mais une machine possède une capacité de calcul très importante ; on peut donc s'appuyer sur cette faculté pour écrire des programmes de résolution de problèmes qui, s'ils ne sont pas encore aussi puissants que le raisonnement humain, offrent tout de même des possibilités intéressantes.

Tout ceci nous mène au point qui nous intéresse : justifier le choix de nos hypothèses de travail. Nous venons de voir que pour un domaine très simple, le Monde des cubes, trouver une méthode de résolution n'est pas évident ; surtout si on la veut suffisamment générique pour qu'elle puisse fonctionner dans d'autres domaines. La question qui se pose alors est la suivante : quelle est pour ce domaine l'efficacité des meilleures méthodes de résolution, qui n'utilisent pas de connaissances supplémentaires, et qui sont suffisamment génériques pour être utilisées avec succès dans d'autres domaines ? La réponse est plutôt décevante. En effet, jusqu'en 1995, les meilleurs programmes génériques de génération de plans connaissaient de grandes difficultés dès lors que l'univers contenait plus de 6 ou 7 cubes. Depuis 1995, de grands progrès ont été effectués, mais la plupart des méthodes dépassent difficilement une vingtaine de cubes. Et si l'univers était dynamique, les difficultés seraient encore plus grandes.

Nous choisirons donc de travailler dans le cadre le plus simple possible, qui présente déjà suffisamment de difficultés pour stimuler notre appétit de recherche... et pour lequel les méthodes de résolution sont loin, à l'heure actuelle, de répondre aux attentes de la majorité des utilisateurs potentiels. Notre travail sera donc fondé sur les hypothèses suivantes :

- *L'univers est statique* : aucun événement impromptu ne peut y survenir. Les seuls changements du monde sont ceux effectués par l'agent, qui évolue seul.
- *L'agent est omniscient* : il possède une connaissance parfaite de l'univers dans lequel il évolue et de la nature de ses propres actions.
- *Les actions ont des effets déterminés et connus* : l'effet de l'exécution d'une action de l'agent est fonction de cette action et de l'état du monde.
- *Les actions sont atomiques* : elles ont un effet immédiat et leur exécution n'est pas interruptible. Elles sont modélisées comme une transformation atomique d'un état du monde vers un autre.

2.2. La représentation des connaissances

Pour préciser le cadre de notre étude, nous devons maintenant montrer comment représenter les informations fournies à un programme de génération de plans (aussi appelé planificateur). A partir d'un état initial, d'un ensemble d'actions et d'un but, un planificateur doit fournir une collection organisée d'actions permettant par leur exécution de faire évoluer l'état initial de l'univers vers un état qui satisfait le but. Il doit aussi être suffisamment générique pour pouvoir résoudre des problèmes dans divers domaines ; il est donc nécessaire d'adopter une représentation adaptée des connaissances. Celle-ci déterminera l'étendue des problèmes que le planificateur sera sensé résoudre.

La simplicité du cadre dans lequel nous nous plaçons doit apparaître dans le formalisme de représentation que nous allons adopter. Plusieurs modèles de représentation ont été proposés dans les premiers temps de l'étude de la planification ; celui que nous retiendrons est le formalisme STRIPS [Fikes et Nilsson 1971], issu du planificateur du même nom qui a jeté les bases de la planification d'aujourd'hui. Il s'agit d'un formalisme basé sur une restriction de la logique du premier ordre.

Un état du monde étant représenté par un ensemble de fluents, l'idée essentielle sur laquelle est basé ce formalisme est la suivante : tout fluent qui n'est pas modifié par l'application d'une action est présent dans l'état du monde résultant de l'application de cette action. Ainsi, la représentation des effets d'une action de type STRIPS ne concernera que ce qui est modifié par l'application de cette action. Une description d'action de type STRIPS est constituée par un triplet d'ensembles de fluents, qui sont :

- Les *préconditions* : ce sont les fluents qui déterminent l'applicabilité de l'action. L'action est applicable sur un état du monde si et seulement si ses préconditions sont contenues dans cet état.
- Les *ajouts* : ce sont les fluents qui sont ajoutés à l'état du monde résultant de l'application de l'action sur un état donné.
- Les *retraits* : ce sont les fluents qui sont retirés de l'état du monde résultant de l'application de l'action sur un état donné.

La version que nous donnons ici du langage STRIPS est la plus simple, mais elle permet de représenter un grand nombre de problèmes. Elle peut être considérablement enrichie, par exemple en ajoutant :

- des négations dans les préconditions : un fluent nié en précondition signifie que ce fluent ne doit pas être présent dans un état du monde pour pouvoir appliquer l'action sur cet état,
- des préconditions disjonctives : l'action peut être appliquée à un état si au moins un des fluents de la disjonction est présent dans cet état,
- des effets conditionnels : lors de l'application de l'action, des effets supplémentaires sont effectués si une condition donnée est remplie par l'état avant l'application de l'action,
- des quantificateurs : ils permettent d'agir sur un nombre indéterminé de fluents possédant une propriété commune.

L'enrichissement du langage STRIPS de cette façon-là (en restant à un niveau symbolique) permet de représenter de façon plus concise et plus claire des problèmes de planification. Ceci a été fait, notamment dans le langage ADL [Pednault 1989] et plus tard dans le langage PDDL [McDermott 1998]. Mais la plupart des ajouts que l'on peut faire à STRIPS (en particulier ceux que nous venons de décrire) améliorent l'expressivité du langage, mais pas sa puissance de représentation ; en d'autres termes, on peut exprimer les mêmes domaines en STRIPS et en ADL, avec toutefois plus de clarté et de concision grâce à ce dernier. Mais une meilleure expressivité du langage de représentation augmente la difficulté de son utilisation : la condition d'applicabilité d'une action sur un état n'est plus un simple test d'inclusion de ses préconditions, mais

provient de l'évaluation d'une formule plus complexe pouvant contenir des quantificateurs, etc. Dans notre souci de simplicité de la représentation pour nous concentrer sur les aspects algorithmiques de la planification, nous emploierons dans cette thèse uniquement le langage STRIPS de base. L'extension à un langage tel que ADL ne semble pas présenter de difficultés insurmontables ; nous déciderons donc de le considérer comme un apport qui peut être fait ultérieurement sur des algorithmes intéressants fonctionnant avec le langage STRIPS.

Afin de simplifier l'écriture des domaines, qui peuvent contenir un très grand nombre d'actions, la description des actions est généralisée en utilisant des variables, afin de les rendre applicables à tous les états du monde ayant certaines caractéristiques communes. Un tel schéma d'action est appelé opérateur, et les actions lui correspondant seront créées par un mécanisme d'instanciation des variables (application de substitutions).

Pour illustrer de façon informelle le langage STRIPS, nous donnons maintenant la représentation du Monde des cubes et du problème que nous avons décrit précédemment. La représentation la plus courante utilise trois prédicats :

- *sur*(*x*, *y*) : le cube *x* est posé sur le cube *y*.
- *sur−table*(*x*) : le cube *x* est posé sur la table..
- *libre*(*x*) : le cube *x* n'est pas recouvert par un autre cube.

Dans l'état initial, les trois cubes A, B et C sont posés à même la table. Cet état est représenté par l'ensemble de fluents suivant :

$$E = \{sur{-}table(A), sur{-}table(B), sur{-}table(C), libre(A), libre(B), libre(C)\}$$

Dans l'état final du problème, le cube A est au-dessus du cube B, qui est au-dessus du cube C ; ce qui peut se traduire par l'ensemble de fluents suivant :

$$B = \{sur(A, B), sur(B, C)\}$$

Lors de la description du domaine, nous avons vu qu'une seule action était possible : celle consistant à déplacer un cube d'une pile de cubes (ou de la table) vers une autre pile de cubes (ou vers la table). Nous allons ici utiliser trois opérateurs : de la table vers une pile, d'une pile vers la table, et d'une pile vers une autre pile :

DéplacerDeTable(*x*, *y*) : *% déplace le cube x depuis la table vers le cube y %*
Préconditions = {*sur−table*(*x*), *libre*(*x*), *libre*(*y*), ($x \neq y$)}
Ajouts = {*sur*(*x*, *y*)}
Retraits = {*sur−table*(*x*), *libre*(*y*)}

DéplacerSurTable(*x*, *y*) : *% déplace le cube x depuis le cube y vers la table %*
Préconditions = {*sur*(*x*, *y*), libre(*x*)}
Ajouts = {*sur−table*(*x*), *libre*(*y*)}
Retraits = {*sur*(*x*, *y*)}

Déplacer(*x*, *y*, *z*) : *% déplace le cube x depuis le cube y vers le cube z %*
Préconditions = {*sur*(*x*, *y*), *libre*(*x*), *libre*(*z*), ($x \neq z$)}
Ajouts = {*sur*(*x*, *z*), *libre*(*y*)}
Retraits = {*sur*(*x*, *y*), *libre*(*z*)}

Remarquons la présence dans les préconditions de contraintes de différence (≠) entre les valeurs des variables. Elles sont nécessaires pour que les actions produites soient correctes : pour les opérateurs DéplacerDeTable et Déplacer, elles interdisent que l'on pose un cube sur lui-même. Il n'y a pas besoin d'autres contraintes, si l'on suppose que l'état initial est correctement formulé : par exemple pour l'opérateur DéplacerSurTable, les variables *x* et *y* ne peuvent pas être égales car un cube ne peut pas être posé sur lui-même.

Pour trouver les actions applicables sur un état *e* à partir d'un opérateur donné, on recherche s'il existe une substitution telle que son application aux préconditions de l'opérateur produise des fluents qui soient tous présents dans l'état *e*, et telle qu'elle satisfasse les contraintes entre les variables. Par exemple, si on considère

l'opérateur DéplacerDeTable et l'état initial *E* du problème, on trouve l'ensemble de substitutions suivant qui peuvent être appliquées à l'opérateur :

$$S = \{\{A/x, B/y\}, \{A/x, C/y\}, \{B/x, A/y\}, \{B/x, C/y\}, \{C/x, A/y\}, \{C/x, B/y\}\}$$

En effet, si on applique la substitution {A/*x*, B/*y*} aux préconditions de l'opérateur, on trouve l'ensemble de fluents suivant :

$$P = \{sur{-}table(A), libre(A), libre(B)\}$$

On a bien $P \subseteq E$, donc l'action DéplacerDeTable(A, B) peut être appliquée à *E*. L'état résultant *F* (représenté dans la Figure 2) de l'application de cette action sur l'état *E* est calculé en retirant de *E* les fluents obtenus en appliquant la substitution *P* aux retraits de l'opérateur DéplacerDeTable et en y ajoutant les fluents obtenus en appliquant la substitution *P* aux ajouts de l'opérateur DéplacerDeTable :

$$F = (E - \{sur{-}table(A), libre(B)\}) \cup \{sur(A, B)\}$$
$$= \{sur(A, B), sur{-}table(B), sur{-}table(C), libre(A), libre(C)\}$$

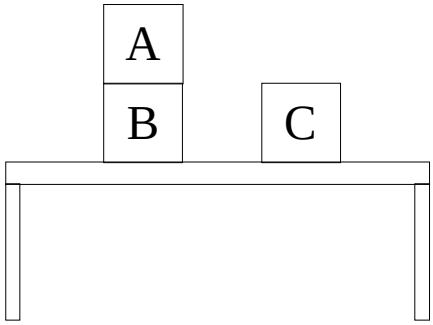

Figure 2 : État résultant de l'application de l'action DéplacerDeTable(A, B) sur l'état initial du problème

Le langage STRIPS est très intéressant dans la mesure où il est simple, permet de modéliser beaucoup de problèmes de planification et est bien adapté pour résoudre le problème du décor (cf. section II.1.1.2, page 11). C'est pourquoi tout notre travail concernera l'étude de méthodes de résolution de problèmes de planification décrits en STRIPS. Le domaine de recherche auquel nous avons consacré nos travaux, la génération de plans avec les hypothèses simplificatrices décrites dans cette section et utilisant le langage STRIPS pour représenter les actions, est appelé *planification classique*.

3. Objectif de notre travail

L'objectif de cette thèse est l'étude et l'amélioration de deux techniques de planification qui ont récemment révolutionné le domaine de recherche de la planification classique, ainsi que des liens existant entre ces méthodes. Même si ces approches ne sont plus à l'heure actuelle considérées comme étant les plus efficaces − et encore, il faudrait étudier de près les critères d'évaluation −, elles n'en gardent pas moins un intérêt certain lié aux idées nouvelles qu'elles ont apportées et à la somme considérable de travaux afférents. Ces deux méthodes n'ont a priori rien à voir ; c'est pourquoi nous ne les étudierons pas suivant leur ordre chronologique d'apparition dans la littérature.

3.1. Les courants de recherche

La seconde (par ordre chronologique) de ces techniques est apparue en 1995, implémentée dans le planificateur Graphplan [Blum et Furst 1995] [Blum et Furst 1997]. Considérée au départ comme une méthode entièrement nouvelle pour la résolution de problèmes de planification et présentée sous la forme d'une "recette de cuisine", il est apparu par la suite qu'elle est fortement liée à d'autres techniques. Elle peut en effet être considérée comme une transformation vers un problème de satisfaction de contraintes dynamique ("Dynamic Constraint Satisfaction Problem", DCSP [Mittal et Falkenhainer 1990]) [Kambhampati, Parker et Lambrecht 1997] [Kambhampati 1999], ou bien comme une implémentation particulière d'un algorithme de recherche avec extraction automatique d'heuristique [Bonet et Geffner 1999].

La première technique, apparue en 1992, est partiellement implémentée dans [Kautz et Selman 1992]. Elle consiste à transformer un problème de planification en un problème de satisfiabilité d'une base de clauses (le

problème SAT), afin de profiter des progrès considérables effectués dans ce domaine, concrétisés notamment par des implémentations remarquablement efficaces des meilleurs algorithmes comme les prouveurs SATZ [Li et Anbulagan 1997] et RELSAT [Bayardo et Schrag 1997]. Cette méthode est aussi motivée par l'emploi de prouveurs SAT utilisant des méthodes de recherche locale, qui se sont souvent avérés très efficaces, comme les prouveurs GSAT [Selman, Levesque et Mitchel 1992] et WALKSAT [Selman, Kautz et Cohen 1994]. La méthode de codage proposée en 1992 n'était pas automatique : il fallait trouver des règles de transformation de manière empirique pour chaque domaine de planification. Ce n'est qu'en 1996 que sont apparues des méthodes de transformation automatiques, l'une d'entre elles étant d'ailleurs inspirée du planificateur Graphplan (cf. [Kautz, McAllester et Selman 1996], [Ernst, Millstein et Weld 1997]).

On voit donc se dessiner un lien entre ces deux méthodes, lien qui a été approfondi dans un premier temps par une implémentation efficace du codage sous forme de base de clauses dans le planificateur BLACKBOX [Kautz et Selman 1999]. Ce dernier utilise une partie de Graphplan afin de créer la structure de données sur laquelle ce dernier travaille (le graphe de planification), puis convertit directement cette structure sous forme d'une base de clauses et enfin utilise un prouveur SAT pour trouver un plan-solution. On peut alors se poser la question suivante : est-il réellement nécessaire d'effectuer deux transformations du problème initial pour utiliser des techniques de satisfiabilité ? La réponse a été apportée récemment dans [Rintanen 1998] et le planificateur DPPlan [Baioletti, Marcugini et Milani 2000], qui présentent deux méthodes de résolution basées sur l'algorithme de Davis et Putnam fonctionnant sans passer par la transformation sous forme de base de clauses. La première utilise l'ensemble complet des actions de base que l'on peut créer en instanciant les opérateurs du domaine, alors que la seconde travaille directement à partir du graphe de planification. Ces méthodes permettent l'emploi de techniques utilisées dans les prouveurs SAT, en particulier la propagation unitaire, qui est connue comme l'amélioration principale de la procédure de Davis et Putnam.

3.2. Présentation des travaux effectués

Dans la partie II, nous présentons les principales méthodes de planification étudiées à l'heure actuelle. Nous montrons comment le domaine de la planification classique a évolué, pour finalement revenir aux premières idées de techniques de résolution de problèmes, basées sur la recherche heuristique. En effet, un des objectifs premiers de la planification était de tenter de simuler le raisonnement humain pour la résolution de problèmes, ce qui a conduit dès le départ à choisir une hypothèse simplificatrice très forte et pas réellement fondée : l'hypothèse de linéarité. Cette hypothèse est basée sur le fait que lorsqu'un être humain se retrouve face à un but constitué de plusieurs parties (des sous-buts), il va considérer soit que ces sous-buts sont tous indépendants les uns des autres, auquel cas il pourra les satisfaire les uns après les autres dans n'importe quel ordre, soit que ces sous-buts sont liés mais que pour résoudre le problème dans sa globalité il suffit de s'attaquer d'abord aux sous-problèmes les plus difficiles. Ce raisonnement est peut-être valable pour un être humain, car ce dernier possède des facultés supplémentaires qui peuvent lui permettre de se sortir de situations délicates (cf. l'exemple du Monde des cubes, section 2.1). Mais pour la machine, c'est une tout autre histoire... Cette hypothèse est beaucoup trop restrictive, et ne permet pas de résoudre des problèmes extrêmement simples. Mais elle a orienté la recherche en planification classique vers une voie particulière, la planification dans les espaces de plans, et a permis la découverte d'algorithmes intéressants. Parallèlement, la recherche heuristique a continué d'évoluer et est à nouveau utilisée en planification.

Dans la partie III, nous présentons en détail Graphplan et l'amélioration que nous lui avons apportée. Nous illustrons d'abord son fonctionnement, au travers du développement complet de la résolution d'un problème de planification simple. Nous présentons ensuite une formalisation de Graphplan et les résultats théoriques qui nous ont conduit à remettre au goût du jour une des idées de base de la planification dans les espaces de plans, l'idée de stratégie du moindre engagement. En effet, une des raisons qui a fait le succès de la planification dans les espaces de plans est le fait que le choix de l'ordonnancement des actions est retardé le plus possible (suivant les stratégies adoptées), ce qui a pour effet de réduire la taille de l'espace de recherche. En un sens, Graphplan possède déjà des capacités de moindre engagement ; il fournit même le plan le plus court en nombre d'ensembles d'actions exécutables en parallèle. Mais nous avons montré que l'on peut effectuer certains choix d'ordonnancement d'actions après avoir trouvé un plan-solution, ce qui apporte dans certains domaines une amélioration significative des performances due à une forte réduction de la taille de l'espace de recherche ; ceci au prix d'un calcul supplémentaire, qui fort heureusement s'avère être de complexité polynomiale. Graphplan est particulièrement bien adapté pour utiliser la stratégie de moindre engagement que nous proposons, mais nous montrons dans la suite que cette dernière peut être employée fructueusement avec une autre technique de planification.

Dans la partie IV, nous présentons le codage d'un problème de planification sous forme de base de clauses en logique propositionnelle. Nous passons en revue les principales techniques de codage proposées dans la littérature, en nous appuyant notamment sur une analyse critique d'un article qui a beaucoup apporté à la formalisation de ces codages [Mali et Kambhampati 1999]. Nous proposons des corrections ou des améliorations pour chacun des codages étudiés.

Dans la partie V, nous présentons la méthode DPPlan de [Baioletti, Marcugini et Milani 2000] qui permet de relier Graphplan à l'utilisation des techniques implémentées dans les prouveurs SAT. Nous analysons en détail l'article qui présente cet algorithme et montrons que ce dernier possède un problème de conception qui le rend inutilisable dans sa version originelle. Nous proposons alors une formalisation de cette technique qui permet de mettre à jour une version "minimale" et correcte de DPPlan ; cette version est minimale dans le sens où elle utilise le minimum de règles de propagation permettant de profiter des propriétés intéressantes de DPPlan. Nous montrons ensuite comment étendre cette version minimale avec les règles de propagation proposées dans la version originelle de DPPlan, et nous montrons enfin que l'on peut tirer partie de notre stratégie de moindre engagement pour améliorer les performances de cet algorithme.

II. Un aperçu général de la planification classique

1. Les premières approches

1.1. Les origines

On considère généralement que la planification trouve ses origines avec les systèmes GPS [Newell et Simon 1963] [Newell et Simon 1972] et QA3 [Green 1969]. La différence entre ces deux approches est que la première se concentre essentiellement sur une technique particulière de résolution de problèmes, sans souci véritable de représentation de la connaissance, alors que la seconde est basée sur l'expression d'un problème de planification dans un formalisme logique, dans le but d'utiliser un démonstrateur de théorèmes générique pour cette logique. Nous verrons dans les sections suivantes que ces deux paradigmes de la recherche en planification sont encore tout à fait d'actualité.

1.1.1. GPS

Le GPS ("General Problem Solver") [Newell et Simon 1963] a été créé dans le but de simuler un raisonnement humain particulier pour la résolution de problèmes généraux. Il manipule des *objets*, qui peuvent être transformés en d'autres objets à l'aide d'*opérateurs*. Ces objets correspondent plus ou moins aux états pour la planification, et il y a en principe un objet initial et un objet but. Les opérateurs qui manipulent ces objets n'ont pas de structure explicite comme les opérateurs de type STRIPS, leur permettant de transformer directement un objet en un autre objet ; ils fonctionnent en réduisant des *différences* entre les objets. Il faut donc lister de façon extensive, pour chaque différence possible entre deux objets, les opérateurs qui permettent de la réduire ; ce qui peut s'avérer très fastidieux pour l'utilisateur et est une méthode de représentation complètement dépendante du domaine d'utilisation et du problème à traiter.

La méthode de résolution de GPS est l'analyse des fins et des moyens ("means-end analysis"). En bref, cette technique consiste à réduire les différences entre deux objets en résolvant des sous-problèmes. Au départ, le problème est constitué d'un objet initial *I* et d'un objet but *G*. On choisit une différence entre *I* et *G*, considérée par une heuristique comme étant la plus difficile à réduire. On sélectionne ensuite un opérateur *o* qui permet de réduire cette différence, et qui transforme un objet *A* en un objet *B*. Le problème initial est alors décomposé en deux sous-problèmes : la réduction des différences entre l'objet initial *I* et l'objet *A*, et la réduction des différences entre l'objet *B* et l'objet but *G*. On rappelle alors récursivement l'algorithme de résolution pour le premier sous-problème, jusqu'à trouver une séquence d'opérateurs qui transforme l'objet *I* en l'objet *A*. En cas de succès, on fait de même pour le second sous-problème. A chaque fois que la résolution d'un sous-problème mène à un échec, c'est-à-dire qu'on ne trouve pas d'opérateur réduisant une différence entre deux objets, l'algorithme effectue un retour arrière.

Cette technique de résolution de problèmes est encore, à l'heure actuelle, utilisée avec succès ; le principal problème de GPS (la représentation des connaissances) étant résolu par l'utilisation des opérateurs de type

STRIPS. On la trouve par exemple dans les planificateurs PRODIGY [Fink et Veloso 1994], UNPOP [McDermott 1996] [McDermott 1999], et System R [Lin 2001], un planificateur qui a concouru avec un certain succès lors de la dernière compétition des planificateurs de la conférence AIPS−2000.

1.1.2. QA3 et le calcul des situations

Le système QA3 [Green 1969] permet de représenter la connaissance grâce à une logique du premier ordre, afin d'utiliser un démonstrateur automatique de théorèmes de cette logique pour trouver un plan-solution. Le problème de planification est représenté sous la forme d'un théorie logique. Le codage proposé est basé sur la notion d'état, ou situation : les actions et les états du monde sont des termes du langage. La notion de *fluent* est introduite afin de représenter le changement d'une propriété à travers le temps : pour une situation donnée, l'ensemble des valeurs des fluents détermine la représentation du monde. Par exemple pour le problème du Monde des cubes que nous avons décrit dans la partie d'introduction (cf. section .2.1, page 5), la propriété suivante est vraie dans la situation initiale, en utilisant le prédicat *sur−table*(*x*, *s*) qui signifie que le cube *x* est posé sur la table dans la situation *s* :

$$sur-table(\mathrm{A}, s) \wedge sur-table(\mathrm{B}, s) \wedge sur-table(\mathrm{C}, s)$$

Les effets de l'action consistant à déplacer un cube depuis la table sur un autre cube à partir de la situation *s* seront représentés de la façon suivante, en utilisant les prédicats *libre*(*x*, *s*) qui signifie qu'aucun cube ne se trouve au-dessus du cube *x* dans la situation *s*, et *sur*(*x*, *y*, *s*) qui signifie que le cube *x* est au-dessus du cube *y* dans la situation *s* :

$$\forall s \forall x \forall y \left(sur-table(x, s) \wedge libre(x, s) \wedge libre(y, s) \rightarrow sur\left(x, y, \mathrm{DéplacerDeTable}(x, y, s)\right)\right)$$

Le terme fonctionnel DéplacerDeTable(*x*, *y*, *s*) représente la situation qui résulte de l'exécution de l'action DéplacerDeTable dans la situation *s*. Ainsi, chaque action sera modélisée dans le langage par une fonction. Le but du problème sera exprimé par la formule *B* suivante, où s_b représente l'état but :

$$B = \exists s_b \left(sur(\mathrm{A}, \mathrm{B}, s_b) \wedge sur(\mathrm{B}, \mathrm{C}, s_b) \wedge sur-table(\mathrm{C}, s_b)\right)$$

Le problème de planification ainsi représenté est donné en entrée à un démonstrateur de théorèmes, qui devra prouver *B*. En cas de succès, il retournera la substitution $\{s / s_b\}$ qui fournira un plan-solution. L'intérêt principal de cette méthode est le codage explicite des préconditions et des effets des opérateurs, ce qui constitue un progrès considérable par rapport à GPS : le codage des opérateurs est tout à fait indépendant du problème que l'on veut résoudre et limite l'introduction d'informations dépendantes du domaine.

L'un des problèmes posés par ces représentations logiques est celui qui a été appelé problème du décor ("frame-problem"), et qui concerne la persistance des informations. En effet, pour qu'un fluent soit conservé d'une situation à l'autre, il faut que d'une manière ou d'une autre ce soit précisé par les axiomes de la représentation du problème. Il peut y avoir deux raisons pour qu'un fluent soit présent dans une situation : soit parce qu'il vient d'être ajouté par une action, auquel cas ceci est géré par le mécanisme d'application des actions, soit parce qu'il était présent dans la situation précédente et qu'il n'est retiré par aucune action. Mais ce dernier cas n'est pas géré directement par cette représentation : il faut rajouter des axiomes du décor, qui spécifient qu'un fluent présent dans une situation se retrouve dans la situation suivante s'il n'est retiré par aucune action. Par exemple, si dans une situation donnée on déplace un cube depuis la table vers un autre cube, alors tous les cubes posés sur d'autres cubes y resteront, ce qui se traduit par l'axiome suivant :

$$\forall s \forall x \forall y \forall z \forall t \left(sur(x, y, s) \rightarrow sur\left(x, y, \mathrm{DéplacerDeTable}(z, t, s)\right)\right)$$

Une variation du formalisme de QA3 est le calcul des situations [McCarthy et Hayes 1969], dont l'avantage principal est qu'il permet d'utiliser des symboles fonctionnels pour représenter les fluents. Le fait qu'un fluent *f* est vrai dans une situation *s* sera indiqué par le prédicat *holds*(*f*, *s*). Ainsi, on pourra écrire des axiomes plus généraux, les fluents pouvant être représentés par des variables. De plus, les termes fonctionnels représentant les actions sont eux aussi séparés de la situation dans laquelle ces actions sont appliquées, en utilisant le symbole fonctionnel *result* ; *result*(*a*, *s*) dénotant la situation résultante de l'application d'une action *a* sur une situation *s*. L'axiome concernant l'effet de l'action DéplacerDeTable décrit plus haut est alors codé de la façon suivante :

$$\forall s \forall x \forall y \left(\begin{array}{c} holds(sur-table(x), s) \wedge holds(libre(x), s) \wedge holds(libre(y), s) \rightarrow \\ holds\left(sur(x, y), result\left(\mathrm{DéplacerDeTable}(x, y), s\right)\right) \end{array} \right)$$

On peut ainsi définir des axiomes du décor plus généraux. Par exemple, l'axiome suivant décrit de façon plus complète que dans la formulation de Green ce qui ne change pas quand on applique l'action DéplacerDeTable (en utilisant le prédicat *diff*(*x*, *y*) qui signifie que *x* doit être différent de *y*) :

$$\forall s \forall f \forall x \forall y \left(\begin{array}{c} holds(f, s) \wedge diff(f, sur-table(x)) \wedge diff(f, libre(y)) \rightarrow \\ holds\left(f, result\left(\text{DéplacerDeTable}(x, y), s\right)\right) \end{array} \right)$$

Mais un des problèmes de ces représentations basées sur la logique du premier ordre est l'explosion combinatoire provoquée par l'ajout indispensable de très nombreux axiomes du décor qui pénalisent lourdement les performances du démonstrateur de théorèmes. C'est une des raisons pour lesquelles ce formalisme a été très vite abandonné, au profit du formalisme STRIPS.

1.2. La planification d'ordre total

1.2.1. STRIPS...

L'histoire moderne de la planification commence avec le planificateur STRIPS [Fikes et Nilsson 1971]. Son principal apport, qui a conditionné la majeure partie de la recherche en planification jusqu'à maintenant, et la conditionnera probablement encore longtemps, est le langage de représentation des connaissances qu'il a introduit (description des opérateurs, des états...) : le langage STRIPS, que nous avons décrit dans la partie d'introduction (cf. section .2.2, page 6). Le langage originellement utilisé par STRIPS était un peu plus élaboré, notamment en ce qui concerne les préconditions des opérateurs (possibilités de formules plus complexes).

L'intérêt principal de ce type de représentation par rapport à celui employé dans GPS est le même que pour le calcul des situations : les préconditions et les effets des opérateurs sont définis précisément, de façon indépendante des états et donc des problèmes. Ainsi, pour tout état, on peut savoir si un opérateur est applicable et quel sera l'état résultant de son application. Les deux principaux avantages du langage STRIPS par rapport au calcul des situations sont, d'une part de permettre un calcul simple des états (sans avoir recours à un démonstrateur de théorèmes), et d'autre part de résoudre le problème du décor : tout fluent appartenant à un état sur lequel on applique un opérateur et qui n'est pas retiré par ce dernier appartient à l'état résultant (propriété de persistance).

L'inconvénient du langage STRIPS par rapport au calcul des situations est sa relative pauvreté d'expression. En effet, le calcul des situations bénéficie de toute la puissance de la logique du premier ordre pour représenter les actions, mais aussi les propriétés du domaine (par exemple, une pile de quatre cubes s'écroule, laissant les quatre cubes sur la table). Il permet aussi d'exprimer facilement des contraintes dépendantes du domaine, afin d'aider à la recherche d'une solution ; par exemple, interdire qu'une action fasse exactement l'opération inverse d'une action qui vient juste d'être effectuée :

$$\forall s \forall f \forall x \forall y \neg \left(holds\left(f, result\left(\text{DéplacerSurTable}(x, y), result\left(\text{DéplacerDeTable}(x, y), s\right)\right)\right)\right)$$

Pour pallier ce problème, de nombreuses extensions au langage STRIPS ont été proposées, parmi lesquelles ADL [Pednault 1989] et PDDL [McDermott 1998] dont nous avons déjà parlé pour les effets conditionnels, la quantification ..., mais aussi des extensions permettant la gestion du temps, la représentation de connaissances du domaine ou encore la gestion de ressources, dont nous reparlerons dans les sections suivantes. En contrepartie, la simplicité du langage STRIPS et de son utilisation permettent a priori de construire des planificateurs plus efficaces que ceux basés sur le calcul des situations.

Le planificateur STRIPS effectue une recherche à la GPS, basée sur l'analyse des fins et des moyens. Mais il pose un problème essentiel, qui résulte d'un choix qui a été fait pour des raisons d'efficacité : l'hypothèse de linéarité. Cette hypothèse spécifie que les sous-buts primitifs d'un problème (les sous-problèmes issus du but initial) sont indépendants, c'est-à-dire qu'ils peuvent être résolus l'un après l'autre, dans n'importe quel ordre, sans nécessiter de retour arrière sur ce choix. Or, on peut exprimer en langage STRIPS des problèmes très simples, qui ne peuvent pas être résolus si on se conforme à cette hypothèse. Le problème du Monde des cubes que nous avons décrit dans la partie d'introduction (cf. section .2.2, page 5) ne peut pas être résolu par STRIPS, si l'on considère que le premier sous-but est *sur*(A, B). En effet, une fois qu'il est résolu par l'application de l'action DéplacerDeTable(A, B), on ne peut pas résoudre le second sous-but *sur*(B, C), puisque *B* n'est plus libre pour le déplacer sur C. Dans ce cas, on dit que les deux sous-buts interagissent l'un avec l'autre.

1.2.2. ... et les autres

Les approches qui vont suivre en planification d'ordre total vont tenter avec plus ou moins de bonheur de résoudre ce problème d'interaction des sous-buts.

Dans le planificateur HACKER [Sussman 1974], apparaît une méthode de protection des intervalles : un sous-but est protégé depuis l'endroit où il a été établi (par une action ou par l'état initial) jusqu'à l'endroit où il est utilisé (comme précondition d'une action ou comme but du problème). Dans le cas où une action tente de retirer un sous-but protégé, la planification est relancée en choisissant un ordre différent pour les sous-buts initiaux. Mais comme HACKER utilise toujours l'hypothèse de linéarité pour la résolution des sous-problèmes primitifs issus de la décomposition du but initial, certains problèmes très simples ne peuvent pas être résolus ; le plus célèbre d'entre eux est un problème du Monde des cubes appelé par la suite l'anomalie de Sussman (cf. Figure 3).

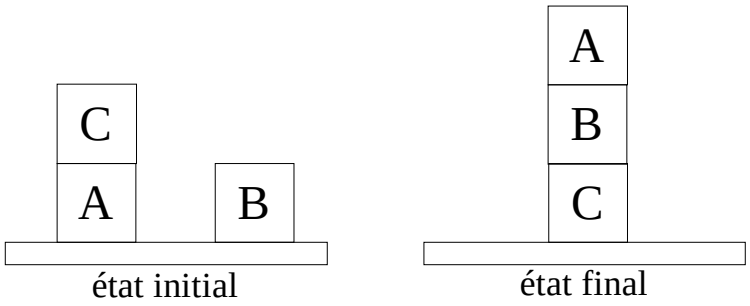

Figure 3 : L'anomalie de Sussman

Les deux sous-buts de ce problème, *sur*(A, B) et *sur*(B, C) ne sont pas indépendants. En effet, si l'on établit d'abord *sur*(B, C) en déplaçant le cube C depuis la table vers le cube B, on ne peut pas résoudre *sur*(A, B) sans défaire *sur*(B, C). De même, si l'on établit *sur*(A, B) en déplaçant le cube C sur la table puis le cube A depuis la table vers le cube B, on ne peut plus résoudre *sur*(B, C) sans défaire *sur*(A, B). HACKER ne peut donc pas résoudre ce problème. Le planificateur INTERPLAN [Tate 1975] complète le traitement de HACKER par la promotion du sous-but dont la résolution viole la protection d'intervalle (c'est-à-dire en le traitant avant le sous-but concerné par la protection), mais s'avère toujours incapable de résoudre l'anomalie de Sussman. Les systèmes qui suivent vont tenter d'apporter une solution aux problèmes de ce type, comme WARPLAN [Warren 1974] qui introduit la technique de régression des actions : celle qui viole la protection d'intervalle est insérée plus avant dans le plan, jusqu'à une position qui ne viole aucune contrainte. On peut citer aussi le système de [Waldinger 1977], qui permet aussi de faire régresser les sous-buts.

1.3. La planification d'ordre partiel

Il semble inévitable que pour être capable de résoudre au moins des problèmes simple comme l'anomalie de Sussman, un planificateur d'ordre total doive effectuer des retours arrières ; soit sur le choix des actions, soit sur l'ordonnancement de ces actions. Une technique nouvelle a alors été proposée pour tenter de résoudre ce problème d'ordonnancement des actions : retarder son apparition le plus longtemps possible. Il s'agit d'une technique dite de moindre engagement ("least commitment"), qui a donné naissance à la planification dans les espaces de plans partiels.

1.3.1. NOAH...

Les stratégies de moindre engagement ont été utilisées pour la première fois dans le planificateur NOAH [Sacerdoti 1975]. Ce dernier travaille sur des plans partiellement ordonnés (appelés réseaux procéduraux dans la terminologie de NOAH) essentiellement formés de noeuds d'actions et de noeuds de buts.

Une première stratégie de moindre engagement dans NOAH concerne l'ordonnancement des actions. Au départ, le but du problème est introduit dans le réseau. Puis, un noeud de but est créé pour chaque sous-but du but initial. Chacun de ces noeuds est ensuite remplacé par le noeud d'une action qui produit le sous-but correspondant, auquel on relie un nouveau noeud de but pour chaque précondition de l'action. Un module spécial est chargé de détecter les interactions et de régler les conflits grâce à des procédures spécifiques, par exemple en ordonnant des actions ou bien en introduisant de nouvelles actions qui résolvent les interactions. Ainsi, les actions ne sont ordonnées que lorsque c'est nécessaire. Ce processus est répété jusqu'à ce qu'il n'y

ait plus de noeud de but dans le réseau, en utilisant si possible des actions déjà présentes dans le réseau. Si le plan résout les buts du problème et qu'il n'y a pas de conflit entre les actions, alors un plan-solution est trouvé. Le problème est que NOAH n'effectue jamais de retour arrière : chaque décision est irrévocable. Il n'est donc pas plus complet que STRIPS ou HACKER, même s'il est capable de résoudre plus de problèmes.

Une autre stratégie de moindre engagement utilisée dans NOAH et que l'on retrouvera dans d'autre approches est l'instanciation tardive des variables des actions. En effet, comme la recherche est effectuée à partir du but, la valeur de certaines variables des actions employées peut rester indéfinie : une action étant utilisée pour remplacer un sous-but qu'elle produit, la valeur de certaines variables présentes dans ses autres effets et dans ses préconditions peut ainsi rester indéterminée jusqu'à ce qu'on ait effectivement besoin de les instancier.

Une des qualités essentielles de NOAH est la séparation explicite qu'il fait entre le traitement des interactions entre les actions et les autres aspects de la planification, comme l'insertion d'actions pour résoudre un sous-but. Ces idées vont conduire ultérieurement à une véritable formalisation de la planification.

1.3.2. ... et les autres

NOAH a inspiré toute une génération de planificateurs, jusque dans les années 90. On peut citer par exemple NONLIN [Tate 1977], qui conserve tous les choix effectués durant la recherche afin de faire des retours arrières, ce qui lui permet d'être complet ; ou encore SIPE [Wilkins 1988], qui permet aussi d'exprimer des axiomes du domaine pour représenter par exemple les effets de bord des actions sur le monde. SIPE gère aussi les ressources nécessaires à l'exécution des actions. D'autres systèmes permettent (entre autres) la représentation du temps pour prendre en compte la durée d'exécution des actions, et permettre ainsi la minimisation du temps d'exécution global d'un plan ; on peut citer par exemple les planificateurs DEVISER [Vere 1983] et FORBIN [Miller, Firby et Dean 1985].

2. La planification comme recherche dans les espaces de plans

Aucun des planificateurs mentionnés jusqu'à présent n'est basé sur une véritable théorie de la planification. Les progrès qui ont été effectués ont souvent été apportés pour résoudre de manière ad hoc tel ou tel problème (problème du décor, interaction entre sous-buts...). Un profond changement s'est produit dans la communauté avec l'apparition du planificateur TWEAK [Chapman 1987], qui est le premier planificateur basé sur une formalisation de la planification.

2.1. TWEAK...

Le coeur de TWEAK est le *critère de vérité* ("Modal Truth Criterion", MTC) qui permet de déterminer la valeur de vérité de tout fluent appartenant aux préconditions des actions d'un plan partiel non-linéaire (un plan non-linéaire étant un plan partiellement ordonné). Ce critère de vérité décrit toutes les manières possibles de corriger un plan partiel en résolvant les interactions entre actions, de façon à ce que ce plan devienne un plan-solution. Les techniques permettant de compléter un plan partiel sont celles des planificateurs d'ordre partiel précédents : insertion d'actions dans le plan, de contraintes d'ordre entre les actions, de contraintes entre les variables d'actions partiellement instanciées... Le critère de vérité de TWEAK peut être interprété comme une procédure non-déterministe qui permet de trouver un plan-solution en utilisant une technique de recherche, par exemple la recherche en profondeur d'abord. Cette recherche est donc effectuée dans un espace de plans partiels : chaque noeud de l'arbre de recherche représente un plan non-linéaire en voie d'élaboration. Un plan-solution est obtenu lorsqu'un noeud contient un plan dont toutes les actions ont leurs préconditions produites par d'autres actions du plan (l'état initial étant considéré comme une action sans préconditions produisant les fluents de l'état initial du problème, et le but comme une action sans effets dont les préconditions sont les fluents du but du problème). Grâce au critère de vérité, TWEAK est sain : toutes les linéarisations d'un plan-solution non-linéaire qui respectent les contraintes d'ordre entre les actions sont des plans-solutions du problème.

Le formalisme de description des problèmes choisi par Chapman est très simple : les actions sont représentées par des opérateurs STRIPS formés d'un ensemble de littéraux en précondition et d'un ensemble de littéraux en post-condition. Les littéraux positifs en post-condition correspondent aux ajouts et les littéraux négatifs correspondent aux retraits. Ces restrictions permettent à Chapman de démontrer que son planificateur est sain et complet, ainsi que plusieurs autres résultats théoriques ; notamment que la planification classique

avec le langage de description d'actions de TWEAK est semi-décidable si l'état initial est infini (en montrant que TWEAK peut simuler une machine de Turing), et qu'elle est indécidable si les effets des opérateurs sont des fonctions de la situation d'entrée, même avec une situation initiale finie.

2.2. ... et les autres

Une grande partie des travaux importants qui vont suivre en planification classique sont basés sur TWEAK. On peut citer notamment :

- Le planificateur SNLP [McAllester et Rosenblitt 1991] ("Systematic Non Linear Planning"), qui est une version systématique de TWEAK, c'est-à-dire qu'il garantit que deux plans partiels quelconques dans les feuilles de l'arbre de recherche conduisent à des plans-solutions dont les linéarisations sont toutes différentes. Des travaux ultérieurs [Knoblock et Yang 1995] montrent que la systématicité ne permet pas forcément à un planificateur non-linéaire d'être plus performant.
- Les travaux de [Jacopin 1993] pour le planificateur PWEAK montrent l'inutilité dans TWEAK d'une des stratégies d'insertion des actions dans un plan partiel pour régler un conflit entre deux actions, le *chevalier blanc* ("White Knight"). Lorsqu'une précondition est retirée par une action s'insérant entre l'action qui établit la précondition et celle qui l'utilise, cette stratégie force l'insertion d'une action qui rétablit la précondition retirée.
- Le planificateur UCPOP [Penberthy et Weld 1992] [Weld 1994], qui utilise comme langage de description des domaines et des problèmes un sous-ensemble important du langage ADL [Pednault 1989]. Le planificateur ainsi obtenu, basé sur SNLP, est sain et complet.
- Les planificateurs UA et TO de [Minton, Bresina et Drummond 1994], ainsi que POCL, TOCL et TOPI de [Barret et Weld 1994]. Ce sont tous des planificateurs effectuant une recherche dans les espaces de plans ; ils diffèrent par leur technique de construction des plans partiels : UA et POCL sont basés sur SNLP et peuvent donc insérer une action n'importe où dans un plan partiellement ordonné, TO et TOCL peuvent insérer une action n'importe où dans un plan totalement ordonné, et TOPI ne peut insérer une action qu'en tête du plan partiel. Ces travaux montrent la supériorité de la planification partiellement ordonnée sur la planification totalement ordonnée, le planificateur TOPI étant même assimilé à un planificateur fonctionnant en chaînage arrière dans les espaces d'états. Mais ces travaux sont relativisés par [Veloso et Blythe 1994], qui montrent que leur planificateur PRODIGY, qui utilise une variante de la technique d'analyse des fins et des moyens, surpasse SNLP (et donc POCL et UA) dans certains domaines particuliers. Le planificateur UNPOP [McDermott 1996] [McDermott 1999] qui utilise aussi une stratégie d'analyse des fins et des moyens guidée par un graphe construit à partir des buts montre aussi de bonnes performances. Nos travaux [Vidal et Régnier 1999] montrent aussi qu'un planificateur fonctionnant en chaînage avant dans les espaces d'états a de meilleures performances que UCPOP dans de nombreux domaines, dès lors que l'on effectue un test de bouclage pour détecter les états déjà rencontrés. Ces résultats sont confirmés par les performances de plusieurs planificateurs récents, qui sont basés sur la planification dans les espaces d'états avec extraction automatique d'heuristique [Bonet, Loerincs et Geffner 1997] [Bonet et Geffner 1998] [Bonet et Geffner 1999].
- Les planificateurs qui gèrent le temps et les ressources, notamment ZENO [Penberthy et Weld 1994], IxTeT ("IndeXed TimE Table") [Laruelle 1994] [Laborie 1995] qui est basé sur le formalisme de logique temporelle de [Ghallab, Alami et Chatila 1987] [Ghallab et Mounir-Alaoui 1989], ou encore TRIPTIC [Rutten et Hertzberg 1993].
- Plusieurs travaux concernant la complexité de la planification classique. [Erol, Nau et Subrahmanian 1992] prouvent que la planification de type TWEAK est semi-décidable si le langage de description d'actions utilise des symboles de fonction ou s'il possède un ensemble infini de symboles de constantes, et qu'elle est décidable si le langage ne contient pas de symbole de fonctions et si l'ensemble des symboles de constantes est fini. [Bylander 1991] montre que le problème de l'existence d'un plan-solution pour la planification de type STRIPS (avec état initial et ensemble des symboles de constantes finis) est PSPACE-complet dans le cas général, et exhibe plusieurs classes particulières dont certaines sont NP−complètes ou même polynomiales.
- On peut enfin citer les travaux de [Bäckström et Klein 1991] [Bäckström 1992], qui proposent un formalisme de représentation des actions différent de STRIPS. Ce formalisme, baptisé SAS^+ ("Simplified Action Structures"), possède la même expressivité que le langage STRIPS classique. Au lieu d'utiliser des

atomes propositionnels pour représenter les faits de l'univers, il utilise des variables d'états multi-valuées. Par exemple pour un problème du Monde des cubes contenant 3 cubes A, B et C, une variable d'état pour chaque cube représente sa position : PositionA peut prendre la valeur B, C, ou Table. Les états et les préconditions et effets des actions sont représentés par des séquences de valeurs pour toutes les variables d'états (avec une valeur *indéfini* dans les actions, pour les variables d'état qui ne sont pas concernées). Ce formalisme a permis de déterminer plusieurs classes de problèmes polynomiaux (avec des algorithmes performants permettant de les résoudre) et NP-complets.

2.3. Un formalisme unificateur : la planification par raffinements

Avec le planificateur TWEAK sont apparues les premières études formelles de la planification, portant essentiellement sur la planification dans les espaces de plans partiels. Un formalisme unificateur de toutes les techniques de planification classique, qu'elles soient d'ordre total ou d'ordre partiel, a été introduit dans [Kambhampati 1994], [Kambhampati, Knoblock et Yang 1995], et [Kambhampati 1997]. Ce formalisme a conduit à l'élaboration du planificateur UCP ("Universal Classical Planner") [Kambhampati 1996] capable d'entrelacer les diverses stratégies de planification afin de tirer parti des avantages de chacune. Ce planificateur est sain, complet, et systématique suivant les stratégies de raffinement employées.

L'idée centrale du cadre unificateur de la planification par raffinements est la suivante : *toute stratégie de planification a pour but de réduire l'ensemble de toutes les séquences d'actions possibles sur un domaine à l'ensemble des séquences pouvant être considérées comme les solutions d'un problème de planification.*

Nous allons maintenant aborder les principaux aspects de ce formalisme, afin d'une part de définir les principes essentiels de la planification par recherche dans les espaces de plans, et d'autre part de fixer un langage qui nous sera utile par la suite.

2.3.1. Les plans partiels

Un planificateur effectue une recherche dans l'espace des *séquences d'actions* que l'on peut définir sur un domaine, l'objectif étant de trouver une *séquence solution* ; c'est-à-dire une séquence dont l'application sur l'état initial du problème conduit à un état qui satisfait le but de ce problème. Ces séquences d'actions sont représentées de façon compacte par des *plans partiels* : un plan partiel peut être vu de façon générale comme un ensemble de contraintes qui éliminent les séquences d'actions qui ne sont pas *candidates*, c'est-à-dire qui ne peuvent pas conduire à des séquences solutions. Un plan partiel est un plan-solution si et seulement si toutes les *linéarisations* (i.e. tris topologiques) des actions de ce plan partiel consistantes avec les contraintes, sont des séquences solutions. On sera intéressé par la recherche des séquences candidates *minimales*, c'est-à-dire qui contiennent le même nombre d'actions que le plan partiel.

Par exemple pour un domaine de planification donné, un plan vide ne contenant aucune contrainte représente toutes les séquences d'actions possibles sur ce domaine. Si l'on introduit une action initiale sans préconditions, dont les effets sont les fluents de l'état initial du problème, et qui est contrainte à être la première action de toutes les séquences, alors on élimine toutes les séquences d'actions qui ne peuvent pas être appliquées dans l'état initial. De même, si l'on introduit une action sans effets, dont les préconditions sont les buts du problème, et qui est contrainte à être la dernière action de toutes les séquences, alors on élimine toutes les séquences d'actions qui ne conduisent pas au but du problème. Ensuite, on utilise diverses stratégies de raffinement qui vont introduire des actions et/ou des contraintes dans les plans partiels afin de restreindre l'ensemble des séquences candidates représentées par ces plans partiels à des plans partiels représentant des ensembles de séquences solutions.

Un plan partiel est composé de trois parties :

1. *L'ensemble des étapes* : ce sont des références aux actions qui constituent un plan partiel. Chaque étape étant identifiée de façon unique, un plan partiel peut donc contenir plusieurs étapes qui font référence à une même action.
2. *L'ensemble des contraintes d'ordre* : ce sont des contraintes précisant l'ordonnancement de deux étapes distinctes. Elles peuvent être de deux types :
   - *contraintes de précédence* : l'étape e_1 *précède* l'étape e_2 signifie que e_1 doit être exécutée avant e_2, mais toute étape e_3 peut être insérée entre e_1 et e_2.
   - *contraintes de contiguïté* : l'étape e_1 est *contiguë* à l'étape e_2 signifie que e_1 doit être exécutée immédiatement avant e_2, interdisant l'insertion de toutes étape e_3 entre e_1 et e_2.

3. *L'ensemble des contraintes auxiliaires* : ce sont des conditions qui doivent être vérifiées par un plan partiel. Elles peuvent être de deux types :
   – *contraintes de préservation d'intervalles* : elles préservent la valeur de vérité d'un fluent sur un intervalle de précédence, c'est-à-dire que les étapes qui vont s'insérer entre les deux étapes de l'intervalle de précédence ne doivent pas avoir pour effet la négation de ce fluent. Ainsi, si une étape e_1 est insérée dans le plan pour établir une précondition p d'une étape e_2, alors on pourra définir une contrainte de précédence entre e_1 et e_2 et protéger la valeur du fluent p par une contrainte de préservation de cet intervalle.
   – *contraintes ponctuelles de valeur de vérité* : elles protègent la valeur de vérité d'un fluent à un endroit précis du plan partiel.

Pour décrire les stratégies de raffinement, nous devons définir la terminologie que nous allons utiliser pour décrire les différentes parties d'un plan partiel :

- La *tête du plan* : il s'agit de l'ensemble maximal d'étapes du début du plan liées deux à deux par des contraintes de contiguïté. La dernière étape de la tête du plan est l'*étape de tête*. Comme ces étapes forment une séquence d'actions applicable dans l'état initial du problème, on peut calculer l'état résultant de son application que l'on nommera l'*état de tête*.
- La *queue du plan* : c'est la partie symétrique de la tête du plan pour la fin du plan partiel. La première étape de la queue du plan est l'*étape de queue*, et l'*état de queue* est l'état partiel défini par la régression des étapes de la queue du plan sur les buts.
- Les *étapes centrales* : ce sont toutes les étapes n'appartenant ni à la tête ni à la queue du plan. La *frange de tête* est l'ensemble des étapes pouvant venir directement après l'étape de tête dans une linéarisation, et la *frange de queue* est l'ensemble des étapes pouvant précéder directement l'étape de queue dans une linéarisation.

2.3.2. Les stratégies de raffinement

Nous allons maintenant passer en revue les principales stratégies de raffinement. Les propriétés essentielles que peut posséder une stratégie de raffinement R s'appliquant sur un plan partiel P et retournant un ensemble de plans partiel $R(P) = \{P'_i\}$sont les suivantes :

- *Complétude* : l'ensemble des séquences solutions représentées par P sont toutes représentées par les plans partiels de $R(P)$. Aucune solution n'est perdue par l'application de R sur P.
- *Progressivité* : l'ensemble des séquences candidates représentées par les plans partiels de $R(P)$ est un sous-ensemble de l'ensemble des séquences candidates représentées par P. Ceci témoigne du fait que la stratégie de raffinement R possède une réelle faculté d'élimination de séquences non candidates. La progressivité est dite *forte* si la longueur des séquences candidates minimales représentées par les plans partiels de $R(P)$ augmente après l'application de R.
- *Systématicité* : après l'application de R, une séquence d'action ne peut être représentée que par un seul plan partiel de $R(P)$.

Raffinement en chaînage avant

Le principe de cette stratégie est d'augmenter progressivement la tête du plan soit par une action n'appartenant pas au plan partiel, soit par une étape de la frange de tête. Une telle action ou étape peut être utilisée lorsque ses préconditions sont vérifiées dans l'état de tête. Une contrainte de contiguïté est ajoutée entre l'étape de tête et la nouvelle étape, qui devient l'étape de tête.

Pour assurer la complétude de cette stratégie, le choix de l'étape doit être un point de retour arrière. Cette stratégie est aussi progressive, puisqu'elle élimine toutes les séquences d'actions qui n'ont pas un préfixe de n actions exécutable ; le préfixe d'une séquence étant défini par les n étapes de la tête du plan. Enfin, elle est systématique car tous les plans partiels retournés ont une tête différente, donc les séquences candidates qu'ils représentent ont un préfixe différent.

Si l'on n'utilise que cette stratégie pour planifier, on sait que l'on a trouvé un plan lorsque l'étape de tête est confondue avec l'étape de queue. Cette stratégie est intéressante pour l'utilisation d'heuristiques sur le domaine, qui s'expriment plus facilement en chaînage avant, et pour élaguer l'arbre de recherche grâce à la connaissance des états déjà rencontrés.

Raffinement en chaînage arrière

Le principe de cette stratégie est d'augmenter progressivement la queue du plan soit par une action n'appartenant pas au plan partiel, soit par une étape de la frange de queue. Une telle action ou étape peut être utilisée lorsqu'un de ses ajouts est présent dans l'état de queue, et lorsqu'aucun de ses retraits n'est inconsistant avec l'état de queue. Une contrainte de contiguïté est ajoutée entre la nouvelle étape et l'étape de tête. La nouvelle étape devient l'étape de tête.

Comme pour le raffinement en chaînage avant, cette stratégie est complète si le choix de l'étape est un point de retour arrière. Elle est aussi progressive et systématique. Son avantage par rapport au raffinement en chaînage avant est qu'elle utilise les connaissances sur le but pour décider quelles actions appliquer. Par contre, l'utilisation d'heuristiques sur le domaine est plus difficile.

Raffinement dans les espaces de plans

Les raffinements dans les espaces de plans sont aussi guidés par les buts, et contraignent moins les plans partiels que les deux stratégies de raffinement précédentes. Les plans partiels produits par cette stratégie représentent donc en général un plus grand nombre de séquences candidates, et donc de séquences solutions potentielles. Cette stratégie fonctionne en trois temps :

1. Sélection d'un sous-but, c'est-à-dire d'une étape *e* du plan partiel, ainsi que d'une précondition *p* de cette étape qui n'est supportée par aucune autre étape du plan. Il ne s'agit pas d'un point de retour arrière pour la complétude de la stratégie de raffinement.
2. Sélection d'une étape *f* (présente dans le plan ou nouvellement créée à partir d'une action) qui établit cette précondition, et création d'une contrainte de précédence entre *f* et *e*. Les étapes du plan partiel qui menacent cet intervalle, c'est-à-dire qui peuvent s'insérer entre *f* et *e* et qui ont des effets contradictoires avec *p*, doivent être contraintes à préserver *p*. Pour cela, on distingue trois techniques, *g* étant une étape qui menace l'intervalle :
   - *promotion* : l'exécution de *g* va précéder l'exécution de *f*, par l'ajout d'une contrainte de précédence entre *g* et *f*.
   - *démotion* : l'exécution de *e* va précéder l'exécution de *g*, par l'ajout d'une contrainte de précédence entre *e* et *g*.
   - *confrontation* : *g* reste exécutable entre *f* et *e*, mais on ajoute une contrainte ponctuelle de valeur de vérité devant *g* qui limite ses effets en supprimant des possibilités d'instanciation de ses variables, afin qu'aucun des effets ne menace l'intervalle une fois totalement instanciés. Cette technique est plus classiquement appelée *séparation* (dans TWEAK par exemple).

   Le choix de l'étape ainsi que celui d'une technique de protection sont des points de retour arrière pour la complétude de la stratégie de raffinement.
3. Protection de l'intervalle (optionnelle) en ajoutant deux contraintes de préservation d'intervalles entre *f* et *e* : la première pour interdire aux prochaines étapes s'insérant entre *f* et *e* de détruire *p*, et la seconde pour leur interdire de créer *p*.

La stratégie de raffinement dans les espaces de plans est complète si l'on respecte les points de retour arrière indiqués. Elle est aussi progressive, puisque l'ajout d'une nouvelle étape ou d'une contrainte de précédence limite l'ensemble des séquences candidates. Enfin, elle est systématique si l'on ajoute les deux contraintes de préservation d'intervalles de l'étape 3.

Raffinements auxiliaires

Les stratégies de raffinement que nous venons d'étudier sont progressives, puisqu'elles réduisent l'ensemble des séquences candidates des plans partiels sur lesquels elles sont appliquées. Mais beaucoup de planificateurs utilisent aussi des stratégies non progressives (n'ayant donc pas de pouvoir d'élimination) qui permettent de réduire les coûts futurs de gestion des plans partiels en créant plusieurs branches dans l'arbre de recherche. On peut en distinguer deux catégories (une troisième catégorie définie dans [Kambhampati 1997] concerne les planificateurs hiérarchiques comme NOAH [Sacerdoti 1975], SIPE [Wilkins 1988] ou IxTeT [Ghallab et Mounir-Alaoui 1989] [Laruelle 1994] [Laborie 1995] ; nous n'abordons pas ce thème dans ce mémoire).

Les raffinements auxiliaires de la première catégorie permettent de réduire le nombre de linéarisations possibles d'un plan partiel. Leur avantage est qu'ils réduisent les coûts futurs de gestion des plans partiels, notamment la vérification des contraintes auxiliaires (contraintes de préservation d'intervalles et contraintes

ponctuelles de valeur de vérité). On trouve dans cette catégorie les raffinements de *préordre*, qui permettent de transformer des plans partiellement ordonnés en plans totalement ordonnés. Les planificateurs TOCL [Barret et Weld 1994] et TO [Minton, Bresina et Drummond 1994] utilisent cette stratégie après chaque raffinement dans les espaces de plans. On trouve aussi le raffinements de *pré-position*, qui considèrent la possibilité pour deux actions d'être contiguës ou pas. Une stratégie de ce type est par exemple utilisée pour implémenter l'analyse des fins et des moyens, comme dans STRIPS [Fikes et Nilsson 1971] et PRODIGY [Fink et Veloso 1994]. Pour la complétude de ces techniques de raffinement, tous les ajouts possibles de contraintes d'ordre pour une étape doivent être considérés.

Les raffinements auxiliaires de la deuxième catégorie sont les raffinements de *pré-satisfaction*, qui permettent que toutes les linéarisations d'un plan partiel soient saines (c'est-à-dire conduisent à des séquences candidates) en forçant le respect des contraintes auxiliaires (protection d'intervalles et contraintes ponctuelles de valeur de vérité). Un plan partiel est donc partagé dans l'arbre de recherche, par ajout de contraintes de type promotion, démotion ou confrontation. Pour la complétude, toutes les techniques de protection pour une étape doivent être envisagées. Les planificateurs SNLP [McAllester et Rosenblitt 1991] et UCPOP [Penberthy et Weld 1992] utilisent cette stratégie afin qu'une étape ajoutée au plan ne constitue pas une menace pour les étapes déjà présentes dans le plan.

2.3.3. La planification par raffinements

L'algorithme principal d'un planificateur utilisant les stratégies de raffinement est composé de quatre étapes :

Raffiner (*P* : plan partiel)
1. *Terminaison* : si une séquence candidate minimale de *P* est solution, la retourner.
2. *Raffinement* : sélectionner une stratégie de raffinement *R*, l'appliquer à *P* afin d'obtenir un ensemble de plans partiels *R*(*P*).
3. *Sélection d'un plan partiel* : sélectionner un plan partiel *P'* de *R*(*P*). Cette étape est un point de retour arrière pour la complétude de l'algorithme.
4. *Appel récursif* : exécuter *Raffiner*(*P'*).

Le plan partiel fourni au premier appel à *Raffiner* contient comme première étape une action sans préconditions dont les effets représentent les fluents de l'état initial, et comme dernière étape une action sans effets dont les préconditions sont les buts du problème.

Il est important de remarquer que le choix d'une stratégie de raffinement n'est pas un point de retour arrière de cet algorithme. En effet, le résultat essentiel démontré par Kambhampati est le suivant : l'algorithme *Raffiner* est complet (resp. systématique) si et seulement les stratégies de raffinement utilisées à l'étape 2 sont complètes (resp. systématiques). Ceci implique évidemment l'utilisation d'un algorithme de recherche complet pour gérer les retours arrières de l'algorithme *Raffiner* et des différentes stratégies de raffinement, par exemple une recherche en largeur d'abord ou un algorithme de recherche heuristique complet comme A* ou IDA* [Pearl 1983] [Korf 1985]. Le planificateur UCP [Kambhampati 1996] montre de bonnes performances en entrelaçant diverses stratégies de raffinement, suivant les domaines sur lesquels il est employé.

2.4. Améliorer l'efficacité de la planification par recherche dans les espaces de plans

Plusieurs techniques permettant d'améliorer les performances des planificateurs qui effectuent une recherche dans les espaces de plans ont retenu notre attention, pour le rapport qu'elles ont avec nos propres travaux sur Graphplan. Bien que les planificateurs d'ordre partiel soient généralement considérés comme étant plus efficaces que les planificateurs d'ordre total [Minton, Bresina et Drummond 1994] [Barret et Weld 1994], il est aussi reconnu qu'ils s'avèrent parfois incapables de résoudre des problèmes de planification très simples [Srinivasan et Howe 1995].

Un des principaux inconvénients de ces planificateurs, parmi lesquels ont peut citer SNLP [McAllester et Rosenblitt 1991], POCL [Barret et Weld 1994], ou encore UCPOP [Penberthy et Weld 1992], provient de leurs stratégies de résolution de conflits. En effet, ils entrelacent les deux stratégies de raffinement suivantes :

- Raffinements dans les espaces de plans (cf. page 19) : après avoir ajouté une étape ou une contrainte de précédence dans le plan partiel, ils vérifient qu'aucune contrainte n'est violée (contraintes de protection

d'intervalles ou contraintes ponctuelles). Plus il y a de contraintes dans le plan, plus cette opération est coûteuse ; et plus il y a de conflits, plus le nombre de plans partiels générés pour résoudre ces conflits est important.

- Raffinements auxiliaires de pré-satisfaction (cf. page 19) : ils résolvent un conflit provoqué par une étape sur une contrainte auxiliaire. Comme précédemment, plus il y a de conflits et plus leur résolution est coûteuse.

Une solution pour éviter l'explosion combinatoire due à de trop nombreux conflits consiste à utiliser les raffinements auxiliaires de préordre ou bien de pré-position, ce qui revient à linéariser certaines parties du plan partiel. Ces techniques peuvent être efficaces dans certains domaines, tels ceux exhibés dans [Veloso et Blythe 1994] qui permettent de démontrer qu'un planificateur d'ordre total comme PRODIGY peut être plus efficace qu'un planificateur d'ordre partiel comme SNLP [McAllester et Rosenblitt 1991], contrairement à ce qui était l'opinion courante à cette époque.

Mais si l'on veut conserver une approche de moindre engagement, il faut continuer à gérer les conflits. Les deux questions qui se sont alors posées sont les suivantes : dans quel ordre doit-on considérer les choix des stratégies de raffinement (actions à utiliser pour établir des préconditions, ou conflits à résoudre), et dans quelle mesure la résolution de certains conflits est-elle utile pour résoudre un problème (c'est-à-dire peut-on remettre leur résolution à plus tard) ?

2.4.1. Heuristiques sur le choix des préconditions à établir et des conflits à résoudre

Après chaque raffinement d'un plan partiel, il faut choisir le prochain raffinement à effectuer : raffinement dans les espaces de plans ou raffinement auxiliaire. Le problème essentiel pour ce choix provient des conflits qui ne sont pas encore résolus. En effet, il se peut qu'un conflit du plan partiel ne puisse être résolu par les techniques de promotion, démotion ou confrontation. Si l'on ne s'en aperçoit pas immédiatement, tous les choix de raffinement que l'on peut faire conduisent inévitablement à un échec. Par exemple sur certains problèmes, 98% des noeuds examinés dans l'arbre de recherche par UCPOP correspondent à des plans partiels comportant un conflit impossible à résoudre [Joslin et Pollack 1996]. De même, il se peut qu'un conflit ne puisse être résolu que d'une seule manière (par exemple, uniquement par promotion par rapport à une étape). Si l'on effectue ce choix immédiatement, il n'y aura pas d'autre possibilité à examiner en cas de retour arrière. Par contre, si on effectue un autre raffinement produisant plusieurs plans partiels, il faudra répéter le même choix de résolution de ce conflit pour tous ces plans partiels et tous leurs descendants dans l'arbre de recherche.

Ces stratégies de résolution de conflits sont utilisées par exemple dans les heuristiques DMIN [Peot et Smith 1993], LCFR et QLCFR [Joslin et Pollack 1994], une variante de QLCFR proposée dans [Srinivasan et Howe 1995], et ZLIFO [Gerevini et Schubert 1996]. De même, il se peut qu'une précondition d'une étape du plan partiel ne puisse plus être établie par aucune autre étape, ou ne puisse être établie que d'une seule façon. De la même façon que pour la résolution des conflits, la décision à prendre est forcée puisqu'il ne reste qu'une seule possibilité. Cette stratégie est utilisée dans les heuristiques LCFR et ses variantes, et ZLIFO.

Lorsque tous les choix forcés ont été effectués, il faut choisir une stratégie de raffinement qui donnera a priori plusieurs embranchements dans l'arbre de recherche. On peut à nouveau appliquer une heuristique afin de faire le meilleur choix possible. Une possibilité consiste à choisir la stratégie qui produira le moins de plans partiels, ce qui est effectué dans LCFR en calculant tous les successeurs potentiels de tous les choix possibles. Le problème est que ce calcul est très coûteux, et les performances globales sont parfois dégradées : même si le nombre de plans partiels est largement inférieur, le temps de calcul est supérieur. Pour éviter ce problème, il est possible de ne calculer qu'une approximation et de la réutiliser pour les raffinements suivants comme dans QLCFR et [Srinivasan et Howe 1995]. Mais ce calcul restant lourd à effectuer et pouvant parfois dégrader le temps de recherche, le choix effectué pour ZLIFO consiste à sélectionner les préconditions à établir suivant une politique LIFO ("Last In First Out") qui donne de bons résultats [Gerevini et Schubert 1996].

2.4.2. Prise en considération tardive de décisions

L'idée de prendre en compte tardivement les conflits est proposée comme une heuristique dans [Peot et Smith 1993], au même titre que l'heuristique DMIN. Elle consiste à ne s'intéresser qu'à l'établissement des préconditions des étapes du plan partiel, sans se préoccuper d'aucun conflit. Ces derniers ne sont résolus qu'à la fin. Le problème est que cette technique ne pourra donner de bons résultats que dans les problèmes où il y a

très peu de contraintes ; mais dans le cas général, la plupart des plans partiels seront rejetés. Les résultats de [Peot et Smith 1993] montrent que cette stratégie n'est pas particulièrement intéressante.

Une technique plus prometteuse apparaît dans [Smith et Peot 1993]. Elle consiste à analyser les relations existant entre les opérateurs du domaine (non instanciés), en créant un graphe d'opérateurs qui contient deux types de noeuds : des noeuds représentant les opérateurs et des noeuds représentant les préconditions de ces opérateurs. Un arc se dirige soit d'une précondition vers l'opérateur qui l'utilise, soit d'un opérateur vers une précondition qui s'unifie avec un effet de cet opérateur. On ajoute ensuite les menaces potentielles sous forme d'arcs dirigés depuis un opérateur vers les préconditions qu'il menace (qui peuvent être des effets d'autres opérateurs). Une analyse de ce graphe permet alors de déterminer que la résolution de certains conflits peut être repoussée jusqu'à la fin de la planification, en faisant la remarque suivante : certaines contraintes d'ordre peuvent toujours résoudre ces conflits, et les autres conflits n'interfèrent pas avec ces contraintes d'ordre. Smith et Peot démontrent que si T est un ensemble de conflits dans le graphe d'opérateurs et que P est un sous-ensemble de ces conflits, alors la résolution des conflits de P peut toujours être effectuée à la fin de la planification si un ensemble de contraintes d'ordre résout ces conflits pour chaque résolution possible des conflits de T−P. Malheureusement, l'utilisation de ce théorème prend un temps exponentiel par rapport au nombre de conflits ce qui le rend difficilement utilisable dans le cas général. Certaines simplifications sont proposées, comme la prise en compte individuelle des conflits au lieu de sous-ensembles quelconques. Les tests préliminaires qui sont relatés dans [Smith et Peot 1993] montrent que cette technique peut aider dans certains cas, mais n'apporte pas grand chose dans les domaines où les opérateurs interfèrent beaucoup. Il ne semble pas y avoir eu de suite immédiate à ces travaux.

Une autre approche proposée dans [Joslin et Pollack 1996] avec le planificateur LC−Descartes consiste à représenter l'ensemble des choix possibles (établissement de préconditions et résolution des menaces) sous la forme d'un problème de satisfaction de contraintes avec domaines dynamiques. Ainsi, après chaque opération de raffinement du plan partiel, la consistance du CSP est vérifiée. Les conflits ne sont donc pas directement résolus : on teste juste si une résolution est possible. Cette approche peut être considérée comme la quintessence des stratégies de moindre engagement, puisque l'on ne considère qu'un seul plan partiel avec son CSP associé représentant les choix que l'on peut faire. Ainsi, la résolution des conflits dans son ensemble est remise à la fin de la planification. Il n'y a donc pas de retours arrières ; mais par contre, la vérification du CSP est très coûteuse puisque c'est un problème NP−complet. Pour pallier cet inconvénient, il s'avère intéressant dans certains cas d'effectuer tout de même des choix et de séparer plusieurs plans partiels dans un arbre de recherche. Le planificateur EC−Descartes de [Joslin et Pollack 1996] montre de meilleures performances que LC−Descartes.

Ces deux approches [Smith et Peot 1993] [Joslin et Pollack 1996] se sont donc heurtées plus ou moins au même problème : la complexité exponentielle induite soit par la détection avant la planification de conflits dont la résolution peut être repoussée à la fin de la planification, soit par la vérification que les conflits pourront toujours être résolus après la planification.

3. La planification par compilation

Après l'apparition de la planification par raffinement de plans partiels initiée par TWEAK, le domaine a subi un autre bouleversement profond avec l'avènement d'un nouveau paradigme, la planification par compilation. L'idée consiste à transformer un problème de planification en une structure particulière puis à effectuer une recherche dans cette structure. Les prémisses de ce changement sont apparus en 1992 avec les premiers résultats concernant la transformation d'un problème de planification exprimé en langage STRIPS en une base de clauses [Kautz et Selman 1992] dont les modèles, correspondant aux plans-solutions, peuvent être extraits à l'aide d'un prouveur SAT. Il s'agit d'un renouveau du paradigme d'une des premières approches de la planification, la planification par démonstration de théorèmes (cf. section 1.1.2, [Green 1969]). Ces premiers résultats, pourtant encourageants, n'ont pas eu un succès immédiat ; ce n'est qu'en 1996 que l'on voit apparaître la continuation directe de ces travaux [Kautz et Selman 1996] [Kautz, McAllester et Selman 1996]. L'idée a pourtant fait son chemin, et en 1995 est apparu Graphplan [Blum et Furst 1995], qui a profondément influencé la majeure partie des recherches effectuées ces dernières années.

3.1. Recherche dans les graphes de planification

Le planificateur Graphplan [Blum et Furst 1995] [Blum et Furst 1997] transforme un problème de planification en une structure appelée *graphe de planification*, puis effectue une recherche dans cette structure. Pour un aperçu détaillé du fonctionnement de Graphplan, le lecteur se reportera à la partie qui lui est consacrée (cf. section III.1, page 41). Ce que l'on peut retenir, c'est que le graphe de planification est un graphe orienté constitué de niveaux successifs contenant deux types de noeuds : les noeuds de fluents et les noeuds d'actions. Le premier niveau contient uniquement les noeuds des fluents qui appartiennent à l'état initial du problème. Le second niveau contient un noeud pour chaque action applicable dans l'état initial, et un noeud de fluent pour chacun de ses effets. Il y a trois types d'arcs : ceux qui vont des préconditions du premier niveau vers les actions du second niveau qui les utilisent, ceux qui vont des actions vers leurs ajouts, et ceux qui vont des actions vers leurs retraits. Afin de résoudre le problème du décor, on ajoute pour chaque fluent du premier niveau une action factice (appelée non-opérateur ou no-op) qui a ce fluent pour unique précondition et pour unique ajout. L'application d'un no-op signifie que le fluent correspondant n'a pas été retiré par une autre action. On calcule ensuite les exclusions mutuelles (aussi appelées mutex) entre les actions du second niveau, qui sont des contraintes binaires représentant l'impossibilité, pour deux actions, d'être appliquées en même temps à ce niveau. On calcule enfin les exclusions mutuelles entre les fluents correspondant aux ajouts et aux retraits des actions du second niveau, qui correspondent à l'impossibilité, pour deux fluents, d'être présents dans un même état. Les niveaux suivants sont calculés en suivant ce schéma, et en tenant compte des exclusions mutuelles entre les fluents du niveau précédent.

Le graphe est étendu niveau par niveau, jusqu'à ce que les buts du problème soient présents dans le dernier niveau et ne contiennent pas d'exclusions mutuelles. On tente alors d'extraire une solution par un algorithme de recherche arrière. En cas d'échec, le graphe est étendu d'un niveau supplémentaire et la recherche est relancée. Ce processus se poursuit jusqu'à l'obtention soit d'un plan-solution, soit d'une condition exprimant l'impossibilité de trouver une solution au problème.

Graphplan a suscité un nombre considérable de travaux. Nous allons maintenant passer en revue les plus importants d'entre eux.

3.1.1. Optimisation de la construction du graphe de planification

Instanciation des opérateurs et analyse de types

Graphplan travaille sur une structure entièrement propositionnelle : les fluents et les actions sont totalement instanciés par les constantes du domaine. L'instanciation des opérateurs peut être faite à chaque extension du graphe, en recherchant tous les unificateurs possibles des préconditions des opérateurs avec les fluents obtenus au niveau précédent. Mais comme une action qui apparaît à un niveau reste présente dans tous les niveaux suivants, la recherche des unificateurs qui ont donné ces actions est redondante à chaque extension du graphe. Devant l'augmentation croissante de la mémoire vive présente dans les machines, il est vite apparu comme un avantage de créer toutes les instanciations possibles des opérateurs par les constantes du domaine avant de planifier, l'applicabilité d'une action à un niveau du graphe se résumant alors à un test d'inclusion. De plus, on peut optimiser la construction du graphe en testant uniquement l'applicabilité des actions qui n'ont pas encore été employées. Le problème de cette méthode est qu'elle génère un grand nombre d'actions qui n'ont rien à voir avec le problème. En effet, les prédicats, et par extension les actions, concernent en général une partie seulement des objets ; beaucoup d'instanciations possibles ne sont donc pas pertinentes. Une possibilité est d'utiliser un langage de représentation typé, et de n'instancier un prédicat qu'avec des constantes dont les types correspondent aux types des arguments de ce prédicat. Mais cette approche impose des efforts supplémentaires pour le codage d'un domaine ; de plus, cette information est souvent codée par des prédicats unaires dans les préconditions des domaines.

Une méthode simple pour instancier efficacement des domaines non typés consiste à repérer les symboles de prédicats présents dans l'état initial d'un problème et qui sont absents des effets des opérateurs [Weld 1999]. On peut considérer que ces symboles de prédicats sont *inertes*. Les fluents formés avec ces prédicats seront présents dans tous les états et contraindront l'instanciation des opérateurs. On peut donc ne calculer que les instanciations des opérateurs compatibles avec ces fluents. De plus, ces derniers ne jouent aucun rôle dans la construction du graphe de planification, puisqu'ils sont présents dans tous les niveaux et ne prennent jamais part aux exclusions mutuelles ; on peut donc les supprimer de l'état initial et des préconditions des actions qu'ils ont permis d'instancier. On peut noter que ces prédicats ne se limitent pas aux prédicats unaires dont on se sert souvent pour typer implicitement les constantes d'un domaine ; on peut aussi trouver des prédicats binaires, par exemple pour représenter les tours de Hanoi où l'on utilise un prédicat binaire $plus{-}petit(d1, d2)$

pour spécifier que le disque *d*1 est plus petit que le disque *d*2. Une généralisation de cette méthode est proposée dans [Koehler et Hoffmann 1999] pour le langage ADL [Pednault 1989], en considérant aussi l'inertie des fluents qui apparaissent après l'instanciation. En effet, quand on instancie les actions partiellement instanciées grâce à l'inertie des prédicats, on peut encore trouver des actions non pertinentes ou des actions ayant des effets conditionnels non pertinents. Ces actions non pertinentes (ou bien les effets conditionnels non pertinents) peuvent aussi être supprimées, grâce à une étude sur l'inertie des fluents qui les constituent, en suivant le même principe que pour l'inertie des prédicats.

Une méthode plus fine d'analyse de types est proposée dans l'outil d'analyse TIM ("Type Inference Module") [Fox et Long 1998]. Les constantes et les variables des opérateurs sont typées par l'analyse d'automates finis que l'on peut construire à partir de la description du domaine : les symboles de prédicat sont les états, les opérateurs constituent les transitions entre les états et les constantes sont les informations véhiculées par les transitions. Grâce à cette analyse, on peut inférer des types plus complexes que ceux trouvés par l'inertie des prédicats, et même plus complexes que ceux que l'on peut définir "à la main". En effet, le type des constantes est défini par le rôle fonctionnel que ces constantes jouent dans les problèmes ; deux objets a priori de même type comme deux roues de voiture peuvent avoir un type inféré différent, si une roue est crevée et que l'on n'a rien pour la réparer (cf. le domaine Tyre World [Fox et Long 1998], page 410).

Construction du graphe guidée par les buts

Dans [Kambhampati 1997] et [Kambhampati, Parker et Lambrecht 1997], le graphe de planification est vu comme une représentation disjonctive de l'espace d'états généré par un planificateur fonctionnant en chaînage avant dans les espaces d'états. Le défaut d'une telle approche est qu'elle n'est pas dirigée par les buts, c'est-à-dire que des actions n'ayant rien à apporter à la résolution du problème sont utilisées et viennent augmenter inutilement le graphe de planification. Dans la planification dans les espaces d'états, ce problème est résolu en faisant du chaînage arrière ; l'idée de construire le graphe de planification à partir des buts est alors naturellement venue [Kambhampati, Parker et Lambrecht 1997]. Le principe en est le suivant : créer un certain nombre de niveaux de fluents et de propositions en arrière depuis les buts, puis reconstruire le graphe normalement en prenant comme premier niveau l'intersection entre l'état initial du problème et les préconditions des actions du dernier niveau construit en arrière. Les niveaux construits par l'arrière servent alors de guides pour déterminer les actions pertinentes à employer pour la construction du graphe. L'idée d'utiliser une telle construction en chaînage arrière pour guider un processus de chaînage avant a été notamment utilisée dans le planificateur UNPOP [McDermott 1996] pour une version moderne de l'analyse des fins et des moyens.

Mais bien que [Kambhampati, Parker et Lambrecht 1997] rapportent des résultats expérimentaux intéressants, cette technique ne semble pas avoir été réutilisée telle quelle pour Graphplan. Son inconvénient majeur est que l'on doit reconstruire entièrement le graphe de planification à chaque itération ; de plus, d'autres techniques pour la sélection des actions pertinentes ont été développées avec succès.

Heuristiques sur la pertinence des fluents de l'état initial

Une autre technique permettant de réduire la taille du graphe de planification est proposée dans [Nebel, Dimopoulos et Koehler 1997] pour le planificateur IPP basé sur Graphplan. Elle est inspirée des travaux sur le planificateur UNPOP [McDermott 1996] [McDermott 1999], à la différence près qu'elle n'est utilisée qu'une seule fois avant de planifier, et qu'elle est calculée à partir des actions totalement instanciées. Elle offre plusieurs heuristiques qui permettent de supprimer de l'état initial les fluents qui ne sont probablement pas utiles pour résoudre le problème. L'inconvénient majeur est donc qu'elle rend Graphplan incomplet, mais il semble qu'elle fournisse pourtant de bons résultats. De plus, il y a de fortes chances pour que Graphplan échoue rapidement si l'heuristique a supprimé trop de fluents ; il reste donc toujours la possibilité de relancer la planification avec l'état initial complet.

Le principe de ce calcul est de construire un graphe ET/OU à partir des buts du problème. La racine de l'arbre est un noeud ET et correspond au but du problème. Les descendants de ce noeud sont des noeuds OU correspondant à chacun des sous-buts. Les descendants de ces noeuds sont les ensembles des préconditions des actions qui établissent ces sous-buts. Un noeud OU est résolu si le fluent qui lui correspond appartient à l'état initial ou si un de ses descendants est résolu. Un noeud ET est résolu si tous ses descendants sont résolus. Un ensemble de fluents utiles pour résoudre un problème est donc constitué par les feuilles d'un sous-arbre solution du graphe. Le but est de trouver les ensembles les plus petits possibles pour l'inclusion, mais il s'agit d'un problème NP−complet donc on doit faire des approximations. De plus, comme on ne tient pas compte des retraits des actions, prendre un seul de ces ensembles sera a priori trop restrictif pour constituer l'état

initial. Plusieurs possibilités sont alors envisagées pour sélectionner un ensemble assez grand, le principe étant de prendre un ensemble suffisamment réduit pour améliorer les performances globales du planificateur mais suffisamment grand pour ne pas rejeter des fluents absolument nécessaires.

Structure circulaire à deux niveaux et calcul des mutex

On trouve dans [Smith et Weld 1999] et [Long et Fox 1999] une construction optimisée du graphe de planification, sous la forme d'une structure circulaire à deux niveaux. Cette construction est basée sur la propriété de monotonicité des actions, des fluents et des exclusions mutuelles : un noeud de fluent ou d'action apparaissant à un niveau *i* du graphe de planification reste présent dans tous les niveaux supérieurs du graphe, et une mutex disparaissant à un niveau *i* du graphe n'apparaîtra plus dans les niveaux supérieurs. Cette propriété permet de construire le graphe de planification sous la forme d'une structure formée d'un seul ensemble de noeuds d'actions et d'un seul ensemble de noeuds de fluents ; pour chacun de ces noeuds, on conserve le niveau d'apparition de l'action ou du fluent correspondant. Pour chaque mutex, on conserve le niveau le plus élevé dans lequel on la rencontre. Les noeuds d'actions sont reliés aux noeuds de fluents qui correspondent à leurs préconditions, leurs ajouts et leur retraits. Pour chaque noeud de fluent, on conserve la liste des actions qui l'utilisent, le produisent ou le retirent à chaque niveau. Cette construction est optimisée dans le planificateur STAN ("STate Analysis") [Long et Fox 1999] par l'utilisation de tableaux de bits.

Une telle construction du graphe de planification permet évidemment un gain de place mémoire important, mais aussi un gain de temps puisqu'on limite la création d'informations complexes. De plus, le calcul des mutex peut être simplifié (cf. [Smith et Weld 1999]) à l'aide de leur catégorisation : certaines mutex seront toujours présentes (par exemple pour des actions qui ont des effets contradictoires) alors que d'autres ne le seront pas. En effet, avec la disparition d'une mutex entre deux fluents, disparaît une des raisons qui fait que deux actions sont mutuellement exclusives. La mutex entre ces deux actions peut donc disparaître, si c'était sa seule raison d'être. Ainsi, certaines mutex ne seront jamais recalculées (celles qui sont permanentes), alors que d'autres le seront. Une fois que ces dernières auront disparu, il sera inutile de les recalculer.

