Bootstrapping classical planning action models

Guillem Francés
Information and Communication Technologies
Universitat Pompeu Fabra
Roc Boronat 138, 08018 Barcelona, Spain
@upf.edu

Sergio Jiménez
Computing and Information Systems
University of Melbourne
Parkville, Victoria 3010, Australia
sjimenez@unimelb.edu.au

Nir Lipovetzky
Computing and Information Systems
University of Melbourne
Parkville, Victoria 3010, Australia
@unimelb.edu.au

Miguel Ramírez
Computing and Information Systems
University of Melbourne
Parkville, Victoria 3010, Australia
@unimelb.edu.au

Abstract

This paper presents a novel approach for learning classical planning action models from minimal input knowledge and exclusively using existing classical planners. First, the paper defines a classical planning compilation to learn action models from examples. The compilation is flexible to different degrees of available input knowledge; it accepts partially specified action models and 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. Second, the paper explains how to autonomously collect informative examples using a classical planner based on pure exploratory search.

Introduction

Off-the-shelf planners reason about action models that correctly and completely capture the possible world transitions (**?**). In addition planning action models allow to effectively address further tasks than plan synthesis, like plan/-goal recognition (**?**). Unfortunately building planning action models is complex, even for planning experts, limiting the potential of automated planning (**?**).

Despite many Machine Learning (ML) techniques are able to compute models from examples (**?**) its application to learning planning action models is not straightforward. First, the inputs to ML algorithms usually are finite numeric vectors encoding objects features while, in planning, the input includes actions so is closer to the representation of procedures and behaviours rather than to object representation. 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 planning action models the output is not a scalar but a model of the possible state transitions. Last but not least, collecting *informative* examples for learning planning action models is challenging. Planning actions include preconditions that are only satisfied by specific sequences of actions, and often, with a low probability of being chosen by chance (**?**).

Learning classical planning 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 (**?**) and on the effective exploration of planning state spaces (**?**), this paper introduces an innovative approach for learning classical planning action models that offers a two-fold contribution:

1. An inductive learning algorithm that minimizes the required input knowledge, that is flexible to different degrees of available input knowledge, and that can be defined as a classical planning compilation.
2. A method for collecting *informative* learning examples using an exploration-based classical planner.

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 $2^{|F|}$.

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 $\mathsf{add}(a) \in \mathcal{L}(F)$, and a set of negative effects $\mathsf{del}(a) \in \mathcal{L}(F)$. 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 $\theta(s, a) = (s \setminus \neg\mathsf{del}(a)) \cup \mathsf{add}(a)$.

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 fluents F are instantiated from predicates, as in PDDL (**?**). There exists a set of pred-

icates 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^n is the n-th Cartesian power of Ω.

