# Planification SAT : Amélioration des codages et traduction automatique

# Planning as satisfiability : improvements of encodings and automated translation

F. Maris[1] P. Régnier[1] V. Vidal[2]

[1] IRIT - Université Paul Sabatier
118, route de Narbonne
31062 TOULOUSE Cedex 04
{maris, regnier}@irit.fr

[2] CRIL - Université d'Artois
rue de l'Université - SP 16
62307 Lens Cedex
vidal@cril.univ-artois.fr

## Résumé

*Cet article présente notre synthèse des travaux actuels sur la planification par satisfaction de bases de clauses (planification SAT). Nous y détaillons les améliorations que nous avons apportées aux codages classiques dans les espaces d'états, de plans et au codage du graphe de planification. Celles-ci sont essentiellement basées sur l'introduction du parallélisme et sur l'utilisation de la relation d'autorisation à la place de celle d'indépendance. Nous présentons notre planificateur TSP (Tunable SAT Planner) qui intègre un traducteur permettant d'automatiser toutes ces méthodes de résolution. Chacun des différents codages peut être ainsi indifféremment utilisé pour traiter un problème de planification donné : la base de clauses automatiquement fournie par le traducteur est donnée à un solveur SAT, et fournit, après traduction inverse du modèle qu'il renvoi, le plan-solution. Cet outil nous permet de montrer l'impact de nos améliorations sur les benchmarks classiques de planification.*

## Mots Clef

Planification, SAT, Codages, Traducteur, SATPLAN, BLACKBOX, GRAPHPLAN.

## Abstract

*This paper presents our synthesis of actual works about planning as satisfiability. We detail the improvements we brought to classical state-space, plan-space and graphplan-based encodings. They are essentially based on the introduction of parallelism and the use of the authorization relation instead of independence. We present our planner TSP (Tunable SAT Planner) which includes a translator that automatizes all these resolution methods. Each of these encodings can thus be used to solve a given planning problem : the base of clauses computed by the translator is given to a SAT solver, which returns, after an inverse translation, the solution plan. This tool allows us to show the impact of our improvements on classical planning benchmarks.*

## Keywords

Planning, SAT, Encodings, Translator, SATPLAN, BLACKBOX, GRAPHPLAN.

# 1 Introduction

## 1.1 La planification STRIPS

La planification–discipline de l'IA–cherche à concevoir des systèmes capables de générer automatiquement, grâce à une procédure formalisée, un résultat articulé, sous la forme d'un système intégré de décisions appelé plan-solution. Ce dernier est une collection organisée de descriptions d'opérations ; il est destiné à guider l'action d'un ou plusieurs agents exécuteurs (systèmes robotiques ou humains) qui doivent agir dans un monde particulier pour atteindre un but préalablement défini. L'exécution est la réalisation d'actions effectuée suivant les directives du plan. Elle vise à réaliser la prédiction que constitue ce plan ; lorsqu'elle est conforme à ce qu'il indique, elle doit permettre (en l'absence d'événements imprévus et si la modélisation du monde est pertinente) de faire évoluer l'univers de l'état initial vers un état satisfaisant le but.
La planification de type STRIPS [13] a maintenant 30 années d'existence. Depuis ce système fondateur, et dans le cadre "classique" de la planification qu'il a délimité–environnement statique, observabilité totale, agent omniscient, actions atomiques et déterministes–de très gros progrès ont été réalisés. Il y a encore huit ans, depuis [9], jusqu'à [17], l'approche de la planification qui était l'objet essentiel des recherches était fondée sur les stratégies de raffinements dans les espaces de plans partiels. Au sein de la communauté, les autres approches étaient marginalisées. En 1995 cependant, l'apparition du planificateur GRAPHPLAN [4, 5] fut à l'origine d'un bouleversement de cet ordre bien établi : ses performances et les idées qui guidaient sa conception ont inspiré un grand nombre de travaux. Ceux-ci ont permis aux algorithmes de planification STRIPS d'augmenter leurs performances de manière si importante que l'on peut maintenant envisager des applications réelles à moyen terme. Actuellement,

certains de ces nouveaux algorithmes, grâce à l'utilisation d'heuristiques inspirées par GRAPHPLAN, permettent de résoudre en quelques dizaines de minutes (pour les ordinateurs classiques) des problèmes complexes qui, même s'ils sont issus de benchmarks, sont largement hors de portée de l'humain. Les plans-solutions générés, bien que non optimaux, comportent quelques milliers d'actions. Plusieurs familles d'algorithmes sont maintenant en concurrence. Chacune possède des avantages et inconvénients qui lui sont propres et les planificateurs font l'objet d'évaluations comparatives régulières dans la compétition internationale IPC [www.dur.ac.uk/d.p.long/competition.html].

### 1.2 Organisation de l'article

Cet article résume des travaux dont on pourra trouver l'intégralité dans [28, 24] et [www.irit.fr/∼Pierre.Regnier]. Dans le chapitre suivant, nous exposons les principes essentiels des algorithmes de planification STRIPS employés par la planification SAT. Le chapitre 3 donne des définitions préliminaires indispensables à la compréhension de l'article. Nous abordons ensuite l'utilisation des codages utilisés dans SATPLAN (chapitre 4) et BLACKBOX (chapitre 5) et les améliorations que nous leur avons apportées. Le chapitre 6 décrit notre planificateur TSP (Tunable SAT Planner) et la comparaison expérimentale qui montre les gains importants procurés par nos modifications. Nous terminons par une prospective.

## 2 Planification STRIPS : algorithmes essentiels

### 2.1 Recherche dans les espaces d'états

Dans la planification par recherche dans les espaces d'états, les noeuds du graphe de recherche représentent les états successifs du monde du problème de planification et les arcs, les actions qui permettent de passer d'un état à un autre. L'algorithme tente de construire un chemin (plan-solution) qui permette de passer de l'état initial du problème à un état-but. Il se termine quand l'état courant contient les buts à atteindre. Parmi les planificateurs actuels les plus performants qui sont basés sur cette technique, on trouve HSP et HSPr [6], FF [15], AltAlt [26], et YAHSP [30]. A partir d'un graphe de planification du problème, ils calculent une heuristique généralement très informative et le plus souvent non admissible inspirée de GRAPHPLAN [4, 5] qui leur permet de guider la recherche de manière très efficace. En contrepartie, ils n'offrent le plus souvent pas de garantie sur l'optimalité en nombre d'actions des plans-solutions. De plus, même si un post-traitement permet de les paralléliser [27, 1], ces plans demeurent généralement moins flexibles que ceux obtenus par d'autres approches.

### 2.2 Recherche dans les espaces de plans

Dans la planification par recherche dans les espaces de plans partiels, les noeuds du graphe de recherche représentent des plans partiels et les arcs, les opérations d'extension de ces plans. L'algorithme cherche à étendre un plan initial qui représente le problème posé, pour obtenir un plan-solution. Il se termine lorsque toutes les préconditions de toutes les actions présentes dans le plan courant sont produites par d'autres actions du plan et que ce dernier ne comporte plus aucune action qui puisse potentiellement interférer avec une autre. Cette technique, introduite par [9], a été l'objet de très nombreux travaux [17]. Elle fut délaissée après l'arrivée de GRAPHPLAN [4, 5], de SATPLAN [18], de BLACKBOX [20] et des planificateurs par recherche heuristique dans les espaces d'états [6, 31]. Elle connaît actuellement un renouveau avec le planificateur REPOP [25] qui utilise des heuristiques basées sur le calcul du graphe de planification de GRAPHPLAN qui guident efficacement la recherche. Cette approche demeure ainsi intéressante car elle permet d'obtenir des plans possédant un haut degré de flexibilité.

