Isomorphismes entre instances et sous-instances STRIPS

Martin Cooper, Arnaud Lequen, Frédéric Maris

To cite this version:

Martin Cooper, Arnaud Lequen, Frédéric Maris. Isomorphismes entre instances et sous-instances STRIPS. Journées Francophones de Programmation par Contraintes, Association Française pour l'Intelligence Artificielle (AFIA), Jun 2022, Saint-Etienne, France. pp.35-42. hal-03819065

HAL Id: hal-03819065

https://hal.science/hal-03819065

Submitted on 18 Oct 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Isomorphismes entre instances et sous-instances STRIPS

Martin C. Cooper, Arnaud Lequen, Frédéric Maris

IRIT, Université de Toulouse, France

{Martin.Cooper, Arnaud.Lequen, Frederic.Maris}@irit.fr

Résumé

Dans le domaine de la planification automatique, décider si deux instances encodées en STRIPS sont isomorphes est la manière la plus élémentaire de comparer deux instances. C'est aussi un cas particulier du problème où l'on se donne deux instances P et P', et l'on cherche un isomorphisme entre P et une sous-instance de P'. Dans cet article, nous nous proposons d'étudier la complexité de ces deux problèmes. On montre que le premier est GI-complet, tandis que le second est NP-complet. Malgré cela, nous proposons un algorithme qui permet de construire un tel isomorphisme, lorsqu'il existe. De même, nous montrons expérimentalement que, sur nos jeux de tests, un pré-traitement basé sur des méthodes de propagation de contraintes permet d'améliorer significativement l'efficacité du solver SAT utilisé par notre algorithme.

Mots-clés

Planification automatique, isomorphisme, complexité, propagation de contraintes

Abstract

Determining whether two STRIPS planning instances are isomorphic is the simplest form of comparison between planning instances. It is also a particular case of the problem concerned with finding an isomorphism between a planning instance P and a sub-instance of another instance P'. In this paper, we study the complexity of both problems. We show that the former is GI-complete, and we prove the latter to be NP-complete. Nonethelss, we propose an algorithm to build an isomorphism, when possible. We report experimental trials on benchmark problems which demonstrate that applying constraint propagation in preprocessing can greatly improve the efficiency of a SAT solver.

Keywords

Planning, isomorphism, complexity, constraint propagation

1 Introduction

Les modèles de planification STRIPS [8] encodent implicitement de grands espaces d'états, souvent impossible à représenter explicitement, mais qui ont cependant une structure claire et régulière. Il est donc raisonnable de penser que deux instances de planification différentes puissent avoir une partie de leurs structures respectives en commun. Cependant, cette similarité n'est pas immédiate à identifier. En effet, pour déterminer si une instance P est une sous-instance d'une autre instance P', il faut associer chaque fluent et chaque action de P à son homologue dans P', tout en respectant une propriété de morphisme. Cela requiert l'exploration de l'espace des fonctions de P vers P', de taille exponentielle en les représentations de ces instances. Il reste que trouver un tel morphisme permet de transférer à une instance certaines informations dont l'on dispose sur l'autre. Par exemple, tout plan-solution de P peut être transformé en un plan-solution pour P' efficacement.

En programmation par contraintes, il est monnaie courante de stocker hors-ligne toutes les solutions d'une instance CSP ou SAT en une forme compilée [1]. Une carte de compilation indique quelles opérations peuvent être effectuées en temps polynomial lors de l'exécution en ligne [7]. Or, il est bien connu qu'une instance de planification STRIPS à horizon fixe peut être compilée en une instance SAT, grâce à l'encodage SATPLAN [10]. Ainsi, pour une instance donnée, tous les plans peuvent être stockés en une forme compilée, du moins en théorie – en pratique, la forme compilée est de taille trop importante pour être gardée en mémoire. Les instances de planification les plus favorables à cette approche par compilation sont celles pour lesquelles le nombre de plans-solutions est faible, ou, au contraire, pour lesquelles l'ordre des actions est peu contraint. Dans le cas où l'on a déjà calculé une forme compilée C' représentant tous les plans-solutions pour P', et que l'on a un problème P' similaire à P, on peut se demander si l'on peut synthétiser un plan pour P à partir de C'. Si P est isomorphe à un sous-problème de P', il suffit alors d'appliquer une suite d'opérations de conditionnement à C' pour obtenir une forme compilée C représentant toutes les solutions de P. C'est dans cette optique-là que l'on cherche à étudier des isomorphismes entre sous-problèmes. Un cas simple, mais crucial, survient lorsque P n'admet aucune solution. Alors, un isomorphisme entre P et un sous-problème de P' est une preuve que P' n'admet aucune solution.

Dans cet article, nous commençons par l'étude du problème SI, qui traite de la synthèse d'un isomorphisme entre deux instances STRIPS de tailles identiques. Notre preuve de la GI-complétude du problème nous permet d'affirmer l'existence d'un algorithme quasi-polynomial pour SI [2]. Nous nous penchons ensuite sur le problème SSI, qui cherche à synthétiser un isomorphisme entre une instance STRIPS et une sous-instance d'une autre instance STRIPS. Nous appe-

