Learning STRIPS action models with classical planning

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

Abstract

This paper presents a novel approach for learning STRIPS action models from examples that compiles the inductive learning task into a classical planning task. Our compilation for this learning task is flexible to different amounts of available input knowledge; it accepts partially specified action models and what is more, the input learning examples can range from a set of plans (with their corresponding initial and final states) to only a set of initial and final states where no action is observed.

Introduction

Besides plan synthesis, planning action models are also useful for plan/goal recognition (**?**). In both tasks, off-the-shelf planners require reasoning about action models that correctly and completely capture the possible world transitions (**?**; **?**). Unfortunately, building such planning action models is complex, even for planning experts, so this knowledge acquisition bottleneck limits the potential of automated planning (**?**).

On the other hand, Machine Learning (ML) techniques are able to compute a wide range of different kinds of models from examples (**?**). However, the application of inductive ML techniques for learning planning action models is not straightforward. First, the inputs to ML algorithms usually are sets of finite numeric vectors encoding the value of sets of objects features. The input for learning planning action models traditionally are sets of observations of plan executions (each with possibly different length). Second, 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). In the case of learning STRIPS action models, the output is not a scalar but a model of the preconditions and the effects of each action that defines the possible state transitions in the given planning domain.

Learning STRIPS action models is a well-studied problem with sophisticated algorithms, like ARMS (**?**), SLAF (**?**) or LOCM (**?**) that do not require full knowledge of all the states traversed by the example plans. Motivated by recent advances on learning generative models with classical planning (**?**; **?**; **?**) this paper introduces an innovative approach for learning classical planning action models that (1) it can be defined as a classical planning compilation and (2), it is flexible to different amounts of available input knowledge.

Background

This section defines the planning models used on this work.

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. $l = f$ 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 then a total assignment of values to fluents, i.e. $|s| = |F|$, so the size of the state space $2^{|F|}$. Explicitly including negative literals $\neg f$ in states simplifies subsequent definitions, but we often 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 positive effects $\mathsf{eff}^+(a) \in \mathcal{L}(F)$, and a set of negative 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 $\theta(s, a) = (s \setminus \mathsf{eff}^-(a)) \cup \mathsf{eff}^+(a)$.

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 $\pi = \langle a_1, \ldots, a_n \rangle$ that induces a state sequence $\langle s_0, s_1, \ldots, s_n \rangle$ 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 following the application of π in I.

