Information Sciences 628 (2023) 148–176

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Learning cost action planning models with perfect precision via constraint propagation

Antonio Garrido

VRAIN, Valencian Research Institute for Artificial Intelligence, Universitat Politecnica de Valencia, Spain

ARTICLE INFO

Keywords:
Learning
Cost action model
Classical planning
Constraint programming
Reliability
Precision

ABSTRACT

Data-driven AI is rapidly gaining importance. In the context of AI planning, a constraint programming formulation for learning action models in a data-driven fashion is proposed. Data comprises plan observations, which are automatically transformed into a set of planning constraints which need to be satisfied. The formulation captures the essence of the action model and unifies functionalities that are individually supported by other learning approaches, such as costs, noise/uncertainty on actions, information on intermediate state observations and mutex reasoning.

Reliability is a key concern in data-driven learning, but existing approaches usually learn action models that can be imprecise, where imprecision here is an error indicator of learning something incorrect. On the contrary, the proposed approach guarantees reliability in terms of perfect precision by using constraint propagation. This means that what is learned is 100% correct (i.e., error-free), not only for the initial observations, but also for future observations. To our knowledge, this is a novelty in action model learning literature. Although perfect precision might potentially limit the amount of learned information, the exhaustive experiments over 20 planning domains show that such amount is comparable, and even better, to ARMS and FAMA, two state-of-the-art benchmarks in action model learning.

1. Introduction

AI planning is a deliberative task to build a plan of actions that, starting from an initial state, achieves a set of goals [1]. Planning requires the definition of action models by using a particular language, such as STRIPS [2], Functional Strips [3], ADL (Action Description Language [4]), or the *de facto* standard language PDDL (Planning Domain Definition Language [5]). In any of these languages, an action model describes the semantics of actions, classically represented in terms of preconditions, which are required before executing an action, and postconditions/effects, which are asserted after the action is executed. With these languages, a planning expert sits down and defines the full semantics of actions according to the physics of the planning scenario [6]. However, building action models from scratch has shown limited because of the difficulty, tedious and error-prone of manually specifying precise and complete action models [7–11], which is becoming a bottleneck for the applicability of planning technology to real-world scenarios with action concurrency [12–15]. Motivated by these limitations, there is a growing interest in the planning community for learning action models, as a particular case of data-driven learning, to help reduce the expert effort and improve the reliability of the models [7,11,16–18].

E-mail address: agarridot@dsic.upv.es.

https://doi.org/10.1016/j.ins.2023.01.111
Received 14 October 2022; Received in revised form 13 January 2023; Accepted 24 January 2023

Available online 31 January 2023

1.1. Learning action models

Learning is the process of gaining knowledge by studying from experience or observation. In AI planning, learning an action model is the process of acquiring the preconditions and effects of actions, by identifying common structures, from input observations. Observations might include sequences of state changes, input graphs that depict the state space, sample instances, input constraints, world transitions, expert demonstrations or, more typically, traces of plan executions [6,12,13]. Consequently, learning here is a cognitive task that can be considered as inverse planning [19]: planning requires an action model to create plans, whereas learning requires an observation of plans to create an action model.

The interest in learning is growing up because it allows us to acquire procedural knowledge that enhances the model of actions, which has a positive impact in future planning tasks. Reliability is a great challenge in learning. Reliable forms of precision, safety and robustness are essential to fully exploit such learned model, and are also rapidly gaining interest (see, for instance, the new specific workshop for Reliable Data-Driven Planning and Scheduling[1] in ICAPS). A learned model can be used to automatically elaborate similar models for similar scenarios, e.g., knowledge transfer or transfer learning [16], but this makes little sense if the models contain errors and/or are unreliable (transferring incorrect knowledge is not recommendable).

There is a wide number of scenarios that benefit from the discovery of a precise model of actions: recognition of past behavior for prediction and anticipation, diagnostic expertise, goal recognition, learning causality models, recommendation, programming and modeling, teleoperation, macro recording, sensing and controlling, robotics motion capturing and planning, explainable planning, etc. [7,14,16,18,20–23]. Learning is appealing because these scenarios include a huge number of tasks, which require accurate expert knowledge and manual tasks that are impractical for complex domains. Reliability is appealing because the learned models need to be precise, i.e., error-free, and faithful to the real world to be practical. After all, decision making is of little interest if the underlying model is incorrect; e.g., a diagnostic or recommendation is more limited if it relies on an incorrect model.

1.2. Current limitations and motivation

In the planning context, current (machine) learning approaches are mainly inductive and require large datasets of observations, e.g., hundreds and even thousands of plan traces, to compute a statistically significant model that optimizes some quality metrics over a reference or *ground truth* model [6,11,24–26]. Several metrics are available, mostly inherited from pattern recognition and classification literature [27]. Validation tests to measure missing information, accuracy and error rates[2] are the most simple metrics. They simply evaluate true + false positive/negative values. Precision and recall are more often used; they are more informative than accuracy and error rates, as they are based on a better understanding and measure of relevance. Briefly, precision provides a notion of soundness (or how correct the learned model is compared to the reference one), while recall gives a notion of completeness (or how much the learned model covers the reference one) [12,13].

Precision and recall are both desired but somewhat conflicting: learning more complete models tends to include some imprecision, whereas more precise models tend to be less complete. Consequently, the main limitation of an optimization-based learning task over a set of observations is twofold: it cannot guarantee soundness (the learned model may contain erroneous information) nor completeness (the learned model may not satisfy a particular observation, i.e., the model is unsafe). In other words, these approaches might learn action models that are incorrect and with undesirable side effects, i.e., models that are not 100% precise and could be potentially risky.

The motivation of this paper is clear. Learning a 100% complete action model is desirable. But learning a 100% precise model, without errors, is more desirable, and a key issue to address reliability. Moreover, in many scenarios (e.g., anticipation, detection of causal relationships and recognition of goals by proactive assistants), precision becomes essential: if one wants to learn something practical, it must be correct. Metaphorically speaking, learning that $2+2=5$ is unacceptable for a reliable scenario.

There are four key aspects that are desirable when learning action models (and have an impact in the applicability of the learning):

1. *Use of a limited number of observations*. In many real-world applications it is expensive, or even impossible, to collect large datasets of observations [16], or to retrieve the complete transitions given by a complex state space [8,28,29]. This is particularly difficult when the observations come from repetitive processes performed by humans, like in learning by demonstration [21].
2. *The learned model must satisfy all the input observations and not only some of them*. Statistical-based learning approaches might learn a little from each observation, but not a model that is 100% valid for any individual sample. In terms of safety, one is interested in learning a model that meets all the observations [30].
3. *The model must be learned with perfect precision*. The learned model must be correct not only for the input collection of observations, which is always limited, but also for any new observation (also an indication of robustness). All in all, what is learned must be 100% precise for all valid (current + future) observations, and what has no absolute precision must not be learned.
4. *The learned model must tolerate some uncertainty*. In terms of robustness, one is also interested in learning a model that supports some uncertainty, e.g., in the observed actions.

This paper presents an approach for learning action models in classical planning and contributes to meet all these general aspects.

[1] https://icaps22.icaps-conference.org/workshops/RDDPS.

[2] Metrics to measure errors are common as, unfortunately, errors are taken as granted.

1.3. Contributions

This paper uses the PDDL formalism for classical planning, where actions have preconditions, effects and also a cost to be learned. The preconditions and effects are defined as Boolean predicates. The cost is learned as a numeric value, a characteristic that is ignored in most existing approaches. Although the numeric information is restricted to costs, this could be easily generalized to other numeric expressions such as resource usage, rewards, penalties, etc. The observations are plan traces, in the form of actions, their start times and total cost. No conflicting observations are present, that is, one observation never contradicts another. This might seem a bit strong, but it is a required assumption when one wants to satisfy all the observations.

This paper proposes a Constraint Programming (CP) approach, which takes advantage of a very powerful type of constraint reasoning named constraint propagation [31–33]. Constraint propagation is an inference process that detects and removes inconsistent variable assignments that cannot participate in any feasible solution. It explicitly forbids assignments to variables because at least one of the constraints cannot be satisfied. Constraint propagation is typically used to reduce a CP problem to another, equivalent but simpler. Moreover, it can be used for consistency tests so that the knowledge derived is propagated and reused for acquiring new additional knowledge [34]. Such knowledge is propagated in the CP formulation to find out what cannot be valid in such formulation (because it violates some constraints) and what must be valid (because it is necessary to satisfy the constraints). This is the essence of learning action models with perfect precision.

The main contributions of this work are:

- It automatically builds a flexible CP formulation to model the constraints of an arbitrary collection of observed plan traces. It accepts different levels of input data: states, actions, exclusion relationships, etc.
- It requires information on the initial and goal state of each plan trace, the sequence of actions that are planned together with their start times, and the plan cost. Unlike some other approaches, the goal state does not need to represent a full state with all the predicates, but only those that represent the planning goals. Also, the intermediate states between actions are not required at all, i.e., the state trajectory is totally unknown.
- It supports input knowledge on mutual exclusion relationships, which can be optionally included. This is a significant difference *w.r.t.* other works, because exclusion relationships are traditionally ignored in the literature.
- It only requires a few tens of plan traces, whereas other learning approaches require datasets with hundreds of plan traces and thousands of actions. In the experiments, no more than 50 traces are used.
- It learns from plan traces with some uncertainty. The actions of the plan are assumed flawless, but there is uncertainty on the exact time when they happen. This noise is common in real-world applications, where observations are captured by inaccurate sensors or that could be occasionally damaged: one knows that an action has been executed, but (s)he is unsure on its time. This makes a difference to some other systems, which assume that plan traces are noiseless, and increases the robustness of the learning.
- It learns cost action planning models precisely. It differs from existing approaches in the incremental way it learns, as constraint propagation learns only the model structure that must be true: it learns a safe model that satisfies all the observations and is also 100% precise. Although ensuring perfect precision might limit the amount of information that is learned (i.e., the recall) only to what is correct, the extensive experiments prove that the amount of information that is learned is comparable (and even higher in some domains) to other approaches that cannot guarantee perfect precision. That is, the model learned is not only precise, but also very complete. The experiments are tested on more than 20 planning domains, whereas most learning approaches use no more than 10 domains.

Although this work is inspired by [13,35], it introduces three important extensions that make it different, represented above as contributions. First, it is a more compact and simplified representation, as it now works with a classical (non-durative) action model. Second, it observes plan traces where the start time of actions is uncertain, which implies noisy observations. Third, and more importantly, it proposes a novel constraint propagation to learn as much as possible from a minimal amount of observations and ensure perfect precision.

1.4. Organization of the paper

This paper is organized as follows:

- Section 2 formalizes the terminology on planning, CP and learning. It also introduces the quality metrics, essential for evaluating the learned models in the paper, and discusses the related work in detail.
- Section 3 constitutes the core of the paper from the formulation point of view. It describes the CP formulation, in terms of the variables and constraints, and provides a working example.
- Section 4 is the core of the paper from the algorithmic point of view. It proposes the algorithms for learning preconditions, effects and costs via constraint propagation, and prove their properties.
- Section 5 presents a complete evaluation of the quality of the learned models. It provides the setup of the experiments, how different elements (i.e., plan library and noise) affect the learning, and compares this approach to others.
- Section 6 discusses the lessons learned after the analysis of the results.
- Section 7 concludes the paper. It highlights the limitations of this work and provides opportunities for future research.

```
(:types locatable city flevel - object
        aircraft person - locatable)

(:predicates (at ?x - locatable ?c - city) (in ?p - person ?a - aircraft)
             (next ?l1 ?l2 - flevel) (fuel-level ?a - aircraft ?l - flevel))

(:action board
 :parameters (?p - person ?a - aircraft ?c - city)
 :precondition (and (at ?p ?c) (at ?a ?c))
 :effect (and (not (at ?p ?c)) (in ?p ?a) (increase (cost) 1)))

(:action refuel
 :parameters (?a - aircraft ?c - city ?l1 - flevel ?l2 - flevel)
 :precondition (and (fuel-level ?a ?l1) (next ?l1 ?l2) (at ?a ?c))
 :effect (and (not (fuel-level ?a ?l1)) (fuel-level ?a ?l2)
              (increase (cost) 2)))
```

Fig. 1. Types, predicates and two operators of the PDDL domain named *zenotravel*. Other (not shown) operators are `debark`, `fly` and `zoom`.

2. Formalization and related work

The objective of this section is twofold. On the one hand, to formalize the concepts and notation on classical planning, CP, learning, and the metrics used in the paper. On the other hand, to present related work and highlight the main differences *w.r.t.* the proposed approach.

2.1. Planning concepts

The planning scenario is defined by following the PDDL structure, which separates the planning domain, problem and plan.

2.1.1. The domain

Let us assume a hierarchy of types $\mathcal{T}$ and a set of Boolean predicates $\mathcal{P}$, with a list of typed parameters over $\mathcal{T}$. PDDL defines a set of $\mathcal{T}$-parameterized operators $\mathcal{O}$ to model a planning domain. The number and type of the parameters restrict the subset of predicates to be used per operator. Each operator $o \in \mathcal{O}$ has a set of positive preconditions ($\mathsf{pre}(o)$) and a set of positive and negative effects that are asserted and retracted, respectively ($\mathtt{eff}(o) = \{\mathtt{eff}^+(o) \cup \mathtt{eff}^-(o)\}$); $\mathsf{pre}(o), \mathtt{eff}^+(o), \mathtt{eff}^-(o) \subseteq \mathcal{P}$. An operator can be applied when its preconditions hold, and the effects happen after its application. The STRIPS version of learning for preconditions and effects is assumed [6,12,24], which means: $\mathtt{eff}^-(o) \subseteq \mathsf{pre}(o)$, $\mathtt{eff}^+(o) \cap \mathtt{eff}^-(o) = \emptyset$ and $\mathsf{pre}(o) \cap \mathtt{eff}^+(o) = \emptyset$. The definition of o is extended with $\mathsf{cost}(o)$, a positive value that represents the cost of applying o. This is very valuable for keeping track of the plan cost and allows planners to optimize it. For simplicity, the cost is assigned to the operator. If the cost depends on the parameters of the operator, e.g., a cost that depends on the distance traveled, different operators with different costs should be defined.

A planning domain is defined as $\delta = \langle \mathcal{T}, \mathcal{P}, \mathcal{O} \rangle$. Fig. 1 depicts part of a transportation domain named *zenotravel*. There are some predicates, such as `(at ?x - locatable ?c - city)`, `(in ?p - person ?a - aircraft)` or `(next ?l1 ?l2 - flevel)` defined over a hierarchy of types `locatable`, `city`, `person`, etc. The predicates are defined over the types and model relations in the domain. For instance, `(at ?x - locatable ?c - city)` represents the fact that a `locatable` (`aircraft` or `person`) is `at` a `city`, `(next ?l1 ?l2 - flevel)` represents that fuel level `?l1` is before `?l2`, etc. Operators[3] use predicates as preconditions and effects. For instance, `board` takes a person `p` `at` a city `c` and loads him/her `in` an aircraft `a`. Clearly, `p` and `a` must be initially `at` the same city and the person is finally `in` `a`, and no longer `at` `c`. `refuel` changes the fuel level of an aircraft `a` located `at` a city from `?l1` to the next level `?l2`. In PDDL the cost is modeled as a particular numeric effect that increases a `cost` expression, represented as $\mathsf{cost}(o)$ for simplicity; in the example, $\mathsf{cost}(\texttt{board})= 1$ and $\mathsf{cost}(\texttt{refuel}) = 2$ no matter their parameters.

Mutual exclusion (mutex) constraints When two predicates in $\mathcal{P}$ cannot hold simultaneously in δ, they are mutex. For instance, in *zenotravel*, ⟨`(at ?x - locatable ?y1 - city)`, `(at ?x - locatable ?y2 - city)`⟩ are mutex if `?y1` $\neq$ `?y2`: in this domain `?x` cannot be at two different cities simultaneously. Similarly, ⟨`(at ?x - person ?y1 - city)`, `(in ?x - person ?y2 - aircraft)`⟩ are mutex, as a person `?x` cannot be `at` a city `?y1` and `in` an aircraft `?y2`. The domain expert knows this while modeling because it is part of the physics of the domain, but this knowledge is not explicitly represented in PDDL. This is only specified if every time one predicate is asserted the other is immediately retracted and vice versa (contradictory effects).

Static information $\mathcal{P}$ may contain predicates that never change in δ, which are known as static. Static predicates are also part of the physics of the domain and they are relevant for the grounding stage, prior to the planning itself. Note the `(next ?l1`

[3] PDDL uses the construct *action* instead of *operator*. To avoid confusion in the notation, this paper distinguishes between operators, i.e., templates with parameters, and actions, i.e., instantiated operators where all parameters are grounded to constant values.

```
1: (refuel plane1 city0 fl1 fl2)
1: (board person2 plane2 city1)
1: (board person1 plane1 city0)
2: (zoom plane1 city0 city1 fl2 fl1 fl0)
2: (zoom plane2 city1 city2 fl3 fl2 fl1)
3: (debark person1 plane1 city1)
...
```

Fig. 2. Fragment of a parallel plan for *zenotravel* with a cost $\mathcal{K}$.

`?l2 - flevel)` predicate in Fig. 1, which represents the natural order between `?l1` and `?l2`, or imagine a potential predicate `(fuel-station ?c - city)`, indicating that there is a fuel station in `?c`, to be used as a precondition for `refuel`. Those predicates will never be used as effects (never asserted/retracted) since no operator can change the physics of *zenotravel*: one fuel level is always before another, and fuel stations do not (dis)appear in cities. Since static predicates always hold, they could eventually be present as preconditions in all the operators, which hinders learning. Static information is not labeled explicitly in PDDL, but can be easily detected.

Input knowledge for learning Most planners perform some kind of reasoning to discover mutex constraints and static information, thus improving the planning task. Similarly, learning can also be improved by providing additional input knowledge. First, giving the list of all predicates $p_i, p_j \in \mathcal{P}$ that are mutex in δ, defined as the set $\mu(\delta) = \{\langle p_i, p_j \rangle\}$. Second, restricting the set of predicates $\mathcal{P}' \subseteq \mathcal{P}$, where $\mathcal{P}'$ contains no static predicates. This knowledge is very valuable for learning, as seen below.

2.1.2. The problem

Let us assume a domain $\delta = \langle \mathcal{T}, \mathcal{P}, \mathcal{O} \rangle$. Given a set of $\mathcal{T}$-typed constant values (*aka* objects), $\mathcal{V}$ and $\mathcal{A}$ are defined. $\mathcal{V}$ is the set of Boolean predicates instantiating these values in $\mathcal{P}$, thus defining a mapping between $\mathcal{P}$ and $\mathcal{V}$ as used in Section 3.2. $\mathcal{A}$ is the set of actions, which are instantiated from the operators $\mathcal{O}$. All the parameters in $\mathcal{A}$ are grounded, and the same for their preconditions + effects, which are thereby instantiated and mapped in $\mathcal{V}$. The number of instantiations depends on the parameters and the possible values for each parameter. For instance, from Fig. 1 and given the constant values {`plane1 - aircraft`, `person1 - person`, `city0 - city`}, $\mathcal{V}$ = { `(at plane1 city0)`, `(at person1 city0)`, `(in person1 plane1)`} and $\mathcal{A}$ = {`(board person1 plane1 city0)`} are obtained. A state $\mathcal{S}$ is defined as an assignment of true/false values to predicates in $\mathcal{V}$. $\mathcal{S}$ is a full state if $|\mathcal{S}| = |\mathcal{V}|$ and a partial state if $|\mathcal{S}| < |\mathcal{V}|$.