Un autre moyen pour analyser les mutex vient du calcul de types effectué par la méthode de [Fox et Long 1998]. Cette analyse de types permet non seulement d'accélérer l'instanciation des actions, mais aussi de déduire des invariants d'état qui peuvent être employés de multiple manière. Par exemple dans le Monde des cubes classique, TIM peut déduire que l'on ne peut pas avoir deux cubes posés sur seul cube ; ce qui sera détecté par Graphplan comme étant une mutex. Ainsi, les invariants d'états trouvés par TIM contiennent entre autres toutes les mutex que trouve Graphplan qui sont permanentes, c'est-à-dire celles que l'on trouve dans tous les niveaux du graphe. Graphplan devant recalculer pour chaque niveau ces mutex, quand il ne s'agit pas d'une relation syntaxique entre des actions (intersection des ajouts d'une action et des retraits d'une autre, etc.), le gain de temps pour la construction du graphe de planification est donc évident. De plus, on peut détecter des mutex permanentes que Graphplan ne peut pas détecter [Fox et Long 2000]. Par exemple, si trois actions mutuellement exclusives deux à deux de façon permanente produisent deux fluents (deux de ces actions en produisant un, la troisième les produisant les deux), Graphplan ne peut pas détecter que les deux fluents sont mutuellement exclusifs. [Fox et Long 2000] exhibent un domaine où cela se produit et où un invariant d'état est déduit, qui permet l'ajout d'une mutex que Graphplan ne trouve pas.

3.1.2. Optimisation de l'extraction de la solution

Utilisation de techniques de satisfaction de contraintes

Comme il est suggéré dans [Kambhampati, Parker et Lambrecht 1997] et démontré dans [Kambhampati 1999] et [Kambhampati 2000], la phase d'extraction de la solution peut être vue comme la résolution d'un problème de satisfaction de contraintes dynamique ("Dynamic Satisfaction Constraint Problem", DCSP [Mittal et Falkenhainer 1990]). En effet, durant cette phase, chaque fluent $f(n)$ à un niveau n du graphe de planification peut être assimilé à une variable d'un DCSP ; l'ensemble des actions qui produisent chaque fluent $f(n)$ constitue son domaine et les mutex produites durant la construction du graphe peuvent être assimilées aux contraintes du DCSP. Graphplan tente alors d'extraire un plan-solution du graphe de planification en assignant une valeur (une action) à chaque variable (un fluent) afin de satisfaire l'ensemble des contraintes (les mutex). L'assignation des valeurs aux variables est un procédé dynamique dans le sens où chaque assignation à un niveau donné active d'autres variables au niveau précédent.

On peut ainsi utiliser des techniques efficaces pour la résolution de problèmes de satisfaction de contraintes, notamment le retour arrière intelligent ("Dynamic Directed Bactracking", DDB), le forward checking et l'apprentissage par explication d'échecs ("Explanation Based Learning", EBL). Les améliorations constatées

pour le temps d'extraction de la solution sont parfois considérables (cf. [Kambhampati 2000], jusqu'à plus de 1000 fois plus rapide).

On peut signaler au passage deux approches qui permettent d'utiliser des techniques de satisfactions de contraintes classiques pour résoudre un problème de planification. La première [van Beek et Chen 1999] transforme manuellement un problème de planification en un CSP, à la manière de la première approche de la planification par satisfiabilité [Kautz et Selman 1992]. La deuxième transforme automatiquement le graphe de planification en un CSP classique [Do et Kambhampati 2000].

Emploi d'heuristiques pour le choix des actions et des sous-buts

Contrairement à ce que Blum et Furst ont avancé pour la version originale de Graphplan [Blum et Furst 1997], l'ordre dans lequel on considère les sous-buts et les actions lors de l'extraction de la solution a une grande importance, surtout si on utilise les techniques CSP décrites dans le paragraphe précédent.

La première heuristique proposée [Kambhampati 2000] consiste à choisir en premier la variable qui a le domaine le plus restreint, c'est-à-dire le sous-but produit par le plus petit nombre d'actions. Cette mesure est dynamique si l'on utilise le forward checking, puisque les actions non utilisables à cause des mutex sont supprimées des domaines des variables ; mais même dans ce cas, le temps d'exécution n'est que faiblement amélioré, et se trouve même dégradé pour certains problèmes.

Des familles d'heuristiques apportant des résultats beaucoup plus intéressants ont été proposées parallèlement dans [Kambhampati et Nigenda 2000] et [Cayrol, Régnier et Vidal 2000b]. Ces heuristiques sont basées sur le niveau d'apparition des actions et des fluents, ce qui constitue apparemment une meilleure mesure du degré de contrainte des variables et des valeurs. Nous étudions et améliorons l'une d'entre elles dans la partie consacrée à Graphplan, section III.6.3, page 88.

Symétrie de la recherche

A cause de la monotonicité des actions, des fluents et des contraintes d'exclusion mutuelle dans le graphe de planification, la recherche effectuée à un niveau i+1 du graphe de planification pour un ensemble de sous-buts B est très semblable à celle effectuée au niveau i pour ce même ensemble B. En effet, l'ensemble A des actions qui produisent les fluents de B au niveau i est un sous-ensemble de l'ensemble A' des actions qui produisent B au niveau i+1. Inversement, les mutex sur les actions de A' au niveau i+1 est un sous-ensemble des mutex sur les actions de A au niveau i. Une technique proposée dans [Zimmerman et Kambhampati 1999] permet de profiter de cette symétrie dans la structure du graphe de planification pour éviter, lors de la recherche au niveau i+1, de refaire le travail qui a été effectué au niveau i. Par exemple, si une contrainte d'exclusion mutuelle ayant causé un retour arrière pendant la recherche au niveau i n'est plus présente au niveau i+1, la recherche au niveau i+1 recommence à l'endroit où la mutex a fait échouer la recherche. Le problème de cette approche est qu'il faut conserver la trace d'un grand nombre d'informations concernant les endroits où la recherche a échoué afin de pouvoir les réutiliser dans les niveaux supérieurs du graphe. Ainsi, si les résultats expérimentaux du planificateur EGBG [Zimmerman et Kambhampati 1999] implémentant ces techniques sont intéressants pour des problèmes de petite taille, les besoins de mémoire vive sont trop importants. L'utilisation des techniques EBL/DDB permettent de limiter la taille des informations stockées, mais cette amélioration reste insuffisante pour une utilisation véritablement efficace de cette technique.

Symétrie dans les problèmes de planification

Une autre symétrie intéressante à exploiter est celle inhérentes aux problèmes de planification. Elle concerne de nombreux domaines, et est une cause d'échec de beaucoup de planificateurs pour des problèmes triviaux. Par exemple, considérons le domaine du Ferry dont l'objectif est de faire traverser une rivière à un certain nombre de voitures. Le ferry ne pouvant embarquer qu'une seule voiture à la fois, il doit effectuer des aller-retours successifs entre les deux rives pour transporter toutes les voitures. Toutes les voitures sont différentes (repérées par une constante différente dans la description du problème), mais elles jouent exactement le même rôle. Ainsi, peu importe que l'on fasse traverser la voiture A avant la voiture B et vice-versa : le résultat sera le même. Les voitures A et B peuvent donc être considérées comme jouant des rôles symétriques, ce que ne sait pas faire automatiquement un planificateur. Supposons que l'on doive faire traverser la rivière à 10 voitures ; le plan forcément linéaire réalisant cette tâche comporte 39 actions. Dans Graphplan, la recherche dans un graphe de planification comportant 38 niveaux va échouer ; et tous les ordonnancements entre les actions seront alors considérés. Toutes les voitures ayant un rôle semblable, l'échec sur un ordonnancement serait suffisant pour savoir que le problème est impossible à résoudre en 38 niveaux.

Une méthode indépendante d'une technique particulière de planification est proposée dans [Fox et Long 2000] pour détecter cette symétrie statique des problèmes de planification. L'utilisation de cette symétrie qui est elle dépendante d'un planificateur a été implémentée avec succès dans le planificateur STAN [Long et Fox 1999]. Le principe consiste à détecter les groupes d'objets symétriques grâce au rôle identique qu'ils ont dans l'état initial et le but du problème, s'ils sont pris en compte de la même façon par tous les opérateurs. Lors de la recherche d'une solution, l'utilisation dans un ordre différent d'objets symétriques sera évité. Les performances relatées dans [Fox et Long 2000] témoignent de l'efficacité de cette méthode pour des domaines triviaux comme le Ferry.

Ordonnancement de sous-buts

La prise en compte des interactions entre les sous-buts permet de diminuer la complexité de la résolution d'un problème, en particulier s'ils sont indépendants les uns des autres [Korf 1987]. Mais la plupart du temps ce n'est pas le cas, et [Koehler 1998b] [Koehler et Hoffmann 2000] proposent une méthode permettant de calculer un ordre des sous-buts dans le cas où ils sont difficilement sérialisables. Dans le cas général, ce calcul est exponentiel, mais une méthode polynomiale est proposée pour obtenir une bonne estimation. Suivant l'utilisation de cette estimation, l'algorithme de planification peut être rendu incomplet. Mais en général, le planificateur IPP [Koehler et al. 1997] modifié pour utiliser cette information montre de bonnes performances.

3.1.3. Expressivité du langage, gestion des ressources et de l'incertitude

Expressivité du langage

Le langage utilisé par la version originale de Graphplan est un langage de type STRIPS très simple, sans négation dans les préconditions des actions. L'utilisation pour Graphplan d'un langage plus expressif (préconditions disjonctives, quantification, effets conditionnels) comme ADL [Pednault 1989] (ou du moins d'un sous-ensemble) s'est donc posée relativement vite.

L'approche la plus simple est celle de [Gazen et Knoblock 1997], dans laquelle un pré-processeur transforme un domaine exprimé sous format ADL en son équivalent sous format STRIPS. Le problème de cette approche est qu'elle multiplie les actions possibles : par exemple, une action ayant une précondition disjonctive sera remplacée par autant d'actions mutuellement exclusives qu'il y a de fluents dans la disjonction, chacune de ces actions ayant un de ces fluents comme précondition. De même, une action contenant un effet conditionnel sera remplacé par deux actions : dans la première, l'effet conditionnel est supprimé et la négation de l'antécédent de l'effet conditionnel est ajouté aux préconditions ; et dans la seconde, les préconditions de l'effet conditionnel sont ajoutées aux préconditions de l'action et le conséquent de l'effet conditionnel est ajouté aux effets de l'action. Graphplan ne nécessite aucune modification, mais ses performances sont sérieusement pénalisées par cette augmentation prohibitive du nombre d'actions.

Une meilleure amélioration de l'expressivité passe donc par une modification de Graphplan, qui a été réalisée dans le planificateur IPP [Koehler et al. 1997]. Comme on le voit dans [Kambhampati, Parker et Lambrecht 1997], le véritable problème se pose pour la prise en compte des effets conditionnels. En effet :

- Pour les préconditions disjonctives : il suffit de rajouter un point de choix dans l'algorithme de recherche pour chaque partie de la disjonction. Il faut aussi faire en sorte qu'une action soit utilisée dans le graphe de planification si l'union d'au moins une des partie de la disjonction avec les préconditions non disjonctives ne comporte pas deux fluents mutuellement exclusifs.
- Pour la quantification : on fait la restriction de ne travailler que sur des domaines typés, ce qui a priori n'enlève rien à l'expressivité. La quantification portera sur les types. Comme on travaille sur des domaines finis, on peut remplacer une expression universellement quantifiée par la base de Herbrand qui lui correspond, et une expression existentielle par une formule de Skolem dans laquelle les constantes de Skolem s'ajouteront aux paramètres de l'action (la quantification existentielle ne pouvant pas être utilisée dans les effets de l'action).
- Pour les effets conditionnels : des modifications complexes de l'algorithme de recherche doivent être faites, afin notamment d'empêcher que les antécédents d'un effet conditionnel que l'on ne souhaite pas appliquer soient présents avant d'appliquer l'action. De plus, il faut faire un choix concernant les mutex. Soit les effets conditionnels n'entrent pas en compte dans le calcul des exclusions mutuelles, comme dans IPP [Koehler et al. 1997], soit il faut étendre la définition des mutex par des règles plus complexes [Anderson, Smith et Weld 1998]. Même si le calcul effectué dans ce dernier cas est plus long, il semble que dans certains domaines cela apporte une importante amélioration.

Gestion des ressources

Une extension de IPP [Koehler et al. 1997] permettant de gérer les ressources est proposée dans [Koehler 1998a]. Mais contrairement aux autres approches permettant de gérer des ressources, souvent basées sur la planification dans les espaces de plans partiels, comme DEVISER [Vere 1983], ZENO [Penberthy et Weld 1994], ou IxTeT [Ghallab et Mounir-Alaoui 1989] [Laruelle 1994] [Laborie 1995], le temps n'est pas géré comme une ressource continue, ce qui limite sérieusement les applications possibles : cette approche n'est pas basée sur un modèle temporel explicite. Les actions sont considérées comme ayant une exécution atomique comme dans Graphplan, et la durée d'une action est considérée comme une ressource produite par l'action, c'est-à-dire comme l'avancement d'un paramètre sur une certaine échelle de temps. Une valeur maximum de cette échelle peut être spécifiée dans les buts, afin de trouver un plan dont l'exécution ne dépassera pas une limite de temps donnée. Mais on ne peut pas spécifier qu'une ressource donnée doit être disponible à un certain instant, ni la durée des effets des actions, etc. C'est au prix de cette sévère limitation que les résultats expérimentaux relatés sont assez bons en comparaison avec ZENO [Penberthy et Weld 1994] ; mais cette approche semble difficilement extensible à une réelle utilisation d'un modèle temporel. En effet, comme dans Graphplan, on ne considère les actions exécutables en parallèle qu'à chaque niveau du graphe, les rendant ainsi instantanées ; ce qui n'a pas de lien véritable avec leur durée et la durée totale d'exécution du plan. Il semble d'ailleurs que ce soit à ce jour la seule approche permettant l'utilisation (limitée) de ressources dans un planificateur basé sur Graphplan.

Une autre extension de Graphplan pour utiliser des actions ayant une durée est proposée dans le planificateur TGP ("Temporal GraphPlan") [Smith et Weld 1999]. La même restriction que dans [Koehler 1998a] est faite : seule la durée de l'action est prise en compte, les préconditions devant être vérifiées durant toute la durée d'exécution de l'action et les effets étant effectifs à la fin de l'exécution de l'action. Par contre, la structure du graphe est modifiée pour prendre en compte la durée des actions par l'utilisation d'une structure circulaire à deux niveaux similaire à celle du planificateur STAN [Long et Fox 1999]. La différence est que le niveau de départ d'un fluent ou d'une action ne correspond pas au niveau d'apparition dans le graphe de planification classique, mais au temps minimum d'apparition calculé d'après la durée des actions. De plus, la notion d'exclusion mutuelle est étendue à l'exclusion entre actions et fluents, afin de permettre le chevauchement de l'exécution des actions. Grâce à ce raisonnement sur les mutex, TGP présente de bonnes performances par rapport à une version de Graphplan qui ne gère pas la durée des actions.

Gestion de l'incertitude

Plusieurs extensions de Graphplan ont été proposées pour gérer l'incertitude sur les actions en environnement totalement ou partiellement observable. Le planificateur CGP ("Conformant GraphPlan") [Smith et Weld 1998] permet de gérer l'incertitude dans les effets des actions (effets "non-déterministes") ainsi que la connaissance partielle de l'état initial. Plusieurs graphes de planification sont créés pour toutes les possibilités correspondant à chaque source d'incertitude, ce qui induit une structure très lourde et des performances peu efficaces. Le planificateur SGP ("Sensory GraphPlan") [Weld, Anderson et Smith 1998] étend CGP à l'utilisation d'actions sensorielles permettant d'acquérir des informations durant l'exécution du plan. Là aussi, un graphe de planification est créé pour chaque résultat possible de l'observation. D'autres approches ont également été proposées pour utiliser des probabilités [Blum et Langford 1998] [Blum et Langford 1999] ou encore des possibilités [Guéré et Alami 1999].

3.2. Planification par satisfiabilité

L'idée d'utiliser un démonstrateur automatique de théorèmes pour résoudre un problème de planification n'est pas neuve (cf. section 1.1.2, page 12, QA3 [Green 1969] et le calcul des situations [McCarthy et Hayes 1969]). Les différences essentielles entre ces dernières et les approches que nous allons étudier maintenant sont d'une part l'utilisation du langage STRIPS pour représenter les problèmes, et d'autre part l'utilisation de la logique propositionnelle au lieu de la logique du premier ordre pour effectuer le codage du problème. Un tel codage occupe certainement une place mémoire beaucoup plus importante ; mais devant l'augmentation constante de la mémoire vive des machines, ce n'est plus réellement un problème : on dépasse généralement la capacité de temps de calcul avant la capacité de mémoire vive.

3.2.1. Motivations

Depuis une dizaine d'années, les prouveurs SAT basés sur la procédure de Davis et Putnam [Davis et Putnam 1960] [Davis, Logemann et Loveland 1962] on fait des progrès considérables[1]. Ceci est dû notamment à :

- Le développement d'algorithmes incomplets, comme GSAT [Selman, Levesque et Mitchel 1992] et WALKSAT [Selman, Kautz et Cohen 1994], qui permettent la résolution d'instances de problèmes particulièrement difficiles pour les prouveurs complets.
- La mise au point d'heuristiques particulièrement efficaces basées sur la propagation unitaire, implémentées par exemple dans le prouveur SATZ [Li et Anbulagan 1997]. Ce dernier bénéficie aussi d'une implémentation très fine qui en fait l'un des prouveurs complets les plus efficaces.
- L'implémentation du retour arrière intelligent, par exemple dans les prouveurs RELSAT [Bayardo et Schrag 1997] et GRASP [Marques Silva et Sakallah 1996].
- Les techniques de choix aléatoire de la variable de branchement dans des algorithmes complets [Gomes, Selman et Kautz 1998]. Le principe consiste à relancer la recherche d'une interprétation à partir de la racine de l'arbre de recherche, après un certain nombre de retours arrières, afin de corriger une défaillance initiale de l'heuristique qui a entraîné la recherche dans une mauvaise direction.

Les approches dédiées à la planification à cette époque (TWEAK, SNLP, UCPOP...) n'offrant pas des performances vraiment satisfaisantes, l'idée de profiter des progrès constants effectués dans le domaine de la satisfiabilité pour résoudre efficacement les problèmes de planification a alors vu le jour [Kautz et Selman 1992]. La planification par satisfiabilité offre également un cadre formel très intéressant, permettant notamment l'utilisation très naturelle de connaissances sur le domaine qui peuvent se représenter sous forme de clauses exprimant des contraintes entre actions et fluents.

3.2.2. Principe

La différence fondamentale entre la planification basée sur la logique des prédicats dans QA3 et le calcul des situations et la planification par satisfiabilité est que cette dernière travaille sur un ensemble fini de variables propositionnelles, alors que la logique des prédicats permet l'utilisation de fonctions pouvant engendrer une infinité de termes du langage. Deux actions identiques pouvant apparaître à des endroits différents d'un même plan, il faut être capable de les différencier. En calcul des prédicats, ceci est fait à l'aide d'un terme fonctionnel qui représente les actions qui les précèdent dans le plan ; en logique propositionnelle, on leur associe une proposition différente (ou éventuellement une conjonction de formules, cf. section 3.2.3, représentation des actions). Or, le problème qui se pose est qu'on ne connaît pas à l'avance la longueur d'un plan-solution du problème. On ne peut donc pas en logique propositionnelle créer un codage unique permettant de résoudre un problème, puisqu'il faudrait créer suffisamment de variables propositionnelles ou de formules pour représenter toutes les actions de tous les plans possibles, c'est-à-dire une infinité.

Une solution à ce problème consisterait à créer un codage représentant tous les plans de longueur 1, puis un codage représentant tous les plans de longueur 2..., jusqu'à trouver un plan-solution par la résolution d'un de ces codages. Les premières approches de la planification par satisfiabilité [Kautz et Selman 1992] [Kautz et Selman 1996] [Kautz, McAllester et Selman 1996] se sont intéressées à l'expression d'un codage pour lequel on connaît à l'avance la longueur minimale d'un plan-solution, puis des solutions ont été proposées pour résoudre un problème sans cette information [Ernst, Millstein et Weld 1997] [Kautz et Selman 1998b] [Baioletti, Marcugini et Milani 1998c] [Kautz et Selman 1999].

En logique des prédicats, l'obtention d'un plan-solution se fait par la démonstration d'un théorème, c'est-à-dire la recherche d'une substitution dont l'application à la formule représentant les buts du problème rend cette dernière valide dans la théorie ; en logique propositionnelle, on recherche un modèle de la formule : tous les modèles différents correspondent à des plans-solutions différents. Pour la complétude de ce procédé, il convient donc que tous les modèles de la formule issue du codage d'un problème correspondent exactement à tous les plans-solutions d'une longueur donnée.

[1] Le lecteur trouvera dans [Castell 1997] et [Le Berre 2000] une description plus complète des progrès effectués dans ce domaine.

3.2.3. Influence du codage

D'un point de vue idéal, tout ce qui concerne les techniques de résolution est laissé de côté ; l'objectif étant de profiter automatiquement des progrès effectués dans le cadre général de la recherche en satisfiabilité. Le problème essentiel qui se pose alors est la compilation d'un problème sous forme d'une base de clauses. Un même problème pouvant être codé de différentes manières, les performances des prouveurs SAT dépendent donc directement de la qualité de ces codages. Cette qualité peut être mesurée par rapport au nombre de variables et de clauses, par rapport à la longueur des clauses, etc. Ces facteurs sont fortement corrélés : diminuer le nombre de variables fait souvent augmenter le nombre de clauses, inversement diminuer le nombre de clauses fait augmenter le nombre de variables. On considère généralement deux aspects d'un codage : sa forme générale, et la représentation des actions.

Il existe essentiellement deux formes de codage : les codages dans les espaces d'états, qui traduisent les transitions entre niveaux successifs et utilisent des frame-axiomes pour transporter la valeur des fluents qui ne sont pas affectés par les actions appliquées à un niveau, et les codages dans les espaces de plans ou causaux, qui traduisent l'insertion d'actions à divers niveaux du plan et ne nécessitent pas l'utilisation de frame-axiomes.

Codages dans les espaces d'états

Les codages dans les espaces d'états ont été les premiers proposés. Ils ont aussi été les plus étudiés, à juste titre car il semble que ce soient les codages qui fournissent les bases les plus restreintes, dont la résolution par les prouveurs SAT est la plus efficace.

Le premier codage proposé [Kautz et Selman 1992], appelé codage "linéaire", est basé sur les principes suivants :

- *État initial et but* : les fluents de l'état initial sont vrais au niveau 0, les fluents qui ne sont pas dans l'état initial sont faux au niveau 0 et les fluents du but sont vrais au niveau maximal k du codage.
- *Application des actions* : une action au niveau i implique ses préconditions au niveau $i-1$, et ses ajouts et la négation de ses retraits au niveau i.
- *Frame-axiomes* : ce sont les frame-axiomes classiques, c'est-à-dire que l'on spécifie pour chaque action que les fluents qu'elle n'affecte pas restent vrais (ou faux) après l'application de l'action. Ces frame-axiomes sont ceux proposés originellement pour le calcul des situations [McCarthy et Hayes 1969].
- *Linéarité* : une action et une seule est appliquée à chaque niveau. Pour chaque niveau, on ajoute une clause formée par la disjonction de toutes les actions qui oblige à utiliser au moins une action ; et pour chaque paire d'actions de ce niveau, on ajoute la négation de la conjonction de ces deux actions qui interdit leur utilisation conjointe au même niveau. De plus, une action "nulle" est nécessaire pour exprimer qu'aucune action n'est utilisée à un niveau si le nombre de niveaux maximum du codage est supérieur à la longueur d'un plan minimal. En effet, pour transporter la valeur des fluents d'un niveau sur un autre, il faut pouvoir déclencher un frame-axiome, et par conséquent utiliser une action. Dans ce cas, on utilisera l'action nulle dont le frame-axiome associé transporte tous les fluents d'un niveau sur l'autre.

Une première amélioration de ce codage, proposée pour le planificateur SATPLAN dans [Kautz, McAllester et Selman 1996], concerne les frame-axiomes. Ils remplacent les frame-axiomes classiques par les frame-axiomes explicatifs [Haas 1987] qui spécifient que si la valeur d'un fluent change, c'est parce que l'on a appliqué une action qui l'ajoute ou le retire. Les clauses associées aux frame-axiomes explicatifs sont de la forme suivante : si un fluent est vrai au niveau $i-1$ et faux au niveau i, alors la disjonction des actions qui retirent ce fluent au niveau i est vraie ; si un fluent est faux au niveau $i-1$ et vrai au niveau i, alors la disjonction des actions qui ajoutent ce fluent au niveau i est vraie. [Kautz, McAllester et Selman 1996] remarquent aussi qu'il n'y a plus besoin de la règle spécifiant que l'on applique au moins une action à un niveau (la disjonction de toutes les actions), ni de l'action "nulle". En effet, les frame-axiomes explicatifs garantissent le transport des valeurs des fluents même si aucune action n'est appliquée.

[Kautz et Selman 1996] remarquent que l'utilisation des frame-axiomes explicatifs permet le parallélisme, contrairement aux frame-axiomes classiques. En effet, lors de l'utilisation de ces derniers, les fluents non affectés par l'utilisation d'une action sont automatiquement transportés au niveau suivant ; on ne peut donc pas appliquer une autre action, qui risquerait de modifier ces fluents. Comme les frame-axiomes explicatifs spécifient uniquement ce qui change, le parallélisme est possible : [Kautz et Selman 1996] proposent simplement de modifier les règles de linéarité afin d'interdire seulement l'application simultanée d'actions ayant des interférences entre effets de l'une et préconditions de l'autre. Les interférences entre ajouts et

retraits sont pris en compte par la règle de description des actions. Ce codage a été repris et étudié plus en détail dans [Ernst, Millstein et Weld 1997], [Mali et Kambhampati 1998] et [Mali et Kambhampati 1999].

Une variation de ces codages dans les espaces d'états est le codage basé sur Graphplan [Kautz, McAllester et Selman 1996], qui apparaît comme une version simplifiée du codage dans les espace d'états avec frame-axiomes explicatifs. La règle concernant les actions est simplifiée : il suffit qu'une action implique ses préconditions. Les interférences entre ajouts et retraits des actions sont traités par des clauses d'exclusion entre actions. Des non-opérateurs semblables à ceux de Graphplan sont ajoutés pour tous les fluents, et sont traités comme les autres actions. Les frame-axiomes spécifiant qu'un fluent vrai à un niveau $i-1$ et faux au niveau i implique la disjonction des actions qui le retirent ne sont plus nécessaires ; quant aux frame-axiomes qui gèrent les ajouts, ils sont modifiés de la façon suivante : un fluent vrai au niveau i implique la disjonction du non-opérateur du fluent et des actions qui l'ajoutent.

Une autre variation de ces codages est le codage basé sur les états, obtenu en supprimant tous les symboles d'action dans le codage dans les espaces d'états avec frame-axiomes explicatifs. Les actions utilisées sont ensuite déduites des modèles trouvés. [Kautz, McAllester et Selman 1996] ne donnent pas de méthode automatique pour obtenir ce codage : ils le calculent à la main pour certains domaines. D'après [Kautz et Selman 1996], il s'agit du codage le plus compact et qui donne les meilleurs résultats. Il n'a pas été réutilisé, à part pour effectuer des comparaisons, son défaut majeur étant l'absence de méthode de génération automatique.

Codages dans les espaces de plans

Ces codages, dont la première version est proposée dans [Kautz, McAllester et Selman 1996], n'utilisent plus de frame-axiomes pour transporter les valeurs des fluents d'un niveau sur l'autre, mais spécifient directement dans les règles qu'une action utilise pour préconditions des fluents présents dans les niveaux inférieurs (pas uniquement au niveau précédent). Il se pose alors le problème de la protection de l'intervalle entre lequel la précondition est produite et l'action qui l'utilise, d'une manière similaire aux planificateurs qui effectuent une recherche dans les espaces de plans partiels comme TWEAK, SNLP ou UCPOP. Les liens causaux sont alors codés sous forme de règles, et protégés par les techniques de promotion ou démotion. Aucune autre étude de ce type de codage ou même d'évaluation expérimentale n'a été faite avant [Mali et Kambhampati 1998] et [Mali et Kambhampati 1999], qui en étudient plusieurs variantes. Ils démontrent que le codage originel proposé dans [Kautz, McAllester et Selman 1996] est inutilement complexe, et proposent deux simplifications possibles. La meilleure de ces variantes, que ce soit en taille ou en performance par rapport à la résolution par un prouveur SAT, est basée sur la technique de protection du chevalier blanc initiée par TWEAK. Mais malgré ces améliorations, la taille du codage dans les espaces d'états avec frame-axiomes explicatifs reste inférieure et ses performances par rapport à la résolution sont meilleures.

Représentation des actions

La *représentation classique* des actions dans ces codages est de la forme $A(c_1, c_2, ..., c_n, i)$, où A est un symbole d'action, $c_1, c_2, ..., c_n$ sont des symboles de constantes pour tous les paramètres de l'action A, et i est le niveau dans lequel peut être appliquée l'action. [Ernst, Millstein et Weld 1997] proposent trois autres méthodes de représentation des actions, données ici par ordre décroissant du nombre maximal (théorique) de variables :

- *Partage simple* : chaque action n-aire est remplacée par n actions. Ainsi, la variable $A(c_1, c_2, ..., c_n, i)$ devient la conjonction des variables $A\text{arg1}(c_1, i)$, $A\text{arg2}(c_2, i)$, ..., et $A\text{arg}n(c_n, i)$. La variable $A\text{arg1}(c_1, i)$ permet de représenter le fait que plusieurs instances de l'action A utilisent la constante c_1 comme premier paramètre.
- *Partage augmenté* : les constantes sont désolidarisées des actions qui les utilisent. Ainsi, la variable $A(c_1, c_2, ..., c_n, i)$ devient la conjonction des variables $A(i)$, $\text{arg1}(c_1, i)$, $\text{arg2}(c_2, i)$, ..., et $\text{arg}n(c_n, i)$. Une action B utilisant c_1 comme premier argument sera la conjonction des variables $B(i)$, $\text{arg1}(c_1, i)$, ...
- *Représentation binaire* : chaque action étant numérotée, une action est représentée par les variables qui représentent les chiffres de ce numéro en notation binaire. Ainsi, s'il y a quatre actions, la première est représentée par $\neg\text{bit1}(i) \wedge \neg\text{bit2}(i)$, la seconde par $\neg\text{bit1}(i) \wedge \text{bit2}(i)$, et ainsi de suite.

Ces trois représentations non classiques des actions permettent la factorisation de certains axiomes du codage : par exemple pour la représentation en partage augmenté, $\neg(\text{arg1}(c_1, i) \wedge \text{arg2}(c_1, i))$ représente le fait

que toutes les actions qui ont la constante c_1 comme premier argument sont mutuellement exclusives avec toutes les actions qui ont la constante c_2 comme second argument.

[Ernst, Millstein et Weld 1997] étudient huit codages, pour les quatre représentations des actions et les deux représentations de frame-axiomes (classiques et explicatifs). En pratique, les codages les plus restreints en nombre de variables et de clauses après l'application de simplifications (typage, propagation unitaire, élimination de littéraux purs...) sont les codages avec frame-axiomes explicatifs et représentation d'action classique et partage simple. Ceci va à l'encontre de la complexité théorique de ces codages ; de plus, les performances des prouveurs SAT sur ces codages sont généralement les meilleures.

3.2.4. Atteignabilité et pertinence

Utilisation du graphe de planification

Le graphe de planification n'est pas seulement une méthode performante pour la représentation compacte de l'espace d'états d'un problème de planification ; ce n'est pas non plus seulement une compilation sous forme de problème de satisfaction de contraintes dynamiques ; c'est aussi (et surtout, à la lumière des récents travaux sur la planification dans les espaces d'états [Bonet et Geffner 1998] [Bonet et Geffner 1999]) une technique efficace d'estimation de l'atteignabilité des fluents et de l'applicabilité des actions pour un problème. Lorsqu'un fluent apparaît pour la première fois à un niveau n du graphe de planification, c'est parce qu'on ne peut pas atteindre un état qui le contienne après l'application d'un plan quelconque comportant $n-1$ ensembles d'actions parallèles ; et si une action apparaît pour la première fois à un niveau n, c'est parce qu'elle ne peut être appliquée dans aucun état obtenu après l'application d'un plan quelconque comportant $n-2$ ensembles d'actions parallèles. Le calcul des exclusions mutuelles entre fluents et actions permet une optimisation de cette estimation.

Les travaux sur la planification par satisfiabilité que nous avons répertoriés jusqu'à présent créent la base de clauses pour un problème à partir de toutes les instanciations possibles des opérateurs du domaine, avec éventuellement des optimisations simples comme l'analyse de types, effectuée par exemple dans [Ernst, Millstein et Weld 1997]. Après avoir été suggérée dans [Kautz et Selman 1996], l'idée de créer un planificateur qui remplace la procédure d'extraction de Graphplan par la transformation du graphe sous forme de base et la recherche d'un modèle de cette base est donc apparue avec les planificateurs C-SATPLAN [Baioletti, Marcugini et Milani 1998b] [Baioletti, Marcugini et Milani 1998c] et surtout BLACKBOX [Kautz et Selman 1998b] [Kautz et Selman 1999]. BLACKBOX permet l'utilisation de la procédure originelle d'extraction de Graphplan, des prouveurs SAT complets SATZ [Li et Anbulagan 1997] et RELSAT [Bayardo et Schrag 1997], du prouveur stochastique WALKSAT [Selman, Kautz et Cohen 1994], le tout avec de multiples options ; notamment l'exécution séquentielle de ces prouveurs sur une certaine durée, avec leurs différentes options de fonctionnement. Par exemple, il peut être intéressant de tester WALKSAT pendant une durée limitée pour essayer de trouver rapidement une solution, puis de lancer SATZ ou RELSAT en cas d'échec pour prouver qu'il n'en existe pas.

Concernant la pertinence des fluents, c'est-à-dire leur utilité potentielle par rapport aux buts du problème, on trouve dans [Do, Srivastava et Kambhampati 2000] une méthode qui permet de calculer, à partir du graphe de planification, des exclusions mutuelles entre fluents et actions correspondant à leur pertinence respective. Cette méthode avait été suggérée dans [Kambhampati, Parker et Lambrecht 1997] mais n'avait jamais été implémentée. Ces exclusions mutuelles (appelées bmutex pour "backward mutex") sont propagées en arrière dans le graphe à partir des buts, l'idée centrale étant que deux actions sont bmutex si elles ont la même utilité par rapport à un ensemble de fluents ; c'est-à-dire que pour produire un ensemble de fluents, il n'est utile d'employer qu'une seule des deux actions. Ainsi, si on emploie une des deux actions et qu'on exclut l'autre, des propagations vont pouvoir être effectuées par le prouveur SAT. Les tests réalisés sur quelques problèmes montrent que les performances obtenues par la résolution avec le prouveur RELSAT sont améliorées de façon modérée (au maximum environ 6 fois plus vite), alors qu'elles sont dégradées avec l'utilisation du prouveur SATZ. L'intérêt de ces bmutex n'apparaît donc pas clairement. Mais ce ne sont là que des résultats préliminaires, et la recherche sur l'extraction d'autres formes de pertinence des fluents et des actions peut se révéler intéressante.

Algorithmes spécialisés

Dans sa version originelle, le graphe de planification fournit des informations d'atteignabilité calculées uniquement à partir d'exclusions mutuelles binaires. Brafman [Brafman 1999] [Brafman 2001] généralise ce calcul par l'algorithme Reachable-k, qui recherche pour chaque niveau les ensembles de fluents et d'actions

de cardinalité inférieure ou égale à k tels qu'à ce niveau aucun état ne puisse contenir un de ces ensembles de fluents, et aucun de ces ensembles d'actions ne soit exécutable sur un de ces états. Ainsi, Reachable−2 calcule exactement les mêmes informations d'atteignabilité que la construction du graphe de planification.

L'objectif de Brafman est de comparer en théorie et en pratique le pouvoir d'élimination de Reachable−k, qui tient compte de la structure du problème du point de vue de la planification, avec les techniques de simplification dans le cadre de la satisfiabilité comme la propagation unitaire et la résolution de clauses binaires[2] (plus généralement la k−clause résolution), qui perdent la structure du problème du point de vue de la planification. En effet, Reachable−1 correspond intuitivement à une sorte de propagation unitaire, et le calcul des exclusions mutuelles serait d'après [Kautz et Selman 1999] une forme limitée de résolution de clauses binaires.

On peut noter qu'il s'agit là des premières études théoriques sur ce sujet, qui a déjà été abordé d'un point de vue pratique notamment dans [Ernst, Millstein et Weld 1997] à propos des différentes représentations d'actions. Brafman démontre plusieurs résultats intéressants[3] :

- Dans le contexte du codage linéaire [Kautz et Selman 1992], Reachable−1 élimine plus de données que la propagation unitaire.
- Dans le contexte du codage linéaire, Reachable−2 et la résolution de clauses binaires ne sont pas comparables. Brafman exhibe l'exemple d'un compteur à 4 bits où Reachable−2 élimine des données que la résolution n'élimine pas et inversement.
- Dans le contexte du codage basé sur Graphplan [Kautz, McAllester et Selman 1996], Reachable−1 et la propagation unitaire éliminent les mêmes données si l'on ignore les contraintes d'indépendance entre actions. Si l'on en tient compte, la propagation unitaire élimine plus de données.
- Dans le contexte du codage basé sur Graphplan à partir de $k \geq 2$, Reachable−k et la k−clause résolution ne sont pas comparables.

Ces résultats théoriques sont confirmés par des tests sur des problèmes du domaine Logistics, du Monde des cubes et des tours de Hanoi : Reachable−1 élimine plus de données que la propagation unitaire dans tous les domaines pour le codage linéaire (le même nombre pour le codage basé sur Graphplan), et Reachable−2 élimine plus de données que la résolution de clauses binaires dans le domaine Logistics mais en élimine moins dans les deux autres domaines, pour les deux codages.

Brafman propose aussi l'algorithme Relevant−k, qui est le symétrique de Reachable−k pour la pertinence des données. La comparaison est effectuée avec la k−clause résolution en tenant compte du but du problème, mais pas de l'état initial. Les résultats théoriques sont similaires : Relevant−1 élimine plus d'informations que la résolution unitaire pour les deux codages (avec certaines restrictions, elle élimine exactement les mêmes informations), tandis que pour $k \geq 2$, aucune méthode n'est meilleure que l'autre. Par contre d'un point de vue pratique, Relevant−2 élimine plus de données que la résolution de clauses binaires pour les domaines Logistics, Monde des cubes et tours de Hanoi.

3.2.5. Spécificités du prouveur

Bien que l'objectif originel ait été d'utiliser directement les prouveurs SAT implémentés dans le cadre de la recherche en satisfiabilité, on trouve quelques travaux s'intéressant à la méthode de recherche suivant deux axes principaux.

Adaptation du prouveur à la planification

Partant de la considération que les valeurs des symboles représentant les fluents sont induites par les valeurs que l'on donne aux symboles représentant les actions, [Giunchiglia, Massarotto et Sebastiani 1998] modifient le prouveur TABLEAU [Crawford et Auton 1996] afin que les seuls choix effectués dans la procédure de Davis et Putnam portent sur les symboles représentant les actions. Les valeurs des symboles représentant les fluents sont alors déduites par simple propagation unitaire. Les performances du prouveur SAT ainsi modifié sont comparées avec la version originale, sur 6 des codages proposés dans [Ernst, Millstein et Weld 1997]

[2] En tenant compte uniquement de l'état initial du problème et pas du but puisqu'il s'agit de calculer les informations d'atteignabilité.

[3] Les résultats théoriques de [Brafman 1999] et [Brafman 2001] diffèrent sensiblement ; nous supposerons que ceux de [Brafman 2001] sont les résultats corrects, puisque la publication de [Brafman 2001] est postérieure à celle de [Brafman 1999] et fournit toutes les preuves.

pour une série de divers problèmes. Les améliorations apportées par cette techniques sont souvent très importantes : plus de 10000 fois moins de temps pour certains problèmes.

Le prouveur MODOC [van Gelder et Okushi 1999] est basé sur des techniques de résolution en chaînage arrière, lui permettant de se focaliser sur les clauses qui correspondent aux buts du problème. De plus, il est capable de déterminer les clauses qui ne sont pas pertinentes pour la résolution du problème par rapport à une interprétation partielle, ce qui lui permet de compléter cette interprétation partielle pour en faire un modèle de la base de clauses. Cette approche est particulièrement adaptée aux problèmes de planification, qui ont une structure particulière due au fait que l'on cherche à prouver un ensemble de buts et que lorsqu'on possède un plan-solution, il est facile de trouver un modèle de la base de clauses correspondant au problème. Les performances de MODOC sont compétitives avec celles du prouveur incomplet WALKSAT [Selman, Kautz et Cohen 1994], mais sont beaucoup moins bonnes que celles du prouveur complet SATO [Zhang 1997] qui possède entre autres une implémentation extrêmement efficace de la propagation unitaire.

Algorithmes de planification basés sur Davis et Putnam

Lorsqu'on crée une base de clauses à partir d'un problème de planification, on peut perdre la structure de ce dernier. [Rintanen 1998] propose un algorithme permettant d'éviter cette transformation, en appliquant une méthode inspirée de la procédure de Davis et Putnam directement sur les actions et les fluents de base que l'on peut construire à partir de la description STRIPS d'un problème, pour une longueur de plan parallèle donnée.

Deux tableaux à deux entrées sont créés : un pour les actions et un autre pour les fluents. Une valeur du tableau représente la valeur de vérité de l'action ou du fluent pour un certain niveau du plan parallèle. Des règles de propagation sont définies, correspondant à la propagation des valeurs effectuée par la procédure de Davis et Putnam sur des clauses issues d'un codage parallèle dans les espaces d'états du problème avec frame-axiomes explicatifs : si la valeur d'un opérateur *o* est assignée à *vrai* à un niveau *i*, alors la valeur des préconditions de *o* est assignée à *vrai* au niveau $i-1$, la valeur des ajouts est assignée à *vrai* au niveau *i*, la valeur des retraits est assignée à *faux* au niveau *i*, et enfin, la valeur des actions qui retirent une précondition de *o* ou dont *o* retire une précondition au niveau *i* est assignée à faux.

D'autres règles de propagation sont appliquées, correspondant aux effets de la propagation unitaire dans un prouveur SAT : si un fluent a la valeur *vrai* au niveau *i* et que tous les opérateurs qui ajoutent ce fluent ont la valeur *faux* au niveau *i* sauf un, alors on peut assigner à ce dernier la valeur *vrai*. De plus, si un fluent a la valeur *faux* à un niveau *i*, les action qui ajoutent ce fluent prennent la valeur *faux* au niveau *i*. Ces règles de propagation sont complétées par un calcul simple d'invariants binaires à partir de l'état initial.

L'algorithme qui implémente ces règles de propagation choisit une action pour laquelle aucune valeur n'a encore été assignée à un certain niveau, et comme dans la procédure de Davis et Putnam, tente dans un premier temps de lui assigner une valeur, puis la valeur opposée. Le choix de cette action et de la valeur est effectué par une fonction heuristique inspirée de l'heuristique de SATZ [Li et Anbulagan 1997] : l'action et la valeur choisies sont celles qui permettront la plus forte réduction du nombre de propositions n'ayant pas encore reçu de valeur après propagation. Les performances relatées sont largement supérieures à celles de Graphplan, mais la plupart du temps inférieures à celles de SATZ et WALKSAT. [Rintanen 1998] observe aussi que l'heuristique qu'il utilise est moins performante que celle de SATZ, puisqu'elle ne calcule pas la possibilité d'effectuer un maximum de propagations unitaires au prochain noeud, ce que fait SATZ par rapport à la taille des clauses après propagation.

Une amélioration de cette méthode est proposée dans le planificateur DPPlan [Baioletti, Marcugini et Milani 2000]. Le principal avantage est l'utilisation du graphe de planification comme structure de base, au lieu des deux tableaux à double entrée de [Rintanen 1998]. Ainsi, on profite de toutes les informations présentes dans le graphe, l'absence de fluents ou d'actions à un niveau étant équivalente à la valeur *faux* pour les propositions correspondantes. Le calcul des exclusions mutuelles entre actions et fluents apporte de plus une information plus précise et complète que le calcul d'invariants binaires de [Rintanen 1998]. Une autre amélioration porte sur les règles de propagation, plus complètes et nombreuses que dans [Rintanen 1998]. Enfin, une liste de buts est maintenue et permet a priori de guider la recherche plus précisément que l'heuristique de [Rintanen 1998], bien qu'aucune comparaison ne soit effectuée : les heuristiques employées dans DPPlan concernent des propriétés locales des noeuds du but, comme le nombre de noeuds d'actions produisant ces buts n'ayant pas encore été assignés. Une propriété importante de cette liste de buts est que lorsqu'elle est vide, on sait que l'on a trouvé une solution : c'est le même principe qui est utilisé dans le prouveur MODOC [van Gelder et Okushi 1999]. Cette fois-ci, sur les quelques tests effectués, DPPlan semble soutenir la comparaison avec SATZ.

3.2.6. Utilisation de connaissances dépendantes du domaine

Si l'utilisation de connaissances dépendantes du domaine (définies par l'utilisateur) n'apparaît pas comme évidente dans le cadre de la recherche dans les graphes de planification (elle n'a, à notre connaissance, jamais été faite), elle semble en revanche très naturelle pour les approches basées sur la satisfiabilité qui disposent de toute l'expressivité de la logique. En effet, les connaissances sur le domaine peuvent généralement s'exprimer sous forme de contraintes, codables directement par des clauses qu'il suffit d'ajouter à la base de clauses représentant un problème. Ces nouvelles clauses vont permettre d'effectuer des coupures supplémentaires dans l'arbre de recherche, éliminant automatiquement les solutions jugées indésirables. Le problème consiste donc d'une part à représenter les connaissances sur le domaine sous une forme utilisable facilement (par exemple sous forme déclarative), et d'autre part à transformer ces connaissances en clauses pouvant être ajoutées à la base de clauses.

Le langage PCDL

Une première approche décrite dans [Baioletti, Marcugini et Milani 1998b] puis dans [Baioletti, Marcugini et Milani 1998c] permet la description des connaissances grâce au langage PCDL ("Planning Constraints Description Language"). Il s'agit d'un langage basé sur une logique du premier ordre, défini originellement pour la planification dans les espaces de plans partiels [Baioletti, Marcugini et Milani 1998a]. L'objectif initial était de compiler directement des formules PCDL dans la description d'un domaine, en modifiant les opérateurs ou au besoin en en créant de nouveaux. Par exemple, pour spécifier que l'on doit toujours utiliser un opérateur *A* avant un opérateur *B*, le compilateur ajoute un nouveau fluent dans les ajouts de *A* et dans les préconditions de *B*. On peut aussi spécifier que cette contrainte ne doit être respectée que si certaines conditions sont vérifiées, comme par exemple des contraintes d'égalité entre les variables de *A* et les variables de *B*. Cette contrainte de précédence permet par exemple de spécifier, dans un domaine de transport d'objets par avions, que le chargement d'un objet dans un avion doit obligatoirement être effectué avant le décollage de cet avion, interdisant ainsi à un avion de voyager sans cargaison.

Pour utiliser pleinement les possibilités du langage PCDL, il faut que le langage de description du domaine soit le plus riche possible ; c'est pourquoi PCDL a été proposé pour le planificateur UCPOP qui dispose du langage ADL. Les codages sous forme de bases de clauses n'ayant été étudiés à ce moment-là que pour une version simple du langage STRIPS, le langage PCDL a été détourné de son objectif premier (la compilation de contraintes dans la description d'un domaine) pour permettre la compilation des contraintes sous forme de clauses dans le planificateur C−SATPLAN [Baioletti, Marcugini et Milani 1998b]. Il semblerait que l'expressivité de PCDL et les performances de C−SATPLAN soient plutôt bonnes, mais aucun résultat expérimental ni aucune comparaison avec d'autres planificateurs ne viennent confirmer ces affirmations.

La logique temporelle

Il existait déjà un langage permettant de représenter des connaissances sous forme déclarative, très expressif et facile d'utilisation, basé sur une logique temporelle du premier ordre, implémenté dans le planificateur TLPlan [Bacchus et Kabanza 1995] [Bacchus et Kabanza 2000]. Ce dernier effectue une recherche en chaînage avant dans les espaces d'états, en vérifiant la validité à chaque état de formules de ce langage qui représentent les connaissances sur le domaine. TLPlan fournit des solutions de bonne qualité et a démontré son efficacité dans de nombreux domaines. C'est pourquoi, après une étude préliminaire [Kautz et Selman 1998a] montrant que le contrôle d'information est faisable dans les approches par satisfiabilité et améliore énormément les performances, on s'est posé la question de l'utilisation du même langage de représentation des connaissances que TLPlan dans les approches par satisfiabilité. [Huang, Selman et Kautz 1999] montrent que les connaissances que l'on peut représenter dans le langage de TLPlan peuvent être classées en trois catégories :

- la première n'implique que des connaissances statiques basées sur l'état initial et le but qui servent à éliminer directement des actions avant même de les utiliser dans la base de clauses,
- la deuxième utilise des connaissances qui dépendent de l'état courant et peuvent être exprimées par ajout de clauses dans la base sans créer de nouvelles variables,
- la troisième utilise des connaissances qui dépendent de l'état courant et ne peuvent être exprimées par ajout de clauses dans la base qu'en créant des nouvelles variables.

Les deux premières catégories sont implémentées dans BLACKBOX [Kautz et Selman 1999] pour plusieurs domaines, et apportent une réduction significative du temps de recherche. La troisième catégorie semble être

inutilisable, car elle augmente la taille de la base de clause de manière trop importante. Malgré le bénéfice apporté par les deux premières catégories, [Huang, Selman et Kautz 1999] déplorent le fait que TLPlan soit toujours plus efficace que BLACKBOX en dépit, disent-ils, des techniques de recherche sophistiquées des prouveurs SAT :

> (...) we were somewhat disappointed that Blackbox with control was not faster than the TLPlan approach, given the more sophisticated search of the SAT solvers. (...)

Ils se consolent en remarquant que les plans fournis par BLACKBOX ont un plus haut degré de parallélisme que ceux fournis par TLPlan, malgré un nombre d'actions largement supérieur : de l'ordre de 50% dans le domaine Logistics. Ce domaine permet le déplacement d'objets ou de personnes entre plusieurs aéroports ou à l'intérieur d'une même ville ; si les 50% d'actions supplémentaires sont des vols en avion, l'affaire risque de ne pas être très rentable. Ceci pose la question de la pertinence de l'optimalité d'un plan par rapport au nombre de niveaux d'actions parallèles, et même par rapport au nombre d'actions. Les niveaux d'un plan ne correspondent pas à des durées : si l'on veut optimiser le temps d'exécution d'un plan, il faut gérer la durée des actions de manière explicite. L'optimalité en nombre de niveaux dans le domaine Logistics par exemple n'a pas vraiment de signification par rapport au monde réel qui est représenté.

3.2.7. Gestion des ressources

Une approche de la planification par satisfiabilité permettant de gérer des ressources métriques est implémentée dans le planificateur LPSAT [Wolfman et Weld 1999]. Le principe consiste à résoudre un problème constitué par une base de clauses et un ensemble de contraintes sous forme d'équations linéaires et d'inégalités, ces contraintes étant déclenchées par l'assignation de la valeur de vérité *vrai* à certaines variables présentes dans des clauses de la base. Par exemple, on peut associer à une variable propositionnelle correspondant au déplacement d'un véhicule une contrainte précisant que la quantité de carburant disponible doit être supérieure à une certaine valeur. Lorsque la valeur de vérité de cette variable passe à *vrai*, il faut s'assurer que cette contrainte est vérifiée, ainsi que les autres contraintes dépendant de cette variable. Dans le cas contraire, un retour arrière dans la procédure de Davis et Putnam doit être effectué, même si la base de clauses en elle-même reste satisfiable. Le prouveur SAT doit donc être capable de détecter l'assignation des variables associées à des contraintes, d'appeler une procédure de vérification de ces contraintes (le moins souvent possible, car l'opération de vérification des contraintes est très coûteuse), et enfin, d'utiliser à bon escient les retours effectués par cette procédure.

Le planificateur LPSAT utilise à cet effet une version modifiée de RELSAT [Bayardo et Schrag 1997] conjointement avec le résolveur de contraintes arithmétiques linéaires CASSOWARY [Borning et al. 1997]. Le choix de RELSAT est motivé en particulier par ses capacités de retour arrière intelligent et d'apprentissage, dans le but de limiter les appels au résolveur de contraintes. Le choix de CASSOWARY est motivé par le fait qu'il a été spécialement conçu pour répondre rapidement lors de modifications mineures de l'ensemble des contraintes.

Les performances rapportées pour une version métrique du domaine Logistics semblent bonnes, notamment par rapport au planificateur ZENO [Penberthy et Weld 1994] qui ne résout même pas le problème le plus facile de cette série. La meilleure adaptation proposée de l'apprentissage de RELSAT à la gestion des contraintes procure une amélioration des performances de l'ordre d'un facteur 1000 pour certains problèmes.

Remarquons qu'il est difficile de juger des performances de ce type de planificateur, en raison du faible nombre d'implémentations et de benchmarks disponibles. De plus, les différences dans l'expressivité du langage de représentation de ces planificateurs augmentent cette difficulté d'évaluation (dans le cas présent, ZENO possède plus de potentialités d'utilisation que LPSAT).

4. La tendance actuelle : la recherche heuristique

4.1. Pourquoi ?

La planification peut être naturellement formulée comme un problème de recherche dans les espaces d'états : à partir d'un état initial, d'un ensemble d'actions qui représentent les transitions entre les états, et d'un but qui caractérise l'ensemble des états que l'on veut atteindre, on recherche une séquence d'actions dont

l'application sur l'état initial permet d'atteindre un des états buts. Pour cela, on dispose de tous les algorithmes de recherche dans les espaces d'états : largeur d'abord, profondeur d'abord, A*, IDA*...

Mais à cause de la nature spécifique des problèmes de planification, qui sont souvent composés de sous-problèmes que l'on peut résoudre indépendamment les uns des autres, les premières formulations de la planification se sont éloignées des méthodes de recherche traditionnelles pour l'élaboration d'algorithmes spécifiques prenant en compte cette particularité. Les premiers planificateurs tel STRIPS étant incapables de résoudre des problèmes extrêmement simples comme l'anomalie de Sussman, il est alors apparu nécessaire de s'intéresser aux interactions entre les sous-problèmes. Ont alors été créées les méthodes visant à protéger les sous-buts, faire régresser les actions dans le plan, retarder les prises de décision (ordonnancement des actions, protection des intervalles) pour réduire le facteur de branchement, et ainsi de suite jusqu'à la formalisation actuelles des méthodes de planification dans les espaces de plans partiels.

L'ironie dans cette succession de méthodes visant essentiellement à régler les problèmes d'interactions entre sous-buts est que finalement, on a abouti à formuler la planification comme un problème de recherche ; la différence étant que cette recherche est effectuée non pas dans les espaces d'états, mais dans les espaces de plans. Ces planificateurs sont donc certes capables de résoudre des problèmes simples, à condition d'utiliser des algorithmes de recherche complets, mais des problèmes un peu plus compliqués leur posent beaucoup de difficultés ; au point qu'une simple recherche en largeur d'abord dans les espaces d'états s'avère souvent plus efficace qu'un planificateur comme UCPOP (cf. [Vidal et Régnier 1999]), pourtant reconnu comme un "state-of-the-art planner"... L'idée de planifier dans les espaces d'états a donc été remise au goût du jour avec les planificateurs ASP [Bonet, Loerincs et Geffner 1997] puis HSP [Bonet et Geffner 1998], ce dernier s'étant avéré tout a fait compétitif avec les planificateurs basés sur la planification par compilation pendant la première compétition de planificateurs, lors de la conférence AIPS−98.

4.2. L'extraction automatique d'heuristiques...

Devant l'obligation d'utiliser des algorithmes de recherche pour un minimum de robustesse, il est devenu nécessaire d'améliorer leur efficacité par rapport à la planification, ce qui passe par l'utilisation d'heuristiques. En effet, les planificateurs les plus efficaces utilisant des connaissances sur le domaine effectuent une recherche en chaînage avant dans les espaces d'états, comme TLPLAN [Bacchus et Kabanza 2000] et TALplanner [Doherty et Kvarnström 1999] [Kvarnström, Doherty et Haslum 2000] [Kvarnström, Doherty et Haslum 2000]. Mais l'objectif restant la résolution de problèmes de planification de façon indépendante du domaine, des méthodes sont apparues visant à extraire automatiquement des heuristiques ; le but étant de réduire l'écart des performances avec la planification dépendante du domaine.

4.2.1. ... par relaxation du problème

Il est connu depuis longtemps que la résolution d'un problème relaxé fournit une heuristique admissible pour le problème originel. Ainsi, on trouve dans [Pearl 1983] une explication de l'heuristique de la distance de Manhattan pour le problème du taquin, basée sur la suppression de certaines préconditions des actions. Mais résoudre un problème relaxé n'est pas polynomial dans le cas général ; c'est pourquoi [Bonet, Loerincs et Geffner 1997] proposent une méthode polynomiale d'estimation de l'heuristique, pour le problème relaxé par la suppression des listes de retraits des actions. Cette heuristique estime la longueur d'un plan minimal qui produit un ensemble de fluents, en définissant le coût d'un fluent *f* à partir d'un état *s* par une fonction h_s qui ajoute 1 au minimum des coûts des ensembles de préconditions des actions qui produisent *f*. Le coût d'un ensemble de fluents est alors défini à l'aide d'une fonction d'agrégation des coûts des fluents qui le composent.

Le coût d'un fluent peut être défini récursivement, pour un état *s* et un fluent *f*, par :

$$h_s(f)=\begin{cases} 0 & \text{si } f\in s \\ i+1 & \text{si } min_{o\in O,\, f\in Add(o)}\left[H_s(Prec(o))\right]=i \\ \infty & \text{sinon} \end{cases}$$

La fonction H_S calcule le coût d'un ensemble de fluents. Plusieurs possibilités sont offertes ; l'heuristique employée dans ASP [Bonet, Loerincs et Geffner 1997], HSP [Bonet et Geffner 1998], HSPr [Bonet et Geffner 1999] et HSP2 [Bonet et Geffner 2001] consiste à additionner le coût de chacun des fluents de l'ensemble :

$$H_s^+(F)=\sum_{f\in F} h_s(f)$$

Cette heuristique additionne la longueur minimale des plans nécessaires à l'obtention de chacun des fluents, sans tenir compte des interactions positives entre les fluents (c'est-à-dire le fait qu'un fluent produit par une action pour un certain sous-but peut servir à l'obtention d'un autre sous-but). Elle n'est donc pas admissible, puisqu'elle surestime largement le nombre d'actions nécessaires pour résoudre le problème relaxé. Néanmoins, elle a prouvé son efficacité lors des compétitions de planificateurs d'AIPS−98 et AIPS−2000, en particulier face aux planificateurs basés sur Graphplan et SATPLAN. Une possibilité pour la rendre admissible serait de calculer le maximum des coûts de chacun des fluents :

$$H_s^{\max}(F) = \max_{f \in F} \left[h_s(f) \right]$$

Mais cette dernière heuristique est très peu informative, et [Haslum et Geffner 2000] proposent plusieurs heuristiques admissibles pour la planification optimale en nombre d'actions.

4.2.2. ... à partir du graphe de planification

Une possibilité pour améliorer l'heuristique additive est proposée dans [Hoffmann 2000] et [Hoffman et Nebel 2000] pour le planificateur FF ("Fast Forward"), qui s'est montré extrêmement performant lors de la compétition d'AIPS−2000. L'idée est basée sur le fait que la résolution du problème relaxé par Graphplan est polynomiale : puisque les retraits des actions ne sont pas considérés, le graphe de planification ne contient aucune exclusion mutuelle, donc la procédure d'extraction de la solution n'effectue aucun retour arrière. Le coût d'un ensemble de fluents est alors constitué par la somme des actions du plan parallèle qui produit ces fluents. Cette heuristique peut être calculée de façon très efficace ; l'essentiel du temps de construction d'un graphe de planification provenant en effet du calcul des exclusions mutuelles. Elle prend en compte les effets positifs des actions, contrairement à l'heuristique additive qui suppose que tous les sous-buts sont indépendants.

Une possibilité pour prendre en compte en partie les effets négatifs des actions est proposée pour le planificateur AltAlt [Nguyen et Kambhampati 2000] [Nguyen et Kambhampati 2001], par le calcul complet du graphe de planification, c'est-à-dire prenant en compte les exclusions mutuelles. Les fluents et les actions apparaissent plus tardivement que dans le graphe fourni par le problème relaxé, ce qui permet d'affiner l'heuristique de FF. Plusieurs heuristiques sont proposées ; la plus performante fait intervenir le niveau d'apparition des fluents pour calculer la longueur d'un plan sans prendre en compte les retraits des actions, et fait aussi intervenir le niveau de disparition des exclusions mutuelles entre fluents afin d'ajuster l'estimation de la longueur du plan, ce qui est une manière indirecte de prise en compte des retraits des actions.

Cette heuristique, baptisée heuristique adjsum2M, est définie pour un ensemble de fluents *F* par :

$$H_{adjsum2M}(F) = H(F) + \Delta_{max}(F)$$

$H(F)$ est une estimation de la longueur d'un plan produisant les fluents de *F* sans prendre en compte les retraits des actions, et $\Delta_{max}(F)$ représente l'écart maximal entre le niveau de disparition d'une mutex et le niveau maximal d'apparition des fluents de la mutex.

Pour calculer $H(F)$, il faut dans un premier temps choisir une action *a* qui produit un fluent de *F*, ce fluent étant choisi parmi ceux qui ont le plus haut niveau d'apparition dans le graphe (*niv*(*f*) dénote le niveau d'apparition d'un fluent) :

$$a \in \left\{ a \mid \exists f \in \left(F \cap Add(a) \right),\ \ niv(f) = \max_{p \in F} \left[niv(p) \right] \right\}$$

On fait ensuite régresser l'ensemble de fluents *F* par l'action *a*, et on ajoute 1 à l'estimation de la longueur d'un plan produisant ce nouvel ensemble de fluents :

$$H(F) = 1 + H\left(F + Prec(a) - Add(a) \right)$$

Ce processus s'arrête lorsque l'on atteint les fluents qui appartiennent à l'état initial, qui ont un coût de 0. Pour calculer $\Delta_{max}(F)$, on calcule pour chaque paire {*p, q*} de fluents de *F* le premier niveau dans lequel les deux fluents ne sont pas mutuellement exclusifs (dénoté par *Niv*({*p, q*}), auquel on soustrait le niveau d'apparition maximum de *p* et de *q*. Si les deux fluents ne sont jamais mutuellement exclusifs, le résultat sera 0, c'est à dire que leur production simultanée ne pose pas de problème. Sinon, le résultat final sera le maximum de ces écarts, qui est une mesure de la difficulté d'établir simultanément tous les fluents de *F*. [Nguyen et Kambhampati 2001] proposent aussi de calculer la somme plutôt que le maximum, mais les mutex sont souvent interdépendants et cette mesure surestime donc trop la difficulté réelle d'obtenir un ensemble de fluents.

Le calcul de $\Delta_{max}(F)$ est défini par :

$$\Delta_{max}(F) = \max_{p,q \in F} \left[Niv(\{p,q\}) - max\left[niv(p), niv(q) \right] \right]$$

Cette heuristique rend le planificateur AltAlt compétitif avec HSP2 [Bonet et Geffner 2001] et FF [Hoffmann 2000], malgré une différence fondamentale dans l'utilisation de ces heuristiques.

4.3. Les algorithmes de recherche...

4.3.1. ... en chaînage avant dans les espaces d'états

L'utilisation sans doute la plus naturelle de ces heuristiques, si l'on pense à la performance des planificateurs dépendants du domaine, est la planification en chaînage avant dans les espaces d'états. Elle a été proposée par [Bonet, Loerincs et Geffner 1997] pour le planificateur ASP, puis plus tard HSP [Bonet et Geffner 1998] et HSP2 [Bonet et Geffner 2001]. Le principe est de calculer l'heuristique pour chaque noeud, et d'utiliser cette estimation par un algorithme de recherche heuristique comme une variante de LRTA* [Korf 1990] pour ASP, une variante de hill-climbing pour HSP, ou encore un algorithme meilleur-d'abord comme WA* pour HSP2. L'avantage de ce dernier, tel qu'il est décrit dans [Pearl 1983], est sa complétude : il conserve tous les noeuds parcourus pour les utiliser en cas de retour arrière. La différence par rapport à un algorithme A* est qu'il pondère l'heuristique par une constante $W \geq 1$, ce qui lui permet d'arriver plus facilement au but avec une dégradation limitée de la solution en terme de longueur du plan. Son utilisation dans HSP2 montre que sur les benchmarks habituels, ses performances ainsi que la qualité de la solution sont très bonnes.

Le planificateur FF [Hoffmann 2000] [Hoffman et Nebel 2000] utilise aussi une variation de hill-climbing lui permettant d'échapper à des minimums locaux. Cet algorithme restant néanmoins incomplet, le planificateur FF est capable de basculer sur une recherche complète WA* si aucune solution n'est trouvée rapidement.

4.3.2. ... en chaînage arrière dans les espaces d'états

Le principal inconvénient de HSP est le recalcul de l'heuristique pour chacun des noeuds parcourus. Ainsi, HSP résolvait plus de problèmes que les autres planificateurs lors de la compétition d'AIPS−98, mais avec des temps de résolution nettement supérieurs. Après une amélioration substantielle de l'implémentation du calcul de l'heuristique ainsi qu'un changement d'algorithme dans HSP2, les performances en chaînage avant se sont nettement améliorées.

Néanmoins, l'idée de ne calculer l'heuristique qu'une seule fois et de l'utiliser en chaînage arrière est apparue avec le planificateur HSPr [Bonet et Geffner 1999], qui utilise le même algorithme WA* que HSP2. Le gain de temps de calcul pour l'heuristique permet de très bonnes performances en comparaison avec HSP2, dans les domaines Logistics et Gripper [Bonet et Geffner 2001]. HSP2 est cependant plus rapide que HSPr dans le domaine des tours de Hanoi, et donne souvent des solutions de meilleure qualité dans les autres domaines.

Le planificateur AltAlt [Nguyen et Kambhampati 2000] utilise un algorithme de recherche en chaînage arrière basé sur celui de HSPr. Mais comme il calcule une heuristique beaucoup plus informée que HSPr, il a de meilleures performances que ce dernier dans sa première version [Bonet et Geffner 1999]. Il faudrait cependant le comparer avec la version de [Bonet et Geffner 2001] : on s'aperçoit en effet que dans [Bonet et Geffner 2001], HSPr est plus rapide que HSP2 dans le domaine du Gripper, alors que dans [Nguyen et Kambhampati 2000] c'est l'inverse.

En conclusion, il n'est pas encore clair qu'une méthode surpasse l'autre, si ce n'est que FF qui utilise le chaînage avant est la plupart du temps supérieur à tous les autres. Il a cependant un comportement un peu erratique, notamment dans le Monde des cubes dans lequel il ne peut trouver de solution à certains problèmes faciles alors qu'il en résout rapidement d'autres plus difficiles.

4.3.3. ... dans les espaces de plans partiels

Signalons enfin une réapparition des algorithmes de planification dans les espaces de plans partiels, avec le planificateur RePOP [Nguyen et Kambhampati 2001] qui utilise une heuristique basée sur le graphe de planification comme dans AltAlt, ainsi que plusieurs améliorations utilisant notamment les mutex calculées lors de la construction du graphe de planification. RePOP est basé sur l'implémentation en Common Lisp de

UCPOP [Penberthy et Weld 1992], et montre de bonnes performances en comparaison avec la version Lisp de AltAlt. Il est souvent un peu plus lent, mais génère des plans de meilleure qualité. On voit ainsi s'ouvrir un spectre important d'utilisations potentielles du calcul automatique d'heuristiques, avec la somme des travaux ayant été effectués en planification dans les espaces de plans partiels.

III. Planifier par recherche dans les graphes de planification

1. Le fonctionnement de Graphplan au travers d'un exemple

Dans cette section, nous illustrons le fonctionnement de Graphplan au travers du développement complet d'un exemple simple. Le lecteur ayant déjà une bonne connaissance du fonctionnement de Graphplan peut se reporter directement à la section 2, page 54.

Les entrées/sorties de Graphplan sont des plus classiques : à partir d'une description de type STRIPS des opérateurs, de l'état initial et du but à atteindre, il fournit un plan-solution. Dans le cas où ce dernier n'existe pas, la recherche se termine par un échec : Graphplan est sain, complet et son exécution se termine. La recherche d'un plan-solution se réalise essentiellement à l'aide de deux procédures :

1. **Expansion du graphe de planification** à l'aide de la description STRIPS des opérateurs et des fluents de l'état initial : chaque opérateur est instancié de toutes les façons possibles, produisant un nouvel ensemble de fluents ainsi que des contraintes sur les fluents et les actions. Cette opération est ensuite répétée à partir de ce nouvel ensemble de fluents, jusqu'à la satisfaction d'une condition nécessaire (mais non suffisante) de validité des buts.
2. **Extraction de la solution** par une recherche arrière dans le graphe de planification. Cette recherche s'effectue en tenant compte des actions, des fluents et surtout des contraintes entre les actions qui ont été détectées à l'étape d'expansion du graphe. Si aucun plan-solution n'est trouvé et que l'on n'a pas atteint le critère d'arrêt de l'algorithme (que nous détaillerons plus tard), on continue à développer le graphe de planification ; sinon il n'y a pas de solution.

1.1. Description du problème

Le problème que nous allons utiliser pour présenter le fonctionnement de Graphplan est un problème très simple inspiré d'une série de problèmes issus du domaine Logistics, fréquemment employé pour tester les performances des planificateurs. Ces problèmes ne sont pas très difficiles à résoudre pour un être humain (du moins ceux pouvant être résolus par un planificateur), mais leur très forte combinatoire met à rude épreuve beaucoup de planificateurs modernes.

Le principe du domaine Logistics est le suivant : divers objets (ou personnes) se trouvent dans des lieux variés, ainsi que plusieurs moyens de locomotion (avions, camions...). Un avion peut se déplacer d'un aéroport à l'autre, embarquer des objets, en débarquer d'autres, le tout sans contraintes de capacité. Les problèmes typiques de ce domaine consistent à déplacer certains objets (ou personnes) d'un endroit vers un

autre, de la façon la plus efficace possible. Un avion peut par exemple prendre plusieurs personnes à un endroit, faire escale à un autre endroit pour en déposer certaines, puis continuer sa route pour en déposer d'autres ailleurs.

Nous allons utiliser une version simplifiée de ce domaine (cf. Figure 4), permettant seulement de déplacer des personnes d'un aéroport à l'autre à l'aide d'avions. Le langage utilisé pour décrire ce domaine est le langage PDDL ("Planning Domain Definition Language"), défini récemment à l'occasion de la première compétition des planificateurs (conférence AIPS−98) [McDermott 1998].

```
(define (domain simple-Logistics)
  (:action EMBARQUER
         :parameters (?personne ?avion ?aeroport)
         :precondition (and (personne ?personne)
                            (avion ?avion)
                            (aeroport ?aeroport)
                            (present ?avion ?aeroport)
                            (present ?personne ?aeroport))
         :effect (and (dans ?personne ?avion)
                      (not (present ?personne ?aeroport))))
  (:action DEBARQUER
         :parameters (?personne ?avion ?aeroport)
         :precondition (and (personne ?personne)
                            (avion ?avion)
                            (aeroport ?aeroport)
                            (present ?avion ?aeroport)
                            (dans ?personne ?avion))
         :effect (and (present ?personne ?aeroport)
                      (not (dans ?personne ?avion))))
  (:action VOLER
         :parameters (?avion ?aeroport1 ?aeroport2)
         :precondition (and (avion ?avion)
                            (aeroport ?aeroport1)
                            (aeroport ?aeroport2)
                            (present ?avion ?aeroport1))
         :effect (and (present ?avion ?aeroport2)
                      (not (present ?avion ?aeroport1)))))
```

Figure 4 : Description du domaine simple−Logistics

Notre problème (cf. Figure 5) consiste simplement à embarquer une personne à l'aéroport de Blagnac pour la déposer à Orly. Nous ne disposons que d'un avion (initialement à Blagnac) et notre monde ne contient que ces deux aéroports et cette seule personne. Ce problème est loin d'exploiter toutes les possibilités de notre domaine, qui peut contenir autant d'aéroports, d'avions et de personnes que l'on veut.

```
(define (problem petit-probleme)
  (:domain simple-Logistics)
  (:init (personne Avrim)
       (avion airbus)
       (aeroport Blagnac) (aeroport Orly)
       (present Avrim Blagnac)
       (present airbus Blagnac))
  (:goal (and (present Avrim Orly))))
```

Figure 5 : Description du problème

Maintenant que nous disposons d'une description complète du domaine et du problème, telle que l'on peut la fournir à un planificateur, nous allons pouvoir nous livrer à certaines simplifications. En effet, lorsque nous parlons d'instancier les opérateurs de toutes les façons possibles, on imagine un grand nombre de fluents produits, un grand nombre d'actions... Or pour notre problème, ce n'est pas le cas. Par exemple, l'opérateur

EMBARQUER ne pourra être instancié que de deux façons possibles : le paramètre **?personne** sera toujours instancié par **Avrim**, le paramètre **?avion** par **airbus** et le paramètre **?aéroport** soit par **Blagnac**, soit par **Orly**. Ceci ne nous fait donc que deux actions possibles pour l'opérateur **EMBARQUER** : **(EMBARQUER Avrim airbus Blagnac)** et **(EMBARQUER Avrim airbus Orly)**. De même, les opérateurs **DEBARQUER** et **VOLER** n'auront chacun que deux instanciations possibles.

De la même façon, les fluents produits seront eux aussi très peu nombreux : **Avrim** est soit présent à **Blagnac**, soit présent à **Orly**, soit dans **airbus** ; ce dernier est présent soit à **Blagnac** soit à **Orly**. De plus, les fluents **(personne Avrim)**, **(avion airbus)**, **(aeroport Blagnac)** et **(aeroport Orly)** ne sont utiles que pour instancier les opérateurs. Ils n'entrent pas vraiment en compte dans la recherche de la solution. Nous pouvons donc utiliser une représentation propositionnelle très simple du problème, donnée dans la Figure 6.

```
FLUENTS :

      pab : (present Avrim Blagnac)
      pao : (present Avrim Orly)

      pvb : (present airbus Blagnac)
      pvo : (present airbus Orly)

      dav : (dans Avrim airbus)

ACTIONS (schéma des actions : preconditions → +ajouts −retraits) :

   EAB = (EMBARQUER Avrim airbus Blagnac) : pab pvb → +dav −pab
   EAO = (EMBARQUER Avrim airbus Orly)    : pao pvo → +dav −pao

   DAB = (DEBARQUER Avrim airbus Blagnac) : dav pvb → +pab −dav
   DAO = (DEBARQUER Avrim airbus Orly)    : dav pvo → +pao −dav

   VBO = (VOLER Blagnac Orly)      : pvb → +pvo −pvb
   VOB = (VOLER Orly Blagnac)      : pvo → +pvb −pvo
```

Figure 6 : Le problème simplifié et réduit à sa forme propositionnelle

1.2. L'expansion du graphe de planification

Le graphe de planification contient deux types de noeuds : les noeuds de fluents et les noeuds d'actions. Ces noeuds sont disposés en niveaux que nous numéroterons à partir de 0. Chaque niveau contient un ensemble de noeuds de fluents (que nous nommerons simplement fluents) et de noeuds d'actions (que nous nommerons actions). Le niveau 0 contient les fluents représentant les faits de l'état initial du problème, et ne contient pas d'action. Trois types d'arcs sont utilisés :

- Les arcs de préconditions : ils connectent les fluents d'un niveau i aux actions d'un niveau i+1, de façon à ce que les fluents du niveau i représentent les préconditions des actions du niveau i+1.
- Les arcs d'ajouts : ils connectent les actions d'un niveau aux fluents de ce même niveau. Les fluents ainsi connectés sont les effets positifs de l'action (ses ajouts, selon la dénomination STRIPS).
- Les arcs de retraits : ils connectent les actions d'un niveau aux fluents de ce même niveau. Les fluents ainsi connectés sont les effets négatifs de l'action (ses retraits, selon la dénomination STRIPS).

Deux types d'actions peuvent se trouver dans le graphe :

- Les actions provenant de l'instanciation des descriptions STRIPS des opérateurs, telles les actions de la Figure 6.
- Les actions permettant de gérer le problème du décor, que nous nommerons no-ops (pour non-opérateurs). Un no-op a comme précondition un fluent d'un niveau i, et comme unique effet l'ajout de ce même fluent au niveau i+1.

Enfin, des contraintes seront attachées à chaque noeud (fluents et actions). Ces contraintes sont des contraintes binaires entre deux fluents ou deux actions du même niveau, signifiant que les deux noeuds sont mutuellement exclusifs à ce niveau. Par exemple, si deux actions sont marquées comme étant mutuellement exclusives à un niveau, une seule de ces deux actions pourra éventuellement faire partie d'un plan-solution (pour ce niveau). Nous détaillerons plus tard la manière dont ces contraintes apparaissent et se propagent à travers le graphe de planification.

Revenons maintenant à notre problème. La première étape va consister à créer le graphe de planification initial, qui ne contient que les fluents du niveau 0, c'est-à-dire les faits de l'état initial. Les fluents d'un même niveau seront représentés verticalement, comme on le voit sur la Figure 7. Le graphe initial ne contient aucune contrainte, puisque les faits de l'état initial existent tous au début du processus de planification.

Figure 7 : Le graphe de planification initial

Comme le fluent représentant le but de notre problème (**pao**) n'est pas présent au niveau 0, nous devons continuer à construire le graphe. Nous allons donc créer le niveau 1 (représenté dans la Figure 8), de la façon suivante :

- Reproduction des fluents du niveau 0 dans le niveau 1 et création des no-ops entre les fluents du niveau 0 et ces mêmes fluents au niveau 1. Le no-op d'une action *A* sera noté N_*A*. Les no-ops seront représentés dans les schémas par des lignes horizontales grisées.
- Création dans le niveau 1 des actions applicables sur les fluents du niveau 0. Seules deux actions ont leurs préconditions présentes dans le niveau 0, et sont donc créées : **EAB** et **VB0**.
- Les fluents ajoutés par ces actions sont ensuite créés dans le niveau 1, s'ils n'existaient pas déjà : on crée donc les fluents **dav** et **pvo**.
- Les arcs entre les actions créées précédemment et leurs effets sont ensuite placés dans le graphe : les traits noirs pleins représentent les ajouts et les traits pointillés gris foncé représentent les retraits.

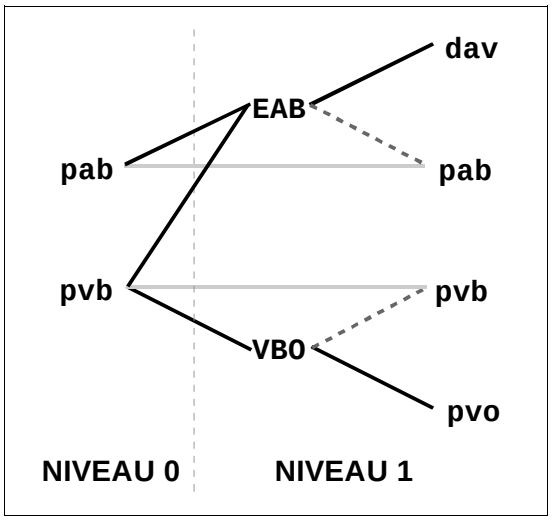

Figure 8 : Création du niveau 1

Après avoir étendu ainsi le graphe, nous devons maintenant nous intéresser aux contraintes d'exclusion mutuelle pouvant exister entre les noeuds du niveau 1 du graphe. Le principe essentiel pour bien comprendre ces contraintes est le suivant : *deux actions d'un niveau donné du graphe de planification ne peuvent faire*

partie d'un plan-solution que si elles peuvent être exécutées dans n'importe quel ordre en produisant des états résultants identiques.

Si l'on considère les deux actions présentes dans le niveau 1 du graphe de planification de notre exemple, on constate :

- Si l'on exécute l'action **EAB** dans l'état initial I = {**pab**, **pvb**}, on se retrouve dans l'état $E1$ = {**dav**, **pvb**}. Le fluent **dav** a été ajouté et le fluent **pab** a été retiré. On peut ensuite exécuter l'action **VBO** car son unique précondition **pvb** est dans $E1$, et on se retrouve alors dans l'état $E2$ = {**dav**, **pvo**}. Le fluent **pvo** a été ajouté et le fluent **pvb** a été retiré. Pour cet ordre entre les actions, il n'y a donc pas de problème : les actions **EAB** et **VBO** peuvent faire partie d'un même plan-solution.
- Si l'on exécute maintenant l'action **VBO** dans l'état initial I = {**pab**, **pvb**}, on se retrouve dans l'état $E1$ = {**pab**, **pvo**}. Par contre, on ne peut pas exécuter l'action **EAB** car sa précondition **pvb** ne se trouve pas dans l'état $E1$.

Les actions **EAB** et **VBO** ne peuvent donc pas faire partie du niveau 1 d'un plan-solution, puisque l'action **VBO** ne peut pas être exécutée avant l'action **EAB**. Il faut donc marquer ces deux actions comme étant mutuellement exclusives. Mais pour trouver toutes les contraintes, on ne vas pas chercher à vérifier l'applicabilité de tous les ordres possibles entre toutes les actions d'un même niveau. Dans notre exemple, c'est faisable car on n'a que peu d'actions ; mais dans le cas général, il faudrait vérifier toutes les combinaisons possibles de plusieurs actions, ce qui se révélerait très coûteux. Les paires d'actions mutuellement exclusives (les *mutex*) se calculent très simplement à partir des règles ci-dessous. Deux actions sont mutuellement exclusives à un niveau i si et seulement si :

1. *Effets inconsistants* : une action retire un fluent que l'autre action établit.
2. *Interférence* : une action retire un fluent qui était déjà présent au niveau i−1, et qui est une précondition de l'autre action.
3. *Préconditions inconsistantes* : une action a pour précondition un fluent qui est mutuellement exclusif avec une précondition de l'autre action.

En appliquant cette définition à notre exemple, on obtient :

- L'action **EAB** retire le fluent **pab** établi par l'action **N_pab**. La paire {**EAB**, **N_pab**} est donc une mutex.
- L'action **VBO** retire le fluent **pvb** établi par l'action **N_pvb**. La paire {**VBO**, **N_pvb**} est donc une mutex.
- L'action **VBO** retire le fluent **pvb** du niveau 1, qui existait déjà au niveau 0. Or ce fluent est une précondition de l'action **EAB**. La paire {**EAB**, **VBO**} est donc une mutex.

Signalons au passage qu'aucun mutex de type 3 (préconditions inconsistantes) n'a été trouvé, ce qui est normal car les fluents du niveau 0, qui correspondent au faits de l'état initial, ne forment bien sûr pas de mutex : ils existent tous ensembles dans l'état initial.

Après avoir calculé ces mutex entre actions, nous devons maintenant calculer les mutex entre fluents. Ce sont des paires de fluents d'un niveau i qui ne peuvent pas être présents dans un même état après l'exécution de certaines actions de ce même niveau. On voit ici par exemple que les fluents **dav** et **pab** ne peuvent pas se trouver dans un même état, puisque la seule façon d'obtenir le fluent **dav** est d'exécuter l'action **EAB**, qui retire le fluent **pab**. La règle de détection des mutex entre fluents est donc la suivante : deux fluents sont mutuellement exclusifs à un niveau i si et seulement si toutes les actions du niveau i qui établissent un des deux fluents sont mutuellement exclusives avec toutes les actions du niveau i qui établissent l'autre fluent.

En appliquant cette règle sur les fluents du niveau 1, on trouve :

- Les actions qui établissent le fluent **dav** (l'action **EAB**) sont mutuellement exclusives avec les actions qui établissent le fluent **pab** (l'action **N_pab**). La paire {**dav**, **pab**} est donc une mutex.
- Les actions qui établissent le fluent **dav** (l'action **EAB**) sont mutuellement exclusives avec les actions qui établissent le fluent **pvo** (l'action **VBO**). La paire {**dav**, **pvo**} est donc une mutex.
- Les actions qui établissent le fluent **pvo** (l'action **pab**) sont mutuellement exclusives avec les actions qui établissent le fluent **pvb** (l'action **N_pvb**). La paire {**pvb**, **pvo**} est donc une mutex.