Likewise we assume that actions in A are instantiated from operator schema. Let $\Omega_v = \{v_1, \ldots, v_v\}$, $\Omega_v \cap \Omega = \emptyset$ be a new set of objects representing variable names defined by the action with the maximum arity in a planning frame. For instance, in the blocksworld $\Omega_v = \{v_1, v_2\}$ since operators *stack* (see Figure 1) and *unstack* have two parameters. Let us define a new set of fluents F_v that results instantiating Ψ but using only the objects in Ω_v, e.g. for the blocksworld example $F_v = \{handempty, holding(v_1), holding(v_2), clear(v_1), clear(v_2), ontable(v_1), ontable(v_2), on(v_1, v_1), on(v_1, v_2), on(v_2, v_1), on(v_2, v_2)\}$.

```
(: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 the *stack* planning operator schema from the blocksworld as represented in PDDL.

We are ready to define an operator schema $\xi \in \Xi$ as a tuple $\langle head(\xi), pre(\xi), add(\xi), del(\xi)\rangle$ where:

- $head(\xi)$, represents the operator *header* defined by a pair $head(\xi) = (name(\xi), pars(\xi))$ built using an action name and a list of variables, $pars(\xi) \in \Omega_v^{ar(\xi)}$. The blocksworld only requires these four operator headers $\{pickup(v_1), putdown(v_1), stack(v_1, v_2), unstack(v_1, v_2)\}$.
- 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

Conditional effects make it possible to repeatedly refer to the same action while their precise effects depend on the current state. 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 $\theta(s, a) = (s \setminus \neg\mathsf{eff}(s, a)) \cup \mathsf{eff}(s, a)$.

Learning classical planning action models

Learning classical planning 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: The operators schema 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 appearing in all the pre-states that correspond to the same operator.

This section formalizes more challenging tasks, for learning classical planning action model, where less input knowledge is available. Formalization is done 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.
- $\Pi = \{\pi_1, \ldots, \pi_t\}$ is the given set of example plans,
- $\Sigma = \{\sigma_1, , \ldots, \sigma_t\}$ is a set of labels s.t. each plan π_i, $1 \leq i \leq t$, has an associated label $\sigma_i = (s_i, s_i')$ where s_i' is the state resulting from executing π_i starting from the state s_i.

A solution to the learning task Λ is a set of operator schema Ξ, with one schema for each operator header, compliant with the predicates in Ψ, the example plans Π, and their labels Σ.

Learning from states

Here we reduce the amount of input knowledge provided to the previous learning task. Now each $\Pi = \{\pi_1, \ldots, \pi_t\}$ does not contain a set of plans but the number of actions of each plan, $\Pi' = \{|\pi_1|, \ldots, |\pi_t|\}$ and redefine the learning task as $\Lambda' = \langle \Psi, \Pi', \Sigma\rangle$. While the previous task can correspond to watching an agent acting in the world, this learnig tasks can be understood as watching only the results of its actions executions knowing the number of diferent actions performed by the agent.

We can go a step further and redefine the learning task as $\Lambda'' = \langle \Psi, \Sigma\rangle$. This learnig tasks corresponds to watching only the results of the plan executions. Now 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.

Learning action models from example states using a classical planner

Our approach for addressing $\Lambda'' = \langle \Psi, \Sigma\rangle$ is compiling this learning task into a classical planning task P_Λ. The intuition behind the compilation is that a solution to P_Λ is a sequence of actions that first, programs the action model (i.e. the $pre(\xi)$, $del(\xi)$ and $add(\xi)$ for each $\xi \in \Xi$) and then, validates the programmed action model in the given set of labels Σ, one after the other.

To formalize the compilation we define t classical planning instances $P_1 = \langle F, A, I_1, G_1\rangle, \ldots, P_t =$

$\langle F, A, I_t, G_t \rangle$, that belong to the same planning frame (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 objects in $\Omega = \{o | o \in s_i \cup s'_i, 1 \leq i \leq t\}$, the set of objects that appear in the fluents $f \in F$ used to define the states in Σ. The set of actions is empty $A = \emptyset$. Finally the initial state I_i, $1 \leq i \leq t$, is given by the state $s_i \in \sigma_i$ while goals G_i are defined by the state $s'_i \in \sigma_i$.

Now we are ready to define the compilation for learning action models using an off-the-shelf classical planner. Given a learning task $\Lambda' = \langle \Psi, \Sigma \rangle$ the compilation outputs a classical planning task $P_\Lambda = \langle F_\Lambda, A_\Lambda, I_\Lambda, G_\Lambda \rangle$ where:

- 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 $\xi \in \Xi$.
  - Fluents $\{test_i\}_{1 \leq i \leq t}$, indicating the example where the programmed model is currently being validated.
  - Fluents $prog1$, $prog2$ and $exec$ indicating whether the solution is programming the preconditions of the action schema, the effects, or it started validating the programmed action models.
- I_Λ, contains the fluents from F that encode the initial state $s_1 \in P_1$, every fluent $pre_f(\xi) \in F_\Lambda$ and $prog1$.
- $G_\Lambda = \{test_i\}$,$1 \leq i \leq t$, indicates that the programmed action model is validated in all the examples in Σ.
- A_Λ replaces the actions in A with actions of three types:
  1. Actions for programming:
     - *Precondition* $f \in F_v$ in the action schema $\xi \in \Xi$:

$$\begin{aligned} \mathsf{pre}(\mathsf{programPre}_{\mathsf{f},\xi}) &= \{pre_f(\xi), \neg prog2, \neg exec\}, \\ \mathsf{cond}(\mathsf{programPre}_{\mathsf{f},\xi}) &= \{\emptyset\} \rhd \{\neg pre_f(\xi)\}. \end{aligned}$$

