Document downloaded from:

http://hdl.handle.net/10251/179924

This paper must be cited as:

Segovia-Aguas, J.; Jiménez-Celorrio, S.; Jonsson, A. (2020). Generalized Planning with Positive and Negative Examples. 9949-9956. https://doi.org/10.1609/aaai.v34i06.6550

The final publication is available at

https://doi.org/10.1609/aaai.v34i06.6550

Copyright

Additional Information

Generalized Planning with Positive and Negative Examples

Javier Segovia-Aguas,[1] **Sergio Jiménez,**[2] **Anders Jonsson**[3]
[1]IRI - Institut de Robòtica i Informàtica Industrial, CSIC-UPC
[2]VRAIN - Valencian Research Institute for Artificial Intelligence, Universitat Politècnica de València
[3]Universitat Pompeu Fabra

Abstract

Generalized planning aims at computing an algorithm-like structure (*generalized plan*) that solves a set of multiple planning instances. In this paper we define *negative examples* for generalized planning as planning instances that must not be solved by a generalized plan. With this regard the paper extends the notion of *validation* of a generalized plan as the problem of verifying that a given generalized plan solves the set of input *positives* instances while it fails to solve a given input set of *negative* examples. This notion of plan validation allows us to define quantitative metrics to asses the generalization capacity of generalized plans. The paper also shows how to incorporate this new notion of plan validation into a compilation for plan synthesis that takes both positive and negative instances as input. Experiments show that incorporating negative examples can accelerate plan synthesis in several domains and leverage quantitative metrics to evaluate the generalization capacity of the synthesized plans.

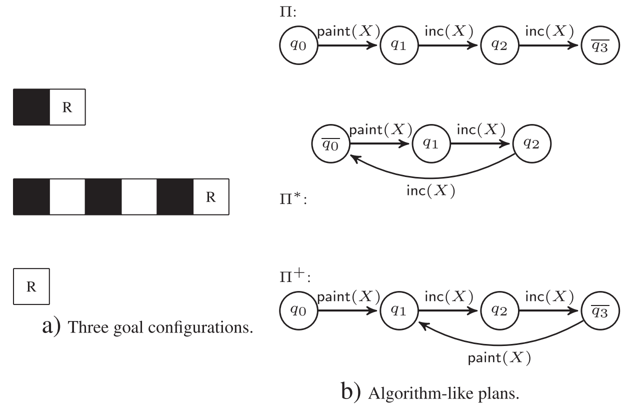

Figure 1: a) Robot in 2×1, 6×1 and 1×1 corridors; b) Three candidate generalized plans.

Introduction

Generalized planning studies the computation of plans that can solve a family of planning instances that share a common structure (Hu and De Giacomo 2011; Srivastava, Immerman, and Zilberstein 2011; Jiménez, Segovia-Aguas, and Jonsson 2019). Since *generalized planning* is computationally expensive, a common approach is to synthesize a generalized plan starting from a set of small instances and validate it on larger instances. This approach is related to the principle of Machine Learning (ML), in which a model is trained on a *training set* and validated on a *test set* (Mitchell 1997).

Traditionally, generalized planning has only focused on solvable instances, both for plan synthesis and for validation (Winner and Veloso 2003; Bonet, Palacios, and Geffner 2010; Hu and Levesque 2011; Srivastava et al. 2011; Hu and De Giacomo 2013; Segovia-Aguas, Jiménez, and Jonsson 2018; Segovia-Aguas, Celorrio, and Jonsson 2019). However, many computational problems in AI benefit from *negative* examples, e.g. SAT approaches that exploit clause learning (Angluin, Frazier, and Pitt 1992), grammar induction (Parekh, Honavar, and others 2000), program synthesis (Alur et al. 2018) and model learning (Camacho and McIlraith 2019). If used appropriately, negative examples can help reduce the solution space and accelerate the search for a solution.

In this paper we introduce *negative examples* for generalized planning as input planning instances that should not be solved by a generalized plan. An intuitive way to come up with negative examples for solutions that generalize is to first synthesize a solution with exclusively positive examples, and identify cases for which the solution did not generalize as desired, somewhat akin to *clause learning* in satisfiability problems (Biere et al. 2009). Imagine that we aim to synthesize the generalized plan $(\mathsf{paint}(X), \mathsf{inc}(X), \mathsf{inc}(X))^+$ that makes a robot paint every odd cell black in a $N \times 1$ corridor, starting from the leftmost cell (we use Kleene notation to represent regular expressions, and Z^+ indicates the repetition of Z at least once). Action $\mathsf{paint}(X)$ paints the current cell black while $\mathsf{inc}(X)$ increments the robot's X coordinate. The *positive example* of a 2×1 corridor (whose goal configuration is illustrated at Figure 1a) is solvable by all three automata plans in Figure 1b) (acceptor states are marked with overlines e.g., $\overline{q}$). These automata plans, namely Π, Π^* and Π^+, compactly represent the three sets of sequential

plans $(\mathsf{paint}(X), \mathsf{inc}(X), \mathsf{inc}(X))$, $(\mathsf{paint}(X), \mathsf{inc}(X), \mathsf{inc}(X))^*$ and $(\mathsf{paint}(X), \mathsf{inc}(X), \mathsf{inc}(X))^+$. Hence the single positive example of a 2×1 corridor is not enough to discriminate among these three generalized plans. Adding a second *positive example*, the 6×1 corridor in Figure 1a), discards plan Π. Adding a third 1×1 *negative example*, where the initial and goal robot cell are the same and no cell is required to be painted, discards Π^* preventing *over-generalization* because Π^* solves this negative example.

The problem of deriving generalized plans has been a longstanding open challenge. Compared to previous work on generalized planning, the contributions of this paper are:

1. **Negative examples** to more precisely specify the semantics of an aimed generalized plan.
2. A new approach for the synthesis of plans that can generalize from **smaller input examples** thanks to negative examples.
3. The definition of quantitative **evaluation metrics** to assess the generalization capacity of generalized plans.

