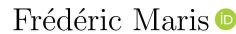
Homomorphisms and Embeddings of STRIPS Planning Models

Arnaud Lequen Martin C. Cooper Frédéric Maris 

Accepted for publication in *Computational Intelligence* on November 18, 2024

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'. One application of such a mapping is to efficiently produce a compiled form containing all solutions to P from a compiled form containing all solutions to P'. We also introduce the notion of *embedding* from an instance P to another instance P', which allows us to deduce that P' has no solution-plan if P is unsolvable. In this paper, we study the complexity of these problems. We show that the first is GI-complete, and can thus be solved, in theory, in quasi-polynomial time. While we prove the remaining problems to be NP-complete, we propose an algorithm to build an isomorphism, when possible. We report extensive experimental trials on benchmark problems which demonstrate conclusively that applying constraint propagation in preprocessing can greatly improve the efficiency of a SAT solver.

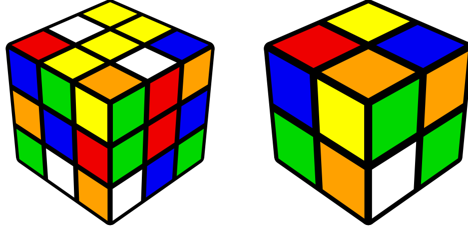
1 Introduction

Automated planning is concerned with finding a course of action in order to achieve a goal. In this paper, we are concerned with deterministic, fully-observable planning tasks. A case in point of such a task is the 3x3x3 Rubik's cube, presented in Figure 1a [a]: even though the puzzle can be described in a few words, there are more than $4.3 \cdot 10^{19}$ different configurations that can be reached through a sequence of legal moves [39]. As a consequence, without further knowledge, finding a solution for this puzzle is far from trivial, as some configurations might require up to 26 moves to be solved [39]. However, nowadays, efficient methods for solving a 3x3x3 Rubik's cube are known. But many variants of the puzzle have emerged: consider for instance the 2x2x2 cube, as depicted in Figure 1b. Despite its smaller size, there are approximately $3 \cdot 10^6$ different configurations. To tackle this puzzle, rather than trying to find a solution *ex nihilo*, one could make the simple observation that a 2x2x2 cube can be seen as a 3x3x3 cube on which edges are ignored. This allows us to transfer the knowledge that we have on the bigger cube to the smaller. In particular, one can readily transform a solution for the 3x3x3 cube into a solution for the 2x2x2 cube, given that the corner pieces are in a similar configuration.

In the more general case of automated planning, planning tasks expressed in STRIPS [18] encode sizeable state-spaces that can rarely be represented explicitly, but that have a clear and somewhat regular structure. Parts of this structure can be, however, common to multiple planning instances, although this similarity is often far from immediate to identify by looking at the STRIPS representation. Indeed, finding whether or not an instance P is equivalent to a sub-instance of another instance P' requires finding a mapping between fluents and operators of the two instances, while respecting a morphism property. This requires the exploration of the exponential search space of mappings from P to P'. Finding such a mapping, however, allows us to carry over significant pieces of information (such as solvability) from one problem to the other.

A classical technique in constraint programming is to store all solutions to a CSP or SAT instance in a compact compiled form [1]. This is performed off-line. A compilation map indicates which operations and transformations can be performed in polynomial time during the on-line stage [13]. STRIPS fixed-horizon planning can be coded as a SAT instance using the classical SATPLAN encoding [27]. So, for a given instance, all plans up to a given length can be stored in a compiled form, at least in theory.

[a] All images of Rubik's cube in this paper have been generated using `https://github.com/Cride5/visualcube`



(a) A 3x3x3 Rubik's cube (b) A 2x2x2 Rubik's cube

Figure 1: A pair of Rubik's cubes. For both of these puzzles, each face can be rotated 90 degrees, clockwise or counter-clockwise, any number of times. The goal is to reach a configuration where for all six colors, all facets of that color are on the same face, so that faces have a homogeneous color. Note that the respective configurations of the corner pieces of these cubes are similar: any sequence of moves that solves the 3x3x3 cube also solves the 2x2x2 cube.

In practice, the compiled form will often be too large to be stored. Types of planning problems which are nevertheless amenable to compilation are those where the number of plans is small or, at the other extreme, there are few constraints on the order of operators. If we have a compiled form C' representing all shortest solution-plans to an instance P' and we encounter a similar problem P, it is natural to ask whether we can synthesize a plan for P from C'. If P is isomorphic to a subinstance of P', then it suffices to apply a sequence of conditioning operations to C' to obtain a compiled form C representing all solutions to P. This was our initial motivation for studying isomorphisms between subproblems. A trivial but important special case occurs when C' is empty, i.e. P' has no solution. In this case, an isomorphism from P to a subproblem of P' is a proof that P has no solution.

In this paper, we will introduce two notions of homomorphisms between planning models. In general, any mapping from a model (a planning instance in our case) to another model that carries over (part of) the structure of the original model is called a homomorphism. If, in addition, the mapping is bijective, then it can be referred to as an isomorphism. We first focus on problem SI, which is concerned with finding an isomorphism between two STRIPS instances of identical size. As we show that the problem is GI-complete, we prove that a quasi-polynomial time algorithm theoretically exists [2].

We then consider problem SSI, which is concerned with finding an isomorphism between a STRIPS instance P and a subinstance of another STRIPS instance P'. Given a STRIPS instance P', we call subinstance of P' any STRIPS instance P'_s which shares the same initial state and goal, and whose set of fluents F'_s (resp. operators O'_s) forms a subset of the fluents F' (resp. operators O') of P', so that the operators of O'_s do not contain fluents of $F' \setminus F'_s$. An isomorphism between P and some subinstance P'_s of P' is called a *subinstance isomorphism*. After showing that the problem of finding such a mapping is NP-complete, we propose an algorithm that finds a subinstance isomorphism if one exists, or that detects that none exists. This algorithm is based on constraint propagation techniques, that allow us to prune impossible associations between elements of P and P', as well as on a reduction to SAT.

So far we have assumed that the two planning instances P and P' have the same initial states and goals (modulo the isomorphism). Even when this is not the case, an isomorphism from P to a subinstance of P' can still be of use. For example, if π is a solution-plan for P, then its image in P' can be converted to a single new operator (commonly called a *macro-operator*) which could be added to P' to facilitate its resolution. We therefore also consider this weaker notion of subinstance isomorphism, that we call *homogeneous subinstance isomorphism*, and the corresponding computational problem SSI-H.

In addition, we also introduce another form of comparison between two STRIPS planning instances, that we call *embedding*. An embedding of P' into a bigger instance P is a form of homomorphism that only takes into account operators of P that are not trivialized by the induced transformation. It is useful in proving the unsolvability of an instance, as an embedding of an unsolvable instance P' into P is a proof that P is also unsolvable. To quickly see an essential difference between the notions of subinstance-isomorphism and embedding, consider an instance P, an instance Q obtained by deleting some operators from P and another instance R obtained by deleting some goals from P: Q is isomorphic

to a subinstance of P whereas R embeds in P. This simple example illustrates the important difference that an instance which is isomorphic to a subinstance (such as Q) is harder to solve than the original instance P, whereas embedded instances (such as R) are easier.

Previous work investigated the complexity of various problems related to finding solution-plans for STRIPS planning instances [7], or focused on the complexity of solving instances from specific domains [23]. More scarcely, problems focused on altering planning models have been studied from a complexity theory point of view, such as the problem concerned with adapting a planning model so that some user-specified plans become feasible [31]. The present paper follows an orthogonal track in that we detect relationships between planning instances rather than directly solving them. Independently of any computational problem, the notions of subinstance isomorphism and embedding may be of interest in explainable AI: a minimal solvable isomorphic subinstance of a solvable instance can be viewed as an explanation of solvability, whereas a minimal unsolvable embedded subinstance can be seen as an explanation of unsolvability.

The paper is organized as follows. In Section 2, we introduce general notation, concepts and constructions that we use throughout this paper. In Section 3 and Section 4, we present our complexity results, for SI and SSI respectively. In Section 5, we present the outline of our algorithm for SSI. We follow the same pattern for SE, as we present our complexity result for the problem in Section 6, and the adapted algorithm in Section 7. Section 8 is then dedicated to the experimental evaluation and discussion. This article is a considerably extended version of a conference paper which studied the notion of subinstance isomorphism but not of embedding [11]. A more detailed version of the present paper is available online [30].

2 Preliminaries

In this section, we present generalities about automated planning, and we give some background on the complexity class GI. In addition, Section 2.3 presents some constructions that we require for the rest of this paper.

2.1 Automated Planning

In this paper, we encode planning tasks in STRIPS. A STRIPS planning instance is a tuple $P = \langle F, I, O, G\rangle$ such that F is a set of *fluents* (propositional variables whose values can change over time), I and G are sets of fluents of F, called the *initial state* and *goal*, and O is a set of *operators*. Operators are of the form $o = \langle \mathsf{pre}(o), \mathsf{eff}(o)\rangle$. $\mathsf{pre}(o) \subseteq F$ is the *precondition* of o and $\mathsf{eff}(o)$ is the *effect* of o, which is a set of literals of F. We will denote $\mathsf{eff}^+(o) = \{f \in F \mid f \in \mathsf{eff}(o)\}$, and $\mathsf{eff}^-(o) = \{f \in F \mid \neg f \in \mathsf{eff}(o)\}$. For any set S of literals of F, we denote $\neg S = \{\neg l \mid l \in S\}$. By a slight abuse of notation, we will denote $\mathsf{pre} : O \to 2^F$ the function $o \mapsto \mathsf{pre}(o)$, and use similar notation with eff^+ and eff^-, as well as with $\mathsf{eff} : O \to 2^F \cup 2^{\neg F}$. In the rest of this paper, we will note $\mathcal{C} = \{\mathsf{pre}, \mathsf{eff}^+, \mathsf{eff}^-\}$. In addition, we apply functions to sets in the natural way: for a function $u : A \to B$, if $C \subseteq A$, then $u(C)$ represents the set $\{\, u(x) \mid x \in C \,\}$.

Note that even if the formalism we present does not allow negative literals in the preconditions, we do not lose in generality: any STRIPS instance with negative preconditions can be translated into an equivalent instance in the formalism we work with, in linear time [21].

A state s is an assignment of truth values to all fluents in F. For notational convenience, we associate s with the set of literals of F which are true in s, so that 2^F is the set of all states. Given an instance $P = \langle F, I, O, G\rangle$, the state that results from the application of $o \in O$ to some state $s \in 2^F$ is the state $s[o] = \big(s \setminus \mathsf{eff}^-(o)\big) \cup \mathsf{eff}^+(o)$, which we only define when o is *applicable* in s, which means that $\mathsf{pre}(o) \subseteq s$. A *solution-plan* is a sequence of operators $o_1, \ldots, o_k$ from O such that the sequence of states $s_0, \ldots, s_k$ defined by $s_0 = I$ and $s_i = s_{i-1}[o]$ (for all $i \in \{1, \ldots, k\}$) satisfies $\mathsf{pre}(o_i) \subseteq s_{i-1}$ (for all $i \in \{1, \ldots, k\}$) and $G \subseteq s_k$. A *plan* is defined similarly but without the conditions concerning I and G.

For example, in a STRIPS formalisation of a 2x2x2 cube, fluents would be used to represent the different states of the cube, while the operators would model the action of rotating a face. A possible way to model the puzzle is sketched in Figure 2. Coding a Rubik's cube in STRIPS would be unwieldy, due to the large number of actions required, but it will allow us to easily illustrate certain notions since it is a well-known puzzle that requires little explanation. Various encodings of the problem into a planning

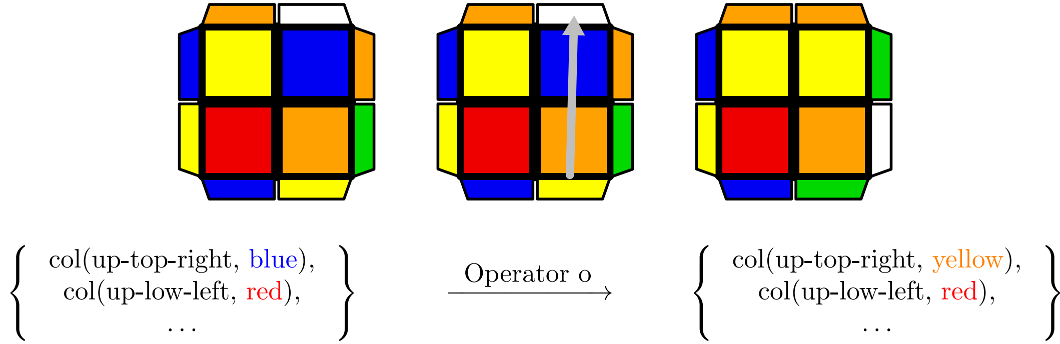


Figure 2: Top view of the 2x2x2 cube presented earlier, as well as the resulting configuration after the right face is rotated of a quarter of turn clockwise. Below both configurations, we partially represent the associated state resulting from a formalisation into STRIPS of the puzzle. For instance, the fluent col(up−top−right, blue), which is true in the first state, represents the fact that, on the upper face of the cube, the top right facet is blue. After some operator o (which, in this case, rotates the right face clockwise) is applied to the state, the facet is no longer blue, but has been replaced by a yellow one, as reflected in the resulting state.

language have been proposed, and both the 2x2x2 and the 3x3x3 Rubik's cube have been modeled in SAS$^+$ [6] (which is a more succinct planning language where variables are multivalued, whereas in STRIPS they are binary since a fluent is either true or false) or in PDDL with conditional effects [36](which is also significantly richer than STRIPS).

Another generic problem that can be compactly coded in STRIPS is the path-finding problem on a graph, described below in Section 2.3 which could easily be generalized to a multi-agent version.

A STRIPS planning instance represents succinctly a more expansive state-space, which we formalise as a *labeled transition system* (LTS). A labeled transition system is a tuple $\Theta = \langle S, L, T, s_I, S^G \rangle$, where S is a finite set of *states*, L is a finite set of *labels*, $T \subseteq S \times L \times S$ is the set of *transitions*, $s_I \in S$ is the *initial state*, and $S^G \subseteq S$ is the set of *goal states*. When given a STRIPS planning instance $P = \langle F, I, O, G \rangle$, the underlying state-space is naturally represented by the LTS $\Theta^P = \langle 2^F, O, T^P, I, S^G \rangle$, where the states of Θ^P are exactly the states of P, the initial state of Θ^P is the same as for P, the labels are the operators, and the transitions T^P of Θ^P between two states $s_1, s_2 \in 2^F$ are the ones that are possible by the application of an operator of P. More formally, we have $T^S = \{\langle s_1, o, s_2 \rangle \in 2^F \times O \times 2^F \mid \mathsf{pre}(o) \subseteq s_1 \text{ and } s_2 = s_1[o]\}$. The goal states of Θ^P are the ones that satisfy the goal condition of P, so that $S^G = \{s \in 2^F \mid G \subseteq s\}$.