     - *Negative effect* $f \in F_v$ in the action schema $\xi \in \Xi$:

$$\begin{aligned} \mathsf{pre}(\mathsf{programDel}_{\mathsf{f},\xi}) &= \{pre_f(\xi), \neg del_f(\xi), \neg add_f(\xi), \neg exec\}, \\ \mathsf{cond}(\mathsf{programDel}_{\mathsf{f},\xi}) &= \{\emptyset\} \rhd \{del_f(\xi), \\ &\qquad \{\emptyset\} \rhd \{prog2\}. \end{aligned}$$

     - *Positive effect* $f \in F_v$ in the action schema $\xi \in \Xi$:

$$\begin{aligned} \mathsf{pre}(\mathsf{programAdd}_{\mathsf{f},\xi}) &= \{\neg pre_f(\xi)), \neg del_f(\xi)), \neg add_f(\xi), \neg exec\}, \\ \mathsf{cond}(\mathsf{programAdd}_{\mathsf{f},\xi}) &= \{\emptyset\} \rhd \{add_f(\xi), \\ &\qquad \{\emptyset\} \rhd \{prog2\}. \end{aligned}$$

  2. Actions for applying an already programmed operator schema $\xi \in \Xi$ bounding it with objects $\omega \subseteq \Omega^{ar(\xi)}$

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

     For instance, these actions define that if an operator is programmed with precondition $holding(v_1) \in F_v$ it implies that $holding(block_1) \in F$ has to be true in the current state if the operator binds variable object $v_1 \in \Omega_v$ with object $block_1 \in \Omega_v$. 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. Actions for changing the active example where the action model is currently being validated.

$$\begin{aligned} \mathsf{pre}(\mathsf{validate_i}) &= G_i \cup \{test_j\}_{j \in 1 \leq j < i} \cup \{exec\}, \\ \mathsf{cond}(\mathsf{validate_i}) &= \{\emptyset\} \rhd \{test_i\}. \end{aligned}$$

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

Proof sketch. Once an operator schema is programmed it cannot be modified and can only be applied. The only way of achieving a *test* fluent is by applying a sequence of programmed operator schema until achieving the goal state defined by its associated label starting from the initial state of the corresponding label. If this is done for all the labels (all the input examples) it means that the programmed action model is compliant with the learning input knowledge and hence, it is a solution to the action model learning task. □

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_Λ. The approach allows also transfer learning where the action model generated for a given sub-task is encoded as *already programmed* for learning new action models in more challenging tasks.

Learning action models from example plans using a classical planner

The compilation can be extended to the learning scenario defined by Λ and Λ' in which a set of plans Π (or only its lengths in the case of Λ') is available. Each plan $\pi_i \in \Pi$, $1 \leq i \leq t$, is a solution to the corresponding classical planning instance $P_i = \langle F, A, I_i, G_i \rangle$ defined above. The compilation extensions are:

- F_Λ includes the new set of fluents $F_\Pi = \{plan(name(\xi), j, \Omega^{ar(\xi)})\}$ for encoding the j steps of the $1 \leq i \leq t$ plans in Π with $F_{\Pi_i} \subseteq F_\Pi$ the fluents encoding the plan corresponding to the i^{th} example (only for the Λ case). 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 the plan step where to start validating the programmed action model. Goals G_Λ are like in the original compilation.
- With respect to the actions in A_Λ,
  1. The actions for programming the preconditions/effects of a given operator schema are the same.

2. The actions for applying an operator schema have an extra precondition $f \in F_{\Pi_i}$ that encodes the current plan step and extra conditional effect $\{at_j\} \rhd \{\neg at_j, at_{j+1}\}_{\forall j \in [1,n]}$ for advancing the plan step.
3. The actions for changing the active test have an extra precondition, $at_{|\Pi_i|}$, to indicate that we simulated the full current plan Π_i and extra conditional effects to load the next plan Π_{i+1} where to validate the programmed action model:

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

Generating informative learning examples

TBD.

Observation: If a predicate is not appearing either in the initial not in the final state it will not appear in the learned action model.

Question: Can we propose a search algorithm that generates states with the full diversity of predicates of a given domain? What about exploring the full state space for small problems (3-4 blocks blocksworld)?

Evaluation

Learning action models from example plans

The performance of our learning approach is evaluated for different degrees of available input knowledge and using different sources for collecting this input knowledge. In all the cases we assess the performance of our learning approach 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 when (1) using *hand-picked* examples, (2) examples collected using the classical *planner* FAST-DOWNWARD (**?**) and (3) examples collected *randomly*. 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

TBD.

Related work

Conclusions

This paper presents a novel approach for learning classical planning action models from minimal input knowledge and using exclusively existing classical planners. Learning action models from examples allows the reformulation of a domain theory. An interesting research direction is the study of domain reformulation using features that allow more compact solutions like the *reachable* or *movable* features in the Sokoban domain.

	Preconditions			Del Effects			Add Effects		
	Hand	Planner	Rand	Hand	Planner	Rand	Hand	Planner	Rand
Blocks	0	41.67 ± 36.32		0	16.70 ± 3.36		0	19.38 ± 19.40	
Grippper	17.78 ± 13.70	40.00 ± 20.00		0	9.17 ± 0.83		0	74.17 ± 15.83	
Miconic	54.17 ± 27.32	61.67 ± 2.89		0	8.33 ± 14.43		0	29.17 ± 18.16	
Visitall	0	100.00		0	20.00		0	0.00	

Table 1: Mean error and standard deviation of the learned models when using hand-picked examples and examples collected using the classical planner Fast-Downward.