The paper is organized as follows. We start with some background notation of classical planning and generalized planning (GP). Then we formalize the concept of a *negative example* for generalized planning. We continue with a description of a generalized planning problem with positive and negative examples that can be compiled to classical planning. We show proofs of soundness and completeness for the two main tasks in GP that are synthesis and validation. We continue with the experiments where we compare the impact of negative examples, and finally we conclude with a discussion on the presented work.

Background

This section formalizes the planning models used in this work as well as *planning programs* (Segovia-Aguas, Celorrio, and Jonsson 2019), an algorithm-like representation for plans that can generalize.

Classical planning with conditional effects

We use F to denote the set of *fluents* (propositional variables) describing a state. A literal l is a valuation of a fluent $f \in F$, i.e. $l = f$ or $l = \neg f$. A set of literals L on F represents a partial assignment of values to fluents (WLOG we assume that L does not assign conflicting values to any fluent). Given L, $\neg L = \{\neg l : l \in L\}$ is the complement of L. Finally, we use $\mathcal{L}(F)$ to denote the set of all literal sets on F, i.e. all partial assignments of values to fluents. A *state* s is a set of literals such that $|s| = |F|$, i.e. a total assignment of values to fluents. The number of states is then $2^{|F|}$.

A *classical planning frame* is a tuple $\Phi = \langle F, A \rangle$, where F is a set of fluents and A is a set of actions with *conditional effects*. Conditional effects can compactly define actions whose precise effects depend on the state where the action is executed. Each action $a \in A$ has a set of literals $\mathsf{pre}(a)$, called the *precondition*, and a set of *conditional effects*, $\mathsf{cond}(a)$. Each conditional effect $C \rhd E \in \mathsf{cond}(a)$ is composed of a set of literals C (the condition) and E (the effect). Action a is *applicable* in state s if and only if $\mathsf{pre}(a) \subseteq s$, and the resulting set of *triggered effects* is

$$\mathsf{eff}(s, a) = \bigcup_{C \rhd E \in \mathsf{cond}(a), C \subseteq s} E,$$

i.e. effects whose conditions hold in s. The result of applying a in s is the *successor state* $\theta(s, a) = (s \setminus \neg\mathsf{eff}(s, a)) \cup \mathsf{eff}(s, a)$.

A *classical planning problem* with conditional effects is a tuple $P = \langle F, A, I, G \rangle$, where F is a set of fluents and A is a set of actions with *conditional effects* as defined for a *planning frame*, I is an initial state and G is a goal condition, i.e. a set of literals to achieve.

A *solution* for a classical planning problem P can be specified using different representation formalisms, e.g. a sequence of actions, a partially ordered plan, a policy, etc. Here we define a *sequential plan* for P as an action sequence $\pi = \langle a_1, \ldots, a_n \rangle$ whose execution induces a state sequence $\langle s_0, s_1, \ldots, s_n \rangle$ such that $s_0 = I$ and, for each i such that $1 \leq i \leq n$, a_i is applicable in s_{i-1} and generates the successor state $s_i = \theta(s_{i-1}, a_i)$. The plan π *solves* P if and only if $G \subseteq s_n$, i.e. if the goal condition is satisfied following the execution of π in I.

Planning programs

Given a planning frame $\Phi = \langle F, A \rangle$, a *planning program* (Segovia-Aguas, Celorrio, and Jonsson 2019) is a sequence of instructions $\Pi = \langle w_0, \ldots, w_n \rangle$. Each instruction w_i, $0 \leq i \leq n$, is associated with a *program line* i and is drawn from the set of instructions $\mathcal{I} = A \cup \mathcal{I}_{go} \cup \{\mathsf{end}\}$, where $\mathcal{I}_{go} = \{\, \mathsf{go}(i', !f) : 0 \leq i' \leq n, f \in F\}$ is the set of *goto instructions*. In other words, each instruction is either a planning action $a \in A$, a goto instruction $\mathsf{go}(i', !f)$ or a termination instruction end.

The execution model for a planning program Π is a *program state* (s, i), i.e. a pair of a planning state $s \subseteq \mathcal{L}(F)$ (with $|s| = |F|$), and a program counter $0 \leq i \leq n$. Given a program state (s, i), the execution of instruction w_i on line i is defined as follows:

- If $w_i \in A$, the new program state is $(s', i + 1)$, where $s' = \theta(s, w)$ is the *successor* for planning state s and action w.
- If $w_i = \mathsf{go}(i', !f)$, the program state becomes $(s, i + 1)$ if $f \in s$, and (s, i') otherwise. Conditions in Goto instructions can represent arbitrary formulae since f can be a *derived fluent* (Lotinac et al. 2016).
- If $w_i = \mathsf{end}$, execution terminates.

To execute a planning program Π on a planning problem $P = \langle F, A, I, G \rangle$, the initial program state is set to $(I, 0)$, i.e. the initial state of P and program line 0. A program $\Pi = \langle w_0, \ldots, w_n \rangle$ *solves* a planning problem $P = \langle F, A, I, G \rangle$ iff the execution terminates in a program state (s, i) that satisfies the goal conditions, i.e. $w_i = \mathsf{end}$ and $G \subseteq s$.

Segovia-Aguas, Celorrio, and Jonsson (2019) contains a detailed analysis of the failure conditions on planning programs, which we summarize here as follows:

Corollary 1 (Planning Program Failure)**.** *If a planning program* Π fails *to solve a planning problem* P*, the only possible sources of failure are:*

1. **Incomplete Program**. *Execution terminates in program state* (s, i) *but the goal condition does not hold, i.e.* $(w_i = \mathsf{end}) \wedge (G \not\subseteq s)$.
2. **Inapplicable Action**. *Executing an action* $w_i \in A$ *in program state* (s, i) *fails because its precondition does not hold, i.e.* $\mathsf{pre}(w) \not\subseteq s$.
3. **Infinite Loop**. *The program execution enters into an infinite loop that never reaches an* end *instruction.*

Generalized planning

We define a *generalized planning problem* as a finite set of classical planning problems $\mathcal{P} = \{P_1, \ldots, P_T\}$ that are defined on the same planning frame Φ. Therefore, $P_1 = \langle F, A, I_1, G_1\rangle, \ldots, P_T = \langle F, A, I_T, G_T\rangle$ share the same fluents and actions but differ in the initial state and goals.

