

Analysis of planning instances without search

Arnaud Lequen^{a,*} and Martin C. Cooper^a

^a *IRIT, Université Toulouse III - Paul Sabatier, 118 route de Narbonne, Toulouse, France*

E-mails: Arnaud.Lequen@irit.fr, Martin.Cooper@irit.fr

Abstract. In order to prove classical planning instances unsolvable, state-of-the-art planners resort to a state-space search. However, we show here that an incomplete, yet computationally efficient criterion is sometimes sufficient to immediately identify as unsolvable a wide range of planning instances. Based on linear and integer programming, we show in this paper how it can be leveraged, should it fail at first. This criterion is the keystone of various techniques we propose to rewrite and enhance the STRIPS model, so as to gather new information about it. If the newly-found bits of information are not sufficient to identify the instance as unsolvable, they can be reinvested later to speed up a complete algorithm.

Keywords: Classical planning, Linear Programming, Unsatisfiability

1. Introduction

Current classical planners resort to a search, with the aim of finding a sequence of actions that satisfies a given goal. They often start with the assumption that such a plan exists, and for the past few decades, significant work has been done on designing more and more efficient techniques to find solution-plans. However, various reasons may lead to an instance not admitting any solution. Search-based planners will then explore the state-space in its entirety, potentially cutting branches of the search tree, until they realize no plan can be found. The detection of states that cannot lead to any solution is often a byproduct of the heuristics used during search: an infinite heuristic value for an admissible heuristic is synonym of a dead-end state.

This is why in the recent years, there has been a renewed interest in detecting unsolvable planning instances, as illustrated by the 2016 Unsolvability International Planning Competition (Unsolvability IPC). Various techniques have been developed in the last couple of decades, such as dead-end formulas [4], traps [9, 13]. However, all of these methods are based on the exploration of the state-space.

In this article, we propose to leverage a linear programming- and integer programming-based criterion to iteratively refine a planning model, to show its unsolvability. The criterion we use is fast to compute, and allows us to quickly recognize a wide range of unsolvable planning instances. However, it is not complete, in the sense that it may not recognize some unsolvable instances as such. Nevertheless, we show how to use it to iteratively refine the planning model, and keep gathering additional information about the instance with the aim that our procedure can detect that it is unsolvable.

Most of our techniques to gather information are based on a simple schema: after testing the solvability of planning instances Π' that are derived from the initial planning problem Π given as input, we deduce additional information about the problem Π if Π' is unsolvable. For instance, if the instance Π' , which is Π where operator a was removed, is proven to be unsolvable, then it means that a appears in all solution-plans of Π . In the case where one can efficiently detect some unsolvable planning instances, then lots of such derived instances Π' can be tested successfully. As the criterion we use is incomplete but fast, even though it often fails to detect unsolvable instances, it still manages to help gather new information, as lots of tests can be made in a reasonable time. As more and more information is known about the planning instance, the mathematical program on which the criterion is based can also be enriched with the new knowledge, so that it can detect additional unsolvable instances.

*Corresponding author. E-mail: Arnaud.Lequen@irit.fr.

More generally, being able to detect planning instances that have no solution can have various applications in itself. For instance, consider the case where an instance models the attacks of a malicious user may perform on a system, with the goal of accessing restricted data. Finding that no sequence of actions may achieve this shows that the system is secure.

The paper is organized as follows. In Section 2, we introduce our formalism and notations for classical planning. In Section 3, we present the mathematical-programming-based criterion we use throughout this paper. In Section 4, we show how to design tests to gather new information about a planning model. In Section 5, we report our experimental trials on standard sets of benchmarks. Section 6 reviews related work, while Section 7 is devoted to a discussion and perspectives on our findings.

2. Background

STRIPS planning instance. A STRIPS planning instance is a tuple $\Pi = \langle F, I, O, G \rangle$ such that F is a set of propositional variables called *fluents*, and I is a set of fluents of F , called the *initial state*. G is a set of *literals* of F , such that no literal appears at the same time as its negation, and is called the *goal*. We will denote G^+ the set of positive literals of G , and G^- the set of fluents that appear negated in G . Finally, O is a set of *operators*: operators $a \in O$ are of the form $a = \langle \text{pre}(a), \text{eff}(a) \rangle$. $\text{pre}(a) \subseteq F$ is the *precondition* of a , which is a set of positive literals. $\text{eff}(a)$ is the *effect* of a , and is a set of literals of F . We will denote $\text{eff}^+(a) = \{f \in F \mid f \in \text{eff}(a)\}$ the set of *positive* effects of a , and $\text{eff}^-(a) = \{f \in F \mid \neg f \in \text{eff}(a)\}$ its *negative* effects.

Note that we define a version of STRIPS with negative goals. The original STRIPS formulation only specified positive goals, and is not any less expressive: any instance with negative goals can be translated, in polynomial time, into an equivalent instance without negative goals. We nonetheless allow negative goals in our formulation of STRIPS, since later in this paper we investigate the possibility of adding negative goals in order to strengthen a STRIPS instance. But one should keep in mind that most planning instances (and in particular, the ones used in our set of benchmarks) come with positive goals only: this is why we assume G^- is empty, except in the section where we consider adding negative goals.

Without loss of generality, we assume that for all operators a , $\text{eff}^+(a) \cap \text{eff}^-(a) = \emptyset$. In addition, we will also suppose that $\text{eff}^+(a) \cap \text{pre}(a) = \emptyset$, otherwise the redundant fluents from the effects can be removed. Any planning instance which does not satisfy these criteria can be transformed, in polynomial time, into an equivalent instance that complies with them.

States and plans. A state s is an assignment of truth values to all fluents in F . For notational convenience, we associate s with the set of fluents of F which are true in s . An operator a can be applied to states of Π that verify its preconditions. More formally, for any state s , if $\text{pre}(a) \subseteq s$ then we define the result of the application of a to s as $s[a] = (s \setminus \text{eff}^-(a)) \cup \text{eff}^+(a)$.

Given an instance $\Pi = \langle F, I, O, G \rangle$, a *plan* is a sequence of operators $\pi = a_1, \dots, a_k$ from O such that there exists a sequence of states s_0, \dots, s_k , such that, for all $i \in 1, \dots, k$, the operator a_i is applicable in s_{i-1} , and that $s_i = s_{i-1}[a_i]$. A plan is a *solution-plan* if we have, in addition, $s_0 = I$, $G^+ \subseteq s_k$ and $G^- \cap s_k = \emptyset$. We say that a fluent f is established (resp. deleted) by some occurrence of an operator $a \in O$ in π if f is false (resp. true) in some state s_i , but true (resp. false) after the application of a , in state $s_{i+1} = s_i[a]$. In the rest of this paper, we will refer to solution-plans as simply plans.

3. Detecting unsolvable instances by LP

This section introduces two equivalent criteria that we use, and extend, to detect a planning instance's unsolvability. These criteria are incomplete, in the sense that they can not detect all unsolvable planning instances by themselves. However, they require very limited computational resources, and are fast to run, as they are based on linear or integer programming. We will show later how to leverage those properties in order to make the most of these criteria when they are not able to detect an instance's unsolvability by themselves.

3.1. Potential-based argument

The first linear programming formulation that we worked with is based on the following argument. Suppose that we have a numerical function $\Phi : F \rightarrow \mathbb{R}^+$, that associates a *potential* to each fluent. We can then naturally define the potential of a state $s \subseteq 2^F$ as $\Phi(s) = \sum_{f \in s} \Phi(f)$. If one can prove that all goal states have a higher potential than the initial state, but the application of any operator a to any state s leads to a state s' of lesser (or equal) potential, then the planning instance has no solution-plan.

