Learning STRIPS action models with classical planning

Abstract

This paper presents a novel approach for learning STRIPS action models from examples that compiles this inductive learning task into classical planning. Interestingly, the compilation approach is flexible to different amounts of available input knowledge; the learning examples can range from a set of plans (with their corresponding initial and final states) to just a set of initial and final states (no intermediate action or state is given). What is more, the compilation accepts partially specified action models and can be used to validate whether certain observations of plan executions follow a given STRIPS action model, even if this model is not fully specified.

Introduction

Besides *plan synthesis* (Ghallab, Nau, and Traverso 2004; Geffner and Bonet 2013), planning action models are also useful for *plan/goal recognition* (Ramírez 2012). At both planning tasks, an automated planner is required to reason about action models that correctly and completely capture the possible world transitions. Unfortunately, building planning action models is complex, even for planning experts, and this knowledge acquisition task is a bottleneck that limits the potential of planning (Kambhampati 2007).

On the other hand, Machine Learning (ML) has shown to be able to compute a wide range of different kinds of models from examples (Michalski, Carbonell, and Mitchell 2013). The application of inductive ML to the learning of STRIPS action models (Fikes and Nilsson 1971), the vanilla action model for planning, is not straightforward though:

- The inputs to ML algorithms (the learning/training data) usually are finite vectors encoding the value of fixed features in a given set of objects. The input for learning planning action models are observations of plan executions (where each plan possibly has different length).
- The traditional output of ML algorithms is a scalar value (an integer, in the case of *classification* tasks, or a real value, in the case of *regression* tasks). When learning STRIPS action models the output is, for each action, the sets of preconditions, negative and positive effects, that define the possible state transitions.

Learning STRIPS action models is a well-studied problem with sophisticated algorithms, like ARMS (Yang, Wu, and Jiang 2007), SLAF (Amir and Chang 2008) or LOCM (Cresswell, McCluskey, and West 2013) that do not require full knowledge of the intermediate states traversed by the example plans. Motivated by recent advances on the synthesis of different kinds of generative models with classical planning (Bonet, Palacios, and Geffner 2009; Segovia-Aguas, Jiménez, and Jonsson 2016; 2017), this paper introduces an innovative approach for learning STRIPS action models that can be defined as a classical planning compilation. The compilation approach is appealing itself because it opens the door to the bootstrapping of planning action models but also because:

1. Is flexible to different amounts of available input knowledge. The learning examples can range from a set of plans (with their corresponding initial and final states) to just a set of initial and final states where no intermediate state or action is observed.

2. Accepts previous knowledge about the structure of the actions in the form of partially specified action models. In the extreme, the compilation could be used to validate whether observed plan executions are valid for a given STRIPS action model.

The first section of the paper presents the classical planning model, its extension to conditional effects (which is a requirement of the proposed compilation) and formalizes the STRIPS action model (the output of the addressed learning task). The second section formalizes the learning of STRIPS action models with regard to different amounts of available input knowledge. The third and fourth sections describe our approach for addressing the formalized learning tasks. Finally, the last sections report the data collected during the empirical evaluation of our approach, discuss the strengths and weaknesses of our approach and propose several opportunities for future research.

Background

This section defines the planning models used on this work and the output of the learning tasks addressed in the paper.

Classical planning

We use F to denote the set of *fluents* (propositional variables) describing a state. A *literal* l is a valuation of a fluent $f \in F$, i.e. either $l = f$ or $l = \neg f$. A set of literals L represents a partial assignment of values to fluents (WLOG we assume that L does not assign conflicting values to any fluent). We use $\mathcal{L}(F)$ to denote the set of all literal sets on F, i.e. all partial assignments of values to fluents.

A *state* s is a total assignment of values to fluents, i.e. $|s| = |F|$, so the size of the state space is $2^{|F|}$. Explicitly including negative literals $\neg f$ in states simplifies subsequent definitions but often, we will abuse notation by defining a state s only in terms of the fluents that are true in s, as is common in STRIPS planning.

A *classical planning frame* is a tuple $\Phi = \langle F, A \rangle$, where F is a set of fluents and A is a set of actions. Each action $a \in A$ comprises three sets of literals:

- $\mathsf{pre}(a) \subseteq \mathcal{L}(F)$, called *preconditions*, that defines the literals that must hold for the action to be applicable.
- $\mathsf{eff}^+(a) \subseteq \mathcal{L}(F)$, called *positive effects*, that defines the fluents set to true by the action application.
- $\mathsf{eff}^-(a) \subseteq \mathcal{L}(F)$, called *negative effects*, that defines the fluents set to false by the application of the action.

We say that an action $a \in A$ is *applicable* in state s iff $\mathsf{pre}(a) \subseteq s$, and the result of applying a in s is the *successor state* $\theta(s, a) = \{s \setminus \mathsf{eff}^-(a)) \cup \mathsf{eff}^+(a)\}$.

A *classical planning problem* is a tuple $P = \langle F, A, I, G \rangle$, where I is an initial state and $G \subseteq \mathcal{L}(F)$ is a goal condition. A *plan* for P is an action sequence $\pi = \langle a_1, \dots, a_n \rangle$ that induces a state sequence $\langle s_0, s_1, \dots, s_n \rangle$ such that $s_0 = I$ and, for each $1 \leq i \leq n$, a_i is applicable in s_{i-1} and generates the successor state $s_i = \theta(s_{i-1}, a_i)$. A plan π *solves* P iff $G \subseteq s_n$, i.e. if the goal condition is satisfied at the last state reached after following the application of π in I. We denote with $|\pi|$ the *plan length*.

Classical planning with conditional effects

Our approach for leaning STRIPS action models is compiling this leaning task into a classical planning task with conditional effects. Conditional effects allow us to compactly define actions whose effects depend on the current state. Many classical planners cope with conditional effects without compiling them away. In fact, supporting conditional effects is a requirement of IPC-2014 (Vallati et al. 2015) and IPC-2018.

Now an action $a \in A$ has a set of literals $\mathsf{pre}(a) \in \mathcal{L}(F)$, called the *precondition*, and a set of *conditional effects* $\mathsf{cond}(a)$. Each conditional effect $C \rhd E \in \mathsf{cond}(a)$ is composed of two sets of literals $C \in \mathcal{L}(F)$, the *condition*, and $E \in \mathcal{L}(F)$, the *effect*.