La Figure 9 récapitule les mutex que nous venons de trouver, avec la notation suivante : $A - B, C, D$ signifie que les paires $\{A, B\}$, $\{A, C\}$ et $\{A, D\}$ sont des mutex. Le bien fondé de ces mutex apparaît

pleinement lorsqu'on se ramène à la forme des actions et des fluents en logique du premier ordre, dont elles sont issues :

- **dav** et **pab** forment une mutex signifie que **Avrim** ne peut être à la fois présent à l'aéroport de Blagnac et dans **airbus** (du moins dans notre monde simplifié !),
- **dav** et **pvo** forment une mutex signifie que **Avrim** ne peut pas être dans **airbus** si ce dernier est à **Orly** : l'action d'embarquer dans l'avion n'a pas pu se faire en même temps que le vol de l'avion (en un seul niveau),
- **pvb** et **pvo** forment une mutex signifie que **airbus** ne peut se trouver à la fois à **Blagnac** et à **Orly**,
- **EAB** et **N_pab** forment une mutex signifie que **Avrim** ne peut embarquer dans **airbus** tout en restant présent dans l'aéroport,
- **EAB** et **VBO** forment une mutex signifie que **Avrim** ne peut embarquer dans **airbus** si ce dernier, au même moment, vole vers **Orly**,
- **VBO** et **N_pvb** forment une mutex signifie que **airbus** ne peut à la fois voler vers **Orly** et rester à **Blagnac**.

```
Mutex entre fluents :
  dav – pab, pvo
  pvb – pvo

Mutex entre actions :
  EAB – N_pab, VBO
  VBO – N_pvb
```

Figure 9 : Les mutex du niveau 1

Le fluent représentant le but du problème (**pao**) n'est toujours pas présent au niveau 1 : nous devons donc continuer à étendre le graphe, par la construction du niveau 2 (voir la Figure 10). Nous allons donc répéter le processus que nous avons utilisé pour la création du niveau 1 :

- Reproduction des fluents du niveau 1 et création des no-ops associés.
- Création des actions applicables. Nous retrouvons les actions **EAB** et **VBO** puisque leurs préconditions sont dans le niveau 1 (grâce aux no-ops). Nous pouvons ensuite créer l'action **DAB** qui a pour préconditions les fluents **dav** et **pvb**, et l'action **VOB** qui a pour préconditions le fluent **pvo**. Les préconditions de l'action **DAO** se trouvent aussi dans le niveau 1 : les fluents **dav** et **pvo**. Mais ces deux fluents forment une mutex, comme nous l'avons vu précédemment. On ne crée donc pas l'action **DAO**.

  En effet, si on la créait, il y aurait un problème lors de la recherche arrière d'un plan-solution. Imaginons que lors de cette recherche, on essaye d'utiliser l'action **DAO** du niveau 2. Il faudrait alors montrer que les préconditions de cette action peuvent être établies depuis le niveau 0, en choisissant pour chacune de ces préconditions une action du niveau 1 qui l'établit. Or, les préconditions de **DAO** forment une mutex, c'est-à-dire que toutes les actions qui établissent un des deux fluents sont mutuellement exclusives avec toutes les actions qui établissent l'autre fluent. On ne pourrait donc pas trouver les deux actions recherchées et on en conclurait que l'on ne peut pas utiliser l'action **DAO** du niveau 2 ; alors autant ne pas l'insérer dans le graphe.
- Création des nouveaux fluents dus aux ajouts des deux nouvelles actions applicables. Ici, ces actions ne créent aucun nouveau fluent.
- Création des arcs d'ajout et de retrait. Si aucun nouveau fluent n'est apparu, il y a par contre deux nouveaux arcs d'ajout et deux nouveaux arcs de retrait. Ces arcs vont provoquer une augmentation très importante du nombre de mutex sur les actions.

Calculons maintenant les mutex sur les actions (pour la liste exhaustive des mutex, voir la Figure 11) :

1. Effets inconsistants : par exemple, l'action **DAB** retire le fluent **dav** que créent les actions **N_dav** et **EAB**, donc les paires {**DAB**, **N_dav**} et {**DAB**, **EAB**} sont des mutex. De même, {**VOB**, **VBO**} et {**VOB**, **N_pvo**} sont des mutex, etc.

2. Interférence : par exemple, l'action **VBO** retire le fluent **pvb**, qui existait déjà au niveau 1 et est une précondition des actions **DAB** et **EAB**. Les paires {**VBO**, **DAB**} et {**VBO**, **EAB**} sont des mutex.
3. Préconditions inconsistantes : par exemple, les fluents **dav** et **pvo** forment une mutex au niveau 1 donc les paires d'actions pour lesquelles une action a pour précondition un des deux fluents et l'autre action a pour précondition l'autre fluent forment des mutex : {**DAB**, **N_pvo**}, {**DAB**, **VOB**}, {**N_dav**, **N_pvo**}, {**N_dav**, **VOB**}, etc.

Après avoir calculé les mutex sur les actions, nous pouvons en déduire les mutex sur les fluents :

- {**dav**, **pab**} est toujours une mutex : le produit cartésien des actions qui produisent **dav** (l'ensemble {**EAB**, **N_dav**}) et des actions qui produisent **pab** (l'ensemble {**DAB**, **N_pab**}) est bien inclus dans les mutex sur les actions du niveau 2, que nous venons de calculer.
- {**pvb**, **pvo**} est toujours une mutex : le produit cartésien des actions qui produisent **pvb** (l'ensemble {**VOB**, **N_pvb**}) et des actions qui produisent **pvo** (l'ensemble {**VBO**, **N_pvo**}) est bien inclus dans les mutex sur les actions du niveau 2.
- {**dav**, **pvo**} n'est plus une mutex : l'action **N_dav** établit **dav**, l'action **VBO** établit **pvo** mais la paire {**N_dav**, **VBO**} n'est pas une mutex. Ceci se comprend bien, puisque le fait qu'**Avrim** soit dans **airbus** avant le décollage (grâce à l'action **N_dav**) est compatible avec le fait que **airbus** soit présent à **Orly** puisque ce vol s'est effectué après l'embarquement d'**Avrim**. Le fait que {**dav**, **pvo**} ne soit plus une mutex au niveau 2 est très important, comme on va le voir par la suite. Cela va en effet nous permettre de créer le fluent qui représente le but du problème.

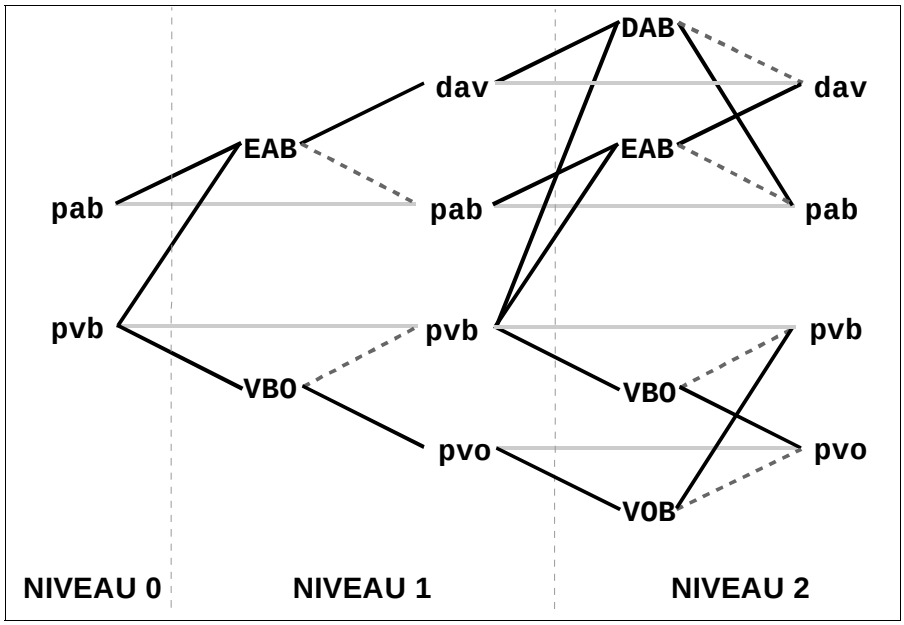

Figure 10 : Création du niveau 2

```
Mutex entre fluents :
  dav – pab
  pvb – pvo

Mutex entre actions :
  DAB – N_dav, EAB, N_pab, N_pvo, VOB
  EAB – N_pab, N_dav, N_pvo, VOB
  VBO – N_pvb, VOB, DAB, EAB, N_pvo
  VOB – N_pvo, N_dav, N_pvb
  N_dav – N_pab, N_pvo
  N_pvb – N_pvo
```

Figure 11 : Les mutex du niveau 2

Le fluent représentant le but du problème (**pao**) n'étant toujours pas présent dans l'ensemble des fluents du niveau 2, nous devons construire un niveau de plus. Nous n'allons pas cette fois détailler le processus de construction (se reporter à la Figure 12 pour la construction du graphe et à la Figure 13 pour le détail des mutex), mais nous devons mettre l'accent sur le fait essentiel de ce niveau : les fluents **dav** et **pvo** ne formant plus une mutex, l'action **DAO** peut enfin être utilisée. Elle produit ainsi pour la première fois le fluent qui représente le but du problème.

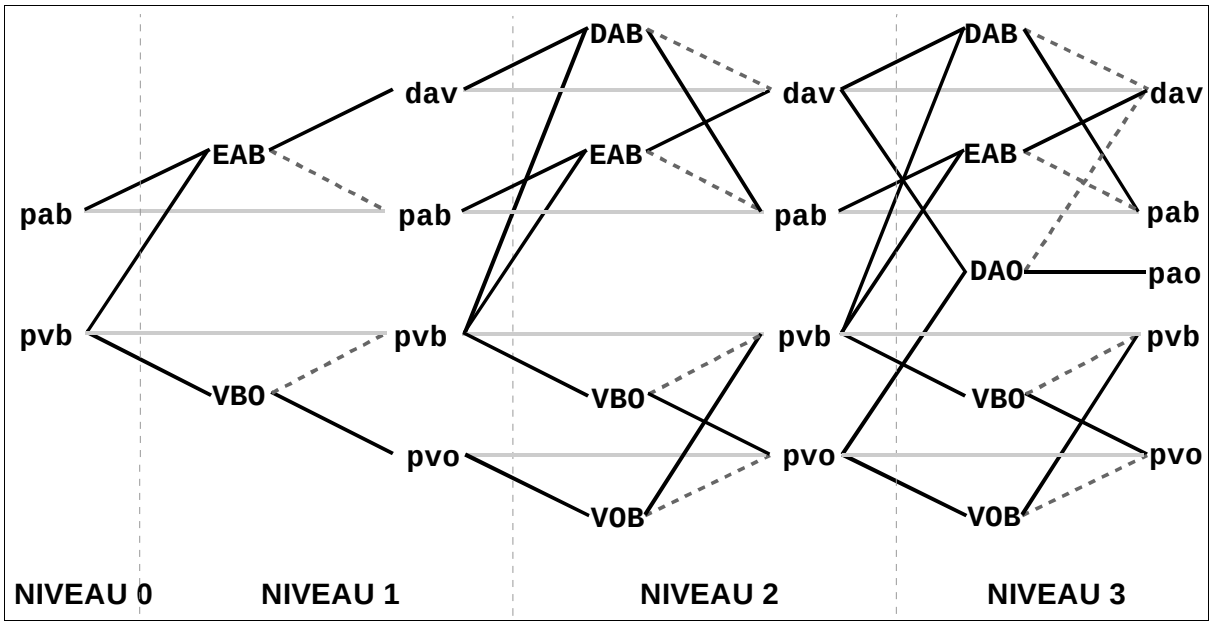

Figure 12 : Création du niveau 3

```
Mutex entre fluents :
  dav – pab, pao
  pao – pab, pvb
  pvb – pvo

Mutex entre actions :
  DAB – N_dav, EAB, DAO, N_pab, N_pvo
  EAB – N_pab, N_dav, N_pvo
  DAO – N_dav, EAB, N_pvb, VBO, N_pab
  VBO – N_pvb, VOB, DAB, EAB, N_pvo
  VOB – N_pvo, DAO, DAB, EAB, N_pvb
  N_dav – N_pab
  N_pvb – N_pvo
```

Figure 13 : Les Mutex du niveau 3

1.3. Extraction du plan-solution

Le fait que le fluent **pao** soit maintenant présent dans l'ensemble des fluents du niveau 3 est une condition nécessaire à la présence d'un plan-solution dans le graphe de planification, mais elle n'est pas suffisante. En effet, comme nous le montrerons plus tard, la construction de ce graphe se fait en temps polynomial. Cela signifierait donc que l'on pourrait résoudre n'importe quel problème de planification en temps polynomial ; or le problème de la planification de type STRIPS est lui-même PSPACE-complet.
Une autre condition nécessaire (mais non suffisante) à l'obtention d'un plan-solution dans un graphe de planification, dans le cas où le but est formé de plusieurs fluents, est qu'il n'existe aucune mutex dans le dernier niveau du graphe entre deux quelconques des fluents présents dans le but. En effet, les fluents formant le but doivent tous être présents dans l'état final, après exécution du plan-solution à partir de l'état initial du problème.

Attaquons-nous maintenant à la recherche d'un plan-solution dans notre graphe de planification. S'il n'en existe pas, il faudra recommencer l'étape d'expansion du graphe de planification en y ajoutant un quatrième niveau ; et ainsi de suite jusqu'à l'obtention soit d'un plan-solution, soit de la condition nécessaire et cette fois suffisante (que nous détaillerons plus tard) de l'inexistence d'un plan-solution.

La recherche d'un plan-solution (dans la version originelle de Graphplan) se fait par la construction d'un arbre ET/OU. Ou plutôt par une recherche dans un arbre ET/OU qui n'est plus à construire, puisqu'il est intégralement présent dans le graphe de planification. Voici comment trouver l'arbre ET/OU présent dans un graphe de planification :

- Les fluents représentant les buts du problème, présents dans le dernier niveau du graphe de planification, constituent les racines de l'arbre ET/OU. Un arbre ET/OU n'ayant habituellement qu'une seule racine, on peut considérer qu'il existe une action virtuelle, appelée RESOUDRE−BUT par exemple, qui aurait comme préconditions les buts du problème et comme effet un unique fluent appelé BUT. La racine de l'arbre ET/OU serait alors le fluent BUT.
- Les fluents du niveau 0 du graphe de planification sont les feuilles de l'arbre ET/OU.
- Les fluents du graphe de planification sont les noeuds OU de l'arbre ET/OU. Un même fluent, présent dans différents niveaux du graphe, est considéré comme étant différent à chacune de ses apparitions.
- Les actions du graphe de planification sont les noeuds ET de l'arbre ET/OU. Un même fluent, présent dans différents niveaux du graphe, est considéré comme étant différent à chacune de ses apparitions.
- Un arc d'ajout, qui relie un fluent à l'action qui l'établit, devient une branche de l'arbre ET/OU et y relie de même le fluent vers l'action.
- Un arc de précondition, qui relie une action à une de ses précondition, devient une branche de l'arbre ET/OU et y relie de même l'action et la précondition.

En suivant ces règles, nous pouvons extraire l'arbre ET/OU présent dans notre graphe de planification (voir la Figure 14).

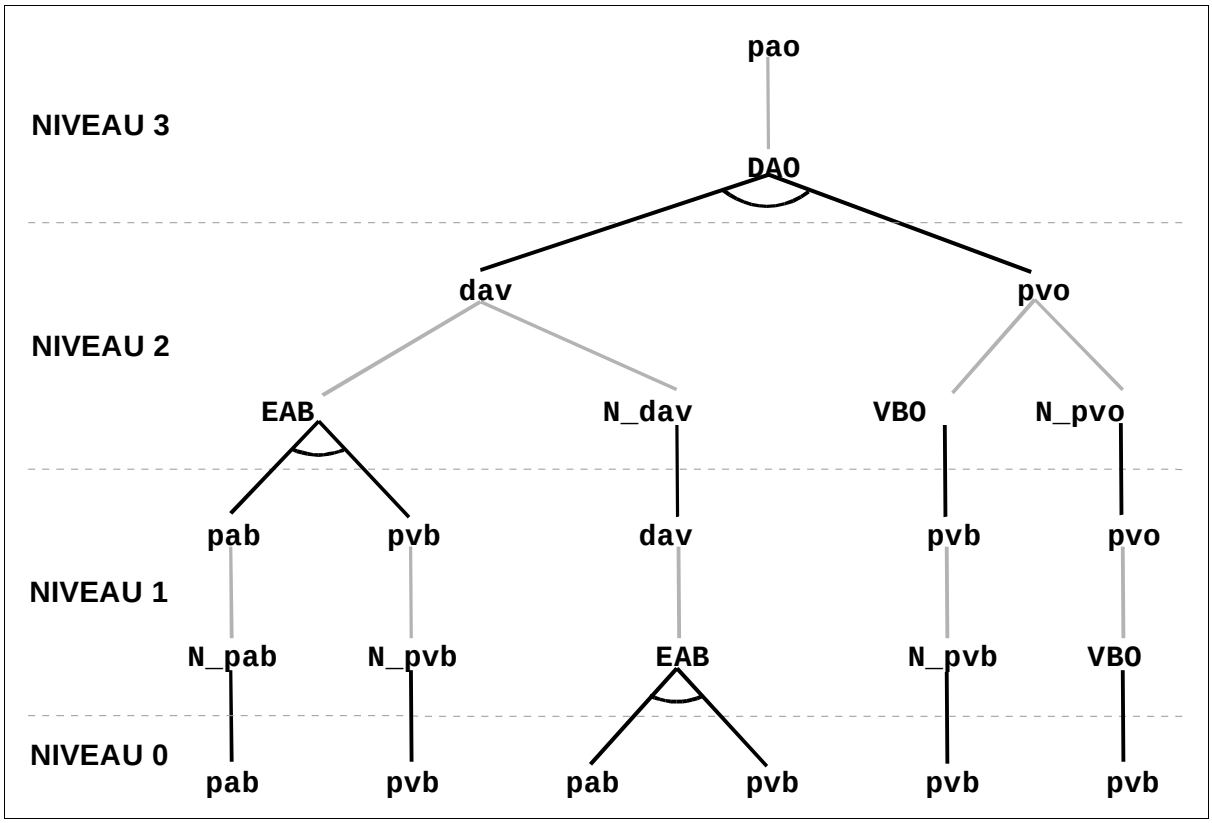

Figure 14 : L'arbre ET/OU issu du graphe de planification après création du niveau 3

Nous voulons maintenant extraire un plan-solution (s'il existe) de l'arbre ET/OU ainsi construit. Nous allons donc essayer de prouver l'ensemble des fluents du niveau 3 qui constituent le but : {**pao**}. Pour cela,

nous allons choisir une action parmi celles qui établissent ce fluent au niveau 3. Ici, un seul choix est disponible : l'action **DAO**. Si la recherche aboutit, cette action fera partie du plan-solution. Nous allons maintenant devoir prouver que toutes les préconditions de cette action peuvent être atteintes à partir de l'état initial du problème. Pour chacun des fluents de l'ensemble {**dav**, **pvo**} au niveau 2, il nous faut trouver une action qui l'établisse, de façon à ce que les deux actions choisies ne soient pas mutuellement exclusives. Les actions du niveau 2 qui établissent **dav** sont **EAB** et **N_dav**, et celles qui établissent **pvo** sont **VBO** et **N_pvo**. Les paires {**EAB**, **VBO**}, {**EAB**, **N_pvo**} et {**N_dav**, **N_pvo**} étant des mutex du niveau 2, le seul choix possible ici est {**N_dav**, **VBO**}. A nouveau, nous allons devoir prouver que l'ensemble des préconditions au niveau 1 de ces deux actions, l'ensemble {**dav**, **pvb**}, peut être atteint depuis l'état initial. Pour chacun des fluents de cet ensemble, nous devons maintenant trouver une action du niveau 1 qui l'établit. Ici il n'y a pas le choix : la seule action qui établit **dav** est **EAB** et la seule action qui établit **pvb** est **N_pvb**. Comme ces deux actions ne forment pas une mutex au niveau 1, la recherche peut continuer. Et elle se termine immédiatement après, puisque les préconditions de ces deux actions sont des fluents du niveau 0, et correspondent donc à des faits de l'état initial du problème sur lesquels il n'y a pas de mutex.

La recherche est donc couronnée de succès, et nous connaissons par conséquent le plan-solution du problème :

- A partir de l'état initial I = {**pab**, **pvb**} exécuter les actions **EAB** et **N_pvb**. Ces deux actions peuvent être exécutées en parallèle sans problème, sinon elles formeraient une mutex. L'exécution de **N_pvb** n'a aucun effet sur I (puisque c'est un no-op), et l'action **EAB** crée le fluent **dav** et retire le fluent **pab**. On se retrouve dans l'état $E1$ = {**dav**, **pvb**}.
- Dans l'état $E1$, on doit exécuter les actions **N_dav** et **VBO**. L'exécution de **N_dav** n'a aucun effet, et l'action **VBO** crée le fluent **pvo** et retire le fluent **pvb**. On se retrouve donc dans l'état $E2$ = {**dav**, **pvo**}.
- Dans l'état $E2$, on doit exécuter l'action **DAO** qui établit le fluent **pao** et retire le fluent **dav**. On se retrouve donc dans un état qui satisfait le but du problème, $E3$ = {**pao**, **pvo**}.

La Figure 15 donne le plan-solution que Graphplan retourne ; c'est-à-dire, pour chaque niveau, l'ensemble des actions qui mènent au but, nettoyé des no-ops et retraduit en logique du premier ordre.

```
NIVEAU 1 :
   (EMBARQUER Avrim Blagnac)

NIVEAU 2 :
   (VOLER Blagnac Orly)

NIVEAU 3 :
   (DEBARQUER Avrim Orly)
```

Figure 15 : Un plan-solution du problème

Une remarque importante, qui témoigne (un peu) de la puissance de Graphplan est la suivante : le graphe de planification est construit en temps polynomial, et la recherche arrière du plan-solution n'a ici nécessité aucun backtrack dans l'arbre ET/OU. En particulier, au niveau 2, on avait le choix entre quatre paires d'actions, mais trois étaient des mutex, ce qui nous a permis de parvenir droit au but, sans aucun retour en arrière. Grâce au calcul des contraintes et à leur propagation lors de la construction du graphe de planification, la partie exponentielle de l'algorithme global de Graphplan (l'extraction de la solution) s'est donc ici déroulée de façon linéaire. Ce ne sera le cas, bien sûr, que pour certains problèmes très faciles à résoudre ; néanmoins une partie de la puissance de Graphplan réside dans ce travail préliminaire de construction du graphe de planification et de détection des incompatibilités binaires entre les actions et entre les fluents.

1.4. La stabilisation du graphe de planification

Si nous n'avions pas trouvé de solution à notre problème après la construction du niveau 3 du graphe de planification, on aurait dû continuer à augmenter le graphe de planification, niveau par niveau. On aurait alors peut-être été confronté au phénomène de la **stabilisation** du graphe de planification. En effet, dans un cadre de planification STRIPS classique (état initial fini, pas de fonction dans la description des opérateurs), on observe le fait suivant : à partir d'un niveau i, tous les niveaux j tels que $j > i$ sont égaux. Par égaux, nous

entendons qu'ils possèdent le même ensemble de fluents, le même ensemble d'actions, les mêmes mutex sur les fluents et sur les actions. Nous dirons alors que la construction du graphe de planification s'est stabilisée au niveau *i*.

L'exemple que nous avons traité ne nous a pas permis de calculer le graphe de planification jusqu'à sa stabilisation : une solution a été trouvée. Nous allons tout de même continuer à calculer le graphe, afin de montrer qu'il atteint sa stabilisation.

Nous construisons donc le niveau 4 (voir la Figure 16). La seule nouveauté par rapport à la construction de l'étape précédente est l'apparition de l'action **EAO**, rendue possible grâce à la création du fluent **pao** au niveau 3. Il apparaît aussi le no-op **N_pao** correspondant au fluent **pao**. On peut remarquer dès maintenant que le niveau 3 contient tous les fluents et toutes les actions que nous avions répertoriés (voir la Figure 6). On peut donc se douter que l'on n'est pas loin de la stabilisation du graphe.

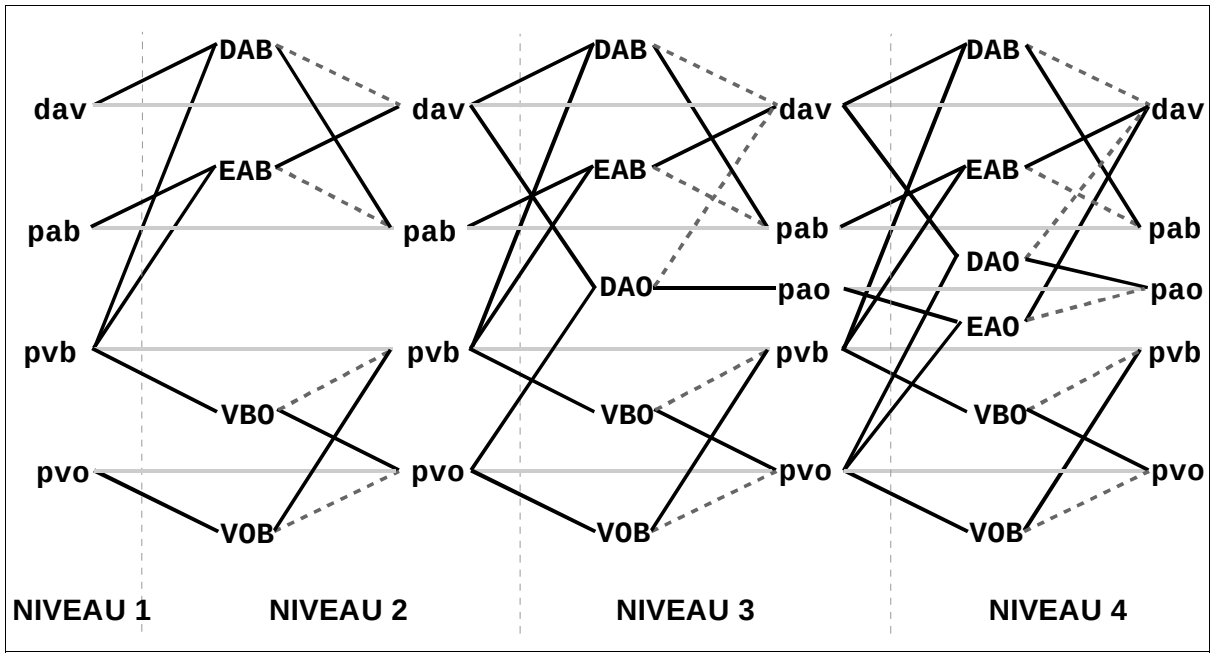

Figure 16 : Création du niveau 4

Nous pouvons maintenant calculer les mutex sur les actions et sur les fluents du niveau 4 (voir la Figure 17). Les seuls nouveaux mutex qui apparaissent sont des mutex sur les actions, et ils concernent les deux actions qui n'étaient pas présentes au niveau 3 : les actions **EAO** et **N_pao**. Par contre, une mutex sur les fluents disparaît : les fluents **pao** et **pvb** sont maintenant supportés par des actions qui ne sont pas mutuellement exclusives, comme par exemple les actions **N_pao** et **VOB**.

```
Mutex entre fluents :
  dav – pab, pao
  pao – pab
  pvb – pvo

Mutex entre actions :
  DAB – N_dav, EAB, DAO, N_pab, N_pvo
  EAB – N_pab, N_dav, N_pvo
  DAO – N_dav, EAB, N_pvb, VBO, N_pab
  VBO – N_pvb, VOB, DAB, EAB, N_pvo
  VOB – N_pvo, DAO, DAB, EAB, N_pvb
  EAO – N_pao, DAO, VOB, DAB, N_dav, EAB, N_pab, N_pvb, VBO
  N_pao – DAB, N_dav, DAO, EAB, N_pab, N_pvb, VBO
  N_dav – N_pab
  N_pvb – N_pvo
```

Figure 17 : Les mutex du niveau 4

Le niveau 4 n'étant pas égal au niveau 3, nous devons continuer la construction du graphe de planification (Figure 18) et la recherche des mutex (Figure 19) en construisant le niveau 5. Nous savons déjà que le graphe contient tous les fluents et toutes les actions possibles, la seule différence par rapport au niveau précédent sera donc peut-être la disparition de certaines mutex.

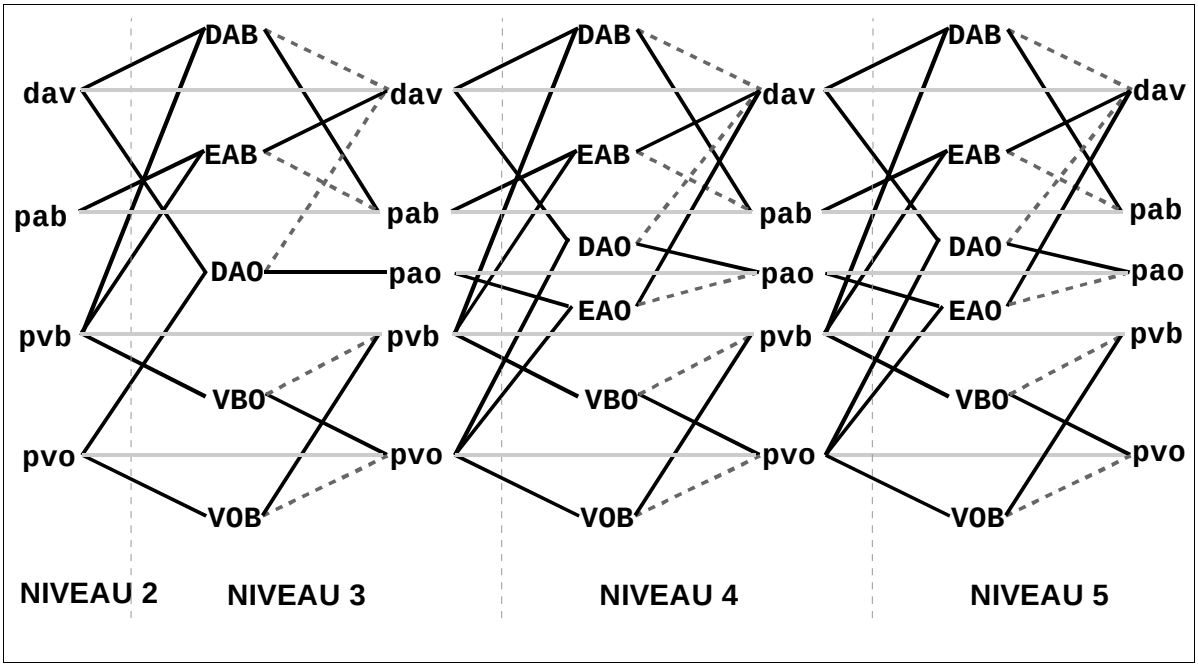

Figure 18 : Création du niveau 5

```
Mutex entre fluents :
  dav – pab, pao
  pao – pab
  pvb – pvo

Mutex entre actions :
  DAB – N_dav, EAB, DAO, N_pab, N_pvo
  EAB – N_pab, N_dav, N_pvo
  DAO – N_dav, EAB, N_pvb, VBO, N_pab
  VBO – N_pvb, VOB, DAB, EAB, N_pvo
  VOB – N_pvo, DAO, DAB, EAB, N_pvb
  EAO – N_pao, DAO, VOB, DAB, N_dav, EAB, N_pab, N_pvb, VBO
  N_pao – DAB, N_dav, DAO, EAB, N_pab
  N_dav – N_pab
  N_pvb – N_pvo
```

Figure 19 : Les mutex du niveau 5

Lors de la création du niveau 4, nous avons vu disparaître la mutex entre les fluents **pao** et **pvb**. Les actions qui utilisent chacune un de ces fluents ne sont donc plus forcément mutuellement exclusives. Ainsi, les mutex {**N_pvb**, **N_pao**} et {**VBO**, **N_pao**} disparaissent. Par contre, tous les mutex sur les fluents demeurent.

Le niveau 5 n'est toujours pas égal au niveau 4, mais la différence entre ces deux niveaux se limite à la disparition de deux mutex sur les actions. Est-il vraiment nécessaire de construire le niveau 6 pour vérifier qu'il est identique au niveau 5 ? Nous savons déjà que le graphe du niveau 6 sera égal au graphe du niveau 5 : mêmes fluents, mêmes actions, mêmes arcs. Une différence entre ces deux niveaux portera donc uniquement sur les mutex entre les fluents ou entre les actions. Or, les mutex entre les fluents du niveau 6 vont dépendre des mutex sur les actions de ce même niveau, qui elles-mêmes dépendent de leurs interactions (effets inconsistants, interférences) et des mutex sur les fluents du niveau 5 (préconditions inconsistantes). Nous pouvons donc remarquer que :

- Au niveau 5, nous connaissions déjà toutes les actions présentes au niveau 6. Nous avons donc déjà trouvé tous les mutex correspondant aux effets inconsistants et aux interactions. En effet, ces mutex sont des mutex permanents : si à un niveau donné, deux actions ont des effets qui se détruisent entre eux ou si une effet d'une action retire une précondition d'une autre, alors à tous les niveaux ou l'on rencontrera ces deux actions (c'est-à-dire tous les niveaux qui suivent le premier niveau où on les a rencontrées pour la première fois), ces deux actions seront toujours mutuellement exclusives. Nous savons donc que les mutex de ce type au niveau 6 seront identiques à ceux du niveau 5.
- Au niveau 5, nous avons retrouvé les mêmes mutex sur les fluents qu'au niveau 4. Les mutex sur les actions correspondant aux préconditions inconsistantes du niveau 6 vont donc être les mêmes que ceux du niveau 5, puisque ils seront calculés à partir des mêmes actions et des mêmes mutex sur les fluents.

Puisque le graphe du niveau 6 est identique à celui du niveau 5 et que les mutex sur les actions du niveau 6 sont les mêmes que ceux du niveau 5, nous pouvons en conclure que les mutex sur les fluents du niveau 6 seront identiques à ceux du niveau 5. La stabilisation du graphe de planification est donc atteinte au niveau 4.

Pour savoir que l'on a atteint le niveau de stabilisation du graphe, on n'a donc pas besoin de calculer un niveau supplémentaire. Il suffit simplement que pour deux niveaux successifs i et i+1, le nombre de fluents et le nombre de mutex sur ces fluents soit égaux. Le niveau de stabilisation du graphe est alors le niveau i. En effet :

- Un fluent, une fois introduit dans le graphe, sera présent dans tous les niveaux suivant son apparition grâce à son no-op. On peut en déduire que si le nombre de fluents du niveau i+1 est le même qu'au niveau i, les fluents du niveau i+1 sont identiques à ceux du niveau i.
- De la même façon, si le nombre de mutex sur les fluents du niveau i+1 est identique à celui des mutex sur les fluents du niveau i, et sachant que les fluents de ces deux niveaux sont les mêmes, on peut en conclure que les mutex sur les fluents du niveau i+1 sont identiques à ceux du niveau i. En effet si ce n'était pas le cas, cela signifierait qu'une mutex entre deux fluents est apparue au niveau i+1, alors qu'elle n'existait pas au niveau i. Or, ceci est impossible.
- L'introduction des actions dans le graphe au niveau i+2 dépend des fluents du niveau i+1 et de leurs mutex. Or, les fluents du niveau i+1 sont identiques à ceux du niveau i et les mutex sur les fluents du niveau i+1 sont identiques à ceux du niveau i. Les actions du niveau i+2 vont donc être identiques à celles du niveau i+1, puisqu'elles dépendent des mêmes fluents et des mêmes mutex sur ces fluents.
- Les actions du niveau i+2 étant identiques à celles du niveau i+1 et les effets de ces actions ne changeant pas, les fluents du niveau i+2 seront donc les mêmes que ceux du niveau i+1.
- Les mutex sur les actions du niveau i+2, du type effets inconsistants et interférence, seront les mêmes que ceux sur les actions du niveau i+1 puisque les actions du niveau i+2 et du niveau i+1 sont les mêmes. Ces mutex ne dépendent en effet que des actions elles-mêmes.
- Les mutex sur les actions du niveau i+2, du type préconditions inconsistantes, seront les mêmes que ceux du niveau i+1. En effet, les actions du niveau i+2 sont les mêmes que celles du niveau i+1, ont les mêmes préconditions et les mêmes mutex sur ces préconditions.
- Enfin, puisque les fluents, les actions et les mutex sur les actions du niveau i+2 sont les mêmes qu'au niveau i+1, les mutex sur les fluents au niveau i+2 seront les mêmes que ceux du niveau i+1.

On peut donc en conclure que les niveaux i+1 et i+2 sont strictement identiques, et donc que le niveau de stabilisation du graphe de planification est le niveau i.

Le fait que le graphe de planification atteigne ainsi un niveau de stabilisation est d'une importance capitale pour la terminaison de l'algorithme, comme nous le verrons par la suite.

2. Définitions de base

2.1. Le problème de planification

Dans la version originale de Graphplan, les opérateurs sont de type STRIPS, sans négation dans leurs préconditions. Nous utilisons une logique du premier ordre *L*, construite à partir des vocabulaires *Vx*, *Vc*, *Vp* qui dénotent respectivement des ensembles finis disjoints deux à deux de symboles de variables, de constantes et de prédicats. Nous n'utilisons pas de symbole de fonction.

Définition 1 (opérateur)

Un *opérateur*, dénoté par *o*, est un triplet $\langle pr, ad, de \rangle$ où *pr*, *ad* et *de* dénotent des ensembles finis de formules atomiques du langage *L*. Prec(*o*), Add(*o*) et Del(*o*) dénotent respectivement les ensembles *pr*, *ad* et *de* de l'opérateur *o*.

Définition 2 (état, fluent)

Un *état* est un ensemble fini de formules atomiques de base (i.e. sans symbole de variable). Une formule atomique de base est aussi appelée un *fluent*. L'ensemble des états que l'on peut construire à partir d'un ensemble *F* fini de fluents sera noté 2^F.

Définition 3 (action)

Une *action* dénotée par *a* est une instance de base $o\theta = \langle pr\theta, ad\theta, de\theta \rangle$ d'un opérateur *o* qui est obtenue par l'application d'une substitution θ définie dans le langage *L* telle que *ad*θ et *de*θ sont des ensembles disjoints. Prec(*a*), Add(*a*), Del(*a*) dénotent respectivement les ensembles *pr*θ, *ad*θ, *de*θ et représentent les préconditions, ajouts et retraits de *a*.

Nous donnons maintenant une première version d'un *problème de planification*. Cette définition correspond à ce que l'on fournit en général en entrée à un logiciel de génération de plans : un ensemble d'opérateurs, un état initial et un but. L'objectif du logiciel est de retourner une solution de ce problème sous forme d'un plan-solution si il existe, et sinon de signifier qu'il n'en existe pas.

Définition 4 (problème de planification)

Un *problème de planification* $\prod$ est un triplet $\langle O, I, B \rangle$ où :

- *O* dénote un ensemble fini d'opérateurs construits à partir du langage *L*,
- *I* dénote un ensemble fini de fluents construits à partir du langage *L* qui représentent l'état initial du problème,
- *B* dénote un ensemble fini de fluents construits à partir du langage *L* qui représentent les buts du problème.

2.2. Les plans et l'application des plans

L'objectif du processus de planification est de trouver une solution à un problème de planification. Celle-ci sera appelée un *plan-solution*. De façon informelle, un plan-solution est une collection organisée d'actions qui permet de faire évoluer la représentation initiale du monde vers un état qui satisfait le but du problème. Deux notions essentielles restent à définir : la structure de cette collection organisée d'actions, et la manière dont cette structure doit être "appliquée" à un état initial pour produire un état final qui, s'il satisfait le but du problème, garantit que l'on a bien un plan-solution. La structure la plus adaptée aux plans que calculent les planificateurs auxquels nous nous intéressons est une structure qui est généralement appelée "plan parallèle". Il y a deux aspects dans cette notion : celui qui concerne le plan, et celui qui concerne le parallélisme entre actions.

Nous allons définir séparément ces deux aspects : dans un premier temps, nous définissons la structure qui représente les plans (la séquence d'ensembles d'actions), et dans un deuxième temps la condition que doit satisfaire ce plan pour être un plan-solution. Cette condition fera intervenir non pas la notion classique de parallélisme, mais une certaine propriété que doivent satisfaire les ensembles d'actions du plan. Cette propriété pourra être le parallélisme tel qu'on le conçoit généralement, mais pas uniquement.

2.2.1. Définitions

Définition 5 (plan)
Un *plan* est une séquence finie d'ensembles finis d'actions. Un plan P est noté $\langle Q_i \rangle_n$, avec $n \in \mathbb{N}$. Si $n = 0$, P est le plan vide : $P = \langle Q_i \rangle_0 = \langle\rangle$; si $n > 0$, P peut être noté $\langle Q_1, Q_2, ..., Q_n \rangle$. Si les ensembles d'actions sont des singletons (i.e. $Q_1 = \{a_1\}$, $Q_2 = \{a_2\}$, ..., $Q_n = \{a_n\}$), le plan associé est une séquence d'actions et sera noté (par abus de langage) $\langle a_1, a_2, ..., a_n \rangle$. L'ensemble des plans construits à partir d'un ensemble d'actions A est notée $(2^A)^*$. L'ensemble des séquences d'actions construites à partir d'un ensemble d'actions A est noté A^*.

Définition 6 (tête, reste, longueur)
Soit A un ensemble d'actions. Nous définissons les fonctions classiques sur les séquences :

- tête : $(2^A)^* - \{\langle\rangle\} \rightarrow 2^A$ est définie par : tête$(\langle Q_1, Q_2, ..., Q_n \rangle) = Q_1$.
- reste : $(2^A)^* - \{\langle\rangle\} \rightarrow (2^A)^*$ est définie par : reste$(\langle Q_1, Q_2, ..., Q_n \rangle) = \langle Q_2, ..., Q_n \rangle$.
- longueur : $(2^A)^* \rightarrow \mathbb{N}$ est définie par : longueur$(\langle Q_i \rangle_n) = n$.

Définition 7 (concaténation de plans)
Soient P et P' deux plans, avec $P = \langle Q_i \rangle_n$ et $P' = \langle Q'_i \rangle_m$. La *concaténation* (notée $\oplus$) de P et P' est définie par :

$$P \oplus P' = (\text{si } n+m = 0 \text{ alors } \langle\rangle \text{ sinon } \langle R_i \rangle_{n+m}, \text{ avec } R_i = (\text{si } 1 \leq i \leq n \text{ alors } Q_i \text{ sinon } Q'_{i-n})).$$

Propriété 1
Soit A un ensemble d'actions. La concaténation de plans possède les propriétés suivantes :

1. élément neutre : $\forall P \in (2^A)^*, P \oplus \langle\rangle = \langle\rangle \oplus P = P$
2. associativité : $\forall P_1, P_2, P_3 \in (2^A)^*, (P_1 \oplus P_2) \oplus P_3 = P_1 \oplus (P_2 \oplus P_3)$ (noté $P_1 \oplus P_2 \oplus P_3$)
3. stabilité pour A^* : $\forall P_1, P_2 \in A^*, P_1 \oplus P_2 \in A^*$

Preuve : trivial.

Définition 8 (linéarisation)
Une *linéarisation* d'un ensemble d'actions $Q = \{a_1, ..., a_n\}$ est une permutation de Q, c'est-à-dire une séquence d'actions S telle qu'il existe une bijection $b : [1, n] \rightarrow Q$ où $S = \langle b(1), ..., b(n) \rangle$. La linéarisation de l'ensemble vide $\{\}$ est la séquence vide $\langle\rangle$. L'ensemble de toutes les linéarisations de Q est dénoté par Lin(Q).

Notations : Si Q est l'ensemble d'actions $Q = \{a_1, ..., a_n\}$, alors :

- l'union des préconditions des éléments de Q est notée Prec(Q): Prec(Q) = Prec(a_1) $\cup$ ... $\cup$ Prec(a_n),
- l'union des ajouts des éléments de Q est notée Add(Q): Add(Q) = Add (a_1) $\cup$ ... $\cup$ Add(a_n),
- l'union des retraits des éléments de Q est notée Del(Q): Del(Q) = Del(a_1) $\cup$ ... $\cup$ Del(a_n).

Nous utilisons la même notation pour la séquence d'actions $Q = \langle a_1, ..., a_n \rangle$.
Si $Q = \varnothing$ ou $Q = \langle\rangle$, Prec(Q) = Add(Q) = Del(Q) = $\varnothing$.

Définition 9 (application d'un ensemble d'actions)
Soient F un ensemble de fluents et A un ensemble d'actions. Soit $\uparrow : 2^F \times (2^A)^* \rightarrow 2^F$, définie par :

$$E \uparrow Q = (E - \text{Del}(Q)) \cup \text{Add}(Q)$$

Notation : si Q est un singleton : $Q = \{a\}$, $E \uparrow \{a\}$ sera noté (par abus de langage) $E \uparrow a$. Lorsqu'il n'y a pas d'ambiguïté, $((...((E \uparrow a_1) \uparrow a_2) \uparrow ...) \uparrow a_n)$ sera noté (par abus de langage) $E \uparrow a_1 \uparrow a_2 \uparrow ... \uparrow a_n$.

Nous définissons maintenant l'application d'un plan à un état, ce qui nous permettra de définir ensuite la notion de plan-solution pour un problème de planification : il s'agira d'un plan possédant certaines propriétés particulières.

Définition 10 (application d'un plan)

Soient F un ensemble de fluents et A un ensemble d'actions.

Soit l'application $\Re : (2^F \cup \{\bot\}) \times (2^A)^* \rightarrow (2^F \cup \{\bot\})$, définie par :

$E \Re P =$

Si $P = \langle\rangle$ ou $E = \bot$
alors E
sinon si Prec(tête(P)) $\subseteq E$
alors ($E \uparrow$ tête(P)) $\Re$ reste(P)
sinon $\bot$.

Notation : lorsqu'il n'y a pas d'ambiguïté, ((...((($E \Re P_1$) $\Re P_2$) $\Re$ ...)$\Re P_n$) sera noté $E \Re P_1 \Re P_2 \Re$ ... $\Re P_n$.

2.2.2. Propriétés

L'application de plans telle que nous venons de la définir possède plusieurs propriétés intéressantes. Une première de ces propriétés est que l'application de la concaténation de plusieurs plans mène au même état résultant que l'application successive de ces mêmes plans :

Propriété 2

Soient E un état et P_1, ..., P_n des plans. On a alors :

$$E \Re (P_1 \oplus ... \oplus P_n) = E \Re P_1 \Re ... \Re P_n$$

Preuve : elle est basée sur le Lemme suivant.

Lemme 1

Soient E un état et P_1, P_2 deux plans. On a alors :

$$E \Re (P_1 \oplus P_2) = E \Re P_1 \Re P_2.$$

Preuve :

Soient E un état et P_1, P_2 deux plans.

Si $E = \bot$:

$E \Re (P_1 \oplus P_2) = \bot \Re (P_1 \oplus P_2) = \bot$

$E \Re P_1 \Re P_2 = (\bot \Re P_1) \Re P_2 = \bot \Re P_2 = \bot$

Si $E \neq \bot$, nous pouvons faire une preuve par récurrence sur longueur(P_1).

- Quand longueur(P_1) = 0 :
  $E \Re (P_1 \oplus P_2) = E \Re (\langle\rangle \oplus P_2) = E \Re P_2$
  $E \Re P_1 \Re P_2 = (E \Re \langle\rangle) \Re P_2 = E \Re P_2$
- Supposons vraie la propriété suivante, pour longueur(P_1) = n :
  $$E \Re (P_1 \oplus P_2) = E \Re P_1 \Re P_2$$
  Démontrons que la propriété est vraie quand longueur(P_1) = n+1 :
  $E \Re (P_1 \oplus P_2)$
  $= E \Re (\langle$tête(P_1)$\rangle \oplus$ reste(P_1) $\oplus P_2$) car longueur(P_1) > 0
  $= E \Re (\langle Q_1\rangle \oplus P'_1 \oplus P_2)$ avec Q_1 = tête(P_1) et P'_1 = reste(P_1)
  Deux cas peuvent se produire :
  – Si Prec(Q_1) $\not\subseteq E$:
    $E \Re (P_1 \oplus P_2) = E \Re (\langle Q_1\rangle \oplus P'_1 \oplus P_2) = \bot$
    $E \Re P_1 \Re P_2 = E \Re (\langle Q_1\rangle \oplus P'_1) \Re P_2 = \bot \Re P_2 = \bot$
  – Si Prec(Q_1) $\subseteq E$:
    $E \Re (P_1 \oplus P_2)$
    $= E \Re (\langle Q_1\rangle \oplus P'_1 \oplus P_2)$
    $= (E \uparrow Q_1) \Re (P'_1 \oplus P_2)$
    $= E' \Re (P'_1 \oplus P_2)$ avec $E' = (E \uparrow Q_1)$
    $E \Re P_1 \Re P_2$
    $= E \Re (\langle Q_1\rangle \oplus P'_1) \Re P_2$
    $= (E \uparrow Q_1) \Re P'_1 \Re P_2$
    $= E' \Re P'_1 \Re P_2$
    $= E' \Re (P'_1 \oplus P_2)$ (hyp. récurrence), car longueur(P'_1) = longueur(reste(P_1)) = n

Preuve de la Propriété 2 :

Soient E un état et P_1, ..., P_n des plans. Nous pouvons faire une preuve par récurrence sur n.

- Si $n = 1$: trivial.
- Si n > 1 : supposons que la propriété suivante soit vraie et démontrons-la pour n+1 :

$$E \,\Re\, (P_1 \oplus ... \oplus P_n) = E \,\Re\, P_1 \,\Re\, ... \,\Re\, P_n$$

$E \,\Re\, (P_1 \oplus ... \oplus P_n \oplus P_{n+1})$
$= E \,\Re\, ((P_1 \oplus ... \oplus P_n) \oplus P_{n+1})$
$= E \,\Re\, (P_1 \oplus ... \oplus P_n) \,\Re\, P_{n+1}$ d'après le Lemme 1
$= E \,\Re\, (P_1 \,\Re\, ... \,\Re\, P_n) \,\Re\, P_{n+1}$ hyp. de récurrence
$= E \,\Re\, P_1 \,\Re\, ... \,\Re\, P_n \,\Re\, P_{n+1}$

Une deuxième propriété intéressante concerne l'application d'une séquence d'actions : si toutes les préconditions des actions d'une séquence sont présentes dans un état et si aucune action de la séquence ne retire une précondition d'une action qui la suit (immédiatement ou non) dans la séquence, alors l'état résultant de l'application de la séquence conduit à un état différent de $\perp$ et correspond à l'application successive des actions de la séquence :

Propriété 3

Soient E un état et $S = \langle a_1, ..., a_n \rangle$ une séquence d'actions non vide telle que : $\text{Prec}(S) \subseteq E$ et $\forall i \in [1, n-1]$, $\text{Prec}(a_{i+1}) \cap (\text{Del}(a_1) \cup ... \cup \text{Del}(a_i)) = \emptyset$. On a alors :

$$E \,\Re\, S = E \uparrow a_1 \uparrow a_2 \uparrow ... \uparrow a_n.$$

Preuve :

Soient E un état et $S = \langle a_1, ..., a_n \rangle$ une séquence d'actions.

Nous pouvons faire une preuve par récurrence sur longueur(S) :

- Quand longueur(S) = 1 : $S = \langle a_1 \rangle$ et $\text{Prec}(a_1) \subseteq E$, donc $E \,\Re\, S = E \,\Re\, \langle a_1 \rangle = E \uparrow a_1$.
- Supposons que la propriété suivante soit vraie quand longueur(S) = n :

  Soit $S = \langle a_i \rangle_n$ avec $\text{Prec}(S) \subseteq E$ et $\forall i \in [1, n-1]$, $\text{Prec}(a_{i+1}) \cap (\text{Del}(a_1) \cup ... \cup \text{Del}(a_i)) = \emptyset$,

  $$E \,\Re\, \langle a_1, a_2, ..., a_n \rangle = E \uparrow a_1 \uparrow a_2 \uparrow ... \uparrow a_n.$$

  Démontrons que la propriété est vraie quand longueur(S) = n+1, avec $S = \langle a_i \rangle_{n+1}$, $\text{Prec}(S) \subseteq E$ et $\forall i \in [1, n+1]$, $\text{Prec}(a_{i+1}) \cap (\text{Del}(a_1) \cup ... \cup \text{Del}(a_i)) = \emptyset$:

$E \,\Re\, \langle a_1, ..., a_{n+1} \rangle$
$= E \,\Re\, (\langle a_1, ..., a_n \rangle \oplus \langle a_{n+1} \rangle)$
$= E \,\Re\, \langle a_1, ..., a_n \rangle \,\Re\, \langle a_{n+1} \rangle$ d'après la Propriété 2
$= ((E \,\Re\, \langle a_1, ..., a_n \rangle) \uparrow a_{n+1}$ car $E \,\Re\, \langle a_1, ..., a_n \rangle \neq \perp$ et $\text{Prec}(a_{n+1}) \subseteq E \,\Re\, \langle a_1, ..., a_n \rangle$
$= E \uparrow a_1 \uparrow ... \uparrow a_n \uparrow a_{n+1}$. (hyp. de récurrence)

La troisième propriété importante est la suivante : l'état résultant de l'application d'un ensemble d'actions sur un état initial est identique à l'état résultant de l'application d'une linéarisation de cet ensemble d'actions, si les préconditions de toutes les actions de cet ensemble sont présentes dans l'état initial et si aucune action de la linéarisation ne retire ni une précondition d'une action qui la suit (immédiatement ou non) dans la linéarisation, ni un ajout d'une action qui la précède (immédiatement ou non) dans la linéarisation.

Propriété 4

Soit E un état, Q un ensemble d'actions tel que $\text{Prec}(Q) \subseteq E$, et $S \in \text{Lin}(Q)$ avec $S = \langle a_1, ..., a_n \rangle$, telle que :

$$\forall i \in [1, n-1], \text{Del}(a_{i+1}) \cap (\text{Add}(a_1) \cup ... \cup \text{Add}(a_i)) = \emptyset$$
$$\text{et } \text{Prec}(a_{i+1}) \cap (\text{Del}(a_1) \cup ... \cup \text{Del}(a_i)) = \emptyset.$$

On a alors :

$$E \,\Re\, \langle Q \rangle = E \,\Re\, S.$$

Preuve : elle est basée sur le lemme suivant.

Lemme 2

Soient A, A_1, ..., A_n, B_1, ..., B_n des ensembles tels que $\forall i \in [1, n-1]$, $A_{i+1} \cap (B_1 \cup ... \cup B_i) = \emptyset$. On a alors :

$$(A - (A_1 \cup ... \cup A_n)) \cup (B_1 \cup ... \cup B_n) = ((...((A - A_1) \cup B_1) - ...) - A_n) \cup B_n.$$

Preuve :

Soient A, A_1, ..., A_n, B_1, ..., B_n des ensembles tels que $\forall i \in [1, n-1]$, $A_{i+1} \cap (B_1 \cup ... \cup B_n) = \varnothing$.
Nous utiliserons les propriétés ensemblistes suivantes :

$$A - (B \cup C) = (A - B) - C \qquad (\alpha)$$

et

$$B \cap C = \varnothing \Rightarrow (A - B) \cup C = (A \cup C) - B \qquad (\beta)$$

Nous pouvons faire une preuve par récurrence sur n :

- Quand $n = 1$: trivial.
- Supposons que la propriété suivante est vraie au rang n et démontrons-la au rang $n+1$:

$$(A - (A_1 \cup ... \cup A_n)) \cup (B_1 \cup ... \cup B_n) = ((...((A - A_1) \cup B_1) - ...) - A_n) \cup B_n$$
$$\text{avec } \forall i \in [1, n-1],\ A_{i+1} \cap (B_1 \cup ... \cup B_i) = \varnothing.$$

Soient A_{n+1} et B_{n+1} tels que $A_{n+1} \cap (B_1 \cup ... \cup B_n) = \varnothing$:

$(A - (A_1 \cup ... \cup A_n \cup A_{n+1})) \cup (B_1 \cup ... \cup B_n \cup B_{n+1})$
$= ((A - (A_1 \cup ... \cup A_n)) - A_{n+1}) \cup (B_1 \cup ... \cup B_n) \cup B_{n+1}$ d'après (α)
$= (((A - (A_1 \cup ... \cup A_n)) \cup (B_1 \cup ... \cup B_n)) - A_{n+1}) \cup B_{n+1}$ d'après (β), car $A_{n+1} \cap (B_1 \cup ... \cup B_n) = \varnothing$
$= ((((...((A - A_1) \cup B_1) - ...) - A_n) \cup B_n) - A_{n+1}) \cup B_{n+1}$ (hyp. de récurrence)

Preuve de la Propriété 4 :

Soit E un état, Q un ensemble d'actions tel que $\text{Prec}(Q) \subseteq E$, et $S \in \text{Lin}(Q)$ avec $S = \langle a_1, ..., a_n \rangle$, telle que $\forall i \in [1, n-1]$, $\text{Del}(a_{i+1}) \cap (\text{Add}(a_1) \cup ... \cup \text{Add}(a_i)) = \varnothing$ et $\text{Prec}(a_{i+1}) \cap (\text{Del}(a_1) \cup ... \cup \text{Del}(a_i)) = \varnothing$.
Comme $\text{Prec}(Q) \subseteq E$, on a :
$E \,\Re\, \langle Q \rangle = E \uparrow Q$
$= (E - \text{Del}(Q)) \cup \text{Add}(Q)$
$= (E - \text{Del}(S)) \cup \text{Add}(S)$
$= (E - \text{Del}(a_1) \cup ... \cup \text{Del}(a_n)) - (\text{Add}(a_1) \cup ... \cup \text{Add}(a_n))$
Comme $\text{Prec}(Q) \subseteq E$ et $\forall i \in [1, n-1]$, $\text{Prec}(a_{i+1}) \cap (\text{Del}(a_1) \cup ... \cup \text{Del}(a_i)) = \varnothing$, on a d'après la Propriété 3 :
$E \,\Re\, S = E \uparrow a_1 \uparrow a_2 \uparrow ... \uparrow a_n$
$= ((...((((E - \text{Del}(a_1)) \cup \text{Add}(a_1)) - \text{Del}(a_2)) \cup \text{Add}(a_2)) - ...) - \text{Del}(a_n)) \cup \text{Add}(a_n)$
Comme $\forall i \in [1, n-1]$, $\text{Del}(a_{i+1}) \cap (\text{Add}(a_1) \cup ... \cup \text{Add}(a_i)) = \varnothing$, on a bien $E \,\Re\, \langle Q \rangle = E \,\Re\, S$ d'après le Lemme 2.

2.3. Les plans-solutions et l'équivalence des problèmes

Nous allons maintenant définir la notion de plan-solution pour un problème de planification. Remarquons une chose importante : nous avons défini les plans comme étant des séquences d'ensembles d'actions, dans l'optique de représenter ce que l'on appelle couramment des plans parallèles. Or, nous n'avons pas encore expliqué ce que nous entendons par le terme "parallélisme", et notre définition des plans-solutions doit faire intervenir cette notion. Notre objectif est ici de donner une définition très générale d'un plan-solution, afin d'obtenir un maximum de possibilités pour la préciser ensuite. Notre définition d'un plan-solution va donc faire intervenir non pas le parallélisme en tant que tel (c'est-à-dire ce qu'il est généralement convenu d'appeler parallélisme), mais une contrainte qui doit être vérifiée par chaque ensemble d'actions du plan. Nous verrons plus tard ce que doit être cette propriété pour obtenir un plan-solution "parallèle" au sens courant du terme. Nous définirons également d'autres propriétés que l'on peut employer pour le calcul de plans-solutions.

Définition 11 (plan-solution d'un problème de planification)

Soient $\prod = \langle O, I, B \rangle$ un problème de planification, $P = \langle Q_i \rangle_n$ un plan, et Φ une contrainte sur les ensembles d'actions. P est un plan-solution de $\prod$ relativement à Φ si et seulement si P satisfait les propriétés suivantes :

1. $\forall i \in [1, n]$, $Q_i = \{o_1\theta_1, ..., o_k\theta_k\}$ avec $k \in \mathbb{N}$ et $\forall m \in [1, k]$, $o_m \in O$ et $o_m\theta_m$ est une action.
2. $\forall i \in [1, n]$, $\Phi(Q_i)$.
3. $B \subseteq I \,\Re\, P$.

L'ensemble des plans-solutions de $\prod$ pour la contrainte Φ sera noté $\prod(\Phi)$. Un plan-solution $P \in \prod(\Phi)$ sera appelé un Φ-plan-solution de $\prod$. Un plan P satisfaisant les propriétés 1 et 2 sera appelé un Φ-plan.

Jusqu'à présent, nous avons défini un problème de planification comme étant un triplet dont le premier élément est un ensemble d'opérateurs. Un planificateur prenant en entrée un tel problème va devoir trouver des substitutions afin de construire des actions pouvant faire partie d'un plan-solution. La plupart des planificateurs, jusqu'à SATPLAN et Graphplan, effectuaient cette recherche de substitutions au fur et à mesure des besoins. Certains planificateurs, comme par exemple les planificateurs fonctionnant en arrière dans les espaces de plans partiels, introduisent même des actions partiellement instanciées dans leurs plans partiels. Les planificateurs SATPLAN et Graphplan, quant à eux, travaillent sur des structures propositionnelles dans lesquelles toutes les actions sont totalement instanciées, et sont toutes présentes dans une même structure : une base de clauses pour SATPLAN, un graphe de planification pour Graphplan. Il est possible de construire cette structure en instanciant au fur et à mesure les opérateurs ; ainsi, on ne calcule que les actions qui doivent réellement faire partie de la structure. L'inconvénient de cette approche est que l'on recalcule sans cesse les mêmes substitutions.

Devant la quantité croissante de mémoire vive disponible dans les machines actuelles, il devient plus intéressant du point de vue des performances et tout à fait réalisable de calculer toutes les actions possibles pour un problème de planification et de résoudre ensuite le problème constitué de cet ensemble d'actions, et de l'état initial et du but du problème de départ. Les deux problèmes sont équivalents, c'est-à-dire :

Définition 12 (équivalence de problèmes de planification)

Soient $\prod$ et $\prod$' deux problèmes de planification. $\prod$ et $\prod$' sont équivalents si et seulement si, pour toute contrainte Φ sur les ensembles d'actions, $\prod(\Phi) = \prod'(\Phi)$.

En nous basant sur la remarque précédente, nous donnons donc une méthode simple qui permet de calculer un problème équivalent à un problème initial, et dont l'ensemble des opérateurs ne contient plus que des actions de base. Nous appellerons un tel problème un problème de base. Le principe de cette méthode est le suivant : à partir de l'état initial, on recherche toutes les actions applicables ; on augmente ensuite l'état initial des ajouts de ces actions, puis on réitère le processus à partir de l'état obtenu jusqu'à ce que l'ensemble des actions produites soit stable. Cette méthode permet également de réaliser une présélection des actions qui pourront réellement être utilisées : ainsi, une partie des actions dont les préconditions ne pourront jamais être produites par l'application d'un plan depuis l'état initial du problème seront automatiquement écartées..

Définition 13 (génération d'un problème de planification de base)

Soit $\prod = \langle O, I, B\rangle$ un problème de planification. La suite ConsPB($\prod$) de génération d'un problème de planification de base est définie par :

$\text{ConsPB}(\prod)_0 = \varnothing$

$\text{ConsPB}(\prod)_{n+1} = \{o\theta \mid o \in O, o\theta \text{ est une action, Prec}(o\theta) \subseteq (I \cup \text{Add}(\text{ConsPB}(\prod)_n))\}$

Théorème 1 (équivalence du problème de planification de base)

Soit $\prod = \langle O, I, B\rangle$ un problème de planification. On a alors :

$$\exists n \geq 0, \forall m > n, \text{ConsPB}(\prod)_n = \text{ConsPB}(\prod)_m = A \text{ et } \prod' = \langle A, I, B\rangle \text{ est équivalent à } \prod.$$

Schéma de preuve :

Soit $\prod = \langle O, I, B\rangle$ un problème de planification. Montrons d'abord par récurrence que ConsPB($\prod$) est croissante pour l'inclusion, c'est-à-dire $\forall n \geq 0$, $\text{ConsPB}(\prod)_n \subseteq \text{ConsPB}(\prod)_{n+1}$.

- Pour $n = 0$: trivial : $\text{ConsPB}(\prod)_0 = \varnothing \subseteq \text{ConsPB}(\prod)_1$.
- Pour $n \geq 0$, supposons que $\text{ConsPB}(\prod)_n \subseteq \text{ConsPB}(\prod)_{n+1}$ et montrons que $\text{ConsPB}(\prod)_{n+1} \subseteq \text{ConsPB}(\prod)_{n+2}$. Soit $a \in \text{ConsPB}(\prod)_{n+1}$. On a alors $\text{Prec}(a) \subseteq I \cup \text{Add}(\text{ConsPB}(\prod)_n))$. Comme $\text{ConsPB}(\prod)_n \subseteq \text{ConsPB}(\prod)_{n+1}$ alors $\text{Add}(\text{ConsPB}(\prod)_n) \subseteq \text{Add}(\text{ConsPB}(\prod)_{n+1})$ donc $\text{Prec}(a) \subseteq I \cup \text{Add}(\text{ConsPB}(\prod)_{n+1}))$ donc $a \in \text{ConsPB}(\prod)_{n+2}$.

Comme l'ensemble des actions de base que l'on peut construire à partir de l'ensemble O des opérateurs est fini dans le langage L et que la suite ConsPB($\prod$) est croissante pour l'inclusion, cette suite devient constante i.e. il existe un entier $n \geq 0$ tel que $\forall m > n$, $\text{ConsPB}(\prod)_n = \text{ConsPB}(\prod)_m = A$. Soit $\prod' = \langle A, I, B\rangle$ le problème de planification défini à partir de A.

Pour montrer que $\prod$' est équivalent à $\prod$, on remarque d'abord que les actions de A sont constituées par l'instanciation d'opérateurs de O dans le langage L, donc les plans-solutions pour $\prod$' sont des plans-solutions pour $\prod$ puisqu'ils sont construits à partir des mêmes actions de base. D'autre part, les actions d'un plan-solution pour $\prod$ constituent simplement une restriction des actions que l'on trouve dans

A puisque ce dernier est augmenté progressivement non seulement des ajouts des actions qui constituent des plans menant au but, mais aussi de toutes les autres actions, qui ne mènent pas forcément au but.

A partir de maintenant, nous ne travaillerons plus que sur des problèmes de planification de base provenant de la recherche de la convergence de la suite ConsPB(∏), équivalent à un problème initial ∏, suivant la méthode de construction donnée dans la Définition 13 et le Théorème 1.

3. Formalisation de Graphplan

Nous allons maintenant nous intéresser à une description formelle de Graphplan. Nous avons vu que Graphplan fonctionne en deux phases entrelacées : la construction du graphe et l'extraction d'un plan-solution dans ce graphe.

Dans un premier temps, nous étudierons donc en détail la construction et les propriétés essentielles des graphes de planification. La définition que nous proposons d'un graphe de planification étend la définition originale de Blum et Furst [Blum et Furst 1995] [Blum et Furst 1997] pour la prise en compte de différentes définitions de parallélisme. En effet, la définition originale d'un graphe de planification est basée sur la notion de parallélisme en planification classique : nous avons vu dans l'exemple développé dans la section 1 que deux actions ne peuvent être appliquées simultanément sur un état que si leur application successive dans n'importe quel ordre conduit à des états identiques. Nous paramètrerons donc la construction du graphe par une relation binaire sur les actions afin d'accepter d'autres types de parallélisme.

Nous étudierons ensuite une méthode simple d'extraction d'une solution dans les graphes de planification paramétrés par une relation binaire entre actions, basée sur la méthode originale proposée par Blum et Furst [Blum et Furst 1995] [Blum et Furst 1997]. Elle présente l'avantage de proposer un critère d'arrêt lorsque le problème n'admet pas de solution. Tout comme la construction du graphe, cette méthode sera paramétrée ; cette fois-ci par une contrainte Φ sur les ensembles d'actions, afin de prendre en compte différentes définitions du parallélisme. Nous montrerons que l'algorithme ainsi obtenu est complet pour la recherche des Φ−plans-solutions d'un problème de planification, moyennant une condition entre la relation binaire utilisée pour la construction du graphe et la contrainte Φ.

3.1. Le graphe de planification

3.1.1. Construction

Nous allons maintenant définir une application qui permet de construire la structure de données essentielle de Graphplan, le *graphe de planification*. Cette structure est appelée un graphe par abus de langage (terminologie employée par ses créateurs), car elle véhicule des informations supplémentaires par rapport à un graphe : des contraintes binaires entre les fluents et les actions, les exclusions mutuelles (mutex). Une exclusion mutuelle à un niveau i entre deux fluents signifie que ces fluents ne peuvent pas faire partie d'un même état après l'application d'un plan de longueur i. Entre deux actions, elle signifie que ces dernières ne peuvent être appliquées en même temps sur un état après l'application d'un plan de longueur $i-1$. La raison de cette exclusion mutuelle entre deux actions peut être soit que deux de leurs préconditions sont mutuellement exclusives, soit qu'elles ne peuvent jamais être exécutées en même temps sur aucun état (ce qui fait intervenir encore une fois la notion de parallélisme).

Pour définir le graphe de planification, nous laissons de côté cette notion de parallélisme dans Graphplan : elle fera l'objet de la section suivante (cf. section 4). La construction d'un graphe de planification sera donc paramétrée par une relation binaire entre les actions, qui pourra être la notion classique de parallélisme telle qu'elle est utilisée dans la version originale de Graphplan, mais qui pourra également être une autre relation, satisfaisant certaines propriétés que nous allons expliciter. La construction du graphe de planification telle que nous allons la définir constitue une généralisation de la définition donnée par Blum et Furst, mais les principaux résultats théoriques restent inchangés : la construction du graphe reste polynomiale et le phénomène de stabilisation du graphe se produit toujours.

La construction d'un graphe de planification pour un problème donné utilise des actions supplémentaires par rapport à celles du problème : les no-ops. Ces actions permettent essentiellement de gérer le problème du décor, en spécifiant qu'un fluent présent à un niveau i sera présent au niveau $i+1$.

Définition 14 (no-op d'un fluent)
Le no-op d'un fluent *f* est l'action dénotée par N_*f* telle que :

$$\text{Prec}(\text{N\_}f) = \{f\},\ \text{Add}(\text{N\_}f) = \{f\},\ \text{Del}(\text{N\_}f) = \emptyset.$$

Définition 15 (graphe de planification)
Soient $\prod = \langle A, I, B\rangle$ un problème de planification, *R* une relation binaire sur *A* et *k* un entier naturel. Le graphe de planification d'ordre *k* pour le problème $\prod$ et la relation *R* est le triplet $\langle Noeuds, Arcs, Mutex\rangle$ défini par : $\langle Noeuds, Arcs, Mutex\rangle = \text{ConsGP}(\prod, R, k)$, où ConsGP est l'application définie récursivement par :

Pour $i = 0$: $\text{ConsGP}(\prod, R, i) = \langle \{f(0) \mid f \in I\}, \emptyset, \emptyset\rangle$.

Pour $i \in [1, k]$: soit $\langle Npr, Apr, Mpr\rangle = \text{ConsGP}(\prod, R, i-1)$.
$\text{ConsGP}(\prod, R, i) = \langle Npr \cup Na \cup Nf, Apr \cup Ap \cup Aa \cup Ar, Mpr \cup Ma \cup Mf\rangle$, avec :

noeuds d'actions :
$$Na = \{a(i) \mid a \in A, (\forall f \in \text{Prec}(a), f(i-1) \in Npr), (\forall \{f, g\} \subseteq \text{Prec}(a), \{f(i-1), g(i-1)\} \notin Mpr)\} \cup \{\text{N\_}f(i) \mid f \notin A, f(i-1) \in Npr\}$$
noeuds de fluents :
$$Nf = \{f(i) \mid f \in \text{Add}(a), a(i) \in Na\}$$
Arcs de précondition :
$$Ap = \{(f(i-1), a(i)) \mid a(i) \in Na, f \in \text{Prec}(a)\}$$
Arcs d'ajout :
$$Aa = \{(a(i), f(i)) \mid a(i) \in Na, f \in \text{Add}(a)\}$$
Arcs de retrait :
$$Ar = \{(a(i), f(i)) \mid a(i) \in Na, f \in \text{Del}(a)\}$$
Mutex entre actions :
$$Ma = \{\{a(i), b(i)\} \mid \{a(i), b(i)\} \subseteq Na, a \neq b, (\neg(a\ R\ b) \text{ ou } (\exists \{f(i-1), g(i-1)\} \in Mpr, f \in \text{Prec}(a), g \in \text{Prec}(b)))\}$$
Mutex entre fluents :
$$Mf = \{\{f(i), g(i)\} \mid \{f(i), g(i)\} \subseteq Nf, f \neq g, (\forall (a(i), f(i)), (b(i), g(i)) \in Aa, \{a(i), b(i)\} \in Ma\}$$

Notations :
Soit *GP* un graphe de planification.

- NoeudsInit(*GP*), NoeudsBut(*GP*) dénotent respectivement l'ensemble des noeuds du niveau 0 (ils correspondent aux fluents de l'état initial), et l'ensemble des noeuds du niveau *k* qui correspondent aux fluents du but.
- Noeuds(*GP*), NoeudsF(*GP*), NoeudsA(*GP*) dénotent respectivement l'ensemble des noeuds, l'ensemble des noeuds de fluents et l'ensemble des noeuds d'actions de *GP*.
- ArcsPrec(*GP*), ArcsAdd(*GP*), ArcsDel(*GP*) dénotent respectivement l'ensemble des arcs de précondition, l'ensemble des arcs d'ajouts et l'ensemble des arcs de retraits de *GP*.
- Mutex(*GP*), MutexF(*GP*), MutexA(*GP*) dénotent respectivement l'ensemble des mutex, l'ensemble des mutex entre fluents et l'ensemble des mutex entre actions de *GP*.

3.1.2. Complexité

Le principal résultat concernant la complexité de la création d'un graphe de planification reste valable pour notre définition étendue du graphe : sa taille et son temps de création sont polynomiaux par rapport à la taille du problème. En effet, peu importe la nature de la relation que l'on choisit pour la création de ce graphe, puisqu'il s'agira toujours d'une relation binaire dont la taille et le temps de calcul sont polynomiaux par rapport à la taille du problème.

Théorème 2 (taille du graphe polynomiale, [Blum et Furst 1997, page 288, theorem 1])
Soit $\prod = \langle A, I, B\rangle$ un problème de planification. Soit *n* le nombre de fluents qui peuvent apparaître dans un graphe de planification ($n = |\text{Add}(A) \cup \text{Del}(A) \cup I|$), *p* le nombre de fluents dans l'état initial ($p = |I|$), et *m* le nombre d'actions ($m = |A|$). Pour toute relation binaire *R* sur les actions dont la taille et le temps de calculs sont polynomiaux par rapport à la taille du problème, la taille d'un graphe de planification d'ordre *k* et le temps de création de ce graphe sont polynomiaux en *n*, *p* et *m*.

Schéma de preuve :

La preuve donnée dans [Blum et Furst 1997] ne fait pas intervenir la nature de la relation *R* entre les actions : il suffit de savoir que sa taille et son temps de calcul sont polynomiaux par rapport à la taille du problème. La preuve de ce théorème reste donc inchangée.

3.1.3. Stabilisation

Nous avons vu dans la section 1.4 qu'à partir d'un niveau *k*, tous les niveaux suivants du graphe de planification sont identiques. Ceci se produit lorsque les noeuds de fluents au niveau *k*+1 et les mutex entre ces noeuds sont identiques à ceux du niveau *k*+1. Cette propriété reste inchangée pour notre définition du graphe de planification, puisque la condition essentielle qui garantit ce phénomène est que deux actions qui ne sont pas mutuellement exclusives à un niveau ne puissent le devenir dans les niveaux suivants. En effet la seule différence entre la définition originale du graphe et la notre consiste dans la possibilité de prendre en compte n'importe quelle relation binaire entre deux actions. Donc, si deux noeuds d'actions sont mutuellement exclusifs car les actions ne font pas partie de la relation binaire *R* ayant servi à la construction du graphe, ces noeuds seront toujours mutuellement exclusifs si on se sert toujours de *R* pour étendre le graphe ; de plus, si deux noeuds d'actions ne sont pas mutuellement exclusifs à partir d'un certain niveau, alors ils ne le deviendront jamais dans les niveaux supérieurs si ceux-ci sont construits avec la même relation *R*.

Définition 16 (niveau de stabilisation du graphe de planification)

Soit *GP* un graphe de planification d'ordre *k*. Le *niveau de stabilisation* de *GP* est le plus petit niveau $i < k$ tel que :

$$\{f \mid f(i) \in \text{NoeudsF}(GP)\} = \{f \mid f(i+1) \in \text{NoeudsF}(GP)\}$$

$$\{\{f, g\} \mid \{f(i), g(i)\} \in \text{MutexF}(GP)\} = \{\{f, g\} \mid \{f(i+1), g(i+1)\} \in \text{MutexF}(GP)\}$$

Lorsque *GP* a atteint son niveau de stabilisation, on dira qu'il est *stabilisé*. Le niveau de stabilisation de *GP* sera noté niveau−stab(*GP*).

Théorème 3 (stabilisation du graphe, [Blum et Furst 1997, page 296, section 5.1])

Soient $\prod = \langle A, I, B\rangle$ un problème de planification et *R* une relation binaire sur les actions. A partir d'un niveau *k*, $\forall i > k$, $GP = \text{ConsGP}(\prod, R, i)$ est stabilisé et :

$$\{n \mid n(i) \in \text{Noeuds}(GP)\} = \{n \mid n(i+1) \in \text{Noeuds}(GP)\}$$

$$\{\{m, n\} \mid \{m(i), n(i)\} \in \text{Mutex}(GP)\} = \{\{m, n\} \mid \{m(i+1), n(i+1)\} \in \text{Mutex}(GP)\}$$

Schéma de preuve :

Encore une fois, la preuve donnée dans [Blum et Furst 1997] ne fait pas intervenir la nature de la relation *R* entre les actions. La preuve est basée sur le fait que si un noeud de fluent ou d'action est présent dans le graphe à un niveau donné, alors il sera présent dans tous les niveaux suivants ; et si une exclusion mutuelle entre deux noeuds de fluents ou d'actions disparaît à un niveau donné, alors elle ne sera jamais présente dans les niveaux suivants. Comme la relation *R* que nous employons porte sur les actions et non les noeuds d'actions, cette propriété reste vraie donc le graphe atteint bien un niveau de stabilisation.

3.2. Extraction de la solution

Nous allons maintenant décrire la phase d'extraction d'un Φ−plan-solution dans un graphe de planification d'ordre *k*. La version que nous donnons ici est une généralisation de la version originelle de Graphplan, par l'utilisation d'une contrainte Φ que doivent vérifier les ensembles d'actions pouvant faire partie d'un plan. Cette extraction est réalisée par deux fonctions mutuellement récursives, **résoudre−but** et **rechercher−actions**.

La fonction résoudre−but : c'est la fonction qui est appelée en premier pour extraire un plan-solution, pour un ensemble de fluents *B* constituant le but du problème. Elle effectue un appel à la fonction **rechercher−actions** qui devra tester tous les ensembles d'actions qui produisent les fluents de *B*. Elle est ensuite rappelée à chaque niveau par la fonction **rechercher−actions**, pour trouver un plan-solution produisant les préconditions des actions que **rechercher−actions** a choisi.

Elle a un autre rôle, qui est de stocker les ensembles de fluents pour lesquels aucun plan-solution n'a été trouvé. En effet, il se peut que l'on revienne à un niveau avec un ensemble de sous-buts que l'on a déjà testé à ce niveau : comme la recherche d'un plan-solution le produisant avait échoué une première fois, elle va échouer à nouveau. Avant de faire appel à **rechercher−action**, on teste donc si l'ensemble de fluents n'a pas déjà été rencontré ; le cas échéant, on peut retourner *échec*. Si l'appel à **rechercher−action** retourne *échec*, l'ensemble de fluents peut être conservé comme étant insoluble à ce niveau. Cette mémoization est effectuée à l'aide de la variable *Nogoods*, qui est un tableau indicé par le niveau concerné : *Nogoods*[*i*] contient tous les ensembles de fluents insolubles rencontrés au niveau *i*. Cette mémoization permet non seulement d'accélérer la recherche, mais aussi de fournir un critère d'arrêt à la fonction principale de Graphplan, comme nous le verrons dans la section suivante.

```
Fonction résoudre−but (GP, Φ, B, i, Nogoods)
;; Entrée :
;; − GP : un graphe de planification d'ordre k ≥ i.
;; − Φ : une contrainte sur les ensembles d'actions.
;; − B : un ensemble de fluents pour lesquels on doit trouver un un Φ−plan-solution de longueur i.
;; − i : un entier qui représente le niveau des noeuds associés aux fluents de B.
;; − Nogoods : le tableau des nogoods.
;; Sortie :
;; − échec si on peut pas extraire de GP un Φ−plan-solution de longueur i pour B.
;; − sinon : un Φ−plan-solution de longueur i pour B.
Début
    Si i = 0 alors retourner ⟨⟩
    Sinon
        Si B ∈ Nogoods[i] alors retourner échec
        Sinon
            P ← rechercher−actions(GP, Φ, B, ∅, i, Nogoods) ;
            Si P = échec alors
                Nogoods[i] ← Nogoods[i] ∪ {B} ;
                retourner échec
            Sinon retourner P
Fin
```

Algorithme 1 : résoudre−but

La fonction rechercher−actions : elle est appelée initialement par la fonction **résoudre−but** afin de trouver un ensemble d'actions qui produisent les fluents d'un ensemble *B*. Elle choisit un fluent de *B* avec la fonction **choisir** (ce qui peut être fait de manière heuristique, cf. [Kambhampati et Nigenda 2000], [Kambhampati 2000], [Cayrol, Régnier et Vidal 2000b], [Cayrol, Régnier et Vidal 2001]). Ce choix ne constitue pas un point de retour arrière ; ce dernier est géré en essayant d'utiliser toutes les actions qui produisent ce fluent, en rappelant à chaque fois la fonction **rechercher−action** avec l'ensemble *B* amputé du fluent qui a été choisi. Lorsque *B* est vide, on a trouvé un ensemble d'actions *Q* dont les noeuds correspondants dans le graphe de planification ne sont pas mutuellement exclusifs deux à deux. La différence entre la version originelle de Graphplan et la version présentée ici est que l'on demande à cet ensemble d'actions de vérifier une contrainte Φ, qui impose des restrictions supplémentaires. Si *Q* vérifie cette contrainte, alors il peut faire partie d'un Φ−plan-solution et on rappelle la fonction **résoudre−but** avec comme ensemble de fluents les préconditions des actions de *Q* ; sinon on retourne *échec*.

Fonction rechercher−actions(*GP*, Φ, *B*, *Q*, *i*, *Nogoods*)
;; Entrée :
;; − *GP* : un graphe de planification d'ordre $k \geq i$.
;; − Φ : une contrainte sur les ensembles d'actions.
;; − *B* : un ensemble de fluents pour lesquels on doit trouver un ensemble d'actions *Q*' tel que $B \subseteq \text{Add}(Q')$.
;; − *Q* : un ensemble d'actions qui produisent les fluents de l'ensemble *B* de l'appel initial à
;; rechercher−actions, que l'on a retiré de *B*.
;; − *i* : un entier qui représente le niveau des noeuds associés aux fluents de *B*.
;; − *Nogoods* : le tableau des nogoods.
;; Sortie :
;; − *échec* si on peut pas trouver une action *a* qui produise un fluent donné $f \in B$, tel que le noeud associé à
;; *a* au niveau *i* n'est mutuellement exclusif avec aucun des noeuds associés aux actions de *Q* au niveau *i*.
;; − sinon : un Φ−plan-solution de longueur *i* pour l'ensemble *B* correspondant à l'appel initial à
;; rechercher−actions.
Début
  Si $B = \emptyset$ **alors**
    Si Φ(*Q*) **alors**
      $P \leftarrow$ résoudre−but(*GP*, Φ, Prec(*Q*), *i*−1, *Nogoods*) ;
      Si *P* = *échec* **alors retourner** *échec*
      Sinon retourner $P \oplus \langle Q \rangle$
    Sinon retourner *échec*
  Sinon
    choisir $f \in B$;
    PourTout $(a(i), f(i)) \in$ ArcsAdd(*GP*) **faire**
      Si $\forall b \in Q, \{a(i), b(i)\} \notin$ MutexA(*GP*) **alors**
        $P \leftarrow$ rechercher−actions(*GP*, Φ, $B - \{f\}$, $Q \cup \{a\}$, *i*, *Nogoods*) ;
        Si $P \neq$ *échec* **alors retourner** *P*
    retourner *échec*
Fin

Algorithme 2 : rechercher−actions

3.3. Fonction principale de Graphplan

La différence par rapport à la version originale de Graphplan est que l'algorithme est maintenant paramétré par une relation binaire entre actions *R* qui sert à la création du graphe, et une contrainte Φ que doivent satisfaire tous les ensembles d'actions qui font partie des plans-solutions. Cette modification permet à l'algorithme de trouver des plans-solutions ayant des propriétés différentes de celles des plans que l'on recherche habituellement en planification classique. Nous devrons ensuite démontrer que notre version étendue de Graphplan possède les mêmes propriétés essentielles que ce dernier, à savoir la complétude et la terminaison, étendues à la recherche des Φ−plans-solutions d'un problème.