A classical planning problem ρ for δ is defined as $\rho = \langle \delta, \mathcal{V}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$, where the initial state $\mathcal{I}$ is a full state ($|\mathcal{I}| = |\mathcal{V}|$), and $\mathcal{G}$ is the goal state. Although $\mathcal{G}$ can be a partial or full state ($|\mathcal{G}| \leq |\mathcal{V}|$), in PDDL it is typically defined as a partial state. $\mathcal{A}$ is the set of potential actions that can be used to reach $\mathcal{G}$ from $\mathcal{I}$.

2.1.3. The plan

Let us assume a problem $\rho = \langle \delta, \mathcal{V}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$. A plan trace, or simply a plan, for ρ is a tuple $\pi(\rho) = \langle \{\langle t_1, a_1 \rangle, \langle t_2, a_2 \rangle \dots \langle t_n, a_n \rangle\}, \mathcal{K} \rangle$. Each $\langle t_i, a_i \rangle$ contains an action $a_i \in \mathcal{A}$ and t_i as the time when a_i happens. Note that different actions can happen at the same time ($t_i = t_j$ for $a_i \neq a_j$ is possible), as parallel plans are used. Also, actions in $\mathcal{A}$ can have several occurrences in $\pi(\rho)$ provided they happen at different times ($t_i \neq t_j$ for $a_i = a_j$ is possible). $\mathcal{K}$ is the cost of the plan. $\pi(\rho)$ induces a chronologically-ordered sequence of full states $\langle \mathcal{S}_0 \dots \mathcal{S}_{end} \rangle$, where $\mathcal{S}_0 = \mathcal{I}$ and $\mathcal{G} \subseteq \mathcal{S}_{end}$. The plan length (*aka* makespan) is the time of the state $\mathcal{S}_{end}$. A plan does not need to be optimal in terms of length or cost, but all their actions are assumed to be relevant and play a role in the plan. In other words, if an action is removed, the plan becomes invalid. Fig. 2 shows a fragment of a parallel plan for *zenotravel*, with the timestamps and actions.

2.2. The CP paradigm

In CP constraints are declared over a set of decision variables that represent the properties a solution must satisfy. This paper focuses on the particular case of a Constraint Satisfaction Problem (CSP), namely $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, where $\mathcal{X}$ is a set of variables, $\mathcal{D}$ represents the domain for each variable and $\mathcal{C}$ is a set of constraints among the variables in $\mathcal{X}$ that restrict their possible values in $\mathcal{D}$.

2.2.1. CSP solving

A CSP is typically solved by using search methods that find an evaluation of values in the corresponding domains to a subset of variables $\mathcal{X}_{ev} \subseteq \mathcal{X}$. An evaluation of values to variables is consistent if it does not violate any of the constraints in $\mathcal{C}$. Such an evaluation is complete if it includes all variables in $\mathcal{X}$, that is, $|\mathcal{X}_{ev}| = |\mathcal{X}|$. An evaluation is a solution for the CSP if it is both consistent and complete.

A CSP can have many solutions, i.e., different consistent and complete evaluations. It is important to note that in absence of a metric over $\mathcal{X}$ one is dealing with a pure satisfaction problem, rather than a Constraint Optimization Problem. Thus, many solutions are possible and equally valid.

2.2.2. Constraint propagation

Constraint propagation is a reasoning mechanism used in CP that reduces an original CSP to another simpler, without changing its solutions, by enforcing consistency [31,32]. Several types of consistency can be considered, such as node, arc and path consistency, which are supported by most propagators with linear, quadratic and cubic complexity, respectively.

Given $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, constraint propagation performs a transformation process that infers a new $\langle \mathcal{X}, \mathcal{D}', \mathcal{C}' \rangle$. The objective is to create new domains in $\mathcal{D}'$, more reduced than $\mathcal{D}$, for the variables $\mathcal{X}$. $\mathcal{C}'$ is a new set of constraints, where some constraints in $\mathcal{C}$ are strengthened, others are removed because they become redundant, and others are new. Ideally, one wants to reduce the domain of each variable $x \in \mathcal{X}$ to just one single value, which would return a solution to the CSP.

Most CSP solvers use some kind of constraint propagation before solving. Unlike CSP solving, constraint propagation usually returns an incomplete evaluation of values to variables, i.e., $|\mathcal{X}_{ev}| \le |\mathcal{X}|$. Rather than solving the CSP completely, constraint propagation reduces the domain of some (no necessarily all) variables to just one value.

If $\mathcal{X}_{prop} \subseteq \mathcal{X}$ is the set of variables with their domains reduced to one value via propagation, any possible solution to the CSP will only need to assign values to the variables in $\mathcal{X} \setminus \mathcal{X}_{prop}$. $\mathcal{X}_{prop}$ is likely to be incomplete, but the values of $\mathcal{X}_{prop}$ will be correct for any possible solution to the CSP.

2.3. Learning from multiple plans. Learning variants

Let us consider a planning domain $\delta = \langle \mathcal{T}, \mathcal{P}, \mathcal{O} \rangle$ and a set of empty operators $\mathcal{O}?$. The sets $\mathcal{O}$ and $\mathcal{O}?$ are equal, but by empty we mean that the preconditions, effects and costs are completely unknown in $\mathcal{O}?$. The name (or a unique identifier) and parameters of operators in $\mathcal{O}?$ must be known. This assumption is minimally necessary to identify common structures and, consequently, very frequent in literature. The parameters are needed to automatically generate a set of candidate predicates that can be potentially learned in every operator. For example, `board(?p - person ?a - aircraft ?c - city){}` and `refuel(?a - aircraft ?c - city ?l1 - flevel ?l2 - flevel){}` are empty operators for the operators of Fig. 1. Let us also consider a set of n problems for δ, where each $\rho_i = \langle \delta, \mathcal{V}_i, \mathcal{I}_i, \mathcal{G}_i, \mathcal{A}_i \rangle$, and their corresponding observed plans $\Pi_n(\delta) = \{\pi(\rho_1) \ldots \pi(\rho_n)\}$, where each $\pi(\rho_i)$ is as given in Fig. 2. For simplicity in the notation, each plan $\pi(\rho_i)$ in the library is extended with its $\mathcal{I}_i$ and $\mathcal{G}_i$, retrieved from ρ_i. The plans use the domain δ, so their actions are always consistent. This means the plans are non-conflicting and they do not induce an over-constrained problem.

The learning task is formalized from a collection of plans $\Pi_n(\delta)$ as the tuple $\mathcal{L}_n = \langle \delta, \Pi_n(\delta), \mu(\delta), \mathcal{O}? \rangle$, where the mutex information given in $\mu(\delta)$ can be optionally empty.

In the literature, a learning task $\mathcal{L}_n$ presents different variants *w.r.t.* the information on the domain, problems and plans:

1. Given a domain $\delta = \langle \mathcal{T}, \mathcal{P}, \mathcal{O} \rangle$, the predicates in $\mathcal{P}$ can contain, or not, static information. A domain without static predicates has a positive impact in learning. Similarly, the learning task benefits from having the mutex information in $\mu(\delta)$.
2. Given a problem $\rho_i = \langle \delta, \mathcal{V}_i, \mathcal{I}_i, \mathcal{G}_i, \mathcal{A}_i \rangle$, $\mathcal{I}_i$ is a full state, but $\mathcal{G}_i$ can be a partial ($|\mathcal{I}_i| < |\mathcal{G}_i|$) or a full state ($|\mathcal{I}_i| = |\mathcal{G}_i|$). Having $\mathcal{G}_i$ defined as a full state is more informative for learning.
3. The collection of plans in $\Pi_n(\delta)$ has an arbitrary size, from one to thousands of plans. Plans in $\Pi_n(\delta)$ can be sequential or parallel, i.e., with concurrent actions. The actions in each $\pi(\rho_i)$ can be fully known or not, and the sequence of intermediate states between actions (*aka* state trajectory [12]) can be observed, or not, as part of the plan trace. Having a full knowledge on actions and states is positive for learning. Additionally, both actions and states can be observed with or without noise. From the action perspective, noise implies: i) uncertainty on the exact time when an action happens (the one supported in this work), ii) a missing action that is not observed, or ii) an action that is observed when it should not. From the state perspective, noise implies that intermediate states are observed with errors. Obviously, noiseless observations are preferred for learning.

These variants lead to different learning approaches. The main advantage of the proposed approach is that it unifies 1, 2 and partly 3, as discussed in Section 2.5.

A solution for the learning task $\mathcal{L}_n$, denoted as $\mathcal{O}(\mathcal{L}_n)$, is an approximation in $\mathcal{O}?$ to the original operators in $\mathcal{O}$, used as the reference model. It is an approximation because the learned operators are not always identical to the original ones, as some preconditions/effects/costs can be missing. $\mathcal{O}(\mathcal{L}_n)$ must satisfy all plans in $\Pi_n(\delta)$ (completeness) and they must imply no contradictions nor mutexes in the states induced by their executions (soundness). Formally, $\mathcal{O}(\mathcal{L}_n)$ must be consistent with the information given in $\mathcal{L}_n$. This means that if actions $\mathcal{A}_i$ in every $\pi(\rho_i)$ are instantiated again according to the preconditions + effects+ cost of operators in $\mathcal{O}(\mathcal{L}_n)$, all plans in $\Pi_n(\delta)$ are consistent: i) $\mathcal{G}_i$ is achieved from $\mathcal{I}_i$; ii) if observed, the sequence of intermediate states between actions encompasses the original state trajectories in every $\pi(\rho_i)$; and iii) the observed cost of $\pi(\rho_i)$ is consistent with the learned costs. Intuitively, all actions could be instantiated from the operators $\mathcal{O}(\mathcal{L}_n)$, and $\Pi_n(\delta)$ would remain consistent.

Fig. 3 shows a possible $\mathcal{O}(\mathcal{L}_n)$ that approximates the original operators of Fig. 1. As can be seen, the preconditions, effects and cost that appear are all correct. However, the precondition `(at ?a ?c)` is not learned in `board` nor in `refuel`. The cost is not learned in `board`, but it is learned in `refuel`. Therefore, the learned model is incomplete but 100% precise.

2.4. Popular metrics used in learning

A learning task $\mathcal{L}_n$ must fill in the operators in $\mathcal{O}?$ by reasoning over the observations of a collection of plans $\Pi(\delta)$. Consequently, learning can be considered as an automated design task to create a new model $\mathcal{O}(\mathcal{L}_n)$ that is syntactically similar to the reference

```
(:action board
 :parameters (?p - person ?a - aircraft ?c - city)
 :precondition (and (at ?p ?c))
 :effect (and (not (at ?p ?c)) (in ?p ?a)))

(:action refuel
 :parameters (?a - aircraft ?c - city ?l1 - flevel ?l2 - flevel)
 :precondition (and (fuel-level ?a ?l1))
 :effect (and (not (fuel-level ?a ?l1)) (fuel-level ?a ?l2)
               (increase (cost) 2)))
```

Fig. 3. Example of two operators learned for *zenotravel*.

model $\mathcal{O}$. The common metrics used in learning action models are inspired by those used in pattern recognition and classification [27], and keep their notation. Let *TP, FP, TN, FN* be true positive, false positive, true negative, and false negative, respectively, in $\mathcal{O}(\mathcal{L}_n)$ calculated over a number *N* of preconditions and effects in $\mathcal{O}$. For each operator, *TP* stands for a precondition/effect that is in both $\mathcal{O}(\mathcal{L}_n)$ and $\mathcal{O}$, whereas a *TN* is not present in any of them. In terms of *TP* and *TN*, $\mathcal{O}(\mathcal{L}_n)$ and $\mathcal{O}$ are syntactically identical. *FP* stands for a precondition/effect that is in $\mathcal{O}(\mathcal{L}_n)$ but not in $\mathcal{O}$, whereas a *FN* is in $\mathcal{O}$ but not in $\mathcal{O}(\mathcal{L}_n)$. In terms of *FP* and *FN*, $\mathcal{O}(\mathcal{L}_n)$ and $\mathcal{O}$ show differences. From the most basic metrics to the most elaborate ones they are:

- Missing information and error counting, defined as *FN*. Intuitively, one count of error occurs if an action precondition or effect is not learned when it should.
- Accuracy and error rate. Accuracy is defined as *(TP + TN)/N*, while error rate is defined as *(FP+ FN)/N* (note that *Error rate = 1-Accuracy*). Accuracy provides relative indicators over the total number of preconditions or effects in $\mathcal{O}$, and error rate the opposite.
- Precision (*aka* positive predictive value) and recall (*aka* sensitivity). Precision is the fraction of relevant instances among the learned ones, and provides a notion of how error-free and sound $\mathcal{O}(\mathcal{L}_n)$ is. It is defined as *TP/(TP + FP)*. Precision = 1 means no *FP*. Recall is the fraction of the total amount of relevant instances that are learned, and provides a notion of how complete $\mathcal{O}(\mathcal{L}_n)$ is. It is defined as *TP/(TP+ FN)*. Recall = 1 means no *FN*. In perfect learning, Precision = Recall = 1, which implies no *FP/FN*.
- F_1 score. It is the harmonic mean of the precision and recall, which analyzes equally both values in one metric. It is defined as *2·Precision·Recall/(Precision + Recall)*. In perfect learning $F_1 = 1$.

2.5. Related work

Classification tasks have been addressed by learning and machine learning. Particularly, they have been solved by natural language inference, recurrent and convolutional neural networks, Markov networks, deep learning models, etc. [27]. These approaches use regular expressions, rules and grammars to identify and classify entities into pre-defined categories from thousands of observations. Learning action models in planning is a type of classification that aims at identifying common structures (mainly preconditions and effects) that are consistent with the input observations. Formally, these observations induce a set of constraints the learned models must satisfy. Therefore, CP paradigms are, a priori, a good starting point for investigation.

2.5.1. Use of CP in constraint learning

A major bottleneck in the use and resolution of CP, and particularly CSPs, is modeling, so constraint acquisition has been studied in the last decades as a type of learning. Such acquisition is an interactive process between the user and the machine that consists in learning a constraint network from solutions, and non-solutions, of the problem to be formulated. Similarly, inductive logic programming [36,37] yields model-theories by constructing first-order clausal theories from examples of solutions and environment knowledge. In [38], authors use event calculus to learn action models from positive and negative solutions. In all these cases, information about (state) changes and some degree of interactivity with the user are required.

CONACQ [39] is an approach that is capable of learning a constraint network from a set of traces and a library of constraints, and has been applied to learn robot behavior in the form of actions [40]. It uses an ad-hoc CP formulation, which relies on preconditions + effects verification and requires sequential positive+ negative plan traces. Although used for learning, it is more related to planning, as the solution represents one possible execution, of many, of the sequence of actions that then might need to be manually adjusted.

ConstraintSeeker [41] and ModelSeeker [42] are also related to constraint acquisition, as they exploit regular structures of constraint problems to find common patterns that apply to group elements. ConstraintSeeker uses the input samples to return a ranked list of matching constraints over a constraint catalog that are presented to normal users, while ModelSeeker generalizes models from positive + negative traces and suggests potential solutions. Both approaches may require interaction with a user, and focus more on categorization than on real learning.

QuAcq [43] learns constraint networks by asking the user to classify partial queries, that is, examples the user needs to classify as solutions or non-solutions. MultiAcq [44] is an extension of QuAcq to make the acquisition process more efficient. In particular, MultiAcq tries to learn more than one constraint (explanation) by query.

Table 1
Learning variants of approaches in recent literature (in chronological order).

Name	Mutex info	$\mathcal{G}_i$	Intermediate information	Noisy observations	Cost learning
ARMS	unsup	partial	acts req states sup	unsup	unsup
LAMP	unsup	partial	acts req states sup	unsup	unsup
AMAN	unsup	partial full	acts req states unsup	mistaken acts	unsup
RIM	unsup	partial	acts req states unsup	unsup	unsup
LOCM	unsup	partial	acts req states unsup	unsup	unsup
LOCM2	unsup	partial	acts req states unsup	unsup	unsup
ASCoL	unsup	partial	acts req states unsup	unsup	unsup
NLOCM	unsup	partial	acts req states unsup	unsup	sup
LOUGA	unsup	partial full	acts req states sup	unsup	unsup
FAMA	unsup	full	acts sup states sup	unsup	unsup
Cube-Space AE	unsup	full	acts unsup states req	noisy states	unsup
FOSR	unsup	-	state space	unsup	unsup
FLGP	unsup	-	state space	unsup	unsup
SAM	unsup	full	acts req states req	unsup	unsup
PAL	unsup	partial full	acts unsup states req	unsup	unsup
LBCRL	unsup	-	state space	unsup	unsup
LPDVR	unsup	full	acts unsup states unsup	unsup	unsup
LvCP	sup	partial full	acts req states sup	start times of acts	sup

All in all, constraint acquisition and inductive logic programming alleviate the significant expertise required for CP modeling. Having many constraints allows for good models, but they are not 100% precise. Although there are several approaches, which even make use of heuristics to accelerate convergence, the main drawback is that they require a number of queries that could be too large for a human user. Also, they require a strong interaction and common knowledge to communicate between the user and the machine.

From a planning perspective, there are three important differences between constraint acquisition/induction and learning action models. First, the underlying constraint model in planning is known as it mainly relies on causal links. The constraints do not need to be acquired, but satisfied. Second, the solution of constraint acquisition/induction is closer to a set of constraints rather than to a real model of actions. Informally, the solution found is similar to the CP formulation proposed in this work (see Section 3). Third, positive and negative traces are needed in acquisition/induction, which in planning is uncommon (negative plans in planning are useless). There is, however, an important similarity between them: they both provide an interesting approach to learn via constraint reasoning. This means that their learning is knowledge-based, rather than statistics-based.

2.5.2. *Learning action models*

Learning classical action models in recent literature has been addressed by different approaches, which do not exploit CP sufficiently. A comprehensive description and classification of them is presented in [7,45]. Therefore, the relevant works are organized according to the learning variants and the metrics presented in Sections 2.3 and 2.4, which are shown in Tables 1 and 2, respectively.

As a summary, Table 1 depicts whether: i) knowledge on mutex information $\mu(\delta)$ is (un)supported; ii) $\mathcal{G}_i$ can be a partial or a full state; iii) intermediate information is required (mandatory), supported (optional) or unsupported for actions and states; iv) noisy information is (un)supported; and v) learning action costs is (un)supported. Table 2 depicts: i) the metric that is used for evaluating the learned model, in some cases individualized for preconditions and effects and in others for the entire model without distinction; ii) if the precision of the learned model can be guaranteed; iii) the dataset size (*w.r.t.* the number of plans and actions); and iv) the number of tested domains.