An action $a \in A$ is *applicable* in state s if and only if $\mathsf{pre}(a) \subseteq s$, and the resulting set of *triggered effects* is

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

i.e. effects whose conditions hold in s.

```
(:action stack
  :parameters (?x1 ?x2 - object)
  :precondition (and (holding ?x1) (clear ?x2))
  :effect (and (not (holding ?x1))
               (not (clear ?x2))
               (handempty) (clear ?x1)
               (on ?x1 ?x2)))
```

Figure 1: Example of a STRIPS operator schema that corresponds to the *stack* action from the *blocksworld* represented in PDDL.

The result of applying action a in a state s is the *successor* state $\theta(s, a) = \{s \setminus \mathsf{eff}_c^-(s, a)) \cup \mathsf{eff}_c^+(s, a)\}$ where $\mathsf{eff}_c^-(s, a) \subseteq triggered(s, a)$ and $\mathsf{eff}_c^+(s, a) \subseteq triggered(s, a)$ are the triggered negative and positive effects, respectively.

The STRIPS action schema and the *variable* objects

This work addresses the learning of PDDL action schemes that follow the STRIPS requirement (McDermott et al. 1998; Fox and Long 2003). Figure 1 shows the schema that corresponds to the *stack* action from a four-operator *blocksworld* (Slaney and Thiébaux 2001).

To formalize the output of the learning task, we assume that there exists a set of *predicates* Ψ and that fluents F are instantiated from these predicates, as in PDDL. Each predicate $p \in \Psi$ has an argument list of arity $ar(p)$. Given a set of objects Ω, the set of fluents F is then induced by assigning objects in Ω to the arguments of predicates in Ψ, i.e. $F = \{p(\omega) : p \in \Psi, \omega \in \Omega^{ar(p)}\}$ s.t. Ω^k is the k-th Cartesian power of Ω. Likewise, we assume that actions $a \in A$ are instantiated from STRIPS operator schemes.

Let $\Omega_v = \{v_i\}_{i=1}^{\max_{a \in A} ar(a)}$ be a new set of objects denoted as *variable names*, $\Omega \cap \Omega_v = \emptyset$, and that is bound by the maximum arity of an action in a given planning frame. For instance, in a three-block blocksworld $\Omega = \{block_1, block_2, block_3\}$ while $\Omega_v = \{v_1, v_2\}$ because the operators with the maximum arity, `stack` and `unstack`, have two parameters each.

Let us define also a new set of fluents F_v, $F \cap F_v = \emptyset$, that results from instantiating Ψ using only *variable objects* in Ω_v. This set defines the elements that can appear in an action schema. In the blocksworld F_v={`handempty`, `holding(`v_1`)`, `holding(`v_2`)`, `clear(`v_1`)`, `clear(`v_2`)`, `ontable(`v_1`)`, `ontable(`v_2`)`, `on(`v_1, v_1`)`, `on(`v_1, v_2`)`, `on(`v_2, v_1`)`, `on(`v_2, v_2`)`}.

We are now ready to define Ξ, a set of operator schema $\xi = \langle head(\xi), pre(\xi), add(\xi), del(\xi) \rangle$ such that:

- $head(\xi) = \langle name(\xi), pars(\xi) \rangle$, is the operator *header* defined by its name and $pars(\xi) = \{v_i\}_{i=1}^{ar(\xi)}$, an enumeration of the variable objects bound by the operator arity. The headers for a 4-operator blocksworld are: `pickup(`v_1`)`, `putdown(`v_1`)`, `stack(`v_1, v_2`)` and `unstack(`v_1, v_2`)`.
- The preconditions $pre(\xi) \subseteq F_v$, the negative effects $del(\xi) \subseteq F_v$, and the positive effects $add(\xi) \subseteq F_v$

such that, $del(\xi) \subseteq pre(\xi)$, $del(\xi) \cap add(\xi) = \emptyset$ and $pre(\xi) \cap add(\xi) = \emptyset$.

Learning STRIPS action models

Learning STRIPS action models from fully available input knowledge, i.e. from plans where the *pre-* and *post-states* of every action in a plan are available, is straightforward. In this case, because intermediate states are available, STRIPS operator schemes are derived lifting the literals that change between the pre and post-state of the corresponding action executions. Preconditions are derived lifting the minimal set of literals that appears in all the pre-states that correspond to the same operator schema.

This section formalizes more challenging learning tasks, where less input knowledge is available:

Learning from (initial, final) state pairs. This learning task corresponds to observing an agent acting in the world but watching only the results of its plan executions. No information about the actions in the plans is given. This learning task is formalized as $\Lambda = \langle \Psi, \Sigma \rangle$:

- Ψ is the set of predicates that define the abstract state space of a given planning domain.
- $\Sigma = \{\sigma_1, \ldots, \sigma_\tau\}$ is a set of labels s.t. each label $\sigma_t = (s_0^t, s_n^t)$, $1 \leq t \leq \tau$, is an $(initial, final)$ state pair that comprises the *final* state s_n^t resulting from executing an unknown plan π_t starting from a given *initial* state s_0^t.

Learning from labeled plans. Here we augment the amount of provided input knowledge with the actions executed by the observed agent and define the learning task as $\Lambda' = \langle \Psi, \Sigma, \Pi \rangle$:

- $\Pi = \{\pi_1, \ldots, \pi_\tau\}$ is a given set of example plans where each plan $\pi_t = \langle a_1^t, \ldots, a_n^t \rangle$, $1 \leq t \leq \tau$, is an action sequence that induces the corresponding state sequence $\langle s_0^t, s_1^t, \ldots, s_n^t \rangle$ such that, for each $1 \leq i \leq n$, a_i^t is applicable in the state s_{i-1}^t and generates the successor state $s_i^t = \theta(s_{i-1}^t, a_i^t)$.

Figure 2 shows an example of a learning task Λ' in the blocksworld. This learning task has a single learning example, $\Pi = \{\pi_1\}$ and $\Sigma = \{\sigma_1\}$, that corresponds to observing the execution of an eight-action plan ($|\pi_1| = 8$) for inverting a four-block tower.

Learning from partially specified action models. We may not require to start learning the STRIPS action models from scratch so we augment here the inputs of the learning task with partially specified operator schemes. In this case the leaning task is defined as $\Lambda'' = \langle \Psi, \Sigma, \Pi, \Xi_0 \rangle$ where:

- Ξ_0 is a partially specified action model in which some preconditions and effects are a priori known.