 ©AFIA2022

lons une telle fonction un *isomorphisme de sous-instance*. Après avoir montré que ce problème est NP-complet, nous proposons un algorithme permettant de déterminer l'existence d'un isomorphisme de sous-instance, ou qui détecte qu'aucun tel morphisme n'existe. Notre algorithme est basé sur une réduction à SAT, renforcée par des techniques de propagation de contraintes, qui nous permettent d'éliminer des associations non-cohérentes entre éléments de P et P'.
Nous avons, jusque-là, considéré que les instances de planification P et P' possédaient des états initiaux et finaux identiques, à isomorphisme près. Même lorsque cette condition est relaxée, un isomorphisme entre P et une sous-instance de P' peut tout de même être utile. Par exemple, si π est un plan-solution pour P, son image dans P' peut alors être convertie en une nouvelle action qui peut être ajoutée à P' afin de faciliter sa résolution. Nous proposons alors cette version plus faible d'isomorphisme de sous-instance, que nous appelons *isomorphisme homogène de sous-instance*. Le problème correspondant est noté $\mathsf{SSI\text{-}H}$.
Précédemment, d'autres travaux ont étudié la complexité de nombreux problèmes associés à la synthèse de plans-solutions pour des instances STRIPS [4], ou se sont penchés sur la complexité de domaines particuliers de planification [9]. Plus rarement, on peut aussi faire état de travaux portant sur la modification d'instances de planification, à l'image du problème traitant de l'adaptation d'une instance afin de rendre faisable un plan donné en entrée [11].
Notre article est organisé comme suit : la Partie 2 présente des notations générales, ainsi que des concepts et constructions que nous utilisons au long de l'article. Dans la Partie 3 et dans la Partie 4, nous présentons nos résultats théoriques pour SI et SSI, respectivement. Dans la Partie 5, nous présentons notre algorithme pour SSI. Enfin, la Partie 6 résume nos résultats expérimentaux, suivis d'une discussion. Par soucis de concision, les démonstrations et de nombreux éléments techniques ont été abrégés, mais peuvent être retrouvés dans une version étendue de cet article [6].

2 Préliminaires

2.1 Planification automatique

Une instance de planification STRIPS est un 4-uplet $P = \langle F, I, O, G\rangle$ tel que F est un ensemble d'éléments appelés *fluents* (des variables propositionnelles dont la valeur peut changer avec le temps), I et G sont des ensembles de littéraux de F, appelés respectivement l'*état initial* et le *but*, et O est un ensemble d'*actions*. Une action est de la forme $o = \langle \mathsf{pre}(o), \mathsf{eff}(o)\rangle$, où $\mathsf{pre}(o)$ et $\mathsf{eff}(o)$ sont la *préconditions* et l'*effet* de o, des ensembles de littéraux de F.
On note $\mathsf{pre}^+(o) = \{f \in F \mid f \in \mathsf{pre}(o)\}$ les fluents positifs de $\mathsf{pre}(o)$, et $\mathsf{pre}^-(o) = \{f \in F \mid \neg f \in \mathsf{pre}(o)\}$ les fluents négatifs. De même, on note $\mathsf{eff}^+(o) = \{f \in F \mid f \in \mathsf{eff}(o)\}$ et $\mathsf{eff}^-(o) = \{f \in F \mid \neg f \in \mathsf{eff}(o)\}$.
Par abus de notation, on note $\mathsf{pre} : O \to 2^F \cup 2^{\neg F}$ la fonction $o \mapsto \mathsf{pre}(o)$, et on utilise des notations similaires pour $\mathsf{pre}^+, \mathsf{pre}^-, \mathsf{eff}, \mathsf{eff}^+$, and eff^-. Dans cet article, on notera $\mathcal{C} = \{\mathsf{pre}^+, \mathsf{pre}^-, \mathsf{eff}^+, \mathsf{eff}^-\}$, et, pour un ensemble quelconque S de littéraux de F, $\neg S = \{\neg l \mid l \in S\}$.

Un état s est une assignation de chaque fluent de F à une valeur de vérité. Nous associerions ici s avec l'*ensemble* des littéraux de F vrais dans s. Etant donnée une instance $P = \langle F, I, O, G\rangle$, un plan $o_1, \ldots, o_k$ de O est une suite d'états $s_0, \ldots, s_k$ satisfaisant, pour $i \in [\![1; k]\!]$, $s_i = (s_{i-1} \setminus \mathsf{eff}^-(o_i)) \cup \mathsf{eff}^+(o_i)$, et $\mathsf{pre}^+(o_i) \subseteq s_{i-1}$, $\mathsf{pre}^-(o_i) \cap s_{i-1} = \varnothing$. Un plan-solution est alors un plan tel que $s_0 = I$ et $G \subseteq s_k$.

2.2 La classe de complexité GI

Dans cette section, nous présentons la classe de complexité GI, pour laquelle SI est complet, comme nous le montrerons dans une section ultérieure. GI est construite autour du problème d'isomorphisme de graphes, qui consiste à trouver une bijection $u : V \to V'$ entre les sommets de deux graphes $\mathcal{G}(V, E)$ et $\mathcal{G}'(V', E')$, telle que les images des sommets reliés entre eux par une arrête dans $\mathcal{G}$ sont aussi reliés dans $\mathcal{G}'$, et réciproquement. Plus formellement :

$$\{x, y\} \in E \text{ ssi } \{u(x), u(y)\} \in E' \tag{1}$$

Définition 1. *La classe de complexité GI est la classe des problèmes pour lesquels il existe une réduction de Turing en temps polynomial au problème d'isomorphisme de graphes.*

La classe de complexité GI contient un grand nombre de problèmes traitant de l'existence d'un isomorphisme entre deux structures non-triviales encodées de manière explicite. De tels problèmes sont souvent complets pour GI, comme par exemple les problèmes d'isomorphisme entre graphes colorés, entre hypergraphes, entre automates [15]...
En l'état des connaissances actuelles, le consensus est que GI est une classe intermédiaire entre P et NP. Plus précisément, le problème d'isomorphisme de graphes peut être résolu en temps quasi-polynomial [2]. Il est donc très peu vraisemblable que le problème soit NP-difficile, malgré le fait qu'aucun algorithme polynomial n'est encore connu.

3 Problème d'isomorphisme STRIPS

Dans cette partie, nous nous intéressons au problème d'isomorphisme entre deux instances STRIPS. Après avoir défini la forme d'isomorphisme STRIPS que nous proposons, nous présentons nos résultats de complexité.