A *planning program* Π solves a given generalized planning problem $\mathcal{P}$ iff Π solves every planning problem $P_t \in \mathcal{P}, 1 \leq t \leq T$.

Segovia-Aguas, Celorrio, and Jonsson (2019) showed that a program Π that solves a generalized planning task $\mathcal{P}$ can be synthesized by defining a new classical planning problem $P_n = \langle F_n, A_n, I_n, G_n\rangle$, where n is a bound on the number of program lines. A solution plan π for P_n programs instructions on the available empty lines (building the n-line program Π), and validates Π on each input problem $P_t \in \mathcal{P}$.

The **fluent set** is defined as $F_n = F \cup F_{pc} \cup F_{ins} \cup F_{test} \cup \{\mathsf{done}\}$, where:

- $F_{pc} = \{\mathsf{pc}_i : 0 \leq i \leq n\}$ models the *program counter*,
- $F_{ins} = \{\mathsf{ins}_{i,w} : 0 \leq i \leq n, w \in \mathcal{I} \cup \{\mathsf{nil}\}\}$ models the program lines (nil indicates an empty line),
- $F_{test} = \{\mathsf{test}_t : 1 \leq t \leq T\}$ indicates the classical planning problem $P_t \in \mathcal{P}$.

Each instruction $w \in \mathcal{I}$ is modeled as a planning action, with one copy end_t of the termination instruction per input problem P_t. Preconditions for the goto and end instructions are defined as $\mathsf{pre}(\mathsf{go}(i', !f)) = \emptyset$ and $\mathsf{pre}(\mathsf{end}_t) = G_t \cup \{\mathsf{test}_t\}$. The authors define two **actions** for each instruction w and line i: a *programming action* $P(w_i)$ for programming w on line i, and an *execution action* $E(w_i)$ that uses the previous execution model to execute w on line i:

$$\begin{aligned}\mathsf{pre}(P(w_i)) &= \mathsf{pre}(w) \cup \{\mathsf{pc}_i, \mathsf{ins}_{i,\mathsf{nil}}\},\\ \mathsf{cond}(P(w_i)) &= \{\emptyset \rhd \{\neg\mathsf{ins}_{i,\mathsf{nil}}, \mathsf{ins}_{i,w}\}\},\\ \mathsf{pre}(E(w_i)) &= \mathsf{pre}(w) \cup \{\mathsf{pc}_i, \mathsf{ins}_{i,w}\}.\end{aligned}$$

The effect of $E(w_i)$ depends on the type of instruction:

$$\begin{aligned}&\mathsf{cond}(E(w_i)) = \mathsf{cond}(w) \cup \{\emptyset \rhd \{\neg\mathsf{pc}_i, \mathsf{pc}_{i+1}\}\}, w \in A,\\ &\mathsf{cond}(E(w_i)) = \{\emptyset \rhd \{\neg\mathsf{pc}_i\}\} \cup \{\{f\} \rhd \{\mathsf{pc}_{i+1}\}\}\\ &\qquad\qquad \cup \{\{\neg f\} \rhd \{\mathsf{pc}_{i'}\}\}, \quad w = \mathsf{go}(i', !f),\\ &\mathsf{cond}(E(w_i)) = \mathsf{reset}_{t+1}, \qquad w = \mathsf{end}_t, t < T,\\ &\mathsf{cond}(E(w_i)) = \{\emptyset \rhd \{\mathsf{done}\}\}, \qquad w = \mathsf{end}_T.\end{aligned}$$

The conditional effect reset_{t+1} resets the program state to $(I_{t+1}, 0)$, preparing execution on the next problem P_{t+1}.

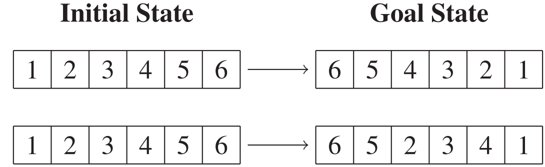

Figure 2: *Positive example* (upper row) and *negative example* (lower row) for the generalized planning task of reversing a list.

The **initial state** is $I_n = I_1 \cup \{\mathsf{pc}_0\} \cup \{\mathsf{ins}_{i,\mathsf{nil}} : 0 \leq i \leq n\}$ indicating that, initially, the program lines are empty and the program counter is at the first line. The **goal** is $G_n = \{\mathsf{done}\}$ and can only be achieved after solving sequentially all the instances in the generalized planning problem.

Negative examples in generalized planning

This section extends the previous generalized planning formalism to include *negative examples* for the validation and synthesis of programs.

Negative examples as classical planning problems

Negative examples are additional solution constraints to more precisely specify the semantics of an aimed generalized plan and prevent undesired generalizations.

Definition 1 (Negative examples in generalized planning)**.** *Given a generalized planning problem* $\mathcal{P}$*, a negative example is a classical planning instance* $P^- = \langle F, A, I^-, G^-\rangle$ *that must not be solved by solutions to* $\mathcal{P}$*.*

Figure 2 shows an input/output pair $(1, 2, 3, 4, 5, 6)/(6, 5, 4, 3, 2, 1)$ that represents a positive example for computing a generalized plan that reverse lists of any size. This example can be encoded as a classical planing problem, where the set of fluents are the state variables necessary for encoding a list of arbitrary size plus two pointers over the list nodes. The initial state encodes, using these fluents, the particular list $(1, 2, 3, 4, 5, 6)$. The goal condition encodes the target list $(6, 5, 4, 3, 2, 1)$. Finally, actions encode the swapping of the content of two pointers as well as actions for moving the pointers along the list. In this regard, the input/output example $(1, 2, 3, 4, 5, 6)/(6, 5, 4, 3, 2, 1)$ is a *positive example* while $(1, 2, 3, 4, 5, 6)/(6, 5, 2, 3, 4, 1)$ or $(4, 3, 2, 1)/(2, 3, 4, 1)$ are *negative examples* for the generalized planning task of reversing lists.