Such a function Φ can be found thanks to the following observation. In any plan, the potential of a state s' solely depends on the previous state s , and on the operator a that was applied such that $s[a] = s'$. In this case, we will say that a induced an increase in potential of $\Delta\Phi_a(s) = \Phi(s') - \Phi(s)$. One can remark that there exists an upper bound for $\Delta\Phi_a(s)$, which does not depend on s but only on a . Indeed, in the limit case, all fluents $f \in \text{eff}^+(a)$ are effectively established by a , but no fluent $f' \in \text{eff}^-(a)$ is destroyed, except when $f' \in \text{eff}^-(a) \cap \text{pre}(a)$. Recall that we assume, without loss of generality, that $\text{eff}^+(a) \cap \text{pre}(a) = \emptyset$.

More formally, let us consider two sets of operators, with regard to some fluent f : on the one hand, the operators that will *surely delete* f when applied, that we denote SD_f ; on the other hand, the operators that could *possibly add* f when applied, denoted PA_f . The latter are operators that may establish f in the resulting state s' depending on whether f is false in the previous state s or not. More formally, the sets are defined as follows:

- $SD_f = \{a \mid f \in \text{eff}^-(a) \cap \text{pre}(a)\}$
- $PA_f = \{a \mid f \in \text{eff}^+(a)\}$

This leads to the following inequality, which models the limit case previously presented. This effectively gives us an upper bound on the change of potential induced by a from any state s , which we denote $\Delta\Phi_a(s)$. Observe that the right-hand side is independent of s .

$$\Delta\Phi_a(s) \leq \sum_{f \in \text{eff}^+(a)} \Phi(f) - \sum_{f \in \text{eff}^-(a) \cap \text{pre}(a)} \Phi(f)$$

Now suppose that, for all operators a , the right-hand side of the previous inequality is negative. It means that applying any operator makes the potential of the state decrease. As a consequence, states that have a higher potential than the initial state cannot be reached. Note that, as the potential of a state is only determined by the potential of the fluents that are true in this state, and all potentials are positive, $\Phi(G^+)$ is a lower bound for the potential of any goal-state. Thus, if we also have that $\Phi(G^+) > \Phi(I)$, then the planning instance has no solution.

The only remaining issue is to check whether such a potential function Φ exists. As Φ is only determined by its values on the various fluents, this can be done with the following set of equations, with the set of variables $V = \{x_f \mid f \in F\}$. Intuitively, x_f corresponds to the potential $\Phi(f)$ of f .

Linear Program 1. Let $\Pi = \langle F, I, O, G \rangle$ be a planning instance. Define $\mathcal{L}_{\Pi}^{\text{pot}}$ to have the following variables and constraints:

Variables: $\{x_f \mid f \in F\}$

Constraints:

$$\sum_{f \in G^+} x_f - \sum_{f \in I} x_f > 0 \tag{1}$$

$$\sum_{f \in \text{eff}^+(a)} x_f - \sum_{f \in \text{eff}^-(a) \cap \text{pre}(a)} x_f \leq 0 \quad (a \in O) \tag{2}$$

$$x_f \geq 0 \quad (f \in F) \tag{3}$$

The following proposition follows from the discussion above.

Proposition 1. Let Π be a STRIPS instance. Suppose that there exists a solution for the Linear Program 1. Then Π has no solution.

Proof. Suppose that the hypothesis is true, and let x_f ($f \in F$) be a solution to Linear Program 1. As outlined in the discussion above, for each state s , we associate the potential:

$$\Phi(s) = \sum_{f \in s} x_f$$

A solution-plan must be valid for any initial state satisfying the initial condition and, in particular, for the initial state $s_0 = I$ in which all fluents not in I are false. Equation (1) tells us that $\Phi(s_0) > \Phi(s_k)$ at (any) goal state s_k . If an operator $a \in O$ transforms state s_{i-1} into state s_i , then the maximum of $\Delta\Phi_a(s_{i-1}) = \Phi(s_i) - \Phi(s_{i-1})$, i.e. the net gain in the score, is attained when each positive effect $f \in \text{eff}^+(a)$ is established (i.e. f was not already true in s_{i-1}) and only those negative effects $f \in \text{eff}^-(a) \cap \text{pre}(a)$ actually occur (i.e. negative effects $f \in \text{eff}^-(a) \setminus \text{pre}(a)$ were already false in s_{i-1}). Equations (2) and (3) tell us that $\Phi(s_i) \leq \Phi(s_{i-1})$ for any consecutive states in a solution-plan s_0, \dots, s_k . Clearly a goal state s_k can never be reached since potentials of all states reachable from s_0 remain less than or equal to $\Phi(s_0)$. \square

Note that the converse is not true: not all unsolvable planning instances are detected by the criterion we propose.

3.2. Dual linear program

The linear program presented in the previous section is hard to interpret, as the concept of potential we introduced has no reality outside of the criterion. However, we show in this section how to transform it into another program that can equivalently allow us to detect some unsolvable instances, but whose result is easier to interpret.

To this effect, we resort to Farkas's lemma. Farkas's lemma is related to the well-known fact that in linear programming, the primal problem is feasible iff the dual problem is feasible. One version of this lemma states that exactly one of the following sets of equations has a solution: either (1) $Ay \geq d$ where $y \geq 0$, or (2) $A^t x \leq 0$ and $d^t x > 0$ where $x \geq 0$, where A is a matrix and x, y and d vectors of the appropriate sizes. Let us consider the set of equations previously mentioned. Applying Farkas's lemma, it has a solution iff the following system has no solution:

Linear Program 2 (\mathcal{L}_{Π}^{op}).

Let $\Pi = \langle F, I, O, G \rangle$ be a planning instance. We define \mathcal{L}_{Π}^{op} as follows:

Variables: $V = \{y_a \mid a \in O\}$

Constraints C:

$$\sum_{a \in PA_f} y_a - \sum_{a \in SD_f} y_a \geq \delta_f^- \quad (f \in F) \quad (4)$$

$$y_a \geq 0 \quad (a \in O) \quad (5)$$

where $\delta_f^- = \mathbb{1}_G(f) - \mathbb{1}_I(f)$ ($\mathbb{1}_S(x)$ being the indicator function of set S : $\mathbb{1}_S(x) = 1$ if $x \in S$, and 0 otherwise). In this context, the variable y_a corresponds to the number of times operator a is executed in some sequence of actions. Note that y_a is positive, but not necessarily integral: this allows us to obtain a polynomial-time relaxation of the STRIPS instance. Inequality (4) states that the number of possible establishments of f minus the number of sure destructions of f must be greater than or equal to δ_f^- . For instance, any fluent that appears positively in the goal but not in the initial state must be established as least once. This dual version of our original linear program provides an alternative insight into the meaning of Proposition 1.

Lemma 1. Let $\Pi = \langle F, I, O, G \rangle$ be a planning instance, \mathcal{L}_{Π}^{op} as defined in Linear Program 2, and π a solution-plan for Π . Let us define $c_{\pi} : O \rightarrow \mathbb{N}$ to be the number of occurrences of operators of O in π . Then the assignment such that, for all $a \in O$, $y_a = c_{\pi}(a)$, is a solution for \mathcal{L}_{Π}^{op} .