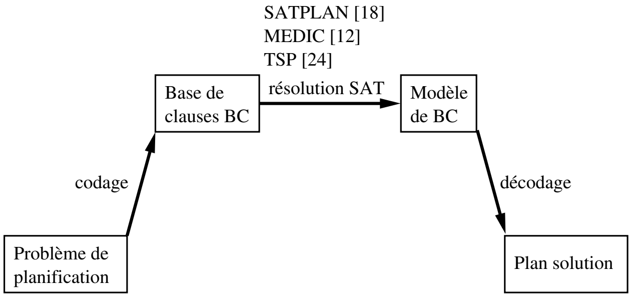
### 2.3 GRAPHPLAN

GRAPHPLAN [4, 5] (et les algorithmes qui en dérivent : IPP [21], STAN [14] et LCGP [8, 28]) travaille en développant d'abord, niveau par niveau et en temps polynomial (fonction du nombre total de fluents du problème, de ceux de l'état initial, et du nombre d'actions), un espace de recherche compact appelé graphe de planification. Pour cette phase (dite de construction), il n'utilise pas toutes les informations indispensables à l'obtention d'un plan-solution qui sont prises en compte au fur et à mesure par les autres approches (exclusions mutuelles entre variables d'état ou actions). Ces contraintes sont simplement calculées et enregistrées à chaque niveau du graphe sous forme d'exclusions mutuelles à la manière des CSP [16]. Cet espace de recherche se développe donc plus facilement, mais, en contrepartie, son achèvement ne coïncide plus avec l'obtention d'un plan-solution qu'une deuxième phase (dite d'extraction) cherche alors à extraire à partir du graphe de planification et des contraintes enregistrées. Les plans ainsi générés peuvent être représentés par des séquences d'ensembles d'actions indépendantes, chaque ensemble correspondant à un niveau du graphe de planification. Cette approche permet de générer des plans optimaux en nombre de niveaux et possédant une bonne flexibilité. Celle-ci reste généralement moins bonne que celle obtenue par recherche dans les espaces de plans partiels [25]).

### 2.4 Les principes de la planification SAT

En 1992, les approches classiques de la planification, basées sur la recherche dans les espaces de plans partiels et dans les espaces d'états, n'offraient pas de performances convainquantes. Kautz et Selman [19] ont alors proposé, avec l'approche SATPLAN, de profiter des progrès constants effectués dans le domaine des techniques SAT pour résoudre efficacement les problèmes de planification. L'utilisation de ces techniques offre également un cadre formel permettant l'utilisation naturelle de connaissances sur le domaine, connaissances représentables sous forme de clauses exprimant des contraintes entre actions et fluents.

Les plans-solutions potentiels au problème de planification posé peuvent être représentés par différents codages dont chacun est une manière d'appréhender la structure de ces plans (codages dans les espaces d'états, dans les espaces de plans). La traduction du problème, par le biais d'un codage donné, fournit une base de clauses qui est ensuite donnée en entrée à un solveur SAT. Celui-ci cherche alors un modèle de cette base de clauses. La traduction inverse du modèle trouvé par le solveur (lorsqu'il existe), fournit un plan-solution au problème initial. Les différents codages fournissent des bases de clauses plus ou moins compactes (en nombre de variables, de clauses) et l'expérimentation nous permettra d'évaluer leurs performances respectives.

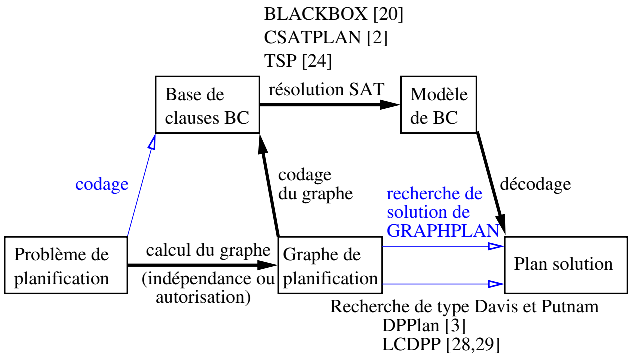



Comme les approches SATPLAN travaillent sur un ensemble fini de variables propositionnelles et que deux actions identiques peuvent apparaître à des endroits différents d'un même plan, on les différencie en leur associant des propositions différentes. La longueur d'un plan-solution ne pouvant être connue à l'avance, on ne peut créer un codage unique permettant de résoudre le problème posé (il faudrait une infinité de propositions représentant toutes les actions de tous les plans possibles). On emploie donc un codage représentant tous les plans d'une longueur fixée en commençant à une longueur minimale (soit 1, soit une longueur obtenue grâce à la construction d'un graphe de planification permettant d'obtenir les fluents du but). Tant qu'un plan-solution n'est pas trouvé par la résolution de ce codage, cette longueur est augmentée. Les plans-solutions correspondent aux modèles de la base de clauses issue du codage ; ils sont donc optimaux en nombre de niveaux. Les approches de type SATPLAN donnent des planificateurs incomplets puisque lorsqu'il n'y a pas de solution, la recherche ne s'arrête pas.

L'autre approche de la planification SAT, utilisée dans BLACKBOX [20] et CSATPLAN [2], est celle qui a donné les meilleurs résultats. Contrairement à SATPLAN, elle ne code pas directement la forme d'un plan-solution potentiel (flèche "codage" sur le schéma ci-après) mais utilise la structure du graphe de planification construit par GRAPHPLAN pour en extraire un plan-solution. Cette phase d'extraction, originellement dévolue dans GRAPHPLAN à une procédure spécifique (flèche "recherche de solution de GRAPHPLAN" sur le schéma) est réalisée, dans BLACKBOX, par un prouveur SAT. Les plans-solutions obtenus sont donc optimaux en nombre de niveaux, comme pour GRAPHPLAN. Le graphe peut être représenté (flèche "codage du graphe" sur le schéma) par :

– le codage de [20] des actions, des fluents, et des exclusions mutuelles des différents niveaux (Codage 6),
– le codage des seules actions et exclusions mutuelles des différents niveaux (Codage 7), puisque la connaissance des actions permet la déduction des fluents.

Une autre approche semblable mais qui n'utilise pas de prouveur SAT consiste à implémenter une procédure de Davis et Putnam [11, 10] travaillant directement sur le graphe de planification pour en extraire un plan-solution (flèche "Davis et Putnam" sur le schéma), cf. [3, 28, 29].




Pour chercher un modèle de la base de clauses, les approches SAT utilisent des prouveurs SAT. Depuis une dizaine d'années, ces derniers ont fait des progrès considérables et ils sont régulièrement comparés. La dernière compétition [www.lri.fr/∼simon/contest03/results/] qui se tenait en liaison avec la conférence SAT'03 (Mai 2003, S. Margherita Ligure, Portofino, Italie) montre que les progrès enregistrés dans ce domaine sont très importants. Plusieurs des nouveaux solveurs étaient ainsi capables de résoudre en 2003 et en moins de 15 mn des problèmes qui étaient restés insolubles en 6 heures lors de la précédente compétition SAT'02 (Mai 2002, Cincinatti, Ohio, USA). Pour la compétition 2003, plusieurs benchmarks de planification générés par notre système ont été testés (industrial/maris/CNF/ferry, industrial/maris/CNF/gripper, industrial/maris/CNF/hanoi). Certains d'entre eux parmi les plus difficiles ont ainsi pu être résolus, alors qu'ils ne l'étaient pas précédemment. Bien que n'importe quel type de solveur (complet ou utilisant des méthodes locales) puisse être utilisé pour trouver un modèle, les plus performants sur les benchmarks structurés de problèmes de planification s'avèrent être les solveurs qui utilisent des méthodes complètes comme Forklift de Goldberg et Novikov.