In this work both *positive* and *negative* examples are classical planning problems $P_1 = \langle F, A, I_1, G_1\rangle, \ldots, P_T = \langle F, A, I_T, G_T\rangle$ defined on the same fluent set F and action set A. Each problem $P_t \in \mathcal{P}$, $1 \leq t \leq T$ encodes an *input* specification in its initial state I_t while G_t encodes the specification of its associated *output*. Although the examples share actions, each action in A can have different interpretations in different states due to *conditional effects*. For instance, back to the example of Figure 1, we can encode individual planning tasks with different corridor sizes (the set of fluents F can include fluents of type $\mathsf{max}(N)$ that encode

SAT		UNSAT
Positives	Negatives	
$P_{1^+}, \dots, P_{T^+}$	$P_{T^++1}, \dots, P_T$	

Figure 3: Taxonomy of instances in *generalized planning*.

different corridor boundaries (Segovia-Aguas, Celorrio, and Jonsson 2019)).

Negative examples should not be confused with *unsolvable planning instances*. The goals of negative examples are reachable from the given initial state (see Figure 3). For instance the goals of the negative example $(1,2,3,4,5,6)/(6,5,2,3,4,1)$ shown in Figure 2 can be reached from the associated initial state by applying the corresponding actions to swap the content of pointers and moving appropriately those pointers. On the other hand $(4,3,2,1)/(1,1,1,1)$ would represent an UNSAT classical planning instance, because $(1,1,1,1)$ is not reachable starting from $(4,3,2,1)$ and provided the mentioned actions for reversing lists.

Program validation with negative examples

In generalized planning the process of *plan validation* is implicitly required as part of *plan synthesis*, since computing a solution plan requires us to validate it on all the given input instances. Next, we extend the notion of validation to consider also negative examples.

Definition 2 (Program Validation with *Positive* and *Negative* examples)**.** *Given a program* Π *and a set of classical planning problems* $\mathcal{P} = \{P_1, \dots, P_T\}$ *labeled either as* positive *or* negative*,* validation *is the task of verifying whether* Π *solves each* $P \in \mathcal{P}$ *labeled as* positive*, while it fails to solve each* $P \in \mathcal{P}$ *that is labeled as* negative*.*

Validating a sequential plan on a classical planing problem is straightforward because either a *validation* proof, or a *failure* proof, is obtained by executing the plan starting from the initial state of the planning problems (Howey, Long, and Fox 2004). Validating a program on a classical planning problem is no longer straightforward because it requires a mechanism for detecting *infinite loops* (checking failure conditions 1. and 2. is however straightforward).

The execution of plans with control flow on a given planning problem is compilable into classical planning. Examples are compilations for GOLOG *procedures* (Baier, Fritz, and McIlraith 2007), *Finite State Controllers* (Bonet, Palacios, and Geffner 2010; Segovia-Aguas, Jiménez, and Jonsson 2018) or *planning programs* (Segovia-Aguas, Jiménez, and Jonsson 2016; Segovia-Aguas, Celorrio, and Jonsson 2019). These compilations encode the *cross product* of the given planning problem and the automata corresponding to the plan to execute. The plan is *valid* iff the compiled problem is solvable. If a classical planner proves the compiled problem is unsolvable, then the plan is *invalid* because its execution necessarily failed.

Besides current planners do not excel at proving that a given problem is *unsolvable* (Eriksson, Röger, and Helmert 2017), none of the cited compilations can identify the precise source of a failed plan execution. Next, we show that the classical planning compilation for the synthesis of *planning programs* (Segovia-Aguas, Celorrio, and Jonsson 2019) can be updated with a mechanism for detecting infinite loops, that is taken from an approach for the computation of infinite plans (Patrizi et al. 2011). This updated compilation can identify the three possible sources of execution failure (namely **incomplete programs**, **inapplicable actions** and **infinite loops**) of a program in a given classical planning problem. What is more, the compilation can be further updated for solving *generalized planning problems with positive and negative examples*.

A compilation for program validation

Given a generalized planning task $\mathcal{P} = \{P_1, \dots, P_T\}$ and a program Π, program validation is compilable into a planning instance $P'_n = \langle F'_n, A'_n, I'_n, G_n \rangle$, that extends P_n from the *background* Section . The extended fluent set is $F'_n = F_n \cup F_{neg} \cup F_{copy}$, where

- $F_{neg} = \{\mathsf{checked}, \mathsf{holds}, \mathsf{stored}, \mathsf{acted}, \mathsf{loop}\}$ contains flags for identifying the source of execution failures,
- $F_{copy} = \{\mathsf{copy}_f, \mathsf{correct}_f : f \in F \cup F_{pc}\}$ contains the fluents used to store a copy of the program state with the aim of identifying infinite loops.

Unlike A_n, the action set A'_n does not contain *programming action* (these actions are only necessary for program synthesis but not for program validation). However, A'_n contains a new type of action called *check action* that verifies whether the precondition of an instruction holds. For an instruction w and line i, the check action $C(w_i)$ is defined as

$$\mathsf{pre}(C(w_i)) = \{\mathsf{pc}_i, \mathsf{ins}_{i,w}, \neg\mathsf{checked}, \neg\mathsf{loop}\},$$
$$\mathsf{cond}(C(w_i)) = \{\emptyset \rhd \{\mathsf{checked}\}\} \cup \{\mathsf{pre}(w) \rhd \{\mathsf{holds}\}\}.$$

After applying $C(w_i)$, execution fails if holds is false, either because the goal condition G_t is not satisfied when we apply a termination instruction end_t, or because the precondition $\mathsf{pre}(w)$ of the action $w \in A$ does not hold (which corresponds precisely to failure conditions 1. and 2. above). A similar mechanism has been previously developed for computing *explanations/excuses* of when a plan cannot be found (Göbelbecker et al. 2010).