La fonction **rechercher−plan−GP** procède de la façon suivante. Tout d'abord, elle étend le graphe de planification jusqu'à un niveau dans lequel les buts du problème sont tous présents dans des noeuds du dernier niveau, aucun de ces derniers n'étant mutuellement exclusifs deux à deux. Si la stabilisation du graphe est atteinte avant cette condition, il n'y a pas de solution. Ensuite, la fonction recherche une solution dans le graphe par un appel à la fonction **résoudre−but**, et en cas d'échec elle étend le graphe d'un niveau. Ce processus est répété jusqu'à l'obtention soit d'un Φ−plan-solution du problème, soit de la condition d'arrêt. Cette condition est basée sur la mémoization des ensembles de sous-buts insolubles à chaque niveau du graphe, effectuée par la procédure **résoudre−but** : le problème n'a pas de solution lorsque le graphe de planification a atteint son niveau de stabilisation (stabilisation(*GP*) prend la valeur *vrai* dans l'algorithme) et le nombre d'ensembles de fluents mémoizés au niveau de la stabilisation reste stable sur deux itérations successives de la boucle.

```
Fonction rechercher−plan−GP(∏ = ⟨A, I, B⟩, Φ, R)
;; Entrée :
;; − ∏ : un problème de planification.
;; − Φ : une contrainte sur les ensembles d'actions.
;; − R : une relation binaire entre actions.
;; Sortie :
;; − échec si ∏(Φ) = ∅.
;; − sinon : un Φ−plan-solution de ∏.
Début
    k ← 0 ;
    GP ← ConsGP(∏, R, k) ;
    TantQue ¬stabilisation(GP) et
            (∃f ∈ B, f(k) ∉ NoeudsF(GP) ou ∃{f, g} ⊆ B, (f(k), g(k)) ∈ MutexF(GP)) faire
        k ← k + 1 ;
        GP ← ConsGP(∏, R, k) ;
    Si ∃f ∈ B, f(k) ∉ NoeudsF(GP) ou ∃{f, g} ⊆ B, (f(k), g(k)) ∈ MutexF(GP) alors retourner échec
    Sinon
        P ← résoudre−but(GP, Φ, B, k, Nogoods) ;
        Si stabilisation(GP) alors
            nb−nogoods ← |Nogoods[niveau−stab(GP)]|
        Sinon
            nb−nogoods ← 0 ;
        TantQue P = échec faire
            k ← k + 1 ;
            GP ← ConsGP(∏, R, k) ;
            P ← résoudre−but(GP, Φ, B, k, Nogoods) ;
            Si P = échec et stabilisation(GP) alors
                Si nb−nogoods = |Nogoods[niveau−stab(GP)]| alors
                    retourner échec
                Sinon
                    nb−nogoods ← |Nogoods[niveau−stab(GP)]| ;
        retourner P
```

Algorithme 3 : rechercher−plan−GP

Pour que notre version étendue de Graphplan soit toujours complète et se termine quand il n'y a pas de solution, il faut que R et Φ vérifient la condition suivante : si un ensemble d'actions Q vérifie la contrainte Φ, alors toutes les paires d'actions de Q doivent appartenir à la relation R. En effet, lors de la création du graphe de planification, toutes les paires de noeuds dont les actions associées n'appartiennent pas à R sont mémorisées comme étant des mutex entre actions. Lors de l'extraction de la solution, tous les ensembles d'actions pour un niveau i contenant deux actions dont les noeuds associés sont mutuellement exclusifs au niveau i sont automatiquement rejetés par la fonction **rechercher−actions**. Donc, si un ensemble d'actions Q satisfaisait la contrainte Φ mais était rejeté parce que deux de ses actions n'appartenaient pas à R, l'algorithme ne serait pas complet pour la recherche des Φ−plans-solutions du problème. Nous définissons donc la compatibilité entre la relation R et la contrainte Φ de la façon suivante :

Définition 17 (Φ−compatibilité)

Soient R une relation binaire entre actions et Φ une contrainte sur les ensembles d'actions. La relation R est Φ−compatible ssi :

$$\forall Q, \Phi(Q) \Rightarrow (\forall \{a, b\} \subseteq Q, a \neq b \Rightarrow a \; R \; b).$$

Théorème 4 (complétude de rechercher−plan−GP)
Soient $\prod$ un problème de planification, Φ une contrainte sur les ensembles d'actions, et R une relation binaire entre actions Φ−compatible. On a alors :

$$\prod(\Phi) \neq \varnothing \Leftrightarrow \textbf{rechercher−plan−GP}(\prod, \Phi, R) \in \prod(\Phi)$$

Schéma de preuve :
La seule différence entre la version originelle de Graphplan et la notre est que l'on rejette les ensembles d'actions qui ne satisfont pas la contrainte Φ. On est donc sûr que si un plan-solution est retourné, c'est bien un Φ−plan-solution : un planificateur basé sur la fonction **rechercher−plan−GP** sera donc sain.
Pour démontrer que si un Φ−plan-solution existe, **rechercher−plan−GP** le retourne, en supposant que Graphplan retourne bien un plan-solution quand il existe, il suffit de montrer qu'aucune solution potentielle n'est rejetée par la vérification de la contrainte Φ ; la condition pour cela étant que la relation R ayant servi à la construction du graphe soit Φ−compatible. En effet si R n'était pas Φ−compatible, alors il existerait au moins une paire d'actions $\{a, b\}$ telle que $\Phi(\{a, b\})$ et $\neg(a\ R\ b)$: la paire $\{a, b\}$ pourrait donc éventuellement faire partie d'un Φ−plan-solution, mais serait rejetée avant même d'être testée par Φ puisque tous les noeuds d'actions $a(i)$ et $b(i)$ seraient mutuellement exclusifs pour tout i.

Théorème 5 (terminaison de rechercher−plan−GP)
Soient $\prod$ un problème de planification, Φ une contrainte sur les ensembles d'actions, et R une relation binaire entre actions Φ−compatible. On a alors :

$$\prod(\Phi) = \varnothing \Leftrightarrow \textbf{rechercher−plan−GP}(\prod, \Phi, R) = \textit{échec}$$

Schéma de preuve :
Encore une fois, la preuve donnée dans [Blum et Furst 1997] ne fait pas intervenir la nature de la relation R entre les actions. De plus, comme nous le verrons dans la section suivante, la version originelle de Graphplan fait implicitement intervenir une relation Φ, vérifiée automatiquement par l'absence d'exclusions mutuelles entre noeuds d'actions. La preuve de la terminaison étant basée sur la stabilisation du graphe de planification et la mémoization des ensemble de fluents insolubles, la nature de la contrainte Φ ne change rien à la preuve de Blum et Furst. En effet, comme la relation R, la contrainte Φ porte sur les actions et non les noeuds d'actions ; ainsi, un ensemble d'actions rejeté à un niveau donné sera aussi rejeté dans les niveaux inférieurs s'il est rencontré, donc un ensemble de fluents mémoizé à un niveau sera aussi mémoizé dans les niveaux inférieurs s'il est rencontré. La preuve du critère d'arrêt de Graphplan étant basée sur cette propriété, sur les no-ops et sur la stabilisation du graphe, il reste donc valide pour **rechercher−plan−GP** avec une relation R Φ−compatible.

4. Le parallélisme dans Graphplan : sémantique et formalisation

La formalisation de Graphplan que nous avons proposée permet de prendre en compte différentes relations afin de gérer le parallélisme. Nous allons nous intéresser maintenant à la sémantique du parallélisme en fonction de l'utilisation de différentes relations.

4.1. Le parallélisme en planification classique : la relation d'indépendance

Comme nous l'avons vu dans le développement de l'exemple de la section 1, la version originale de Graphplan utilise la notion usuelle de parallélisme en planification classique : Graphplan contraint fortement le choix des actions pouvant être appliquées en parallèle, afin de s'assurer que l'exécution parallèle ou séquentielle des actions d'un plan conduit au même état résultant. Pour obtenir ce résultat avec l'utilisation d'une description d'actions de type STRIPS, toutes les actions d'un même ensemble d'actions d'un plan doivent être indépendantes deux à deux, c'est-à-dire que leurs effets ne doivent pas être contradictoires (une action ne doit pas retirer un ajout d'une autre action) et elles ne doivent pas interférer (une action ne doit pas retirer une précondition d'une autre action).

Pour obtenir ce même résultat, nous allons définir une relation binaire entre deux actions qui servira à la construction du graphe de planification, et une contrainte sur les ensembles d'actions qui servira à l'extraction d'un plan-solution :

Définition 18 (indépendance entre actions)
Deux actions a et b telles que $a \neq b$ sont *indépendantes* (noté $a \# b$) ssi :

$$(\text{Add}(a) \cup \text{Prec}(a)) \cap \text{Del}(b) = \varnothing \text{ et } (\text{Add}(b) \cup \text{Prec}(b)) \cap \text{Del}(a) = \varnothing.$$

Propriété 5
La relation d'indépendance entre actions est irréflexive et symétrique.

Preuve : triviale.

Définition 19 (ensemble d'actions indépendant)
Un ensemble d'actions Q est *indépendant* (noté $\Phi_{\#}(Q)$) ssi les actions qui le composent sont indépendantes deux à deux, c'est-à-dire :

$$\forall \{a, b\} \subseteq Q, a \neq b \Rightarrow a \# b$$

ce qui équivaut à :

$$\forall \{a, b\} \subseteq Q, a \neq b \Rightarrow (\text{Prec}(a) \cup \text{Add}(a)) \cap \text{Del}(b) = \varnothing.$$

Le Théorème 6 établit la propriété essentielle des $\Phi_{\#}-$plans : les actions d'un $\Phi_{\#}-$plan qui peuvent être exécutées en parallèle produisent le même état résultant quand elles sont exécutées séquentiellement, quel que soit leur ordre d'exécution.

Théorème 6 (égalité de l'état résultant par applications parallèles et séquentielles d'un $\Phi_{\#}-$plan)
Soit E un état et P un $\Phi_{\#}-$plan non vide, avec $P = \langle Q_1, ..., Q_n \rangle$. On a alors :

$$E \Re P \neq \perp \Rightarrow \forall S_1 \in \text{Lin}(Q_1), ..., \forall S_n \in \text{Lin}(Q_n), E \Re P = E \Re (S_1 \oplus ... \oplus S_n).$$

Preuve : elle est basée le lemme suivant.

Lemme 3
Soit Q un ensemble d'actions tel que $\Phi_{\#}(Q)$ et $\text{Prec}(Q) \subseteq E$. On a alors :

$$\forall S \in \text{Lin}(Q), E \Re \langle Q \rangle = E \Re S.$$

Preuve :
Soit Q un ensemble d'actions tel que $\Phi_{\#}(Q)$ et $\text{Prec}(Q) \subseteq E$. Soit $S \in \text{Lin}(Q)$, avec $S = \langle a_1, ..., a_n \rangle$.
Comme $\Phi_{\#}(Q)$, alors :

$$\forall \{a, b\} \subseteq Q, a \neq b \Rightarrow (\text{Prec}(a) \cup \text{Add}(a)) \cap \text{Del}(b) = \varnothing$$

ce qui implique que :

$$\forall i \in [1, n-1], \text{Del}(a_{i+1}) \cap (\text{Add}(a_1) \cup ... \cup \text{Add}(a_i)) \text{ et } \text{Prec}(a_{i+1}) \cap (\text{Del}(a_1) \cup ... \cup \text{Del}(a_i)) = \varnothing.$$

D'après la Propriété 4, comme $\text{Prec}(Q) \subseteq E$, on a bien $E \Re \langle Q \rangle = E \Re S$.

Preuve du Théorème 6 :
Soit E un état et P un $\Phi_{\#}-$plan non vide, avec $P = \langle Q_1, ..., Q_n \rangle$ tel que $E \Re P \neq \perp$ (ce qui implique $E \neq \perp$).
Nous pouvons faire une preuve par récurrence sur la longueur de P.

- Quand longueur(P) = 1: $P = \langle Q \rangle$. Comme $E \Re \langle Q \rangle \neq \perp$, $\text{Prec}(Q) \subseteq E$ et $\Phi_{\#}(Q)$. D'après le Lemme 3, $\forall S \in \text{Lin}(\text{tête}(P)), E \Re P = E \Re S$.
- Supposons que la propriété suivante est vraie pour longueur(P) = n, avec $P = \langle Q_1, ..., Q_n \rangle$:
  $$E \Re P \neq \perp \Rightarrow \forall S_1 \in \text{Lin}(Q_1), ..., \forall S_n \in \text{Lin}(Q_n), E \Re P = E \Re (S_1 \oplus ... \oplus S_n)$$
  Démontrons-la au rang $n+1$, avec $P = \langle Q_1, ..., Q_n, Q_{n+1} \rangle$ et $E \Re P \neq \perp$:

$E \Re P$	
$= E \Re \langle Q_1, ..., Q_n, Q_{n+1} \rangle$	
$= E \Re (\langle Q_1, ..., Q_n \rangle \oplus \langle Q_{n+1} \rangle)$	
$= E \Re \langle Q_1, ..., Q_n \rangle \Re \langle Q_{n+1} \rangle$	d'après la Propriété 2.
$= E \Re (S_1 \oplus ... \oplus S_n) \Re \langle Q_{n+1} \rangle$	$\forall S_1 \in \text{Lin}(Q_1), ..., \forall S_n \in \text{Lin}(Q_n)$, (hyp. de récurrence).
$= E \Re (S_1 \oplus ... \oplus S_n) \Re S_{n+1}$	$\forall S_1 \in \text{Lin}(Q_1), ..., \forall S_n \in \text{Lin}(Q_n), \forall S_{n+1} \in \text{Lin}(Q_{n+1})$, d'après le Lemme 3, car $E \Re P \neq \perp$ implique $\text{Prec}(Q_{n+1}) \subseteq E \Re (S_1 \oplus ... \oplus S_n)$ et $\Phi_{\#}(Q_{n+1})$.
$= E \Re (S_1 \oplus ... \oplus S_n \oplus S_{n+1})$	$\forall S_1 \in \text{Lin}(Q_1), ..., \forall S_n \in \text{Lin}(Q_n), \forall S_{n+1} \in \text{Lin}(Q_{n+1})$, d'après la Propriété 2.

4.2. Relaxation du parallélisme classique : les relations d'autorisation

Comme nous venons de le voir, les conditions imposées sur les actions par la relation d'indépendance sont très fortes : elles permettent de garantir que toutes les exécutions possibles d'un plan mènent au même état résultant, quelles que soient les linéarisations ou le parallélisme des ensembles d'actions du plan. L'idée principale des relations d'autorisation consiste à relâcher partiellement ces conditions, afin d'avoir une plus grande liberté dans le choix des actions pouvant faire partie d'un même ensemble d'actions d'un plan. La production de ce plan devra ainsi en être facilitée.

Mais en faisant une telle modification, nous ne saurons plus si les actions appartenant à un même ensemble d'actions d'un plan (à un même niveau du plan) peuvent ou non être exécutées en parallèle tel qu'on l'entendait jusqu'à présent, puisqu'elles ne seront plus forcément indépendantes. La sémantique de ces plans devra donc être précisée. Cependant, l'idée principale de Graphplan est conservée : les actions que l'on considérera simultanément pour faire partie d'un plan-solution seront traitées comme un tout, c'est-à-dire que la condition d'applicabilité d'un tel ensemble sur un état sera toujours que l'ensemble des préconditions de toutes les actions soit inclus dans cet état, et l'état résultant contiendra toujours l'ensemble des ajouts de toutes les actions. Ceci est dû au fait que l'algorithme d'extraction (cf. section 3.3) recherche un plan possédant certaines propriétés (comme l'indépendance entre actions d'un même ensemble, et maintenant l'autorisation), pouvant être appliqué par l'application $\Re$.

En relaxant une partie des contraintes imposées sur les actions par la relation d'indépendance, nous définissons deux relations plus flexibles, qui ne sont plus symétriques : les relations d'autorisation. L'idée de la première de ces relations, que nous appellerons autorisation forte, a déjà été brièvement introduite dans un cadre différent dans [Dimopoulos, Nebel et Koehler 1997] : le codage d'un problème de planification en logique non monotone et la recherche de modèles stables. Nous étendons ici l'utilisation de cette relation à Graphplan, et montrons comment en conserver les propriétés intéressantes : complétude et terminaison.

4.2.1. Autorisation forte

Le principe de cette relation est le suivant : une action *a* autorise fortement une action *b* si *b* peut toujours être exécutée après *a* et conserve les effets de *a*. De plus, l'exécution simultanée de *a* et *b* produit le même état que l'exécution de *a* puis de *b*. Afin de garantir ce résultat, une propriété plus faible que l'indépendance est suffisante : *a* ne retire aucune précondition de *b* et les effets de *a* et de *b* ne sont pas contradictoires, c'est-à-dire que *a* ne retire aucun ajout de *b* et *b* ne retire aucun ajout de *a*.

Définition 20 (autorisation forte)

Une action *a autorise fortement* une action $b \neq a$ (noté $a \angle b$) ssi :

$$\mathrm{Add}(a) \cap \mathrm{Del}(b) = \emptyset \text{ et } (\mathrm{Prec}(b) \cup \mathrm{Add}(b)) \cap \mathrm{Del}(a) = \emptyset.$$

Une action *a interdit fortement* une action *b* ssi *a* n'autorise pas fortement *b*.

La relation d'autorisation forte conduit à une nouvelle contrainte définissant les ensembles d'actions qui peuvent faire partie d'un plan. En effet, ces ensembles ne seront plus indépendants et nous devons préciser la sémantique des plans que nous voulons obtenir. Une possibilité serait de prendre la même définition que pour l'indépendance d'un ensemble d'actions, en remplaçant l'indépendance entre actions par l'autorisation. Mais la relation d'autorisation forte n'est pas symétrique comme la relation d'indépendance, et cela pose un problème. N'oublions pas que l'on souhaite toujours trouver un plan-solution à un problème de planification de format STRIPS. Par exemple, imaginons le problème du Monde des cubes suivant : dans l'état initial, trois cubes A, B et C sont posés sur la table. L'état initial *I* est donc :

$$I = \{sur{-}table(\mathrm{A}), sur{-}table(\mathrm{B}), sur{-}table(\mathrm{C}), libre(\mathrm{A}), libre(\mathrm{B}), libre(\mathrm{C})\}$$

Le but du problème, notoirement impossible à réaliser, est le suivant :

$$B = \{sur(\mathrm{A}, \mathrm{B}), sur(\mathrm{B}, \mathrm{C}), sur(\mathrm{C}, \mathrm{A})\}$$

Un plan dont l'application par $\Re$ permet de résoudre ce problème est :

$$P = \langle\{\text{DéplacerDeTable(A, B)}, \text{DéplacerDeTable(B, C)}, \text{DéplacerDeTable(C, A)}\}\rangle$$

On s'aperçoit que DéplacerDeTable(B, C) autorise DéplacerDeTable(A, B), mais que DéplacerDeTable(A, B) n'autorise pas DéplacerDeTable(B, C) puisqu'elle retire *libre*(B) qui est une précondition de DéplacerDeTable(B, C). De même, DéplacerDeTable(C, A) autorise DéplacerDeTable(B, C) mais pas l'inverse, et enfin DéplacerDeTable(A, B) autorise DéplacerDeTable(C, A) mais pas l'inverse. On voit donc

que si on prend chaque couple d'actions, on a toujours une des deux actions qui autorise l'autre. Or ce plan mène à un état qui ne peut pas exister ; il faut donc l'interdire. On voit apparaître ici un phénomène de cycle, qui fait que l'on ne peut pas trouver un plan séquentiel à partir de ce plan parallèle : quelle que soit l'action que l'on exécute en premier, elle retire forcément une précondition d'une des deux autres actions, ce qui empêche l'exécution de cette dernière.

Les ensembles d'actions qui nous intéressent seront donc les ensembles pour lesquels une action de chaque couple d'actions autorise l'autre, et pour lesquels il existe au moins une exécution séquentielle possible. L'indépendance d'un ensemble d'actions permettait que toutes les exécutions séquentielles soient possibles ; une seule séquence est maintenant nécessaire. Ceci nous conduit donc aux deux définitions suivantes :

Définition 21 (séquence fortement autorisée, linéarisations fortement autorisées)

Une séquence d'actions $S = \langle a_i \rangle_n$ est *fortement autorisée* ssi $\forall i, j \in [1, n], i < j \Rightarrow a_i \angle a_j$, ce qui équivaut à :

$$\mathrm{Del}(S) \cap \mathrm{Add}(S) = \emptyset \text{ et } \forall i \in [1, n-1], \mathrm{Prec}(a_{i+1}) \cap (\mathrm{Del}(a_1) \cup ... \cup \mathrm{Del}(a_i)) = \emptyset.$$

L'ensemble des *linéarisations fortement autorisées* d'un ensemble d'actions Q, noté $\mathrm{Lin}_\angle(Q)$, est défini par :

$$\mathrm{Lin}_\angle(Q) = \{S \in \mathrm{Lin}(Q) \mid S \text{ est fortement autorisée}\}.$$

Définition 22 (ensemble d'actions fortement autorisé)

Un ensemble d'actions Q est *fortement autorisé* (noté $\Phi_\angle(Q)$) ssi $\mathrm{Lin}_\angle(Q) \neq \emptyset$.

Ainsi, un ensemble d'actions est fortement autorisé si les actions qu'il contient n'ont pas d'effets contradictoires et si on peut trouver un ordre sur ces actions tel qu'aucune action ne retire une précondition d'une action qui la suit. L'intérêt de ces ensembles fortement autorisés est que l'on n'a pas besoin de connaître cet ordre pour calculer l'état résultant de l'application de cet ensemble, et qu'il suffit que l'ensemble des préconditions des actions de l'ensemble soit inclus dans l'état initial pour pouvoir l'appliquer. Ce comportement "global" de l'ensemble est identique à celui des ensembles indépendants, ce qui se traduit par le théorème suivant, proche du Théorème 6 :

Théorème 7 (égalité de l'état résultant par applications parallèles et séquentielles d'un $\Phi_\angle$−plan)

Soit E un état et P un $\Phi_\angle$−plan non vide, avec $P = \langle Q_1, ..., Q_n \rangle$. On a alors :

$$E \,\Re\, P \neq \perp \Rightarrow \forall S_1 \in \mathrm{Lin}_\angle(Q_1), ..., \forall S_n \in \mathrm{Lin}_\angle(Q_n), E \,\Re\, P = E \,\Re\, (S_1 \oplus ... \oplus S_n).$$

Preuve : elle est basée sur le lemme suivant.

Lemme 4

Soit Q un ensemble d'actions tel que $\mathrm{Prec}(Q) \subseteq E$. On a alors :

$$\forall S \in \mathrm{Lin}_\angle(Q), E \,\Re\, \langle Q \rangle = E \,\Re\, S.$$

Preuve :

Soit Q un ensemble d'actions tel que $\mathrm{Prec}(Q) \subseteq E$. Soit $S \in \mathrm{Lin}_\angle(Q)$, avec $S = \langle a_1, ..., a_n \rangle$. S est donc telle que $\mathrm{Del}(S) \cap \mathrm{Add}(S) = \emptyset$, ce qui implique :

$$\forall i \in [1, n-1], \mathrm{Del}(a_{i+1}) \cap (\mathrm{Add}(a_1) \cup ... \cup \mathrm{Add}(a_i)) = \emptyset.$$

Comme on a aussi $\forall i \in [1, n-1], \mathrm{Prec}(a_{i+1}) \cap (\mathrm{Del}(a_1) \cup ... \cup \mathrm{Del}(a_i)) = \emptyset$, alors d'après la Propriété 4, comme $\mathrm{Prec}(Q) \subseteq E$, on a bien $E \,\Re\, \langle Q \rangle = E \,\Re\, S$.

Preuve du Théorème 7 :

Identique à la preuve du Théorème 6, en remplaçant Lemme 3 par Lemme 4, $\Phi_\#$ par $\Phi_\angle$ et Lin par $\mathrm{Lin}_\angle$.

4.2.2. Autorisation faible

Nous sommes allés plus loin dans la relaxation des contraintes entre les actions, en affaiblissant la relation d'autorisation forte. En effet, l'intérêt principal de cette dernière réside dans le fait que l'exécution simultanée de deux actions produit le même état que leur exécution séquentielle. Mais de la façon dont nous avons défini l'application d'une action sur un état, c'est-à-dire en retirant les retraits d'une action puis en ajoutant ses ajouts, on s'aperçoit que l'on peut aller encore plus loin dans la relaxation des contraintes. En effet, supposons

que deux actions a et b sont telles que a autorise fortement b, à l'exception d'un fluent f retiré par a et produit par b. Si l'on exécute simultanément a et b sur un état E, l'état résultant F sera égal à $E \uparrow \{a, b\}$, c'est-à-dire :

$$F = (E - \text{Del}(\{a, b\})) \cup \text{Add}(\{a, b\}).$$

On voit que le fluent f appartient forcément à l'état F, puisqu'il appartient à Add(b) et donc à Add($\{a, b\}$). Si maintenant on exécute l'action a puis l'action b, l'état résultant G est égal à $E \uparrow a \uparrow b$, c'est-à-dire :

$$G = (((E - \text{Del}(a)) \cup \text{Add}(a)) - \text{Del}(b)) \cup \text{Add}(b)$$

On voit que là aussi, le fluent f appartient à l'état G, puisqu'il appartient à Add(b). De plus, on peut démontrer que $F = G$. La relation d'autorisation faible permettra donc qu'une action a autorisant une action b retire des ajouts de b. Ainsi, l'exécution simultanée de a et b conduira toujours au même état résultant. Il faut toutefois noter que l'exécution parallèle de ces deux actions sera impossible (comme pour la relation d'autorisation forte).

Définition 23 (autorisation faible)

Une action *a autorise faiblement* une action $b \neq a$ (noté $a \underline{\angle} b$) ssi :

$$\text{Add}(a) \cap \text{Del}(b) = \emptyset \text{ et } \text{Prec}(b) \cap \text{Del}(a) = \emptyset.$$

Une action *a interdit faiblement* une action b ssi a n'autorise pas faiblement b.

Comme pour l'autorisation forte, nous devons maintenant définir les ensembles d'actions qui nous intéressent pour construire les plans.

Définition 24 (séquence faiblement autorisée, linéarisations faiblement autorisées)

Une séquence d'actions $\langle a_i \rangle_n$ est *faiblement autorisée* ssi $\forall i, j \in [1, n], i < j \Rightarrow a_i \underline{\angle} a_j$, ce qui équivaut à :

$$\forall i \in [1, n-1], \text{Del}(a_{i+1}) \cap (\text{Add}(a_1) \cup ... \cup \text{Add}(a_i)) = \emptyset \text{ et } \text{Prec}(a_{i+1}) \cap (\text{Del}(a_1) \cup ... \cup \text{Del}(a_i)) = \emptyset.$$

L'ensemble des *linéarisations faiblement autorisées* d'un ensemble d'actions Q, noté $\text{Lin}_{\underline{\angle}}(Q)$, est défini par :

$$\text{Lin}_{\underline{\angle}}(Q) = \{S \in \text{Lin}(Q) \mid S \text{ est fortement autorisée}\}.$$

On peut remarquer que cette définition correspond à une partie des prémisses de la Propriété 4, propriété essentielle pour la preuve des théorèmes intéressants sur les plans que nous définissons. Finalement, une séquence faiblement autorisée est telle qu'aucune action ne retire une précondition d'une action qui la suit, ni un ajout d'une action qui la précède. Il s'agit donc de la condition minimale sur les ensembles d'actions leur permettant de vérifier la Propriété 4.

Définition 25 (ensemble d'actions faiblement autorisé)

Un ensemble d'actions Q est *faiblement autorisé* (noté $\Phi_{\underline{\angle}}(Q)$) ssi $\text{Lin}_{\underline{\angle}}(Q) \neq \emptyset$.

Tout comme pour les ensembles fortement autorisés, l'intérêt des ensembles faiblement autorisés est que l'on n'a pas besoin de connaître un ordre sur les actions pour calculer l'état résultant de l'application de cet ensemble, et qu'il suffit que l'ensemble des préconditions des actions de l'ensemble soit inclus dans l'état initial pour pouvoir l'appliquer. Ceci se traduit donc par le théorème suivant :

Théorème 8 (égalité de l'état résultant par applications parallèles et séquentielles d'un $\Phi_{\underline{\angle}}$−plan)

Soit E un état et P un $\Phi_{\underline{\angle}}$−plan non vide, avec $P = \langle Q_1, ..., Q_n \rangle$. On a alors :

$$E \,\Re\, P \neq \perp \Rightarrow \forall S_1 \in \text{Lin}_{\underline{\angle}}(Q_1), ..., \forall S_n \in \text{Lin}_{\underline{\angle}}(Q_n), E \,\Re\, P = E \,\Re\, (S_1 \oplus ... \oplus S_n).$$

Preuve : elle est basée sur le lemme suivant.

Lemme 5

Soit Q un ensemble d'actions tel que $\text{Prec}(Q) \subseteq E$. On a alors :

$$\forall S \in \text{Lin}_{\underline{\angle}}(Q), E \,\Re\, \langle Q \rangle = E \,\Re\, S.$$

Preuve :

> Soit Q un ensemble d'actions tel que $\text{Prec}(Q) \subseteq E$. Soit $S \in \text{Lin}_{\underline{\angle}}(Q)$, avec $S = \langle a_1, ..., a_n \rangle$. S est donc telle que :

$\forall i \in [1, n-1]$, $\text{Del}(a_{i+1}) \cap (\text{Add}(a_1) \cup ... \cup \text{Add}(a_i)) = \varnothing$ et $\text{Prec}(a_{i+1}) \cap (\text{Del}(a_1) \cup ... \cup \text{Del}(a_i)) = \varnothing$.
D'après la Propriété 4, comme $\text{Prec}(Q) \subseteq E$, on a bien $E \,\Re\, \langle Q \rangle = E \,\Re\, S$.

Preuve du Théorème 8 :
Identique à la preuve du Théorème 6 en remplaçant Lemme 3 par Lemme 5, $\Phi_{\#}$ par $\Phi_{\underline{\angle}}$ et Lin par $\text{Lin}_{\underline{\angle}}$.

4.3. Correspondances entre les plans

Nous venons de définir trois relations binaires entre actions, et les contraintes sur les ensembles d'actions qui leur sont associées. L'intérêt du relâchement des contraintes qui permettaient le parallélisme, comme nous le verrons dans la partie expérimentale, est d'améliorer l'efficacité des planificateurs calculant des plans parallèles basés sur l'indépendance entre actions. Si dans un premier temps on relâche certaines contraintes sur les plans, il va falloir dans un second temps s'assurer que l'on ne perd aucune information par rapport à des plans basés sur l'indépendance.

Tout d'abord, on démontre facilement qu'un ensemble d'actions indépendant est fortement autorisé, et qu'un ensemble d'actions fortement autorisé est faiblement autorisé.

Propriété 6
Soit Q un ensemble d'actions. On a alors :

$$\Phi_{\#}(Q) \Rightarrow \Phi_{\angle}(Q) \text{ et } \Phi_{\angle}(Q) \Rightarrow \Phi_{\underline{\angle}}(Q)$$

Preuve :
Immédiate d'après les définitions de $\Phi_{\#}$, $\Phi_{\angle}$ et $\Phi_{\underline{\angle}}$.

Il en découle immédiatement qu'un plan-solution pour l'indépendance est un plan-solution pour l'autorisation forte, et qu'un plan-solution pour l'autorisation forte est un plan-solution pour l'autorisation faible. Ainsi, si un planificateur est complet pour la recherche de plans pour l'autorisation forte ou faible, alors il trouvera forcément tous les plans-solutions possibles pour l'indépendance, c'est-à-dire tous les plans-solutions que trouve un planificateur tel que Graphplan dans sa version originale.

Théorème 9 (équivalence des plans-solutions pour l'indépendance et l'autorisation)
Soit $\prod$ un problème de planification et P un plan. On a alors :

$$P \in \prod(\Phi_{\#}) \Rightarrow P \in \prod(\Phi_{\angle})$$
$$P \in \prod(\Phi_{\angle}) \Rightarrow P \in \prod(\Phi_{\underline{\angle}})$$

Preuve :
Soit $\prod = \langle A, I, B \rangle$ un problème de planification et P un plan tel que $P \in \prod(\Phi_{\#})$. D'après la Propriété 6, on démontre immédiatement que P est un $\Phi_{\angle}$−plan. Comme $P \in \prod(\Phi_{\#})$, $B \subseteq I \,\Re\, P$ donc on a bien $P \in \prod(\Phi_{\angle})$.
De même, on démontre que $P \in \prod(\Phi_{\angle}) \Rightarrow P \in \prod(\Phi_{\underline{\angle}})$.

Et enfin, le résultat essentiel : un plan-solution basé sur l'autorisation forte ou faible peut être transformé en un plan-solution basé sur l'indépendance en recherchant une linéarisation autorisée de chacun des ensembles d'actions qui le composent.

Théorème 10 (transformation de $\Phi_{\angle}$−plans et $\Phi_{\underline{\angle}}$−plans en $\Phi_{\#}$−plans)
Soit $\prod$ un problème de planification et $P = \langle Q_1, ... Q_n \rangle$ un plan. On a alors :

$$P \in \prod(\Phi_{\angle}) \Rightarrow \forall S_1 \in \text{Lin}_{\angle}(Q_1), ..., \forall S_n \in \text{Lin}_{\angle}(Q_n), (S_1 \oplus ... \oplus S_n) \in \prod(\Phi_{\#})$$
$$P \in \prod(\Phi_{\underline{\angle}}) \Rightarrow \forall S_1 \in \text{Lin}_{\underline{\angle}}(Q_1), ..., \forall S_n \in \text{Lin}_{\underline{\angle}}(Q_n), (S_1 \oplus ... \oplus S_n) \in \prod(\Phi_{\#})$$

Preuve :
Soit $\prod$ un problème de planification et $P = \langle Q_1, ... Q_n \rangle$ un plan.
Si $P \in \prod(\Phi_{\angle})$, alors $B \subseteq I \,\Re\, P$. On a donc :
$\forall S_1 \in \text{Lin}_{\angle}(Q_1), ..., \forall S_n \in \text{Lin}_{\angle}(Q_n), I \,\Re\, P = I \,\Re\, (S_1 \oplus ... \oplus S_n)$ d'après le Théorème 7
$\Rightarrow \forall S_1 \in \text{Lin}_{\angle}(Q_1), ..., \forall S_n \in \text{Lin}_{\angle}(Q_n), B \subseteq I \,\Re\, (S_1 \oplus ... \oplus S_n)$
$\Rightarrow \forall S_1 \in \text{Lin}_{\angle}(Q_1), ..., \forall S_n \in \text{Lin}_{\angle}(Q_n), (S_1 \oplus ... \oplus S_n) \in \prod(\Phi_{\#})$

car ($S_1 \oplus ... \oplus S_n$) est une séquence d'actions, et est donc un $\Phi_{\#}$−plan.
De même, on démontre que $\forall S_1 \in \text{Lin}_{\trianglelefteq}(Q_1)$, ..., $\forall S_n \in \text{Lin}_{\trianglelefteq}(Q_n)$, $(S_1 \oplus ... \oplus S_n) \in \prod(\Phi_{\#})$ en utilisant le Théorème 8.

5. La recherche de plans-solutions

Maintenant que nous avons défini plusieurs possibilités pour la gestion du parallélisme, nous allons montrer comment les utiliser concrètement pour la recherche des plans-solutions qui leur correspondent.

5.1. Utilisation de la relation d'indépendance

Plusieurs possibilités s'offrent à nous pour la production des $\Phi_{\#}$−plans-solutions. Les deux principaux aspects à considérer sont les suivants : le coût du calcul de la relation binaire entre actions *R*, qui intervient dans la construction du graphe de planification mais aussi dans l'extraction de la solution, et le coût du calcul de la contrainte $\Phi_{\#}$ pour chaque ensemble d'actions lors de l'extraction de la solution.

5.1.1. Version originale de Graphplan

La solution choisie dans la version originale de Graphplan consiste à calculer la relation *R* de la manière la plus complète possible, afin de trouver le plus possible de mutex entre actions. Une conséquence de ceci est que la vérification de la contrainte $\Phi_{\#}$ est automatiquement réalisée. Nous pouvons donc utiliser directement la relation d'indépendance entre actions #. Il suffit de vérifier qu'elle est $\Phi_{\#}$−compatible :

Propriété 7
La relation # est $\Phi_{\#}$−compatible.

Preuve :
Immédiate, puisque $\Phi_{\#}(Q) \Rightarrow \forall \{a, b\} \subseteq Q, a \neq b \Rightarrow a \# b$.

Nous pouvons donc maintenant donner la version originale de Graphplan :

```
Fonction GP (∏)
Début
    rechercher−plan−GP(∏, Φ#, #)
Fin
```

Algorithme 4 : GP

D'après la Propriété 7 et les théorèmes 4 et 5, la fonction **GP** possède les propriétés de complétude et de terminaison.

Nous avons dit précédemment que le calcul exhaustif de la relation d'indépendance permet d'éviter de vérifier que les ensembles devant faire partie du plan-solution vérifient la contrainte $\Phi_{\#}$. En effet, d'après la définition de $\Phi_{\#}$, $(\forall \{a, b\} \subseteq Q, a \neq b \Rightarrow a \# b) \Rightarrow \Phi_{\#}(Q)$. Donc, comme la fonction **GP** calcule de façon complète la relation d'indépendance entre les actions, la vérification de la contrainte $\Phi_{\#}$ est inutile. Ceci permet de simplifier le début de la fonction **rechercher−actions** de la façon suivante :

```
Fonction rechercher−actions(GP, Φ, B, Q, i, Nogoods)
Début
    Si B = ∅ alors
        P ← résoudre−but(GP, Φ, Prec(Q), i−1, Nogoods) ;
        Si P = échec alors retourner échec
        Sinon retourner P ⊕ ⟨Q⟩
    (...)
```

Algorithme 5 : rechercher−actions (GP)

5.1.2. Version "minimale" de Graphplan

Une autre possibilité consiste, à l'inverse de la précédente, à considérer que toutes les paires d'actions appartiennent à la relation *R*. La relation ainsi définie sera donc trivialement $\Phi_{\#}$−compatible :

Définition 26 (relation $\#_{max}$)
La relation $\#_{max}$ est une relation binaire entre actions définie par :

$$\forall a \neq b,\ a \#_{max} b$$

Propriété 8
La relation $\#_{max}$ est $\Phi_{\#}$−compatible.

Preuve :
Triviale.

```
Fonction GPMin (∏)
Début
    rechercher−plan−GP(∏, Φ#, #max)
Fin
```

Algorithme 6 : GPMin

D'après la Propriété 8 et les théorèmes 4 et 5, la fonction **GPMin** possède les propriétés de complétude et de terminaison.

La version de Graphplan ainsi obtenue est "minimale" en ce sens que le graphe de planification obtenu ne comportera aucune mutex entre actions, donc par conséquent aucune mutex entre fluents. Tout le travail de calcul des contraintes est donc reporté dans la phase d'extraction de la solution, qui en sera considérablement alourdie. En effet, la disparition des exclusions mutuelles entraîne l'augmentation du nombre d'actions que l'on peut utiliser à chaque niveau. Comme on recherche toujours des $\Phi_{\#}$−plans-solutions, le nombre de niveaux nécessaires à l'obtention d'une solution sera le même que pour la fonction **GP**. Le graphe sera donc plus large que pour **GP**, et ne contiendra pas autant d'informations. On peut donc penser que cette version sera beaucoup moins efficace que la précédente.

On peut tout de même simplifier la fonction **rechercher−actions**, en supprimant la vérification des mutex puisqu'il n'y en a plus, et inclure directement le calcul de vérification de $\Phi_{\#}$:

```
Fonction rechercher−actions(GP, Φ, B, Q, i, Nogoods)
Début
    Si B = ∅ alors
        Si (∀{a, b} ⊆ Q, a ≠ b ⇒ a # b) alors
            P ← résoudre−but(GP, Φ, Prec(Q), i−1, Nogoods) ;
            Si P = échec alors retourner échec
            Sinon retourner P ⊕ ⟨Q⟩
        Sinon retourner échec
    Sinon
        choisir f ∈ B ;
        PourTout a tel que (a(i), f(i)) ∈ ArcsAdd(GP) faire
            P ← rechercher−actions(GP, Φ, B − {f}, Q ∪ {a}, i, Nogoods) ;
            Si P ≠ échec alors retourner P ;
        retourner échec
Fin
```

Algorithme 7 : rechercher−actions (GPMin)

5.2. Utilisation des relations d'autorisation

Nous allons maintenant montrer comment utiliser les relations d'autorisation dans Graphplan. Les deux relations d'autorisation que nous avons définies étant très ressemblantes et leur utilisation plus complexe que pour la relation d'indépendance, nous allons décrire étape par étape leur inclusion dans Graphplan.

5.2.1. Construction du graphe de planification

Tout d'abord, nous devons définir les relations binaires que nous emploierons pour la construction du graphe de planification et montrer qu'elles sont Φ−compatibles. En effet, nous ne pouvons pas utiliser directement les relations d'autorisation comme pour l'indépendance : deux noeuds d'actions $a(i)$ et $b(i)$ sont mémorisés comme étant mutuellement exclusifs lors de la construction du graphe si $\neg(a\ R\ b)$ ou $\neg(b\ R\ a)$. En d'autres termes, si on employait directement une relation d'autorisation, $a(i)$ et $b(i)$ seraient mutuellement exclusifs si a n'autorisait pas b ou b n'autorisait pas a. Nous devons donc définir la relation $a\ R\ b$ comme étant : a autorise b ou b autorise a. Ainsi, deux noeuds seront mutuellement exclusifs si [$\neg(a\ R\ b)$ ou $\neg(b\ R\ a)$], c'est-à-dire a n'autorise pas b ou b n'autorise pas a.

Commençons par définir la relation binaire pour l'utilisation de la relation d'autorisation forte :

Définition 27 (relation $R_{\angle}$)

La relation $R_{\angle}$ est une relation binaire entre actions définie par :

$$\forall a \neq b,\ a\ R_{\angle}\ b \Leftrightarrow (a \angle b) \text{ ou } (b \angle a)$$

Propriété 9

La relation $R_{\angle}$ est $\Phi_{\angle}$−compatible.

Preuve :

Immédiate, puisque :
$\Phi_{\angle}(Q)$
$\Rightarrow \exists\ S \in \text{Lin}_{\angle}(Q)$, telle que $S = \langle a_i \rangle_n$ et $\forall i, j \in [1, n],\ i < j \Rightarrow a_i \angle a_j$
$\Rightarrow \forall \{a, b\} \subseteq Q,\ a \neq b \Rightarrow (a \angle b)$ ou $(b \angle a)$
$\Rightarrow \forall \{a, b\} \subseteq Q,\ a \neq b \Rightarrow a\ R_{\angle}\ b.$

De même, nous définissons la relation binaire pour l'utilisation de la relation d'autorisation faible :

Définition 28 (relation $R_{\underline{\angle}}$)

La relation $R_{\underline{\angle}}$ est une relation binaire entre actions définie par :

$$\forall a \neq b,\ a\ R_{\underline{\angle}}\ b \Leftrightarrow (a \mathrel{\underline{\angle}} b) \text{ ou } (b \mathrel{\underline{\angle}} a)$$

Propriété 10

La relation $R_{\underline{\angle}}$ est $\Phi_{\underline{\angle}}$−compatible.

Preuve :

Immédiate, puisque :
$\Phi_{\underline{\angle}}(Q)$
$\Rightarrow \exists\ S \in \text{Lin}_{\underline{\angle}}$, telle que $S = \langle a_i \rangle_n$ et $\forall i, j \in [1, n],\ i < j \Rightarrow a_i \mathrel{\underline{\angle}} a_j$
$\Rightarrow \forall \{a, b\} \subseteq Q,\ a \neq b \Rightarrow (a \mathrel{\underline{\angle}} b)$ ou $(b \mathrel{\underline{\angle}} a)$
$\Rightarrow \forall \{a, b\} \subseteq Q,\ a \neq b \Rightarrow a\ R_{\underline{\angle}}\ b.$

Ces définitions de la relation R utilisée pour la construction du graphe impliquent que l'on va trouver moins de paires de noeuds d'actions mutuellement exclusifs qu'avec la version originelle de Graphplan (fonction **GP**). Dans le pire des cas, on en trouvera autant. Par conséquent, un niveau n contiendra plus d'actions qu'un même niveau n pour **GP** car les actions pourront être employées plus tôt. Ce phénomène ressemble à celui de la version "minimale" de Graphplan (fonction **GPMin**), à la différence que pour les relations d'autorisation, un graphe d'ordre k contiendra des $\Phi_{\angle}$−plans ou des $\Phi_{\underline{\angle}}$−plans qui pourront être transformés en $\Phi_{\#}$−plans de longueur supérieure à k. En d'autres termes, un graphe de planification construit à l'aide d'une relation d'autorisation comportant n niveaux nous servira à trouver des $\Phi_{\#}$−plans de longueur supérieure ou égale à k : il contiendra donc plus de plans-solutions potentiels qu'un graphe comportant k niveaux pour **GP**.

5.2.2. Extraction de la solution

Nous allons maintenant détailler la vérification de l'autorisation d'un ensemble d'actions dans la fonction **rechercher−actions**. Si R est une relation d'autorisation, un ensemble d'actions Q est autorisé si et seulement si l'ensemble des linéarisations de Q pour R n'est pas vide. Cette contrainte peut être vérifiée en utilisant un algorithme de tri topologique modifié (linéaire par rapport au nombre de noeuds et d'arcs, cf. [Melhorn 1984]) qui teste si le graphe suivant contient un circuit :

Définition 29 (graphe d'autorisation)
Soit $Q = \{a_1, ..., a_n\}$ un ensemble d'actions et R une relation d'autorisation. Le graphe d'autorisation $GA_R(N, C)$ de Q pour la relation R est un graphe orienté défini par :

- N est l'ensemble des noeuds tels que pour chaque action a_i il existe un seul noeud de N noté $n(a_i)$: $N = \{n(a_1), ..., n(a_n)\}$,
- C est l'ensemble des arcs qui représentent les contraintes d'ordre entre les actions : il y a un arc de $n(a)$ vers $n(b)$ ssi l'exécution de a **doit précéder** l'exécution de b, c'est-à-dire :

$$\forall \{a, b\} \subseteq Q, (n(a), n(b)) \in C \Leftrightarrow a \neq b \text{ et } \neg(b\ R\ a).$$

En effet, on peut démontrer que :

Théorème 11 (vérification de l'autorisation d'un ensemble d'actions)
Soient Q un ensemble d'actions, R une relation d'autorisation et $GA_R(N, C)$ le graphe d'autorisation de Q pour la relation R. On a alors :

$$GA_R \text{ ne contient pas de circuit} \Leftrightarrow \Phi_R(Q)$$

Preuve :

Soient Q un ensemble d'actions, R une relation d'autorisation et $GA_R(N, C)$ le graphe d'autorisation de Q pour la relation R. Nous savons que GA_R n'a pas de circuit ssi il existe un ordre topologique sur les noeuds de GA_R (ce qui implique un ordre sur les actions) :
$N = \{n(a_1), ..., n(a_n)\}$ avec $\forall\ 1 \leq i < j \leq n, (n(a_j), n(a_i)) \notin C$
$\Leftrightarrow \forall\ 1 \leq i < j \leq n, \neg(\neg(a_i\ R\ a_j))$
$\Leftrightarrow \forall\ 1 \leq i < j \leq n, a_i\ R\ a_j$
$\Leftrightarrow \Phi_R(Q)$.

Nous allons donc utiliser l'algorithme **stratifier** ci-dessous pour prouver qu'un ensemble d'actions est autorisé. Mais cet algorithme ne se contente pas de répondre à la question "est-ce que l'ensemble d'actions Q est autorisé ?" (cf. Théorème 12, page 76) ; il retourne aussi un $\Phi_{\#}$−plan P tel que $E\ \Re\ \langle Q \rangle = E\ \Re\ P$ (cf. Théorème 13, page 77). Nous avons divisé cet algorithme de tri topologique en deux fonctions, car la seconde (**tri−topologique**) sera utilisée plus tard par l'algorithme qui recherche le réordonnancement optimal d'un plan-solution.

```
Fonction stratifier(Q, R)
;; Entrée :
;; − Q : un ensemble d'actions.
;; − R : une relation d'autorisation.
;; Sortie :
;; − échec si ¬Φ_R(Q)
;; − sinon : un Φ_#−plan P tel que E ℜ ⟨Q⟩ = E ℜ P.
Début
    GA_R(N, C) ← le graphe d'autorisation de Q pour la relation R ;
    retourner tri−topologique(GA_R)
Fin
```

Algorithme 8 : stratifier

Fonction tri−topologique(G(*N*, *C*))
;; Entrée :
;; − G(*N*, *C*) : un graphe orienté. *N* est l'ensemble des noeuds associés aux actions,
;; *C* est l'ensemble des arcs.
;; Sortie :
;; − *échec* si G contient un circuit.
;; − sinon : un $\Phi_{\#}$−plan *P* correspondant au tri topologique modifié des noeuds de *N*.
Début
sans−pred ← ∅ ;
P ← ⟨⟩ ;
TantQue $N \neq \emptyset$ **faire**
$sans-pred \leftarrow \{n_1 \in N \mid \forall n_2 \in N, (n_2, n_1) \notin C)\}$;
Si *sans−pred* = ∅ **alors retourner** *échec* ;
$P \leftarrow P \oplus \langle \{a \mid n(a) \in sans-pred\} \rangle$;
$N \leftarrow N - sans-pred$;
$C \leftarrow C - \{(n_1, n_2) \in C \mid n_1 \in sans-pred\}$;
retourner *P*
Fin

Algorithme 9 : tri−topologique

Maintenant, nous devons démontrer que le résultat fourni par cet algorithme est correct :

Théorème 12 (validité de la fonction stratifier)
Soient *Q* un ensemble d'actions et *R* une relation d'autorisation. On a alors :

$$\Phi_R(Q) \Leftrightarrow \textbf{stratifier}(Q, R) \neq \textit{échec}$$

Preuve :

Soient *Q* un ensemble d'actions et *R* une relation d'autorisation. Soit $GA_R(N, C)$ le graphe d'autorisation de *Q* pour la relation *R*.
Montrons d'abord que $\neg\Phi_R(Q) \Rightarrow$ **stratifier**(*Q*, *R*) = *échec*.
Si $\neg\Phi_R(Q)$, alors d'après le Théorème 11 GA_R contient un circuit. L'algorithme va le détecter et retourner *échec*, car lors d'une itération chaque noeud présent dans le graphe aura au moins un prédécesseur. En effet, à chaque itération, l'algorithme détecte tous les noeuds sans prédécesseurs et les retire du graphe ; or les noeuds d'un circuit ont au moins un prédécesseur. Ces noeuds ne peuvent donc pas être retirés.
Montrons ensuite que **stratifier**(*Q*, *R*) = *échec* $\Rightarrow \neg\Phi_R(Q)$.
Si l'algorithme retourne *échec*, on est dans la situation suivante : on essaie de retirer les noeuds sans prédécesseurs d'un graphe $GA_R'(N', C')$, avec $N' \subseteq N$, $N' \neq \emptyset$ et $C' \subseteq C$. Comme $N' \neq \emptyset$, il reste des noeuds dans GA_R'. Mais comme ces noeuds ont au moins un prédécesseur, GA_R' contient un circuit. D'après le Théorème 11, l'ensemble $Q' = \{a \mid n(a) \in N'\}$ est tel que $\neg\Phi_R(Q')$. Mais comme $Q' \subseteq Q$, on peut en conclure $\neg\Phi_R(Q)$.

5.2.3. Transformation de la solution

Nous allons maintenant montrer comment transformer un Φ_R−plan, *R* étant une relation d'autorisation, en un $\Phi_{\#}$−plan de longueur minimale. En effet, par rapport à la sémantique usuelle du parallélisme entre actions, les actions d'un Φ_R−plan sont insuffisamment ordonnées pour être directement exécutées à cause des interactions possibles entre elles. La transformation que nous étudions est composée de deux étapes :

1. Le Φ_R−plan est tout d'abord transformé en un $\Phi_{\#}$−plan en utilisant directement la fonction **stratifier** sur chacun des ensembles d'actions qui composent le Φ_R−plan, et en concaténant les plans obtenus.
2. Le $\Phi_{\#}$−plan ainsi obtenu est ensuite réordonnancé de façon optimale en utilisant l'algorithme polynomial de [Régnier et Fade 1991], qui a été révisé et formalisé par [Bäckström 1998]. En effet, Bäckström démontre que cet algorithme calcule un réordonnancement optimal en nombre de niveaux du plan, c'est-à-dire en nombre d'ensembles d'actions indépendants.

Le détail de ces deux étapes est donné dans les deux paragraphes suivants.

Transformation en un $\Phi_{\#}$−plan

Nous savons que l'on peut utiliser la fonction **stratifier** pour tester l'autorisation d'un ensemble d'actions ; nous devons maintenant prouver que les Φ_R−plans qu'il retourne peuvent être utilisés pour transformer un Φ_R−plan en un $\Phi_{\#}$−plan.

Théorème 13 (transformation en $\Phi_{\#}$−plan par la fonction stratifier)
Soient E un état, $P = \langle Q_1, ..., Q_n \rangle$ un plan et R une relation d'autorisation tels que P est un Φ_R−plan. On a alors :

$$E \Re P = E \Re (\textbf{stratifier}(Q_1, R) \oplus ... \oplus \textbf{stratifier}(Q_n, R))$$

Preuve : elle est basée sur les deux lemmes suivants.

Lemme 6
Soient Q un ensemble d'actions et R une relation d'autorisation, tels que **stratifier**$(Q, R) = \langle Q_1, ..., Q_n \rangle$. On a alors :

$$\forall S_1 \in \text{Lin}(Q_1), ..., \forall S_n \in \text{Lin}(Q_n), S_1 \oplus ... \oplus S_n \in \text{Lin}_R(Q)$$

Preuve :

Soit Q un ensemble d'actions et R une relation d'autorisation, tels que **stratifier**$(Q, R) = \langle Q_1, ..., Q_n \rangle$. Soient $S_1 \in \text{Lin}(Q_1), ..., S_n \in \text{Lin}(Q_n)$.
La solution retournée par la fonction **stratifier** est telle que $Q = Q_1 \cup ... \cup Q_n$, avec $Q_1 \cap ... \cap Q_n = \varnothing$. En effet, **stratifier** place chaque action de l'ensemble initial Q dans un et un seul ensemble Q_i ; on peut donc en déduire que $S_1 \oplus ... \oplus S_n \in \text{Lin}(Q)$.
Nous devons maintenant prouver que $S_1 \oplus ... \oplus S_n \in \text{Lin}_R(Q)$.
Supposons que $S_1 \oplus ... \oplus S_n \notin \text{Lin}_R(Q)$. Nous avons alors deux possibilités :

- Soit $\exists i \in [1, n]$ tel que $S_i = \langle a_1, ..., a_m \rangle$ et $\exists a_j, a_k \in S_i$ avec $j < k$, telles que $\neg(a_j \, R \, a_k)$.
  Nous avons alors $a_j \in Q_i$ et $a_k \in Q_i$. Par construction, si ces deux actions sont dans le même ensemble d'actions, c'est parce que leur noeuds respectifs dans le graphe d'autorisation n'ont pas de prédécesseur dans la même itération. En particulier, n(a_k) n'est pas un prédécesseur de n(a_j). Nous avons donc $\neg(\neg(a_j \, R \, a_k))$, c'est-à-dire $a_j \, R \, a_k$: il y a contradiction.
- Soit $\exists i, j \in [1, n]$ tel que $i < j$, et $\exists a \in S_i$, $\exists b \in S_j$ telles que $\neg(a \, R \, b)$.
  Mais si a est dans dans un ensemble d'actions qui a été trouvé par l'algorithme avant celui qui contient b, c'est parce que a n'a pas de prédécesseur dans le graphe d'autorisation dans lequel b est présent. On a donc n(b) ne précède pas n(a), c'est-à-dire $\neg(\neg(a \, R \, b))$, et donc $a \, R \, b$: il y a contradiction.

Lemme 7
Soit E un état, Q un ensemble d'actions et R une relation d'autorisation, tels que $\Phi_R(Q)$. On a alors :

$$E \Re \langle Q \rangle \neq \bot \Rightarrow E \Re \langle Q \rangle = E \Re \textbf{stratifier}(Q, R).$$

Preuve :

Soit E un état, Q un ensemble d'actions et R une relation d'autorisation, tels que $\Phi_R(Q)$ et $E \Re \langle Q \rangle \neq \bot$. Comme $\Phi_R(Q)$, alors d'après le Théorème 12 : **stratifier**$(Q, R) \neq$ *échec* et donc **stratifier**$(Q, R) = \langle Q_1, ..., Q_n \rangle$. On a alors :

$E \Re$ **stratifier**(Q, R)
$= E \Re \langle Q_1, ..., Q_n \rangle$
$= E \Re (S_1 \oplus ... \oplus S_n)$ $\quad \forall S_1 \in \text{Lin}(Q_1), ..., \forall S_n \in \text{Lin}(Q_n)$, d'après le Théorème 6.
$= E \Re S'$ $\quad$ avec $S' \in \text{Lin}_R(Q)$, d'après le Lemme 6.

Comme $E \Re \langle Q \rangle \neq \bot$, on a forcément $\text{Prec}(Q) \subseteq E$, donc d'après les lemmes 4 et 5 :

$$\forall S \in \text{Lin}_R(Q), E \Re \langle Q \rangle = E \Re S$$

On peut en déduire que :

$$E \Re \langle Q \rangle = E \Re S' = E \Re \textbf{stratifier}(Q, R).$$

Preuve du Théorème 13 :

Soient E un état, $P = \langle Q_1, ..., Q_n \rangle$ un plan et R une relation d'autorisation tels que P est un Φ_R−plan.
Nous pouvons faire une preuve par récurrence sur la longueur de P.

- Quand longueur(P) = 1: $E \,\Re\, P = E \,\Re$ **stratifier**(tête(P), R) d'après le Lemme 7.
- Supposons que la propriété suivante soit vraie au rang n, avec $P = \langle Q_1, ..., Q_n \rangle$ qui est un Φ_R−plan :
  - $E \,\Re\, P = E \,\Re$ (**stratifier**(Q_1, R) ⊕ ... ⊕ **stratifier**(Q_n, R))

  Démontrons-la au rang n+1, avec $P' = \langle Q_1, ..., Q_{n+1} \rangle$ qui est un Φ_R−plan :

$E \,\Re\, P'$	
$= E \,\Re \langle Q_1, ..., Q_n, Q_{n+1} \rangle$	
$= E \,\Re \langle Q_1, ..., Q_n \rangle \oplus \langle Q_{n+1} \rangle$	
$= E \,\Re \langle Q_1, ..., Q_n \rangle \,\Re \langle Q_{n+1} \rangle$	d'après la Propriété 2
$= E \,\Re$ (**stratifier**(Q_1, R) ⊕ ... ⊕ **stratifier**(Q_n, R)) $\Re \langle Q_{n+1} \rangle$	(hyp. de récurrence)
$= E \,\Re$ (**stratifier**(Q_1, R) ⊕ ... ⊕ **stratifier**(Q_n, R)) $\Re$ **stratifier**(Q_{n+1}, R)	d'après le Lemme 7
$= E \,\Re$ (**stratifier**(Q_1, R) ⊕ ... ⊕ **stratifier**(Q_n, R) ⊕ **stratifier**(Q_{n+1}, R))	d'après la Propriété 2

Il est bien évident qu'un $\Phi_\#$−plan obtenu par ce calcul est un $\Phi_\#$−plan-solution du problème de planification pour lequel on l'a cherché :

Corollaire 1

Soit $\prod$ un problème de planification, $P = \langle Q_1, ..., Q_n \rangle$ un plan et R une relation d'autorisation. On a alors :

$$P \in \prod(\Phi_R) \Rightarrow \textbf{stratifier}(Q_1, R) \oplus ... \oplus \textbf{stratifier}(Q_n, R) \in \prod(\Phi_\#)$$

Preuve :

Immédiate d'après le Théorème 13.

On peut noter que comme tous les ensembles d'actions d'un Φ_R−plan doivent être autorisés pour faire partie du plan-solution en utilisant la fonction **stratifier**, nous pouvons "mémoizer" pour chaque ensemble d'actions et à chaque niveau le résultat de ce test ; ainsi, quand un plan-solution sera trouvé, nous pourrons éviter de faire la transformation décrite ci-dessus et calculer directement le réordonnancement optimal du plan comme indiqué dans le paragraphe suivant.

Recherche du réordonnancement optimal

Comme on vient de le montrer, le $\Phi_\#$−plan calculé en appliquant la fonction **stratifier** sur chacun des ensembles d'actions qui le compose est un plan-solution du problème de planification pour lequel on l'a cherché : on pourrait se contenter de le retourner tel quel. Mais ce plan peut être facilement optimisé en utilisant l'algorithme PRF (cf. [Régnier et Fade 1991], révisé et formalisé par [Bäckström 1998], page 119) afin de trouver un réordonnancement optimal de ce plan en nombre de niveaux du plan (c'est-à-dire en nombre d'ensemble d'actions indépendants).

Cette étape est composée de deux parties, comme la vérification de la contrainte d'autorisation pour un ensemble d'actions. Tout d'abord, on construit un graphe qui représente les contraintes entre les actions du plan (c'est-à-dire les relations d'ordre et d'indépendance entre actions). Nous utilisons ensuite la fonction **tri−topologique** sur ce graphe, qui trouve le $\Phi_\#$−plan correspondant à ce graphe. Ce $\Phi_\#$−plan est minimal en nombre de niveaux, et son application sur un état retourne le même résultat que l'application du plan originel sur ce même état.

Définition 30 (graphe d'ordre partiel)

Soit $P = \langle Q_1, ... Q_n \rangle$ un $\Phi_\#$−plan. Le *graphe d'ordre partiel* GOP(N, C) de P est un graphe dirigé défini par :

- N est l'ensemble des noeuds tels que pour chaque action $a \in Q_i$, $\forall i \in [1, n]$, il existe un seul noeud de N noté n(a),
- C est l'ensemble des arcs qui représentent les contraintes d'ordre entre les actions : il y a un arc de n(a) vers n(b) ssi l'exécution de a **doit précéder** l'exécution de b, c'est-à-dire :

$$(\text{n}(a), \text{n}(b)) \in C \Leftrightarrow (a \in Q_k, b \in Q_p, 1 \le k < p \le n \text{ et } (\neg(b \trianglelefteq a) \text{ ou } \text{Add}(a) \cap \text{Prec}(b) \neq \varnothing))$$

L'algorithme suivant calcule le $\Phi_\#$−plan correspondant au réordonnancement optimal d'un Φ_R−plan P passé en paramètre, avec R une relation d'autorisation. Il procède en trois étapes : transformation de P en un $\Phi_\#$−plan P' par l'application de la fonction **stratifier** sur chacun des ensembles constituant P, construction du

graphe d'ordre partiel de *P*', puis utilisation de la fonction **tri−topologique** sur ce graphe pour calculer le réordonnancement optimal du plan. Toutes les preuves se trouvent dans [Bäckström 1998].

Fonction réordonnancer−plan(*P*, *R*)
;; Entrée :
;; − *P* : un Φ_R−plan.
;; − *R* : une relation d'autorisation.
;; Sortie :
;; − le réordonnancement optimal de *P*.
Début
    Si *P* = ⟨⟩ **alors retourner** *P* ;
    ;; avec *P* = ⟨Q_1, ..., Q_n⟩ :
    P' ← (**stratifier**(Q_1, *R*) ⊕ ... ⊕ **stratifier**(Q_n, *R*)) ;
    GOP(*N*, *C*) ← le graphe d'ordre partiel de *P*' ;
    retourner tri−topologique(GOP)
Fin

Algorithme 10 : réordonnancer−plan

Théorème 14 (réordonnancement d'un plan-solution)
Soient *E* un état, *R* une relation d'autorisation et *P* un Φ_R−plan. On a alors :

$$E \Re P = E \Re (\textbf{réordonnancer−plan}(P, R))$$

Preuve :
Voir [Bäckström 1998].

Corollaire 2
Soit $\prod$ un problème de planification, *P* un plan et *R* une relation d'autorisation. On a alors :

$$P \in \prod(\Phi_R) \Rightarrow \textbf{réordonnancer−plan}(P, R) \in \prod(\Phi_\#)$$

Preuve :
Immédiate d'après le Théorème 14.

Une autre solution pour transformer un Φ_R−plan en un $\Phi_\#$−plan est de le transformer directement en ajoutant la condition suivante dans la construction du graphe d'ordre partiel : il y a un arc entre une action *a* et une action *b* qui appartiennent à un même ensemble autorisé si l'action *b* n'autorise pas l'action *a*. Il s'agit de la même condition de présence d'un arc dans le graphe que pour la construction du graphe d'autorisation. Cette version a été présentée dans [Cayrol, Régnier et Vidal 2000b].

5.2.4. Synthèse

Nous pouvons maintenant présenter nos nouvelles versions de Graphplan qui utilisent les relations d'autorisation pour produire des $\Phi_\#$−plans-solutions. Les noms de ces générateurs de plans seront préfixés par LCGP, pour "Least Committed GraphPlan". "Least Commitment" en anglais signifie "moindre engagement" : il s'agit d'une des idées de base de la planification dans les espaces de plans, qui nous a fortement inspiré ce travail.

En un sens, la version originelle de Graphplan est déjà un planificateur qui utilise une stratégie de moindre engagement, puisqu'il produit des plans parallèles minimaux en nombre d'ensembles d'actions parallèles (indépendantes). Nous poussons encore plus loin cette idée de moindre engagement, et montrons que l'on peut construire des planificateurs complets basés sur Graphplan différant le choix de l'ordonnancement des actions dans la phase d'extraction de la solution.

Par rapport aux approches de [Smith et Peot 1993] et [Joslin et Pollack 1996] (cf. section II.2.4.2, page 21) sur la prise en compte tardive de décisions dans la planification dans les espaces de plans partiels, l'intérêt de notre approche est que :

- la détection de conflits dont on peut remettre à plus tard la résolution, effectuée pendant la construction du graphe de planification,

- la vérification de l'existence potentielle d'une solution pendant la recherche grâce à la fonction **stratifier**,
- et enfin la prise en compte effective de ces conflits pour retourner un $\Phi_{\#}$−plan-solution grâce à la fonction **réordonnancer−plan**,

sont des opérations polynomiales par rapport à la taille du problème.

Les deux premières versions de LCGP calculent de façon complète les relations binaires d'autorisation entre actions pendant la construction du graphe de planification, en utilisant les relations $R_{\angle}$ et $R_{\underline{\angle}}$. Tout comme pour la fonction **GP**, la vérification de la contrainte d'autorisation des ensembles d'actions en sera facilitée pendant l'extraction de la solution. La première version de LCGP utilise la relation d'autorisation forte :

Fonction LCGP$_{\angle}$($\prod$)
Début
    $P \leftarrow$ rechercher−plan−GP($\prod$, $\Phi_{\angle}$, $R_{\angle}$) ;
    Si P = *échec* **alors retourner** *échec* ;
    retourner réordonnancer−plan(P, $\angle$)
Fin

Algorithme 11 : LCGP$_{\angle}$

D'après la Propriété 9, les théorèmes 4 et 5 et le Corollaire 2, **LCGP**$_{\angle}$ possède les propriétés de complétude et de terminaison pour la recherche de $\Phi_{\#}$−plans-solutions.

La deuxième version de LCGP utilise la relation d'autorisation faible :

Fonction LCGP$_{\underline{\angle}}$($\prod$)
Début
    $P \leftarrow$ rechercher−plan−GP($\prod$, $\Phi_{\underline{\angle}}$, $R_{\underline{\angle}}$) ;
    Si P = *échec* **alors retourner** *échec* ;
    retourner réordonnancer−plan(P, $\underline{\angle}$)
Fin

Algorithme 12 : LCGP$_{\underline{\angle}}$

D'après la Propriété 10, les théorèmes 4 et 5 et le Corollaire 2, **LCGP**$_{\underline{\angle}}$ possède les propriétés de complétude et de terminaison pour la recherche de $\Phi_{\#}$−plans-solutions.

Les deux autres versions de LCGP utilisent la relation binaire $\#_{max}$, comme la fonction **GPMin**. En effet, cette relation est trivialement $\Phi_{\angle}$−compatible et $\Phi_{\underline{\angle}}$−compatible :

Propriété 11
La relation $\#_{max}$ est $\Phi_{\angle}$−compatible et $\Phi_{\underline{\angle}}$−compatible.

Preuve :
Triviale.

Fonction LCGPMin$_{\angle}$($\prod$)
Début
    $P \leftarrow$ rechercher−plan−GP($\prod$, $\Phi_{\angle}$, $\#_{max}$) ;
    Si P = *échec* **alors retourner** *échec* ;
    retourner réordonnancer−plan(P, $\angle$)
Fin

Algorithme 13 : LCGPMin$_{\angle}$

D'après la Propriété 11, les théorèmes 4 et 5 et le Corollaire 2, **LCGPMin**$_{\angle}$ possède les propriétés de complétude et de terminaison pour la recherche de $\Phi_{\#}$−plans-solutions.

Fonction LCGPMin$_{\trianglelefteq}$($\prod$)
Début
    $P \leftarrow$ rechercher−plan−GP($\prod$, $\Phi_{\trianglelefteq}$, $\#_{max}$) ;
    Si P = *échec* **alors retourner** *échec* ;
    retourner réordonnancer−plan(P, $\trianglelefteq$)
Fin

Algorithme 14 : LCGPMin$_{\trianglelefteq}$

D'après la Propriété 11, les théorèmes 4 et 5 et le Corollaire 2, **LCGPMin$_{\trianglelefteq}$** possède les propriétés de complétude et de terminaison pour la recherche de $\Phi_{\#}$−plans-solutions.

Les quatre versions de LCGP que nous venons de définir doivent vérifier la contrainte d'autorisation forte ou faible sur les ensembles d'actions qui peuvent faire partie d'un plan-solution. Le début de la fonction **rechercher−actions** peut donc être réécrit de la façon suivante :

Fonction rechercher−actions(*GP*, Φ, *B*, *Q*, *i*, Nogoods)
Début
    Si $B = \emptyset$ **alors**
        Si ($\Phi = \Phi_{\trianglelefteq}$ et stratifier(Q, $\trianglelefteq$) ≠ *échec*) ou ($\Phi = \Phi_{\trianglelefteq}$ et stratifier(Q, $\trianglelefteq$) ≠ *échec*) **alors**
            $P \leftarrow$ résoudre−but(*GP*, Φ, Prec(Q), i−1, Nogoods) ;
            Si P = *échec* **alors retourner** *échec*
            Sinon retourner $P \oplus \langle Q \rangle$
        Sinon retourner *échec*
    (...)

Algorithme 15 : rechercher−actions (LCGP)

5.3. Exemple

Nous allons maintenant reprendre l'exemple traité avec Graphplan dans la section 1, et le traiter avec LCGP$_{\trianglelefteq}$. La description du domaine, du problème, des fluents et des actions de base reste la même (voir les figures 4, 5 et 6, pages 42 et 43).

5.3.1. L'expansion du graphe de planification

Tout d'abord, nous allons créer le graphe de planification initial (Figure 20). Ce graphe ne change pas par rapport à celui de la section 1.

Figure 20 : Le graphe de planification initial

Le fluent qui représente le but (**pao**) n'étant pas présent dans les fluents du niveau 0, nous devons continuer à développer le graphe en créant le niveau 1 (voir la Figure 21). La construction est identique à ce que nous avions déjà vu : création des no-ops à partir des propositions du niveau 0, création dans le niveau 1 des actions applicables, création des fluents de leur ensemble des ajouts, création des arcs entre les préconditions situées dans le niveau 0 et les actions qui les utilisent, et enfin création des arcs d'ajout et de retrait entre les actions du niveau 1 et leurs effets.

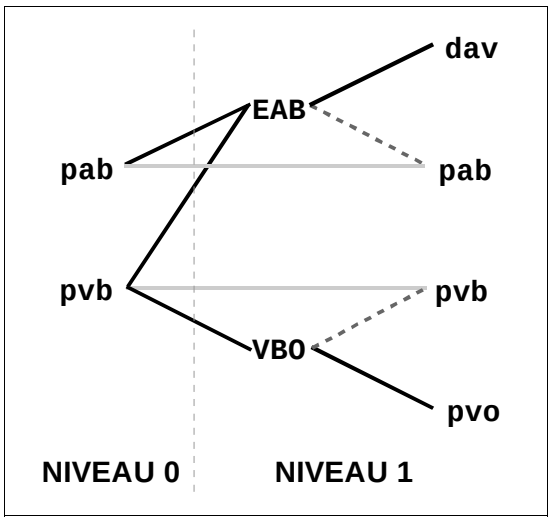

Figure 21 : Création du niveau 1

Après construction de ce niveau, on ne voit toujours pas de différence par rapport à ce même graphe développé par Graphplan (voir la Figure 8). La différence va se situer au niveau du calcul des mutex entre actions (voir la Figure 22 pour un récapitulatif des mutex) :

- L'action **EAB** retire **pab**, qui est une précondition de l'action **N_pab**. On a donc ¬(**EAB** $\trianglelefteq$ **N_pab**). De plus, **pab** est un ajout de **N_pab**, on a donc ¬(**N_pab** $\trianglelefteq$ **EAB**). Les actions **EAB** et **N_pab** interfèrent, donc la paire {**EAB**, **N_pab**} forment une mutex. Ces deux actions formaient déjà une mutex avec Graphplan.
- De la même façon, la paire {**VBO**, **N_pvb**} est une mutex, car **VBO** retire **pvb** qui est à la fois une précondition et un ajout de l'action **N_pvb**. Ces deux actions formaient déjà une mutex avec Graphplan.
- L'action **VBO** retire le fluent **pvb**, qui est la précondition de l'action **EAB**. Avec Graphplan, ces deux actions formaient une mutex. Mais maintenant, on peut voir que :
  Add(**EAB**) ∩ Del(**VBO**) = {**dav**} ∩ {**pvb**} = ∅
  Prec(**VBO**) ∩ Del(**EAB**) = {**pvb**} ∩ {**pab**} = ∅
  On peut en conclure que l'action **EAB** autorise l'action **VBO** (**EAB** $\trianglelefteq$ **VBO**). Ces deux actions n'interfèrent pas et ne forment donc pas une mutex : l'action **EAB** peut très bien être exécutée avant l'action **VBO**.

Nous ne trouvons donc que deux mutex sur les actions, contrairement à Graphplan qui en trouvait trois. Cette différence va maintenant se répercuter sur les mutex entre fluents :

- Les actions qui établissent le fluent **dav** (l'action **EAB**) sont toujours mutuellement exclusives avec les actions qui établissent le fluent **pab** (l'action **N_pab**). La paire {**dav**, **pab**} est donc une mutex : on retrouve le même résultat qu'avec Graphplan.
- Les actions qui établissent le fluent **pvo** (l'action **VBO**) sont toujours mutuellement exclusives avec les actions qui établissent le fluent **pvb** (l'action **N_pvb**). La paire {**pvb**, **pvo**} est donc une mutex : on retrouve le même résultat qu'avec Graphplan.
- Les actions qui établissent le fluent **dav** (l'action **EAB**) ne sont plus mutuellement exclusives avec les actions qui établissent le fluent **pvo** (l'action **VBO**). *La paire* {**dav**, **pvo**} *n'est donc pas une mutex, contrairement à ce que l'on trouvait avec Graphplan.*

Si le niveau 1 construit par LCGP est identique à celui construit par Graphplan, on peut déjà se douter que cela ne va pas être le cas pour le niveau 2. En effet, ce dernier dépend des mutex sur les fluents du niveau 1, qui sont différentes de celles que l'on trouvait avec Graphplan.

```
Mutex entre fluents :
  dav – pab
  pvb – pvo

Mutex entre actions :
  EAB – N_pab
  VBO – N_pvb
```

Figure 22 : Les mutex du niveau 1

En particulier (et c'est là un point très intéressant), *les fluents* **dav** *et* **pvo** *ne formant pas une mutex, nous allons pouvoir utiliser l'action* **DAO**. Or, cette action produit le fluent **pao**, qui est le but de notre problème. Avec Graphplan, nous n'avions pas pu utiliser l'action **DAO** au niveau 2 : nous avions dû attendre le niveau 3 pour pouvoir nous en servir. Nous allons donc pouvoir essayer d'extraire un plan-solution au niveau 2.

Nous ne savons pas encore si le fait de pouvoir se servir de l'action **DAO** dès le niveau 2 est positif ou négatif : va-t-on perdre du temps en effectuant la recherche de la solution au niveau 2 (si elle n'aboutit pas), ou bien va-t-on en gagner en évitant de construire le troisième niveau et en effectuant la recherche sur un graphe plus petit (si elle aboutit) ?

Construisons donc le graphe de planification jusqu'au niveau 2 (voir la Figure 23). Il est presque identique à celui que l'on avait trouvé avec Graphplan, à ceci prés que l'on peut utiliser l'action **DAO**.

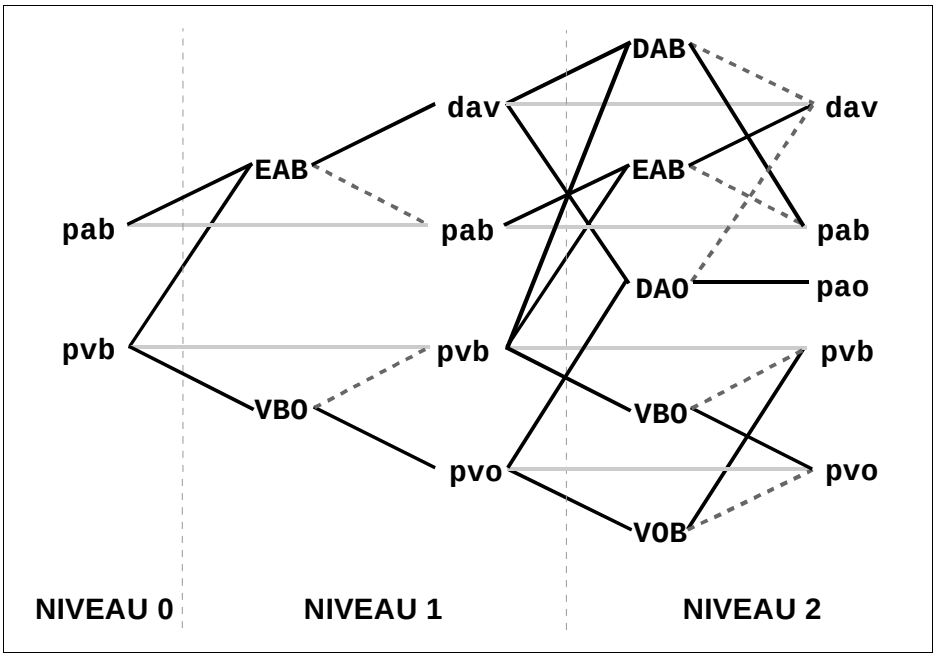

Figure 23 : Création du niveau 2

Calculons maintenant les mutex sur les actions. D'un point de vue global, nous allons en trouver plus qu'avec Graphplan, pour le niveau 2 du graphe. Mais si l'on ne considère que les mutex sur les actions que Graphplan avait utilisé (c'est-à-dire toutes les actions sauf **DAO**) alors on en trouve moins maintenant. Les mutex que l'on a en plus concernent l'action **DAO**.

Nous allons maintenant détailler le calcul des mutex sur les actions, afin de bien comprendre la différence avec Graphplan. Nous allons décomposer ce calcul selon les trois types de mutex : effets inconsistants, interférences et préconditions inconsistantes.

1. Effets inconsistants : dans un premier temps, on recherche les interdictions entre actions.

   (1) Add(**N_dav**) ∩ Del(**DAB**) = {**dav**} ⇒ ¬(**N_dav** ≪ **DAB**)
   (2) Add(**EAB**) ∩ Del(**DAB**) = {**dav**} ⇒ ¬(**EAB** ≪ **DAB**)
   (3) Add(**N_dav**) ∩ Del(**DAO**) = {**dav**} ⇒ ¬(**N_dav** ≪ **DAO**)
   (4) Add(**EAB**) ∩ Del(**DAO**) = {**dav**} ⇒ ¬(**EAB** ≪ **DAO**)

(5) Add(**DAB**) ∩ Del(**EAB**) = {**pab**} ⇒ ¬(**DAB** ≺ **EAB**)
(6) Add(**N_pab**) ∩ Del(**EAB**) = {**pab**} ⇒ ¬(**N_pab** ≺ **EAB**)
(7) Add(**N_pvb**) ∩ Del(**VBO**) = {**pvb**} ⇒ ¬(**N_pvb** ≺ **VBO**)
(8) Add(**VOB**) ∩ Del(**VBO**) = {**pvb**} ⇒ ¬(**VOB** ≺ **VBO**)
(9) Add(**VBO**) ∩ Del(**VOB**) = {**pvo**} ⇒ ¬(**VBO** ≺ **VOB**)
(10) Add(**N_pvo**) ∩ Del(**VOB**) = {**pvo**} ⇒ ¬(**N_pvo** ≺ **VOB**)

On peut déduire de ces interdictions un certain nombre de mutex :

(2) et (5) ⇒ {**EAB(2)**, **DAB(2)**} ∈ MutexA(*GP*)
(8) et (9) ⇒ {**VOB(2)**, **VBO(2)**} ∈ MutexA(*GP*)

2. Interférences : dans un premier temps, on recherche les interdictions entre actions.

(11) Prec(**N_dav**) ∩ Del(**DAB**) = {**dav**} ⇒ ¬(**DAB** ≺ **N_dav**)
(12) Prec(**DAO**) ∩ Del(**DAB**) = {**dav**} ⇒ ¬(**DAB** ≺ **DAO**)
(13) Prec(**DAB**) ∩ Del(**DAO**) = {**dav**} ⇒ ¬(**DAO** ≺ **DAB**)
(14) Prec(**N_dav**) ∩ Del(**DAO**) = {**dav**} ⇒ ¬(**DAO** ≺ **N_dav**)
(15) Prec(**N_pab**) ∩ Del(**EAB**) = {**pab**} ⇒ ¬(**EAB** ≺ **N_pab**)
(16) Prec(**DAB**) ∩ Del(**VBO**) = {**pvb**} ⇒ ¬(**VBO** ≺ **DAB**)
(17) Prec(**EAB**) ∩ Del(**VBO**) = {**pvb**} ⇒ ¬(**VBO** ≺ **EAB**)
(18) Prec(**N_pvb**) ∩ Del(**VBO**) = {**pvb**} ⇒ ¬(**VBO** ≺ **N_pvb**)
(19) Prec(**DAO**) ∩ Del(**VOB**) = {**pvo**} ⇒ ¬(**VOB** ≺ **DAO**)
(20) Prec(**N_pvo**) ∩ Del(**VOB**) = {**pvo**} ⇒ ¬(**VOB** ≺ **N_pvo**)

Avec ces nouvelles interdictions, on peut déduire le reste des mutex effets inconsistants − interférences :

(1) et (11) ⇒ {**DAB(2)**, **N_dav(2)**} ∈ MutexA(*GP*)
(12) et (13) ⇒ {**DAB(2)**, **DAO(2)**} ∈ MutexA(*GP*)
(3) et (14) ⇒ {**DAO(2)**, **N_dav(2)**} ∈ MutexA(*GP*)
(6) et (15) ⇒ {**EAB(2)**, **N_pab(2)**} ∈ MutexA(*GP*)
(7) et (18) ⇒ {**VBO(2)**, **N_pvb(2)**} ∈ MutexA(*GP*)
(10) et (20) ⇒ {**VOB(2)**, **N_pvo(2)**} ∈ MutexA(*GP*)

3. Préconditions inconsistantes : le calcul de ces mutex se fait de façon classique (voir la Figure 24 pour le détail des mutex). Deux actions sont inconsistantes si une précondition de la première forme une mutex avec une précondition de la deuxième.

On peut remarquer que l'action **EAB** interdit l'action **VBO**, bien que le contraire soit faux (17). Graphplan aurait déclaré la paire {**EAB**, **VBO**} comme étant une mutex, ce que nous ne faisons pas ici. De même pour les paires {**VBO**, **DAB**}, {**VOB**, **DAO**} et {**EAB**, **DAO**}. C'est en ce sens que l'on voit que même si on obtient plus de mutex que Graphplan au niveau 2 (22 contre 20), sur des actions identiques on en a moins.