One of the most incipient works in planning learning was [46], as a way to improve partially known action models by using intermediate states in plans. ARMS [6] was a pioneer approach to learn action models, without cost, from scratch. In particular, the

Table 2
Evaluation and metrics of the learning task in recent approaches. “-” means that this information is not provided in the paper.

Name	Metric used	Precision guaranteed?	Dataset size (plans & acts)	Domains
ARMS	pre: error counting eff: redundancy	no	160 plans 1600-4320 acts	6
LAMP	pre: error rate eff: error rate	no	100-200 plans 1300-6200 acts	4
AMAN	pre: error rate eff: error rate	no	40-200 plans -	3
RIM	model accuracy	no	30-150 plans -	3
LOCM	model adequacy (similar to accuracy)	no	-	5
LOCM2	-	no	-	10
ASCoL	false positive/negative of static relations	no	1-24 plans 8-360 acts	15
NLOCM	error counting in operators cost	no	100-1000 plans 1000-10000 acts	40
LOUGA	pre: error rate eff: error rate	no	160 plans 800-3200 acts	5
FAMA	pre: precision + recall eff: precision+ recall	no	1-10 plans 10-100 acts	15
Cube-Space AE	model accuracy	no	360 plans 30 acts	6
FOSR	validation test (similar to accuracy)	no	-	4
FLGP	validation test (similar to accuracy)	no	-	6
SAM	model safety	no	3-9 plans -	12
PAL	-	no	112 plans -	3
LBCRL	-	no	-	4
LPDVR	model differences (similar to accuracy)	no	-	8
LvCP	pre: precision + recall+ F_1 eff: precision+ recall+ F_1 cost: precision+ recall+ F_1	yes	10-50 plans 30-3250 acts	22

action model is initially unknown; it only requires the sequence of actions as noiseless plans, and the information of observable states can be omitted. ARMS uncovers a number of constraints, as a set of clauses, from the actions of the input plans to confine the space of learned models. Each clause is associated with a weight representing the priority in satisfying the constraint, thus building a weighted propositional MAX-SATisfiability problem. As an optimization problem, it tries to minimize the error counting and redundancy rates which returns models that are not 100% precise (see Table 2).

LAMP [11] supports more expressiveness than ARMS and learns action models with quantifiers and logical implications from observations of plans and partial intermediate states. LAMP encodes propositional candidate formulas that are passed to a Markov Logic Network for selecting the most likely subsets of candidates, which are finally transformed into learned action models. These models summarize the observed plans as much as possible, but they can be imprecise.

AMAN [26] was the first successful approach to learn action models where observed actions are allowed to be with noise (observation of state information is unsupported). This means that some actions in a plan may be mistakenly observed, giving birth to uncertainty. AMAN generates a set of domain models, builds a probabilistic graphical model to capture the physics of the domain and finally learns the model that best explains the observed plans. Again, the learned model can be partially imprecise.

Similarly to ARMS, RIM [47] uses a MAX-SAT approach, where the constraints are derived from the plan traces, as well as the preconditions + effects from an incomplete action model. This allows RIM to learn a refined model of actions and macro actions that maximize the model accuracy, although it cannot guarantee 100% of precision.

LOCM [48] started a family of inductive learning systems that use Finite State Machines to collect the transitions in the plan traces. LOCM induces action models without the need of initial, goal or intermediate states. LOCM2 [49] generalizes the domain representation of LOCM to allow multiple state machines, with each machine characterized by a set of transitions. This allows LOCM2 to deal with a higher coverage of domains where action models can be learned, but no metrics for adequate comparisons are used in its evaluation [49]. Following the path of LOCM and LOCM2, ASCoL [50] does not learn an action model in its own, but it exploits graph analysis to automatically identify static relations missed in already acquired action models. NLOCM [51] is a Numeric extension of LOCM that extends the finite state automata of LOCM to identify the structural elements and deal with numeric weights on the transitions. NLOCM uses LOCM2 as a previous step to learn the action model, and then uses CP to learn action costs from plan traces (with only the final cost of the plans as extra information). Basically, it encodes a linear constraint, such that the sum of the costs of the individual actions equals the cost of the entire plan, as in the proposed approach. NLOCM also considers the operator parameters that are important for incurring cost, which is beyond the work of this paper. NLOCM is the first, and only approach in recent literature, that supports cost learning. However, it does not support noisy observations, nor guarantees perfect precision and requires huge datasets.

LOUGA [24] combines an ad-hoc and a genetic algorithm to learn preconditions and effects, respectively, from plan traces where the goals are not strictly necessary. LOUGA supports partial intermediate states. For the effects, an individual is encoded where each gene represents whether a predicate is part of the positive, negative effects, or none of them. The preconditions are learned by using a more efficient and specific algorithm. According to [24], LOUGA shows better results than ARMS in a few domains, but it cannot guarantee perfect precision.

FAMA [12] automatically compiles the task of learning actions into a planning task which is solved by a planner. It supports incomplete, or empty, plan traces and information on intermediate states that can be partial, which reduces the required observability to the minimum. However, it requires both an initial and final (goal) full state. FAMA provides better values for precision and recall than ARMS, but it cannot guarantee 100% of precision.

Cube-Space AE [52] proposes the Cube-Space AutoEncoder neuro-symbolic architecture, which is trained to produce an effective discrete state transition model to learn symbolic representations of an action model. It uses neural networks to extract the action effects from (probably) noisy image-based observations of the environment, which means the domains need to be image-based; it requires both the initial and goal full state as raw visual inputs. It is mainly focused on learning effects and, according to the authors, the precondition learning needs to be improved. It cannot ensure perfect precision.

A SAT formulation to learn First-Order Symbolic Representations from non-symbolic data, which is denoted here as FOSR, is introduced in [8]. It emphasizes a different idea to learn from state graphs rather than from plan traces. In particular, the objective is to find the simplest model (first-order representations structured in terms of objects and relations) which explains the structure of the input graphs that capture the state space. The main advantage is that it does not assume knowledge of the operators or predicates, as they are all learned from the input. However, the resulting domain representation is only correct for the observed problems, and not necessarily for others, which means perfect precision is not guaranteed. In [28], a Formulation to Learn General Policies (or models), which is denoted here as FLGP, is presented. Similarly to [8], the learning scheme uses state spaces, in terms of sample instances, which do not require their plans. The idea is to learn policies mapped from the state transitions. This formulation is represented as an optimization task and solved via Weighted Max-SAT. It cannot guarantee 100% of precision.

TempAMLSI [9] is a Temporal Action Model Learning approach based on grammar induction, which learns a deterministic finite automaton corresponding to the regular grammar generating the action traces. Unlike most learning approaches, it requires positive + negative traces to induce the preconditions and effects of the actions; it learns an intermediate classical domain and then converts it into a temporal domain. TempAMLSI supports a high level of action concurrency for temporal planning, beyond the scope of this paper, so it is not included in the Tables 1 and 2, which restrict to classical planning.

SAM [30] is a Safe Action Model learning approach that deals with reliability in terms of model safety. It learns an action model that is sound *w.r.t.* all the observations and guarantees the actions achieve the intended effects to meet the goals, like in the proposed approach. SAM does not aim at learning a 100% precise model, but one that is sufficient for finding a safe plan for all the observations. Additionally, it requires plan traces that include not only all the actions but also all the intermediate states, which means a high level of observability.

PAL [53] interleaves planning, acting and learning using an initial (possibly incomplete and incorrect) draft planning domain and a sequence of intermediate states. It uses a different approach for learning, which focuses on finding an abstraction of the continuous environment, with continuous states, into a finite set of states that can scale up to large state spaces. However, it cannot guarantee perfect precision.

The thread initiated in [8] is continued in recent works. First, in [29], with Language-Based Causal Representation Learning, which is denoted as LBCRL. The internal structure of the states is recovered from state graphs, where the states are black-boxes with unknown structure. It learns the representations over a first-order causal domain-independent language, like PDDL, with a known structure (syntax) and semantics. The representation is expressed and solved using a SAT solver. The authors state their approach is evaluated on four domains, but no additional results are provided. The idea of learning causality is very useful in other works, particularly in those for industrial automation [14], to describe the behavior of the models mathematically. Second, [54] presents a formulation to Learn Planning Domains on Visual Representations, denoted here as LPDVR, representing the initial and goal states. The idea is to learn the domain along with the grounding of the (learned) predicates so that they can be evaluated on any parsed image, which requires vision modules for object recognition to map images into the parsed representations. In all these woks, precision cannot be ensured.

The proposed approach is based on the works in [13,35]. It uses a similar CP formulation to encode preconditions and effects. The model learned in those works is for temporal planning rather than for classical planning, so they are not included in the comparative Tables 1 and 2. In [13], only one plan is observed for learning (one-shot learning), which is extended in [35] with more plans. Both works support mutex information, but they do not support noisy observations on the start time of the actions, nor guarantee perfect precision learning, which are key differences. The distinctive features of the proposed approach, named LvCP (Learning via Constraint Propagation) in the Tables, *w.r.t.* other works are its flexibility and coverage. First, it deals with, and without, static information and supports mutex information. Second, the problem goals $\mathcal{G}_i$ can be defined as partial or full states. Third, the plans can include both parallel and sequential actions. A partial or full sequence of intermediate states is unnecessary, but can be supported

Variables: $x_1, x_2, x_3, x_4, x_5 \in \{1..4\}$
Reference model to be learned: $\{x_1 = 1, x_2 = 2, x_3 = 3, x_4 = 4, x_5 = 4\}$
Constraint1: $x_2 > 1$
Constraint2: $x_1 + x_2 = 3$
Constraint3: $x_3 + x_4 = 7$

Fig. 4. CSP solving *vs.* constraint propagation in learning. A motivating example.

as part of the plan observations as well. Fourth, noise is supported in the form of uncertainty on the timestamps of actions, which means a more robust model. Fifth, action costs can be learned. Sixth, it learns an action model that meets all the observations, which means a safe action model like in SAM. Seventh, the constraint propagation approach ensures perfect precision, which is a novel feature that differs from all existing approaches. Additionally, the experiments also evaluate the recall and F_1 score, individualized for preconditions, effects and cost.

2.5.3. *CSP vs. SAT*

Constraint satisfaction and propositional SATisfiability are closely related frameworks [55]. Currently, SAT solvers are very efficient and some approaches use SAT in one way or another, such as CONACQ, MultiAcq, ARMS, RIM, FOSR, FLGP or LBCRL. From the modeling point of view, CSP formulations can be mapped into SAT ones and vice versa. However, non-boolean variables make the SAT encoding more tedious. On the contrary, there is no need of clauses to ensure that a CSP variable is given a value, nor to ensure that each CSP variable is given only one value [55]. Hence, the CSP encoding is easier. This is the first reason why using a CSP formulation is more appealing in this work.

From the propagation point of view, in many cases there is a direct correspondence between a propagation method for CSP and another one for SAT [56]. But it has been proved that different encodings can have a serious impact on the level of consistency achieved [55]. First, translating a CSP to SAT can be beneficial only if some particular encodings are used. Second, propagation on SAT encodings does less work than achieving some levels of consistency on the original CSP. The interest in propagation is the second reason to use a CSP formulation rather than a SAT one. Note, therefore, that one positive consequence is that most of the constraints of the formulation in Section 3 can be turned into SAT clauses and managed by modern SAT solvers.

3. CP formulation

3.1. *Motivating example. Ensuring perfect precision via constraint propagation*

In order to show how constraint propagation guarantees perfect precision, the example in Fig. 4 is used. There are five integer variables $x_1 \dots x_5$ whose values need to be learned, provided some observations that impose the three constraints depicted. Rather than learning an action model from plan traces, it learns here the values of the variables from the constraints observed; but the underlying idea is the same. The reference model to learn is $\{x_1 = 1, x_2 = 2, x_3 = 3, x_4 = 4, x_5 = 4\}$. Any value learned that does not match with the reference value will be imprecise. Obviously, any current or future observation must be consistent with that model because there are no conflicting observations.

When solving the CSP, a possible solution/model is $sol_1 = \{x_1 = 1, x_2 = 2, x_3 = 4, x_4 = 3, x_5 = 1\}$. But in fact, there are other possible solutions, e.g., $sol_2 = \{x_1 = 1, x_2 = 2, x_3 = 4, x_4 = 3, x_5 = 2\}$, $sol_3 = \{x_1 = 1, x_2 = 2, x_3 = 3, x_4 = 4, x_5 = 1\}$, etc. These solutions are imprecise. Since every solution gives value to all variables, none of the three solutions matches without errors the reference model, because x_5 is learned incorrectly. Moreover, these solutions are only valid for the current set of constraints. Given a new "Constraint4: $x_1 + x_5 > 4$", these solutions will lead to a contradiction or inconsistency, which is highly undesirable: if one learns something, it must be correct not only for the current observations, but also for any future observation.

Let us now consider a constraint propagation approach. From Fig. 4, the iterative process runs over the values of the domains and posts constraints to check the (path) consistency of the model. If constraint $x_1 \neq 1$ is posted, the propagation detects a contradiction. This means that $x_1 = 1$ in every possible solution. Then, it tries x_2 and posts $x_2 \neq 1$, but the propagation does not find an inconsistency. After unposting that constraint, it posts $x_2 \neq 2$ and the propagation detects an inconsistency. Therefore, $x_2 = 2$ in every solution. Finally, it tries $\{1..4\}$-values with x_3, x_4 and x_5, but no inconsistencies are detected. This means these three variables are still open and not learned.

Note that solving this CSP finds solutions that assign values to all five variables, but they introduce imprecision in form of errors. On the contrary, propagation only returns $\{x_1 = 1, x_2 = 2\}$, which is incomplete but 100% precise. Constraint propagation makes no assumption about the part of the solution in which there is insufficient information to learn with perfect precision. Moreover, $\{x_1 = 1, x_2 = 2\}$ do not need to be rechecked for new observations. If Constraint4 is also given, the values for x_1 and x_2 are still correct. After posting + propagating $x_5 \neq 4$, an inconsistency is detected, which means that $x_5 = 4$. Consequently, the propagation finds the evaluation $\{x_1 = 1, x_2 = 2, x_5 = 4\}$ which matches perfectly the reference model for the three variables. As can be seen, it learns less but better. If future constraints for this model are given, everything that has been assigned/learned remains correct and the model keeps its robustness.

The computational complexity of the process of successive propagation steps depends on the domain size of the unassigned variables and the complexity of the propagation in itself. The propagation steps are run several times, but it is important to note that each propagation has polynomial complexity. On the contrary, solving has a non-polynomial complexity.

Note that if the model has only one solution, the result of solving subsumes that of propagation. But when several solutions are possible, which is very common when the number of observations is limited, assigning values to $\mathcal{X} \setminus \mathcal{X}_{prop}$ variables is imprecise

Table 3
Formulation of variables. Without loss of generality, domains of X1 and X4 are modeled in $\mathbb{Z}^+$.

Id.	Variable	Domain	Description
X1	$\mathsf{cost}(o)$	$\mathbb{Z}^+ \cup \{0\}$	cost of o
X2	$\mathsf{pre}(p, o)$	$\{false, pre\}$	symbolic var with the type of precondition p
X3	$\mathtt{eff}(p, o)$	$\{false, add, del\}$	symbolic var with the type of effect p
X4	$\mathsf{start}(o_\pi)$	$\mathbb{Z}^+ \cup \{0\}$	start time of o_π
X5	$\mathsf{sup}(p_\pi, o_\pi)$	$\emptyset \cup \{o'_\pi\} \in \pi$	supporters for causal link $\langle o'_\pi, p_\pi, o_\pi \rangle$

and risky. On the contrary, when reliability and precision are important, constraint propagation is an appealing way to learn, and addressing a learning task $\mathcal{L}_n$ via constraint propagation is very promising.

3.2. Preprocessing

The CP formulation represents a learning task $\mathcal{L}_n = \langle \delta, \Pi_n(\delta), \mu(\delta), \mathcal{O}? \rangle$, as defined in Section 2.3, which encodes the preconditions, effects and cost of each operator, and the actions of the observed plans. In order to learn the preconditions + effects of an operator, the approach first needs to populate a potential set of candidate predicates.

Given the domain δ and the set $\mathcal{O}?$, this approach defines the alphabet of $o \in \mathcal{O}?$, denoted as $\alpha(o)$, as the set of all predicates $p \in \mathcal{P}$ whose types belong to the types of the parameters of o. For instance, according to the types, predicates and parameters of `board` in Fig. 1, α(`board`)={`(at ?p - person ?c - city)`, `(at ?a - aircraft ?c - city)`, `(in ?p - person ?a - aircraft)`}.

The candidates of an operator o are defined as the two-set tuple $\gamma(o) = \langle \{p_i\}_{\mathsf{pre}}, \{p_i\}_{\mathsf{eff}} \rangle$, where $p_i \in \alpha(o)$. $\{p_i\}_{\mathsf{pre}}$ denotes all candidates that can be preconditions of o, whereas $\{p_i\}_{\mathsf{eff}}$ denotes all candidates that can be either positive or negative effects of o, identified as $\mathtt{eff}^+(o)$ or $\mathtt{eff}^-(o)$ respectively; $|\gamma(o)| = 2 \cdot |\alpha(o)|$. Note that $|\gamma(o)|$ represents the number of candidates of o, no the number of possible combinations of subsets of preconditions + effects, which is exponential.

In this preprocessing step, a mapping between the operators in the domain and the actions in a plan is created. For each $\pi \in \Pi_n(\delta)$, let o_π be an action in π that is the result of grounding an operator $o \in \mathcal{O}?$. Let $p \in \alpha(o)$ be a predicate of o, which can be used as a precondition/effect, that is instantiated for an action o_π, and denoted as p_π. For instance, given the operator o= `(board ?p - person ?a - aircraft ?c - city)` and the predicate p= `(at ?p - person ?c - city)`, it creates a mapping with a possible action o_π = `(board person1 plane1 city0)` and p_π = `(at person1 city0)`. The underlying idea of this mapping is to simplify the CP formulation: if p is learned as a precondition of o, any p_π and o_π must be consistent with such decision in π. For instance, if `(at ?p - person ?c - city)` is learned as a precondition of `(board ?p - person ?a - aircraft ?c - city)`, then `(at person1 city0)` will be a precondition of `(board person1 plane1 city0)`; and analogously for another action o_π = `(board person2 plane2 city0)` and its precondition `(at person2 city0)`.

3.3. Variables

The decision variables are inspired by [35], but the current model is more compact. They are organized in two blocks, as depicted in Table 3. The former are the variables for each $o \in \mathcal{O}?$ and $p \in \alpha(o)$ (X1..X3). The latter are the variables for o_π and p_π, which must be repeated for every plan $\pi \in \Pi_n(\delta)$ (X4..X5).