2.2 The Complexity Class GI

This section introduces the complexity class GI, for which SI is later shown to be complete. GI is built around the Graph Isomorphism problem, which consists in determining whether two graphs have the same structure. Formally, a graph $\mathcal{G}$ is a pair (V, E), where V is a set of elements called *vertices* and E is a set of (unordered) pairs called *edges*. $\mathcal{G}$ is said to be *undirected* when edges are unordered pairs (denoted $\{v_1, v_2\}$ with $v_1, v_2 \in V$), and *directed* when edges are ordered pairs (denoted (v_1, v_2)).

The Graph Isomorphism problem consists in determining the existence of a bijection $u : V \to V'$ between the vertices of two undirected graphs $\mathcal{G}(V, E)$ and $\mathcal{G}'(V', E')$, such that the images of vertices linked by an edge in $\mathcal{G}$ are also linked by an edge in $\mathcal{G}'$ (and vice-versa). Formally, we require that the following condition holds:

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

Definition 1. *The complexity class GI is the class of problems with a polynomial-time Turing reduction to the Graph Isomorphism problem.*

Complexity class GI contains numerous problems concerned with the existence of an isomorphism between two non-trivial structures encoded explicitly. Such problems are often complete for the class.

For instance, the problems of finding an isomorphism between colored graphs, hypergraphs, automata, etc. are GI-complete [52]. In particular, we later use the following result:

Proposition 1 (Zemlyachenko *et al.* 1985[52], Ch. 4, Sec. 15)**.** *The* Directed Graph Isomorphism problem *is GI-complete.*

As with the Graph Isomorphism problem, an isomorphism between directed graphs $\mathcal{G}(V, E)$ and $\mathcal{G}'(V', E')$ is a bijection $u : V \to V'$ such that $(x, y) \in E$ iff $(u(x), u(y)) \in E'$.

In this paper, we consider another category of structures, called Finite Model, defined below. Finite models are also such that the related isomorphism existence problem is GI-complete.

Definition 2. *A Finite Model is a tuple $M = \langle V, R_1, \ldots, R_n \rangle$ where V is a finite non-empty set and each R_i is a relation on elements of V with a finite number of arguments.*

Let $M = \langle V, R_1, \ldots, R_n \rangle$ and $M' = \langle V', R'_1, \ldots, R'_n \rangle$ be two finite models. An isomorphism between M and M' is a bijection $u : V \to V'$ such that, for any $i \in \{1, \ldots, n\}$, for any set of elements $v_1, \ldots, v_m$ with m the arity of R_i, $R_i(v_1, \ldots, v_m)$ iff $R'_i(u(v_1), \ldots, u(v_m))$.

Proposition 2 (Zemlyachenko *et al.* 1985[52], Ch. 4, Sec. 15)**.** *The* Finite Model Isomorphism problem *is GI-complete.*

Class GI is believed to be an intermediate class between P and NP: the Graph Isomorphism problem can indeed be solved in quasi-polynomial time [2]. Although the problem is thought not to be NP-complete, no polynomial time algorithm is known.

2.3 Graph encodings into STRIPS

In this section, we present two ways to encode a graph $\mathcal{G} = (V, E)$ into a planning problem $P = \langle F, I, O, G \rangle$. These constructions are needed at various points in the rest of this paper, and only differ in that the first one concerns directed graphs, while the second one applies to undirected graphs.

The intuition behind these constructions is that they model an agent that can move on the graph, resting on vertices and moving along edges. An agent being on vertex v would thus be denoted by the state $\{v\}$, where all fluents other than v are false.

In order to make the construction and resulting proofs simpler to read, for any pair $(v_s, v_t) \in F^2$, we will denote $\mathsf{move}(v_s, v_t)$ the operator that represents a movement from vertex v_s to vertex v_t. Keeping in mind that $F = V$, we have, more formally:

$$\mathsf{move}(v_s, v_t) = \langle \{v_s\}, \{v_t\} \cup \neg(V \setminus \{v_t\}) \rangle$$

In the following construction, the vertices (resp. edges) of $\mathcal{G}$ are in bijection with the fluents (resp. operators) of P. In particular, we do not allow multi-edges.

Construction 1. *Let $\mathcal{G} = (V, E)$ be a directed graph. Let us build the planning problem $P_{\mathcal{G}} = \langle F, I, O, G \rangle$, where:*

$$\begin{aligned} F &= V \\ O &= \{\mathsf{move}(v_s, v_t) \mid (v_s, v_t) \in E\} \\ G &= I = \varnothing \end{aligned}$$

In the case of undirected graphs, the construction is essentially the same, except that moves are possible in both directions. This gives us the following definition:

Construction 2. *Let $\mathcal{G} = (V, E)$ be an undirected graph. Let us build the planning problem $P_{\mathcal{G}} = \langle F, I, O, G \rangle$, where F, I and G are defined as in Construction 1, but where:*

$$O = \{\mathsf{move}(v_s, v_t) \mid \{v_s, v_t\} \in E\}$$

Note that in Construction 2, we have $\mathsf{move}(v_s, v_t) \in O$ iff $\mathsf{move}(v_t, v_s) \in O$.

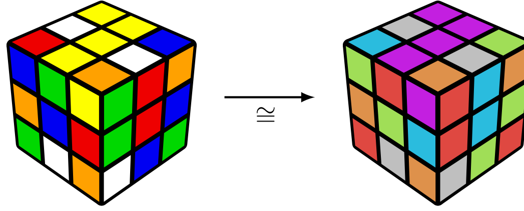


Figure 3: Two Rubik's cubes of the same shape, that are isomorphic. The one on the right is a copy of the left one, except that colors have been replaced consistently.

3 STRIPS Isomorphism Problem

This section is concerned with the problem of finding an isomorphism between two STRIPS planning problems. After introducing the notion of isomorphism between STRIPS instances that we use throughout this paper, we define problem SI more formally, and settle its complexity.

Intuitively, two STRIPS planning instances are isomorphic when they encode the exact same problem, but do not necessarily agree on the names given to their fluents and operators. For instance, let us consider the pair of 3x3x3 cubes depicted in Figure 3, where the one on the right-hand side is a the exact same cube as the other, but recolored: blue facets have been recolored in light green, yellow facets in purple, etc. The STRIPS instance that models the right cube is the same as the STRIPS instance of the left cube, except that fluents of the form color (X, yellow) have been replaced by fluents of the form color (X, purple), where X designates a facet. An isomorphism between both instances is then a bijective mapping that associates each fluent and operator of a problem to its counterpart in the other.

Definition 3 (Isomorphism between STRIPS instances)**.** *Let $P = \langle F, I, O, G\rangle$ and $P' = \langle F', I', O', G'\rangle$ be two STRIPS instances. An* isomorphism *from P to P' is a pair (υ, ν) of bijections $\upsilon : F \to F'$ and $\nu : O \to O'$ that respect the following three conditions:*

$$\forall o \in O,\ \nu(o) = \langle \upsilon(\mathit{pre}(o)), \upsilon(\mathit{eff}(o))\rangle \quad (2)$$

$$\upsilon(I) = I' \quad (3)$$

$$\upsilon(G) = G' \quad (4)$$

Where, by a slight abuse of notation, for any two sets F_1 and F_2 of fluents of F,

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

An immediate property of this definition is that it carries over all plans: any sequence $o_1, \dots, o_n$ of operators of O is a plan for P if, and only if, the corresponding plan $\nu(o_1), \dots, \nu(o_n)$ is a plan for P'. This homomorphism property is enforced by equation (2). Similarly, all solution-plans carry over, as enforced by the additional conditions defined in equations (3) and (4).

We can now introduce the problem SI formally, in order to analyze its complexity:

Problem 1. *STRIPS Isomorphism problem SI*
Input*:* *Two STRIPS instances P and P'*
Output*:* *An isomorphism (υ, ν) between P and P', if one exists*

Proposition 3. *The decision problem corresponding to SI is GI-complete*

The rest of this section is dedicated to the proof of this result. We first show the GI-hardness of the problem, and then that it belongs to GI.

Lemma 1. *SI is GI-hard*

Proof. The proof consists in a reduction from the Directed Graph Isomorphism problem to SI. Let $(\mathcal{G}, \mathcal{G}')$ be an instance of the Directed Graph Isomorphism problem, where $\mathcal{G} = (V, E)$ and $\mathcal{G}' = (V', E')$ are directed graphs. The proof relies on Construction 1, which gives us in polynomial time the STRIPS planning problems $P_{\mathcal{G}}$ and $P_{\mathcal{G}'}$.

We show that there exists an isomorphism $u : V \to V'$ between $\mathcal{G}$ and $\mathcal{G}'$ iff there exists an isomorphism (υ, ν) between $P_{\mathcal{G}}$ and $P_{\mathcal{G}'}$. The main idea consists in, first, identifying mappings u and υ, and second, showing that the morphism condition between the edges of graphs $\mathcal{G}$ and $\mathcal{G}'$ is enforced by the morphism condition on the operators of STRIPS instances $P_{\mathcal{G}}$ and $P_{\mathcal{G}'}$, and vice-versa.

($\Rightarrow$) Suppose that there exists a graph isomorphism $u : V \to V'$ between $\mathcal{G}$ and $\mathcal{G}'$, and let us show that there exists an isomorphism between $P_{\mathcal{G}}$ and $P_{\mathcal{G}'}$. We define the transformation ν on elements of O by $\nu(\langle \mathsf{pre}(o), \mathsf{eff}(o)\rangle) = \langle u(\mathsf{pre}(o)), u(\mathsf{eff}(o))\rangle$. We will show that $\nu : O \to O'$ is well-defined and that the pair (u, ν) forms an isomorphism between $P_{\mathcal{G}}$ and $P_{\mathcal{G}'}$. For any $o \in O$, by construction, there exists a unique pair $(v_1, v_2) \in V^2$ such that $o = \mathsf{move}(v_1, v_2)$. Thus, we have that

$$
\begin{aligned}
o \in O \ & \textit{iff } (v_1, v_2) \in E \\
& \textit{iff } (u(v_1), u(v_2)) \in E' \qquad \text{(since u is a graph isomorphism)} \\
& \textit{iff } \mathsf{move}(u(v_1), u(v_2)) \in O' \qquad \text{(by definition of move in Construction 1)} \\
& \textit{iff } \nu(\mathsf{move}(v_1, v_2)) \in O' \qquad \text{(by our definition of ν)} \\
& \textit{iff } \nu(o) \in O'
\end{aligned}
$$

Thus, we have shown that $P_{\mathcal{G}}$ and $P_{\mathcal{G}'}$ are isomorphic.

($\Leftarrow$) Suppose that $P_{\mathcal{G}}$ and $P_{\mathcal{G}'}$ are isomorphic, and that there exists an isomorphism (υ, ν) between them. We will show that there exists an isomorphism between $\mathcal{G}$ and $\mathcal{G}'$. By hypothesis, we have $\upsilon : F \to F'$ (or $\upsilon : V \to V'$) and $\nu : O \to O'$ two bijections.

In the following, we will denote by g and g' the bijections $g : E \to O$ and $g' : E' \to O'$, that exist by the construction (e.g. $g((v_1, v_2)) = \mathsf{move}(v_1, v_2)$).

Let us show that the function υ is a graph isomorphism between $\mathcal{G}$ and $\mathcal{G}'$. We have that, for any $e = (v_1, v_2) \in E$, $g(e) = \mathsf{move}(v_1, v_2) \in O$. So $\nu \circ g(e) = \nu(\mathsf{move}(v_1, v_2))$, and as such, $\nu \circ g(e) = \mathsf{move}(\upsilon(v_1), \upsilon(v_2)) \in O'$. Then, $g'^{-1} \circ \nu \circ g(e) = (\upsilon(v_1), \upsilon(v_2))$, but also $g'^{-1} \circ \nu \circ g(e) \in E'$. As a consequence, $(\upsilon(v_1), \upsilon(v_2)) \in E'$.

With similar arguments, as g, g' and ν are bijections, the converse can be shown. As a consequence, ν is a graph isomorphism between $\mathcal{G}$ and $\mathcal{G}'$. □

Lemma 2. *The decision problem corresponding to* SI *is in GI*

Proof. The proof follows a reduction from SI to the Finite Model isomorphism problem, as introduced in Definition 2.

The demonstration is based on the following construction: for any planning problem $P = \langle F, I, O, G\rangle$, we build the finite model

$$
\begin{aligned}
& M_P = \langle V, \mathcal{R}_F, \mathcal{R}_I, \mathcal{R}_O, \mathcal{R}_G, \mathcal{R}_{\mathsf{pre}}, \mathcal{R}_{\mathsf{eff}+}, \mathcal{R}_{\mathsf{eff}-}\rangle \\
& V = F \cup O \\
& \text{For } X \in \{F, I, O, G\},\ \mathcal{R}_X = X \\
& \text{For each } \mathcal{S} \in \mathcal{C},\ \mathcal{R}_{\mathcal{S}} = \left\{(o, f) \in V^2 \mid o \in O \textit{ and } f \in \mathcal{S}(o)\right\}
\end{aligned}
$$

where we recall that $\mathcal{C} = \{\mathsf{pre}, \mathsf{eff}^+, \mathsf{eff}^-\}$. Note that F and O form a partition of V. We will show that any two STRIPS planning problems P and P' are isomorphic iff M_P and $M_{P'}$ are isomorphic.

Let us denote $M_P = \langle V, \mathcal{R}_F, \ldots, \mathcal{R}_{\mathsf{eff}-}\rangle$ and $M_{P'} = \langle V', \mathcal{R}'_F, \ldots, \mathcal{R}'_{\mathsf{eff}-}\rangle$

($\Rightarrow$) Suppose that there exists an isomorphism (υ, ν) between P and P'. Define the mapping $g : V \to V'$ such that, for $x \in V$,

$$
g(x) = \begin{cases} \upsilon(x) & \text{if } x \in F \\ \nu(x) & \text{if } x \in O \end{cases} \tag{5}
$$

g is immediately a bijection, by hypothesis on (υ, ν). In addition, for $X \in \{F, I, O, G\}$, $\mathcal{R}_X(v)$ iff $\mathcal{R}'_X(g(v))$, by hypothesis on (υ, ν).

Let $o \in O$, $p \in F$. We have that, for any $\mathcal{S} \in \mathcal{C} = \{\mathsf{pre}, \mathsf{eff}^+, \mathsf{eff}^-\}$,

$$
\begin{aligned}
\mathcal{R}_{\mathcal{S}}(o, p) \ & \textit{iff } o \in O \textit{ and } p \in \mathcal{S}(o) && (6) \\
& \textit{iff } \nu(o) \in O' \textit{ and } \upsilon(p) \in \mathcal{S}(\nu(o)) && (7) \\
& \textit{iff } \mathcal{R}'_{\mathcal{S}}(\nu(o), \upsilon(p)) \\
& \textit{iff } \mathcal{R}'_{\mathcal{S}}(g(o), g(p))
\end{aligned}
$$

The passage from (6) to (7) is by definition of the isomorphism. The other equivalences follow mostly by definition. This proves that M_P and $M_{P'}$ are isomorphic.