On peut aussi remarquer que les mutex concernant le fluent **DAO** que l'on trouve à ce niveau, sont trouvées par Graphplan au niveau 3 : ainsi en est-il des paires {**DAO**, **N_dav**}, {**DAO**, **DAB**}, {**DAO**, **N_pvb**} et {**DAO**, **VBO**}. Par contre, Graphplan trouve au niveau 3 des mutex que l'on ne trouve pas avec LCGP au niveau 2, et qui ne le seront donc jamais : les paires {**DAO**, **EAB**}, {**DAO**, **N_pab**} et {**DAO**, **VOB**}. Intuitivement, on sent que le graphe de planification que l'on vient d'obtenir, développé jusqu'au niveau 2, peut très bien contenir des plans qui ne seraient trouvés par Graphplan qu'à partir du niveau 3.

```
Mutex entre fluents :
  dav – pab, pao
  pao – pab
  pvb – pvo

Mutex entre actions :
  DAB – N_dav, EAB, DAO, N_pab, N_pvo
  EAB – N_pab, N_dav, N_pvo, VOB
  DAO – N_dav, EAB, N_pvb, VBO, N_pab
  VBO – N_pvb, VOB, N_pvo
  VOB – N_pvo, DAB, N_pvb
  N_dav – N_pab
  N_pvb – N_pvo
```

Figure 24 : Les mutex du niveau 2

Achevons maintenant la construction du graphe par le calcul des mutex sur les fluents. Comme le fluent **pao** qui représente le but est présent dans le dernier niveau développé, nous allons pouvoir tenter d'extraire un plan-solution.

5.3.2. Extraction du plan-solution

Nous devons tout d'abord extraire l'arbre ET/OU du graphe de planification (voir la Figure 25). L'extraction de cet arbre se fait exactement comme avec Graphplan. Comme le graphe est développé seulement jusqu'au niveau 2, l'arbre ET/OU est nettement moins important.

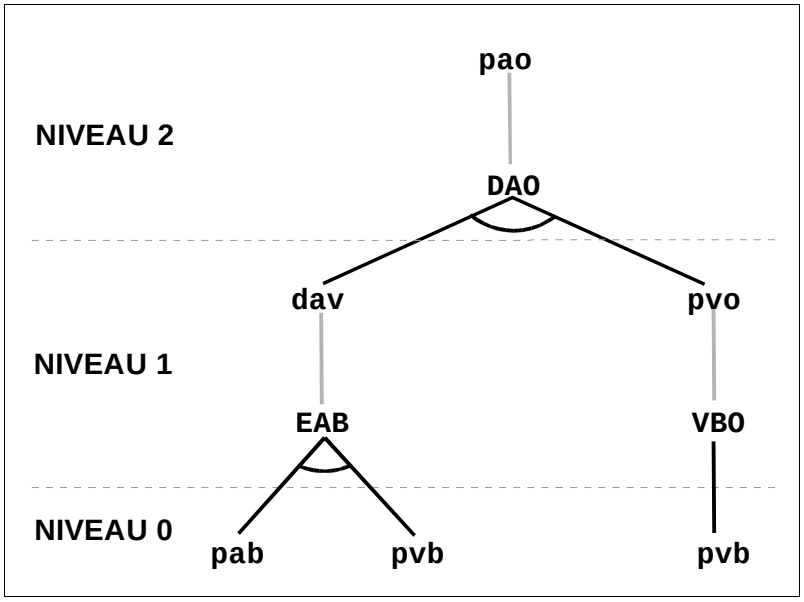

Figure 25 : L'arbre ET/OU issu du graphe de planification après création du niveau 2

Nous allons ensuite tenter d'extraire un plan-solution de cet arbre. Pour établir le fluent **pao**, il faut obligatoirement utiliser l'action **DAO** au niveau 2. Cette dernière fera donc partie du plan-solution s'il existe.

Nous devons ensuite essayer de voir si les préconditions de **DAO** peuvent être établies en même temps, c'est-à-dire par deux actions autorisées du niveau 1. On ne peut plus réellement parler d'actions parallèles ici, terme plutôt réservé aux actions indépendantes de Graphplan. Les préconditions de **DAO** sont les fluents **dav** et **pvo**, établis respectivement par les actions **EAB** et **VBO**. Il n'y a pas d'autre choix possible : si un plan-solution existe, ces actions en feront partie.

Nous devons ensuite vérifier que l'ensemble Q = {**EAB**, **VBO**} est autorisé. Nous savons déjà que **EAB** et **VBO** ne forment pas une mutex, mais ceci n'est en général pas suffisant pour être sûr qu'un ensemble est autorisé : il faut qu'une linéarisation autorisée existe. Dans notre cas, il n'y a que deux actions : si Q n'est pas

autorisé, c'est forcément que **EAB** et **VBO** forment une mutex. Comme ce n'est pas le cas, *Q* est autorisé : il peut faire partie d'un plan-solution.

Pour terminer, nous devons voir si les préconditions de **EAB** et **VBO** peuvent être établies en même temps ; ce qui est évidemment le cas puisque ces préconditions se trouvant dans le niveau 0, elles sont vraies dans l'état initial du problème.

La recherche d'un $\Phi_{\trianglelefteq}$−plan-solution est donc un succès : nous trouvons le $\Phi_{\trianglelefteq}$−plan $P = \langle\{$**EAB**, **VBO**$\}$, $\{$**DAO**$\}\rangle$. Ce plan n'est pas du tout identique au plan que l'on avait trouvé avec Graphplan : ce dernier était plus long d'un niveau, les actions **EAB** et **VBO** n'étaient pas dans un même ensemble d'actions puisqu'elles ne sont pas indépendantes, et deux no-ops se trouvaient dans le plan. Le $\Phi_{\trianglelefteq}$−plan *P* est ensuite transformé en un $\Phi_{\#}$−plan en utilisant la fonction **stratifier** :

P' = **stratifier**($\{$**EAB**, **VBO**$\}$, $\trianglelefteq$) $\oplus$ **stratifier**($\{$**DAO**$\}$, $\trianglelefteq$)
= $\langle$**EAB**, **VBO**$\rangle \oplus \langle$**DAO**$\rangle$
= $\langle$**EAB**, **VBO**, **DAO**$\rangle$

Ce plan *P'* doit normalement être réordonnancé en utilisant la fonction **réordonnancer−plan**, mais on voit tout de suite que le plan produit sera le même. On trouve donc le $\Phi_{\#}$−plan-solution qu'avait trouvé Graphplan, quand on en supprime les no-ops (voir la Figure 15).

6. Évaluation expérimentale

6.1. Conditions expérimentales

Nous allons présenter plusieurs séries d'expérimentations effectuées avec les planificateurs GP et LCGP$_{\trianglelefteq}$. Ces deux planificateurs partagent la majeure partie de leur code : leurs différences sont minimales, ainsi qu'il a été montré dans la section 5. La partie commune contient des améliorations connues de Graphplan : les techniques EBL/DDB de [Kambhampati 1999] [Kambhampati 2000] et une construction du graphe de planification inspirée de [Long et Fox 1999] et [Smith et Weld 1999] : une structure circulaire à deux niveaux. GP et LCGP$_{\trianglelefteq}$ sont implémentés en Allegro Common Lisp 5.0. Tous les tests ont été effectués avec un Pentium−II 450Mhz avec 256Mo de RAM, tournant sous Debian GNU/Linux 2.1.

6.2. Comparaisons entre plusieurs planificateurs basés sur Graphplan dans le domaine Logistics

Nous présentons les résultats des tests effectués sur des problèmes du domaine Logistics issus de la distribution BLACKBOX[4] [Kautz et Selman 1999] entre LCGP$_{\trianglelefteq}$ (que nous appellerons simplement LCGP par la suite) et trois planificateurs basés sur Graphplan : IPP[5] v4.0 [Nebel, Dimopoulos et Koehler 1997], STAN[6] v3.0 [Fox et Long 1998] et GP. IPP et STAN sont des planificateurs hautement optimisés, implémentés en C pour IPP et en C++ pour STAN. L'heuristique utilisée pour l'extraction de la solution dans GP et LCGP est l'heuristique originale de Graphplan, l'heuristique "no-ops first" : la première action qui est choisie pour établir un fluent est son no-op, et les autres actions sont choisies dans n'importe quel ordre. Nous présenterons ensuite une heuristique plus performante, que nous utiliserons dans les autres séries de tests. Les résultats de cette première série sont présentés dans la Table 2.

- Parmi les trois planificateurs basés sur Graphplan (qui utilisent la relation d'indépendance), STAN est le plus efficace. Deux raisons peuvent expliquer ce résultat : la recherche EBL/DDB est implémentée dans ce planificateur, et il ne conserve que les actions qui sont pertinentes pour chaque problème grâce à un outil d'analyse de type utilisé avant la planification, au moment de la création des actions de base. Ensuite vient GP, qui résout un peu moins de problèmes que STAN mais quand même beaucoup plus que IPP. GP est en général plus rapide que IPP, sauf dans deux problèmes. Ceci peut être expliqué par la recherche EBL/DDB qu'effectue GP.
- Notre planificateur, LCGP, résout tous les problèmes avec d'excellentes performances en comparaison des autres planificateurs. STAN est cependant plus rapide que LCGP sur 9 problèmes, mais les performances de LCGP seraient sûrement grandement améliorées s'il possédait les mêmes avantages que STAN (une

[4] BLACKBOX est téléchargeable à l'adresse `http://www.research.att.com/~kautz/blackbox/index.html`
[5] IPP est téléchargeable à l'adresse `http://www.informatik.uni-freiburg.de/~koehler/ipp.html`
[6] STAN est téléchargeable à l'adresse `http://www.dur.ac.uk/~dcs0www/research/stanstuff/stanpage.html`

implémentation en C++ et un outil performant d'analyse de type). Dans la plupart des problèmes, la construction du graphe de planification prend la plus grande partie du temps : le temps de recherche est alors négligeable. Quelques problèmes seulement (*log.c*, *log017*, *log020*, *log023*) prennent relativement plus de temps à cause de la difficulté de la recherche dans la seconde phase. L'amélioration apportée par l'utilisation de la relation d'autorisation est évidente : LCGP fonctionne environ 1800 fois plus vite que GP dans les problèmes résolus par les deux planificateurs.

Problèmes	**Temps CPU (sec.)**				**Ratio tempsGP/ tempsLCGP**	**Actions**				**Niveaux**		
										GP	**LCGP**	
	IPP	**STAN**	**GP**	**LCGP**		**IPP**	**STAN**	**GP**	**LCGP**		**(+)**	**(++)**
log.easy	0,06	0,05	0,41	0,32	1,29	25	25	25	25	9	9	6
rocket.a	23,09	–	16,29	0,49	33,05	30		30	28	7	7	4
rocket.b	34,40	24,34	5,85	0,40	14,62	26	26	26	26	7	7	4
log.a	2 174,07	4,34	164,01	1,23	133,56	54	54	54	54	11	11	7
log.b	5 820,92	5,67	76 402,09	1,78	43 043,43	45	44	45	45	13	13	8
log.c	–	6 135,85	≥86 400	227,69	≥ 379		52		53	≥ 11	13	8
log.d	–	22 105,24	–	3,89			68		73		15	9
log.d3	–	≥86 400	≥86 400	4,46	≥19 355				72	≥ 13	13	8
log.d1	–	–	≥86 400	23,88	≥3 619				68	≥ 14	17	10
log010	1 861,77	0,59	28,53	3,28	8,70	43	43	42	41	10	11	7
log011	–	218,03	8 635,85	2,18	3 965,04		48	48	49	11	11	7
log012	74,40	0,77	6,61	1,17	5,64	38	38	38	38	8	8	5
log013	–	523,24	4 526,88	3,70	1 222,82		67	67	66	11	11	7
log014	122,65	1,17	6,04	5,00	1,21	70	71	70	75	10	11	7
log015	–	≥86 400	≥86 400	4,26	≥20 267				61	≥ 11	13	7
log016	–	–	–	4,13					40		16	9
log017	–	–	≥86 400	101,07	≥ 855				44	≥ 16	17	10
log018	–	3,04	40,74	5,58	7,31		48	52	50	11	11	7
log019	–	5,77	14,46	2,60	5,56		47	46	50	11	12	7
log020	–	–	≥86 400	578,01	≥ 149				87	≥ 14	15	9
log021	–	3 259,27	2 594,30	4,20	617,25		63	63	66	11	12	7
log022	–	–	≥86 400	4,18	≥20 690				74	≥ 14	15	9
log023	–	232,42	≥86 400	90,50	≥ 955		61		61	≥ 13	13	8
log024	–	286,80	3 349,58	3,96	846,71		64	64	67	12	13	8
log025	–	1,46	12,62	3,38	3,74		57	58	56	12	13	8
log026	–	0,64	4,85	3,10	1,56		51	50	50	12	12	8
log027	–	23,15	≥86 400	3,41	≥25 345		71		72	≥ 13	14	8
log028	–	–	≥86 400	11,61	≥7 445				78	≥ 14	14	9
log029	3 558,05	1,19	8,27	5,96	1,39	46	49	46	46	10	11	7
log030	–	1,11	49,54	3,06	16,21		52	52	52	13	13	8
Moy. (*)	1 518,82	1 563,53	5 325,94	2,85	1 866,28	41,89	52,33	48,67	49,11	10,50	10,89	6,78
Moy. (**)	-	-	≥34 280,96	36,95	≥927,82	-	-	-	55,57	≥11,5	12,37	7,53

(+) nombre de niveaux du plan après la transformation par **réordonnancer−plan**
(++) nombre de niveaux du plan avant la transformation par **réordonnancer−plan**
(*) moyenne sur les problèmes résolus (cellules blanches). Pour LCGP, moyenne des problèmes résolus par GP.
(**) moyenne sur les 30 problèmes.
cellules grises : échec de la résolution du problème correspondant.
Un tiret (−) indique que le problème correspondant n'a pas pu être traité à cause d'un manque de mémoire.

Table 1 : Comparaison entre plusieurs planificateurs basés sur Graphplan dans le domaine Logistics

Une des particularités du domain Logistics est que les plans peuvent contenir un grand nombre d'actions parallèles. Ainsi, GP, IPP et STAN trouvant beaucoup d'actions indépendantes, le graphe de planification est beaucoup moins contraint (par rapport au nombre d'actions) que dans d'autres domaines classiques, comme par exemple le Monde des cubes avec un bras manipulateur. Cependant, de nombreuses contraintes trouvées par Graphplan peuvent être relaxées par LCGP, pour devenir des contraintes d'autorisation. Par exemple, avec GP, les deux actions "embarquer un paquet dans un avion à l'endroit A" et "cet avion vole de l'endroit A vers l'endroit B" ne sont pas indépendantes : une précondition de la première action (l'avion doit être à l'endroit A) est retirée par la seconde action. Avec LCGP, la première action autorise la seconde ; elles peuvent donc apparaître simultanément dans un ensemble autorisé. Ainsi, les très bonnes performances de LCGP dans ce domaine sont dus principalement à la réduction de la taille de l'espace de recherche.

Aucun de ces planificateurs ne produit des solutions optimales en nombre d'actions, mais leurs plans contiennent approximativement le même nombre d'actions. LCGP n'est pas optimal dans le sens de Graphplan, en nombre de niveaux par rapport à la relation d'indépendance ; il est optimal en nombre de

niveaux par rapport à la relation d'autorisation. Après transformation par la procédure **réordonnancer−plan**, le plan produit est de longueur supérieure ou égale au plan produit par GP. Cependant, la qualité du plan en nombre d'actions ne semble pas être vraiment dégradée : en moyenne pour les problèmes résolus par les deux planificateurs, les plans trouvés par GP contiennent 48,67 actions, tandis que ceux trouvés par LCGP contiennent 49,11 actions. De plus, LCGP donne la meilleure solution pour plusieurs problèmes (cf. *log010*, *log013*, *log025*...).

6.3. Les heuristiques dans la phase de recherche de GP et LCGP

Dans cette section, nous présentons l'heuristique que GP et LCGP utiliseront pendant la phase d'extraction de la solution dans les prochaines séries de tests. Cette heuristique est indépendante du domaine et réduit beaucoup le temps de recherche. Elle combine l'heuristique initiale de Graphplan, l'heuristique "no-ops first", et l'heuristique "level-based" proposée simultanément pour LCGP dans [Cayrol, Régnier et Vidal 2000b] et pour Graphplan dans [Kambhampati et Nigenda 2000]. En combinant ces deux heuristiques, nous profitons de leurs deux qualités essentielles : qualité de la solution en nombre d'actions pour l'heuristique "no-ops first", et amélioration du temps de recherche pour l'heuristique "level-based heuristic".

Si l'on considère la phase d'extraction de la solution comme la résolution d'un problème de satisfaction de contraintes dynamique (DCSP) [Kambhampati 1999] [Kambhampati 2000], deux ordres entrent en ligne de compte : l'ordre dans lequel on considère les variables à assigner et l'ordre dans lequel on considère les valeurs à employer. Afin d'utiliser l'heuristique classique en CSP : "la variable la plus contrainte en premier et la valeur la moins contrainte en premier", nous avons besoin d'une mesure quantitative de ces contraintes.

L'heuristique originale de Graphplan "no-ops first" utilise en premier un no-op pour établir un fluent, et ensuite les autres actions disponibles. Ce choix semble raisonnable pour produire des plans contenant le moins d'actions possibles, car les fluents seront produits de préférence par des no-ops, ce qui évite dans une certaine mesure d'insérer d'autres actions dans le plan. Cependant, cette heuristique ne donne pas d'informations sur les contraintes entre les fluents ou les actions et n'est donc pas appropriée d'un point de vue CSP. Nos résultats expérimentaux dans [Cayrol, Régnier et Vidal 2000b] et ceux de [Kambhampati et Nigenda 2000] démontrent clairement que dans de nombreux domaines cette stratégie conduit à une détérioration du temps de recherche.

La taille du domaine des variables est une autre heuristique employée dans [Kambhampati 2000] pour mesurer le degré de contrainte d'une variable. Selon ce critère, un fluent est dit plus contraint qu'un autre si moins d'actions le produisent ; d'après l'étude expérimentale de [Kambhampati 2000], Graphplan va au maximum 4 fois plus vite en utilisant cette heuristique.

Aucune de ces deux heuristiques n'est réellement "informative", car elles ne mesurent pas vraiment la difficulté d'extraction de la solution. En effet, ce n'est pas parce que plusieurs actions produisent un fluent que l'obtention de ce dernier est plus aisée. Cette information concerne seulement le niveau courant ; afin d'améliorer l'heuristique, nous avons besoin de mesurer la difficulté d'asserter un fluent en prenant en compte les différents niveaux du graphe de planification (du niveau 0 jusqu'au niveau courant).

L'heuristique "level-based" proposée dans [Cayrol, Régnier et Vidal 2000b] et [Kambhampati et Nigenda 2000] utilise comme mesure le niveau d'apparition d'un fluent (ou d'une action), c'est-à-dire le numéro du niveau du graphe de planification dans lequel ce fluent (ou cette action) apparaît pour la première fois. Nous pouvons raisonnablement penser que plus le niveau d'apparition d'un fluent est élevé, plus l'obtention de ce fluent est difficile ; en effet :

- Afin d'être produit, un fluent dont le niveau d'apparition est élevé requiert un plan possédant beaucoup de niveaux. Généralement, un plan qui a besoin de n niveaux pour asserter un fluent p contient plus d'actions (en incluant les no-ops) qu'un autre plan dans lequel le fluent p apparaît à un niveau $n' < n$. Ainsi, un fluent donné est en général plus difficile à obtenir qu'un autre fluent ayant un niveau d'apparition moins élevé.
- Les exclusions mutuelles entre fluents (ou entre actions) tendent à disparaître dans les niveaux supérieurs du graphe de planification. Ainsi, les fluents qui ont un niveau d'apparition élevé sont établis en utilisant des actions apparues récemment, qui ont donc plus de chances d'être encore impliquées dans des exclusions mutuelles que des actions plus anciennes.

Ainsi, pour un fluent, un niveau d'apparition élevé dénote deux difficultés : le plan pour obtenir ce fluent contient généralement un grand nombre d'actions (en incluant les no-ops), et les actions qui le produisent on plus de chances d'être impliquées dans des mutex.

Le niveau d'apparition semble donc être une mesure caractéristique du degré de contrainte d'un fluent. Ainsi, pour l'heuristique de choix des variables, nous assignerons une valeur en premier aux fluents qui ont un niveau d'apparition élevé. En ce qui concerne l'heuristique de choix des valeurs des domaines, un raisonnement similaire montre que les actions qui ont un niveau d'apparition bas sont en général plus aisées à produire.

Nos expérimentations (cf. Table 1) prouvent qu'il est réellement intéressant de combiner cette heuristique "level-based" avec l'heuristique "no-ops first" afin de profiter des qualités des deux : accélération de la recherche (heuristique "level-based") et qualité de la solution en nombre d'actions (heuristique "no-ops first"). Par conséquent, l'heuristique qui est implémentée dans GP et LCGP utilise comme heuristique de choix des variables l'heuristique "le plus haut niveau d'apparition en premier" et comme heuristique de choix des valeurs "le plus bas niveau d'apparition en premier". Les mauvais résultats dans le domaine des Tours de Hanoi peuvent être expliqués par le fait qu'avec le codage standard de ce domaine, le niveau d'apparition des fluents du but est toujours 0, excepté pour le disque le plus large : les fluents "disque 1 sur disque 2", "disque 2 sur disque 3", ..., appartiennent à la fois à l'état initial et au but. Une solution serait de coder ce domaine d'une façon plus "informative", en indiquant sur quelles tiges se trouvent les disques : "disque 1 sur disque 2 sur tige A" appartiendrait à l'état initial, et "disque 1 sur disque 2 sur tige C" appartiendrait au but.

Problèmes	Temps CPU (sec.)			Ratio No−opsF/		Actions			Niveaux	
	No−opsF	Level	LeveN	Level	LevelN	No−ops	Level	LevelN	(+)	(++)
ferry6	3,05	0,30	0,36	10,17	8,47	23	23	23	23	12
ferry8	387,51	2,51	3,06	154,39	126,64	31	31	31	31	16
gripper6	1,45	0,39	0,49	3,74	2,96	17	17	17	11	6
gripper8	165,81	8,02	7,61	20,68	21,79	23	23	23	15	8
bw-large-a	3,42	2,49	2,57	1,37	1,33	12	12	12	12	12
bw-large-b	257,65	19,13	36,05	13,47	7,15	18	18	18	18	18
log020	578,01	8,38	9,20	68,97	62,83	87	93	87	15	9
log023	90,50	3,77	4,30	24,01	21,05	61	65	61	13	8
hanoi5	8,41	10,48	27,30	0,80	0,31	32	32	32	32	21

(+) nombre de niveaux du plan après la transformation par **réordonnancer−plan**
(++) nombre de niveaux du plan avant la transformation par **réordonnancer−plan**
No−opsF : heuristique no-ops first.
Level : heuristique basée sur les niveaux.
LevelN : heuristique basée sur les niveaux avec "no-ops first".

Table 2: Bénéfices de l'heuristique pour LCGP

6.4. Comparaison GP vs. LCGP dans les domaines du Ferry et du Gripper

Par comparaison avec GP, dans les domaines du ferry et du Gripper, LCGP n'est pas aussi efficace que dans le domaine Logistics ; mais il est toujours plus rapide que GP (cf. tables 3 et 4). La colonne "Noeuds développés" dans les tables représentent le nombre d'ensembles de sous-buts que l'algorithme essaie de satisfaire. Il est intéressant de remarquer que dans le domaine du Ferry, dont tous les problèmes ont une solution séquentielle, les graphes de planification produits par LCGP sont environ deux fois plus courts que ceux de GP. En effet, dans LCGP, les actions "embarquer une voiture sur la rive A" et "naviguer de la rive A vers la rive B" peuvent appartenir à un même ensemble autorisé ; le même phénomène se produit pour les actions "débarquer une voiture sur la rive B" et "naviguer de la rive B vers la rive A".

Ce phénomène se produit aussi dans le domaine du Gripper : les graphes de planification produits par LCGP sont aussi environ deux fois plus courts que ceux de GP. Ce domaine autorise un certain parallélisme entre les actions : avec GP, le robot peut tenir une balle dans chacune de ses pinces. Avec LCGP, l'action de se déplacer d'une pièce dans l'autre peut être considérée simultanément avec les actions de saisir une balle dans chaque pince. Mais contrairement au domaine du Ferry, dans lequel l'accélération entre les deux planificateurs semble se stabiliser autour de 6,5, dans le domaine du Gripper l'accélération augmente en fonction de la difficulté du problème.

Sous-buts	Temps CPU (sec.)		Ratio TempsGP/ TempsLCGP	Noeuds développés		Actions		Niveaux		
								GP	LCGP	
	GP	LCGP		GP	LCGP	GP	LCGP		(+)	(++)
1	0,02	0,01	1,54	4	3	3	3	3	3	2
2	0,03	0,02	1,30	15	5	7	7	7	7	4
3	0,06	0,04	1,58	139	21	11	11	11	11	6
4	0,17	0,06	2,67	646	92	15	15	15	15	8
5	0,56	0,14	4,00	2 310	351	19	19	19	19	10
6	1,96	0,36	5,47	6 998	997	23	23	23	23	12
7	6,29	1,06	5,94	19 125	2 614	27	27	27	27	14
8	18,61	3,06	6,07	48 846	6 657	31	31	31	31	16
9	51,62	7,82	6,60	118 195	14 786	35	35	35	35	18
10	143,54	22,07	6,50	275 921	37 686	39	39	39	39	20
11	385,27	58,16	6,62	626 157	84 930	43	43	43	43	22
12	1 021,97	159,85	6,39	1 392 793	190 266	47	47	47	47	24

(+) nombre de niveaux du plan après la transformation par **réordonnancer−plan**
(++) nombre de niveaux du plan avant la transformation par **réordonnancer−plan**

Table 3 : Comparaison GP vs. LCGP dans le domaine du Ferry

Sous-buts	Temps CPU (sec.)		Ratio TimeGP/ TimeLCGP	Noeuds développés		Actions		Niveaux		
								GP	LCGP	
	GP	LCGP		GP	LCGP	GP	LCGP		(+)	(++)
2	0,04	0,03	1,30	4	3	5	5	3	3	2
4	0,15	0,07	2,17	435	48	11	11	7	7	4
6	3,27	0,49	6,65	9 533	1 272	17	17	11	11	6
8	57,89	7,61	7,61	127 804	15 332	23	23	15	15	8
10	765,05	89,19	8,58	1 233 178	128 664	29	29	19	19	10
12	9 455,39	927,60	10,19	9 176 365	861 096	35	35	23	23	12

(+) nombre de niveaux du plan après la transformation par **réordonnancer−plan**
(++) nombre de niveaux du plan avant la transformation par **réordonnancer−plan**

Table 4 : Comparaison GP vs. LCGP dans le domaine du Gripper

6.5. Comparaison GP vs. LCGP dans le Monde des cubes

6.5.1. Version de Prodigy

La version de Prodigy du Monde des cubes utilise 6 opérateurs et un bras manipulateur, et il n'y a aucun parallélisme à exploiter, même pour LCGP : les graphes de planification produits par GP et LCGP sont rigoureusement identiques. Ainsi, la phase d'extraction de la solution est effectuée de la même façon. On pourrait cependant penser que LCGP soit plus lent que GP à cause de la vérification de la contrainte d'autorisation des ensembles d'actions. Mais comme il n'y a pas de parallélisme, un ensemble d'actions considéré durant la recherche ne contient qu'une seule "vraie" action (toutes les autres sont des no-ops) ; or, la vérification de l'autorisation peut n'être vérifiée que sur le sous-ensemble des "vraies" actions d'un ensemble. En effet :

- un no-op autorise toujours un autre no-op ;
- si une action n'autorise pas un no-op, alors ce no-op n'autorise pas cette action, et ainsi ils sont mutuellement exclusifs (et vice versa).

Ainsi, un ensemble d'actions qui contient au plus deux "vraies" actions est autorisé si et seulement si il ne contient pas de mutex. Ceci explique pourquoi LCGP et GP ont des performances quasiment identiques dans ce domaine (cf. Table 5).

Problèmes	Temps CPU (sec.)		Ratio TempsGP/ TempsLCGP	Noeuds développés		Actions		Niveaux		
								GP	LCGP	
	GP	LCGP		GP	LCGP	GP	LCGP		(+)	(++)
bw−simple	0,02	0,02	1,00	3	3	2	2	2	2	2
bw−sussman	0,06	0,06	1,00	7	7	6	6	6	6	6
bw−reversal4	0,08	0,08	0,99	9	9	8	8	8	8	8
bw−large−a	2,02	2,07	0,98	218	218	12	12	12	12	12
bw−large−b	33,31	34,16	0,98	15 563	15 563	18	18	18	18	18
Moyenne	7,10	7,28	0,98	–	–	9,20	9,20	9,20	9,20	9,20

(+) nombre de niveaux du plan après la transformation par **réordonnancer−plan**
(++) nombre de niveaux du plan avant la transformation par **réordonnancer−plan**

Table 5 : Comparaison GP vs. LCGP dans le Monde des cubes (version Prodigy)

6.5.2. Version à trois opérateurs

Nous avons comparé GP et LCGP dans la version du Monde des cubes avec trois opérateurs, décrite dans l'introduction de cette thèse. La différence essentielle avec la version de Prodigy de ce domaine est la possibilité d'appliquer des actions en parallèle. LCGP profite de ce parallélisme, et des actions mutuellement exclusives pour GP deviennent autorisées pour LCGP. Par exemple, si les deux cubes A et B sont au sommet de deux piles différentes, les actions "déplacer A vers la table" et "déplacer B sur A" sont mutuellement exclusives car la seconde retire le fait que le cube A est libre, précondition de la première action ; en revanche, la première action autorise la seconde.

En terme de performance en temps CPU (cf. Table 6), LCGP est toujours plus rapide que GP (en moyenne 426 fois). Deux problèmes (*Nineteen* et *P16*) ne sont pas résolus par GP à cause du manque de mémoire vive. En effet, LCGP requiert environ 5,38 niveaux pour résoudre les problèmes tandis que GP en requiert 7,13. En terme de qualité de la solution en nombre d'actions, LCGP est légèrement meilleur que GP (13,13 actions pour LCGP contre 13,75 pour GP), bien que la longueur du plan en nombre de niveaux soit généralement plus importante pour LCGP que pour GP (8,63 niveaux contre 7,13 pour GP). Encore une fois, la perte d'optimalité en nombre de niveaux semble n'avoir aucune influence sur la qualité de la solution en terme de nombre d'actions du plan-solution.

Un fait remarquable qui se produit dans ce domaine avec les problèmes que nous avons testés avec LCGP, est que le nombre de noeuds développés correspond exactement au nombre de niveaux du graphe de planification (en comptant le niveau 0, qui n'apparaît pas dans les plans-solutions puisqu'il ne contient pas d'actions). En d'autres termes, une fois que LCGP a trouvé un ensemble d'actions autorisé qui produit l'ensemble des sous-buts en vérifiant les mutex et l'autorisation, il ne retourne jamais en arrière, même dans les problèmes les plus difficiles. Ce phénomène ne se produit pas avec GP : il y a toujours au moins un retour arrière (problèmes *Bw−12steps* et *Nine*), et beaucoup plus dans les problèmes plus difficiles.

Problèmes	Temps CPU (sec.)		Ratio TempsGP/ TempsLCGP	Noeuds développés		Actions		Niveaux		
								GP	LCGP	
	GP	LCGP		GP	LCGP	GP	LCGP		(+)	(++)
Bw−12step	0,35	0,33	1,06	6	5	6	6	4	6	4
Nine	2,43	0,76	3,20	6	4	10	9	4	5	3
Eleven	12,67	6,04	2,10	8	5	13	14	5	7	4
Fifteen	17 920,31	55,90	320,59	81 402	7	22	18	8	11	6
Nineteen	–	162,80	–	–	7	–	26	–	10	6
P8	3,69	1,51	2,44	15	9	13	15	10	13	8
P10	11,45	4,73	2,42	18	6	14	12	7	7	5
P12	36 180,19	11,82	3 060,41	1 821 844	6	16	15	9	10	5
P14	73,29	46,11	1,59	13	9	16	16	10	10	8
P16	–	216,27	–	–	9	–	26	–	13	8

(+) nombre de niveaux du plan après la transformation par **réordonnancer−plan**
(++) nombre de niveaux du plan avant la transformation par **réordonnancer−plan**
Un tiret (−) indique que le problème correspondant n'a pas pu être traité à cause d'un manque de mémoire.

Table 6 : Comparaison GP vs. LCGP dans le Monde des cubes (version à trois opérateurs)

6.6. Comparaison GP vs. LCGP dans les domaines Mprime et Mystery

Nous finissons ces expérimentations avec les domaines Mprime et Mystery. Ces domaines sont très proches et acceptent des actions parallèles. Nous avons utilisé les séries de problèmes créés pour la première

compétition de générateurs de plans lors de la conférence AIPS−98. Contrairement aux problèmes des autres domaines que nous avons testés, certains problèmes de ces domaines n'admettent pas de solution (cf. tables 7 et 8. La principale difficulté de ces domaines est la construction du graphe de planification ; l'extraction de la solution est alors triviale. Les problèmes qui ne sont pas présents dans ces tables (*Mprime−X−6*, *Mprime−X−10*, ...) ne sont résolus ni par GP ni par LCGP à cause d'un manque de mémoire vive durant la construction du graphe.

Ces domaines sont les seuls que nous ayons trouvés dans lesquels GP est légèrement meilleur que LCGP : dans le domaine Mprime, GP est 1,01 fois meilleur que LCGP, et dans le domain Mystery GP est 1,05 fois meilleur que LCGP. La raison en est la suivante : bien que parfois plus courts en nombre de niveaux, les graphes de planification produits par LCGP contiennent dans leurs derniers niveaux plus d'actions et de fluents que ceux de GP. Par exemple pour le problème *Mprime−Y−2* résolu en 6,99 secondes par GP et en 7,41 secondes par LCGP, le dernier niveau (le septième) pour GP contient 835 action contre 861 pour le dernier niveau (le sixième) pour LCGP. LCGP calcule aussi plus d'exclusions mutuelles que GP : 188 820 contre 187 386. A un niveau inférieur ou identique, les graphes construits par LCGP contiennent plus de plans potentiels que ceux construits par GP mais ils sont plus difficiles à créer en raison du grand nombre d'actions et de fluents. Et comme l'extraction du plan-solution est triviale pour les deux planificateurs, LCGP n'en profite pas.

Nous pouvons cependant remarquer que les meilleures performances de GP dans ces domaines par rapport à LCGP n'ont aucune commune mesure avec les meilleures performances de LCGP dans les autres domaines : l'accélération la plus importante dans ces domaines est de 1,3 (problèmes *Mprime−X−8* et *Mysty−X−2*), ce qui n'est rien comparé à une accélération comme 39 524 dans le domaine Logistics (problème *log016*). En ce qui concerne la qualité de la solution en nombre d'actions, GP est meilleur que LCGP dans le domaine Mprime (en moyenne 0,7% d'actions en plus pour LCGP) tandis que LCGP est meilleur que GP dans le domaine Mystery (en moyenne 3,9% d'actions en plus pour GP).

Problèmes	**Temps CPU (sec.)**		**Ratio TimeGP/ TimeLCGP**	**Noeuds développés**		**Actions**		**Niveaux**		
								GP	**LCGP**	
	GP	**LCGP**		**GP**	**LCGP**	**GP**	**LCGP**		**(+)**	**(++)**
Mprime−X−1	1,95	1,17	1,67	6	5	6	8	5	5	4
Mprime−X−2	24,38	29,90	0,82	8	5	9	10	5	5	4
Mprime−X−3	2,48	1,98	1,25	5	4	4	4	4	4	3
Mprime−X−4	1,30	1,15	1,13	8	8	9	10	7	7	6
Mprime−X−5	48,74	53,21	0,92	0	0	–	–	10	–	9
Mprime−X−7	3,41	2,91	1,17	0	0	–	–	10	–	8
Mprime−X−8	9,22	12,02	0,77	6	6	10	10	5	5	5
Mprime−X−9	5,25	5,25	1,00	6	5	8	8	5	5	4
Mprime−X−11	3,51	2,26	1,55	12	6	8	8	7	7	5
Mprime−X−12	5,74	6,22	0,92	6	10	6	9	5	6	5
Mprime−X−16	14,88	15,04	0,99	37	36	10	6	5	5	4
Mprime−X−17	32,57	25,95	1,26	5	4	5	5	4	4	3
Mprime−X−19	101,33	121,91	0,83	7	6	8	8	6	7	5
Mprime−X−21	135,71	131,03	1,04	0	0	–	–	14	–	12
Mprime−X−25	0,77	0,63	1,24	5	4	4	4	4	4	3
Mprime−X−26	9,01	6,61	1,36	6	5	6	6	5	5	4
Mprime−X−27	13,01	5,68	2,29	7	4	7	7	4	4	3
Mprime−X−28	4,24	2,01	2,11	20	6	9	7	7	7	5
Mprime−X−29	7,02	3,83	1,83	5	4	5	6	4	5	3
Mprime−Y−1	5,23	4,85	1,08	5	4	4	4	4	4	3
Mprime−Y−2	6,99	7,41	0,94	8	7	7	8	7	7	6
Mprime−Y−4	6,66	6,46	1,03	5	4	5	4	4	4	3
Mprime−Y−5	1,12	1,17	0,96	5	5	7	6	4	5	4
Moyenne	19,33	19,51	0,99	7,48	6,00	6,85	6,90	5,87	5,25	4,83

(+) nombre de niveaux du plan après la transformation par **réordonnancer−plan**
(++) nombre de niveaux du plan avant la transformation par **réordonnancer−plan**
Un tiret (−) indique que le problème correspondant n'admet pas de solution.

Table 7 : Comparaison GP vs. LCGP dans le domaine Mprime

Problèmes	Temps CPU (sec.)		Ratio TempsGP/ TempsLCGP	Noeuds développés		Actions		Niveaux		
								GP	LCGP	
	GP	LCGP		GP	LCGP	GP	LCGP		(+)	(++)
Mysty−X−1	0,81	0,54	1,48	6	5	5	6	5	6	4
Mysty−X−2	22,31	29,14	0,77	6	8	9	10	5	5	4
Mysty−X−3	2,19	1,75	1,25	5	4	4	4	4	4	3
Mysty−X−4	2,70	2,57	1,05	0	0	–	–	14	–	13
Mysty−X−5	43,91	51,09	0,86	0	0	–	–	10	–	9
Mysty−X−7	3,19	2,80	1,14	0	0	–	–	10	–	8
Mysty−X−8	198,11	193,08	1,03	0	0	–	–	21	–	21
Mysty−X−9	5,33	5,03	1,06	6	5	8	8	5	5	4
Mysty−X−11	1,85	1,15	1,61	12	6	7	7	7	7	5
Mysty−X−12	2,23	2,70	0,82	0	0	–	–	8	–	8
Mysty−X−15	88,37	94,56	0,93	7	6	12	7	6	6	5
Mysty−X−16	11,93	9,78	1,22	0	0	–	–	9	–	7
Mysty−X−17	19,48	16,72	1,16	5	4	4	4	4	4	3
Mysty−X−18	38,58	41,04	0,94	0	0	–	–	18	–	18
Mysty−X−19	23,94	26,65	0,90	7	6	6	7	6	7	5
Mysty−X−20	68,51	63,16	1,08	8	7	13	13	7	7	6
Mysty−X−21	132,89	131,69	1,01	0	0	–	–	14	–	12
Mysty−X−23	129,35	141,87	0,91	0	0	–	–	11	–	10
Mysty−X−24	283,71	318,21	0,89	0	0	–	–	25	–	25
Mysty−X−25	0,81	0,81	1,00	5	4	4	4	4	4	3
Mysty−X−26	4,98	4,39	1,14	7	6	8	6	6	6	5
Mysty−X−27	1,95	1,57	1,24	7	4	7	7	4	4	3
Mysty−X−28	0,97	0,72	1,35	22	6	7	7	7	7	5
Mysty−X−29	1,71	1,62	1,05	5	4	4	4	4	4	3
Mysty−X−30	18,92	18,33	1,03	7	6	10	10	6	6	5
Moyenne	44,35	46,44	0,95	4,60	3,24	7,20	6,93	8,80	5,47	7,76

(+)nombre de niveaux du plan après la transformation par **réordonnancer−plan**
(++)nombre de niveaux du plan avant la transformation par **réordonnancer−plan**
Un tiret (−) indique que le problème correspondant n'admet pas de solution.

Table 8 : Comparaison GP vs. LCGP dans le domaine Mystery

IV. Planifier par satisfaction de bases de clauses

1. Introduction

Dans cette partie, nous étudions différents codages pour la planification par satisfiabilité, en nous basant essentiellement sur l'article de [Mali et Kambhampati 1999] qui a beaucoup apporté à la formalisation de ces codages, notamment par l'utilisation de notations précises. Grâce à ces dernières, on s'aperçoit par exemple que le codage basé sur les espaces de plans proposé initialement dans [Kautz, McAllester et Selman 1996] est inutilement complexe et peut être grandement simplifié.

Pour chacun des codages étudiés, nous proposons des corrections par rapport à [Mali et Kambhampati 1999], ainsi que diverses améliorations permettant entre autres de produire des codages plus compacts, et de prendre en compte le parallélisme dans les codages basés sur les espaces de plans.

1.1. Notations

Nous utiliserons les notations classiques en logique propositionnelle. Les connecteurs pour l'implication ($\Rightarrow$) et l'équivalence ($\Leftrightarrow$) sont les moins prioritaires, donc on supprimera certaines parenthèses :

$$((a \wedge b) \Rightarrow (c \vee d)) \equiv (a \wedge b \Rightarrow c \vee d)$$

Soit $A = \{a_1, a_2, ..., a_n\}$ un ensemble fini quelconque et *f* une règle de réécriture.

- $\bigwedge_{a \in A} f(a)$ dénote la formule $F = \left(f(a_1) \wedge f(a_2) \wedge ... \wedge f(a_n) \right)$. Si $A = \varnothing$, alors $F = \top$.
- $\bigvee_{a \in A} f(a)$ dénote la formule $F = \left(f(a_1) \vee f(a_2) \vee ... \vee f(a_n) \right)$. Si $A = \varnothing$, alors $F = \bot$.

Soit *P*[*x*] une formule dépendant de *x* (c'est-à-dire contenant au moins une occurrence libre de *x*). *P*[*a*] dénote la formule *P*[*x*] à laquelle on a appliqué la substitution {*a* / *x*}.
Soit $PA = \{a \in A \mid P[a] \text{ vraie}\} = \{b_1, b_2, ..., b_m\}$.

- $\left(\bigwedge_{a \in A \mid P[a]} f(a) \right)$ dénote la formule $F = \left(f(b_1) \wedge f(b_2) \wedge ... \wedge f(b_m) \right)$. Si $PA = \varnothing$, alors $F = \top$.
- $\left(\bigvee_{a \in A \mid P[a]} f(a) \right)$ dénote la formule $F = \left(f(b_1) \vee f(b_2) \vee ... \vee f(b_m) \right)$. Si $PA = \varnothing$, alors $F = \bot$.

Soient F_1 et F_2 deux formules, et soit *G* une propriété :

- $\left[G \hookrightarrow F_1 \middle| F_2\right]$ dénote la règle de réécriture suivante : (si *G* est satisfaite, alors F_1 sinon F_2).

1.2. Exemple

Pour illustrer les différents codages et les problèmes qui s'y rapportent, nous nous servirons du même problème de planification dans toute cette partie. Nous donnons dans la Figure 26 le codage STRIPS de ce problème, ainsi que les ensembles de fluents correspondant à quatre catégories : *F* est l'ensemble total des fluents, *Fp* est l'ensemble des fluents qui apparaissent dans les préconditions des actions, *Fa* est l'ensemble des fluents qui apparaissent dans les ajouts des actions, et *Fd* est l'ensemble des fluents qui apparaissent dans les retraits des actions (pour *O* un ensemble d'actions) :

- $Fp = \{ f \in F \mid \exists a \in O \text{ et } f \in \text{Prec}(a) \}$
- $Fa = \{ f \in F \mid \exists a \in O \text{ et } f \in \text{Add}(a) \}$
- $Fd = \{ f \in F \mid \exists a \in O \text{ et } f \in \text{Del}(a) \}$

Action *A* **:**	**Ensemble des actions :** $O = \{A, B, C\}$
− Prec(*A*) = {*a*}	**Ensembles de fluents :**
− Add(*A*) = {*b*}	− $F = \{a, b, c, d, e\}$
− Del(*A*) = {*c*}	− $Fp = \{a, c, d\}$
Action *B* **:**	− $Fa = \{b, d, e\}$
− Prec(*B*) = {*c*}	− $Fd = \{c, d\}$
− Add(*B*) = {*d*}	**État initial :** $I = \{a, c\}$
− Del(*B*) = {}	**But :** $G = \{e\}$
Action *C* **:**	
− Prec(*C*) = {*d*}	**Le problème :**
− Add(*C*) = {*e*}	$\prod = \{O, I, G\}$
− Del(*C*) = {d}	

Figure 26 : Codage d'un problème sous forme STRIPS

La résolution de ce problème nécessite l'application successive de l'action *B* et de l'action *C*. Nous développerons donc les différents codages sur 3 niveaux, le niveau 0 ne contenant que les fluents de l'état initial. Ce nombre de niveaux du codage sera dénoté dans les formules représentant les différents codages par l'entier *k*.

2. Codages dans les espaces d'états

L'idée de ces codages est de prouver la validité d'un plan en s'intéressant à la *transition entre niveaux successifs*, en partant de l'état initial et en se dirigeant vers le but. Le parallélisme est autorisé, au travers de la notion d'*indépendance* entre actions simultanées. Ce codage nécessite l'intégration de la notion de *frame-axiome*, pour exprimer le fait que les fluents d'un niveau qui ne sont pas affectés par les actions du niveau suivant sont conservés. Nous décrivons deux techniques possibles : la première, décrite dans [Mali et Kambhampati 1999], utilise des frame-axiomes explicatifs ; la deuxième, sur laquelle sont basés les travaux sur BLACKBOX [Kautz et Selman 1999], utilise des no-ops.

2.1. Codage dans les espaces d'états avec frame-axiomes explicatifs

2.1.1. Syntaxe et sémantique des propositions

Les règles du codage dans les espaces d'états produisent deux formes de propositions :

1. Les propositions de la forme *A*(*i*), où *A* est une action appartenant à l'ensemble des actions *O* et *i* est un entier naturel, représentent le fait que l'action *A* est appliquée à un niveau *i* du plan si et seulement si *A*(*i*) a la valeur de vérité *vrai*.

2. Les propositions de la forme *f*(*i*), où *f* est un fluent appartenant à l'ensemble des fluents *F* et *i* est un entier naturel, représentent le fait que le fluent *f* est présent au niveau *i* si et seulement si *f*(*i*) a la valeur de vérité *vrai*. Ceci signifie que *f* est présent dans l'état courant du monde après l'application successive de toutes les actions du niveau 1 jusqu'au niveau *i* qui sont associées à des propositions (voir ci-dessus) ayant la valeur de vérité *vrai*.

Remarque : la présence de parenthèses dans l'écriture d'une proposition ne doit pas être confondue avec une notation de prédicats de la logique du premier ordre. Ces parenthèses peuvent tout à fait être remplacées par un autre symbole : *A*(1) pourrait s'écrire *A*−1, voire même *A*#1 ou encore *A*1.

2.1.2. Codage MK99[7]

Ce codage se traduit par les règles suivantes (cf. Figure 27) :

1. **État initial et but :** les fluents de l'état initial sont vrais au niveau 0, tous ceux qui n'en font pas partie sont faux au niveau 0, et les fluents du but sont vrais au niveau *k*.
2. **Préconditions et effets des actions :** une action du niveau *i* implique la conjonction de ses préconditions au niveau *i*−1, de ses ajouts au niveau *i* et des négations de ses retraits au niveau *i*.
3. **Frame-axiomes explicatifs :** si un fluent vrai au niveau *i*−1 devient faux au niveau *i*, alors la disjonction des actions du niveau *i* qui peuvent le rendre faux doit être vraie : une action au moins qui le retire doit avoir été appliquée. Si un fluent faux au niveau *i*−1 devient vrai au niveau *i*, alors la disjonction des actions du niveau *i* qui peuvent l'établir est vraie : une action au moins qui l'établit doit avoir été appliquée.

$$\textbf{1.}\quad \left(\bigwedge_{f\in I} f(0)\right)\wedge\left(\bigwedge_{f\in(F-I)} \neg f(0)\right)\wedge\left(\bigwedge_{f\in G} f(k)\right)$$

$$\textbf{2.}\quad \bigwedge_{i\in[1,k]}\ \bigwedge_{a\in O}\left(a(i) \Rightarrow \left(\bigwedge_{f\in \mathrm{Prec}(a)} f(i-1)\right)\wedge\left(\bigwedge_{f\in \mathrm{Add}(a)} f(i)\right)\wedge\left(\bigwedge_{f\in \mathrm{Del}(a)} \neg f(i)\right)\right)$$

$$\textbf{3.}\quad \bigwedge_{i\in[1,k]}\ \bigwedge_{f\in F}\left(\left(f(i-1)\wedge\neg f(i)\right) \Rightarrow \bigvee_{a\in O \mid f\in \mathrm{Del}(a)} a(i)\right)$$

$$\bigwedge_{i\in[1,k]}\ \bigwedge_{f\in F}\left(\left(\neg f(i-1)\wedge f(i)\right) \Rightarrow \bigvee_{a\in O \mid f\in \mathrm{Add}(a)} a(i)\right)$$

Figure 27 : Codage MK99 dans les espaces d'états avec frame-axiomes explicatifs

La Figure 28 donne le codage du problème $\prod$ en suivant ces schémas, et la Figure 29 présente la restriction des modèles (trouvés en résolvant ce codage) aux propositions qui correspondent à des symboles d'actions. Ces restrictions des modèles devraient normalement correspondre aux plans-solutions du problème de planification ; or ici, deux des modèles trouvés ne correspondent pas à des plans-solutions : seuls les modèles 1 et 3 sont associés à des plans valides qui résolvent le problème.

[7] Nous utiliserons l'expression "codage MK99" pour chacun des codages étudiés pour faire référence à leur version originelle dans [Mali et Kambhampati 1999].

1. $a(0) \wedge c(0) \wedge \neg b(0) \wedge \neg d(0) \wedge \neg e(0) \wedge e(2)$

2. $\big(A(1) \Rightarrow a(0) \wedge b(1) \wedge \neg c(1)\big) \wedge \big(B(1) \Rightarrow c(0) \wedge d(1)\big) \wedge \big(C(1) \Rightarrow d(0) \wedge e(1) \wedge \neg d(1)\big) \wedge$
$\big(A(2) \Rightarrow a(1) \wedge b(2) \wedge \neg c(2)\big) \wedge \big(B(2) \Rightarrow c(1) \wedge d(2)\big) \wedge \big(C(2) \Rightarrow d(1) \wedge e(2) \wedge \neg d(2)\big)$

3. $\big(a(0) \wedge \neg a(1) \Rightarrow \bot\big) \wedge \big(b(0) \wedge \neg b(1) \Rightarrow \bot\big) \wedge \big(c(0) \wedge \neg c(1) \Rightarrow A(1)\big) \wedge \big(d(0) \wedge \neg d(1) \Rightarrow C(1)\big) \wedge$
$\big(e(0) \wedge \neg e(1) \Rightarrow \bot\big) \wedge \big(a(1) \wedge \neg a(2) \Rightarrow \bot\big) \wedge \big(b(1) \wedge \neg b(2) \Rightarrow \bot\big) \wedge \big(c(1) \wedge \neg c(2) \Rightarrow A(2)\big) \wedge$
$\big(d(1) \wedge \neg d(2) \Rightarrow C(2)\big) \wedge \big(e(1) \wedge \neg e(2) \Rightarrow \bot\big) \wedge \big(\neg a(0) \wedge a(1) \Rightarrow \bot\big) \wedge \big(\neg b(0) \wedge b(1) \Rightarrow A(1)\big) \wedge$
$\big(\neg c(0) \wedge c(1) \Rightarrow \bot\big) \wedge \big(\neg d(0) \wedge d(1) \Rightarrow B(1)\big) \wedge \big(\neg e(0) \wedge e(1) \Rightarrow C(1)\big) \wedge \big(\neg a(1) \wedge a(2) \Rightarrow \bot\big) \wedge$
$\big(\neg b(1) \wedge b(2) \Rightarrow A(2)\big) \wedge \big(\neg c(1) \wedge c(2) \Rightarrow \bot\big) \wedge \big(\neg d(1) \wedge d(2) \Rightarrow B(2)\big) \wedge \big(\neg e(1) \wedge e(2) \Rightarrow C(2)\big)$

Clauses : 42 **Propositions :** 21 (6 actions, 15 fluents)

Figure 28 : Codage MK99 dans les espaces d'états du problème Π

Modèles :
1. $\{B(1), C(2)\}$
2. $\{A(1), B(1), C(2)\}$
3. $\{B(1), A(2), C(2)\}$
4. $\{A(1), B(1), A(2), C(2)\}$

Figure 29 : Modèles trouvés pour le codage MK99 dans les espaces d'états avec frame-axiomes explicatifs du problème Π

2.1.3. Problèmes et modifications

1. **Pas de prise en compte des interactions croisées :** il manque les règles interdisant à une action de retirer une précondition d'une autre action à un même niveau (interactions croisées), en particulier : *A*(1) retire *c* qui est une précondition de *B*(1). Or, il est précisé dans la section 2.2 de [Mali et Kambhampati 1999] que des actions peuvent être exécutées à un même niveau, mais qu'une action ne doit retirer ni une précondition, ni un effet d'une autre action. Ceci correspond bien évidemment à la relation d'indépendance que nous avons étudiée dans la partie consacrée à Graphplan (cf. section III.4.1, page 66). Dans [Mali et Kambhampati 1998], on trouve une règle pour prendre en compte les interactions croisées dans un codage dans les espaces d'états (cf. Figure 30).

4. $\bigwedge_{i \in [1,k]} \; \bigwedge_{a_1 \in O} \; \bigwedge_{f \in \mathrm{Prec}(a_1)} \; \bigwedge_{a_2 \in O \mid a_2 \neq a_1 \wedge f \in \mathrm{Del}(a_2)} \neg\big(a_1(i) \wedge a_2(i)\big)$

Figure 30 : Codage [Mali et Kambhampati 1998] des interactions croisées pour les espaces d'états

Cette règle pourrait s'appliquer ici, mais ne produit pas exactement le résultat désiré. En effet, si une action *A* retire une précondition d'une action *B* et que *B* retire une précondition de *A*, on ajoutera les deux clauses équivalentes suivantes pour chaque niveau *i* du codage :

$$\neg\big(A(i) \wedge B(i)\big) \quad \text{et} \quad \neg\big(B(i) \wedge A(i)\big).$$

De plus, la même clause sera ajoutée plusieurs fois à la base si une action retire plusieurs préconditions d'une même action. Et enfin, si deux actions ont des effets contradictoires, ce n'est pas la peine de rajouter (si besoin était) une clause pour les interactions croisées : ceci est pris en compte par la règle 2. Nous proposons donc de remplacer la règle 4 par la règle 4a (cf. Figure 31). Grâce à cette règle, une clause d'exclusion mutuelle entre deux actions est ajoutée à la base si elle n'a pas été déjà ajoutée, si les deux actions ont une interaction croisée, et si les deux actions n'ont pas d'effets contradictoires. Dans notre problème, on ajoutera les clauses :

$$\neg\big(A(1) \wedge B(1)\big) \quad \text{et} \quad \neg\big(A(2) \wedge B(2)\big).$$

On trouve maintenant uniquement les modèles 1 et 3 qui correspondent aux plans-solutions.

2. **Propositions et clauses inutiles :** la règle 3 génère des clauses inutiles lorsqu'un fluent n'est ajouté ni par l'état initial ni par une action, ou bien lorsque ce fluent n'est retiré par aucune action. Par exemple, le fluent *a* est ajouté par l'état initial mais n'est jamais retiré, donc la clause $\left(a(0)\wedge\neg a(1)\Rightarrow\bot\right)$ est inutile. On peut alors se passer des frame-axiomes explicatifs concernant un tel fluent. Nous proposons donc de remplacer la condition d'appartenance d'un fluent à l'ensemble des fluents dans la règle 3 de la façon suivante :
   - Pour les frame-axiomes de retrait : il faut que le fluent puisse exister à un moment donné, donc être élément de *I* ou de *Fa*. Il faut qu'il puisse être retiré par une action donc être élément de *Fd*. Il doit donc être élément de $(I \cup Fa) \cap Fd$.
   - Pour les frame-axiomes d'ajout : il faut que le fluent puisse ne pas exister à un moment donné, donc ne pas être élément de *I* ou être élément de *Fd*. Il faut aussi qu'il puisse être ajouté par une action donc être élément de *Fa*. Il doit donc être élément de $((F-I) \cup Fd) \cap Fa$.

La Figure 31 donne les règles modifiées du codage dans les espaces d'états avec frame-axiomes explicatifs, et la Figure 32, le codage modifié du problème $\prod$.

1a. $\left(\bigwedge_{f\in I} f(0)\right)\wedge\left(\bigwedge_{f\in(F-I)} \neg f(0)\right)\wedge\left(\bigwedge_{f\in G} f(k)\right)$

2a. $\bigwedge_{i\in[1,k]}\ \bigwedge_{a\in O}\left(a(i)\ \Rightarrow\ \left(\bigwedge_{f\in \mathrm{Prec}(a)} f(i-1)\right)\wedge\left(\bigwedge_{f\in \mathrm{Add}(a)} f(i)\right)\wedge\left(\bigwedge_{f\in \mathrm{Del}(a)} \neg f(i)\right)\right)$

3a. $\bigwedge_{i\in[1,k]}\ \bigwedge_{f\in((I\cup Fa)\cap Fd)}\left(\left(f(i-1)\wedge\neg f(i)\right)\ \Rightarrow\ \bigvee_{a\in O\,|\,f\in \mathrm{Del}(a)} a(i)\right)$

$\bigwedge_{i\in[1,k]}\ \bigwedge_{f\in(((F-I)\cup Fd)\cap Fa)}\left(\left(\neg f(i-1)\wedge f(i)\right)\ \Rightarrow\ \bigvee_{a\in O\,|\,f\in \mathrm{Add}(a)} a(i)\right)$

4a. $\bigwedge_{i\in[1,k]}\ \bigwedge_{(a_m,a_n)\in O^2\,|\,m<n\wedge\left((\mathrm{Prec}(a_m)\cap\mathrm{Del}(a_n)\neq\varnothing)\vee(\mathrm{Del}(a_m)\cap\mathrm{Prec}(a_n)\neq\varnothing)\right)\wedge\ (\mathrm{Add}(a_m)\cap\mathrm{Del}(a_n)=\varnothing)\wedge(\mathrm{Del}(a_m)\cap\mathrm{Add}(a_n)=\varnothing)} \neg\left(a_m(i)\wedge a_n(i)\right)$

Figure 31 : Codage modifié dans les espaces d'états avec frame-axiomes explicatifs

1a. $a(0)\wedge c(0)\wedge\neg b(0)\wedge\neg d(0)\wedge\neg e(0)\wedge e(2)$

2a. $\left(A(1)\Rightarrow a(0)\wedge b(1)\wedge\neg c(1)\right)\wedge\left(B(1)\Rightarrow c(0)\wedge d(1)\right)\wedge\left(C(1)\Rightarrow d(0)\wedge e(1)\wedge\neg d(1)\right)\wedge$
$\left(A(2)\Rightarrow a(1)\wedge b(2)\wedge\neg c(2)\right)\wedge\left(B(2)\Rightarrow c(1)\wedge d(2)\right)\wedge\left(C(2)\Rightarrow d(1)\wedge e(2)\wedge\neg d(2)\right)$

3a. $\left(c(0)\wedge\neg c(1)\Rightarrow A(1)\right)\wedge\left(d(0)\wedge\neg d(1)\Rightarrow C(1)\right)\wedge\left(c(1)\wedge\neg c(2)\Rightarrow A(2)\right)\wedge$
$\left(d(1)\wedge\neg d(2)\Rightarrow C(2)\right)\wedge\left(\neg b(0)\wedge b(1)\Rightarrow A(1)\right)\wedge\left(\neg d(0)\wedge d(1)\Rightarrow B(1)\right)\wedge$
$\left(\neg e(0)\wedge e(1)\Rightarrow C(1)\right)\wedge\left(\neg b(1)\wedge b(2)\Rightarrow A(2)\right)\wedge\left(\neg d(1)\wedge d(2)\Rightarrow B(2)\right)\wedge$
$\left(\neg e(1)\wedge e(2)\Rightarrow C(2)\right)$

4a. $\neg\left(A(1)\wedge B(1)\right)\wedge\neg\left(A(2)\wedge B(2)\right)$

Clauses : 34 **Propositions :** 21 (6 actions, 15 fluents)

Figure 32 : Codage modifié dans les espaces d'états avec frame-axiomes explicatifs du problème $\prod$

2.2. Codage dans les espaces d'états avec no-ops

Ce codage diffère très peu du précédent. L'idée d'utiliser des no-ops à la place des frame-axiomes explicatifs vient du planificateur Graphplan et a été initiée dans [Kautz, McAllester et Selman 1996]. La version que nous présentons ici est un peu différente, puisque nous représentons les effets des actions (ajouts et retraits) afin de coder les interactions entre effets (au lieu de clauses binaires d'exclusion entre actions) ; ceci afin de montrer les différences avec le codage avec frame-axiomes explicatifs.

Les seuls symboles nouveaux par rapport au codage précédent sont les propositions associées aux no-ops : elles sont de la forme N_*f*(*i*), où *f* est un fluent et *i* est un entier naturel, et représentent le fait que le fluent *f* présent au niveau *i*−1 est présent au niveau *i* si N_*f*(*i*) a la valeur de vérité *vrai*.

Il faut rajouter à la règle 2 du codage précédent le fait que le no-op d'un fluent *f* appliqué au niveau *i* implique sa précondition au niveau *i*−1 et son effet au niveau *i*, selon le même principe que les autres actions. Par contre, la règle 3 qui concernait les frame-axiomes explicatifs est entièrement modifiée pour prendre en compte les no-ops. L'intérêt essentiel des no-ops est que l'on n'a plus à se soucier du cas où un fluent présent au niveau *i*−1 n'est plus présent au niveau *i*. Dans le codage précédent, il fallait une règle disant que l'on a appliqué une action qui l'a retiré ; maintenant, le no-op est automatiquement appliqué sauf dans le cas où une action retire le fluent. La nouvelle règle exprime donc le fait que si un fluent est présent à un niveau, c'est soit parce que l'on a appliqué son no-op soit parce que l'on a appliqué une action qui le produit.

La Figure 33 donne les règles du codage dans les espaces d'états avec no-ops, et la Figure 34 le codage de notre problème suivant ces règles. On peut constater que ce codage produit plus de clauses et de propositions que le codage dans les espaces d'états avec frame-axiomes explicatifs ; par contre, dans ce dernier, on avait 10 clauses de taille 3, toutes les autres étant de taille 1 ou 2. Maintenant, on n'a plus que 6 clauses de taille 3. L'utilisation de no-ops est en fait une factorisation du codage avec frame-axiomes explicatifs. L'introduction des symboles de no-ops produit des clauses de taille 2 par la règle 2, ce qui permet d'éliminer des clauses de taille supérieure ou égale à 3 de la règle 3.

1. $\left(\bigwedge_{f\in I} f(0)\right)\wedge\left(\bigwedge_{f\in(F-I)} \neg f(0)\right)\wedge\left(\bigwedge_{f\in G} f(k)\right)$

2. $\bigwedge_{i\in[1,k]}\ \bigwedge_{a\in O}\left(a(i) \Rightarrow \left(\bigwedge_{f\in \mathrm{Prec}(a)} f(i-1)\right)\wedge\left(\bigwedge_{f\in \mathrm{Add}(a)} f(i)\right)\wedge\left(\bigwedge_{f\in \mathrm{Del}(a)} \neg f(i)\right)\right)$

$\bigwedge_{i\in[1,k]}\ \bigwedge_{f\in(I\cup Fa)}\left(\mathrm{N\_}f(i) \Rightarrow f(i-1)\wedge f(i)\right)$

3. $\bigwedge_{i\in[1,k]}\ \bigwedge_{f\in(I\cup Fa)}\left(f(i) \Rightarrow \mathrm{N\_}f(i)\vee\left(\bigvee_{a\in O \mid f\in \mathrm{Add}(a)} a(i)\right)\right)$

4. $\bigwedge_{i\in[1,k]}\ \bigwedge_{(a_m,a_n)\in O^2 \mid m<n\wedge\left((\mathrm{Prec}(a_m)\cap\mathrm{Del}(a_n)\neq\varnothing)\vee(\mathrm{Del}(a_m)\cap\mathrm{Prec}(a_n)\neq\varnothing)\right)\wedge (\mathrm{Add}(a_m)\cap\mathrm{Del}(a_n)=\varnothing)\wedge(\mathrm{Del}(a_m)\cap\mathrm{Add}(a_n)=\varnothing)} \neg\left(a_m(i)\wedge a_n(i)\right)$

Figure 33 : Codage dans les espaces d'états avec no-ops

1. $a(0)\wedge c(0)\wedge\neg b(0)\wedge\neg d(0)\wedge\neg e(0)\wedge e(2)$

2. $\left(A(1)\Rightarrow a(0)\wedge b(1)\wedge\neg c(1)\right)\wedge\left(B(1)\Rightarrow c(0)\wedge d(1)\right)\wedge\left(C(1)\Rightarrow d(0)\wedge e(1)\wedge\neg d(1)\right)\wedge$
$\left(A(2)\Rightarrow a(1)\wedge b(2)\wedge\neg c(2)\right)\wedge\left(B(2)\Rightarrow c(1)\wedge d(2)\right)\wedge\left(C(2)\Rightarrow d(1)\wedge e(2)\wedge\neg d(2)\right)$
$\left(\mathrm{N\_}a(1)\Rightarrow a(0)\wedge a(1)\right)\wedge\left(\mathrm{N\_}b(1)\Rightarrow b(0)\wedge b(1)\right)\wedge\left(\mathrm{N\_}c(1)\Rightarrow c(0)\wedge c(1)\right)\wedge$
$\left(\mathrm{N\_}d(1)\Rightarrow d(0)\wedge d(1)\right)\wedge\left(\mathrm{N\_}e(1)\Rightarrow e(0)\wedge e(1)\right)\wedge\left(\mathrm{N\_}a(2)\Rightarrow a(1)\wedge a(2)\right)\wedge$
$\left(\mathrm{N\_}b(2)\Rightarrow b(1)\wedge b(2)\right)\wedge\left(\mathrm{N\_}c(2)\Rightarrow c(1)\wedge c(2)\right)\wedge\left(\mathrm{N\_}d(2)\Rightarrow d(1)\wedge d(2)\right)\wedge$
$\left(\mathrm{N\_}e(2)\Rightarrow e(1)\wedge e(2)\right)$

3. $\left(a(1)\Rightarrow \mathrm{N\_}a(1)\right)\wedge\left(b(1)\Rightarrow \mathrm{N\_}b(1)\vee A(1)\right)\wedge\left(c(1)\Rightarrow \mathrm{N\_}c(1)\right)\wedge\left(d(1)\Rightarrow \mathrm{N\_}d(1)\vee B(1)\right)\wedge$
$\left(e(1)\Rightarrow \mathrm{N\_}e(1)\vee C(1)\right)\wedge\left(a(2)\Rightarrow \mathrm{N\_}a(2)\right)\wedge\left(b(2)\Rightarrow \mathrm{N\_}b(2)\vee A(2)\right)\wedge\left(c(2)\Rightarrow \mathrm{N\_}c(2)\right)\wedge$
$\left(d(2)\Rightarrow \mathrm{N\_}d(2)\vee B(2)\right)\wedge\left(e(2)\Rightarrow \mathrm{N\_}e(2)\vee C(2)\right)$

4. $\neg\left(A(1)\wedge B(1)\right)\wedge\neg\left(A(2)\wedge B(2)\right)$

Clauses : 54 **Propositions :** 31 (6 actions, 10 no−ops, 15 fluents)

Figure 34 : Codage dans les espaces d'états avec no-ops du problème Π

3. Codages dans les espaces de plans

Contrairement aux codages que l'on vient d'étudier, les codages basés sur les espaces de plans ne s'intéressent plus uniquement aux transitions entre niveaux successifs du plan. Ils sont fondés sur la notion de *causalité* entre les actions du plan dans son ensemble. Dans les codages dans les espaces d'états, la cause première de la présence d'un fluent à un niveau peut être l'application soit d'une action, soit d'un frame-axiome du même niveau. Dans ce dernier cas, la cause de la présence de ce frame-axiome est la présence du fluent à un niveau antérieur, etc. De plus, ce fluent peut être utilisé par d'autres actions ; ce qui fait que la notion de causalité entre l'action qui crée le fluent et l'action qui l'utilise n'apparaît pas clairement. Les codages que nous allons étudier maintenant explicitent cette relation de cause à effet entre les actions, ce qui implique l'obligation de *protéger* l'effet d'une action s'il établit la précondition d'une autre action.

De plus, on ne considère plus de la même manière l'insertion des actions dans le plan : une action n'apparaît pas à un niveau du plan parce que ses préconditions sont satisfaites au niveau précédent (ce qui est une façon linéaire de voir les choses), mais parce qu'une action qui la précède les établit et qu'une action qui la suit réclame ses effets (ce qui est une façon causale de voir les choses).

Les variantes du codage dans les espaces de plans proviennent des différentes stratégies utilisées pour l'insertion des actions dans le plan (n'importe où dans le plan, ou bien suivant un ordre préétabli des niveaux du plan...) et la protection des liens de cause à effet (promotion, demotion, chevalier blanc... cf. [Jacopin 1993] [Weld 1994] [Kambhampati 1997]).

Dans les codages dans les espaces de plans, les actions ne sont pas traitées de la même façon que dans les codages dans les espaces d'états. Dans ces derniers, on utilise directement les actions et les fluents dans les règles qui gèrent leur insertion dans le plan, ainsi que dans les frame-axiomes. Maintenant, le raisonnement ne va plus se faire sur toutes les actions à la fois mais sur les actions qui font (ou peuvent faire) partie d'un plan-solution. Comme a priori, on ne connaît pas ces actions, il faut établir une correspondance entre toutes les actions disponibles et les actions qui font partie des plans-solutions : c'est le rôle de la partie commune des codages dans les espaces de plans, et c'est la raison pour laquelle est introduite la notion d'*étape*. Dans les codages originels de [Kautz, McAllester et Selman 1996] et [Mali et Kambhampati 1999], les actions qui peuvent faire partie d'un plan sont associées à des symboles qui représentent les étapes. Un ordre sur ces étapes détermine un ordre total pour l'exécution des actions. Nous allons modifier cette définition pour pouvoir introduire le parallélisme dans les codages dans les espaces de plans. Dorénavant, une étape sera un ensemble d'actions indépendantes deux à deux, donc exécutables en parallèle. Pour les codages originaux, les étapes seront limitées à des singletons par le biais de la règle de la partie commune qui interdit tout parallélisme. De plus, nous assimilerons une étape à une action unique dont les ensembles de préconditions, d'ajouts et de retraits sont respectivement l'union des préconditions, l'union des ajouts et l'union des retraits des actions appartenant à l'étape. Nous dirons alors qu'une étape a pour précondition un fluent, et ajoute (ou retire) un fluent.

3.1. Syntaxe et sémantique des propositions.

Les règles des différentes formes de codages dans les espaces de plans produisent plusieurs formes de propositions :

1. Les propositions de la forme $(p_i = a)$, où p_i est une étape du plan et a est une action, représentent le fait que l'étape p_i est l'action a si et seulement si $(p_i = a)$ a la valeur de vérité *vrai*. Selon le codage MK99, une étape représente donc une action unique et il n'y a pas de parallélisme.

2. Les propositions de la forme $(a \in p_i)$, où p_i est une étape du plan et a est une action, représentent le fait que l'action a appartient à l'étape p_i si et seulement si $(a \in p_i)$ a la valeur de vérité *vrai*. Les propositions de cette forme remplacerons les propositions de la forme précédente dans les modifications que nous apporterons aux codages dans les espaces de plans, et considèrent qu'une étape représente un ensemble d'actions. Ceci nous permettra donc d'introduire du parallélisme dans ces codages.

3. Les propositions de la forme $Adds(p_i, f)$, où p_i est une étape du plan et f est un fluent, représentent le fait que l'étape p_i contient une action qui ajoute le fluent f si et seulement si $Adds(p_i, f)$ a la valeur de vérité *vrai*.

4. Les propositions de la forme $Needs(p_i, f)$, où p_i est une étape du plan et f est un fluent, représentent le fait que l'étape p_i contient une action qui possède f comme précondition si et seulement si $Needs(p_i, f)$ a la valeur de vérité *vrai*.

5. Les propositions de la forme *Dels*(p_i, *f*), où p_i est une étape du plan et *f* est un fluent, représentent le fait que l'étape p_i contient une action qui retire le fluent *f* si et seulement si *Dels*(p_i, *f*) a la valeur de vérité *vrai*.

6. Les propositions de la forme $p_i \xrightarrow{f} p_j$, où p_i et p_j sont deux étapes du plan et *f* est un fluent, représentent un lien causal, c'est-à-dire le fait que l'étape p_i produit le fluent *f* qui est une précondition de l'étape p_j si et seulement si $p_i \xrightarrow{f} p_j$ a la valeur de vérité *vrai*.

7. Les propositions de la forme $(p_i \ll p_j)$, où p_i et p_j sont deux étapes du plan, représentent le fait que l'étape p_i précède l'étape p_j, c'est-à-dire que l'action associée à p_i (dans le cas du codage MK99 de la partie commune) ou l'ensemble d'actions indépendantes associées à p_i (dans le cas du codage modifié de la partie commune) doivent être exécutés avant la ou les actions associées à l'étape p_j, si et seulement si $(p_i \ll p_j)$ a la valeur de vérité *vrai*. Ces propositions sont utilisées uniquement pour représenter un ordre partiel sur les étapes (cf. section 4.3). Dans les codages où l'on considère que les étapes sont ordonnées suivant leur numéro d'indice (cf. section 4.4 et section 4.5), elles sont inutiles.

3.2. Partie commune des codages dans les espaces de plans

3.2.1. Codage MK99

Le codage proposé dans [Mali et Kambhampati 1999] reprend la seule étude qui avait été faite jusqu'à présent sur les codages dans les espaces de plans [Kautz, McAllester et Selman 1996]. Dans [Kautz, McAllester et Selman 1996], un seul type de codage était étudié : le codage dans les espaces de plans avec liens causaux, protection d'intervalles et ordre partiel sur les étapes. Dans [Mali et Kambhampati 1999], deux variantes de ce codage sont proposées, et ont une complexité (en espace) plus restreinte. Ce que nous appelons la partie commune de ces codages correspond donc à toutes les règles du codage [Kautz, McAllester et Selman 1996] qui ne gèrent pas l'insertion des étapes dans le plan ou la protection des intervalles.

Une remarque importante que l'on peut faire avant d'étudier ces règles, est que le parallélisme n'est pas autorisé dans ce codage. En effet, chaque étape du plan correspond à une action, sachant que les étapes sont exécutées l'une après l'autre suivant un certain ordre. Ceci est assez paradoxal, dans la mesure où traditionnellement, les algorithmes de recherche dans les espaces d'états construisent des plans séquentiels (contrairement aux codages que l'on a vu), et que les algorithmes de recherche dans les espaces de plans construisent des plans parallèles. Nous verrons par la suite qu'une légère modification du codage de la partie commune permet de construire des plans parallèles.

Le codage MK99 de la partie commune se traduit par les règles suivantes (cf. Figure 35) :

1. **Correspondance étape ↔ action :** une étape peut être soit une action, soit *Φ* (l'action nulle, ou no-op).
2. **Unicité des étapes :** une étape correspond à une et une seule action (ou à *Φ*) à un niveau donné du plan. C'est cette règle qui interdit le parallélisme entre les actions.
3. **État initial et but :** l'état initial et le but sont considérés respectivement par l'étape *I* et l'étape *G*. Les ajouts de *I* sont vrais, tous les fluents qui ne sont pas dans *I* sont faux, et les préconditions du but sont vraies. Il n'y a pas pour ces étapes de considérations de niveau : implicitement, *I* est l'étape du niveau 0, et les préconditions de *G* peuvent être établies n'importe quand, *G* étant l'étape du niveau *k*+1.
4. **Correspondance (préconditions et effets d'une action) → (préconditions et effets d'une étape) :** si l'étape est associée à une action (différente de *Φ*), alors la conjonction de ses ajouts, de ses préconditions et des retraits de cette dernière est vraie.
5. **Correspondance (préconditions et effets d'une étape) → (action correspondante) :** si une étape ajoute, retire, ou a pour précondition un fluent, alors cette étape peut correspondre à n'importe quelle action qui a le même comportement vis-à-vis de ce fluent.

La Figure 36 montre l'application de ces règles pour le codage de notre problème. On peut déjà constater qu'il y a plus de deux fois plus de clauses que pour le codage dans les espaces d'états !

1. $\bigwedge_{i\in[1,k]}\left(\bigvee_{a\in O}(p_i=a)\vee(p_i=\Phi)\right)$

2. $\bigwedge_{i\in[1,k]}\ \bigwedge_{a_1\in O}\ \bigwedge_{a_2\in O\mid a_1\neq a_2}\neg\left((p_i=a_1)\wedge(p_i=a_2)\right)$

$\bigwedge_{i\in[1,k]}\ \bigwedge_{a\in O}\neg\left((p_i=a)\wedge(p_i=\Phi)\right)$

3. $\left(\bigwedge_{f\in I}Adds(I,f)\right)\wedge\left(\bigwedge_{f\in(F-I)}\neg Adds(I,f)\right)\wedge\left(\bigwedge_{f\in G}Needs(G,f)\right)$

4. $\bigwedge_{i\in[1,k]}\ \bigwedge_{a\in O}\left((p_i=a)\Rightarrow\left(\bigwedge_{f\in\mathrm{Add}(a)}Adds(p_i,f)\right)\wedge\left(\bigwedge_{f\in\mathrm{Prec}(a)}Needs(p_i,f)\right)\wedge\left(\bigwedge_{f\in\mathrm{Del}(a)}Dels(p_i,f)\right)\right)$

5. $\bigwedge_{i\in[1,k]}\ \bigwedge_{f\in F}\left(Adds(p_i,f)\ \Rightarrow\bigvee_{a\in O\mid f\in\mathrm{Add}(a)}(p_i=a)\right)$

$\bigwedge_{i\in[1,k]}\ \bigwedge_{f\in F}\left(Dels(p_i,f)\ \Rightarrow\bigvee_{a\in O\mid f\in\mathrm{Del}(a)}(p_i=a)\right)$

$\bigwedge_{i\in[1,k]}\ \bigwedge_{f\in F}\left(Needs(p_i,f)\ \Rightarrow\bigvee_{a\in O\mid f\in\mathrm{Prec}(a)}(p_i=a)\right)$

Figure 35 : Partie commune du codage MK99 dans les espaces de plans

1. $\left((p_1=A)\vee(p_1=B)\vee(p_1=C)\vee(p_1=\Phi)\right)\wedge\left((p_2=A)\vee(p_2=B)\vee(p_2=C)\vee(p_2=\Phi)\right)$

2. $\neg\left((p_1=A)\wedge(p_1=B)\right)\wedge\neg\left((p_1=A)\wedge(p_1=C)\right)\wedge\neg\left((p_1=B)\wedge(p_1=A)\right)\wedge\neg\left((p_1=B)\wedge(p_1=C)\right)\wedge$
$\neg\left((p_1=C)\wedge(p_1=A)\right)\wedge\neg\left((p_1=C)\wedge(p_1=B)\right)\wedge\neg\left((p_2=A)\wedge(p_2=B)\right)\wedge\neg\left((p_2=A)\wedge(p_2=C)\right)\wedge$
$\neg\left((p_2=B)\wedge(p_2=A)\right)\wedge\neg\left((p_2=B)\wedge(p_2=C)\right)\wedge\neg\left((p_2=C)\wedge(p_2=A)\right)\wedge\neg\left((p_2=C)\wedge(p_2=B)\right)\wedge$
$\neg\left((p_1=A)\wedge(p_1=\Phi)\right)\wedge\neg\left((p_1=B)\wedge(p_1=\Phi)\right)\wedge\neg\left((p_1=C)\wedge(p_1=\Phi)\right)\wedge$
$\neg\left((p_2=A)\wedge(p_2=\Phi)\right)\wedge\neg\left((p_2=B)\wedge(p_2=\Phi)\right)\wedge\neg\left((p_2=C)\wedge(p_2=\Phi)\right)$

3. $Adds(I,a)\wedge Adds(I,c)\wedge\neg Adds(I,b)\wedge\neg Adds(I,d)\wedge\neg Adds(I,e)\wedge Needs(G,e)$

4. $\left((p_1=A)\Rightarrow Needs(p_1,a)\wedge Adds(p_1,b)\wedge Dels(p_1,c)\right)\wedge\left((p_1=B)\Rightarrow Needs(p_1,c)\wedge Adds(p_1,d)\right)\wedge$
$\left((p_1=C)\Rightarrow Needs(p_1,d)\wedge Adds(p_1,e)\wedge Dels(p_1,d)\right)\wedge\left((p_2=A)\Rightarrow Needs(p_2,a)\wedge Adds(p_2,b)\wedge Dels(p_2,c)\right)\wedge$
$\left((p_2=B)\Rightarrow Needs(p_2,c)\wedge Adds(p_2,d)\right)\wedge\left((p_2=C)\Rightarrow Needs(p_2,d)\wedge Adds(p_2,e)\wedge Dels(p_2,d)\right)$

5. $\left(Adds(p_1,a)\Rightarrow\bot\right)\wedge\left(Adds(p_1,b)\Rightarrow(p_1=A)\right)\wedge\left(Adds(p_1,c)\Rightarrow\bot\right)\wedge\left(Adds(p_1,d)\Rightarrow(p_1=B)\right)\wedge$
$\left(Adds(p_1,e)\Rightarrow(p_1=C)\right)\wedge\left(Adds(p_2,a)\Rightarrow\bot\right)\wedge\left(Adds(p_2,b)\Rightarrow(p_2=A)\right)\wedge\left(Adds(p_2,c)\Rightarrow\bot\right)\wedge$
$\left(Adds(p_2,d)\Rightarrow(p_2=B)\right)\wedge\left(Adds(p_2,e)\Rightarrow(p_2=C)\right)$
$\left(Needs(p_1,a)\Rightarrow(p_1=A)\right)\wedge\left(Needs(p_1,b)\Rightarrow\bot\right)\wedge\left(Needs(p_1,c)\Rightarrow(p_1=B)\right)\wedge\left(Needs(p_1,d)\Rightarrow(p_2=C)\right)\wedge$
$\left(Needs(p_1,e)\Rightarrow\bot\right)\wedge\left(Needs(p_2,a)\Rightarrow(p_2=A)\right)\wedge\left(Needs(p_2,b)\Rightarrow\bot\right)\wedge\left(Needs(p_2,c)\Rightarrow(p_2=B)\right)\wedge$
$\left(Needs(p_2,d)\Rightarrow(p_2=C)\right)\wedge\left(Needs(p_2,e)\Rightarrow\bot\right)$
$\left(Dels(p_1,a)\Rightarrow\bot\right)\wedge\left(Dels(p_1,b)\Rightarrow\bot\right)\wedge\left(Dels(p_1,c)\Rightarrow(p_1=A)\right)\wedge\left(Dels(p_1,d)\Rightarrow(p_1=C)\right)\wedge$
$\left(Dels(p_1,e)\Rightarrow\bot\right)\wedge\left(Dels(p_2,a)\Rightarrow\bot\right)\wedge\left(Dels(p_2,b)\Rightarrow\bot\right)\wedge\left(Dels(p_2,c)\Rightarrow(p_2=A)\right)\wedge$
$\left(Dels(p_2,d)\Rightarrow(p_1=C)\right)\wedge\left(Dels(p_2,e)\Rightarrow\bot\right)$

Clauses : 78 **Propositions :** 44 (8 actions, 36 fluents)

Figure 36 : Partie commune du codage MK99 dans les espaces de plans du problème $\prod$

3.2.2. Modifications

Nous pouvons faire plusieurs remarques concernant ce codage :

1. **L'action Φ :** pourquoi considérer cette action différemment des autres actions ? En effet, il serait bien plus simple de l'inclure dans l'ensemble *O* des actions de base. Elle serait alors traitée comme les autres. De plus, elle n'est utile que si on code un problème sur plus de niveaux qu'il n'en est nécessaire pour le résoudre. Afin d'éviter ceci, on peut très bien employer l'approche de BLACKBOX [Kautz et Selman 1999] : incrémenter le nombre de niveaux du codage de 1 à chaque échec. On trouve ainsi le plan de taille minimum. Quoi qu'il en soit, ceci est un autre problème et concerne l'utilisation que l'on veut faire d'un tel codage. Nous allons donc supprimer cette action, sachant qu'elle peut ou non faire partie de l'ensemble *O* des actions de base. Les règles 1 et 2 s'en trouvent donc simplifiées. Nous verrons dans la 4ème remarque que nous pouvons même supprimer la règle 1.
2. **Clauses inutiles :** la règle 2 fait apparaître beaucoup de clauses équivalentes. Pour éviter ceci, il suffit de fusionner les deux conjonctions portant sur les actions en une seule conjonction portant sur des couples d'actions, en repérant ces dernières par leur indice associé, et en spécifiant que l'indice de la première action du couple doit être inférieur (strictement) à celui de la seconde action du couple.
3. **Propositions (et clauses) inutiles :** pourquoi faire apparaître (règle 5) des propositions telles que $Adds(p_1, a)$ ou $Needs(p_2, b)$ alors que *a* n'apparaît jamais dans les listes d'ajouts des actions, et que *b* n'est précondition d'aucune action ? Nous proposons donc de différencier les trois sous-ensembles qui constituent l'ensemble *F* des fluents : l'ensemble *Fp* des fluents qui apparaissent dans les préconditions des actions, l'ensemble *Fa* des fluents qui apparaissent dans les ajouts des actions, l'ensemble *Fd* des fluents qui apparaissent dans les retraits des actions. Nous modifions donc la règle 5 pour ne plus faire apparaître de propositions ni de clauses inutiles. De plus, en faisant ceci, on se rend compte que la règle 4 peut être supprimée si on remplace les implications de la règle 5 par des équivalences. Une autre conséquence de ce choix est qu'il n'est plus nécessaire de spécifier dans la règle 3 les fluents qui n'appartiennent pas à l'état initial, si l'on modifie en conséquence les parties spécifiques de chaque codage. Nous allons donc laisser la règle 3 telle quelle, sachant que lorsque l'on modifiera les codages des parties spécifiques, il faudra enlever de la partie commune le codage des fluents qui n'appartiennent pas à l'état initial.
4. **Parallélisme :** pourquoi interdire le parallélisme, alors qu'il peut permettre de développer un codage sur beaucoup moins de niveaux ? En effet, en interdisant le parallélisme, on va devoir considérer toutes les linéarisations possibles d'actions qui pourraient être exécutées en parallèle. Il est extrêmement simple de rajouter le parallélisme dans ce codage : il suffit de considérer que les étapes du plan (les p_i) représentent non plus une action unique du plan, mais un ensemble d'actions indépendantes. Plusieurs modifications sont alors nécessaires : la règle 1 devient inutile, la règle 2 doit être modifiée pour n'exclure que les paires d'actions qui ne peuvent pas être appliquées en parallèle (les actions qui ne sont pas indépendantes), et les formules des règles 2, 4 et 5 signifiant qu'une étape du plan est égale à une action (par ex. $(p_i = a)$) doivent maintenant être modifiées pour représenter l'appartenance d'une action à une étape du plan : $(a \in p_i)$. Une conséquence de ce choix, en particulier la suppression de la règle 1, est que l'action Φ est automatiquement codée par l'absence d'information sur les actions contenues dans une étape. En effet, si un modèle ne fait pas apparaître qu'au moins une action appartient à une étape donnée, c'est que cette étape est l'ensemble vide. On peut donc assimiler une étape vide à une étape contenant seulement l'action Φ. On pourrait aussi envisager d'obliger les étapes à contenir au moins une action, en rétablissant la règle 1 du codage MK99 et en supprimant l'action Φ. Ceci pourrait être intéressant dans la mesure où l'on connaîtrait exactement la taille du plan en nombre d'étapes.

