Learning STRIPS action models with classical planning

Diego Aineto and **Sergio Jiménez** and **Eva Onaindia**
Departamento de Sistemas Informáticos y Computación
Universitat Politécnica de Valéncia.
Camino de Vera s/n. 46022 Valencia, Spain
{dieaigar,serjice,onaindia}@dsic.upv.es

Abstract

This paper presents a novel approach for learning STRIPS action models from examples that compiles this inductive learning task into a classical planning task. 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 where neither actions nor intermediate states are given. What is more, partially specified action models can be used to feed the compilation and the compilation can also be used to validate whether certain observations of plan executions follow a given STRIPS action model.

Introduction

Besides *plan synthesis* (Ghallab, Nau, and Traverso 2004; Geffner and Bonet 2013), planning action models are also useful for *plan/goal recognition* (Ramírez Jávega 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 automated 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, the vanilla action model for planning (Fikes and Nilsson 1971), is not straightforward though:

- The inputs to ML algorithms (the learning data) usually are finite vectors that encode the value of fixed features for a given set of objects. The input for learning planning action models traditionally are sets of observations of plan executions (each with possibly a different length).
- The traditional output of off-the-shelf ML techniques 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 are, for each action, the sets of preconditions, negative and positive effects, that define the possible state transitions of a given planning domain.

The learning of 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 learning 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. A compilation approach is appealing because an off-the-shelf planner can be used to address the inductive learning task opening the door to the bootstrapping of action models in planning. In addition, a compilation approach presents the following contributions:

1. It 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 actions or intermediate states are observed.
2. Can exploit previous knowledge about the action model in the form of partially specified action models. In the extreme, it can be used to validate whether observed plan executions are valid for a given STRIPS action model.

The paper is organized as follows. The first section presents the classical planning model, its extension to conditional effects (which is a requirement for the proposed compilation) and formalizes the STRIPS action model, the output of the addressed learning task. The second section formalizes the task of learning STRIPS action models with regard to different amounts of input knowledge (observations of plan executions). The third section describes our approach for addressing the learning of STRIPS action models by compiling it into classical planning and how previous knowledge can be introduced in the compilation. The fourth section reports the collected data during the empirical evaluation of the compilation approach. Finally, the last section discusses the strengths and weaknesses of the compilation approach for learning action models and proposes some opportunities of future research.

Background

This section defines the planning models used on this work and the STRIPS action model, output of the addressed learning task.

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) and let $\neg L = \{\neg l : l \in L\}$ be the complement of L. 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 then 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 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 $a \in A$ has a set of literals $\mathsf{pre}(a) \in \mathcal{L}(F)$, called *preconditions*, a set of effects $\mathsf{eff}(a) \in \mathcal{L}(F)$. 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 a new state $\theta(s, a) = (s \setminus \neg\mathsf{eff}(a)) \cup \mathsf{eff}(a)$.

A *classical planning problem* is a tuple $P = \langle F, A, I, G\rangle$, where I is an initial state and $G \in \mathcal{L}(F)$ 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 $s_i = \theta(s_{i-1}, a_i)$. The plan π *solves* P if and only if $G \subseteq s_n$, i.e. if the goal condition is satisfied in the state reached after following the application of π in I.

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 particular effects depend on the current state. Many classical planners cope with conditional effects without compiling them away. In fact, the support of conditional effects was a requirement for participating at the IPC-2014 (Vallati et al. 2015).

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

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

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

The STRIPS action model

This work addresses the learning action schema in the STRIPS language (Fikes and Nilsson 1971). To define the output of this learning task, we assume that the fluents in F are instantiated from predicates, as in PDDL (McDermott et al. 1998; Fox and Long 2003). There exists a set of predicates Ψ, each $p \in \Psi$ with 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 are instantiated from a STRIPS operator schema. Figure 1 shows the STRIPS operator schema that corresponds to the *stack* action from the blocksworld (Slaney and Thiébaux 2001) represented in PDDL.