Proof. Let y_a ($a \in O$) be as defined above. We will show that this is a solution for \mathcal{L}_{Π}^{op} . For each fluent f , let us denote e_f the number of times a fluent is established during the execution of π , and d_f the number of times it is destroyed. Recall that a fluent f is established (resp. deleted) by some occurrence of an operator $a \in O$ in π if f is false (resp. true) before the application of the operator, but true (resp. false) after. As π is a solution plan, we have that:

$$\mathbb{1}_{G^+}(f) - \mathbb{1}_I(f) \leq e_f - d_f$$

which can be shown by case disjunction on whether f is in I , G^+ , both or neither. We denote the inequalities above in a more concise way:

$$\delta_f^- \leq e_f - d_f$$

In addition, in the extreme case, f is established in π at most as many times as there are occurrences of operators a with $f \in \text{eff}^+(a)$. Hence,

$$e_f \leq \sum_{a \in PA_f} y_a$$

Similarly, the only operators $a \in O$ whose applications are guaranteed to destroy f are such that $f \in \text{pre}^+(a) \cap \text{eff}^-(a)$. Thus,

$$d_f \geq \sum_{a \in SD_f} y_a$$

By combining both inequalities above, we have

$$\begin{aligned} \delta_f^- &\leq e_f - d_f \\ &\leq \sum_{a \in PA_f} y_a - \sum_{a \in SD_f} y_a \end{aligned} \tag{6}$$

which means that y_a ($a \in O$) satisfies the constraints of the form of inequality (4) of \mathcal{L}_{Π}^{op} . As a consequence, as each y_a is also positive, this is a solution to \mathcal{L}_{Π}^{op} . \square

The contrapositive of Lemma 1 is an alternative proof that, if \mathcal{L}_{Π}^{op} has no solution, then neither has Π . But it allows us to show more than that, as we have the following corollaries, that we use later on:

Corollary 1. *If \mathcal{L}_{Π}^{op} has no integral solution, then the associated planning instance Π has no solution.*

Proof. The proof is immediate, as each operator appears an integral number of times in any solution-plan π . \square

Corollary 2. *Optimising the value of y_a within \mathcal{L}_{Π}^{op} leads to a bound on the number of times $a \in O$ must occur in a plan.*

Linear Program 2 is, in fact, a linear programming formulation of the state equation heuristic [2], as previously shown in Pommerening *et al.* [11]. Its efficiency for detecting unsolvable planning instances has been shown before, as it is part of the Aidos planner, which won the Unsat IPC in 2016 [12]. The planner uses the LP formulation of the operator counting heuristic to detect dead-ends during search, working on a finite domain representation (FDR) of the instance. We, however, do not resort to search, but show how to rewrite the model directly, and how to adapt the linear program accordingly.

Even though we introduced \mathcal{L}_{Π}^{op} as a linear program, we showed with Lemma 1 that one can also see it as an integer program. Solving an integer program is notoriously harder and slower than solving a linear program. As the integral solutions of the set of equations form a subset of its set of rational solutions, testing the solvability of the program over integral solutions is more likely to prove that the associated planning instance has no solution. Note that Farkas's lemma does not apply in the integral case, hence the need for Lemma 1.

In the next section, we show that, in the case where the criterion introduced here fails, it can still be leveraged to gather additional information about Π .

4. Enhancing the planning problem

This section is dedicated to extending and adding information to the initial planning instance, mainly with the goal of proving it unsolvable. Through various methods, we either add or remove elements from the input model Π , or add information about Π that is not directly encodable into the model, but that can nevertheless still be included in the linear program or to make deductions. In order to do so, we will resort to two kinds of methods. In the first ones, we build variations of Π so that, if one of these variations can be deemed unsolvable through the previous linear program, then some additional information about Π can be deduced. In the second method, we do not consider *per se* a variation Π' of Π , but we directly modify the linear program \mathcal{L}_{Π}^{op} associated to Π , so that if it is unsolvable, we can deduce new specific information about Π .

In the following, we call *operation* any such method. In the specific case where the operation answers a boolean question (e.g. Is an action removable?), we call it a *test*.

Note that ultimately, our goal is to detect unsolvable instances as such. As the set of solution-plans is empty, its elements and itself satisfy various otherwise uncommon properties. For instance, any operator can be removed without altering the set of solutions. This is why we search for properties on elements of Π that are unlikely to appear in solvable instances, but that are reasonable in our setting.

In the rest of this section, we illustrate the previous general principles through various operations, that allow us to find new information about the planning instance given as input. As our goal is to detect unsolvable instances, in the following, we assume that the criterion could not detect, at first, that the instance is unsolvable and that we have to gather additional information in order to do so.

4.1. Operator counts and landmarks

Landmark detection. An operator $a \in O$ is a landmark for Π if a occurs at least once in every solution-plan. We maintain throughout our procedure a set $L \subseteq O$ of landmarks. With regard to our framework, we can test if an operator is a landmark by removing it from the model and testing if the instance can be deemed unsolvable. More formally,

Lemma 2. *Let $\Pi = \langle F, I, O, G \rangle$ and $a \in O$. If $\Pi|_a = \langle F, I, O \setminus \{a\}, G \rangle$ is unsolvable, then a is a landmark.*

This allows us to define the landmark detection test below, where $\Pi|_a$ is as defined in Lemma 2.

LMDet

If the linear program $\mathcal{L}_{\Pi|_a}^{op}$ does not have a solution
then add a to the set of landmarks L

Operator count. One can generalize the notion of landmark, by counting the least number of times an operator appears in any solution-plan. This is why we maintain a function $n^- : O \rightarrow \mathbb{N}$, such that $n^-(a)$ is (a lower bound on) the least number of occurrences of action a in any plan. Likewise, we define $n^+(a)$ as (an upper bound on) the maximum number of times a appears in any plan. With this notation, $a \in O$ is a landmark iff $n^-(a) \geq 1$.

Reasoning on the number of occurrences of some operator $a \in O$ can be done through Linear Program 2. Indeed, as the variables correspond to the number of occurrences of each operator in some sequence of actions, one only has to find lower and upper bounds for each variable y_a in a solution of LP 2. This is why one can compute approximate values for $n^+(a)$ and $n^-(a)$ through an integral variation of our linear program, that we present below:

Integer Program 1 ($\mathcal{L}_{\Pi}^{opt}(a)$).

Let $\Pi = \langle F, I, O, G \rangle$ be a planning instance, with $O = \{a_1, \dots, a_m\}$, and $\mathcal{L}_{\Pi}^{op}(V, C)$ the associated Linear Program 2. For $a \in O$, define $\mathcal{L}_{\Pi}^{opt}(a)$ with the following variables and constraints:

Variables: $\{y_a \mid a \in O\}$

Constraints: Same as \mathcal{L}_{Π}^{op} except that the y_a are integral.

Objective function $g : \mathbb{N}^m \rightarrow \mathbb{N}$:

$$g : y_{a_1}, \dots, y_{a_m} \mapsto y_a$$

Lemma 3. Let Π be a planning instance, $a \in O$ an operator and consider the integer program $\mathcal{L}_{\Pi}^{opt}(a)$ with objective function g . Then minimizing (resp. maximizing) g yields a lower (resp. an upper) bound on the value of $n^-(a)$ (resp. $n^+(a)$).