Définition 2 (Isomorphisme entre instances STRIPS)**.** *Soient $P = \langle F, I, O, G\rangle$ et $P' = \langle F', I', O', G'\rangle$ deux instances STRIPS. Un isomorphisme entre P et P' est une paire (υ, ν) de bijections $\upsilon : F \to F'$ et $\nu : O \to O'$ t.q.*

$$\forall o \in O,\ \nu(o) = \langle \upsilon(\mathsf{pre}(o)), \upsilon(\mathsf{eff}(o))\rangle \tag{2}$$
$$\upsilon(I) = I' \tag{3}$$
$$\upsilon(G) = G' \tag{4}$$

Où, pour deux ensembles disjoints $F_1, F_2 \subseteq F$ de fluents,

$$\upsilon(F_1 \cup \neg F_2) = \upsilon(F_1) \cup \neg\upsilon(F_2)$$

Une conséquence immédiate de cette définition est qu'un tel isomorphisme conserve l'espace des plans : toute suite

d'actions $o_1, \ldots, o_n \in O$ est un plan pour P si, et seulement si, la suite associée $\nu(o_1), \ldots, \nu(o_n)$ est un plan pour P'. Cette propriété de morphisme est assurée par l'équation (2). De la même façon, tous les plans-solutions sont conservés, et ce grâce aux conditions imposées par les équations (3) et (4). On en vient alors au problème SI, que l'on définit formellement afin d'établir sa complexité :

Problème 1. *Problème d'Isomorphisme STRIPS SI*

Entrée *Deux instances STRIPS P et P'*

Sortie *Un isomorphisme (υ, ν) entre P et P', s'il en existe un*

Proposition 1. *SI est GI-complet*

Démonstration. Voir [6] □

Par soucis de concision, la preuve peut se retrouver dans la version étendue de cet article. Ce résultat de GI-complétude tient encore si l'on n'impose pas que les états initiaux et buts respectifs de P et P' soient en bijections. Ainsi, cela signifie que la majeure partie de la difficulté de SI réside en la complexité d'associer correctement les structures internes respectives des espaces d'états de chaque instance, et que des propriétés supplémentaires sur certains états (comme être un état final ou initial) n'influencent pas significativement la complexité du problème.

4 Problème d'isomorphisme de sous-instance STRIPS

Dans cette partie, nous présentons les problèmes SSI-H et SSI, qui traitent de la synthèse (de deux formes différentes) d'isomorphismes entre une instance de planification P et d'une sous-isntance d'une autre instance STRIPS P'. Dans cette section, on identifie la complexité de ces deux problèmes, en prouvant leur NP-complétude. On utilise ce résultat pour proposer, dans la partie subséquente, un algorithme pour SSI et SSI-H. Cet algorithme est basé sur une réduction vers SAT, enrichi par une étape de pré-traitement s'appuyant sur de la propagation de contraintes.
On commence par introduire la notion d'*isomorphisme homogène de sous-instances*, qui est une forme d'isomorphisme entre P et une sous-instance de P' qui ne conserve cependant pas l'état initial ni le but. Il s'agit donc d'associer l'espace d'états complet du problème P à une partie de l'espace d'états de P', sans pour autant tenir compte des états initiaux ou finaux de ces instances.

Définition 3 (Isomorphisme homogène de sous-instance)**.** *Soient deux instances STRIPS $P = \langle F, I, O, G \rangle$ et $P' = \langle F', I', O', G' \rangle$. Un* isomorphisme homogène de sous-instances *de P à P' est un couple (υ, ν) de fonctions injectives $\upsilon : F \to F'$ et $\nu : O \to O'$ qui respectent la condition (2) de la Définition 2.*

Problème 2. *Isomorphisme homogène de sous-instance STRIPS SSI-H*

Entrée *Deux instances STRIPS P et P'*

Sortie *Un isomorphisme homogène de sous-instances (υ, ν) entre P et P', s'il en existe un*

Un isomorphisme homogène de sous-instances entre P et P' peut s'avérer utile, par exemple, dans le cas où l'on a réussi à compiler l'ensemble des plans pour P' et que l'on souhaite extraire un plan pour P. La définition suivante d'isomorphisme de sous-instance suivante, plus forte que dans le cas homogène, prend en compte les états initiaux et les buts. Cette notion nous permet de ne conserver que les plans-*solutions* d'un problème à l'autre.

Définition 4 (Isomorphisme de sous-instance)**.** *Un* isomorphisme de sous-instance *entre P et P' est un isomorphisme de sous-instance homogène qui respecte de surcroît les conditions (3) et (4) de la Définition 2.*

Problème 3. *Isomorphisme de sous-instance STRIPS SSI*

Entrée *Deux instances STRIPS P et P'*

Sortie *Un isomorphisme de sous-instance (υ, ν) entre P et P', s'il en existe un*

La principale différence entre SI et SSI est que, pour SSI, la condition sur la bijectivité de υ et ν est relaxée. L'injectivité des deux fonctions est cependant toujours requise, afin d'empêcher les fluents et actions d'être fusionnés ensemble par la fonction. Les autres conditions restent inchangées.
Le résultat principal de cette partie est présenté ci-dessous. La preuve se base sur une réduction à partir du problème d'isomorphisme de sous-graphes, dont la NP-complétude est un résultat bien établi [5].

Proposition 2. *SSI et SSI-H sont NP-complets*

Démonstration. Voir [6] □

5 Un algorithme pour SSI

Dans cette partie, nous présentons notre algorithme pour le problème SSI, dont le pseudo-code est fourni dans l'Algorithme 1. Cet algorithme repose sur la compilation du problème en une formule propositionnelle, qui est ensuite passée à un solveur SAT. La procédure est renforcée par un pré-traitement, basé sur la propagation de contraintes, qui nous permet d'éliminer des associations incohérentes entre les fluents et actions respectifs de chaque instance.
Pour deux instances STRIPS P et P', l'algorithme retourne un isomorphisme de sous-instance (υ, ν), lorsqu'il en existe un. L'Algorithme 1 comporte deux étapes principales. La première étape, qui s'étend de la ligne 2 à la ligne 8, consiste à éliminer autant d'associations inconsistantes que possible entre les fluents (resp. actions) du problèmes P et les fluents (resp. actions) du problème P', dès lors que l'on a détecté une incohérence d'ordre syntaxique qui a été éventuellement propagée (comme l'on présente plus bas). La deuxième étape, qui débute à la ligne 9, consiste en une phase de recherche, ce qui passe par l'encodage du problème en une formule en FNC, qui est ensuite passée à un solveur SAT.