($\Leftarrow$) Suppose that M_P and $M_{P'}$ are isomorphic, and that $g : V \rightarrow V'$ is an isomorphism between the two models. Let us define $\upsilon = g_{|F}$ (resp. $\nu = g_{|O}$) the restriction of g on the subdomain F (resp. O). Clearly, we have that $\upsilon : F \rightarrow F'$, as otherwise there would exist an element $v \in V$ such that $\mathcal{R}_F(v)$ but *without* $\mathcal{R}'_F(g(v))$, violating the isomorphism hypothesis on g. Similarly, we have that $\nu : O \rightarrow O'$.

With similar arguments as above, we have, for any $o \in O$,

$$\begin{aligned} & o = \langle \mathsf{pre}(o), \mathsf{eff}(o) \rangle \\ & \textit{iff } \forall p \in F, \forall \mathcal{S} \in \mathcal{C}, p \in \mathcal{S}(o) \Leftrightarrow \mathcal{R}_S(o, p) && (8) \\ & \textit{iff } \forall p \in F, \forall \mathcal{S} \in \mathcal{C}, p \in \mathcal{S}(o) \Leftrightarrow \mathcal{R}'_S(g(o), g(p)) && (9) \\ & \textit{iff } \forall p \in F, \forall \mathcal{S} \in \mathcal{C}, p \in \mathcal{S}(o) \Leftrightarrow g(p) \in \mathcal{S}(g(o)) && (10) \\ & \textit{iff } g(o) = \langle g(\mathsf{pre}(o)), g(\mathsf{eff}(o)) \rangle && (11) \\ & \textit{iff } \nu\,(o) = \langle \upsilon\,(\mathsf{pre}(o))\,, \upsilon\,(\mathsf{eff}(o)) \rangle \end{aligned}$$

The relations between the first line and (8), as well as between (9) and (10) hold by construction of M_P and $M_{P'}$. The equivalence between (8) and (9) comes from the hypothesis that g is an isomorphism.

This proves that (υ, ν) is a homomorphism, and thus an isomorphism by choice of its domain and codomain. $\square$

The results still hold if we do not enforce conditions (3) and (4), that the initial and goal states of P and P' are in bijection. Indeed, the hardness proof relies on a reduction from the Graph Isomorphism problem, with graphs that do not have initial or goal nodes, which renders trivial the initial and goal states of the construction. Conversely, the proof that SI belongs to class GI can include, or not, the relations $\mathcal{R}_I$ and $\mathcal{R}_G$ that take into account the information concerning initial and goal states, and still remain correct for the version of SI without conditions on the initial and goal states. This means that the hardness of SI comes from matching the inner structure of the state-space, and that additional properties on some states (like being initial states or goal states) do not impact significantly the complexity of the problem. This is consistent with our intuition of class GI: it is known that finding a color-preserving isomorphism between colored graphs (i.e., an isomorphism that conserves a given property on nodes) is also a problem that is complete for this class [52].

An isomorphism between a planning instance and itself is called an *automorphism*, or *symmetry*. Intuitively, a symmetry is a set of pairs of elements of the model (for instance operators or fluents) such that each element is interchangeable with the element it is paired with, in a way that does not modify the behaviour of the model, but only its description. It is well known that, when it comes to solving a CSP[9], a SAT instance [41], or a planning instance [19], finding the symmetries of the model as a preprocessing step can prove fruitful, since it can dramatically reduce the size of the search space (when symmetry-breaking techniques are implemented). In planning, in particular, finding the symmetries of a STRIPS planning instance boils down to finding operators or fluents that have similar roles, and preventing the planner from exploring branches of the search space that are too similar.

4 The STRIPS Subinstance isomorphism problem

Let us now introduce problems SSI-H and SSI, which are concerned with finding (different kinds of) isomorphisms between a planning instance P and some subinstance of another STRIPS instance P'. In this section, we settle the complexity of both problems, and show that they are NP-complete. We use this result in order to propose, in the next section, an algorithm for SSI and SSI-H. This algorithm is based on a reduction to SAT, assisted by a preprocessing phase that relies on constraint propagation.

In essence, a subinstance of a planning instance $P = \langle F, O, I, G \rangle$ is a planning instance P_s which shares the same initial state and goal as P, but whose fluents and operators F_s and O_s are subsets of the fluents and operators F and O of P. Note that any operator that occurs in O_s also appears *as is* in O, and is solely defined on the fluents F_s.

We begin by introducing the notion of *homogeneous subinstance isomorphism*. It is concerned with finding an isomorphism between P and a subinstance of P', which does not necessarily conserve the initial state and goal. That is to say, it maps operators of P to operators of P', and fluents alike, in a

way that ensures that all plans (and not just solution-plans) can be transferred from P' to P. In the case of such a homomorphism between two versions of the Rubik's cube, it would consist in mapping each operator of P to an equivalent operator in P' even though P' may have extra operators (such as 180-degree rotations for instance, which are equivalent to a sequence of two operators that rotate the same face).

Definition 4 (Homogeneous subinstance isomorphism)**.** *Consider two STRIPS instances $P = \langle F, I, O, G\rangle$ and $P' = \langle F', I', O', G'\rangle$. A* homogeneous subinstance isomorphism *from P to P' is a pair (υ, ν) consisting of the* injective *mapping $\upsilon : F \to F'$ and the mapping $\nu : O \to O'$ that respect condition (2) of Definition 3.*

Problem 2. *STRIPS Homogeneous Subinstance Isomorphism SSI-H*
Input*:* *Two STRIPS instances P and P'*
Output*:* *A* homogeneous subinstance isomorphism *(υ, ν) between P and P', if one exists*

A homogeneous subinstance isomorphism between P and P' is useful, for instance, in the case when the smaller problem P has already been solved: the image of a solution plan for P could be added as an operator to P' as a high-level action. In the case of a Rubik's cube, suppose that P and P' have the same set of operators, but the goal of P is simply to have one face all red whereas P' has the traditional goal that all faces are a uniform colour. Then there is a homogeneous subinstance isomorphism from P to P' and a solution-plan for P can be converted into an operator that could be added to P'. Such additional operators (often called *macro-operators* [28]) have been widely studied, as they have the potential to significantly speed up search [28]. The problem of learning them has been previously addressed [29, 5, 35], and indeed, a common approach is to reuse plans from a problem that has already been solved [17]. Our work provides a formal basis to detect instances whose solution-plans can be translated into a macro-operator for another instance. Indeed, let P'' be a modified version of P', so that a plan for P has been added to P'' as a new operator. Then there is a subinstance isomorphism between P' and P'' but in this case, the two problems have the same initial states and goals. This leads in turn to the following more precise notion of subinstance isomorphism which takes into account the information provided by the initial state and goal. This allows us to carry over *solution-plans* from one problem to the other: a solution for the smaller problem directly provides a solution for the larger problem.

Definition 5 (Subinstance isomorphism)**.** *A* subinstance isomorphism *from P to P' is a homogeneous subinstance isomorphism that respects conditions (3) and (4) of Definition 3.*

Proposition 4. *If there is a subinstance isomorphism from P to P' and P has a solution-plan, then so does P' (the image of the plan under the subinstance isomorphism).*

One application of Proposition 4 is in plan compilation. Suppose that there is a subinstance homomorphism from P to P' and that all solutions to P' (up to a certain length L) are stored in a compiled form that allows conditioning in polynomial time [13]. Then we can find the compiled form of all solutions of length at most L to (the isomorphic image of) P in polynomial time. Note, however, that if the aim is to detect unsolvable instances, the notion of embedding introduced in Section 6 provides a stronger test than the contrapositive of Proposition 4.

Problem 3. *STRIPS Subsintance Isomorphism SSI*
Input*:* *Two STRIPS instances P and P'*
Output*:* *A subinstance isomorphism (υ, ν) between P and P', if one exists*

The main difference between SI and SSI is that, in SSI, we relax the condition on the bijectivity of υ and ν, to account for the possible difference in size between P and P'. Injectivity of υ is still required in order to prevent fluents from being merged together by the mapping (but injectivity of ν and surjectivity of both mappings are relaxed). All other conditions remain the same. Note that, even when the injectivity of the mapping ν over the operators is not enforced, ν is still often injective. This is due to the injectivity of υ: two operators of O that have the same image by ν must also have the same preconditions, the same positive effects, and the same negative effects. Thus, except in the case where two operators are identical (except maybe for their names), the injectivity of ν is a consequence of the injectivity of υ, even when not explicitely enforced.

The main result of this section is presented below. The proof is based on a reduction from the Subgraph Matching problem [51], which is known to be NP-complete [10]. As such, we introduce that problem before stating our result. Essentially, it consists in finding a mapping g, that defines an isomorphism between an undirected graph $\mathcal{G}$ and the subgraph $(g(V), E' \cap g(V) \times g(V))$ of $\mathcal{G}'$ (i.e. the induced subgraph of $\mathcal{G}'$ on $g(V)$).

Problem 4. *Subgraph Matching problem*

Input: *Two undirected graphs $\mathcal{G}(V, E)$ and $\mathcal{G}'(V', E')$*

Output: *An injective mapping $g : V \to V'$ such that, for any $v_1, v_2 \in V$, $\{v_1, v_2\} \in E$ iff $\{g(v_1), g(v_2)\} \in E'$.*

Proposition 5. *The decision problem corresponding to* SSI *is NP-complete*

Proof. In order to prove that SSI belongs to NP, it suffices to resort to the certificate-based definition of the class NP, and observe that the mappings υ and ν constitute a polynomial size certificate that can be checked in polynomial time.

The proof that SSI is NP-hard consists in a reduction from the Subgraph Matching problem, which is straightforward with the construction that we proposed earlier.

Let $(\mathcal{G}, \mathcal{G}')$ be a pair of undirected graphs, an instance of the Subgraph Matching problem, and let us follow Construction 2 to build planning problems $P_{\mathcal{G}}$ and $P_{\mathcal{G}'}$. We show that there exists a subgraph matching g between $\mathcal{G}$ and $\mathcal{G}'$ iff there exists a subinstance isomorphism between $P_{\mathcal{G}}$ and $P_{\mathcal{G}'}$.

($\Rightarrow$) Suppose that there exists a subgraph matching $g : V \to V'$ between $\mathcal{G}$ and $\mathcal{G}'$. Then by construction, as $F = V$ and $F' = V'$, g is also an injective mapping between F and F'. In addition, let us define the mapping $\nu : O \to O'$ such that $\nu : \mathsf{move}(v_1, v_2) \mapsto \mathsf{move}(g(v_1), g(v_2))$. ν is well-defined, as $\{v_1, v_2\} \in E$ iff $\{g(v_1), g(v_2)\} \in E'$, so $\mathsf{move}(v_1, v_2) \in O$ iff $\mathsf{move}(g(v_1), g(v_2)) \in O'$. In addition, as g is injective, so is ν. As a consequence, (g, ν) is a subinstance isomorphism between $P_{\mathcal{G}}$ and $P_{\mathcal{G}'}$.

($\Leftarrow$) Suppose that there exists a subinstance isomorphism (υ, ν) between $P_{\mathcal{G}}$ and $P_{\mathcal{G}'}$. As above, $\upsilon : V \to V'$ is an injective mapping. In addition, we have that

$$
\begin{aligned}
&(v_1, v_2) \in E \\
\text{iff } &\mathsf{move}(v_1, v_2) \in O \qquad \text{(by definition of Construction 2)} \\
\text{iff } &\nu\left(\mathsf{move}(v_1, v_2)\right) \in O' \qquad \text{(since } \nu \text{ is a mapping from } O \text{ to } O'\text{)} \\
\text{iff } &\mathsf{move}(\upsilon(v_1), \upsilon(v_2)) \in O' \qquad \text{(by definition of a subinstance isomorphism)} \\
\text{iff } &(\upsilon(v_1), \upsilon(v_2)) \in E' \qquad \text{(by definition of Construction 2)}
\end{aligned}
$$

As a consequence, υ is a subgraph matching between $\mathcal{G}$ and $\mathcal{G}'$. □

In addition, it is clear that SSI-H is in NP. As the above proof of NP-hardness of SSI is independent of the initial and goal states, it also applies to the problem SSI-H.

Corollary 1. *The decision problem corresponding to* SSI-H *is NP-complete*

5 An algorithm for SSI

In this section, we present an algorithm for solving the SSI problem, for which the pseudo-code is presented in Algorithm 1. This algorithm is based on a compilation of the problem into a propositional formula, which is then passed to a SAT solver. It is completed by a preprocessing step, based on constraint propagation, that allows us to prune impossible mappings early on.

Given two STRIPS instances P and P', the algorithm outputs, when possible, a subinstance isomorphism (υ, ν). Algorithm 1 consists in two main phases. The first phase, that spans lines 2 to 8, consists in pruning as many associations between fluents (resp. operators) of problem P and fluents (resp. operators) of problem P' that are impossible, because of some syntactical inconsistencies (described below) that are then propagated. The second phase, that starts at line 9, consists in a search phase, by means of an encoding of the problem into a CNF formula, that is then passed to a SAT solver.

Algorithm 1 to find a subinstance isomorphism

Input: Two STRIPS instances P and P'
Output: A subinstance isomorphism between P and P' if one exists

```
1:  D := Initialize_domains(P, P')
    /* Prune impossible associations */
2:  Q := F ∪ O
3:  while Q ≠ ∅ do
4:      v := Q.Pop()
5:      Revise_domain(D, v)
6:      if D has been updated then
7:          if D(v) = ∅ then return UNSAT
8:          else  Q.Add({v' | v' related to v})
    /* Search phase through a SAT solver */
9:  φ := Encode_to_SAT(P, P', D)
10: return Interpret(Solver.Find_model(φ))
```

5.1 Pruning invalid associations

By *association* between fluents, we mean a pair $(f, f') \in F \times F'$ such that f' is a candidate for the value of $\upsilon(f)$. Similarly, we call an *association* between operators a pair $(o, o') \in O \times O'$ such that o' is a candidate for the value of $\nu(o)$. Detecting early on associations that can not be part of a valid subinstance isomorphism reduces the size of the search space.

In order to prune as many inconsistent associations as possible, we use a technique similar to constraint propagation, as commonly found in the constraint programming literature. The general idea is to maintain, for each fluent $f \in F$ of P, a *domain* $\mathcal{D}(f) \subseteq F'$ of fluents of P', that consists of the plausible candidates for the value of $\upsilon(f)$. Similarly, each operator $o \in O$ is assigned a domain $\mathcal{D}(o) \subseteq O'$, containing the plausible candidates for $\nu(o)$. In the following, we will call fluents and operators *variables*. The aim of the procedure presented below is to trim the domains of the different variables, thus alleviating the load left to the SAT solver.