Nous proposons donc un nouveau codage (cf. Figure 37) pour la partie commune des codages dans les espaces de plans. Il en résulte un codage beaucoup plus compact pour notre problème (cf. Figure 38).

1a. $\bigwedge_{i\in[1,k]} \bigwedge_{(a_m,a_n)\in O^2 \mid m<n\wedge\neg(a_m \# a_n)} \neg\Big((a_m\in p_i)\wedge(a_n\in p_i)\Big)$

2a. $\Big(\bigwedge_{f\in I} Adds(I,f)\Big)\wedge\Big(\bigwedge_{f\in G} Needs(G,f)\Big)\wedge\Big(\bigwedge_{f\in(F-I)} \neg Adds(I,f)\Big)$

3a. $\bigwedge_{i\in[1,k]} \bigwedge_{f\in Fa}\Big(Adds(p_i,f) \Leftrightarrow \bigvee_{a\in O \mid f\in \mathrm{Add}(a)} (a\in p_i)\Big)$

$\bigwedge_{i\in[1,k]} \bigwedge_{f\in Fd}\Big(Dels(p_i,f) \Leftrightarrow \bigvee_{a\in O \mid f\in \mathrm{Del}(a)} (a\in p_i)\Big)$

$\bigwedge_{i\in[1,k]} \bigwedge_{f\in Fp}\Big(Needs(p_i,f) \Leftrightarrow \bigvee_{a\in O \mid f\in \mathrm{Prec}(a)} (a\in p_i)\Big)$

Figure 37 : Partie commune modifiée du codage dans les espaces de plans

1a. $\neg\Big((A\in p_1)\wedge(B\in p_1)\Big)\wedge\neg\Big((B\in p_1)\wedge(C\in p_1)\Big)\wedge\neg\Big((A\in p_2)\wedge(B\in p_2)\Big)\wedge\neg\Big((B\in p_2)\wedge(C\in p_2)\Big)$

2a. $Adds(I,a)\wedge Adds(I,c)\wedge\neg Adds(I,b)\wedge\neg Adds(I,d)\wedge\neg Adds(I,e)\wedge Needs(G,e)$

3a. $\Big(Adds(p_1,b)\Leftrightarrow(A\in p_1)\Big)\wedge\Big(Adds(p_1,d)\Leftrightarrow(B\in p_1)\Big)\wedge\Big(Adds(p_1,e)\Leftrightarrow(C\in p_1)\Big)\wedge$
$\Big(Adds(p_2,b)\Leftrightarrow(A\in p_2)\Big)\wedge\Big(Adds(p_2,d)\Leftrightarrow(B\in p_2)\Big)\wedge\Big(Adds(p_2,e)\Leftrightarrow(C\in p_2)\Big)\wedge$
$\Big(Needs(p_1,a)\Leftrightarrow(A\in p_1)\Big)\wedge\Big(Needs(p_1,c)\Leftrightarrow(B\in p_1)\Big)\wedge\Big(Needs(p_1,d)\Leftrightarrow(C\in p_1)\Big)\wedge$
$\Big(Needs(p_2,a)\Leftrightarrow(A\in p_2)\Big)\wedge\Big(Needs(p_2,c)\Leftrightarrow(B\in p_2)\Big)\wedge\Big(Needs(p_2,d)\Leftrightarrow(C\in p_2)\Big)\wedge$
$\Big(Dels(p_1,c)\Leftrightarrow(A\in p_1)\Big)\wedge\Big(Dels(p_1,d)\Leftrightarrow(C\in p_1)\Big)\wedge\Big(Dels(p_2,c)\Leftrightarrow(A\in p_2)\Big)\wedge$
$\Big(Dels(p_2,d)\Leftrightarrow(C\in p_2)\Big)$

Clauses : 42 **Propositions :** 28 (6 actions, 22 fluents)

Figure 38 : Partie commune modifiée du codage dans les espaces de plans du problème Π

3.3. Codage par liens causaux, protection d'intervalles et ordre partiel

Il s'agit de la seule version d'un codage dans les espaces de plans proposée avant [Mali et Kambhampati 1999] ; ce codage a seulement été étudié dans [Kautz, McAllester et Selman 1996]. L'établissement des préconditions se fait par un codage direct des liens causaux de la forme $p_i \xrightarrow{f} p_j$, qui permettent de spécifier que l'étape p_i produit le fluent *f* qui est une précondition de l'étape p_j. La protection de *f* dans l'intervalle compris entre les deux étapes se fait par promotion (l'étape qui menace le fluent *f* est placée avant p_i) ou demotion (l'étape qui menace *f* est placée après p_j). Les numéros des étapes ne correspondent pas à l'ordre d'exécution des actions ; ce dernier est donné par une relation de précédence qui définit un ordre partiel sur les actions. Ce codage peut être utilisé avec les deux versions de la partie commune que nous avons étudiées.

3.3.1. Codage MK99

Il faut rajouter les règles suivantes (cf. Figure 39) à la partie commune du codage dans les espaces de plans :

6. **Établissement des préconditions :** chaque fluent est supporté par un lien causal entre l'étape qui le produit (qui peut être l'état initial) et l'étape qui l'utilise en tant que précondition. On peut remarquer que l'on ne se soucie pas de savoir s'il existe réellement une action qui produit ce fluent et une action qui l'utilise.
7. **Liens causaux :** la présence d'un lien causal supportant un fluent *f* entre une étape p_i et une étape p_j implique que l'étape p_i ajoute *f*, que l'étape p_j a pour précondition *f* et que l'étape p_i précède l'étape p_j.

8. **Protection d'intervalles :** s'il y a un lien causal qui supporte un fluent *f* entre une étape p_i et une étape p_j, et si une étape p_q retire *f*, alors p_q doit précéder p_i (promotion) ou p_j doit précéder p_q (demotion).
9. **Propriétés de la relation de précédence :** transitive, antisymétrique et irréflexive.

6. $\bigwedge_{i\in[1,k]} \bigwedge_{f\in F} \left(Needs(p_i, f) \Rightarrow \left(\bigvee_{j\in[1,k]|\ j\neq i} \left(p_j \xrightarrow{f} p_i \right) \right) \vee I \xrightarrow{f} p_i \right)$

7. $\bigwedge_{i\in[1,k]} \bigwedge_{j\in[1,k]|\ i\neq j} \bigwedge_{f\in F} \left(p_i \xrightarrow{f} p_j \Rightarrow Adds(p_i, f) \wedge Needs(p_j, f) \wedge (p_i \ll p_j) \right)$

8. $\bigwedge_{i\in[1,k]} \bigwedge_{j\in[1,k]|\ i\neq j} \bigwedge_{q\in[1,k]|\ q\neq i\wedge q\neq j} \bigwedge_{f\in F} \left(p_i \xrightarrow{f} p_j \wedge Dels(p_q, f) \Rightarrow (p_q \ll p_i) \vee (p_j \ll p_q) \right)$

9. $\bigwedge_{i\in[1,k]} \bigwedge_{j\in[1,k]|\ i\neq j} \bigwedge_{q\in[1,k]|\ q\neq i\wedge q\neq j} \left((p_i \ll p_j) \wedge (p_j \ll p_q) \Rightarrow (p_i \ll p_q) \right)$

$\bigwedge_{i\in[1,k]} \bigwedge_{j\in[1,k]|\ i\neq j} \neg\left((p_i \ll p_j) \wedge (p_j \ll p_i) \right)$

$\bigwedge_{i\in[1,k]} \neg(p_i \ll p_i)$

Figure 39 : Codage MK99 par liens causaux, protection d'intervalles et ordre partiel

Le codage qui en résulte pour notre problème est donné dans la Figure 40. La Figure 41 montre les modèles trouvés avec les deux codages de la partie commune.

6. $\left(Needs(p_1, a) \Rightarrow I \xrightarrow{a} p_1 \vee p_2 \xrightarrow{a} p_1 \right) \wedge \left(Needs(p_1, b) \Rightarrow I \xrightarrow{b} p_1 \vee p_2 \xrightarrow{b} p_1 \right) \wedge$
$\left(Needs(p_1, c) \Rightarrow I \xrightarrow{c} p_1 \vee p_2 \xrightarrow{c} p_1 \right) \wedge \left(Needs(p_1, d) \Rightarrow I \xrightarrow{d} p_1 \vee p_2 \xrightarrow{d} p_1 \right) \wedge$
$\left(Needs(p_1, e) \Rightarrow I \xrightarrow{e} p_1 \vee p_2 \xrightarrow{e} p_1 \right) \wedge \left(Needs(p_2, a) \Rightarrow I \xrightarrow{a} p_2 \vee p_1 \xrightarrow{a} p_2 \right) \wedge$
$\left(Needs(p_2, b) \Rightarrow I \xrightarrow{b} p_2 \vee p_1 \xrightarrow{b} p_2 \right) \wedge \left(Needs(p_2, c) \Rightarrow I \xrightarrow{c} p_2 \vee p_1 \xrightarrow{c} p_2 \right) \wedge$
$\left(Needs(p_2, d) \Rightarrow I \xrightarrow{d} p_2 \vee p_1 \xrightarrow{d} p_2 \right) \wedge \left(Needs(p_2, e) \Rightarrow I \xrightarrow{e} p_2 \vee p_1 \xrightarrow{e} p_2 \right)$

7. $\left(p_1 \xrightarrow{a} p_2 \Rightarrow Adds(p_1, a) \wedge Needs(p_2, a) \wedge (p_1 \ll p_2) \right) \wedge \left(p_2 \xrightarrow{a} p_1 \Rightarrow Adds(p_2, a) \wedge Needs(p_1, a) \wedge (p_2 \ll p_1) \right) \wedge$
$\left(p_1 \xrightarrow{b} p_2 \Rightarrow Adds(p_1, b) \wedge Needs(p_2, b) \wedge (p_1 \ll p_2) \right) \wedge \left(p_2 \xrightarrow{b} p_1 \Rightarrow Adds(p_2, b) \wedge Needs(p_1, b) \wedge (p_2 \ll p_1) \right) \wedge$
$\left(p_1 \xrightarrow{c} p_2 \Rightarrow Adds(p_1, c) \wedge Needs(p_2, c) \wedge (p_1 \ll p_2) \right) \wedge \left(p_2 \xrightarrow{c} p_1 \Rightarrow Adds(p_2, c) \wedge Needs(p_1, c) \wedge (p_2 \ll p_1) \right) \wedge$
$\left(p_1 \xrightarrow{d} p_2 \Rightarrow Adds(p_1, d) \wedge Needs(p_2, d) \wedge (p_1 \ll p_2) \right) \wedge \left(p_2 \xrightarrow{d} p_1 \Rightarrow Adds(p_2, d) \wedge Needs(p_1, d) \wedge (p_2 \ll p_1) \right) \wedge$
$\left(p_1 \xrightarrow{e} p_2 \Rightarrow Adds(p_1, e) \wedge Needs(p_2, e) \wedge (p_1 \ll p_2) \right) \wedge \left(p_2 \xrightarrow{e} p_1 \Rightarrow Adds(p_2, e) \wedge Needs(p_1, e) \wedge (p_2 \ll p_1) \right)$

8. –

9. $\neg\left((p_1 \ll p_2) \wedge (p_2 \ll p_1) \right) \wedge\ \neg\left((p_2 \ll p_1) \wedge (p_1 \ll p_2) \right) \wedge\ \neg(p_1 \ll p_1) \wedge\ \neg(p_2 \ll p_2)$

Clauses : 43 **Propositions nouvelles :** 22

Figure 40 : Codage MK99 par liens causaux, protection d'intervalles et ordre partiel du problème Π

Modèles (codage [MK99]):
Tous les modèles :

$$\left\{ X \cup Y \cup Z \;\middle|\; \begin{array}{r} X \in \left\{ \{p_1 = \Phi\}, \{p_1 = A\}, \{p_1 = B\}, \{p_1 = C\} \right\} \\ Y \in \left\{ \{p_2 = \Phi\}, \{p_2 = A\}, \{p_2 = B\}, \{p_2 = C\} \right\} \\ Z \in \left\{ \{(p_1 \ll p_2)\}, \{(p_2 \ll p_1)\} \right\} \end{array} \right\}$$

Modèles corrects :
$\{p_1 = B, p_2 = C, (p_1 \ll p_2)\}$ $\{p_1 = C, p_2 = B, (p_2 \ll p_1)\}$

Modèles (codage modifié):
Tous les modèles :

$$\left\{ X \cup Y \cup Z \;\middle|\; \begin{array}{r} X \in \left\{ \{\}, \{A \in p_1\}, \{B \in p_1\}, \{C \in p_1\}, \{A \in p_1, C \in p_1\} \right\} \\ Y \in \left\{ \{\}, \{A \in p_2\}, \{B \in p_2\}, \{C \in p_2\}, \{A \in p_2, C \in p_2\} \right\} \\ Z \in \left\{ \{(p_1 \ll p_2)\}, \{(p_2 \ll p_1)\} \right\} \end{array} \right\}$$

Modèles corrects :
$\{B \in p_1, C \in p_2, (p_1 \ll p_2)\}$ $\{B \in p_1, A \in p_2, C \in p_2, (p_1 \ll p_2)\}$
$\{C \in p_1, B \in p_2, (p_2 \ll p_1)\}$ $\{A \in p_1, C \in p_1, B \in p_2, (p_2 \ll p_1)\}$

Figure 41 : Modèles trouvés pour le codage MK99 par liens causaux, protection d'intervalles et ordre partiel du problème $\prod$

Seuls 2 modèles parmi les 32 modèles trouvés avec le codage MK99 de la partie commune et 4 modèles parmi les 50 modèles trouvés avec le codage modifié de la partie commune correspondent à des plans valides. Les autres modèles correspondent soit à des plans valides mais qui ne résolvent pas le but, soit à des plans qui ne sont pas valides car toutes les préconditions de toutes les actions ne sont pas établies par l'état initial ou par une action antérieure.

3.3.2. Problèmes et modifications

1. **Pas de prise en compte du but :** certains plans extraits des modèles sont valides, mais ne résolvent pas le problème (par exemple les modèles $\{p_1 = B, p_2 = A, (p_1 \ll p_2)\}$, $\{A \in p_1, A \in p_2, (p_2 \ll p_1)\}$). En effet, d'après la règle 6 (établissement des préconditions), les préconditions des étapes 1 à *k* doivent être supportées par un lien causal avec une étape antérieure ou avec l'état initial. Mais cette règle ne précise pas que les préconditions de l'étape *G* doivent elles aussi être supportées par un lien causal avec une étape du plan ou avec l'état initial. Il faut donc le rajouter à la règle 6. Il faut aussi modifier la règle 7 qui explicite les liens causaux : la deuxième étape du lien causal peut être le but. Enfin, il faut compléter la règle 8 qui permet de protéger un lien causal : s'il y a un lien causal entre une étape p_i et le but et qu'une étape p_j menace ce lien, alors l'étape p_j doit obligatoirement précéder l'étape p_i (elle ne peut pas suivre l'étape du but, puisque cette dernière est l'étape finale du plan).
2. **Prise en compte incomplète de l'état initial :** l'état initial est pris en compte dans la règle 6 (ce qui n'était pas le cas pour le but, comme on vient de le voir) mais pas dans la règle 7. On trouve donc des modèles dont les préconditions des actions n'ont été établies ni dans l'état initial, ni par une autre action (par exemple les modèles $\{p_1 = C, p_2 = A, (p_1 \ll p_2)\}$ et $\{C \in p_2, (p_2 \ll p_1)\}$). En effet, s'il y a un lien causal entre l'état initial et une étape, il faut préciser la signification de ce lien dans la règle 7 : l'étape en question a pour précondition le fluent supporté par le lien causal.
3. **Plans incorrects :** certains modèles ne correspondent pas à des plans valides car une action retire une précondition d'une action qui la suit (par exemple les modèles $\{p_1 = A, p_2 = B, (p_1 \ll p_2)\}$ et $\{B \in p_1, A \in p_2, (p_2 \ll p_1)\}$). En effet, de la même façon qu'il faut protéger les liens causaux mettant en question l'étape du but pour la règle 8, il faut compléter cette dernière par la protection des liens causaux mettant en jeu l'état initial : s'il y a un lien causal entre l'état initial et une étape et qu'une autre étape menace ce lien, alors cette dernière étape doit suivre la deuxième étape du lien causal.

4. **Clauses inutiles :** de même que pour la partie commune du codage, des propositions et des clauses inutiles apparaissent : par exemple, $Needs(p_1, b)$ apparaît alors que *b* n'est précondition d'aucune action. On peut donc supprimer cette proposition, ainsi que la clause (issue de la règle 6) dans laquelle elle intervient. D'autres propositions sont inutiles, par exemple $I \xrightarrow{d} p_1$: d ne fait pas partie de l'état initial. Mais il ne faut pas supprimer la clause de la règle 6 dans laquelle elle apparaît, puisque *d* peut être ajouté par l'action *B*, donc par une étape du plan. On peut par contre supprimer la proposition de la clause de la règle 6. De même, la proposition $p_1 \xrightarrow{c} p_2$ qui apparaît dans une clause issue de la règle 7 est inutile : aucune action n'a *c* pour effet, donc aucune étape (à part l'état initial) ne peut l'ajouter. On peut donc supprimer la clause de la règle 7 dans laquelle cette proposition intervient, et on peut supprimer cette proposition de la clause de la règle 6 dans laquelle elle apparaît. Pour régler ce problème de propositions et de clauses inutiles, nous employons la même solution que pour la partie commune : le partage de l'ensemble des fluents en trois sous-ensembles (préconditions, ajouts, retraits).
   La règle 9 génère aussi des clauses équivalentes, voire même des clauses inutiles : la 3ème partie de cette règle ne sert à rien puisqu'on s'assure toujours que la relation de précédence porte sur des étapes différentes. Nous supprimons donc cette partie de la règle 9, et nous modifions les domaines des indices de la deuxième partie de la règle 9 en utilisant des inégalités plutôt que des différences.

La Figure 42 montre les règles modifiées du codage par liens causaux, protection d'intervalles et ordre partiel, et la Figure 43 le codage modifié de notre problème. On trouve maintenant les modèles corrects indiqués dans la Figure 41.

6a. $$\bigwedge_{i\in[1,k]} \bigwedge_{f\in Fp} \left(Needs(p_i, f) \Rightarrow \left[f\in Fa \hookrightarrow \bigvee_{j\in[1,k]\,|\, j\neq i} p_j \xrightarrow{f} p_i \,\middle|\, \bot \right] \vee \left[f\in I \hookrightarrow (I \xrightarrow{f} p_i) \,\middle|\, \bot \right] \right)$$
$$\bigwedge_{f\in G} \left(Needs(G, f) \Rightarrow \left[f\in Fa \hookrightarrow \bigvee_{i\in[1,k]} p_i \xrightarrow{f} G \,\middle|\, \bot \right] \vee \left[f\in I \hookrightarrow (I \xrightarrow{f} G) \,\middle|\, \bot \right] \right)$$

7a. $$\bigwedge_{i\in[1,k]} \bigwedge_{j\in[1,k]\,|\, i\neq j} \bigwedge_{f\in(Fn\cap Fa)} \left(p_i \xrightarrow{f} p_j \Rightarrow Adds(p_i, f) \wedge Needs(p_j, f) \wedge (p_i \ll p_j) \right)$$
$$\bigwedge_{i\in[1,k]} \bigwedge_{f\in(G\cap Fa)} \left(p_i \xrightarrow{f} G \Rightarrow Adds(p_i, f) \right)$$
$$\bigwedge_{i\in[1,k]} \bigwedge_{f\in(Fp\cap I)} \left(I \xrightarrow{f} p_i \Rightarrow Needs(p_i, f) \right)$$

8a. $$\bigwedge_{i\in[1,k]} \bigwedge_{j\in[1,k]\,|\, i\neq j} \bigwedge_{q\in[1,k]\,|\, q\neq i\wedge q\neq j} \bigwedge_{f\in(Fp\cap Fa\cap Fd)} \left(p_i \xrightarrow{f} p_j \wedge Dels(p_q, f) \Rightarrow (p_q \ll p_i) \vee (p_j \ll p_q) \right)$$
$$\bigwedge_{i\in[1,k]} \bigwedge_{j\in[1,k]\,|\, i\neq j} \bigwedge_{f\in(G\cap Fa\cap Fd)} \left(p_i \xrightarrow{f} G \wedge Dels(p_j, f) \Rightarrow (p_j \ll p_i) \right)$$
$$\bigwedge_{i\in[1,k]} \bigwedge_{j\in[1,k]\,|\, i\neq j} \bigwedge_{f\in(Fp\cap I\cap Fd)} \left(I \xrightarrow{f} p_i \wedge Dels(p_j, f) \Rightarrow (p_i \ll p_j) \right)$$
$$\bigwedge_{i\in[1,k]} \bigwedge_{f\in(G\cap I\cap Fd)} \left(I \xrightarrow{f} G \wedge Dels(p_i, f) \Rightarrow \bot \right)$$

9a. $$\bigwedge_{i\in[1,k]} \bigwedge_{j\in[1,k]\,|\, j\neq i} \bigwedge_{s\in[1,k]\,|\, s\neq i\wedge s\neq j} \left((p_i \ll p_j) \wedge (p_j \ll p_s) \Rightarrow (p_i \ll p_s) \right)$$
$$\bigwedge_{i\in[1,k]} \bigwedge_{j\in[i+1,k]} \neg \left((p_i \ll p_j) \wedge (p_j \ll p_i) \right)$$

Figure 42 : Codage modifié par liens causaux, protection d'intervalles et ordre partiel

6a. $\left(Needs(p_1,a)\Rightarrow I\xrightarrow{a}p_1\right)\wedge\left(Needs(p_1,c)\Rightarrow I\xrightarrow{c}p_1\right)\wedge\left(Needs(p_1,d)\Rightarrow p_2\xrightarrow{d}p_1\right)\wedge\left(Needs(p_2,a)\Rightarrow I\xrightarrow{a}p_2\right)\wedge$
$\left(Needs(p_2,c)\Rightarrow I\xrightarrow{c}p_2\right)\wedge\left(Needs(p_2,d)\Rightarrow p_1\xrightarrow{d}p_2\right)\wedge\left(Needs(G,e)\Rightarrow p_1\xrightarrow{e}G\vee p_2\xrightarrow{e}G\right)$

7a. $\left(p_1\xrightarrow{d}p_2\Rightarrow Adds(p_1,d)\wedge Needs(p_2,d)\wedge(p_1\ll p_2)\right)\wedge\left(p_2\xrightarrow{d}p_1\Rightarrow Adds(p_2,d)\wedge Needs(p_1,d)\wedge(p_2\ll p_1)\right)\wedge$
$\left(I\xrightarrow{a}p_1\Rightarrow Needs(p_1,a)\right)\wedge\left(I\xrightarrow{a}p_2\Rightarrow Needs(p_2,a)\right)\wedge\left(I\xrightarrow{c}p_1\Rightarrow Needs(p_1,c)\right)\wedge\left(I\xrightarrow{c}p_2\Rightarrow Needs(p_2,c)\right)\wedge$
$\left(p_1\xrightarrow{e}G\Rightarrow Adds(p_1,e)\right)\wedge\left(p_2\xrightarrow{e}G\Rightarrow Adds(p_2,e)\right)$

8a. $\left(I\xrightarrow{c}p_1\wedge Dels(p_2,c)\Rightarrow(p_1\ll p_2)\right)\wedge\left(I\xrightarrow{c}p_2\wedge Dels(p_1,c)\Rightarrow(p_2\ll p_1)\right)$

9a. $\neg\left((p_1\ll p_2)\wedge(p_2\ll p_1)\right)$

Clauses : 32 **Propositions nouvelles :** 10

Figure 43 : Codage modifié par liens causaux, protection d'intervalles et ordre partiel du problème Π

3.4. Codage par liens causaux, protection d'intervalles et étapes contiguës

On a vu que dans le codage précédent, plusieurs modèles correspondent au même plan-solution, suivant l'ordre partiel entre les étapes. Le but qui nous préoccupe maintenant est de simplifier le codage précédent, en contraignant les étapes à suivre un ordre prédéfini. Nous considérons maintenant que les étapes sont ordonnées suivant l'ordre croissant de leur indice, et nous n'aurons donc plus besoin de la relation de précédence. La protection des intervalles se fait non plus en choisissant un ordre partiel sur les étapes par promotion ou demotion, puisque les étapes sont déjà ordonnées, mais en interdisant qu'une étape comprise dans un intervalle défini par un lien causal menace ce dernier.

3.4.1. Codage MK99

Il faut rajouter les règles de la Figure 44 à la partie commune du codage dans les espaces de plans :

6. **Établissement des préconditions :** chaque fluent est supporté par un lien causal entre l'étape qui le produit (qui peut être l'état initial) et l'étape qui l'utilise en tant que précondition. On peut remarquer que l'on ne se soucie pas de savoir s'il existe réellement une action qui produit ce fluent et une action qui l'utilise.
7. **Liens causaux et protection d'intervalles :** la présence d'un lien causal supportant un fluent *f* entre une étape p_i et une étape p_j implique que l'étape p_i ajoute *f* et que l'étape p_j a pour précondition *f*. De plus, une étape qui retire *f* ne peut pas s'insérer entre p_i et p_j.

6. $\bigwedge_{i\in[1,k]}\ \bigwedge_{f\in F}\left(Needs(p_i,f)\ \Rightarrow\ \left(\bigvee_{j\in[1,i-1]}\left(p_j\xrightarrow{f}p_i\right)\right)\vee I\xrightarrow{f}p_i\right)$

7. $\bigwedge_{i\in[1,k-1]}\ \bigwedge_{j\in[i+1,k]}\ \bigwedge_{f\in F}\left(p_i\xrightarrow{f}p_j\ \Rightarrow\ Adds(p_i,f)\wedge Needs(p_j,f)\wedge\left(\bigwedge_{q\in[i+1,j-1]}\neg Dels(p_q,f)\right)\right)$

Figure 44 : Codage MK99 par liens causaux, protection d'intervalles et étapes contiguës

Le codage qui en résulte pour notre problème est donné dans la Figure 45. La Figure 46 montre les modèles trouvés avec les deux codages de la partie commune.

Seul 1 modèle parmi les 16 modèles trouvés avec le codage MK99 de la partie commune et 2 modèles parmi les 25 modèles trouvés avec le codage modifié de la partie commune correspondent à des plans valides. Les autres modèles correspondent soit à des plans valides mais qui ne résolvent pas le but, soit à des plans qui ne sont pas valides car toutes les préconditions de toutes les actions ne sont pas établies par l'état initial ou par une action antérieure.

6. $\left(Needs(p_1,a)\Rightarrow I\xrightarrow{a}p_1\right)\wedge\left(Needs(p_1,b)\Rightarrow I\xrightarrow{b}p_1\right)\wedge\left(Needs(p_1,c)\Rightarrow I\xrightarrow{c}p_1\right)\wedge$
$\left(Needs(p_1,d)\Rightarrow I\xrightarrow{d}p_1\right)\wedge\left(Needs(p_1,e)\Rightarrow I\xrightarrow{e}p_1\right)\wedge\left(Needs(p_2,a)\Rightarrow\left(I\xrightarrow{a}p_2\vee p_1\xrightarrow{a}p_2\right)\right)\wedge$
$\left(Needs(p_2,b)\Rightarrow I\xrightarrow{b}p_2\vee p_1\xrightarrow{b}p_2\right)\wedge\left(Needs(p_2,c)\Rightarrow I\xrightarrow{c}p_2\vee p_1\xrightarrow{c}p_2\right)\wedge$
$\left(Needs(p_2,d)\Rightarrow I\xrightarrow{d}p_2\vee p_1\xrightarrow{d}p_2\right)\wedge\left(Needs(p_2,e)\Rightarrow I\xrightarrow{e}p_2\vee p_1\xrightarrow{e}p_2\right)$

7. $\left(p_1\xrightarrow{a}p_2\Rightarrow Adds(p_1,a)\wedge Needs(p_2,a)\right)\wedge\left(p_1\xrightarrow{b}p_2\Rightarrow Adds(p_1,b)\wedge Needs(p_2,b)\right)\wedge$
$\left(p_1\xrightarrow{c}p_2\Rightarrow Adds(p_1,c)\wedge Needs(p_2,c)\right)\wedge\left(p_1\xrightarrow{d}p_2\Rightarrow Adds(p_1,d)\wedge Needs(p_2,d)\right)\wedge$
$\left(p_1\xrightarrow{e}p_2\Rightarrow Adds(p_1,e)\wedge Needs(p_2,e)\right)$

Clauses : 20 **Propositions nouvelles :** 15

Figure 45 : Codage MK99 par liens causaux, protection d'intervalles et étapes contiguës du problème Π

Modèles (codage [MK99]):
Tous les modèles :
$$\left\{X\cup Y\mid \begin{array}{l} X\in\left\{\{p_1=\Phi\},\{p_1=A\},\{p_1=B\},\{p_1=C\}\right\}\\ Y\in\left\{\{p_2=\Phi\},\{p_2=A\},\{p_2=B\},\{p_2=C\}\right\}\end{array}\right\}$$
Modèles corrects :
$\{p_1=B, p_2=C\}$

Modèles (codage modifié):
Tous les modèles :
$$\left\{X\cup Y\mid \begin{array}{l} X\in\left\{\{\},\{A\in p_1\},\{B\in p_1\},\{C\in p_1\},\{A\in p_1, C\in p_1\}\right\}\\ Y\in\left\{\{\},\{A\in p_2\},\{B\in p_2\},\{C\in p_2\},\{A\in p_2, C\in p_2\}\right\}\end{array}\right\}$$
Modèles corrects :
$\{B\in p_1, C\in p_2\}$ $\{B\in p_1, A\in p_2, C\in p_2\}$

Figure 46 : Modèles trouvés pour le codage MK99 par liens causaux, protection d'intervalles et étapes contiguës du problème Π

3.4.2. Problèmes et modifications

1. **Pas de prise en compte du but :** certains plans extraits des modèles sont valides, mais ne résolvent pas le problème (par exemple les modèles $\{p_1=B, p_2=A\}$, $\{B\in p_1, B\in p_2\}$). En effet, d'après la règle 6 (établissement des préconditions), les préconditions des étapes 1 à *k* doivent être supportées par un lien causal avec une étape antérieure ou avec l'état initial. Mais cette règle ne précise pas que les préconditions de l'étape *G* doivent elles aussi être supportées par un lien causal avec une étape du plan ou avec l'état initial. Il faut donc le rajouter à la règle 6. Il faut aussi modifier la règle 7 qui explicite les liens causaux et permet la protection des intervalles : une des deux étapes du lien causal peut être le but. La destruction du fluent supporté par le lien causal doit donc être interdite.
2. **Prise en compte incomplète de l'état initial :** l'état initial est pris en compte dans la règle 6 (ce qui n'était pas le cas pour le but, comme on vient de le voir) mais pas dans la règle 7. On trouve donc des modèles pour lesquels les préconditions des actions n'ont été établies ni dans l'état initial, ni par une autre action (par exemple les modèles $\{p_1=C, p_2=A\}$ et $\{C\in p_2\}$). En effet, s'il y a un lien causal entre l'état initial et une étape, il faut préciser la signification de ce lien dans la règle 7 : l'étape en question a pour précondition le fluent supporté par le lien causal.

3. **Plans incorrects :** certains modèles ne correspondent pas à des plans valides car une action retire une précondition d'une action qui la suit (par exemple les modèles $\{p_1 = A, p_2 = B\}$ et $\{A \in p_1, B \in p_2\}$). En effet, de la même façon qu'il faut protéger les liens causaux mettant en question l'étape du but pour la règle 7, il faut compléter cette dernière pour la protection des liens causaux mettant en jeu l'état initial : s'il y a un lien causal entre l'état initial et une étape, il faut interdire la destruction du fluent supporté par le lien causal. De même, s'il y a un lien causal entre l'état initial et l'étape du but, il faut interdire la destruction du fluent supporté par le lien causal.
4. **Clauses inutiles :** de même que pour la partie commune du codage, apparaissent des propositions et des clauses inutiles : par exemple, $Needs(p_1, b)$ apparaît alors que *b* n'est précondition d'aucune action. On peut donc supprimer cette proposition, ainsi que la clause (issue de la règle 6) dans laquelle elle intervient. D'autres propositions sont inutiles, par exemple $I \xrightarrow{d} p_1$: *d* ne fait pas partie de l'état initial. Mais il ne faut pas supprimer la clause de la règle 6 dans laquelle elle apparaît, puisque *d* peut être ajouté par une action. On peut par contre supprimer cette proposition de la clause dans laquelle elle apparaît. De même, la proposition $p_1 \xrightarrow{c} p_2$ qui apparaît dans une clause issue de la règle 7 est inutile : aucune action n'a *c* pour effet, donc aucune étape (à part l'état initial) ne peut l'ajouter. On peut donc supprimer la clause de la règle 7 dans laquelle cette proposition intervient, et on peut supprimer cette proposition de la clause de la règle 6 dans laquelle elle apparaît. Nous allons donc employer la même solution que pour la partie commune : le partage de l'ensemble des fluents en trois sous-ensembles (préconditions, ajouts, retraits).

La Figure 47 montre les règles modifiées du codage par liens causaux, protection d'intervalles et étapes contiguës, et la Figure 48 le codage modifié de notre problème. On trouve maintenant les modèles corrects indiqués dans la Figure 46.

6a. $$\bigwedge_{i\in[1,k]} \bigwedge_{f\in Fp} \left(Needs(p_i, f) \Rightarrow \left[f \in Fa \hookrightarrow \bigvee_{j\in[1,i-1]} p_j \xrightarrow{f} p_i \,\middle|\, \bot \right] \vee \left[f \in I \hookrightarrow (I \xrightarrow{f} p_i) \,\middle|\, \bot \right] \right)$$
$$\bigwedge_{f\in G} \left(Needs(G, f) \Rightarrow \left[f \in Fa \hookrightarrow \bigvee_{i\in[1,k]} p_i \xrightarrow{f} G \,\middle|\, \bot \right] \vee \left[f \in I \hookrightarrow (I \xrightarrow{f} G) \,\middle|\, \bot \right] \right)$$

7a. $$\bigwedge_{i\in[1,k-1]} \bigwedge_{j\in[i+1,k]} \bigwedge_{f\in(Fp\cap Fa)} \left(p_i \xrightarrow{f} p_j \Rightarrow \begin{array}{l} Adds(p_i, f) \wedge Needs(p_j, f) \\ \wedge \left[f \in Fd \hookrightarrow \bigwedge_{q\in[i+1,j-1]} \neg Dels(p_q, f) \,\middle|\, \top \right] \end{array} \right)$$
$$\bigwedge_{i\in[1,k]} \bigwedge_{f\in(Fp\cap I)} \left(I \xrightarrow{f} p_i \Rightarrow Needs(p_i, f) \wedge \left[f \in Fd \hookrightarrow \bigwedge_{j\in[1,i-1]} \neg Dels(p_j, f) \,\middle|\, \top \right] \right)$$
$$\bigwedge_{i\in[1,k]} \bigwedge_{f\in(Fa\cap G)} \left(p_i \xrightarrow{f} G \Rightarrow Adds(p_i, f) \wedge \left[f \in Fd \hookrightarrow \bigwedge_{j\in[i+1,k]} \neg Dels(p_j, f) \,\middle|\, \top \right] \right)$$
$$\bigwedge_{f\in(I\cap G)} \left(I \xrightarrow{f} G \Rightarrow \left[f \in Fd \hookrightarrow \bigwedge_{i\in[1,k]} \neg Dels(p_i, f) \,\middle|\, \top \right] \right)$$

Figure 47 : Codage modifié par liens causaux, protection d'intervalles et étapes contiguës

6a. $\left(Needs(p_1,a)\Rightarrow I\xrightarrow{a}p_1\right)\wedge\left(Needs(p_1,c)\Rightarrow I\xrightarrow{c}p_1\right)\wedge\left(Needs(p_1,d)\Rightarrow\bot\right)\wedge$
$\left(Needs(p_2,a)\Rightarrow I\xrightarrow{a}p_2\right)\wedge\left(Needs(p_2,c)\Rightarrow I\xrightarrow{c}p_2\right)\wedge\left(Needs(p_2,d)\Rightarrow p_1\xrightarrow{d}p_2\right)\wedge$
$\left(Needs(G,e)\Rightarrow p_1\xrightarrow{e}G\vee p_2\xrightarrow{e}G\right)$

7a. $\left(p_1\xrightarrow{d}p_2\Rightarrow Adds(p_1,d)\wedge Needs(p_2,d)\right)\wedge\left(I\xrightarrow{a}p_1\Rightarrow Needs(p_1,a)\right)\wedge$
$\left(I\xrightarrow{c}p_1\Rightarrow Needs(p_1,c)\right)\wedge\left(I\xrightarrow{a}p_2\Rightarrow Needs(p_2,a)\right)\wedge\left(I\xrightarrow{c}p_2\Rightarrow Needs(p_2,c)\right)\wedge$
$\left(p_1\xrightarrow{e}G\Rightarrow Adds(p_1,e)\right)\wedge\left(p_2\xrightarrow{e}G\Rightarrow Adds(p_2,e)\right)$

Clauses : 15 **Propositions nouvelles :** 7

Figure 48 : Codage modifié par liens causaux, protection d'intervalles et étapes contiguës du problème Π

3.5. Codage du "chevalier blanc"

Ce codage n'utilise plus de liens causaux, comme ceux que l'on vient de voir. Ceci permet évidemment de réduire le nombre de variables, puisqu'on ne va pas en utiliser d'autres que celles présentes dans la partie commune du codage dans les espaces d'états. Les liens de causalité entre les actions seront exprimés "directement" : si une étape a besoin d'un fluent, c'est qu'une étape qui la précède l'a créé. La protection des intervalles va se faire par la technique du chevalier blanc ou "white-knight" introduit par Chapman dans son planificateur TWEAK [Chapman 1987].

3.5.1. Codage MK99

Il faut rajouter les règles suivantes (cf. Figure 49) à la partie commune du codage dans les espaces de plans :

6. **Établissement des préconditions :** la précondition d'une étape d'un niveau *i* doit être ajoutée par une étape qui la précède, c'est-à-dire soit par une étape d'un niveau *j* tel que $1 \le j < i$, soit par l'étape *I*.
7. **Chevalier blanc :** si une première étape (ou l'étape du but *G*) a pour précondition le fluent *f* au niveau *i*, et qu'une deuxième étape retire ce fluent avant que la première étape puisse l'utiliser, c'est-à-dire à un niveau *j* tel que $1 \le j \le i-2$, alors il doit y avoir une troisième étape qui rétablit *f* à un niveau *q* tel que $j < q < i$.

6. $\bigwedge_{i\in[2,k]}\ \bigwedge_{f\in F}\left(Needs(p_i,f)\ \Rightarrow\ \left(\bigvee_{j\in[1,i-1]}Adds(p_j,f)\right)\vee Adds(I,f)\right)$

7. $\bigwedge_{i\in[3,k]}\ \bigwedge_{j\in[1,i-2]}\ \bigwedge_{f\in F}\left(Needs(p_i,f)\wedge Dels(p_j,f)\ \Rightarrow\ \bigvee_{q\in[j+1,i-1]}Adds(p_q,f)\right)$
$\bigwedge_{i\in[1,k-1]}\ \bigwedge_{f\in G}\left(Needs(G,f)\wedge Dels(p_i,f)\ \Rightarrow\ \bigvee_{j\in[i+1,k]}Adds(p_j,f)\right)$

Figure 49 : Codage MK99 du chevalier blanc

Le codage qui en résulte pour notre problème est donné dans la Figure 50. La Figure 51 montre les modèles trouvés avec les deux façons de coder la partie commune.

Plusieurs problèmes apparaissent : sur les 13 modèles trouvés en utilisant le codage MK99 de la partie commune, seul le modèle 13 correspond à un plan valide qui donne le bon résultat. Les plans 1 à 8 correspondent à des plans valides, mais ne mènent pas au bon résultat. Les modèles 9 à 12 ne correspondent pas à des plans valides. De même, sur les 17 modèles trouvés en utilisant le codage modifié, seuls les modèles 16 et 17 sont corrects. Les plans 1 à 8 correspondent à des plans valides, mais ne mènent pas au bon résultat. Les modèles 9 à 15 ne correspondent pas à des plans valides. Nous allons maintenant étudier les problèmes posés par le codage du chevalier blanc, et proposer une solution pour y remédier.

6. $\Big(Needs(p_2, a) \Rightarrow Adds(I, a) \vee Adds(p_1, a)\Big) \wedge \Big(Needs(p_2, b) \Rightarrow Adds(I, b) \vee Adds(p_1, b)\Big) \wedge$
$\Big(Needs(p_2, c) \Rightarrow Adds(I, c) \vee Adds(p_1, c)\Big) \wedge \Big(Needs(p_2, d) \Rightarrow Adds(I, d) \vee Adds(p_1, d)\Big) \wedge$
$\Big(Needs(p_2, e) \Rightarrow Adds(I, e) \vee Adds(p_1, e)\Big)$

7. $\Big(Needs(G, e) \wedge Dels(p_1, e) \Rightarrow Adds(p_2, e)\Big)$

Clauses : 6 **Propositions nouvelles :** 0

Figure 50 : Codage MK99 du chevalier blanc pour le problème $\prod$

Modèles (codage [MK99]):		**Modèles (codage modifié):**	
1. $\{p_1=\Phi, p_2=\Phi\}$	8. $\{p_1=B, p_2=B\}$	1. $\{\}$ (donc $p_1=p_2=\varnothing$)	10. $\{C\in p_1\}$ (donc $p_2=\varnothing$)
2. $\{p_1=A, p_2=\Phi\}$	9. $\{p_1=A, p_2=B\}$	2. $\{A\in p_1\}$ (donc $p_2=\varnothing$)	11. $\{C\in p_1, A\in p_1\}$ (donc $p_2=\varnothing$)
3. $\{p_1=\Phi, p_2=A\}$	10. $\{p_1=C, p_2=\Phi\}$	3. $\{A\in p_2\}$ (donc $p_1=\varnothing$)	12. $\{C\in p_1, A\in p_2\}$
4. $\{p_1=B, p_2=\Phi\}$	11. $\{p_1=C, p_2=A\}$	4. $\{B\in p_1\}$ (donc $p_2=\varnothing$)	13. $\{C\in p_1, B\in p_2\}$
5. $\{p_1=\Phi, p_2=B\}$	12. $\{p_1=C, p_2=B\}$	5. $\{B\in p_2\}$ (donc $p_1=\varnothing$)	14. $\{C\in p_1, A\in p_1, A\in p_2\}$
6. $\{p_1=A, p_2=A\}$	13. $\{p_1=B, p_2=C\}$	6. $\{A\in p_1, A\in p_2\}$	15. $\{C\in p_1, A\in p_1, B\in p_2\}$
7. $\{p_1=B, p_2=A\}$		7. $\{B\in p_1, A\in p_2\}$	16. $\{B\in p_1, C\in p_2\}$
		8. $\{B\in p_1, B\in p_2\}$	17. $\{B\in p_1, A\in p_2, C\in p_2\}$
		9. $\{A\in p_1, B\in p_2\}$	

Figure 51 : Modèles trouvés pour le codage MK99 du chevalier blanc du problème $\prod$

3.5.2. Problèmes et modifications

1. **Pas de prise en compte du but :** certains des plans extraits des modèles sont valides, mais ne résolvent pas le problème (les modèles 1 à 8 des deux codages). En effet, d'après la règle 6 du codage du chevalier blanc, les préconditions des étapes 2 à *k* doivent être établies par une action antérieure ou par l'état initial. Mais rien n'indique que les préconditions de l'étape *G* doivent elles aussi être établies par une étape du plan ou par l'état initial. Il faut donc le rajouter à la règle 6 (cf. Figure 52).
2. **Prise en compte incomplète de l'état initial :** les modèles 10 à 12 du codage MK99 de la partie commune et les modèles 10 à 15 du codage modifié de la partie commune sont incorrects car l'état initial n'est pas pris en compte pour les actions de l'étape 1 : l'indice *i* de la règle 6 ne doit pas démarrer à 2, mais à 1, sinon on ne peut pas garantir que les préconditions des actions de l'étape 1 sont établies.
3. **Plans incorrects :** les modèles 9 des deux codages et le modèle 15 du codage modifié de la partie commune ne correspondent pas à des plans valides : l'action *A* retire le fluent *c*, donc on ne peut pas appliquer l'action *B* à la suite de *A*. Ceci vient du fait que la règle du chevalier blanc (la règle 7) ne suffit pas : elle ne prend pas en compte le fait qu'une action peut avoir une de ses préconditions retirée par une action qui la précède. Pour remédier à ceci, il suffit de modifier les limites des indices de la règle 7 de façon à interdire le fait qu'une action d'une étape *i* retire une précondition d'une étape *i*+1 (sur la première partie de la règle 7) : l'indice *j* doit s'arrêter à *i*−1 et non à *i*−2. Ainsi, la disjonction à droite de l'implication sera égale à $\perp$. De même, pour la seconde partie de la règle 7 (qui concerne le but), il faut que l'indice *i* s'arrête à *k* et non pas *k*−1 : une action à l'étape *k* peut retirer un fluent du but. Enfin, l'indice *i* de la règle 7 ne doit pas commencer à 3 mais à 2, car une action de l'étape 1 peut retirer un fluent dont a besoin une action de l'étape 2.
4. **Clauses inutiles :** de même que pour la partie commune du codage, apparaissent des propositions et des clauses inutiles : par exemple, $Needs(p_1, b)$ apparaît alors que b n'est précondition d'aucune action. On peut donc supprimer cette proposition, ainsi que la clause (issue de la règle 6) dans laquelle elle intervient. De façon identique, la clause issue de la règle 7 est inutile, car le fluent *e* n'est retiré par aucune action. D'autres propositions sont inutiles, par exemple $Adds(I, d)$: d ne fait pas partie de l'état initial. Mais il ne faut pas supprimer la clause dans laquelle elle apparaît, puisque *d* peut être ajoutée par une action. De

même, $Adds(p_1, c)$ est inutile, mais comme *c* est dans l'état initial, il ne faut pas supprimer la clause entière. Nous allons donc employer la même solution que pour la partie commune : le partage de l'ensemble des fluents en trois sous-ensembles (préconditions, ajouts, retraits).

La Figure 52 montre le codage modifié du chevalier blanc, et la Figure 53 le codage modifié de notre problème. Cette fois-ci, on trouve les bons modèles : le modèle 13 pour le codage MK99 de la partie commune, et les modèles 16 et 17 pour le codage modifié de la partie commune.

6a. $\bigwedge_{i\in[1,k]} \bigwedge_{f\in Fp} \left(Needs(p_i, f) \Rightarrow \left[f\in Fa \hookrightarrow \bigvee_{j\in[1,i-1]} Adds(p_j, f) \,\middle|\, \bot \right] \vee \left[f\in I \hookrightarrow Adds(I, f) \,\middle|\, \bot \right] \right)$

$\bigwedge_{f\in G} \left(Needs(G, f) \Rightarrow \left[f\in Fa \hookrightarrow \bigvee_{i\in[1,k]} Adds(p_i, f) \,\middle|\, \bot \right] \vee \left[f\in I \hookrightarrow Adds(I, f) \,\middle|\, \bot \right] \right)$

7a. $\bigwedge_{i\in[2,k]} \bigwedge_{j\in[1,i-1]} \bigwedge_{f\in(Fp\cap Fd)} \left(Needs(p_i, f) \wedge Dels(p_j, f) \Rightarrow \left[f\in Fa \hookrightarrow \bigvee_{q\in[j+1,i-1]} Adds(p_q, f) \,\middle|\, \bot \right] \right)$

$\bigwedge_{i\in[1,k]} \bigwedge_{f\in(G\cap Fd)} \left(Needs(G, f) \wedge Dels(p_i, f) \Rightarrow \left[f\in Fa \hookrightarrow \bigvee_{j\in[i+1,k]} Adds(p_j, f) \,\middle|\, \bot \right] \right)$

Figure 52 : Codage du chevalier blanc modifié

6a. $\left(Needs(p_1, a) \Rightarrow Adds(I, a)\right) \wedge \left(Needs(p_1, c) \Rightarrow Adds(I, c)\right) \wedge \left(Needs(p_1, d) \Rightarrow \bot\right) \wedge$
$\left(Needs(p_2, a) \Rightarrow Adds(I, a)\right) \wedge \left(Needs(p_2, c) \Rightarrow Adds(I, c)\right) \wedge \left(Needs(p_2, d) \Rightarrow Adds(p_1, d)\right) \wedge$
$\left(Needs(G, e) \Rightarrow Adds(p_1, e) \vee Adds(p_2, e)\right)$

7a. $\left(Needs(p_2, c) \wedge Dels(p_1, c) \Rightarrow \bot\right)$

Clauses : 8 **Propositions nouvelles :** 0

Figure 53 : Codage du chevalier blanc modifié du problème Π

4. Conclusion et perspectives

Dans cette partie, nous avons présenté une analyse critique de [Mali et Kambhampati 1999]. Nous avons montré que les codages présentés dans cet article sont, pour la plupart, incorrects. Mais il n'en reste pas moins que malgré ces erreurs, ces différents codages présentent un intérêt certain.

Tout d'abord pour la forme dans laquelle ils sont présentés : bien que critiquable dans MK99 (nous avons amélioré certaines notations, en particulier en ce qui concerne les notations indicées), cette forme est bien plus claire et compréhensible que la notation de [Kautz, McAllester et Selman 1996].

Ensuite, cet article présente deux codages originaux : le codage par liens causaux, protection d'intervalles et étapes contiguës qui est une forme améliorée (et surtout simplifiée) du codage [Kautz, McAllester et Selman 1996], et le codage du chevalier blanc. Le fait que les auteurs aient su dégager une forme simplifiée du codage [Kautz, McAllester et Selman 1996] montre tout l'intérêt d'utiliser une notation efficace : on s'aperçoit immédiatement que le codage [Kautz, McAllester et Selman 1996] est inutilement complexe.

Après avoir corrigé ces codages, nous avons montré que l'on pouvait les simplifier : tout d'abord en évitant de construire des formules dont on sait d'avance qu'elles sont inutiles, mais surtout en introduisant le parallélisme dans les codages dans les espaces de plans. Ceci est extrêmement aisé : il suffit en effet de supprimer certaines clauses correspondant à des contraintes binaires entre les actions. Non seulement on supprime ainsi des clauses, mais en plus, dans les domaines où il est possible d'avoir des actions parallèles, le codage est développé sur beaucoup moins de niveaux. Par exemple dans le domaine Logistics, il est fréquent d'avoir des plans comprenant une cinquantaine d'actions, pour seulement une dizaine de niveaux. Ce qui signifie qu'introduire du parallélisme permet de diviser la taille du codage au moins par un facteur 5.

Nous nous interrogeons actuellement sur les liens existant entre ces différents codages (ainsi qu'avec d'autres codages existant dans la littérature), et nous pensons que la plupart d'entre eux (voire tous !) dérivent d'une forme commune de codage. Un codage qui paraît ainsi très intuitif est le suivant : si une étape de niveau

i a pour précondition un fluent *f*, alors soit une étape crée *f* au niveau *i*−1, soit une étape crée *f* au niveau *i*−2 et aucune étape ne le retire au niveau *i*−1, etc. Ceci peut s'exprimer par les formules suivantes (pour un codage développé jusqu'au niveau 4, sans tenir compte de l'état initial) :

$$\text{F1} = Needs(p_4, f) \Rightarrow Adds(p_3, f) \vee \Big(Adds(p_2, f) \wedge \neg Dels(p_3, f) \Big) \vee \Big(Adds(p_1, f) \wedge \neg Dels(p_2, f) \wedge \neg Dels(p_3, f) \Big)$$

$$\text{F2} = Adds(p_2, f) \wedge Dels(p_2, f) \Rightarrow \bot$$

$$\text{F3} = Adds(p_3, f) \wedge Dels(p_3, f) \Rightarrow \bot$$

On peut remarquer que la formule F1 n'est pas une clause, mais une DNF. Il va donc falloir la transformer en CNF afin de pouvoir utiliser un prouveur SAT classique :

$$\text{F1 } identical \left\{ \begin{array}{l} \text{c1} = \neg Needs(p_4, f) \vee Adds(p_3, f) \vee Adds(p_2, f) \vee Adds(p_1, f) \\ \text{c2} = \neg Needs(p_4, f) \vee Adds(p_3, f) \vee Adds(p_2, f) \vee \neg Dels(p_2, f) \quad \%\text{résolution possible avec F2}\% \\ \text{c3} = \neg Needs(p_4, f) \vee Adds(p_3, f) \vee Adds(p_2, f) \vee \neg Dels(p_3, f) \quad \%\text{subsumée par c6}\% \\ \text{c4} = \neg Needs(p_4, f) \vee Adds(p_3, f) \vee \neg Dels(p_3, f) \vee Adds(p_1, f) \quad \%\text{subsumée par c6}\% \\ \text{c5} = \neg Needs(p_4, f) \vee Adds(p_3, f) \vee \neg Dels(p_3, f) \vee \neg Dels(p_2, f) \quad \%\text{subsumée par c6}\% \\ \text{c6} = \neg Needs(p_4, f) \vee Adds(p_3, f) \vee \neg Dels(p_3, f) \quad \%\text{résolution possible avec F3}\% \end{array} \right\}$$

$$\text{F2} = Adds(p_2, f) \wedge Dels(p_2, f) \Rightarrow \bot$$

$$\text{F3} = Adds(p_3, f) \wedge Dels(p_3, f) \Rightarrow \bot$$

Finalement, on obtient les clauses suivantes...

$$Needs(p_4, f) \Rightarrow Adds(p_3, f) \vee Adds(p_2, f) \vee Adds(p_1, f)$$

$$Needs(p_4, f) \wedge Dels(p_2, f) \Rightarrow Adds(p_3, f)$$

$$Needs(p_4, f) \wedge Dels(p_3, f) \Rightarrow \bot$$

... et on se rend compte qu'il s'agit là du codage du chevalier blanc, qui ne serait alors que la forme clausale de notre codage "intuitif".

V. Planifier par l'utilisation de la procédure de Davis et Putnam sur les graphes de planification

1. Introduction

Nous allons maintenant nous intéresser à une méthode d'extraction de la solution qui permet de relier la planification et la satisfiabilité de bases de clauses. Cette méthode, baptisée DPPlan, a été présentée pour la première fois dans [Baioletti, Marcugini et Milani 2000]. DPPlan est un mécanisme d'extraction qui se substitue à celui de Graphplan, de la même façon qu'un prouveur SAT peut être utilisé pour trouver un plan-solution à partir d'un graphe de planification (comme le fait le système BLACKBOX [Kautz et Selman 1998b] [Kautz et Selman 1999]). Ce mécanisme est lui aussi basé sur la procédure de Davis et Putnam. Les avantages de DPPlan par rapport à un prouveur SAT sont les suivants :

- Il n'est pas nécessaire de transformer le graphe de planification en une base de clauses. Le graphe est exploité tel quel, en attachant à chaque noeud une valeur qui représente son état. Par exemple, un noeud d'action a la valeur *vrai* s'il fait partie du plan-solution courant, *faux* s'il ne peut pas en faire partie et *indéfini* si son état n'est pas encore spécifié. Le gain en place mémoire peut se révéler considérable : une base de clauses engendrée à partir d'un graphe de planification peut avoir une taille très importante.
- L'utilisation du graphe de planification pour l'extraction de la solution permet de conserver la structure du problème, et d'employer des heuristiques classiques en planification. En effet, la transformation d'un problème en une base de clauses fait perdre la structure de ce dernier, et les heuristiques employées par les prouveurs SAT sont des heuristiques qui n'ont a priori aucun lien (apparent ?) avec les heuristiques indépendantes du domaine que l'on peut utiliser en planification et qui ont démontré leur efficacité [Bonet et Geffner 1998] [Bonet et Geffner 1999] [Bonet et Geffner 2001] [Cayrol, Régnier et Vidal 2000b] [Kambhampati et Nigenda 2000]. DPPlan permet d'utiliser ces heuristiques, ce qui peut accélérer considérablement la phase d'extraction de la solution.
- Le mécanisme reconnu comme étant le plus efficace pour améliorer la procédure de Davis et Putnam [Davis et Putnam 1960] [Davis, Logemann et Loveland 1962], la propagation unitaire, fait partie intégrante de DPPlan. Elle permet en particulier, dans DPPlan, de propager des valeurs négatives (absence

d'un fluent, interdiction d'utiliser une action) ; ce qui n'existe pas dans Graphplan puisque le mécanisme de sélection des actions ne propage pas, au travers des exclusions mutuelles, le fait qu'une action est interdite pour un plan-solution.

- La procédure de recherche n'est plus unidirectionnelle vers l'arrière, niveau par niveau, comme dans Graphplan : elle est guidée par des heuristiques sur le choix des variables de branchement et se fait toujours vers l'arrière, mais n'est plus contrainte à respecter les niveaux successifs du graphe ; de plus, la propagation des valeurs se fait aussi bien vers l'arrière que vers l'avant.
- DPPlan offre, contrairement aux prouveurs SAT, un cadre tout à fait approprié à l'utilisation des relations d'autorisation. En effet, comme nous l'avons montré dans la partie consacrée à Graphplan, la vérification de la contrainte d'autorisation d'un ensemble d'actions doit être effectuée pendant l'extraction de la solution ; il va donc maintenant être possible de l'inclure dans DPPlan, puisque toutes les informations nécessaires sont présentes dans le graphe de planification.

2. La version originale de DPPlan

Nous présentons dans cette section la version originale de DPPlan, telle qu'elle est décrite dans [Baioletti, Marcugini et Milani 2000]. Cet algorithme comporte une erreur de conception qui le rend inutilisable tel quel, ce que nous démontrons à l'aide d'un contre-exemple très simple. Nous proposons ensuite une légère modification de l'algorithme qui permet de résoudre ce problème. Ceci nous amènera par la suite à analyser DPPlan en profondeur et à en proposer une version correcte, substantiellement simplifiée et améliorée.

2.1. Algorithme de DPPlan

A chaque noeud du graphe de planification, que ce soit un noeud de fluent ou un noeud d'action, est attachée une valeur qui représente son état. Si l'on raisonne en termes de satisfiabilité d'une base de clauses, un noeud du graphe correspond à un symbole propositionnel appartenant à un certain nombre de clauses ; cette valeur représente en quelque sorte sa valeur de vérité. La différence avec un problème de satisfiabilité classique, est qu'ici les noeuds de fluents peuvent avoir quatre valeurs possibles autres qu'*indéfini* : deux valeurs positives (*asserté, requis*) et deux valeurs négatives (*nié, requis−faux*). Les noeuds d'actions quant à eux, peuvent prendre une valeur positive (*vrai*) ou une valeur négative (*faux*). Pour simplifier, nous emploierons le terme fluent pour désigner à la fois un noeud de fluent et le fluent attaché à ce noeud ; de même que le terme action pour désigner les noeuds d'actions et les actions.

La valeur d'un fluent n_f, notée Valeur(n_f), est choisie parmi les valeurs suivantes :

- *indéfini* : le fluent est dans un état indéterminé. Sa valeur peut être modifiée à tout moment, sans provoquer une erreur de la recherche.
- *asserté* : le fluent fait partie de la liste des ajouts d'une action utilisée dans le plan-solution courant (cette action a la valeur *vrai*). Un fluent ayant cette valeur ne peut plus changer d'état.
- *requis* : le fluent fait partie de la liste des préconditions d'une action utilisée dans le plan-solution courant (cette action a la valeur *vrai*). L'algorithme devra alors trouver une action qui produit ce fluent ; la valeur de ce dernier passera alors à *asserté*.
- *nié* : le fluent fait partie de la liste des retraits d'une action utilisée dans le plan-solution courant (cette action a la valeur *vrai*). Un fluent ayant cette valeur ne peut plus changer d'état.
- *requis−faux* : le fluent doit passer à l'état *nié*, soit à cause d'une exclusion mutuelle avec un autre fluent, soit parce qu'il est prouvé qu'il ne peut plus prendre une autre valeur. Nous verrons plus loin que c'est cette valeur qui est à la base de l'erreur de conception de DPPlan.

Les actions, quant à elles, peuvent prendre l'une des valeurs suivantes :

- *indéfini* : l'action est dans un état indéterminé. Elle peut changer de valeur à tout moment, sans provoquer une erreur de la recherche.
- *vrai* : l'action fait partie du plan solution. Ses préconditions prennent la valeur *requis*, ses ajouts prennent la valeur *asserté*, et ses retraits prennent la valeur *nié*.

- *faux* : l'action ne peut pas faire partie du plan solution. Ses préconditions, ses ajouts et ses retraits changent de valeur en fonction de toutes les actions qui les ajoutent ou les retirent, suivant certaines règles de propagation.

Lorsqu'un noeud a une valeur positive (resp. négative), la tentative de lui associer une valeur négative (resp. positive) provoque un échec de la recherche. L'algorithme doit alors effectuer un retour arrière. Le changement de valeur d'un noeud provoque une propagation dans le reste du graphe, ce qui est à mettre en relation avec la propagation unitaire dans le cadre d'un prouveur SAT. En effet, la base de clauses obtenue par transformation d'un graphe de planification comporte essentiellement des clauses binaires ; l'affectation d'une valeur à une variable permet donc un grand nombre de propagations unitaires. Le même phénomène se reproduit dans DPPlan.

Nous allons maintenant décrire les principales fonctions de DPPlan. Nous utilisons dans cet algorithme un mécanisme de gestion d'échappements dynamiques, ce qui nous permet une description plus complète de l'algorithme que celle proposée dans [Baioletti, Marcugini et Milani 2000], et qui reste simple à écrire et facilement implémentable. Ce mécanisme est une version simplifiée des instructions "catch" et "throw" du langage Common Lisp [Steele 1989]. Il utilise les deux fonctions suivantes :

- **récupère**(<étiquette−échappement>, <instructions>) : cette fonction évalue les instructions, qui peuvent contenir des appels de fonctions. Durant cette évaluation, deux cas se présentent :
  - La fonction **échappement** n'est jamais rencontrée. Les instructions sont évaluées normalement et la fonction **récupère** retourne le résultat de la dernière instruction.
  - La fonction **échappement** est exécutée, avec comme argument la même étiquette d'échappement que celle de l'appel à la fonction **récupère**. De plus, durant l'évaluation des instructions, il ne doit pas y avoir eu d'appel à la fonction **récupère** avec cette étiquette d'échappement. L'évaluation des instructions est alors stoppée, et la fonction **récupère** retourne l'étiquette d'échappement. Plusieurs appels à la fonction **récupère** avec la même étiquette d'échappement peuvent être imbriqués, par exemple lors d'appels récursifs à une même fonction : c'est alors le dernier appel effectué qui est pris en compte.
- **échappement**(<nom−échappement>) : un échappement se produit vers le plus récent appel de la fonction **récupère** qui a la même étiquette d'échappement.

L'algorithme utilise deux variables globales (afin d'en simplifier l'écriture) : *GP*, le graphe de planification, et *liste−buts* dont nous reparlerons plus loin. Cette variable servira à représenter l'ensemble des fluents dont la valeur est *requis* ou *requis−faux* ; ce qui permettra l'utilisation d'heuristiques pour le choix des variables de branchement et donnera un critère de succès pour la recherche d'un plan-solution.

La fonction principale DPPlan : elle effectue trois opérations : l'initialisation des valeurs des noeuds du graphe de planification, l'appel à la fonction d'extraction de la solution et le retour de la solution. L'appel à la fonction d'extraction de la solution se fait par deux appels emboîtés de **récupère**, afin de récupérer le nom d'un échappement qui indique le succès ou non de la recherche. La récupération du plan-solution par la fonction **récupération−plan** peut se faire très simplement en collectant niveau par niveau les actions dont la valeur est *vrai*. En effet, on peut démontrer qu'en associant la valeur *faux* à toutes les actions qui sont indéfinies (cf. Théorème 15, page 131), on obtient bien un plan-solution.

```
Fonction DPPlan()
Début
    résultat ← récupère(succès,
                    récupère(inconsistance,
                        initialiser() ;
                        recherche())) ;
    Si résultat = succès alors retourner récupération−plan(GP)
    Sinon retourner échec
Fin
```

Algorithme 16 : DPPlan

La fonction initialiser : elle effectue l'initialisation des valeurs des noeuds du graphe de planification.

- Dans un premier temps, tous les noeuds reçoivent la valeur *indéfini*.
- Ensuite, les fluents qui font partie de l'état initial doivent recevoir la valeur *asserté*, ce qui se fait par appel à la fonction **asserter**. En effet, on peut considérer que l'état initial est une action, appliquée sur un état vide de fluents, qui n'a pas de précondition et produit tous les fluents de l'état initial du problème.
- Enfin, les fluents qui font partie du but du problème doivent recevoir la valeur *requis*, ce qui se fait par appel à la fonction **requérir**. On peut là aussi considérer que le but du problème est un fluent, qui est le seul effet d'une action dont les préconditions sont les fluents du but.

On peut noter que la fonction **initialiser** peut provoquer un échec de la recherche à cause des propagations effectuées par les fonctions **asserter** et **requérir**. C'est alors la fonction DPPlan qui rattrape l'échappement *inconsistance* rencontré dans les fonctions de propagation.

```
Fonction initialiser()
Début
    Pour tout n ∈ Noeuds(GP) faire
        Valeur(n) ← indéfini ;
    Pour tout n_f ∈ NoeudsInit(GP) faire
        asserter(n_f) ;
    Pour tout n_f ∈ NoeudsBut(GP) faire
        requérir(n_f)
Fin % initialiser %
```

Algorithme 17 : initialiser (DPPlan)

La fonction recherche : c'est la fonction principale de l'algorithme. Elle est très proche de la version de base de la procédure de Davis et Putnam, avec cependant une différence essentielle, qui vient du fait que les fluents peuvent prendre une valeur parmi deux valeurs positives et deux valeurs négatives. Les valeurs *requis* et *requis−faux* sont des valeurs temporaires : tant qu'il reste un fluent ayant une de ces deux valeurs, la recherche n'est pas terminée et il faut qu'une action soit appliquée, qui produise ou retire ce fluent.

Le principe de la fonction **recherche** est similaire à celui de la procédure de Davis et Putnam. Tout d'abord, s'il ne reste plus de fluent ayant la valeur *requis* ou *requis−faux*, on peut démontrer qu'un plan-solution existe : il suffit d'associer la valeur *nié* (pour un fluent) ou la valeur *faux* (pour une action) à tous les noeuds ayant la valeur *indéfini*. Ceci correspond dans Davis et Putnam au fait que toutes les variables ont une valeur et qu'il n'y a pas inconsistance : la base est trivialement consistante.

Sinon, et il s'agit là d'une différence avec Davis et Putnam, la fonction vérifie qu'il reste des noeuds ayant la valeur *indéfini*. Si ce n'est pas le cas, la recherche échoue et un retour arrière est effectué. Ce test paraît étrange et nous montrerons qu'il est inutile. En effet, comment peut-on avoir affecté une valeur à tous les noeuds du graphe sans provoquer d'inconsistance ou trouver un plan-solution ?

S'il reste des noeuds ayant la valeur *indéfini*, on sauvegarde les valeurs de tous les noeuds du graphe et la liste des buts à l'aide de la fonction **sauvegarder−valeurs−buts**, dans une variable locale à la fonction **recherche**. On fait ensuite le choix d'un noeud de branchement et d'un booléen grâce à la fonction **choisir**. Le booléen dénote le choix de l'affectation d'une valeur positive ou négative au noeud. On affecte alors une valeur à ce noeud en fonction de la valeur du booléen en utilisant la fonction **assigner** ; puis, on appelle la fonction **recherche**. Si cette dernière échoue, les valeurs des noeuds qui ont été affectés par l'assignation et la recherche sont restaurées, ainsi que la liste des buts (fonction **restaurer−valeurs−buts**) et on relance la recherche avec la valeur opposée du booléen. La fonction **assigner** se charge alors de faire appel à la fonction de propagation qui correspond. En cas d'échec dans la seconde branche, la fonction se termine par un échappement *inconsistance* : un retour arrière est donc effectué dans l'arbre de recherche. Il s'agit ici d'un retour arrière chronologique, puisque le dernier rattrapage de l'échappement *inconsistance* provient soit de l'appel précédent à la fonction **recherche**, auquel cas il reste des branches à explorer ; soit de la fonction **DPPlan**, auquel cas il n'y a pas de plan-solution.

Les fluents ayant la valeur *requis* (resp. *requis−faux*) sont stockés à l'aide de la variable *liste−buts*, ce qui permet l'utilisation d'heuristiques pour le choix des noeuds à traiter : il est en effet très intéressant de s'occuper d'abord de ces noeuds, avant ceux qui ont la valeur *indéfini*, puisque l'on doit absolument leur affecter la valeur *asserté* (resp. *nié*).

```
Fonction recherche()
Début
    Si liste−buts = ∅ alors échappement(succès) ;
    Si ∃n ∈ Noeuds(GP) | Valeur(n) = indéfini alors
        choisir(n, b) ;
        VB ← sauvegarder−valeurs−buts() ;
        récupère(inconsistance,
            assigner(n, b) ;
            recherche()) ;
        restaurer−valeurs−buts (VB) ;
        récupère(inconsistance,
            assigner(n, not b) ;
            recherche()) ;
    échappement(inconsistance)
Fin
```

Algorithme 18 : recherche (DPPlan)

La fonction assigner : elle effectue un appel à la fonction qui va assigner au noeud la valeur désirée puis propager les informations acquises lors de ce changement d'état. D'après l'algorithme original de DPPlan, le noeud peut être soit une action, soit un fluent. Notons ici que les heuristiques a priori les plus efficaces concernent les fluents ayant la valeur *requis* ou *requis−faux*, pour lesquels c'est une action qui est choisie (une action qui ajoute le fluent quand il a la valeur *requis*, une action qui retire le fluent quand il a la valeur *requis−faux*). On peut se demander quel intérêt il peut y avoir à choisir des fluents comme noeuds de branchement de la fonction **recherche**, surtout pour leur donner la valeur *requis* ou *requis−faux*. En effet, pourquoi donner la valeur *requis* à un fluent qui n'est précondition d'aucune action utilisée pour l'instant ? Quelle est alors la signification de cette valeur ?

```
Fonction assigner(n, b)
Début
    Si n ∈ NoeudsA(GP) alors
        Si b alors utiliser(n)
        Sinon exclure(n)
    Sinon
        Si b alors requérir(n)
        Sinon requérir−faux(n)
Fin
```

Algorithme 19 : assigner (DPPlan)

La fonction utiliser : elle spécifie qu'une action fait partie du plan-solution courant en lui affectant la valeur *vrai*, et effectue les propagations induites par ce changement d'état. En effet, si une action fait partie du plan-solution, elle sera appliquée, donc :

- Ses préconditions devront être produites par d'autres actions (appel à la fonction **requérir**).
- Ses ajouts et ses retraits devront être effectués (appels aux fonctions **asserter** et **nier**).
- Les actions mutuellement exclusives avec l'action utilisée ne peuvent pas être appliquées au même niveau, elles sont donc exclues (appel à la fonction **exclure**).

Signalons que l'échappement *inconsistance* est produit avant le traitement si le noeud avait précédemment la valeur *faux*.

```
Fonction utiliser(n_a)
Début
    Cas de
        Valeur(n_a) = faux
            échappement(inconsistance)
        Valeur(n_a) = indéfini
            Valeur(n_a) ← vrai ;
            Pour tout (n_f, n_a) ∈ ArcsPrec(GP) faire
                requérir(n_f) ;
            Pour tout (n_a, n_f) ∈ ArcsAdd(GP) faire
                asserter(n_f) ;
            Pour tout (n_a, n_f) ∈ ArcsDel(GP) faire
                nier(n_f) ;
            Pour tout {n_a, n_b} ∈ MutexA(GP) faire
                exclure(n_b)
Fin
```

Algorithme 20 : utiliser (DPPlan)

La fonction asserter : elle spécifie que le fluent passé en paramètre est présent soit dans l'état initial du problème, soit dans l'état courant du monde après l'application d'une action dont la valeur est *vrai*. Si ce fluent possédait déjà une valeur négative, l'échappement *inconsistance* est produit. Si ce fluent avait la valeur *requis*, on est sûr maintenant qu'une action le supporte et on peut donc lui affecter la valeur *asserté* et le retirer de la liste des buts. Si son état était indéfini, on lui donne la valeur *asserté* et une propagation est possible : tous les fluents mutuellement exclusifs avec le fluent traité doivent recevoir une valeur négative, puisqu'ils ne peuvent pas faire partie de l'état du monde à ce niveau ; on fait donc appel à la fonction *requérir−faux* puisque l'on ne sait pas encore si une action les retire effectivement.

```
Fonction asserter(n_f)
Début
    Cas de
        Valeur(n_f) ∈ {nié, requis−faux} :
            échappement(inconsistance)
        Valeur(n_f) = requis :
            Valeur(n_f) ← asserté ;
            liste−buts ← liste−buts − {n_f}
        Valeur(n_f) = indéfini :
            Valeur(n_f) ← asserté ;
            Pour tout {n_f, n_h} ∈ MutexF(GP) faire
                requérir−faux(n_h)
Fin
```

Algorithme 21 : asserter (DPPlan)

La fonction requérir : elle spécifie que le fluent passé en paramètre doit être présent dans l'état courant du monde après l'application d'une action, mais que l'on ne sait pas encore quelle action appliquer. Si ce fluent possédait déjà une valeur négative, l'échappement *inconsistance* est produit. Si le fluent avait déjà la valeur *asserté*, alors on sait qu'une action qui produit le fluent avait été appliquée et on ne fait rien. Par contre, si aucune action qui produit le fluent n'a la valeur *vrai*, on sait qu'il faudra plus tard en choisir une ; on change donc la valeur du fluent à *requis* et on le place dans la liste des buts. C'est cette liste de buts qui fait la puissance de DPPlan. On pourrait s'en passer en donnant directement au fluent la valeur *asserté* et en n'utilisant plus de liste de buts, mais dans ce cas il faudrait trouver une valeur pour tous les noeuds du graphe. La méthode de gestion utilisée pour cette liste de buts implique l'existence d'un plan-solution lorsque plus aucun fluent n'a la valeur *requis* (ou *requis−faux*, cf. plus loin). Elle permet l'emploi d'heuristiques puisque lorsque un fluent a la valeur *requis*, alors une des actions qui le produit doit être utilisée. Le choix du noeud

de branchement dans l'algorithme de recherche sera donc guidé par les fluents de cette liste. Deux propagations sont possibles :

- La première est identique à celle de la fonction **asserter** : faire appel à la fonction **requérir−faux** pour tous les fluents mutuellement exclusifs avec le fluent considéré.
- La seconde concerne les actions qui retirent le fluent : elles doivent être exclues. Cette propagation est effectuée de façon implicite lors de l'appel à la fonction **asserter** : en effet, l'appel à cette fonction suppose que l'on utilise une action, et donc les actions mutuellement exclusives avec cette dernière sont exclues, et en particulier celles qui retirent le fluent considéré.

```
Fonction requérir(n_f)
Début
    Cas de
        Valeur(n_f) ∈ {nié, requis−faux} :
            échappement(inconsistance)
        Valeur(n_f) = indéfini :
            Valeur(n_f) ← requis ;
            liste−buts ← liste−buts ∪ {n_f} ;
            Pour tout {n_f, n_h} ∈ MutexF(GP) faire
                requérir−faux(n_h) ;
            Pour tout (n_a, n_f) ∈ ArcsDel(GP) faire
                exclure(n_a)
Fin
```

Algorithme 22 : requérir (DPPlan)

La fonction nier : c'est l'équivalent de la fonction **asserter**, pour les valeurs négatives des fluents. Elle n'est appelée que par la fonction **utiliser**, et spécifie que le fluent passé en paramètre sera absent de l'état courant du monde après l'application d'une action. Si ce fluent possédait déjà une valeur positive, l'échappement *inconsistance* est produit. Si ce fluent avait la valeur *requis−faux*, on est sûr maintenant qu'une action le retire et on peut donc lui affecter la valeur *nié* et le retirer de la liste des buts. Si son état était indéfini, on lui donne la valeur *nié* et toutes les actions qui ajoutent ou retirent ce fluent sont exclues : elles recevront la valeur *faux*.

```
Fonction nier(n_f)
Début
    Cas de
        Valeur(n_f) ∈ {asserté, requis} :
            échappement(inconsistance)
        Valeur(n_f) = requis−faux :
            Valeur(n_f) ← nié ;
            liste−buts ← liste−buts − {¬n_f}
        Valeur(n_f) = indéfini :
            Valeur(n_f) ← nié ;
            Pour tout (n_f, n_a) ∈ ArcsPrec(GP) faire
                exclure(n_a) ;
            Pour tout (n_a, n_f) ∈ ArcsAdd(GP) faire
                exclure(n_a)
Fin
```

Algorithme 23 : nier (DPPlan)

Remarquons que si un fluent a la valeur *nié*, c'est qu'il ne peut pas faire partie de l'état courant du monde (après application des actions des niveaux précédents). Or, un fluent peut ne pas faire partie d'un état pour deux raisons : soit parce qu'une action le retire au niveau précédent (ce qui correspond à l'appel de la fonction **nier** par la fonction **utiliser**), soit parce qu'aucune action utilisée ne le produit. Or, ce cas est

traité différemment dans DPPlan : lorsqu'aucune action ne produit un fluent (cas détecté par la fonction **exclure**, voir page suivante), on ne fait pas appel à la fonction **nier** mais à la fonction **requérir−faux** qui est le pendant de la fonction **requérir**, dans la mesure ou il va falloir trouver une action qui retire le fluent. Or dans le cas présent, il paraît évident que si aucune action ne produit un fluent, il faudrait affecter à ce dernier la valeur *nié* puisque l'on est sûr qu'il ne sera pas présent dans l'état courant.

La fonction requérir−faux : c'est l'équivalent de la fonction **requérir**, pour les valeurs négatives des fluents. Elle spécifie que le fluent passé en paramètre doit être absent de l'état courant du monde, éventuellement après l'application d'une action encore ignorée. Si le fluent possédait déjà une valeur positive, l'échappement *inconsistance* est produit. Si le fluent avait déjà la valeur *nié*, alors on sait qu'une action qui retire le fluent a été appliquée et on ne fait rien. Par contre, si aucune action qui retire le fluent n'a la valeur *vrai*, on sait qu'il faudra plus tard en choisir une ; on change donc la valeur du fluent à *requis−faux* et on le place dans la liste des buts. Les mêmes propagations que pour la fonction **nier** sont possibles : l'exclusion des actions qui ajoutent ou utilisent le fluent.

C'est ici qu'il y a un problème de conception dans l'algorithme de DPPlan : en effet, il n'est pas obligatoire d'avoir à appliquer une action pour qu'un fluent soit absent de l'état courant : il suffit qu'aucune action applicable ne le produise. Or, la méthode de gestion de la liste de buts de DPPlan impose qu'il faille trouver une action qui retire le fluent. Le cas ou aucune action ne le produit n'est pas pris en compte.

```
Fonction requérir−faux(n_f)
Début
    Cas de
        Valeur(n_f) ∈ {asserté, requis} :
            échappement(inconsistance)
        Valeur(n_f) = indéfini :
            Valeur(n_f) ← requis−faux ;
            liste−buts ← liste−buts ∪ {¬n_f} ;
            Pour tout (n_f, n_a) ∈ ArcsPrec(GP) faire
                exclure(n_a) ;
            Pour tout (n_a, n_f) ∈ ArcsAdd(GP) faire
                exclure(n_a)
Fin
```

Algorithme 24 : requérir−faux (DPPlan)

La fonction exclure : c'est l'équivalent de la fonction **utiliser**, pour les valeurs négatives des actions. Elle spécifie qu'une action ne peut pas faire partie du plan-solution courant en lui affectant la valeur *faux*, et effectue les propagations possibles dues à ce changement d'état.

Les propagations sont plus complexes que dans la fonction **utiliser** : exclure une action ne signifie pas que les fluents qu'elle produit doivent être niés, une autre action pouvant les produire. Les propagations sur les ajouts et les retraits concernent donc l'ensemble des actions qui produisent ou retirent les fluents des listes d'ajouts et de retraits. En effet, pour les ajouts de l'action exclue :

- Si toutes les actions qui ajoutent un même fluent ont la valeur *faux*, ce fluent ne peut pas avoir une valeur positive. On fait donc appel à la fonction **requérir−faux** afin de trouver une action qui le retire.

  Il y a ici un problème, puisque l'on ne considère pas le cas où le fluent n'existait pas au niveau précédent ; auquel cas il n'y aurait pas besoin de trouver une action qui le retire.
- S'il ne reste qu'une seule action ayant la valeur *indéfini* qui peut produire un fluent et que ce dernier a la valeur *requis*, alors on peut utiliser cette dernière action.

Et pour les retraits de l'action exclue :

- Si toutes les actions qui retirent un même fluent ont la valeur *faux*, alors :
  - si le fluent a la valeur *indéfini* et qu'au niveau précédent il a la valeur *asserté* ou *requis*, alors il doit prendre la valeur *asserté*. On appelle la fonction **requérir**, afin de trouver une action qui le produise.