A solution to Λ is a set of operator schema Ξ that is compliant just with the predicates in Ψ, and the given set of initial and final states Σ. In this learning scenario, a solution must not only determine a possible STRIPS action model but also the plans π_t, $1 \leq t \leq \tau$ that explain the given labels Σ using the learned STRIPS model. A solution to Λ' is a set of STRIPS operator schema Ξ (with one schema

```
;;; Predicates in Ψ
(handempty) (holding ?o  - object)
(clear ?o - object) (ontable ?o - object)
(on ?o1 - object ?o2 - object)
```

```
;;; Plan π1
0: (unstack A B)
1: (putdown A)
2: (unstack B C)
3: (stack B A)
4: (unstack C D)
5: (stack C B)
6: (pickup D)
7: (stack D C)
```

;;; Label $\sigma_1 = (s_0^1, s_n^1)$

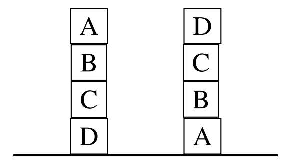

Figure 2: Example of a task for learning a STRIPS action model in the blocksworld.

$\xi = \langle head(\xi), pre(\xi), add(\xi), del(\xi) \rangle$ for each action with a different name in the example plans Π) compliant with the predicates in Ψ, the example plans Π, and their corresponding labels Σ. Finally a solution to Λ'' is a set of STRIPS operator schema Ξ compliant as well with the provided partially specified action model Ξ_0.

Learning STRIPS action models with planning

Our approach for addressing a learning task Λ, Λ' or Λ'', is compiling it into a classical planning task with conditional effects. The intuition behind the compilation is that a solution to the resulting classical planning task is a sequence of actions that:

1. Programs the STRIPS action model Ξ. The solution plan has a *prefix* that, for each $\xi \in \Xi$, determines the fluents from F_v that belong to its $pre(\xi)$, $del(\xi)$ and $add(\xi)$ sets.
2. Validates the programmed STRIPS action model Ξ using the given input knowledge (the set of labels Σ, and Π and/or Ξ_0 if available). For every label $\sigma_t \in \Sigma$, the solution plan has a postfix that produces a final state s_n^t starting from the corresponding initial state s_0^t using the programmed action model Ξ. We call this process the validation of the programmed STRIPS action model Ξ, at the learning example $1 \leq t \leq \tau$.

To formalize our compilations we first define $1 \leq t \leq \tau$ classical planning instances $P_t = \langle F, \emptyset, I_t, G_t \rangle$ that belong to the same planning frame (i.e. same fluents and actions and differ only in the initial state and goals). The fluents F are built instantiating the predicates in Ψ with the objects appearing in the input labels Σ. Formally $\Omega = \{o | o \in \bigcup_{1 \leq t \leq \tau} obj(s_0^t)\}$, where obj is a function that returns the set of objects that appear in a fully specified state. The set of actions, $A = \emptyset$, is empty because the action model is initially unknown. Finally, the initial state I_t is given by the state $s_0^t \in \sigma_t$ while goals G_t, are defined by the state $s_n^t \in \sigma_t$.

Now we are ready to formalize the compilations. We start with Λ, because this learning task requires the less input knowledge. Given a learning task $\Lambda = \langle \Psi, \Sigma \rangle$

the compilation outputs a classical planning task $P_\Lambda = \langle F_\Lambda, A_\Lambda, I_\Lambda, G_\Lambda\rangle$:

- F_Λ extends F with:
  - Fluents representing the programmed action model $pre_f(\xi)$, $del_f(\xi)$ and $add_f(\xi)$, for every $f \in F_v$ and $\xi \in \Xi$. If a fluent $pre_f(\xi)/del_f(\xi)/add_f(\xi)$ holds, it means that f is a precondition/negative effect/positive effect in the STRIPS operator schema $\xi \in \Xi$. For instance in the working example the preconditions of the $stack$ schema are represented by the fluents $\mathtt{pre\_holding\_stack\_}v_1$ and $\mathtt{pre\_clear\_stack\_}v_2$ set to true.
  - A fluent $mode_{prog}$ indicating whether the operator schemes are being programmed or they are being validated (already programmed).
  - Fluents $\{test_t\}_{1\leq t\leq\tau}$, indicating the learning example where the programmed action model is being validated.
- I_Λ contains the fluents from F that encode s_0^1 (the initial state of the first learning example), every $pre_f(\xi) \in F_\Lambda$ (our compilation assumes that initially any operator schema is programmed with every possible precondition, no negative effect and no positive effect) and $mode_{prog}$ set to true.
- $G_\Lambda = \bigcup_{1\leq t\leq\tau}\{test_t\}$, indicates that the programmed action model is validated in all the learning examples.
- A_Λ contains actions of three different kinds:
  1. The actions for *programming* an operator schema $\xi \in \Xi$, which includes:
     - Actions for **removing** a *precondition* $f \in F_v$ from the action schema $\xi \in \Xi$.

$$\begin{aligned}\mathsf{pre}(\mathsf{programPre}_{\mathsf{f},\xi}) =& \{\neg del_f(\xi), \neg add_f(\xi),\\ & mode_{prog}, pre_f(\xi)\},\\ \mathsf{cond}(\mathsf{programPre}_{\mathsf{f},\xi}) =& \{\emptyset\} \rhd \{\neg pre_f(\xi)\}.\end{aligned}$$

     - Actions for **adding** a *negative* or a *positive* effect $f \in F_v$ to the action schema $\xi \in \Xi$.

$$\begin{aligned}\mathsf{pre}(\mathsf{programEff}_{\mathsf{f},\xi}) =& \{\neg del_f(\xi), \neg add_f(\xi),\\ & mode_{prog}\},\\ \mathsf{cond}(\mathsf{programEff}_{\mathsf{f},\xi}) =& \{pre_f(\xi)\} \rhd \{del_f(\xi)\},\\ & \{\neg pre_f(\xi)\} \rhd \{add_f(\xi)\}.\end{aligned}$$

  2. The actions for *applying* an already programmed operator schema $\xi \in \Xi$ bound with the objects $\omega \subseteq \Omega^{ar(\xi)}$. We assume that the operators headers are known so the binding of the operator schema is done implicitly by order of appearance of the action parameters, i.e. the variables in $pars(\xi)$ are bound to the objects in ω appearing at the same position. Figure 3 shows the PDDL encoding of the action for applying a programmed operator $stack$.