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

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

Let Ω_v be a new set of objects, $\Omega \cap \Omega_v = \emptyset$, that represents *variable names*. The cardinality of this set, $|\Omega_v|$, is given by the maximum arity of an action in a given planning frame Φ. For instance, in a three-block blocksworld $\Omega = \{block_1, block_2, block_3\}$ while $\Omega_v = \{v_1, v_2\}$ because the operators `stack` and `unstack` are the ones with the maximum arity (two parameters each). Let us define also a new set of fluents F_v that results instantiating Ψ but using only the *variable objects* in Ω_v. 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 a STRIPS action model Ξ as 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 corresponding action name and a enumeration of the variable names bound by the arity of the operator, $pars(\xi) \in \Omega_v^{ar(\xi)}$. 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. a set of plans where the *pre-* and *post-states* of every action in a plan are available, is straightforward. In this case, a STRIPS operator schema is derived by lifting the literals that change between the pre and post-state of the corresponding action executions. Likewise, the 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 tasks, for learning STRIPS action models, where less input knowledge is available. Next, these learning tasks are formalized according to the available amount of input knowledge.

Learning from labeled plans

This learning task is formalized as $\Lambda = \langle \Psi, \Pi, \Sigma \rangle$:

- Ψ, the set of predicates that define the abstract state space of a given planning domain.
- $\Pi = \{\pi_1, \ldots, \pi_\tau\}$, the given set of example plans s.t. 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 $s_i^t = \theta(s_{i-1}^t, a_i^t)$.
- $\Sigma = \{\sigma_1, \ldots, \sigma_\tau\}$, is a set of labels s.t. each plan π_t, $1 \leq t \leq \tau$, has an associated label $\sigma_t = (s_0^t, s_n^t)$ where s_n^t is the state resulting from executing the plan π_t starting from the state s_0^t.

To illustrate this, Figure 2 shows an example of a task for learning the STRIPS action model of the blocksworld. Figure 2 shows the content of Ψ, $\Pi = \{\pi_1\}$ and $\Sigma = \{\sigma_1\}$, this learning task has a single learning example, that corresponds to observing a plan execution for inverting a tower of four blocks.

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

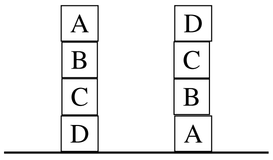

Figure 2: Example of a task for learning the STRIPS action model of the blocksworld from a single learning example.

A solution to a learning task Λ is a set of STRIPS operator schema Ξ (with one schema $\xi = \langle head(\xi), pre(\xi), add(\xi), del(\xi) \rangle$ for each action with a different name in the example plans) that is compliant with the predicates in Ψ, the example plans Π, and their labels Σ.

Learning from initial/final states

Here we reduce the amount of input knowledge provided to the learning task. First, $\Pi = \{\pi_1, \ldots, \pi_\tau\}$ is replaced by $\Pi' = \{|\pi_1|, \ldots, |\pi_\tau|\}$ i.e. Π' that does not contain a set of plans but the number of actions of each plan. The learning task is hence redefined as $\Lambda' = \langle \Psi, \Pi', \Sigma \rangle$. While the previous learning task, Λ, corresponds to watching an agent acting in the world, this new learning task Λ' can be understood as watching only the result of its plan execution but knowing the number of actions performed by the agent. In the case of the learning example shown in Figure 2, $|\pi_1| = 8$.

Finally, we go one step further and redefine a third learning task $\Lambda'' = \langle \Psi, \Sigma \rangle$ that corresponds to watching only the results of the plan executions. In this case no information about the executed plans is given. A solution to the Λ'' learning task is a set of operator schema Ξ that is compliant only with the predicates in Ψ, and the given set of initial and final states Σ.

In these two cases Λ' and Λ'', a solution must not only determine a possible STRIPS action model but also the plans that explain the given labels using the learned model (this information about the actions is no longer given in the learning examples). However, we assume that the headers of the operators schema are known.