Proof. The proof is a consequence of Lemma 1. We consider only the case where g is minimized, as the proof for the other case is mostly identical. We denote $n_{\mathcal{L}}^-$ the value obtained by minimizing g in $\mathcal{L}_{\Pi}^{opt}(a)$, where $a \in O$ is fixed. Suppose for a contradiction that $n^-(a) < n_{\mathcal{L}}^-$. Then there exists a plan π_a where a occurs exactly $n^-(a)$ times, by definition. By Lemma 1, there exists a solution Y_{π_a} for \mathcal{L}_{Π}^{op} where $Y_{\pi_a}(a) = n^-(a) < n_{\mathcal{L}}^-$, which contradicts the optimality of $n_{\mathcal{L}}^-$. Consequently, we have $n_{\mathcal{L}}^- \leq n^-(a)$. \square

This gives the following two tests.

OpCount⁺(a)

If the value $n_{\mathcal{L}}^+$ obtained by *maximizing* g in $\mathcal{L}_{\Pi}^{opt}(a)$ is bounded
 then set the current value of $n^+(a)$ to $n_{\mathcal{L}}^+$

OpCount⁻(a)

If the value $n_{\mathcal{L}}^-$ obtained by *minimizing* g in $\mathcal{L}_{\Pi}^{opt}(a)$ is non-zero
 then set the current value of $n^-(a)$ to $n_{\mathcal{L}}^-$

In the rest of this paper, we will often use the notation $\text{OpCount}(a)$ to refer to the successive application of $\text{OpCount}^-(a)$ and $\text{OpCount}^+(a)$.

Using operator counts. Once a non-trivial value for some $n^+(a)$ or some $n^-(a)$ has been found (i.e. a finite or non-zero value, respectively), one can reintroduce it into the linear program in the form of additional constraints. These constraints can be introduced in either \mathcal{L}_{Π}^{op} or \mathcal{L}_{Π}^{opt} , as both programs use the same sets of variables and constraints. As the variables of the linear programs correspond to the number of occurrences of operators in some plan, adding these constraints is straightforward for every $a \in O$:

$$y_a \leq n^+(a)$$

$$y_a \geq n^-(a)$$

4.2. Operator mutexes

Operator mutexes are unordered pairs of operators $\{a_1, a_2\}$ that can not both appear in the same solution-plan for Π . Such operators can still be part of some solution-plan, on their own. As an illustration, suppose that a_1 and a_2 both include fluent f in their respective positive preconditions and in their delete effects. If f does not belong to the add effects of any operator of Π , then a_1 and a_2 are operators mutexes, as they are in competition for the non-renewable resource f . We maintain, throughout the execution of our procedure, a set M_O of operator mutexes. One may think that operator mutexes do not occur often in planning models, as it is a property that concerns the set of solution-plans as a whole. However, our aim is to detect unsolvable planning instances, which have by definition no solution-plan: all pairs of operators are thus operator mutexes.

Finding operator mutexes through linear programming. In order to check if two operators $a_1, a_2 \in O$ are operator mutexes, it suffices to build the linear program $\mathcal{L}_{\Pi}^{opm}(a_1, a_2)$ and check if it is feasible or not. In essence, $\mathcal{L}_{\Pi}^{opm}(a_1, a_2)$ consists of the same set of constraints as \mathcal{L}_{Π}^{op} , except that it has the following additional constraints:

$$y_{a_1} \geq 1 \quad (7)$$

$$y_{a_2} \geq 1 \quad (8)$$

If $\mathcal{L}_{\Pi}^{opm}(a_1, a_2)$ does not admit a solution, then it means that both operators can not appear simultaneously in a plan, and thus are operator mutexes. Note that adding only one of the two inequalities above could also make the linear system unsolvable. In that case, this would mean that either a_1 or a_2 does not appear in any solution-plan. This is something that we test before resorting to operator mutex tests. More generally, we check if some operators can be removed from the planning model. We present various techniques for doing so in the next section. Below, we introduce the notation for the test corresponding to the linear program we presented above:

OpMut (a_1, a_2)

If the linear program $\mathcal{L}_{\Pi}^{opm}(a_1, a_2)$ does not admit a solution
 then add $\{a_1, a_2\}$ to the set of operator mutexes M_O

Using operator mutexes. Including the information we have about operator mutexes into the linear program is straightforward. Knowing that $\{a_1, a_2\}$ is an operator mutex translates into barring certain solutions of \mathcal{L}_{Π}^{op} from appearing. Concretely, we wish to ensure that, when $y_{a_1} \geq 1$, then $y_{a_2} = 0$, and conversely.

In order to achieve so, we introduce *binary* variables of the form $u_a \in \{0, 1\}$, for each operator $a \in O$. We wish to enforce that $u_a = 1$ whenever $y_a \geq 1$. This can be done through the following set of constraints, which also bars operator mutexes from appearing.

$$y_a - n^+(a) \cdot u_a \leq 0 \quad (a \in O) \quad (9)$$

$$u_{a_1} + u_{a_2} \leq 1 \quad (\{a_1, a_2\} \in M_O) \quad (10)$$

4.3. Detection of removable actions

This section is concerned with finding operators $a \in O$ that never appear in any solution-plan. Even though some such operators can be detected statically by the parser that we use, some others require additional computation. We present various techniques that allow us to detect if an operator can be immediately removed from the planning instance, without altering its set of solutions.

Through a modification of the linear program. We start by extending \mathcal{L}_{Π}^{op} into $\mathcal{L}_{\Pi}^{ro}(a)$ through the addition of the constraint $y_a \geq 1$. If $\mathcal{L}_{\Pi}^{ro}(a)$ has no solution, then Π has no solution where a occurs at least once, and a can thus be removed from the model.

We do not elaborate on this argument further, as it is a special case of the technique seen in Section 4.1. Indeed, it is equivalent to show that $n^+(a) = 0$, as it ensures that a does not occur in any solution-plan. However, this argument allows us to find removable operators that are not detected by a test proposed later in this subsection.

Unreachable preconditions. A simple way to prove that some operator a will never be part of any plan, is to prove that no reachable state satisfies its precondition. This can be done by testing that the planning instance $\Pi_a^{\text{pre}} = \langle F, I, O, \text{pre}(a) \rangle$ is unsolvable.

Removing some operators relaxes the linear program $\mathcal{L}_{\Pi}^{\text{op}}$, by the deletion of some of the associated variables and constraints. As a consequence, it can help prove some instances unsolvable. We introduce below the notation for the associated test:

Prelmp

If the linear program $\mathcal{L}_{\Pi_a^{\text{pre}}}^{\text{op}}$ does not have a solution
 then remove a from the set of operators O

Dead-end operators. As it is possible to test whether or not there exists a reachable state where a can be applied, it is natural to ask the opposite: does a always lead to a dead-end, where no goal state can be reached?

This paragraph is dedicated to finding such operators, called *dead-end operators*. In order to do so, we need to restrict ourselves to the few fluents that appear in all states resulting from the application of a , that is to say, the fluents that are true after a is applied either because of the effects of a , or by inertia. Indeed, these fluents are the only ones for which we have enough information about their truth value to reason about. Let $F_a = \text{pre}(a) \cup \text{eff}^+(a) \cup \text{eff}^-(a)$. For any set S of literals of F , and $E \subseteq F$, we note $S|_E$ the projection of S over the fluents E . Likewise, we denote $a|_E = \langle \text{pre}(a)|_E, \text{eff}(a)|_E \rangle$ the projection of operator a over E . For any $O' \subseteq O$, we also note $O'|_E = \{a|_E \mid a \in O'\}$. This leads us to the following lemma:

Lemma 4. Let $\Pi = \langle F, I, O, G \rangle$ be a planning instance and $\Pi^{\text{post}} = \langle F_a, I_a^{\text{post}}, O|_{F_a}, G|_{F_a} \rangle$, where $I_a^{\text{post}} = ((\text{pre}(a) \setminus \text{eff}^-(a)) \cup \text{eff}^+(a)) \cap F_a$. If Π_a^{post} is unsolvable, then a is a dead-end operator in Π .