Différents codages pour la planification SAT ont été proposés depuis l'article original de [19] jusqu'à [12, 20, 23]. Les modifications que nous apportons simplifient tous ces codages, introduisent le parallélisme dans les codages d'espaces de plans, emploient la relation d'autorisation et produisent des codages plus compacts (en nombre de clauses, de variables et de niveaux des plans-solutions). L'expérimentation réalisée avec notre planificateur TSP montre une amélioration importante des performances. Faute de place, nous ne donnerons pas ici tous les codages testés mais seulement ceux qui se sont avérés être les plus performants. L'ensemble des codages est disponible dans [24, 28] et sur le site [www.irit.fr/∼Pierre.Regnier].

# 3 Définitions préliminaires

Nous donnons ici quelques définitions indispensables à la compréhension de l'article. Dans le cadre de la planification STRIPS, nous utilisons une logique du premier ordre $L$, construite à partir des vocabulaires $V_x$, $V_c$, $V_p$ qui dénotent respectivement des ensembles finis disjoints deux à deux de symboles de variables, de constantes et de prédicats. Nous n'utilisons pas de fonctions. Pour les notations utilisées dans les codages, nous utilisons les définitions classiques de logique propositionnelle. Pour deux formules $F_1$ et $F_2$ et une propriété $G$, "si $G$ est satisfaite, alors $F_1$ sinon $F_2$" s'écrira en utilisant la règle de réécriture suivante : $(G \mapsto F_1 \mid F_2)$.

**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. $Prec(o)$, $Add(o)$, $Del(o)$ dénotent respectivement les ensembles pr, ad, de et représentent les préconditions, ajouts et retraits de l'opérateur o.*

**Définition 2 (état, fluent)** *Un état E est un ensemble fini de formules atomiques de base (sans symbole de variable) du langage L. Une formule atomique de base est aussi appelée un fluent.*

**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, obtenue par l'application d'une substitution (définie dans le langage L) telle que $ad\theta$ et $de\theta$ sont des ensembles disjoints. $Prec(a)$, $Add(a)$ et $Del(a)$ dénotent respectivement les ensembles $pr\theta$, $ad\theta$, $de\theta$ et représentent les préconditions, ajouts et retraits de l'action a.*

Pour un ensemble d'actions $A = \{a_1, ..., a_n\}$, nous utiliserons les notations :
- $Prec(A) = Prec(a_1) \cup \ldots \cup Prec(a_n)$
- $Add(A) = Add(a_1) \cup \ldots \cup Add(a_n)$
- $Del(A) = Del(a_1) \cup \ldots \cup Del(a_n)$

**Définition 4 (application d'une action à un état)** *L'état $E'$ résultant de l'application de l'action a à un état E (noté $E' = E \uparrow a$) se calcule en retirant $Del(a)$ à l'état E et en rajoutant $Add(a)$ : $E' = (E - Del(a)) \cup Add(a)$.*

**Définition 5 (problème de planification)** *Un problème de planification $\Pi$ est un triplet $\langle O, I, G\rangle$ 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,*
- *G dénote un ensemble fini de fluents construits à partir du langage L qui représentent les buts du problème.*

Pour un problème de planification $\Pi$, nous utiliserons également les ensembles suivants :
- $F_p$ : ensemble des fluents préconditions des actions,
- $F_a$ : ensemble des fluents ajouts des actions,
- $F_d$ : ensemble des fluents retraits des actions.

Nous définissons maintenant la notion d'indépendance entre actions et pour un ensemble d'actions : lorsque cette relation est vérifiée, elle permet, quel que soit l'ordre d'exécution d'un ensemble d'actions indépendant $Q$, d'obtenir un seul et même état résultant.

**Définition 6 (indépendance entre deux actions)** *Deux actions a et b telles que $a \neq b$ sont indépendantes (noté a#b) ssi : $(Add(a) \cup Prec(a)) \cap Del(b) = \emptyset$ et $(Add(b) \cup Prec(b)) \cap Del(a) = \emptyset$*

**Définition 7 (ensemble d'actions indépendant)** *Un ensemble d'actions $Q = \{a_1, ..., a_n\}$ est indépendant ssi les actions qui le composent sont indépendantes deux à deux, c'est-à-dire ssi : $\forall \{a, b\} \subseteq Q, a \neq b \Rightarrow (Prec(a) \cup Add(a)) \cap Del(b) = \emptyset$*

**Définition 8 (application d'un ensemble d'actions indépendantes à un état)** *L'état $E'$ résultant de l'application d'un ensemble d'actions indépendantes Q à un état E (noté $E' = E \uparrow Q$) se calcule en retirant $Del(Q)$ à l'état initial et en rajoutant $Add(Q)$ : $E' = (E - Del(Q)) \cup Add(Q)$.*

Nous définissons maintenant la notion d'autorisation [7, 8] entre actions et pour un ensemble d'actions : lorsque cette relation, moins contraignante que celle d'indépendance, est vérifiée, elle permet, pour un ordre d'exécution déterminé d'un ensemble d'actions autorisé Q, d'obtenir un état résultant par un calcul identique à celui effectué pour l'indépendance.

**Définition 9 (autorisation entre deux actions)** *Deux actions a et b telles que $a \neq b$ sont autorisées (noté $a \angle b$) ssi : $Add(a) \cap Del(b) = \emptyset$ et $Prec(b) \cap Del(a) = \emptyset$*

**Définition 10 (ensemble d'actions autorisé)** *Un ensemble d'actions $Q = \{a_1, ..., a_n\}$ est autorisé ssi il existe un ordre d'exécution de ces actions tel que les ajouts des actions $a_1, \ldots, a_i$ soient conservés par $a_{i+1}$ et que l'exécution de $a_{i+1}$ reste possible après celle des actions qui la précèdent. C'est-à-dire ssi : $\forall i \in [1, n-1], Del(a_{i+1}) \cap (Add(a_1) \cup \ldots \cup Add(a_i)) = \emptyset$ et $Prec(a_{i+1}) \cap (Del(a_1) \cup \ldots \cup Del(a_i)) = \emptyset$*

**Définition 11 (application d'un ensemble d'actions autorisées à un état)** *L'état $E'$ résultant de l'application d'un ensemble d'actions autorisées Q à un état E (noté $E' = E \uparrow Q$) se calcule comme pour un ensemble d'actions indépendantes) : $E' = (E - Del(Q)) \cup Add(Q)$.*

# 4 Les différents codages SAT

## 4.1 SATPLAN : Codages dans les espaces d'états

Ces codages sont basés sur les transitions entre les niveaux successifs du plan, depuis l'état initial jusqu'au but. Le parallélisme y est codé grâce à la notion d'indépendance entre actions simultanées. Pour conserver les fluents non affectés par les actions du niveau suivant, on code la notion de frame-axiome. Nous décrivons d'abord la technique la plus efficace (en termes de compacité et de temps de résolution, cf. [23]) qui utilise des frame-axiomes explicatifs, ainsi que les modifications que nous lui avons apportées. Nous évoquons ensuite la technique de [20] qui utilise des no-ops.

**Codage dans les espaces d'états avec frame-axiomes explicatifs.** Les règles du codage dans les espaces d'états produisent des propositions de forme $A(i)$ qui représentent le fait que l'action $A$ est appliquée à un niveau $i$ du plan ssi $A(i)$ a la valeur de vérité vrai ($A$ est une action, $i$ un entier naturel), et des propositions de forme $f(i)$ qui représentent le fait que le fluent $f$ est présent au niveau $i$ ssi $f(i)$ a la valeur de vérité vrai ($f$ est un fluent, $i$ un entier naturel). Lorsque le fluent $f$ apparaît au niveau $i$, cela signifie qu'il est présent après l'application successive de toutes les actions associées à des propositions qui sont vraies, du niveau 1 jusqu'au niveau $i$. Le nombre d'actions et le nombre de niveaux étant finis, nous sommes dans le cadre de la logique propositionnelle. L'utilisation de parenthèses autorise une notation prédicative d'une logique d'ordre supérieur mais la base de clauses obtenue reste propositionnelle. Le codage de [23] comporte trois règles :

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 :
   - Frame-axiomes de retrait : 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).
   - Frame-axiomes d'ajout : 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 doit être vraie (une action au moins qui l'établit doit avoir été appliquée).