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 P_Λ, $P_{\Lambda'}$ or $P_{\Lambda''}$. The intuition behind these compilations is that a solution to the resulting classical planning task is a sequence of actions that:

1. Programs the STRIPS action model Ξ. For each $\xi \in \Xi$, the solution plan determines the literals that belong to the $pre(\xi)$, $del(\xi)$ and $add(\xi)$ sets.
2. Validates the programmed the STRIPS action model Ξ using the given input knowledge (the given set of labels Σ, and Π if available). For every label $1 \leq t \leq \tau$, then Ξ is used to produce a final state s_n^t starting from the corresponding initial state s_0^t. We call this process the validation of the programmed STRIPS Ξ at the learning example $1 \leq t \leq \tau$. If information about the corresponding plan π_t is available, it is used to constrain the validation of Ξ.

Figure 3 shows a classical plan for solving a learning task of the Λ class. This classical plan programs and validates the operator schema `stack`, from the blocksworld, using the plan π_1 and label σ_1 shown in Figure 2. In particular the steps $[0, 8]$ are the actions for programming the preconditions of the operator, steps $[9, 13]$ are the actions for programming the operator effects and steps $[14, 22]$ are the actions for validating the programmed operator. The details of the compilation are given in the remaining of the section.

To formalize our compilations we first define $1 \leq t \leq \tau$ classical planning instances $P_t = \langle F, \emptyset, I_t, G_t \rangle$, one for each

```
 0 : (program_pre_clear_stack_v1)
 1 : (program_pre_handempty_stack)
 2 : (program_pre_holding_stack_v2)
 3 : (program_pre_on_stack_v1_v1)
 4 : (program_pre_on_stack_v1_v2)
 5 : (program_pre_on_stack_v2_v1)
 6 : (program_pre_on_stack_v2_v2)
 7 : (program_pre_ontable_stack_v1)
 8 : (program_pre_ontable_stack_v2)
 9 : (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 3: Example of a plan for programing and validating the operator schema $unstack$ using the plan π_1 and label σ_1 shown in Figure 2 as well as previously specified operator schema for $pickup$, $putdown$ and $unstack$.

leaning example, that belong to the same planning frame (i.e. same fluents and actions and differ only in the initial state and goals). The set of fluents F is built instantiating the predicates in Ψ with the set of different objects appearing in the input learning examples. 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 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 our compilations for learning STRIPS action models with classical planning. We start with Λ'' because it is the learning task that requires the smallest amount of input knowledge, and incrementally extend the formalized compilation until addressing Λ, the learning task with the largest amount of input knowledge.

Given a learning task $\Lambda'' = \langle \Psi, \Sigma \rangle$ the compilation outputs a classical planning task $P_{\Lambda''} = \langle F_{\Lambda''}, A_{\Lambda''}, I_{\Lambda''}, G_{\Lambda''} \rangle$ such that:

- $F_{\Lambda''}$ 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 $\xi \in \Xi$.
  - A fluent $mode_{prog}$ indicating whether the operator schema are being programmed or they are already programmed operators that are being validated in the learning examples.
  - Fluents $\{test_t\}_{1 \leq t \leq \tau}$, indicating the learning example where the programmed action model is being validated.
- $I_{\Lambda''}$, contains the fluents from F that encode s_0^1 (i.e. the initial state of the first learning example), every $pre_f(\xi) \in F_\Lambda$ (note that our compilation assumes that initially, any operator schema, contains all the possible preconditions, no negative effect and no positive effect) and the $mode_{prog}$ set to true.
- $G_{\Lambda''} = \{test_t\}, 1 \leq t \leq \tau$, indicates that the programmed action model is validated in all the learning examples.
- $A_{\Lambda''}$ contains actions of three different kinds:
  1. The actions for programming the operator schema which includes:
     - The actions for removing a *precondition* $f \in F_v$ from the action schema $\xi \in \Xi$.

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

     - The 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 $\xi \in \Xi$ bounded with the objects $\omega \subseteq \Omega^{ar(\xi)}$

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

     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 in the same position. To illustrate this, Figure 4 shows the PDDL encoding of the action for applying an already programmed operator schema $stack$.
  3. The actions for updating the learning example $1 \leq t \leq \tau$ where the programmed action model is validated.

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

Lemma 1. *Any classical plan π that solves $P_{\Lambda''}$ induces an action model Ξ that solves the learning task Λ''.*

Proof sketch. Once the preconditions of an operator schema Ξ 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 operator preconditions are programmed, because of the preconditions in the $programPre_{f,\xi}$ actions). Furthermore, once the operator schema are programmed they can only be applied because of the $mode_{prog}$ fluent. To solve the classical planning task $P_{\Lambda''}$, there is only one way of achieving a goal $\{test_t\}$,$1 \leq t \leq \tau$ that is by executing an applicable sequence of programmed operator schema that achieves the final state s_n^t defined by the associated label σ_t, starting from the initial s_0^t defined in this same label. If this is achieved for all the input examples of the learning task Λ'' (i.e. for all the labels σ_t, $1 \leq t \leq \tau$), it means that the programmed action model Ξ is compliant with all the input knowledge and hence, it is a solution to Λ''. □