X1 models the cost of the operator.[4] X2 represents whether p is, or not, a precondition in o, labeled as *pre* and *false*, respectively. X3 represents if p is learned as a positive or negative effect in o (*add* or *del*, respectively); *false* means that p is not an effect. X4 is the start time of action o_π; its value can be known (noiseless observations) or unknown (noisy observations with uncertainty on the start time). X5 represents the cause-effect relationship, *aka* causal link: action o'_π supports p_π, which is required by o_π. If $\mathsf{pre}(p, o) = false$, meaning that p is not a precondition of o (and, consequently, p_π is not a precondition of o_π), then $\mathsf{sup}(p_\pi, o_\pi) = \emptyset$, which denotes an empty supporter.

The number of variables of the formulation is polynomial in the number of predicates and actions in the plans. Let U_α be the upper bound on the size of the alphabets of the operators. Let n be the number of input plans in $\mathcal{L}_n$, and U_π be the upper bound on the number of actions in these plans. The number of variables is bounded by $O(U_\alpha \cdot n \cdot U_\pi)$.

In addition to actions in $\pi \in \Pi_n(\delta)$, where $\Pi_n(\delta) = \{\pi(\rho_1) \ldots \pi(\rho_n)\}$, two dummy actions[5] are created per problem $\rho \in \{\rho_1 \ldots \rho_n\}$, where $\rho = \langle \delta, \mathcal{V}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$. First, init_ρ represents $\mathcal{I}$ ($\mathsf{cost}(\mathsf{init}_\rho) = \mathsf{start}(\mathsf{init}_\rho) = 0$). For init_ρ it does not create pre and sup variables because it has no preconditions. init_ρ has as many $\mathtt{eff}(p_i, \mathsf{init}_\rho) = add$ as $p_i \in \mathcal{I}$ and $\mathtt{eff}(p_j, \mathsf{init}_\rho) = del$ for the false predicates $\neg p_j \in \mathcal{I}$, as $\mathcal{I}$ is a full state. Second, goal_ρ represents $\mathcal{G}$ ($\mathsf{cost}(\mathsf{goal}_\rho) = 0$ and $\mathsf{start}(\mathsf{goal}_\rho) = length(\pi)$, with the plan length of π). goal_ρ has as many $\mathsf{pre}(p_i, \mathsf{goal}_\rho) = pre$ as $p_i \in \mathcal{G}$, thus supporting both a partial or a full goal state. goal_ρ has no $\mathtt{eff}$ variables because it has no effects.

[4] The cost is defined per operator. If the cost really depends on the parameters, different per instantiated action of the same operator, we would need to create a dummy operator per action, with the same preconditions + effects but different cost. This would be equivalent to have a domain with more operators but with the same structure.

[5] Dummy actions are mapped to dummy operators with the same name.

Table 4
Formulation of planning and mutex constraints.

Id.	Constraint
C1	**if** $(\mathtt{eff}(p,o)=del)$ **then** $\mathsf{pre}(p,o)=pre$
C2	**if** $(\mathsf{pre}(p,o)=pre)$ **then** $\mathtt{eff}(p,o)\neq add$
C3	$\forall\ o$: $(\exists\ \mathsf{pre}(p,o)\neq false)$ **and** $(\exists\ \mathtt{eff}(p,o)\neq false)$
C4	**iff** $(\mathsf{pre}(p,o)=false)$ **then** $\mathsf{sup}(p_\pi,o_\pi)=\emptyset$
C5	**if** $(\mathtt{eff}(p,o)=false)$ **then** $\forall\ o'_\pi$ that requires p_π: $\mathsf{sup}(p_\pi,o'_\pi)\neq o_\pi$
C6	**if** $(\mathtt{eff}(p,o)=add)$ **and** $(\mathtt{eff}(p,o')=del)$ **then** $\mathsf{start}(o_\pi)\neq\mathsf{start}(o'_\pi)$
C7	**if** $(\mathsf{sup}(p_\pi,o_\pi)=o'_\pi)$ **then** $\mathsf{start}(o'_\pi)<\mathsf{start}(o_\pi)$
C8	**if** $(\mathsf{sup}(p_\pi,o_\pi)=o'_\pi)$ **and** $(\exists o^{threat}\mid\mathtt{eff}(p,o^{threat})=del)$ **and** $(o_\pi\neq o_\pi^{threat})$ **then** $(\mathsf{start}(o_\pi^{threat})<\mathsf{start}(o'_\pi))$ **or** $(\mathsf{start}(o_\pi^{threat})>\mathsf{start}(o_\pi))$
C9	$\forall\ o'_\pi$: $(\exists\ \mathsf{sup}(p_\pi,o_\pi)=o'_\pi)$
C10	$\sum occurrences(o_\pi)\cdot\mathsf{cost}(o)=\mathcal{K}_\pi$
C11	$\forall\ \langle p_i,p_j\rangle\in\mu(\delta)$:
C11.1	**not** $((\mathsf{pre}(p_i,o)=pre)$ **and** $(\mathsf{pre}(p_j,o)=pre))$
C11.2	**not** $((\mathtt{eff}(p_i,o)=add)$ **and** $(\mathtt{eff}(p_j,o)=add))$
C11.3	**if** $(\mathtt{eff}(p_i,o)=add)$ **and** $(\mathsf{pre}(p_j,o)=pre)$ **then** $\mathtt{eff}(p_j,o)=del$

The formulation of variables is both operator- and action-oriented, but including observations of intermediate partial/full states is straightforward. To do this, an observation obs needs to be defined, analogous to goal. $\mathsf{obs}(p,t)$ is a dummy action that starts at time t, with only one precondition p, which is the value observed for p ($\mathsf{pre}(p,\mathsf{obs}(p,t))=pre$, $\mathsf{sup}(p,\mathsf{obs}(p,t))\neq\emptyset$), and no eff variables at all. The formulation can include as many observations as necessary.

3.4. Constraints

Table 4 depicts the constraints, which are organized in three blocks. First, the constraints that deal only with operators (C1..C3). Second, the constraints that combine operators and actions (C4..C10). Clearly, the operator variables impose constraints that must be consistent with the actions and vice versa. Third, the constraints when mutex information is given in $\mu(\delta)$ (C11).

C1 and C2 are, respectively, necessary for the STRIPS assumption that negative effects are required as preconditions, and preconditions are not part of the positive effects. C3 ensures that every operator has at least one precondition and one effect. Although C3 might seem too restrictive, it is included to improve the learning; e.g., an operator without preconditions can start indiscriminately, whereas learning an operator without effects is of little use in a plan. Obviously, C3 could be removed if necessary. C4 is an *if-and-only-if* constraint which represents that if p is not a precondition in o, its instantiated version p_π will not be used as a supporter in o_π, and vice versa. Intuitively, no precondition means no need to be supported. Note, however, that C4 does not prevent p_π to be achieved. It only means that p is not a precondition in o and, consequently, p_π is not a precondition of o_π. Similarly in C5, if p is not an effect in o then no instantiated action o_π will be a supporter of the mapped p_π for any other action o'_π. C6 ensures that if two operators have contradictory effects, their instantiated actions cannot happen at the same time. In planning, this avoids effect interference. C7 represents the causal link $\langle o'_\pi,p_\pi,o_\pi\rangle$, which means that the supporter o'_π of p_π must happen before the requirer o_π. C8 encodes the way to solve a possible threat to the causal link $\langle o'_\pi,p_\pi,o_\pi\rangle$: if there exists a threatening operator o^{threat}, its mapped action o_π^{threat} cannot break the causal link and it must be moved forward or backward (*aka* promotion or demotion in planning). C9 encodes that every action in the plan is used as a supporter. This means that actions in plans appear due to at least one causal relationship. C9 could be removed if necessary, but it is assumed that every action in a plan is relevant. C10 encodes the linear constraint for the cost $\mathcal{K}_\pi$ of plan π, where $occurrences(o_\pi)$ is the number of instances of o_π in π.[6] Finally, C11 includes the constraints to ensure that two mutex predicates cannot hold together. C11.1 represents that two mutex predicates cannot be preconditions of the same operator. C11.2 is the analogous constraint for positive effects. Finally, C11.3 is useful to learn negative effects. Given two mutex predicates $\langle p_i,p_j\rangle$, if p_i is a positive effect, then p_j must be a negative effect, provided p_j appears as a precondition.

The number of constraints of the formulation is also polynomial in the number of predicates and actions in the plans. If U_α represents the upper bound on the size of the alphabets, n is the number of plans, and U_π is the upper bound on the number of actions, the number of constraints is bounded by C8, i.e., $\mathrm{O}(U_\alpha\cdot n\cdot U_\pi^3)$. Intuitively, U_π to the third power is because solving C8 implies dealing with three actions.

3.5. A simple working example

Let us consider the operator `board` of Fig. 1, with α(`board`)={`(at ?p - person ?c - city)`, `(at ?a - aircraft ?c - city)`, `(in ?p - person ?a - aircraft)`}. Let us also consider the two actions `(board person2 plane2 city1)` and `(board person1 plane1 city0)` of the plan of Fig. 2. According to Table 3, the variables of the formulation are depicted

[6] Note that the cost learning enforced by C10 is independent from the remaining part of the model, as it only involves the X1 variable; it could be calculated a posteriori by linear regression of the costs. Hence, although X1 + C10 could be modeled separately from the precondition+effect model, the formulation unifies all the elements to be learned.

```
cost(board)
pre((at ?p - person ?c - city),board)
pre((at ?a - aircraft ?c - city),board)
pre((in ?p - person ?a - aircraft),board)
eff((at ?p - person ?c - city),board)
eff((at ?a - aircraft ?c - city),board)
eff((in ?p - person ?a - aircraft),board)
start((board person2 plane2 city1))
sup((at person2 city1),(board person2 plane2 city1))
sup((at plane2 city1),(board person2 plane2 city1))
sup((in person2 plane2),(board person2 plane2 city1))
start((board person1 plane1 city0))
sup((at person1 city0),(board person1 plane1 city0))
sup((at plane1 city0),(board person1 plane1 city0))
sup((in person1 plane1),(board person1 plane1 city0))
```

Fig. 5. Example of CP variables for `board`.

```
if (eff((at ?p - person ?c - city),board)=del) then
  pre((at ?p - person ?c - city),board)=pre
if (eff((at ?a - aircraft ?c - city),board)=del) then
  pre((at ?a - aircraft ?c - city),board)=pre
if (eff((in ?p - person ?a - aircraft),board)=del) then
  pre((in ?p - person ?a - aircraft),board)=pre
if (pre((at ?p - person ?c - city),board)=pre) then
  eff((at ?p - person ?c - city),board)≠add
if (pre((at ?a - aircraft ?c - city),board)=pre) then
  eff((at ?a - aircraft ?c - city),board)≠add
if (pre((in ?p - person ?a - aircraft),board)=pre) then
  eff((in ?p - person ?a - aircraft),board)≠add
not ((pre((at ?p - person ?c - city),board)=pre) and
     (pre((in ?p - person ?a - aircraft),board)=pre))
not ((eff((at ?p - person ?c - city),board)=add) and
     (eff((in ?p - person ?a - aircraft),board)=add))
if (eff((at ?p - person ?c - city),board)=add) and
  (pre((in ?p - person ?a - aircraft),board)=pre) then
  eff((in ?p - person ?a - aircraft),board)=del
if (eff((in ?p - person ?a - aircraft),board)=add) and
  (pre((at ?p - person ?c - city),board)=pre) then
  eff((at ?p - person ?c - city),board)=del
```

Fig. 6. Example of CP (C1, C2 and C11.*) constraints for `board`.

in Fig. 5. The constraints are created for these variables according to Table 4, and are shown in Fig. 6. Most constraints are straightforward (e.g., the first six constraints in Fig. 6 represent C1 and C2 for `board`). C5, C8 and C9 constraints, which use universal and existential quantification, need to be programmatically unrolled in the solver by using array-based constraints.

The mutex constraints are also straightforward. Let us optionally consider that ⟨`(at ?p - person ?c - city)`, `(in ?p - person ?a - aircraft)`⟩ are mutex. According to C11.*, the last four constraints of Fig. 6 need to be formulated.

4. On the use of constraint propagation for perfect precision learning

Once the CP formulation for a learning task $\mathcal{L}_n$ is created, the typical next step is to solve it. As discussed in Section 3.1, a solving process gives value to all variables and, when several solutions are possible, this leads to imprecision. On the contrary, constraint propagation is used to ensure perfect precision on what is learned. The idea is to analyze the candidates $\gamma(o) = \langle \{p_i\}_{\text{pre}}, \{p_i\}_{\text{eff}} \rangle$ of every operator $o \in \mathcal{O}?$ to learn only those $\text{pre}(o) \in \{p_i\}_{\text{pre}}$ and $\text{eff}^+(o), \text{eff}^-(o) \in \{p_i\}_{\text{eff}}$ that guarantee 100% of precision. Similarly it is proceeded with $\text{cost}(o)$.

Initially, all $\text{pre}(o), \text{eff}^+(o), \text{eff}^-(o)$ are empty. The propagation approach follows the algorithm described in Fig. 7, which iteratively propagates constraints over the formulation created for all operators (step 1). The algorithm repeats until no further propagation occurs (loop of step 2). Steps 3–9 are for preconditions. The initial domain of $\text{pre}(p, o)$ is {*false,pre*}. If "$\text{pre}(p, o) \neq \textit{false}$" is posted and the propagation[7] detects an inconsistency, then p cannot be a precondition (detecting "$\text{pre}(p, o) = \textit{false}$" is equivalent to "$\text{pre}(p, o) \neq \textit{pre}$"). Otherwise, "$\text{pre}(p, o) \neq \textit{pre}$" is propagated to detect if p must be a precondition. Note that after learning that p must, or cannot, be a precondition, p is removed from $\{p_i\}_{\text{pre}}$ as it is no longer a candidate (steps 6 and 9). This is useful if the algorithm is applied

[7] This propagation simply invokes the function of the solver to check path consistency.

```
1: for all $o \in \mathcal{O}?$ do
2:   repeat
3:     {learning preconditions}
4:     for all $p \in \{p_i\}_{\mathsf{pre}}$ do
5:       if propagate("$\mathsf{pre}(p,o) \neq \mathit{false}$") detects an inconsistency then
6:         remove $p$ from $\{p_i\}_{\mathsf{pre}}$ {post "$\mathsf{pre}(p,o) = \mathit{false}$"}
7:       else if propagate("$\mathsf{pre}(p,o) \neq \mathit{pre}$") detects an inconsistency then
8:         add $p$ to $\mathsf{pre}(o)$ {post "$\mathsf{pre}(p,o) = \mathit{pre}$"}
9:         remove $p$ from $\{p_i\}_{\mathsf{pre}}$
10:    {learning positive and negative effects}
11:    for all $p \in \{p_i\}_{\mathsf{eff}}$ do
12:      if propagate("$\mathsf{eff}(p,o) \neq \mathit{false}$") detects an inconsistency then
13:        remove $p$ from $\{p_i\}_{\mathsf{eff}}$ {post "$\mathsf{eff}(p,o) = \mathit{false}$"}
14:      else if propagate("$\mathsf{eff}(p,o) \neq \mathit{add}$") detects an inconsistency then
15:        add $p$ to $\mathsf{eff}^+(o)$ {post "$\mathsf{eff}(p,o) = \mathit{add}$"}
16:        remove $p$ from $\{p_i\}_{\mathsf{eff}}$
17:      else if propagate("$\mathsf{eff}(p,o) \neq \mathit{del}$") detects an inconsistency then
18:        add $p$ to $\mathsf{eff}^-(o)$ {post "$\mathsf{eff}(p,o) = \mathit{del}$"}
19:        remove $p$ from $\{p_i\}_{\mathsf{eff}}$
20:  until no propagation occurs
```

Fig. 7. Algorithm for learning preconditions and effects via propagation.

```
1: {solve to find a complete evaluation $\mathsf{cost}(o) = ev(o), \forall\, o \in \mathcal{O}?$}
2: solve-CSP()
3: {learning the costs}
4: for all $o \in \mathcal{O}?$ do
5:   if propagate("$\mathsf{cost}(o) \neq ev(o)$") detects an inconsistency then
6:     $\mathsf{cost}(o) \leftarrow ev(o)$ {post "$\mathsf{cost}(o) = ev(o)$"}
7:     remove $o$ from $\mathcal{O}?$
```

Fig. 8. Algorithm for learning costs via solving + propagation.

several times, as in every new application the size of the candidates will be reduced and the loop will be shorter. Ideally, one wants to finish with an empty $\{p_i\}_{\mathsf{pre}}$, meaning that every predicate is assigned a *false* or *pre* value. However, this is not always possible as in some cases p cannot be ensured to be or not a precondition; that is, in the current formulation both options are still possible and no inconsistency is detected. Like in the motivating example of Section 3.1, some variables may remain open (not learned). Steps 10–19 are for effects, and the behavior is similar to preconditions. The initial domain of $\mathsf{eff}(p,o)$ is {*false,add,eff*}, which entails one more propagation step. Steps 12–13 learn if p cannot be an effect. Steps 14–16 learn positive effects, whereas 17–19 learns negative effects. Again, one wants to finish with an empty $\{p_i\}_{\mathsf{eff}}$, but this is not always possible.

Additionally, the approach needs to learn the cost of the operators. The underlying idea of the propagation process is similar to the one for preconditions and effects. However, there is an important difference here: the domain of $\mathsf{cost}(o)$ variables is unknown and could be potentially large. In other words, propagating constraints for all the values of the domain "$\mathsf{cost}(o) \neq 1$", "$\mathsf{cost}(o) \neq 2$", "$\mathsf{cost}(o) \neq 3$", etc. to detect inconsistencies is neither sensible nor practical. A mixed approach of solving + propagation is used, which still guarantees perfect precision.

The part of the model for learning costs is given by the X1 variables and the C10 constraints of Tables 3 and 4, respectively. Therefore, the algorithm only needs to give values (i.e., solve) to the X1 variables. Although the solving process can return an imprecise solution, the assigned value to each variable can be used for the propagation and detection of inconsistencies. Clearly, if a constraint that negates a variable-value assignment leads to an inconsistency, such a value must be assigned to that variable in every possible solution, which means that it matches the reference cost with absolute precision.

The algorithm in Fig. 8 describes the process. Step 2 solves the resulting CSP and retrieves the first solution found, which returns a possible evaluation $ev(o)$ to each $\mathsf{cost}(o)$. This evaluation is used in propagation (steps 3–7). In step 5, if "$\mathsf{cost}(o) \neq ev(o)$" is posted and the propagation detects an inconsistency, it can guarantee that $\mathsf{cost}(o) = ev(o)$. Consequently, $\mathsf{cost}(o)$ does not need to be rechecked in future applications of this algorithm (step 7). If no inconsistency is detected, $\mathsf{cost}(o)$ cannot be precisely learned yet and the value of $\mathsf{cost}(o)$ remains open.