In this work we assume that the fluents in F are instantiated from predicates, as in PDDL (**?**; **?**). 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 each action in A is instantiated from an STRIPS operator schema. Figure 1 shows the *stack* operator schema for a STRIPS blocksworld represented in PDDL.

```
(:action stack
  :parameters (?x1 ?x2)
  :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 *stack* operator schema for a STRIPS blocksworld represented in PDDL.

Let Ω_v be a new set of objects, $\Omega \cap \Omega_v = \emptyset$, that represent *variable names*. $|\Omega_v|$ is given by the action with the maximum arity in a planning frame. For instance, in a three-block blocksworld $\Omega = \{block_1, block_2, block_3\}$ and $\Omega_v = \{v_1, v_2\}$ because the operators `stack` and `unstack` are the ones with the maximum arity (two parameters each).

Let us define a new set of fluents F_v that results instantiating Ψ but using only the *variable objects* 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 operator schema as a tuple $\xi = \langle head(\xi), pre(\xi), add(\xi), del(\xi)\rangle$:

- $head(\xi) = \langle name(\xi), pars(\xi)\rangle$, represents an operator *header* defined by its corresponding action name and a list of variables, $pars(\xi) \in \Omega_v^{ar(\xi)}$. The headers for the blocksworld operators 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 positive effects $add(\xi) \subseteq F_v$, and the negative effects $del(\xi) \subseteq F_v$ such that, $del(\xi) \subseteq pre(\xi)$, $del(\xi) \cap add(\xi) = \emptyset$ and $pre(\xi) \cap add(\xi) = \emptyset$.

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. We use conditional effects because they 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, the support of PDDL conditional effects was a requirement for participating at IPC-2014 (**?**).

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 $\theta(s, a) = (s \setminus \mathsf{eff}^-(s, a)) \cup \mathsf{eff}^+(s, a)$, where $\mathsf{eff}^-(s, a)$ and $\mathsf{eff}^+(s, a)$ are the negative and positive effects in $\mathsf{eff}(s, a)$.

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, the operators schema are derived lifting the literals that change between the pre and post-state of the corresponding action executions. Likewise, preconditions are derived lifting the minimal set of literals appearing in all the pre-states that correspond to the same operator.

This section formalizes more challenging tasks, for learning STRIPS action models, where less input knowledge is available. Next we formalize these learning tasks according to the available 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. This set includes the predicates for defining the headers of the operators schema.
- $\Pi = \{\pi_1, \ldots, \pi_\tau\}$, the given set of example plans.
- $\Sigma = \{\sigma_1, , \ldots, \sigma_\tau\}$, a set of labels s.t. each plan π_t, $1 \leq t \leq \tau$, has a label $\sigma_t = (s_0^t, s_n^t)$ where s_n^t is the state resulting from executing π_t starting from the state s_0^t.

A solution to the learning task Λ is a set of operator schema Ξ (one schema for each operator header) 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. Now $\Pi = \{\pi_1, \ldots, \pi_\tau\}$ 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, so the learning task is 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 results of its actions executions knowing the number of diferent actions performed by the agent.

Finally, we can go a 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 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, a solution must not only synthesize the action models but also the actions that produced the given labels (this information is no longer given in the learning examples).

Learning STRIPS action models with classical planning

Our approach for addressing a learning task Λ, Λ' or Λ'', is to compile it into a classical planning task 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. For each $\xi \in \Xi$, determines the literals that belong to the sets $pre(\xi)$, $del(\xi)$ and $add(\xi)$.
2. Validates the programmed action model with the given set of labels $\Sigma = \{\sigma_1, \ldots, \sigma_\tau\}$. For every $1 \leq t \leq \tau$, the programmed action model Ξ is used to produce a final state s_n^t starting from their corresponding initial state s_0^t.

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. share the 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 objects appearing in the example plans or their corresponding labels. Formally $\Omega = \{o | o \in \bigcup_{1 \leq t \leq \tau} obj(\pi_t) \cup obj(s_0^t) \cup obj(s_n^t)\}$. The set of actions is empty, $A = \emptyset$, i.e. the action model is unkown. 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 using classical planning with conditional effects. We start with Λ'' because it is the learning task that requires the least 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 of the operator ξ.
  - Fluents $\{test_t\}_{1 \leq t \leq \tau}$, indicating the example where the programmed model is being validated.
  - Fluents $mode_{prog}$ and $mode_{val}$ indicating whether the operator schema are being programmed or the programmed operators are 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 fluent $pre_f(\xi) \in F_\Lambda$ (initially all operators have all the possible preconditions) and $mode_{prog}$.
- $G_{\Lambda''} = \{test_t\}$,$1 \leq t \leq \tau$, indicates that the programmed action model is validated in all the learning examples.
- $A_{\Lambda''}$ contains actions of three different types:
  1. The actions for programming the operator schema. This includes the actions for removing a *precondition* $f \in F_v$ from the action schema $\xi \in \Xi$.

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