 ©AFIA2022

Algorithme 1 trouver_isomorphisme_sous-instance

Entrée : Deux instances STRIPS P et P'
Sortie : Un isomorphisme de sous-instance entre P et P' s'il en existe un

1: Initialiser_domaines(F, O)
/* Elimination des associations impossibles */
2: $Q := F \cup O$
3: **tant que** $Q \neq \varnothing$ **faire**
4: $v := Q$.Pop()
5: $r :=$ Réviser(v)
6: **si** r **alors**
7: **si** $\mathcal{D}(v) = \varnothing$ **alors retourner** UNSAT
8: **sinon** Q.Ajouter($\{v' \mid v'$ lié à $v\}$)
/* Phase de recherche avec SAT */
9: $\varphi :=$ Encoder_en_FNC($P, P', \mathcal{D}$)
10: **retourner** Interpréter(Solveur.Trouver_modèle(φ))

5.1 Élimination des associations invalides

On appelle *association* entre fluents un couple $(f, f') \in F \times F'$ tel que f' est un candidat pour la valeur de $\upsilon(f)$. De même, on appelle association entre actions un couple $(o, o') \in O \times O'$ tel que o' est un candidat pour la valeur de $\upsilon(o)$. Compte tenu de la taille des problèmes considérés, il est crucial de détecter aussi tôt que possible les associations n'appartenant à aucun isomorphisme de sous-instance valide, afin de réduire la taille de l'espace de recherche.

Afin d'éliminer autant d'associations inconsistantes que possible, nous utilisons une technique basée sur la propagation de contraintes, telle que couramment étudiée dans la littérature de programmation par contraintes. L'idée générale est de maintenir, pour chaque fluent $f \in F$ de P, un *domaine* $\mathcal{D}(f) \subseteq F'$ de fluents de P', qui représente les candidats plausibles pour la valeur de $\upsilon(f)$. De manière analogue, chaque action $o \in O$ se voit assigner un domaine $\mathcal{D}(o) \subseteq O'$. Par la suite, on appelle *variable* tout fluent ou action. La procédure que l'on présente ci-dessous a pour objectif de réduire les domaines des différentes variables, dans l'optique d'alléger la charge passée au solveur SAT.

La première étape consiste en l'initialisation des domaines. Pour chaque fluent $f \in F$, on a $\mathcal{D}(f) = F'$. Les domaines des actions sont, pour leur part, initialisés en fonction de leur *profil d'action*. Pour chaque action $o \in O \cup O'$, on définit le vecteur $\mathsf{profile}(o) \in \mathbb{N}^6$, que l'on appelle le *profil* de o. Ce vecteur est une abstraction numérique de quelques caractéristiques de l'action, pensée de sorte à ce que si $\mathsf{profile}(o) \neq \mathsf{profile}(o')$, alors les deux actions $o \in O$ et $o' \in O'$ ne peuvent être associés.

Dans la pratique, $\mathsf{profile}(o)$ compte le nombre de fluents positifs et négatifs dans les préconditions et effets de o, ainsi que dans le nombre de fluents qui sont des *additions strictes* ou des *suppressions strictes*. Un fluent f est une *addition stricte* pour l'action o si $f \in \mathsf{pre}^-(o) \wedge f \in \mathsf{eff}^+(o)$, et est une *suppression stricte* si $f \in \mathsf{pre}^+(o) \wedge f \in \mathsf{eff}^-(o)$. Enfin, on initialise le domaine de chaque action $o \in O$:

$$\mathcal{D}(o) = \{o' \in O' \mid \mathsf{profile}(o') = \mathsf{profile}(o)\}$$

La seconde étape consiste à propager les contraintes supplémentaires posées par les restrictions fraîchement identifiés du domaine. La technique que l'on propose est basée sur l'idée de cohérence d'arc, omniprésente dans le domaine de la programmation par contraintes. Elle consiste à éliminer du domaine des fluents (resp. actions) les fluents-candidats (resp. actions-candidates) qui ne sont pas supportés par le domaine d'une action (resp. fluent). Cela permet de faire la jonction entre les contraintes posées par les fluents et les contraintes posées par les actions.

Plus précisément, soit un fluent $f \in F$. Lorsqu'une action $o \in O$ est tel que f apparaît (positivement ou négativement) dans ses préconditions ou ses effets, alors on dit que *o dépend* de f. On note alors $d(f)$ l'ensemble des actions qui dépendent de f. On utilise des notations similaires pour $d(f')$ lorsque $f' \in F'$. Plaçons-nous maintenant dans le cas où $\upsilon(f) = f'$. Une conséquence de l'équation (2) de la Définition 2 est que chaque action de $d(f)$ doit avoir son image par ν dans $d(f')$. Sinon, on aurait que f apparaît dans $\mathsf{pre}(o)$ ou $\mathsf{eff}(o)$, mais $\upsilon(f)$ n'apparaîtrait ni dans $\upsilon(\mathsf{pre}(o))$, ni dans $\upsilon(\mathsf{eff}(o))$. Ainsi, si pour une action donnée $o \in d(f)$, aucun candidat pour son image n'est dans $d(f')$ (*i.e.*, $\mathcal{D}(o) \cap d(f') = \varnothing$), alors on a que f' ne peut pas être choisi pour l'image de f.