Formal properties

Lemma 1. *Guarantee of precision. The propagation process ensures learning with perfect precision.*

Proof. Let us consider $\delta = \langle \mathcal{T}, \mathcal{P}, \mathcal{O} \rangle$, where $\mathcal{O}$ is the reference model and $\mathcal{L}_n = \langle \delta, \Pi_n(\delta), \mu(\delta), \mathcal{O}? \rangle$ is the learning task. Let us also consider an operator $o \in \mathcal{O}?$ in the formulation created for $\mathcal{L}_n$. The three decision variables are $\mathsf{cost}(o)$, and $\mathsf{pre}(p,o)/\mathsf{eff}(p,o)$ given a predicate p. The possible values to propagate are: {*value1* $= ev(o)$} for $\mathsf{cost}(o)$; {*value1* $=$ *false*, *value2* $=$ *pre*} for $\mathsf{pre}(p,o)$; and {*value1* = *false*, *value2* = *add*, *value3* = *del*} for $\texttt{eff}(p, o)$. The algorithm will consider each of these variables, denoted here simply as *var*, and its possible values {*value1, value2, value3*}, which depend on *var*.

If constraint "*var* ≠ *value1*" is posted and propagated, two situations are possible. If an inconsistency is detected, it means that at least one constraint for $\mathcal{L}_n$ is violated, and "*var* = *value1*" must hold to avoid such a violation and to satisfy all constraints in $\mathcal{L}_n$. Therefore, *value1* is learned precisely for *var* in $o \in \mathcal{O}$? and matches in the corresponding o of the reference model $\mathcal{O}$. In short, a true positive (*TP*) has been found in o. On the contrary, if no inconsistency is detected, the value for *var* could be *value1* or other, as it cannot be ensured yet. In consequence, the algorithm needs to continue and try with *value2* and, whenever necessary, with *value3*. In general, if an inconsistency is detected within this iterative algorithm, it ensures a value for *var*; otherwise, no value for *var* can be guaranteed.

Let us now consider that a new (non-conflicting) plan for $\mathcal{O}$ is observed. Now, there exists a $\Pi_{n+1}(\delta)$ and the learning task becomes $\mathcal{L}_{n+1}$. On the one hand, any new assignment found by propagation in $\mathcal{L}_{n+1}$ must satisfy $\mathcal{L}_n$, because $\mathcal{L}_n$ is a subtask of $\mathcal{L}_{n+1}$. On the other hand, any assignment that was found by previous propagation in $\mathcal{L}_n$ must satisfy $\mathcal{L}_{n+1}$, because it was learned as a matching (*TP*) in $\mathcal{O}$. In other words, the new assignments must satisfy the previous observations, and the previous assignments must satisfy the new observations. This ensures learning with perfect precision for the current observed plans ($\Pi_n(\delta)$) and any future one ($\Pi_{n+1}(\delta)$), which, indirectly, is also good for robustness. This is applicable to variables for preconditions, effects and costs. □

Lemma 2. *The computational complexity of each propagation step in Algorithm of Fig. 7 is polynomial in the number of variables of the formulation.*

Proof. The propagation step depends on the type of consistency to be guaranteed in the constraint network. Path consistency is used, with cubic complexity in the number of variables of the formulation. As indicated in Section 3.3, the number of variables of the formulation is bounded by $O(U_\alpha \cdot n \cdot U_\pi)$. Consequently, the complexity of each propagation step in Fig. 7 is bounded by $O((U_\alpha \cdot n \cdot U_\pi)^3)$. Note that the number of variables of the real formulation might be higher if the solver internally creates dummy variables to deal only with binary constraints. □

Lemma 3. *The computational complexity of each propagation step in Algorithm of Fig. 8 is polynomial in the number of X1 variables.*

Proof. The only type of variable of the formulation involved for cost propagation is X1 (see C10 in Table 4). The size of X1 is given by the number of operators $|\mathcal{O}|$. If path consistency is used, the complexity of each propagation step in Fig. 8 is bounded by $O(|\mathcal{O}|^3)$.[8] Again, the number of variables might be higher if the solver creates dummy variables. □

Note that both algorithms imply several propagation steps. But, the main advantage of these two algorithms is that they can be applied in different ways. For instance, given a learning task $\mathcal{L}_n$ with a collection of n plans, one can create a complete formulation for the n plans and run the algorithms only once, just in the end. Alternatively, one can create the formulation incrementally and run the algorithms after processing k plans. Applying the algorithms $\frac{n}{k}$ times might require more running time, but in some cases it pays off: the execution of each propagation step becomes simpler and faster, as the candidate sets $\gamma(o)$ shrink and some $\mathrm{cost}(o)$ are learned and made fixed.[9] This also reduces the complexity and number of propagation steps in future applications of the algorithms, which is an additional benefit. In this work, the algorithm for learning preconditions and effects in Fig. 7 is run after each plan is processed ($k = 1$, so n times per $\mathcal{L}_n$), and the algorithm for learning costs in Fig. 8 only one time in the end ($k = n$, so one time per $\mathcal{L}_n$).

5. Evaluation

This section evaluates the CP formulation in different IPC[10] domains. From a quality perspective, it learns an action model and assess the learned model *vs.* the reference model given by the domain in terms of precision, recall and the F_1 score. A variable of the formulation that finally has an instantiated value is considered as learned. If it is correctly learned, it is measured as *TP* or *TN*. If the value is wrongly learned, it is measured as *FP* (although this is impossible when precision = 1). If the variable is not instantiated, it is measured as *FN*.

This section runs experiments, to depict the previous quality indicators and running times, under different types of learning variants to analyze the impact of:

- Learning in planning domains that contain, or not, static information and use mutex information whenever available.
- Ranging the size of the input collection of observed plans, from 1 to 50 plans.
- Considering noise in the observations; in particular, uncertainty on the exact time when actions are observed.

[8] Despite the complexity of the propagation of step 5 is polynomial, the complexity of the complete Algorithm in Fig. 8 is non-polynomial because of the "solve-CSP()" invoked in step 2.

[9] Intuitively, this can be seen as a way to improve the *learning power* of the propagation, that is, how informative each propagation step is. Typically, the number of propagation steps reduces in successive executions of the algorithm.

[10] *International Planning Competition*, https://www.icaps-conference.org/competitions.

Table 5
Features of the planning domains and input datasets used in the experiments.

		static info		no static info				
name	# ops	# pred	# alpha	# pred	# alpha	# acts	# init	# goals
blocksworld*	4			27	52	11	25	3
depot*	5			37	64	11	35	2
driverlog	6	28	64	26	48	13	47	6
elevator/miconic	4	16	36	12	20	36	270	9
ferry*	3			14	22	14	18	6
floortile	7	44	182	37	64	7	55	3
grid	5	31	88	27	74	7	36	1
gripper*	3			14	20	15	18	4
hanoi	1	8	30	7	18	5	34	2
logistics	6	24	36	22	32	65	73	10
npuzzle	1	7	12	6	8	3	38	12
openstacks	3	19	46	17	34	9	42	4
pathways	5	24	100	19	32	8	96	1
pegsol-netbenefit	1	10	24	9	12	5	135	31
pegsol-sequential	3	29	72	27	48	8	145	24
satellite	5	26	50	18	36	22	61	8
scananalyzer	3	29	224	26	64	11	51	8
sokoban	3	33	108	26	48	15	276	2
storage	5	38	86	33	66	6	39	1
transport-seq	3	20	36	17	24	14	42	2
visitall	1	5	12	4	8	20	113	14
zenotravel	5	28	56	24	36	6	23	3

From a comparative perspective, this approach is contrasted *vs.* ARMS and FAMA, which have become benchmarks in action model learning. The idea is to analyze the impact of input knowledge on mutex information and observability of intermediate states between actions. Finally, although the formulation is mainly designed to be addressed via constraint propagation, its behavior is also analyzed when it is addressed via a CSP solving task, where perfect precision cannot be guaranteed (recall that the solving task assigns values to all variables, which might introduce imprecision in form of errors, as shown in Section 3.1).

5.1. Setup

The experiments use the STRIPS version of 22 well-known IPC domains,[11] 18 of which have static information. The domains used are those that can be parsed; other domains do not use types or use constants, which are unsupported functionalities in the current parser. Two versions are learned: with and without static predicates. Table 5 presents the domains and their main features in terms of total number of operators, predicates to learn and alphabet size for the two given versions. There are 4 domains that do not use static predicates: they are marked with "*" and only have the no static version. Understanding the mutex information is not easy, so it is only given when it is very intuitive, ranging from 1 (*pegsol-netbenefit*, *satellite* and *visitall*) to 12 (*floortile*) pair/s of predicates. This means the mutexes used as input are usually incomplete.

The input dataset is populated with up to 80 parallel plan traces per domain. The planning problems come from the IPC examples, after arbitrarily modifying their initial and goal states. All plans are generated by LPG [57], so plan optimality is not guaranteed. The three last columns of Table 5 represent average values for the number of actions in the observed plans, number of predicates in the initial state and in the goals. Note that all goals are defined as partial states, which makes the learning task more difficult.

Two important facts can be noticed in Table 5. First, the alphabet size of the static versions is larger than the no static versions (e.g., over 2x in *sokoban* and *floortile*, and over 3x in *pathways* and *scananalyzer*), which shows the increase in the complexity of learning with static information. Second, the number of actions and goals does not need to be particularly large; up to 15 actions and 10 goals are enough in most domains.

The formulation has been implemented in Choco (http://www.choco-solver.org), an open-source Java library that provides an object-oriented API for CP. Choco not only allows us to model a problem and solve it, but it also provides a propagation method, as a set of filtering algorithms, to eliminate values from domain variables that can lead to contradictions. This method is the one that is invoked in the propagation steps of the algorithms of Figs. 7 and 8. The running time per learning task is limited to 300 s on an Intel i5-6400 @ 2.70GHz with 8GB of RAM. If this limit is exceeded, the task is considered unsolvable.

5.2. Quality evaluation

5.2.1. Size of the CP formulation

For this experiment, 50 learning scenarios $\mathcal{L}_1, \mathcal{L}_2 \ldots \mathcal{L}_{50}$ are created, with an input collection of $1, 2 \ldots 50$ noiseless plans, respectively, taken from the input dataset. Each scenario is an aggregation of tasks. In particular, there are 30 different tasks

[11] The cost of each operator is arbitrary. In the tests, the cost is given by the length of the operator name.

Table 6
Average size of the formulations $\mathcal{L}_1, \mathcal{L}_2 \ldots \mathcal{L}_{50}$.

	static info		no static info	
name	# vars	# consts	# vars	# consts
blocksworld*			100	45064
depot*			122	8422
driverlog	127	22817	100	21804
elevator/miconic	209	11325	144	7606
ferry*			96	110988
floortile	132	9487	62	3184
grid	131	22136	105	21424
gripper*			71	38558
hanoi	61	6568	40	4400
logistics	134	13333	114	10746
npuzzle	41	415	30	292
openstacks	127	42056	96	22412
pathways	109	3431	54	894
pegsol-netbenefit	89	1204	68	1003
pegsol-sequential	132	11464	105	11253
satellite	116	31670	91	28083
scananalyzer	99	6624	66	3291
sokoban	165	5241	81	2701
storage	101	8295	73	6758
transport-seq	124	18935	77	10737
visitall	121	4070	78	3211
zenotravel	73	3854	56	2804

per scenario, with different plans, which means a total number of $50 \cdot 30 = 1500$ learning tasks. It is important to note that, in each scenario, the algorithm for learning preconditions and effects via propagation (see Fig. 7) is run once per plan in the task (1 time in $\mathcal{L}_1$, 2 times in $\mathcal{L}_2 \ldots$ 50 times in $\mathcal{L}_{50}$), but the algorithm for learning costs (see Fig. 8) is executed only once per task (1 time in $\mathcal{L}_1$, $\mathcal{L}_2 \ldots \mathcal{L}_{50}$).

In order to have a clear picture of the size of the formulations for the 1500 tasks, Table 6 shows the average values on the number of variables and constraints for the static and no static versions. The size of the formulations highly depends on the domain complexity, that is, the number of operators, predicates and how they are related. There is an important difference between the static and no static versions. The number of variables and constraints can be more than 2x in static *vs.* no static.

5.2.2. Impact of the size of the collection of plans

This section analyzes the relevance and scalability in terms of the number of input plans. It runs the learning tasks, from $\mathcal{L}_1$ to $\mathcal{L}_{50}$, with noiseless plans. For space reasons, the individual results for recall and F_1 per each of the 22 domains are not included here, but for the interested reader the 44 figures are available on-line to explore the behavior in every particular domain.[12] In those figures, the recall and F_1 lines are practically constant through all the tasks in most domains. The average results, only for $\mathcal{L}_{10}, \mathcal{L}_{20} \ldots \mathcal{L}_{50}$, for recall and F_1 are shown in Figs. 9 and 10, respectively. Note that the learning tasks are addressed via propagation, so the precision is always 1. The number of input plans has little impact, as the difference between the results for $\mathcal{L}_{10}$ and $\mathcal{L}_{50}$ is around 0.2 and 0.1 for recall and F_1, respectively. The amount of information that can be learned via propagation tends to converge after 20-30 plans. The recall and F_1 scores are over 0.6 and 0.7, respectively, for the preconditions + effects, and a little lower for the cost.

Table 7 focuses on $\mathcal{L}_{50}$ and depicts the recall and F_1 score per domain. It shows the individual indicators for preconditions and positive + negative effects, in the static and no static versions, and cost. Cost learning is not affected by having, or not, static information because it only depends on the actions. Three important conclusions can be extracted here:

- Static information is harmful for learning preconditions and makes the learning task more complex. Actually, in planning, static information is only used for grounding matters and is usually ignored while planning. This complexity is particularly significant in *sokoban*, which models the popular puzzle game with large amounts of static predicates. In *sokoban*, mutex information is not intuitive and difficult to capture, so negative effects are not learned. Despite this, the average results are still good in the static version, mainly for the effects. In average, the no static version can reach over 10% of global improvement in the recall thanks to a better learning of preconditions.
- Finding a unique interpretation about how the structure of the domain impacts on the quality of the learning is difficult (or impossible). We have not found a set of factors that leads to a better/worse learning. In general, the learning is better if the number of operators and their parameters is small, but a high number of predicates and, concretely, high values of the alphabet make the learning more difficult. The number of constant values (objects) defined in the problem is less important, because their use is restricted by the parameters in operators. After all, the learning relies on observed actions, which are grounded versions

[12] http://agarridot.blogs.upv.es/recall_and_f1_figures.

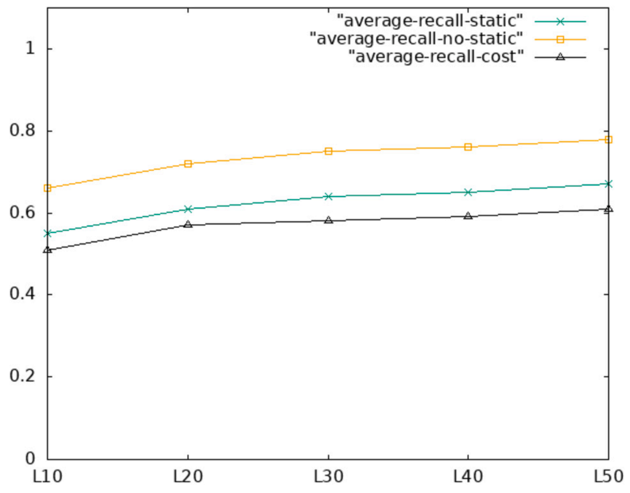

Fig. 9. Average results of the recall *w.r.t.* the input plans: $\mathcal{L}_{10} \ldots \mathcal{L}_{50}$.

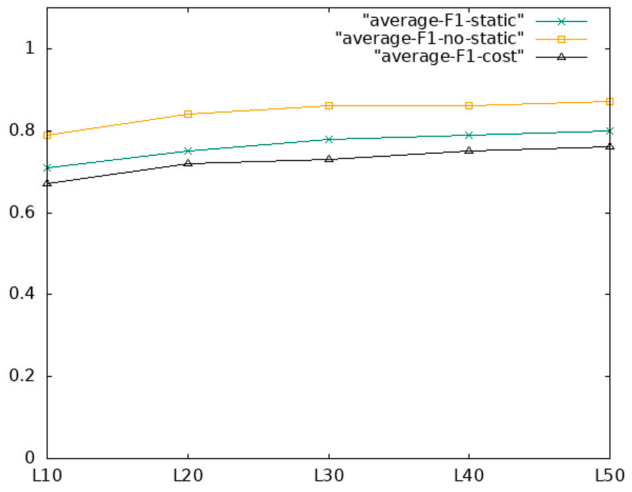

Fig. 10. Average results of the F_1 score *w.r.t.* the input plans: $\mathcal{L}_{10} \ldots \mathcal{L}_{50}$.

of the operators. Using more negative effects hinders the learning, but the relations between the predicates and how they appear (together or separately) in the operators seems to have a stronger effect in the learning. Consequently, there is no a clear answer about which factor is the most decisive.

- The cost is perfectly learned in many domains. However, there are domains where it is impossible to learn the exact cost, particularly when operators work in pairs. For instance, `board` and `debark` in *zenotravel* always appear in pairs, as no person remains forever in an aircraft, but other operators appear freely. In *blocksworld* all operators work in pairs: `unstack` is always planned with `stack` or `put-down`, as a block is always grabbed + ungrabbed in the plans. This means that the number of occurrences of two operators in a plan is the same, which leads to an inconclusive C10 constraint of Table 4. For instance, $x \cdot (\text{cost}(o_1) + \text{cost}(o_2)) = 4$ leads to several solutions when $x = occurrences(o_1) = occurrences(o_2) = 1$: $\{\text{cost}(o_1) = 1, \text{cost}(o_2) = 3\}$, $\{\text{cost}(o_1) = 2, \text{cost}(o_2) = 2\}$, $\{\text{cost}(o_1) = 3, \text{cost}(o_2) = 1\}$, etc. The propagation step can find out the cost of the two operators as a pair, but cannot guarantee the individual costs.

Focusing on the running time, that is, the time for learning via constraint propagation, Table 8 compares the average values for the learning tasks $\mathcal{L}_{10} \ldots \mathcal{L}_{50}$ in all the domains and their static and no static versions. Dealing with static information is harmful and increases the running times. There are some domains where the time is much larger in the static versions than in the respective no static versions (e.g., *elevator*, *floortile*, *pathways* and *transport-seq*). On average, the time for the static version is up to two times higher in $\mathcal{L}_{40}$ and $\mathcal{L}_{50}$ *vs.* the no static version. This shows that the static information has a negative impact in the scalability, though still tractable.