The first step is to initialize the domains. For each fluent $f \in F$, we set its initial domain $\mathcal{D}(f)$ to either $I' \setminus G'$, $G' \setminus I'$, $I' \cap G'$ or $F' \setminus (I' \cup G')$, depending if f is in I only, G only, in both or in neither, respectively. Note that if the algorithm is given an SSI-H instance, the distinction between domains made above does not hold anymore. In this case, for all fluents $f \in F$, we set $\mathcal{D}(f) = F'$.

The initial assignment of the domains of operators $o \in O$, however, is based on *operator profiles*. For each operator $o \in O \cup O'$, we define the vector $\mathsf{profile}(o) \in \mathbb{N}^4$, called the *profile* of o. This vector numerically abstracts some characteristics of the operator, so that an operator $o \in O$ cannot be associated to operator $o' \in O'$ if $\mathsf{profile}(o) \neq \mathsf{profile}(o')$. In practice, $\mathsf{profile}(o)$ consists of the number of fluents in the precondition of o, the number of positive and negative fluents in the effect of o, and its number of *strict-delete* fluents. A fluent f is said to be *strict-delete* if $f \in \mathsf{pre}(o) \wedge f \in \mathsf{eff}^-(o)$. Then, we initialize the domain of each $o \in O$ so that

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

The second step is to propagate the additional constraints posed by these newly-found restrictions of the domains. The technique we propose is based on the concept of arc consistency, which is ubiquitous in the field of constraint programming. The idea consists in eliminating, from the domains of fluents (resp. operators), the candidate fluents (resp. operators) that have no support in the domain of some operator (resp. fluent).

More specifically, let us consider a fluent $f \in F$. When an operator $o \in O$ is such that f appears, negated or not, in its precondition or effect, then we say that o *depends* on f. Let us denote $d(f)$ the set of operators that depend on f. When $f' \in F'$, we define $d(f')$ in a similar fashion. Now suppose that $\upsilon(f) = f'$. As a consequence of equation (2) of Definition 3, each operator of $d(f)$ must have its image by ν in $d(f')$. Otherwise, f would appear in $\mathsf{pre}(o)$ or $\mathsf{eff}(o)$, but $\upsilon(f)$ would not appear in $\upsilon(\mathsf{pre}(o))$

nor $\upsilon(\mathsf{eff}(o))$. Thus, if for some operator $o \in d(f)$ no candidate operator for its image is in $d(f')$ (*i.e.*, $\mathcal{D}(o) \cap d(f') = \varnothing$), then it means that f' can not be chosen as the image of f.

In the following, we refine the argument of last paragraph by identifying $\mathsf{pre}(o)$ with $\mathsf{pre}(o')$ and $\mathsf{eff}(o)$ with $\mathsf{eff}(o')$. We thus have the following constraint for $\mathcal{D}(f)$, where $\mathcal{C} = \{\mathsf{pre}, \mathsf{eff}^+, \mathsf{eff}^-\}$:

$$\mathcal{D}(f) \subseteq \left\{ f' \;\middle|\; \begin{array}{l} \forall o \in O, \forall \mathcal{S} \in \mathcal{C} \text{ s.t. } f \in \mathcal{S}(o), \\ \exists o' \in \mathcal{D}(o) \text{ s.t. } f' \in \mathcal{S}(o') \end{array} \right\} \tag{12}$$

A similar case can be made for operators. Let $o \in O$ be any operator, and consider a candidate operator $o' \in O'$. In order for the morphism property to hold, in the case where $\nu(o) = o'$, for every fluent f of $\mathsf{pre}(o)$, for instance, there must exist in $\mathsf{pre}(o')$ a fluent that belongs to $\mathcal{D}(f)$. More generally and more formally, we have the following:

$$\mathcal{D}(o) \subseteq \left\{ o' \mid \forall \mathcal{S} \in \mathcal{C}, \forall f \in \mathcal{S}(o), \exists f' \in \mathcal{D}(f) \cap \mathcal{S}(o') \right\} \tag{13}$$

Algorithmically, we enforce these constraints using an adaptation of AC3 [34, 40]. The algorithm revolves around the revision of the variables' domains. Revising a variable v boils down to checking that all elements of its domain still comply with the necessary condition evoked earlier, which is either equation (12) if v is a fluent, or equation (13) if v is an operator. The main loop, depicted in Algorithm 1, then consists in revising all fluents and operators iteratively, by maintaining a queue Q of variables to revise (line 1). The algorithm begins by revising once each variable. If, during the revision of a variable v, the domain of v is altered by the procedure, then all variables that are related to v are added to the set of variables to revise later on (lines 5 to 9). We say that v' is related to v if v is a fluent and $v' \in d(v)$, or conversely.

If the domain of a variable is empty, then no isomorphism exists, and the procedure ends prematurely (line 7). Otherwise, the loop ends when there is no variable left to revise.

This procedure is often not sufficient to conclude, but greatly alleviates the pressure on the search phase, which we present in the following section.

5.2 Propositional encoding for SSI and SSI-H

In this section, we build the propositional formula φ evoked earlier, from whose models an isomorphism can be extracted. φ is built on the set of variables $\mathit{Var}(\varphi)$, where

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

The propositional variable f_i^j represents the association of fluent $i \in F$ to fluent $j \in F'$. Likewise, o_r^s represents the association of $r \in O$ to $s \in O'$.

In the rest of this section, we show how to build formula φ, which encodes the SSI problem input to Algorithm 1. φ consists of the conjunction of the following formulas, each of which enforces a different property.

The formula presented in (14) enforces that each fluent has an image which is unique. Similarly, by swapping f_i^j variables for o_i^j and adapting the domains of i and j, we enforce that each operator has an image by ν.

$$\bigwedge_{i \in F} \left(\bigvee_{j \in \mathcal{D}(i)} f_i^j \;\wedge \bigwedge_{\substack{j,k \in \mathcal{D}(i) \\ j \neq k}} (\neg f_i^j \vee \neg f_i^k) \right) \tag{14}$$

We now need to ensure that υ is injective, which is done through (15)

$$\bigwedge_{i \in F'} \bigwedge_{\substack{j,k \in F \\ j \neq k}} \neg f_j^i \vee \neg f_k^i \tag{15}$$

Note that, in Definition 4, we did not require the mapping on operators ν to be injective. However, even when not specifically enforced, ν is still injective in any (homogeneous) subinstance isomorphism, except potentially when two operators that have the exact same precondition and effects exist in P. Since this

is never the case in our set of benchmarks, we also enforce the injectivity of ν, in order to make the search for a mapping more efficient.

The morphism property is enforced by formulas (16) and (17), for each $\mathcal{S} \in \mathcal{C}$. More precisely, (16) ensures that, for any $\mathcal{S} \in \mathcal{C}$ and for any operator $o \in O$, we have $\upsilon(\mathcal{S}(o)) \subseteq \mathcal{S}(\nu(o))$. Conversely, (17) ensures that $\mathcal{S}(\nu(o)) \subseteq \upsilon(\mathcal{S}(o))$.

$$\bigwedge_{\substack{r \in O \\ s \in O'}} \left(o_r^s \longrightarrow \bigwedge_{i \in \mathcal{S}(r)} \bigvee_{j \in \mathcal{S}(s)} f_i^j \right) \tag{16}$$

$$\bigwedge_{\substack{r \in O \\ s \in O'}} \left(o_r^s \longrightarrow \bigwedge_{j \in \mathcal{S}(s)} \bigvee_{i \in \mathcal{S}(r)} f_i^j \right) \tag{17}$$

The formulas presented in (14) and (15), are immediately in CNF, and the size of their conjunction is in $\mathcal{O}(|F| \cdot |F'|^2 + |O| \cdot |O'|^2)$ assuming $|F| \leq |F'|$ and $|O| \leq |O'|$. In addition, the formulas presented in (16) and (17) can be readily converted into CNF by duplicating the implication in each clause, and then have a size $\mathcal{O}(|O| \cdot |O'| \cdot |F| \cdot |F'|)$.

The preprocessing step presented in Section 5.1 allows us to find φ. Indeed, if it is known that fluent $i \in F$ (resp. $r \in O$) cannot be mapped to fluent $j \in F'$ (resp. $s \in O'$), then f_i^j (resp. o_r^s) is necessarily false in any model of φ. As a consequence, as all formulas are in CNF, every positive occurrence of f_i^j is removed in the clauses of φ, while clauses where f_i^j appears negatively are simplified.

In order to adapt the algorithm for SSI-H, it suffices to initialize the domain of all fluents to F'. The others formulas and the rest of the algorithm remains the same.

An experimental evaluation of the algorithm can be found in Section 8, along with an evaluation of the adaptation of the same algorithm for a different notion of homomorphism, that we present in the next section.

6 STRIPS Embedding Problem

In this section, we introduce the notion of *embedding* between two planning instances P and P', which allows us to deduce that P has no solution-plan if P' has no solution-plan. We can define the notion of embedding of P' in P informally by saying that for each fluent $f' \in F'$ of P' there is a corresponding fluent $\upsilon(f') \in F$ in P and that when all fluents that are not in the image of υ are ignored, P' is isomorphic to a simplified version of P. In this context, simplifying a problem means (possibly) weakening its goals and the preconditions of operators, and (possibly) strengthening its initial state. This implies that if P' has no solution, then P also has no solution. Embedding an unsolvable problem into another problem P can help understand why P can not be solved. An example with Rubik's cubes is presented in Figure 4.

Definition 6. *Let $P = \langle F, O, I, G \rangle$ and $P' = \langle F', O', I', G' \rangle$ be two planning problems. An* embedding *of P' in P is a pair of functions $\upsilon : F' \to F$ and $\nu : O \to O'$ such that υ is injective and for each operator $o \in O$ that verifies*

$$(\mathit{eff}^+(o) \cup \mathit{eff}^-(o)) \cap \upsilon(F') \neq \emptyset \tag{18}$$

we have:

$$\upsilon\left(\mathit{eff}^+(\nu(o))\right) = \mathit{eff}^+(o) \cap \upsilon\left(F'\right) \tag{19}$$

$$\upsilon\left(\mathit{eff}^-(\nu(o))\right) = \mathit{eff}^-(o) \cap \upsilon\left(F'\right) \tag{20}$$

$$\upsilon\left(\mathit{pre}(\nu(o))\right) \subseteq \mathit{pre}(o) \cap \upsilon\left(F'\right) \tag{21}$$

In addition, we require that

$$\upsilon\left(G'\right) \subseteq G \cap \upsilon\left(F'\right) \tag{22}$$

$$\upsilon\left(I'\right) \supseteq I \cap \upsilon\left(F'\right) \tag{23}$$

When there exists an embedding of P' in P, we simply say that P' *embeds* in P.

Proposition 6. *If P' embeds in P and P' has no solution-plan, then neither does P.*

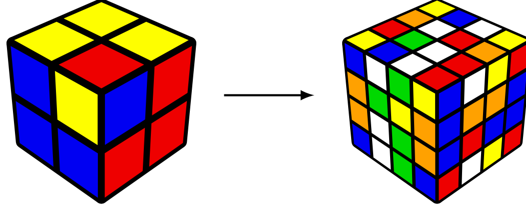


Figure 4: Through an invariance argument, it can be proven that the 2x2x2 cube on the left can not be solved. This cube can be embedded into the 4x4x4 cube on the right: the corners of the 2x2x2 cube can be mapped onto the corners of the 4x4x4 cube. When one only considers these parts of the 4x4x4 cube, we end up with a puzzle where each move that has an effect on the corner pieces can be mimicked on the smaller cube, and where the goal is to correctly position the corners. Other moves are ignored: for instance, moves that rotate one of the center slices leave the corners in place and are of no interest in this case. As no solution exists for the 2x2x2 cube, no sequence of moves exists that would correctly position the corner pieces on the 4x4x4, as we would end up with a contradiction otherwise. Thus, the 4x4x4 cube presented here can not be solved.

Proof. Let υ, ν be the functions defining the embedding. We define $\upsilon(P')$ in the natural way, as a copy of P' in which each fluent $f \in F'$ is replaced throughout by $\upsilon(f)$. Clearly, since υ is injective, P' and $\upsilon(P')$ are isomorphic, and as P' does not have a solution-plan, then neither does $\upsilon(P')$. Now suppose by contraposition that P has a solution-plan π. Let $\tilde{O}$ be the set of operators $o \in O$ such that $(\mathsf{eff}^+(o) \cup \mathsf{eff}^-(o)) \cap \upsilon(F') \neq \emptyset$. Then deleting from π all operators not in $\tilde{O}$ leaves a sequence of operators $\tilde{\pi}$ such that $\nu(\tilde{\pi})$ is a solution-plan for $\upsilon(P')$. ◻

As mentioned in Section 4, the notion of embedding is a generalisation of the notion of subinstance isomorphism in the sense that it allows us to detect more unsolvable instances than the contrapositive of Proposition 4. In fact, as the following proposition shows, we obtain an embedding by considering the *inverse* of the mapping between fluents in the subinstance isomorphism.