$$\begin{aligned}\mathsf{pre}(\mathsf{apply}_{\xi,\omega}) =& \{pre_f(\xi) \implies p(\omega)\}_{\forall p\in\Psi, f=p(pars(\xi))},\\ \mathsf{cond}(\mathsf{apply}_{\xi,\omega}) =& \{del_f(\xi)\} \rhd \{\neg p(\omega)\}_{\forall p\in\Psi, f=p(pars(\xi))},\\ & \{add_f(\xi)\} \rhd \{p(\omega)\}_{\forall p\in\Psi, f=p(pars(\xi))},\\ & \{mode_{prog}\} \rhd \{\neg mode_{prog}\}.\end{aligned}$$

```
(:action apply_stack
  :parameters (?o1 - object ?o2 - object)
  :precondition
   (and (or (not (pre_on_stack_v1_v1)) (on ?o1 ?o1))
        (or (not (pre_on_stack_v1_v2)) (on ?o1 ?o2))
        (or (not (pre_on_stack_v2_v1)) (on ?o2 ?o1))
        (or (not (pre_on_stack_v2_v2)) (on ?o2 ?o2))
        (or (not (pre_ontable_stack_v1)) (ontable ?o1))
        (or (not (pre_ontable_stack_v2)) (ontable ?o2))
        (or (not (pre_clear_stack_v1)) (clear ?o1))
        (or (not (pre_clear_stack_v2)) (clear ?o2))
        (or (not (pre_holding_stack_v1)) (holding ?o1))
        (or (not (pre_holding_stack_v2)) (holding ?o2))
        (or (not (pre_handempty_stack)) (handempty)))
  :effect
   (and (when (del_on_stack_v1_v1) (not (on ?o1 ?o1)))
        (when (del_on_stack_v1_v2) (not (on ?o1 ?o2)))
        (when (del_on_stack_v2_v1) (not (on ?o2 ?o1)))
        (when (del_on_stack_v2_v2) (not (on ?o2 ?o2)))
        (when (del_ontable_stack_v1) (not (ontable ?o1)))
        (when (del_ontable_stack_v2) (not (ontable ?o2)))
        (when (del_clear_stack_v1) (not (clear ?o1)))
        (when (del_clear_stack_v2) (not (clear ?o2)))
        (when (del_holding_stack_v1) (not (holding ?o1)))
        (when (del_holding_stack_v2) (not (holding ?o2)))
        (when (del_handempty_stack) (not (handempty)))
        (when (add_on_stack_v1_v1) (on ?o1 ?o1))
        (when (add_on_stack_v1_v2) (on ?o1 ?o2))
        (when (add_on_stack_v2_v1) (on ?o2 ?o1))
        (when (add_on_stack_v2_v2) (on ?o2 ?o2))
        (when (add_ontable_stack_v1) (ontable ?o1))
        (when (add_ontable_stack_v2) (ontable ?o2))
        (when (add_clear_stack_v1) (clear ?o1))
        (when (add_clear_stack_v2) (clear ?o2))
        (when (add_holding_stack_v1) (holding ?o1))
        (when (add_holding_stack_v2) (holding ?o2))
        (when (add_handempty_stack) (handempty))
        (when (modeProg) (not (modeProg)))))
```

Figure 3: Action for applying an already programmed operator schema $stack$ as encoded in PDDL.

  3. The actions for *validating* the current learning example $1 \leq t \leq \tau$.

$$\begin{aligned}\mathsf{pre}(\mathsf{validate_t}) =& G_t \cup \{test_j\}_{j\in 1\leq j<t}\\ & \cup \{\neg test_j\}_{j\in t\leq j\leq\tau} \cup \{\neg mode_{prog}\},\\ \mathsf{cond}(\mathsf{validate_t}) =& \{\emptyset\} \rhd \{test_t\}.\end{aligned}$$

Lemma 1. *Any classical plan π that solves P_Λ induces an action model Ξ that solves the learning task Λ.*

Proof sketch. The compilation forces that once the preconditions of an operator schema $\xi \in \Xi$ are programmed, they cannot be altered. The same happens with the positive and negative effects that define an operator schema $\xi \in \Xi$ (besides they can only be programmed after the preconditions are programmed). Once operator schemes are programmed they can only be applied because of the $mode_{prog}$ fluent. To solve P_Λ, there is only one way of achieving goals $\{test_t\}$, $1 \leq t \leq \tau$: executing an applicable sequence of programmed operator schemes that reaches the final state s_n^t, defined in σ_t, starting from s_0^t. If this is achieved for all the input examples $1 \leq t \leq \tau$, it means that the programmed action model Ξ is compliant with the provided input knowledge and hence, it is a solution to Λ. $\square$

The compilation is *complete* in the sense that it does not discard any possible STRIPS action model.

Constraining the learning hypothesis space with additional input knowledge

Here we show that further input knowledge can be used to constrain the space of the possible action models and make the learning of STRIPS action models more practicable.

Labeled plans

Here we extend our compilation to address the learning scenario where a set of plans is available. Given a learning task $\Lambda' = \langle \Psi, \Sigma, \Pi \rangle$, the compilation outputs a classical planning task $P_{\Lambda'} = \langle F_{\Lambda'}, A_{\Lambda'}, I_{\Lambda'}, G_{\Lambda'} \rangle$ that extends P_Λ as follows:

- $F_{\Lambda'}$ extends F_Λ with $F_\Pi = \{plan(name(\xi), \Omega^{ar(\xi)}, j)\}$, the fluents to code the steps of the plans in Π, where $F_{\pi_t} \subseteq F_\Pi$ encodes $\pi_t \in \Pi$. Fluents at_j and $next_{j,j_2}$, $1 \leq j < j2 \leq n$, are also added to represent the current plan step and to iterate through the steps of a plan.
- $I_{\Lambda'}$ extends I_Λ with the fluents from F_Π that encode the plan $\pi_1 \in \Pi$, the fluents at_1 and $\{next_{j,j_2}\}$, $1 \leq j < j2 \leq n$, for indicating where to start validating the programmed action model. Goals are the same as in the previous compilation $G_{\Lambda'} = G_\Lambda = \bigcup_{1 \leq t \leq \tau} \{test_t\}$.
- With respect to $A_{\Lambda'}$.
  1. The actions for *programming* the preconditions/effects of a given operator schema $\xi \in \Xi$ are the same.
  2. The actions for *applying* an already programmed operator have an extra precondition $f \in F_\Pi$, that encodes the current plan step, and an extra conditional effects $\{at_j\} \rhd \{\neg at_j, at_{j+1}\}_{\forall j \in [1,n]}$ for advancing to the next plan step. This mechanism forces that these actions are only applied as in the example plans.
  3. The actions for *validating* the current learning example have an extra precondition, $at_{|\pi_t|}$, to indicate that the current plan π_t was fully executed and extra conditional effects to unload plan π_t and load the next plan π_{t+1}:

     $\{\emptyset\} \rhd \{\neg at_{|\pi_t|}, at_1\}, \{f\} \rhd \{\neg f\}_{f \in F_{\pi_t}}, \{\emptyset\} \rhd \{f\}_{f \in F_{\pi_{t+1}}}$.