Dans la suite, on précise l'argument tout juste énoncé, en identifiant $\mathsf{pre}^+(o)$ et $\mathsf{pre}^+(o'), \ldots, \mathsf{eff}^-(o)$ et $\mathsf{eff}^-(o')$. On obtient alors les contraintes suivantes pour $\mathcal{D}(f)$, dans lesquelles on note $\mathcal{C} = \{\mathsf{pre}^+, \mathsf{pre}^-, \mathsf{eff}^+, \mathsf{eff}^-\}$:

$$\mathcal{D}(f) \subseteq \left\{ f' \;\middle|\; \begin{array}{l} \forall o \in O, \forall \mathcal{S} \in \mathcal{C} \text{ s.t. } f \in \mathcal{S}(o), \\ \exists o' \in \mathcal{D}(o) \text{ s.t. } f' \in \mathcal{S}(o') \end{array} \right\} \quad (5)$$

Le même exercice peut être fait pour les actions. Soit $o \in O$ une action quelconque, et considérons un candidat pour son image $o' \in O'$. Afin de satisfaire la propriété de morphisme, dans le cas où l'on a effectivement $\nu(o) = o'$, on doit trouver dans $\mathsf{pre}^+(o')$ un fluent qui appartient à $\mathcal{D}(f)$, et ce pour chaque fluent $f \in \mathsf{pre}^+(o')$. Plus généralement, et plus formellement, cela impose les conditions suivantes :

$$\mathcal{D}(o) \subseteq \{o' \mid \forall \mathcal{S} \in \mathcal{C}, \forall f \in \mathcal{S}(o), \exists f' \in \mathcal{D}(f) \cap \mathcal{S}(o')\} \quad (6)$$

Algorithmiquement, on s'assure de la satisfaction de ces contraintes en utilisant une version adapté de AC3 [13, 14]. On retrouve en particulier la sous-routine de révision de la cohérence des domaines des variables.

Réviser une variable v consiste à vérifier que l'ensemble des éléments de son domaine sont conformes aux conditions nécessaires évoquées précédemment, c'est-à-dire l'équation (5) si v est un fluent, ou (6) si v est une action. La boucle principale, dépeinte dans l'Algorithme 1, effectue donc une révision itérative de tous les fluents et actions, en maintenant une file Q des variables à réviser (ligne 1). Cette file est initialisée avec l'ensemble des variables, afin que chacune soit révisée au moins une fois.

Si, lors de la révision d'une variable v, le domaine de v est modifié par la procédure, alors toutes les variables qui sont liées à v sont ajoutées à la file des variables à réviser (lignes 5 à 9). On dit que v' est liée à la variable v

si v est un fluent et $v' \in d(v)$, ou inversement. Si le domaine d'une variable est vide, alors on est assuré qu'aucun isomorphisme de sous-instance n'existe, et l'algorithme se termine immédiatement (lignes 6 et 7). Sinon, la boucle se termine lorsque la file des variables à réviser est vide.
Cette procédure allège donc la charge restant à la phase de recherche, que l'on présente dans la partie suivante.

5.2 Encodage en une instance SAT

Dans cette partie, nous présentons notre construction de la formule propositionnelle φ évoquée plus tôt, dont les modèles permettent de trouver un isomorphisme de sous-instance. φ est construit sur l'ensemble de variable *Var*(φ), que l'on définit comme suit :

$$Var(\varphi) = \left\{ f_i^j \;\middle|\; i \in F, j \in F' \right\} \cup \{ o_r^s \mid r \in O, s \in O' \}$$

Une variable propositionnelle de la forme f_i^j représente l'association d'un fluent $i \in F$ à un fluent $j \in F'$. De même, o_r^s représente l'association de $r \in O$ à $s \in O'$.
φ encode une instance de SSI passée en entrée à l'Algorithme 1. La formule consiste en la conjonction des formules suivantes, qui nous assurent chacune une propriété différente sur un modèle.
La formule présentée dans l'équation (7) nous assure de l'unicité de l'image de chaque fluent. On obtient la même propriété sur les images des actions en remplaçant les variables de la forme f_i^j par des variables o_i^j, et en adaptant les domaines de i et j en conséquence.

$$\bigwedge_{i \in F} \left(\bigvee_{j \in \mathcal{D}(i)} f_i^j \;\wedge \bigwedge_{\substack{j,k \in \mathcal{D}(i) \\ j \neq k}} (\neg f_i^j \vee \neg f_i^k) \right) \tag{7}$$

Afin d'assurer l'injectivité de v et ν, on ajoute l'équation (8) pour les fluents, qui est aussi adaptée comme précédemment dans le cas des actions.

$$\bigwedge_{i \in F'} \bigwedge_{\substack{j,k \in F \\ j \neq k}} \neg f_j^i \vee \neg f_k^i \tag{8}$$

La propriété de morphisme est assurée par les formules (9) et (10), pour chaque $\mathcal{S} \in \mathcal{C} = \{\mathsf{pre}^+, \mathsf{pre}^-, \mathsf{eff}^+, \mathsf{eff}^-\}$. Plus précisément, (9) assure que, pour tout $\mathcal{S} \in \mathcal{C}$ et pour $o \in O$ quelconque, on a $v\,(\mathcal{S}(o)) \subseteq \mathcal{S}(\nu\,(o))$. Réciproquement, (10) nous assure que $\mathcal{S}(\nu\,(o)) \subseteq v\,(\mathcal{S}(o))$.

$$\bigwedge_{\substack{r \in O \\ s \in O'}} \left(o_r^s \longrightarrow \bigwedge_{i \in \mathcal{S}(r)} \bigvee_{j \in \mathcal{S}(s)} f_i^j \right) \tag{9}$$

$$\bigwedge_{\substack{r \in O \\ s \in O'}} \left(o_r^s \longrightarrow \bigwedge_{j \in \mathcal{S}(s)} \bigvee_{i \in \mathcal{S}(r)} f_i^j \right) \tag{10}$$

Finalement, il ne reste plus qu'à conserver l'état initial et le but (i.e., nous assurer du respect des équations (3) et (4)). On note alors I^+ (resp. I^-) l'ensemble des fluents positifs (resp. négatifs) dans I, et on utilise des notations similaires pour G, I' et G'. Pour chaque $\mathcal{T} \in \{I^+, I^-, G^+, G^-\}$, ainsi que pour l'ensemble $\mathcal{T}' \in \{I'^+, I'^-, G'^+, G'^-\}$ correspondant, on ajoute alors les formules suivantes :

$$\bigwedge_{i \in \mathcal{T}} \bigvee_{j \in \mathcal{T}'} f_i^j \quad \wedge \quad \bigwedge_{j \in \mathcal{T}'} \bigvee_{i \in \mathcal{T}} f_i^j \tag{11}$$