Each execution action $E(w_i)$ is defined as before, but we add precondition $\{\mathsf{checked}, \mathsf{holds}\}$ and the conditional effect $\emptyset \rhd \{\neg\mathsf{checked}, \neg\mathsf{holds}, \mathsf{acted}\}$. As a result, $C(w_i)$ has to be applied before $E(w_i)$, and $E(w_i)$ is only applicable if execution does not fail (i.e. if holds is true).

To identify *infinite loops* A'_n is extended with three new actions:

- store, which stores a copy of the current program state.

$$\mathsf{pre}(\mathsf{store}) = \{\neg\mathsf{checked}, \neg\mathsf{stored}, \mathsf{acted}\},$$
$$\mathsf{cond}(\mathsf{store}) = \{\emptyset \rhd \{\mathsf{stored}, \neg\mathsf{acted}\}\} \cup \{\{f\} \rhd \{\mathsf{copy}_f\} : \forall f \in F \cup F_{pc}\}.$$

This action can be applied only once, after an action execution $E(w_i)$ and before checking an action $C(w_i)$. It simply uses conditional effects to store a copy of the program state (s, i) in the fluents of type copy_f.

- compare, which compares the current program state (s, i) with the stored copy.

$$\begin{aligned}\mathsf{pre}(\mathsf{compare}) &= \{\neg\mathsf{checked}, \mathsf{stored}, \mathsf{acted}, \neg\mathsf{loop}\},\\ \mathsf{cond}(\mathsf{compare}) &= \{\emptyset \rhd \{\neg\mathsf{stored}, \neg\mathsf{acted}, \mathsf{loop}\}\}\\ &\cup\{\{f, \mathsf{copy}_f\} \rhd \{\mathsf{correct}_f\} : f \in F \cup F_{pc}\}\\ &\cup\{\{\neg f, \neg\mathsf{copy}_f\} \rhd \{\mathsf{correct}_f\} : f \in F \cup F_{pc}\}.\end{aligned}$$

The result of the comparison is in the fluents of type $\mathsf{correct}_f$. Note that acted is not true after applying store so, to satisfy the precondition of compare, we have to apply an execution action first (otherwise the current program state would trivially equal the stored copy). For a fluent f to be correct, either it is true in both the current program state and the stored copy, or it is false in both.

- process, which processes the outcome of the comparison.

$$\begin{aligned}\mathsf{pre}(\mathsf{process}) &= \{\mathsf{loop}\} \cup \{\mathsf{correct}_f : f \in F \cup F_{pc}\},\\ \mathsf{cond}(\mathsf{process}) &= \{\emptyset \rhd \{\neg\mathsf{loop}, \mathsf{checked}\}\}.\end{aligned}$$

This action can only be applied if all fluents in $F \cup F_{pc}$ are correct, adds fluent checked, and resets other auxiliary fluents to false. The purpose of adding checked is to match the state of other failure conditions (checked is true and holds is false).

Finally, A'_n contain also actions skip_t, $1 \leq t \leq T$, that terminate program execution as a result of a failure condition. These actions are applicable once a failure condition is detected, of either type (checked is true and holds is false).

$$\begin{aligned}\mathsf{pre}(\mathsf{skip}_t) &= \{\mathsf{test}_t, \mathsf{checked}, \neg\mathsf{holds}\},\\ \mathsf{cond}(\mathsf{skip}_t) &= \mathsf{cond}(\mathsf{end}_t)\\ &\cup \{\emptyset \rhd \{\neg\mathsf{checked}, \neg\mathsf{stored}\}\}\\ &\cup \{\emptyset \rhd \{\neg\mathsf{copy}_f, \neg\mathsf{correct}_f : f \in F \cup F_{pc}\}\}.\end{aligned}$$

Note that the action applied immediately before skip_t identifies the source of the execution failure of the program Π on P_t. This action is either:

1. $C(\mathsf{end}_{t,i})$, identifying an **incomplete program**.
2. $C(w_i)$ such that $w \in A$, which proves that $w \in A$ is an **inapplicable action**.
3. process, identifying that the execution of the program entered an **infinite loop**.

The precondition $\neg$stored is added to all check actions $C(\mathsf{end}_{t,i})$, to avoid saving a stored copy of the program state from one input instance to the next. The goal condition is the same as before and in the initial state I'_n the instructions of the program Π are already programmed in the initial state:

$$I'_n = I_1 \cup \{\mathsf{pc}_0\} \cup \{\mathsf{ins}_{i,w} : 0 \leq i \leq n \wedge w_i \in \Pi\}.$$

Program synthesis with positive and negative examples

We define a *generalized planning problem with positive and negative examples* as a finite set of classical planning instances $\mathcal{P} = \{P_1, \ldots, P_{T^+}, \ldots, P_T\}$ that belong to the same planning frame. In this set there are T^+ positive and T^- negative instances such that $T = T^+ + T^-$ (see Figure 3). We assume that at least one positive instance is necessary ($T^+ > 0$) because otherwise, the one-instruction program $0.end$, covers any negative instance whose goals are not satisfied in the initial state.

To synthesize programs for generalized planning with positive and negative examples we extend the compilation P'_n with *programming instructions*. The output of the final extension of the compilation is a new planning instance $P''_n = \langle F''_n, A''_n, I''_n, G_n\rangle$:

- The fluent set F''_n is identical to that of the compilation P'_n, except that F''_n now includes a fluent negex, which is used to constrain the application of actions $E(\mathsf{end}_{t,i})$ and skip_t, respectively. By adding a precondition $\neg$negex to $E(\mathsf{end}_{t,i})$ and a precondition negex to skip_t, we require program execution to succeed for positive examples, and to fail for negative examples.
- The action set A''_n is identical to A'_n except that we reintroduce *programming actions* $P(w_i)$ in the action set A''_n and we add a precondition $\neg$negex to $E(\mathsf{end}_{t,i})$ and a precondition negex to skip_t to require program execution to succeed for positive examples, and to fail for negative examples. Moreover, precondition negex is added to all actions related to infinite loop detection (e.g. store, compare and process).
- All program lines are now empty in I''_n (so they can be programmed) and the goal condition is still $G_n = \{\mathsf{done}\}$, like in the original compilation.