Partially specified action models

Known preconditions and effects are encoded as fluents $pre_f(\xi)$, $del_f(\xi)$ and $add_f(\xi)$ set to true at the initial state $I_{\Lambda'}$. The corresponding programming actions ($\mathsf{programPre}_{\mathsf{f},\xi}$ and $\mathsf{programEff}_{\mathsf{f},\xi}$) become unnecessary and are removed from $A_{\Lambda'}$ making the classical planning task $P_{\Lambda'}$ easier to be solved.

To illustrate this, the classical plan of Figure 4 is a solution to a learning task $\Lambda'' = \langle \Psi, \Sigma, \Pi, \Xi_0 \rangle$ for getting the blocksworld action model where the schemes for `pickup`, `putdown` and `unstack` belong to Ξ_0 and are fully known in advance. This classical plan programs and validates the operator schema `stack` from the blocksworld (previously specified operator schemes for *pickup*, *putdown* and *unstack* are given), using the plan π_1 and label σ_1 shown in Figure 2. The plan steps $[0, 8]$ are the actions for programming the preconditions of the `stack` operator, steps $[9, 13]$ are the actions for programming the operator effects and steps $[14, 22]$ are actions for validating the programmed operators applying the actions in the plan π_1 shown in the Figure 2.

```
00 : (program_pre_clear_stack_v1)
01 : (program_pre_handempty_stack)
02 : (program_pre_holding_stack_v2)
03 : (program_pre_on_stack_v1_v1)
04 : (program_pre_on_stack_v1_v2)
05 : (program_pre_on_stack_v2_v1)
06 : (program_pre_on_stack_v2_v2)
07 : (program_pre_ontable_stack_v1)
08 : (program_pre_ontable_stack_v2)
09 : (program_eff_clear_stack_v1)
10 : (program_eff_clear_stack_v2)
11 : (program_eff_handempty_stack)
12 : (program_eff_holding_stack_v1)
13 : (program_eff_on_stack_v1_v2)
14 : (apply_unstack a b i1 i2)
15 : (apply_putdown a i2 i3)
16 : (apply_unstack b c i3 i4)
17 : (apply_stack b a i4 i5)
18 : (apply_unstack c d i5 i6)
19 : (apply_stack c b i6 i7)
20 : (apply_pickup d i7 i8)
21 : (apply_stack d c i8 i9)
22 : (validate_1)
```

Figure 4: Example of a plan for programming and validating the operator schema *stack* using the plan π_1 and label σ_1 shown in Figure 2 as well as previously specified operator schemes for *pickup*, *putdown* and *unstack*.

In the extreme, when a fully specified STRIPS action model Ξ is given, the compilation validates whether an observed plan follows the given model. In this case, if a solution plan is found to $P_{\Lambda'}$, it means that the given STRIPS action model is *valid* for the given set of examples. If $P_{\Lambda'}$ is unsolvable it means that the given STRIPS action model is invalid since it is not compliant with all the given examples. Tools for plan validation like VAL (Howey, Long, and Fox 2004) could also be used at this point.

Static predicates

A *static predicate* $p \in \Psi$ is a predicate that does not appear in the effects of any action schema (Fox and Long 1998). Therefore, one can get rid of the mechanism for programming these predicates as the effect of any action schema while keeping the compilation complete. Formally, given a static predicate p:

- Fluents $del_f(\xi)$ and $add_f(\xi)$, such that $f \in F_v$ is an instantiation of the static predicate p in the set of *variable objects* Ω_v, can be discarded for every $\xi \in \Xi$.
- Actions $\mathsf{programEff}_{\mathsf{f},\xi}$ (s.t. $f \in F_v$ is an instantiation of p in Ω_v) can also be discarded for every $\xi \in \Xi$.

Static predicates can also constrain the space of possible preconditions by looking at the given set of labels Σ. In particular one can assume that if a precondition $f \in F_v$ (s.t. $f \in F_v$ is an instantiation of a static predicate in Ω_v) is not compliant with the labels in Σ it means that is not possible and then, fluents $pre_f(\xi)$ and actions $\mathsf{programPre_{f,\xi}}$ can be discarded for every $\xi \in \Xi$. For instance in the *zenotravel* domain $pre\_next\_board\_v1\_v1$, $pre\_next\_debark\_v1\_v1$, $pre\_next\_fly\_v1\_v1$, $pre\_next\_zoom\_v1\_v1$, $pre\_next\_refuel\_v1\_v1$ can be discarded (as well as their corresponding programming actions) because a precondition `(next ?v1 ?v1 - flevel)` cannot be compliant with any state in Σ.

Likewise, static predicates can constrain the space of possible preconditions looking at the given set of example plans Π. Fluents $pre_f(\xi)$ and actions $\mathsf{programPre_{f,\xi}}$ are discardable for every $\xi \in \Xi$ if a precondition $f \in F_v$ (s.t. $f \in F_v$ is an instantiation of a static predicate in Ω_v) is not possible according to Π. Back to the *zenotravel* domain, if a example plan $\pi_t \in \Pi$ contains the action `(fly plane1 city2 city0 fl3 fl2)` and the corresponding label $\sigma_t \in \Sigma$ contains the static literal `(next fl2 fl3)` but does not contain `(next fl2 fl2)`, `(next fl3 fl3)` or `(next fl3 fl2)` the only possible precondition including the static predicate is $pre\_next\_fly\_v1\_v2$.

In the evaluation of our approach we do not assume that the set of static predicates is given but compute a set of *potential* static predicates from the input to the learning task.

Evaluation

This section evaluates the performance of our approach for learning STRIPS action models starting from different amounts of available input knowledge. The set of learning examples is fixed for all the experiments so we evaluate the effect of the amount of input knowledge in the quality of the learned models.