Ce codage ne prend pas en compte les interactions croisées : interdire à une action de retirer une précondition d'une autre action à un même niveau nécessite une quatrième règle, décrite dans [22], et qui fournit un codage complet. Ce codage produit cependant des propositions et clauses inutiles éliminées par notre version (cf. Codage 1) : notre quatrième règle n'ajoute une clause d'exclusion mutuelle entre actions que si cette dernière n'a pas déjà été ajoutée par la règle 2 qui prend en compte les effets contradictoires (l'exclusion n'est ajoutée que si les deux actions ont une interaction croisée et pas d'effets contradictoires). La troisième règle est aussi modifiée pour supprimer des clauses inutiles lorsqu'un fluent n'est ajouté ni par l'état initial ni par une action, ou lorsqu'il n'est retiré par aucune action. Pour les frame-axiomes de retrait, il faut permettre au fluent d'exister à un moment donné, il doit donc appartenir à $I$ ou à $F_a$. Il lui faut aussi pouvoir être retiré par une action, il doit donc appartenir à $F_d$. Il doit donc être un élément de $(I \cup F_a) \cap F_d$. Pour les frame-axiomes d'ajout, il faut permettre au fluent d'exister à un moment donné, il ne doit donc pas appartenir à $I$ ou être élément de $F_d$. Il lui faut aussi pouvoir être ajouté par une action et donc être élément de $F_a$. Il doit donc appartenir à $((F-I) \cup F_d) \cap F_a$.

**Introduction de la relation d'autorisation.** Dans le codage précédent, la règle 4 interdit l'apparition simultanée de deux actions non indépendantes à un même niveau. Or la présence à un même niveau de deux actions non indépendantes mais autorisées peut souvent permettre d'obtenir un état résultant identique à celui qui est obtenu en répartissant ces deux actions sur deux niveaux successifs [7, 8]. En remplaçant la relation d'indépendance par celle d'autorisation, on réduit le nombre de niveaux du plan-solution, la taille de la base de clauses qui lui correspond, et on améliore donc les performances (cf. partie 6). Il faut seulement rajouter la vérification, pour le plan correspondant au modèle renvoyé par le solveur, de l'existence d'une séquence d'actions autorisée pour chacun des niveaux successifs. Ce test se fait en un temps polynomial, et indépendamment du solveur. Si cette condition est vérifiée, le plan retourné est un plan-solution, sinon, on rajoute une clause interdisant le modèle associé et on relance le solveur au même niveau. Ce cas particulier ne s'est cependant jamais présenté sur aucun des benchmarks testés. La nouvelle règle 4 (qui code le fait qu'aucune action n'autorise l'autre) est représentée dans le Codage 2.

**Autres codages dans les espaces d'états.** On peut utiliser des no-ops à la place des frame-axiomes explicatifs. Cette idée vient du planificateur GRAPHPLAN et a été proposée pour la première fois dans [18]. Ce codage produit plus de propositions et de clauses que celui des espaces d'états avec frame-axiomes explicatifs (il élimine des clauses de taille supérieure ou égale à 3 produites par la règle 3 du codage avec frame-axiomes et rajoute des clauses de taille 2). Faute de place, nous ne le détaillerons pas ici car il est moins performant (cf. [28, 24] pour sa description complète et les tests).

## 4.2 SATPLAN : Codages dans les espaces de plans

Ces codages ne traduisent plus seulement les transitions qui s'opèrent entre niveaux successifs du plan, mais expriment maintenant les relations de causalité entre ses actions (cf. [32] pour une introduction à la planification dans les espaces de plans). Dans les espaces d'états, l'application des actions est envisagée séquentiellement : une action peut apparaître à un niveau du plan parce que ses préconditions sont satisfaites au niveau précédent. Ici, une action ap-

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. $\bigwedge_{i\in[1,k]} \bigwedge_{a\in O} \left(a_i \Rightarrow \left(\bigwedge_{f\in Prec(a)} f(i-1)\right) \wedge \left(\bigwedge_{f\in Add(a)} f(i)\right) \wedge \left(\bigwedge_{f\in Del(a)} \neg f(i)\right)\right)$
3. $\bigwedge_{i\in[1,k]} \bigwedge_{f\in((I\cup F_a)\cap F_d)} \left(f(i-1) \wedge \neg f(i) \Rightarrow_{a\in O f\in Del(a)} \bigvee a(i)\right)$
   $\bigwedge_{i\in[1,k]} \bigwedge_{f\in(((F-I)\cup F_d)\cap F_a)} \left(\neg f(i-1) \wedge f(i) \Rightarrow_{a\in O|f\in Add(a)} \bigvee a(i)\right)$
4. $\bigwedge_{i\in[1,k]} \bigwedge_{(a_m,a_n)\in O^2|m<n\wedge((Prec(a_m)\cap Del(a_n)\neq\emptyset)\vee(Del(a_m)\cap Prec(a_n)\neq\emptyset))\wedge((Add(a_m)\cap Del(a_n)=\emptyset)\wedge(Del(a_m)\cap Add(a_n)=\emptyset))} (\neg a_m(i) \vee \neg a_n(i))$

Codage 1 : espaces d'états, frame-axiomes explicatifs

4. $\bigwedge_{i\in[1,k]} \bigwedge_{(a_m,a_n)\in O^2|m<n\wedge((Add(a_m)\cap Del(a_n)\neq\emptyset)\vee(Del(a_m)\cap Prec(a_n)\neq\emptyset))\wedge((Del(a_m)\cap Add(a_n)\neq\emptyset)\vee(Prec(a_m)\cap Del(a_n)\neq\emptyset))} (\neg a_m(i) \vee \neg a_n(i))$

Codage 2 : changement de la règle 4 du Codage 1 pour l'autorisation

1. $\bigwedge_{i\in[1,k]} \left(\bigvee_{a\in O} (a \in p_i)\right)$
2. $\bigwedge_{i\in[1,k]} \bigwedge_{(a_m,a_n)\in O^2|m<n\wedge((Add(a_m)\cup Prec(a_m))\cap Del(a_n)\neq\emptyset\vee((Add(a_n)\cup Prec(a_n))\cap Del(a_m)\neq\emptyset))} (\neg(a_m \in p_i) \vee \neg(a_n \in p_i))$
3. $\left(\bigwedge_{f\in I} Adds(Init,f)\right) \wedge \left(\bigwedge_{f\in(F-I)} \neg Adds(Init,f)\right) \wedge \left(\bigwedge_{f\in G} Needs(Goal,f)\right)$
4. $\bigwedge_{i\in[1,k]} \bigwedge_{f\in F_a} \left(Adds(p_i,f) \Leftrightarrow_{a\in O|f\in Add(a)} \bigvee (a\in p_i)\right) \wedge \bigwedge_{i\in[1,k]} \bigwedge_{f\in F_d} \left(Dels(p_i,f) \Leftrightarrow_{a\in O|f\in Del(a)} \bigvee (a\in p_i)\right)$
   $\bigwedge_{i\in[1,k]} \bigwedge_{f\in F_p} \left(Needs(p_i,f) \Leftrightarrow_{a\in O|f\in Prec(a)} \bigvee (a\in p_i)\right)$

Codage 3 : espaces de plans, partie commune

paraît à un niveau du plan parce qu'une action d'un niveau antérieur (pas forcément le niveau précédent) établit ses préconditions et qu'une action qui suit réclame ses effets. Le plan est considéré comme un ensemble interdépendant d'actions. L'existence de différents codages découle de la traduction des différentes stratégies de recherche utilisés dans les espaces de plans. Mali et Kambhampati [23] ont montré que ces codages étaient moins performants, en termes de compacité et de temps de résolution, que le meilleur codage dans les espaces d'états (celui avec frame-axiomes explicatifs). Nous décrivons d'abord la technique la plus performante dans les espaces de plans, qui code la technique du "chevalier blanc", ainsi que les modifications que nous lui avons apportées. Nous évoquons ensuite, sans les détailler, faute de place, les techniques qui utilisent les liens causaux et la protection d'intervalles.