Using a larger collection of plans in a learning task typically increases the running time in the range $\mathcal{L}_{10} \ldots \mathcal{L}_{30}$. However, when the learning task is more constrained (and this happens when more plans are used, i.e., in the range $\mathcal{L}_{30} \ldots \mathcal{L}_{50}$), the propagation algorithm learns more information and the number of future propagation steps decreases. In other words, each propagation step is

Table 7
Average metrics for preconditions, effects and cost in $\mathcal{L}_{50}$. Learning is done via propagation, so Precision = 1. Each x/y pair stands for recall/F$_1$ score. Bold font represents perfect learning, i.e., F$_1$ = Precision = Recall = 1.

	static info				no static info				
name	pre	eff$^+$	eff$^-$	global	pre	eff$^+$	eff$^-$	global	cost
blocksworld*					0.89/0.94	0.89/0.94	0.89/0.94	0.89/0.94	0.00/0.00
depot*					0.53/0.69	0.80/0.89	0.80/0.89	0.71/0.83	0.20/0.33
driverlog	0.47/0.64	0.93/0.97	0.93/0.97	0.78/0.88	0.54/0.71	0.93/0.97	0.93/0.97	0.80/0.89	0.17/0.29
elevator/miconic	0.44/0.62	1.00/1.00	1.00/1.00	0.81/0.90	1.00/1.00	1.00/1.00	1.00/1.00	1.00/**1.00**	0.25/0.40
ferry*					0.50/0.67	0.75/0.86	0.75/0.86	0.67/0.80	0.33/0.50
floortile	0.63/0.77	1.00/1.00	1.00/1.00	0.88/0.93	1.00/1.00	1.00/1.00	1.00/1.00	1.00/**1.00**	1.00/**1.00**
grid	0.22/0.36	0.81/0.90	0.54/0.70	0.52/0.69	0.29/0.45	0.81/0.90	0.54/0.70	0.55/0.71	0.20/0.33
gripper*					0.33/0.50	0.50/0.67	0.25/0.40	0.36/0.53	0.33/0.50
hanoi	0.75/0.86	1.00/1.00	0.50/0.67	0.75/0.86	1.00/1.00	1.00/1.00	0.50/0.67	0.83/0.91	1.00/**1.00**
logistics	0.67/0.80	1.00/1.00	1.00/1.00	0.89/0.94	0.80/0.89	1.00/1.00	1.00/1.00	0.93/0.97	0.17/0.29
npuzzle	0.67/0.80	1.00/1.00	1.00/1.00	0.89/0.94	1.00/1.00	1.00/1.00	1.00/1.00	1.00/**1.00**	1.00/**1.00**
openstacks	0.54/0.71	0.89/0.94	0.89/0.94	0.78/0.87	0.70/0.82	0.89/0.94	0.89/0.94	0.83/0.91	1.00/**1.00**
pathways	0.08/0.14	0.81/0.89	0.20/0.33	0.36/0.53	0.75/0.86	0.83/0.91	0.20/0.33	0.59/0.75	1.00/**1.00**
pegsol-netbenefit	0.75/0.86	1.00/1.00	1.00/1.00	0.92/0.96	1.00/1.00	1.00/1.00	1.00/1.00	1.00/**1.00**	1.00/**1.00**
pegsol-sequential	0.82/0.90	1.00/1.00	1.00/1.00	0.94/0.97	1.00/1.00	1.00/1.00	1.00/1.00	1.00/**1.00**	1.00/**1.00**
satellite	0.06/0.11	0.50/0.67	0.25/0.40	0.27/0.42	0.22/0.36	0.50/0.67	0.25/0.40	0.32/0.49	0.20/0.33
scananalyzer	0.36/0.53	0.50/0.67	0.50/0.67	0.45/0.63	0.50/0.67	0.50/0.67	0.50/0.67	0.50/0.67	0.67/0.80
sokoban	0.00/0.00	0.11/0.20	0.00/0.00	0.04/0.07	0.63/0.77	0.67/0.80	0.56/0.71	0.62/0.76	1.00/**1.00**
storage	0.44/0.62	1.00/1.00	0.80/0.89	0.75/0.86	0.69/0.82	1.00/1.00	0.80/0.89	0.83/0.91	1.00/**1.00**
transport-seq	0.40/0.57	0.60/0.75	0.60/0.75	0.53/0.70	0.84/0.91	0.97/0.99	0.97/0.99	0.93/0.96	0.33/0.50
visitall	0.35/0.52	0.87/0.93	0.73/0.85	0.65/0.79	0.70/0.82	0.87/0.93	0.73/0.85	0.77/0.87	1.00/**1.00**
zenotravel	0.57/0.73	1.00/1.00	1.00/1.00	0.86/0.92	0.80/0.89	1.00/1.00	1.00/1.00	0.93/0.97	0.60/0.75
Average	0.46/0.63	0.83/0.91	0.72/0.84	0.67/0.80	0.71/0.83	0.86/0.92	0.75/0.86	0.78/0.87	0.61/0.76

Table 8
Average running times (s) for learning via propagation *w.r.t.* the input plans: $\mathcal{L}_{10} \dots \mathcal{L}_{50}$. Information is shown for the two versions: with and without static predicates.

	static info					no static info				
name	$\mathcal{L}_{10}$	$\mathcal{L}_{20}$	$\mathcal{L}_{30}$	$\mathcal{L}_{40}$	$\mathcal{L}_{50}$	$\mathcal{L}_{10}$	$\mathcal{L}_{20}$	$\mathcal{L}_{30}$	$\mathcal{L}_{40}$	$\mathcal{L}_{50}$
blocksworld*						4.87	5.71	5.06	3.12	2.52
depot*						1.33	1.41	1.49	0.98	0.44
driverlog	4.64	3.98	5.32	4.92	4.68	3.37	2.81	3.40	3.06	2.72
elevator/miconic	1.23	1.24	1.39	1.46	1.60	0.01	0.01	0.01	0.01	0.01
ferry*						8.16	8.97	8.60	7.73	6.58
floortile	1.84	1.87	8.56	7.20	5.33	0.07	0.04	0.14	0.09	0.09
grid	6.00	6.97	12.19	10.93	9.78	4.71	5.39	9.75	8.51	7.45
gripper*						2.22	2.66	4.26	4.56	3.61
hanoi	0.60	0.86	0.87	0.83	0.39	0.17	0.21	0.19	0.19	0.08
logistics	1.40	0.62	0.29	0.24	0.58	0.47	0.20	0.09	0.07	0.20
npuzzle	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
openstacks	5.27	6.68	6.50	8.27	6.95	2.00	2.25	1.80	1.84	1.49
pathways	0.77	0.96	1.29	1.08	0.86	0.02	0.02	0.03	0.03	0.03
pegsol-netbenefit	0.03	0.04	0.06	0.07	0.05	0.01	0.01	0.01	0.01	0.01
pegsol-sequential	1.57	1.35	1.33	2.27	5.34	0.79	0.47	0.16	0.25	0.60
satellite	1.25	3.28	4.17	6.93	7.33	0.70	1.78	2.26	3.82	4.04
scananalyzer	0.62	0.46	0.49	4.35	4.47	0.09	0.04	0.05	0.26	0.28
sokoban	1.93	1.41	1.58	4.13	4.71	0.16	0.11	0.14	0.41	0.47
storage	3.58	3.50	2.78	1.16	0.52	1.59	1.47	1.17	0.51	0.21
transport-seq	4.10	1.94	1.59	1.29	2.02	1.18	0.31	0.07	0.07	0.12
visitall	0.23	0.15	0.19	0.23	0.25	0.09	0.06	0.08	0.12	0.11
zenotravel	0.13	0.41	0.44	0.71	0.41	0.02	0.06	0.06	0.10	0.06
Average	1.96	1.99	2.73	3.12	3.07	1.46	1.55	1.77	1.63	1.42

more informative/powerful, thus detecting more inconsistencies and learning more information. This highly depends on the structure of the domain and the internal relations between its predicates and/or actions, so a unique explanation cannot be generalized.

5.2.3. Impact of noise in the start time of actions

Rather than using a collection of plans where the start time of the actions is observed correctly, this section now includes uncertainty on those times (to deal with certain noise) and runs two experiments: i) $\mathsf{start}(o_\pi) \pm 2$, and ii) $\mathsf{start}(o_\pi) \pm 4$. In the former, there is uncertainty of 5 possible values on the start time and in the latter the uncertainty is of 9 values: the plan lengths remain unchanged, but the start times are now blurred. It is important to note that noisy observations do not mean that some constraints

Table 9
Average metrics for preconditions and effects in $\mathcal{L}_{50}$ when noise on the action start time is considered. Only the no static version of the domains is shown. Learning is done via propagation, so Precision = 1. Each x/y pair stands for recall/F$_1$ score. Bold font represents perfect learning, i.e., F$_1$ = Precision = Recall = 1.

	$\mathsf{start}(o_\pi) \pm 2 = 5$ possible start times				$\mathsf{start}(o_\pi) \pm 4 = 9$ possible start times			
name	pre	eff$^+$	eff$^-$	global	pre	eff$^+$	eff$^-$	global
blocksworld	0.89/0.94	0.89/0.94	0.89/0.94	0.89/0.94	0.89/0.94	0.89/0.94	0.89/0.94	0.89/0.94
depot	0.53/0.69	0.80/0.89	0.80/0.89	0.71/0.83	0.53/0.69	0.80/0.89	0.80/0.89	0.71/0.83
driverlog	0.33/0.50	0.57/0.73	0.57/0.73	0.49/0.66	0.33/0.50	0.57/0.73	0.57/0.73	0.49/0.66
elevator/miconic	1.00/1.00	1.00/1.00	1.00/1.00	1.00/**1.00**	1.00/1.00	1.00/1.00	1.00/1.00	1.00/**1.00**
ferry	0.50/0.67	0.75/0.86	0.75/0.86	0.67/0.80	0.50/0.67	0.73/0.84	0.73/0.84	0.65/0.79
floortile	1.00/1.00	1.00/1.00	1.00/1.00	1.00/**1.00**	0.98/0.99	0.97/0.98	1.00/1.00	0.98/0.99
grid	0.23/0.37	0.70/0.82	0.42/0.59	0.45/0.62	0.23/0.37	0.70/0.82	0.42/0.59	0.45/0.62
gripper	0.33/0.50	0.50/0.67	0.25/0.40	0.36/0.53	0.33/0.50	0.50/0.67	0.25/0.40	0.36/0.53
hanoi	1.00/1.00	1.00/1.00	0.50/0.67	0.83/0.91	1.00/1.00	1.00/1.00	0.50/0.67	0.83/0.91
logistics	0.80/0.89	1.00/1.00	1.00/1.00	0.93/0.97	0.80/0.89	1.00/1.00	1.00/1.00	0.93/0.97
npuzzle	1.00/1.00	1.00/1.00	1.00/1.00	1.00/**1.00**	1.00/1.00	1.00/1.00	1.00/1.00	1.00/**1.00**
openstacks	0.27/0.43	0.39/0.56	0.39/0.56	0.35/0.52	0.27/0.43	0.39/0.56	0.39/0.56	0.35/0.52
pathways	0.75/0.86	0.83/0.91	0.20/0.33	0.59/0.75	0.75/0.86	0.83/0.91	0.20/0.33	0.59/0.75
pegsol-netbenefit	1.00/1.00	1.00/1.00	1.00/1.00	1.00/**1.00**	1.00/1.00	1.00/1.00	1.00/1.00	1.00/**1.00**
pegsol-sequential	0.51/0.68	0.51/0.68	0.53/0.70	0.52/0.68	0.44/0.62	0.44/0.62	0.44/0.62	0.44/0.62
satellite	0.11/0.20	0.40/0.57	0.25/0.40	0.25/0.40	0.11/0.20	0.40/0.57	0.25/0.40	0.25/0.40
scananalyzer	0.50/0.67	0.50/0.67	0.50/0.67	0.50/0.67	0.50/0.67	0.49/0.66	0.50/0.67	0.50/0.67
sokoban	0.63/0.77	0.67/0.80	0.56/0.71	0.62/0.76	0.63/0.77	0.67/0.80	0.56/0.71	0.62/0.76
storage	0.53/0.69	0.84/0.91	0.64/0.78	0.67/0.80	0.53/0.69	0.80/0.89	0.60/0.75	0.64/0.78
transport-seq	0.57/0.73	0.60/0.75	0.60/0.75	0.59/0.74	0.57/0.73	0.60/0.75	0.60/0.75	0.59/0.74
visitall	0.70/0.82	0.87/0.93	0.73/0.85	0.77/0.87	0.70/0.82	0.87/0.93	0.73/0.85	0.77/0.87
zenotravel	0.60/0.75	0.84/0.91	0.57/0.73	0.67/0.80	0.50/0.67	0.77/0.87	0.57/0.73	0.61/0.76
Average	0.63/0.77	0.76/0.86	0.64/0.78	0.68/0.81	0.62/0.76	0.75/0.85	0.64/0.78	0.67/0.80

might be unsatisfied. The learned model must satisfy all the constraints, including those induced by the noise. This is also an indication of the robustness of the model.

Table 9 shows the results of recall and F$_1$ for $\mathcal{L}_{50}$, now focusing on the no static version of the domains. The cost learning is not affected by the noisy observations, so the results in the last column of Table 7 remain unchanged no matter the noise. The intuition is that dealing with noise will make the metrics worse. After all, the plans are now more uncertain and the formulation might not be able to learn much information with perfect precision. In fact, the C7 and C8 constraints of Table 4, which formulate the causal links and threat resolution, are much more complex when uncertainty on the start times is present. The negative impact of the noise in the learning grows in those domains where the potential number of causal links is larger. However, the results show that the impact of noise is not very harmful and the model remains robust; the average recall/F$_1$ is 0.78/0.87 without noise, 0.68/0.81 with 5 possible start times and 0.67/0.80 with 9 possible start times. It is also noticeable that, despite the noise, the perfect learning is achieved in 4 domains.

Focusing on the running time, Table 10 compares the average values for the learning tasks $\mathcal{L}_{10} \dots \mathcal{L}_{50}$ in all the domains and the two versions of uncertainty on the start time of the actions. Like in Table 9, only the no static version of the domains is shown. Two conclusions can be extracted from this experiment:

- Including noise in the start time of actions has a negative effect in the running time. Now, solving a learning task requires more time because the amount of information that can be precisely learned is more reduced. Intuitively, the learning power of each propagation step is reduced (being less informative), which means that more propagation steps are run. This can be noted by comparing the values of Table 10 with the values of Table 8, where no noise is included. In Table 8, the average times are below 2s for the version without static predicates, whereas in Table 10 all average times are over 2s.
- Including more or less uncertainty does not modify the formulation size of a learning task. Therefore, there is no difference between $\mathsf{start}(o_\pi) \pm 2$ and $\mathsf{start}(o_\pi) \pm 4$ in terms of variables/constraints of the formulation. Although more uncertainty might reduce the learning power of the propagation, which leads to more propagation steps, there is not a substantial difference in the average times of $\mathsf{start}(o_\pi) \pm 2$ and $\mathsf{start}(o_\pi) \pm 4$. This shows that the model is also robust *w.r.t.* the running time for different levels of uncertainty.

5.2.4. Semantic vs. syntactic evaluation

As pointed out in Section 2.4, most metrics used in learning are syntactically-oriented, as they evaluate the learned model *w.r.t.* a reference or unique ground truth model. However, one learned model might slightly differ from a reference one, while the underlying model is still consistent. In other words, two syntactically-different models can be semantically equivalent. This is the reason why many learning approaches also use a semantic-oriented evaluation to assess whether the learned model can reproduce, with no contradictions or inconsistencies, unknown samples or plan observations [6,12,13,24,35]. In short, these approaches split the initial plan dataset into two disjoint sets: one for learning and one for testing. The idea is to assess how well the learned model captures

Table 10
Average running times (s) for learning via propagation *w.r.t.* uncertainty on the start time of the actions. The input plans are $\mathcal{L}_{10} \ldots \mathcal{L}_{50}$. Only the no static version of the domains is shown.

	start$(o_\pi) \pm 2$					start$(o_\pi) \pm 4$				
name	$\mathcal{L}_{10}$	$\mathcal{L}_{20}$	$\mathcal{L}_{30}$	$\mathcal{L}_{40}$	$\mathcal{L}_{50}$	$\mathcal{L}_{10}$	$\mathcal{L}_{20}$	$\mathcal{L}_{30}$	$\mathcal{L}_{40}$	$\mathcal{L}_{50}$
blocksworld	5.13	5.71	5.31	3.21	2.50	8.79	5.96	5.15	3.45	2.50
depot	2.15	2.08	1.55	0.95	0.40	2.70	2.75	1.57	0.95	0.42
driverlog	4.74	4.68	6.75	8.31	8.61	4.25	4.02	6.13	7.42	7.67
elevator/miconic	0.10	0.09	0.08	0.08	0.09	0.12	0.09	0.07	0.08	0.09
ferry	10.81	11.27	9.75	8.62	7.55	11.95	12.77	11.69	9.08	7.20
floortile	0.20	0.05	0.14	0.11	0.10	0.20	0.11	1.03	0.18	0.10
grid	5.04	5.63	10.21	9.53	8.22	4.42	4.82	8.85	8.09	7.22
gripper	5.61	2.71	4.19	4.67	3.63	5.17	2.53	3.96	4.36	3.41
hanoi	0.15	0.20	0.18	0.18	0.08	0.15	0.19	0.18	0.18	0.07
logistics	1.05	0.42	0.18	0.13	0.31	1.29	0.41	0.17	0.12	0.29
npuzzle	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
openstacks	3.54	5.04	5.59	8.37	7.44	3.63	5.16	5.36	7.99	7.07
pathways	0.06	0.06	0.07	0.06	0.05	0.06	0.06	0.09	0.07	0.06
pegsol-netbenefit	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
pegsol-sequential	1.04	1.41	2.21	4.37	7.59	0.96	1.32	2.11	4.21	9.69
satellite	8.41	2.01	2.69	4.48	4.74	7.55	1.82	2.41	4.05	4.31
scananalyzer	0.21	0.04	0.05	0.23	0.42	0.92	0.23	0.06	0.23	0.39
sokoban	0.16	0.11	0.14	0.40	0.44	0.15	0.10	0.12	0.37	0.43
storage	2.99	3.22	2.45	0.94	0.43	2.74	2.97	2.22	0.87	0.39
transport-seq	1.47	0.70	0.50	0.44	0.91	1.38	0.64	0.44	0.41	0.84
visitall	0.08	0.07	0.08	0.11	0.10	0.07	0.06	0.07	0.10	0.09
zenotravel	0.10	0.37	0.28	0.38	0.21	0.10	0.41	0.33	0.51	0.31
Average	2.41	2.09	2.38	2.53	2.45	2.57	2.11	2.37	2.40	2.39

the physics of the planning domain or, equivalently, how precise is the learned model to explain unknown samples. Formally, the semantic success ratio is defined as $success\ ratio = \frac{PS}{|dataset|}$, where PS counts the number of positive samples on a test $dataset$ that are consistent with the learned model.

A semantic evaluation of the proposed constraint propagation approach is straightforward. By Lemma 1, what is learned is 100% precise and consistent for any present or future plan observation of the reference model. If several action models are semantically equivalent, learning via constraint propagation will be restricted to the common part of those models. Therefore, the model learned will be consistent with all models, which means $success\ ratio = 1$.

