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 using exclusively existing classical planners. First, the paper defines a classical planning compilation to learn action models from just pairs of initial and final states. Second, the paper explains how to 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 (Geffner and Bonet 2013). Building such models is complex even for planning experts (Kambhampati 2007).

In Machine Learning (ML) models are not hand-coded but computed from examples (Michalski, Carbonell, and Mitchell 2013). Unfortunately, the application of off-the-shelf ML techniques to learning planning action models is not straightforward. On the one hand ML examples typically represent objects encoded as an assignment of values to a finite set of features while planning tasks include actions and are more related to procedures or behaviours, than objects. On the other hand, 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) while classical planning tasks are traditionally defined with declarative generative models. In addition the collection of *informative* examples for learning planning action models is complex. Planning actions include preconditions that are only satisfied by specific sequences of actions and often with a low probability of being chosen by chance. Therefore simple exploration approaches, s.t. random walks, easily undersample planning state spaces (Fern, Yoon, and Givan 2004).

This work focuses on learning action models for classical planners. This is a well-studied problem where the dynamics of a given action can be captured lifting the literals that change between the pre and post-state of an action execution. There are sophisticated learning approaches for this task, like ARMS (Yang, Wu, and Jiang 2007) or LOCM (Cresswell, McCluskey, and West 2013) systems that do not require full knowledge of the states traversed by the example plans. In this work aims going one step beyond these systems and study the task of learning classical planning action models when the only available information is pairs of initial and final states.

Motivated by recent advances on effective exploration of planning state spaces () and on the learning of complex structures with classical planners (Segovia-Aguas, Jiménez, and Jonsson 2017), this paper introduces an innovative approach for learning classical planning action models. The contribution of this work are two-fold:

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

Background

Here we define the planning models we use 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 a total assignment of values to fluents, i.e. $|s| = |F|$, so the number of states is $2^{|F|}$.

Under this formalism, 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 the *precondition*, a set of positive effects $\mathsf{add}(a) \in \mathcal{L}(F)$ and a set of del 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 $\theta(s, a) = (s \setminus \neg\mathsf{del}(a)) \cup \mathsf{add}(a)$.

Given a planning frame $\Phi = \langle F, A\rangle$, a *classical planning problem* is a tuple $P = \langle F, A, I, G\rangle$, where I is an initial state and G is a goal condition, i.e. a set of literals on F. 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 i such that $1 \leq i \leq n$, 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.

We assume that fluents are instantiated from predicates, as in PDDL (Fox and Long 2003). Specifically, there exists a set of predicates Ψ, and 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. $F = \{p(\omega) : p \in \Psi, \omega \in \Omega^{ar(p)}\}$ where, given a set X, X^n is the n-th Cartesian power of X.

Likewise we assume actions in $a \in A$ are instantiated from operator schema Ξ, i.e. $A = \{\xi(\omega) : \xi \in \Xi, \omega \in \Omega^{ar(\xi)}\}$. An operator schema $\xi \in \Xi$, is represented by an operator *header*: that contains a unique symbol, $name(\xi)$, and a list of variables, $pars(\xi)$. The operator *body* consists of three sets of lifted predicates: the *preconditions*, $pre(\xi)$, the *positive effects*, $add(\xi)$, and the *negative effects*, $del(\xi)$. Figure 1 shows an example of planning action schema from the blocksworld as 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 the *stack* planning action schema from the blocksworld as represented in PDDL.

Classical Planning with Conditional Effects

Conditional effects make it possible to repeatedly refer to the same action even though their precise effects depend on the current state. In this case each 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 from minimal input knowledge

This section formalizes the learning task we address in the paper. First we define the learning task of computing a planning action model from a given set of lifted predicates and a set of plans labelled with their corresponding initial and final states pairs. This task is defined as $\Lambda = \langle \Psi, \Pi, \Sigma\rangle$:

- Ψ is the set of lifted predicates that define the abstract state space of a given planning domain,
- $\Pi = \{\pi_1, \ldots, \pi_t\}$ is the given set of example plans,
- $\Sigma = \{\sigma_1, , \ldots, \sigma_t\}$ is a set of labels with each plan π_i, $1 \leq i \leq t$, has an associated label $\sigma_i = (s_i, s'_i)$ such that 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 that is compliant with the predicates in Ψ, the example plans Π, and their labels Σ.