Dans tous ces codages, les ensembles d'actions indépendantes deux à deux qui peuvent faire partie d'un plan sont associés à des symboles d'étapes. Un ordre sur ces étapes détermine un ordre total pour l'exécution des ensembles d'actions indépendantes. Les règles de ces différents codages produisent plusieurs types de propositions :

1. Les propositions $(a \in p)$, où $p$ est une étape du plan et $a$ une action, sont vraies ssi l'action $a$ appartient à l'étape $p$. Ces propositions permettent d'autoriser le parallélisme.
2. Les propositions $Adds(p,f)$, où $p$ est une étape du plan et $f$ est un fluent, sont vraies ssi l'étape $p$ contient une action qui ajoute le fluent $f$.
3. Les propositions $Needs(p,f)$, où $p$ est une étape du plan et $f$ est un fluent, sont vraies ssi l'étape $p$ contient une action qui nécessite $f$ comme précondition.
4. Les propositions $Dels(p,f)$, où $p$ est une étape du plan et $f$ est un fluent, sont vraies ssi l'étape $p$ contient une action qui détruit le fluent $f$.
5. Les propositions $p_i \xrightarrow{f} p_j$, où $p_i$ et $p_j$ sont deux étapes du plan et $f$ un fluent, représentent un lien causal et sont vraies ssi l'étape $p_i$ produit le fluent $f$ qui est une précondition de l'étape $p_j$.
6. Les propositions $p_i \prec p_j$, où $p_i$ et $p_j$ sont deux étapes du plan, sont vraies ssi l'étape $p_i$ précède l'étape $p_j$ ; les actions de l'ensemble d'actions indépendantes associées à $p_i$ doivent donc être exécutées avant les actions associées à $p_j$. Ces propositions sont utilisées pour traduire un ordre partiel sur les étapes.

**Partie commune des codages dans les espaces de plans.** Elle permet d'établir une correspondance entre toutes les actions disponibles et celles qui font partie des étapes des plans-solutions. Son codage par [23] utilise cinq règles :

1. Équivalence entre étape et action : une étape peut être soit une action, soit $\Phi$ (l'action nulle, ou no-op).
2. Unicité des étapes : une étape correspond à une et une seule action (ou à $\Phi$) à un niveau donné du plan. Cette règle interdisant le parallélisme entre actions est supprimée de notre codage.
3. Étape $Init$ de niveau 0 qui produit l'état initial : $Adds(Init,f)$ est vraie pour tous les fluents $f$ de $I$

et fausse pour les autres. Elle définit aussi l'étape *Goal* de niveau $k+1$, où $k$ est le niveau le plus haut considéré pour une autre étape, dont les préconditions sont les fluents de l'état-but : $Needs(Goal, f)$ est vraie pour tous les fluents $f$ de l'état-but.

4. Si l'étape $p$ est associée à une action $a$ (différente de $\Phi$), alors la conjonction des ajouts, des préconditions et des retraits de $a$ est vraie : $Adds(p, f)$ est vraie pour les fluents $f$ de $Add(a)$, $Needs(p, f)$ est vraie pour les fluents $f$ de $Prec(a)$, $Dels(p, f)$ est vraie pour les fluents $f$ de $Del(a)$.
5. Si une étape ajoute, retire ou a pour précondition un fluent, alors elle peut correspondre à n'importe quelle action ayant un comportement identique sur ce fluent.

Dans notre codage de la partie commune (Codage 3), l'action $\Phi$ n'est plus considérée comme une action particulière et elle peut ou non faire partie de l'ensemble $O$ de toutes les actions. Par la suite, nous ne la considérerons pas car son introduction n'apporte rien et dégrade généralement les performances. La première règle est conservée sous forme modifiée pour obtenir un nombre d'étapes égal au nombre de niveaux du plan-solution. La deuxième règle du codage de [23] est modifiée pour autoriser le parallélisme.

**Codage du "chevalier blanc".** Ce codage (Codage 4) est actuellement le plus compact des codages dans les espaces de plans en nombre de variables et de clauses produite ; il est également le plus performant par son temps de résolution. Il exprime directement les liens causaux : si une étape a besoin d'un fluent, c'est qu'une étape précédente doit l'avoir créé. On n'utilise ainsi que les variables déjà présentes dans la partie commune. La protection d'intervalles se réalise en codant la technique du "chevalier blanc" introduite par le planificateur TWEAK [9]. Pour cela, on ajoute deux règles à la partie commune :

1. Établissement des préconditions : la précondition d'une étape d'un niveau $i$ doit être ajoutée par une étape précédente.
2. Chevalier blanc : si une étape (ou l'étape but *Goal*) a pour précondition le fluent $f$ au niveau $i$, et qu'une autre le retire au niveau $j$ avant que la première puisse l'utiliser, alors il doit y avoir une troisième étape qui rétablit $f$ à un niveau $q$ tel que $j < q < i$.

Les corrections que nous avons apportées aux règles du codage de [23] permettent de supprimer des clauses inutiles et des modèles qui correspondent à des plans ne résolvant pas le problème.

**Introduction de la relation d'autorisation.** Comme pour les codages dans les espaces d'états (cf. partie 4.2), nous introduisons la relation d'autorisation pour réduire le nombre de niveaux du plan-solution, la taille de la base de clauses qui lui correspond, et améliorer les performances (cf. partie 6). Pour cela, la règle 2 de la partie commune est modifiée (cf. Codage 5).

**Autres codages dans les espaces de plans.** Les deux codages que nous évoquons maintenant utilisent les liens causaux et la protection d'intervalle. Faute de place, nous ne les détaillerons pas ici car ils s'avèrent être moins performants que le codage du "chevalier blanc" (cf. [28, 24] pour leur description complète et les tests).
*Codage par liens causaux, protection d'intervalles et ordre partiel* : l'établissement des préconditions se fait par le codage direct des liens causaux. La proposition $p_i \xrightarrow{f} p_j$ traduit le fait que l'étape $p_i$ produit le fluent $f$, précondition de l'étape $p_j$. La protection de $f$ dans l'intervalle compris entre les deux étapes se fait par promotion (l'étape menaçant $f$ est placée avant $p_i$), ou par rétrogradation (l'étape menaçant $f$ est placée après $p_j$). Les indices des étapes ne correspondent pas à l'ordre d'exécution des actions qui est donné par une relation de précédence de la forme $p_i \prec p_j$. Notre codage corrige le codage de [23] en supprimant des clauses inutiles et des modèles ne correspondant pas à des plans-solutions.
*Codage par liens causaux, protection d'intervalles et étapes contiguës* : ce codage simplifie le précédent dans lequel plusieurs modèles correspondent au même plan-solution, à la numérotation des étapes près. Les étapes doivent maintenant suivre un ordre prédéfini d'indice croissant, ce qui rend inutile la relation de précédence. La protection d'intervalles ne nécessite plus la promotion ou la rétrogradation avec un ordre partiel sur les étapes, mais se réalise en interdisant qu'une étape comprise dans un intervalle défini par un lien causal ne le menace. Notre codage apporte des corrections similaires aux précédentes.

## 4.3 BLACKBOX : les différents codages

**Codage du graphe avec actions, fluents et mutex.** Le codage de [20] contient quatre règles :

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, ses préconditions sont vraies au niveau précédent.
3. Établissement des fluents : si un fluent est vrai à un niveau supérieur à l'état initial, la disjonction des actions qui peuvent le produire au niveau précédent est vraie (no-ops compris s'il est présent à ce niveau). Ces clauses correspondent aux arcs d'ajout calculés à la construction du graphe.
4. Exclusions mutuelles : les actions mutuellement exclusives ne peuvent pas être vraies en même temps. Ces mutex concernent l'indépendance entre actions et les mutex d'atteignabilité trouvés lors de la construction du graphe.