```
(: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 4: Action for applying an already programmed operator schema *stack* as encoded in PDDL.

Constraining the hypothesis space with previous action models

Interestingly, the compilation does not require to start the learning of STRIPS action models from scratch and accepts partially specified operator schema where some preconditions are effects are a priori known.

The known preconditions and effects can be encoded as fluents $pre_f(\xi)$, $del_f(\xi)$ and $add_f(\xi)$ set to true that are part of the initial state $I_{\Lambda''}$. Likewise, the corresponding programming actions ($\mathsf{programPre}_{\mathsf{f},\xi}$ and $\mathsf{programEff}_{\mathsf{f},\xi}$) can be removed from $A_{\Lambda''}$ reducing the number of applicable actions and making the classical planning task $P_{\Lambda''}$ easier to be solved. With this regard, the classical plan of Figure 3 is a solution to a learning task for getting the blocksworld action model where the operator schema for `pickup`, `putdown` and `unstack` are known in advance.

In the extreme, the compilation can also be used to validate whether an observed plan follows a given STRIPS action model. In this case the action model to validate is coded in the initial state $I_{\Lambda''}$, any programmed action is removed from $A_{\Lambda''}$ and the $mode_{prog}$ fluent is set to false. If a solution plan is found to $P_{\Lambda''}$ it means that the given STRIPS action model is *valid* for the given examples. If $P_{\Lambda''}$ is unsolvable it means that the given STRIPS action model is invalid since it cannot satisfy the given examples. Tools for plan validation like VAL (Howey, Long, and Fox 2004) can also be used at this point but note that the compilation approach does not require a plan so the validation of an STRIPS action model can be done with just an initial and a final state.

Constraining the hypothesis space with example plans

Here we extend our compilation to address the learning scenario defined by Λ where, a set of plans Π is available so it is used to reduce the space of possible action models. Given a learning task $\Lambda = \langle \Psi, \Pi, \Sigma \rangle$, 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_Λ extends $F_{\Lambda''}$ by including also the fluents $F_\Pi = \{plan(name(\xi), j, \Omega^{ar(\xi)})\}$ to code the j steps of a plan in Π, with $F_{\Pi_t} \subseteq F_\Pi$ encoding $\pi_t \in \Pi$. We denote π_t a solution plan to a classical planning instance P_t. In addition fluents at_j and $next_{j,j_2}$, $1 \leq j < j2 \leq n$, are also added to represent the plan steps where the programmed action model is validated.
- I_Λ is extended with the fluents from F_{Π_1} that encode the plan $\pi_1 \in \Pi$ for solving P_1, and 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 $G_\Lambda = G''_\Lambda$ so again require that the programmed action model is validated in all the learning examples.
- With respect to A_Λ.