Theorem 1 (Soundness)**.** *Any plan π that solves the planning instance P''_n induces a planning program Π that solves the corresponding generalized planning task with positive and negative examples $\mathcal{P} = \{P_1, \ldots, P_{T^+}, \ldots, P_T\}$.*

Proof. To solve the planning instance P''_n, a solution π has to use *programming actions* $P(w_i)$ to program the instructions on empty program lines, effectively inducing a planning program Π. Once an instruction w is programmed on line i, it cannot be altered and can only be executed using the given execution model (that is, using a *check action* $C(w_i)$ to test its precondition, followed by an *execution action* $E(w_i)$ to apply its effects). To achieve the goal $G_n = \{\mathsf{done}\}$, π has to simulate the execution of Π on each input instance P_t, $1 \leq t \leq T$, terminating with either $E(\mathsf{end}_{t,i})$ or skip_t, which are the only two actions that allow us to move on to the next input instance (if $t < T$) or add fluent done (if $t = T$). Because of the preconditions of $E(\mathsf{end}_{t,i})$ and skip_t, termination has to end with $E(\mathsf{end}_{t,i})$ if P_t is a positive example, and with skip_t if P_t is negative proving that Π solves each positive example and fails to solve each negative example. $\square$

Theorem 2 (Completeness)**.** *If there exists a program* Π *within* n *program lines that solves* $\mathcal{P} = \{P_1, \dots, P_{T^+}, \dots, P_T\}$ *then there exists a classical plan* π *that solves* P''_n*.*

Proof. We can build a prefix of plan π using programming actions that insert the instructions of Π on each program line. Now, we determine the remaining actions of π building a postfix that simulates the execution of Π on each input instance $P_t \in \mathcal{P}$. Since Π solves each positive example and fails to solve each negative example, it means that there exists an action sequence that simulates the execution of Π on P_t and ends with action $E(\mathsf{end}_{t,i})$ (if P_t is a positive example) and with skip_t (if P_t is negative). □

Experiments

This section reports the empirical performance of our approach for the *synthesis* and *evaluation* of programs for generalized planning[1]. All experiments are run on an *Intel Core i5 2.90GHz x 4* with a memory limit of 4GB and 600 seconds of planning timeout. In order to compare with previous approaches, we use Fast Downward (Helmert 2006) in the LAMA-2011 setting (Richter, Westphal, and Helmert 2011) to synthesize and evaluate programs using the presented compilations.

Experiments are carried out over the following generalized planning tasks. The *Green Block* consist of a tower of blocks where only one greenish block exists and must be collected. In *Fibonacci* the aim is to output the correct number in the Fibonacci sequence. In *Gripper* a robot has to move a number of balls from room A to room B, and in *List* the aim is to visit all elements of a linked list. Finally, in *Triangular Sum* we must calculate the triangular sum represented with the formula $y = \sum_0^N x$. We also introduce in this paper *RoboPainter* (RP), where a robot should paint, given different constraints, odd cells in a corridor (see Figure 1).

Computing programs with positive and negative examples

For the synthesis of programs that solve the previous generalized planning tasks, we compare two versions of our compilation, *PN-Lite* and *PN*, with the results from some problems whose solutions where solved and reported as *"One Procedure"* in Segovia-Aguas, Jiménez, and Jonsson (2016). We use *PN* to denote the version with positive and negative examples that detect the three possible failures of a planning program, whereas *PN-Lite* is a simpler sound version that detects *incomplete programs* and *inapplicable actions* but not *infinite loops*.

In this experiment we have run almost 100 random configurations with at most 5 instances that could be either positive or negative (where at least one is forced to be positive, see the previous section). The idea behind this experiment is to evaluate the use of negative examples as counter-examples to prevent undesired generalizations of programs that are synthesized from small input instances. Some domains from previous approaches are simple enough that they can generalize from few positive instances, so our compilations will only add complexity to the domain, requiring extra searching time required for failure detection.

In Table 1, columns *PN-Lite* and *PN* report the obtained results when we synthesize programs that are validated on both *positive* and *negative* examples. Recall that the P''_n compilation has additional fluents and actions compared to P_n, which imposes an extra searching time. However, including negative examples often makes it possible to synthesize programs from fewer positive examples and with fewer fluents (planning instances of smaller size) which, in general, increases the percentage of programs found. Also, the process of synthesis from few examples is a benefit in generalized planning compilations akin to few-shot learning (Lake, Salakhutdinov, and Tenenbaum 2015; Camacho and McIlraith 2019).

We briefly describe the best solutions that generalize for each domain in Table 1. In *Green Block*, we repeat the drop and unstack instructions while the green block is not hold, then we collect the holding green block. In the *Fibonacci* domain there are 4 variables called A, B, C and D that represent F_n, n, F_{n-1} and F_{n-2} respectively. The program assigns C to D, then A to C, then adds D to A and decreases B, repeating this sequence while B is different from 0. The planning program found in *Gripper* picks up a ball with the left hand, moves to room B, drops the ball, and goes back until no more balls are in room A. The *List* program visits the current node, moves to the next node and repeats the process until it reaches the tail. Finally, the program for *Triangular Sum* has a variable A initialized to 0 and countdown variable B that is added iteratively to A.

Evaluating generalized plans with test sets of positive and negative examples

Negative examples are useful for defining quantitative metrics that evaluate the coverage of generalized plans with respect to a *test set* of unseen examples. Given a labeled classical planning instance P and a program Π:

- If P is labeled as *positive* and Π solves P this means P is a **true positive**. Otherwise, if Π fails to solve P this means P is a **false positive**.
- If P is labeled as a *negative* example and Π solves P this means P is a **false negative**. Otherwise, if Π fails to solve P this means P is a **true negative**.