Les formules définies par (7), (8), et (11) sont immédiatement en FNC, et la taille de leur conjonction est un $\mathcal{O}(|F| \cdot |F'|^2 + |O| \cdot |O'|^2)$, avec l'hypothèse que $|F| \leq |F'|$ si $|O| \leq |O'|$ (sinon, la conclusion est immédiate). Par ailleurs, les formules définies par (9) et 10) peuvent être immédiatement converties en FNC en dupliquant les implications dans chaque clause. Elle ont alors une taille $\mathcal{O}(|O| \cdot |O'| \cdot |F| \cdot |F'|)$.
L'étape de pré-traitement présentée dans la Partie 5.1 nous permet de simplifier φ. En, effet, si l'on sait que le fluent $i \in F$ (resp. l'action $r \in O$) ne peut pas être assigné au fluent $j \in F'$ (resp. $s \in O'$), alors f_i^j (resp. o_r^s) est nécessairement faux dans tout modèle de φ. Ainsi, nos formules étant en FNC, toutes les occurrences positives de f_i^j peuvent être supprimées dans les clauses de φ, tandis que les clauses où f_i^j apparaît négativement peuvent être simplifiées.
Pour adapter l'algorithme précédent à SSI-H, il suffit de supprimer les formules de (11), et de conserver le reste des formules et de l'algorithme.

6 Évaluation empirique

L'Algorithme 1 a fait l'objet d'une implémentation en Python 3.10, ce qui nous a permis de le tester sur des instances de SSI et de SSI-H. L'analyse syntaxique a été confiée au module dédié de TouISTPlan [3], ce qui nous permet de convertir des instances de planification PDDL en une représentation STRIPS. Comme solveur SAT, nous avons eu recours à Maple LCM [12], gagnant du *main track* de la compétition SAT 2017. L'ensemble des ressources nécessaires à la reproduction des expériences présentées ici peut être trouvé sur la page web dédiée [1].
Les expériences ont été effectuées sur une machine fonctionnant sous Rocky Linux 8.5, dont le processeur était un Intel Xeon E5-2667 v3 processor, en utilisant au plus 8Go de mémoire vive et 4 threads par test.
Notre jeu de test est tiré de huit ensembles utilisés dans l'*International Planning Competition* : Blocks, Gripper, Hanoi, Rovers, Satellite, Sokoban, et TSP. Pour chacun de ces domaines, nous avons créé ce que l'on a nommé des *instances de morphismes STRIPS*, qui sont des paires d'instances du même domaine. Nous avons effectué cette opération pour chaque paire possible d'instances de planification, et ce pour chaque domaine pris en compte. Ainsi, une instance de morphisme STRIPS est à la fois une instance de SSI et une instance de SSI-H. Nous avons donc pu évaluer notre implémentation pour chacun de ces deux problèmes, sur le même jeu de tests.

1. `https://github.com/arnaudlequen/PDDLIsomorphismFinder`

Domaine	SSI-H		
	CP	NoCP	Simp. Moy.
blocks	172	96	76.1%
gripper	210	189	74.9%
hanoi	74	75	0.2%
rovers	19	6	97.4%
satellite	34	22	79.1%
sokoban	204	0	98.6%
tsp	376	374	0.7%

Domaine	SSI		
	CP	NoCP	Simp. Moy.
blocks	166	93	76.2%
gripper	90	84	75.1%
hanoi	85	82	0.2%
rovers	16	6	97.3%
satellite	38	23	78.4%
sokoban	205	4	98.6%
tsp	265	266	1.0%

TABLE 1 – Nombre d'instances de SSI-H et SSI pour lesquelles notre algorithme termine en moins de 600 secondes. Pour chaque problème, les deux premières colonnes présentent le nombre d'instance de morphisme STRIPS qui ont pu être résolues avec et sans, respectivement, l'étape de pré-traitement basée sur une propagation des contraintes. La dernière colonne montre le pourcentage moyen de clauses qui ont pu être éliminées dans l'encodage en FNC φ, grâce à l'étape d'élimination des associations.

L'objectif de ces évaluations expérimentales est double. D'une part, il s'agit de montrer que, malgré la difficulté théorique du problème, il est en pratique possible d'identifier un isomorphisme (homogène) de sous-instances en temps raisonnable, et ce pour des problèmes de tailles non-triviales. D'autre part, il s'agit de tester l'efficacité de notre technique de pré-traitement présentée en Partie 5.1, afin de montrer que le coût supplémentaire ne surpasse pas l'avantage qu'il donne à l'algorithme de recherche.

La couverture absolue de notre implémentation sur notre jeu de tests est présentée dans la Table 1 pour SSI et SSI-H. Le tableau montre le nombre d'instances du jeu de tests pour lesquelles notre implémentation termine avec les contraintes spatiales et temporelles allouées.

On remarque tout d'abord que les problèmes SSI-H et SSI sont souvent comparables en terme de difficulté, à quelques domaines près qui font figure d'exceptions. Parmi ces exceptions, on compte par exemple TSP et Gripper, pour lesquels relaxer la condition sur l'état initial et le but permet de résoudre, respectivement, 40% et 133% plus d'instances. Pour ces deux domaines, cela peut s'expliquer par les contraintes supplémentaires imposées par SSI. En effet, une de leurs conséquences immédiates est que toutes les paires d'instances différentes de TSP (ou de Gripper) sont des instances négatives de SSI, qui posent davantage de difficultés au solveur SAT que des instances positives.

De plus, les bénéfices de l'étape de pré-traitement dépassent presque toujours le coût de l'opération. En effet, l'immense majorité des instances que notre algorithme peut résoudre sans pré-traitement peuvent aussi être résolues une fois le pré-traitement activé. A plus forte raison, on peut surtout constater que le pré-traitement améliore grandement les performances générales de notre algorithme, tant et si bien que certains domaines, précédemment hors de portée pour notre algorithme, deviennent faisables. Parmi de tels cas extrêmes, on peut citer le domaine Sokoban, pour lequel notre algorithme est impuissant sans l'étape préalable d'élimination des associations inconsistantes. En effet, des 204 instances résolues par notre implémentation, aucune n'est résolue une fois le pré-traitement désactivé. Dans le cas général, cependant, on observe une augmentation du nombre d'instances résolues qui, sans pour autant changer d'ordre de grandeur, reste significatif. Par exemple, pour le domaine Satellite, dans le cas de SSI, 34 instances sont résolues lorsque la propagation de contraintes est activée, alors que seulement 22 peuvent être décidées sans pré-traitement.