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 operator have an extra precondition $f \in F_{\Pi_t}$ that encodes the current plan step and extra conditional effects $\{at_j\} \rhd \{\neg at_j, at_{j+1}\}_{\forall j \in [1,n]}$ for advancing the plan step.
3. The actions for updating the active test have an extra precondition, $at_{|\Pi_t|}$, to indicate that the current plan Π_t was fully executed and extra conditional effects to load

the next plan Π_{t+1}:
$\{f\} \rhd \{\neg f\}_{f \in F_{\Pi_t}}, \{\emptyset\} \rhd \{f\}_{f \in F_{\Pi_{t+1}}}, \{\emptyset\} \rhd \{\neg at_{|\pi_t|}, at_1\}$.

Evaluation

We evaluated our learning approach for different amounts of available input knowledge. In all the experiments the classical planning instances resulting from our compilations are addressed by the SAT-based classical planner MADAGASCAR (Rintanen 2014) given its ability to deal with planning instances populated with dead-ends.

The domains used in the evaluations are domains taken from different IPCs that satisfy the STRIPS requirement.

Validating action models with classical planning

Learning action models from example plans

Here we assess the performance of our learning approach when addressing the Λ learning task. The quality of the learned models is quantified using the *precision* and *recall* metrics frequently used in *pattern recognition* (Davis and Goadrich 2006). This metrics are computed in the learned models with respect to the actual models taken as reference. Table 1 summarizes the obtained results.

	Pre		**Add**		**Del**	
	P	**R**	**P**	**R**	**P**	**R**
Blocks	1.0	1.0	1.0	1.0	1.0	1.0
Visitall	1.0	0.5	1.0	1.0	1.0	1.0
Gripper	1.0	0.6	1.0	1.0	1.0	1.0
Miconic	1.0	0.3	1.0	1.0	1.0	1.0
Floortile	0.2	0.2	0.8	0.5	1.0	0.9
Pegsol	-	-	-	-	-	-

Table 1: The *precision* and *recall* values achieved by the learned models in the different domains.

Table 2 reports the planning time, and prepossessing time used to solve the classical planning task that results from our compilation as well as the plan length of the found solutions.

Domain	Total time	Preprocess	Plan length
Blocks	0.06	0.01	72
Visitall	0.43	0.40	37
Gripper	0.03	0.00	43
Miconic	0.12	0.09	59
Floortile	0.70	0.57	141
Pegsol	-	-	-

Table 2: Planning time, preprocessing time and Plan length.

Learning action models from example states

When the set of input planning task is not available, the planner determines also the actions that must satistying the input labels so actions can reformulated and still be compliant with the learning examples. For instance a blocksworld can be learned where the operator `stack` is defined with the preconditions and effects of the `unstack` operator and vice versa.

Related work

The LIVE system (Shen and Simon, 1989) was an extension of the General Problem Solver (GPS) framework (Ernst and Newell, 1969) with a learning component. LIVE alternated problem solving with model learning to automatically define operators. The decision about when to alternate depended onsurprises, that is situations where an action effects violated its predicted model. EXPO (Gil, 1992) generated plans with the PRODIGY system (Minton, 1988), monitored the plans execution, detected differences in the predicted and the observed states and constructed a set of specific hypotheses to fix those differences. Then the EXPO filtered the hypotheses heuristically. OBSERVER (Wang, 1994) learned operators by monitoring expert agents and applying the version spaces algorithm (Mitchell, 1997) to the observations. When the system already had an operator representation, the preconditions were updated by removing facts that were not present in the new observations pre-state; the effects were augmented by adding facts that were in the observations delta-state. All of these early works were based on direct liftings of the observed states. They also benefit from experience beyond simple interaction with the environment such as exploratory plans or external teachers, but none provided a theoretical justification for this second source of knowledge. The work recently reported in (Walsh and Littman, 2008) succeeds in bounding the number of interactions the learner must complete to learn the preconditions and effects of a STRIPS action model. This work shows that learning STRIPS operators from pure interaction with the environment, can require an exponential number of samples, but that limiting the size of the precondition lists enable sample-efficient learning (polynomial in the number of actions and predicates of the domain). The work also proves that efficient learning is also possible without this limit if an agent has access to an external teacher that can provide solution traces on demand.