Our notion of *positive* and *negative examples* allows us to adopt metrics from ML for the evaluation of planning solutions that generalize with respect to a *test set*. These metrics are more informative than simply counting the number of positive examples covered by a solution and also consider the errors made over the *test set* (Davis and Goadrich 2006):

- **Precision**, $pr(\Pi) = \frac{p}{(p+p^-)}$, and **Recall**, $re(\Pi) = \frac{p}{(p+n^-)}$, where p is the number of *positive* examples solved by Π, p^- the number of *false positives* (classical planing problems labeled as *negative* that are solved by

[1] The source code, benchmarks and scripts are in the Automated Programming Framework (Segovia-Aguas 2017) such that any experimental data in the paper can be reproduced.

	n	$max(T)$	Only Positive		PN-Lite		PN	
			Avg. Search(s)	Found(%)	Avg. Search(s)	Found(%)	Avg. Search(s)	Found(%)
RoboPainter	5	5	**64.58**	50%	140.44	**60%**	86.43	40%
Green Block	4	5	**45.40**	81.25%	154.35	67.5%	99.12	**90%**
Fibonacci	5	5	190.14	25%	**93.96**	**27.5%**	-	0%
Gripper	4	5	**48.19**	**43.75%**	67.77	27.5%	107.14	27.5%
List	3	5	**0.04**	31.25%	0.07	27.5%	0.21	**45%**
T. Sum	3	5	143.36	**100%**	192.13	**100%**	**141.74**	**100%**

Table 1: *Program synthesis with positive and negative examples*: number of program lines (n), max number of input instances (T), average search time (secs) and, percentage of found solutions (with *only positive* and with *positive and negative* examples).

	$re(\Pi_{\text{P}})$	$pr(\Pi_{\text{P}})$	$ac(\Pi_{\text{P}})$	$re(\Pi_{\text{PN-Lite}})$	$pr(\Pi_{\text{PN-Lite}})$	$ac(\Pi_{\text{PN-Lite}})$	$re(\Pi_{\text{PN}})$	$pr(\Pi_{\text{PN}})$	$ac(\Pi_{\text{PN}})$
RoboPainter	71.74%	**100.00%**	95.19%	70.90%	**100.00%**	94.58%	**75.63%**	**100.00%**	**95.57%**
Green Block	68.86%	80.93%	91.48%	60.48%	75.00%	88.76%	**80.89%**	**88.38%**	**94.43%**
Fibonacci	17.86%	71.43%	85.47%	**22.22%**	**100.00%**	**85.97%**	-%	-%	-%
Gripper	41.54%	85.71%	87.68%	**62.38%**	**88.73%**	**91.35%**	35.44%	84.88%	86.30%
List	**8.08%**	**72.73%**	**81.89%**	5.96%	65.00%	81.00%	4.75%	47.22%	80.27%
T. Sum	71.74%	**100.00%**	95.19%	**75.63%**	**100.00%**	**95.57%**	70.90%	**100.00%**	94.58%

Table 2: *Program evaluation wrt a set of positive and negative tests* using the *recall*, *precision* and *accuracy* metrics. The -% symbol refers either to value not found because of an invalid operation.

Π) and n^- is the number of *false negatives* (instances labeled as positive examples that cannot be solved by Π).

- **Accuracy** is a more informed metric frequently used in ML, $ac(\Pi) = \frac{p+n}{p+n+p^-+n^-}$. In our case, n represents the number of *negative* examples unsolved by the program Π.

For this experiment we considered the planning program as given, i.e. the computed programs in the previous synthesis experiment. Then we compile a set of positive and negative instances but include the planning program in the initial state instead of having *empty* lines (as in the P_n and P'_n compilations). The execution of the computed programs on these instances must solve the positive instances while failing to solve the negative instances, verifying plan failure due to a failed condition or the detection of an infinite loop, as explained in the previous section.

We have used a framework for validating planning programs that reports *success* or specifies the *source of failure* when executing the program for each randomly generated planning instance. The results are shown in Table 2 where we report the *precision*, *accuracy* and *recall* of programs synthesized using *only positive examples*, and programs synthesized using the *positive and negative examples*. The list domain is the only case where positive examples yield to a better accuracy, while the rest of domains using positive and negatives improves only positives in all metrics.

Conclusion

Generalized planning provides an interesting framework to bridge the gap between ML and *AI planning* (Geffner 2018). On the one hand *generalized planning* follows a model based approach with declarative definitions of actions and goals, as *AI planning*. On the other hand *generalized planning*, as inductive ML, computes solutions that generalize over a set of input examples. *Generalized planning* has however little work dedicated to the assessment of the generality of plans beyond the given input planning tasks (positive examples only). ML however, has a long tradition on the empirical assessment of the generality of solutions. The fact that our compilation identifies the source of failures of program execution on a particular planning instance allows us to introduce negative examples and to bring the evaluation machinery from ML to define evaluation metrics that empirically assess the generality of plans beyond the given input planning tasks, e.g. using *test sets*.

Model checking (Clarke, Grumberg, and Peled 1999) provides effective solvers for automatically verifying correctness properties for diverse finite-state systems. More precisely when actions have non-deterministic effects, program validation becomes complex since it requires proving that all the possible program executions reach the goals. In such a scenario, *model checking* (Clarke, Grumberg, and Peled 1999) and *non-deterministic planning*, like POMDP planning, are definitely more suitable approaches for plan validation (Hoffmann 2015). An interesting research direction is to study how to leverage *model checking* techniques for the synthesis of generalized planning form both positive and negative examples. *Plan constraints* are also a compact way of expressing *negative* information for planning and reduce the space of possible solution plans. Plan constraints have already been introduced to different planning models using the LTL formalism (Baier, Bacchus, and McIlraith 2009; Bauer and Haslum 2010; Patrizi, Lipovetzky, and Geffner 2013; Camacho et al. 2017).