5.3. Comparison to ARMS and FAMA

This section compares the proposed approach to ARMS and FAMA, which are two successful approaches for action model learning. They both show important gain when the sequence of intermediate states between actions is also observed. Therefore, they define an observability degree, ranging from 0% to 100%, that measures the probability of observing a predicate p in a state at time t. This can be easily included in the proposed formulation by using $\mathsf{obs}(p,t)$ dummy actions. FAMA requires the definition of the goals as a full state, which is easily supported by the formulation. ARMS and FAMA do not support cost learning, nor uncertainty on the start time of the actions, so the comparison is restricted to the cases and the 12 domains manageable by them, i.e., *blocksworld*, *driverlog*, *ferry*, *floortile*, *grid*, *gripper*, *hanoi*, *npuzzle*, *satellite*, *transport-sequential*, *visitall* and *zenotravel*. All domains used in the comparison represent the versions with static information.

The experiment given in [12] has been reproduced to learn from a dataset of 10 sequential plans ($\mathcal{L}_{10}$), with 10 actions each, per domain. Figs. 11, 12 and 13 depict the average results for precision, recall and F_1 score, respectively. ARMS and FAMA cannot reason on mutex information, so for the LvCP approach, the results with and without mutex information are shown. Learning via Constraint Propagation always returns perfect precision, no matter if mutex information is given or not. This can be seen in Fig. 11, where precision = 1 for LvCP. As expected (see Fig. 12), the recall values are slightly lower in LvCP since ensuring precision reduces the amount of information learned. However, it is important to note that: i) LvCP with mutex information shows better recall than ARMS in low (0-30%) observability degrees, and ii) mutex information is only relevant for learning in low observability degrees. Once the observability degree of 40% is passed, the mutex information becomes irrelevant, particularly for the negative effects that can now be directly observed. In other words, mutex information is beneficial only in absence of intermediate state observations, and its practical effectiveness disappears when the observability degree increases. In Fig. 13, the F_1 score of LvCP is very competitive and shows a constant behavior beyond the observability degree of 40%.

Focusing on the running time, Tables 11 and 12 show the average values for the observability degrees when reasoning and not reasoning with mutex information, respectively. Three conclusions can be extracted here:

- Increasing the observability degree has not a significant impact in the running time when the collection of plans is not high (only 10 plans are considered in this experiment). We have run some additional experiments with more than 10 input plans and

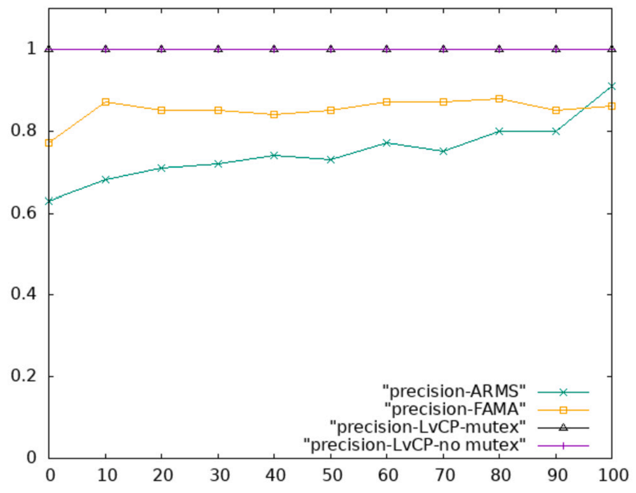

Fig. 11. Average results of the precision for different observability degrees.

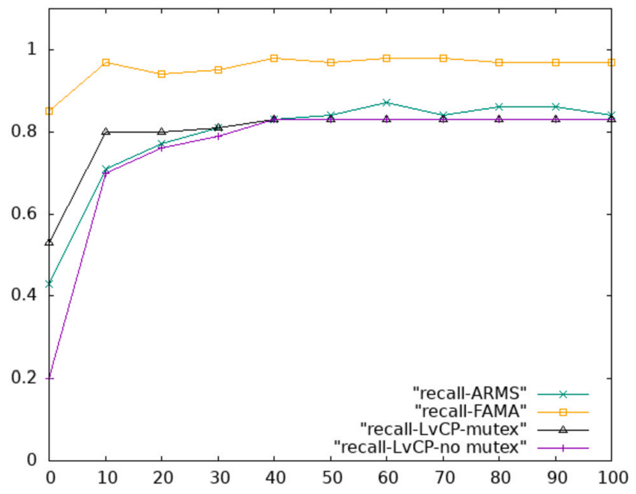

Fig. 12. Average results of the recall for different observability degrees.

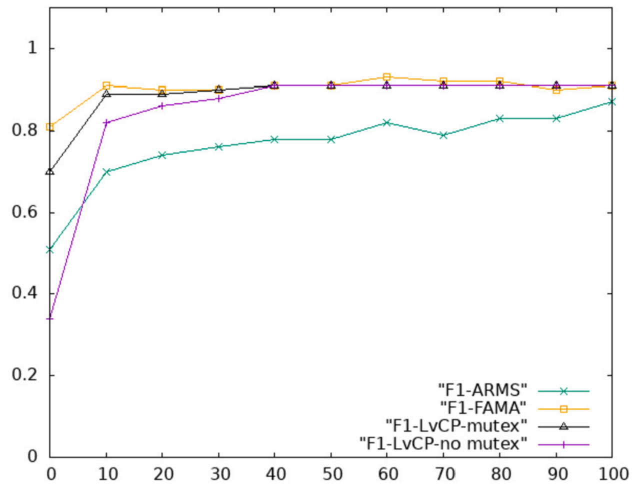

Fig. 13. Average results of the F_1 score for different observability degrees.

Table 11
Average running times (s) for learning via propagation with mutex information for different observability degrees ($0 \ldots 100\%$). Only the static version of the domains is shown.

name	0	10	20	30	40	50	60	70	80	90	100
blocksworld	4.64	2.73	2.13	2.35	1.82	2.19	2.34	2.33	2.34	2.46	2.49
driverlog	1.00	1.08	1.00	1.23	1.02	1.08	1.05	1.13	1.20	1.22	1.22
ferry	1.16	0.97	0.88	1.00	0.90	1.00	1.03	1.03	1.05	1.16	1.10
floortile	8.91	8.83	9.07	8.30	8.44	8.85	9.10	10.03	10.35	10.76	10.90
grid	6.54	5.02	5.96	6.53	5.81	6.07	5.42	5.56	5.35	5.65	5.66
gripper	1.12	0.80	0.71	0.77	0.68	0.68	0.72	0.73	0.79	0.85	0.87
hanoi	1.25	1.46	2.13	1.69	1.68	1.91	2.07	2.22	2.42	2.66	2.68
npuzzle	0.38	0.43	0.59	0.49	0.60	0.56	0.62	0.63	0.68	0.78	0.73
satellite	0.40	0.40	0.68	0.33	0.40	0.38	0.47	0.46	0.53	0.55	0.54
transport-seq	0.62	0.54	0.67	0.52	0.57	0.62	0.60	0.69	0.76	0.84	0.84
visitall	1.54	1.13	1.36	1.38	1.42	1.63	1.54	1.71	1.79	1.91	1.96
zenotravel	1.14	1.32	1.30	1.00	1.00	1.07	1.06	1.06	1.11	1.19	1.17
Average	2.39	2.06	2.21	2.13	2.03	2.17	2.17	2.30	2.36	2.50	2.51

Table 12
Average running times (s) for learning via propagation with no mutex information for different observability degrees ($0 \ldots 100\%$). Only the static version of the domains is shown.

name	0	10	20	30	40	50	60	70	80	90	100
blocksworld	4.31	2.22	1.87	1.85	2.24	2.18	2.21	2.31	2.54	2.45	2.54
driverlog	1.31	1.11	0.92	1.07	1.15	1.03	1.04	1.15	1.22	1.19	1.22
ferry	1.41	1.15	1.36	1.27	0.88	1.10	1.11	1.07	1.14	1.13	1.19
floortile	8.79	8.96	8.66	8.54	8.82	9.41	9.94	10.38	11.98	10.29	11.01
grid	6.54	6.67	5.98	5.28	5.12	4.93	6.37	5.76	5.79	5.40	5.80
gripper	1.48	0.76	0.79	0.66	0.66	0.76	0.95	0.79	0.84	0.81	0.90
hanoi	1.50	1.40	1.58	1.74	2.05	2.13	2.46	2.64	2.81	2.78	3.00
npuzzle	0.56	0.42	0.41	0.50	0.52	0.55	0.63	0.67	0.76	0.70	0.76
satellite	0.40	0.37	0.31	0.40	0.38	0.39	0.50	0.53	0.48	0.49	0.54
transport-seq	0.61	0.59	0.67	0.57	0.63	0.67	0.85	0.77	0.73	0.78	0.87
visitall	1.40	1.21	1.30	1.31	1.41	1.52	1.85	1.79	1.78	1.88	2.01
zenotravel	1.46	1.24	1.08	1.11	1.07	1.12	1.53	1.19	1.12	1.17	1.30
Average	2.48	2.18	2.08	2.03	2.08	2.15	2.45	2.42	2.60	2.42	2.60

the running time starts to increase when the observability degree is over 50%, because of the overhead of using $\mathsf{obs}(p,t)$ dummy actions in many plans.

- The most interesting observability degree for the running time moves around 30-60%. Intuitively, such a degree provides enough observations to improve the learning power of the propagation while it is not too high to provoke an overhead due to the observations.
- The differences on average values between Tables 11 and 12 are minimal, which shows that reasoning with mutex information is not very complex (the C11.* constraints are not expensive).

Solving the learning task The main advantage of learning via constraint propagation is that the learned model is 100% precise, though it might be incomplete. On the contrary, if one needs to learn a complete model (i.e., all variables want to be learned), one will probably need to instantiate variables without enough information. This means that absolute precision cannot be ensured, that is, precision ≤ 1. In such a case, one only needs to use a CSP solving approach, as described in Section 2.2.1. This is the test that is presented here, where the running time per learning task remains limited to 300 s.

The experiment of [12] is reproduced again, but rather than learning via constraint propagation, it now solves the CSP and retrieves its first solution (note that in a satisfaction problem all solutions are equally valid). The objective is to find a complete action model to check how much the recall increases, how much the precision decreases, and to compare the new results *vs.* Figs. 11, 12 and 13. Intuitively, one expects to obtain higher values of recall, as more information is learned, but values of precision below 1, as perfect precision cannot be guaranteed now.

Figs. 14, 15 and 16 depict the new average results for precision, recall and F_1 score, respectively, under a CSP solving approach. The precision results are very good: they are always over 0.9. Also, they practically keep a constant line for solving with and without mutex information. The recall values are very close to 1: 0.98-0.99 values for observability degrees higher than 20%. In comparison to Fig. 12, the recall has increased around 0.15-0.2. The results for F_1 are also very good and keep a constant line. The main conclusion is that the solving approach for the CP formulation presents very high and steady indicators, which are better than the ones for ARMS and FAMA, particularly in terms of precision and F_1 score. Although learning a complete action model cannot ensure absolute precision, the loss of precision is less than 10%, while the increase in the recall is close to 20%. This means that the CP formulation

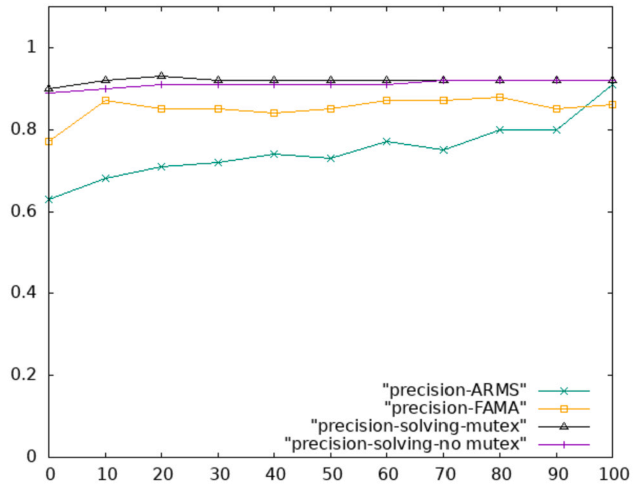

Fig. 14. Average results of the precision for different observability degrees under a solving approach (now Precision ≤ 1).

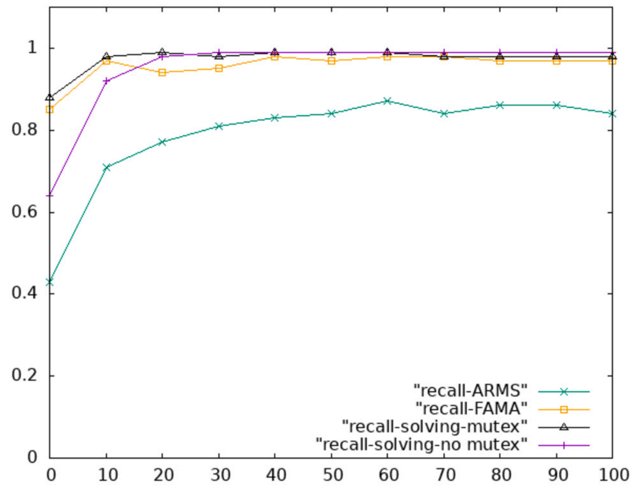

Fig. 15. Average results of the recall for different observability degrees under a solving approach (now Precision ≤ 1).

is very appealing for learning, both for perfect and imperfect precision. Also, the use of mutex here, where partial observability on the intermediate states is available, becomes unnecessary.

Finally, Tables 13 and 14 show the running times under a solving approach for different observability degrees when reasoning and not reasoning with mutex information, respectively. The twofold conclusions that can be extracted are:

- When the collection of plans is not high, e.g., $\mathcal{L}_{10}$ like in this case, using a solving approach for the learning task is very fast. Also, increasing the observability degree has not a significant impact in the running time, which means the observations do not provoke much overhead. Obviously, the running time increases in learning tasks where the collection of plans is above 30-40.
- A comparison between Tables 11 and 12 (learning via propagation) and Tables 13 and 14 (learning via solving) shows that the running times for a solving approach are lower. However, it is important to recall that the values in Tables 11 and 12 imply 10 executions (once per plan in $\mathcal{L}_{10}$) of the propagation algorithm of Fig. 7, whereas the values in Tables 13 and 14 imply only one solving task. In other words, the computational complexity of a propagation step is better than the complexity of solving. However, the algorithm of Fig. 7 is run several times, and also its multiple propagation steps, whereas the solving task is run only once.

6. Discussion and lessons learned

This section analyzes and discusses the main results obtained from the evaluation and the lessons learned:

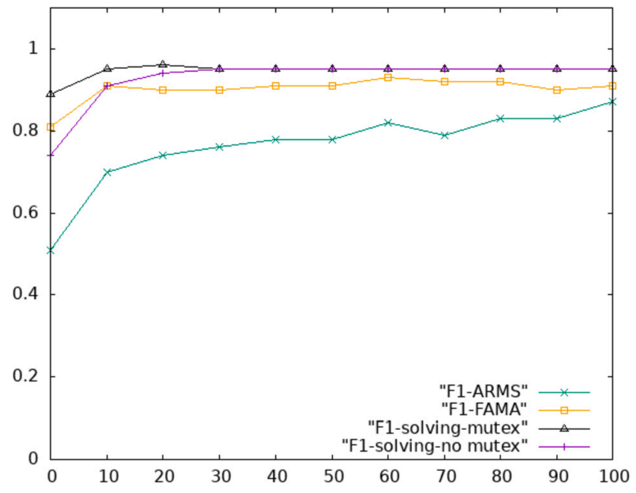

Fig. 16. Average results of the F_1 score for different observability degrees under a solving approach (now Precision ≤ 1).

Table 13
Average running times (s) for learning under a solving approach with mutex information for different observability degrees ($0\ldots100\%$). Only the static version of the domains is shown.

name	0	10	20	30	40	50	60	70	80	90	100
blocksworld	0.51	0.70	0.56	0.90	0.65	0.92	0.74	5.15	0.83	1.16	0.88
driverlog	0.19	0.18	0.18	0.19	0.21	0.22	0.22	0.21	0.23	0.23	0.25
ferry	0.49	0.52	0.51	0.60	0.56	0.60	0.60	0.62	0.65	0.69	0.70
floortile	0.52	0.48	0.48	0.51	0.56	0.53	0.55	0.74	0.60	0.62	0.63
grid	0.51	0.50	0.60	0.72	0.57	0.54	0.56	0.58	0.61	0.60	4.06
gripper	0.52	0.59	0.97	1.25	0.65	0.67	0.86	0.75	0.92	0.84	0.84
hanoi	0.44	0.39	0.63	0.57	0.53	0.54	0.55	0.60	0.80	0.67	0.74
npuzzle	0.26	0.35	0.35	0.34	0.38	0.42	0.62	0.45	0.46	3.74	3.69
satellite	0.08	0.09	0.10	0.10	0.09	0.12	0.11	0.11	0.15	0.13	0.13
transport-seq	0.17	0.15	0.16	0.18	0.36	0.20	0.22	0.21	0.23	0.26	0.26
visitall	1.53	5.72	1.01	1.34	1.30	1.35	1.17	1.25	1.28	1.36	1.39
zenotravel	0.15	0.17	0.24	0.17	0.19	0.21	0.21	0.19	0.21	0.21	0.21
Average	0.45	0.82	0.48	0.57	0.50	0.53	0.53	0.91	0.58	0.88	1.15

Table 14
Average running times (s) for learning under a solving approach with no mutex information for different observability degrees ($0\ldots100\%$). Only the static version of the domains is shown.

name	0	10	20	30	40	50	60	70	80	90	100
blocksworld	3.68	0.54	0.67	0.66	0.67	0.70	0.75	0.77	1.05	0.86	0.97
driverlog	0.21	0.18	0.18	0.18	0.24	0.21	0.21	0.20	0.24	0.26	0.23
ferry	1.25	0.50	0.54	0.53	0.58	0.58	0.62	0.87	0.68	0.65	0.73
floortile	0.51	3.09	0.48	0.48	0.57	0.52	0.65	0.61	0.68	0.58	0.63
grid	3.26	0.54	0.53	0.51	4.63	0.55	0.72	0.72	0.61	0.74	0.62
gripper	0.54	0.59	0.60	4.14	0.64	0.82	1.05	1.14	0.82	0.79	0.87
hanoi	0.71	0.55	0.44	0.51	0.59	0.52	0.62	0.65	0.91	0.66	0.87
npuzzle	3.15	0.29	0.30	0.30	0.36	0.38	0.43	0.46	0.52	0.48	0.54
satellite	0.13	0.09	0.08	0.10	0.12	0.12	0.14	0.11	0.11	0.14	0.15
transport-seq	0.18	0.16	0.17	0.17	0.20	0.21	0.23	0.23	0.23	0.22	0.26
visitall	6.50	0.86	6.29	0.93	1.02	1.07	1.81	1.20	1.29	1.31	8.72
zenotravel	0.16	0.18	0.17	0.16	0.18	0.31	0.21	0.19	0.21	0.37	0.22
Average	1.69	0.63	0.87	0.72	0.82	0.50	0.62	0.60	0.61	0.59	1.23

- The static information, which is typically used for grounding matters in many planning domains, increases the size of the alphabet of an operator o ($\alpha(o)$) and the candidate predicates to be used as preconditions/effects ($\gamma(o)$). Consequently, dealing with static information increases the size of the formulation, which has a negative effect in its complexity, the quality of the learning, the running time and the scalability in general.
- A priori, the intuition is that one will need many observed plans to learn with perfect precision. However, this is not necessarily the case. Almost identical results are obtained when 20 or more plans are considered. This is an important difference *w.r.t.*

other works that need hundreds of plans (see Table 2). In other words: i) the CP formulation learns as much as possible from a minimal amount of observations, and ii) it does not require many plans that increase the running times, particularly when static information is present.