Plus spécifiquement, pour presque toutes les instances de test, le pré-traitement permet une réduction non-négligeable de la taille de l'encodage propositionnel. C'est ce qui est montré dans les colonnes intitulées "Simp. Moy." dans la Table 1, qui représentent la proportion moyenne de clauses simplifiées grâce à l'étape d'élimination des associations inconsistantes. Les plus grands pourcentages de clauses simplifiées sont trouvés dans les domaines qui contiennent peu de symétries. Par exemple, pour le domaine Rovers, les fluents représentent des entités qui sont souvent de types différents, et qui sont donc affectés de manières très différentes par les actions. Cela peut s'illustrer par des actions de la forme navigate(rover, x, y), qui ont un profil qui leur est propre, et qui sont peu nombreuses. Par conséquent, leurs domaines respectifs sont de tailles très réduites, ce qui est très favorable à notre algorithme.

Au contraire, les domaines contenant un grand nombre de symétries ne profitent que très peu de l'élimination d'associations. C'est le cas du domaine Hanoi, pour lequel toutes les actions ont le même profil : à l'exception des informations données par l'état initial et le but, tous les disques sont *a priori* interchangeables, ce qui ne permet pas à notre pré-traitement d'obtenir des résultats significatifs. Les seuls fragments d'informations à mêmes de guider la phase de recherche se retrouvent dans l'état initial, ce que nous pensons être un argument permettant d'expliquer la couverture légèrement supérieur de SSI, comparé à SSI-H.

Pour certaines instances de notre jeu de test, le pré-traitement est suffisant pour conclure qu'il n'existe aucun isomorphisme (homogène) de sous-instances. Cela se produit lorsqu'une variable voit son domaine réduit à l'ensemble vide. Dans ce cas, l'algorithme peut alors faire l'économie de la phase de recherche entière. Ainsi, notre algorithme est plus efficace lorsqu'il s'agit de détecter qu'il n'existe aucun isomorphisme (homogène) de sous-instances. C'est ainsi que notre étape d'élimination des associations inconsistantes nous permet d'augmenter

Domaine	SSI-H CP	Comp.	Résol.	Temps total
blocks	0.5	93.3	76.8	170.3
gripper	0.2	23.5	9.8	33.4
hanoi	0.3	43.9	78.0	122.0
rovers	1.8	168.9	2.2	171.4
satellite	0.4	116.4	48.8	165.4
sokoban	1.7	222.7	2.3	225.2
tsp	0.2	50.7	46.7	97.5

Domaine	SSI CP	Comp.	Résol.	Temps total
blocks	0.4	83.3	94.6	178.1
gripper	0.1	11.2	35.2	46.5
hanoi	0.3	70.9	47.8	118.9
rovers	1.7	180.7	2.7	183.5
satellite	0.4	85.0	10.3	95.4
sokoban	1.7	220.6	1.4	222.3
tsp	0.1	14.6	26.6	41.2

TABLE 2 – Temps moyen, en secondes, passé dans chacune des trois étapes principales de l'algorithme : pré-traitement (CP), compilation vers SAT, et résolution, respectivement. La dernière colonne résume le temps total moyen utilisé par l'algorithme. Nous ne présentons ici que les résultats pour les instances qui ont pu être décidées avec succès (positivement ou négativement) : les résultats pour SSI-H et SSI sont donc non-comparables.

Domaine	SSI-H Somme max. $\|P\|$	$\|P'\|$	Somme	$\|P\|$ max. $\|P\|$	$\|P'\|$
blocks	57	4642	4699	534	534
gripper	510	510	1020	510	510
hanoi	13	3328	3341	391	391
rovers	276	2667	2943	920	920
satellite	147	2066	2213	608	920
sokoban	2212	2286	4498	2212	2286
tsp	182	930	1112	462	462

Domaine	SSI-H Somme max. $\|P\|$	$\|P'\|$	Somme	$\|P\|$ max. $\|P\|$	$\|P'\|$
blocks	57	4642	4699	534	534
gripper	510	510	1020	510	510
hanoi	13	6953	6966	391	513
rovers	276	2667	2943	920	920
satellite	147	2610	2757	608	920
sokoban	2212	2286	4498	2212	2286
tsp	90	930	1020	380	380

TABLE 3 – Tailles des plus grandes instances pouvant être résolues par notre implémentation, avec les contraintes spatiales et temporelles imposées, pour SSI-H et pour SSI. Dans les trois premières colonnes, on considère la somme des tailles des instances de planifications qui constituent le problème de morphisme STRIPS. Dans les colonnes suivantes, on considère la taille de P, plus petite instance parmi les deux formant l'instance de morphisme STRIPS.

grandement notre couverture des instances de morphismes STRIPS qui sont négatives, tandis que notre performances sur les instances positives est plus modeste, quoique significatif. Ces résultats sont compilés dans la Figure 1.

La Table 2 montre que le temps que prend l'étape de pré-traitement est négligeable devant le reste de l'algorithme. Plus précisément, que ce soit dans les domaines où elle élimine un grand nombre d'associations, ou dans les domaines où son efficacité est plus limitée, la propagation de contraintes prend rarement plus de quelques secondes. C'est ainsi que les instances où le pré-traitement permet de conclure sont résolues quasi-immédiatement.

Dans la Table 3, nous présentons quelques résultats concernant les tailles absolues des problèmes que nous avons été en mesure de résoudre. A chaque instance de planification STRIPS $P = \langle F, I, O, G\rangle$, on associe une taille $|P| = |F| + |O|$. Dans la mesure où une instance de morphisme STRIPS a deux dimensions principales, que l'on représente par les tailles respectives des instances de planification qui la constituent, on présente deux manières différentes de mesurer la taille d'une instance de SSI.