As explained in this paper we aim to reduce the amount of input knowledge provided to the learning task so we redefine it as $\Lambda' = \langle \Psi, \Sigma\rangle$:

- Ψ, the set of predicates,
- $\Sigma = \{\sigma_1, , \ldots, \sigma_t\}$ the set of state pairs $\sigma_i = (s_i, s'_i)$.

A solution to the Λ' learning task is a set of operator schema Ξ that is compliant with the predicates in Ψ, and the given set of initial and final states.

Learning action models from states using a classical planner

Our approach for addressing $\Lambda' = \langle \Psi, \Sigma\rangle$ is compiling this learning task into a classical planning task that can later be solved by an off-the-shelf classical planner. The intuition behind the compilation is that a solution to P_Λ is a sequence of actions that first, programs the action action model (i.e. the preconditions, del and add effects of each action schema $\xi \in \Xi$) and then, sequentially, validates the programmed action model in their labels Σ one after the other.

To formalize the compilation we first 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 $\Phi = \langle F, A\rangle$ (i.e. share the same fluents and actions and differ only in the initial state and goals). Let Ω be the set of objects that appear in the states Σ, i.e., $\Omega = \{o | o \in s_i \cup s'_i, 1 \leq i \leq t\}$. Then F and A are the set of fluents and actions built instantiating the predicates in Ψ with the objects in Ω. Finally the initial state I_i, $1 \leq i \leq t$, is given by the state $s_i \in \sigma_i$ and the goals are defined by the state $s'_i \in \sigma_i$.

Now we are ready to define the compilation for learning action models using a 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. They have the form $header(name(\xi), \Omega_v^{ar(\xi)})$, $pre_p(name(\xi), \Omega_v^{ar(p)})$, $del_p(name(\xi), \Omega_v^{ar(p)})$ and $add_p(name(\xi), \Omega_v^{ar(p)})$ where $p \in \Psi$, $\xi \in \Xi$ and $\Omega_v = \{var_1, \ldots, var_v\}$ is a new set of objects, $\Omega_v \cap \Omega = \emptyset$, representing variable names. The number of *variable* objects, i.e. $|\Omega_v|$, is given by the action with the maximum arity in Π. For instance, for the blocksworld $\Omega_v = \{var_1, var_2\}$ since actions $stack$ and $unstack$ have two parameters.

  - Fluents $\{test_i\}_{1\leq i\leq t}$, indicating the plan where the programmed model is currently being validated.

- I_Λ, contains the fluents from F that encode the initial state $s_1 \in P_1$ and the header of the actions.
- $G_\Lambda = \{test_i\}$,$1 \leq i \leq t$, indicates that the programmed model is validated in all the examples.
- A_Λ replaces the actions in A with actions of three types:

1. The actions for programming an action schema:
   - A *precondition* with predicate $p \in \Psi$ and variables $\upsilon \in \Omega_v^{ar(p)}$ in the action schema $\xi \in \Xi$:

$$\begin{aligned}\mathsf{pre}(\mathsf{programPre}_{\xi,\mathsf{p}(\upsilon)}) =&\{\neg pre_\xi(p(\upsilon)), \neg del_\xi(p(\upsilon)),\\ &\neg add_\xi(p(\upsilon))\}.\\ \mathsf{cond}(\mathsf{programPre}_{\xi,\mathsf{p}(\upsilon)}) =&\{\emptyset\} \rhd \{pre_\xi(p(\upsilon))\}.\end{aligned}$$

   - A *negative effect* with predicate $p \in \Psi$ and variables $\upsilon \in \Omega_v^{ar(p)}$ in the action schema $\xi \in \Xi$:

$$\begin{aligned}\mathsf{pre}(\mathsf{programDel}_{\xi,\mathsf{p}(\upsilon)}) =&\{pre_\xi(p(\upsilon)), \neg del_\xi(p(\upsilon)),\\ &\neg add_\xi(p(\upsilon))\}.\\ \mathsf{cond}(\mathsf{programDel}_{\xi,\mathsf{p}(\upsilon)}) =&\{\emptyset\} \rhd \{del_\xi(p(\upsilon))\}.\end{aligned}$$