Proof. Suppose by contradiction that a is not a dead-end operator in Π , but that Π_a^{post} is unsolvable. Then there exists a solution-plan for Π where a occurs at least once. Let us note $\pi = a_1 \cdots a_k$ this plan, and let $i \in \{1, \dots, k\}$ be the greatest index such that $a_i = a$. In the case where $i = k$, we immediately have $I_a^{\text{post}} \subseteq G|_{F_a}$, which means that Π_a^{post} is solvable and thus results in a contradiction.

Let us now suppose that $i \neq k$. Then $\pi^{\text{post}} = a_{i+1}|_{F_a} \cdots a_k|_{F_a}$ is a solution-plan for Π^{post} , as proven below. Indeed, let $s_0, s_1 \cdots s_k$ be the set of states associated to π , and let us show by recurrence that the sequence of states associated to the plan π^{post} in Π_a^{post} is $s_i|_{F_a} \cdots s_k|_{F_a}$. First, we prove that $I_a^{\text{post}} = s_i \cap F_a$. As s_i results from the application of $a_i = a$, we have that $I_a^{\text{post}} \subseteq s_i$, and thus $I_a^{\text{post}} \subseteq s_i \cap F_a$. Conversely, let us show that $s_i \cap F_a \subseteq I_a^{\text{post}}$. Suppose that $f \in s_i \cap F_a$, and let us proceed by case disjunction on $f \in F_a = \text{pre}(a) \cup \text{eff}^+(a) \cup \text{eff}^-(a)$. If $f \in \text{eff}^+(a)$, then $f \in I_a^{\text{post}}$. Necessarily, we have that $f \notin \text{eff}^-(a)$, because otherwise $f \notin s_i$ as s_i results from the application of a . Then, we otherwise have $f \in \text{pre}(a) \setminus \text{eff}^-(a)$, and thus, we have $f \in I_a^{\text{post}}$. Hence, $I_a^{\text{post}} = s_i \cap F_a$.

We have shown that the initial state of the sequence of states associated to π^{post} is $s_i|_{F_a}$. To show the property for the remaining states, it suffices to check that, for any two states s, s' of Π , and for any operator $a' \in O$, if we have $s' = s[a']$, then $s'|_{F_a} = s|_{F_a}[a'|_{F_a}]$. As a consequence, we immediately have that $s_k|_{F_a} \subseteq G|_{F_a}$, and π^{post} is a solution-plan for Π_a^{post} , which is a contradiction. Thus, no solution-plan π where a occurs exists, and a is a dead-end operator. \square

OpDLock

If the linear program $\mathcal{L}_{\Pi_a^{\text{post}}}^{\text{op}}$ is unsolvable
 then remove a from the set of operators O

Experimental trials showed that no operator could be proved to be a dead-end operator. This is why, in Section 5, no results are reported about the above test. However, we still included this test to show that some very small problems derived from the input instance can be of interest.

4.4. Extended goals

In this section, we propose various methods to find more precise goal states. More particularly, we try to add new literals to the goal, be they positive or negative. Suppose, for instance, that some fluent $f \in G^+$ can only be true if some other fluent f' is true. Then one can immediately add f' to G^+ . These more precise goals make the program richer and hence more likely to detect unsolvable instances. G^- can also be extended in a similar way.

This can be done in our framework through the following simple observation. Let $f \in F$, and $\Pi_{+f}^G = \langle F, I, O, G \cup \{f\} \rangle$. If Π_{+f}^G is unsolvable, then f can be added to the negative goals of Π . Indeed, no goal state s_G such that $s_G \models f$ is reachable: necessarily, in any goal state s_G , we have $s_G \models \neg f$. Conversely, let $\Pi_{-f}^G = \langle F, I, O, G \cup \{\neg f\} \rangle$. If Π_{-f}^G is unsolvable, then f can be safely added to the goals of Π without changing the set of solutions.

FNegGoal

If the linear program $\mathcal{L}_{\Pi_{+f}^G}^{op}$ is unsolvable
then add f to the negative goals of Π

There is, of course, a symmetrically equivalent test **FPosGoal** that we could have defined. However, in order to detect positive goals, we would need to test Π_{-f}^G , which has negative goals. These negative goals are not used in the formulation of Linear Program 2: as a consequence, our criterion would be powerless at detecting that Π_{-f}^G is unsolvable. In addition, we tested **FNegGoal** in order to assess the ability of our linear programming criterion to detect the consumption of resources.

4.5. Fluent mutexes and unreachable fluents

A fluent mutex is a set of fluents $M \subseteq F$ for which all states s reachable from the initial state I are such that $s \not\models M$. Some tests presented previously can be seen as testing whether some subset $M \subseteq F$ is a fluent mutex. Let us consider for instance the **PreImp** test presented in Section 4.3: for some operator $a \in O$, checking that $\Pi_a^{\text{pre}} = \langle F, I, O, \text{pre}(a) \rangle$ is unsolvable (and thus that operator a can be removed from the instance) is equivalent to checking that $\text{pre}(a)$ is a mutex. However, our criterion allows us to check if any set of fluents $F' \subseteq F$ is a mutex, by testing the unsolvability of $\Pi_{F'}^{\text{mut}} = \langle F, I, O, F' \rangle$.

FMut(F')

If the linear program $\mathcal{L}_{\Pi_{F'}^{\text{mut}}}^{op}$ is unsolvable
then F' is a fluent mutex

The criterion does not detect all fluent mutexes, and each candidate set of fluents has to be tested individually. Thus, not all fluent mutexes can be detected in reasonable time, as there exists an exponential number of candidates. Finding which sets are interesting to test is a problem in itself; even more so since one has to know how to make use of the newly-found information that some $M \subseteq F$ is a mutex.

In the general case, we could not find a way to reinvest into the linear program the knowledge that a set of fluents is a mutex. Indeed, Linear Program 2 reasons over the number of times operators (have to) occur in a plan. As a consequence, we do not have any obvious way to reason about properties concerning states, which is precisely what fluent mutexes are. For that reason, we do not include in our routine a computation of mutexes through our linear program, even though we can detect a range of fluent mutexes.

However, some fluents are always false, in the sense that no plan will ever establish them. We call the fluents *unreachable fluents*, and they can be detected with the same argument as above:

FRreach(f)

If the linear program $\mathcal{L}_{\Pi_{\{f\}}^{\text{mut}}}^{op}$ is unsolvable
then f is an impossible fluent

Even though these fluents appear very rarely, as will be shown in the experimental trials, it remains linear to test for all fluents whether they are unreachable or not: thus, the computational burden is significantly lower than for other fluent “mutexes”. When an unreachable fluent is detected, one can project the whole instance on fluents $F \setminus \{f\}$. Theoretically, one could also remove operators that have f in their positive preconditions: however, any such operator a would also be detected by test **PreImp(a)**, which is more likely to succeed.

Set	Unsat	Total
bag-transport	19	29
bottleneck	25	25
cave-diving	1	25
chessboard-pebbling	23	23
over-tpp	2	30
pegsol-row5	14	15
tetris	20	20
<i>Remaining</i>	0	180
Total	104	347

Table 1

Summary of the results returned by the LP-based criterion, run on the Unsat planning competition benchmark set. Each line corresponds to a domain, which is a set of instances modelling similar problems. The first column reports instances on which our criterion succeeds, while the second column reports the total number of instances in the benchmark set. Domains for which no instance could be solved are summed up in the last line labeled *Remaining*.