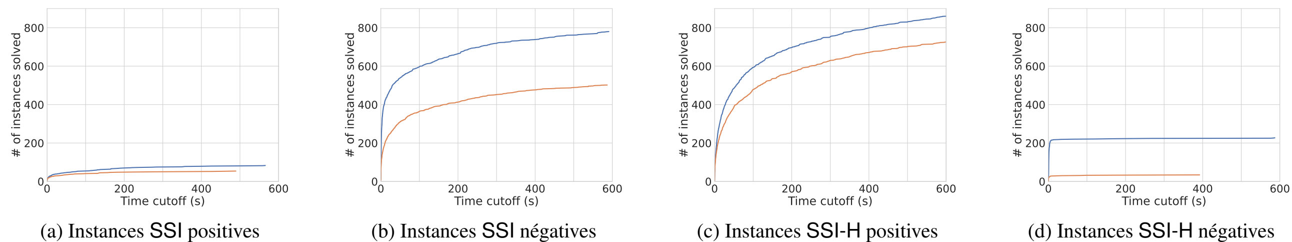
Dans les trois premières colonnes de la Table 3, pour une instance de morphisme STRIPS donnée, on considère la somme des tailles des deux instances de planification constituant l'instance, et on présente l'instance maximisant cette somme. Cependant, avec cette mesure, P' est souvent de taille disproportionnée par rapport à P. Ce déséquilibre peut être expliqué par le fait que l'encodage en une formule propositionnelle est de temps et de taille $\mathcal{O}(|O| \cdot |O'| \cdot |F| \cdot |F'|)$, comme mentionné précédemment. Dans les deux dernières colonnes, on considère l'ordre lexicographique sur les couples $(|P|, |P'|)$. On représente, pour chaque problème, l'instance maximisant cette métrique.

7 Conclusion

Dans cet article, nous nous sommes tout d'abord intéressés au problème SI, qui traite de la synthèse d'un isomorphisme entre deux instances STRIPS, et nous avons montré sa GI-complétude. Ensuite, nous avons introduit la notion d'isomorphisme de sous-instance STRIPS, ainsi que les problèmes associés SSI et SSI-H. En sus de prouver la NP-complétude de ces deux problèmes, nous avons proposé un algorithme permettant de les résoudre, basé sur des techniques de propagation de contraintes, ainsi que sur une compilation vers SAT.

Notre évaluation expérimentale de cet algorithme a permis de montrer que des techniques classiques de propagation de contraintes, utilisées en pré-traitement, permettent d'améliorer grandement les performances d'un solveur SAT. Cela dit, bien que l'étape de pré-traitement en elle-même n'est que très peu coûteuse, tous les domaines de notre jeu de test n'en ont pas tiré le même parti.

 ©AFIA2022



(a) Instances SSI positives (b) Instances SSI négatives (c) Instances SSI-H positives (d) Instances SSI-H négatives

FIGURE 1 – Nombre d'instances SSI et SSI-H que notre implémentation peut résoudre, en fonction du temps laissé à l'algorithme. La courbe bleue (resp. orange) correspond au cas où le pré-traitement est activé (resp. désactivé).

De manière plus générale, notre travail fait office d'illustration d'un problème ouvert plus large, qui consiste à identifier quelles caractéristiques de problèmes de NP les rendent plus enclins à bénéficier de cette approche hybride entre programmation par contraintes et résolution SAT.

Remerciements

Les auteurs tiennent à remercier les relecteurs anonymes de cet article, dont les commentaires nous ont permis d'améliorer la forme et le fond de ce papier.

Références

[1] Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis. Consistency restoration and explanations in dynamic csps application to configuration. *Artif. Intell.*, 135(1-2) :199–234, 2002.

[2] László Babai. Group, graphs, algorithms : the graph isomorphism problem. In *Proceedings of the International Congress of Mathematicians*, pages 3319–3336. World Scientific, 2018.

[3] Djamila Baroudi, Maël Valais, and Frédéric Maris. Touistplan.

[4] Tom Bylander. The computational complexity of propositional strips planning. *Artificial Intelligence*, 69(1-2) :165–204, 1994.

[5] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *3rd Annual ACM Symposium on Theory of Computing*, pages 151–158. ACM, 1971.

[6] Martin C. Cooper, Arnaud Lequen, and Frédéric Maris. Isomorphisms between STRIPS problems and sub-problems. In *28th International Conference on Principles and Practice of Constraint Programming (CP 2022) (To appear)*, 2022.

[7] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *J. Artif. Intell. Res.*, 17 :229–264, 2002.

[8] Richard Fikes and Nils J. Nilsson. STRIPS : A new approach to the application of theorem proving to problem solving. *Artif. Intell.*, 2(3/4) :189–208, 1971.

[9] Malte Helmert. Complexity results for standard benchmark domains in planning. *Artificial Intelligence*, 143(2) :219–262, 2003.

[10] Henry A. Kautz and Bart Selman. Planning as satisfiability. In Bernd Neumann, editor, *ECAI 92*, pages 359–363. John Wiley and Sons, 1992.

[11] Songtuan Lin and Pascal Bercher. Change the world - how hard can that be ? On the computational complexity of fixing planning models. In Zhi-Hua Zhou, editor, *IJCAI-21*, pages 4152–4159, 8 2021.

[12] Mao Luo, Chu-Min Li, Fan Xiao, Felip Manyà, and Zhipeng Lü. An effective learnt clause minimization approach for cdcl sat solvers. In *IJCAI-17*, pages 703–711, 2017.

[13] Alan K Mackworth. Consistency in networks of relations. *Artificial intelligence*, 8(1) :99–118, 1977.

[14] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.

[15] Viktor N Zemlyachenko, Nickolay M Korneenko, and Regina I Tyshkevich. Graph isomorphism problem. *Journal of Soviet Mathematics*, 29(4) :1426–1481, 1985.