Proposition 7. *Let P and P' be STRIPS planning instances with non-empty sets of fluents. If (υ, ν) is a subinstance isomorphism from P to P', then there is an embedding $(\tilde{\upsilon}, \nu)$ of P' in P.*

Proof. It suffices to define $\tilde{\upsilon}$ and show that the conditions of an embedding are satisfied. Since (υ, ν) is a subinstance isomorphism, υ is injective, hence invertible on its image $\upsilon(F)$. Define $\tilde{\upsilon}(q) = \upsilon^{-1}(q)$ for $q \in \upsilon(F)$ and $\tilde{\upsilon}(q) = p_0$ otherwise, where $p_0 \in F$ is an arbitrary fluent. Clearly $\tilde{\upsilon}(F') = F$, so to show that $(\tilde{\upsilon}, \nu)$ is an embedding P' in P, it suffices to show:

1. $\forall o \in O$, $\tilde{\upsilon}(\mathsf{eff}(\nu(o))) = \mathsf{eff}(o)$ and $\tilde{\upsilon}(\mathsf{pre}(\nu(o))) \subseteq \mathsf{pre}(o)$,
2. $\tilde{\upsilon}(G') \subseteq G$, and
3. $\tilde{\upsilon}(I') \supseteq I$

But, since (υ, ν) is a subinstance isomorphism from P to P', we have

1. $\forall o \in O$, $\mathsf{eff}(\nu(o)) = \upsilon(\mathsf{eff}(o))$ and $\mathsf{pre}(\nu(o)) = \upsilon(\mathsf{pre}(o))$,
2. $G' = \upsilon(G)$, and
3. $I' = \upsilon(I)$

Applying $\tilde{\upsilon}$ to both sides of each equation, we obtain:

1. $\forall o \in O$, $\tilde{\upsilon}(\mathsf{eff}(\nu(o))) = \tilde{\upsilon}(\upsilon(\mathsf{eff}(o))) = \mathsf{eff}(o)$ and $\tilde{\upsilon}(\mathsf{pre}(\nu(o))) = \tilde{\upsilon}(\upsilon(\mathsf{pre}(o))) = \mathsf{pre}(o)$,
2. $\tilde{\upsilon}(G') = \tilde{\upsilon}(\upsilon(G)) = G$, and
3. $\tilde{\upsilon}(I') = \tilde{\upsilon}(\upsilon(I)) = I$

which completes the proof. ◻

Let us now give examples to highlight some differences between subinstance isomorphisms and embeddings. As a first example, there is an embedding between the two instances shown in Figure 4 but no subinstance isomorphism since the number of preconditions and effects of each operator is strictly greater in the 4x4x4 cube compared to the 2x2x2 cube. As a second example, consider the generic path-finding problem in a graph $\mathcal{G}$ encoded as a STRIPS instance $P_{\mathcal{G}}$, described in Section 2.3. If the initial state is a and the goal state b, then a solution-plan is a path from vertex a to vertex b. For simplicity, in the discussion that follows, we assume the same initial state and goal in each instance. If $\mathcal{G}_1$ is a proper subgraph of $\mathcal{G}_2$, in the sense that $\mathcal{G}_2$ has extra edges, then there is a subinstance isomorphism from $P_{\mathcal{G}_1}$ to $P_{\mathcal{G}_2}$, but no embedding from $P_{\mathcal{G}_1}$ to $P_{\mathcal{G}_2}$ (since the operators corresponding to the extra edges in $\mathcal{G}_2$ have no image in $P_{\mathcal{G}_1}$). Proposition 7 tells us that there is an embedding in the opposite direction, from $P_{\mathcal{G}_2}$ to $P_{\mathcal{G}_1}$. If, on the other hand, the set of vertices V_1 of $\mathcal{G}_1$ is a proper subset of the set of vertices V_2 of $\mathcal{G}_2$ and that $\mathcal{G}_2$ restricted to the vertices V_1 is a proper subgraph of $\mathcal{G}_1$, then $P_{\mathcal{G}_1}$ embeds in $P_{\mathcal{G}_2}$, but there is no subinstance isomorphism, in either direction, between $P_{\mathcal{G}_1}$ and $P_{\mathcal{G}_2}$ (since there is either some operator in $P_{\mathcal{G}_1}$ that has no image in $P_{\mathcal{G}_2}$ or there is some fluent in $P_{\mathcal{G}_2}$ with no image in $P_{\mathcal{G}_1}$).

In the rest of this section, we prove the NP-completeness of the problem concerned with finding whether an embedding of P' in P exists.

Problem 5. *STRIPS Embedding* SE

Input: *Two STRIPS instances P and P'*

Output: *An embedding (υ, ν) of P' in P, if one exists*

The proof is based on a reduction from the decision problem k-Independent Set, which we introduce below.

Definition 7 (Independent Set)**.** *Let $\mathcal{G}(V, E)$ be a graph. An independent set is a set $V' \subseteq V$ such that, for any $v_i, v_j \in V'$, we have $\{v_i, v_j\} \notin E$.*

Problem 6. k-*Independent Set*

Input: *A graph $\mathcal{G}(V, E)$*

An integer k

Output: Yes iff *there exists an independent set of size k in $\mathcal{G}$*

Proposition 8. k-*Independent Set is NP-complete[20]*

Proposition 9. *The decision problem corresponding to* SE *is NP-complete*

Proof. The problem is immediately in NP. In the following, in order to prove the NP-hardness of the problem, we build a reduction from k-Independent Set.

Let $\mathcal{G}(V, E)$ be an instance of k-Independent Set, and let us denote $V = \{v_1, \dots, v_{|V|}\}$. We can assume, without loss of generality, that $\mathcal{G}$ does not contain any reflexive edge (otherwise, we can remove the associated node, as they can not be part of any solution). Let us define the planning problem $P = \langle V, \varnothing, O, \varnothing \rangle$ such that

$$O = \{ \langle \varnothing, \{v_i, v_j\} \rangle \mid \{v_i, v_j\} \in E \}$$

In addition, we build $P' = \langle F', \varnothing, O', \varnothing \rangle$ such that $F' = \{1, \dots, k\}$ and

$$O' = \{ \langle \varnothing, \varnothing \rangle \} \cup \{ \langle \varnothing, \{i\} \rangle \mid i \in \{1, \dots, k\} \}$$

Let us show that there exists an embedding of P' in P iff $\mathcal{G}$ is a positive k-Independent Set instance.

($\Rightarrow$) Suppose that there exists an embedding $\upsilon : F' \to V$, $\nu : O \to O'$, and let us show that $\upsilon(F')$ is an independent set of $\mathcal{G}$ of size k.

Firstly, as υ is injective, we immediately have that $\upsilon(F')$ is of size k. Secondly, let $v_i, v_j \in \upsilon(F')$, with $v_i \neq v_j$. Let us show that $\{v_i, v_j\} \notin E$.

Suppose by contradiction that $\{v_i, v_j\} \in E$. Then by construction of O, $o = \langle \varnothing, \{v_i, v_j\} \rangle \in O$. We immediately have that o verifies condition (18), as $\mathsf{eff}^+(o) = \{v_i, v_j\} \subseteq \upsilon(F')$. As a consequence, o also verifies condition (19), i.e.,

$$\upsilon\left(\mathsf{eff}^+(\nu(o))\right) = \mathsf{eff}^+(o) \cap \upsilon\left(F'\right)$$

On the right-hand-side, we have that $\mathsf{eff}^+(o) \cap \upsilon(F') = \{v_i, v_j\}$, which is a set of size exactly 2. However, on the left-hand-side, we have, by definition of the domain of ν, that $\nu(o) = \langle \varnothing, \varnothing \rangle$ or $\nu(o) =$

$\langle \varnothing, \{v_l\} \rangle$ for some $v_l \in V$. In both cases, we have $|\mathsf{eff}^+(\nu(o))| \leq 1$, and thus $|\upsilon\left(\mathsf{eff}^+(\nu(o))\right)| \leq 1$, which is a contradiction.

As a consequence, we have that $\{v_i, v_j\} \notin E$, and $\upsilon(F')$ is a k-independent set.

($\Leftarrow$) Suppose that $\mathcal{G}$ is a positive instance of k-Independent Set. Then there exist k different vertices of V, that form an independent set. Let us denote them $v_1, \ldots, v_k \in V$ without loss of generality.

Let us build $\upsilon : F' \to V$ such that $\upsilon(i) = v_i$ for all $i \in \{1, \ldots, k\}$, so that $\upsilon(F')$ is an independent set. υ is immediately injective. In addition, we define $\nu : O \to O'$ such that

$$\nu\left(\langle \varnothing, \{v_i, v_j\} \rangle\right) = \begin{cases} \langle \varnothing, \{l\} \rangle \text{ if } l = i \text{ or } l = j \text{ for some } l \in F' \\ \langle \varnothing, \varnothing \rangle \text{ otherwise} \end{cases}$$

Let us show that this forms a correct embedding. As $\mathsf{pre}(o) = \mathsf{eff}^-(o) = \varnothing$, and $\mathsf{pre}(o') = \mathsf{eff}^-(o') = \varnothing$ for any $o' \in O'$, conditions (20) and (21) are always satisfied. All that is left to show is that condition (19) is satisfied when required, i.e., when (18) is satisfied.

Let $o = \langle \varnothing, \{v_i, v_j\} \rangle \in O$, and let us then suppose that $\mathsf{eff}^+(o) \cap \upsilon(F') \neq \varnothing$. We necessarily have $\mathsf{eff}^+(o) \cap \upsilon(F') = \{v_i\}$ for some v_i. Otherwise, we would have $\mathsf{eff}^+(o) \cap \upsilon(F') = \{v_i, v_j\}$, where $v_i \neq v_j$, but where $v_i, v_j \in \upsilon(F')$. By construction of the elements of O, we would have $\{v_i, v_j\} \in E$, which is impossible because they are vertices of the independent set, by hypothesis.

As a consequence, $o = \langle \varnothing, \{v_i, v_j\} \rangle$, with $i \in \{1, \ldots, k\} = F'$ and $j \in \{k+1, \ldots, |V|\}$. We have the following:

$$\begin{aligned} \upsilon\left(\mathsf{eff}^+(\nu(o))\right) &= \upsilon\left(\mathsf{eff}^+(\langle \varnothing, \{i\} \rangle)\right) \\ &= \upsilon(\{i\}) \\ &= \{v_i\} \\ &= \mathsf{eff}^+(o) \cap \upsilon\left(F'\right) \end{aligned}$$

Condition (19) is thus satisfied. As the initial state and goal are empty for both problems, conditions (22) and (23) are immediately satisfied. υ and ν therefore form a correct embedding of P' into P. □

7 Adapted algorithm for SE

In this section, we show how to adapt Algorithm 1, originally proposed for subinstance isomorphisms, to the case of SE. More precisely, we propose adapted constraints for the preprocessing step, as well as an adequate propositional encoding for SE. These steps being the only required modifications, the outline of the algorithm remains unchanged.

7.1 Constraint Propagation

In this section, we show how constraint propagation allows us to prune a number of inconsistent associations, as previously. For a start, we consider a variable for each operator $o \in O$, and a variable for each fluent $f' \in F'$, so that the respective domains of each of those variables contain the candidates for their image through υ or ν.

In addition, to make the propagation of constraints more efficient, we introduce variables of the form u_f, where $f \in F$ and $\mathcal{D}(u_f) \subseteq \{\top, \bot\}$. They translate the *usefulness* of their associated fluents. A fluent f is said to be useful in an embedding if it belongs to $\upsilon(F')$. More generally, we will say that a fluent is *useful* if it belongs to the image of *every* possible embedding. Intuitively, useful fluents are fluents that are likely to play a part in the enforcement of the properties imposed in Definition 6. This is why keeping in memory the domains of fluents of the form u_f allows us to know if f is useful ($\mathcal{D}(u_f) = \{\top\}$), potentially useful ($\mathcal{D}(u_f) = \{\top, \bot\}$) or not useful ($\mathcal{D}(u_f) = \{\bot\}$). Informally, when an operator of O has a useful fluent in its effect (appearing positively or negatively) and thus satisfies (18), we say that the operator is *active*. It means that the equations (19) to (21) have to be satisfied for this operator. This notion, however, is mostly useful in the propositional encoding, and we thus elaborate further on this point in the appropriate section.

7.1.1 Initial domains

First, as with subinstance isomorphisms, the initialization of the domains of fluents is straightforward: each fluent $f' \in F'$ has an initial domain set to either $I \setminus G$, $G \setminus I$, $I \cap G$ or $F \setminus (I \cup G)$, depending on whether f' is in I' only, G' only, both or neither, respectively.

We also use operator profiles to initialize the domains of operators. As previously, an operator profile is a vector of $\mathbb{N}^4$ that bears the information of the size of the precondition, the (positive and negative) effect of the operator, as well as the number of fluents that are strict-delete. However, in our current case, an operator $o \in O$ that has a different profile from $o' \in O'$ can still be mapped to o', depending on the set of fluents that constitute the image of υ.

We will only consider the domains of active operators, in the sense that operators that are not active do not have to fulfill any condition. So, let us suppose that o is active, and that we wish to map o to o' (i.e., $\nu(o) = o'$). Then it has to satisfy the condition:

$$\underbrace{\upsilon\left(\mathsf{pre}(\nu(o))\right)}_{|\cdot|=|\mathsf{pre}(o')|} \subseteq \underbrace{\mathsf{pre}(o) \cap \upsilon\left(F'\right)}_{|\cdot|\leq|\mathsf{pre}(o)|}$$

The equality on the left-hand side uses the fact that υ is injective. So as a consequence, we have that necessarily, $|\mathsf{pre}(o')| \leq |\mathsf{pre}(o)|$. A similar case can be made for effects, which allows us to define an operator profile $\mathsf{profile}(o)$ for each operator o of either problem. As a consequence, an operator o can only be mapped to o' if $\mathsf{profile}(o') \leq \mathsf{profile}(o)$, where $\leq$ compares each element of each tuple to its counterpart in the other tuple.

7.1.2 Propagation

In the following, we denote $\mathcal{C}_{\mathsf{eff}} = \{\mathsf{eff}^+, \mathsf{eff}^-\}$. The constraint below indicates that fluent f' can only be mapped to a fluent f if all operators o where f appears in eff can be matched with some $o' \in O'$. We also check that $\top \in \mathcal{D}(u_f)$, because if it is not the case, then mapping f' to f leads to a contradiction.

$$\mathcal{D}(f') \subseteq \left\{ f \;\middle|\; \begin{array}{l} \top \in \mathcal{D}(u_f) \quad \wedge \\ \forall o \in O,\ \forall \mathcal{S} \in \mathcal{C}_{\mathsf{eff}}\ s.t.\ f \in \mathcal{S}(o),\ \exists o' \in \mathcal{D}(o)\ s.t.\ f' \in \mathcal{S}(o') \end{array} \right\} \tag{24}$$

The constraint below only applies for operators that are active. We however maintain the domain of every operator as if it were active, as an inactive operator can be mapped to any other operator - in fact, our algorithm does not bother giving an image to inactive operators, as any operator would suffice. Note that, by enforcing a constraint for active operators only, we do not interfere incorrectly with the other constraints. Indeed, (24) and (26) are only concerned with operators that are active, and as a consequence, we are not at risk of pruning an otherwise possibly valid association. The first line below indicates that mapping o to o' can only be done if all fluents in $\mathsf{eff}(o)$ that are necessarily useful can be matched by some fluent of $\mathsf{eff}(o') \subseteq F'$. The second indicates that all fluents in $\mathsf{eff}(o')$ (resp. $\mathsf{pre}(o')$) must map to a fluent in $\mathsf{eff}(o)$ (resp. $\mathsf{pre}(o)$).

$$\begin{aligned} \mathcal{D}(o) \subseteq \quad & \{\, o' \mid \forall \mathcal{S} \in \mathcal{C}_{\mathsf{eff}}, \forall f \in \mathcal{S}(o)\ s.t.\ \mathcal{D}(u_f) = \{\top\}, \exists f' \in \mathcal{S}(o')\ s.t.\ f \in \mathcal{D}(f') \,\} \cap \\ & \{\, o' \mid \forall \mathcal{S} \in \mathcal{C}, \forall f' \in \mathcal{S}(o'), \exists f \in \mathcal{D}(f')\ s.t.\ \top \in \mathcal{D}(u_f) \ \wedge\ f \in \mathcal{S}(o) \,\} \end{aligned} \tag{25}$$

Despite the constraint only being applicable for active operators, it still allows us to simplify a number of clauses. Clauses found in (32), (34) and (35) (in the propositional encoding given in Section 7.2), which are the only formulas involving variables that model associations between operators, all have a_o for hypothesis, which is a variable that represents that o is active. In other words, the clauses that are simplified as a consequence of (25) are actually clauses that would have been satisfied nonetheless if o weren't active. All that is left to do is ensuring that some fluents can be detected as useful, as otherwise, the above constraint would be powerless. This is why the rest of this section, as well as the next, is dedicated to methods for finding useful fluents, both through constraint propagation techniques and through a more ad-hoc criterion.