Setup. Experiments are run on an Intel Core i5 3.10 GHz x 4 with 4 GB of RAM. The domains used in the evaluation are IPC domains that satisfy the STRIPS requirement (Fox and Long 2003), taken from the PLANNING.DOMAINS repository (Muise 2016).

The planner. The classical planner we use to solve the instances that result from our compilations is MADAGASCAR (Rintanen 2014). We use MADAGASCAR because its ability to deal with planning instances populated with dead-ends. In addition, SAT-based planners can apply the actions for programming preconditions in a single planning step (in parallel) because these actions do not interact. Actions for programming action effects can also be applied in a single planning step reducing significantly the planning horizon.

The evaluation metrics. The quality of the learned models is quantified with the *precision* and *recall* metrics. These two metrics are frequently used in *pattern recognition*, *information retrieval* and *binary classification* and are more informative that simply counting the number of errors in the learned model or computing the *symmetric difference* between the learned and the reference model (Davis and Goadrich 2006).

	Pre		**Add**		**Del**			
	P	**R**	**P**	**R**	**P**	**R**	**P**	**R**
Blocks	0.33	0.33	0.60	0.50	0.33	0.33	0.42	0.39
Driverlog	1.0	0.43	0.30	1.0	1.0	0.75	0.77	0.73
Ferry	1.0	0.67	0.50	1.0	1.0	1.0	0.83	0.89
Floortile	0.67	0.40	0.50	0.40	1.0	0.40	0.72	0.40
Gripper	1.0	0.50	1.0	1.0	1.0	1.0	1.0	0.83
Miconic	0.0	0.0	0.33	0.50	0.0	0.0	0.11	0.17
Satellite	1.0	0.14	0.67	1.0	1.0	1.0	0.89	0.71
Transport	1.0	0.25	0.33	1.0	1.0	0.50	0.78	0.58
Zenotravel	-	-	-	-	-	-	-	-
Grid	-	-	-	-	-	-	-	-
	0.75	0.34	0.53	0.80	0.79	0.63	0.69	0.59

Table 1: *Precision* and *recall* learning from state pairs.

	Total time	Preprocess	Plan length
Blocks	0.73	0.00	69
Driverlog	0.98	0.04	104
Ferry	0.09	0.00	107
Floortile	6.42	0.15	126
Gripper	0.03	0.00	47
Miconic	0.04	0.00	68
Satellite	4.34	0.10	126
Transport	12.21	0.53	54
Zenotravel	-	-	-
Grid	-	-	-

Table 2: Planning times (secs) and plan length.

Intuitively precision gives a notion of *soundness* while recall gives a notion of the *completeness* of the learned models. Formally, $Precision = \frac{tp}{tp+fp}$, where tp is the number of true positives (predicates that correctly appear in the action model) and fp is the number of false positives (predicates appear in the learned action model that should not appear). On the other hand recall is formally defined as $Recall = \frac{tp}{tp+fn}$ where fn is the number of false negatives (predicates that should appear in the learned action model but are missing).

Learning from (initial,final) state pairs

Here we assess the performance of our learning approach when addressing learning tasks of the Λ kind. This is the configuration with the minimum amount of available input knowledge, where input plans are not available so the planner must determine as well the actions that satisfy the input labels. Table 1 summarizes the obtained results. Precision (**P**) and recall (**R**) are computed separately for the preconditions (**Pre**), positive effects (**Add**) and negative Effects (**Del**) while last two columns (and the last row) report averages.

In this scenario precision and recall have low values. This is not surprising because the learning hypothesis space is so low constrained that actions can be reformulated and still be compliant with the inputs (e.g. the blocksworld operator `stack` could be *learned* with the preconditions and effects of the `unstack` operator and vice versa). Table 2 reports the total planning time, the preprocessing time (in seconds) invested by MADAGASCAR to solve the classical planning in-

	Pre		Add		Del			
	P	**R**	**P**	**R**	**P**	**R**	**P**	**R**
Blocks	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Driverlog	1.0	0.43	0.67	0.86	1.0	0.86	0.89	0.71
Ferry	0.80	0.57	1.0	1.0	1.0	1.0	0.93	0.86
Floortile	0.52	0.68	0.64	0.82	0.83	0.91	0.66	0.80
Gripper	1.0	0.67	1.0	1.0	1.0	1.0	1.0	0.89
Miconic	0.75	0.33	0.50	0.50	0.75	1.0	0.67	0.61
Satellite	0.60	0.21	1.0	1.0	1.0	0.75	0.87	0.65
Transport	1.0	0.40	0.71	1.0	1.0	0.80	0.90	0.73
Visitall	1.0	0.50	1.0	1.0	1.0	1.0	1.0	0.83
Zenotravel	1.0	0.36	0.70	1.0	1.0	0.71	0.90	0.69
Hanoi	1.0	0.50	1.0	1.0	1.0	1.0	1.0	0.83
Grid	0.53	0.47	0.75	0.86	0.78	1.0	0.69	0.78
	0.85	0.51	0.83	0.92	0.95	0.92	0.88	0.78

Table 3: *Precision* and *recall* learning from labeled plans.

	Total time	Preprocess	Plan length
Blocks	0.04	0.00	72
Driverlog	0.10	0.06	127
Ferry	0.06	0.03	54
Floortile	2.42	1.64	168
Gripper	0.03	0.01	43
Miconic	0.06	0.03	57
Satellite	0.20	0.14	67
Transport	0.70	0.64	74
Visitall	0.21	0.15	40
Zenotravel	2.21	2.17	83
Hanoi	0.12	0.06	48
Grid	4.54	4.51	86

Table 4: Planning time, preprocessing time and plan length learning from labeled plans.

	Pre		Add		Del			
	P	**R**	**P**	**R**	**P**	**R**	**P**	**R**
Blocks	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Driverlog	1.0	0.50	0.75	0.86	1.0	0.71	0.92	0.69
Ferry	0.80	0.57	1.0	1.0	1.0	1.0	0.93	0.86
Floortile	0.68	0.68	0.89	0.73	1.0	0.82	0.86	0.74
Gripper	1.0	0.67	1.0	1.0	1.0	1.0	1.0	0.89
Miconic	0.89	0.89	1.0	0.75	0.75	1.0	0.88	0.88
Satellite	0.82	0.64	1.0	1.0	1.0	0.75	0.94	0.80
Transport	1.0	0.70	0.71	1.0	1.0	0.80	0.90	0.83
Visitall	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Zenotravel	1.0	0.71	0.70	1.0	1.0	0.71	0.90	0.81
Hanoi	0.75	0.75	1.0	1.0	1.0	1.0	0.92	0.92
Grid	0.73	0.65	1.0	0.86	0.75	0.86	0.83	0.79
	0.89	0.73	0.92	0.93	0.96	0.89	0.92	0.85

Table 5: The *precision* and *recall* exploiting static predicates.