5. Experimental evaluation

Our implementation was done in Python 3.10, basing ourselves on the Fast Downward parser [7]. For linear programs, we resorted to the GLOP solver [10], while integer programs were solved with Gurobi [6]. We also used Google ORTools [10] to interface between our program and the solvers. We ran our experiments on a machine running Rocky Linux 8.5, powered by an Intel Xeon E5-2667 v3 processor, with a 30-minutes cutoff and using at most 16GB of memory per instance. Our code is available online¹.

In addition to the evaluation of the linear program, we also implemented a procedure based on the observations of Section 4. The main loop of this procedure consists in executing sequentially a predetermined list of operations and tests, until the instance is detected as unsolvable or the list is depleted. We elaborate further on this in Section 5.2.

We wished to evaluate our program on two different aspects: first, its ability to detect unsolvable instances, and second, its ability to find additional information when it could not conclude.

Our set of benchmarks consists of the unsolvable instances from the unplannability track of the International Planning Competition 2016 (Unsat IPC), for which we report our results on unsolvable instances. The Unsat IPC also includes solvable instances, which we tested our program on, as a sanity check, with success.

5.1. LP-based criteria

In this section, we show that our LP-based criterion suffices to detect a wide range of unsolvable planning instances. Our results are reported in details in Table 1.

In essence, about 30% of all instances of the Unsat IPC are almost immediately found to be unsolvable by the sole use of the criterion. These results however vary greatly from one domain to the other, in a very dichotomous fashion: either the domain is (almost) entirely solved through the criterion, either few to no instances can be deemed unsolvable. In the case of domain bag-transport, which seems to be in-between, all instances the criterion has been tried on are actually found to be unsolvable: however, as the last 10 instances are too big to be parsed, we could not run the test on them. We can also note that both linear- and integer-programming-based criteria yield the same results, and that solving the IP-based program did not allow us to detect more unsolvable instances than through the linear programming formulation.

Both programs are however very lightweight: for most domains, building and solving the program required less than a few seconds. In addition, for the vast majority of instances, the criteria required little more than a few tenths of a second to complete. This further justifies our use of the program in the iterative procedure that we present in the next section.

¹<https://github.com/arnaudlequen/MPRefinement>

Our program fails entirely on some domains, where no instance can be solved. While this is often because our criterion simply fails to detect the instance’s unsolvability, this can also be due to the size of the model. This is the case of bag-gripper, where the first instance has 5681 fluents and 60604 operators, which prevents us from building the associated linear program. In our assessments of the performances of the criteria, the limitation always came from memory.

5.2. Iterative refinement of the model

In the case where the criterion could not immediately detect that an instance Π is unsolvable, one can resort to the several operations previously introduced. In addition, the order in which operations are executed is also critical. Consider for instance an operator a that is both recognized as a landmark and as a removable operator by our operations. In the case where the operator is first removed, then it can not be detected as a landmark, and we thus missed an opportunity to return that the instance is unsolvable. In the case where a is first detected as a landmark, then our routine terminates successfully by detecting that the instance is unsolvable.

5.2.1. Sequences of operations

We present below the different lists of operations that we chose. Note that all sequences start and end with a simple test of solvability with the criterion: initially with only the information contained in the STRIPS model, and then with the information that was incrementally gathered after each series of operations. The exact sequences of operations can be found in Appendix A.

Linear. This sequence comprises all tests and operations that are linear in the size of the instance, i.e. that only require one argument. We tried to put first the tests that were the most likely to succeed, so that the followings tests and operations that come after have more information to work with. We successively apply the following tests on all relevant elements, in that order: LMDet, PreImp, OpCount, FReach, and FNegGoal. By that, we mean that we run LMDet(a) for all $a \in O$, then PreImp(a) for all $a \in O$, etc.

Quadratic. In addition to the tests found in the Linear sequence, the Quadratic sequence has a sequence of operator mutexes tests OpMut, which comes right after the Linear sequence’s operations. There exists a quadratic (in the size of the instance) number of such tests, as one has to consider every pair of operators successively. Even though the Linear sequence is itself costly in terms of computation, it still builds fewer linear programs than the operator mutexes tests alone.

OperatorPreImpossible. As will be reported later, the PreImp tests that check an operator’s reachability are our most successful ones. We wished to gauge the time it requires and its possible impact on the model by itself.

OperatorCount. This sequence consists in finding lower on the number of times each operator has to appear in any plan, then upper bounds on the number of times each operator can appear in any plan. It aims to show that a linear number of integer programs to optimize can be done in reasonable time, while also providing interesting information.

5.2.2. Results

We present our results below. As we prune out instances that can be immediately identified as unsolvable, domains that are immediately found unsolvable by the criterion are not reported.

Linear sequence. Table 2 shows statistics for the Linear sequence. The main goal of our routine was to extract additional information from the model, either to directly prove the instance unsolvable, or so that another procedure that comes after can show the instance unsolvable more easily. We could indeed notice that our algorithm was sometimes enough to detect unsolvable instances that are otherwise not detected as such by the criterion. There are few examples of such instances (about 9.5% of the entire benchmark set), and they are grouped in only two domains (cave-diving and pegsol). Nonetheless, they suffice to show that a well-chosen sequence of operations can sometimes replace a search, and that our work paves the way for further research in that regard.

In the cases where our procedure could not conclude, it still manages to gather valuable information about the planning instance. For example, on some domains, almost a third of all operators are pruned on average, among instances on which our procedure terminates.

Set	Diff.	Operators			Others		
		PreImp	OpCount	Rem.	LMDet	FReach	FNegGoal
cave-diving (14)	+11	10.0%	14.1%	10.4%	1.1%	4.8%	3.0%
diagnosis (19)	0	0%	57.0%	11.6%	18.3%	4.6%	17.6%
doc-transfer (5)	0	13.0%	26.4%	27.9%	1.7%	0.0%	39.8%
over-nomystery (2)	0	33.4%	25.7%	34.8%	2.1%	0%	7.4%
over-rovers (8)	0	27.9%	17.2%	29.3%	0%	<0.1%	0%
over-tpp (8)	0	7.4%	54.8%	24.7%	0.3%	0.3%	0%
pegsol (24)	+24	13.6%	N/A	13.6%	0.8%	N/A	N/A
sliding-tiles (20)	0	0%	0%	0%	0%	0%	69.2%

Table 2

Statistics for the Linear sequence. The first column with the name of the domain also reports the total number of instances for which the procedure terminated entirely within the time and memory limits. The “Difference” (Diff.) column shows the number of instances that could be found unsolvable during the execution of the procedure, compared to the single use of the criterion reported in Table 1. The next set of columns shows stats for operations related to the deletion of operators. The first pair of columns show the percentage of success of each test (OpCount comprising both OpCount⁺ and OpCount⁻), while the last column of the set shows the average total percentage of operators removed at the end of the sequence of tests. The last three columns show the percentage of success of three other tests. N/A values indicate that no such test was performed as the program terminated before.

The termination of our procedure is, however, the main issue of this sequence of operations, which is too computationally costly, and often stops early because of the time and memory limits imposed. In some domains, very few instances could be run through the entire sequence of operations: such domains include over-nomystery, where this sequence terminated on only 2 instances out of the 24 that could be parsed.