Some useful fluents can be detected through the following constraint, which allows us to revise partially the usefulness of a fluent. Its role is mainly to prune $\top$ of the domain of the fluent, and thus marking a fluent as not useful. Conversely, in some rare cases, it can also detect that a fluent is useful, but detecting fluent usefulness is mostly left to a criterion that we present in the next section.

The second line, in the first set, translates the fact that if f were to make o active, then o has to have a possible image in which f finds a match.

$$\mathcal{D}(u_f) \subseteq \left\{ \top \;\middle|\; \begin{array}{l} \exists f' \in F' \; s.t. \; f \in \mathcal{D}(f') \land \\ \forall o \in O, \forall \mathcal{S} \in \mathcal{C}_{\text{eff}} \; s.t. \; f \in \mathcal{S}(o), \exists o' \in \mathcal{D}(o), \exists f' \in \mathcal{S}(o'), f \in \mathcal{D}(f') \end{array} \right\} \cup \quad (26)$$
$$\left\{ \bot \mid \nexists f' \in F' \; s.t. \; \mathcal{D}(f') = \{f\} \right\}$$

7.1.3 Fluent usefulness

As shown in (25), detecting that a fluent is useful is crucial for operators to be pruned efficiently. However, (26) can only detect fluent usefulness in the very specific case where a fluent's domain is a singleton. It is reasonable to assume that this will not occur often. We thus propose an additional method to detect useful fluents more efficiently, which is based on the following simple lemma that generalizes the above criterion:

Lemma 3. *Let $U \subseteq F$ and $f'_1, \ldots, f'_n \in F'$ be such that $\mathcal{D}(f'_1) = \ldots = \mathcal{D}(f'_n) = U$.*

- *If $|U| = n$, then every $f \in U$ is useful.*
- *If $|U| < n$, then no embedding exists.*

Proof. In the first case, if n fluents of $U \subseteq F$ have to be the images of n fluents of F' through an injective mapping υ, then υ is surjective on U, if such a mapping exists.

In the second case, the restriction $\upsilon' : \{f'_1, \ldots, f'_n\} \to U$ of υ should be injective, but can not be, as $n > |U|$. Thus, such a mapping does not exist. □

A stronger version of the lemma could be obtained by replacing the condition on the equality of the domains by the condition $\mathcal{D}(f'_i) \subseteq U$ for all $i \in \{1, \ldots, m\}$. However, finding sets U of fluents for which the number of domains that are subsets of U is greater than $|U|$ is a computationally costly problem. This justifies our choice to use our simpler version of the lemma, along with the fact that our experiments tend to indicate that it is sufficient in practice. This is not surprising: as our STRIPS representations come from planning instances encoded in PDDL, fluents and operators follow structures similar to one another. In particular, in typed PDDL instances, STRIPS fluents that result from PDDL objects of the same type often have identical domains.

Algorithmically, this criterion is used during the execution of AC3. More precisely, we add a token in the queue of variables to revise, that indicates that the above procedure (based on Lemma 3) should be called. This token is inserted between the fluent variables, that are revised first, and the operator variables, that are revised next. This ensures that as many useful fluents as possible are detected before any operator is revised, hence maximising the effectiveness of the revisions of operators.

7.2 Propositional encoding for SE

We now build the formula φ such that models of φ correspond to embeddings as defined previously (Definition 6). The variables of φ are:

$$Var(\varphi) = \left\{ f_i^j \;\middle|\; i \in F, j \in F' \right\} \cup \quad (27)$$
$$\left\{ o_r^s \mid r \in O, s \in O' \right\} \cup \quad (28)$$
$$\left\{ u_i \mid i \in F \right\} \cup \quad (29)$$
$$\left\{ a_r \mid r \in O \right\} \quad (30)$$

The propositional variable f_i^j represents that fluent $j \in F'$ is mapped to $i \in F$. Similarly, o_r^s represents that operator $r \in O$ is mapped to $s \in O'$. For the sake of clarity, despite the homomorphisms υ and ν being asymmetric, we use a consistent notation for fluents and operators. Indeed, even though we have $\upsilon : F' \to F$ and $\nu : O \to O'$, subscripts of our propositional variables represent elements (fluents or operators) of P, while superscripts always represent elements of P'. This allows us to remain consistent with the notation used in Section 5.

The notion of *active* operator, introduced earlier (Section 7.1), also allows us to make our encoding more succinct. Propositional variable a_r represents that $r \in O$ is active, which means that there is a useful fluent in its effect. Only active operators have to verify the morphism properties of Definition 6. For similar reasons, we use variables of the form u_i for each fluent $i \in F$, which represent the fact that i is *useful*, as introduced in Section 7.1.

We now show how to build the propositional encoding in itself. The formula φ consists in the conjunction of the sets of formulas presented below.

The following formula enforces that υ is correctly defined, and has an image for each element of its domain. We also add a similar formula where f_i^j is replaced by o_i^j, in order to ensure an image for each element of the domain of ν, adapting the domain and codomain as needed. An additional adaptation that we perform is that we check if the operator is active: if it is not, its image can be chosen to be an arbitrary operator, and indeed there is no need to find the image itself.

$$\bigwedge_{j \in F'} \left(\bigvee_{i \in \mathcal{D}(j)} f_i^j \ \wedge \bigwedge_{\substack{i_1, i_2 \in \mathcal{D}(j) \\ i_1 \neq i_2}} (\neg f_{i_1}^j \vee \neg f_{i_2}^j) \right) \tag{31}$$

$$\bigwedge_{r \in O} \left(\left(a_r \rightarrow \bigvee_{s \in \mathcal{D}(r)} o_r^s \right) \wedge \bigwedge_{\substack{s,t \in \mathcal{D}(r) \\ s \neq t}} (\neg o_r^s \vee \neg o_r^t) \right) \tag{32}$$

In addition, we enforce the injectivity of υ with the following formula:

$$\bigwedge_{i \in F} \bigwedge_{\substack{j,k \in F' \\ j \neq k}} \neg f_i^j \vee \neg f_i^k \tag{33}$$

Let $\mathcal{S} \in \{\mathsf{eff}^+, \mathsf{eff}^-\}$ and recall that $\upsilon : F' \rightarrow F$. The following formula enforces that $\upsilon\,(\mathcal{S}(\nu\,(o))) \subseteq \mathcal{S}(o) \cap \upsilon\,(F')$ for operators o that are active. It means that, if operator $r \in O$ is active and mapped to $s \in O'$ (*i.e.*, if $\nu\,(r) = s$), then for each fluent j in $\mathcal{S}(s) = \mathcal{S}(\nu\,(r))$, there must exist a fluent $i \in \mathcal{S}(r) \subseteq F$ such that $\upsilon\,(j) = i$ (and thus i is useful).

$$\bigwedge_{r \in O} \left(a_r \rightarrow \bigwedge_{s \in O'} \left(o_r^s \rightarrow \bigwedge_{j \in \mathcal{S}(s)} \left(\bigvee_{i \in \mathcal{S}(r)} f_i^j \right) \right) \right) \tag{34}$$

We also apply the above formula with $\mathcal{S} = \mathsf{pre}$, which immediately enforces (21).

Similarly, the following formula enforces that $\upsilon\,(\mathcal{S}(\nu\,(o))) \supseteq \mathcal{S}(o) \cap \upsilon\,(F')$. It starts by considering an operator $r \in O$ that is active (*i.e.*, that must satisfy conditions (19) and (20)), and then all fluents $i \in F$ that are useful and in $\mathcal{S}(r)$ (i.e. fluents i that are necessarily in $\mathcal{S}(r) \cap \upsilon\,(F')$). We then find the operator $s \in O'$ that is such that $\nu\,(r) = s$ (when o_r^s is not verified, the implication is true). In $\mathcal{S}(s) = \mathcal{S}(\nu\,(r))$, we must then find some fluent j that is such that $\upsilon\,(j) = i$. Hence the formula.

$$\bigwedge_{r \in O} \left(a_r \rightarrow \bigwedge_{i \in \mathcal{S}(r)} \left(u_i \rightarrow \bigwedge_{s \in O'} \left(o_r^s \rightarrow \bigvee_{j \in \mathcal{S}(s)} f_i^j \right) \right) \right) \tag{35}$$

Finally, the following formula enforces (23). It considers all fluents i that are useful and in I, and requires that they are associated to some fluent in I'.

$$\bigwedge_{i \in I} \left(u_i \rightarrow \bigvee_{j \in I'} f_i^j \right) \tag{36}$$

Propositional variables of the form u_i and a_o do not immediately represent objects of P or P', but rather properties about the homomorphism that we are building. We express them in the following formulas.

A fluent $i \in F$ is useful if it belongs to $\upsilon(F')$. As a consequence, we have, for every $i \in F$:

$$u_i \leftrightarrow \bigvee_{j \in F'} f_i^j$$

which is, in CNF,

$$\left(\neg u_i \vee \bigvee_{j \in F'} f_i^j\right) \quad \wedge \quad \bigwedge_{j \in F'} \left(u_i \vee \neg f_i^j\right) \tag{37}$$

In addition, an operator $o \in O$ is active if it satisfies condition (18). In the following formula, we write $f \in \mathsf{var}(\mathsf{eff}(o))$ to denote the set of fluents $f \in F$ such that either $f \in \mathsf{eff}(o)$ or $\neg f \in \mathsf{eff}(o)$. We thus have:

$$\bigwedge_{o \in O} \bigwedge_{i \in \mathsf{var}(\mathsf{eff}(o))} u_i \rightarrow a_o$$

Note that this does not perfectly translate our definition of active operator: we do *not* have that $o \in O$ is active *iff* it satisfies (18), but only that, if (18) is satisfied, then necessarily, it has to be marked active. It suffices for our encoding, as no problem arises when an operator o is marked as active during solving, even though it's not: an inactive operator can be mapped to any other operator. In particular, the set of models of the formula remains unchanged, compared to the case where we enforce strictly the activeness of operators.

8 Experimental evaluation

We implemented Algorithm 1 in Python 3.10, and used it to solve SSI, SSI-H and SE instances. We used the parser and translator of the Fast Downward planning system [24]. The SAT solver we used was Maple LCM [33], winner of the main track at SAT 2017. Experiments were run on a machine running Rocky Linux 8.5, powered by an Intel Xeon E5-2667 v3 processor, using at most 8GB of RAM, 4 threads, and 300 seconds per test. The code, sets of benchmarks, as well as our full results are available online[b].

Our set of benchmarks is formed out of ten sets found in previous International Planning Competitions, namely Barman, Blocks, Ferry, Gripper, Hanoi, Rovers, Satellite, Sokoban, TSP and Visitall. For each of these domains, we created what we call *STRIPS matching instances*, which are pairs of instances of the same domain. We did this for each possible pair of planning instances of each considered domain. Consequently, each domain of our set of benchmarks has a number of STRIPS matching instances that is quadratic in the number of planning instances in the associated IPC domain. We did not generate any new instance.

A STRIPS matching instance is a valid input for problems SSI, SSI-H and SE simultaneously. We thus evaluated our algorithm adapted for all three problems on the same set of benchmarks. We only report domains for which at least one instance can be solved. In particular, as no instance of Rovers was solved for any of the problems we considered, the domain does not appear in the results.

The goal of the experiments is twofold. First, the aim is to demonstrate that, despite the theoretical hardness of the problems, it is possible to find (homogeneous) subinstance isomorphisms and embeddings in reasonable time for problems of non-trivial size - or at least prove the absence of such mappings. Second, the goal is to show the efficiency of the pruning techniques presented in Section 5.1 and Section 7.1, that is to say, to prove that the additional cost of the preprocessing is outbalanced by the speed-up it provides to the search phase.

8.1 Absolute coverage

The coverage of our implementation on our set of benchmarks is shown in Table 1 for SSI, SSI-H, and SE. The table shows the absolute and relative numbers of instances of each problem on which our implementation terminates within the time and memory cutoffs.

The first point we notice is that problems SSI-H and SSI are often closely comparable in terms of hardness, except for some particular domains. These include domains TSP and Gripper, for which 25% and 216% more instances are solved when imposing no condition on the initial state and goal. For both

[b] `https://github.com/arnaudlequen/PDDLIsomorphismFinder`

Domain	SSI-H			SSI			SE		
	CP	NoCP	Av. Simp.	CP	NoCP	Av. Simp.	CP	NoCP	Av. Simp.
barman	0	0	N/A%	0	0	N/A%	19	0	100.0%
blocks	125	61	81.2%	118	57	81.3%	261	136	80.9%
ferry	100	0	65.9%	134	0	66.1%	199	161	90.0%
gripper	196	134	70.1%	62	60	69.5%	210	73	89.0%
hanoi	48	43	41.5%	52	47	41.5%	153	109	84.6%
satellite	26	16	87.1%	30	16	93.6%	167	41	97.2%
sokoban	189	0	100.0%	189	0	100.0%	73	0	100.0%
tsp	332	324	34.3%	263	255	38.5%	444	252	95.1%

Table 1: Number of instances of SSI-H, SSI and SE on which the implementation of our algorithm terminates within 300 seconds. For each problem, the first pair of columns shows the number of STRIPS matching instances solved with and without the constraint propagation-based preprocessing, respectively. The last column shows the average percentage of clauses that were eliminated from the propositional encoding, thanks to the pruning step. When the pruning step allowed us to conclude immediately (thus skipping the encoding into SAT altogether), we consider that all clauses have been simplified. As no SSI-H or SSI instance of Barman could be decided, the average number of simplified clauses is not applicable.

domains, we believe this to be due to the additional constraints in SSI, that turn all instances of these domains into negative ones. Not only are these negative instances harder for the SAT solver to decide, but also the pruning step is of little to no help. This is a consequence of the abundance of symmetries in these domains, on which we elaborate in the next section.

Finding embeddings, however, is significantly different from finding subinstance isomorphisms. In our entire set of benchmarks, our algorithm could not identify a single positive SE instance, except for STRIPS matching instances that were built upon two identical STRIPS instances. This leads us to believe that embeddings between real-world planning instances are rare, which does not necessarily render the notion of embedding impractical. Indeed, a small embeddable unsolvable instance could possibly be synthesized out of a larger unsolvable instance, and serve as a certificate of unsolvability, for example. We can also note that the IPC planning instances have been crafted to be solvable, which precludes the application of fast detection of unsolvability via a small embeddable unsolvable instance.