	Total time	Preprocess	Plan length
Blocks	0.03	0.00	72
Driverlog	0.12	0.07	102
Ferry	0.04	0.03	54
Floortile	0.67	0.57	77
Gripper	0.01	0.00	43
Miconic	0.04	0.00	41
Satellite	0.18	0.12	60
Transport	0.56	0.51	60
Visitall	0.17	0.15	36
Zenotravel	1.03	1.00	65
Hanoi	0.09	0.06	39
Grid	3.39	3.35	72

Table 6: Planning time and plan length exploiting static predicates.

stances that result from our compilation as well as the number of actions in the solutions. Values for the *Zenotravel* and *Grid* domains are not reported since the compilation could not be solved within a time bound of 1000 seconds.

Learning from labeled plans

Here we assess the performance of our learning approach when addressing learning tasks of the Λ' kind where labeled plans are available. Tables 3 and 4 summarize the obtained results. Precision and recall values are now acceptable, except for the still low value of the preconditions recall, indicating there are many missing preconditions in the learned models. All the learning tasks are now solved in a few seconds time.

We repeat now the evaluation but exploiting potential static predicates that are computed from the set of labels Σ. The potential *static predicates* considered is the set of predicates s.t. every predicate instantiation appears unaltered in the initial and final states, for every $\sigma_t \in \Sigma$. These static predicates are used to constrain the space of possible action models as explained in the previous section. Tables 5 and 6 summarize the obtained results.

We can observe that identifying static predicates drives to models with larger precondition *recall*. This fact evidences that since many of the missing preconditions corresponded to static predicates, there were no incentive to learn these preconditions as they always hold (Gregory and Cresswell 2015). When static predicates are identified, the resulting compilation is much compact and produces smaller planning/instantiation times. Interestingly, one can identify the domains with static predicates by just looking at the reported plan length. In these domains some preconditions corresponding to static predicates are directly derived from the learning examples so less programming actions are required.

Learning from partially specified action models

Here we evaluate the ability of the compilation to support partially specified action models (including identification of static predicates). In this case, instead of learning the action models from scratch, the model of half of the actions is given as input of the learning task. Tables 7 and 8 summarize the obtained results for only the *unknown* actions (considering the *known* action models would mean reporting higher scores since *precision* and *recall* in the *known* models is 1.0). The tables confirm the trend: more input knowledge drives to better models and requires less planning time. Likewise smaller solution plans are required since it is only necessary to program half of the actions (the other half is input knowledge). *Visitall* is excluded from this evaluation because it only contains one action schema.

	Pre		Add		Del			
	P	R	P	R	P	R	P	R
Blocks	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Driverlog	1.0	0.71	1.0	1.0	1.0	1.0	1.0	0.91
Ferry	1.0	0.67	1.0	1.0	1.0	1.0	1.0	0.89
Floortile	0.75	0.60	1.0	0.80	1.0	0.80	0.92	0.73
Gripper	1.0	0.50	1.0	1.0	1.0	1.0	1.0	0.83
Miconic	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Satellite	1.0	0.57	1.0	1.0	1.0	1.0	1.0	0.86
Transport	1.0	0.75	1.0	1.0	1.0	1.0	1.0	0.92
Zenotravel	1.0	0.67	1.0	1.0	1.0	0.67	1.0	0.78
Grid	0.78	0.58	1.0	1.0	0.75	0.75	0.84	0.78
	0.95	0.71	1.0	0.98	0.98	0.92	0.98	0.87

Table 7: *Precision* and *recall* in the learned models starting from partially specified actions.

	Total time	Preprocess	Plan length
Blocks	0.07	0.01	54
Driverlog	0.06	0.03	59
Ferry	0.06	0.03	45
Floortile	0.43	0.42	55
Gripper	0.03	0.01	35
Miconic	0.03	0.01	34
Satellite	0.14	0.14	47
Transport	0.32	0.32	42
Zenotravel	1.10	1.09	44
Grid	3.18	3.15	52

Table 8: Planning time and plan length when starting from partially specified action models.

Past and future of learning action models

Back in the 90's various systems aimed learning operators mostly via interaction with the environment. LIVE captured and formulated observable features of objects and used them to acquire and refine operators (Shen and Simon 1989). OBSERVER updated preconditions and effects by removing and adding facts, respectively, accordingly to observations (Wang 1995). These early works were based on direct lifting of the observed states and supported by exploratory plans or external teachers.

Action model learning has also been studied in domains where there is partial or missing state observability. ARMS works when no or partial intermediate states are given. It defines a set of weighted constraints that must hold for the plans to be correct, and solves the weighted propositional satisfiability problem with a MAX-SAT solver (Yang, Wu, and Jiang 2007). Consequently, the action models output by ARMS may be inconsistent with some of the examples. SLAF also deals with partial observability (Amir and Chang 2008). Given a formula representing the initial belief state, a sequence of executed actions and the corresponding partially observed states, it builds a complete explanation of observations by models of actions through a CNF formula. The learning algorithm updates the formula of the belief state with every action and observation in the sequence. This update makes sure that the new formula represents all the transition relations consistent with the actions and observations. The formula returned at the end includes all consistent models, which can then be retrieved with additional processing.

LOCM only requires the example training plas as input without need for providing information about predicates or states (Cresswell, McCluskey, and West 2013). This makes LOCM be most likely the learning approach that works with the least information possible. The lack of available information is addressed by LOCM by exploting assumptions about the kind of domain model it has to generate. Particularly, it assumes a domain consists of a collection of objects (sorts) whose defined set of states can be captured by a parameterized Finite State Machine (FSM). The intuitive assumptions of LOCM yield a learning model heavily reliant on the kind of domain structure. The inability of LOCM to properly derive domain theories where the state of a sort is subject to different FSMs is later overcome by LOCM2 by forming separate FSMs, each containing a subset of the full transition set for the sort (Cresswell and Gregory 2011). LOP (LOCM with Optimized Plans (Gregory and Cresswell 2015)), the last contribution of the LOCM family, addresses the problem of inducing static predicates. Because LOCM approaches induce similar models for domains with similar structures, they face problems at generating models for domains that are only distinguished by whether or not they contain static relations (e.g. *blocksworld* and *freecell*). In order to mitigate this drawback, LOP applies a post-processing step after the LOCM analysis which requires additional information about the plans, namely a set of optimal plans to be used in the learning phase.