– si le fluent a la valeur *requis−faux*, alors au niveau précédent il doit avoir aussi la valeur *requis−faux* puisqu'aucune action ne pourrait le retirer s'il avait la valeur *asserté* ou *requis* au niveau précédent. On fait donc appel à la fonction **requérir−faux** sur ce fluent au niveau précédent.

> Soulignons encore une fois le problème de DPPlan : le fluent garde la valeur *requis−faux* et est retiré de la liste des buts, alors que l'on demande qu'il soit faux au niveau précédent. Ce fluent devrait passer à l'état *nié*.

- Si une seule des actions qui retirent un même fluent a la valeur *indéfini* (tous les autres ayant la valeur *faux*), si le fluent a la valeur *requis−faux*, et si au niveau précédent il a la valeur *asserté* ou *requis*, alors il faut utiliser cette dernière action puisque c'est la seule qui peut encore retirer le fluent.

Afin de repérer ces différents cas, la fonction **exclure** utilise les fonctions auxiliaires suivantes, ayant pour argument un fluent :

- **nombre−établisseurs−possibles** (resp. **nombre−destructeurs−possibles**) retourne −1 si au moins une action qui produit (resp. retire) le fluent passé en paramètre a la valeur *vrai*, sinon le nombre d'actions qui produisent (resp. retirent) le fluent et ont la valeur *indéfini*.
- **établisseur−possible** (resp. **destructeur−possible**) retourne une action dont la valeur est *indéfini* et qui produit (resp. retire) le fluent passé en paramètre. Ces deux fonctions ne seront utilisées que dans le cas où les fonctions précédentes retournent 1.

```
Fonction exclure(n_a)
Début
    Cas de
        Valeur(n_a) = vrai :
            échappement(inconsistance)
        Valeur(n_a) = indéfini :
            Valeur(n_a) ← faux ;
            Pour tout (n_a, n_f) ∈ ArcsAdd(GP) faire
                Cas de
                    nombre−établisseurs−possibles(n_f) = 0 :
                        requérir−faux(n_f)
                    nombre−établisseurs−possibles(n_f) = 1 et Valeur(n_f) = requis :
                        utiliser(établisseur−possible(n_f))
                FinCas ;
            Pour tout (n_a, n_f) ∈ ArcsDel(GP) faire
                % avec n_f = f(i) %
                Cas de
                    nombre−destructeurs−possibles(n_f) = 0 :
                        Cas de
                            Valeur(n_f) = indéfini et Valeur(f(i−1)) ∈ {asserté, requis} :
                                requérir(n_f)
                            Valeur(n_f) = requis−faux :
                                liste−buts ← liste−buts − {¬n_f} ;
                                requérir−faux(f(i−1))
                        FinCas ;
                    nombre−destructeurs−possibles(n_f) = 1 et Valeur(n_f) = requis−faux et
                    Valeur(f(i−1)) ∈ {asserté, requis} :
                        utiliser(destructeur−possible(n_f))
                FinCas
Fin
```

Algorithme 25 : exclure (DPPlan)

2.2. Contre-exemple au bon fonctionnement de DPPlan

Nous présentons maintenant la trace d'un contre-exemple très simple qui démontre que la version originale de DPPlan ne fonctionne pas correctement. La description STRIPS de cet exemple comporte seulement deux actions et trois fluents :

Prec(A) = {a} Add(A) = {b} Del(A) = {}
Prec(B) = {a} Add(B) = {c} Del(B) = {a}

L'état initial du problème est I = {a} et le but est G = {b}. Le graphe de planification de ce problème, développé jusqu'au niveau 1, est donné dans la Figure 54. Il contient trois noeuds d'actions et quatre noeuds de fluents.

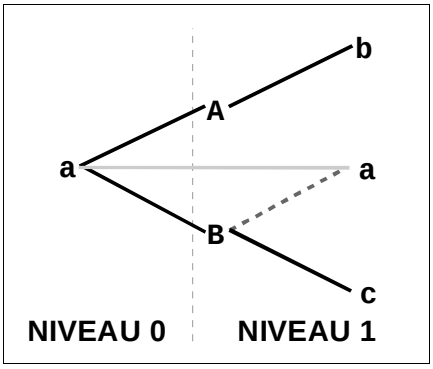

Figure 54 : Graphe de planification du contre-exemple de la version originale de DPPlan

Les paires de noeuds d'actions mutuellement exclusifs sont {A(1), B(1)} et {N_a(1), B(1)}. Les paires de noeuds de fluents mutuellement exclusifs sont {a(1), c(1)} et {b(1), c(1)}.

Nous allons développer le fonctionnement de l'algorithme sur cet exemple, en détaillant les appels de fonctions et les valeurs des principales variables. Le signe "→ OK" après l'appel à une fonction signalera que le noeud concerné avait déjà la valeur demandée et qu'il n'y a ni production d'échappement ni propagation, la fonction se terminant immédiatement.

initialiser()
Valeur(a(0)) ← *indéfini*
Valeur(a(1)) ← *indéfini*
Valeur(b(1)) ← *indéfini*
Valeur(c(1)) ← *indéfini*
Valeur(A(1)) ← *indéfini*
Valeur(B(1)) ← *indéfini*
asserter(*a*(0)) % $a \in I$
Valeur(a(0)) ← *asserté*
requérir(*b*(1)) % $b \in G$
Valeur(b(1)) ← *requis*
liste−buts ← {b(1)}
requérir−faux(*c*(1)) % {b(1), c(1)} ∈ MutexF(GP)
Valeur(c(1)) ← *requis−faux*
liste−buts ← {b(1), ¬c(1)}
exclure(*B*(1)) % $c \in$ Add(B)
Valeur(B(1)) ← *faux*
requérir−faux(*c*(1)) → OK % il n'y a plus d'établisseur possible pour c(1)
requérir(*a*(1)) % il n'y a plus de destructeur possible pour a(1)
Valeur(a(1)) ← *requis*
liste−buts ← {a(1), b(1), ¬c(1)}
requérir−faux(*c*(1)) → OK % {a(1), c(1)} ∈ MutexF(GP)
exclure(*B*(1)) → OK % $a \in$ Del(B)
Fin de l'appel à initialiser

recherche()
  choisir(*A*(1), Vrai)
  VB ← **sauvegarder−valeurs−buts()**
  assigner(*A*(1), Vrai)
    utiliser(*A*(1)) % Valeur(*b*(1)) = *requis* et *b* ∈ Add(*A*)
      Valeur(*A*(1)) ← *vrai*
      requérir(*a*(0)) → OK % *a* ∈ Prec(*A*)
      asserter(*b*(1)) % *b* ∈ Add(*A*)
        Valeur(*b*(1)) ← *asserté*
        liste−buts ← {*a*(1), ¬*c*(1)}
      exclure(*B*(1)) → OK % {*A*(1), *B*(1)} ∈ MutexA(*GP*)
  recherche()
    appel à **choisir** : pas d'action qui retire *c*(1) → échec
  restaurer−valeurs−buts()
  assigner(*A*(1), Faux)
    exclure(*A*(1))
      ...

On voit qu'il est inutile de poursuivre l'exécution de DPPlan, puisque l'on vient d'échouer dans une branche conduisant normalement à la solution. En effet, on a obtenu que l'action *A* soit appliquée au niveau 1, produisant le but, et que l'action *B* qui retire ce dernier ne soit pas appliquée. Or, il se produit le phénomène que nous avons mentionné dans la section précédente : l'algorithme essaye de trouver une action qui retire le fluent *c* alors qu'il n'est pas présent, puisqu'aucune action qui le produit n'est appliquée. Il faudrait que le fluent soit nié lorsque toutes les actions qui le produisent sont exclues ; or, sa valeur est toujours *requis−faux* et il reste présent dans la liste des buts.

2.3. Une tentative de correction de DPPlan : DPPC

Afin de mieux comprendre les modifications que nous apporterons à DPPlan, nous allons dans un premier temps corriger l'algorithme original. En effet, on peut déterminer précisément l'endroit où se situe le problème, et apporter à la fonction concernée la correction (très naturelle) qui s'impose. Nous verrons que cette seule modification démontre clairement l'inutilité de la valeur *requis−faux* pour les fluents, et donc de la fonction **requérir−faux**. Nous verrons dans les sections suivantes comment supprimer cette fonction et améliorer globalement l'algorithme ainsi obtenu.

Le problème qui se pose dans le contre-exemple est le suivant : le noeud *c*(1) reste présent dans la liste des buts avec la valeur *requis−faux*, alors que toutes les actions qui pourraient le produire sont exclues (ici, l'unique action *B*). Dans ce cas, il paraît naturel de changer l'état du noeud *c*(1) en lui assignant la valeur *nié*, et de le retirer de la liste des buts. En effet, toutes les actions pouvant le produire étant exclues, le fluent correspondant à ce noeud ne peut être présent dans aucun état à ce niveau ; il n'est donc pas besoin de chercher une action qui le retire. La question qui se pose est donc la suivante : quelle(s) fonction(s) modifier pour qu'un fluent soit nié, et éventuellement retiré de la liste des buts, lorsque toutes les actions qui le produisent sont exclues ?

Il y a un endroit précis dans l'algorithme où l'on s'intéresse au cas où toutes les actions qui produisent un fluent sont exclues : c'est dans la fonction **exclure**. Lorsqu'on exclut une action, on regarde tous les fluents produits par cette action ; et lorsqu'un de ces derniers n'a plus d'établisseurs possibles, on fait appel à la fonction **requérir−faux** sur ce fluent. Or on vient de voir qu'il faudrait plutôt nier ce dernier, puisqu'en aucun cas il ne pourra être présent dans un état à ce niveau ; de plus, il était peut-être présent dans la liste des buts avec la valeur *requis−faux*, et dans ce cas il faut le sortir de la liste des buts et lui donner la valeur *nié*. Au lieu d'appeler la fonction **requérir−faux** dans ce cas, on va donc appeler la fonction **nier**. Si le fluent avait la valeur *indéfini*, on lui affectera la valeur *nié* et toutes les actions qui pourraient l'utiliser sont exclues ; s'il avait la valeur *requis−faux*, il est retiré de la liste des buts et sa valeur devient *nié*. La fonction **exclure** devient donc :

```
Fonction exclure(n_a)
Début
    Cas de
        Valeur(n_a) = vrai :
            échappement(inconsistance)
        Valeur(n_a) = indéfini :
            Valeur(n_a) ← faux ;
            Pour tout (n_a, n_f) ∈ ArcsAdd(GP) faire
                Cas de
                    nombre−établisseurs−possibles(n_f) = 0 :
                        nier(n_f)                          % avant : requérir−faux(n_f)
                    nombre−établisseurs−possibles(n_f) = 1 et Valeur(n_f) = requis :
                        utiliser(établisseur−possible(n_f))
                FinCas ;
            (...)                                          % fin inchangée %
```

Algorithme 26 : exclure (DPPC)

L'algorithme trouve maintenant une solution au problème de la section 2.2 : le plan-solution est ⟨{A, N_*a*}⟩, qui peut être simplifié en ⟨A⟩. Le déroulement de l'algorithme est identique, jusqu'au moment où l'on exclut le noeud *B*(1). Au lieu de faire appel à la fonction **requérir−faux** sur le noeud *c*(1), on fait maintenant appel à la fonction **nier** : ¬*c*(1) n'est donc pas ajouté à la liste des buts ; quant à *a*(1), il sera asserté par l'utilisation de N_*a*(1).

Nous devons cependant faire une remarque importante sur l'utilité de la fonction r**equérir−faux**. En effet, lorsque l'on fait appel à cette fonction pour un fluent, il se produit le phénomène suivant dans le cas où ce dernier a la valeur *indéfini* : toutes les actions qui le produisent sont exclues. Or si on les exclut toutes, on va forcément déclencher la règle de la fonction **exclure** qui spécifie que si toutes les actions qui produisent un fluent sont exclues, il faut nier ce fluent. Donc, lorsqu'on exclura la dernière action qui produit le fluent, ce fluent sera nié. Comme sa valeur est *requis−faux*, il prendra la valeur *nié* et sera retiré de la liste des buts. Son passage à la valeur *requis−faux* et dans la liste des buts aura donc été temporaire : en entrant dans la fonction **requérir−faux**, il sera effectué, et en sortie de cette même fonction, il sera défait ; ceci apparaît donc comme tout à fait inutile. Nous supprimerons donc la fonction **requérir−faux** et étudierons les modifications que cela entraîne sur l'ensemble de l'algorithme.

3. Étude formelle de DPPlan

Nous allons maintenant effectuer une étude formelle de DPPlan, en nous basant sur les liens entre l'extraction d'un plan-solution dans un graphe de planification et la satisfiabilité de la base de clauses qui correspond à ce graphe. Cette étude nous permettra de proposer deux versions "minimales" de DPPlan, c'est-à-dire comportant le moins de propagations possibles. Nous verrons ainsi les conditions minimales pour gérer de façon optimale la liste des buts. Nous donnerons des éléments de preuve pour la démonstration de la complétude de ces versions de DPPlan, c'est-à-dire le fait qu'elles trouvent un plan-solution dans un graphe de planification s'il en existe au moins un.

Notons toutefois que nous ne démontrerons pas que si on remplace la procédure originelle d'extraction d'un plan-solution de Graphplan par un algorithme de type DPPlan, le planificateur ainsi construit retourne *échec* dans le cas où un problème de planification n'admet pas de solution. En effet, tout comme dans les approches du type SATPLAN et BLACKBOX, il reste le problème de la terminaison. Ce que l'on peut démontrer, c'est que si un plan-solution existe, alors on le trouve ; mais s'il n'en existe pas, on ne peut pas le signifier dans le cas général, contrairement à Graphplan. Pour ce faire, Graphplan utilise une propriété basée sur la mémoization d'ensembles de sous-buts insolubles à un niveau donné ; notre algorithme n'effectuant aucune mémoization, nous sommes pour l'instant dans l'incapacité de formuler un critère d'arrêt. Remarquons que l'on peut quand même retourner *échec* si deux fluents du but sont mutuellement exclusifs au niveau de stabilisation du graphe, ce qui constitue en pratique un critère d'arrêt intéressant (cf. domaines Mprime et Mystery, section 6.6, page 156).

Cette étude nous permettra de donner dans la section suivante une nouvelle version de DPPlan, basée sur la seconde version minimale, dans laquelle nous inclurons le plus possible de règles de propagation.

3.1. Correspondance avec la satisfiabilité d'une base de clauses

Nous allons maintenant établir une correspondance entre un graphe de planification et son codage sous forme de base de clauses, puis montrer les répercussions des fonctions de DPPlan sur cette base de clauses. En effet, on peut voir en DPPlan une implémentation particulière de la procédure de Davis et Putnam. La différence avec un prouveur SAT classique est qu'ici, on connaît à l'avance la forme des clauses que l'on traite : on utilise donc cette connaissance que l'on a de la structure de la base de clauses pour implémenter directement les propagations (unitaires et autres). De plus, le choix des variables de branchement est guidé par la liste des buts qui permet non seulement d'employer des heuristiques qui améliorent l'efficacité de la recherche de la solution, mais aussi (et c'est sans doute le point le plus important) qui fournit un critère spécifiant que l'on peut immédiatement trouver un modèle de la base de clauses (et donc un plan-solution).

Le codage sous forme de base de clauses que nous utiliserons pour la démonstration est le codage dans les espaces d'états avec no-ops extrait du graphe de planification. Il s'agit du codage utilisé par le planificateur BLACKBOX [Kautz et Selman 1999], qui leur a été inspiré par Graphplan (cf. [Kautz, McAllester et Selman 1996]). Nous le nommerons codage GP. Les règles de transformation, décrites dans la Figure 55, ont la signification suivante :

1. **État initial et but :** les fluents de l'état initial et du but sont vrais.
2. **Préconditions des actions :** si une action est utilisée, cela implique que ses préconditions sont vraies au niveau précédent.
3. **Établissement des fluents :** si un fluent est vrai à un niveau supérieur à l'état initial, cela implique que la disjonction des actions qui peuvent le produire au niveau précédent est vraie (no-op compris s'il est présent à ce niveau). Ces clauses sont déterminées par les arcs d'ajouts calculés pendant la construction du graphe de planification.
4. **Exclusions mutuelles :** les actions qui correspondent à des noeuds mutuellement exclusifs ne peuvent pas être vraies en même temps. Ces mutex concernent l'indépendance entre actions et les mutex supplémentaires d'atteignabilité trouvées lors de la construction du graphe.

1. $\left(\bigwedge_{n_f \in \text{NoeudsInit}(GP)} n_f \right) \wedge \left(\bigwedge_{n_f \in \text{NoeudsBut}(GP)} n_f \right)$

2. $\bigwedge_{(n_f, n_a) \in \text{ArcsPrec}(GP)} \left(n_a \Rightarrow n_f \right)$

3. $\bigwedge_{n_f \in (\text{NoeudsF}(GP) - \text{NoeudsInit}(GP))} \left(n_f \Rightarrow \bigvee_{(n_a, n_f) \in \text{ArcsAdd}(GP)} n_a \right)$

4. $\bigwedge_{\{n_a, n_b\} \in \text{MutexA}(GP)} \neg \left(n_a \wedge n_b \right)$

Figure 55 : Codage GP

Notation : Soit *BC* une base de clauses issue du codage GP d'un problème de planification. *BC*(1), *BC*(2), *BC*(3), *BC*(4) dénotent respectivement les formules de *BC* correspondant aux règles 1, 2, 3 et 4 du codage GP. *BC* sera dite d'ordre *k* si elle correspond au codage d'un graphe de planification d'ordre *k*.

En observant ces règles, on s'aperçoit que les propositions qui concernent les fluents peuvent être supprimées. Notre but étant de trouver un plan d'actions, l'état des fluents nous importe peu. Pour utiliser un prouveur SAT, on peut avoir intérêt à utiliser ce codage tel quel, puisque supprimer des variables va probablement faire augmenter le nombre de clauses du codage, voire la taille des clauses. Mais puisqu'avec DPPlan, on ne crée pas la base de clauses, il peut être intéressant d'effectuer cette transformation. Voici les simplifications que l'on peut apporter à ce codage (*BC* étant la base de clauses issue du codage GP d'un problème :

1. Les variables des clauses unitaires de *BC*(1) qui concernent les fluents de l'état initial ne se retrouvent par ailleurs que dans des clauses de *BC*(2), en partie conséquent. En affectant la valeur de vérité *vrai* à ces variables, les clauses de *BC*(1) et *BC*(2) qui les contiennent sont satisfaites et on peut les supprimer de la base.
2. Les variables des clauses unitaires de *BC*(1) qui concernent les fluents du but ne se retrouvent par ailleurs que dans des clauses de *BC*(3), en partie antécédent. En affectant la valeur de vérité *vrai* à ces variables, les clauses de *BC*(1) qui les contiennent sont satisfaites et on peut les supprimer de la base. De plus, on peut simplifier les clauses de *BC*(3) qui les contiennent en ne gardant que la disjonction de la partie conséquent.
3. Enfin, les variables concernant les fluents en partie conséquent des clauses de *BC*(2) qui ne sont pas dans l'état initial ne se retrouvent par ailleurs que dans des clauses de *BC*(3), en partie antécédent. On peut remplacer les clauses de *BC*(2) et *BC*(3) qui concernent les mêmes fluents par leurs résolvantes. En effet, la valeur de vérité d'une variable représentant un fluent ainsi éliminé peut être déterminée en fonction de la valeur de vérité des variables représentant les actions : une variable n_f représentant un fluent prendra la valeur de vérité *vrai* si il existe une clause de *BC*(2) $n_a \Rightarrow n_f$ telle que la variable n_a a la valeur de vérité *vrai* dans le codage simplifié, sinon elle prendra la valeur de vérité *faux*.

La base simplifiée comprend donc les règles suivantes (codage GPA, Figure 56) :

1. **But :** pour chacun des fluents du but, il y a au moins une action qui le produit qui doit avoir été appliquée.
2. **Établissement des préconditions :** les préconditions de chaque action du plan, si elles ne se trouvent pas dans l'état initial, doivent être établies par au moins une action.
3. **Exclusions mutuelles :** règle 4 du codage GP inchangée.

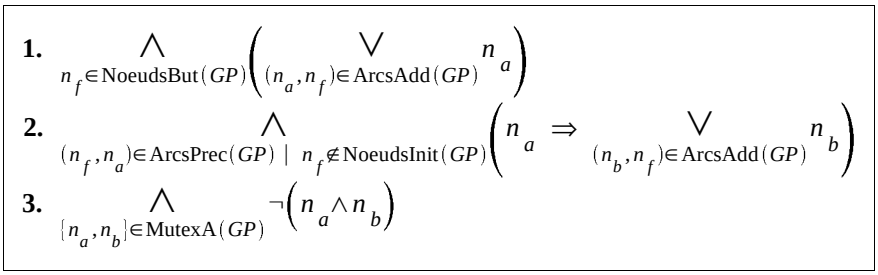

Figure 56 : Codage GPA

Notation : Soit *BC* une base de clauses issue du codage GPA d'un problème de planification. *BC*(1), *BC*(2), *BC*(3) dénotent respectivement les formules de *BC* correspondant aux règles 1, 2 et 3 du codage GPA. *BC* sera dite d'ordre *k* si elle correspond au codage d'un graphe de planification d'ordre *k*.

L'objectif de DPPlan est de calculer un plan-solution, qui corresponde à un modèle de la base de clauses de la manière la plus directe possible ; c'est-à-dire en évitant de considérer des actions qui manifestement n'apportent rien à la résolution. On cherche pour cela à calculer des interprétations partielles particulières qui permettront de trouver immédiatement un modèle de la base de clauses, sans avoir à chercher une valeur pour les propositions encore indéfinies. Ces interprétations partielles correspondent dans DPPlan à l'ensemble des noeuds de fluents ou d'actions qui ont reçu une valeur positive ou négative ; grâce au codage GPA, nous n'avons plus que des noeuds d'actions, ce qui simplifie considérablement le problème.

Définition 31 (interprétation partielle)
Soit *BC* une base de clauses d'ordre *k* obtenue par le codage GPA d'un problème de planification. Une *interprétation partielle IP* de *BC* est un ensemble de littéraux de *BC* tel que :

$$\forall C \in BC,\ IP^{\wedge} \wedge C \text{ est consistant.}$$

Afin de restreindre la recherche d'une interprétation partielle, nous allons maintenir un ensemble de clauses qui doivent être satisfaites (ces clauses correspondent à la liste des buts dans DPPlan) : lorsque cet ensemble sera vide, on pourra trouver immédiatement un modèle de la base de clauses. Le travail essentiel de DPPlan consiste à maintenir cet ensemble des clauses à satisfaire ; tout le reste n'est que propagations supplémentaires améliorant l'efficacité de la résolution. Comme nous le verrons par la suite, c'est pour cela que l'utilisation

des fluents est intéressante. Nous allons d'abord nous attacher à construire une version minimale de DPPLan basée sur la codage GPA.

Définition 32 (ensemble des clauses à satisfaire)

Soit *BC* une base de clauses d'ordre *k* obtenue par le codage GPA d'un problème de planification. Soit *IP* une interprétation partielle de *BC*. L'*ensemble des clauses à satisfaire* de *BC* pour *IP*, noté *CAS*(*BC*, *IP*), est défini par :

$$CAS(BC, IP) = \{C \mid C \in BC(1)^{\{\}}, C^{\{\}} \cap IP = \emptyset\} \cup \{C \mid C \in BC(2)^{\{\}}, C = (n_a \Rightarrow D), n_a \in IP, D^{\{\}} \cap IP = \emptyset\}$$

La raison pour laquelle on doit s'occuper en priorité de la satisfaction de ces clauses est la suivante :

- Les clauses qui concernent les fluents du but doivent être satisfaites, c'est-à-dire que pour chaque fluent du but, une action au moins qui le produit doit avoir été appliquée.
- Si une action est utilisée à partir du niveau 2, c'est qu'au moins une action (ou son no-op) a produit chacune de ses préconditions ; les clauses de *BC*(2) qui concernent les actions qui sont appliquées (la proposition qui leur correspond en partie antécédent appartient à l'interprétation partielle) doivent donc être satisfaites, c'est-à-dire qu'au moins une action qui produit chacune des préconditions de l'action doit être appliquée.

La deuxième notion que nous définissons est la complétion d'une interprétation partielle. En effet, la recherche d'une solution dans DPPlan peut s'arrêter lorsque la liste des buts est vide ; on peut alors trouver directement un plan-solution, c'est-à-dire un modèle de la base de clauses correspondante. Nous montrons donc comment compléter une interprétation partielle pour en faire une interprétation de la base de clauses. Nous démontrerons ensuite que cette interprétation est un modèle de la base de clauses, à condition que l'ensemble des clauses à satisfaire soit vide.

Définition 33 (complétion d'une interprétation partielle)

Soit *BC* une base de clauses d'ordre *k* obtenue par le codage GPA d'un problème de planification. Soit *IP* une interprétation partielle de *BC*. La *complétion* de *IP* pour *BC*, notée *Comp*(*BC*, *IP*), est l'ensemble de littéraux défini par :

$$Comp(BC, IP) = IP \cup \{\neg n_a \mid n_a \in \text{NoeudsA}(GP), n_a \notin IP\}$$

Théorème 15 (calcul d'un modèle par complétion)

Soit *BC* une base de clauses d'ordre *k* obtenue par le codage GPA d'un problème de planification. Soit *IP* une interprétation partielle de *BC*. On a alors :

$$CAS(BC, IP) = \emptyset \Leftrightarrow Comp(BC, IP) \text{ est un modèle de } BC.$$

Preuve :

Soit *BC* une base de clauses d'ordre *k* obtenue par le codage GPA d'un problème de planification. Soit *IP* une interprétation partielle de *BC*.

⇒) *IP* est telle que $CAS(BC, IP) = \emptyset$. Il est immédiat de voir que *Comp*(*BC*, *IP*) est une interprétation de *BC* ; nous allons montrer que *Comp*(*BC*, *IP*) satisfait toutes les clauses de *BC*.

- Satisfaction des clauses de *BC*(1) :
  Soit $C \in BC(1)$. Comme $CAS(BC, IP) = \emptyset$, $C^{\{\}} \cap IP \neq \emptyset$ donc $IP^{\wedge} \models C$ car *C* est une disjonction de littéraux positifs. Comme $IP \subseteq Comp(BC, IP)$, $Comp(BC, IP)^{\wedge} \models C$.
  On a donc bien $Comp(BC, IP)^{\wedge} \models BC(1)$.
- Satisfaction des clauses de *BC*(2) :
  Soit $C \in BC(2)$, avec $C = (n_a \Rightarrow D)$. Si $\neg n_a \in IP$ ou $n_a \notin IP$, $\neg n_a \in Comp(BC, IP)$ donc $Comp(BC, IP)^{\wedge} \models C$. Si $n_a \in IP$, comme $CAS(BC, IP) = \emptyset$, on a :
  $D^{\{\}} \cap IP \neq \emptyset$
  $\Rightarrow Comp(BC, IP)^{\wedge} \models D$ car *D* est une conjonction de littéraux positifs
  $\Rightarrow Comp(BC, IP)^{\wedge} \models C$.
  On a donc bien $Comp(BC, IP)^{\wedge} \models BC(2)$.
- Satisfaction des clauses de *BC*(3) :
  Soit $C \in BC(3)$, avec $C = \neg(n_a \wedge n_b)$. Si $n_a \in IP$, alors $n_b \notin IP$ car $IP^{\wedge} \wedge C$ est consistant puisque *IP* est une interprétation partielle de *BC*. On a donc $\neg n_b \in Comp(BC, IP)$ donc $Comp(BC, IP)^{\wedge} \models C$. Si

$n_a \notin IP$ ou $\neg n_a \in IP$, $\neg n_a \in Comp(BC, IP)$ donc $Comp(BC, IP)^\wedge \models C$.
On a donc bien $Comp(BC, IP)^\wedge \models BC(3)$.
Finalement, $Comp(BC, IP)^\wedge \models BC(1) \wedge BC(2) \wedge BC(3)$ donc $Comp(BC, IP)$ est bien un modèle de BC.
⇐) $Comp(BC, IP)$ est un modèle de BC. Nous allons maintenant montrer que $CAS(BC, IP) = \emptyset$. Supposons que $CAS(BC, IP) \neq \emptyset$.
La première possibilité est qu'il existe une clause $C \in BC(1)$ telle que $C^{\{\}} \cap IP = \emptyset$. Par définition de $Comp(BC, IP)$ on a $\forall n_a \in C^{\{\}}, \neg n_a \in Comp(BC, IP)$. La clause C n'est pas satisfaite, donc $Comp(BC, IP)$ n'est pas un modèle de BC.
La deuxième possibilité est qu'il existe une clause $C \in BC(2)$ telle que :

$$C = (n_a \Rightarrow D),\ n_a \in IP,\ D^{\{\}} \cap IP = \emptyset.$$

Par définition de $Comp(BC, IP)$ on a $\forall n_b \in D^{\{\}}, \neg n_b \in Comp(BC, IP)$. Comme $n_a \in IP$, la clause C n'est pas satisfaite donc $Comp(BC, IP)$ n'est pas un modèle de BC. Il y a contradiction, donc on a bien $CAS(BC, IP) = \emptyset$.

Un corollaire de ce théorème, trivial mais important, est que l'ensemble des clauses à satisfaire d'un modèle de BC est vide. Ainsi, si l'on démontre que l'exécution de DPPlan revient à rechercher toutes les interprétations d'une base d'ordre k dont la liste des clauses à satisfaire est vide, alors DPPlan est complet pour la recherche des modèles de cette base, et donc pour la recherche des plans-solutions de longueur k.

Corollaire 3

Soit BC une base de clauses d'ordre k obtenue par le codage GPA d'un problème de planification. Soit I une interprétation de BC. On a alors :

I est un modèle de $BC \Leftrightarrow I$ est une interprétation partielle de BC et $CAS(BC, I) = \emptyset$

Preuve :

Soit BC une base de clauses d'ordre k obtenue par le codage GPA d'un problème de planification. Soit I une interprétation de BC.
⇒) Si I est un modèle de BC, alors I satisfait toutes les clauses de BC donc I est une interprétation partielle de BC. Par définition de la complétion d'une interprétation partielle, $I = Comp(BC, I)$ donc d'après le Théorème 15, $CAS(BC, I) = \emptyset$.
⇐) Si I est une interprétation partielle de BC et $CAS(BC, I) = \emptyset$, alors d'après le Théorème 15 $Comp(BC, I)$ est un modèle de BC. Comme I est une interprétation de BC, $I = Comp(BC, I)$ donc I est un modèle de BC.

3.2. Version "minimale" de DPPlan

Nous allons maintenant décrire **DPPMin**, une version de DPPlan de laquelle toutes les propagations qui ne sont pas nécessaires sont supprimées. Le but est de montrer que cette version permet d'exploiter les propriétés associées à la base de clauses correspondant au graphe de planification, que nous avons décrites dans la section précédente. Nous donnerons ensuite des éléments de la preuve de la complétude de cette version de DPPlan. Pour cela, nous montrerons l'équivalence de la recherche effectuée par DPPMin avec la recherche de la satisfiabilité de la base de clauses (associée au graphe de planification) par une procédure de Davis et Putnam. Mais il s'agit d'une recherche particulière puisqu'on maintient une liste des buts qui correspond à l'ensemble des clauses à satisfaire pour la base de clauses, afin de stopper la recherche lorsque l'on est sûr qu'un plan-solution (un modèle de la base de clauses) existe.

3.2.1. Algorithme de DPPMin

Au couple formé par un graphe de planification GP et par l'ensemble E des noeuds d'actions qui ont reçu la valeur *vrai* ou *faux*, on peut associer à un endroit précis de l'algorithme le couple unique formé par la base de clauses BC issue du codage GPA de ce graphe et par l'interprétation partielle IP de cette base de clause correspondant à E. En effet, à chaque noeud de E correspond une variable propositionnelle de BC. IP contient donc les variables correspondant aux noeuds de E, sous forme d'un littéral positif (resp. négatif) pour un noeud de E ayant la valeur *vrai* (resp. *faux*). Dans l'algorithme, IP est bien une interprétation partielle de BC à l'endroit où elle est consistante avec chaque clause de BC, c'est-à-dire après avoir testé la présence d'une inconsistance (cf. fonction **recherche**, page 134). Dans la description de DPPMin, nous allons donc établir un parallèle entre les opérations effectuées sur le graphe de planification et la base de clauses qui lui correspond.

La fonction principale DPPMin : elle est quasiment identique à la fonction **DPPlan**, à ceci près que l'appel à la fonction d'initialisation ne peut plus provoquer un échec. En effet, cette fonction va simplement affecter la valeur *indéfini* à toutes les actions et construire la liste des buts (qui correspond à l'ensemble des clauses à satisfaire). On peut aussi supprimer l'échappement dynamique. En effet, ce type d'échappement était introduit en raison de la complexité des propagations ; dans cette version simple de DPPlan, nous supprimons les échappements et nous nous servons uniquement de la valeur de retour des fonctions, c'est-à-dire d'un booléen (Vrai et Faux).

```
Fonction DPPMin()
Début
    initialiser() ;
    Si recherche() alors retourner récupération−plan(GP)
    Sinon retourner échec
Fin
```

Algorithme 27 : DPPMin

La fonction initialiser : elle affecte la valeur *indéfini* à toutes les actions du graphe et construit la liste des buts initiale. Pour maintenir cette liste des buts, nous nous servirons des fluents. En effet, un fluent permet de repérer de manière unique les clauses de *BC*(1) qui correspondent au but : si une action qui produit ce but est appliquée, alors la clause correspondant à ce fluent est satisfaite et le fluent peut être retiré de la liste des buts. Un fluent permet également de repérer de manière unique toutes les clauses de *BC*(2) dont l'action en partie antécédent a pour précondition ce fluent : si une action qui produit ce fluent est appliquée, alors toutes ces clauses sont satisfaites en même temps et le fluent peut être retiré de la liste des buts.

```
Fonction initialiser()
Début
    Pour tout n_a ∈ NoeudsA(GP) faire
        Valeur(n_a) ← indéfini ;
    liste−buts ← NoeudsBut(GP)
Fin
```

Algorithme 28 : initialiser (DPPMin)

La fonction recherche : elle comprend plusieurs différences importantes par rapport à la fonction **recherche** pour DPPlan. En effet, comme on supprime toutes les propagations qui ne sont pas nécessaires à la complétude de l'algorithme :

- La liste des buts peut contenir des fluents tels que pour les produire on ait déjà utilisé une action, parce qu'elle servait à produire d'autres fluents de la liste des buts. L'utilisation de cette action permet donc de satisfaire d'autres clauses appartenant à l'ensemble des clauses à satisfaire. Il faut donc vérifier si le fluent de la liste des buts que l'on choisit n'est pas déjà produit par une action, auquel cas il peut être supprimé de la liste des buts. On voit là l'intérêt de l'utilisation des valeurs *requis* et *asserté* pour les fluents, et de la propagation des informations correspondantes. Pour l'instant, on vérifie que le fluent choisi n'est pas déjà produit par une action ; le cas échéant, on le supprime de la liste des buts et on relance la recherche.
- La liste des buts peut contenir des fluents pour lesquels toutes les actions qui les produisent ont déjà été exclues. En effet, lorsqu'on exclut une action, on ne propage aucune information ; un fluent de la liste des buts peut donc ne posséder aucune action qui le produise. Cela signifie que toutes les clauses de *BC*(1) ou *BC*(2) qu'il représente et qui sont dans l'ensemble des clauses à satisfaire sont falsifiées : l'interprétation partielle ne peut pas être complétée pour être un modèle, il faut effectuer un retour arrière.
- On ne va plus effectuer de propagation sur les noeuds mutuellement exclusifs ; des contraintes d'exclusion mutuelle peuvent donc être violées, c'est-à-dire des clauses de *BC*(3) falsifiées. Il faut donc effectuer la vérification des mutex entre actions.

De plus, la fonction **choisir** retourne maintenant une action produisant un fluent de la liste des buts (les buts négatifs disparaissent), et plus de booléen : on tentera en premier lieu d'utiliser l'action choisie. En effet, on cherche un plan d'actions, et on veut éliminer les fluents de la liste des buts : on a peut donc avoir intérêt à

essayer en premier lieu d'utiliser les actions qui produisent ces derniers. Ce choix est discutable, mais pour cette version minimale de DPPlan il convient tout à fait. On peut maintenant effectuer directement les appels à **utiliser** et **exclure** sans passer par la fonction **assigner**, cette fonction se réduisant à une seule ligne dans chaque cas.

```
Fonction recherche()
Début
    Si (∃n_f ∈ liste−buts | ∀(n_a, n_f) ∈ ArcsAdd(GP), Valeur(n_a) = faux) ou
      (∃{n_a, n_b} ∈ MutexA(GP) | Valeur(n_a) = vrai et Valeur(n_b) = vrai) alors
        retourner Faux
    Sinon
        % A cet endroit, une interprétation partielle IP correspond à l'ensemble E des noeuds
        % ayant reçu une valeur pour la base de clauses BC correspondant à GP.
        Si liste−buts = ∅ alors retourner Vrai
        Sinon
            choisir(n_a, n_f) ;
            VB ← sauvegarder−valeurs−buts() ;
            liste−buts ← liste−buts − {n_f} ;
            Si ∃(n_b, n_f) ∈ ArcsAdd(GP), Valeur(n_b) = vrai alors retourner recherche()
            Sinon
                utiliser(n_a) ;
                Si recherche() alors retourner Vrai
                Sinon
                    restaurer−valeurs−buts(VB) ;
                    exclure(n_a) ;
                    retourner recherche()
Fin
```

Algorithme 29 : recherche (DPPMin)

La fonction utiliser : elle est considérablement simplifiée par rapport à la version précédente de DPPlan. Elle effectue deux opérations :

- D'abord, l'action passée en paramètre prend la valeur *vrai* ; on n'a pas besoin de tester qu'elle a bien la valeur *indéfini*. En effet, la fonction **choisir** n'est pas utilisée s'il existe dans la liste des buts un fluent pour lequel toutes les actions qui le produisent ont la valeur *faux*. Si toutes les actions n'ont pas la valeur *faux*, alors il en existe au moins une qui a soit la valeur *vrai*, soit la valeur *indéfini*. Si elle a la valeur *vrai*, alors la fonction **utiliser** n'est pas appliquée : la recherche est directement relancée en supprimant le fluent de la liste des buts. Cette fonction n'est donc utilisée que dans le cas où l'action a bien la valeur *indéfini* et où le fluent pour laquelle on l'utilise n'est encore produit par aucune action.
- Ensuite, les fluents des préconditions de l'action utilisée sont ajoutés à la liste des buts. Ils peuvent s'y trouver déjà, ou bien être déjà produits par une action utilisée ; aucune vérification n'est faite. On voit là encore l'intérêt d'employer les valeurs *requis* et *asserté* pour les fluents. Rajouter ces fluents dans la liste des buts revient par rapport à la base de clauses, à rajouter toutes les clauses de *BC*(2) dont l'action en partie antécédent correspond à l'action utilisée, dans l'ensemble des clauses à satisfaire.

```
Fonction utiliser(n_a)
Début
    Valeur(n_a) ← vrai ;
    % avec n_a = a(i) %
    Si i > 1 alors
        Pour tout (n_f, n_a) ∈ ArcsPrec(GP) faire
            liste−buts ← liste−buts ∪ {n_f} ;
Fin
```

Algorithme 30 : utiliser (DPPMin)

La fonction exclure : il ne reste qu'une seule opération, la mise à *faux* de la valeur de l'action. On ne vérifie pas que toutes les actions produisant un fluent appartenant à la liste des buts sont exclues, c'est-à-dire que des clauses de *BC*(1) ou *BC*(2) deviennent falsifiées. Cette vérification est effectuée dans la fonction **recherche**.

```
Fonction exclure(n_a)
Début
    Valeur(n_a) ← faux ;
Fin
```

Algorithme 31 : exclure (DPPMin)

Nous pouvons maintenant énoncer le théorème de complétude de **DPPMin** :

Théorème 16 (complétude de DPPMin)
Soit *GP* le graphe de planification pour un problème de planification d'ordre *k*. La fonction **DPPMin** retourne un plan-solution de longueur *k* s'il en existe un.

Schéma de preuve :
Une possibilité de preuve est d'effectuer le parallèle entre un couple ⟨*GP*, *E*⟩ où *GP* est un graphe de planification et *E* est l'ensemble des noeuds d'actions dont la valeur est *vrai* ou *faux*, et le couple ⟨*BC*, *IP*⟩ où *BC* est la base de clauses issue du codage GPA de *GP* est *IP* est l'interprétation partielle de *BC* correspondant à *E*.
Il faut d'abord démontrer que (α) *IP* peut être étendue à un modèle de *BC* si et seulement si *E* peut être étendu à un ensemble dont on peut extraire un plan-solution. Cette démonstration est à la base du planificateur BLACKBOX [Kautz et Selman 1999], mais n'a (à notre connaissance) jamais été publiée.
On peut ensuite démontrer que la fonction **recherche** a un comportement similaire à celui de la procédure de Davis et Putnam exécutée sur le couple ⟨*BC*, *IP*⟩ correspondant au couple ⟨*GP*, *E*⟩, la seule différence étant la gestion de l'ensemble des clauses à satisfaire :

- D'abord, on vérifie que la base n'est pas trivialement inconsistante, c'est-à-dire qu'une clause de *BC* n'est pas falsifiée par l'interprétation partielle. Les seules clauses d'un codage GPA pouvant être falsifiées sont les clauses de *BC*(1) et *BC*(2) appartenant à *CAS*(*BC*, *IP*), et les clauses de *BC*(3). En effet, les clauses n'appartenant pas à *CAS*(*BC*, *IP*) sont soit déjà satisfaites, soit satisfiables par complétion de l'interprétation partielle (si *BC* est consistante).
- Ensuite, on teste si la base de clauses est trivialement consistante, c'est-à-dire que l'ensemble des clauses à satisfaire *CAS*(*BC*, *IP*) est vide. Le cas échéant, la base est consistante et on peut trouver un modèle, d'après le Théorème 15. D'après (α), il existe une extension de *E* dont on peut extraire un plan-solution.
- Enfin, on choisit un fluent *f* de la liste des buts, ainsi qu'une action *a* qui le produit dont la valeur est soit *vrai*, soit *indéfini*. Ce fluent et cette action représentent soit la clause de *BC*(1) dans laquelle *f* est un but du problème et *a* est une action de la disjonction des actions qui produisent *f*, soit l'ensemble des *n* clauses de *BC*(2) telles que *f* appartient aux préconditions de *n* actions (chacune étant en partie antécédent de ces *n* clauses) et *a* est une action de la disjonction en partie conséquent. Ces clauses de *BC*(1) ou *BC*(2) sont soit déjà satisfaites, soit appartiennent à *CAS*(*BC*, *IP*).
  On teste donc la valeur des actions qui produisent *f* : si une action a la valeur *vrai*, alors les clauses de l'ensemble des clauses à satisfaire que représentent ce fluent sont satisfaites, et on peut relancer la

recherche.
Sinon, la valeur de l'action est *indéfini*. On teste dans un premier temps la valeur *vrai* en appelant la fonction **utiliser** puis la fonction **recherche**, et ensuite la valeur *faux* en appelant la fonction **exclure** puis la fonction **recherche**. La fonction **utiliser** ajoute dans la liste des buts les préconditions de l'action utilisée, qui correspondent à toutes les nouvelles clauses à mettre dans l'ensemble des clauses à satisfaire (elles peuvent être déjà satisfaites). Pour la base de clauses, ceci correspond au choix d'une variable de branchement et à la création de deux branches dans l'arbre de recherche : une pour la valeur positive et l'autre pour la valeur négative.

3.2.2. Exemple de fonctionnement de DPPMin

Reprenons maintenant l'exemple de la section 2.2 et traitons le avec DPPMin.

initialiser()
Valeur(A(1)) ← *indéfini*
Valeur(B(1)) ← *indéfini*
liste−buts ← {b(1)} % $b \in G$
Fin de l'appel à initialiser.

recherche()
choisir(*A*(1), *b*(1))
liste−buts ← {}
utiliser(*A*(1)) % $b \in$ Add(A)
Valeur(A(1)) ← *vrai*
recherche()
retourner Vrai % *liste−buts* = ∅
Fin de l'appel à recherche.

La recherche se termine donc par un succès, le plan-solution retourné étant le bon.

La première remarque que l'on peut faire sur le fonctionnement de **DPPMin** est qu'il effectue beaucoup moins de propagations que la version originale de DPPlan, ce qui paraît être un avantage sur cet exemple puisque l'on trouve la solution plus rapidement. Mais pour un problème plus difficile, un nombre restreint de propagations entraînera fatalement un nombre plus important de choix à effectuer dans la fonction **recherche**, ce qui provoquera une sérieuse dégradation des performances, comme nous le verrons dans la partie expérimentale.

La deuxième remarque que l'on peut faire est que le modèle que l'on extrait de la base de clauses correspondant au graphe de planification n'est pas une représentation exacte de la réalité. En effet, comme le noeud du no-op N_*a(*1) n'a pas reçu de valeur, alors par complétion de l'interprétation partielle il recevra la valeur *faux*, ce qui ne correspond pas à la réalité. Mais ce phénomène n'a aucune importance, puisque seules les "vraies" actions nous intéressent. A partir d'un plan-solution constitué uniquement de "vraies" actions, il est facile de reconstituer tous les états successifs et par conséquent les no-ops employés. Ceci s'est produit ici parce que le fluent *a* n'est utile que dans l'état initial. S'il avait fait parti du but, la valeur du noeud *a*(1) aurait été *requis* et on aurait forcément utilisé le no-op N_*a(*1).

3.3. Seconde version "minimale" de DPPlan : amélioration de la gestion de la liste des buts

3.3.1. Idées de base

Le principal défaut de **DPPMin** est sa très mauvaise gestion de la liste des buts, qui implique que l'on doive vérifier que la base de clauses correspondant au graphe de planification est trivialement inconsistante ou pas, ce qui peut s'avérer assez laborieux à effectuer. En effet, dans **DPPMin**, cette vérification est faite à chaque passage dans la fonction **recherche** de façon globale sur la base.

Comme à chaque fois que l'on affecte une valeur à une action, on sait exactement quelles répercussions cela va avoir sur les clauses de la base, il semble judicieux de faire cette vérification uniquement sur les

clauses concernées ; d'où l'idée de donner des valeurs aux fluents. On n'a pas besoin de considérer que ces fluents font partie de la base de clauses, et de revenir au codage GP du graphe de planification. On a vu que les fluents de la liste des buts représentent chacun un ensemble de clauses ; il suffit donc de considérer que les valeurs que l'on donnera à ces fluents sont une façon de repérer l'état de chacune des clauses de la base :

- La valeur *asserté* :
  - Si le fluent fait partie du but du problème, alors la clause de *BC*(1) qui correspond à ce fluent est satisfaite par l'interprétation partielle, c'est-à-dire que l'on a utilisé une action qui produit ce fluent. On peut enlever la clause de l'ensemble des clauses à satisfaire, c'est-à-dire enlever le fluent de la liste des buts.
  - Si le fluent ne fait pas partie du but, les clauses de *BC*(2) qui correspondent à ce fluent sont satisfaites. Ces clauses sont telles qu'une action ayant ce fluent pour précondition implique la disjonction des actions qui produisent ce fluent. On a donc utilisé une de ces actions pour produire le fluent. On peut enlever le fluent de la liste des buts s'il en faisait partie.
- La valeur *requis* :
  - Si le fluent fait partie du but du problème, alors la clause de *BC*(1) qui correspond à ce fluent n'est pas encore satisfaite par l'interprétation partielle. Elle doit donc faire partie de l'ensemble des clauses à satisfaire, c'est-à-dire que le fluent doit faire partie de la liste des buts.
  - Si le fluent ne fait pas partie du but, les clauses de *BC*(2) qui correspondent à ce fluent ne sont pas encore satisfaites, et elles ne pourront pas être satisfaites par complétion de l'interprétation partielle : l'action en partie antécédent a la valeur *vrai*. Il faut donc affecter la valeur *vrai* à une action qui produit le fluent, qui fait donc partie de la liste des buts.
- La valeur *nié* :
  - si le fluent fait partie du but du problème, alors la clause de *BC*(1) qui correspond à ce fluent est falsifiée par l'interprétation partielle car toutes les actions de cette clause ont la valeur *faux*. Il y a échec de la recherche.
  - Si le fluent ne fait pas partie du but, deux cas peuvent se produire :
    - Soit les clauses de *BC*(2) qui correspondent à ce fluent ne font pas partie de l'ensemble des clauses à satisfaire, et elles n'en feront jamais partie : elles seront alors satisfaites par complétion de l'interprétation partielle.
    - Soit une des clauses de *BC*(2) qui correspond à ce fluent fait partie de l'ensemble des clauses à satisfaire, et il y a échec de la recherche puisque l'action en partie antécédent a la valeur *vrai* tandis que les d'actions en partie conséquent ont la valeur *faux*.

Si à chaque appel de la fonction **recherche**, les valeurs des fluents ont bien la sémantique décrite ci-dessus, alors on est sûr que la base de clauses n'est pas trivialement inconsistante et que la liste des buts ne contient pas de fluent pour lequel une action qui le produit a la valeur *vrai* : la fonction **recherche** en sera considérablement simplifiée. Le problème consiste donc à maintenir la valeur de ces fluents. Nous proposons ici une seconde version minimale de DPPlan, **DPPMin2**, qui effectue cette opération.

3.3.2. Algorithme de DPPMin2

La fonction principale DPPMin2 : elle est identique à la fonction **DPPMin**.

```
Fonction DPPMin2()
Début
    initialiser() ;
    Si recherche() alors retourner récupération−plan(GP)
    Sinon retourner échec
Fin
```

Algorithme 32 : DPPMin2

La fonction initialiser : par rapport à la fonction **initialiser** de **DPPMin**, elle affecte la valeur *indéfini* aux fluents et affecte la valeur *requis* aux fluents qui font partie du but du problème.

```
Fonction initialiser()
Début
    Pour tout n ∈ Noeuds(GP) faire
        Valeur(n) ← indéfini ;
    Pour tout n_f ∈ NoeudsBut(GP) faire
        Valeur(n_f) ← requis ;
    liste−buts ← NoeudsBut(GP) ;
Fin
```

Algorithme 33 : initialiser (DPPMin2)

La fonction recherche : elle ne teste plus si la base est trivialement inconsistante. De plus, si un fluent fait partie de la liste des buts, alors ce fluent a la valeur *requis* donc les clauses qu'il représente font partie de l'ensemble des clauses à satisfaire tel qu'il est décrit dans la Définition 32. La fonction **choisir** retourne donc une action ayant la valeur *indéfini* permettant de satisfaire un ensemble de clauses représentées par un fluent de la liste des buts. On ne teste donc pas si ces clauses sont déjà satisfaites. Les fonctions **utiliser** et **exclure** retournent maintenant un booléen indiquant si la base de clauses est trivialement inconsistante ou non, et vont maintenir de façon précise l'ensemble des clauses à satisfaire.

```
Fonction recherche()
Début
    Si liste−buts = ∅ alors retourner Vrai
    Sinon
        choisir(n_a) ;
        VB ← sauvegarder−valeurs−buts() ;
        Si utiliser(n_a) et recherche() alors retourner Vrai
        Sinon
            restaurer−valeurs−buts(VB) ;
            retourner exclure(n_a) et recherche()
Fin
```

Algorithme 34 : recherche (DPPMin2)

La fonction utiliser : une action peut se trouver :

- En partie antécédent de clauses de *BC*(2). On repère ces clauses par les préconditions de l'action. Il faut alors trouver une interprétation partielle telle que ces clauses soient satisfaites. Pour chacune d'entre elles, on vérifie que le fluent qui les repère n'a pas la valeur *nié*, auquel cas la clause est falsifiée par l'interprétation partielle. Si ce fluent a la valeur *asserté*, la clause est satisfaite donc on ne fait rien. S'il a la valeur requis, la clause fait toujours partie de l'ensemble des clauses à satisfaire donc on ne fait rien. Enfin, s'il a la valeur *indéfini*, la clause doit faire partie de l'ensemble des clauses à satisfaire donc on ajoute le fluent à la liste des buts et on lui donne la valeur *requis.*
- Dans des clauses de *BC*(1) ou en partie conséquent de clauses de *BC*(2). On repère ces clauses par les ajouts de l'action. Elles sont alors satisfaites, le fluent qui les repère prend donc la valeur *asserté*. S'il avait précédemment la valeur *requis*, il faut l'enlever de la liste des buts puisque les clauses qu'il repère sont maintenant satisfaites. Remarquons qu'il ne pouvait pas avoir la valeur *nié*, sinon on n'aurait pas pu utiliser l'action qui le produit puisqu'il aurait eu la valeur *faux*. En effet, un fluent ne prend la valeur *nié* que lorsque toutes les actions qui le produisent ont la valeur *faux* (cf. fonction **exclure**).
- Dans des clauses de *BC*(3). Il faut alors vérifier que ces clauses ne sont pas falsifiées dans l'interprétation courante.

```
Fonction utiliser(n_a)
Début
    Valeur(n_a) ← vrai ;
    % avec n_a = a(i) %
    Si i > 1 alors
        Pour tout (n_f, n_a) ∈ ArcsPrec(GP) faire
            Cas de
                Valeur(n_f) = nié :
                    retourner Faux
                Valeur(n_f) = indéfini :
                    Valeur(n_f) ← requis ;
                    liste−buts ← liste−buts ∪ {n_f}
            FinCas ;
    Pour tout (n_a, n_f) ∈ ArcsAdd(GP) faire
        Cas de
            Valeur(n_f) = requis :
                Valeur(n_f) ← asserté ;
                liste−buts ← liste−buts − {n_f}
            Valeur(n_f) = indéfini :
                Valeur(n_f) ← asserté
        FinCas ;
    Pour tout {n_a, n_b} ∈ MutexA(GP) faire
        Si Valeur(n_b) = vrai alors retourner Faux ;
    retourner Vrai
Fin
```

Algorithme 35 : utiliser (DPPMin2)

La fonction exclure : si l'action se trouve en partie antécédent de clauses de *BC*(2), ces clauses sont satisfaites, il n'y a rien à faire. S'il se trouve dans une clause de *BC*(1) ou en partie conséquent d'une clause de *BC*(2) repérée par un ajout de l'action, deux cas peuvent se produire :

- Soit il y a des actions en partie conséquent dont la valeur est *vrai* ou *indéfini*, donc la clause est soit déjà satisfaite soit encore satisfiable. Il n'y a donc rien à faire.
- Soit toutes les actions en partie conséquent ont la valeur *faux.* Deux cas se présentent alors : soit la clause fait partie de l'ensemble des clauses à satisfaire, auquel cas elle est falsifiée par l'interprétation partielle et il y a échec de la recherche ; soit elle ne fait pas partie de l'ensemble des clauses à satisfaire, auquel cas on interdit qu'elle en fasse jamais partie en affectant la valeur *nié* au fluent qui la repère. En effet, dans la fonction **utiliser**, on ajoute un fluent à la liste des buts s'il n'a pas la valeur *nié*.

```
Fonction exclure(n_a)
Début
    Valeur(n_a) ← faux ;
    Pour tout (n_a, n_f) ∈ ArcsAdd(GP) faire
        Si nombre−établisseurs−possibles(n_f) = 0 alors
            Cas de
                Valeur(n_f) = requis :
                    retourner Faux
                Valeur(n_f) = indéfini :
                    Valeur(n_f) ← nié ;
    retourner Vrai
Fin
```

Algorithme 36 : exclure (DPPMin2)

Nous pouvons maintenant énoncer le théorème de complétude de **DPPMin2** :

Théorème 17 (complétude de DPPMin2)
Soit *GP* le graphe de planification pour un problème de planification d'ordre *k*. La fonction **DPPMin2** retourne un plan-solution de longueur *k* s'il en existe un.

Schéma de preuve :
La seule différence avec **DPPMin** est que l'on gère mieux la liste des buts, en utilisant des valeurs pour les fluents. Ainsi, la liste des buts ne contient plus que des fluents ayant la valeur *requis*, pour lesquels il faut absolument trouver une action qui les produise ; les fluents pour lesquels ce n'est plus la peine ont en effet la valeur *asserté*. De plus, on sait qu'il existe au moins une action qui peut les produire, car cette vérification est faite à chaque fois que l'on exclut une action. Ceci permet donc de simplifier la fonction **recherche**.
La preuve sera donc basée sur celle de **DPPMin**, en montrant que l'on ne rajoute pas de clause dans l'ensemble des clauses à satisfaire si elles sont déjà satisfaites, et que l'on détecte plus tôt qu'une clause appartenant à cet ensemble est falsifiée.

4. Un nouvel algorithme pour DPPlan

4.1. Idées de base

Les modifications que nous allons apporter à DPPlan sont basées sur les remarques que nous avons faites sur les fonctions de propagation de la version originale de DPPlan et sur la seconde version minimale de DPPlan. Cette dernière va servir de base à notre nouvelle version de DPPlan, à laquelle nous allons ajouter toutes les propagations possibles.

L'idée essentielle de ces modifications est que le problème de la version originale de DPPlan est basé sur une mauvaise prise en compte de la valeur *requis−faux*, mais aussi sur le fait que cette valeur est inutile, en particulier à cause des diverses règles de propagation. En effet, nous avons vu que la fonction **DPPMin2** est complète et n'utilise en aucune manière une valeur *requis−faux* pour les fluents.

Les améliorations que nous apportons concernent plusieurs aspects de DPPlan :

- Suppression de la valeur *requis−faux* et par conséquent de la fonction **requérir−faux**. En effet, nous avons vu dans la première tentative de correction de DPPlan et dans l'exemple qui suivait que cette valeur ne sert à rien, puisqu'un fluent passe automatiquement (c'est-à-dire dans la même suite de propagations, sans passage dans la fonction **recherche**) de la valeur *requis−faux* à la valeur *nié*. De plus, pour une gestion de la liste des buts comme nous l'avons étudiée pour **DPPMin2**, cette valeur est tout à fait inutile.

- Simplification des fonctions **recherche**, **choisir** et **assigner**. En effet, la liste des buts ne contient plus maintenant que des fluents ayant la valeur *requis*, et pour lesquels il reste au moins une action qui les produit ayant la valeur *indéfini*. Il semble évident que pour réduire cette liste des buts, on doive utiliser une telle action, puis ensuite l'exclure en cas d'échec. De plus, les meilleures heuristiques indépendantes du domaine étudiées récemment (cf. [Bonet et Geffner 1999], [Bonet et Geffner 2001]) portent sur l'utilisation des actions, et non sur leur exclusion. L'objectif est tout de même, ne l'oublions pas, de construire un plan d'actions... il nous paraît donc naturel que les "bonnes" heuristiques dirigent le choix des actions à insérer dans un plan, et non le choix des actions à rejeter.

- Suppression de certaines propagations, qui sont effectuées de manière redondante, dans les fonctions **asserter** et **nier**. La détection de telles redondances n'est pas une tâche aisée, au vu des nombreuses interactions existant entre les fonctions de propagation. Il sera donc difficile de déterminer si notre nouvelle version de DPPlan ne comporte plus de propagation redondante ; mais quoi qu'il en soit, le gain en temps de calcul n'est pas énorme puisqu'un appel redondant à une fonction de propagation est interrompu immédiatement, en fonction de la valeur du noeud passé en paramètre.
- Ajout de règles de propagation inspirées de celles de la fonction **exclure** dans les fonctions **requérir** et **nier**. Par exemple, pour chaque ajout d'une action que l'on exclut, on regarde combien il reste d'établisseurs possibles. Dans le cas où il n'en reste qu'un, on l'utilise si l'ajout en question a la valeur

requis. Pourquoi ne pas faire la même chose dans la fonction *requérir* ? S'il ne reste qu'un seul établisseur possible, on l'utilisera.

- Amélioration des règles de propagation de la fonction **exclure**, notamment par une prise en compte améliorée des no-ops.

4.2. Nouvel algorithme de DPPlan : DPP

Cette nouvelle version est basée sur une version modularisée de **DPPMin2** par les fonctions **asserter**, **requérir** et **nier** que l'on peut facilement extraire des fonctions **utiliser** et **exclure**, utilisant à nouveau les échappements dynamiques non locaux.

La fonction **DPP** qui effectue l'appel à la fonction d'initialisation et lance la recherche ainsi que la fonction **initialiser** restent inchangées par rapport à la version originale de DPPlan ; les premières modifications concernent donc la fonction **recherche**.

```
Fonction DPP()
Début
    résultat ← récupère(succès,
        récupère(inconsistance,
            initialiser() ;
            recherche())) ;
    Si résultat = succès alors retourner récupération−plan(GP)
    Sinon retourner échec
Fin
```

Algorithme 37 : DPP

```
Fonction initialiser()
Début
    Pour tout n ∈ Noeuds(GP) faire
        Valeur(n) ← indéfini ;
    Pour tout n_f ∈ NoeudsInit(GP) faire
        asserter(n_f) ;
    Pour tout n_f ∈ NoeudsBut(GP) faire
        requérir(n_f)
Fin
```

Algorithme 38 : initialiser (DPP)

La fonction recherche : il s'agit de la fonction **recherche** utilisée par **DPPMin2**, à laquelle on a rajouté l'utilisation des échappements dynamiques.

```
Fonction recherche()
Début
    Si liste−buts = ∅ alors échappement(succès)
    Sinon
        choisir(n_a) ;
        VB ← sauvegarder−valeurs−buts() ;
        récupère(inconsistance,
            utiliser(n_a) ;
            recherche()) ;
        restaurer−valeurs−buts (VB) ;
        récupère(inconsistance,
            exclure(n_a) ;
            recherche()) ;
    échappement(inconsistance)
Fin
```

Algorithme 39 : recherche (DPP)

La fonction utiliser : la seule différence par rapport à la version originale de DPPlan est la suppression des propagations sur les retraits de l'action considérée. En effet, nous avons vu dans la version minimale de DPPlan que ces propagations ne sont absolument pas nécessaires ; de plus, elles sont redondantes avec les propagations sur les actions mutuellement exclusives : si l'action retire un fluent donné, alors elle va être mutuellement exclusive avec toutes les actions qui ajoutent ce fluent (dans le cas où on utilise la relation d'indépendance). On va donc exclure toutes ces actions, et lorsqu'elles vont être toutes exclues, le fluent en question va être nié par la règle de la fonction **exclure**. On n'a donc pas besoin de le nier immédiatement.

> De plus, nous verrons dans la section suivante que la suppression de cette règle de propagation redondante permet d'utiliser la relation d'autorisation faible. On pourra ainsi avoir à un même niveau une action n_a qui produit un fluent f et une action n_b qui retire ce même fluent, avec n_a et n_b ayant la valeur *vrai* et f ayant la valeur asserté. Si on conservait la règle qui nie les retraits, ce serait impossible ; or l'action n_b peut très bien autoriser faiblement l'action n_a.

Les différences par rapport à la fonction **utiliser** de la version minimale de DPPlan sont les suivantes :

- Il faut rétablir le test sur la valeur du fluent : s'il a déjà la valeur *faux*, il faut produire un échappement. En effet, ce cas peut très bien se produire grâce aux propagations, en particulier à cause de l'utilisation des no-ops : on utilisera explicitement un no-op, en sachant qu'un échappement va être produit si besoin est.
- Le traitement effectué sur les préconditions et les ajouts de l'action considérée est extrait de la fonction **utiliser**, en introduisant les fonctions **requérir** et **asserter**. Elles vont effectuer un traitement identique à celui qui était fait dans **utiliser**, avec cependant des propagations supplémentaires pour **requérir**.
- La propagation sur les actions mutuellement exclusives avec l'action considérée est effectuée par la fonction **exclure**. En effet, si l'on se réfère au codage GPA, on se retrouve avec des clauses unitaires dans *BC*(3) qu'il faut satisfaire. La seule façon de les satisfaire est donc d'exclure la seconde action.

```
Fonction utiliser(n_a)
Début
    Cas de
        Valeur(n_a) = faux :
            échappement(inconsistance)
        Valeur(n_a) = indéfini :
            Valeur(n_a) ← vrai ;
            Pour tout (n_f, n_a) ∈ ArcsPrec(GP) faire
                requérir(n_f) ;
            Pour tout (n_a, n_f) ∈ ArcsAdd(GP) faire
                asserter(n_f) ;
            Pour tout {n_a, n_b} ∈ MutexA(GP) faire
                exclure(n_b)
Fin
```

Algorithme 40 : utiliser (DPP)

La fonction asserter : elle est simplifiée par la suppression de la règle de propagation qui concerne les fluents qui sont mutuellement exclusifs avec le fluent considéré. En effet, la fonction **asserter** est appelée uniquement par la fonction **utiliser**. Or cette dernière exclut toutes les actions mutuellement exclusives avec l'action que l'on utilise. Deux fluents sont mutuellement exclusifs lorsque toutes les actions qui les produisent sont mutuellement exclusives deux à deux ; ainsi, toutes les actions qui produisent les fluents mutuellement exclusifs avec le fluent considéré sont exclues, et donc ces fluents sont automatiquement niés par la fonction **exclure**, par la règle de propagation qui nie un fluent si toutes les actions qui le produisent sont exclues.

Par rapport à la version minimale de DPPlan, on a juste rétabli le test de la valeur *nié* du fluent. Ce test n'est pas obligatoire, puisque si le fluent a été nié précédemment, les actions qui le produisent doivent être exclues. Mais par le jeu des propagations, le fluent peut être asserté avant que les actions qui le produisent soient exclues. Avec ce test, on détecte plus tôt l'inconsistance.

```
Fonction asserter(n_f)
Début
    Cas de
        Valeur(n_f) = nié :
            échappement(inconsistance)
        Valeur(n_f) = requis :
            Valeur(n_f) ← asserté ;
            liste−buts ← liste−buts − {n_f}
        Valeur(n_f) = indéfini :
            Valeur(n_f) ← asserté
Fin
```

Algorithme 41 : asserter (DPP)

La fonction requérir : elle est améliorée par l'utilisation de règles de propagation inspirées de celles de la fonction **exclure** :

- Lorsque le fluent a une valeur positive au niveau précédent, alors il y a au moins une action qui est utilisée : le no-op du fluent considéré. On peut donc utiliser directement ce no-op, qui va asserter le fluent.

  > Nous verrons dans la section suivante que ce cas doit être supprimé pour une prise en compte optimale de la relation d'autorisation faible. En effet, un fluent peut être vrai dans deux niveaux successifs sans que son no-op soit utilisé : deux actions faiblement autorisées dont l'une retire le fluent et l'autre le rajoute peuvent être utilisées à un même niveau.

- Lorsque toutes les actions qui produisent le fluent considéré sont exclues, il y a échec de la recherche : on produit un échappement. Il n'est pas indispensable de considérer ce cas : en effet, il sera forcément pris en compte par la fonction **exclure**. Comme le fluent prend la valeur *requis* et comme il n'y a plus

d'établisseurs possibles pour ce fluent, on a forcément fait appel à **exclure** et la règle de propagation de cette dernière va être déclenchée. Elle ne l'a pas été de suite du fait des appels imbriqués aux règles de propagation, mais comme le cas se présente, on peut détecter plus tôt l'inconsistance.

- Lorsqu'il ne reste qu'une seule action qui supporte le fluent considéré, alors il faut forcément utiliser cette action. Il n'y a donc pas besoin de changer la valeur du fluent à *requis* et d'exécuter les propagations ; il suffit d'utiliser l'action, qui va asserter le fluent et effectuer elle-même les propagations nécessaires. Ce cas n'avait pas pu être détecté par la fonction **exclure** lors de l'exclusion des autres actions qui ajoutent le fluent (s'il y en avait), puisque le fluent avait la valeur *indéfini*. La règle correspondante n'avait donc pas été déclenchée.
- Lorsqu'il reste plusieurs actions qui produisent le fluent considéré, on effectue les mêmes calculs que l'ancienne fonction **requérir**, à ceci près que l'on ne fait plus appel à la fonction **requérir−faux** pour les fluents mutuellement exclusifs avec le fluent considéré, mais à la fonction **nier**. Lorsque ces fluents sont niés au niveau précédent, il n'y a pas de problème : aucune action qui les retire n'est nécessaire. C'est le cas qui n'était pas pris en compte dans la version originale de DPPlan. Nous avons vu dans la seconde version minimale de DPPlan que le cas où une action doit les retirer est automatiquement pris en compte dans les fonctions **nier** et **exclure**, et qu'il n'y a donc pas besoin d'effectuer le choix d'une telle action dans la fonction **recherche**. Il nous faut noter une différence pour l'appel à la fonction **nier**. Cette fonction a maintenant besoin de deux paramètres : le fluent à nier, et un booléen qui indique s'il est nécessaire ou non d'effectuer toutes les propagations de la fonction **nier**. En effet, la fonction **nier** exclut les actions qui produisent le fluent que l'on nie ; or ces actions peuvent être déjà exclues : ce cas se produit dans la fonction **exclure**. Leur exclusion étant redondante, on en empêche l'exécution dans la fonction **nier** en positionnant le booléen à Faux. Ici, ces propagations sont utiles. En effet, les actions qui produisent les fluents mutuellement exclusifs avec le fluent considéré ne sont pas exclues dans la fonction **requérir**, puisque l'on ne sait pas quelle action produit ce fluent : on ne peut donc pas exclure automatiquement les actions mutuellement exclusives avec une action que l'on ne connaît pas encore.

```
Fonction requérir(n_f)
Début
    Cas de
        Valeur(n_f) = nié :
            échappement(inconsistance)
        Valeur(n_f) = indéfini :
            % avec n_f = f(i) %
            Si Valeur(f(i−1)) ∈ {asserté, requis} alors
                utiliser(N_f(i))
            Sinon
                Cas de
                    nombre−établisseurs−possibles(n_f) = 0 :
                        échappement(inconsistance)
                    nombre−établisseurs−possibles(n_f) = 1 :
                        utiliser(établisseur−possible(n_f))
                    autres cas :
                        Valeur(n_f) ← requis ;
                        liste−buts ← liste−buts ∪ {n_f} ;
                        Pour tout {n_f, n_h} ∈ MutexF(GP) faire
                            nier(n_h, Vrai) ;
                        Pour tout (n_a, n_f) ∈ ArcsDel(GP) faire
                            exclure(n_a)
Fin
```

Algorithme 42 : requérir (DPP)

La fonction nier : elle est modifiée par l'ajout de règles de propagation inspirées de celles de la fonction **exclure**. Les différents cas de propagation sont conditionnés, comme dans la fonction **exclure**, par le nombre d'actions qui peuvent retirer le fluent que l'on nie :

- Lorsqu'il ne reste qu'une seule action qui retire le fluent considéré et que ce dernier a une valeur positive au niveau précédent, alors il faut forcément utiliser cette action. Il n'y a donc pas besoin d'effectuer les propagations ; il suffit d'utiliser l'action, qui va effectuer elle-même les propagations nécessaires. Ce cas n'avait pas pu être détecté par la fonction **exclure** lors de l'exclusion des autres actions qui retirent le fluent (s'il y en avait), si le fluent n'avait pas encore une valeur positive au niveau précédent. La règle correspondante n'avait donc pas été déclenchée.
- Lorsqu'il reste plusieurs actions qui retirent le fluent considéré, ou bien s'il n'en reste qu'une et que le fluent n'a pas une valeur positive au niveau précédent, on effectue les mêmes calculs que l'ancienne fonction **nier**. En effet dans ces cas, on ne sait pas si le fluent n'existait pas au niveau précédent, auquel cas il n'y a pas besoin de le retirer, ou bien s'il était présent et que l'on doit le retirer, ou encore si une action le retire bel et bien. Dans l'ancienne version de DPPlan, il aurait fallu faire appel à **requérir−faux** dans le cas où aucune action ne le retire (pour l'instant), et ensuite on aurait cherché une action qui le retire effectivement, le cas où le fluent était nié au niveau précédent n'étant pas bien pris en compte.
- Si toutes les actions qui retirent le fluent considéré ont la valeur *faux*, alors ce dernier doit posséder la valeur *nié* au niveau précédent. On fait donc appel à la fonction **nier** sur le fluent au niveau précédent, en demandant à effectuer toutes les propagations possibles.

```
Fonction nier(n_f, toutes−propagations)
Début
    Cas de
        Valeur(n_f) ∈ {asserté, requis} :
            échappement(inconsistance)
        Valeur(n_f) = indéfini :
            % avec n_f = f(i) %
            Valeur(n_f) ← nié ;
            Si nombre−destructeurs−possibles(n_f) = 1 et Valeur(f(i−1)) ∈ {asserté, requis} alors
                utiliser(destructeur−possible(n_f))
            Sinon
                Si toutes−propagations alors
                    Pour tout (n_a, n_f) ∈ ArcsAdd(GP) faire
                        exclure(n_a) ;
                Pour tout (n_f, n_a) ∈ ArcsPrec(GP) faire
                    exclure(n_a) ;
                Si nombre−destructeurs−possibles(n_f) = 0 alors
                    nier(f(i−1), Vrai)
Fin
```

Algorithme 43 : nier (DPP)

La fonction exclure : elle est modifiée pour la prise en compte de la disparition de la fonction **requérir−faux**, ce qui permet d'affiner certaines règles de propagation. Pour les ajouts de l'action exclue :

- Si toutes les actions qui ajoutent un même fluent ont la valeur *faux*, ce dernier ne peut pas avoir une valeur positive. On ne fait plus appel à la fonction **requérir−faux** afin de trouver une action qui retire ce fluent, car le fluent n'existait peut-être pas au niveau précédent. On fait donc appel à la fonction **nier**, qui va prendre en compte cette possibilité comme on l'a vu pour **DPPMin2**. Il n'est pas nécessaire de demander toutes les propagations de la fonction **nier**, puisque toutes les actions qui produisent le fluent sont exclues (booléen à Faux).
- S'il ne reste qu'une seule action qui peut produire un fluent et ce dernier a la valeur *requis*, alors il faut utiliser cette action.

Et pour les retraits de l'action exclue :

- Si toutes les actions qui retirent un même fluent ont la valeur *faux*, alors :
  - si le fluent a la valeur *asserté* ou *requis* au niveau précédent, il doit maintenant prendre la valeur *asserté*. Dans la version originale de DPPlan, on s'assurait en outre qu'il avait la valeur *indéfini*, avant

de faire appel à la fonction **requérir**. Or, si un fluent est présent dans un état, absent de l'état suivant et si aucune action ne peut le retirer, il est bien évident qu'il y a une contradiction : on doit produire un échappement. De plus, si un fluent est présent dans un état et qu'aucune action ne peut le retirer, alors il y a au moins une action qui le produit au niveau suivant, que l'on peut utiliser immédiatement : son no-op. Au lieu de faire appel à la fonction **requérir**, on utilise donc le no-op, ce qui a pour conséquence de faire échouer la recherche dans le cas où le fluent considéré a la valeur *nié*.

- si le fluent a la valeur *nié* (dans la version originale, on regardait s'il avait la valeur *requis−faux*), alors au niveau précédent il doit avoir aussi la valeur *nié* puisqu'aucune action ne pourrait le retirer s'il avait la valeur *asserté* ou *requis*. On fait donc appel à la fonction **nier** sur le fluent au niveau précédent, en demandant d'effectuer toutes les propagations possibles.

- Si une seule des actions qui retirent un même fluent a la valeur *indéfini* (toutes les autres ayant la valeur *faux*), si le fluent a la valeur *nié*, et si au niveau précédent il a la valeur *asserté* ou *requis*, alors il faut utiliser cette dernière action puisque c'est la seule qui peut encore retirer le fluent.

```
Fonction exclure(n_a)
Début
    Cas de
        Valeur(n_a) = vrai :
            échappement(inconsistance)
        Valeur(n_a) = indéfini :
            Valeur(n_a) ← faux ;
            Pour tout (n_a, n_f) ∈ ArcsAdd(GP) faire
                Cas de
                    nombre−établisseurs−possibles(n_f) = 0 :
                        nier(n_f, Faux)
                    nombre−établisseurs−possibles(n_f) = 1 et Valeur(n_f) = requis :
                        utiliser(établisseur−possible(n_f))
                FinCas ;
            Pour tout (n_a, n_f) ∈ ArcsDel(GP) faire
                % avec n_f = f(i) %
                Cas de
                    nombre−destructeurs−possibles(n_f) = 0 :
                        Cas de
                            Valeur(f(i−1)) ∈ {asserté, requis} :
                                utiliser(N_f(i))
                            Valeur(n_f) = nié :
                                nier(f(i−1), Vrai)
                        FinCas
                    nombre−destructeurs−possibles(n_f) = 1 et Valeur(n_f) = nié
                    et Valeur(f(i−1)) ∈ {asserté, requis} :
                        utiliser(destructeur−possible(n_f))
Fin
```

Algorithme 44 : exclure (DPP)

4.3. Exemple

Nous allons maintenant développer le fonctionnement de ce nouvel algorithme sur l'exemple de la section 2.2.

initialiser()
Valeur($a(0)$) ← *indéfini*
Valeur($a(1)$) ← *indéfini*
Valeur($b(1)$) ← *indéfini*

```
Valeur(c(1)) ← indéfini
Valeur(A(1)) ← indéfini
Valeur(B(1)) ← indéfini
asserter(a(0))                                  % a ∈ I
    Valeur(a(0)) ← asserté
requérir(b(1))                                  % b ∈ G
    utiliser(A(1))                              % un seul établisseur possible pour b(1)
        Valeur(A(1)) ← vrai
        requérir(a(0)) → OK                     % a ∈ Prec(A)
        asserter(b(1))                          % b ∈ Add(A)
            Valeur(b(1)) ← asserté
        exclure(B(1))                           % {A(1), B(1)} ∈ MutexA(GP)
            Valeur(B(1)) ← faux
            nier(c(1), F)                       % il n'y a plus d'établisseur possible pour c(1)
                Valeur(c(1)) ← nié
            utiliser(N_a(1)))                   % il n'y a plus de destructeur possible pour a(1)
                                                % et a(0) est asserté
                Valeur(N_a(1)) ← vrai
                requérir(a(0)) → OK             % a ∈ Prec(N_a)
                asserter(a(1))                  % a ∈ Add(N_a)
                    Valeur(a(1)) ← asserté
                exclure(B(1)) → OK              % {B(1), N_a(1)} ∈ MutexA(GP)
Fin de l'appel à initialiser.

recherche()
    échappement(succès)                         % liste−buts = ∅
Fin de l'appel à recherche.
```

Cette fois-ci encore, l'algorithme s'achève sur un succès de la recherche. Le plan retourné est donc $\langle\{A, N\_a\}\rangle$ qui peut être simplifié en $\langle A\rangle$. On peut constater que l'initialisation des valeurs des noeuds du graphe par la fonction **initialiser** suffit pour trouver la solution, grâce aux règles de propagation implantées dans les fonctions **requérir** et **nier**.

5. Utilisation des relations d'autorisation dans DPPlan

Les algorithmes d'extraction de solution basés sur DPPlan offrent un cadre tout à fait approprié pour l'utilisation des relations d'autorisation, contrairement à l'utilisation des prouveurs SAT. Comme nous l'avons étudié dans la partie consacrée à Graphplan, cette utilisation se fait en deux temps :

- Pendant la construction du graphe : les exclusions mutuelles entre noeuds d'actions et de fluents sont calculées à l'aide d'une relation binaire comme l'indépendance (relation #, déf. 18 page 67), l'autorisation forte (relation $R_{\angle}$, déf. 27 page 74) ou l'autorisation faible (relation $R_{\underline{\angle}}$, déf. 28 page 74). Comme les algorithmes basés sur DPPlan ne concernent que l'extraction de la solution, la construction du graphe peut être effectuée avec n'importe quelle relation binaire R, pourvu qu'elle soit Φ_R−compatible.
- Pendant l'extraction de la solution : il faut vérifier que les ensembles d'actions retenus pour faire partie du plan-solution vérifient la contrainte Φ_R, R étant la relation utilisée lors de la construction du graphe. Dans LCGP, cette vérification se faisait en deux temps : d'abord en choisissant un ensemble d'actions ne comportant pas d'exclusions mutuelles, ensuite en vérifiant la contrainte d'autorisation. Dans DPPlan, nous allons vérifier cette contrainte à chaque fois que l'on utilise une action.

Mais la difficulté dans DPPlan, au vu de la complexité des propagations effectuées, va être de s'assurer que l'on n'exclut pas de solution potentiellement valable. En effet, DPPlan a été conçu pour l'utilisation de la relation d'indépendance ; on a donc inclus le plus possible de propagations par rapport à cette relation. Mais

comme nous allons le voir, certaines d'entre elles rejettent des solutions acceptables pour la relation d'autorisation faible ; la contrainte d'autorisation forte ne nous posera aucun problème.

5.1. Vérification des contraintes d'autorisation

A chaque fois qu'une action n_a prend la valeur *vrai*, nous allons vérifier que l'ensemble des actions du même niveau que n_a qui ont la valeur *vrai* vérifient la contrainte d'autorisation choisie (forte ou faible). Pour toutes les versions de DPPlan que nous avons étudiées, le seul endroit où une action prend la valeur *vrai* est dans la fonction **utiliser**. Juste après avoir changé la valeur de l'action, nous appelons donc la fonction **stratifier** (cf. section III.5.2.2, page 75) pour vérifier que la contrainte d'autorisation Φ est vérifiée. Cette contrainte paramètre toutes les fonctions des différentes versions de DPPlan.

Pour **DPPMin** et **DPPMin2** :

```
Fonction utiliser(n_a, Φ)
Début
    Valeur(n_a) ← vrai ;
    % avec n_a = a(i) et Q = {a | Valeur(a(i)) = vrai} %
    Si (Φ = Φ_∠ et stratifier(Q, ∠) = échec) ou (Φ = Φ_≦ et stratifier(Q, ≦) = échec) alors
        retourner Faux ;
    (...)
Fin
```

Algorithme 45 : utiliser (LCDPPMin et LCDPPMin2)

Pour **DPPC** (version corrigée de DPPlan, cf. section 2.3 page 127) et **DPP** (version modifiée de DPPlan, cf. section 4 page 140) :

```
Fonction utiliser(n_a, Φ)
Début
    Cas de
        Valeur(n_a) = faux
            échappement(inconsistance)
        Valeur(n_a) = indéfini
            Valeur(n_a) ← vrai ;
            % avec n_a = a(i) et Q = {a | Valeur(a(i)) = vrai} %
            Si (Φ = Φ_∠ et stratifier(Q, ∠) = échec) ou (Φ = Φ_≦ et stratifier(Q, ≦) = échec) alors
                échappement(inconsistance) ;
            (...)
Fin
```

Algorithme 46 : utiliser (LCDPP et LCDPPC)

5.2. Modifications pour l'utilisation de la relation d'autorisation forte

La différence entre la relation d'autorisation forte et la relation d'indépendance est qu'une action peut retirer une précondition d'une autre action à un même niveau. Dans toutes les versions de DPPlan que nous avons étudiées, le seul endroit où on exclut une action pour cette raison est dans la procédure **utiliser**, lorsque l'on exclut les actions mutuellement exclusives avec l'action utilisée. Or, ces exclusions mutuelles sont déterminées lors de la construction du graphe ; elles prennent donc en compte la relation d'autorisation forte. Il n'y a donc aucune modification à apporter aux fonctions de propagation.

5.3. Modifications pour l'utilisation de la relation d'autorisation faible

La différence entre la relation d'autorisation faible et la relation d'autorisation forte est qu'une action peut retirer un ajout d'une autre action à un même niveau. Ceci a une conséquence fondamentale : un fluent devrait pouvoir avoir à la fois la valeur *asserté* (ou *requis*) et la valeur *nié* ! Bien sûr, cette solution n'est pas viable. En effet, un fluent doit avoir la valeur *nié* lorsque toutes les actions qui le produisent sont exclues. Dans ce cas, il ne faut surtout pas accepter que le noeud prenne aussi la valeur *asserté* ou *requis*, puisqu'on ne trouvera aucune action qui le produise.

Il nous faut donc trouver une solution plus subtile, en nous posant la question : à quel moment un fluent doit vraiment prendre la valeur *nié* ? La réponse est la suivante : c'est quand plus aucune action ne peut produire ce fluent et quand il ne peut pas coexister avec un autre fluent au même niveau. Nous allons donc éliminer toutes les propagations qui ne respectent pas ces contraintes. En effet, il suffit qu'il reste les propagations minimales que nous avons étudiées dans **DPPMin** et **DPPMin2** pour que l'algorithme reste complet et fournisse des plans-solutions corrects.

Pour **DPPMin** et **DPPMin2** : il n'y a aucune modification à faire. Dans **DPPMin**, les fluents n'ont pas de valeur. Les propagations étant réduites au strict minimum, le problème ne se pose pas. Dans **DPPMin2**, le seul cas où un fluent prend la valeur *nié* est bien un des deux cas obligatoires.

Pour **DPPC**, plusieurs modifications sont nécessaires :

- Dans la fonction **utiliser**, il ne faut plus nier les retraits de l'action utilisée. Nous avons vu que cette propagation est de toute façon redondante avec l'exclusion des actions mutuellement exclusives avec l'action utilisée ; la perte de cette règle ne doit donc pas provoquer une perte d'efficacité. Cette fonction devient identique à la fonction **utiliser** de DPP.
- Dans la fonction **requérir**, il ne faut plus exclure les actions qui retirent le fluent que l'on requiert. En effet, un fluent peut maintenant être retiré par une action et néanmoins produit par une autre.