In Table 2, we present a few results on the absolute sizes of the problems that we solved during our experiments, within the time and memory limits. For a STRIPS planning problem $P = \langle F, I, O, G \rangle$, we denote $|P| = |F| + |O|$. As a STRIPS matching instance has two main dimensions, represented by the respective sizes of the planning instances that constitute it, we present two different ways of measuring the size of a matching instance. In the first set of columns of Table 2, the sum of both planning instances is considered, and we report the size of the STRIPS matching instance that maximizes that sum. With this metric, the larger instance is often disproportionately bigger than the smaller instance. This imbalance can be explained by the fact that the encoding into a propositional formula is of time and size $\mathcal{O}(|O| \cdot |O'| \cdot |F| \cdot |F'|)$, for all three problems. This kind of situation often represents the limit of what could be dealt with with our program, when trying to extract the solutions of a small problem out of a larger database for the domain (in the case of SSI) or detecting unsolvability of a large instance by embedding a small instance which is known to be unsolvable (in the case of SE).

In the second set of columns, to measure the size of an SSI or SSI-H instance, we consider the lexicographic order on pairs $(|P|, |P'|)$, and we report the biggest problem with respect to that metric. Likewise, we consider the lexicographic order on pairs $(|P'|, |P|)$ for SE. This gives an order of magnitude of the size of the biggest problem that can be identified as a subinstance of another one, under ideal conditions on this other instance.

8.2 Impact of the preprocessing

Our first observation is that the preprocessing step almost never holds back the algorithm: almost all instances of our test sets that can be solved without preprocessing are also solved when the preprocessing

Domain	SSI-H					SSI				
	Maximum sum			Max $\lvert P\rvert$		Maximum sum			Max $\lvert P\rvert$	
	$\lvert P\rvert$	$\lvert P'\rvert$	Sum	$\lvert P\rvert$	$\lvert P'\rvert$	$\lvert P\rvert$	$\lvert P'\rvert$	Sum	$\lvert P\rvert$	$\lvert P'\rvert$
blocks	73	3265	3338	505	586	73	3265	3338	505	505
ferry	434	730	1164	542	562	442	730	1172	562	562
gripper	464	604	1068	492	492	576	576	1152	576	576
hanoi	23	2853	2876	358	358	23	4887	4910	358	358
satellite	184	1845	2029	644	644	103	2187	2290	644	696
sokoban	2617	2703	5320	2617	2703	2617	2703	5320	2617	2703
tsp	108	990	1098	378	418	108	990	1098	418	418

Domain	SE					Average
	Maximum sum			Max $\lvert P'\rvert$		
	$\lvert P\rvert$	$\lvert P'\rvert$	Sum	$\lvert P\rvert$	$\lvert P'\rvert$	Op/Fluent
barman	3184	3184	6368	3184	3184	5.87
blocks	5290	73	5363	766	766	1.25
ferry	730	730	1460	730	730	1.09
gripper	604	604	1208	604	604	1.25
hanoi	13682	60	13742	625	625	8.04
satellite	646540	696	647236	15712	10895	84.77
sokoban	2703	1881	4584	2703	1881	0.72
tsp	990	378	1368	504	504	6.78

Table 2: Sizes of the biggest instances that can be solved by our implementation within the time and memory limits, for SSI-H, SSI and SE, as well as statistics on the set of benchmarks. For each problem, in the first set of columns, we consider the sum of the sizes of the planning instances that constitute the STRIPS matching instance. In the second set, we consider the size of the smallest planning instance among the pair that constitutes the instance. The last column of the second table gives statistics for the whole domain, by reporting the average number of operators per fluent.

step is performed. In all of the sets of benchmarks for which we report the results here, no significant slowdown of our algorithm was incurred by the preprocessing.

Furthermore, in many sets of benchmarks and for all three problems, the preprocessing greatly improves the overall performance of our implementation, so much so that some previously infeasible domains are now within the range of our algorithm. Such extreme cases include Sokoban, for which our algorithm is powerless without the pruning step: all 189 (resp. 73) instances solved by our implementation are outside the range of the preprocessing-less version of the algorithm, for SSI-H and SSI (resp. SE). In most cases, however, we observe a significant increase in the coverage of the algorithm, which remains nonetheless within the same order of magnitude. For example, for the Satellite domain in the case of SSI, 30 instances are solved when constraint propagation is enabled, whereas only 16 can be settled without it.

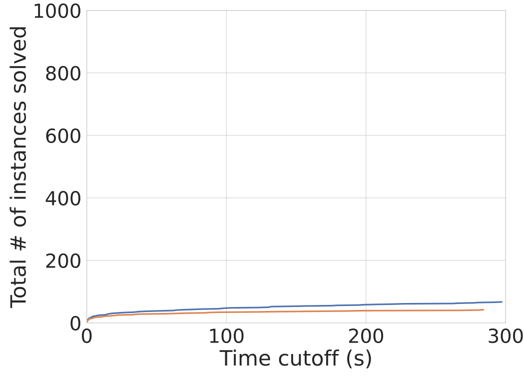
More specifically, in almost every case, the preprocessing step leads to a reduction of the size of the propositional encoding. This is shown by the columns labeled "Av. Simp." in Table 1, which represent the average proportion of clauses that are simplified as a consequence of the pruning step. Note that these particular columns only consider the instances for which the algorithm did not terminate before the end of the preprocessing, and thus had to resort to a propositional encoding. We remark that the highest percentages of simplified clauses are found in domains that contain little to no symmetries. For example, in Rovers, fluents represent entities that often have different types, and that are affected in different ways by operators. For instance, operators of the form navigate(rover, x, y) have a unique profile, and are not numerous. As a consequence, their respective domains remain small, which is something our algorithm makes the most of, especially in the case of subinstance isomorphisms.

On the contrary, for domains that contain lots of symmetries, the pruning step does not remove a significant number of associations, be it for SSI-H, SSI or SE. This is the case in Hanoi, where all operators have the same profile: except for the information provided by the initial and goal states, all disks are interchangeable, which does not allow our preprocessing to draw any conclusive result. The only information that can guide the search is encoded in the initial state, which we believe partly explains the slightly greater coverage of SSI over SSI-H.

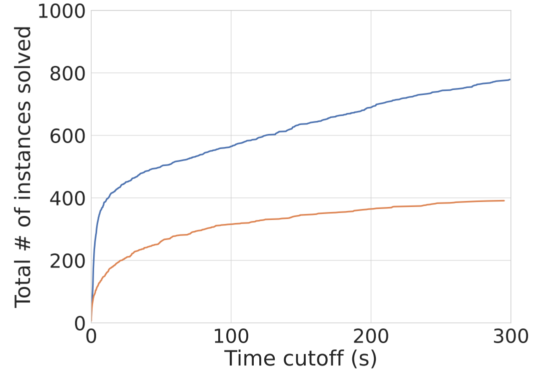
In some instances of our set of benchmarks, pruning alone suffices to find negative matching instances (i.e. instances that have no solution), be it for subinstance isomorphisms or embeddings. This happens when the majority of associations between fluents or between operators are ruled out, and the domain of some variable becomes empty. In these cases, our algorithm can return UNSAT prematurely, skipping the search phase altogether. As a direct consequence, our algorithm is most effective in detecting negative instances of the problems we consider. This is why the pruning step allows us to significantly increase our coverage on STRIPS matching instances that are negative, as shown in Figure 5, while our performance on positive instances is more modest, although significant.

In Table 3, we can also see that, in the majority of the domains we consider, the additional time required by the constraint propagation phase is negligible compared to the rest of the algorithm. In fact, be it in domains where it prunes many associations or in domains where its efficiency is limited, constraint propagation rarely takes more than a handful of seconds. As a consequence, some instances that would otherwise require a substantial amount of time are now solved almost immediately. This is made clear in Figure 5b, for instance: solving the easiest 400 negative SSI instances requires 5 minutes when pruning is not enabled, while it takes a handful of seconds when pruning is enabled.

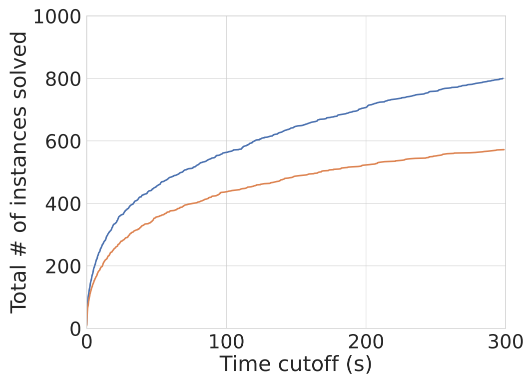
The most notable exception to this, however, is domain Satellite, for which the preprocessing step seems to be the most costly in time. However, it is also crucial, as the domain is generally hard for the preprocessing-less algorithm, which underperforms on this domain compared to other sets of benchmarks. In fact, for SE, in about 65% of this set of benchmarks, the preprocessing is not only necessary, but also sufficient to find that the instance is unsolvable: in most cases, it allows the algorithm to cut short, and detect a negative instance right away. When it comes to positive instances, only 2 additional positive instances are found with constraint propagation activated. Thus, most instances do not even require the SAT solving phase, hence the lower average time spent in compilation or solving. A similar phenomenon occurs for Barman, in the case of SE.



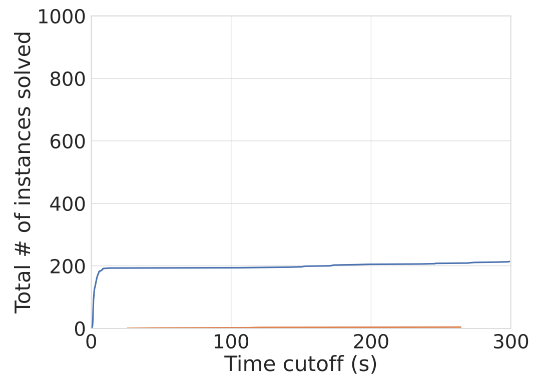
(a) SSI instances with positive outcome



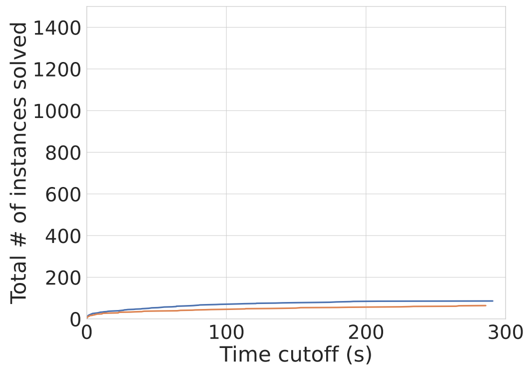
(b) SSI instances with negative outcome



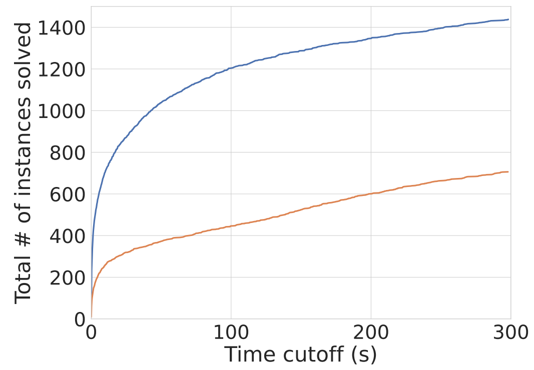
(c) SSI-H instances with positive outcome



(d) SSI-H instances with negative outcome



(e) SE instances with positive outcome



(f) SE instances with negative outcome

Figure 5: Number of SSI (top), SSI-H (middle) and SE (bottom) instances that can be solved by our implementation, as a function of the time cutoff. Blue/orange (upper/lower) curves correspond respectively to with/without pruning (constraint propagation preprocessing). The left column reports positive instances (for which a homomorphism could be found), while the right column reports negative instances (for which no homomorphism exists).

Domain	SSI-H				SSI				SE			
	CP	Comp.	Solving	Total	CP	Comp.	Solving	Total	CP	Comp.	Solving	Total
barman	-	-	-	-	-	-	-	-	6.6	0.0	0.0	6.6
blocks	0.3	58.3	32.4	90.7	0.3	52.4	40.8	93.3	2.0	73.3	22.6	96.1
ferry	0.2	76.3	114.3	190.7	0.2	78.3	106.8	185.2	1.3	41.4	8.1	49.6
gripper	0.2	26.2	27.6	53.8	0.1	16.4	28.4	44.9	0.6	15.5	1.4	16.9
hanoi	0.2	28.9	38.9	67.8	0.2	40.5	29.7	70.3	0.4	51.1	15.5	66.8
satellite	0.4	54.9	14.8	69.8	47.8	29.2	5.0	81.8	34.1	25.3	2.8	62.0
sokoban	2.5	0.0	0.0	2.5	2.5	0.0	0.0	2.5	1.0	0.0	0.0	1.0
tsp	0.1	26.8	21.4	48.2	0.1	14.4	19.3	33.8	0.1	26.3	8.1	34.4

Table 3: Average time, in seconds, spent in each of the main three steps of the algorithm: pruning (CP), compilation to SAT, and solving, respectively. The last column summarizes the average total running time of the algorithm for each domain of each problem. We only report instances that were successfully solved (either positively or negatively): results for SSI-H, SSI and SE are thus non-comparable, and Barman does not have statistics for SSI-H nor SSI, as no such instance was solved.

9 Related work

Horčík and Fišer[26] proposed a notion of endomorphism for classical planning tasks in finite domain representation (FDR). Mathematically, an endomorphism is a homomorphism where the domain and the codomain are identical. As STRIPS is a special case of FDR where variables have binary domains, their definition of an *FDR endomorphism* is directly comparable to (a special case of) our notions of homomorphisms. In essence, compared to a subinstance isomorphism, an FDR endomorphism only requires that $\mathsf{pre}(\nu(o)) \subseteq \upsilon(\mathsf{pre}(o))$, while we require equality (even though in practice, the CSP encoding Horčík and Fišer propose also requires it). In addition, our mapping between fluents is injective, whereas FDR endomorphisms do not impose an *injective* mapping between facts (i.e. literals, corresponding to variable-value assignments) but they do impose that each fact maps to a fact representing an assignment to the same variable. In the case of boolean variables, such as the ones we study in this paper, this means that an endomorphism may map a fluent p to its negation $\neg p$. However, if we assume all goals and preconditions are positive (as we have done in this paper) and given that goals and preconditions must map respectively to goals and preconditions, this prevents fluents occurring in goals or preconditions mapping to negative fluents. Of course, generalising our notion of subinstance isomorphism to allow positive fluents to map to negative fluents would be an interesting extension if there may be negative goals or preconditions. The main difference between our notions resides in their intent: while we wish to keep the planning model intact, Horčík and Fišer try to reduce its size as much as possible, by folding it over itself and deleting redundant operators.