Compiling the learning task into a a classical planning task is a general and flexible solution as it allows to accommodate different amounts of input knowledge. This scheme opens up a path for solving further tasks other than learning action models or plan validation – when a fully specified action model Ξ is given. Thus, by replacing the learning examples of Π by sequences of states (observations), noisy or missing fluents could be introduced in the states, and so the learning task $\Lambda = \langle \Psi, \Sigma, \mathcal{O}, \Xi \rangle$, where $\mathcal{O}$ is the noisy or partial observations, would amount to a plan recognition task where the domain theory is given (Sohrabi, Riabov, and Udrea 2016). Furthermore, the compilation scheme can be extended to learn other type of generative models that comprise partially-specified solution plans like HTN models or full-specified solutions like FSM or behaviour trees.

Conclusions

The paper presented a novel approach for learning STRIPS action models from examples using classical planning. The approach is flexible to different amounts of available input knowledge and accepts partially specified action models. As far as we know, this is the first work on learning action models exclusively using an *off-the-shelf* classical planner. Recently, Stern and Juba proposed a classical planning compilation for learning action models but following the *finite domain* representation for the state variables and did not report experimental results since the compilation was not implemented.

Our evaluation shows that, when example plans are available, we can compute accurate action models from small sets of learning examples and investing small learning times (less than a second time in most of the domains). When action plans are not available, our approach is still able to produce action models compliant with the input information. In this case, since learning is not constrained by actions it can reformulate operators changing their semantics.

The size of the compiled classical planning instances depends on the number of input examples. Generating *informative* examples for learning planning action models is still a challenging open issue. Planning actions include preconditions that are only satisfied by specific sequences of actions, and often, with a low probability of being chosen by chance (Fern, Yoon, and Givan 2004). The success of recent algorithms for exploring planning tasks (Francés et al. 2017) motivates the development of novel techniques that autonomously collect the learning examples.

References

Amir, E., and Chang, A. 2008. Learning partially observable deterministic action models. *Journal of Artificial Intelligence Research* 33:349–402.

Bonet, B.; Palacios, H.; and Geffner, H. 2009. Automatic derivation of memoryless policies and finite-state controllers using classical planners. In *ICAPS*.

Cresswell, S., and Gregory, P. 2011. Generalised domain model acquisition from action traces. In *ICAPS*.

Cresswell, S. N.; McCluskey, T. L.; and West, M. M. 2013. Acquiring planning domain models using LOCM. *The Knowledge Engineering Review* 28(02):195–213.

Davis, J., and Goadrich, M. 2006. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd international conference on Machine learning*, 233–240. ACM.

Fern, A.; Yoon, S. W.; and Givan, R. 2004. Learning domain-specific control knowledge from random walks. In *ICAPS*, 191–199.

Fikes, R. E., and Nilsson, N. J. 1971. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence* 2(3-4):189–208.

Fox, M., and Long, D. 1998. The automatic inference of state invariants in tim. *Journal of Artificial Intelligence Research* 9:367–421.

Fox, M., and Long, D. 2003. Pddl2. 1: An extension to pddl for expressing temporal planning domains. *J. Artif. Intell. Res.(JAIR)* 20:61–124.

Francés, G.; Ramrez, M.; Lipovetzky, N.; and Geffner, H. 2017. Purely declarative action representations are overrated: Classical planning with simulators. In *International Joint Conference on Artificial Intelligence*.

Geffner, H., and Bonet, B. 2013. A concise introduction to models and methods for automated planning.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: theory and practice*. Elsevier.

Gregory, P., and Cresswell, S. 2015. Domain model acquisition in the presence of static relations in the LOP system. In *ICAPS*, 97–105.

Howey, R.; Long, D.; and Fox, M. 2004. Val: Automatic plan validation, continuous effects and mixed initiative planning using pddl. In *Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on*, 294–301. IEEE.

Kambhampati, S. 2007. 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*.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. Pddl-the planning domain definition language.

Michalski, R. S.; Carbonell, J. G.; and Mitchell, T. M. 2013. *Machine learning: An artificial intelligence approach*. Springer Science & Business Media.

Muise, C. 2016. Planning. domains. *ICAPS system demonstration*.

Ramírez, M. 2012. *Plan recognition as planning*. Ph.D. Dissertation, Universitat Pompeu Fabra.

Rintanen, J. 2014. Madagascar: Scalable planning with sat. *Proceedings of the 8th International Planning Competition (IPC-2014)*.

Segovia-Aguas, J.; Jiménez, S.; and Jonsson, A. 2016. Hierarchical finite state controllers for generalized planning. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, 3235–3241. AAAI Press.

Segovia-Aguas, J.; Jiménez, S.; and Jonsson, A. 2017. Generating context-free grammars using classical planning. In *International Joint Conference on Artificial Intelligence*.

Shen, W., and Simon, H. A. 1989. Rule creation and rule learning through environmental exploration. In *International Joint Conference on Artificial Intelligence*, 675–680.

Slaney, J., and Thiébaux, S. 2001. Blocks world revisited. *Artificial Intelligence* 125(1-2):119–153.

Sohrabi, S.; Riabov, A. V.; and Udrea, O. 2016. Plan recognition as planning revisited. In *International Joint Conference on Artificial Intelligence, IJCAI-16*, 3258–3264.

Stern, R., and Juba, B. 2017. Efficient, safe, and probably approximately complete learning of action models. *IJCAI*.

Vallati, M.; Chrpa, L.; Grzes, M.; McCluskey, T. L.; Roberts, M.; and Sanner, S. 2015. The 2014 international planning competition: Progress and trends. *AI Magazine* 36(3):90–98.

Wang, X. 1995. Learning by observation and practice: An incremental approach for planning operator acquisition. In *In Proceedings of the 12th International Conference on Machine Learning*, 549–557.

Yang, Q.; Wu, K.; and Jiang, Y. 2007. Learning action models from plan examples using weighted max-sat. *Artificial Intelligence* 171(2-3):107–143.