   - A *positive effect* with predicate $p \in \Psi$ and variables $\upsilon \in \Omega_v^{ar(p)}$ in the action schema $\xi \in \Xi$:

$$\begin{aligned}\mathsf{pre}(\mathsf{programAdd}_{\xi,\mathsf{p}(\upsilon)}) =&\{\neg pre_\xi(p(\upsilon)), \neg del_\xi(p(\upsilon)),\\ &\neg add_\xi(p(\upsilon))\}.\\ \mathsf{cond}(\mathsf{programAdd}_{\xi,\mathsf{p}(\upsilon)}) =&\{\emptyset\} \rhd \{add_\xi(p(\upsilon))\}.\end{aligned}$$

2. The actions for applying an operator schema $\xi \in \Xi$ (that is already programmed with variables $\upsilon \in \Omega_v^{ar(\xi)}$) and that is bound with objects $\upsilon' \in \Omega^{ar(\xi)}$)

$$\begin{aligned}\mathsf{pre}(\mathsf{apply}_{\xi,\upsilon,\upsilon'}) =&\{\neg header(\xi(\upsilon))\} \cup\\ &\{\neg pre_\xi(p(\upsilon)) \vee p(\upsilon')\}_{\forall p\in\Psi}.\\ \mathsf{cond}(\mathsf{apply}_{\xi,\upsilon,\upsilon'}) =&\{del_\xi(p(\upsilon))\} \rhd \{\neg p(\upsilon')\}_{\forall p\in\Psi},\\ &\{add_\xi(p(\upsilon))\} \rhd \{p(\upsilon')\}_{\forall p\in\Psi}.\end{aligned}$$

3. The actions for changing the active test where the model is currently being validated.

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

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 action schema is programmed it can only be executed. The only way of achieving a test fluent is by achieving the goal state in its associated label. If this is done for all the input examples it means that the programmed model is compliant with the learning input knowledge and hence, solves the action model learning task. □

Learning action models from plans using a classical planner

Interestingly the compilation is also valid for partially specified action models since the known preconditions and effects can be part of the initial state. With this regard the approach allows also transfer learning in which we generate the action model for a given sub-task and then encode this model as already programmed actions for learning new or more challenging action models. Evenmore, the compilation is extensible to the scenario in which a set of action plans Π is available s.t., each plan $\pi_i \in \Pi$, $1 \leq i \leq t$, is a solution to the corresponding classical planning instance $P_i = \langle F, A, I_i, G_i\rangle$. In this case:

- F_Λ have to include the fluents $plan(name(\xi), i, \Omega^{ar(a)})$ for encoding the plans in Π. Fluents at_i and $next_{i,i_2}$, $1 \leq i < i2 \leq n$, indicate the plan step where the programmed model is being validated and n is the max length of a plan in Π.
- I_Λ is extended with the fluents $plan(name(\xi), i, \Omega^{ar(a)})$, $1 \leq i \leq |\pi_1|$ that encode the plan $\pi_1 \in \Pi$ for solving P_1, and the fluents at_1 and $\{next_{i,i_2}\}$, $1 \leq i < i2 \leq n$, for indicating that the plan step where to start validating the programmed model.
- The actions for applying an operator schema have an extra precondition $plan(name(\xi), i, \Omega^{ar(a)})$ and an extra conditional effect $\{at_i\} \rhd \{\neg at_i, at_{i+1}\}_{\forall i\in 1\leq i<n}$.
- The actions for changing the active test have an extra precondition, $at_{|\Pi_i|}$, to indicate that we simulated the full current plan and extra conditional states to load the next plan where to validate the model,

$$\begin{aligned}&\{\emptyset\} \rhd \{\neg at_{|\pi_i|}, at_1\},\\ &\{\emptyset\} \rhd \{\neg plan(\xi, k, \upsilon\}_{1\leq k<|\Pi_i|},\\ &\{\emptyset\} \rhd \{plan(\xi, k, \upsilon\}_{1\leq k<|\Pi_{i+1}|}.\end{aligned}$$

Generating the learning examples

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

Evaluation

Related work

Conclusions

References

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.

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

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.

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

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*, volume 22, 1601. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.

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

Segovia-Aguas, J.; Jiménez, S.; and Jonsson, A. 2017. Unsupervised learning of planning tasks. In *International Conference on Automated Planning and Scheduling*.

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.