**Codage du graphe avec actions et mutex.** Notre codage simplifie le précédent et conduit aux trois règles suivantes :

1. But : pour chacun des fluents du but, au moins une action qui le produit doit avoir été appliquée.

1. $\bigwedge_{i\in[1,k]} \bigwedge_{f\in F_p} \Big[Needs(p_i,f) \Rightarrow \Big(f \in F_a \mapsto \bigvee_{j\in[1,i-1]} Adds(p_j,f) \mid \bot\Big) \vee (f \in I \Rightarrow Adds(Init,f) \mid \bot)\Big]$
   $\bigwedge_{f\in G} \Big[Needs(Goal,f) \Rightarrow \Big(f \in F_a \mapsto \bigvee_{i\in[1,k]} Adds(p_i,f) \mid \bot\Big) \vee (f \in I \mapsto Adds(Init,f) \mid \bot)\Big]$
2. $\bigwedge_{i\in[2,k]} \bigwedge_{j\in[1,i-1]} \bigwedge_{f\in(F_p\cap F_d)} \Big[Needs(p_i,f) \wedge Dels(p_j,f) \Rightarrow \Big(f \in F_a \mapsto \bigvee_{q\in[j+1,i-1]} Adds(p_q,f) \mid \bot\Big)\Big]$
   $\bigwedge_{i\in[1,k]} \bigwedge_{f\in(G\cap F_d)} \Big[Needs(Goal,f) \wedge Dels(p_i,f) \Rightarrow \Big(f \in F_a \mapsto \bigvee_{j\in[i+1,k]} Adds(p_i,f) \mid \bot\Big)\Big]$

Codage 4 : espaces de plans, chevalier blanc

2. $\bigwedge_{i\in[1,k]} \bigwedge_{\substack{(a_m,a_n)\in O^2 \mid m<n \wedge ((Add(a_m)\cap Del(a_n)\neq\emptyset)\vee(Del(a_m)\cap Prec(a_n)\neq\emptyset))\wedge \\ ((Del(a_m)\cap Add(a_n)\neq\emptyset)\vee(Prec(a_m)\cap Del(a_n)\neq\emptyset))}} (\neg(a_m \in p_i) \vee \neg(a_n \in p_i))$

Codage 5 : changement de la règle 2 du Codage 3 pour l'autorisation

1. $\Big(\bigwedge_{n_f\in Init(GP)} n_f\Big) \wedge \Big(\bigwedge_{n_f\in But(GP)} n_f\Big)$
2. $\bigwedge_{(n_f,n_a)\in ArcsPrec(GP)} (n_a \Rightarrow n_f)$
3. $\bigwedge_{n_f\in(NoeudsF(GP)\setminus Init(PG))} \Big(n_f \Rightarrow \bigvee_{(n_a,n_f)\in ArcsAdd(GP)} n_a\Big)$
4. $\bigwedge_{\{n_a,n_b\}\in MutexA(PG)} (\neg n_a \vee \neg n_b)$

Codage 6 : [20] graphe, actions, fluents et mutex

2. Établissement des préconditions : les préconditions de chaque action du plan, si elles ne sont pas dans l'état initial, doivent être établies par au moins une action.
3. Exclusions mutuelles : règle 4 du codage de [20].

1. $\bigwedge_{n_f\in But(GP)} \Big(\bigvee_{(n_a,n_f)\in ArcsAdd(GP)} n_a\Big)$
2. $\bigwedge_{(n_f,n_a)\in ArcsPrec(GP)} \Big(\bigvee_{(n_a,n_f)\in ArcsAdd(GP)} n_a\Big)$
3. $\bigwedge_{\{n_a,n_b\}\in MutexA(PG)} (\neg n_a \vee \neg n_b)$

Codage 7 : graphe, actions et mutex

Ces deux formes de codage du graphe de planification diffèrent par le nombre de variables et de clauses produites pour un même problème (nombre de variables moindre et nombre de clauses plus important pour notre codage).

**Introduction de la relation d'autorisation.** La seule modification à apporter, pour introduire l'autorisation, consiste à utiliser un graphe construit avec cette relation à la place de celle d'indépendance [7, 8]. Pour la traduction, on utilisera ensuite l'ensemble des actions mutex suivant la relation d'autorisation à la place des actions mutex suivant la relation d'indépendance.

# 5 Résultats expérimentaux

## 5.1 Le planificateur TSP

Pour étudier équitablement l'impact de nos modifications, nous avons implémenté le planificateur TSP (Tunable SAT Planner) qui intègre un traducteur permettant d'automatiser la planification SAT et peut indifféremment utiliser chacun des codages précédents. TSP peut aussi être employé pour produire des benchmarks pour les solveurs SAT. Le traducteur est écrit en C, le générateur d'instances et la construction du graphe en Caml ; ces programmes s'exécutent en temps polynomial. TSP est téléchargeable sur le site [www.irit.fr/∼Pierre.Regnier], il fonctionne sur PC sous linux, ou sur station Sun sous Solaris. Il prend en entrée le domaine et le problème de planification (fichiers PDDL), le choix des règles de codage à utiliser (au format TSPL, cf. [24]) et celui du solveur à employer. La complexité maximale en temps pour l'interprétation des codages par TSP est en $O(K^2.M^2)$ ou $O(K^2.M^3)$ suivant les codages ($K$ est la limite supérieure de longueur de plan recherché et $M$ le plus grand cardinal des ensembles définis). A partir des fichiers PDDL, le générateur d'instances produit le problème instancié :

- Pour les codages SATPLAN, le traducteur l'utilise suivant les règles de codage choisies, pour produire la base de clauses correspondant aux plans-solutions potentiels d'une longueur fixée (1 par défaut ou la longueur minimale d'un plan-solution si on utilise le graphe de planification pour la déterminer). TSP lance ensuite le solveur SAT pour en chercher un modèle. Si il n'en trouve pas, le traducteur incrémente la longueur du plan cherché en régénérant la base et en relançant le solveur, jusqu'à ce qu'il trouve un modèle ou qu'il arrive à une borne supérieure prédéfinie (arbitrairement fixée ou qui peut être déterminée par le niveau de stabilisation du graphe).
- Pour les codages BLACKBOX, TSP construit le graphe de planification (indépendant ou autorisé) puis le traducteur l'utilise, suivant les règles de codage choisies, pour produire la base de clauses correspondant aux plans-solutions potentiels susceptibles d'en être extraits. TSP lance ensuite le solveur SAT pour en chercher un modèle. Si il n'en trouve pas, TSP poursuit la construction du graphe au niveau suivant et il itère le processus jusqu'à ce qu'il trouve un modèle ou que le graphe se stabilise sans solution au problème posé.

Dans tous les cas, lorsqu'un modèle est trouvé, TSP utilise un algorithme linéaire de traduction inverse pour obtenir le plan-solution correspondant.