Shleyfman *et al.* [44] defined the notion of *structural symmetry* for STRIPS planning instances. A structural symmetry is, in essence, a STRIPS Isomorphism as we defined, from an instance Π to itself, except that Equation (3) guaranteeing the stability of the initial state is not enforced. They show that a wide range of heuristics are invariant by structural symmetry, albeit their study excludes abstraction-based heuristics. Sievers *et al.* [46] extended the notion of structural symmetry to PDDL instances, and defined it as a permutation of constants, variables and predicates that conserves the semantics of the action schemas, as well as the initial and goal states. They prove that a structural symmetry of a PDDL instance induces a structural symmetry on the STRIPS instance obtained through grounding. Later, Röger *et al.* [38] showed that structural symmetries on PDDL instances can be leveraged to optimize its grounding, by pruning irrelevant operators through a relaxed reachability analysis.

A wide variety of notions of homomorphisms are defined not on the planning model itself, as we do in this paper, but on structures derived from it. Such structures notably include Labeled Transitions Systems (LTS), which are, in our context, graphs where edges are labeled with the names of operators, a node is designated as the initial state, and a set of nodes are designated as goal states. LTS's are naturally used to represent the state-space underlying any STRIPS instance. Symmetries of this state-space have also been studied, although through more compact representations, such as the Problem Description

Graph (PDG) [37, 44].

Abstractions aim at creating equivalence classes between the states of the LTS $\mathcal{T}$, in order to build an abstract LTS $\mathcal{T}^\alpha$ on which some desirable properties and features of the original LTS $\mathcal{T}$ are carried over. Bäckström and Jonsson [3] proposed a framework for analysing LTS-based abstractions, and understanding how some mathematical properties of an abstraction translate in terms of the structure of the set of paths it carries over to the abstract LTS. Horčík and Fišer studied the connection between the notions of FDR endomorphism previously mentioned, and the notion of LTS endomorphism, which they introduced in the same article [26]. The aim is similar to their previous method: computing a homomorphism from (a factored version of) the state-space to itself allows them to remove redundant operators.

In a preprocessing step, Pattern Databases (PDBs) [14] also reduce the size of the LTS underlying an FDR model by mapping it to a more concise one, obtained through *syntactic projection*. Syntactic projections take as input a set of fluents $S \subseteq F$, and forget all other fluents of the planning instance P, resulting in an instance $P_{|S}$. In our terminology, there is an embedding from the syntactic projection $P_{|S}$ to the original problem P: the mappings υ, ν that define this embedding are both identity. Embedding can be said to be a more general notion, in that when P' embeds in P, the problems P' and $P_{|\upsilon(F')}$ (where F' is the set of fluents of P') are not necessarily identical, notably because only inclusion rather than equality is imposed between preconditions, goals and the inital state (Equations (21), (22), and (23)).

More generally, computing the perfect heuristic on an abstract LTS $\mathcal{T}^\alpha$ is a common technique. The projection of the problem Π onto a subset of its fluents induces an LTS that is an abstract search space for Π, on which the perfect heuristic can be computed, and extrapolated to the states of Π [14]. In addition to PDBs, methods based on abstractions of LTS include the Merge & Shrink framework [45], or Cartesian abstractions [42]. However, in general, these works focus on computing the most relevant abstraction of a given structure, and the abstract LTS that results. Ultimately, the goal is to perform some computations on an abstract LTS, as they would have been impossible to achieve on the original LTS. In contrast, our work focuses on finding homomorphisms between pre-existing STRIPS models, without even considering the underlying state-space explicitly. In particular, the mappings that we create and work with are not necessarily surjective.

The embedding of an unsolvable planning instance Π into another instance Π' is, in itself, a proof that Π' is unsolvable. Even more so, when Π is small enough to be humanly understandable, it provides an explanation of why Π' is, in fact, unsolvable. Even though research in planning was historically focused on finding plans, there has been, in the recent years, a surge in interest in unsolvability detection. A wealth of techniques have been tailored to better handle unsolvable instances. This includes heuristics such as Merge & Shrink [25], PDBs [50], or even logic-based heuristics [48]. Other such techniques include, for instance, dead-end detection [12] and traps [32]. In addition to adapting already existing methods, some other work propose new, more specific approaches, sometimes inspired from other domains where unsatisfiability detection is more crucial, such as constraint programming [4] or propositional logic satisfiability [49], but also sometimes more ad-hoc [8].

In the case where an instance is solvable, a plan acts as a certificate of solvability. But in the case of unsolvable instance, there exists no immediate counterpart. Akin to proofs of unsolvability found in the SAT community, a few methods have been recently proposed to certify plan non-existence: Eriksson *et al* [15] have shown that a form of inductive certificates are suitable for planning instances, and later extended their work with a more flexible proof system [16]. Both methods can be adapted on existing planning systems. Other works have focused on explaining why a planner failed to find a plan, by providing humanly intelligible reasons and excuses [47, 22]. In this article, we showed that an embedding from P' to P, given that P' is a smaller planning instance that was proven to be unsolvable, can serve as a certificate of unsolvability for P. More than that, if P' is small enough that the reasons why it is unsolvable are clear, an embedding can help forge an intuition on why P itself is unsolvable.

10 Conclusion

In this article, we introduced the problem SI of finding an isomorphism between two planning problems, and showed that it is GI-complete. Afterwards, we introduced the notions of subinstance isomorphism and embedding, as well as the associated problems SSI and SSI-H, on the one hand, and SE on the other.

In addition to proving the NP-completeness of these problems, we proposed a generic algorithm for them, based on constraint propagation techniques and a reduction to SAT. We chose to use a reduction to SAT to take advantage of the efficiency of state-of-the-art SAT solvers, but, of course, a direct coding into CSP could take advantage of the automatic constraint propagation of CSP solvers.

The experimental evaluation of this algorithm shows that traditional constraint propagation in a preprocessing step can greatly improve the efficiency of SAT solvers. Even though some planning domains benefited greatly from an almost costless and effective pruning step, some others were left almost unchanged after the (fortunately short) preprocessing. A common characteristic of these latter problems is that they all have a significant amount of symmetries, which might be harnessed to some extent. Recent theoretical results seem to indicate that, in practice, this could be done with a reasonable computational effort. Indeed, finding a set of generators for the group of symmetries of a planning instance is a GI-complete problem [43], as well as the first step in the direction of breaking symmetries in the search for a homomorphism between planning instances. More generally, as the study of symmetries in constraint programming is a well-established field [9], we believe this avenue of future research to be promising.

On a more general note, it remains an interesting open question to identify which characteristics of problems in NP make them amenable to this hybrid CP-SAT approach.

Subinstance isomorphism and embedding define two distinct partial orders between STRIPS instances. This could help to perceive hidden structure in the space of all planning instances. In this vein, an interesting avenue for future research is the definition of formal explanations of (un)solvability via minimal solvable isomorphic subinstances or minimal unsolvable embedded subinstances.

References

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

[2] László Babai. Group, graphs, algorithms: the graph isomorphism problem. In Boyan Sirakov, Paulo Ney de Souza, and Marcelo Viana, editors, *Proceedings of the International Congress of Mathematicians*, pages 3319–3336, 2018.

[3] Christer Bäckström and Peter Jonsson. Abstracting abstraction in search with applications to planning. In Gerhard Brewka, Thomas Eiter, and Sheila A. McIlraith, editors, *KR*, 2012.

[4] Christer Bäckström, Peter Jonsson, and Simon Ståhlberg. Fast detection of unsolvable planning instances using local consistency. In Malte Helmert and Gabriele Röger, editors, *SOCS*, 2013.

[5] Adi Botea, Markus Enzenberger, Martin Müller, and Jonathan Schaeffer. Macro-ff: Improving AI planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research*, 24:581–621, 2005.

[6] Clemens Büchner, Patrick Ferber, Jendrik Seipp, and Malte Helmert. A comparison of abstraction heuristics for rubik's cube. In *ICAPS Workshop on Heuristics and Search for Domain-independent Planning*, 2022.

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

[8] Remo Christen, Salomé Eriksson, Florian Pommerening, and Malte Helmert. Detecting unsolvability based on separating functions. In Akshat Kumar, Sylvie Thiébaux, Pradeep Varakantham, and William Yeoh, editors, *ICAPS*, pages 44–52, 2022.

[9] David A. Cohen, Peter Jeavons, Christopher Jefferson, Karen E. Petrie, and Barbara M. Smith. Symmetry definitions for constraint satisfaction problems. *Constraints*, 11(2-3):115–137, 2006.

[10] 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, 1971.

[11] Martin C. Cooper, Arnaud Lequen, and Frédéric Maris. Isomorphisms between STRIPS problems and sub-problems. In Christine Solnon, editor, *CP 2022*, volume 235 of *LIPIcs*, pages 13:1–13:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[12] Bence Cserna, William J. Doyle, Jordan S. Ramsdell, and Wheeler Ruml. Avoiding dead ends in real-time heuristic search. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *AAAI*, pages 1306–1313, 2018.

[13] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[14] Stefan Edelkamp. Planning with pattern databases. In *ECP*, 01 2001.

[15] Salomé Eriksson, Gabriele Röger, and Malte Helmert. Unsolvability certificates for classical planning. In Laura Barbulescu, Jeremy Frank, Mausam, and Stephen F. Smith, editors, *ICAPS*, pages 88–97, 2017.

[16] Salomé Eriksson, Gabriele Röger, and Malte Helmert. A proof system for unsolvable planning tasks. In Mathijs de Weerdt, Sven Koenig, Gabriele Röger, and Matthijs T. J. Spaan, editors, *ICAPS*, pages 65–73, 2018.

[17] Richard Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3(1-3):251–288, 1972.

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

[19] Maria Fox and Derek Long. The detection and exploitation of symmetry in planning problems. In Thomas Dean, editor, *IJCAI*, pages 956–961, 1999.

[20] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[21] Hector Geffner and Blai Bonet. *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool Publishers, 2013.

[22] Moritz Göbelbecker, Thomas Keller, Patrick Eyerich, Michael Brenner, and Bernhard Nebel. Coming up with good excuses: What to do when no plan can be found. In Ronen I Brafman, Hector Geffner, Jörg Hoffmann, and Henry A Kautz, editors, *ICAPS*, pages 81–88, 2010.

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

[24] Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[25] Jörg Hoffmann, Peter Kissmann, and Alvaro Torralba. "distance"? who cares? tailoring merge-and-shrink heuristics to detect unsolvability. In Torsten Schaub, Gerhard Friedrich, and Barry O'Sullivan, editors, *ECAI 2014*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 441–446. IOS Press, 2014.

[26] Rostislav Horčík and Daniel Fišer. Endomorphisms of classical planning tasks. In *AAAI*, pages 11835–11843, 2021.

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

[28] Richard E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26(1):35–77, 1985.

[29] Richard Earl Korf. *Learning to solve problems by searching for macro-operators*. PhD thesis, Carnegie Mellon University, USA, 1983. AAI8425820.

[30] Arnaud Lequen, Martin C. Cooper, and Frédéric Maris. Homomorphisms and embeddings of strips planning models. arXiv, 2024.

[31] 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*, pages 4152–4159, 8 2021.

[32] Nir Lipovetzky, Christian J. Muise, and Hector Geffner. Traps, invariants, and dead-ends. In Amanda Jane Coles, Andrew Coles, Stefan Edelkamp, Daniele Magazzeni, and Scott Sanner, editors, *ICAPS*, pages 211–215, 2016.

[33] Mao Luo, Chu-Min Li, Fan Xiao, Felip Manyà, and Zhipeng Lü. An effective learnt clause minimization approach for CDCL SAT solvers. In Carles Sierra, editor, *IJCAI 2017*, pages 703–711, 2017.

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

[35] Steven Minton. Selectively generalizing plans for problem-solving. In Aravind K. Joshi, editor, *IJCAI*, pages 596–599, 1985.

[36] Bharath Muppasani, Vishal Pallagani, Biplav Srivastava, and Forest Agostinelli. On solving the rubik's cube with domain-independent planners using standard representations. arXiv, 2023.

[37] Nir Pochter, Aviv Zohar, and Jeffrey S. Rosenschein. Exploiting problem symmetries in state-based planners. In Wolfram Burgard and Dan Roth, editors, *AAAI*, 2011.

[38] Gabriele Röger, Silvan Sievers, and Michael Katz. Symmetry-based task reduction for relaxed reachability analysis. In *ICAPS*, pages 208–217, 2018.

[39] Tomas Rokicki and Morley Davidson. God's number is 26 in the quarter-turn metric. `https://web.archive.org/web/20220906131819/https://www.cube20.org/qtm/`, 2022. Accessed: 2022-09-6.

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

[41] Karem A. Sakallah. Symmetry and satisfiability. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 509–570. IOS Press, second edition edition, 2021.

[42] Jendrik Seipp and Malte Helmert. Counterexample-guided cartesian abstraction refinement for classical planning. *Journal of Artificial Intelligence Research*, 62:535–577, 2018.

[43] Alexander Shleyfman and Peter Jonsson. Computational complexity of computing symmetries in finite-domain planning. *Journal of Artificial Intelligence Research*, 70:1183–1221, may 2021.

[44] Alexander Shleyfman, Michael Katz, Malte Helmert, Silvan Sievers, and Martin Wehrle. Heuristics and symmetries in classical planning. In Blai Bonet and Sven Koenig, editors, *AAAI*, pages 3371–3377, 2015.

[45] Silvan Sievers and Malte Helmert. Merge-and-shrink: A compositional theory of transformations of factored transition systems. *Journal of Artificial Intelligence Research*, 71:781–883, 2021.

[46] Silvan Sievers, Gabriele Röger, Martin Wehrle, and Michael Katz. Structural symmetries of the lifted representation of classical planning tasks. *ICAPS Workshop on Heuristics and Search for Domain-independent Planning*, 1:67–74, 2017.

[47] Sarath Sreedharan, Siddharth Srivastava, David E. Smith, and Subbarao Kambhampati. Why can't you do that HAL? explaining unsolvability of planning tasks. In Sarit Kraus, editor, *IJCAI*, pages 1422–1430, 2019.

[48] Simon Ståhlberg, Guillem Francès, and Jendrik Seipp. Learning generalized unsolvability heuristics for classical planning. In Zhi-Hua Zhou, editor, *IJCAI*, pages 4175–4181, 2021.

[49] Marcel Steinmetz and Jörg Hoffmann. State space search nogood learning: Online refinement of critical-path dead-end detectors in planning. *Artificial Intelligence*, 245:1–37, 2017.

[50] Alvaro Torralba. Sympa: Symbolic perimeter abstractions for proving unsolvability. *UIPC 2016 planner abstracts*, pages 8–11, 2016.

[51] Julian R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.

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