Acknowledgments

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement no. 731761, IMAGINE; and it is partially supported by grant TIN-2015-67959 and the Maria de Maeztu Units of Excellence Programme MDM-2015-0502, MEC, Spain. Sergio Jiménez

is supported by the *Ramon y Cajal* program, RYC-2015-18009, the Spanish MINECO project TIN2017-88476-C2-1-R, and the *generalitat valenciana* project GV/2019/082. Anders Jonsson is partially supported by the Spanish grants TIN2015-67959 and PCIN-2017-082.

References

Alur, R.; Singh, R.; Fisman, D.; and Solar-Lezama, A. 2018. Search-based program synthesis. *Communications of the ACM* 61(12):84–93.

Angluin, D.; Frazier, M.; and Pitt, L. 1992. Learning conjunctions of horn clauses. *Machine Learning* 9(2-3):147–164.

Baier, J. A.; Bacchus, F.; and McIlraith, S. A. 2009. A heuristic search approach to planning with temporally extended preferences. *Artificial Intelligence* 173(5-6):593–618.

Baier, J. A.; Fritz, C.; and McIlraith, S. A. 2007. Exploiting procedural domain control knowledge in state-of-the-art planners. In *International Conference on Automated Planning and Scheduling*, 26–33.

Bauer, A., and Haslum, P. 2010. Ltl goal specifications revisited. In *ECAI*, volume 10, 881–886.

Biere, A.; Heule, M.; van Maaren, H.; and Walsh, T. 2009. Conflict-driven clause learning sat solvers. *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications* 131–153.

Bonet, B.; Palacios, H.; and Geffner, H. 2010. Automatic derivation of finite-state machines for behavior control. In *AAAI*.

Camacho, A., and McIlraith, S. A. 2019. Learning interpretable models expressed in linear temporal logic. In *International Conference on Automated Planning and Scheduling*, volume 29, 621–630.

Camacho, A.; Triantafillou, E.; Muise, C.; Baier, J. A.; and McIlraith, S. A. 2017. Non-deterministic planning with temporally extended goals: Ltl over finite and infinite traces. In *AAAI*.

Clarke, E. M.; Grumberg, O.; and Peled, D. 1999. *Model checking*. MIT press.

Davis, J., and Goadrich, M. 2006. The relationship between precision-recall and roc curves. In *ICML*, 233–240. ACM.

Eriksson, S.; Röger, G.; and Helmert, M. 2017. Unsolvability certificates for classical planning. In *International Conference on Automated Planning and Scheduling*.

Geffner, H. 2018. Model-free, model-based, and general intelligence. In *IJCAI*.

Göbelbecker, M.; Keller, T.; Eyerich, P.; Brenner, M.; and Nebel, B. 2010. Coming up with good excuses: What to do when no plan can be found. *Cognitive Robotics* 10081.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research* 26:191–246.

Hoffmann, J. 2015. Simulated penetration testing: From "dijkstra" to "turing test++". In *International Conference on Automated Planning and Scheduling*, 364–372.

Howey, R.; Long, D.; and Fox, M. 2004. Val: Automatic plan validation, continuous effects and mixed initiative planning using pddl. In *ICTAI*, 294–301. IEEE.

Hu, Y., and De Giacomo, G. 2011. Generalized planning: Synthesizing plans that work for multiple environments. In *International Joint Conferences on Artificial Intelligence*.

Hu, Y., and De Giacomo, G. 2013. A generic technique for synthesizing bounded finite-state controllers. In *International Conference on Automated Planning and Scheduling*.

Hu, Y., and Levesque, H. J. 2011. A correctness result for reasoning about one-dimensional planning problems. In *International Joint Conferences on Artificial Intelligence*, 2638–2643.

Jiménez, S.; Segovia-Aguas, J.; and Jonsson, A. 2019. A review of generalized planning. *The Knowledge Engineering Review* 34.

Lake, B. M.; Salakhutdinov, R.; and Tenenbaum, J. B. 2015. Human-level concept learning through probabilistic program induction. *Science* 350(6266):1332–1338.

Lotinac, D.; Segovia-Aguas, J.; Jiménez, S.; and Jonsson, A. 2016. Automatic generation of high-level state features for generalized planning. In *International Joint Conferences on Artificial Intelligence*.

Mitchell, T. M. 1997. *Machine Learning*. New York, NY, USA: McGraw-Hill, Inc., 1 edition.

Parekh, R.; Honavar, V.; et al. 2000. Grammar inference, automata induction, and language acquisition. *Handbook of natural language processing* 727–764.

Patrizi, F.; Lipoveztky, N.; De Giacomo, G.; and Geffner, H. 2011. Computing infinite plans for ltl goals using a classical planner. In *International Joint Conferences on Artificial Intelligence*.

Patrizi, F.; Lipovetzky, N.; and Geffner, H. 2013. Fair ltl synthesis for non-deterministic systems using strong cyclic planners. In *International Joint Conferences on Artificial Intelligence*.

Richter, S.; Westphal, M.; and Helmert, M. 2011. Lama 2008 and 2011. In *International Planning Competition*.

Segovia-Aguas, J.; Celorrio, S. J.; and Jonsson, A. 2019. Computing programs for generalized planning using a classical planner. *Artificial Intelligence*.

Segovia-Aguas, J.; Jiménez, S.; and Jonsson, A. 2016. Generalized planning with procedural domain control knowledge. In *International Conference on Automated Planning and Scheduling*.

Segovia-Aguas, J.; Jiménez, S.; and Jonsson, A. 2018. Computing hierarchical finite state controllers with classical planning. *Journal of Artificial Intelligence Research* 62:755–797.

Segovia-Aguas, J. 2017. Automated programming framework. https://github.com/aig-upf/automated-programming-framework. Accessed: 2019-11-12.

Srivastava, S.; Immerman, N.; Zilberstein, S.; and Zhang, T. 2011. Directed search for generalized plans using classical planners. In *International Conference on Automated Planning and Scheduling*, 226–233.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2011. A new representation and associated algorithms for generalized planning. *Artificial Intelligence* 175(2):615–647.

Winner, E., and Veloso, M. 2003. Distill: Learning domain-specific planners by example. In *ICML*, 800–807.