## 5.2 Conditions des tests

Les tests ont été effectués sur un PC sous Linux Red-Hat 6.1 qui intègre un CPU Athlon 500 et 256 Mo

| | Variables | | Clauses | | | Temps CPU(s) | | | Niveaux | |
|---|---|---|---|---|---|---|---|---|---|---|
| | MK99/Cod. 1 | Cod. 2 | MK99 | Cod. 1 | Cod. 2 | MK99 | Cod. 1 | Cod. 2 | MK99/Cod. 1 | Cod. 2 |
| Bworld-3c | 132 | 102 | 1046 | 842 | 590 | 0,38 | 0,33 | 0,2 | 4 | 3 |
| Bworld-4c | 360 | 292 | 5803 | 4213 | 2799 | 3,77 | 3,15 | 1,73 | 5 | 4 |
| Bworld-5c | 810 | 680 | 22834 | 15934 | 9834 | 38,1 | 32,69 | 17,4 | 6 | 5 |
| Bworld-6c | 1374 | 708 | 60491 | 41681 | 14024 | 228,71 | 207,3 | 37,84 | 6 | 3 |
| Bworld-7c | * | 1106 | * | * | 29375 | * | * | 170,4 | * | 3 |
| Bworld-8c | * | 1632 | * | * | 56034 | * | * | 612,42 | * | 3 |
| Gripper-1 | 62 | 44 | 267 | 249 | 171 | 0,12 | 0,12 | 0,08 | 3 | 2 |
| Gripper-2 | 102 | 72 | 536 | 476 | 324 | 0,2 | 0,19 | 0,12 | 3 | 2 |
| Gripper-3 | 310 | 184 | 1965 | 1671 | 967 | 1,58 | 1,62 | 0,49 | 7 | 4 |
| Gripper-4 | 398 | 236 | 2810 | 2306 | 1332 | 2,72 | 2,79 | 0,79 | 7 | 4 |
| Gripper-5 | 750 | 420 | 5903 | 4693 | 2579 | 88,97 | 107,25 | 3,25 | 11 | 6 |
| Gripper-6 | 886 | 496 | 7580 | 5864 | 3220 | 218,13 | 164,12 | 5,28 | 11 | 6 |
| Gripper-7 | * | 752 | * | * | 5199 | * | * | 152,84 | * | 8 |
| Gripper-8 | * | 852 | * | * | 6180 | * | * | 236,71 | * | 8 |
| Ferry-2 | 142 | 85 | 655 | 599 | 351 | 0,38 | 0,38 | 0,17 | 7 | 4 |
| Ferry-3 | 298 | 168 | 1599 | 1401 | 777 | 1,63 | 1,58 | 0,44 | 11 | 6 |
| Ferry-4 | 510 | 279 | 3079 | 2599 | 1403 | 18,78 | 25,06 | 1,12 | 15 | 8 |
| Ferry-5 | 778 | 418 | 5191 | 4241 | 2253 | 1999,37 | 619,93 | 2,67 | 19 | 10 |
| Ferry-6 | * | 585 | * | * | 3351 | * | * | 11,2 | * | 12 |
| Ferry-7 | * | 780 | * | * | 4721 | * | * | 51,95 | * | 14 |
| Ferry-8 | * | 1003 | * | * | 6387 | * | * | 237,58 | * | 16 |
| Logistics-2323 | 855 | 585 | 4008 | 3648 | 2196 | 13,61 | 19,37 | 5,98 | 6 | 4 |
| Logistics-4323 | 1812 | 1227 | 10050 | 8727 | 5172 | 113,64 | 170,25 | 49,21 | 9 | 6 |
| Logistics-2325 | 2117 | 1502 | 10472 | 9512 | 5903 | 147,27 | 217,27 | 69,76 | 10 | 7 |
| Logistics-4325 | 2720 | 1841 | 15838 | 13705 | 7996 | 410,6 | 669,03 | 187,27 | 9 | 6 |

TAB. 1 – Codages dans les espaces d'états avec frame-axiomes explicatifs

| | Variables | | | Clauses | | | Temps CPU(s) | | | Niveaux | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | MK99 | Cod. 4 | Cod. 5 | MK99 | Cod. 4 | Cod. 5 | MK99 | Cod. 4 | Cod. 5 | MK99 | Cod. 4 | Cod. 5 |
| Bworld-3c | 234 | 230 | 176 | 2020 | 1228 | 709 | 0,62 | 0,53 | 0,29 | 4 | 4 | 3 |
| Bworld-4c | 677 | 563 | 455 | 16454 | 6256 | 3098 | 12,65 | 5,29 | 2,62 | 6 | 5 | 4 |
| Bworld-5c | * | 1174 | 984 | * | 24278 | 10463 | * | 52,85 | 24,41 | 6 | 6 | 5 |
| Bworld-6c | * | * | 965 | * | * | 14425 | * | * | 51,16 | * | * | 3 |
| Gripper-1 | 114 | 111 | 77 | 586 | 367 | 216 | 0,17 | 0,16 | 0,1 | 3 | 3 | 2 |
| Gripper-2 | 289 | 176 | 122 | 2481 | 718 | 392 | 0,95 | 0,29 | 0,16 | 5 | 3 | 2 |
| Gripper-3 | 694 | 537 | 315 | 8536 | 2857 | 1210 | 9,05 | 3,11 | 0,78 | 9 | 7 | 4 |
| Gripper-4 | 1069 | 682 | 400 | 16781 | 4060 | 1636 | 76,73 | 5,44 | 1,29 | 11 | 7 | 4 |
| Gripper-5 | * | 1283 | 713 | * | 9053 | 3268 | * | 1101,91 | 6,1 | * | 11 | 6 |
| Gripper-6 | * | 1508 | 838 | * | 11546 | 4024 | * | 1456,52 | 9,21 | * | 11 | 6 |
| Gripper-7 | * | * | 1271 | * | * | 6678 | * | * | 131,5 | * | * | 8 |
| Gripper-8 | * | * | 1436 | * | * | 7844 | * | * | 64,36 | * | * | 8 |
| Ferry-2 | 277 | 270 | 159 | 1525 | 1056 | 491 | 0,8 | 0,71 | 0,24 | 7 | 7 | 4 |
| Ferry-3 | 576 | 565 | 315 | 4198 | 2779 | 1128 | 5,47 | 4,25 | 0,77 | 11 | 11 | 6 |
| Ferry-4 | 979 | 964 | 523 | 8843 | 5678 | 2107 | 87,83 | 35,94 | 2,38 | 15 | 15 | 8 |
| Ferry-5 | * | 1467 | 783 | * | 10041 | 3488 | * | * | 7,09 | 19 | 19 | 10 |
| Ferry-6 | * | * | 1095 | * | * | 5331 | * | * | 19,04 | * | * | 12 |
| Ferry-7 | * | * | 1459 | * | * | 7696 | * | * | 53,76 | * | * | 14 |
| Ferry-8 | * | * | 1875 | * | * | 10643 | * | * | 382,04 | * | * | 16 |
| Logistics-2323 | * | 1398 | 948 | * | 5880 | 2845 | * | 28,76 | 10,43 | 11 | 6 | 4 |
| Logistics-4323 | * | 2841 | 1914 | * | 15111 | 6738 | * | 244,5 | 84,09 | 27 | 9 | 6 |
| Logistics-2325 | * | 3462 | 2445 | * | 16542 | 8295 | * | 340,2 | 128 | 17 | 10 | 7 |
| Logistics-4325 | * | 4219 | 2842 | * | 23457 | 10278 | * | 895,22 | 306,01 | 15 | 9 | 6 |

TAB. 2 – Codages dans les espaces de plans avec le chevalier blanc

SDRAM. Le caractère "-" indique que le problème n'a pas été résolu en moins de 3600 secondes et "*" signifie que la mémoire allouée est insuffisante pour la résolution. Pour tous les tests, nous avons utilisé le solveur Chaff 2 [www.ee.princeton.edu/∼chaff]. Les benchmarks utilisés (Gripper, Ferry, Blocks-word, Logistics, Hanoï) sont ceux des compétitions IPC, leur description se trouve sur le site [www.dur.ac.uk/d.p.long/competition.html]. Leurs caractéristiques les rendent complémentaires.