Quadratic sequence. Table 3 shows statistics for the Quadratic sequence. As the first part of the Quadratic sequence consists in the exact same sequence of operations as the Linear sequence, there are at least as many instances solved by the Quadratic sequence as by the Linear sequence. This is why domains cave-diving and pegsol only exhibit instances that are entirely solved: our procedure detects these instances as unsolvable before reaching the OpMut tests (and then running out of time, in the case of cave-diving).

In general, the use of a quadratic number of operations quickly becomes problematic: compared to the Linear sequence, the Quadratic sequence can be run in its entirety on significantly fewer instances. In addition, very few tests are actually successful, as even with the most receptive instances, only a few percents of all tests succeed. However, the main issue with our sequence of operations is that it considers all pairs of operators, and checks if they can be marked as mutexes. The quadratic number of such tests is then responsible for the computational cost of the sequence, but the test in itself remains an integer program that can be solved almost instantly.

Yet, even if we were to find operators mutexes, they would be hard to reinvest in the linear program. Indeed, to be included in the linear program, an operator mutex $\{a_1, a_2\}$ requires that $n^+(a_1)$ and $n^+(a_2)$ are finite (see Equation (9)), which is common but not always the case, as reported in Table 5. As very few pairs of operators are detected as mutexes, the next logical step to make the most of operator mutexes is to more efficiently choose which pairs to test, to avoid performing unpromising tests.

Individual tests. Tables 4 and 5 summarize the statistics for the other sequences, that mostly consists of series of one or two of the same operations. However, we do not report comprehensive results for all remaining sequences: indeed, in the case of the OperatorDeadLock sequence, no test answered positively, and no dead-end operator could be found.

Nonetheless, the results for the other sequences of operations are encouraging. Be it for the sequence centered on PreImp or the one focused on OpCount operations, a significant proportion of operators could be removed. In some cases, it suffices to show that the instance was not solvable, as is the case for the cave-diving or pegsol domains. However, the time required for the computation is significant, which is discussed in the next section.

An interesting property of our OpCount tests is that they do not always return 0, 1 or $+\infty$: they sometimes find that some operator a can be applied at most c times, where c is an integer such that $c \geq 2$.

Set	Diff.	Operator mutexes		Av. $ O $
		Av. mut/op	Success %	
cave-diving (12)	+12	N/A	N/A	1113
diagnosis (15)	0	4.8	3.6%	136
over-rovers (3)	0	8.2	6.2%	313
over-tpp (3)	0	0.8	<0.1%	1675
pegsol (24)	+24	N/A	N/A	77
sliding-tiles (10)	0	0	0%	193

Table 3

Statistics for the Quadratic sequence. As in Table 2, the Diff. column reports the instances found unsolvable by our routine, but on which the criterion failed to conclude by itself. The second column reports the average number of mutexes an operator finds itself in, and the third column reports the average percentage of success of OpMut tests. This latter column could also be interpreted as the average proportion of operators that are found to be mutex with another fixed operator. The last column reports the average number of operators per instance. Note that our reports concern far fewer instances than for the Linear sequence, as the computational cost is significantly greater for the sequence reported here. Domains document-transfer and over-nomystery are not reported as no instance finished before the cutoff.

Set	Diff.	OperatorPreImpossible	
		Removed	Time (s)
bag-barman (4)	0	77.2%	1177.8
cave-diving (17)	+4	6.5%	147.8
diagnosis (20)	0	0%	6.4
document-transfer (13)	0	0%	475.7
over-nomystery (10)	0	18.8%	587.8
over-rovers (11)	0	21.9%	370.2
over-tpp (14)	0	<0.1%	268.1
pegsol (24)	+6	16.4%	0.6
sliding-tiles (20)	0	0%	5.6

Table 4

Performances of the OperatorPreImpossible sequence. The number in parenthesis in each row indicates the instances the sequence terminated on. The Diff. column shows the number of instances solved thanks to the iterative refinement, the Rem. column shows the average percentage of operators that could be removed thanks to the operation, and the Time column shows, in seconds, the average time per instance.

Note that these sequences of tests are not as powerful as the Linear sequence, when it comes to detecting unsolvable instances. This seems to indicate that the combination of different kinds of operations is crucial to draw conclusions, and studying their interactions is crucial in designing more powerful sequences.

6. Related work

The surge in interest for unsolvability detection, in the last decade, has been embodied by the first Unsolvability Planning Competition in 2016. The competition saw various adaptations of techniques that have shown themselves efficient for finding plans, in a state-space search. Such methods include heuristics specifically tailored for unsolvability detection, such as a Merge & Shrink-based heuristic [8] (which precedes the competition). Such heuristics rely on abstractions that do not preserve distance, but merely solvability.

Another heuristic that was successfully adapted was the operator-counting heuristic [2, 11, 17]. The heuristic is based on a relaxation of the orderings of the operators. Previous works showed that it admits a linear programming formulation, similar to the Linear Program 1 that we propose. However, while we only optimize the variable associated to the count of a single operator, the objective function that they minimize is the total cost of the plan. The adaptation of the linear program to the case of unsolvability detection, was carried out by the Fast Downward-based

Set	Diff.	OperatorCount			
		OpCount ⁻	OpCount ⁺	Removed	Time (s)
bag-barman (0)	-	-	-	-	-
cave-diving (17)	0	0.9%	28.0%	7.0%	329.3
diagnosis (20)	0	16.9%	96.3%	19.5%	91.6
document-transfer (8)	0	1.7%	50.7%	29.8%	643.2
over-nomystery (3)	0	1.4%	87.2%	3.9%	746.2
over-rovers (9)	0	0%	62.4%	5.2%	455.1
over-tpv (9)	0	0.3%	65.4%	20.2%	428.8
pegsol (24)	+22	0%	8.2%	3.0%	0.51
sliding-tiles (20)	0	0%	0%	0%	19.4

Table 5

Performances of the OperatorCount sequence. The number in parenthesis in each row indicates the instances the sequence terminated on, and the *Diff.* column shows the number of instances solved thanks to the iterative refinement. OpCount⁻ and OpCount⁺ columns report the average percentage of success of their respective operations. The *Removed* column shows the average percentage of operators that could be removed thanks to the OpCount⁺ operations. The *Time* column shows, in seconds, the average time per instance.

unsolvability planner Aidos [12]. It consists in checking the existence of a solution, in the same way as for Linear Program 2. However, Aidos uses this component in a state-space search, in order to detect dead-ends.

More generally, be it in unsolvable or in solvable planning tasks, the early detection of states that can not lead to a goal makes can help prune out whole branches of the search space. In the case of dead-end detection [4], various works have focused on the elaboration of formulas that can be efficiently evaluated, and whose only models are states that can not lead to a goal state. The notion of dead-end formula has been generalized with the notion of traps [9]: a formula ϕ such that, once it's verified in a state s , all states reachable from s will satisfy it too. A formula ϕ that is inconsistent with the goal then shows that the current branch is not worth exploring.

In the case where our algorithm does not manage to find that the task is unsolvable, it still manages to remove unnecessary elements from the planning model, to make the task easier for the next algorithm. Various other methods prune the model in a preprocessing step: in [1], the authors show that invariants in the form of mutexes can be leveraged to remove operators that will never be part of a plan. In [5], it is shown how to combine symmetries of the planning task and operator mutexes to find operators that are redundant, in the sense that removing them preserves at least one solution-plan.