La fonction **requérir** devient donc :

```
Fonction requérir(n_f, Φ)
Début
    Cas de
        Valeur(n_f) ∈ {nié, requis−faux} :
            échappement(inconsistance)
        Valeur(n_f) = indéfini :
            Valeur(n_f) ← requis ;
            liste−buts ← liste−buts ∪ {n_f} ;
            Pour tout {n_f, n_h} ∈ MutexF(GP) faire
                requérir−faux(n_h, Φ) ;
Fin
```

Algorithme 47 : requérir (LCDPPC)

Pour **DPP**, plusieurs modifications sont là aussi nécessaires :

- Dans la fonction **requérir**, nous devons faire la même modification que pour **DPPC** : il ne faut plus exclure les actions qui retirent le fluent que l'on requiert.
- Toujours dans la fonction **requérir**, on ne peut plus utiliser le no-op correspondant au fluent considéré, dans le cas où ce dernier a une valeur positive au niveau précédent. En effet, un fluent peut très bien avoir une valeur positive à un niveau, être retiré par une action, puis ajouté par une action que la première autorise faiblement. Il n'y a aucune raison (et c'est même impossible dans ce cas) pour que cette action soit le no-op correspondant au fluent considéré. En effet, si une action retire le fluent, alors elle interdit le no-op puisque le fluent en question est la précondition du no-op.

La fonction **requérir** devient donc :

```
Fonction requérir(n_f, Φ)
Début
    Cas de
        Valeur(n_f) = nié :
            échappement(inconsistance)
        Valeur(n_f) = indéfini :
            Cas de
                nombre−établisseurs−possibles(n_f) = 0 :
                    échappement(inconsistance)
                nombre−établisseurs−possibles(n_f) = 1 :
                    utiliser(établisseur−possible(n_f), Φ)
                autres cas :
                    Valeur(n_f) ← requis ;
                    liste−buts ← liste−buts ∪ {n_f} ;
                    Pour tout {n_f, n_h} ∈ MutexF(GP) faire
                        nier(n_h, Vrai, Φ)
Fin
```

Algorithme 48 : requérir (LCDPP)

6. Évaluation expérimentale

6.1. Conditions expérimentales

Nous allons maintenant présenter plusieurs séries d'expérimentations, effectuées avec les planificateurs suivants :

- **DPPC**, **LCDPPC :** versions corrigées de DPPlan, utilisant respectivement la relation d'indépendance (cf. section 2.3) et la relation d'autorisation faible (cf. section 5).
- **DPP**, **LCDPP :** versions modifiées de DPPlan, utilisant respectivement la relation d'indépendance (cf. section 4) et la relation d'autorisation faible (cf. section 5).
- **LCGPO :** Graphplan avec la relation d'autorisation faible et la procédure d'extraction originelle (cf. section III.4). On ne conserve que les ensembles de sous-buts insolubles à chaque niveau minimaux pour l'inclusion.
- **LCGP :** Graphplan avec la relation d'autorisation faible et la procédure d'extraction utilisant les techniques EBL/DDB de [Kambhampati 1999] [Kambhampati 2000].

Ces planificateurs sont implémentés en Allegro Common Lisp 5.0 et partagent la majeure partie de leur code. La version de LCGP que nous employons maintenant est légèrement améliorée par rapport à celle utilisée pour les expérimentations de la section III.6, notamment par un calcul un peu plus rapide des exclusions mutuelles et une consommation en mémoire vive plus réduite. L'heuristique employée par ces planificateurs est celle décrite dans la section III.6.3 : les fluents préférés sont ceux qui ont le niveau d'apparition le plus élevé dans le graphe de planification, et les actions préférées sont celles qui ont le niveau d'apparition le plus bas.

Dans le Monde des cubes et les domaines Mprime et Mystery, nous effectuons aussi la comparaison avec le planificateur FF[8] [Hoffmann 2000] [Hoffman et Nebel 2000], qui s'est distingué lors de la deuxième compétition de planificateurs d'AIPS−2000 par des performances exceptionnelles par rapport à la plupart de ses concurrents. Il s'agit d'un planificateur fonctionnant en chaînage avant dans les espaces d'états, utilisant une heuristique extraite automatiquement de la description du domaine, basée sur la résolution du problème relaxé par la suppression des retraits des opérateurs (cf. section II.4). Il est implémenté en C et bénéficie de plusieurs améliorations, comme une heuristique d'ordonnancement des sous-buts [Koehler et Hoffmann 2000] et une procédure efficace d'instanciation des opérateurs [Koehler et Hoffmann 1999]. Nous ne rapportons pas

[8] FF est téléchargeable à l'adresse `http://www.informatik.uni-freiburg.de/~hoffmann/ff.html`

les résultats de FF dans les domaines Logistics, Ferry et Gripper car il se montre très largement supérieur à tous nos planificateurs (quelques centièmes ou dixièmes de secondes pour la résolution des problèmes les plus difficiles). Nous montrons donc les résultats de FF dans les domaines où il présente quelques difficultés, ce qui peut justifier l'utilisation des techniques que nous avons décrites dans ce mémoire.

Tous les tests ont été réalisés sur un Pentium−II 450Mhz avec 512Mo de RAM, fonctionnant sous Debian GNU/Linux 2.1. Le temps maximal de résolution est fixé à 1 heure.

6.2. Comparaison entre versions corrigées et améliorées de DPPlan

Nous comparons d'abord DPPC, DPP, LCDPPC et LCDPP sur plusieurs problèmes issus des domaines Ferry, Gripper, Monde des cubes version Prodigy et Logistics (cf. Table 11). On constate que pour ces problèmes, les versions améliorées de DPPlan avec relation d'indépendance ou d'autorisation sont plus rapides que les versions corrigées, d'un facteur variant entre 1 et 3,4. Ainsi, les simplifications et améliorations que nous avons apportées sur l'algorithme de DPPlan s'en trouvent justifiées d'un point de vue pratique.

Problèmes	Temps CPU (sec.)				Ratio DPPC/ DPP	Ratio LCDPPC/ LCDPP
	DPPC	DPP	LCDPPC	LCDPP		
Ferry4	7,70	3,18	0,13	0,09	2,42	1,49
Ferry5	475,33	142,34	0,56	0,38	3,34	1,48
Gripper6	779,64	285,57	0,57	0,42	2,73	1,37
Bw−large−a	2,95	2,22	3,19	2,35	1,33	1,36
Bw−large−b	43,83	12,75	43,46	13,85	3,44	3,14
Log018	306,09	148,93	5,54	4,73	2,06	1,17
Log021	148,95	106,18	3,72	3,71	1,40	1,00
Log030	225,02	121,35	2,59	2,56	1,85	1,01

Table 9 : Comparaison entre versions corrigées et améliorées de DPPlan

6.3. Comparaison dans le domaine Logistics

Nous comparons dans ce domaine les planificateurs DPP, LCDPP, LCGPO et LCGP (cf. Table 10) sur les 30 problèmes de la distribution BLACKBOX [Kautz et Selman 1999].

Nous pouvons faire les remarques suivantes :

- Comme on pouvait s'y attendre, DPP résout le moins de problèmes : c'est le seul à utiliser la relation d'indépendance pour la construction du graphe et l'extraction de la solution. Il résout 17 problèmes avec un temps de calcul moyen de 298 secondes, ce que l'on peut comparer avec la version originelle de Graphplan avec extraction EBL/DDB (GP, cf. Table 2 page 89) qui résolvait 18 problèmes avec un temps de calcul moyen de 5325 secondes. Ce dernier disposait néanmoins de 24 heures de calcul, et on voit que DPP est généralement plus rapide que GP sur les problèmes résolus par les deux planificateurs.
- LCDPP résout 29 problèmes, et est environ 118 fois plus rapide que DPP sur les 17 problèmes résolus par les deux planificateurs. Ceci démontre encore une fois l'intérêt de l'utilisation de la relation d'autorisation dans ce domaine.
- LCGP est le plus rapide des trois planificateurs utilisant la relation d'autorisation faible : il résout les 30 problèmes avec un temps de calcul moyen de 3,88 secondes. Mais sa procédure d'extraction effectue du retour arrière intelligent (techniques EBL/DDB), alors que le retour arrière de LCDPP est géré de manière chronologique. C'est pourquoi nous avons comparé LCDPP avec LCGPO qui effectue aussi du retour arrière chronologique. On voit donc que LCDPP résout 29 problèmes avec un temps de calcul moyen de 17 secondes alors que LCGPO n'en résout que 21 avec un temps de calcul moyen de 170 secondes. Il serait donc intéressant d'étudier l'utilisation du retour arrière intelligent dans LCDPP, qui a fait ses preuves pour les prouveurs SAT (cf. RELSAT [Bayardo et Schrag 1997] et GRASP [Marques Silva et Sakallah 1996]).
- La qualité de la solution en nombre d'actions est quasiment identique pour tous les planificateurs : environ 47 actions en moyenne sur les problème résolus par tous les planificateurs.

Problèmes	Temps CPU (sec.)				Ratio DPP/	Actions				Niveaux		
										DPP	LCDPP	
	DPP	LCDPP	LCGPO	LCGP	LCDPP	DPP	LCDPP	LCGPO	LCGP		(+)	(++)
log.easy	0,35	0,31	0,27	0,27	1,11	25	25	25	25	9	9	6
rocket.a	1,71	0,35	0,47	0,33	4,85	30	28	26	28	7	7	4
rocket.b	2,56	0,37	0,43	0,32	6,98	26	26	26	26	7	7	4
log.a	2,09	0,93	165,50	0,77	2,26	54	54	54	54	11	11	7
log.b	≥3 600	2,23	10,47	1,14	≥1 618		45	45	45	≥ 12	13	8
log.c	≥3 600	2,72	≥3 600	2,26	≥1 323		55	53	53	≥ 12	13	8
log.d	≥3 600	3,66	968,11	2,50	≥ 985		73	72	72	≥ 13	15	9
log.d3	≥3 600	4,67	2,97	2,68	≥ 772		72	72	72	≥ 12	13	8
log.d1	≥3 600	5,87	≥3 600	3,16	≥ 613		68	68	68	≥ 15	17	10
log010	5,59	2,74	2,17	2,35	2,04	43	41	41	41	10	12	7
log011	879,98	1,64	83,22	1,47	536,90	49	50	50	50	11	12	7
log012	1,63	1,03	3,47	0,90	1,59	39	38	38	38	8	8	5
log013	2 826,34	2,80	2,58	2,40	1 008,68	67	66	66	66	11	11	7
log014	6,06	5,10	6,18	2,96	1,19	71	75	75	75	10	11	7
log015	≥3 600	4,71	≥3 600	2,57	≥ 764		63	64	63	≥ 11	13	7
log016	≥3 600	4,82	101,31	1,97	≥ 747		40	40	40	≥ 14	16	9
log017	≥3 600	381,35	189,79	10,65	≥ 9		43	43	43	≥ 14	17	10
log018	148,93	4,73	3,67	3,95	31,49	50	52	49	50	11	11	7
log019	15,09	2,23	15,12	1,92	6,77	49	46	48	48	11	11	7
log020	≥3 600	≥3 600	≥3 600	42,36	1,00				87	≥ 12	≥ 9	≥ 9
log021	106,18	3,71	95,41	2,87	28,64	65	64	64	63	11	13	7
log022	≥3 600	5,78	≥3 600	2,82	≥ 623		75	75	75	≥ 13	15	9
log023	≥3 600	4,31	≥3 600	2,81	≥ 835		61	61	61	≥ 13	13	8
log024	634,11	2,62	27,24	2,30	242,49	64	67	67	67	12	13	8
log025	54,25	2,64	≥3 600	2,27	20,52	57	56	57	57	12	14	8
log026	9,21	2,63	1 878,35	2,19	3,50	50	52	51	50	12	13	8
log027	≥3 600	2,82	≥3 600	2,40	≥1 277		71	71	73	≥ 13	13	8
log028	≥3 600	25,93	≥3 600	5,71	≥ 139		78	77	78	≥ 13	14	9
log029	6,88	6,48	3,62	3,99	1,06	46	48	46	46	10	10	7
log030	121,35	2,56	2,00	0,30	47,35	52	52	52	52	13	13	8
Moy. (*)	298,00	2,51	143,11	1,83	118,55	46,93	46,93	46,57	46,64	10,14	10,71	6,57
Moy. (**)	283,66	16,96	169,64	3,82	16,73	46,93	47,13	54,34	54,45	10,14	12,34	7,48
Moy. (***	≥1 720,74	≥136,39	≥1 198,74	3,82	–	–	–	–	–	≥11,43	≥12,23	≥7,53

(+) nombre de niveaux du plan après la transformation par **réordonnancer−plan**
(++) nombre de niveaux du plan avant la transformation par **réordonnancer−plan**
(*) moyenne sur les problèmes résolus par tous les planificateurs.
(**) moyenne sur les problèmes résolus par le planificateur correspondant à la colonne (cellules blanches).
(***) moyenne sur les 30 problèmes.
cellules grises : échec de la résolution du problème correspondant.

Table 10 : Comparaison dans le domaine Logistics

6.4. Comparaison dans les domaines du Ferry et du Gripper

Nous comparons maintenant DPP, LCDPP, LCGPO et LCGP dans les domaines du Ferry et du Gripper. Dans ces domaines, LCDPP est beaucoup plus rapide que DPP mais est largement surpassé par LCGPO et LCGP (cf. tables 9 et 12). L'utilisation du retour arrière intelligent pour LCGP n'apporte pas un gain de temps aussi bon dans le domaine du Ferry que dans le domaine Logistics : au mieux un facteur 2. Il est un peu plus utile dans le domaine du Gripper. Par contre, LCGPO tire certainement parti de la mémoization des ensembles de sous-buts insolubles, ce qui n'est pas effectué dans DPP et LCDPP. En effet dans ces domaines très combinatoires, la plupart des objets ont un rôle symétrique ; la recherche effectuée à un niveau est donc très semblable à celle effectuée aux niveaux inférieurs. Il serait donc intéressant d'étudier l'utilisation de techniques de mémoization dans DPP et LCDPP. On peut remarquer que le rapport entre le temps de calcul de DPP et celui de LCDPP croît de manière beaucoup plus importante qu'entre GP et LCGP (cf. tables 3 et 4 page 90), où il se stabilisait aux environs de 6 dans le domaine du Ferry et augmentait faiblement dans le domaine du Gripper.

Sous-buts	Temps CPU (sec.)				Ratio DPP/ LCDPP	Actions		Niveaux		
								DPP	LC*	
	DPP	LCDPP	LCGPO	LCGP		DPP	LC*		(+)	(++)
1	0,02	0,02	0,02	0,01	1,10	3	3	3	3	2
2	0,03	0,03	0,02	0,02	1,04	7	7	7	7	4
3	0,13	0,04	0,03	0,03	3,17	11	11	11	11	6
4	3,18	0,09	0,05	0,05	35,36	15	15	15	15	8
5	142,34	0,38	0,15	0,13	371,63	19	19	19	19	10
6	≥3 600	2,68	0,61	0,48	≥1 345		23	≥ 21	23	12
7	≥3 600	23,93	2,55	1,68	≥ 150		27	≥ 21	27	14
8	≥3 600	235,98	9,55	5,42	≥ 15		31	≥ 21	31	16
9	≥3 600	2 739,43	33,33	15,49	≥ 1		35	≥ 21	35	18

(+) nombre de niveaux du plan après la transformation par **réordonnancer−plan**
(++) nombre de niveaux du plan avant la transformation par **réordonnancer−plan**
cellules grises : échec de la résolution du problème correspondant.

Table 11 : Comparaison dans le domaine du Ferry

Sous-buts	Temps CPU (sec.)				Ratio DPP/ LCDPP	Actions		Niveaux		
								DPP	LC*	
	DPP	LCDPP	LCGPO	LCGP		DPP	LC*		(+)	(++)
2	0,03	0,03	0,02	0,02	0,94	3	3	3	3	2
4	0,26	0,07	0,06	0,06	3,75	7	7	7	7	4
6	285,57	0,42	1,08	0,55	676,69	11	11	11	11	6
8	≥3 600	22,55	44,63	13,05	≥159,66		15	≥ 12	15	8
10	≥3 600	2 800,90	900,24	163,94	≥1,29		19	≥ 12	19	10
12	≥3 600	≥3 600	≥3 600	1 458,46			23	≥ 12	23	12

(+) nombre de niveaux du plan après la transformation par **réordonnancer−plan**
(++) nombre de niveaux du plan avant la transformation par **réordonnancer−plan**
cellules grises : échec de la résolution du problème correspondant.

Table 12 : Comparaison dans le domaine du Gripper

6.5. Comparaison dans le Monde des cubes

6.5.1. Version Prodigy

Nous comparons maintenant DPP, LCDPP, LCGP et FF dans le Monde des cubes avec un bras manipulateur (cf. Table 15). Les problèmes employés sont les 28 premiers problèmes de ce domaine utilisés pendant la compétition d'AIPS−2000.

Comme pour la comparaison entre GP et LCGP, l'utilisation de la relation d'autorisation faible n'apporte rien dans ce domaine où aucun parallélisme ni relaxation de ce dernier n'est possible. Ainsi, les temps de calcul de DPP et LCDPP sont très proches ; les faibles différences enregistrées s'expliquent par l'environnement Common Lisp utilisé, qui a parfois tendance à ne pas être tout à fait "déterministe", notamment dans la gestion des tables de hashage.

Encore une fois, LCGP est plus rapide que DPP et LCDPP et résout plus de problèmes : 25 problèmes sont résolus par LCGP contre 21 pour DPP et LCDPP. En temps de calcul moyen sur les problèmes résolus par les deux planificateurs, LCGP met 6,20 secondes contre 77 secondes pour DPP. Quant à FF, il présente de très bonnes performances ; il s'est d'ailleurs montré capable de résoudre des problèmes bien plus difficiles dans ce domaine lors de la compétition d'AIPS−2000. Il fournit de plus des solutions de très bonne qualité en nombre d'actions, par rapport aux plans optimaux fournis dans ce domaine par LCGP. On peut toutefois noter que deux problèmes résolus facilement par nos trois planificateurs ne sont pas résolus par FF à cause d'un dépassement (rapide) de la capacité de mémoire vive.

Problèmes	Temps CPU (sec.)				Ratio DPP/	Actions				Niveaux		
										DPP	LCDPP	
	DPP	LCDPP	LCGP	FF	LCDPP	DPP	LCDPP	LCGP	FF		(+)	(++)
04−0	0,13	0,14	0,10	0,01	0,93	6	6	6	6	6	6	6
04−1	0,14	0,14	0,11	0,00	0,98	10	10	10	10	10	10	10
04−2	0,12	0,12	0,10	0,00	0,95	6	6	6	6	6	6	6
05−0	0,35	0,36	0,24	0,01	0,97	12	12	12	12	12	12	12
05−1	0,30	0,31	0,23	0,00	0,95	10	10	10	10	10	10	10
05−2	0,33	0,35	0,24	0,01	0,95	16	16	16	16	16	16	16
06−0	0,62	0,66	0,45	0,01	0,94	12	12	12	20	12	12	12
06−1	0,52	0,55	0,36	0,00	0,95	10	10	10	10	10	10	10
06−2	0,69	0,73	0,44	0,01	0,95	20	20	20	20	20	20	20
07−0	0,92	0,98	0,69	0,01	0,93	20	20	20	20	20	20	20
07−1	102,23	107,11	3,58	0,02	0,95	22	22	22	22	22	22	22
07−2	7,95	8,24	1,14	0,10	0,97	20	20	20	22	20	20	20
08−0	21,06	16,43	2,53	0,01	1,28	18	18	18	18	18	18	18
08−1	18,97	20,87	3,18	≥18,23	0,91	20	20	20		20	20	20
08−2	1,51	1,64	1,05	0,08	0,92	16	16	16	16	16	16	16
09−0	36,01	47,75	5,76	≥17,75	0,75	30	30	30		30	30	30
09−1	12,05	12,20	2,79	0,04	0,99	28	28	28	28	28	28	28
09−2	7,39	7,32	1,72	0,01	1,01	26	26	26	26	26	26	26
10−0	425,71	383,64	3,12	0,02	1,11	34	34	34	34	34	34	34
10−1	≥3 600	≥3 600	505,46	1,78				32	38	≥ 31		≥ 31
10−2	271,83	348,76	21,14	0,08	0,78	34	34	34	34	34	34	34
11−0	≥3 600	≥3 600	614,94	0,02	1,00			32	34	≥ 30		≥ 30
11−1	≥3 600	≥3 600	≥3 600	≥16,16						≥ 29		≥ 29
11−2	609,65	649,94	77,85	0,01	0,94	34	34	34	34	34	34	34
12−0	≥3 600	≥3 600	897,91	0,38				34	44	≥ 31		≥ 31
12−1	≥3 600	≥3 600	36,05	0,02				34	34	≥ 34		≥ 34
13−0	≥3 600	≥3 600	≥3 600	0,03					42	≥ 40		≥ 40
13−1	≥3 600	≥3 600	≥3 600	27,38					46	≥ 42		≥ 42
Moy. (*)	77,03	81,03	6,20	0,02	0,95	18,63	18,63	18,63	19,16	18,63	18,63	18,63
Moy. (**)	72,31	76,58	87,25	1,20	0,94	19,24	19,24	21,44	24,08	19,24	19,24	19,24
Moy. (***)	≥954,25	≥957,45	≥463,61	2,94	–	–	–	–	–	≥22,89	≥19,24	≥22,89

(+) nombre de niveaux du plan après la transformation par **réordonnancer−plan**
(++) nombre de niveaux du plan avant la transformation par **réordonnancer−plan**
(*) moyenne sur les problèmes résolus par tous les planificateurs.
(**) moyenne sur les problèmes résolus par le planificateur correspondant à la colonne (cellules blanches).
(***) moyenne sur les 30 problèmes.
cellules grises : échec de la résolution du problème correspondant.

Table 13 : Comparaison dans le Monde des cubes (version Prodigy)

6.5.2. Version à trois opérateurs

Nous comparons maintenant DPP, LCDPP, LCGP et FF dans le Monde des cubes avec un bras manipulateur. Les problèmes employés sont les 50 premiers problèmes de ce domaine utilisés pendant la compétition d'AIPS−2000, avec la description des opérateurs telle qu'elle est faite dans l'introduction de cette thèse.

Dans ce domaine, DPP se montre très inférieur à LCDPP : il ne résout que 17 problèmes avec un temps de calcul moyen de 22 secondes, alors que ces mêmes problèmes sont résolus par LCDPP en un temps moyen de 0,62 secondes. Il s'agit aussi du seul domaine dans lequel LCDPP se montre clairement supérieur à LCGP, car il résout les 50 problèmes alors que LCGP n'en résout que 39. Il se produit dans ce domaine un phénomène très intéressant, que nous avions déjà mentionné lors de la comparaison entre GP et LCGP : les planificateurs utilisant la relation d'autorisation n'effectuent jamais aucun retour arrière dans leur arbre de recherche. Pour LCDPP, le fait qu'une action ne peut pas être utilisée dans la fonction **recherche** apparaît dans les propagations faites par la fonction **utiliser** ; le cas échéant, l'action est exclue et il n'y a jamais de retour arrière sur ce choix, de même que sur l'utilisation positive d'une action si la fonction **utiliser** ne produit pas d'échappement. Dans LCGP, il n'y a jamais de retour arrière dans le sens où on ne remonte jamais d'un niveau : une fois qu'on a trouvé un ensemble d'actions autorisé qui produit un ensemble de fluents, ce choix n'est jamais remis en question. Par contre, trouver un tel ensemble est difficile pour LCGP lorsqu'il y a beaucoup de fluents à un niveau, qui nécessitent donc beaucoup d'actions pour les produire : il y a donc des

retour arrières dans la fonction **rechercher−actions** à un même niveau. LCGP se montre donc incapable de résoudre les problèmes les plus difficiles.

Problèmes	Temps CPU (sec.)				Ratio DPP/	Actions				Niveaux		
										DPP	LCDPP	
	DPP	LCDPP	LCGP	FF	LCDPP	DPP	LCDPP	LCGP	FF		(+)	(++)
04−0	0,12	0,03	0,02	0,03	4,21	3	3	3	3	3	3	1
04−1	0,11	0,06	0,06	0,01	1,66	5	6	5	5	5	6	4
04−2	0,10	0,06	0,05	0,00	1,68	3	3	3	3	3	3	2
05−0	0,24	0,14	0,13	0,00	1,69	6	7	6	6	5	6	4
05−1	0,33	0,14	0,12	0,00	2,42	6	5	5	5	5	5	3
05−2	0,35	0,14	0,13	0,01	2,40	9	8	8	9	8	8	5
06−0	0,74	0,18	0,16	0,01	4,13	8	9	6	10	6	6	3
06−1	0,80	0,25	0,23	0,01	3,23	5	5	5	5	5	5	2
06−2	1,68	0,31	0,28	0,01	5,51	13	10	10	11	10	10	6
07−0	1,45	0,62	0,58	0,01	2,32	11	12	10	12	10	11	7
07−1	1,48	0,61	0,57	0,01	2,44	13	11	11	12	8	8	5
07−2	16,08	0,61	0,56	0,02	26,45	11	10	11	13	9	9	5
08−0	22,53	0,67	0,62	0,01	33,88	9	11	9	9	8	9	3
08−1	10,41	1,12	1,05	0,21	9,30	12	12	10	16	8	9	4
08−2	2,75	1,15	1,08	0,01	2,40	9	9	8	8	7	7	4
09−0	≥3 600	2,25	2,14	0,23	≥1 597		16	17	27	≥ 13	15	8
09−1	34,52	2,25	2,14	0,02	15,37	16	16	14	16	13	13	9
09−2	280,37	2,23	2,13	1,28	125,56	16	18	13	23	13	16	8
10−0	≥3 600	3,98	3,87	0,03	≥ 905		17	17	19	≥ 13	17	9
10−1	≥3 600	3,90	3,78	0,06	≥ 923		18	17	24	≥ 12	14	7
10−2		4,01	3,86	0,04			24	17	22		21	9
11−0		6,95	6,51	0,03			18	16	16		14	7
11−1		2,35	2,25	13,44			17	16	25		12	3
11−2		6,95	6,57	0,03			19	18	17		16	8
12−0		11,68	11,23	0,07			21	18	22		14	9
12−1		11,96	11,24	0,03			24	17	17		21	10
13−0		18,91	18,60	0,06			31	22	25		28	10
13−1		19,55	18,72	0,73			23	28	25		22	11
14−0		28,83	26,87	0,05			27	20	19		16	9
14−1		22,33	20,92	0,10			26	20	26		18	5
15−0		39,09	38,48	≥87,24			31	24			22	6
15−1		44,72	78,76	0,12			27	27	37		25	13
16−0		65,84	67,34	≥98,61			29	28			25	12
16−1		64,41	75,65	0,10			33	27	27		24	14
17−0		72,20	70,32	≥99,23			28	27			18	6
17−1		92,15	98,25	≥142,02			42	29			37	12
18−0		126,67	≥3 600	≥69,38			44				33	11
18−1		129,46	145,98	0,23			32	32	41		30	14
19−0		197,72	≥3 600	≥117,72			36				31	17
19−1		197,39	≥3 600	≥118,72			54				46	14
20−0		238,49	683,97	≥252,29			42	33			25	16
20−1		244,71	304,14	≥112,6			37	37			36	18
21−0		326,33	≥3 600	0,42			39		44		39	20
21−1		324,39	≥3 600	≥217,69			60				47	19
22−0		351,08	≥3 600	1,95			40		43		29	11
22−1		417,37	≥3 600	5,97			44		43		24	14
23−0		574,43	≥3 600	0,55			59		43		49	15
23−1		579,72	≥3 600	≥344,85			60				43	12
24−0		559,64	≥3 600	≥297,45			44				29	9
24−1		674,63	≥3 600	0,59			61		50		51	14
Moy. (*)	22,00	0,62	0,58	0,10	35,45	9,12	9,12	8,06	9,76	7,41	7,88	4,41
Moy. (**)	22,00	109,49	43,83	0,70	0,20	9,12	25,56	16,51	20,47	7,41	20,50	8,94
Moy. (***)	≥558,7	109,49	≥826,2	≥39,69	–	–	25,56	–	–	≥8,2	20,50	8,94

(+) nombre de niveaux du plan après la transformation par **réordonnancer−plan**

(++) nombre de niveaux du plan avant la transformation par **réordonnancer−plan**

(*) moyenne sur les problèmes résolus par tous les planificateurs.

(**) moyenne sur les problèmes résolus par le planificateur correspondant à la colonne (cellules blanches).

(***) moyenne sur les 30 problèmes.

cellules grises : échec de la résolution du problème correspondant.

Table 14 : Comparaison dans le Monde des cubes (version à trois opérateurs)

Quant à FF, il résout 38 problèmes avec un temps de calcul moyen de 0,7 secondes : encore une fois, quand il trouve une solution, il la trouve très rapidement. LCDPP résout plus de problèmes, mais plus lentement : c'est la construction du graphe de planification qui lui prend le plus temps (le temps de recherche sur ces

problèmes ne dépasse jamais 10 secondes pour LCDPP). Par contre, FF résout des problèmes plus difficiles que ceux présents dans ce tableau, dans lesquels LCDPP échoue par manque de mémoire vive. Par rapport à la qualité de la solution, les deux planificateurs sont à peu prés équivalents : en moyenne, les plans des 38 problèmes que FF résout contiennent 20,47 actions pour FF et 20,29 pour LCDPP.

6.6. Comparaison dans les domaines Mprime et Mystery

Dans ces domaines, la relation d'autorisation ne procure pas à LCDPP un avantage décisif sur DPP. Si LCDPP est un peu plus rapide que DPP dans le domaine Mprime (cf. Table 13), il est en revanche un peu plus lent que ce dernier dans le domaine Mystery (cf. Table 14). Ils résolvent tous les deux 32 problèmes sur les 35 du domaine Mprime, et 29 sur les 30 du domaine Mystery. Le temps de recherche est négligeable pour tous les problèmes, sauf un : le problème Mprime−X−6, qui n'est pas résolu en une heure de calcul. Les autres problèmes non résolus le sont par manque de mémoire vive. En effet, les graphes de planification comportent un grand nombre d'actions, de fluents et d'exclusions mutuelles : ces dernières se comptent par millions.

Mais ce calcul très lourd du graphe de planification permet toutefois à DPP et LCDPP de résoudre plus de problèmes que FF, qui en résout 31 dans le domaine Mprime et 18 dans le domaine Mystery. Quand il trouve une solution, FF la donne généralement très rapidement (environ 24 secondes de temps de calcul moyen dans le domaine Mprime, et 0,8 secondes dans le domaine Mystery). Le problème de ces domaines pour FF est que l'on peut atteindre en chaînage avant des états à partir desquels aucun état but n'est accessible ; parfois même il n'y a pas de solution. Comme FF utilise une variante de l'algorithme de recherche Hill-climbing, l'heuristique tombe fréquemment dans des minimums locaux desquels FF ne parvient pas à sortir. Il est alors capable de passer sur une recherche complète de type WA*, mais son heuristique trop peu informée ne lui permet pas souvent de trouver une solution par rapport aux contraintes en temps de calcul et en mémoire vive.

Les bonnes performances de DPP et LCDPP, en comparaison notamment avec les autres planificateurs basés sur Graphplan utilisés pendant la première compétition de planificateurs d'AIPS−98 pour laquelle ont été créés ces domaines, peut sans doute s'expliquer par une meilleure gestion de la mémoire ; en particulier pour le stockage des exclusions mutuelles. La structure que nous avons employée semble en effet particulièrement économique. Elle est basée sur la monotonicité des mutex dans le graphe de planification : deux actions (ou deux fluents) mutuellement exclusives à un niveau i le sont dans tous les niveaux $j < i$ où elles sont toutes les deux présentes, et lorsqu'elles ne sont plus mutuellement exclusives à un niveau k, elles ne le seront plus dans tous les niveaux $l > k$. Ce qui caractérise une mutex est donc son niveau de disparition, que l'on peut stocker dans un tableau à deux entrées indicé par les deux actions (ou fluents). Pour réduire encore l'occupation de la mémoire, on peut utiliser une table triangulaire en repérant les actions et les fluents par un entier. Par exemple, la table triangulaire de la Figure 57 représente les exclusions mutuelles pour 6 actions A, B, C, D, E, F repérées respectivement par les numéros 1, 2, 3, 4, 5, 6. Les actions A et B sont mutex jusqu'au niveau 4, A et C jusqu'au niveau 2, C et E jusqu'au niveau 8, etc. L'avantage de cette structure est qu'elle est commune au graphe entier, donc on ne stocke pas une mutex dans tous les niveaux où elle apparaît. De plus, on peut limiter la taille des cases du tableau à un octet par exemple, ce qui permet tout de même de représenter un graphe de planification de 256 niveaux. Dans les séries de tests que nous avons effectuées on dépasse rarement 20 niveaux. Ainsi, pour un problème comportant 5000 actions sur le dernier niveau du graphe (ce qui est un problème de taille importante pour Graphplan), les mutex seront stockées sur environ 95Mo (en limitant le graphe à 256 niveaux).

2(B)	3(C)	4(D)	5(E)	6(F)	
4	**2**		**1**	**2**	**1(A)**
		3			**2(B)**
			8	**4**	**3(C)**
			7		**4(D)**
				6	**5(E)**

Figure 57 : Stockage des exclusions mutuelles

Problèmes	Temps CPU (sec.)			Ratio DPP/ LCDPP	Actions			Niveaux		
								DPP	LCDPP	
	DPP	LCDPP	FF		DPP	LCDPP	FF		(+)	(++)
Mprime−X−1	1,53	0,86	0,02	1,77	8	8	5	5	5	4
Mprime−X−2	15,62	18,61	0,15	0,84	9	10	10	5	6	4
Mprime−X−3	2,09	1,80	0,04	1,16	4	4	4	4	4	3
Mprime−X−4	1,01	0,90	0,03	1,12	9	9	10	7	7	6
Mprime−X−5	32,48	33,66	≥1 869,29	0,96	–	–		10	–	9
Mprime−X−6	≥3 600	≥3 600	1,36				14	≥ 8		≥ 7
Mprime−X−7	2,59	2,28	0,02	1,14	–	–	–	10	–	8
Mprime−X−8	7,26	9,25	0,21	0,78	10	10	10	5	5	5
Mprime−X−9	4,03	4,11	0,09	0,98	8	8	10	5	5	4
Mprime−X−10			31,41				12			
Mprime−X−11	2,58	1,70	0,03	1,52	9	7	9	7	7	5
Mprime−X−12	3,90	4,16	0,11	0,94	8	9	10	5	6	5
Mprime−X−13	211,03	256,41	≥3 600	0,82	16	17		8	9	7
Mprime−X−14			306,85				18			
Mprime−X−15	378,97	270,97	1,65	1,40	6	17	8	6	8	5
Mprime−X−16	10,38	8,13	0,12	1,28	11	6	7	5	5	4
Mprime−X−17	23,08	21,04	0,39	1,10	5	5	4	4	4	3
Mprime−X−18	947,57	850,11	≥1 178,27	1,11	68	53		14	17	13
Mprime−X−19	51,42	74,50	0,64	0,69	8	9	9	6	7	5
Mprime−X−20	738,74	510,35	1,59	1,45	15	19	13	7	8	6
Mprime−X−21	89,20	83,04	≥2 286,3	1,07	–	–		14	–	12
Mprime−X−22	484,72	547,07	594,61	0,89	13	15	23	8	8	7
Mprime−X−23	180,06	224,89	2,75	0,80	16	12	14	8	9	7
Mprime−X−24	123,71	100,49	1,53	1,23	10	10	9	6	7	5
Mprime−X−25	0,28	0,24	0,01	1,20	4	4	4	4	4	3
Mprime−X−26	5,25	4,31	0,09	1,22	7	9	10	5	5	4
Mprime−X−27	7,84	3,87	0,39	2,03	7	7	5	4	4	3
Mprime−X−28	3,13	1,60	0,04	1,96	9	8	7	7	7	5
Mprime−X−29	4,93	2,87	0,16	1,71	4	6	4	4	5	3
Mprime−X−30	179,81	108,65	0,95	1,65	15	15	11	6	6	5
Mprime−Y−1	5,18	4,56	0,06	1,14	4	5	4	4	4	3
Mprime−Y−2	5,29	5,62	0,20	0,94	7	11	7	7	8	6
Mprime−Y−3	349,70	410,63	62,26	0,85	14	21	22	10	12	9
Mprime−Y−4	5,71	5,85	0,08	0,98	4	4	4	4	4	3
Mprime−Y−5	0,95	1,01	0,03	0,94	7	6	7	4	4	4
Moy. (*)	92,85	83,94	23,87	1,11	8,56	9,41	8,89	5,79	6,07	4,79
Moy. (**)	121,25	111,67	32,51	1,09	10,86	11,17	9,47	6,50	6,55	5,47
Moy. (***)	≥226,67	≥217,38	≥284,05	–	–	–	–	≥6,55	≥6,55	≥5,52

(+) nombre de niveaux du plan après la transformation par **réordonnancer−plan**
(++) nombre de niveaux du plan avant la transformation par **réordonnancer−plan**
(*) moyenne sur les problèmes résolus par tous les planificateurs.
(**) moyenne sur les problèmes résolus par le planificateur correspondant à la colonne (cellules blanches).
(***) moyenne sur les 35 problèmes.
Un tiret (−) indique que le problème correspondant n'admet pas de solution.
cellules grises : échec de la résolution du problème correspondant.

Table 15 : Comparaison dans le domaine Mprime

Problèmes	Temps CPU (sec.)			Ratio DPP/	Actions			Niveaux		
								DPP	LCDPP	
	DPP	LCDPP	FF	LCDPP	DPP	LCDPP	FF		(+)	(++)
Mysty−X−1	0,26	0,48	0,01	0,55	5	6	5	5	6	4
Mysty−X−2	14,36	17,19	0,14	0,84	9	9	10	5	5	4
Mysty−X−3	1,95	1,65	0,04	1,18	4	4	4	4	4	3
Mysty−X−4	2,02	1,79	≥373,76	1,13	–	–		14	–	13
Mysty−X−5	31,85	32,49	≥1 876,54	0,98	–	–		10	–	9
Mysty−X−6	108,93	150,63	≥3 091,08	0,72	19	17		9	9	8
Mysty−X−7	2,61	2,20	0,02	1,19	–	–	–	10	–	8
Mysty−X−8	133,71	118,10	≥1 767,56	1,13	–	–		21	–	19
Mysty−X−9	3,44	3,08	0,23	1,12	8	8	8	5	5	4
Mysty−X−10	1 115,28	1 556,13	≥3 600	0,72	11	8		8	8	7
Mysty−X−11	1,38	0,86	0,02	1,61	7	7	9	7	7	5
Mysty−X−12	1,69	1,70	≥1 341,15	0,99	–	–		8	–	8
Mysty−X−13	211,72	259,46	≥3 600	0,82	16	18		8	10	7
Mysty−X−14			26,71				12			
Mysty−X−15	56,62	59,84	0,53	0,95	6	18	8	6	8	5
Mysty−X−16	9,62	8,52	≥1 952,04	1,13	–	–		9	–	7
Mysty−X−17	19,99	17,32	0,31	1,15	4	4	4	4	4	3
Mysty−X−18	28,16	28,94	0,07	0,97	–	–	–	18	–	18
Mysty−X−19	21,14	22,84	11,39	0,93	7	7	8	6	7	5
Mysty−X−20	50,81	44,34	0,24	1,15	16	16	13	7	8	6
Mysty−X−21	90,01	82,21	≥2 333,8	1,09	–	–		14	–	12
Mysty−X−22	309,40	284,85	≥1 202,5	1,09	–	–		14	–	12
Mysty−X−23	82,58	87,17	≥3 166,72	0,95	–	–		11	–	10
Mysty−X−24	200,84	212,87	≥2 203,51	0,94	–	–		25	–	25
Mysty−X−25	0,26	0,23	0,02	1,12	4	4	4	4	4	3
Mysty−X−26	3,50	3,07	0,59	1,14	6	8	7	6	6	5
Mysty−X−27	1,71	1,40	0,03	1,22	7	7	5	4	4	3
Mysty−X−28	0,87	0,62	0,01	1,41	7	7	7	7	7	5
Mysty−X−29	1,52	1,42	0,03	1,07	4	4	4	4	4	3
Mysty−X−30	12,91	11,96	0,12	1,08	10	11	11	6	7	5
Moy. (*)	13,03	12,79	0,81	1,02	6,93	8,00	7,13	6,35	5,73	6,92
Moy. (**)	86,87	103,91	2,25	0,84	8,33	9,06	7,44	8,93	6,28	7,79
Moy. (***)	≥86,87	≥103,91	≥884,97	–	–	–	–	≥8,93	≥6,28	≥7,79

(+) nombre de niveaux du plan après la transformation par **réordonnancer−plan**
(++) nombre de niveaux du plan avant la transformation par **réordonnancer−plan**
(*) moyenne sur les problèmes résolus par tous les planificateurs.
(**) moyenne sur les problèmes résolus par le planificateur correspondant à la colonne (cellules blanches).
(***) moyenne sur les 30 problèmes.
Un tiret (−) indique que le problème correspondant n'admet pas de solution.
cellules grises : échec de la résolution du problème correspondant.

Table 16 : Comparaison dans le domaine Mystery

VI. Conclusion et perspectives

1. Ce qui a été fait...

Nous avons présenté dans ce mémoire deux techniques de planification classique apparues voici quelques années, qui ont eu un impact considérable sur la communauté : la recherche dans les graphes de planification et la planification par satisfiabilité. Ces approches ont suscité un nombre très important de travaux, et sont encore à l'heure actuelle source d'inspiration pour les plus récents développements.

Les plans-solutions que calcule le planificateur Graphplan, qui a initié la recherche dans les graphes de planification, sont des plans parallèles basés sur la propriété d'indépendance entre actions. Cette propriété soumet les actions que l'on peut considérer comme étant exécutables en parallèle dans le monde réel à une contrainte très forte, garantissant que toutes les exécutions séquentielles d'un plan parallèle mènent au même état résultant. Après une analyse formelle de la structure de ces plans, nous avons montré que la relation d'indépendance peut être remplacée par des relations moins contraignantes, les relations d'autorisation. Ces relations garantissent que toutes les exécutions séquentielles d'un plan parallèle respectant une contrainte d'autorisation, qui peut être vérifiée en temps polynomial, mènent au même état résultant. De plus, nous avons montré qu'il existe une transformation polynomiale d'un plan parallèle basé sur une relation d'autorisation en un plan parallèle basé sur la relation d'indépendance.

Nous avons alors étudié l'utilisation de ces relations d'autorisation dans le planificateur Graphplan, et montré que le coût engendré par leur prise en compte était là aussi polynomial : que ce soit pour la construction du graphe de planification, où l'utilisation d'une relation binaire d'autorisation est effectué de façon similaire à l'utilisation de la relation binaire d'indépendance, que pour l'extraction d'une solution lors de la vérification de l'autorisation des actions considérées simultanément.

L'intérêt de ces relations d'autorisation se révèle d'un point de vue pratique, et s'inscrit dans un cadre plus général : les stratégies de moindre engagement en planification classique. Ces dernières ont pour but de retarder le plus judicieusement possible l'introduction des contraintes sur les plans partiels, en particulier les contraintes d'ordre entre les actions. Mais le choix des contraintes que l'on repousse à plus tard ou que l'on considère immédiatement est difficile dans les techniques antérieures de planification, comme la planification par recherche dans les espaces de plans partiels. Ainsi, le planificateur UCPOP [Penberthy et Weld 1992] effectue souvent des choix de moindre engagement qui le conduisent à raffiner des plans partiels ne pouvant pas mener à des solutions potentielles [Srinivasan et Howe 1995]. Certaines heuristiques, comme DMIN [Peot et Smith 1993] et LCFR [Joslin et Pollack 1994], permettent d'améliorer la stratégie de moindre engagement à considérer, mais ce choix reste heuristique. [Smith et Peot 1993] montrent que le choix d'une "bonne" stratégie de résolution tardive de conflits pour la planification dans les espaces de plans partiels est un problème difficile. Le planificateur LC−Descartes [Joslin et Pollack 1996] ne force aucune contrainte immédiatement, mais teste la consistance d'un CSP ; ce qui s'avère très coûteux en temps de calcul et conduit ses concepteurs au planificateur EC−Descartes, qui effectue certains choix plus tôt.

L'utilisation d'une relation d'autorisation dans Graphplan peut être considérée comme une stratégie de moindre engagement ; mais contrairement aux approches que l'on vient de citer, elle n'est pas heuristique et n'induit qu'une difficulté polynomiale sur la recherche d'une solution. D'après les expérimentations que nous avons conduites sur plusieurs séries de benchmarks classiques, l'amélioration apportée à Graphplan se révèle

parfois considérable pour certains problèmes : le planificateur LCGP que nous avons implémenté ne nécessite que quelques secondes de temps de calcul au lieu de plus de 24 heures dans le domaine Logistics. Ceci s'explique par une forte réduction de la taille du graphe de planification. Lorsque les performances sont dégradées, ce n'est que de manière très limitée par rapport à Graphplan.

L'autre technique de planification que nous avons étudiée est la planification par satisfiabilité. L'objectif initial de cette approche étant l'utilisation de prouveurs SAT développés dans le cadre de la recherche en satisfiabilité pour la recherche d'un plan-solution, le problème essentiel est la transformation d'un problème de planification en une base de clauses. Nous avons étudié en détail l'article [Mali et Kambhampati 1999], qui reprend les principaux codages et en introduit de nouveaux sous le point de vue espaces d'états − espaces de plans. Un des intérêts de cet article est de proposer des notations claires et précises pour décrire ces codages, qui permettent notamment de montrer que le codage dans les espaces de plans proposé initialement dans [Kautz, McAllester et Selman 1996] est inutilement complexe. Les codages proposés dans [Mali et Kambhampati 1999] comportent quelques erreurs, que nous avons illustrées à l'aide d'un exemple simple. Nous avons alors proposé les corrections appropriées, ainsi que de nombreuses améliorations portant notamment sur la réduction de la taille de ces codages en nombre de variables et de clauses et sur la prise en compte du parallélisme dans les codages dans les espaces de plans.

Nous avons enfin étudié le planificateur DPPlan [Baioletti, Marcugini et Milani 2000] qui permet d'effectuer une recherche "à la Davis et Putnam" directement sur un graphe de planification, c'est-à-dire en évitant la transformation du graphe en une base de clauses. L'article qui décrit cet algorithme comporte lui aussi une anomalie, que nous avons mise à jour à l'aide d'un contre-exemple simple. Ceci nous a amené à étudier en détail ce planificateur en nous basant sur le codage du graphe de planification sous forme d'une base de clauses, et à en proposer deux versions comportant un minimum de règles de propagation. Nous avons alors montré que la version originale de DPPlan peut être simplifiée par la suppression d'un des états possibles des fluents, l'état *requis−faux*. Nous avons alors proposé l'algorithme DPP, une version correcte et simplifiée de DPPlan, dans laquelle nous avons introduit des propagations supplémentaires. Nous avons ensuite montré comment prendre en compte les relations d'autorisation dans DPP, ce qui peut être fait très simplement et a été implémenté dans le planificateur LCDPP. Les expérimentations montrent que l'utilisation des relations d'autorisation dans DPPlan est tout aussi intéressante que dans Graphplan.

Nous avons aussi montré que face à un planificateur extrêmement efficace comme FF, l'utilisation d'une méthode basée sur Graphplan ou DPPlan permet de trouver des solutions à certains problèmes que FF ne parvient pas à résoudre par une recherche locale rapide.

2. ... et ce qui reste à faire.

Comme dans toute thèse qui se respecte, de nombreux points restent à explorer :

- Dans tous les domaines que nous avons pu tester, nous n'avons pas trouvé de différences notables dans le fonctionnement de LCGP et LCDPP avec la relation d'autorisation forte ou faible. Il n'apparaît donc pas clairement que l'autorisation faible soit une amélioration de l'autorisation forte dont l'idée avait été rapidement suggérée par [Dimopoulos, Nebel et Koehler 1997] dans un cadre d'utilisation différent. Nous avons commencé quelques expérimentations sur des domaines aléatoires (ce qui n'a pas beaucoup de sens pour la planification) qui auraient tendance à montrer que suivant les cas, l'une ou l'autre des deux relations peut se révéler supérieure. Il serait donc intéressant de définir les caractéristiques d'un domaine inspiré du réel qui permettraient de différencier ces relations.
- La partie sur le codage d'un problème de planification sous forme de base de clauses ne demande qu'à être poursuivie. Il serait intéressant d'étudier de façon exhaustive tous les codages proposés dans la littérature, et d'établir des liens entre ces codages. Il semble par exemple que le codage par liens causaux, protection d'intervalles et ordre partiel est inutilement complexe et que sa simplification "naturelle" pourrait être le codage par liens causaux, protection d'intervalles et étapes contiguës. Il serait aussi intéressant d'étudier différentes représentations d'actions dans ces codages et de réaliser une étude expérimentale montrant comment ces codages se comportent par rapport aux simplifications comme la propagation unitaire, etc.
- LCDPP est le plus souvent moins performant que LCGP, mais peut être meilleur si l'on n'utilise pas de retour arrière intelligent. De plus, LCDPP n'effectue aucune mémoization. Il faudrait donc étudier si LCDPP peut être amélioré de cette façon ; ce qui semble possible puisque le retour arrière intelligent est utilisé avec succès dans le cadre de la recherche en satisfiabilité.

- Il serait intéressant d'implémenter des heuristiques utilisées par les prouveurs SAT dans LCDPP. Un des avantages de ce dernier étant de pouvoir utiliser des heuristiques plus liées au problème de planification, on peut tout de même se poser la question. De plus, il faudrait tester expérimentalement dans quelle mesure une approche de type DPPlan peut être intéressante par rapport à l'utilisation d'un prouveur SAT.
- Enfin, une question cruciale : ces techniques de résolution ont-elles un avenir face aux approches actuelles de la planification par recherche heuristique ? Le graphe de planification, lui, reste intéressant du point de vue de l'extraction d'heuristique, comme dans le planificateur AltAlt [Nguyen et Kambhampati 2000]. Il serait à ce propos intéressant de voir si la relation d'autorisation peut être bénéfique dans ce cadre. Mais les techniques d'extraction de la solution comme celles de Graphplan avec retour arrière intelligent et mémoization, l'utilisation d'un prouveur SAT sur une base de clauses, ou bien la recherche de type DPPlan, ont-elles encore un intérêt ? Dans nos tests, nous avons montré que ces techniques peuvent parfois être compétitives avec un planificateur performant, FF. Ceci peut suggérer l'intérêt d'une approche hybride, qui lancerait une recherche de ce type après l'échec d'une recherche locale rapide. Mais les progrès de la planification par recherche heuristique risquent peut-être dans un avenir proche de rendre caduques cette supposition.

Table des matiè res

Table des définitions

Table des théorèmes

Table des algorithmes

Liste des tableaux

Bibliographie

[Anderson, Smith et Weld 1998] Anderson C., Smith D.E., Weld D.S., Conditional effects in Graphplan, in: Proceedings of the 4th International Conference on Artificial Intelligence Planning and Scheduling (AIPS−98), Pittsburgh, PA, USA, 1998, pp. 44−53.

[Bacchus et Kabanza 1995] Bacchus F., Kabanza F., Using temporal logic to control search in a forward-chaining planner, in: M. Ghallab, A. Milani (Eds.), New directions in AI Planning, IOS Press, Amsterdam, Netherlands, 1996, pp. 141−153.

[Bacchus et Kabanza 2000] Bacchus F., Kabanza F., Using temporal logic to express search control knowledge for planning, Artificial Intelligence 116 (2000) 123−191.

[Bäckström 1992] Bäckström C., Computational complexity of reasoning about plans, Ph.D Thesis, Department of Computer Science and Information Science, Linköping University, Linköping, Sweden, 1992.

[Bäckström 1998] Bäckström C., Computational aspects of reordering plans, Journal of Artificial Intelligence Research 9 (1998) 99−137.

[Bäckström et Klein 1991] Bäckström C., Klein I., Parallel non-binary planning in polynomial time, in: Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI−91), Sydney, Australia, 1991, pp. 268−273.

[Baioletti, Marcugini et Milani 1998a]Baioletti M., Marcugini S., Milani A., Encoding planning constraints into partial order planning domains, in: Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning (KR−98), Trento, Italy, 1998, 608−616.

[Baioletti, Marcugini et Milani 1998b]Baioletti M., Marcugini S., Milani A., C−SATPlan: a SATPlan-based tool for planning with constraints, in: Working Notes of Workshop Planning As Combinatorial Search, 4th International Conference on Artificial Intelligence Planning and Scheduling (AIPS−98), Pittsburgh, PA, USA, 1998.

[Baioletti, Marcugini et Milani 1998c] Baioletti M., Marcugini S., Milani A., An extension of SATPLAN for planning with constraints, in: Proceedings of the 8th International Conference on Artificial Intelligence: Methodology, Systems, and Applications (AIMSA−98), Sozopol, Bulgaria, 1998, pp. 39−49.

[Baioletti, Marcugini et Milani 2000] Baioletti M., Marcugini S., Milani A., DPPlan: An algorithm for fast solutions extraction from a planning graph, in: Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS−2000), Breckenridge, CO, USA, 2000, pp. 13−21.

[Barret et Weld 1994] Barret A., Weld D.S., Partial-order planning: Evaluating possible efficiency gains, Artificial Intelligence 67 (1994) 71−112.

[Bayardo et Schrag 1997] Bayardo R.J.J., Schrag R.C., Using CSP look-back techniques to solve real-world SAT instances, in: Proceedings of the 15th National Conference on Artificial Intelligence (AAAI−98), Madison, WI, USA, 1998, pp. 203−208.

[Blum et Furst 1995] Blum A.L., Furst M.L., Fast planning through planning-graphs analysis, in: Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI−95), Montréal, Québec, Canada, 1995, pp. 1636−1642.

[Blum et Furst 1997] Blum A.L., Furst M.L., Fast planning through planning-graphs analysis, Artificial Intelligence 90 (1997) 281−300.

[Blum et Langford 1998] Blum A.L., Langford J.C., Probabilistic planning in the Graphplan framework, in: Working Notes of Workshop Planning As Combinatorial Search, 4th International Conference on Artificial Intelligence Planning and Scheduling (AIPS−98), Pittsburgh, PA, USA, 1998.

[Blum et Langford 1999] Blum A.L., Langford J.C., Probabilistic Planning in the Graphplan Framework, in: Proceedings of the 5th European Conference on Planning (ECP−99), Durham, UK, 1999, pp. 319−332.

[Bonet et Geffner 1998] Bonet B., Geffner H., HSP: Heuristic search planner, Planning Competition of the 4th International Conference on Artificial Intelligence Planning and Scheduling (AIPS−98), Pittsburgh, PA, USA, 1998.

[Bonet et Geffner 1999] Bonet B., Geffner H., Planning as heuristic search: New results, in: Proceedings of the 5th European Conference on Planning (ECP−99), Durham, UK, 1999, pp. 360−372.

[Bonet et Geffner 2001] Bonet B., Geffner H., Planning as heuristic search, to be published in Artificial Intelligence, Special issue on Heuristic Search.

[Bonet, Loerincs et Geffner 1997] Bonet B., Loerincs G., Geffner H., A robust and fast action selection mechanism for planning, in: Proceedings of the 14th National Conference on Artificial Intelligence (AAAI−97), Providence, Rhode Island, USA, 1997, pp. 714−719.

[Borning et al. 1997] Borning A., Marriot K., Stuckey P., Xiao Y., Solving linear arithmetic constraints for user interface applications, in: Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology (UIST−97), Banff, Alberta, Canada, 1997, pp. 87−96.

[Brafman 1999] Brafman R.I., Reachability, relevance, resolution and the planning as satisfiability approach, in: Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI−99), Stockholm, Sweden, 1999, pp. 976−981.

[Brafman 2001] Brafman R.I., On reachability, relevance, and resolution in the planning as satisfiability approach, Journal of Artificial Intelligence Research 14 (2001) 1−28.

[Bylander 1991] Bylander T., Complexity results for planning, in: Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI−91), Sydney, Australia, 1991, pp. 274−279.

[Castell 1997] Castell T., Consistance et déduction en logique propositionnelle, Thèse de doctorat de l'Université Paul Sabatier, Institut de Recherche en Informatique de Toulouse, Toulouse, 1997.

[Cayrol, Régnier et Vidal 2000a] Cayrol M., Régnier P., Vidal V., LCGP : Une amélioration de Graphplan par relâchement de contraintes entre actions simultanées, dans : Actes du 12ème Congrès de Reconnaissance des Formes et Intelligence Artificielle (RFIA−2000), Paris, France, 2000, pp. 79−88.

[Cayrol, Régnier et Vidal 2000b] Cayrol M., Régnier P., Vidal V., New results about LCGP, a least committed Graphplan, in: Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS−2000), Breckenridge, CO, USA, 2000, pp. 273−282.

[Cayrol, Régnier et Vidal 2001] Cayrol M., Régnier P., Vidal V., Least commitment in Graphplan, Artificial Intelligence 130 (2001) 85−118.

[Chapman 1987] Chapman D., Planning for conjunctive goals, Artificial Intelligence 32 (1987) 333−377.

[Crawford et Auton 1996] Crawford J., Auton L., Experimental results on the crossover point in satisfiability problems, Artificial Intelligence 81 (1996) 31−58.

[Davis, Logemann et Loveland 1962] Davis M., Logemann G., Loveland D., A computing procedure for quantification theory, Communications of the ACM 5 (1962) 394−397.

[Davis et Putnam 1960] Davis M., Putnam H., A computing procedure for quantification theory, Journal of the ACM 7 (1960) 201−215.

[Dimopoulos, Nebel et Koehler 1997] Dimopoulos Y., Nebel B., Koehler J., Encoding planning problems in nonmonotonic logic programs, in: Proceedings of the 4th European Conference on Planning (ECP−97), Toulouse, France, 1997, pp. 169−181.

[Do et Kambhampati 2000] Do B.M., Kambhampati S., Solving planning-graph by compiling it into CSP, in: Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS−2000), Breckenridge, CO, USA, 2000, pp. 82−91.

[Do, Srivastava et Kambhampati 2000] Do B.M., Srivastava B., Kambhampati S., Investigating the effect of relevance and reachability constraints on SAT encodings of planning, in: Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS−2000), Breckenridge, CO, USA, 2000, pp. 308−314.

[Doherty et Kvarnström 1999] Doherty P., Kvarnström J., TALplanner: An empirical investigation of a temporal logic-based forward chaining planner, in: Proceedings of the 6th International Workshop on Temporal Representation and Reasoning (TIME−99), Orlando, FL, USA, 1999, pp. 47−54.

[Ernst, Millstein et Weld 1997] Ernst M., Millstein T., Weld D.S., Automatic SAT−compilation of planning problems, in: Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI−97), Nagoya, Japan, 1997, pp. 1169−1176.

[Erol, Nau et Subrahmanian 1992] Erol K., Nau D.S., Subrahmanian V.S., On the complexity of domain-independent planning, in: Proceedings of the 10th National Conference on Artificial Intelligence (AAAI−92), San Jose, CA, USA, 1992, pp. 381−386.

[Fikes et Nilsson 1971] Fikes R.E., Nilsson N.J., STRIPS: A new approach to the application of theorem proving to problem solving, Artificial Intelligence 2 (1971) 189−208.

[Fink et Veloso 1994] Fink E., Veloso M., Prodigy planning algorithm, Technical Report 94−123, CMU School of Computer Science, 1994.

[Fox et Long 1998] Fox M., Long D., The automatic inference of state invariants in TIM, Journal of Artificial Intelligence Research 9 (1998) 367−421.

[Fox et Long 1999] Fox M., Long D., The detection and exploitation of symmetry in planning problems, in: Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI−99), Stockholm, Sweden, 1999, pp. 956−961.

[Fox et Long 2000] Fox M., Long D., Utilizing automatically infered invariants in graph construction and search, in: Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS−2000), Breckenridge, CO, USA, 2000, pp. 102−111.

[Gazen et Knoblock 1997] Gazen B.C, Knoblock C.A., Combining the expressivity of UCPOP with the efficiency of Graphplan, in: Proceedings of the 4th European Conference on Planning (ECP−97), Toulouse, France, 1997, pp. 221−233.

[Gerevini et Schubert 1996] Gerevini A., Schubert L., Accelerating partial-order planners: Some techniques for effective search control and pruning, Journal of Artificial Intelligence Research 5 (1996) 95−137.

[Gerevini et Schubert 1998] Gerevini A., Schubert L., Inferring state constraints for domain-independent planning, in: Proceedings of the 15th National Conference on Artificial Intelligence (AAAI−98), Madison, WI, USA, 1998, pp. 905−912.

[Ghallab, Alami et Chatila 1987] Ghallab M., Alami R., Chatila R., Dealing with time in planning and execution monitoring, in: Proceedings of the 4th International Symposium on Robotics Research, 1987.

[Ghallab et Mounir-Alaoui 1989] Ghallab M., Mounir-Alaoui A., Managing efficiently temporal relations through indexed spanning trees, in: Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI−89), Detroit, MI, USA, 1989, pp. 1297−1303.

[Giunchiglia, Massarotto et Sebastiani 1998] Giunchiglia E., Massarotto A., Sebastiani R., Act, and the rest will follow: Exploiting determinism in planning as satisfiability, in: Proceedings of the 15th National Conference on Artificial Intelligence (AAAI−98), Madison, WI, USA, 1998, pp. 948−953.

[Gomes, Selman et Kautz 1998] Gomes C.P., Selman B., Kautz H., Boosting combinatorial search through randomization, in: Proceedings of the 15th National Conference on Artificial Intelligence (AAAI−98), Madison, WI, USA, 1998, pp. 431−437.

[Green 1969] Green C., Application of theorem proving to planning, in: Proceedings of the 1st International Joint Conference on Artificial Intelligence (IJCAI−69), Washington, DC, USA, 1969, pp. 219−239.

[Guéré et Alami 1999] Guéré E., Alami R., A possibilistic planner that deals with non-determinism and contingency, in: Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI−99), Stockholm, Sweden, 1999, pp. 996−1001.

[Haas 1987] Haas A., The case for domain-specific frame axioms, in: F.M. Brown (Ed.), The Frame Problem in Artificial Intelligence, Morgan Kaufmann, Los Altos, CA, 1987.

[Haslum et Geffner 2000] Haslum P., Geffner H., Admissible heuristics for optimal planning, in: Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS−2000), Breckenridge, CO, USA, 2000, pp. 140−194.

[Hoffmann 2000] Hoffmann J., A heuristic for domain-independent planning and its use in a enforced hill-climbing algorithm, Technical Report 133, Albert Ludwigs University, Freiburg, Germany, 2000.

[Hoffman et Nebel 2000] Hoffman J. Nebel B., The FF planning system: Fast plan generation through heuristic search, submitted to Journal of Artificial Intelligence Research.

[Huang, Selman et Kautz 1999] Huang Y., Selman B., Kautz H., Control knowledge in planning: Benefits and tradeoffs, in: Proceedings of the 16th National Conference on Artificial Intelligence (AAAI−99), Orlando, FL, USA, 1999, pp. 511−517.

[Jacopin 1993] Jacopin E., Algorithmique de l'interaction : Le cas de la planification. Thèse de doctorat de l'Université Paris VI, Laboratoire Formes et Intelligence Artificielle, Paris, 1993.

[Joslin et Pollack 1994] Joslin D., Pollack M.E., Least-cost flaw repair: A plan refinement strategy for partial-order planning, in: Proceedings of the 12th National Conference on Artificial Intelligence (AAAI−94), Seattle, WA, USA, 1994, pp. 1004−1009.

[Joslin et Pollack 1996] Joslin D., Pollack M.E., Is "early commitment" in plan generation ever a good idea ?, in: Proceedings of the 13th National Conference on Artificial Intelligence (AAAI−96), Portland, OR, USA, 1996, pp. 1188−1193.

[Kambhampati 1994] Kambhampati S., Refinement search as a unifying framework for analyzing planning algorithms, in: Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning (KR−94), Bonn, Germany, 1994, pp. 329−340.

[Kambhampati 1996] Kambhampati S., Universal Classical Planner: An algorithm for unifying state-space and plan-space planning, in: M. Ghallab, A. Milani (Eds.), New directions in AI Planning, IOS Press, Amsterdam, Netherlands, 1996, pp. 61−75.

[Kambhampati 1997] Kambhampati S., Refinement planning as a unifying framework for plan synthesis, AI Magazine 18 (1997) 67−97.

[Kambhampati 1999] Kambhampati S., Improving Graphplan's search with EBL & DDB techniques, in: Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI−99), Stockholm, Sweden, 1999, pp. 982−987.

[Kambhampati 2000] Kambhampati S., Planning graph as a (dynamic) CSP: Exploiting EBL, DDB and other CSP techniques in Graphplan, Journal of Artificial Intelligence Research 12 (2000) 1−34.

[Kambhampati, Knoblock et Yang 1995] Kambhampati S., Knoblock C.A., Yang Q., Planning as refinement search: A unified framework for evaluating design tradeoffs in partial-order planning, Artificial Intelligence 76 (1995) 167−238.

[Kambhampati et Nigenda 2000] Kambhampati S., Nigenda R. S., Distance-based goal-ordering heuristics for Graphplan, in: Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS−2000), Breckenridge, CO, USA, 2000, pp. 315−322.

[Kambhampati, Parker et Lambrecht 1997] Kambhampati S., Parker E, Lambrecht E., Understanding and extending Graphplan, in: Proceedings of the 4th European Conference on Planning (ECP−97), Toulouse, France, 1997, pp. 260−272.

[Kautz, McAllester et Selman 1996] Kautz H., McAllester D., Selman B., Encoding plans in propositional logic, in: Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning (KR-96), Cambridge, MA, USA, 1996, pp. 374−384.

[Kautz et Selman 1992] Kautz H., Selman B., Planning as Satisfiability, in: Proceedings of the 10th European Conference on Artificial Intelligence (ECAI-92), Vienna, Austria, 1992, pp. 359−363.

[Kautz et Selman 1996] Kautz H., Selman B., Pushing the Envelope: Planning, propositional logic and stochastic search, in: Proceedings of the 13th National Conference on Artificial Intelligence (AAAI−96), Portland, OR, USA, 1996, pp. 1194−1201.

[Kautz et Selman 1998a] Kautz H., Selman B., The role of domain-specific knowledge in the planning as satisfiability framework, in: Proceedings of the 4th International Conference on Artificial Intelligence Planning and Scheduling (AIPS−98), Pittsburgh, PA, USA, 1998, pp. 181−189.

[Kautz et Selman 1998b] Kautz H., Selman B., BLACKBOX: A new approach to the application of theorem proving to problem solving, in: Working Notes of Workshop Planning As Combinatorial Search, 4th International Conference on Artificial Intelligence Planning and Scheduling (AIPS−98), Pittsburgh, PA, USA, 1998.

[Kautz et Selman 1999] Kautz H., Selman B., Unifying SAT-based and Graph-based Planning, in: Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI−99), Stockholm, Sweden, 1999, pp. 318−325.

[Knoblock et Yang 1995] Knoblock C.A., Yang Q., A comparison of the SNLP and TWEAK planning algorithms, in: Procceedings of the Symposium of the Foundation of Automatic Planning, Stanford, CA, 1995.

[Koehler 1998a] Koehler J., Planning under resources constraints, in: Proceedings of the 13th European Conference on Artificial Intelligence (ECAI−98), Brighton, UK, 1998, pp. 489−493.

[Koehler 1998b] Koehler J., Solving complex planning tasks through extraction of subproblems, in: Proceedings of the 4th International Conference on Artificial Intelligence Planning and Scheduling (AIPS−98), Pittsburgh, PA, USA, 1998, pp. 62−69.

[Koehler et Hoffmann 1999] Koehler J., Hoffmann J., Handling of inertia in a planning system, Technical Report 122, Albert Ludwigs University, Freiburg, Germany, 1999.

[Koehler et Hoffmann 2000] Koehler J., Hoffmann J., On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm, Journal of Artificial Intelligence Research 12 (2000) 338−386.

[Koehler et al. 1997] Koehler J., Nebel B., Hoffmann J., Dimopoulos Y., Extending planning-graphs to an ADL subset, in: Proceedings of the 4th European Conference on Planning (ECP−97), Toulouse, France, 1997, pp. 273−285.

[Koehler et Schuster 2000] Koehler J., Schuster K., Elevator control as a planning problem, in: Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS−2000), Breckenridge, CO, USA, 2000, pp. 331−338.

[Korf 1985] Korf R.E., Depth-first iterative deepening: An optimal admissible tree search, Artificial Intelligence 27 (1985) 97−109.

[Korf 1987] Korf R.E., Planning as search: A quantitative approach, Artificial Intelligence 33 (1987) 65−88.

[Korf 1990] Korf R.E., Real-time heuristic search, Artificial Intelligence 42 (1990) 189−211.

[Kvarnström, Doherty et Haslum 2000] Kvarnström J., Doherty P., Haslum P., Extending TALplanner with concurrency and resources, in: Proceedings of the 14th European Conference on Artificial Intelligence (ECAI−2000), Berlin, Germany, 2000, pp. 501−505.

[Laborie 1995] Laborie P., IxTeT : Une approche intégrée pour la gestion de ressources et la synthèse de plans, Thèse de doctorat de l'Ecole Nationale Supérieure des Télécommunications, Paris, 1995.

[Laruelle 1994] Laruelle H., Planification temporelle et exécution de tâches en robotique, Thèse de doctorat de l'Université Paul Sabatier, Laboratoire d'Automatique et d'Analyse des Systèmes, Toulouse, 1994.

[Le Berre 2000] Le Berre D., Autour de SAT : Le calcul d'impliquants P-restreints, algorithmes et applications, Thèse de doctorat de l'Université Paul Sabatier, Institut de Recherche en Informatique de Toulouse, Toulouse, 2000.

[Li et Anbulagan 1997] Li C.M., Anbulagan, Heuristics based on unit propagation for satisfiability problems, in: Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI−97), Nagoya, Japan, 1997, pp. 366−371.

[Lin 2001] Lin F., A planner called R, Extended abstract for AI Journal, 2001.

[Long et Fox 1999] Long D., Fox M., The efficient implementation of the plan-graph in STAN, Journal of Artificial Intelligence Research 10 (1999) 87−115.

[Mali et Kambhampati 1998] Mali A., Kambhampati S., Refinement-based Planning as satisfiability, in: Working Notes of Workshop Planning As Combinatorial Search, 4th International Conference on Artificial Intelligence Planning and Scheduling (AIPS−98), Pittsburgh, PA, USA, 1998.

[Mali et Kambhampati 1999] Mali A., Kambhampati S., On the utility of plan-space (causal) encodings, in: Proceedings of the 16th National Conference on Artificial Intelligence (AAAI−99), Orlando, FL, USA, 1999, pp. 557−563.

[Marques Silva et Sakallah 1996] Marques Silva J.P., Sakallah K.A., GRASP − A new search algorithm for satisfiability, in: Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD−96), San Jose, CA, USA, 1996, pp. 220−227.

[McAllester et Rosenblitt 1991] Mcallester D.A., Rosenblitt D., Systematic nonlinear planning, in: Proceedings of the 9th National Conference on Artificial Intelligence (AAAI−91), Anaheim, CA, USA, 1991, pp. 634−639.

[McCarthy et Hayes 1969] McCarthy J., Hayes P.J., Some philosophical problems from the standpoint of AI, Machine Intelligence 4 (1969).

[McDermott 1996] McDermott D.V., A heuristic estimator for means-end analysis in planning, in: Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems (AIPS−96), Edinburgh, UK, 1996, pp. 142−149.

[McDermott 1998] McDermott D.V., PDDL, The Planning Domain Definition Language, Technical Report, Yale University, (http://ftp.cs.yale.edu/pub/macdermott), 1998.

[McDermott 1999] McDermott D.V., Using regression-match graphs to control search in planning, Artificial Intelligence 109 (1999) 111−159.

[Melhorn 1984] Mehlhorn K., Data structures and algorithms 2: Graph algorithms and NP-Completness, Springer-Verlag, Berlin, 1984.

[Miller, Firby et Dean 1985] Miller D.P., Firby R.J., Dean T., Deadlines, travel time and robot problem solving, in: Proceedings of the 9th International Joint Conference on Artificial Intelligence (IJCAI−85), Los angeles, CA, USA, 1985, pp. 1052−1054.

[Minton, Bresina et Drummond 1994] Minton S., Bresina J., Drummond M., Total-order and partial-order planning: A comparative analysis, Journal of Artificial Intelligence Research 2 (1994) 222−262.

[Mittal et Falkenhainer 1990] Mittal S., Falkenhainer B., Dynamic constraint satisfaction problems, in: Proceedings of the 8th National Conference on Artificial Intelligence (AAAI−90), Boston, MA, USA, 1990, pp. 25−32.

[Nebel, Dimopoulos et Koehler 1997] Nebel B., Dimopoulos Y., Koehler J., Ignoring irrelevant facts and operators in plan generation, in: Proceedings of the 4th European Conference on Planning (ECP−97), Toulouse, France, 1997, pp. 338−350.

[Newell et Simon 1963] Newell A., Simon H.A., GPS, a program that simulates human thought, Computers and Thought, McGrawHill, New-York, 1963.

[Newell et Simon 1972] Newell A., Simon H.A., Human Problem Solving, Prentice Hall, Englewood Cliffs, NJ, 1972.

[Nguyen et Kambhampati 2000] Nguyen X.L., Kambhampati S., Extracting effective and admissible state space heuristics from the planning-graph, in: Proceedings of the 17th National Conference on Artificial Intelligence (AAAI−2000), Austin, TX, USA, 2000, 798−805.

[Nguyen et Kambhampati 2001] Nguyen X.L., Kambhampati S., Reviving partial order planning, to be published in: Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI−2001), 2001.

[Pearl 1983] Pearl J., Heuristics, Morgan Kaufmann, San Mateo, CA, 1983.

[Pednault 1989] Pednault E.P.D., ADL: Exploring the middle ground between STRIPS and the situation calculus, in: Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning (KR−89), Toronto, Canada, 1989, pp. 324−332.

[Penberthy et Weld 1992] Penberthy J.S., Weld D.S., UCPOP: A sound, complete, partial order planner for ADL, in: Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR−92), Cambridge, MA, USA, 1992, pp. 103−114.

[Penberthy et Weld 1994] Penberthy J.S., Weld D.S., Temporal planning with continuous change, in: Proceedings of the 12th National Conference on Artificial Intelligence (AAAI−94), Seattle, WA, USA, 1994, pp. 1010−1015.

[Peot et Smith 1993] Peot M.A., Smith D.E., Threat-removal strategies for partial-order planning, in: Proceedings of the 11th National Conference on Artificial Intelligence (AAAI−93), Washington, DC, USA, 1993, pp. 492−499.

[Refanidis et Vlahavas 1999] Refanidis I., Vlahavas I., GRT: A domain independent heuristic for strips worlds based on greedy regression tables, in: Proceedings of the 5th European Conference on Planning (ECP−99), Durham, UK, 1999, pp. 347−359.

[Régnier et Fade 1991] Régnier P., Fade B., Complete determination of parallel actions and temporal optimization in linear plans of actions, in: Proceedings of the European Workshop on Planning (EWSP−91), Sankt Augustin, Swiss, 1991, pp. 100−111.

[Rintanen 1998] Rintanen J., A planning algorithm not based on directionnal search, in: Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning (KR−98), Trento, Italy, 1998, pp. 617−624.

[Rutten et Hertzberg 1993] Rutten E., Hertzberg J., Temporal planner = nonlinear planner + time map manager, AI Communications 6 (1993) 18−26.

[Sacerdoti 1975] Sacerdoti E.D., The non-linear nature of plans, in: Proceedings of the 4th International Joint Conference on Artificial Intelligence (IJCAI−75), Tbilisi, USSR, 1975, pp. 206−214.

[Sebastia, Onaindia et Marzal 2000] Sebastia L., Onaindia E., Marzal E., A Graph-based approach for POCL Planning, in: Proceedings of the 14th European Conference on Artificial Intelligence (ECAI−2000), Berlin, Germany, 2000, pp. 531−535.

[Selman, Kautz et Cohen 1994] Selman B., Kautz H., Cohen B., Noise strategies for improving local search, in: Proceedings of the 12th National Conference on Artificial Intelligence (AAAI−94), Seattle, WA, USA, 1994, pp. 337−343.

[Selman, Levesque et Mitchel 1992] Selman B., Levesque H., Mitchel D., A new method for solving hard satisfiability problems, in: Proceedings of the 10th National Conference on Artificial Intelligence (AAAI−92), San Jose, CA, USA, 1992, pp. 459−465.

[Smith et Peot 1993] Smith D.E., Peot M.A., Postponing threats in partial-order planning, in: Proceedings of the 11th National Conference on Artificial Intelligence (AAAI−93), Washington, DC, USA, 1993, pp. 500−506.

[Smith et Weld 1998] Smith D.E., Weld D.S., Conformant Graphplan, in: Proceedings of the 15th National Conference on Artificial Intelligence (AAAI−98), Madison, WI, USA, 1998, pp. 889−896.

[Smith et Weld 1999] Smith D.E., Weld D.S., Temporal Planning with mutual exclusion reasoning, in: Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI−99), Stockholm, Sweden, 1999, pp. 326−333.

[Srinivasan et Howe 1995] Srinivasan R., Howe A.E., Comparison of methods for improving search efficiency in a partial-order planner, in: Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI−95), Montréal, Québec, Canada, 1995, pp. 1620−1626.

[Steele 1989] Steele G.L., Common Lisp the Language, 2nd Edition, Digital Press, Woburn, MA, 1989.

[Sussman 1974] Sussman G.J., The virtuous nature of bugs, in: Proceedings of the First Conference of the Society for the study of AI and the simulation of Behaviour, Brighton, UK, 1974.

[Tate 1975] Tate A., Interacting goals and their use, in: Proceedings of the 4th International Joint Conference on Artificial Intelligence (IJCAI−75), Tbilisi, USSR, 1975, pp. 215−218.

[Tate 1977] Tate A., Generating project networks, in: Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI−77), Cambridge, MA, USA, 1977, pp. 888−893.

[van Beek et Chen 1999] van Beek P., Chen X., Cplan: A constraint programming approach to planning, in: Proceedings of the 16th National Conference on Artificial Intelligence (AAAI−99), Orlando, FL, USA, 1999, pp. 585−590.

[van Gelder et Okushi 1999] van Gelder A., Okushi F., A propositional theorem prover to solve planning and other problems, Annals of Mathematics and Artificial Intelligence 26 (1999) 87−112.

[Veloso et Blythe 1994] Veloso M., Blythe J., Linkability: Examining causal link commitments in partial-order planning, in: Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems (AIPS−94), Chicago, IL, USA, 1994, pp. 170−175.

[Vere 1983] Vere S., Planning in time: Windows and durations for activities and goals, IEEE Trans. on Pattern Analysis and Machine Intelligence 5 (1983) 246−267.

[Vidal et Régnier 1999] Vidal V., Régnier P., Total order planning is better than we thought, in: Proceedings of the 16th National Conference on Artificial Intelligence (AAAI−99), Orlando, FL, USA, 1999, pp. 591−596.

[Waldinger 1977] Waldinger R., Achieving several goals simultaneously, Machine Intelligence 8 (1977) 80−136.

[Warren 1974] Warren D., Warplan: A system for generating plans, Memo 76, Department of Computational Logic, University of Edinburgh, Aedin, 1974.

[Wilkins 1988] Wilkins D.E., Practical planning, Morgan Kaufman, San Mateo, CA, 1988.

[Weld 1994] Weld D.S., An introduction to least commitment planning, AI Magazine 15 (1994) 27−61.

[Weld 1999] Weld D.S., Recent advances in AI planning, AI Magazine 20 (1999) 93−123.

[Weld, Anderson et Smith 1998] Weld D.S., Anderson C.R., Smith D.E., Extending Graphplan to handle uncertainty and sensing actions, in: Proceedings of the 15th National Conference on Artificial Intelligence (AAAI−98), Madison, WI, USA, 1998, pp. 897−904.

[Wolfman et Weld 1999] Wolfman S.A., Weld D.S., The LPSAT engine and its application to resource planning, in: Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI−99), Stockholm, Sweden, 1999, pp. 310−317.

[Zhang 1997] Zhang H., SATO: An efficient propositional prover, in: Proceedings of the 14th International Conference on Automated Deduction (ICAD−97), 1997, pp. 272−275.

[Zimmerman et Kambhampati 1999] Zimmerman T., Kambhampati S., Exploiting symmetry in the planning-graph via explanation-guided search, in: Proceedings of the 16th National Conference on Artificial Intelligence (AAAI−99), Orlando, FL, USA, 1999, pp. 605−611.

TITLE : Search in planning-graphs, satisfiability and least commitment strategies. The LCGP and LCDPP systems.

ABSTRACT

This thesis deals with Artificial Intelligence planning. After presenting the main methods, particularly search in planning-graphs and planning as satisfiability, we analyse in detail the Graphplan planner. A formal study of its plans lead us to renew a basic idea of planning in partial plan space, the least commitment strategies. Indeed, one of the reasons that made the succes of this planning paradigm is the fact that some choices about the ordering of actions are delayed as most as possible (depending on the strategies), which permits a reduction of the search space. We show that some particular choices about the ordering of actions can be made after a solution-plan is found, which leads to significant improvements of the performances due to the search space reduction. The induced calculus is polynomial.

We then present the encoding of a planning problem into a propositional logic formula. We describe the main encoding methods, doing a critical analysis of a reference article. We propose corrections and improvements for each of these encodings.

We finally present the DPPlan method that allows the use of SAT technics within a planning-graph. We analyse the article that describes this algorithm and show that DPPlan suffers from a conception problem. We then propose a formalization of this method that permits to exhibit a "minimal" and correct version of DPPlan, and show how to extend it with the propagation rules of the original algorithm. We then describe how to take advantage of our least commitment strategy within DPPlan.

KEYWORDS

Artificial Intelligence, Planning, STRIPS, Parallel plans, Constraint systems, Independence, Least commitment, Authorization, Graphplan, Propositional logic, Satisfiability, Satplan, Davis and Putnam.

AUTEUR : Vincent VIDAL

TITRE : Recherche dans les graphes de planification, satisfiabilité et stratégies de moindre engagement. Les systèmes LCGP et LCDPP.

DIRECTEUR DE THÈSE : Michel CAYROL

LIEU ET DATE DE SOUTENANCE : IRIT − UPS, le 10 juillet 2001

RESUME

Cette thèse s'inscrit dans le cadre de la planification de tâches en Intelligence Artificielle. Après avoir présenté les principales méthodes, en particulier la recherche dans les graphes de planification et la planification par satisfiabilité, nous analysons en détail le planificateur Graphplan. Une étude formelle des plans qu'il calcule nous conduit à réactualiser une idée de base de la planification dans les espaces de plans : le moindre engagement. En effet, une des raisons du succès de ce type de planification est le fait que le choix de l'ordonnancement des actions est retardé le plus possible (suivant les stratégies), ce qui permet de réduire la taille de l'espace de recherche. Nous montrons que l'on peut effectuer certains choix d'ordonnancement d'actions après avoir trouvé un plan-solution, ce qui apporte dans certains domaines une amélioration significative des performances due à une forte réduction de la taille de l'espace de recherche. Le calcul supplémentaire induit est de complexité polynomiale.

Nous présentons ensuite le codage d'un problème de planification sous forme de base de clauses en logique propositionnelle. Nous passons en revue les principales techniques de codage proposées dans la littérature, en nous appuyant notamment sur une analyse critique d'un article de référence. Nous proposons des corrections ou des améliorations pour chacun des codages étudiés.

Nous présentons enfin la méthode DPPlan qui permet d'utiliser des techniques SAT sur un graphe de planification. Nous analysons l'article qui décrit cet algorithme et montrons que ce dernier possède un problème de conception. Nous proposons alors une formalisation de cette technique qui permet d'exhiber une version "minimale" et correcte de DPPlan, et montrons comment l'étendre avec les règles de propagation de la version originelle. Nous montrons enfin comment profiter de notre stratégie de moindre engagement dans DPPlan.

MOTS−CLÉS

Intelligence Artificielle, Planification, STRIPS, Systèmes de contraintes, Plans parallèles, Indépendance, Moindre engagement, Autorisation, Graphplan, Logique propositionnelle, Satisfiabilité, Satplan, Davis et Putnam.

DISCIPLINE ADMINISTRATIVE

Informatique

INTITULÉ ET ADRESSE DE L'U.F.R. OU DU LABORATOIRE

Institut de Recherche en Informatique de Toulouse, UMR 5505 CNRS−INP−UPS, Université Paul Sabatier, 118 Route de Narbonne, 31062 Toulouse Cedex 4