  The actions for adding a *negative* or *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 $\xi \in \Xi$ bounding it with objects $\omega \subseteq \Omega^{ar(\xi)}$

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

  For instance, these actions define that if an operator is programmed with the precondition $holding(v_1) \in F_v$ it *implies* ($\implies$) that $holding(block_1) \in F$ has to be true in the current state if the operator binds the variable object $v_1 \in \Omega_v$ with the regular object $block_1 \in \Omega$. The operator binding is done implicitly, i.e. variables in $pars(\xi)$ are bound to the objects in ω appearing in the same position.
  3. The actions for changing the learning example $1 \leq t \leq \tau$ where the programmed action model is validated.

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

Lemma 1. *Any classical plan π that solves $P_{\Lambda''}$ induces a valid action model 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 (besides they can only be programmed after the operator preconditions are programmed). Furthermore, once an operator schema is programmed it can only be applied. The only way of achieving a goal $\{test_t\}$,$1 \leq t \leq \tau$ is by executing an applicable sequence of programmed operator schema that achieves the final state defined by the associated label σ_t, starting from the initial defined in that label. If this is done for all the input examples of the learning task (for all the labels), it means that the programmed action model Ξ is compliant with the learning input knowledge and hence, it is a solution to the action model learning task. □

Interestingly, the compilation is valid for partially specified action models since known preconditions and effects (fluents $pre_f(\xi)$, $del_f(\xi)$ and $add_f(\xi)$) can be part of the initial state $I_{\Lambda''}$ and the corresponding programming actions ($\mathsf{programPre}_{\mathsf{f},\xi}$ and $\mathsf{programEff}_{\mathsf{f},\xi}$) be removed from $A_{\Lambda''}$ making the classical planning task $P_{\Lambda''}$ easier.

Table 1: Mean error and standard deviation of the learned models.

Constraining the hypothesis space with example plans

Here we extend our compilation to address the learning scenario defined by Λ and Λ' where a set of plans Π (or at least its lengths in the case of Λ') is available. We denote $\pi_t \in \Pi$ as a solution plan to the classical planning instances P_t introduced above.

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 includes the fluents $F_\Pi = \{plan(name(\xi), j, \Omega^{ar(\xi)})\}$ to code the j steps of the plans in Π with $F_{\Pi_t} \subseteq F_\Pi$ encoding the t^{th} plan. In addition fluents at_j and $next_{j,j_2}$, $1 \leq j < j2 \leq n$, represent the plan step where the programmed action model is validated (n is the max length of a plan in Π).
- 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. $G_\Lambda = G''_\Lambda$.
- With respect to A_Λ, the actions for programming the preconditions/effects of a given operator are the same.
  1. 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.
  2. The actions for changing 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} where the programmed operators are validated:

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

Evaluation

The performance of our learning approach is evaluated for different degrees of available input knowledge and using different sources for collecting this input knowledge.

Learning action models from example plans

Here we assess the performance of our learning approach for addressing the Λ learning task. The evaluation is done using the cardinality of the *symmetric difference* sets that are computed between the set of preconditions, del and add effects (1), in the learned model and (2), in the actual models. In all the experiments the compilation is solved using the SAT-based classical planner MADAGASCAR (**?**).

Table 1 shows the mean error and standard deviation of the learned models with respect to the actual action models. The standard deviation provides a measure of how this error is distributed among the different operators in the domain. If this deviation is 0 it means that is equally distributed in all the domain operators.

Learning action models from example states

When the set of input planning task is not available the *symmetric difference* is not a good measure for evaluating the quality of the learned models. In this situation the planner determines the actions for satistying the input labels so actions can be completely interpreted in totally different way.

Conclusions

The paper presented a novel approach for learning STRIPS action models from examples exclusively using classical planning. The evaluation show that when example plans are available the approach allows to learn action models that are very close to the models used as reference. Interestingly when action plans are not available the results of the learning task are action models that satisfy the input knowledge but that is not constrained to the reference model. An interesting research direction related to this problem is then study of domain reformulation using actions in a different way, reducing the set of actions or learning features that allow more compact solutions like the *reachable* or *movable* features in the Soybean domain. Learning action models from examples allows the reformulation of a domain theory.

With regard to the efficiency of the compilation the size of the compilation output depends also on the number of examples. Empirical results show that our approach is able to generate non-trivial models from very small data sets.

Last but not least, 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 (**?**). In addition, motivated by the success of recent algorithms for exploring planning tasks (**?**), we do not assume that a learning set of plans is given apriori but instead, we autonomously collect the learning examples.