Our algorithm also learns information that is not explicitly expressible in a STRIPS planning instance. In [15], the authors draw inspiration from a well-known technique in SAT solving, to learn clauses that recognize dead-ends, through a conflict-driven approach during search. They also show how to learn traps online [14]. Learning is ubiquitous in generalized planning, which is a domain concerned with the synthesis of generalized plans, which are procedures that solve multiple instances. For instance, previous work [16] proposed to learn heuristics in the form of logical formulas, out of a set of small examples instances, so as to recognize unsolvable planning instances.

In [3], another polynomial criterion is proposed to immediately detect a class of unsolvable instances without resorting to search. The authors synthesize a function that separates the initial state from all goal states, through a linear combination of features valued in a finite field. Akin to our criterion, their technique is incomplete, but it is very efficient at detecting parity arguments.

7. Discussion

7.1. Additional operations

We designed more operations than presented in this paper, but we only report those for which a non-trivial amount of tests answered positively. Operations that never succeeded include operator ordering tests: given a_1, a_2 two operators, does a_1 always appear before a_2 in all solution-plans? This is the case, for instance, if a_1 has some initial

fluent f in its precondition, a_2 destroys f , and no other operator can establish f . In order to use our framework to test that a_1 cannot occur after a_2 , one can check that the preconditions of a_1 cannot be reached once a_2 has been applied. But to do so, the instance has to be projected on the only fluents whose value is known after the application of a_2 . As a result, the linear program corresponding to the newly-formed instance is loosely constrained, and in our experiments did not allow us to conclude.

Other such tests include checking if some fluent f can be added to the (negative) precondition of operator a . This can be done by testing if $\text{pre}(a) \cup \{f\}$ is reachable from I . If not, then $\neg f$ can be added to the preconditions of a . Such tests sometimes succeed, but the proportion of tests that do is often negligible compared to the cost of testing each pair of $F \times O$.

7.2. Perspectives

Section 5 showed that, when our criterion failed to show an instance unsolvable, it was still possible to extract additional information from the model by leveraging the criterion. Even more so, in some cases, otherwise undetected unsolvable instances could be identified as such by this means. Yet, there is still a lot of room for improvement: a more in-depth study of our operations, as well as their interactions, could help us fine-tune the algorithm, and tailor more effective sequences of operations. Indeed, not all sequences of tests are equal in all aspects, and finding a sequence that avoid unnecessary computations is a way to optimize our algorithm, and to boost its detection power.

In our tests, we choose to simply run pre-determined sequences of operations and tests. This means that, regardless of how tests succeed or fail, the algorithm will linearly go through the same sequence of operations, except if it can show preemptively that an instance is unsolvable. However, the outcome of some test may help in finding which step to take next. For instance, after finding that an operator is a landmark, it might be interesting to check right away if it can be removed. This can be done through a stack of operations, on which are added the operations that are made relevant by the result of another previous operation.

One of the main weaknesses of our iterative refinement algorithm is its computational cost. Even the most lightweight sequences, such as the OperatorPreImpossible sequence, takes significant time to complete. Our program builds each linear program from scratch each time a test is performed. However, very few constraints differ from one linear program to the other; thus, one could modify only these constraints from one test to the next, in order to save significant time. In addition to that, the operations are mostly independent from one another: as a consequence, one could perform multiple operations in parallel with minimum loss.

8. Conclusion

In this article, we showed that a simple criterion was sometimes enough to prove that a planning instance is unsolvable. Even though our program is non-optimised, we have still managed to show that resorting to a search is not always necessary, as reasoning on the model directly can suffice. Even when our procedure fails, it still gathers valuable information about the instance, that can help a complete procedure terminate faster.

Other operations and tests can be thought of, and included in our framework. The most important point would be to ensure that the information that they bring is related to the other operations (e.g. checking if an operator a 's preconditions are removable, and checking if $n^+(a) \leq 0$), or at least that the new information can be reinvested in the linear program.

Acknowledgements

The authors would like to thank the reviewers of *17èmes Journées d'Intelligence Artificielle Fondamentale* for their insightful comments.

This work was supported by the AI Interdisciplinary Institute ANITI, funded by the French program "Investing for the Future - PIA3" under grant agreement no. ANR-19-PI3A-0004.

Appendix A. Sequences of operations and tests

Below are shown the exact sequences of operations and tests of each sequence tested on the benchmark sets. Tests that require an argument are run on all possible arguments (ex. PreImp tests are run on all operators) and those that require two arguments are run on all possible pairs of such arguments (ex. OpMut tests are run on all pairs of operators).

Quadratic.

- Criterion
- LMDet
- PreImp
- OpCount⁻
- OpCount⁺
- FReach
- FNegGoal
- OpMut
- Criterion

Linear.

- Criterion
- LMDet
- PreImp
- OpCount⁻
- OpCount⁺
- FReach
- FNegGoal
- Criterion

OperatorCount.

- Criterion
- OpCount⁻
- OpCount⁺
- Criterion

OperatorPreImpossible.

- Criterion
- PreImp
- Criterion

References

- [1] Vidal Alcázar and Alvaro Torralba. A reminder about the importance of computing and exploiting invariants in planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 25, pages 2–6, 2015.
- [2] Blai Bonet. An admissible heuristic for SAS+ planning obtained from the state equation. In Francesca Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence*, pages 2268–2274, 2013.
- [3] 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, *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling, ICAPS 2022*, pages 44–52, 2022.
- [4] 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, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18)*, pages 1306–1313, 2018.
- [5] Daniel Fišer, Alvaro Torralba, and Alexander Shleyfman. Operator mutexes and symmetries for simplifying planning tasks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 7586–7593, 2019.
- [6] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023.
- [7] Malte Helmert. The fast downward planning system. *JAIR*, 26:191–246, 2006.
- [8] Malte Helmert, Patrik Haslum, Jörg Hoffmann, and Raz Nissim. Merge-and-shrink abstraction: A method for generating lower bounds in factored state spaces. *J. ACM*, 61(3):16:1–16:63, 2014.
- [9] 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, *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling, ICAPS*, pages 211–215, 2016.
- [10] Laurent Perron and Vincent Furnon. Or-tools.
- [11] Florian Pommerening, Gabriele Röger, Malte Helmert, and Blai Bonet. Lp-based heuristics for cost-optimal planning. In Steve A. Chien, Minh Binh Do, Alan Fern, and Wheeler Ruml, editors, *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS*, 2014.
- [12] Jendrik Seipp, Florian Pommerening, Silvan Sievers, Martin Wehrle, Chris Fawcett, and Yusra Alkhazraji. Fast Downward Aidos. *Unsolvability International Planning Competition: planner abstracts*, pages 28–38, 2016.
- [13] Marcel Steinmetz, Jörg Hoffmann, Alisa Kovtunova, and Stefan Borgwardt. Classical planning with avoid conditions. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022*, pages 9944–9952, 2022.
- [14] Marcel Steinmetz and Jörg Hoffmann. Search and learn: On dead-end detectors, the traps they set, and trap learning. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 4398–4404, 2017.
- [15] 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.
- [16] Simon Ståhlberg, Guillem Francès, and Jendrik Seipp. Learning generalized unsolvability heuristics for classical planning. In Zhi-Hua Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, pages 4175–4181, 2021.
- [17] Menkes Van Den Briel, J Benton, Subbarao Kambhampati, and Thomas Vossen. An LP-based heuristic for optimal planning. In *Principles and Practice of Constraint Programming—CP 2007: 13th International Conference*, pages 651–665. Springer, 2007.