Others systems have tried to learn more expressive action models for deterministic planning in fully observable environments. Examples would include the learning of conditional costs for AP actions (Jess Lanchas and Borrajo, 2007) or the learning of conditional effects with quantifiers (Zhuo et al., 2008).

In addition action model learning has been studied in domains where there is partial state observability. ARMS uses the same kind od learning examples but assumes the exampls are given and proceeds in two phases. In the first phase, ARMS extracts frequent action sets from plans that share a common set of parameters. ARMS also finds some frequent literal-action pairs with the help of the initial state and the goal state that provide an initial guess on the actions preconditions, and effects. In the second phase, ARMS uses the frequent action sets and literal-action pairs to define a set of weighted constraints that must hold in order to make the plans correct. Then, ARMS solves the resulting weighted MAX-SAT problem and produces action models from the solution of the SAT problem. This process iterates until all actions are modelled. For a complex planning domain that involves hundreds of literals and actions, the corresponding weighted MAX-SAT representation is likely to be too large to be solved efficiently as the number of clauses can reach up to tens of thousands. For that reason ARMS implements a hill-climbing method that models the actions approximately. Consequently, the ARMS output is a model which may be inconsistent with the examples.

(Amir and Chang, 2008) introduced an algorithm that tractably generates all the STRIPS-like models that could have lead to a set of observations. Given a formula representing the initial belief state, a sequence of executed actions and the corresponding observed states(where partial observations of states are given), it builds a complete explanation of observations by models of actions through a Conjunctive Normal Form (CNF) formula. By linking the possible states of fluents to the effect propositions in the action models, the complexity of the CNF encoding can be controlled to find exact solutions efficiently in some circumstances. 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. Unlike the previous approaches, the one described in (Mour ao et al., 2008) deals with both missing and noisy predicates in the observations. For each action in a given domain, they use kernel perceptrons to learn predictions of the domain properties that change because of the action execution. LOCM (Cresswellet al., 2009) induces action schemas without being provided with any information about initial, goal or intermediate state descriptions for the example action sequences. LOCM receives descriptions of plans or plan fragments, uses them to create states machines for the different domain objects and extracts the action schemas from these state machines.

(Stern and Juba 2017).

Conclusions

The paper presented a novel approach for learning STRIPS action models from examples exclusively using classical planning. Interestingly the approach is flexible to different amounts of available input knowledge and accepts partially specified action models.

The empirical evaluation shows that, when example plans are available, our approach can compute accurate action models. 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 change the semantics of the operators. An interesting research direction related to this issue is *domain reformulation* to use actions in a more efficient way, reduce the set of actions identifying dispensable information or exploitin features that allow more compact solutions like the *reachable* or *movable* features in the Sokoban domain.

With regard to efficiency, the size of the compiled classical planning instances depend on the number of input examples. On the other hand the empirical results show that our approach, since is based on inferences instead of statistics, is able to generate non-trivial models from very small data sets.

This research opens the door to the bootstrapping of action model. That is that a planner collects by its own the data that it is necessary to define an action model for a given planning task. With this regard, collecting *informative* examples for learning planning action models is 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 recent success of recent algorithms for exploring planning tasks (Francés et al. 2017) motivates the development of novel approaches that do not assume that a learning set of plans is given apriori but instead, autonomously collect the learning examples.

Acknowledgment

Diego Aineto is partially supported by the *FPU* program funded by the Spanish government. Sergio Jiménez is partially supported by the *Ramon y Cajal* program funded by the Spanish government.

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. 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. 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.

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.

Ramírez Jávega, 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*.

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

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.

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.