## 5.3 Résultats des tests

Nous comparons ici les codages présentés précédemment (cf. tables 1 et 2). Ce sont les plus intéressants. Pour la totalité des tests on se reportera à [24]. Les codages notés MK99 sont ceux de [23]. Dans les espaces d'états, on remarquera une diminution sensible du nombre de variables, de clauses et de niveaux, et une diminution souvent très importante des temps CPU (en passant de MK99 à Codage 1 puis à Codage 2). Même chose dans les espaces de plans (en passant de MK99 à Codage 4 puis à Codage 5) avec, en plus, une diminution du nombre de niveaux de MK99 à Codage 4, due à l'introduction du parallélisme. Finalement, le codage 2 (espaces d'états, frame-axiomes explicatifs, autorisation) s'avère être le plus performant (aussi bien en compacité que par ses performances pour la résolution). Les codages qui viennent ensuite, sont le codage 5 (espaces de plans, chevalier blanc, autorisation), le codage 1 (espaces d'états, frame-axiomes explicatifs) et enfin le codage 4 (espaces de plans, chevalier blanc). Nos améliorations ne modifient pas la hiérarchie antérieure des performances, ainsi, le codage le plus compact dans les espaces d'états surclasse toujours celui qui est le plus compact dans les espaces de plans, qu'on leur intègre ou non la relation d'autorisation.

# 6 Conclusion

Le travail que nous avons présenté ici porte sur plusieurs aspects de la planification SAT :

- Des améliorations significatives des codages dans les espaces d'états, de plans et du graphe de planification.
- L'implémentation du planificateur TSP qui intègre un traducteur spécifique, et permet de mesurer objectivement les performances des différents codages.
- L'expérimentation qui montre, avec nos codages, une diminution significative du nombre de clauses, de variables et de niveaux nécessaires pour représenter les plans-solutions. Cette diminution permet d'augmenter fortement les performances de la résolution. Notre codage 2 (espaces d'états, frame-axiomes explicatifs, autorisation) s'avère ainsi être le plus performant (aussi bien en compacité que par ses performances pour la résolution). Les codages qui viennent ensuite, sont, dans l'ordre, le codage 5 (espaces de plans, chevalier blanc, autorisation), le codage 1 (espaces d'états, frame-axiomes explicatifs) et enfin le codage 4 (espaces de plans, chevalier blanc).

Pour continuer d'améliorer la planification SAT, plusieurs voies peuvent être poursuivies :

- Le perfectionnement des codages.
- L'utilisation d'heuristiques spécifiques aux problèmes permettant de guider le solveur SAT.
- L'implémentation de solveurs dédiés comme DPPLAN [3] et LCDPP [28, 29] qui travaillent directement à partir du graphe de planification.

Ces possibilités permettent de penser que les méthodes de planification SAT, qui sont assez récentes, peuvent encore progresser. Certes, les progrès considérables de la recherche heuristique font douter qu'elle puisse parvenir à concurrencer ces derniers algorithmes du point de vue de la taille des problèmes résolus et de la rapidité de traitement. Mais elle offre des atouts indéniables : une certaine forme d'optimalité des plans produits et un cadre formel permettant de représenter des connaissances sur le domaine.

# Références


[1] C. Bäckström. Computationnal aspects of reordering plans. *JAIR*, vol. 9, pages 99–137, 1998.

[2] M. Baioletti, S. Marcugini, and A. Milani. An extension of SATPLAN for planning with constraints. Dans *Proc. AIMSA-98*, 1998.

[3] M. Baioletti, S. Marcugini, and A. Milani. DPPlan : An algorithm for fast solutions extraction from a planning graph. Dans *Proc. AIPS-2000*, pages 13–21, 2000.

[4] A. Blum and M. Furst. Fast planning through planning-graphs analysis. Dans *Proc. IJCAI-95*, pages 1636–1642, 1995.

[5] A. Blum and M. Furst. Fast planning through planning-graphs analysis. *Artificial Intelligence*, vol. 90, n° 1-2, pages 281–300, 1997.

[6] B. Bonet and H. Geffner. Planning as heuristic search : New results. Dans *Proc. ECP-99*, pages 360–372, 1999.

[7] M. Cayrol, P. Régnier, and V. Vidal. New results about LCGP, a least committed Graphplan. Dans *Proc. AIPS-2000*, pages 273–282, 2000.

[8] M. Cayrol, P. Régnier, and V. Vidal. Least commitment in Graphplan. *Artificial Intelligence*, vol. 130, n° 1, pages 85–118, 2001.

[9] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, vol. 32, n° 3, pages 333–377, 1987.

[10] M. Davis, G. Logemann, and D. Loveland. A computing procedure for quantification theory. *Communications of the ACM*, vol. 5, pages 394–397, 1962.

[11] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, vol. 7, pages 201–215, 1960.

[12] M. Ernst, T. Millstein, and D.S. Weld. Automatic SAT-compilation of planning problems. Dans *Proc. IJCAI-97*, 1997.

[13] R.E. Fikes and N.J. Nilsson. STRIPS : a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, vol. 2, n° 3-4, pages 189–208, 1971.

[14] M. Fox and D. Long. The automatic inference of state invariants in TIM. *JAIR*, vol. 9, pages 367–421, 1998.

[15] J. Hoffmann and B. Nebel. The FF planning system : Fast plan generation through heuristic search. *JAIR*, vol. 14, pages 253–302, 2001.

[16] S. Kambhampati. Planning graph as a (dynamic) CSP : Exploiting EBL, DDB and other CSP techniques in Graphplan. *JAIR*, vol. 12, pages 1–34, 2000.

[17] S. Kambhampati and B. Srivastava. Universal Classical Planner : An Algorithm for Unifying State-space and Plan-space Planning. Dans *Proc. EWSP-95*, pages 93–94, 1995.

[18] H. Kautz, D. McAllester, and B. Selman. Encoding plans in propositional logic. Dans *Proc. KR-96*, pages 374–384, 1996.

[19] H. Kautz and B. Selman. Planning as satisfiability. Dans *ECAI-92*, pages 359–363, 1992.

[20] H. Kautz and B. Selman. Unifying SAT-based and Graph-based planning. Dans *Proc. IJCAI-99*, pages 318–325, 1999.

[21] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending planning-graphs to an ADL subset. Dans *Proc. ECP-97*, pages 273–285, 1997.

[22] A. Mali and S. Kambhampati. Refinement-based planning as satisfiability. Dans *Proc. Workshop planning as combinatorial search, AIPS-98*, 1998.

[23] A. Mali and S. Kambhampati. On the utility of plan-space (causal) encodings. Dans *Proc. AAAI-99*, pages 557–563, 1999.

[24] F. Maris, R. Régnier, and V. Vidal. Planification SAT : amélioration des codages, automatisation de la traduction et étude comparative. Rapport technique IRIT 2002-39-R, Université Paul Sabatier, 2002.

[25] X.L. Nguyen and S. Kambhampati. Reviving partial order planning. Dans *Proc. IJCAI-2001*, pages 459–466, 2001.

[26] X.L. Nguyen, S. Kambhampati, and R.S. Nigenda. Planning graph as the basis for deriving heuristics for plan synthesis by state space and CSP search. *Artificial Intelligence*, vol. 135, n° 1-2, pages 73–123, 2002.

[27] P. Régnier and B. Fade. Complete determination of parallel actions and temporal optimization in linear plans of actions. Dans *Proc. EWSP-91*, pages 100–111, 1991.

[28] V. Vidal. *Recherche dans les graphes de planification, satisfiabilité et stratégies de moindre engagement. Les systèmes LCGP et LCDPP*. Thèse de doctorat, IRIT, Université Paul Sabatier, 10 Juillet 2001.

[29] V. Vidal. Le moindre engagement dans une approche de la planification basée sur la procédure de Davis et Putnam. Dans *Proc. JNPC-02*, pages 239–253, 2002.

[30] V. Vidal. A lookahead strategy for heuristic search planning. Rapport technique 2002-35-R, IRIT, Université Paul Sabatier, Toulouse, France, 2002.

[31] V. Vidal and P. Régnier. Total order planning is better than we thought. Dans *Proc. AAAI-99*, pages 591–596, 1999.

[32] D. Weld. An introduction to least commitment planning. *AI Magazine*, vol. 15, n° 4, pages 27–61, 1994.