- The cost is perfectly learned ($F_1 = 1$) in many domains, which is an unsupported feature of most approaches in literature (see Table 1). However, there are domains where such perfect learning is impossible because one operator and its inverse always appear together, e.g., *blocksworld*. Actually, this is not a limitation of the formulation in itself, but it is intrinsic to the domain definition.
- The use of mutex is only useful to learn negative effects. This is a direct consequence of the PDDL formalism for predicates. If a state variable per predicate could be defined, similarly to the SAS^+ formalism, asserting a new value would automatically mean to delete the previous value, thus making negative effects and mutex reasoning unnecessary. After all, there is no necessity to learn negative effects if they are not required nor directly observed. This is an open issue that requires further research. However, when the observability degree increases or the sequence of intermediate states is observed, such mutex reasoning becomes irrelevant. In any case, reasoning on mutex is not computationally complex.
- Having noisy observations does not affect the formulation size, but it has a negative impact in the learned model because it includes uncertainty. In fact, finding the right causal links in a formulation with noise is more difficult, thus reducing the power of the propagation algorithm. Fortunately, the quality of the learning is not significantly worse and, the difference between having a small *vs.* medium amount of noise is minimal, also in terms of the running time. This is an indication of the approach to learn robust models.
- Learning via propagation is excellent for precision, but less complete; although the experiments achieve solutions with around 70% of recall. The computational complexity of each propagation step is polynomial (see Lemma 2 and 3), but the propagation steps need to be run several times. On the contrary, learning via CSP solving is more complete, but less precise and computationally more complex (it is non-polynomial); although the solving process needs to be run only once. The experiments have shown that both learning options are interesting as the formulation leads to safe, very precise and complete learned models, no matter if propagation or solving is used.

7. Conclusions

This work has proposed a constraint propagation approach to learn a planning action model with perfect precision. Starting from an arbitrary collection of plan observations and a set of empty operators, it formulates all the necessary constraints to learn the preconditions, effects and costs. Since it model constraints in terms of Partial-Order Causal-Link (POCL) planning, thus explicitly modeling the causal links and threat resolution, it can better capture the causality of the observed plans and, consequently, of the action model.

The general contribution of this work is its adaptability to accept different levels of input data. It unifies features that are individually supported by other approaches in literature: input knowledge on mutex when available, partial and full goal states, information on intermediate states, noise on action start times, and cost learning. Using a CP approach has several advantages for learning, which form the specific contributions of the work.

First, the definition of variables in the formulation facilitates the use of an empty model of operators or a more complete one. For instance, one might start from a partial model where some preconditions, effects or costs are known. This can be easily done by restricting the domain of the variables, e.g., $\mathsf{pre}(p,o) = pre$ means that one knows that p is a precondition of o, whereas $\mathtt{eff}(p,o) = false$ means that p is not an effect of o. This definition of variables tolerates uncertainty/noise on cost, actions or observations of intermediate information. This is very useful, as dealing with noise is indispensable in some real scenarios that require robust approaches.

Second, the learned model satisfies all the constraints imposed by the observations (*aka* safe learning), and not just some of them like in other approaches. This means that what is learned is 100% correct for the current observations. Moreover, the number of observations is very flexible: from one to many. In the experiments, what is learned via propagation tends to converge after 30-40 plan observations, and the complexity of using more plans does not pay off. Additional experiments, with more than 50 plans in some domains, have been studied, but the learning task becomes very expensive and the results hardly improve.

Third, the learning process can be addressed from two perspectives. On the one hand, using CP propagation learns with perfect precision; i.e., what is learned is 100% correct and reliable not only for the current observations, but also for any new observation over the original planning model (*aka* robust learning). Although obtaining a complete solution just by propagation is uncommon, it is possible, and more often than expected, that many variables have domains that are reduced to just one value. Intuitively, learning via propagation learns only what is correct, and keeps open those decisions where no more knowledge can be extracted. This is indispensable when reliability is a key issue, where it is better to learn step by step in a safe way, rather than learning and having to retract later. The propagation algorithms can be run from 1 to n times, which allows for an incremental update of the action model with new information without the necessity to retract previously learned part of the model. On the other hand, one can use a CSP solver to learn a complete model (expecting higher values of recall), but without ensuring perfect precision in the cases where more than one model can be learned from the given observations. Solving the formulation means solving a satisfaction problem. Although different metrics to specify preferences over the space of possible solutions have been investigated, a metric that always leads to the best learned models has not been found. Obviously, any learned model can be tuned by experts before its application to real-world planning.

Fourth, the formulation is solver-independent, meaning that an arbitrary solver can be used for its resolution. Also, most of the constraints are logical implications that can be turned into Conjunctive Normal Form (CNF) clauses to use SAT and Satisfiability Modulo Theories (SMT) solvers. This investigation remains as future work.

This work has some limitations. First, the CP formulation shows scalability problems when addressing learning tasks $\mathcal{L}_n$ in complex planning domains with values of n over 50-75. The size of the formulation grows significantly when the collection of input plans is large, particularly in domains with a lot of static information, and the running time might become intractable. Second, this work assumes no conflicting observations. This assumption could be relaxed to use sensors that are not flawless. This would require to define soft constraints in the formulation and satisfy as many observations as possible: an optimization-based learning task would be necessary, rather than a satisfaction one, and the idea of safe learning would disappear. Third, a more general concept of noise within the plans could be used, where some actions are not properly observed or observed when they should not. Again, this is equivalent to work with conflicting observations and would require optimization instead of satisfaction. This would likely reduce the power of the propagation approach.

Solving these limitations is part of future work. Additionally, there are some challenges that need more research. First, adopting a SAS$^+$-inspired formalism to avoid the necessity of negative effects and mutex reasoning. Second, using stronger, more powerful, propagation algorithms to learn more complete action models and deal with optimization tasks. The important tradeoff here is the amount of learning *vs.* the complexity of the approach. Finally, analyzing the usability of the model learned with perfect precision and limited recall for other planning tasks, such as simplification of planning domains, plan validation or goal recognition.

Data availability

Data will be made available on request.

Acknowledgements

This work has been partially supported by grant PID2021-127647NB-C22 funded by MCIN/AEI/10.13039/501100011033 and by "ERDF A way of making Europe", and by the Spanish MINECO project TIN2017-88476-C2-1-R.

References

[1] M. Ghallab, D. Nau, P. Traverso, Automated Planning. Theory and Practice, Morgan Kaufmann, 2004.

[2] R. Fikes, N. Nilsson, STRIPS: a new approach to the application of theorem proving to problem solving, Artif. Intell. 2 (1971) 189–208, https://doi.org/10.1016/0004-3702(71)90010-5.

[3] H. Geffner, Functional STRIPS: a more flexible language for planning and problem solving, in: Logic-Based Artificial Intelligence, 2000.

[4] E. Pednault, ADL: exploring the middle ground between strips and the situation calculus, in: Proc. Int. Conference on Principles of Knowledge Representation and Reasoning (KR-89), Morgan Kaufmann, San Francisco, CA, 1989, pp. 324–332, https://dl.acm.org/doi/10.5555/112922.112954.

[5] M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, D. Wilkins, PDDL - The Planning Domain Definition Language, AIPS-98 Planning Competition Committee, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.212, 1998.

[6] Q. Yang, K. Wu, Y. Jiang, Learning action models from plan examples using weighted MAX-SAT, Artif. Intell. 171 (2–3) (2007) 107–143, https://doi.org/10.1016/j.artint.2006.11.005.

[7] D. Aineto, S. Jimenez, E. Onaindia, A comprehensive framework for learning declarative action models, J. Artif. Intell. Res. 74 (2022) 1091–1123.

[8] B. Bonet, H. Geffner, Learning first-order symbolic representations for planning from the structure of the state space, in: Proc. of the 24th European Conference on Artificial Intelligence (ECAI 2020), 2020, pp. 2322–2329.

[9] M. Grand, D. Pellier, H. Fiorino TempAMLSI, Temporal action model learning based on grammar induction, in: Proceedings of the International Workshop of Knowledge Engineering for Planning and Scheduling (KEPS-ICAPS), 2021.

[10] S. Kambhampati, Model-lite planning for the web age masses: the challenges of planning with incomplete and evolving domain models, in: Proceedings of the National Conference on Artificial Intelligence (AAAI-07), vol. 22(2), 2007, pp. 1601–1604.

[11] H.H. Zhuo, Q. Yang, D.H. Hu, L. Li, Learning complex action models with quantifiers and logical implications, Artif. Intell. 174 (18) (2010) 1540–1569, https://doi.org/10.1016/j.artint.2010.09.007.

[12] D. Aineto, S. Jiménez, E. Onaindia, Learning action models with minimal observability, Artif. Intell. 275 (2019) 104–137, https://doi.org/10.1016/j.artint.2019.05.003.

[13] A. Garrido, S. Jimenez, Learning temporal action models via constraint programming, in: Proc. of the 24th European Conference on Artificial Intelligence (ECAI 2020), 2020, pp. 2362–2369.

[14] M. Krantz, A. Windmann, R. Heesch, L. Moddemann, O. Niggemann, On a uniform causality model for industrial automation, https://arxiv.org/abs/2209.09618, 2022.

[15] H.H. Zhuo, S. Kambhampati, Model-lite planning: case-based vs. model-based approaches, Artif. Intell. 246 (2017) 1–21.

[16] S. Jialin Pan, Q. Yang, A survey on transfer learning, IEEE Trans. Knowl. Data Eng. 22 (10) (2010) 1345–1359.

[17] S. Jiménez, T. De la Rosa, S. Fernández, F. Fernández, D. Borrajo, A review of machine learning for automated planning, Knowl. Eng. Rev. 27 (4) (2012) 433–467, https://doi.org/10.1017/S026988891200001X.

[18] L. Liu, S. Wang, B. Hu, Q. Qiong, D. Rosenblum, Learning structures of interval-based Bayesian networks in probabilistic generative model for human complex activity recognition, Pattern Recognit. 81 (2018) 545–561, https://doi.org/10.1016/j.patcog.2018.04.022.

[19] C.L. Baker, R. Saxe, J.B. Tenenbaum, Action understanding as inverse planning, Cognition 113 (2009) 329–349.

[20] M. Cashmore, A. Collins, B. Krarup, S. Krivic, D. Magazzeni, D. Smith, Towards explainable ai planning as a service, in: Proc. 2nd ICAPS Workshop on Explainable Planning (XAIP-2019), 2019, pp. 1–9.

[21] C. Lauretti, F. Cordella, A.L. Ciancio, E. Trigili, J.M. Catalan, F.J. Badesa, S. Crea, S.M. Pagliara, S. Sterzi, N. Vitiello, N. Garcia Aracil, L. Zollo, Learning by demonstration for motion planning of upper-limb exoskeletons, Front. Neurorobot. 12 (2018) 1–5, https://doi.org/10.3389/fnbot.2018.00005.

[22] R. Pereira, N. Oren, F. Meneguzzi, Landmark-based approaches for goal recognition as planning, Artif. Intell. 279 (2020) 1–49.

[23] S. Sohrabi, A. Riabov, O. Udrea, Plan recognition as planning revisited, in: Proc. International Joint Conference on AI (IJCAI-2016), 2016, pp. 3258–3264.

[24] J. Kucera, R. Barták, LOUGA: learning planning operators using genetic algorithms, in: Pacific Rim Knowledge Acquisition Workshop, PKAW-18, 2018, pp. 124–138.
[25] K. Mourão, L.S. Zettlemoyer, R.P.A. Petrick, M. Steedman, Learning STRIPS operators from noisy and incomplete observations, in: Conference on Uncertainty in Artificial Intelligence, UAI-12, 2012, pp. 614–623, https://arxiv.org/abs/1210.4889.
[26] H.H. Zhuo, S. Kambhampati, Action-model acquisition from noisy plan traces, in: International Joint Conference on Artificial Intelligence, IJCAI-13, 2013, pp. 2444–2450, https://dl.acm.org/doi/10.5555/2540128.2540479.
[27] S. Minaee, N. Kalchbrenner, E. Cambria, N. Nikzad, M. Chenaghlu, J. Gao, Deep learning based text classification: a comprehensive review, https://arxiv.org/pdf/2004.03705.pdf, 2020.
[28] G. Frances, B. Bonet, H. Geffner, Learning general planning policies from small examples without supervision, in: Proceedings of the National Conference on Artificial Intelligence (AAAI-2021), 2021, pp. 11801–11808.
[29] B. Bonet, H. Geffner, Language-based causal representation learning, https://arxiv.org/abs/2207.05259, 2022.
[30] B. Juba, H. Le, R. Stern, Safe learning of lifted action models, in: Proc. International Conference on Principles of Knowledge Representation and Reasoning (KR-2021), 2021, pp. 379–389.
[31] K. Apt, The essence of constraint propagation, Theor. Comput. Sci. 221 (1–2) (1999) 179–210.
[32] C. Bessiere, Constraint propagation, in: Foundations of Artificial Intelligence. Handbook of Constraint Programming, Elsevier, 2006, pp. 29–83.
[33] R. Dechter, Constraint Processing, Morgan Kaufmann, 2003.
[34] U. Dorndorf, E. Pesch, T. Phan-Huy, Constraint propagation techniques for the disjunctive scheduling problem, Artif. Intell. 122 (2000) 189–240.
[35] A. Garrido, Learning temporal action models from multiple plans: a constraint satisfaction approach, Eng. Appl. Artif. Intell. 108 (2022) 104590.
[36] S. Benson, Learning action models for reactive autonomous agents, Ph.D. thesis, Stanford University, 1996, https://dl.acm.org/doi/book/10.5555/892612.
[37] S. Muggleton, L. Raedt, Inductive logic programming: theory and methods, J. Log. Program. 19–20 (1994) 629–679, https://doi.org/10.1016/0743-1066(94)90035-3.
[38] G. Sablon, D. Boulanger, Using the event calculus to integrate planning and learning in an intelligent autonomous agent, in: Proc. of the Workshop on Planning Current Trends in AI Planning, 1994, pp. 254–265.
[39] C. Bessiere, R. Coletta, F. Koriche, B. O'Sullivan, Acquiring constraint networks using a SAT-based version space algorithm, in: Proc. of the AAAI Conference on Artificial Intelligence, 2006, pp. 1565–1568.
[40] M. Paulin, C. Bessiere, J. Sallantin, Automatic design of robot behaviors through constraint network acquisition, in: Proc. of IEEE International Conference on Tools with Artificial Intelligence (ICTAI'08), 2008, pp. 275–282.
[41] N. Beldiceanu, H. Simonis, A constraint seeker: finding and ranking global constraints from examples, in: Proc. of the Int. Conference on Principles and Practice of Constraint Programming (CP-2011), 2011, pp. 12–26.
[42] N. Beldiceanu, H. Simonis, A model seeker: extracting global constraint models from positive examples, in: Proc. of the Int. Conference on Principles and Practice of Constraint Programming (CP-2012), 2012, pp. 141–157.
[43] C. Bessiere, R. Coletta, E. Hebrard, G. Katsirelos, N. Lazaar, N. Narodytska, C. Quimper, T. Walsh, Constraint acquisition via partial queries, in: Proc. International Joint Conference on AI (IJCAI-2013), 2013, pp. 475–481, https://dl.acm.org/doi/10.5555/2540128.2540198.
[44] R. Arcangioli, C. Bessiere, N. Lazaar, Multiple constraint acquisition, in: Proc. International Joint Conference on AI (IJCAI-2016), 2016, pp. 698–704.
[45] A. Arora, H. Fiorino, D. Pellier, M. Métivier, S. Pesty, A review of learning planning action models, Knowl. Eng. Rev. 33 (2018).
[46] Y. Gil, Learning by experimentation: incremental refinement of incomplete planning domains, in: Proc. Int. Conference on Machine Learning, 1994, pp. 87–95.
[47] H.H. Zhuo, T.A. Nguyen, S. Kambhampati, Refining incomplete planning domain models through plan traces, in: International Joint Conference on Artificial Intelligence, IJCAI-13, 2013, pp. 2451–2458, https://dl.acm.org/doi/abs/10.5555/2540128.2540480.
[48] S.N. Cresswell, T. McCluskey, M. West, Acquiring planning domain models using LOCM, Knowl. Eng. Rev. 28 (2) (2013) 195–213, https://doi.org/10.1017/S0269888912000422.
[49] S. Cresswell, P. Gregory, Generalised domain model acquisition from action traces, in: Proc. Int. Conference on Automated Planning and Scheduling (ICAPS-2011), 2011, pp. 42–49, https://dl.acm.org/doi/abs/10.5555/3038485.3038492.
[50] R. Jilani, A. Crampton, D. Kitchin, M. Vallati, ASCoL: a tool for improving automatic planning domain model acquisition, in: Proc. Italian Association for Artificial Intelligence, 2015, pp. 438–451.
[51] P. Gregory, A. Lindsay, Domain model acquisition in domains with action costs, in: Proc. Int. Conference on Automated Planning and Scheduling (ICAPS-2016), 2011, pp. 149–157, https://dl.acm.org/doi/abs/10.5555/3038594.3038613.
[52] M. Asai, C. Muise, Learning neural-symbolic descriptive planning models via cube-space priors: the voyage home (to STRIPS), in: Proc. International Joint Conference on AI (IJCAI-2020), 2020, pp. 2676–2682.
[53] L. Lamanna, A. Gerevini, A. Saetti, L. Serafini, P. Traverso, On-line learning of planning domains from sensor data in PAL: scaling up to large state spaces, in: Proceedings of the National Conference on Artificial Intelligence (AAAI-2021), 2021, pp. 11862–11869.
[54] A. Occhipinti, B. Bonet, H. Geffner, Learning first-order symbolic planning representations that are grounded, in: Proc. ICAPS Workshop on Bridging the Gap Between AI Planning and Reinforcement Learning (PRL), 2022.
[55] T. Walsh, SAT v CSP, in: Proc. of the Int. Conference on Principles and Practice of Constraint Programming (CP-2000), 2000, pp. 441–456, https://link.springer.com/chapter/10.1007/3-540-45349-0_32.
[56] Y. Dimopoulos, K. Stergiou, Propagation in CSP and SAT, in: Proc. of the Int. Conference on Principles and Practice of Constraint Programming (CP-2006), 2006, pp. 137–151.
[57] A. Gerevini, A. Saetti, I. Serina, Planning through stochastic local search and temporal action graphs in LPG, J. Artif. Intell. Res. 20 (2003) 239–290, https://doi.org/10.1613/jair.1183.